The Coq Reference Manual

Release 8.14.1

The Coq Development Team

Dec 12, 2021

CONTENTS

1 Introduction **1**

2 Specification language **3**
2.1 Core language . . . 3
2.2 Language extensions . . . 99

3 Proofs **219**
3.1 Basic proof writing . . . 219
3.2 Automatic solvers and programmable tactics . . . 411
3.3 Creating new tactics . . . 455

4 Using Coq **522**
4.1 Libraries and plugins . . . 522
4.2 Command-line and graphical tools . . . 560

5 Appendix **596**
5.1 History and recent changes . . . 596

Bibliography **755**

Glossary **759**

Command Index **761**

Tactic Index **765**

Flags, options and Tables Index **769**

Gallina Index **771**

Errors and Warnings Index **772**

Attribute Index **778**

Index **779**

CHAPTER

ONE

INTRODUCTION

This is the reference manual of Coq. Coq is an interactive theorem prover. It lets you formalize mathematical concepts and then helps you interactively generate machine-checked proofs of theorems. Machine checking gives users much more confidence that the proofs are correct compared to human-generated and -checked proofs. Coq has been used in a number of flagship verification projects, including the CompCert verified C compiler[3], and has served to verify the proof of the four color theorem[4] (among many other mathematical formalizations).

Users generate proofs by entering a series of tactics that constitute steps in the proof. There are many built-in tactics, some of which are elementary, while others implement complex decision procedures (such as `lia`, a decision procedure for linear integer arithmetic). *Ltac* and its planned replacement, *Ltac2*, provide languages to define new tactics by combining existing tactics with looping and conditional constructs. These permit automation of large parts of proofs and sometimes entire proofs. Furthermore, users can add novel tactics or functionality by creating Coq plugins using OCaml.

The Coq kernel, a small part of Coq, does the final verification that the tactic-generated proof is valid. Usually the tactic-generated proof is indeed correct, but delegating proof verification to the kernel means that even if a tactic is buggy, it won't be able to introduce an incorrect proof into the system.

Finally, Coq also supports extraction of verified programs to programming languages such as OCaml and Haskell. This provides a way of executing Coq code efficiently and can be used to create verified software libraries.

To learn Coq, beginners are advised to first start with a tutorial / book. Several such tutorials / books are listed at https://coq.inria.fr/documentation.

This manual is organized in three main parts, plus an appendix:

- **The first part presents the specification language of Coq**, that allows to define programs and state mathematical theorems. *Core language* presents the language that the kernel of Coq understands. *Language extensions* presents the richer language, with notations, implicits, etc. that a user can use and which is translated down to the language of the kernel by means of an "elaboration process".
- **The second part presents proof mode**, the central feature of Coq. *Basic proof writing* introduces this interactive mode and the available proof languages. *Automatic solvers and programmable tactics* presents some more advanced tactics, while *Creating new tactics* is about the languages that allow a user to combine tactics together and develop new ones.
- **The third part shows how to use Coq in practice.** *Libraries and plugins* presents some of the essential reusable blocks from the ecosystem and some particularly important extensions such as the program extraction mechanism. *Command-line and graphical tools* documents important tools that a user needs to build a Coq project.
- In the appendix, *History and recent changes* presents the history of Coq and changes in recent releases. This is an important reference if you upgrade the version of Coq that you use. The various indexes are very useful to **quickly browse the manual and find what you are looking for.** They are often the main entry point to the manual.

The full table of contents is presented below:

[3] http://compcert.inria.fr/
[4] https://github.com/math-comp/fourcolor

Note: License

This material (the Coq Reference Manual) may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub). Options A and B are not elected.

CHAPTER

TWO

SPECIFICATION LANGUAGE

2.1 Core language

At the heart of the Coq proof assistant is the Coq kernel. While users have access to a language with many convenient features such as *notations*, *implicit arguments*, etc. (presented in the *next chapter*), those features are translated into the core language (the Calculus of Inductive Constructions) that the kernel understands, which we present here. Furthermore, while users can build proofs interactively using tactics (see Chapter *Basic proof writing*), the role of these tactics is to incrementally build a "proof term" which the kernel will verify. More precisely, a proof term is a *term* of the Calculus of Inductive Constructions whose *type* corresponds to a theorem statement. The kernel is a type checker which verifies that terms have their expected types.

This separation between the kernel on one hand and the *elaboration engine* and *tactics* on the other follows what is known as the de Bruijn criterion (keeping a small and well delimited trusted code base within a proof assistant which can be much more complex). This separation makes it necessary to trust only a smaller, critical component (the kernel) instead of the entire system. In particular, users may rely on external plugins that provide advanced and complex tactics without fear of these tactics being buggy, because the kernel will have to check their output.

2.1.1 Basic notions and conventions

This section provides some essential notions and conventions for reading the manual.

We start by explaining the syntax and lexical conventions used in the manual. Then, we present the essential vocabulary necessary to read the rest of the manual. Other terms are defined throughout the manual. The reader may refer to the glossary index for a complete list of defined terms. Finally, we describe the various types of settings that Coq provides.

Syntax and lexical conventions

Syntax conventions

The syntax described in this documentation is equivalent to that accepted by the Coq parser, but the grammar has been edited to improve readability and presentation.

In the grammar presented in this manual, the terminal symbols are black (e.g. **forall**), whereas the nonterminals are green, italic and hyperlinked (e.g. *term*). Some syntax is represented graphically using the following kinds of blocks:

item **?** An optional item.

item **+** A list of one or more items.

item ***** An optional list of items.

item (+ **s**) A list of one or more items separated by "s" (e.g. **item$_1$ s item$_2$ s item$_3$**).

item (* **s**) An optional list of items separated by "s".

item$_1$ | **item$_2$** | **...** Alternatives (either **item$_1$** or **item$_2$** or ...).

Precedence levels[5] that are implemented in the Coq parser are shown in the documentation by appending the level to the nonterminal name (as in ***term100*** or ***ltac_expr3***).

Note: Coq uses an extensible parser. Plugins and the *notation system* can extend the syntax at run time. Some notations are defined in the *prelude*, which is loaded by default. The documented grammar doesn't include these notations. Precedence levels not used by the base grammar are omitted from the documentation, even though they could still be populated by notations or plugins.

Furthermore, some parsing rules are only activated in certain contexts (*proof mode*, *custom entries*...).

Warning: Given the complexity of these parsing rules, it would be extremely difficult to create an external program that can properly parse a Coq document. Therefore, tool writers are advised to delegate parsing to Coq, by communicating with it, for instance through SerAPI[6].

See also:

`Print Grammar`

Lexical conventions

Blanks Space, newline and horizontal tab are considered blanks. Blanks are ignored but they separate tokens.

Comments Comments are enclosed between `(*` and `*)`. They can be nested. They can contain any character. However, embedded ***string*** literals must be correctly closed. Comments are treated as blanks.

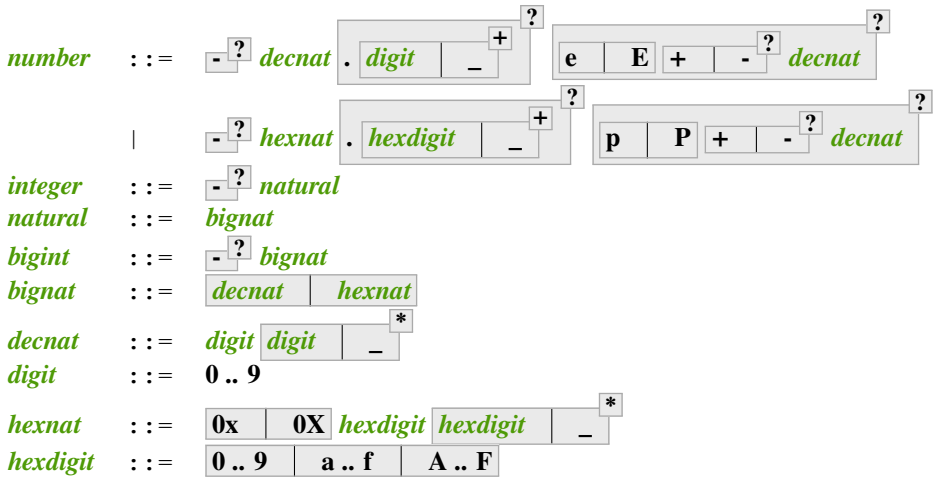
Identifiers Identifiers, written ***ident***, are sequences of letters, digits, _ and `'`, that do not start with a digit or `'`. That is, they are recognized by the following grammar (except that the string _ is reserved; it is not a valid identifier):

ident	::=	*first_letter* (*subsequent_letter*)*
first_letter	::=	**a .. z** | **A .. Z** | **_** | *unicode_letter*
subsequent_letter	::=	*first_letter* | *digit* | **'** | *unicode_id_part*

All characters are meaningful. In particular, identifiers are case-sensitive. `unicode_letter` non-exhaustively includes Latin, Greek, Gothic, Cyrillic, Arabic, Hebrew, Georgian, Hangul, Hiragana and Katakana characters, CJK ideographs, mathematical letter-like symbols and non-breaking space. `unicode_id_part` non-exhaustively includes symbols for prime letters and subscripts.

Numbers Numbers are sequences of digits with an optional fractional part and exponent, optionally preceded by a minus sign. Hexadecimal numbers start with `0x` or `0X`. ***bigint*** are integers; numbers without fractional nor exponent parts. ***bignat*** are non-negative integers. Underscores embedded in the digits are ignored, for example `1_000_000` is the same as `1000000`.

[5] https://en.wikipedia.org/wiki/Order_of_operations
[6] https://github.com/ejgallego/coq-serapi



integer and ***natural*** are limited to the range that fits into an OCaml integer (63-bit integers on most architectures). ***bigint*** and ***bignat*** have no range limitation.

The *standard library* provides some *interpretations* for ***number***. The `Number Notation` mechanism offers the user a way to define custom parsers and printers for ***number***.

Strings Strings begin and end with `"` (double quote). Use `""` to represent a double quote character within a string. In the grammar, strings are identified with `string`.

The `String Notation` mechanism offers the user a way to define custom parsers and printers for `string`.

Keywords The following character sequences are keywords defined in the main Coq grammar that cannot be used as identifiers (even when starting Coq with the `-noinit` command-line flag):

```
_ Axiom CoFixpoint Definition Fixpoint Hypothesis Parameter Prop
SProp Set Theorem Type Variable as at cofix else end
fix for forall fun if in let match return then where with
```

The following are keywords defined in notations or plugins loaded in the *prelude*:

```
IF by exists exists2 using
```

Note that loading additional modules or plugins may expand the set of reserved keywords.

Other tokens The following character sequences are tokens defined in the main Coq grammar (even when starting Coq with the `-noinit` command-line flag):

```
! #[ % & ' ( () ) * + , - ->
. .( .. ... / : ::= := :> ; < <+ <- <:
<<: <= = => > >-> >= ? @ @{ [ ] _
`( `{ { {| | }
```

The following character sequences are tokens defined in notations or plugins loaded in the *prelude*:

```
** [= |- || ->
```

Note that loading additional modules or plugins may expand the set of defined tokens.

When multiple tokens match the beginning of a sequence of characters, the longest matching token is used. Occasionally you may need to insert spaces to separate tokens. For example, if ~ and ~~ are both defined as tokens, the inputs ~ ~ and ~~ generate different tokens, whereas if ~~ is not defined, then the two inputs are equivalent.

Essential vocabulary

This section presents the most essential notions to understand the rest of the Coq manual: *terms* and *types* on the one hand, *commands* and *tactics* on the other hand.

term Terms are the basic expressions of Coq. Terms can represent mathematical expressions, propositions and proofs, but also executable programs and program types.

Here is the top-level syntax of terms. Each of the listed constructs is presented in a dedicated section. Some of these constructs (like **`term_forall_or_fun`**) are part of the core language that the kernel of Coq understands and are therefore described in *this chapter*, while others (like **`term_if`**) are language extensions that are presented in *the next chapter*.

term ::= *term_forall_or_fun*
| *term_let*
| *term_if*
| *term_fix*
| *term_cofix*
| *term100*

term100 ::= *term_cast*
| *term10*

term10 ::= *term_application*
| *one_term*

one_term ::= *term_explicit*
| *term1*

term1 ::= *term_projection*
| *term_scope*
| *term0*

term0 ::= *qualid_annotated*
| *sort*
| *primitive_notations*
| *term_evar*
| *term_match*
| *term_record*
| *term_generalizing*
| [| *term* (*;*)* | *term* : *type*? |] *univ_annot*?
| *term_ltac*
| (*term*)

qualid_annotated ::= *qualid* *univ_annot*?

Note: Many *commands* and *tactics* use **`one_term`** (in the syntax of their arguments) rather than **`term`**. The former need to be enclosed in parentheses unless they're very simple, such as a single identifier. This avoids confusing a space-separated list of terms or identifiers with a **`term_application`**.

---

type To be valid and accepted by the Coq kernel, a term needs an associated type. We express this relationship by "x *of type* T", which we write as "$x : T$". Informally, "$x : T$" can be thought as "x *belongs to* T".

The Coq kernel is a type checker: it verifies that a term has the expected type by applying a set of typing rules (see *Typing rules*). If that's indeed the case, we say that the term is well-typed.

A special feature of the Coq language is that types can depend on terms (we say that the language is dependently-typed[7]). Because of this, types and terms share a common syntax. All types are terms, but not all terms are

[7] https://en.wikipedia.org/wiki/Dependent_type

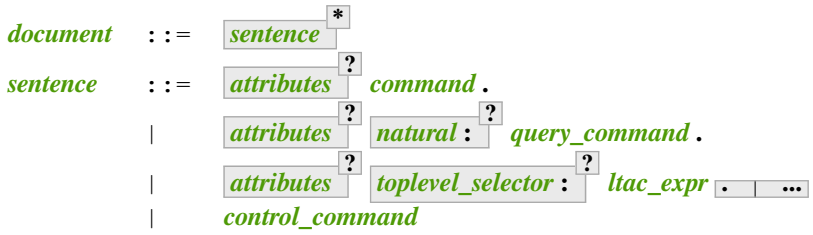
types:

type ::= *term*

Intuitively, types may be viewed as sets containing terms. We say that a type is inhabited if it contains at least one term (i.e. if we can find a term which is associated with this type). We call such terms witnesses. Note that deciding whether a type is inhabited is undecidable[8].

Formally, types can be used to construct logical foundations for mathematics alternative to the standard "set theory"[9]: we call such logical foundations "type theories"[10]. Coq is based on the Calculus of Inductive Constructions, which is a particular instance of type theory.

sentence Coq documents are made of a series of sentences that contain *commands* or *tactics*, generally terminated with a period and optionally decorated with *attributes*.



ltac_expr syntax supports both simple and compound *tactics*. For example: `split` is a simple tactic while `split; auto` combines two simple tactics.

command A `command` can be used to modify the state of a Coq document, for instance by declaring a new object, or to get information about the current state.

By convention, command names begin with uppercase letters. Commands appear in the HTML documentation in blue or gray boxes after the label "Command". In the pdf, they appear after the boldface label "Command:". Commands are listed in the command_index. Example:

Command: **`Comments`** *one_term* | *string* | *natural* *

Prints "Comments ok" and does not change the state of the document.

tactic A `tactic` specifies how to transform the current proof state as a step in creating a proof. They are syntactically valid only when Coq is in *proof mode*, such as after a `Theorem` command and before any subsequent proof-terminating command such as `Qed`. See *Proof mode* for more on proof mode.

By convention, tactic names begin with lowercase letters. Tactic appear in the HTML documentation in blue or gray boxes after the label "Tactic". In the pdf, they appear after the boldface label "Tactic:". Tactics are listed in the tactic_index.

[8] https://en.wikipedia.org/wiki/Undecidable_problem
[9] https://en.wikipedia.org/wiki/Set_theory
[10] https://en.wikipedia.org/wiki/Type_theory

Settings

There are several mechanisms for changing the behavior of Coq. The *attribute* mechanism is used to modify the behavior of a single *sentence*. The *flag*, *option* and *table* mechanisms are used to modify the behavior of Coq more globally in a document or project.

Attributes

An attribute modifies the behavior of a single sentence. Syntactically, most commands and tactics can be decorated with attributes (cf. ***sentence***), but attributes not supported by the command or tactic will trigger *This command does not support this attribute*.

attributes ::= **#[** *attribute* (*)(,) **]** *legacy_attr* (*)

attribute ::= *ident* *attr_value* (?)

attr_value ::= **=** *string*
| **=** *ident*
| **(** *attribute* (*)(,) **)**

legacy_attr ::= **Local** | **Global**
| **Polymorphic** | **Monomorphic**
| **Cumulative** | **NonCumulative**
| **Private**
| **Program**

The order of top-level attributes doesn't affect their meaning. `#[foo,bar]`, `#[bar,foo]`, `#[foo]#[bar]` and `#[bar]#[foo]` are equivalent.

Boolean attributes take the form ***ident*** $_{attr}$ **=** **yes** | **no** (?). When the **yes** | **no** value is omitted, the default is **yes**.

The legacy attributes (***legacy_attr***) provide an older, alternate syntax for certain attributes. They are equivalent to new attributes as follows:

Legacy attribute	New attribute
`Local`	*local*
`Global`	*global*
`Polymorphic, Monomorphic`	*universes(polymorphic)*
`Cumulative, NonCumulative`	*universes(cumulative)*
`Private`	*private(matching)*
`Program`	*program*

Attributes appear in the HTML documentation in blue or gray boxes after the label "Attribute". In the pdf, they appear after the boldface label "Attribute:". Attributes are listed in the attribute_index.

Warning: `This command does not support this attribute: ident.`
This warning is configured to behave as an error by default. You may turn it into a normal warning by using the *Warnings* option:

```
Set Warnings "unsupported-attributes".
#[ foo ] Comments.
```

(continues on next page)

(continued from previous page)

```
Toplevel input, characters 0-18:
> #[ foo ] Comments.
> ^^^^^^^^^^^^^^^^^^
Warning: This command does not support this attribute: foo.
[unsupported-attributes,parsing]
```

Flags, Options and Tables

The following types of settings can be used to change the behavior of Coq in subsequent commands and tactics (see *Locality attributes supported by Set and Unset* for a more precise description of the scope of these settings):

- A flag has a boolean value, such as `Universe Polymorphism`.
- An option generally has a numeric or string value, such as `Firstorder Depth`.
- A table contains a set of `string`s or `qualid`s.
- In addition, some commands provide settings, such as `Extraction Language`.

setting_name ::= *ident* +

Flags, options and tables are identified by a series of identifiers. By convention, each of the identifiers start with an initial capital letter.

Flags, options and tables appear in the HTML documentation in blue or gray boxes after the labels "Flag", "Option" and "Table". In the pdf, they appear after a boldface label. They are listed in the options_index.

Command: **Set** *setting_name* (*integer* | *string*) ?

If *setting_name* is a flag, no value may be provided; the flag is set to on. If *setting_name* is an option, a value of the appropriate type must be provided; the option is set to the specified value.

This command supports the `local`, `global` and `export` attributes. They are described *here*.

Warning: **There is no flag or option with this name: "***setting_name***".**

This warning message can be raised by `Set` and `Unset` when *setting_name* is unknown. It is a warning rather than an error because this helps library authors produce Coq code that is compatible with several Coq versions. To preserve the same behavior, they may need to set some compatibility flags or options that did not exist in previous Coq versions.

Command: **Unset** *setting_name*

If *setting_name* is a flag, it is set to off. If *setting_name* is an option, it is set to its default value.

This command supports the `local`, `global` and `export` attributes. They are described *here*.

Command: **Add** *setting_name* (*qualid* | *string*) +

Adds the specified values to the table *setting_name*.

Command: **Remove** *setting_name* (*qualid* | *string*) +

Removes the specified value from the table *setting_name*.

Command: **Test** *setting_name* (**for** (*qualid* | *string*) +) ?

If *setting_name* is a flag or option, prints its current value. If *setting_name* is a table: if the `for` clause is specified, reports whether the table contains each specified value, otherwise this is equivalent to `Print Table`. The `for` clause is not valid for flags and options.

Error: There is no flag, option or table with this name: "*setting_name*".
This error message is raised when calling the *Test* command (without the `for` clause), or the *Print Table* command, for an unknown ***setting_name***.

Error: There is no qualid-valued table with this name: "*setting_name*".
Error: There is no string-valued table with this name: "*setting_name*".
These error messages are raised when calling the *Add* or *Remove* commands, or the *Test* command with the `for` clause, if ***setting_name*** is unknown or does not have the right type.

Command: Print Options
Prints the current value of all flags and options, and the names of all tables.

Command: Print Table *setting_name*
Prints the values in the table ***setting_name***.

Command: Print Tables
A synonym for *Print Options*.

Locality attributes supported by `Set` and `Unset`

The *Set* and *Unset* commands support the mutually exclusive *local*, *export* and *global* locality attributes (or the `Local`, `Export` or `Global` prefixes).

If no attribute is specified, the original value of the flag or option is restored at the end of the current module but it is *not* restored at the end of the current section.

Newly opened modules and sections inherit the current settings.

Note: We discourage using the *global* locality attribute with the *Set* and *Unset* commands. If your goal is to define project-wide settings, you should rather use the command-line arguments `-set` and `-unset` for setting flags and options (see *By command line options*).

2.1.2 Sorts

sort	::=	**Set**
	|	**Prop**
	|	**SProp**
	|	**Type**
	|	**Type @{ _ }**
	|	**Type @{** *universe* **}**
universe	::=	**max (** *universe_expr* (**+**,) **)**
	|	*universe_expr*
universe_expr	::=	*universe_name* [**+** *natural*]?

The types of types are called sorts.

All sorts have a type and there is an infinite well-founded typing hierarchy of sorts whose base sorts are SProp, Prop and Set.

The sort Prop intends to be the type of logical propositions. If M is a logical proposition then it denotes the class of terms representing proofs of M. An object m belonging to M *witnesses* the fact that M is provable. An object of type Prop is called a proposition. We denote propositions by ***form***. This constitutes a semantic subclass of the syntactic class ***term***.

The sort SProp is like Prop but the propositions in SProp are known to have irrelevant proofs (all proofs are equal). Objects of type SProp are called strict propositions. See *SProp (proof irrelevant propositions)* for information about using SProp, and [GCST19] for meta theoretical considerations.

The sort Set intends to be the type of small sets. This includes data types such as booleans and naturals, but also products, subsets, and function types over these data types. We denote specifications (program types) by ***specif***. This constitutes a semantic subclass of the syntactic class ***term***.

SProp, Prop and Set themselves can be manipulated as ordinary terms. Consequently they also have a type. Because assuming simply that Set has type Set leads to an inconsistent theory [Coq86], the language of CIC has infinitely many sorts. There are, in addition to the base sorts, a hierarchy of universes $\mathsf{Type}(i)$ for any integer $i \geq 1$.

Like Set, all of the sorts $\mathsf{Type}(i)$ contain small sets such as booleans, natural numbers, as well as products, subsets and function types over small sets. But, unlike Set, they also contain large sets, namely the sorts Set and $\mathsf{Type}(j)$ for $j < i$, and all products, subsets and function types over these sorts.

Formally, we call $\mathcal{S}$ the set of sorts which is defined by:

$$\mathcal{S} \equiv \{\mathsf{SProp}, \mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}$$

Their properties, such as $\mathsf{Prop} : \mathsf{Type}(1)$, $\mathsf{Set} : \mathsf{Type}(1)$, and $\mathsf{Type}(i) : \mathsf{Type}(i+1)$, are described in *Subtyping rules*.

The user does not have to mention explicitly the index i when referring to the universe $\mathsf{Type}(i)$. One only writes `Type`. The system itself generates for each instance of `Type` a new index for the universe and checks that the constraints between these indexes can be solved. From the user point of view we consequently have Type : Type. We shall make precise in the typing rules the constraints between the indices.

Implementation issues In practice, the Type hierarchy is implemented using algebraic universes. An algebraic universe u is either a variable (a qualified identifier with a number) or a successor of an algebraic universe (an expression $u+1$), or an upper bound of algebraic universes (an expression $\max(u_1, ..., u_n)$), or the base universe (the expression 0) which corresponds, in the arity of template polymorphic inductive types (see Section *Well-formed inductive definitions*), to the predicative sort Set. A graph of constraints between the universe variables is maintained globally. To ensure the existence of a mapping of the universes to the positive integers, the graph of constraints must remain acyclic. Typing expressions that violate the acyclicity of the graph of constraints results in a `Universe inconsistency` error.

See also:

Printing universes, *Explicit Universes*.

2.1.3 Functions and assumptions

Binders

open_binders ::= *name*+ : *type*
| *binder*+
name ::= _
| *ident*
binder ::= *name*
| (*name*+ : *type*)
| (*name* : *type*? := *term*)
| *implicit_binders*
| *generalizing_binder*
| (*name* : *type* | *term*)
| ' *pattern0*

Various constructions such as `fun`, `forall`, `fix` and `cofix` *bind* variables. A binding is represented by an identifier. If the binding variable is not used in the expression, the identifier can be replaced by the symbol _. When the type of a bound variable cannot be synthesized by the system, it can be specified with the notation **(*ident* : *type*)**. There is also a notation for a sequence of binding variables sharing the same type: **(*ident*+ : *type*)**. A binder can also be any pattern prefixed by a quote, e.g. `'(x,y)`.

Some constructions allow the binding of a variable to value. This is called a "let-binder". The entry ***binder*** of the grammar accepts either an assumption binder as defined above or a let-binder. The notation in the latter case is **(*ident* := *term*)**. In a let-binder, only one variable can be introduced at the same time. It is also possible to give the type of the variable as follows: **(*ident* : *type* := *term*)**.

Lists of ***binder***s are allowed. In the case of `fun` and `forall`, it is intended that at least one binder of the list is an assumption otherwise fun and forall gets identical. Moreover, parentheses can be omitted in the case of a single sequence of bindings sharing the same type (e.g.: `fun (x y z : A) => t` can be shortened in `fun x y z : A => t`).

Functions (fun) and function types (forall)

term_forall_or_fun	::=	**forall** *open_binders* **,** *term*
	|	**fun** *open_binders* **=>** *term*

The expression **fun** ***ident*** **:** ***type*** **=>** ***term*** defines the *abstraction* of the variable ***ident***, of type ***type***, over the term ***term***. It denotes a function of the variable ***ident*** that evaluates to the expression ***term*** (e.g. `fun x : A => x` denotes the identity function on type `A`). The keyword `fun` can be followed by several binders as given in Section *Binders*. Functions over several variables are equivalent to an iteration of one-variable functions. For instance the expression **fun** ***ident***$_i$+ **:** ***type*** **=>** ***term*** denotes the same function as **fun** ***ident***$_i$ **:** ***type*** **=>** ***term***+. If a let-binder occurs in the list of binders, it is expanded to a let-in definition (see Section *Let-in definitions*).

The expression **forall** ***ident*** **:** ***type*****,** ***term*** denotes the *product* of the variable ***ident*** of type ***type***, over the term ***term***. As for abstractions, `forall` is followed by a binder list, and products over several variables are equivalent to an iteration of one-variable products. Note that ***term*** is intended to be a type.

If the variable ***ident*** occurs in ***term***, the product is called *dependent product*. The intention behind a dependent product `forall x : A, B` is twofold. It denotes either the universal quantification of the variable `x` of type `A` in the proposition `B` or the functional dependent product from `A` to `B` (a construction usually written $\Pi_{x:A}.B$ in set theory).

Non dependent product types have a special notation: `A -> B` stands for `forall _ : A, B`. The *non dependent product* is used both to denote the propositional implication and function types.

Function application

term_application	::=	*term1* *arg*+
	|	**@** *qualid_annotated* *term1*+
arg	::=	**(** *ident* **:=** *term* **)**
	|	*term1*

term$_{fun}$ ***term*** denotes applying the function ***term***$_{fun}$ to *term*.

term$_{fun}$ ***term***$_i$+ denotes applying ***term***$_{fun}$ to the arguments ***term***$_i$. It is equivalent to **(** **…** **(** ***term***$_{fun}$ ***term***$_1$ **)** **…** **)** ***term***$_n$: associativity is to the left.

The notation **(*ident* := *term*)** for arguments is used for making explicit the value of implicit arguments (see Section *Explicit applications*).

Assumptions

Assumptions extend the global environment with axioms, parameters, hypotheses or variables. An assumption binds an *ident* to a *type*. It is accepted by Coq only if *type* is a correct type in the global environment before the declaration and if *ident* was not previously defined in the same module. This *type* is considered to be the type (or specification, or statement) assumed by *ident* and we say that *ident* has type *type*.

Command: *assumption_token* Inline (*natural*)? (*assumpt*)+ | *assumpt*

assumption_token ::= Axiom | Axioms
| Conjecture | Conjectures
| Parameter | Parameters
| Hypothesis | Hypotheses
| Variable | Variables

assumpt ::= *ident_decl*+ *of_type*

ident_decl ::= *ident* *univ_decl*?

of_type ::= : | :> *type*

These commands bind one or more *ident*(s) to specified *type*(s) as their specifications in the global environment. The fact asserted by *type* (or, equivalently, the existence of an object of this type) is accepted as a postulate. They accept the *program* attribute.

Axiom, *Conjecture*, *Parameter* and their plural forms are equivalent. They can take the *local* *attribute*, which makes the defined *ident*s accessible by *Import* and its variants only through their fully qualified names.

Similarly, *Hypothesis*, *Variable* and their plural forms are equivalent. Outside of a section, these are equivalent to **Local Parameter**. Inside a section, the *ident*s defined are only accessible within the section. When the current section is closed, the *ident*(s) become undefined and every object depending on them will be explicitly parameterized (i.e., the variables are *discharged*). See Section *Section mechanism*.

:> If specified, *ident_decl* is automatically declared as a coercion to the class of its type. See *Implicit Coercions*.

The **Inline** clause is only relevant inside functors. See *Module*.

Example: Simple assumptions

```
Parameter X Y : Set.
Parameter (R : X -> Y -> Prop) (S : Y -> X -> Prop).
Axiom R_S_inv : forall x y, R x y <-> S y x.
```

Error: *ident* **already exists.**

Warning: *ident* **is declared as a local axiom**
Warning generated when using *Variable* or its equivalent instead of **Local Parameter** or its equivalent.

Note: We advise using the commands *Axiom*, *Conjecture* and *Hypothesis* (and their plural forms) for logical postulates (i.e. when the assertion *type* is of sort Prop), and to use the commands *Parameter* and *Variable* (and their plural forms) in other cases (corresponding to the declaration of an abstract object of the given type).

2.1.4 Definitions

Let-in definitions

term_let ::= **let** *name* [**: type**]? **:=** *term* **in** *term*
| **let** *name* [*binder*]+ [**: type**]? **:=** *term* **in** *term*
| *destructuring_let*

let *ident* **:=** $term_1$ **in** $term_2$ represents the local binding of the variable *ident* to the value $term_1$ in $term_2$.

let *ident* [*binder*]+ **:=** $term_1$ **in** $term_2$ is an abbreviation for **let** *ident* **:= fun** [*binder*]+ **=>** $term_1$ **in** $term_2$.

See also:

Extensions of the `let ... in ...` syntax are described in *Irrefutable patterns: the destructuring let variants*.

Type cast

term_cast ::= *term10* **: type**
| *term10* **<: type**
| *term10* **«: type**

The expression *term10* **:** *type* is a type cast expression. It enforces the type of *term10* to be *type*.

term10 **<:** *type* specifies that the virtual machine will be used to type check that *term10* has type *type* (see *vm_compute*).

term10 **<<:** *type* specifies that compilation to OCaml will be used to type check that *term10* has type *type* (see *native_compute*).

Top-level definitions

Definitions extend the global environment by associating names to terms. A definition can be seen as a way to give a meaning to a name or as a way to abbreviate a term. In any case, the name can later be replaced at any time by its definition.

The operation of unfolding a name into its definition is called *delta-reduction*. A definition is accepted by the system if and only if the defined term is well-typed in the current context of the definition and if the name is not already used. The name defined by the definition is called a constant and the term it refers to is its body. A definition has a type, which is the type of its *body*.

A formal presentation of constants and environments is given in Section *Typing rules*.

Command: **Definition** | **Example** *ident_decl def_body*

def_body ::= [*binder*]* [**: type**]? **:=** [*reduce*]? *term*
| [*binder*]* **: type**
reduce ::= **Eval** *red_expr* **in**

These commands bind *term* to the name *ident* in the global environment, provided that *term* is well-typed. They can take the *local attribute*, which makes the defined *ident* accessible by *Import* and its variants only

through their fully qualified names. If **reduce** is present then *ident* is bound to the result of the specified computation on *term*.

These commands also support the `universes(polymorphic)`, `program` (see *Program Definition*), `canonical`, `bypass_check(universes)`, `bypass_check(guard)`, and `using` attributes.

If *term* is omitted, *type* is required and Coq enters proof mode. This can be used to define a term incrementally, in particular by relying on the `refine` tactic. In this case, the proof should be terminated with `Defined` in order to define a *constant* for which the computational behavior is relevant. See *Entering and exiting proof mode*.

The form **Definition** *ident* **:** *type* **:=** *term* checks that the type of *term* is definitionally equal to *type*, and registers *ident* as being of type *type*, and bound to value *term*.

The form **Definition** *ident* *binder** **:** *type* **:=** *term* is equivalent to **Definition** *ident* **:** **forall** *binder** **,** *type* **:=** **fun** *binder** **=>** *term*.

See also:

`Opaque`, `Transparent`, `unfold`.

Error: *ident* **already exists.**

Error: **The term** *term* **has type** *type* **while it is expected to have type** *type'*.

Assertions and proofs

An assertion states a proposition (or a type) for which the proof (or an inhabitant of the type) is interactively built using *tactics*. Assertions cause Coq to enter *proof mode* (see *Proof mode*). Common tactics are described in the *Basic proof writing* chapter. The basic assertion command is:

Command:

thm_token *ident_decl* *binder** **:** *type* **with** *ident_decl* *binder** **:** *type* (**with** ...)*

thm_token ::= **Theorem**
| **Lemma**
| **Fact**
| **Remark**
| **Corollary**
| **Proposition**
| **Property**

After the statement is asserted, Coq needs a proof. Once a proof of *type* under the assumptions represented by *binder*s is given and validated, the proof is generalized into a proof of **forall** *binder** **,** *type* and the theorem is bound to the name *ident* in the global environment.

These commands accept the `program` attribute. See *Program Lemma*.

Forms using the **with** clause are useful for theorems that are proved by simultaneous induction over a mutually inductive assumption, or that assert mutually dependent statements in some mutual co-inductive type. It is equivalent to `Fixpoint` or `CoFixpoint` but using tactics to build the proof of the statements (or the *body* of the specification, depending on the point of view). The inductive or co-inductive types on which the induction or coinduction has to be done is assumed to be non ambiguous and is guessed by the system.

Like in a `Fixpoint` or `CoFixpoint` definition, the induction hypotheses have to be used on *structurally smaller* arguments (for a `Fixpoint`) or be *guarded by a constructor* (for a `CoFixpoint`). The verification that recursive proof arguments are correct is done only at the time of registering the lemma in the global environment. To know if

the use of induction hypotheses is correct at some time of the interactive development of a proof, use the command *Guarded*.

This command accepts the *bypass_check(universes)*, *bypass_check(guard)*, and *using* attributes.

Error: The term *term* has type *type* which should be Set, Prop or Type.

Error: *ident* already exists.
The name you provided is already defined. You have then to choose another name.

**Error: Nested proofs are discouraged and not allowed by default.
This error probably means that you forgot to close the last "Proof.
" with "Qed." or "Defined.". If you really intended to use nested proofs,
you can do so by turning the "Nested Proofs Allowed" flag on.**
You are asserting a new statement when you're already in proof mode. This feature, called nested proofs, is disabled by default. To activate it, turn the *Nested Proofs Allowed* flag on.

Proofs start with the keyword *Proof*. Then Coq enters the proof mode until the proof is completed. In proof mode, the user primarily enters tactics (see *Basic proof writing*). The user may also enter commands to manage the proof mode (see *Proof mode*).

When the proof is complete, use the *Qed* command so the kernel verifies the proof and adds it to the global environment.

Note:

1. Several statements can be simultaneously asserted provided the *Nested Proofs Allowed* flag was turned on.
2. Not only other assertions but any command can be given while in the process of proving a given assertion. In this case, the command is understood as if it would have been given before the statements still to be proved. Nonetheless, this practice is discouraged and may stop working in future versions.
3. Proofs ended by *Qed* are declared *opaque*. Their content cannot be unfolded (see *Applying conversion rules*), thus realizing some form of *proof-irrelevance*. Proofs that end with *Defined* can be unfolded.
4. *Proof* is recommended but can currently be omitted. On the opposite side, *Qed* (or *Defined*) is mandatory to validate a proof.
5. One can also use *Admitted* in place of *Qed* to turn the current asserted statement into an axiom and exit proof mode.

2.1.5 Conversion rules

Coq has conversion rules that can be used to determine if two terms are equal by definition in CIC, or *convertible*. Conversion rules consist of reduction rules and expansion rules. Equality is determined by converting both terms to a normal form, then verifying they are syntactically equal (ignoring differences in the names of bound variables by *alpha-conversion*).

See also:

Applying conversion rules, which describes tactics that apply these conversion rules.

Reductions convert terms to something that is incrementally closer to its normal form. For example, *zeta-reduction* removes **let *ident* := *term*$_1$ in *term*$_2$** constructs from a term by replacing ***ident*** with ***term*$_1$** wherever ***ident*** appears in ***term*$_2$**. The resulting term may be longer or shorter than the original.

```
Eval cbv zeta in let i := 1 in i + i.
    = 1 + 1
         : nat
```

Expansions are reductions applied in the opposite direction, for example expanding `2 + 2` to `let i := 2 in i + i`. While applying reductions gives a unique result, the associated expansion may not be unique. For example, `2 + 2` could also be expanded to `let i := 2 in i + 2`. Reductions that have a unique inverse expansion are also referred to as contractions.

The normal form is defined as the result of applying a particular set of conversion rules (beta-, delta-, iota- and zeta-reduction and eta-expansion) repeatedly until it's no longer possible to apply any of them.

Sometimes the result of a reduction tactic will be a simple value, for example reducing `2*3+4` with `cbv beta delta iota` to `10`, which requires applying several reduction rules repeatedly. In other cases, it may yield an expression containing variables, axioms or opaque contants that can't be reduced.

The useful conversion rules are shown below. All of them except for eta-expansion can be applied with conversion tactics such as *cbv*:

Conversion name	Description
beta-reduction	eliminates `fun`
delta-reduction	replaces a defined variable or constant with its definition
zeta-reduction	eliminates `let`
eta-expansion	replaces a term `f` of type `forall a : A, B` with `fun x : A => f x`
match-reduction	eliminates `match`
fix-reduction	replaces a `fix` with a *beta-redex*; recursive calls to the symbol are replaced with the `fix` term
cofix-reduction	replaces a `cofix` with a *beta-redex*; recursive calls to the symbol are replaced with the `cofix` term
iota-reduction	match-, fix- and cofix-reduction together

Applying conversion rules describes tactics that only apply conversion rules. (Other tactics may use conversion rules in addition to other changes to the proof state.)

α-conversion

Two terms are α-convertible if they are syntactically equal ignoring differences in the names of variables bound within the expression. For example `forall x, x + 0 = x` is α-convertible with `forall y, y + 0 = y`.

β-reduction

β-reduction reduces a beta-redex, which is a term in the form `(fun x => t) u`. (Beta-redex is short for "beta-reducible expression", a term from lambda calculus. See Beta reduction[11] for more background.)

Formally, in any *global environment* E and *local context* Γ, the beta-reduction rule is:

Beta

$$\overline{E[\Gamma] \vdash ((\lambda x : T.\ t)\ u)\ \triangleright_\beta\ t\{x/u\}}$$

We say that $t\{x/u\}$ is the *β-contraction* of $((\lambda x : T.\ t)\ u)$ and, conversely, that $((\lambda x : T.\ t)\ u)$ is the *β-expansion* of $t\{x/u\}$.

[11] https://en.wikipedia.org/wiki/Beta_normal_form#Beta_reduction

Terms of the *Calculus of Inductive Constructions* enjoy some fundamental properties such as confluence, strong normalization, subject reduction. These results are theoretically of great importance but we will not detail them here and refer the interested reader to [Coq85].

δ-reduction

δ-reduction replaces variables defined in *local contexts* or *constants* defined in the *global environment* with their values. Unfolding means to replace a constant by its definition. Formally, this is:

Delta-Local

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x := t : T) \in \Gamma}{E[\Gamma] \vdash x \ \triangleright_{\Delta} \ t}$$

Delta-Global

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c := t : T) \in E}{E[\Gamma] \vdash c \ \triangleright_{\delta} \ t}$$

Delta-reduction only unfolds *constants* that are marked transparent. Opaque is the opposite of transparent; *delta-reduction* doesn't unfold opaque constants.

ι-reduction

A specific conversion rule is associated with the inductive objects in the global environment. We shall give later on (see Section *Well-formed inductive definitions*) the precise rules but it just says that a destructor applied to an object built from a constructor behaves as expected. This reduction is called ι-reduction and is more precisely studied in [PM93a, Wer94].

ζ-reduction

ζ-reduction removes *let-in definitions* in terms by replacing the defined variable by its value. One way this reduction differs from δ-reduction is that the declaration is removed from the term entirely. Formally, this is:

Zeta

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad E[\Gamma] \vdash u : U \qquad E[\Gamma :: (x := u : U)] \vdash t : T}{E[\Gamma] \vdash \mathsf{let}\ x := u : U\ \mathsf{in}\ t \ \triangleright_{\zeta} \ t\{x/u\}}$$

η-expansion

Another important concept is η-expansion. It is legal to identify any term t of functional type $\forall x : T,\ U$ with its so-called η-expansion

$$\lambda x : T.\ (t\ x)$$

for x an arbitrary variable name fresh in t.

Note: We deliberately do not define η-reduction:

$$\lambda x : T.\ (t\ x) \ \not\triangleright_{\eta} \ t$$

This is because, in general, the type of t need not be convertible to the type of $\lambda x : T.\ (t\ x)$. E.g., if we take f such that:

$$f \ : \ \forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$$

then

$$\lambda x : \mathsf{Type}(1).\ (f\ x)\ :\ \forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$$

We could not allow

$$\lambda x : \mathsf{Type}(1).\ (f\ x)\ \rhd_{\eta}\ f$$

because the type of the reduced term $\forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$ would not be convertible to the type of the original term $\forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$.

Examples

Example: Simple delta, fix, beta and match reductions

+ is a *notation* for `Nat.add`, which is defined with a *`Fixpoint`*.

```
Print Nat.add.
    Nat.add =
    fix add (n m : nat) {struct n} : nat :=
      match n with
      | 0 => m
      | S p => S (add p m)
      end
         : nat -> nat -> nat

    Arguments Nat.add (_ _)%nat_scope

Goal 1 + 1 = 2.
    1 goal

      ============================
      1 + 1 = 2

cbv delta.
    1 goal

      ============================
      (fix add (n m : nat) {struct n} : nat :=
         match n with
         | 0 => m
         | S p => S (add p m)
         end) 1 1 = 2

cbv fix.
    1 goal

      ============================
      (fun n m : nat =>
       match n with
       | 0 => m
       | S p =>
           S
             ((fix add (n0 m0 : nat) {struct n0} : nat :=
                 match n0 with
```

(continues on next page)

(continued from previous page)

```
                | 0 => m0
                | S p0 => S (add p0 m0)
                end) p m)
       end) 1 1 = 2

cbv beta.
    1 goal

      ============================
      match 1 with
      | 0 => 1
      | S p =>
          S
            ((fix add (n m : nat) {struct n} : nat :=
                match n with
                | 0 => m
                | S p0 => S (add p0 m)
                end) p 1)
      end = 2

cbv match.
    1 goal

      ============================
      S
        ((fix add (n m : nat) {struct n} : nat :=
            match n with
            | 0 => m
            | S p => S (add p m)
            end) 0 1) = 2
```

The term can be fully reduced with `cbv`:

```
Goal 1 + 1 = 2.
    1 goal

      ============================
      1 + 1 = 2

cbv.
    1 goal

      ============================
      2 = 2
```

Proof Irrelevance

It is legal to identify any two terms whose common type is a strict proposition $A : \mathsf{SProp}$. Terms in a strict propositions are therefore called *irrelevant*.

Convertibility

Let us write $E[\Gamma] \vdash t \rhd u$ for the contextual closure of the relation t reduces to u in the global environment E and local context Γ with one of the previous reductions β, δ, ι or ζ.

We say that two terms t_1 and t_2 are *βδιζη-convertible*, or simply convertible, or *definitionally equal*, in the global environment E and local context Γ iff there exist terms u_1 and u_2 such that $E[\Gamma] \vdash t_1 \rhd ... \rhd u_1$ and $E[\Gamma] \vdash t_2 \rhd ... \rhd u_2$ and either u_1 and u_2 are identical up to irrelevant subterms, or they are convertible up to η-expansion, i.e. u_1 is $\lambda x : T.\ u_1'$ and $u_2 x$ is recursively convertible to u_1', or, symmetrically, u_2 is $\lambda x : T.\ u_2'$ and $u_1 x$ is recursively convertible to u_2'. We then write $E[\Gamma] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2$.

Apart from this we consider two instances of polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive types (see below) convertible

$$E[\Gamma] \vdash t\ w_1 ... w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'$$

if we have subtypings (see below) in both directions, i.e.,

$$E[\Gamma] \vdash t\ w_1 ... w_m \leq_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'$$

and

$$E[\Gamma] \vdash t\ w_1' ... w_m' \leq_{\beta\delta\iota\zeta\eta} t\ w_1 ... w_m.$$

Furthermore, we consider

$$E[\Gamma] \vdash c\ v_1 ... v_m =_{\beta\delta\iota\zeta\eta} c'\ v_1' ... v_m'$$

convertible if

$$E[\Gamma] \vdash v_i =_{\beta\delta\iota\zeta\eta} v_i'$$

and we have that c and c' are the same constructors of different instances of the same inductive types (differing only in universe levels) such that

$$E[\Gamma] \vdash c\ v_1 ... v_m : t\ w_1 ... w_m$$

and

$$E[\Gamma] \vdash c'\ v_1' ... v_m' : t'\ w_1' ... w_m'$$

and we have

$$E[\Gamma] \vdash t\ w_1 ... w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'.$$

The convertibility relation allows introducing a new typing rule which says that two convertible well-formed types have the same inhabitants.

2.1.6 Typing rules

The underlying formal language of Coq is a Calculus of Inductive Constructions (CIC) whose inference rules are presented in this chapter. The history of this formalism as well as pointers to related work are provided in a separate chapter; see *Early history of Coq*.

The terms

The expressions of the CIC are *terms* and all terms have a *type*. There are types for functions (or programs), there are atomic types (especially datatypes)... but also types for proofs and types for the types themselves. Especially, any object handled in the formalism must belong to a type. For instance, universal quantification is relative to a type and takes the form "*for all x of type T, P*". The expression "*x of type T*" is written "$x : T$". Informally, "$x : T$" can be thought as "*x belongs to T*".

Terms are built from sorts, variables, constants, abstractions, applications, local definitions, and products. From a syntactic point of view, types cannot be distinguished from terms, except that they cannot start by an abstraction or a constructor. More precisely the language of the *Calculus of Inductive Constructions* is built from the following rules.

1. the sorts SProp, Prop, Set, Type(i) are terms.
2. variables, hereafter ranged over by letters x, y, etc., are terms
3. constants, hereafter ranged over by letters c, d, etc., are terms.
4. if x is a variable and T, U are terms then $\forall x : T,\ U$ (`forall x:T, U` in Coq concrete syntax) is a term. If x occurs in U, $\forall x : T,\ U$ reads as "for all x of type T, U". As U depends on x, one says that $\forall x : T,\ U$ is a *dependent product*. If x does not occur in U then $\forall x : T,\ U$ reads as "if T then U". A *non dependent product* can be written: $T \to U$.
5. if x is a variable and T, u are terms then $\lambda x : T.\ u$ (`fun x:T => u` in Coq concrete syntax) is a term. This is a notation for the λ-abstraction of λ-calculus [Bar81]. The term $\lambda x : T.\ u$ is a function which maps elements of T to the expression u.
6. if t and u are terms then $(t\ u)$ is a term (`t u` in Coq concrete syntax). The term $(t\ u)$ reads as "t applied to u".
7. if x is a variable, and t, T and u are terms then let $x := t : T$ in u is a term which denotes the term u where the variable x is locally bound to t of type T. This stands for the common "let-in" construction of functional programs such as ML or Scheme.

Free variables. The notion of free variables is defined as usual. In the expressions $\lambda x : T.\ U$ and $\forall x : T,\ U$ the occurrences of x in U are bound.

Substitution. The notion of substituting a term t to free occurrences of a variable x in a term u is defined as usual. The resulting term is written $u\{x/t\}$.

The logical vs programming readings. The constructions of the CIC can be used to express both logical and programming notions, accordingly to the Curry-Howard correspondence between proofs and programs, and between propositions and types [CFC58, dB72, How80].

For instance, let us assume that nat is the type of natural numbers with zero element written 0 and that `True` is the always true proposition. Then $\to$ is used both to denote nat $\to$ nat which is the type of functions from nat to nat, to denote True→True which is an implicative proposition, to denote nat $\to$ Prop which is the type of unary predicates over the natural numbers, etc.

Let us assume that `mult` is a function of type nat $\to$ nat $\to$ nat and `eqnat` a predicate of type nat $\to$ nat $\to$ Prop. The λ-abstraction can serve to build "ordinary" functions as in $\lambda x :$ nat. (mult x x) (i.e. `fun x:nat => mult x x` in Coq notation) but may build also predicates over the natural numbers. For instance $\lambda x :$ nat. (eqnat x 0) (i.e. `fun x:nat => eqnat x 0` in Coq notation) will represent the predicate of one variable x which asserts the equality of x with 0. This predicate has type nat $\to$ Prop and it can be applied to any expression of type nat, say t, to give an object $P\ t$ of type Prop, namely a proposition.

Furthermore `forall x:nat, P x` will represent the type of functions which associate with each natural number n an object of type $(P\ n)$ and consequently represent the type of proofs of the formula "$\forall x.\ P(x)$".

Typing rules

As objects of type theory, terms are subjected to *type discipline*. The well typing of a term depends on a local context and a global environment.

Local context. A *local context* is an ordered list of declarations of *variables*. The declaration of a variable x is either an *assumption*, written $x : T$ (where T is a type) or a *definition*, written $x := t : T$. Local contexts are written in brackets, for example $[x : T;\ y := u : U;\ z : V]$. The variables declared in a local context must be distinct. If Γ is a local context that declares x, we write $x \in \Gamma$. Writing $(x : T) \in \Gamma$ means there is an assumption or a definition giving the type T to x in Γ. If Γ defines $x := t : T$, we also write $(x := t : T) \in \Gamma$. For the rest of the chapter, $\Gamma :: (y : T)$ denotes the local context Γ enriched with the local assumption $y : T$. Similarly, $\Gamma :: (y := t : T)$ denotes the local context Γ enriched with the local definition $(y := t : T)$. The notation $[]$ denotes the empty local context. Writing $\Gamma_1; \Gamma_2$ means concatenation of the local context Γ_1 and the local context Γ_2.

Global environment. A *global environment* is an ordered list of *declarations*. Global declarations are either *assumptions*, *definitions* or declarations of inductive objects. Inductive objects declare both constructors and inductive or coinductive types (see Section *Theory of inductive definitions*).

In the global environment, *assumptions* are written as $(c : T)$, indicating that c is of the type T. *Definitions* are written as $c := t : T$, indicating that c has the value t and type T. We shall call such names *constants*. For the rest of the chapter, the $E;\ c : T$ denotes the global environment E enriched with the assumption $c : T$. Similarly, $E;\ c := t : T$ denotes the global environment E enriched with the definition $(c := t : T)$.

The rules for inductive definitions (see Section *Theory of inductive definitions*) have to be considered as assumption rules in which the following definitions apply: if the name c is declared in E, we write $c \in E$ and if $c : T$ or $c := t : T$ is declared in E, we write $(c : T) \in E$.

Typing rules. In the following, we define simultaneously two judgments. The first one $E[\Gamma] \vdash t : T$ means the term t is well-typed and has type T in the global environment E and local context Γ. The second judgment $\mathcal{WF}(E)[\Gamma]$ means that the global environment E is well-formed and the local context Γ is a valid local context in this global environment.

A term t is well typed in a global environment E iff there exists a local context Γ and a term T such that the judgment $E[\Gamma] \vdash t : T$ can be derived from the following rules.

W-Empty

$$\frac{}{\mathcal{WF}([])[]}$$

W-Local-Assum

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$

W-Local-Def

$$\frac{E[\Gamma] \vdash t : T \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x := t : T)]}$$

W-Global-Assum

$$\frac{E[] \vdash T : s \qquad s \in \mathcal{S} \qquad c \notin E}{\mathcal{WF}(E;\ c : T)[]}$$

W-Global-Def

$$\frac{E[] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E;\ c := t : T)[]}$$

Ax-SProp

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{SProp} : \mathsf{Type}(1)}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Prop} : \mathsf{Type}(1)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Set} : \mathsf{Type}(1)}$$

Ax-Type

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Type}(i) : \mathsf{Type}(i+1)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma \ \text{ or } \ (x := t : T) \in \Gamma \text{ for some } t}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T) \in E \ \text{ or } \ (c := t : T) \in E \text{ for some } t}{E[\Gamma] \vdash c : T}$$

Prod-SProp

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{SProp}}{E[\Gamma] \vdash \forall\ x : T, U : \mathsf{SProp}}$$

Prod-Prop

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Prop}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Prop}}$$

Prod-Set

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Prop}, \mathsf{Set}\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Set}}$$

Prod-Type

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Type}(i)\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(i)}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Type}(i)}$$

Lam

$$\frac{E[\Gamma] \vdash \forall x : T,\ U : s \qquad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.\ t : \forall x : T,\ U}$$

App

$$\frac{E[\Gamma] \vdash t : \forall x : U,\ T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

Let

$$\frac{E[\Gamma] \vdash t : T \qquad E[\Gamma :: (x := t : T)] \vdash u : U}{E[\Gamma] \vdash \mathsf{let}\ x := t : T\ \mathsf{in}\ u : U\{x/t\}}$$

Note: **Prod-Prop** and **Prod-Set** typing-rules make sense if we consider the semantic difference between Prop and Set:

- All values of a type that has a sort Set are extractable.
- No values of a type that has a sort Prop are extractable.

Note: We may have $\mathsf{let}\ x := t : T\ \mathsf{in}\ u$ well-typed without having $((\lambda x : T.\ u)\ t)$ well-typed (where T is a type of t). This is because the value t associated with x may be used in a conversion rule (see Section *Conversion rules*).

Subtyping rules

At the moment, we did not take into account one rule between universes which says that any term in a universe of index i is also a term in the universe of index $i + 1$ (this is the *cumulativity* rule of CIC). This property extends the equivalence relation of convertibility into a *subtyping* relation inductively defined by:

1. if $E[\Gamma] \vdash t =_{\beta\delta\iota\zeta\eta} u$ then $E[\Gamma] \vdash t \leq_{\beta\delta\iota\zeta\eta} u$,
2. if $i \leq j$ then $E[\Gamma] \vdash \mathsf{Type}(i) \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(j)$,
3. for any i, $E[\Gamma] \vdash \mathsf{Set} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$,
4. $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Set}$, hence, by transitivity, $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$, for any i (note: SProp is not related by cumulativity to any other term)
5. if $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta\eta} U$ and $E[\Gamma :: (x : T)] \vdash T' \leq_{\beta\delta\iota\zeta\eta} U'$ then $E[\Gamma] \vdash \forall x : T,\ T' \leq_{\beta\delta\iota\zeta\eta} \forall x : U,\ U'$.
6. if $\mathsf{Ind}\ [p]\ (\Gamma_I := \Gamma_C)$ is a universe polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive type (see below) and $(t : \forall \Gamma_P, \forall \Gamma_{Arr(t)}, S) \in \Gamma_I$ and $(t' : \forall \Gamma'_P, \forall \Gamma'_{Arr(t)}, S') \in \Gamma_I$ are two different instances of *the same* inductive type (differing only in universe levels) with constructors

   $$[c_1 : \forall \Gamma_P, \forall T_{1,1} ... T_{1,n_1},\ t\ v_{1,1} ... v_{1,m};\ ...;\ c_k : \forall \Gamma_P, \forall T_{k,1} ... T_{k,n_k},\ t\ v_{k,1} ... v_{k,m}]$$

   and

   $$[c_1 : \forall \Gamma'_P, \forall T'_{1,1} ... T'_{1,n_1},\ t'\ v'_{1,1} ... v'_{1,m};\ ...;\ c_k : \forall \Gamma'_P, \forall T'_{k,1} ... T'_{k,n_k},\ t'\ v'_{k,1} ... v'_{k,m}]$$

   respectively then

   $$E[\Gamma] \vdash t\ w_1 ... w_m \leq_{\beta\delta\iota\zeta\eta} t'\ w'_1 ... w'_m$$

(notice that t and t' are both fully applied, i.e., they have a sort as a type) if

$$E[\Gamma] \vdash w_i =_{\beta\delta\iota\zeta\eta} w'_i$$

for $1 \leq i \leq m$ and we have

$$E[\Gamma] \vdash T_{i,j} \leq_{\beta\delta\iota\zeta\eta} T'_{i,j}$$

and

$$E[\Gamma] \vdash A_i \leq_{\beta\delta\iota\zeta\eta} A'_i$$

where $\Gamma_{Arr(t)} = [a_1 : A_1;\ ...;\ a_l : A_l]$ and $\Gamma'_{Arr(t)} = [a_1 : A'_1;\ ...;\ a_l : A'_l]$.

The conversion rule up to subtyping is now exactly:

Conv

$$\frac{E[\Gamma] \vdash U : s \qquad E[\Gamma] \vdash t : T \qquad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}$$

Normal form. A term which cannot be any more reduced is said to be in *normal form*. There are several ways (or strategies) to apply the reduction rules. Among them, we have to mention the *head reduction* which will play an important role (see Chapter *Tactics*). Any term t can be written as $\lambda x_1 : T_1. ...\lambda x_k : T_k. (t_0\ t_1 ... t_n)$ where t_0 is not an application. We say then that t_0 is the *head of* t. If we assume that t_0 is $\lambda x : T.\ u_0$ then one step of β-head reduction of t is:

$$\lambda x_1 : T_1. ...\lambda x_k : T_k. (\lambda x : T.\ u_0\ t_1 ... t_n) \ \triangleright\ \lambda (x_1 : T_1) ... (x_k : T_k). (u_0\{x/t_1\}\ t_2 ... t_n)$$

Iterating the process of head reduction until the head of the reduced term is no more an abstraction leads to the *β-head normal form* of t:

$$t \triangleright ... \triangleright \lambda x_1 : T_1. ...\lambda x_k : T_k. (v\ u_1 ... u_m)$$

where v is not an abstraction (nor an application). Note that the head normal form must not be confused with the normal form since some u_i can be reducible. Similar notions of head-normal forms involving δ, ι and ζ reductions or any combination of those can also be defined.

Admissible rules for global environments

From the original rules of the type system, one can show the admissibility of rules which change the local context of definition of objects in the global environment. We show here the admissible rules that are used in the discharge mechanism at the end of a section.

Abstraction. One can modify a global declaration by generalizing it over a previously assumed constant c. For doing that, we need to modify the reference to the global declaration in the subsequent global environment and local context by explicitly applying this constant to the constant c.

Below, if Γ is a context of the form $[y_1 : A_1;\ ...;\ y_n : A_n]$, we write $\forall x : U,\ \Gamma\{c/x\}$ to mean $[y_1 : \forall x : U,\ A_1\{c/x\};\ ...;\ y_n : \forall x : U,\ A_n\{c/x\}]$ and $E\{|\Gamma|/|\Gamma|c\}$ to mean the parallel substitution $E\{y_1/(y_1\ c)\}...\{y_n/(y_n\ c)\}$.

First abstracting property:

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' := t : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' := \lambda x : U.\ t\{c/x\} : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c:U;\ E';\ \mathsf{Ind}\,[p]\,(\Gamma_I\ :=\ \Gamma_C)\,;\ E'')[\Gamma]}{\mathcal{WF}\ \begin{array}{l}(E;\ c:U;\ E';\ \mathsf{Ind}\,[p+1]\,(\forall x:U,\ \Gamma_I\{c/x\}\ :=\ \forall x:U,\ \Gamma_C\{c/x\})\,;\ E''\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\})\\ [\Gamma\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\}]\end{array}}$$

One can similarly modify a global declaration by generalizing it over a previously defined constant c. Below, if Γ is a context of the form $[y_1 : A_1;\ ...;\ y_n : A_n]$, we write $\Gamma\{c/u\}$ to mean $[y_1 : A_1\{c/u\};\ ...;\ y_n : A_n\{c/u\}]$.

Second abstracting property:

$$\frac{\mathcal{WF}(E;\ c:=u:U;\ E';\ c':=t:T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c:=u:U;\ E';\ c':=(\mathsf{let}\ x:=u:U\ \mathsf{in}\ t\{c/x\}):T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c:=u:U;\ E';\ c':T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c:=u:U;\ E';\ c':T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c:=u:U;\ E';\ \mathsf{Ind}\,[p]\,(\Gamma_I\ :=\ \Gamma_C)\,;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c:=u:U;\ E';\ \mathsf{Ind}\,[p]\,(\Gamma_I\{c/u\}\ :=\ \Gamma_C\{c/u\})\,;\ E'')[\Gamma]}$$

Pruning the local context. If one abstracts or substitutes constants with the above rules then it may happen that some declared or defined constant does not occur any more in the subsequent global environment and in the local context. One can consequently derive the following property.

First pruning property:

$$\frac{\mathcal{WF}(E;\ c:U;\ E')[\Gamma] \qquad c\ \mathsf{does\ not\ occur\ in}\ E'\ \mathsf{and}\ \Gamma}{\mathcal{WF}(E;E')[\Gamma]}$$

Second pruning property:

$$\frac{\mathcal{WF}(E;\ c:=u:U;\ E')[\Gamma] \qquad c\ \mathsf{does\ not\ occur\ in}\ E'\ \mathsf{and}\ \Gamma}{\mathcal{WF}(E;E')[\Gamma]}$$

The Calculus of Inductive Constructions with impredicative Set

Coq can be used as a type checker for the Calculus of Inductive Constructions with an impredicative sort Set by using the compiler option `-impredicative-set`. For example, using the ordinary `coqtop` command, the following is rejected,

---

Example

```
Fail Definition id: Set := forall X:Set,X->X.
    The command has indeed failed with message:
    The term "forall X : Set, X -> X" has type "Type"
    while it is expected to have type "Set"
    (universe inconsistency: Cannot enforce Set+1 <= Set).
```

---

while it will type check, if one uses instead the `coqtop -impredicative-set` option..

The major change in the theory concerns the rule for product formation in the sort Set, which is extended to a domain in any sort:

ProdImp

$$\frac{E[\Gamma]\vdash T:s \qquad s\in\mathcal{S} \qquad E[\Gamma::(x:T)]\vdash U:\mathsf{Set}}{E[\Gamma]\vdash\forall x:T,\ U:\mathsf{Set}}$$

This extension has consequences on the inductive definitions which are allowed. In the impredicative system, one can build so-called *large inductive definitions* like the example of second-order existential quantifier (`exSet`).

There should be restrictions on the eliminations which can be performed on such definitions. The elimination rules in the impredicative system for sort Set become:

Set1

$$\frac{s \in \{\mathsf{Prop}, \mathsf{Set}\}}{[I : \mathsf{Set} | I \to s]}$$

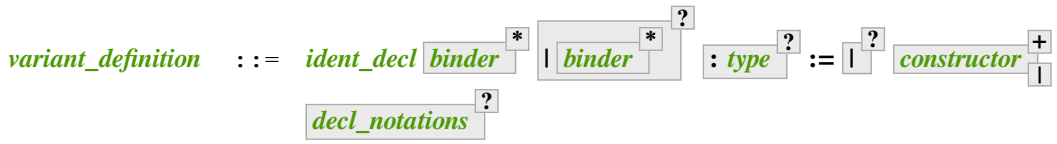
Set2

$$\frac{I \text{ is a small inductive definition} \qquad s \in \{\mathsf{Type}(i)\}}{[I : \mathsf{Set} | I \to s]}$$

2.1.7 Variants and the `match` construct

Variants

Command: `Variant` *variant_definition* `with` *variant_definition* *



The *Variant* command is similar to the *Inductive* command, except that it disallows recursive definition of types (for instance, lists cannot be defined using *Variant*). No induction scheme is generated for this variant, unless the *Nonrecursive Elimination Schemes* flag is on.

This command supports the *universes(polymorphic)*, *universes(template)*, *universes(cumulative)*, and *private(matching)* attributes.

Error: `The` *natural* `th argument of` *ident* `must be` *ident* `in` *type*.

Private (matching) inductive types

Attribute: `private(matching)`

This *attribute* can be used to forbid the use of the `match` construct on objects of this inductive type outside of the module where it is defined. There is also a legacy syntax using the `Private` prefix (cf. ***legacy_attr***).

The main use case of private (matching) inductive types is to emulate quotient types / higher-order inductive types in projects such as the HoTT library[12].

Example

```
Module Foo.
    Interactive Module Foo started

#[ private(matching) ] Inductive my_nat := my_O : my_nat | my_S : my_nat -> my_nat.
    my_nat is defined
```

(continues on next page)

[12] https://github.com/HoTT/HoTT

(continued from previous page)

```
Check (fun x : my_nat => match x with my_O => true | my_S _ => false end).
    fun x : my_nat => match x with
                      | my_O => true
                      | my_S _ => false
                      end
         : my_nat -> bool

End Foo.
    Module Foo is defined

Import Foo.
Fail Check (fun x : my_nat => match x with my_O => true | my_S _ => false end).
    The command has indeed failed with message:
    case analysis on a private type.
```

Definition by cases: match

Objects of inductive types can be destructured by a case-analysis construction called *pattern matching* expression. A pattern matching expression is used to analyze the structure of an inductive object and to apply specific treatments accordingly.

term_match ::= **match** *case_item* (+ ,) **return** *term100* (?) **with** (?) *eqn* (* |) **end**

case_item ::= *term100* (**as** *name*) (?) (**in** *pattern*) (?)

eqn ::= (*pattern* (+ ,) (+ |)) **=>** *term*

pattern ::= *pattern10* **:** *term*
| *pattern10*

pattern10 ::= *pattern1* **as** *name*
| *pattern1* *pattern1* (*)
| **@** *qualid* *pattern1* (*)

pattern1 ::= *pattern0* **%** *scope_key*
| *pattern0*

pattern0 ::= *qualid*
| {| *qualid* **:=** *pattern* (*) |}
| _
| (*pattern* (+ |))
| *number*
| *string*

Note that the ***pattern ::= pattern10 : term*** production is not supported in **match** patterns. Trying to use it will give this error:

Error: Casts are not supported in this pattern.

This paragraph describes the basic form of pattern matching. See Section *Multiple and nested pattern matching* and Chapter *Extended pattern matching* for the description of the general form. The basic form of pattern matching is char-

acterized by a single *case_item* expression, an *eqn* restricted to a single *pattern* and *pattern* restricted to the form *qualid* *ident*[*] .

The expression **match** *term* **return** *term100*[?] **with** *pattern$_i$* **=>** *term$_i$*[+ |] **end** denotes a *pattern matching* over the term *term* (expected to be of an inductive type I). The *term$_i$* are the *branches* of the pattern matching expression. Each *pattern$_i$* has the form *qualid ident* where *qualid* must denote a constructor. There should be exactly one branch for every constructor of I.

The **return** *term100* clause gives the type returned by the whole match expression. There are several cases. In the *non dependent* case, all branches have the same type, and the **return** *term100* specifies that type. In this case, **return** *term100* can usually be omitted as it can be inferred from the type of the branches[1].

In the *dependent* case, there are three subcases. In the first subcase, the type in each branch may depend on the exact value being matched in the branch. In this case, the whole pattern matching itself depends on the term being matched. This dependency of the term being matched in the return type is expressed with an *ident* clause where *ident* is dependent in the return type. For instance, in the following example:

```
Inductive bool : Type := true : bool | false : bool.
Inductive eq (A:Type) (x:A) : A -> Prop := eq_refl : eq A x x.
Inductive or (A:Prop) (B:Prop) : Prop :=
  | or_introl : A -> or A B
  | or_intror : B -> or A B.

Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false) :=
  match b as x return or (eq bool x true) (eq bool x false) with
  | true => or_introl (eq bool true true) (eq bool true false) (eq_refl bool true)
  | false => or_intror (eq bool false true) (eq bool false false) (eq_refl bool false)
  end.
```

the branches have respective types "`or (eq bool true true) (eq bool true false)`" and "`or (eq bool false true) (eq bool false false)`" while the whole pattern matching expression has type "`or (eq bool b true) (eq bool b false)`", the identifier `b` being used to represent the dependency.

Note: When the term being matched is a variable, the `as` clause can be omitted and the term being matched can serve itself as binding name in the return type. For instance, the following alternative definition is accepted and has the same meaning as the previous one.

```
Definition bool_case (b:bool) : or (eq bool b true) (eq bool b false) :=
match b return or (eq bool b true) (eq bool b false) with
| true => or_introl (eq bool true true) (eq bool true false) (eq_refl bool true)
| false => or_intror (eq bool false true) (eq bool false false) (eq_refl bool false)
end.
```

The second subcase is only relevant for annotated inductive types such as the equality predicate (see Section *Equality*), the order predicate on natural numbers or the type of lists of a given length (see Section *Matching objects of dependent types*). In this configuration, the type of each branch can depend on the type dependencies specific to the branch and the whole pattern matching expression has a type determined by the specific dependencies in the type of the term being matched. This dependency of the return type in the annotations of the inductive type is expressed with a clause in the form **in** *qualid* _[+] *pattern*[+], where

- *qualid* is the inductive type of the term being matched;
- the holes _ match the parameters of the inductive type: the return type is not dependent on them.

[1] Except if the inductive type is empty in which case there is no equation that can be used to infer the return type.

- each *pattern* matches the annotations of the inductive type: the return type is dependent on them
- in the basic case which we describe below, each *pattern* is a name *ident*; see *Patterns in in* for the general case

For instance, in the following example:

```
Definition eq_sym (A:Type) (x y:A) (H:eq A x y) : eq A y x :=
match H in eq _ _ z return eq A z x with
| eq_refl _ _ => eq_refl A x
end.
```

the type of the branch is `eq A x x` because the third argument of `eq` is `x` in the type of the pattern `eq_refl`. On the contrary, the type of the whole pattern matching expression has type `eq A y x` because the third argument of eq is y in the type of H. This dependency of the case analysis in the third argument of `eq` is expressed by the identifier `z` in the return type.

Finally, the third subcase is a combination of the first and second subcase. In particular, it only applies to pattern matching on terms in a type with annotations. For this third subcase, both the clauses `as` and `in` are available.

There are specific notations for case analysis on types with one or two constructors: `if ... then ... else ...` and `let (...,...) := ... in ...` (see Sections *Pattern-matching on boolean values: the if expression* and *Irrefutable patterns: the destructuring let variants*).

2.1.8 Record types

The `Record` construction is a macro allowing the definition of records as is done in many programming languages. Its syntax is described in the grammar below. In fact, the `Record` macro is more general than the usual record types, since it allows also for "manifest" expressions. In this sense, the `Record` construction allows defining "signatures".

Command: **Record** | **Structure** *record_definition* **with** *record_definition* *

record_definition ::= > ? *ident_decl* *binder* * : *sort* ? := *ident* ? { *record_field* * ; ? } ?

record_field ::= #[*attribute* * ,] * *name* *field_body* ? | *natural* ? *decl_notations* ?

field_body ::= *binder* * *of_type*

| *binder* * *of_type* := *term*

| *binder* * := *term*

term_record ::= {| *field_def* * ; ? |}

field_def ::= *qualid* *binder* * := *term*

Each *record_definition* defines a record named by *ident_decl*. The constructor name is given by *ident*. If the constructor name is not specified, then the default name **Build_***ident* is used, where *ident* is the record name.

If *sort* is omitted, the default sort is Type. Notice that the type of an identifier can depend on a previously-given identifier. Thus the order of the fields is important. *binder* parameters may be applied to the record as a whole or to individual fields.

> ? If provided, the constructor name is automatically declared as a coercion from the class of the last field type to the record name (this may fail if the uniform inheritance condition is not satisfied). See *Implicit Coercions*.

Notations can be attached to fields using the *decl_notations* annotation.

Record and *Structure* are synonyms.

This command supports the *universes(polymorphic)*, *universes(template)*, *universes(cumulative)*, and *private(matching)* attributes.

More generally, a record may have explicitly defined (a.k.a. manifest) fields. For instance, we might have: **Record** *ident* *binder** **:** *sort* **:=** **{** $ident_1$ **:** $type_1$ **;** $ident_2$ **:=** $term_2$ **;** $ident_3$ **:** $type_3$ **}**. in which case the correctness of $type_3$ may rely on the instance $term_2$ of $ident_2$ and $term_2$ may in turn depend on $ident_1$.

Example

The set of rational numbers may be defined as:

```
Record Rat : Set := mkRat
 { sign : bool
 ; top : nat
 ; bottom : nat
 ; Rat_bottom_cond : 0 <> bottom
 ; Rat_irred_cond :
     forall x y z:nat, (x * y) = top /\ (x * z) = bottom -> x = 1
 }.
    Rat is defined
    sign is defined
    top is defined
    bottom is defined
    Rat_bottom_cond is defined
    Rat_irred_cond is defined
```

Note here that the fields `Rat_bottom_cond` depends on the field `bottom` and `Rat_irred_cond` depends on both `top` and `bottom`.

Let us now see the work done by the `Record` macro. First the macro generates a variant type definition with just one constructor: **Variant** *ident* *binder** **:** *sort* **:=** $ident_0$ *binder** **.**

To build an object of type *ident*, provide the constructor $ident_0$ with the appropriate number of terms filling the fields of the record.

Example

Let us define the rational $1/2$:

```
Theorem one_two_irred : forall x y z:nat, x * y = 1 /\ x * z = 2 -> x = 1.
Admitted.

Definition half := mkRat true 1 2 (O_S 1) one_two_irred.
Check half.
```

Alternatively, the following syntax allows creating objects by using named fields, as shown in this grammar. The fields do not have to be in any particular order, nor do they have to be all present if the missing ones can be inferred or prompted for (see *Program*).

```
Definition half' :=
  {| sign := true;
     Rat_bottom_cond := O_S 1;
     Rat_irred_cond := one_two_irred |}.
    half' is defined
```

The following settings let you control the display format for types:

Flag: `Printing Records`
When this *flag* is on (this is the default), use the record syntax (shown above) as the default display format.

You can override the display format for specified types by adding entries to these tables:

Table: `Printing Record` *qualid*
This *table* specifies a set of qualids which are displayed as records. Use the *Add* and *Remove* commands to update the set of qualids.

Table: `Printing Constructor` *qualid*
This *table* specifies a set of qualids which are displayed as constructors. Use the *Add* and *Remove* commands to update the set of qualids.

This syntax can also be used for pattern matching.

```
Eval compute in (
  match half with
  | {| sign := true; top := n |} => n
  | _ => 0
  end).
    = 1
         : nat
```

The macro generates also, when it is possible, the projection functions for destructuring an object of type *ident*. These projection functions are given the names of the corresponding fields. If a field is named _ then no projection is built for it. In our example:

```
Eval compute in top half.
    = 1
         : nat

Eval compute in bottom half.
    = 2
         : nat

Eval compute in Rat_bottom_cond half.
    = O_S 1
         : 0 <> bottom half
```

An alternative syntax for projections based on a dot notation is available:

```
Eval compute in half.(top).
    = 1
         : nat
```

Flag: `Printing Projections`
This *flag* activates the dot notation for printing.

Example

```
Set Printing Projections.
Check top half.
    half.(top)
         : nat
```

---

term_projection ::= *term0* .(*qualid* *arg**)
| *term0* .(@ *qualid* *term1**)

Syntax of Record projections

The corresponding grammar rules are given in the preceding grammar. When *qualid* denotes a projection, the syntax ***term0*.(*qualid*)** is equivalent to ***qualid term0***, the syntax ***term0*.(*qualid* *arg*+)** to ***qualid* *arg*+ *term0***. and the syntax ***term0*.(@*qualid* *term0*+)** to **@***qualid* *term0*+ *term0***. In each case, *term0* is the object projected and the other arguments are the parameters of the inductive type.

Note: Records defined with the `Record` keyword are not allowed to be recursive (references to the record's name in the type of its field raises an error). To define recursive records, one can use the `Inductive` and `CoInductive` keywords, resulting in an inductive or co-inductive record. Definition of mutually inductive or co-inductive records are also allowed, as long as all of the types in the block are records.

Note: Induction schemes are automatically generated for inductive records. Automatic generation of induction schemes for non-recursive records defined with the `Record` keyword can be activated with the *Nonrecursive Elimination Schemes* flag (see *Generation of induction principles with Scheme*).

Warning: *ident* cannot be defined.
It can happen that the definition of a projection is impossible. This message is followed by an explanation of this impossibility. There may be three reasons:

1. The name *ident* already exists in the global environment (see *Axiom*).
2. The *body* of *ident* uses an incorrect elimination for *ident* (see *Fixpoint* and *Destructors*).
3. The type of the projections *ident* depends on previous projections which themselves could not be defined.

Error:
Records declared with the keyword Record or Structure cannot be recursive.
The record name *ident* appears in the type of its fields, but uses the keyword `Record`. Use the keyword `Inductive` or `CoInductive` instead.

Error: Cannot handle mutually (co)inductive records.
Records cannot be defined as part of mutually inductive (or co-inductive) definitions, whether with records only or mixed with standard definitions.

During the definition of the one-constructor inductive definition, all the errors of inductive definitions, as described in Section *Inductive types*, may also occur.

See also:

Coercions and records in section *Classes as Records* of the chapter devoted to coercions.

Primitive Projections

Flag: `Primitive Projections`
This *flag* turns on the use of primitive projections when defining subsequent records (even through the `Inductive` and `CoInductive` commands). Primitive projections extended the Calculus of Inductive Constructions with a new binary term constructor `r.(p)` representing a primitive projection `p` applied to a record object `r` (i.e., primitive projections are always applied). Even if the record type has parameters, these do not appear in the internal representation of applications of the projection, considerably reducing the sizes of terms when manipulating parameterized records and type checking time. On the user level, primitive projections can be used as a replacement for the usual defined ones, although there are a few notable differences.

Flag: `Printing Primitive Projection Parameters`
This compatibility *flag* reconstructs internally omitted parameters at printing time (even though they are absent in the actual AST manipulated by the kernel).

Primitive Record Types

When the *`Primitive Projections`* flag is on, definitions of record types change meaning. When a type is declared with primitive projections, its `match` construct is disabled (see *Primitive Projections* though). To eliminate the (co-)inductive type, one must use its defined primitive projections.

For compatibility, the parameters still appear to the user when printing terms even though they are absent in the actual AST manipulated by the kernel. This can be changed by unsetting the *`Printing Primitive Projection Parameters`* flag.

There are currently two ways to introduce primitive records types:

1. Through the `Record` command, in which case the type has to be non-recursive. The defined type enjoys eta-conversion definitionally, that is the generalized form of surjective pairing for records: `r = Build_R (r.(`p_1`) ... r.(`p_n`))`. Eta-conversion allows to define dependent elimination for these types as well.
2. Through the `Inductive` and `CoInductive` commands, when the *body* of the definition is a record declaration of the form `Build_R {` p_1 `:` t_1`; ... ;` p_n `:` t_n `}`. In this case the types can be recursive and eta-conversion is disallowed. These kind of record types differ from their traditional versions in the sense that dependent elimination is not available for them and only non-dependent case analysis can be defined.

Reduction

The basic reduction rule of a primitive projection is p_i `(Build_R` t_1 ... t_n`)` $\rightarrow_\iota$ t_i. However, to take the δ flag into account, projections can be in two states: folded or unfolded. An unfolded primitive projection application obeys the rule above, while the folded version delta-reduces to the unfolded version. This allows to precisely mimic the usual unfolding rules of *constants*. Projections obey the usual `simpl` flags of the `Arguments` command in particular. There is currently no way to input unfolded primitive projections at the user-level, and there is no way to display unfolded projections differently from folded ones.

Compatibility Projections and `match`

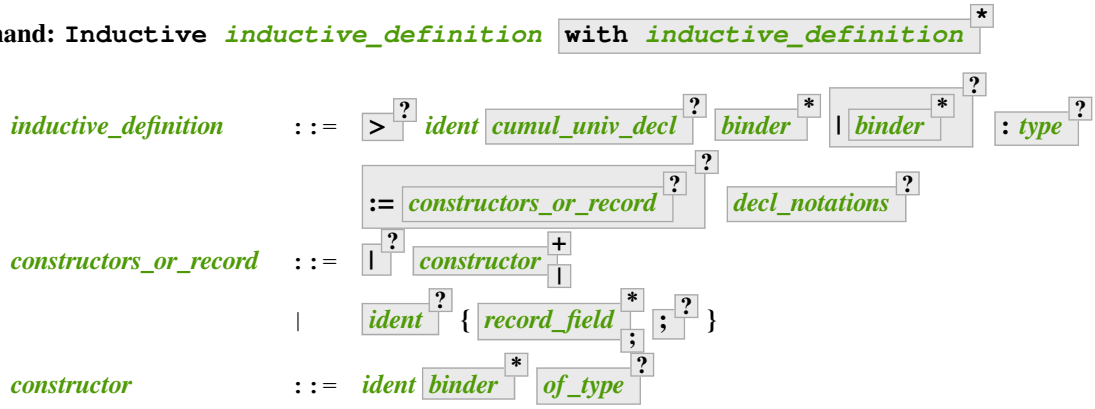
To ease compatibility with ordinary record types, each primitive projection is also defined as an ordinary *constant* taking parameters and an object of the record type as arguments, and whose *body* is an application of the unfolded primitive projection of the same name. These constants are used when elaborating partial applications of the projection. One can distinguish them from applications of the primitive projection if the `Printing Primitive Projection Parameters` flag is off: For a primitive projection application, parameters are printed as underscores while for the compatibility projections they are printed as usual.

Additionally, user-written `match` constructs on primitive records are desugared into substitution of the projections, they cannot be printed back as `match` constructs.

2.1.9 Inductive types and recursive functions

Inductive types

Command: **`Inductive`** *inductive_definition* **`with`** *inductive_definition*



This command defines one or more inductive types and its constructors. Coq generates destructors depending on the universe that the inductive type belongs to.

The destructors are named ***ident*`_rect`**, ***ident*`_ind`**, ***ident*`_rec`** and ***ident*`_sind`**, which respectively correspond to elimination principles on `Type`, `Prop`, `Set` and `SProp`. The type of the destructors expresses structural induction/recursion principles over objects of type ***ident***. The *constant* ***ident*`_ind`** is always generated, whereas ***ident*`_rec`** and ***ident*`_rect`** may be impossible to derive (for example, when ***ident*** is a proposition).

This command supports the *`universes(polymorphic)`*, *`universes(template)`*, *`universes(cumulative)`*, *`bypass_check(positivity)`*, *`bypass_check(universes)`*, and *`private(matching)`* attributes.

Mutually inductive types can be defined by including multiple ***inductive_definition*s**. The ***ident*s** are simultaneously added to the global environment before the types of constructors are checked. Each ***ident*** can be used independently thereafter. See *Mutually defined inductive types*.

If the entire inductive definition is parameterized with ***binder*s**, the parameters correspond to a local context in which the entire set of inductive declarations is interpreted. For this reason, the parameters must be strictly the same for each inductive type. See *Parameterized inductive types*.

Constructor ***ident*s** can come with ***binder*s**, in which case the actual type of the constructor is **`forall`** ***binder*** **`,`** ***type***.

Error: **`Non strictly positive occurrence of`** ***ident*** **`in`** ***type*`.`**

The types of the constructors have to satisfy a *positivity condition* (see Section *Positivity Condition*). This

condition ensures the soundness of the inductive definition. Positivity checking can be disabled using the *Positivity Checking* flag or the *bypass_check(positivity)* attribute (see *Controlling Typing Flags*).

Error: The conclusion of *type* is not valid; it must be built from *ident*.
The conclusion of the type of the constructors must be the inductive type *ident* being defined (or *ident* applied to arguments in the case of annotated inductive types — cf. next section).

The following subsections show examples of simple inductive types, simple annotated inductive types, simple parametric inductive types, mutually inductive types and private (matching) inductive types.

Simple inductive types

A simple inductive type belongs to a universe that is a simple *sort*.

Example

The set of natural numbers is defined as:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
    nat is defined
    nat_rect is defined
    nat_ind is defined
    nat_rec is defined
    nat_sind is defined
```

The type nat is defined as the least `Set` containing `O` and closed by the `S` constructor. The names `nat`, `O` and `S` are added to the global environment.

This definition generates four elimination principles: `nat_rect`, `nat_ind`, `nat_rec` and `nat_sind`. The type of `nat_ind` is:

```
Check nat_ind.
    nat_ind
         : forall P : nat -> Prop,
           P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

This is the well known structural induction principle over natural numbers, i.e. the second-order form of Peano's induction principle. It allows proving universal properties of natural numbers (`forall n:nat, P n`) by induction on `n`.

The types of `nat_rect`, `nat_rec` and `nat_sind` are similar, except that they apply to, respectively, `(P:nat->Type)`, `(P:nat->Set)` and `(P:nat->SProp)`. They correspond to primitive induction principles (allowing dependent types) respectively over sorts `Type, `Set and `SProp.

In the case where inductive types don't have annotations (the next section gives an example of annotations), a constructor can be defined by giving the type of its arguments alone.

Example

```
Inductive nat : Set := O | S (_:nat).
```

Simple annotated inductive types

In annotated inductive types, the universe where the inductive type is defined is no longer a simple ***sort***, but what is called an arity, which is a type whose conclusion is a ***sort***.

Example

As an example of annotated inductive types, let us define the `even` predicate:

```
Inductive even : nat -> Prop :=
| even_0 : even O
| even_SS : forall n:nat, even n -> even (S (S n)).
    even is defined
    even_ind is defined
    even_sind is defined
```

The type `nat->Prop` means that `even` is a unary predicate (inductively defined) over natural numbers. The type of its two constructors are the defining clauses of the predicate `even`. The type of `even_ind` is:

```
Check even_ind.
    even_ind
         : forall P : nat -> Prop,
           P O ->
           (forall n : nat, even n -> P n -> P (S (S n))) ->
           forall n : nat, even n -> P n
```

From a mathematical point of view, this asserts that the natural numbers satisfying the predicate `even` are exactly in the smallest set of naturals satisfying the clauses `even_0` or `even_SS`. This is why, when we want to prove any predicate `P` over elements of `even`, it is enough to prove it for `O` and to prove that if any natural number `n` satisfies `P` its double successor `(S (S n))` satisfies also `P`. This is analogous to the structural induction principle we got for `nat`.

Parameterized inductive types

In the previous example, each constructor introduces a different instance of the predicate `even`. In some cases, all the constructors introduce the same generic instance of the inductive definition, in which case, instead of an annotation, we use a context of parameters which are ***binder***s shared by all the constructors of the definition.

Parameters differ from inductive type annotations in that the conclusion of each type of constructor invokes the inductive type with the same parameter values of its specification.

Example

A typical example is the definition of polymorphic lists:

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined
```

In the type of `nil` and `cons`, we write "`list A`" and not just "`list`". The constructors `nil` and `cons` have these types:

```
Check nil.
    nil
         : forall A : Set, list A

Check cons.
    cons
         : forall A : Set, A -> list A -> list A
```

Observe that the destructors are also quantified with `(A:Set)`, for example:

```
Check list_ind.
    list_ind
         : forall (A : Set) (P : list A -> Prop),
           P (nil A) ->
           (forall (a : A) (l : list A), P l -> P (cons A a l)) ->
           forall l : list A, P l
```

Once again, the names of the constructor arguments and the type of the conclusion can be omitted:

```
Inductive list (A:Set) : Set := nil | cons (_:A) (_:list A).
```

Note:

- The constructor type can recursively invoke the inductive definition on an argument which is not the parameter itself.

  One can define :

  ```
  Inductive list2 (A:Set) : Set :=
  | nil2 : list2 A
  | cons2 : A -> list2 (A*A) -> list2 A.
      list2 is defined
      list2_rect is defined
      list2_ind is defined
      list2_rec is defined
      list2_sind is defined
  ```

  that can also be written by specifying only the type of the arguments:

  ```
  Inductive list2 (A:Set) : Set :=
  | nil2
  | cons2 (_:A) (_:list2 (A*A)).
      list2 is defined
      list2_rect is defined
      list2_ind is defined
      list2_rec is defined
      list2_sind is defined
  ```

  But the following definition will give an error:

  ```
  Fail Inductive listw (A:Set) : Set :=
  | nilw : listw (A*A)
  | consw : A -> listw (A*A) -> listw (A*A).
      The command has indeed failed with message:
      In environment
      listw : Set -> Set
  ```

(continues on next page)

(continued from previous page)

```
A : Set
Unable to unify "listw (A * A)%type" with "listw A".
```

because the conclusion of the type of constructors should be `listw A` in both cases.

- A parameterized inductive definition can be defined using annotations instead of parameters but it will sometimes give a different (bigger) sort for the inductive definition and will produce a less convenient rule for case elimination.

Flag: `Uniform Inductive Parameters`
When this *flag* is set (it is off by default), inductive definitions are abstracted over their parameters before type checking constructors, allowing to write:

```
Set Uniform Inductive Parameters.
Inductive list3 (A:Set) : Set :=
| nil3 : list3
| cons3 : A -> list3 -> list3.
    list3 is defined
    list3_rect is defined
    list3_ind is defined
    list3_rec is defined
    list3_sind is defined
```

This behavior is essentially equivalent to starting a new section and using *`Context`* to give the uniform parameters, like so (cf. *Section mechanism*):

```
Section list3.
Context (A:Set).
    A is declared

Inductive list3 : Set :=
| nil3 : list3
| cons3 : A -> list3 -> list3.
    list3 is defined
    list3_rect is defined
    list3_ind is defined
    list3_rec is defined
    list3_sind is defined

End list3.
```

For finer control, you can use a `|` between the uniform and the non-uniform parameters:

```
Inductive Acc {A:Type} (R:A->A->Prop) | (x:A) : Prop
  := Acc_in : (forall y, R y x -> Acc y) -> Acc x.
```

The flag can then be seen as deciding whether the `|` is at the beginning (when the flag is unset) or at the end (when it is set) of the parameters when not explicitly given.

See also:

Section *Theory of inductive definitions* and the *`induction`* tactic.

Mutually defined inductive types

Example: Mutually defined inductive types

A typical example of mutually inductive data types is trees and forests. We assume two types `A` and `B` that are given as variables. The types can be declared like this:

```
Parameters A B : Set.

Inductive tree : Set := node : A -> forest -> tree

with forest : Set :=
| leaf : B -> forest
| cons : tree -> forest -> forest.
```

This declaration automatically generates eight induction principles. They are not the most general principles, but they correspond to each inductive part seen as a single inductive definition.

To illustrate this point on our example, here are the types of `tree_rec` and `forest_rec`.

```
Check tree_rec.
    tree_rec
         : forall P : tree -> Set,
           (forall (a : A) (f : forest), P (node a f)) -> forall t : tree, P t


Check forest_rec.
    forest_rec
         : forall P : forest -> Set,
           (forall b : B, P (leaf b)) ->
           (forall (t : tree) (f0 : forest), P f0 -> P (cons t f0)) ->
           forall f1 : forest, P f1
```

Assume we want to parameterize our mutual inductive definitions with the two type variables `A` and `B`, the declaration should be done as follows:

```
Inductive tree (A B:Set) : Set := node : A -> forest A B -> tree A B

with forest (A B:Set) : Set :=
| leaf : B -> forest A B
| cons : tree A B -> forest A B -> forest A B.
```

Assume we define an inductive definition inside a section (cf. *Section mechanism*). When the section is closed, the variables declared in the section and occurring free in the declaration are added as parameters to the inductive definition.

See also:

A generic command *Scheme* is useful to build automatically various mutual induction principles.

Recursive functions: fix

term_fix : := **let fix** *fix_decl* **in** *term*

| **fix** *fix_decl* (**with** *fix_decl*)+ **for** *ident* ?

fix_decl : := *ident* *binder** *fixannot*? **:** *type*? **:=** *term*

fixannot : := **{ struct** *ident* **}**

| **{ wf** *one_term ident* **}**

| **{ measure** *one_term* *ident*? *one_term*? **}**

The expression "**fix** $ident_1$ $binder_1$ **:** $type_1$ **:=** $term_1$ **with … with** $ident_n$ $binder_n$ **:** $type_n$ **:=** $term_n$ **for** $ident_i$" denotes the i-th component of a block of functions defined by mutual structural recursion. It is the local counterpart of the `Fixpoint` command. When $n = 1$, the "**for** $ident_i$" clause is omitted.

The association of a single fixpoint and a local definition have a special syntax: **let fix** *ident* *binder** **:= term in** stands for **let** *ident* **:= fix** *ident* *binder** **:= term in**. The same applies for co-fixpoints.

Some options of ***fixannot*** are only supported in specific constructs. **fix** and **let fix** only support the **struct** option, while **wf** and **measure** are only supported in commands such as `Fixpoint` (with the `program` attribute) and `Function`.

Top-level recursive functions

This section describes the primitive form of definition by recursion over inductive objects. See the `Function` command for more advanced constructions.

Command: Fixpoint *fix_definition* **with** *fix_definition**

fix_definition : := *ident_decl* *binder** *fixannot*? **:** *type*? **:=** *term*? *decl_notations*?

Allows defining functions by pattern matching over inductive objects using a fixed point construction. The meaning of this declaration is to define *ident* as a recursive function with arguments specified by the *binder*s such that *ident* applied to arguments corresponding to these *binder*s has type *type*, and is equivalent to the expression *term*. The type of *ident* is consequently **forall** *binder** **,** *type* and its value is equivalent to **fun** *binder** **=>** *term*.

This command accepts the `program`, `bypass_check(universes)`, and `bypass_check(guard)` attributes.

To be accepted, a `Fixpoint` definition has to satisfy syntactical constraints on a special argument called the decreasing argument. They are needed to ensure that the `Fixpoint` definition always terminates. The point of the **{struct** *ident***}** annotation (see ***fixannot***) is to let the user tell the system which argument decreases along the recursive calls.

The **{struct** *ident***}** annotation may be left implicit, in which case the system successively tries arguments from left to right until it finds one that satisfies the decreasing condition.

`Fixpoint` without the `program` attribute does not support the **wf** or **measure** clauses of ***fixannot***. See *Program Fixpoint*.

The **with** clause allows simultaneously defining several mutual fixpoints. It is especially useful when defining functions over mutually defined inductive types. Example: *Mutual Fixpoints*.

If *term* is omitted, *type* is required and Coq enters proof mode. This can be used to define a term incrementally, in particular by relying on the `refine` tactic. In this case, the proof should be terminated with `Defined` in order to define a *constant* for which the computational behavior is relevant. See *Entering and exiting proof mode*.

This command accepts the `using` attribute.

Note:

- Some fixpoints may have several arguments that fit as decreasing arguments, and this choice influences the reduction of the fixpoint. Hence an explicit annotation must be used if the leftmost decreasing argument is not the desired one. Writing explicit annotations can also speed up type checking of large mutual fixpoints.
- In order to keep the strong normalization property, the fixed point reduction will only be performed when the argument in position of the decreasing argument (which type should be in an inductive definition) starts with a constructor.

Example

One can define the addition function as :

```
Fixpoint add (n m:nat) {struct n} : nat :=
match n with
| O => m
| S p => S (add p m)
end.
    add is defined
    add is recursively defined (guarded on 1st argument)
```

The match operator matches a value (here `n`) with the various constructors of its (inductive) type. The remaining arguments give the respective values to be returned, as functions of the parameters of the corresponding constructor. Thus here when `n` equals `O` we return `m`, and when `n` equals `(S p)` we return `(S (add p m))`.

The match operator is formally described in Section *The match ... with ... end construction*. The system recognizes that in the inductive call `(add p m)` the first argument actually decreases because it is a *pattern variable* coming from `match n with`.

Example

The following definition is not correct and generates an error message:

```
Fail Fixpoint wrongplus (n m:nat) {struct n} : nat :=
match m with
| O => n
| S p => S (wrongplus n p)
end.
    The command has indeed failed with message:
    Recursive definition of wrongplus is ill-formed.
    In environment
    wrongplus : nat -> nat -> nat
    n : nat
    m : nat
    p : nat
```

(continues on next page)

(continued from previous page)

```
    Recursive call to wrongplus has principal argument equal to
    "n" instead of a subterm of "n".
    Recursive definition is:
    "fun n m : nat => match m with
                      | O => n
                      | S p => S (wrongplus n p)
                      end".
```

because the declared decreasing argument `n` does not actually decrease in the recursive call. The function computing the addition over the second argument should rather be written:

```
Fixpoint plus (n m:nat) {struct m} : nat :=
match m with
| O => n
| S p => S (plus n p)
end.
    plus is defined
    plus is recursively defined (guarded on 2nd argument)
```

Example

The recursive call may not only be on direct subterms of the recursive variable `n` but also on a deeper subterm and we can directly write the function `mod2` which gives the remainder modulo 2 of a natural number.

```
Fixpoint mod2 (n:nat) : nat :=
match n with
| O => O
| S p => match p with
         | O => S O
         | S q => mod2 q
         end
end.
    mod2 is defined
    mod2 is recursively defined (guarded on 1st argument)
```

Example: Mutual fixpoints

The size of trees and forests can be defined the following way:

```
Fixpoint tree_size (t:tree) : nat :=
match t with
| node a f => S (forest_size f)
end
with forest_size (f:forest) : nat :=
match f with
| leaf b => 1
| cons t f' => (tree_size t + forest_size f')
end.
    tree_size is defined
    forest_size is defined
    tree_size, forest_size are recursively defined (guarded respectively on␣
 ↪1st,
    1st arguments)
```

Theory of inductive definitions

Formally, we can represent any *inductive definition* as $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$ where:

- Γ_I determines the names and types of inductive types;
- Γ_C determines the names and types of constructors of these inductive types;
- p determines the number of parameters of these inductive types.

These inductive definitions, together with global assumptions and global definitions, then form the global environment. Additionally, for any p there always exists $\Gamma_P = [a_1 : A_1;\ ...;\ a_p : A_p]$ such that each T in $(t : T) \in \Gamma_I \cup \Gamma_C$ can be written as: $\forall \Gamma_P, T'$ where Γ_P is called the *context of parameters*. Furthermore, we must have that each T in $(t : T) \in \Gamma_I$ can be written as: $\forall \Gamma_P, \forall \Gamma_{Arr(t)}, S$ where $\Gamma_{Arr(t)}$ is called the *Arity* of the inductive type t and S is called the sort of the inductive type t (not to be confused with $\mathcal{S}$ which is the set of sorts).

Example

The declaration for parameterized lists is:

$$\mathsf{Ind}\ [1]\left([\mathsf{list} : \mathsf{Set} \to \mathsf{Set}]\ :=\ \begin{bmatrix} \mathsf{nil} & : & \forall A : \mathsf{Set},\ \mathsf{list}\ A \\ \mathsf{cons} & : & \forall A : \mathsf{Set},\ A \to \mathsf{list}\ A \to \mathsf{list}\ A \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

Example

The declaration for a mutual inductive definition of tree and forest is:

$$\mathsf{Ind}\ [0]\left(\begin{bmatrix} \mathsf{tree} & : & \mathsf{Set} \\ \mathsf{forest} & : & \mathsf{Set} \end{bmatrix}\ :=\ \begin{bmatrix} \mathsf{node} & : & \mathsf{forest} \to \mathsf{tree} \\ \mathsf{emptyf} & : & \mathsf{forest} \\ \mathsf{consf} & : & \mathsf{tree} \to \mathsf{forest} \to \mathsf{forest} \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive tree : Set :=
| node : forest -> tree
with forest : Set :=
| emptyf : forest
| consf : tree -> forest -> forest.
```

Example

The declaration for a mutual inductive definition of even and odd is:

$$\mathsf{Ind}\ [0]\left(\begin{bmatrix} \mathsf{even} & : & \mathsf{nat} \to \mathsf{Prop} \\ \mathsf{odd} & : & \mathsf{nat} \to \mathsf{Prop} \end{bmatrix}\ :=\ \begin{bmatrix} \mathsf{even}_\mathsf{O} & : & \mathsf{even}\ 0 \\ \mathsf{even}_\mathsf{S} & : & \forall n,\ \mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{S}\ n) \\ \mathsf{odd}_\mathsf{S} & : & \forall n,\ \mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{S}\ n) \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive even : nat -> Prop :=
| even_O : even 0
| even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
| odd_S : forall n, even n -> odd (S n).
```

Types of inductive objects

We have to give the type of constants in a global environment E which contains an inductive definition.

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \qquad (a : A) \in \Gamma_I}{E[\Gamma] \vdash a : A}$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \qquad (c : C) \in \Gamma_C}{E[\Gamma] \vdash c : C}$$

Example

Provided that our global environment E contains inductive definitions we showed before, these two inference rules above enable us to conclude that:

$$E[\Gamma] \vdash \mathsf{even} : \mathsf{nat} \to \mathsf{Prop}$$
$$E[\Gamma] \vdash \mathsf{odd} : \mathsf{nat} \to \mathsf{Prop}$$
$$E[\Gamma] \vdash \mathsf{even}_{\mathsf{O}} : \mathsf{even}\ \mathsf{O}$$
$$E[\Gamma] \vdash \mathsf{even}_{\mathsf{S}} : \forall n : \mathsf{nat},\ \mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{S}\ n)$$
$$E[\Gamma] \vdash \mathsf{odd}_{\mathsf{S}} : \forall n : \mathsf{nat},\ \mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{S}\ n)$$

Well-formed inductive definitions

We cannot accept any inductive definition because some of them lead to inconsistent systems. We restrict ourselves to definitions which satisfy a syntactic criterion of positivity. Before giving the formal rules, we need a few definitions:

Arity of a given sort

A type T is an *arity of sort* s if it converts to the sort s or to a product $\forall x : T,\ U$ with U an arity of sort s.

Example

$A \to \mathsf{Set}$ is an arity of sort Set. $\forall A : \mathsf{Prop},\ A \to \mathsf{Prop}$ is an arity of sort Prop.

Arity

A type T is an *arity* if there is a $s \in \mathcal{S}$ such that T is an arity of sort s.

Example

$A \to \mathsf{Set}$ and $\forall A : \mathsf{Prop},\ A \to \mathsf{Prop}$ are arities.

Type of constructor

We say that T is a *type of constructor of* I in one of the following two cases:

- T is $(I\ t_1 ... t_n)$
- T is $\forall x : U,\ T'$ where T' is also a type of constructor of I

Example

nat and $\mathsf{nat} \to \mathsf{nat}$ are types of constructor of nat. $\forall A : \mathsf{Type},\ \mathsf{list}\ A$ and $\forall A : \mathsf{Type},\ A \to \mathsf{list}\ A \to \mathsf{list}\ A$ are types of constructor of list.

Positivity Condition

The type of constructor T will be said to *satisfy the positivity condition* for a constant X in the following cases:

- $T = (X\ t_1 ... t_n)$ and X does not occur free in any t_i
- $T = \forall x : U,\ V$ and X occurs only strictly positively in U and the type V satisfies the positivity condition for X.

Strict positivity

The constant X *occurs strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X\ t_1 ... t_n)$ and X does not occur in any of t_i
- T converts to $\forall x : U,\ V$ and X does not occur in type U but occurs strictly positively in type V
- T converts to $(I\ a_1 ... a_m\ t_1 ... t_p)$ where I is the name of an inductive definition of the form

$$\mathsf{Ind}\ [m]\ (I : A\ :=\ c_1 : \forall p_1 : P_1, ... \forall p_m : P_m,\ C_1;\ ...;\ c_n : \forall p_1 : P_1, ... \forall p_m : P_m,\ C_n)$$

(in particular, it is not mutually defined and it has m parameters) and X does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1...m}$ of I satisfy the nested positivity condition for X

Nested Positivity

The type of constructor T of I *satisfies the nested positivity condition* for a constant X in the following cases:

- $T = (I\ b_1 ... b_m\ u_1 ... u_p)$, I is an inductive type with m parameters and X does not occur in any u_i
- $T = \forall x : U,\ V$ and X occurs only strictly positively in U and the type V satisfies the nested positivity condition for X

Example

For instance, if one considers the following variant of a tree type branching over the natural numbers:

```
Inductive nattree (A:Type) : Type :=
| leaf : nattree A
| natnode : A -> (nat -> nattree A) -> nattree A.
```

Then every instantiated constructor of `nattree A` satisfies the nested positivity condition for `nattree`:

- Type `nattree A` of constructor `leaf` satisfies the positivity condition for `nattree` because `nattree` does not appear in any (real) arguments of the type of that constructor (primarily because `nattree` does not have any (real) arguments) ... (bullet 1)
- Type `A → (nat → nattree A) → nattree A` of constructor `natnode` satisfies the positivity condition for `nattree` because:
  - `nattree` occurs only strictly positively in `A` ... (bullet 1)
  - `nattree` occurs only strictly positively in `nat → nattree A` ... (bullet 3 + 2)
  - `nattree` satisfies the positivity condition for `nattree A` ... (bullet 1)

Correctness rules

We shall now describe the rules allowing the introduction of a new inductive definition.

Let E be a global environment and Γ_P, Γ_I, Γ_C be contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1;\ ...;\ I_k : \forall \Gamma_P, A_k]$, and Γ_C is $[c_1 : \forall \Gamma_P, C_1;\ ...;\ c_n : \forall \Gamma_P, C_n]$. Then

W-Ind

$$\frac{\mathcal{WF}(E)[\Gamma_P] \qquad (E[\Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1...n}}{\mathcal{WF}(E;\ \mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C))[]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1...k$ and $i = 1...n$,
- p is the number of parameters of $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1...k$ we have that A_j is an arity of sort s_j and $I_j \notin E$,
- for $i = 1...n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 ... I_k$ and $c_i \notin E$.

One can remark that there is a constraint between the sort of the arity of the inductive type and the sort of the type of its constructors which will always be satisfied for the impredicative sorts SProp and Prop but may fail to define inductive type on sort Set and generate constraints between universes for inductive types in the Type hierarchy.

Example

It is well known that the existential quantifier can be encoded as an inductive definition. The following declaration introduces the second-order existential quantifier $\exists X.P(X)$.

```
Inductive exProp (P:Prop->Prop) : Prop :=
| exP_intro : forall X:Prop, P X -> exProp P.
```

The same definition on Set is not allowed and fails:

```
Fail Inductive exSet (P:Set->Prop) : Set :=
exS_intro : forall X:Set, P X -> exSet P.
    The command has indeed failed with message:
    Large non-propositional inductive types must be in Type.
```

It is possible to declare the same inductive definition in the universe Type. The `exType` inductive definition has type $(\mathsf{Type}(i) \to \mathsf{Prop}) \to \mathsf{Type}(j)$ with the constraint that the parameter X of $\mathsf{exT}_{\mathsf{intro}}$ has type $\mathsf{Type}(k)$ with $k < j$ and $k \leq i$.

```
Inductive exType (P:Type->Prop) : Type :=
exT_intro : forall X:Type, P X -> exType P.
    exType is defined
    exType_rect is defined
    exType_ind is defined
    exType_rec is defined
    exType_sind is defined
```

Example: Negative occurrence (first example)

The following inductive definition is rejected because it does not satisfy the positivity condition:

```
Fail Inductive I : Prop := not_I_I (not_I : I -> False) : I.
    The command has indeed failed with message:
    Non strictly positive occurrence of "I" in "(I -> False) -> I".
```

If we were to accept such definition, we could derive a contradiction from it (we can test this by disabling the *Positivity Checking* flag):

```
Definition I_not_I : I -> ~ I := fun i =>
  match i with not_I_I not_I => not_I end.
    I_not_I is defined

Lemma contradiction : False.
Proof.
  enough (I /\ ~ I) as [] by contradiction.
  split.
  - apply not_I_I.
    intro.
    now apply I_not_I.
  - intro.
    now apply I_not_I.
Qed.
```

Example: Negative occurrence (second example)

Here is another example of an inductive definition which is rejected because it does not satify the positivity condition:

```
Fail Inductive Lam := lam (_ : Lam -> Lam).
    The command has indeed failed with message:
    Non strictly positive occurrence of "Lam" in "(Lam -> Lam) -> Lam".
```

Again, if we were to accept it, we could derive a contradiction (this time through a non-terminating recursive function):

```
Fixpoint infinite_loop l : False :=
  match l with lam x => infinite_loop (x l) end.
    infinite_loop is defined
    infinite_loop is recursively defined (guarded on 1st argument)


Check infinite_loop (lam (@id Lam)) : False.
    infinite_loop (lam (id (A:=Lam))) : False
         : False
```

Example: Non strictly positive occurrence

It is less obvious why inductive type definitions with occurences that are positive but not strictly positive are harmful. We will see that in presence of an impredicative type they are unsound:

```
Fail Inductive A: Type := introA: ((A -> Prop) -> Prop) -> A.
    The command has indeed failed with message:
    Non strictly positive occurrence of "A" in "((A -> Prop) -> Prop) -> A".
```

If we were to accept this definition we could derive a contradiction by creating an injective function from $A \to \mathsf{Prop}$ to A.

This function is defined by composing the injective constructor of the type A with the function $\lambda x.\lambda z.z = x$ injecting any type T into $T \to \mathsf{Prop}$.

```
Definition f (x: A -> Prop): A := introA (fun z => z = x).
    f is defined

Lemma f_inj: forall x y, f x = f y -> x = y.
Proof.
  unfold f; intros ? ? H; injection H.
  set (F := fun z => z = y); intro HF.
  symmetry; replace (y = x) with (F y).
  + unfold F; reflexivity.
  + rewrite <- HF; reflexivity.
Qed.
```

The type $A \to \mathsf{Prop}$ can be understood as the powerset of the type A. To derive a contradiction from the injective function f we use Cantor's classic diagonal argument.

```
Definition d: A -> Prop := fun x => exists s, x = f s /\ ~s x.
    d is defined

Definition fd: A := f d.
    fd is defined

Lemma cantor: (d fd) <-> ~(d fd).
Proof.
```

(continues on next page)

(continued from previous page)

```
  split.
  + intros [s [H1 H2]]; unfold fd in H1.
    replace d with s.
    * assumption.
    * apply f_inj; congruence.
  + intro; exists d; tauto.
Qed.

Lemma bad: False.
Proof.
  pose cantor; tauto.
Qed.
```

This derivation was first presented by Thierry Coquand and Christine Paulin in [CP90].

Template polymorphism

Inductive types can be made polymorphic over the universes introduced by their parameters in Type, if the minimal inferred sort of the inductive declarations either mention some of those parameter universes or is computed to be Prop or Set.

If A is an arity of some sort and s is a sort, we write $A_{/s}$ for the arity obtained from A by replacing its sort with s. Especially, if A is well-typed in some global environment and local context, then $A_{/s}$ is typable by typability of all products in the Calculus of Inductive Constructions. The following typing rule is added to the theory.

Let $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ be an inductive definition. Let $\Gamma_P = [p_1 : P_1;\ ...;\ p_p : P_p]$ be its context of parameters, $\Gamma_I = [I_1 : \forall \Gamma_P, A_1;\ ...;\ I_k : \forall \Gamma_P, A_k]$ its context of definitions and $\Gamma_C = [c_1 : \forall \Gamma_P, C_1;\ ...;\ c_n : \forall \Gamma_P, C_n]$ its context of constructors, with c_i a constructor of I_{q_i}. Let $m \leq p$ be the length of the longest prefix of parameters such that the m first arguments of all occurrences of all I_j in all C_k (even the occurrences in the hypotheses of C_k) are exactly applied to $p_1 ... p_m$ (m is the number of *recursively uniform parameters* and the $p - m$ remaining parameters are the *recursively non-uniform parameters*). Let $q_1, ..., q_r$, with $0 \leq r \leq m$, be a (possibly) partial instantiation of the recursively uniform parameters of Γ_P. We have:

Ind-Family

$$\frac{\left\{\begin{array}{l}\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E\\ (E[] \vdash q_l : P'_l)_{l=1...r}\\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1...l-1})_{l=1...r}\\ 1 \leq j \leq k\end{array}\right.}{E[] \vdash I_j\ q_1 ... q_r : \forall [p_{r+1} : P_{r+1};\ ...;\ p_p : P_p], (A_j)_{/s_j}}$$

provided that the following side conditions hold:

- $\Gamma_{P'}$ is the context obtained from Γ_P by replacing each P_l that is an arity with P'_l for $1 \leq l \leq r$ (notice that P_l arity implies P'_l arity since $E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1...l-1}$);
- there are sorts s_i, for $1 \leq i \leq k$ such that, for $\Gamma_{I'} = [I_1 : \forall \Gamma_{P'}, (A_1)_{/s_1};\ ...;\ I_k : \forall \Gamma_{P'}, (A_k)_{/s_k}]$ we have $(E[\Gamma_{I'}; \Gamma_{P'}] \vdash C_i : s_{q_i})_{i=1...n}$;
- the sorts s_i are all introduced by the inductive declaration and have no universe constraints beside being greater than or equal to Prop, and such that all eliminations, to Prop, Set and Type(j), are allowed (see Section *Destructors*).

Notice that if $I_j\ q_1 ... q_r$ is typable using the rules **Ind-Const** and **App**, then it is typable using the rule **Ind-Family**. Conversely, the extended theory is not stronger than the theory without **Ind-Family**. We get an equiconsistency result

by mapping each $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ occurring into a given derivation into as many different inductive types and constructors as the number of different (partial) replacements of sorts, needed for this derivation, in the parameters that are arities (this is possible because $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$ well-formed implies that $\mathsf{Ind}\,[p]\,(\Gamma_{I'} := \Gamma_{C'})$ is well-formed and has the same allowed eliminations, where $\Gamma_{I'}$ is defined as above and $\Gamma_{C'} = [c_1 : \forall \Gamma_{P'}, C_1;\ ...;\ c_n : \forall \Gamma_{P'}, C_n]$). That is, the changes in the types of each partial instance $q_1 ... q_r$ can be characterized by the ordered sets of arity sorts among the types of parameters, and to each signature is associated a new inductive definition with fresh names. Conversion is preserved as any (partial) instance $I_j\ q_1 ... q_r$ or $C_i\ q_1 ... q_r$ is mapped to the names chosen in the specific instance of $\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C)$.

Warning: The restriction that sorts are introduced by the inductive declaration prevents inductive types declared in sections to be template-polymorphic on universes introduced previously in the section: they cannot parameterize over the universes introduced with section variables that become parameters at section closing time, as these may be shared with other definitions from the same section which can impose constraints on them.

Flag: `Auto Template Polymorphism`

This *flag*, enabled by default, makes every inductive type declared at level Type (without annotations or hiding it behind a definition) template polymorphic if possible.

This can be prevented using the `universes(template=no)` attribute.

Template polymorphism and full universe polymorphism (see Chapter *Polymorphic Universes*) are incompatible, so if the latter is enabled (through the `Universe Polymorphism` flag or the `universes(polymorphic)` attribute) it will prevail over automatic template polymorphism.

Warning: `Automatically declaring` *`ident`* `as template polymorphic.`

Warning `auto-template` can be used (it is off by default) to find which types are implicitly declared template polymorphic by `Auto Template Polymorphism`.

An inductive type can be forced to be template polymorphic using the `universes(template)` attribute: in this case, the warning is not emitted.

Attribute: `universes(template` `=` `yes` | `no` `?` `)`

This *boolean attribute* can be used to explicitly declare an inductive type as template polymorphic, whether the `Auto Template Polymorphism` flag is on or off.

Error: `template and polymorphism not compatible`

This attribute cannot be used in a full universe polymorphic context, i.e. if the `Universe Polymorphism` flag is on or if the `universes(polymorphic)` attribute is used.

Error:
`Ill-formed template inductive declaration: not polymorphic on any universe.`

The attribute was used but the inductive definition does not satisfy the criterion to be template polymorphic.

When `universes(template=no)` is used, it will prevent an inductive type to be template polymorphic, even if the `Auto Template Polymorphism` flag is on.

Attribute: `universes(notemplate)`

Deprecated since version 8.13: Use `universes(template=no)` instead.

In practice, the rule **Ind-Family** is used by Coq only when all the inductive types of the inductive definition are declared with an arity whose sort is in the Type hierarchy. Then, the polymorphism is over the parameters whose type is an arity of sort in the Type hierarchy. The sorts s_j are chosen canonically so that each s_j is minimal with respect to the hierarchy $\mathsf{Prop} \subset \mathsf{Set}_p \subset \mathsf{Type}$ where Set_p is predicative Set. More precisely, an empty or small singleton inductive definition (i.e. an inductive definition of which all inductive types are singleton – see Section *Destructors*) is set in Prop, a small non-singleton inductive type is set in Set (even in case Set is impredicative – see *The Calculus of Inductive Constructions with impredicative Set*), and otherwise in the Type hierarchy.

Note that the side-condition about allowed elimination sorts in the rule **Ind-Family** avoids to recompute the allowed elimination sorts at each instance of a pattern matching (see Section *Destructors*). As an example, let us consider the following definition:

Example

```
Inductive option (A:Type) : Type :=
| None : option A
| Some : A -> option A.
```

As the definition is set in the Type hierarchy, it is used polymorphically over its parameters whose types are arities of a sort in the Type hierarchy. Here, the parameter A has this property, hence, if `option` is applied to a type in Set, the result is in Set. Note that if `option` is applied to a type in Prop, then, the result is not set in Prop but in Set still. This is because `option` is not a singleton type (see Section *Destructors*) and it would lose the elimination to Set and Type if set in Prop.

Example

```
Check (fun A:Set => option A).
    fun A : Set => option A
         : Set -> Set

Check (fun A:Prop => option A).
    fun A : Prop => option A
         : Prop -> Set
```

Here is another example.

Example

```
Inductive prod (A B:Type) : Type := pair : A -> B -> prod A B.
```

As `prod` is a singleton type, it will be in Prop if applied twice to propositions, in Set if applied twice to at least one type in Set and none in Type, and in Type otherwise. In all cases, the three kind of eliminations schemes are allowed.

Example

```
Check (fun A:Set => prod A).
    fun A : Set => prod A
         : Set -> Type -> Type

Check (fun A:Prop => prod A A).
    fun A : Prop => prod A A
         : Prop -> Prop

Check (fun (A:Prop) (B:Set) => prod A B).
    fun (A : Prop) (B : Set) => prod A B
         : Prop -> Set -> Set

Check (fun (A:Type) (B:Prop) => prod A B).
    fun (A : Type) (B : Prop) => prod A B
         : Type -> Prop -> Type
```

Note: Template polymorphism used to be called "sort-polymorphism of inductive types" before universe polymorphism (see Chapter *Polymorphic Universes*) was introduced.

Destructors

The specification of inductive definitions with arities and constructors is quite natural. But we still have to say how to use an object in an inductive type.

This problem is rather delicate. There are actually several different ways to do that. Some of them are logically equivalent but not always equivalent from the computational point of view or from the user point of view.

From the computational point of view, we want to be able to define a function whose domain is an inductively defined type by using a combination of case analysis over the possible constructors of the object and recursion.

Because we need to keep a consistent theory and also we prefer to keep a strongly normalizing reduction, we cannot accept any sort of recursion (even terminating). So the basic idea is to restrict ourselves to primitive recursive functions and functionals.

For instance, assuming a parameter $A : \mathsf{Set}$ exists in the local context, we want to build a function length of type $\mathsf{list}\ A \to \mathsf{nat}$ which computes the length of the list, such that $(\mathsf{length}\ (\mathsf{nil}\ A)) = \mathsf{O}$ and $(\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)) = (\mathsf{S}\ (\mathsf{length}\ l))$. We want these equalities to be recognized implicitly and taken into account in the conversion rule.

From the logical point of view, we have built a type family by giving a set of constructors. We want to capture the fact that we do not have any other way to build an object in this type. So when trying to prove a property about an object m in an inductive type it is enough to enumerate all the cases where m starts with a different constructor.

In case the inductive definition is effectively a recursive one, we want to capture the extra property that we have built the smallest fixed point of this recursive equation. This says that we are only manipulating finite objects. This analysis provides induction principles. For instance, in order to prove $\forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l))$ it is enough to prove:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ (\mathsf{length}\ (\mathsf{nil}\ A)))$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)))$

which given the conversion equalities satisfied by length is the same as proving:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ \mathsf{O})$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ (\mathsf{length}\ l)))$

One conceptually simple way to do that, following the basic scheme proposed by Martin-Löf in his Intuitionistic Type Theory, is to introduce for each inductive definition an elimination operator. At the logical level it is a proof of the usual induction principle and at the computational level it implements a generic operator for doing primitive recursion over the structure.

But this operator is rather tedious to implement and use. We choose in this version of Coq to factorize the operator for primitive recursion into two more primitive operations as was first suggested by Th. Coquand in [Coq92]. One is the definition by pattern matching. The second one is a definition by guarded fixpoints.

The match ... with ... end construction

The basic idea of this operator is that we have an object m in an inductive type I and we want to prove a property which possibly depends on m. For this, it is enough to prove the property for $m = (c_i\ u_1 ... u_{p_i})$ for each constructor of I. The Coq term for this proof will be written:

$$\mathsf{match}\ m\ \mathsf{with}\ (c_1\ x_{11} ... x_{1p_1}) \Rightarrow f_1 | ... | (c_n\ x_{n1} ... x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

In this expression, if m eventually happens to evaluate to $(c_i\ u_1 ... u_{p_i})$ then the expression will behave as specified in its i-th branch and it will reduce to f_i where the $x_{i1} ... x_{ip_i}$ are replaced by the $u_1 ... u_{p_i}$ according to the ι-reduction.

Actually, for type checking a match...with...end expression we also need to know the predicate P to be proved by case analysis. In the general case where I is an inductively defined n-ary relation, P is a predicate over $n + 1$ arguments: the n first ones correspond to the arguments of I (parameters excluded), and the last one corresponds to object m. Coq can sometimes infer this predicate but sometimes not. The concrete syntax for describing this predicate uses the as...in...return construction. For instance, let us assume that I is an unary predicate with one parameter and one argument. The predicate is made explicit using the syntax:

$$\mathsf{match}\ m\ \mathsf{as}\ x\ \mathsf{in}\ I\ \_\ a\ \mathsf{return}\ P\ \mathsf{with}\ (c_1\ x_{11} ... x_{1p_1}) \Rightarrow f_1 | ... | (c_n\ x_{n1} ... x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

The as part can be omitted if either the result type does not depend on m (non-dependent elimination) or m is a variable (in this case, m can occur in P where it is considered a bound variable). The in part can be omitted if the result type does not depend on the arguments of I. Note that the arguments of I corresponding to parameters *must* be _, because the result type is not generalized to all possible values of the parameters. The other arguments of I (sometimes called indices in the literature) have to be variables (a above) and these variables can occur in P. The expression after in must be seen as an *inductive type pattern*. Notice that expansion of implicit arguments and notations apply to this pattern. For the purpose of presenting the inference rules, we use a more compact notation:

$$\mathsf{case}(m, (\lambda a x.P), \lambda x_{11} ... x_{1p_1}.f_1\ |...|\ \lambda x_{n1} ... x_{np_n}.f_n)$$

Allowed elimination sorts. An important question for building the typing rule for match is what can be the type of $\lambda ax.P$ with respect to the type of m. If $m : I$ and $I : A$ and $\lambda ax.P : B$ then by $[I : A|B]$ we mean that one can use $\lambda ax.P$ with m in the above match-construct.

Notations. The $[I : A|B]$ is defined as the smallest relation satisfying the following rules: We write $[I|B]$ for $[I : A|B]$ where A is the type of I.

The case of inductive types in sorts Set or Type is simple. There is no restriction on the sort of the predicate to be eliminated.

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

The case of Inductive definitions of sort Prop is a bit more complicated, because of our interpretation of this sort. The only harmless allowed eliminations, are the ones when predicate P is also of sort Prop or is of the morally smaller sort SProp.

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop}|I \to s]}$$

Prop is the type of logical propositions, the proofs of properties P in Prop could not be used for computation and are consequently ignored by the extraction mechanism. Assume A and B are two propositions, and the logical disjunction $A \vee B$ is defined inductively by:

Example

```
Inductive or (A B:Prop) : Prop :=
or_introl : A -> or A B | or_intror : B -> or A B.
```

The following definition which computes a boolean value by case over the proof of `or A B` is not accepted:

Example

```
Fail Definition choice (A B: Prop) (x:or A B) :=
match x with or_introl _ _ a => true | or_intror _ _ b => false end.
    The command has indeed failed with message:
    Incorrect elimination of "x" in the inductive type "or":
    the return type has sort "Set" while it should be "SProp" or "Prop".
    Elimination of an inductive object of sort Prop
    is not allowed on a predicate in sort Set
    because proofs can be eliminated only to build proofs.
```

From the computational point of view, the structure of the proof of `(or A B)` in this term is needed for computing the boolean value.

In general, if I has type Prop then P cannot have type $I \to$ Set, because it will mean to build an informative proof of type $(P\ m)$ doing a case analysis over a non-computational object that will disappear in the extracted program. But the other way is safe with respect to our interpretation we can have I a computational object and P a non-computational one, it just corresponds to proving a logical property of a computational object.

In the same spirit, elimination on P of type $I \to$ Type cannot be allowed because it trivially implies the elimination on P of type $I \to$ Set by cumulativity. It also implies that there are two proofs of the same property which are provably different, contradicting the proof-irrelevance property which is sometimes a useful axiom:

Example

```
Axiom proof_irrelevance : forall (P : Prop) (x y : P), x=y.
    proof_irrelevance is declared
```

The elimination of an inductive type of sort Prop on a predicate P of type $I \to$ Type leads to a paradox when applied to impredicative inductive definition like the second-order existential quantifier `exProp` defined above, because it gives access to the two projections on this type.

Empty and singleton elimination. There are special inductive definitions in Prop for which more eliminations are allowed.

Prop-extended

$$\frac{I \text{ is an empty or singleton definition} \qquad s \in \mathcal{S}}{[I : \mathsf{Prop} | I \to s]}$$

A *singleton definition* has only one constructor and all the arguments of this constructor have type Prop. In that case, there is a canonical way to interpret the informative extraction on an object in that type, such that the elimination on any sort s

is legal. Typical examples are the conjunction of non-informative propositions and the equality. If there is a hypothesis $h : a = b$ in the local context, it can be used for rewriting not only in logical propositions but also in any type.

Example

```
Print eq_rec.
    eq_rec =
    fun (A : Type) (x : A) (P : A -> Set) => eq_rect x P
         : forall (A : Type) (x : A) (P : A -> Set),
           P x -> forall y : A, x = y -> P y

    Arguments eq_rec [A]%type_scope _ _%function_scope _ _ _

Require Extraction.
    [Loading ML file extraction_plugin.cmxs ... done]

Extraction eq_rec.
    (** val eq_rec : 'a1 -> 'a2 -> 'a1 -> 'a2 **)

    let eq_rec _ f _ =
      f
```

An empty definition has no constructors, in that case also, elimination on any sort is allowed.

Inductive types in SProp must have no constructors (i.e. be empty) to be eliminated to produce relevant values.

Note that thanks to proof irrelevance elimination functions can be produced for other types, for instance the elimination for a unit type is the identity.

Type of branches. Let c be a term of type C, we assume C is a type of constructor for an inductive type I. Let P be a term that represents the property to be proved. We assume r is the number of parameters and s is the number of arguments.

We define a new type $\{c : C\}^P$ which represents the type of the branch corresponding to the $c : C$ constructor.

$$\begin{array}{ll} \{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P & \equiv (P\ t_1 \ldots\ t_s\ c) \\ \{c : \forall x : T,\ C\}^P & \equiv \forall x : T,\ \{(c\ x) : C\}^P \end{array}$$

We write $\{c\}^P$ for $\{c : C\}^P$ with C the type of c.

Example

The following term in concrete syntax:

```
match t as l return P' with
| nil _ => t1
| cons _ hd tl => t2
end
```

can be represented in abstract syntax as

$$\mathsf{case}(t, P, f_1 | f_2)$$

where

$$\begin{array}{rcl} P & = & \lambda l.\ P' \\ f_1 & = & t_1 \\ f_2 & = & \lambda(hd : \mathsf{nat}).\ \lambda(tl : \mathsf{list\ nat}).\ t_2 \end{array}$$

According to the definition:

$$\{(\mathsf{nil\ nat})\}^P \equiv \{(\mathsf{nil\ nat}) : (\mathsf{list\ nat})\}^P \equiv (P\ (\mathsf{nil\ nat}))$$

$$\begin{aligned}\{(\mathsf{cons\ nat})\}^P &\equiv \{(\mathsf{cons\ nat}) : (\mathsf{nat} \to \mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ &\equiv \forall n : \mathsf{nat},\ \{(\mathsf{cons\ nat}\ n) : (\mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ &\equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ \{(\mathsf{cons\ nat}\ n\ l) : (\mathsf{list\ nat})\}^P \\ &\equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ (P\ (\mathsf{cons\ nat}\ n\ l)).\end{aligned}$$

Given some P then $\{(\mathsf{nil\ nat})\}^P$ represents the expected type of f_1, and $\{(\mathsf{cons\ nat})\}^P$ represents the expected type of f_2.

Typing rule. Our very general destructor for inductive definition enjoys the following typing rule

match

$$\frac{\begin{array}{l}E[\Gamma] \vdash c : (I\ q_1 ... q_r\ t_1 ... t_s) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 ... q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 ... q_r)\}^P)_{i=1...l}\end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|...|f_l) : (P\ t_1 ... t_s\ c)}$$

provided I is an inductive type in a definition $\mathsf{Ind}\ [r]\ (\Gamma_I\ :=\ \Gamma_C)$ with $\Gamma_C = [c_1 : C_1;\ ...;\ c_n : C_n]$ and $c_{p_1} ... c_{p_l}$ are the only constructors of I.

Example

Below is a typing rule for the term shown in the previous example:

list example

$$\frac{\begin{array}{l}E[\Gamma] \vdash t : (\mathsf{list\ nat}) \\ E[\Gamma] \vdash P : B \\ [(\mathsf{list\ nat})|B] \\ E[\Gamma] \vdash f_1 : \{(\mathsf{nil\ nat})\}^P \\ E[\Gamma] \vdash f_2 : \{(\mathsf{cons\ nat})\}^P\end{array}}{E[\Gamma] \vdash \mathsf{case}(t, P, f_1|f_2) : (P\ t)}$$

Definition of ι-reduction. We still have to define the ι-reduction in the general case.

An ι-redex is a term of the following form:

$$\mathsf{case}((c_{p_i}\ q_1 ... q_r\ a_1 ... a_m), P, f_1|...|f_l)$$

with c_{p_i} the i-th constructor of the inductive type I with r parameters.

The ι-contraction of this term is $(f_i\ a_1 ... a_m)$ leading to the general reduction rule:

$$\mathsf{case}((c_{p_i}\ q_1 ... q_r\ a_1 ... a_m), P, f_1|...|f_l) \triangleright_\iota (f_i\ a_1 ... a_m)$$

Fixpoint definitions

The second operator for elimination is fixpoint definition. This fixpoint may involve several mutually recursive definitions. The basic concrete syntax for a recursive set of mutually recursive declarations is (with Γ_i contexts):

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with ... with}\ f_n(\Gamma_n) : A_n := t_n$$

The terms are obtained by projections from this set of declarations and are written

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with ... with}\ f_n(\Gamma_n) : A_n := t_n\ \mathsf{for}\ f_i$$

In the inference rules, we represent such a term by

$$\mathsf{Fix}\ f_i\{f_1 : A'_1 := t'_1 ... f_n : A'_n := t'_n\}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalized) with respect to the bindings in the context Γ_i, namely $t'_i = \lambda\Gamma_i.t_i$ and $A'_i = \forall\Gamma_i, A_i$.

Typing rule

The typing rule is the expected one for a fixpoint.

Fix

$$\frac{(E[\Gamma] \vdash A_i : s_i)_{i=1...n} \qquad (E[\Gamma;\ f_1 : A_1;\ ...;\ f_n : A_n] \vdash t_i : A_i)_{i=1...n}}{E[\Gamma] \vdash \mathsf{Fix}\ f_i\{f_1 : A_1 := t_1 ... f_n : A_n := t_n\} : A_i}$$

Any fixpoint definition cannot be accepted because non-normalizing terms allow proofs of absurdity. The basic scheme of recursion that should be allowed is the one needed for defining primitive recursive functionals. In that case the fixpoint enjoys a special syntactic restriction, namely one of the arguments belongs to an inductive type, the function starts with a case analysis and recursive calls are done on variables coming from patterns and representing subterms. For instance in the case of natural numbers, a proof of the induction principle of type

$$\forall P : \mathsf{nat} \to \mathsf{Prop},\ (P\ \mathsf{O}) \to (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))) \to \forall n : \mathsf{nat},\ (P\ n)$$

can be represented by the term:

$$\lambda P : \mathsf{nat} \to \mathsf{Prop}.\ \lambda f : (P\ \mathsf{O}).\ \lambda g : (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))).$$
$$\mathsf{Fix}\ h\{h : \forall n : \mathsf{nat},\ (P\ n) := \lambda n : \mathsf{nat}.\ \mathsf{case}(n, P, f | \lambda p : \mathsf{nat}.\ (g\ p\ (h\ p)))\}$$

Before accepting a fixpoint definition as being correctly typed, we check that the definition is "guarded". A precise analysis of this notion can be found in [Gimenez94]. The first stage is to precise on which argument the fixpoint will be decreasing. The type of this argument should be an inductive type. For doing this, the syntax of fixpoints is extended and becomes

$$\mathsf{Fix}\ f_i\{f_1/k_1 : A_1 := t_1 ... f_n/k_n : A_n := t_n\}$$

where k_i are positive integers. Each k_i represents the index of parameter of f_i, on which f_i is decreasing. Each A_i should be a type (reducible to a term) starting with at least k_i products $\forall y_1 : B_1,\ ...\forall y_{k_i} : B_{k_i},\ A'_i$ and B_{k_i} an inductive type.

Now in the definition t_i, if f_j occurs then it should be applied to at least k_j arguments and the k_j-th argument should be syntactically recognized as structurally smaller than y_{k_i}.

The definition of being structurally smaller is a bit technical. One needs first to define the notion of *recursive arguments of a constructor*. For an inductive definition $\mathsf{Ind}\ [r]\ (\Gamma_I := \Gamma_C)$, if the type of a constructor c has the form $\forall p_1 : P_1,\ ...\forall p_r : P_r,\ \forall x_1 : T_1,\ ...\forall x_m : T_m,\ (I_j\ p_1 ... p_r\ t_1 ... t_s)$, then the recursive arguments will correspond to T_i in which one of the I_l occurs.

The main rules for being structurally smaller are the following. Given a variable y of an inductively defined type in a declaration $\mathsf{Ind}\ [r]\ (\Gamma_I := \Gamma_C)$ where Γ_I is $[I_1 : A_1;\ ...;\ I_k : A_k]$, and Γ_C is $[c_1 : C_1;\ ...;\ c_n : C_n]$, the terms structurally smaller than y are:

- $(t\ u)$ and $\lambda x : U.\ t$ when t is structurally smaller than y.
- case$(c, P, f_1 ... f_n)$ when each f_i is structurally smaller than y. If c is y or is structurally smaller than y, its type is an inductive type I_p part of the inductive definition corresponding to y. Each f_i corresponds to a type of constructor $C_q \equiv \forall p_1 : P_1,\ ..., \forall p_r : P_r,\ \forall y_1 : B_1,\ ... \forall y_m : B_m,\ (I_p\ p_1 ... p_r\ t_1 ... t_s)$ and can consequently be written $\lambda y_1 : B'_1.\ ... \lambda y_m : B'_m.\ g_i$. ($B'_i$ is obtained from B_i by substituting parameters for variables) the variables y_j occurring in g_i corresponding to recursive arguments B_i (the ones in which one of the I_l occurs) are structurally smaller than y.

The following definitions are correct, we enter them using the *Fixpoint* command and show the internal representation.

Example

```
Fixpoint plus (n m:nat) {struct n} : nat :=
match n with
| O => m
| S p => S (plus p m)
end.
    plus is defined
    plus is recursively defined (guarded on 1st argument)


Print plus.
    plus =
    fix plus (n m : nat) {struct n} : nat :=
      match n with
      | 0 => m
      | S p => S (plus p m)
      end
         : nat -> nat -> nat

    Arguments plus (_ _)%nat_scope

Fixpoint lgth (A:Set) (l:list A) {struct l} : nat :=
match l with
| nil _ => O
| cons _ a l' => S (lgth A l')
end.
    lgth is defined
    lgth is recursively defined (guarded on 2nd argument)

Print lgth.
    lgth =
    fix lgth (A : Set) (l : list A) {struct l} : nat :=
      match l with
      | nil _ => 0
      | cons _ _ l' => S (lgth A l')
      end
         : forall A : Set, list A -> nat

    Arguments lgth _%type_scope _

Fixpoint sizet (t:tree) : nat := let (f) := t in S (sizef f)
with sizef (f:forest) : nat :=
match f with
| emptyf => O
| consf t f => plus (sizet t) (sizef f)
end.
```

(continues on next page)

(continued from previous page)

```
    sizet is defined
    sizef is defined
    sizet, sizef are recursively defined (guarded respectively on 1st,
    1st arguments)

Print sizet.
    sizet =
    fix sizet (t : tree) : nat :=
      let (f) := t in S (sizef f)
    with sizef (f : forest) : nat :=
      match f with
      | emptyf => 0
      | consf t f0 => plus (sizet t) (sizef f0)
      end
    for
    sizet
         : tree -> nat
```

Reduction rule

Let F be the set of declarations: $f_1/k_1 : A_1 := t_1 ... f_n/k_n : A_n := t_n$. The reduction for fixpoints is:

$$(\mathsf{Fix}\ f_i\{F\}\ a_1 ... a_{k_i}) \triangleright_\iota t_i\{f_k/\mathsf{Fix}\ f_k\{F\}\}_{k=1...n}\ a_1 ... a_{k_i}$$

when a_{k_i} starts with a constructor. This last restriction is needed in order to keep strong normalization and corresponds to the reduction for primitive recursive operators. The following reductions are now possible:

$$\begin{aligned} \mathsf{plus}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O}))\ (\mathsf{S}\ \mathsf{O}) &\triangleright_\iota \mathsf{S}\ (\mathsf{plus}\ (\mathsf{S}\ \mathsf{O})\ (\mathsf{S}\ \mathsf{O})) \\ &\triangleright_\iota \mathsf{S}\ (\mathsf{S}\ (\mathsf{plus}\ \mathsf{O}\ (\mathsf{S}\ \mathsf{O}))) \\ &\triangleright_\iota \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) \end{aligned}$$

Mutual induction

The principles of mutual induction can be automatically generated using the Scheme command described in Section *Generation of induction principles with Scheme*.

2.1.10 Co-inductive types and co-recursive functions

Co-inductive types

The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a *finite* number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors. Infinite objects are introduced by a non-ending (but effective) process of construction, defined in terms of the constructors of the type.

More information on co-inductive definitions can be found in [Gimenez95, Gimenez98, GimenezCasteran05].

Command: **`CoInductive`** ***`inductive_definition`*** **`with`** ***`inductive_definition`*** `*`

This command introduces a co-inductive type. The syntax of the command is the same as the command *`Inductive`*. No principle of induction is derived from the definition of a co-inductive type, since such principles only make sense for inductive types. For co-inductive types, the only elimination principle is case analysis.

This command supports the *universes(polymorphic)*, *universes(template)*, *universes(cumulative)*, *private(matching)*, *bypass_check(universes)*, *bypass_check(positivity)*, and *using* attributes.

Example

The type of infinite sequences of natural numbers, usually called streams, is an example of a co-inductive type.

```
CoInductive Stream : Set := Seq : nat -> Stream -> Stream.
```

The usual destructors on streams `hd:Stream->nat` and `tl:Str->Str` can be defined as follows:

```
Definition hd (x:Stream) := let (a,s) := x in a.
Definition tl (x:Stream) := let (a,s) := x in s.
```

Definitions of co-inductive predicates and blocks of mutually co-inductive definitions are also allowed.

Example

The extensional equality on streams is an example of a co-inductive type:

```
CoInductive EqSt : Stream -> Stream -> Prop :=
  eqst : forall s1 s2:Stream,
           hd s1 = hd s2 -> EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

In order to prove the extensional equality of two streams `s1` and `s2` we have to construct an infinite proof of equality, that is, an infinite object of type `(EqSt s1 s2)`. We will see how to introduce infinite objects in Section *Top-level definitions of co-recursive functions*.

Caveat

The ability to define co-inductive types by constructors, hereafter called *positive co-inductive types*, is known to break subject reduction. The story is a bit long: this is due to dependent pattern-matching which implies propositional η-equality, which itself would require full η-conversion for subject reduction to hold, but full η-conversion is not acceptable as it would make type checking undecidable.

Since the introduction of primitive records in Coq 8.5, an alternative presentation is available, called *negative co-inductive types*. This consists in defining a co-inductive type as a primitive record type through its projections. Such a technique is akin to the *co-pattern* style that can be found in e.g. Agda, and preserves subject reduction.

The above example can be rewritten in the following way.

```
Set Primitive Projections.
CoInductive Stream : Set := Seq { hd : nat; tl : Stream }.
    Stream is defined
    hd is defined
    tl is defined

CoInductive EqSt (s1 s2: Stream) : Prop := eqst {
  eqst_hd : hd s1 = hd s2;
  eqst_tl : EqSt (tl s1) (tl s2);
}.
    EqSt is defined
```

(continues on next page)

(continued from previous page)

```
    eqst_hd is defined
    eqst_tl is defined
```

Some properties that hold over positive streams are lost when going to the negative presentation, typically when they imply equality over streams. For instance, propositional η-equality is lost when going to the negative presentation. It is nonetheless logically consistent to recover it through an axiom.

```
Axiom Stream_eta : forall s: Stream, s = Seq (hd s) (tl s).
    Stream_eta is declared
```

More generally, as in the case of positive coinductive types, it is consistent to further identify extensional equality of coinductive types with propositional equality:

```
Axiom Stream_ext : forall (s1 s2: Stream), EqSt s1 s2 -> s1 = s2.
    Stream_ext is declared
```

As of Coq 8.9, it is now advised to use negative co-inductive types rather than their positive counterparts.

See also:

Primitive Projections for more information about negative records and primitive projections.

Co-recursive functions: cofix

term_cofix ::= **let cofix** *cofix_body* **in** *term*

| **cofix** *cofix_body* (**with** *cofix_body*)+ **for** *ident* ?

cofix_body ::= *ident* *binder** **:** *type* ? **:=** *term*

The expression "**cofix** $ident_1$ $binder_1$ **:** $type_1$ **with … with** $ident_n$ $binder_n$ **:** $type_n$ **for** $ident_i$" denotes the i-th component of a block of terms defined by a mutual guarded co-recursion. It is the local counterpart of the `CoFixpoint` command. When $n = 1$, the "**for** $ident_i$" clause is omitted.

Top-level definitions of co-recursive functions

Command: CoFixpoint *cofix_definition* (**with** *cofix_definition*)*

cofix_definition ::= *ident_decl* *binder** (**:** *type* ?) ? (**:=** *term*) ? *decl_notations* ?

This command introduces a method for constructing an infinite object of a coinductive type. For example, the stream containing all natural numbers can be introduced applying the following method to the number `O` (see Section *Co-inductive types* for the definition of `Stream`, `hd` and `tl`):

```
CoFixpoint from (n:nat) : Stream := Seq n (from (S n)).
    from is defined
    from is corecursively defined
```

Unlike recursive definitions, there is no decreasing argument in a co-recursive definition. To be admissible, a method of construction must provide at least one extra constructor of the infinite object for each iteration. A syntactical guard condition is imposed on co-recursive definitions in order to ensure this: each recursive call in the definition must be protected by at least one constructor, and only by constructors. That is the case in the former definition,

where the single recursive call of `from` is guarded by an application of `Seq`. On the contrary, the following recursive function does not satisfy the guard condition:

```
Fail CoFixpoint filter (p:nat -> bool) (s:Stream) : Stream :=
  if p (hd s) then Seq (hd s) (filter p (tl s)) else filter p (tl s).
    The command has indeed failed with message:
    Recursive definition of filter is ill-formed.
    In environment
    filter : (nat -> bool) -> Stream -> Stream
    p : nat -> bool
    s : Stream
    Unguarded recursive call in "filter p (tl s)".
    Recursive definition is:
    "fun (p : nat -> bool) (s : Stream) =>
     if p (hd s)
     then {| hd := hd s; tl := filter p (tl s) |}
     else filter p (tl s)".
```

The elimination of co-recursive definition is done lazily, i.e. the definition is expanded only when it occurs at the head of an application which is the argument of a case analysis expression. In any other context, it is considered as a canonical expression which is completely evaluated. We can test this using the command *Eval*, which computes the normal forms of a term:

```
Eval compute in (from 0).
    = (cofix from (n : nat) : Stream := {| hd := n; tl := from (S n) |}) 0
         : Stream

Eval compute in (hd (from 0)).
    = 0
         : nat

Eval compute in (tl (from 0)).
    = (cofix from (n : nat) : Stream := {| hd := n; tl := from (S n) |}) 1
         : Stream
```

As in the *Fixpoint* command, the **with** clause allows simultaneously defining several mutual cofixpoints.

If ***term*** is omitted, ***type*** is required and Coq enters proof mode. This can be used to define a term incrementally, in particular by relying on the *refine* tactic. In this case, the proof should be terminated with *Defined* in order to define a *constant* for which the computational behavior is relevant. See *Entering and exiting proof mode*.

2.1.11 Section mechanism

Sections are naming scopes that permit creating section-local declarations that can be used by other declarations in the section. Declarations made with *Variable*, *Hypothesis*, *Context*, *Let*, *Let Fixpoint* and *Let CoFixpoint* (or the plural variants of the first two) within sections are local to the section.

In proofs done within the section, section-local declarations are included in the *local context* of the initial goal of the proof. They are also accessible in definitions made with the *Definition* command.

Sections are opened by the *Section* command, and closed by *End*. Sections can be nested. When a section is closed, its local declarations are no longer available. Global declarations that refer to them will be adjusted so they're still usable outside the section as shown in this *example*.

Command: `Section` *ident*

Opens the section named *ident*. Section names do not need to be unique.

Command: **`End`** ***`ident`***

Closes the section or module named *ident*. See *Terminating an interactive module or module type definition* for a description of its use with modules.

After closing the section, the local declarations (variables and local definitions, see *Variable*) are *discharged*, meaning that they stop being visible and that all global objects defined in the section are generalized with respect to the variables and local definitions they each depended on in the section.

Error: **`There is nothing to end.`**

Error: **`Last block to end has name`** ***`ident`***`.`

Note: Most commands, such as the *Hint* commands, *Notation* and option management commands that appear inside a section are canceled when the section is closed.

Command: **`Let`** ***`ident_decl def_body`***

Command: **`Let Fixpoint`** ***`fix_definition`*** **`with`** ***`fix_definition`*** `*`

Command: **`Let CoFixpoint`** ***`cofix_definition`*** **`with`** ***`cofix_definition`*** `*`

These are similar to *Definition*, *Fixpoint* and *CoFixpoint*, except that the declared *constant* is local to the current section. When the section is closed, all persistent definitions and theorems within it that depend on the constant will be wrapped with a ***`term_let`*** with the same declaration.

As for *Definition*, *Fixpoint* and *CoFixpoint*, if ***`term`*** is omitted, ***`type`*** is required and Coq enters proof mode. This can be used to define a term incrementally, in particular by relying on the *refine* tactic. In this case, the proof should be terminated with *Defined* in order to define a constant for which the computational behavior is relevant. See *Entering and exiting proof mode*.

Command: **`Context`** ***`binder`*** `+`

Declare variables in the context of the current section, like *Variable*, but also allowing implicit variables, *Implicit generalization*, and let-binders.

```
Context {A : Type} (a b : A).
Context `{EqDec A}.
Context (b' := b).
```

See also:

Section *Binders*. Section *Sections and contexts* in chapter *Typeclasses*.

Example: Section-local declarations

```
Section s1.

Variables x y : nat.
    x is declared
    y is declared
```

The command *Let* introduces section-wide *Let-in definitions*. These definitions won't persist when the section is closed, and all persistent definitions which depend on `y'` will be prefixed with `let y' := y in`.

```
Let y' := y.
Definition x' := S x.
Definition x'' := x' + y'.
```

```
Print x'.
    x' = S x
         : nat

Print x''.
    x'' = x' + y'
         : nat


End s1.

Print x'.
    x' = fun x : nat => S x
         : nat -> nat

    Arguments x' _%nat_scope

Print x''.
    x'' = fun x y : nat => let y' := y in x' x + y'
         : nat -> nat -> nat

    Arguments x'' (_ _)%nat_scope
```

Notice the difference between the value of `x'` and `x''` inside section `s1` and outside.

2.1.12 The Module System

The module system extends the Calculus of Inductive Constructions providing a convenient way to structure large developments as well as a means of massive abstraction.

Modules and module types

Access path. An access path is denoted by p and can be either a module variable X or, if p' is an access path and id an identifier, then $p'.id$ is an access path.

Structure element. A structure element is denoted by e and is either a definition of a *constant*, an assumption, a definition of an inductive, a definition of a module, an alias of a module or a module type abbreviation.

Structure expression. A structure expression is denoted by S and can be:

- an access path p
- a plain structure Struct e; ...; e End
- a functor Functor$(X : S)\ S'$, where X is a module variable, S and S' are structure expressions
- an application $S\ p$, where S is a structure expression and p an access path
- a refined structure S with $p := p'$ or S with $p := t : T$ where S is a structure expression, p and p' are access paths, t is a term and T is the type of t.

Module definition. A module definition is written Mod$(X : S\,[:= S'])$ and consists of a module variable X, a module type S which can be any structure expression and optionally a module implementation S' which can be any structure expression except a refined structure.

Module alias. A module alias is written ModA$(X == p)$ and consists of a module variable X and a module path p.

Module type abbreviation. A module type abbreviation is written ModType($Y := S$), where Y is an identifier and S is any structure expression .

Using modules

The module system provides a way of packaging related elements together, as well as a means of massive abstraction.

Command:

Module (**Import** | **Export**)? *ident* *module_binder** *of_module_type*? (**:=** *module_expr_inl*)?

module_binder ::= ((**Import** | **Export**)? *ident*+ **:** *module_type_inl*)
module_type_inl ::= **!** *module_type*
| *module_type* *functor_app_annot*?
functor_app_annot ::= **[inline at level** *natural* **]**
| **[no inline]**
module_type ::= *qualid*
| (*module_type*)
| *module_type module_expr_atom*
| *module_type* **with** *with_declaration*
with_declaration ::= **Definition** *qualid* *univ_decl*? **:=** *term*
| **Module** *qualid* **:=** *qualid*
module_expr_atom ::= *qualid*
| (*module_expr_atom*+)
of_module_type ::= **:** *module_type_inl*
| **<:** *module_type_inl**
module_expr_inl ::= **!** *module_expr_atom*+
| *module_expr_atom*+ *functor_app_annot*?

Defines a module named *ident*. See the examples *here*.

The **Import** and **Export** flags specify whether the module should be automatically imported or exported.

Specifying *module_binder** starts a functor with parameters given by the *module_binder*s. (A *functor* is a function from modules to modules.)

of_module_type specifies the module type. **<:** *module_type_inl*+ starts a module that satisfies each *module_type_inl*.

:= *module_expr_inl*+ specifies the body of a module or functor definition. If it's not specified, then the module is defined *interactively*, meaning that the module is defined as a series of commands terminated with *End* instead of in a single *Module* command. Interactively defining the *module_expr_inl*s in a series of *Include* commands is equivalent to giving them all in a single non-interactive *Module* command.

The ! prefix indicates that any assumption command (such as *Axiom*) with an **Inline** clause in the type of the functor arguments will be ignored.

Command:

Module Type *ident* *module_binder** **<:** *module_type_inl** (**:=** *module_type_inl*+)?

Defines a module type named *ident*. See the example *here*.

Specifying *module_binder* * starts a functor type with parameters given by the *module_binder*s.

:= *module_type_inl* + <+ specifies the body of a module or functor type definition. If it's not specified, then the module type is defined *interactively*, meaning that the module type is defined as a series of commands terminated with `End` instead of in a single `Module Type` command. Interactively defining the *module_type_inl*s in a series of `Include` commands is equivalent to giving them all in a single non-interactive `Module Type` command.

Terminating an interactive module or module type definition

Interactive modules are terminated with the `End` command, which is also used to terminate *Sections*. **End** *ident* closes the interactive module or module type `ident`. If the module type was given, the command verifies that the content of the module matches the module type. If the module is not a functor, its components (*constants*, inductive types, submodules etc.) are now available through the dot notation.

Error: No such label *ident*.

Error: Signature components for label *ident* do not match.

Error: The field *ident* is missing in *qualid*.

Note:

1. Interactive modules and module types can be nested.
2. Interactive modules and module types can't be defined inside of *sections*. Sections can be defined inside of interactive modules and module types.
3. Hints and notations (the *Hint* and `Notation` commands) can also appear inside interactive modules and module types. Note that with module definitions like:

   Module $ident_1$ **:** *module_type* **:=** $ident_2$**.**

   or

   Module $ident_1$ **:** *module_type***. Include** $ident_2$**. End** $ident_1$**.**

   hints and the like valid for $ident_1$ are the ones defined in *module_type* rather then those defined in $ident_2$ (or the module body).
4. Within an interactive module type definition, the `Parameter` command declares a *constant* instead of defininig a new axiom (which it does when not in a module type definition).
5. Assumptions such as `Axiom` that include the **Inline** clause will be automatically expanded when the functor is applied, except when the function application is prefixed by `!`.

Command: Include *module_type_inl* <+ *module_expr_inl* *

Includes the content of module(s) in the current interactive module. Here *module_type_inl* can be a module expression or a module type expression. If it is a high-order module or module type expression then the system tries to instantiate *module_type_inl* with the current interactive module.

Including multiple modules is a single `Include` is equivalent to including each module in a separate `Include` command.

Command: Include Type *module_type_inl* + <+

Deprecated since version 8.3: Use `Include` instead.

Command:

Declare Module [**Import** | **Export**]? *ident* [*module_binder*]* : *module_type_inl*

Declares a module *ident* of type *module_type_inl*.

If *module_binder*s are specified, declares a functor with parameters given by the list of *module_binder*s.

Command: Import [*filtered_import*]+

filtered_import ::= *qualid* ([*qualid* (..)?]+,)?

If *qualid* denotes a valid basic module (i.e. its module type is a signature), makes its components available by their short names.

Example

```
Module Mod.
Definition T:=nat.
Check T.
End Mod.
Check Mod.T.

Fail Check T.
    The command has indeed failed with message:
    The reference T was not found in the current environment.

Import Mod.
Check T.
    T
         : Set
```

Some features defined in modules are activated only when a module is imported. This is for instance the case of notations (see *Notations*).

Declarations made with the *local* attribute are never imported by the *Import* command. Such declarations are only accessible through their fully qualified name.

Example

```
Module A.
Module B.
Local Definition T := nat.
End B.
End A.
Import A.

Check B.T.
    Toplevel input, characters 6-9:
    > Check B.T.
    >       ^^^
    Error: The reference B.T was not found in the current environment.
```

Appending a module name with a parenthesized list of names will make only those names available with short names, not other names defined in the module nor will it activate other features.

The names to import may be *constants*, inductive types and constructors, and notation aliases (for instance, Ltac definitions cannot be selectively imported). If they are from an inner module to the one being imported, they must be prefixed by the inner path.

The name of an inductive type may also be followed by `(..)` to import it, its constructors and its eliminators if they exist. For this purpose "eliminator" means a *constant* in the same module whose name is the inductive type's name suffixed by one of `_sind`, `_ind`, `_rec` or `_rect`.

Example

```
Module A.
Module B.
Inductive T := C.
Definition U := nat.
End B.
Definition Z := Prop.
End A.
Import A(B.T(..), Z).

Check B.T.
    B.T
         : Prop

Check B.C.
    B.C
         : B.T

Check Z.
    Z
         : Type

Fail Check B.U.
    The command has indeed failed with message:
    The reference B.U was not found in the current environment.

Check A.B.U.
    A.B.U
         : Set
```

Command: `Export` *`filtered_import`* +

Similar to *`Import`*, except that when the module containing this command is imported, the ***`qualid`*** + are imported as well.

The selective import syntax also works with Export.

Error: *`qualid`* `is not a module.`

Warning: `Trying to mask the absolute name` *`qualid`*`!`

Command: `Print Module` *`qualid`*
Prints the module type and (optionally) the body of the module ***`qualid`***.

Command: `Print Module Type` *`qualid`*
Prints the module type corresponding to ***`qualid`***.

Flag: `Short Module Printing`
This *flag* (off by default) disables the printing of the types of fields, leaving only their names, for the commands `Print Module` and `Print Module Type`.

Examples

Example: Defining a simple module interactively

```
Module M.
Definition T := nat.
Definition x := 0.

Definition y : bool.
    1 goal

      ============================
      bool

exact true.
    No more goals.

Defined.
End M.
```

Inside a module one can define *constants*, prove theorems and do anything else that can be done in the toplevel. Components of a closed module can be accessed using the dot notation:

```
Print M.x.
    M.x = 0
          : nat
```

Example: Defining a simple module type interactively

```
Module Type SIG.
Parameter T : Set.
Parameter x : T.
End SIG.
```

Example: Creating a new module that omits some items from an existing module

Since **`SIG`**, the type of the new module **`N`**, doesn't define **`y`** or give the body of **`x`**, which are not included in **`N`**.

```
Module N : SIG with Definition T := nat := M.
    Module N is defined

Print N.T.
    N.T = nat
          : Set

Print N.x.
    *** [ N.x : N.T ]
```

(continues on next page)

(continued from previous page)

```
Fail Print N.y.
    The command has indeed failed with message:
    N.y not a defined object.
```

The definition of `N` using the module type expression `SIG` with `Definition T := nat` is equivalent to the following one:

```
Module Type SIG'.
Definition T : Set := nat.
Parameter x : T.
End SIG'.

Module N : SIG' := M.
```

If we just want to be sure that our implementation satisfies a given module type without restricting the interface, we can use a transparent constraint

```
Module P <: SIG := M.


Print P.y.
    P.y = true
         : bool
```

Example: Creating a functor (a module with parameters)

```
Module Two (X Y: SIG).
Definition T := (X.T * Y.T)%type.
Definition x := (X.x, Y.x).
End Two.
```

and apply it to our modules and do some computations:

```
Module Q := Two M N.

Eval compute in (fst Q.x + snd Q.x).
    = N.x
         : nat
```

Example: A module type with two sub-modules, sharing some fields

```
Module Type SIG2.
  Declare Module M1 : SIG.
  Module M2 <: SIG.
    Definition T := M1.T.
    Parameter x : T.
  End M2.
End SIG2.

Module Mod <: SIG2.
  Module M1.
```

(continues on next page)

(continued from previous page)

```
    Definition T := nat.
    Definition x := 1.
  End M1.
Module M2 := M.
End Mod.
```

Notice that `M` is a correct body for the component `M2` since its `T` component is `nat` as specified for `M1.T`.

Typing Modules

In order to introduce the typing system we first slightly extend the syntactic class of terms and environments given in section *The terms*. The environments, apart from definitions of *constants* and inductive types now also hold any other structure elements. Terms, apart from variables, *constants* and complex terms, also include access paths.

We also need additional typing judgments:

- $E[] \vdash \mathcal{WF}(S)$, denoting that a structure S is well-formed,
- $E[] \vdash p : S$, denoting that the module pointed by p has type S in the global environment E.
- $E[] \vdash S \longrightarrow \overline{S}$, denoting that a structure S is evaluated to a structure $\overline{S}$ in weak head normal form.
- $E[] \vdash S_1 <: S_2$, denoting that a structure S_1 is a subtype of a structure S_2.
- $E[] \vdash e_1 <: e_2$, denoting that a structure element e_1 is more precise than a structure element e_2.

The rules for forming structures are the following:

WF-STR

$$\frac{\mathcal{WF}(E; E')[]}{E[] \vdash \mathcal{WF}(\mathsf{Struct}\ E'\ \mathsf{End})}$$

WF-FUN

$$\frac{E; \mathsf{Mod}(X : S)[] \vdash \mathcal{WF}(\overline{S'})}{E[] \vdash \mathcal{WF}(\mathsf{Functor}(X : S)\ S')}$$

Evaluation of structures to weak head normal form:

WEVAL-APP

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Functor}(X : S_1)\ S_2 \quad E[] \vdash S_1 \longrightarrow \overline{S_1} \\ E[] \vdash p : S_3 \quad E[] \vdash S_3 <: \overline{S_1} \end{array}}{E[] \vdash S\ p \longrightarrow S_2\{p/X, t_1/p_1.c_1, ..., t_n/p_n.c_n\}}$$

In the last rule, $\{t_1/p_1.c_1, ..., t_n/p_n.c_n\}$ is the resulting substitution from the inlining mechanism. We substitute in S the inlined fields $p_i.c_i$ from $\mathsf{Mod}(X : S_1)$ by the corresponding delta-reduced term t_i in p.

WEVAL-WITH-MOD

$$\frac{\begin{array}{c} E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X : S_1); e_{i+2}; ...; e_n\ \mathsf{End} \\ E; e_1; ...; e_i[] \vdash S_1 \longrightarrow \overline{S_1} \qquad E[] \vdash p : S_2 \\ E; e_1; ...; e_i[] \vdash S_2 <: \overline{S_1} \end{array}}{\begin{array}{c} E[] \vdash S\ \mathsf{with}\ X := p \longrightarrow \\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{ModA}(X == p); e_{i+2}\{p/X\}; ...; e_n\{p/X\}\ \mathsf{End} \end{array}}$$

WEVAL-WITH-MOD-REC

$$\frac{E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X_1 : S_1); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash S_1\ \mathsf{with}\ p := p_1 \longrightarrow \overline{S_2}}{E[] \vdash S\ \mathsf{with}\ X_1.p := p_1 \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X : \overline{S_2}); e_{i+2}\{p_1/X_1.p\}; ...; e_n\{p_1/X_1.p\}\ \mathsf{End}}$$

WEVAL-WITH-DEF

$$\frac{E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Assum}()(c : T_1); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash \mathsf{Def}()(c := t : T)) <: \mathsf{Assum}()(c : T_1)}{E[] \vdash S\ \mathsf{with}\ c := t : T \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Def}()(c := t : T); e_{i+2}; ...; e_n\ \mathsf{End}}$$

WEVAL-WITH-DEF-REC

$$\frac{E[] \vdash S \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X_1 : S_1); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash S_1\ \mathsf{with}\ p := p_1 \longrightarrow \overline{S_2}}{E[] \vdash S\ \mathsf{with}\ X_1.p := t : T \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X : \overline{S_2}); e_{i+2}; ...; e_n\ \mathsf{End}}$$

WEVAL-PATH-MOD1

$$\frac{E[] \vdash p \longrightarrow \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Mod}(X : S\,[:= S_1]); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash S \longrightarrow \overline{S}}{E[] \vdash p.X \longrightarrow \overline{S}}$$

WEVAL-PATH-MOD2

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{Mod}(X : S\,[:= S_1]) \in E \qquad E[] \vdash S \longrightarrow \overline{S}}{E[] \vdash X \longrightarrow \overline{S}}$$

WEVAL-PATH-ALIAS1

$$\frac{E[] \vdash p \longrightarrow\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{ModA}(X == p_1); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash p_1 \longrightarrow \overline{S}}{E[] \vdash p.X \longrightarrow \overline{S}}$$

WEVAL-PATH-ALIAS2

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{ModA}(X == p_1) \in E \qquad E[] \vdash p_1 \longrightarrow \overline{S}}{E[] \vdash X \longrightarrow \overline{S}}$$

WEVAL-PATH-TYPE1

$$\frac{E[] \vdash p \longrightarrow\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{ModType}(Y := S); e_{i+2}; ...; e_n\ \mathsf{End} \qquad E; e_1; ...; e_i[] \vdash S \longrightarrow \overline{S}}{E[] \vdash p.Y \longrightarrow \overline{S}}$$

WEVAL-PATH-TYPE2

$$\frac{\mathcal{WF}(E)[] \qquad \mathsf{ModType}(Y := S) \in E \qquad E[] \vdash S \longrightarrow \overline{S}}{E[] \vdash Y \longrightarrow \overline{S}}$$

Rules for typing module:

MT-EVAL

$$\frac{E[] \vdash p \longrightarrow \overline{S}}{E[] \vdash p : \overline{S}}$$

MT-STR

$$\frac{E[] \vdash p : S}{E[] \vdash p : S/p}$$

The last rule, called strengthening is used to make all module fields manifestly equal to themselves. The notation S/p has the following meaning:

- if $S \longrightarrow$ Struct $e_1; ...; e_n$ End then $S/p =$ Struct $e_1/p; ...; e_n/p$ End where e/p is defined as follows (note that opaque definitions are processed as assumptions):
  - Def$()(c := t : T)/p =$ Def$()(c := t : T)$
  - Assum$()(c : U)/p =$ Def$()(c := p.c : U)$
  - Mod$(X : S)/p =$ ModA$(X == p.X)$
  - ModA$(X == p')/p =$ ModA$(X == p')$
  - Ind$[\Gamma_P](\Gamma_C := \Gamma_I)/p =$ Ind$_p()[\Gamma_P](\Gamma_C := \Gamma_I)$
  - Ind$_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)/p =$ Ind$_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I)$
- if $S \longrightarrow$ Functor$(X : S')\ S''$ then $S/p = S$

The notation Ind$_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ denotes an inductive definition that is definitionally equal to the inductive definition in the module denoted by the path p. All rules which have Ind$[\Gamma_P](\Gamma_C := \Gamma_I)$ as premises are also valid for Ind$_p()[\Gamma_P](\Gamma_C := \Gamma_I)$. We give the formation rule for Ind$_p()[\Gamma_P](\Gamma_C := \Gamma_I)$ below as well as the equality rules on inductive types and constructors.

The module subtyping rules:

MSUB-STR

$$\frac{\begin{array}{c} E; e_1; ...; e_n[] \vdash e_{\sigma(i)} <: e'_i \text{ for } i = 1..m \\ \sigma : \{1...m\} \to \{1...n\} \text{ injective} \end{array}}{E[] \vdash \text{Struct } e_1; ...; e_n \text{ End} <: \text{ Struct } e'_1; ...; e'_m \text{ End}}$$

MSUB-FUN

$$\frac{E[] \vdash \overline{S'_1} <: \overline{S_1} \qquad E; \text{Mod}(X : S'_1)[] \vdash \overline{S_2} <: \overline{S'_2}}{E[] \vdash \text{Functor}(X : S_1)S_2 <: \text{Functor}(X : S'_1)S'_2}$$

Structure element subtyping rules:

ASSUM-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash \text{Assum}()(c : T_1) <: \text{Assum}()(c : T_2)}$$

DEF-ASSUM

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash \text{Def}()(c := t : T_1) <: \text{Assum}()(c : T_2)}$$

ASSUM-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash c =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash \mathsf{Assum}()(c : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

DEF-DEF

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash \mathsf{Def}()(c := t_1 : T_1) <: \mathsf{Def}()(c := t_2 : T_2)}$$

IND-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta\eta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C \qquad E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I}{E[] \vdash \mathsf{Ind}\,[\Gamma_P]\,(\Gamma_C \;:=\; \Gamma_I) <: \mathsf{Ind}\,[\Gamma'_P]\,(\Gamma'_C \;:=\; \Gamma'_I)}$$

INDP-IND

$$\frac{E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta\eta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C \qquad E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I}{E[] \vdash \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) <: \mathsf{Ind}\,[\Gamma'_P]\,(\Gamma'_C \;:=\; \Gamma'_I)}$$

INDP-INDP

$$\frac{\begin{array}{c} E[] \vdash \Gamma_P =_{\beta\delta\iota\zeta\eta} \Gamma'_P \qquad E[\Gamma_P] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C \\ E[\Gamma_P; \Gamma_C] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I \qquad E[] \vdash p =_{\beta\delta\iota\zeta\eta} p' \end{array}}{E[] \vdash \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I) <: \mathsf{Ind}_{p'}()[\Gamma'_P](\Gamma'_C := \Gamma'_I)}$$

MOD-MOD

$$\frac{E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{Mod}(X : S_2)}$$

ALIAS-MOD

$$\frac{E[] \vdash p : S_1 \qquad E[] \vdash S_1 <: S_2}{E[] \vdash \mathsf{ModA}(X == p) <: \mathsf{Mod}(X : S_2)}$$

MOD-ALIAS

$$\frac{E[] \vdash p : S_2 \qquad E[] \vdash S_1 <: S_2 \qquad E[] \vdash X =_{\beta\delta\iota\zeta\eta} p}{E[] \vdash \mathsf{Mod}(X : S_1) <: \mathsf{ModA}(X == p)}$$

ALIAS-ALIAS

$$\frac{E[] \vdash p_1 =_{\beta\delta\iota\zeta\eta} p_2}{E[] \vdash \mathsf{ModA}(X == p_1) <: \mathsf{ModA}(X == p_2)}$$

MODTYPE-MODTYPE

$$\frac{E[] \vdash S_1 <: S_2 \qquad E[] \vdash S_2 <: S_1}{E[] \vdash \mathsf{ModType}(Y := S_1) <: \mathsf{ModType}(Y := S_2)}$$

New environment formation rules

WF-MOD1

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{\mathcal{WF}(E; \mathsf{Mod}(X : S))[]}$$

WF-MOD2

$$\frac{E[] \vdash S_2 <: S_1 \qquad \mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S_1) \qquad E[] \vdash \mathcal{WF}(S_2)}{\mathcal{WF}(E; \mathsf{Mod}(X : S_1 := S_2))[]}$$

WF-ALIAS

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash p : S}{\mathcal{WF}(E; \mathsf{ModA}(X == p))[]}$$

WF-MODTYPE

$$\frac{\mathcal{WF}(E)[] \qquad E[] \vdash \mathcal{WF}(S)}{\mathcal{WF}(E; \mathsf{ModType}(Y := S))[]}$$

WF-IND

$$\frac{\begin{array}{c}\mathcal{WF}(E; \mathsf{Ind}\,[\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I))[] \\ E[] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_n; \mathsf{Ind}\,[\Gamma'_P]\,(\Gamma'_C\ :=\ \Gamma'_I)\,; ...\ \mathsf{End} \\ E[] \vdash \mathsf{Ind}\,[\Gamma'_P]\,(\Gamma'_C\ :=\ \Gamma'_I) <: \mathsf{Ind}\,[\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I)\end{array}}{\mathcal{WF}(E; \mathsf{Ind}_p()[\Gamma_P](\Gamma_C := \Gamma_I))[]}$$

Component access rules

ACC-TYPE1

$$\frac{E[\Gamma] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Assum}()(c : T); ...\ \mathsf{End}}{E[\Gamma] \vdash p.c : T}$$

ACC-TYPE2

$$\frac{E[\Gamma] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Def}()(c := t : T); ...\ \mathsf{End}}{E[\Gamma] \vdash p.c : T}$$

Notice that the following rule extends the delta rule defined in section *Conversion rules*

ACC-DELTA

$$\frac{E[\Gamma] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Def}()(c := t : U); ...\ \mathsf{End}}{E[\Gamma] \vdash p.c \rhd_\delta t}$$

In the rules below we assume Γ_P is $[p_1 : P_1; ...; p_r : P_r]$, Γ_I is $[I_1 : A_1; ...; I_k : A_k]$, and Γ_C is $[c_1 : C_1; ...; c_n : C_n]$.

ACC-IND1

$$\frac{E[\Gamma] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Ind}\,[\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I)\,; ...\ \mathsf{End}}{E[\Gamma] \vdash p.I_j : (p_1 : P_1)...(p_r : P_r)A_j}$$

ACC-IND2

$$\frac{E[\Gamma] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Ind}\,[\Gamma_P]\,(\Gamma_C\ :=\ \Gamma_I)\,; ...\ \mathsf{End}}{E[\Gamma] \vdash p.c_m : (p_1 : P_1)...(p_r : P_r)C_m I_j(I_j\, p_1 ... p_r)_{j=1...k}}$$

ACC-INDP1

$$\frac{E[] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I); ...\ \mathsf{End}}{E[] \vdash p.I_i \rhd_\delta p'.I_i}$$

ACC-INDP2

$$\frac{E[] \vdash p :\ \mathsf{Struct}\ e_1; ...; e_i; \mathsf{Ind}_{p'}()[\Gamma_P](\Gamma_C := \Gamma_I); ...\ \mathsf{End}}{E[] \vdash p.c_i \rhd_\delta p'.c_i}$$

Libraries and qualified names

Names of libraries

The theories developed in Coq are stored in *library files* which are hierarchically classified into *libraries* and *sublibraries*. To express this hierarchy, library names are represented by qualified identifiers qualid, i.e. as list of identifiers separated by dots (see *Qualified identifiers*). For instance, the library file `Mult` of the standard Coq library `Arith` is named `Coq.Arith.Mult`. The identifier that starts the name of a library is called a *library root*. All library files of the standard library of Coq have the reserved root Coq but library filenames based on other roots can be obtained by using Coq commands (coqc, coqtop, coqdep, …) options `-Q` or `-R` (see *By command line options*). Also, when an interactive Coq session starts, a library of root `Top` is started, unless option `-top` or `-notop` is set (see *By command line options*).

Qualified identifiers

qualid ::= *ident* *field_ident**
field_ident ::= *.ident*

Library files are modules which possibly contain submodules which eventually contain constructions (axioms, parameters, definitions, lemmas, theorems, remarks or facts). The *absolute name*, or *full name*, of a construction in some library file is a qualified identifier starting with the logical name of the library file, followed by the sequence of submodules names encapsulating the construction and ended by the proper name of the construction. Typically, the absolute name `Coq.Init.Logic.eq` denotes Leibniz' equality defined in the module Logic in the sublibrary `Init` of the standard library of Coq.

The proper name that ends the name of a construction is the short name (or sometimes base name) of the construction (for instance, the short name of `Coq.Init.Logic.eq` is `eq`). Any partial suffix of the absolute name is a *partially qualified name* (e.g. `Logic.eq` is a partially qualified name for `Coq.Init.Logic.eq`). Especially, the short name of a construction is its shortest partially qualified name.

Coq does not accept two constructions (definition, theorem, …) with the same absolute name but different constructions can have the same short name (or even same partially qualified names as soon as the full names are different).

Notice that the notion of absolute, partially qualified and short names also applies to library filenames.

Visibility

Coq maintains a table called the name table which maps partially qualified names of constructions to absolute names. This table is updated by the commands *Require*, *Import* and *Export* and also each time a new declaration is added to the context. An absolute name is called visible from a given short or partially qualified name when this latter name is enough to denote it. This means that the short or partially qualified name is mapped to the absolute name in Coq name table. Definitions with the *local* attribute are only accessible with their fully qualified name (see *Top-level definitions*).

It may happen that a visible name is hidden by the short name or a qualified name of another construction. In this case, the name that has been hidden must be referred to using one more level of qualification. To ensure that a construction always remains accessible, absolute names can never be hidden.

Example

```
Check 0.
    0
         : nat


Definition nat := bool.
```

(continues on next page)

(continued from previous page)

```
    nat is defined


Check 0.
    0
         : Datatypes.nat


Check Datatypes.nat.
    Datatypes.nat
         : Set


Locate nat.
    Constant Top.nat
    Inductive Coq.Init.Datatypes.nat
      (shorter name to refer to it in current context is Datatypes.nat)
```

See also:

Commands *Locate*.

Libraries and filesystem

Note: The questions described here have been subject to redesign in Coq 8.5. Former versions of Coq use the same terminology to describe slightly different things.

Compiled files (`.vo` and `.vio`) store sub-libraries. In order to refer to them inside Coq, a translation from file-system names to Coq names is needed. In this translation, names in the file system are called *physical* paths while Coq names are contrastingly called *logical* names.

A logical prefix Lib can be associated with a physical path using the command line option `-Q path Lib`. All subfolders of path are recursively associated with the logical path `Lib` extended with the corresponding suffix coming from the physical path. For instance, the folder `path/foo/Bar` maps to `Lib.foo.Bar`. Subdirectories corresponding to invalid Coq identifiers are skipped, and, by convention, subdirectories named `CVS` or `_darcs` are skipped too.

Thanks to this mechanism, `.vo` files are made available through the logical name of the folder they are in, extended with their own basename. For example, the name associated with the file `path/foo/Bar/File.vo` is `Lib.foo.Bar.File`. The same caveat applies for invalid identifiers. When compiling a source file, the `.vo` file stores its logical name, so that an error is issued if it is loaded with the wrong loadpath afterwards.

Some folders have a special status and are automatically put in the path. Coq commands automatically associate a logical path to files in the repository tree rooted at the directory from where the command is launched, `coqlib/user-contrib/`, the directories listed in the `$COQPATH`, `${XDG_DATA_HOME}/coq/` and `${XDG_DATA_DIRS}/coq/` environment variables (see XDG base directory specification[13]) with the same physical-to-logical translation and with an empty logical prefix.

The command line option `-R` is a variant of `-Q` which has the strictly same behavior regarding loadpaths, but which also makes the corresponding `.vo` files available through their short names in a way similar to the *Import* command. For instance, `-R path Lib` associates the file `/path/foo/Bar/File.vo` with the logical name `Lib.foo.Bar.File` but allows this file to be accessed through the short names `foo.Bar.File`, `Bar.File` and `File`. If several files with identical base name are present in different subdirectories of a recursive loadpath, which of these files is found first

[13] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

may be system- dependent and explicit qualification is recommended. The `From` argument of the `Require` command can be used to bypass the implicit shortening by providing an absolute root to the required file (see *Compiled files*).

There also exists another independent loadpath mechanism attached to OCaml object files (`.cmo` or `.cmxs`) rather than Coq object files as described above. The OCaml loadpath is managed using the option `-I path` (in the OCaml world, there is neither a notion of logical name prefix nor a way to access files in subdirectories of path). See the command *`Declare ML Module`* in *Compiled files* to understand the need of the OCaml loadpath.

See *By command line options* for a more general view over the Coq command line options.

Controlling the scope of commands with locality attributes

Many commands have effects that apply only within a specific scope, typically the section or the module in which the command was called. Locality *attributes* can alter the scope of the effect. Below, we give the semantics of each locality attribute while noting a few exceptional commands for which *`local`* and *`global`* attributes are interpreted differently.

Attribute: `local`

This *attribute* limits the effect of the command to the current scope (section or module).

The `Local` prefix is an alternative syntax for the *`local`* attribute (see ***`legacy_attr`***).

Note:

- For some commands, this is the only locality supported within sections (e.g., for *`Notation`*, *`Ltac`* and *Hint* commands).
- For some commands, this is the default locality within sections even though other locality attributes are supported as well (e.g., for the *`Arguments`* command).

> **Warning: Exception:** when *`local`* is applied to *`Definition`*, *`Theorem`* or their variants, its semantics are different: it makes the defined objects available only through their fully-qualified names rather than their unqualified names after an *`Import`*.

Attribute: `export`

This *attribute* makes the effect of the command persist when the section is closed and applies the effect when the module containing the command is imported.

Commands supporting this attribute include *`Set`*, *`Unset`* and the *Hint* commands, although the latter don't support it within sections.

Attribute: `global`

This *attribute* makes the effect of the command persist even when the current section or module is closed. Loading the file containing the command (possibly transitively) applies the effect of the command.

The `Global` prefix is an alternative syntax for the *`global`* attribute (see ***`legacy_attr`***).

> **Warning: Exception:** for a few commands (like *`Notation`* and *`Ltac`*), this attribute behaves like *`export`*.

> **Warning:** We strongly discourage using the *`global`* locality attribute because the transitive nature of file loading gives the user little control. We recommend using the *`export`* locality attribute where it is supported.

2.1.13 Primitive objects

Primitive Integers

The language of terms features 63-bit machine integers as values. The type of such a value is *axiomatized*; it is declared through the following sentence (excerpt from the `PrimInt63` module):

```
Primitive int := #int63_type.
```

This type can be understood as representing either unsigned or signed integers, depending on which module is imported or, more generally, which scope is open. `Uint63` and `uint63_scope` refer to the unsigned version, while `Sint63` and `sint63_scope` refer to the signed one.

The `PrimInt63` module declares the available operators for this type. For instance, equality of two unsigned primitive integers can be determined using the `Uint63.eqb` function, declared and specified as follows:

```
Primitive eqb := #int63_eq.
Notation "m '==' n" := (eqb m n) (at level 70, no associativity) : uint63_scope.

Axiom eqb_correct : forall i j, (i == j)%uint63 = true -> i = j.
```

The complete set of such operators can be found in the `PrimInt63` module. The specifications and notations are in the `Uint63` and `Sint63` modules.

These primitive declarations are regular axioms. As such, they must be trusted and are listed by the `Print Assumptions` command, as in the following example.

```
From Coq Require Import Uint63.
Lemma one_minus_one_is_zero : (1 - 1 = 0)%uint63.
Proof. apply eqb_correct; vm_compute; reflexivity. Qed.

Print Assumptions one_minus_one_is_zero.
    Axioms:
    sub : int -> int -> int
    eqb_correct : forall i j : int, (i =? j)%uint63 = true -> i = j
    eqb : int -> int -> bool
```

The reduction machines implement dedicated, efficient rules to reduce the applications of these primitive operations.

The extraction of these primitives can be customized similarly to the extraction of regular axioms (see *Program extraction*). Nonetheless, the `ExtrOCamlInt63` module can be used when extracting to OCaml: it maps the Coq primitives to types and functions of a `Uint63` module (including signed functions for `Sint63` despite the name). That OCaml module is not produced by extraction. Instead, it has to be provided by the user (if they want to compile or execute the extracted code). For instance, an implementation of this module can be taken from the kernel of Coq.

Literal values (at type `Uint63.int`) are extracted to literal OCaml values wrapped into the `Uint63.of_int` (resp. `Uint63.of_int64`) constructor on 64-bit (resp. 32-bit) platforms. Currently, this cannot be customized (see the function `Uint63.compile` from the kernel).

Primitive Floats

The language of terms features Binary64 floating-point numbers as values. The type of such a value is *axiomatized*; it is declared through the following sentence (excerpt from the `PrimFloat` module):

```
Primitive float := #float64_type.
```

This type is equipped with a few operators, that must be similarly declared. For instance, the product of two primitive floats can be computed using the `PrimFloat.mul` function, declared and specified as follows:

```
Primitive mul := #float64_mul.
Notation "x * y" := (mul x y) : float_scope.

Axiom mul_spec : forall x y, Prim2SF (x * y)%float = SF64mul (Prim2SF x) (Prim2SF y).
```

where `Prim2SF` is defined in the `FloatOps` module.

The set of such operators is described in section *Floats library*.

These primitive declarations are regular axioms. As such, they must be trusted, and are listed by the `Print Assumptions` command.

The reduction machines (*`vm_compute`*, *`native_compute`*) implement dedicated, efficient rules to reduce the applications of these primitive operations, using the floating-point processor operators that are assumed to comply with the IEEE 754 standard for floating-point arithmetic.

The extraction of these primitives can be customized similarly to the extraction of regular axioms (see *Program extraction*). Nonetheless, the `ExtrOCamlFloats` module can be used when extracting to OCaml: it maps the Coq primitives to types and functions of a `Float64` module. Said OCaml module is not produced by extraction. Instead, it has to be provided by the user (if they want to compile or execute the extracted code). For instance, an implementation of this module can be taken from the kernel of Coq.

Literal values (of type `Float64.t`) are extracted to literal OCaml values (of type `float`) written in hexadecimal notation and wrapped into the `Float64.of_float` constructor, e.g.: `Float64.of_float (0x1p+0)`.

Primitive Arrays

The language of terms features persistent arrays as values. The type of such a value is *axiomatized*; it is declared through the following sentence (excerpt from the `PArray` module):

```
Primitive array := #array_type.
```

This type is equipped with a few operators, that must be similarly declared. For instance, elements in an array can be accessed and updated using the `PArray.get` and `PArray.set` functions, declared and specified as follows:

```
Primitive get := #array_get.
Primitive set := #array_set.
Notation "t .[ i ]" := (get t i).
Notation "t .[ i <- a ]" := (set t i a).

Axiom get_set_same : forall A t i (a:A), (i < length t) = true -> t.[i<-a].[i] = a.
Axiom get_set_other : forall A t i j (a:A), i <> j -> t.[i<-a].[j] = t.[j].
```

The rest of these operators can be found in the `PArray` module.

These primitive declarations are regular axioms. As such, they must be trusted and are listed by the `Print Assumptions` command.

The reduction machines (*vm_compute*, *native_compute*) implement dedicated, efficient rules to reduce the applications of these primitive operations.

The extraction of these primitives can be customized similarly to the extraction of regular axioms (see *Program extraction*). Nonetheless, the `ExtrOCamlPArray` module can be used when extracting to OCaml: it maps the Coq primitives to types and functions of a `Parray` module. Said OCaml module is not produced by extraction. Instead, it has to be provided by the user (if they want to compile or execute the extracted code). For instance, an implementation of this module can be taken from the kernel of Coq (see `kernel/parray.ml`).

Coq's primitive arrays are persistent data structures. Semantically, a set operation `t.[i <- a]` represents a new array that has the same values as `t`, except at position `i` where its value is `a`. The array `t` still exists, can still be used and its values were not modified. Operationally, the implementation of Coq's primitive arrays is optimized so that the new array `t.[i <- a]` does not copy all of `t`. The details are in section 2.3 of [CF07]. In short, the implementation keeps one version of `t` as an OCaml native array and other versions as lists of modifications to `t`. Accesses to the native array version are constant time operations. However, accesses to versions where all the cells of the array are modified have O(n) access time, the same as a list. The version that is kept as the native array changes dynamically upon each get and set call: the current list of modifications is applied to the native array and the lists of modifications of the other versions are updated so that they still represent the same values.

2.1.14 Polymorphic Universes

Author Matthieu Sozeau

General Presentation

Warning: The status of Universe Polymorphism is experimental.

This section describes the universe polymorphic extension of Coq. Universe polymorphism makes it possible to write generic definitions making use of universes and reuse them at different and sometimes incompatible universe levels.

A standard example of the difference between universe *polymorphic* and *monomorphic* definitions is given by the identity function:

```
Definition identity {A : Type} (a : A) := a.
```

By default, *constant* declarations are monomorphic, hence the identity function declares a global universe (say `Top.1`) for its domain. Subsequently, if we try to self-apply the identity, we will get an error:

```
Fail Definition selfid := identity (@identity).
    The command has indeed failed with message:
    The term "@identity" has type "forall A : Type, A -> A"
    while it is expected to have type "?A"
    (unable to find a well-typed instantiation for "?A": cannot ensure that
    "Type@{identity.u0+1}" is a subtype of "Type@{identity.u0}").
```

Indeed, the global level `Top.1` would have to be strictly smaller than itself for this self-application to type check, as the type of `(@identity)` is `forall (A : Type@{Top.1}), A -> A` whose type is itself `Type@{Top.1+1}`.

A universe polymorphic identity function binds its domain universe level at the definition level instead of making it global.

```
Polymorphic Definition pidentity {A : Type} (a : A) := a.
```

```
About pidentity.
    pidentity@{u} : forall {A : Type}, A -> A

    pidentity is universe polymorphic
    Arguments pidentity {A}%type_scope _
    pidentity is transparent
    Expands to: Constant Top.pidentity
```

It is then possible to reuse the constant at different levels, like so:

```
Definition selfpid := pidentity (@pidentity).
```

Of course, the two instances of `pidentity` in this definition are different. This can be seen when the *Printing Universes* flag is on:

```
Print selfpid.
    selfpid =
    pidentity@{selfpid.u0} (@pidentity@{selfpid.u1})
         : forall A : Type@{selfpid.u1}, A -> A
    (* {selfpid.u1 selfpid.u0} |= selfpid.u1 < selfpid.u0 *)

    Arguments selfpid _%type_scope _
```

Now `pidentity` is used at two different levels: at the head of the application it is instantiated at `Top.3` while in the argument position it is instantiated at `Top.4`. This definition is only valid as long as `Top.4` is strictly smaller than `Top.3`, as shown by the constraints. Note that this definition is monomorphic (not universe polymorphic), so the two universes (in this case `Top.3` and `Top.4`) are actually global levels.

When printing `pidentity`, we can see the universes it binds in the annotation `@{Top.2}`. Additionally, when *Printing Universes* is on we print the "universe context" of `pidentity` consisting of the bound universes and the constraints they must verify (for `pidentity` there are no constraints).

Inductive types can also be declared universes polymorphic on universes appearing in their parameters or fields. A typical example is given by monoids:

```
Polymorphic Record Monoid := { mon_car :> Type; mon_unit : mon_car;
  mon_op : mon_car -> mon_car -> mon_car }.

Print Monoid.
```

The Monoid's carrier universe is polymorphic, hence it is possible to instantiate it for example with `Monoid` itself. First we build the trivial unit monoid in `Set`:

```
Definition unit_monoid : Monoid :=
  {| mon_car := unit; mon_unit := tt; mon_op x y := tt |}.
```

From this we can build a definition for the monoid of `Set`-monoids (where multiplication would be given by the product of monoids).

```
Polymorphic Definition monoid_monoid : Monoid.
  refine (@Build_Monoid Monoid unit_monoid (fun x y => x)).
Defined.

Print monoid_monoid.
    monoid_monoid@{u} =
    {|
      mon_car := Monoid@{Set};
```

(continues on next page)

(continued from previous page)

```
  mon_unit := unit_monoid;
  mon_op := fun x _ : Monoid@{Set} => x
|}
     : Monoid@{u}
(* u |= Set < u *)
```

As one can see from the constraints, this monoid is "large", it lives in a universe strictly higher than `Set`.

Polymorphic, Monomorphic

Attribute: `universes(polymorphic = yes | no ?)`

This *boolean attribute* can be used to control whether universe polymorphism is enabled in the definition of an inductive type. There is also a legacy syntax using the `Polymorphic` prefix (see ***legacy_attr***) which, as shown in the examples, is more commonly used.

When `universes(polymorphic=no)` is used, global universe constraints are produced, even when the *Universe Polymorphism* flag is on. There is also a legacy syntax using the `Monomorphic` prefix (see ***legacy_attr***).

Attribute: `universes(monomorphic)`

Deprecated since version 8.13: Use *universes(polymorphic=no)* instead.

Flag: `Universe Polymorphism`

This *flag* is off by default. When it is on, new declarations are polymorphic unless the *universes(polymorphic=no)* attribute is used to override the default.

Many other commands can be used to declare universe polymorphic or monomorphic *constants* depending on whether the *Universe Polymorphism* flag is on or the *universes(polymorphic)* attribute is used:

- *Lemma*, *Axiom*, etc. can be used to declare universe polymorphic constants.
- Using the *universes(polymorphic)* attribute with the *Section* command will locally set the polymorphism flag inside the section.
- *Variable*, *Context*, *Universe* and *Constraint* in a section support polymorphism. See *Universe polymorphism and sections* for more details.
- Using the *universes(polymorphic)* attribute with the *Hint Resolve* or *Hint Rewrite* commands will make *auto* / *rewrite* use the hint polymorphically, not at a single instance.

Cumulative, NonCumulative

Attribute: `universes(cumulative = yes | no ?)`

Polymorphic inductive types, coinductive types, variants and records can be declared cumulative using this *boolean attribute* or the legacy `Cumulative` prefix (see ***legacy_attr***) which, as shown in the examples, is more commonly used.

This means that two instances of the same inductive type (family) are convertible based on the universe variances; they do not need to be equal.

When the attribtue is off, the inductive type is non-cumulative even if the *Polymorphic Inductive Cumulativity* flag is on. There is also a legacy syntax using the `NonCumulative` prefix (see ***legacy_attr***).

This means that two instances of the same inductive type (family) are convertible only if all the universes are equal.

Error: `The cumulative attribute can only be used in a polymorphic context.`
Using this attribute requires being in a polymorphic context, i.e. either having the *`Universe Polymorphism`* flag on, or having used the *`universes(polymorphic)`* attribute as well.

Note: `#[ universes(polymorphic = yes?), universes(cumulative = yes | no?) ]` can be abbreviated into `#[ universes(polymorphic = yes?, cumulative = yes | no?) ]`.

Attribute: `universes(noncumulative)`
Deprecated since version 8.13: Use *`universes(cumulative=no)`* instead.

Flag: `Polymorphic Inductive Cumulativity`
When this *flag* is on (it is off by default), it makes all subsequent *polymorphic* inductive definitions cumulative, unless the *`universes(cumulative=no)`* attribute is used to override the default. It has no effect on *monomorphic* inductive definitions.

Consider the examples below.

```
Polymorphic Cumulative Inductive list {A : Type} :=
| nil : list
| cons : A -> list -> list.

Print list.
    Inductive list@{u} (A : Type@{u}) : Type@{max(Set,u)} :=
        nil : list@{u} | cons : A -> list@{u} -> list@{u}.
    (* *u |=  *)

    Arguments list {A}%type_scope
    Arguments nil {A}%type_scope
    Arguments cons {A}%type_scope _ _
```

When printing `list`, the universe context indicates the subtyping constraints by prefixing the level names with symbols.

Because inductive subtypings are only produced by comparing inductives to themselves with universes changed, they amount to variance information: each universe is either invariant, covariant or irrelevant (there are no contravariant subtypings in Coq), respectively represented by the symbols =, + and *.

Here we see that `list` binds an irrelevant universe, so any two instances of `list` are convertible: $E[\Gamma] \vdash \mathsf{list@\{}i\mathsf{\}}\ A =_{\beta\delta\iota\zeta\eta} \mathsf{list@\{}j\mathsf{\}}\ B$ whenever $E[\Gamma] \vdash A =_{\beta\delta\iota\zeta\eta} B$ and this applies also to their corresponding constructors, when they are comparable at the same type.

See *Conversion rules* for more details on convertibility and subtyping. The following is an example of a record with non-trivial subtyping relation:

```
Polymorphic Cumulative Record packType := {pk : Type}.
    packType is defined
    pk is defined

About packType.
    packType@{u} : Type@{u+1}
    (* +u |=  *)
```

(continues on next page)

(continued from previous page)

```
    packType is universe polymorphic
    Expands to: Inductive Top.packType
```

`packType` binds a covariant universe, i.e.

$$E[\Gamma] \vdash \mathsf{packType@\{}i\mathsf{\}} =_{\beta\delta\iota\zeta\eta} \mathsf{packType@\{}j\mathsf{\}} \text{ whenever } i \leq j$$

Specifying cumulativity

The variance of the universe parameters for a cumulative inductive may be specified by the user.

For the following type, universe `a` has its variance automatically inferred (it is irrelevant), `b` is required to be irrelevant, `c` is covariant and `d` is invariant. With these annotations `c` and `d` have less general variances than would be inferred.

```
Polymorphic Cumulative Inductive Dummy@{a *b +c =d} : Prop := dummy.
    Dummy is defined
    Dummy_rect is defined
    Dummy_ind is defined
    Dummy_rec is defined
    Dummy_sind is defined

About Dummy.
    Dummy@{a b c d} : Prop
    (* *a *b +c =d |=  *)

    Dummy is universe polymorphic
    Expands to: Inductive Top.Dummy
```

Insufficiently restrictive variance annotations lead to errors:

```
Fail Polymorphic Cumulative Record bad@{*a} := {p : Type@{a}}.
    The command has indeed failed with message:
    Incorrect variance for universe Top.45: expected * but cannot be less restrictive␣
 ↪than +.
```

An example of a proof using cumulativity

```
Set Universe Polymorphism.
Set Polymorphic Inductive Cumulativity.

Inductive eq@{i} {A : Type@{i}} (x : A) : A -> Type@{i} := eq_refl : eq x x.

Definition funext_type@{a b e} (A : Type@{a}) (B : A -> Type@{b})
:= forall f g : (forall a, B a),
                (forall x, eq@{e} (f x) (g x))
                -> eq@{e} f g.

Section down.
   Universes a b e e'.
   Constraint e' < e.
   Lemma funext_down {A B}
     (H : @funext_type@{a b e} A B) : @funext_type@{a b e'} A B.
   Proof.
```

(continues on next page)

(continued from previous page)

```
    exact H.
  Defined.
End down.
```

Cumulativity Weak Constraints

Flag: `Cumulativity Weak Constraints`
When set, which is the default, this *flag* causes "weak" constraints to be produced when comparing universes in an irrelevant position. Processing weak constraints is delayed until minimization time. A weak constraint between `u` and `v` when neither is smaller than the other and one is flexible causes them to be unified. Otherwise the constraint is silently discarded.

This heuristic is experimental and may change in future versions. Disabling weak constraints is more predictable but may produce arbitrary numbers of universes.

Global and local universes

Each universe is declared in a global or local context before it can be used. To ensure compatibility, every *global* universe is set to be strictly greater than `Set` when it is introduced, while every *local* (i.e. polymorphically quantified) universe is introduced as greater or equal to `Set`.

Conversion and unification

The semantics of conversion and unification have to be modified a little to account for the new universe instance arguments to polymorphic references. The semantics respect the fact that definitions are transparent, so indistinguishable from their *bodies* during conversion.

This is accomplished by changing one rule of unification, the first- order approximation rule, which applies when two applicative terms with the same head are compared. It tries to short-cut unfolding by comparing the arguments directly. In case the *constant* is universe polymorphic, we allow this rule to fire only when unifying the universes results in instantiating a so-called flexible universe variables (not given by the user). Similarly for conversion, if such an equation of applicative terms fail due to a universe comparison not being satisfied, the terms are unfolded. This change implies that conversion and unification can have different unfolding behaviors on the same development with universe polymorphism switched on or off.

Minimization

Universe polymorphism with cumulativity tends to generate many useless inclusion constraints in general. Typically at each application of a polymorphic *constant* `f`, if an argument has expected type `Type@{i}` and is given a term of type `Type@{j}`, a $j \leq i$ constraint will be generated. It is however often the case that an equation $j = i$ would be more appropriate, when `f`'s universes are fresh for example. Consider the following example:

```
Definition id0 := @pidentity nat 0.

Print id0.
    id0@{} = pidentity@{Set} 0
         : nat
```

This definition is elaborated by minimizing the universe of `id0` to level `Set` while the more general definition would keep the fresh level `i` generated at the application of `id` and a constraint that `Set` $\leq i$. This minimization process is

applied only to fresh universe variables. It simply adds an equation between the variable and its lower bound if it is an atomic universe (i.e. not an algebraic max() universe).

Flag: `Universe Minimization ToSet`
Turning this *flag* off (it is on by default) disallows minimization to the sort `Set` and only collapses floating universes between themselves.

Explicit Universes

universe_name ::= *qualid*
| **Set**
| **Prop**

univ_annot ::= **@{** *universe_level* * **}**

universe_level ::= **Set**
| **Prop**
| **Type**
| **_**
| *qualid*

univ_decl ::= **@{** *ident* * + ? | *univ_constraint* * , + ? ? **}**

cumul_univ_decl ::= **@{** + | = | * ? *ident* * + ? | *univ_constraint* * , + ? ? **}**

univ_constraint ::= *universe_name* **<** | **=** | **<=** *universe_name*

The syntax has been extended to allow users to explicitly bind names to universes and explicitly instantiate polymorphic definitions.

Command: `Universe` *`ident`* +
In the monomorphic case, declares new global universes with the given names. Global universe names live in a separate namespace. The command supports the *`universes(polymorphic)`* attribute (or the `Polymorphic` legacy attribute) only in sections, meaning the universe quantification will be discharged for each section definition independently.

Error: `Polymorphic universes can only be declared inside sections, use Monomorphic Universe instead.`

Command: `Constraint` *`univ_constraint`* + ,
Declares new constraints between named universes.

If consistent, the constraints are then enforced in the global environment. Like *`Universe`*, it can be used with the *`universes(polymorphic)`* attribute (or the `Polymorphic` legacy attribute) in sections only to declare constraints discharged at section closing time. One cannot declare a global constraint on polymorphic universes.

Error: `Undeclared universe` *`ident`*`.`

Error: `Universe inconsistency.`

Error:
`Polymorphic universe constraints can only be declared inside sections, use Monomorphic Constraint instead`

Printing universes

Flag: `Printing Universes`

Turn this *flag* on to activate the display of the actual level of each occurrence of `Type`. See *Sorts* for details. This wizard flag, in combination with *`Printing All`* can help to diagnose failures to unify terms apparently identical but internally different in the Calculus of Inductive Constructions.

Command: `Print` `Sorted`? `Universes` `Subgraph ( qualid* )`? *`string`*?

This command can be used to print the constraints on the internal level of the occurrences of Type (see *Sorts*).

The **`Subgraph`** clause limits the printed graph to the requested names (adjusting constraints to preserve the implied transitive constraints between kept universes).

The **`Sorted`** clause makes each universe equivalent to a numbered label reflecting its level (with a linear ordering) in the universe hierarchy.

`string` is an optional output filename. If *`string`* ends in `.dot` or `.gv`, the constraints are printed in the DOT language, and can be processed by Graphviz tools. The format is unspecified if `string` doesn't end in `.dot` or `.gv`.

Polymorphic definitions

For polymorphic definitions, the declaration of (all) universe levels introduced by a definition uses the following syntax:

```
Polymorphic Definition le@{i j} (A : Type@{i}) : Type@{j} := A.

Print le.
    le@{i j} =
    fun A : Type@{i} => A
         : Type@{i} -> Type@{j}
    (* i j |= i <= j *)

    Arguments le _%type_scope
```

During refinement we find that `j` must be larger or equal than `i`, as we are using `A : Type@{i} <= Type@{j}`, hence the generated constraint. At the end of a definition or proof, we check that the only remaining universes are the ones declared. In the term and in general in proof mode, introduced universe names can be referred to in terms. Note that local universe names shadow global universe names. During a proof, one can use *`Show Universes`* to display the current context of universes.

It is possible to provide only some universe levels and let Coq infer the others by adding a + in the list of bound universe levels:

```
Fail Definition foobar@{u} : Type@{u} := Type.
    The command has indeed failed with message:
    Universe {Top.73} is unbound.

Definition foobar@{u +} : Type@{u} := Type.
    foobar is defined

Set Printing Universes.
Print foobar.
    foobar@{u u0} = Type@{u0}
         : Type@{u}
    (* u u0 |= u0 < u *)
```

This can be used to find which universes need to be explicitly bound in a given definition.

Definitions can also be instantiated explicitly, giving their full instance:

```
Check (pidentity@{Set}).
    pidentity@{Set}
         : ?A -> ?A
    where
    ?A : [ |- Set]

Monomorphic Universes k l.
Check (le@{k l}).
    le@{k l}
         : Type@{k} -> Type@{l}
    (* {} |= k <= l *)
```

User-named universes and the anonymous universe implicitly attached to an explicit `Type` are considered rigid for unification and are never minimized. Flexible anonymous universes can be produced with an underscore or by omitting the annotation to a polymorphic definition.

```
Check (fun x => x) : Type -> Type.
    (fun x : Type@{Top.78} => x) : Type@{Top.78} -> Type@{Top.79}
         : Type@{Top.78} -> Type@{Top.79}
    (* {Top.79 Top.78} |= Top.78 <= Top.79 *)

Check (fun x => x) : Type -> Type@{_}.
    (fun x : Type@{Top.80} => x) : Type@{Top.80} -> Type@{Top.80}
         : Type@{Top.80} -> Type@{Top.80}
    (* {Top.80} |=  *)


Check le@{k _}.
    le@{k k}
         : Type@{k} -> Type@{k}

Check le.
    le@{Top.83 Top.83}
         : Type@{Top.83} -> Type@{Top.83}
    (* {Top.83} |=  *)
```

Flag: `Strict Universe Declaration`
Turning this *flag* off allows one to freely use identifiers for universes without declaring them first, with the semantics that the first use declares it. In this mode, the universe names are not associated with the definition or proof once it has been defined. This is meant mainly for debugging purposes.

Flag: `Private Polymorphic Universes`
This *flag*, on by default, removes universes which appear only in the *body* of an opaque polymorphic definition from the definition's universe arguments. As such, no value needs to be provided for these universes when instantiating the definition. Universe constraints are automatically adjusted.

Consider the following definition:

```
Lemma foo@{i} : Type@{i}.
    1 goal

      ============================
      Type@{i}

Proof. exact Type. Qed.
```

(continues on next page)

(continued from previous page)

```
    No more goals.

Print foo.
    foo@{i} =
    Type@{Top.86}
         : Type@{i}
    (* Public universes:
    i |= Set < i
    Private universes:
    {Top.86} |= Top.86 < i *)
```

The universe `Top.xxx` for the `Type` in the *body* cannot be accessed, we only care that one exists for any instantiation of the universes appearing in the type of `foo`. This is guaranteed when the transitive constraint `Set <= Top.xxx < i` is verified. Then when using the *constant* we don't need to put a value for the inner universe:

```
Check foo@{_}.
    foo@{Top.87}
         : Type@{Top.87}
    (* {Top.87} |= Set < Top.87 *)
```

and when not looking at the *body* we don't mention the private universe:

```
About foo.
    foo@{i} : Type@{i}
    (* i |= Set < i *)

    foo is universe polymorphic
    foo is opaque
    Expands to: Constant Top.foo
```

To recover the same behaviour with regard to universes as `Defined`, the *Private Polymorphic Universes* flag may be unset:

```
Unset Private Polymorphic Universes.

Lemma bar : Type. Proof. exact Type. Qed.
    1 goal

      ============================
      Type@{Top.88}


    No more goals.

About bar.
    bar@{u u0} : Type@{u}
    (* u u0 |= u0 < u *)

    bar is universe polymorphic
    bar is opaque
    Expands to: Constant Top.bar

Fail Check bar@{_}.
    The command has indeed failed with message:
    Universe instance should have length 2.

Check bar@{_ _}.
```

(continues on next page)

(continued from previous page)

```
    bar@{Top.91
    Top.92}
         : Type@{Top.91}
    (* {Top.92 Top.91} |= Top.92 < Top.91 *)
```

Note that named universes are always public.

```
Set Private Polymorphic Universes.
Unset Strict Universe Declaration.

Lemma baz : Type@{outer}. Proof. exact Type@{inner}. Qed.
    1 goal

      ============================
      Type@{outer}


    No more goals.

About baz.
    baz@{outer inner} : Type@{outer}
    (* outer inner |= inner < outer *)

    baz is universe polymorphic
    baz is opaque
    Expands to: Constant Top.baz
```

Universe polymorphism and sections

Variables, *Context*, *Universe* and *Constraint* in a section support polymorphism. This means that the universe variables and their associated constraints are discharged polymorphically over definitions that use them. In other words, two definitions in the section sharing a common variable will both get parameterized by the universes produced by the variable declaration. This is in contrast to a "mononorphic" variable which introduces global universes and constraints, making the two definitions depend on the *same* global universes associated with the variable.

It is possible to mix universe polymorphism and monomorphism in sections, except in the following ways:

- no monomorphic constraint may refer to a polymorphic universe:

```
Section Foo.

  Polymorphic Universe i.
  Fail Constraint i = i.
    The command has indeed failed with message:
    Cannot add monomorphic constraints which refer to section polymorphic␣
↪universes.
```

This includes constraints implicitly declared by commands such as *Variable*, which may need to be used with universe polymorphism activated (locally by attribute or globally by option):

```
Fail Variable A : (Type@{i} : Type).
    The command has indeed failed with message:
    Cannot add monomorphic constraints which refer to section polymorphic␣
↪universes.
```

(continues on next page)

(continued from previous page)

```
Polymorphic Variable A : (Type@{i} : Type).
    A is declared
```

(in the above example the anonymous `Type` constrains polymorphic universe `i` to be strictly smaller.)

- no monomorphic *constant* or inductive may be declared if polymorphic universes or universe constraints are present.

These restrictions are required in order to produce a sensible result when closing the section (the requirement on *constants* and inductive types is stricter than the one on constraints, because constants and inductives are abstracted by *all* the section's polymorphic universes and constraints).

2.1.15 SProp (proof irrelevant propositions)

> **Warning:** The status of strict propositions is experimental.
>
> In particular, conversion checking through bytecode or native code compilation currently does not understand proof irrelevance.

This section describes the extension of Coq with definitionally proof irrelevant propositions (types in the sort SProp, also known as strict propositions) as described in [GCST19].

Use of SProp may be disabled by passing `-disallow-sprop` to the Coq program or by turning the *`Allow StrictProp`* flag off.

Flag: `Allow StrictProp`
This *flag* enables or disables the use of SProp. It is enabled by default. The command-line flag `-disallow-sprop` disables SProp at startup.

Error: `SProp is disallowed because the "Allow StrictProp" flag is off.`

Some of the definitions described in this document are available through `Coq.Logic.StrictProp`, which see.

Basic constructs

The purpose of SProp is to provide types where all elements are convertible:

```
Theorem irrelevance (A : SProp) (P : A -> Prop) : forall x : A, P x -> forall y : A,␣
↪P y.
    1 goal

      A : SProp
      P : A -> Prop
      ============================
      forall x : A, P x -> forall y : A, P y

Proof.
intros * Hx *.
    1 goal

      A : SProp
      P : A -> Prop
      x : A
      Hx : P x
      y : A
```

(continues on next page)

(continued from previous page)

```
  ============================
  P y

exact Hx.
No more goals.

Qed.
```

Since we have definitional *η-expansion* for functions, the property of being a type of definitionally irrelevant values is impredicative, and so is SProp:

```
Check fun (A:Type) (B:A -> SProp) => (forall x:A, B x) : SProp.
fun (A : Type) (B : A -> SProp) => (forall x : A, B x) : SProp
     : forall A : Type, (A -> SProp) -> SProp
```

In order to keep conversion tractable, cumulativity for SProp is forbidden, unless the *Cumulative StrictProp* flag is turned on:

```
Fail Check (fun (A:SProp) => A : Type).
The command has indeed failed with message:
In environment
A : SProp
The term "A" has type "SProp" while it is expected to have type "Type".

Set Cumulative StrictProp.
Check (fun (A:SProp) => A : Type).
fun A : SProp => A : Type
     : SProp -> Type
```

We can explicitly lift strict propositions into the relevant world by using a wrapping inductive type. The inductive stops definitional proof irrelevance from escaping.

```
Inductive Box (A:SProp) : Prop := box : A -> Box A.
Arguments box {_} _.

Fail Check fun (A:SProp) (x y : Box A) => eq_refl : x = y.
The command has indeed failed with message:
In environment
A : SProp
x : Box A
y : Box A
The term "eq_refl" has type "x = x" while it is expected to have type
"x = y" (cannot unify "x" and "y").

Definition box_irrelevant (A:SProp) (x y : Box A) : x = y
  := match x, y with box x, box y => eq_refl end.
```

In the other direction, we can use impredicativity to ”squash” a relevant type, making an irrelevant approximation.

```
Definition iSquash (A:Type) : SProp
  := forall P : SProp, (A -> P) -> P.
Definition isquash A : A -> iSquash A
  := fun a P f => f a.
Definition iSquash_sind A (P : iSquash A -> SProp) (H : forall x : A, P (isquash A x))
  : forall x : iSquash A, P x
  := fun x => x (P x) (H : A -> P x).
```

Or more conveniently (but equivalently)

```
Inductive Squash (A:Type) : SProp := squash : A -> Squash A.
```

Most inductives types defined in SProp are squashed types, i.e. they can only be eliminated to construct proofs of other strict propositions. Empty types are the only exception.

```
Inductive sEmpty : SProp := .

Check sEmpty_rect.
    sEmpty_rect
         : forall (P : sEmpty -> Type) (s : sEmpty), P s
```

Note: Eliminators to strict propositions are called `foo_sind`, in the same way that eliminators to propositions are called `foo_ind`.

Primitive records in SProp are allowed when fields are strict propositions, for instance:

```
Set Primitive Projections.
Record sProd (A B : SProp) : SProp := { sfst : A; ssnd : B }.
```

On the other hand, to avoid having definitionally irrelevant types in non-SProp sorts (through record η-extensionality), primitive records in relevant sorts must have at least one relevant field.

```
Set Warnings "+non-primitive-record".
Fail Record rBox (A:SProp) : Prop := rbox { runbox : A }.
    The command has indeed failed with message:
    The record rBox could not be defined as a primitive record
    [non-primitive-record,record]

Record ssig (A:Type) (P:A -> SProp) : Type := { spr1 : A; spr2 : P spr1 }.
```

Note that `rBox` works as an emulated record, which is equivalent to the Box inductive.

Encodings for strict propositions

The elimination for unit types can be encoded by a trivial function thanks to proof irrelevance:

```
Inductive sUnit : SProp := stt.
Definition sUnit_rect (P:sUnit->Type) (v:P stt) (x:sUnit) : P x := v.
```

By using empty and unit types as base values, we can encode other strict propositions. For instance:

```
Definition is_true (b:bool) : SProp := if b then sUnit else sEmpty.

Definition is_true_eq_true b : is_true b -> true = b
  := match b with
     | true => fun _ => eq_refl
     | false => sEmpty_ind _
     end.

Definition eq_true_is_true b (H:true=b) : is_true b
  := match H in _ = x return is_true x with eq_refl => stt end.
```

Definitional UIP

Flag: `Definitional UIP`
This *flag*, off by default, allows the declaration of non-squashed inductive types with 1 constructor which takes no argument in SProp. Since this includes equality types, it provides definitional uniqueness of identity proofs.

Because squashing is a universe restriction, unsetting *Universe Checking* is stronger than setting *Definitional UIP*.

Definitional UIP involves a special reduction rule through which reduction depends on conversion. Consider the following code:

```
Set Definitional UIP.

Inductive seq {A} (a:A) : A -> SProp :=
  srefl : seq a a.

Axiom e : seq 0 0.
Definition hidden_arrow := match e return Set with srefl _ => nat -> nat end.

Check (fun (f : hidden_arrow) (x:nat) => (f : nat -> nat) x).
```

By the usual reduction rules `hidden_arrow` is a stuck match, but by proof irrelevance `e` is convertible to `srefl 0` and then by congruence `hidden_arrow` is convertible to `nat -> nat`.

The special reduction reduces any match on a type which uses definitional UIP when the indices are convertible to those of the constructor. For `seq`, this means a match on a value of type `seq x y` reduces if and only if `x` and `y` are convertible.

Such matches are indicated in the printed representation by inserting a cast around the discriminee:

```
    hidden_arrow = match e : seq 0 0 with
                   | srefl _ => nat -> nat
                   end
         : Set
```

Non Termination with UIP

The special reduction rule of UIP combined with an impredicative sort breaks termination of reduction [AC19]:

```
Axiom all_eq : forall (P Q:Prop), P -> Q -> seq P Q.
    all_eq is declared


Definition transport (P Q:Prop) (x:P) (y:Q) : Q
:= match all_eq P Q x y with srefl _ => x end.
    transport is defined


Definition top : Prop := forall P : Prop, P -> P.
    top is defined


Definition c : top :=
  fun P p =>
  transport
  (top -> top)
  P
```

(continues on next page)

(continued from previous page)

```
  (fun x : top => x (top -> top) (fun x => x) x)
  p.
    c is defined


Fail Timeout 1 Eval lazy in c (top -> top) (fun x => x) c.
    The command has indeed failed with message:
    Timeout!
```

The term `c (top -> top) (fun x => x) c` infinitely reduces to itself.

Issues with non-cumulativity

During normal term elaboration, we don't always know that a type is a strict proposition early enough. For instance:

```
Definition constant_0 : ?[T] -> nat := fun _ : sUnit => 0.
```

While checking the type of the constant, we only know that `?[T]` must inhabit some sort. Putting it in some floating universe `u` would disallow instantiating it by `sUnit : SProp`.

In order to make the system usable without having to annotate every instance of SProp, we consider SProp to be a subtype of every universe during elaboration (i.e. outside the kernel). Then once we have a fully elaborated term it is sent to the kernel which will check that we didn't actually need cumulativity of SProp (in the example above, `u` doesn't appear in the final term).

This means that some errors will be delayed until `Qed`:

```
Lemma foo : Prop.
Proof. pose (fun A : SProp => A : Type); exact True.


Fail Qed.
    The command has indeed failed with message:
    In environment
    A : SProp
    The term "A" has type "SProp" while it is expected to have type "Type".

Abort.
```

Flag: `Elaboration StrictProp Cumulativity`
Unset this *flag* (it is on by default) to be strict with regard to SProp cumulativity during elaboration.

The implementation of proof irrelevance uses inferred "relevance" marks on binders to determine which variables are irrelevant. Together with non-cumulativity this allows us to avoid retyping during conversion. However during elaboration cumulativity is allowed and so the algorithm may miss some irrelevance:

```
Fail Definition late_mark := fun (A:SProp) (P:A -> Prop) x y (v:P x) => v : P y.
    The command has indeed failed with message:
    In environment
    A : SProp
    P : A -> Prop
    x : A
    y : A
    v : P x
    The term "v" has type "P x" while it is expected to have type "P y".
```

The binders for `x` and `y` are created before their type is known to be `A`, so they're not marked irrelevant. This can be avoided with sufficient annotation of binders (see `irrelevance` at the beginning of this chapter) or by bypassing the conversion check in tactics.

```
Definition late_mark := fun (A:SProp) (P:A -> Prop) x y (v:P x) =>
  ltac:(exact_no_check v) : P y.
```

The kernel will re-infer the marks on the fully elaborated term, and so correctly converts `x` and `y`.

Warning: `Bad relevance`
This is a developer warning, disabled by default. It is emitted by the kernel when it is passed a term with incorrect relevance marks. To avoid conversion issues as in `late_mark` you may wish to use it to find when your tactics are producing incorrect marks.

Flag: `Cumulative StrictProp`
Set this *flag* (it is off by default) to make the kernel accept cumulativity between SProp and other universes. This makes typechecking incomplete.

2.2 Language extensions

Elaboration extends the language accepted by the Coq kernel to make it easier to use. For example, this lets the user omit most type annotations because they can be inferred, call functions with implicit arguments which will be inferred as well, extend the syntax with notations, factorize branches when pattern-matching, etc. In this chapter, we present these language extensions and we give some explanations on how this language is translated down to the core language presented in the *previous chapter*.

2.2.1 Existential variables

Existential variables represent as yet unknown values.

term_evar : : =
| _
| ?[*ident*]
| ?[?*ident*]
| ?*ident* @{ *ident* := *term* (+ ;) } ?

Coq terms can include existential variables that represent unknown subterms that are eventually replaced with actual subterms.

Existential variables are generated in place of unsolved implicit arguments or "_" placeholders when using commands such as `Check` (see Section *Requests to the environment*) or when using tactics such as *`refine`*, as well as in place of unsolved instances when using tactics such that *`eapply`*. An existential variable is defined in a context, which is the context of variables of the placeholder which generated the existential variable, and a type, which is the expected type of the placeholder.

As a consequence of typing constraints, existential variables can be duplicated in such a way that they possibly appear in different contexts than their defining context. Thus, any occurrence of a given existential variable comes with an instance of its original context. In the simple case, when an existential variable denotes the placeholder which generated it, or is used in the same context as the one in which it was generated, the context is not displayed and the existential variable is represented by "?" followed by an identifier.

```
Parameter identity : forall (X:Set), X -> X.
    identity is declared


Check identity _ _.
    identity ?X ?y
         : ?X
    where
    ?X : [ |- Set]
    ?y : [ |- ?X]


Check identity _ (fun x => _).
    identity (forall x : ?S, ?S0) (fun x : ?S => ?y)
         : forall x : ?S, ?S0
    where
    ?S : [ |- Set]
    ?S0 : [x : ?S |- Set]
    ?y : [x : ?S |- ?S0]
```

In the general case, when an existential variable **?*ident*** appears outside its context of definition, its instance, written in the form **{** *ident* **:=** *term* ***** **;** **}**, is appended to its name, indicating how the variables of its defining context are instantiated. Only the variables that are defined in another context are displayed: this is why an existential variable used in the same context as its context of definition is written with no instance. This behaviour may be changed: see *Explicit displaying of existential instances for pretty-printing*.

```
Check (fun x y => _) 0 1.
    (fun x y : nat => ?y) 0 1
         : ?T@{x:=0; y:=1}
    where
    ?T : [x : nat  y : nat |- Type]
    ?y : [x : nat  y : nat |- ?T]
```

Existential variables can be named by the user upon creation using the syntax **?[*ident*]**. This is useful when the existential variable needs to be explicitly handled later in the script (e.g. with a named-goal selector, see *Goal selectors*).

Inferable subterms

Expressions often contain redundant pieces of information. Subterms that can be automatically inferred by Coq can be replaced by the symbol _ and Coq will guess the missing piece of information.

Explicit displaying of existential instances for pretty-printing

Flag: `Printing Existential Instances`
This *flag* (off by default) activates the full display of how the context of an existential variable is instantiated at each of the occurrences of the existential variable.

```
Check (fun x y => _) 0 1.
    (fun x y : nat => ?y) 0 1
         : ?T@{x:=0; y:=1}
    where
    ?T : [x : nat  y : nat |- Type]
    ?y : [x : nat  y : nat |- ?T]
```

(continues on next page)

(continued from previous page)

```
Set Printing Existential Instances.

Check (fun x y => _) 0 1.
    (fun x y : nat => ?y@{x:=x; y:=y}) 0 1
         : ?T@{x:=0; y:=1}
    where
    ?T : [x : nat  y : nat |- Type]
    ?y : [x : nat  y : nat |- ?T@{x:=x; y:=y}]
```

Solving existential variables using tactics

Instead of letting the unification engine try to solve an existential variable by itself, one can also provide an explicit hole together with a tactic to solve it. Using the syntax `ltac:(tacexpr)`, the user can put a tactic anywhere a term is expected. The order of resolution is not specified and is implementation-dependent. The inner tactic may use any variable defined in its scope, including repeated alternations between variables introduced by term binding as well as those introduced by tactic binding. The expression `tacexpr` can be any tactic expression as described in *Ltac*.

```
Definition foo (x : nat) : nat := ltac:(exact x).
    foo is defined
```

This construction is useful when one wants to define complicated terms using highly automated tactics without resorting to writing the proof-term by means of the interactive proof engine.

2.2.2 Implicit arguments

An implicit argument of a function is an argument which can be inferred from contextual knowledge. There are different kinds of implicit arguments that can be considered implicit in different ways. There are also various commands to control the setting or the inference of implicit arguments.

The different kinds of implicit arguments

Implicit arguments inferable from the knowledge of other arguments of a function

The first kind of implicit arguments covers the arguments that are inferable from the knowledge of the type of other arguments of the function, or of the type of the surrounding context of the application. Especially, such implicit arguments correspond to parameters dependent in the type of the function. Typical implicit arguments are the type arguments in polymorphic functions. There are several kinds of such implicit arguments.

Strict Implicit Arguments

An implicit argument can be either strict or non strict. An implicit argument is said to be *strict* if, whatever the other arguments of the function are, it is still inferable from the type of some other argument. Technically, an implicit argument is strict if it corresponds to a parameter which is not applied to a variable which itself is another parameter of the function (since this parameter may erase its arguments), not in the body of a match, and not itself applied or matched against patterns (since the original form of the argument can be lost by reduction).

For instance, the first argument of

```
cons: forall A:Set, A -> list A -> list A
```

in module `List.v` is strict because `list` is an inductive type and `A` will always be inferable from the type `list A` of the third argument of `cons`. Also, the first argument of `cons` is strict with respect to the second one, since the first argument is exactly the type of the second argument. On the contrary, the second argument of a term of type

```
forall P:nat->Prop, forall n:nat, P n -> ex nat P
```

is implicit but not strict, since it can only be inferred from the type `P n` of the third argument and if `P` is, e.g., `fun _ => True`, it reduces to an expression where `n` does not occur any longer. The first argument `P` is implicit but not strict either because it can only be inferred from `P n` and `P` is not canonically inferable from an arbitrary `n` and the normal form of `P n`. Consider, e.g., that `n` is 0 and the third argument has type `True`, then any `P` of the form

```
fun n => match n with 0 => True | _ => anything end
```

would be a solution of the inference problem.

Contextual Implicit Arguments

An implicit argument can be *contextual* or not. An implicit argument is said to be *contextual* if it can be inferred only from the knowledge of the type of the context of the current expression. For instance, the only argument of:

```
nil : forall A:Set, list A
```

is contextual. Similarly, both arguments of a term of type:

```
forall P:nat->Prop, forall n:nat, P n \/ n = 0
```

are contextual (moreover, `n` is strict and `P` is not).

Reversible-Pattern Implicit Arguments

There is another class of implicit arguments that can be reinferred unambiguously if all the types of the remaining arguments are known. This is the class of implicit arguments occurring in the type of another argument in position of reversible pattern, which means it is at the head of an application but applied only to uninstantiated distinct variables. Such an implicit argument is called *reversible- pattern implicit argument*. A typical example is the argument `P` of nat_rec in

```
nat_rec : forall P : nat -> Set, P 0 ->
  (forall n : nat, P n -> P (S n)) -> forall x : nat, P x
```

(`P` is reinferable by abstracting over `n` in the type `P n`).

See *Controlling reversible-pattern implicit arguments* for the automatic declaration of reversible-pattern implicit arguments.

Implicit arguments inferable by resolution

This corresponds to a class of non-dependent implicit arguments that are solved based on the structure of their type only.

Maximal and non-maximal insertion of implicit arguments

When a function is partially applied and the next argument to apply is an implicit argument, the application can be interpreted in two ways. If the next argument is declared as *maximally inserted*, the partial application will include that argument. Otherwise, the argument is *non-maximally inserted* and the partial application will not include that argument.

Each implicit argument can be declared to be inserted maximally or non maximally. In Coq, maximally inserted implicit arguments are written between curly braces "{ }" and non-maximally inserted implicit arguments are written in square brackets "[ ]".

See also:

Maximal Implicit Insertion

Trailing Implicit Arguments

An implicit argument is considered *trailing* when all following arguments are implicit. Trailing implicit arguments must be declared as maximally inserted; otherwise they would never be inserted.

Error: Argument *name* is a trailing implicit, so it can't be declared non maximal. Please use { } instead of [ ].

For instance:

```
Fail Definition double [n] := n + n.
    The command has indeed failed with message:
    Argument n is a trailing implicit, so it can't be declared non maximal.
    Please use { } instead of [ ].
```

Casual use of implicit arguments

If an argument of a function application can be inferred from the type of the other arguments, the user can force inference of the argument by replacing it with _.

Error: Cannot infer a term for this placeholder.

Coq was not able to deduce an instantiation of a "_".

Declaration of implicit arguments

Implicit arguments can be declared when a function is declared or afterwards, using the *Arguments* command.

Implicit Argument Binders

implicit_binders ::= { *name*+ : *type*? }
| [*name*+ : *type*?]

In the context of a function definition, these forms specify that *name* is an implicit argument. The first form, with curly braces, makes *name* a maximally inserted implicit argument. The second form, with square brackets, makes *name* a non-maximally inserted implicit argument.

For example:

```
Definition id {A : Type} (x : A) : A := x.
    id is defined
```

declares the argument A of id as a maximally inserted implicit argument. A may be omitted in applications of id but may be specified if needed:

```
Definition compose {A B C} (g : B -> C) (f : A -> B) := fun x => g (f x).
    compose is defined


Goal forall A, compose id id = id (A:=A).
    1 goal

      ============================
      forall A : Type, compose id id = id
```

For non-maximally inserted implicit arguments, use square brackets:

```
Fixpoint map [A B : Type] (f : A -> B) (l : list A) : list B :=
  match l with
  | nil => nil
  | cons a t => cons (f a) (map f t)
  end.
    map is defined
    map is recursively defined (guarded on 4th argument)


Print Implicit map.
    map : forall [A B : Type], (A -> B) -> list A -> list B

    Arguments A, B are implicit
```

For (co-)inductive datatype declarations, the semantics are the following: an inductive parameter declared as an implicit argument need not be repeated in the inductive definition and will become implicit for the inductive type and the constructors. For example:

```
Inductive list {A : Type} : Type :=
| nil : list
| cons : A -> list -> list.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined


Print list.
    Inductive list (A : Type) : Type :=  nil : list | cons : A -> list -> list.

    Arguments list {A}%type_scope
    Arguments nil {A}%type_scope
    Arguments cons {A}%type_scope _ _
```

One can always specify the parameter if it is not uniform using the usual implicit arguments disambiguation syntax.

The syntax is also supported in internal binders. For instance, in the following kinds of expressions, the type of each declaration present in *binder** can be bracketed to mark the declaration as implicit: * **fun**

(*ident***:forall** *binder* * **,** *type***) => ***term***, * forall (***ident***:forall** *binder* * **,** *type***),** *type*, * **let** *ident* *binder* * **:=** *term* **in** *term*, * **fix** *ident* *binder* * **:=** *term* **in** *term* and * **cofix** *ident* *binder* * **:=** *term* **in** *term*.

Here is an example:

```
Axiom Ax :
  forall (f:forall {A} (a:A), A * A),
  let g {A} (x y:A) := (x,y) in
  f 0 = g 0 0.
    Ax is declared
```

Warning: `Ignoring implicit binder declaration in unexpected position`

This is triggered when setting an argument implicit in an expression which does not correspond to the type of an assumption or to the *body* of a definition. Here is an example:

```
Definition f := forall {y}, y = 0.
    map is defined
    map is recursively defined (guarded on 4th argument)
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined
    Ax is declared
    Toplevel input, characters 24-25:
    > Definition f := forall {y}, y = 0.
    >                         ^
    Warning: Ignoring implicit binder declaration in unexpected position.
    [unexpected-implicit-declaration,syntax]
    f is defined
```

Warning: `Making shadowed name of implicit argument accessible by position`

This is triggered when two variables of same name are set implicit in the same block of binders, in which case the first occurrence is considered to be unnamed. Here is an example:

```
Check let g {x:nat} (H:x=x) {x} (H:x=x) := x in 0.
    Toplevel input, characters 0-50:
    > Check let g {x:nat} (H:x=x) {x} (H:x=x) := x in 0.
    > ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Warning: Making shadowed name of implicit argument accessible by position.
    [shadowed-implicit-name,syntax]
    let g := fun (x : nat) (H : x = x) (x0 : ?A@{x0:=x}) (_ : x0 = x0) => x0 in 0
         : nat
    where
    ?A : [x0 : nat  H : x0 = x0  x : ?A |- Type] (x cannot be used)
```

Mode for automatic declaration of implicit arguments

Flag: `Implicit Arguments`
This *flag* (off by default) allows to systematically declare implicit the arguments detectable as such. Auto-detection of implicit arguments is governed by flags controlling whether strict and contextual implicit arguments have to be considered or not.

Controlling strict implicit arguments

Flag: `Strict Implicit`
When the mode for automatic declaration of implicit arguments is on, the default is to automatically set implicit only the strict implicit arguments plus, for historical reasons, a small subset of the non-strict implicit arguments. To relax this constraint and to set implicit all non strict implicit arguments by default, you can turn this *flag* off.

Flag: `Strongly Strict Implicit`
Use this *flag* (off by default) to capture exactly the strict implicit arguments and no more than the strict implicit arguments.

Controlling contextual implicit arguments

Flag: `Contextual Implicit`
By default, Coq does not automatically set implicit the contextual implicit arguments. You can turn this *flag* on to tell Coq to also infer contextual implicit argument.

Controlling reversible-pattern implicit arguments

Flag: `Reversible Pattern Implicit`
By default, Coq does not automatically set implicit the reversible-pattern implicit arguments. You can turn this *flag* on to tell Coq to also infer reversible-pattern implicit argument.

Controlling the insertion of implicit arguments not followed by explicit arguments

Flag: `Maximal Implicit Insertion`
Assuming the implicit argument mode is on, this *flag* (off by default) declares implicit arguments to be automatically inserted when a function is partially applied and the next argument of the function is an implicit one.

Combining manual declaration and automatic declaration

When some arguments are manually specified implicit with binders in a definition and the automatic declaration mode in on, the manual implicit arguments are added to the automatically declared ones.

In that case, and when the flag *`Maximal Implicit Insertion`* is set to off, some trailing implicit arguments can be inferred to be non-maximally inserted. In this case, they are converted to maximally inserted ones.

Example

```
Set Implicit Arguments.
Axiom eq0_le0 : forall (n : nat) (x : n = 0), n <= 0.
    eq0_le0 is declared

Print Implicit eq0_le0.
    eq0_le0 : forall [n : nat], n = 0 -> n <= 0

    Argument n is implicit

Axiom eq0_le0' : forall (n : nat) {x : n = 0}, n <= 0.
    Argument n is a trailing implicit, so it has been declared maximally
    inserted.
    eq0_le0' is declared

Print Implicit eq0_le0'.
    eq0_le0' : forall {n : nat}, n = 0 -> n <= 0

    Arguments n, x are implicit and maximally inserted
```

Explicit applications

In presence of non-strict or contextual arguments, or in presence of partial applications, the synthesis of implicit arguments may fail, so one may have to explicitly give certain implicit arguments of an application. Use the **(*ident* := *term*)** form of *arg* to do so, where *ident* is the name of the implicit argument and *term* is its corresponding explicit term. Alternatively, one can deactivate the hiding of implicit arguments for a single function application using the **@*qualid_annotated*** ***term1***+ form of *term_application*.

Example: Syntax for explicitly giving implicit arguments (continued)

```
Parameter X : Type.
    X is declared

Definition Relation := X -> X -> Prop.
    eq0_le0 is declared
    Argument n is a trailing implicit, so it has been declared maximally
    inserted.
    eq0_le0' is declared
    X is declared
    Relation is defined

Definition Transitivity (R:Relation) := forall x y:X, R x y -> forall z:X, R y z -> R␣
↪x z.
    Transitivity is defined

Parameters (R : Relation) (p : Transitivity R).
    R is declared
    p is declared

Arguments p : default implicits.
Print Implicit p.
    p : forall [x y : X], R x y -> forall z : X, R y z -> R x z

    Arguments x, y, z are implicit
```

(continues on next page)

(continued from previous page)

```
Parameters (a b c : X) (r1 : R a b) (r2 : R b c).
    a is declared
    b is declared
    c is declared
    r1 is declared
    r2 is declared

Check (p r1 (z:=c)).
    p r1 (z:=c)
         : R b c -> R a c


Check (p (x:=a) (y:=b) r1 (z:=c) r2).
    p r1 r2
         : R a c
```

Displaying implicit arguments

Command: `Print Implicit` *reference*
Displays the implicit arguments associated with an object, identifying which arguments are applied maximally or not.

Displaying implicit arguments when pretty-printing

Flag: `Printing Implicit`
By default, the basic pretty-printing rules hide the inferable implicit arguments of an application. Turn this *flag* on to force printing all implicit arguments.

Flag: `Printing Implicit Defensive`
By default, the basic pretty-printing rules display implicit arguments that are not detected as strict implicit arguments. This "defensive" mode can quickly make the display cumbersome so this can be deactivated by turning this *flag* off.

See also:

Printing All.

Interaction with subtyping

When an implicit argument can be inferred from the type of more than one of the other arguments, then only the type of the first of these arguments is taken into account, and not an upper type of all of them. As a consequence, the inference of the implicit argument of "=" fails in

```
Fail Check nat = Prop.
    The command has indeed failed with message:
    The term "Prop" has type "Type" while it is expected to have type
    "Set" (universe inconsistency: Cannot enforce Set+1 <= Set).
```

but succeeds in

```
Check Prop = nat.
    Prop = nat
         : Prop
```

Deactivation of implicit arguments for parsing

term_explicit ::= @ *qualid_annotated*

This syntax can be used to disable implicit arguments for a single function.

Example

The function `id` has one implicit argument and one explicit argument.

```
Check (id 0).
    id 0
         : nat

Definition id' := @id.
    id' is defined
```

The function `id'` has no implicit argument.

```
Check (id' nat 0).
    id' nat 0
         : nat
```

Flag: `Parsing Explicit`
Turning this *flag* on (it is off by default) deactivates the use of implicit arguments.

In this case, all arguments of *constants*, inductive types, constructors, etc, including the arguments declared as implicit, have to be given as if no arguments were implicit. By symmetry, this also affects printing.

Example

We can reproduce the example above using the *`Parsing Explicit`* flag:

```
Set Parsing Explicit.
Definition id' := id.
    id' is defined

Unset Parsing Explicit.
Check (id 1).
    id 1
         : nat

Check (id' nat 1).
    id' nat 1
         : nat
```

Implicit types of variables

It is possible to bind variable names to a given type (e.g. in a development using arithmetic, it may be convenient to bind the names `n` or `m` to the type `nat` of natural numbers).

Command: **`Implicit`** **`Type`** | **`Types`** *reserv_list*

reserv_list ::= (*simple_reserv*)+
| *simple_reserv*

simple_reserv ::= *ident*+ **:** *type*

Sets the type of bound variables starting with *ident* (either *ident* itself or *ident* followed by one or more single quotes, underscore or digits) to *type* (unless the bound variable is already declared with an explicit type, in which case, that type will be used).

Example

```
Require Import List.

Implicit Types m n : nat.

Lemma cons_inj_nat : forall m n l, n :: l = m :: l -> n = m.
    1 goal

  ============================
  forall m n (l : list nat), n :: l = m :: l -> n = m

Proof. intros m n. Abort.
    1 goal

  m, n : nat
  ============================
  forall l : list nat, n :: l = m :: l -> n = m


Lemma cons_inj_bool : forall (m n:bool) l, n :: l = m :: l -> n = m.
    1 goal

  ============================
  forall (m n : bool) (l : list bool), n :: l = m :: l -> n = m

Abort.
```

Flag: **`Printing Use Implicit Types`**
By default, the type of bound variables is not printed when the variable name is associated with an implicit type which matches the actual type of the variable. This feature can be deactivated by turning this *flag* off.

Implicit generalization

generalizing_binder ::= `(*typeclass_constraint* + ,)
| `{ *typeclass_constraint* + , }
| `[*typeclass_constraint* + ,]

typeclass_constraint ::= ! ? *term*
| { *name* } : ! ? *term*
| *name* : ! ? *term*

term_generalizing ::= `{ *term* }
| `(*term*)

Implicit generalization is an automatic elaboration of a statement with free variables into a closed statement where these variables are quantified explicitly. Use the `Generalizable` command to designate which variables should be generalized.

It is activated within a binder by prefixing it with `, and for terms by surrounding it with `{ }, or `[ ] or `( ).

Terms surrounded by `{ } introduce their free variables as maximally inserted implicit arguments, terms surrounded by `[ ] introduce them as non-maximally inserted implicit arguments and terms surrounded by `( ) introduce them as explicit arguments.

Generalizing binders always introduce their free variables as maximally inserted implicit arguments. The binder itself introduces its argument as usual.

In the following statement, `A` and `y` are automatically generalized, `A` is implicit and `x`, `y` and the anonymous equality argument are explicit.

```
Generalizable All Variables.

Definition sym `(x:A) : `(x = y -> y = x) := fun _ p => eq_sym p.
    sym is defined


Print sym.
    sym =
    fun (A : Type) (x y : A) (p : x = y) => eq_sym p
         : forall (A : Type) (x y : A), x = y -> y = x

    Arguments sym {A}%type_scope _ _ _
```

Dually to normal binders, the name is optional but the type is required:

```
Check (forall `{x = y :> A}, y = x).
    forall (A : Type) (x y : A), x = y -> y = x
         : Prop
```

When generalizing a binder whose type is a typeclass, its own class arguments are omitted from the syntax and are generalized using automatic names, without instance search. Other arguments are also generalized unless provided. This produces a fully general statement. this behaviour may be disabled by prefixing the type with a `!` or by forcing the typeclass name to be an explicit application using `@` (however the later ignores implicit argument information).

```
Class Op (A:Type) := op : A -> A -> A.

Class Commutative (A:Type) `(Op A) := commutative : forall x y, op x y = op y x.
Instance nat_op : Op nat := plus.
    nat_op is defined


Set Printing Implicit.
Check (forall `{Commutative }, True).
    forall (A : Type) (H : Op A), Commutative A H -> True
         : Prop

Check (forall `{Commutative nat}, True).
    forall H : Op nat, Commutative nat H -> True
         : Prop

Fail Check (forall `{Commutative nat _}, True).
    The command has indeed failed with message:
    Typeclass does not expect more arguments

Fail Check (forall `{!Commutative nat}, True).
    The command has indeed failed with message:
    The term "Commutative nat" has type "Op nat -> Prop"
    which should be Set, Prop or Type.

Arguments Commutative _ {_}.
Check (forall `{!Commutative nat}, True).
    @Commutative nat nat_op -> True
         : Prop

Check (forall `{@Commutative nat plus}, True).
    @Commutative nat Nat.add -> True
         : Prop
```

Multiple binders can be merged using `,` as a separator:

```
Check (forall `{Commutative A, Hnat : !Commutative nat}, True).
    forall (A : Type) (H : Op A),
    @Commutative A H -> @Commutative nat nat_op -> True
         : Prop
```

Command:

Generalizable | **Variable** | **Variables** | *ident*+ | **All Variables** | **No Variables**

Controls the set of generalizable identifiers. By default, no variables are generalizable.

This command supports the *global* attribute.

The **Variable** | **Variables** *ident*+ form allows generalization of only the given ***ident***s. Using this command multiple times adds to the allowed identifiers. The other forms clear the list of ***ident***s.

The **All Variables** form generalizes all free variables in the context that appear under a generalization delimiter. This may result in confusing errors in case of typos. In such cases, the context will probably contain some unexpected generalized variables.

The **No Variables** form disables implicit generalization entirely. This is the default behavior (before any *Generalizable* command has been entered).

2.2.3 Extended pattern matching

Authors Cristina Cornes and Hugo Herbelin

This section describes the full form of pattern matching in Coq terms.

Variants and extensions of `match`

Multiple and nested pattern matching

The basic version of `match` allows pattern matching on simple patterns. As an extension, multiple nested patterns or disjunction of patterns are allowed, as in ML-like languages (cf. *Multiple patterns* and *Nested patterns*).

The extension just acts as a macro that is expanded during parsing into a sequence of match on simple patterns. Especially, a construction defined using the extended match is generally printed under its expanded form (see *Printing Matching*).

Pattern-matching on boolean values: the if expression

term_if ::= **if** *term* [**as** *name*?] **return** *term100*? **then** *term* **else** *term*

For inductive types with exactly two constructors and for pattern matching expressions that do not depend on the arguments of the constructors, it is possible to use a `if ... then ... else` notation. For instance, the definition

```
Definition not (b:bool) :=
match b with
| true => false
| false => true
end.
    not is defined
```

can be alternatively written

```
Definition not (b:bool) := if b then false else true.
    not is defined
```

More generally, for an inductive type with constructors $ident_1$ and $ident_2$, the following terms are equal:

if $term_0$ [**as** *name*?] **return** *term*? **then** $term_1$ **else** $term_2$

match $term_0$ [**as** *name*?] **return** *term*? **with** | $ident_1$ _* => $term_1$ | $ident_2$ _* => $term_2$ **end**

Example

```
Check (fun x (H:{x=0}+{x<>0}) =>
match H with
| left _ => true
| right _ => false
end).
```

(continues on next page)

(continued from previous page)

```
fun (x : nat) (H : {x = 0} + {x <> 0}) => if H then true else false
     : forall x : nat, {x = 0} + {x <> 0} -> bool
```

Notice that the printing uses the `if` syntax because `sumbool` is declared as such (see *Controlling pretty-printing of match expressions*).

Irrefutable patterns: the destructuring let variants

Pattern-matching on terms inhabiting inductive type having only one constructor can be alternatively written using `let ... in ...` constructions. There are two variants of them.

destructuring_let ::= **let (** *name* **)** **as** *name* **return** *term100* **:=** *term* **in** *term*

| **let '** *pattern* **:=** *term* **return** *term100* **in** *term*

| **let '** *pattern* **in** *pattern* **:=** *term* **return** *term100* **in** *term*

First destructuring let syntax

The expression **let (** $ident_i$ **)** **:=** $term_0$ **in** $term_1$ performs case analysis on $term_0$ whose type must be an inductive type with exactly one constructor. The number of variables $ident_i$ must correspond to the number of arguments of this constructor. Then, in $term_1$, these variables are bound to the arguments of the constructor in $term_0$. For instance, the definition

```
Definition fst (A B:Set) (H:A * B) := match H with
| pair x y => x
end.
    fst is defined
```

can be alternatively written

```
Definition fst (A B:Set) (p:A * B) := let (x, _) := p in x.
    fst is defined
```

Notice that reduction is different from regular `let ... in ...` construction since it happens only if $term_0$ is in constructor form. Otherwise, the reduction is blocked.

The pretty-printing of a definition by matching on a irrefutable pattern can either be done using `match` or the `let` construction (see Section *Controlling pretty-printing of match expressions*).

If term inhabits an inductive type with one constructor `C`, we have an equivalence between

```
let (ident₁, …, identₙ) [dep_ret_type] := term in term'
```

and

```
match term [dep_ret_type] with
C ident₁ … identₙ => term'
end
```

Second destructuring let syntax

Another destructuring let syntax is available for inductive types with one constructor by giving an arbitrary pattern instead of just a tuple for all the arguments. For example, the preceding example can be written:

```
Definition fst (A B:Set) (p:A*B) := let 'pair x _ := p in x.
    fst is defined
```

This is useful to match deeper inside tuples and also to use notations for the pattern, as the syntax `let 'p := t in b` allows arbitrary patterns to do the deconstruction. For example:

```
Definition deep_tuple (A:Set) (x:(A*A)*(A*A)) : A*A*A*A :=
let '((a,b), (c, d)) := x in (a,b,c,d).
    deep_tuple is defined


Notation " x 'With' p " := (exist _ x p) (at level 20).
    Identifier 'With' now a keyword


Definition proj1_sig' (A:Set) (P:A->Prop) (t:{ x:A | P x }) : A :=
let 'x With p := t in x.
    proj1_sig' is defined
```

When printing definitions which are written using this construct it takes precedence over let printing directives for the datatype under consideration (see Section *Controlling pretty-printing of match expressions*).

Controlling pretty-printing of match expressions

The following commands give some control over the pretty-printing of `match` expressions.

Printing nested patterns

Flag: `Printing Matching`
The Calculus of Inductive Constructions knows pattern matching only over simple patterns. It is however convenient to re-factorize nested pattern matching into a single pattern matching over a nested pattern.

When this *flag* is on (default), Coq's printer tries to do such limited re-factorization. Turning it off tells Coq to print only simple pattern matching problems in the same way as the Coq kernel handles them.

Factorization of clauses with same right-hand side

Flag: `Printing Factorizable Match Patterns`
When several patterns share the same right-hand side, it is additionally possible to share the clauses using disjunctive patterns. Assuming that the printing matching mode is on, this *flag* (on by default) tells Coq's printer to try to do this kind of factorization.

Use of a default clause

Flag: `Printing Allow Match Default Clause`
When several patterns share the same right-hand side which do not depend on the arguments of the patterns, yet an extra factorization is possible: the disjunction of patterns can be replaced with a _ default clause. Assuming that the printing matching mode and the factorization mode are on, this *flag* (on by default) tells Coq's printer to use a default clause when relevant.

Printing of wildcard patterns

Flag: `Printing Wildcard`
Some variables in a pattern may not occur in the right-hand side of the pattern matching clause. When this *flag* is on (default), the variables having no occurrences in the right-hand side of the pattern matching clause are just printed using the wildcard symbol "_".

Printing of the elimination predicate

Flag: `Printing Synth`
In most of the cases, the type of the result of a matched term is mechanically synthesizable. Especially, if the result type does not depend of the matched term. When this *flag* is on (default), the result type is not printed when Coq knows that it can re- synthesize it.

Printing matching on irrefutable patterns

If an inductive type has just one constructor, pattern matching can be written using the first destructuring let syntax.

Table: `Printing Let` *`qualid`*
This *table* specifies a set of qualids for which pattern matching is displayed using a let expression. Note that this only applies to pattern matching instances entered with `match`. It doesn't affect pattern matching explicitly entered with a destructuring `let`. Use the *`Add`* and *`Remove`* commands to update this set.

Printing matching on booleans

If an inductive type is isomorphic to the boolean type, pattern matching can be written using `if` … `then` … `else` …. This table controls which types are written this way:

Table: `Printing If` *`qualid`*
This *table* specifies a set of qualids for which pattern matching is displayed using `if` … `then` … `else` …. Use the *`Add`* and *`Remove`* commands to update this set.

This example emphasizes what the printing settings offer.

Example

```
Definition snd (A B:Set) (H:A * B) := match H with
| pair x y => y
end.
    snd is defined
```

(continues on next page)

(continued from previous page)

```
Test Printing Let for prod.
    Cases on elements of prod are printed using a `let' form


Print snd.
    snd =
    fun (A B : Set) (H : A * B) => let (_, y) := H in y
         : forall A B : Set, A * B -> B

    Arguments snd (_ _)%type_scope _


Remove Printing Let prod.

Unset Printing Synth.

Unset Printing Wildcard.

Print snd.
    snd =
    fun (A B : Set) (H : A * B) => match H return B with
                                   | (x, y) => y
                                   end
         : forall A B : Set, A * B -> B

    Arguments snd (_ _)%type_scope _
```

Conventions about unused pattern-matching variables

Pattern-matching variables that are not used on the right-hand side of => are considered the sign of a potential error. For instance, it could result from an undetected misspelled constant constructor. By default, a warning is issued in such situations.

Warning: `Unused variable` *`ident`* `catches more than one case.`
This indicates that an unused pattern variable *ident* occurs in a pattern-matching clause used to complete at least two cases of the pattern-matching problem.

The warning can be deactivated by using a variable name starting with _ or by setting `Set Warnings "-unused-pattern-matching-variable"`.

Here is an example where the warning is activated.

Example

```
Definition is_zero (o : option nat) := match o with
| Some 0 => true
| x => false
end.
    is_zero is defined
```

Patterns

The full syntax of `match` is presented in *Definition by cases: match*. Identifiers in patterns are either constructor names or variables. Any identifier that is not the constructor of an inductive or co-inductive type is considered to be a variable. A variable name cannot occur more than once in a given pattern. It is recommended to start variable names by a lowercase letter.

If a pattern has the form `c x` where `c` is a constructor symbol and x is a linear vector of (distinct) variables, it is called *simple*: it is the kind of pattern recognized by the basic version of match. On the opposite, if it is a variable `x` or has the form `c p` with `p` not only made of variables, the pattern is called *nested*.

A variable pattern matches any value, and the identifier is bound to that value. The pattern "_" (called "don't care" or "wildcard" symbol) also matches any value, but does not bind anything. It may occur an arbitrary number of times in a pattern. Alias patterns written **(*pattern* as *ident*)** are also accepted. This pattern matches the same values as *pattern* does and *ident* is bound to the matched value. A pattern of the form ***pattern* | *pattern*** is called disjunctive. A list of patterns separated with commas is also considered as a pattern and is called *multiple pattern*. However multiple patterns can only occur at the root of pattern matching equations. Disjunctions of *multiple patterns* are allowed though.

Since extended `match` expressions are compiled into the primitive ones, the expressiveness of the theory remains the same. Once parsing has finished only simple patterns remain. The original nesting of the `match` expressions is recovered at printing time. An easy way to see the result of the expansion is to toggle off the nesting performed at printing (use here *Printing Matching*), then by printing the term with *Print* if the term is a *constant*, or using the command *Check*.

The extended `match` still accepts an optional *elimination predicate* given after the keyword `return`. Given a pattern matching expression, if all the right-hand-sides of => have the same type, then this type can be sometimes synthesized, and so we can omit the return part. Otherwise the predicate after return has to be provided, like for the basicmatch.

Let us illustrate through examples the different aspects of extended pattern matching. Consider for example the function that computes the maximum of two natural numbers. We can write it in primitive syntax by:

```
Fixpoint max (n m:nat) {struct m} : nat :=
  match n with
  | O => m
  | S n' => match m with
            | O => S n'
            | S m' => S (max n' m')
            end
  end.
```

Multiple patterns

Using multiple patterns in the definition of `max` lets us write:

```
Fixpoint max (n m:nat) {struct m} : nat :=
    match n, m with
    | O, _ => m
    | S n', O => S n'
    | S n', S m' => S (max n' m')
    end.
```

which will be compiled into the previous form.

The pattern matching compilation strategy examines patterns from left to right. A match expression is generated **only** when there is at least one constructor in the column of patterns. E.g. the following example does not build a match expression.

```
Check (fun x:nat => match x return nat with
                    | y => y
                    end).
    fun x : nat => x
         : nat -> nat
```

Aliasing subpatterns

We can also use **as** *ident* to associate a name to a sub-pattern:

```
Fixpoint max (n m:nat) {struct n} : nat :=
  match n, m with
  | O, _ => m
  | S n' as p, O => p
  | S n', S m' => S (max n' m')
  end.
```

Nested patterns

Here is now an example of nested patterns:

```
Fixpoint even (n:nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S n') => even n'
  end.
```

This is compiled into:

```
Unset Printing Matching.
Print even.
    even =
    fix even (n : nat) : bool :=
      match n with
      | 0 => true
      | S n0 => match n0 with
                | 0 => false
                | S n' => even n'
                end
      end
         : nat -> bool

    Arguments even _%nat_scope
```

In the previous examples patterns do not conflict with, but sometimes it is comfortable to write patterns that admit a non trivial superposition. Consider the boolean function `lef` that given two natural numbers yields `true` if the first one is less or equal than the second one and `false` otherwise. We can write it as follows:

```
Fixpoint lef (n m:nat) {struct m} : bool :=
  match n, m with
  | O, x => true
  | x, O => false
  | S n, S m => lef n m
  end.
```

Note that the first and the second multiple pattern overlap because the couple of values `O O` matches both. Thus, what is the result of the function on those values? To eliminate ambiguity we use the *textual priority rule:* we consider patterns to be ordered from top to bottom. A value is matched by the pattern at the ith row if and only if it is not matched by some pattern from a previous row. Thus in the example, `O O` is matched by the first pattern, and so `(lef O O)` yields true.

Another way to write this function is:

```
Fixpoint lef (n m:nat) {struct m} : bool :=
  match n, m with
  | O, x => true
  | S n, S m => lef n m
  | _, _ => false
  end.
```

Here the last pattern superposes with the first two. Because of the priority rule, the last pattern will be used only for values that do not match neither the first nor the second one.

Terms with useless patterns are not accepted by the system. Here is an example:

```
Fail Check (fun x:nat =>
              match x with
              | O => true
              | S _ => false
              | x => true
              end).
    The command has indeed failed with message:
    Pattern "x" is redundant in this clause.
```

Disjunctive patterns

Multiple patterns that share the same right-hand-side can be factorized using the notation *pattern* + , + | . For instance, `max` can be rewritten as follows:

```
Fixpoint max (n m:nat) {struct m} : nat :=
  match n, m with
  | S n', S m' => S (max n' m')
  | 0, p | p, 0 => p
  end.
```

Similarly, factorization of (not necessarily multiple) patterns that share the same variables is possible by using the notation *pattern* + | . Here is an example:

```
Definition filter_2_4 (n:nat) : nat :=
  match n with
  | 2 as m | 4 as m => m
  | _ => 0
  end.
```

Nested disjunctive patterns are allowed, inside parentheses, with the notation **(** *pattern* + | **)** , as in:

```
Definition filter_some_square_corners (p:nat*nat) : nat*nat :=
  match p with
  | ((2 as m | 4 as m), (3 as n | 5 as n)) => (m,n)
```

(continues on next page)

(continued from previous page)

```
  | _ => (0,0)
  end.
```

About patterns of parametric types

Parameters in patterns

When matching objects of a parametric type, parameters do not bind in patterns. They must be substituted by "_". Consider for example the type of polymorphic lists:

```
Inductive List (A:Set) : Set :=
| nil : List A
| cons : A -> List A -> List A.
```

We can check the function *tail*:

```
Check
  (fun l:List nat =>
     match l with
     | nil _ => nil nat
     | cons _ _ l' => l'
     end).
    fun l : List nat => match l with
                        | nil _ => nil nat
                        | cons _ _ l' => l'
                        end
         : List nat -> List nat
```

When we use parameters in patterns there is an error message:

```
Fail Check
  (fun l:List nat =>
     match l with
     | nil A => nil nat
     | cons A _ l' => l'
     end).
    The command has indeed failed with message:
    The parameters do not bind in patterns; they must be replaced by '_'.
```

Flag: `Asymmetric Patterns`
This *flag* (off by default) removes parameters from constructors in patterns:

```
Set Asymmetric Patterns.
Check (fun l:List nat =>
  match l with
  | nil => nil _
  | cons _ l' => l'
  end).
    fun l : List nat => match l with
                        | @nil _ => nil nat
                        | @cons _ _ l' => l'
                        end
         : List nat -> List nat

Unset Asymmetric Patterns.
```

Implicit arguments in patterns

By default, implicit arguments are omitted in patterns. So we write:

```
Arguments nil {A}.
Arguments cons [A] _ _.
Check
  (fun l:List nat =>
     match l with
     | nil => nil
     | cons _ l' => l'
     end).
    fun l : List nat => match l with
                        | nil => nil
                        | cons _ l' => l'
                        end
         : List nat -> List nat
```

But the possibility to use all the arguments is given by "`@`" implicit explicitations (as for terms, see *Explicit applications*).

```
Check
  (fun l:List nat =>
     match l with
     | @nil _ => @nil nat
     | @cons _ _ l' => l'
     end).
    fun l : List nat => match l with
                        | nil => nil
                        | cons _ l' => l'
                        end
         : List nat -> List nat
```

Matching objects of dependent types

The previous examples illustrate pattern matching on objects of non- dependent types, but we can also use the expansion strategy to destructure objects of dependent types. Consider the type `listn` of lists of a certain length:

```
Inductive listn : nat -> Set :=
| niln : listn 0
| consn : forall n:nat, nat -> listn n -> listn (S n).
```

Understanding dependencies in patterns

We can define the function length over `listn` by:

```
Definition length (n:nat) (l:listn n) := n.
```

Just for illustrating pattern matching, we can define it by case analysis:

```
Definition length (n:nat) (l:listn n) :=
  match l with
  | niln => 0
  | consn n _ _ => S n
  end.
```

We can understand the meaning of this definition using the same notions of usual pattern matching.

When the elimination predicate must be provided

Dependent pattern matching

The examples given so far do not need an explicit elimination predicate because all the right hand sides have the same type and Coq succeeds to synthesize it. Unfortunately when dealing with dependent patterns it often happens that we need to write cases where the types of the right hand sides are different instances of the elimination predicate. The function `concat` for `listn` is an example where the branches have different types and we need to provide the elimination predicate:

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
 listn (n + m) :=
  match l in listn n return listn (n + m) with
  | niln => l'
  | consn n' a y => consn (n' + m) a (concat n' y m l')
  end.
```

The elimination predicate is `fun (n:nat) (l:listn n) => listn (n+m)`. In general if `m` has type `(I q1 … qr t1 … ts)` where `q1, …, qr` are parameters, the elimination predicate should be of the form `fun y1 … ys x : (I q1 … qr y1 … ys ) => Q`.

In the concrete syntax, it should be written : `match m as x in (I _ … _ y1 … ys) return Q with … end`. The variables which appear in the `in` and `as` clause are new and bounded in the property `Q` in the return clause. The parameters of the inductive definitions should not be mentioned and are replaced by _.

Multiple dependent pattern matching

Recall that a list of patterns is also a pattern. So, when we destructure several terms at the same time and the branches have different types we need to provide the elimination predicate for this multiple pattern. It is done using the same scheme: each term may be associated with an `as` clause and an `in` clause in order to introduce a dependent product.

For example, an equivalent definition for `concat` (even though the matching on the second term is trivial) would have been:

```
Fixpoint concat (n:nat) (l:listn n) (m:nat) (l':listn m) {struct l} :
 listn (n + m) :=
  match l in listn n, l' return listn (n + m) with
  | niln, x => x
  | consn n' a y, x => consn (n' + m) a (concat n' y m x)
  end.
```

Even without real matching over the second term, this construction can be used to keep types linked. If `a` and `b` are two `listn` of the same length, by writing

```
Check (fun n (a b: listn n) =>
 match a, b with
 | niln, b0 => tt
 | consn n' a y, bS => tt
 end).
```

we have a copy of `b` in type `listn 0` resp. `listn (S n')`.

Patterns in `in`

If the type of the matched term is more precise than an inductive applied to variables, arguments of the inductive in the `in` branch can be more complicated patterns than a variable.

Moreover, constructors whose types do not follow the same pattern will become impossible branches. In an impossible branch, you can answer anything but False_rect unit has the advantage to be subterm of anything.

To be concrete: the `tail` function can be written:

```
Definition tail n (v: listn (S n)) :=
  match v in listn (S m) return listn m with
  | niln => False_rect unit
  | consn n' a y => y
  end.
```

and `tail n v` will be subterm of `v`.

Using pattern matching to write proofs

In all the previous examples the elimination predicate does not depend on the object(s) matched. But it may depend and the typical case is when we write a proof by induction or a function that yields an object of a dependent type. An example of a proof written using `match` is given in the description of the tactic *`refine`*.

For example, we can write the function `buildlist` that given a natural number `n` builds a list of length `n` containing zeros as follows:

```
Fixpoint buildlist (n:nat) : listn n :=
  match n return listn n with
  | O => niln
  | S n => consn n 0 (buildlist n)
  end.
```

We can also use multiple patterns. Consider the following definition of the predicate less-equal `Le`:

```
Inductive LE : nat -> nat -> Prop :=
  | LEO : forall n:nat, LE 0 n
  | LES : forall n m:nat, LE n m -> LE (S n) (S m).
```

We can use multiple patterns to write the proof of the lemma `forall (n m:nat), (LE n m) \/ (LE m n)`:

```
Fixpoint dec (n m:nat) {struct n} : LE n m \/ LE m n :=
  match n, m return LE n m \/ LE m n with
  | O, x => or_introl (LE x 0) (LEO x)
  | x, O => or_intror (LE x 0) (LEO x)
  | S n as n', S m as m' =>
      match dec n m with
      | or_introl h => or_introl (LE m' n') (LES n m h)
      | or_intror h => or_intror (LE n' m') (LES m n h)
      end
  end.
```

In the example of `dec`, the first match is dependent while the second is not.

The user can also use match in combination with the tactic *`refine`* to build incomplete proofs beginning with a `match` construction.

Pattern-matching on inductive objects involving local definitions

If local definitions occur in the type of a constructor, then there are two ways to match on this constructor. Either the local definitions are skipped and matching is done only on the true arguments of the constructors, or the bindings for local definitions can also be caught in the matching.

Example

```
Inductive list : nat -> Set :=
| nil : list 0
| cons : forall n:nat, let m := (2 * n) in list m -> list (S (S m)).
```

In the next example, the local definition is not caught.

```
Fixpoint length n (l:list n) {struct l} : nat :=
  match l with
  | nil => 0
  | cons n l0 => S (length (2 * n) l0)
  end.
```

But in this example, it is.

```
Fixpoint length' n (l:list n) {struct l} : nat :=
  match l with
  | nil => 0
  | @cons _ m l0 => S (length' m l0)
  end.
```

Note: For a given matching clause, either none of the local definitions or all of them can be caught.

Note: You can only catch let bindings in mode where you bind all variables and so you have to use `@` syntax.

Note: this feature is incoherent with the fact that parameters cannot be caught and consequently is somehow hidden. For example, there is no mention of it in error messages.

Pattern-matching and coercions

If a mismatch occurs between the expected type of a pattern and its actual type, a coercion made from constructors is sought. If such a coercion can be found, it is automatically inserted around the pattern.

Example

```
Inductive I : Set :=
  | C1 : nat -> I
  | C2 : I -> I.

Coercion C1 : nat >-> I.
```

```
Check (fun x => match x with
                | C2 O => 0
                | _ => 0
                end).
    fun x : I => match x with
                 | C1 _ | _ => 0
                 end
         : I -> nat
```

When does the expansion strategy fail?

The strategy works very like in ML languages when treating patterns of non-dependent types. But there are new cases of failure that are due to the presence of dependencies.

The error messages of the current implementation may be sometimes confusing. When the tactic fails because patterns are somehow incorrect then error messages refer to the initial expression. But the strategy may succeed to build an expression whose sub-expressions are well typed when the whole expression is not. In this situation the message makes reference to the expanded expression. We encourage users, when they have patterns with the same outer constructor in different equations, to name the variable patterns in the same positions with the same name. E.g. to write `(cons n O x) => e1` and `(cons n _ x) => e2` instead of `(cons n O x) => e1` and `(cons n' _ x') => e2`. This helps to maintain certain name correspondence between the generated expression and the original.

Here is a summary of the error messages corresponding to each situation:

Error: The constructor *ident* expects *natural* arguments.
Error: The variable ident is bound several times in pattern term
Error:
Found a constructor of inductive type term while a constructor of term is expected
Patterns are incorrect (because constructors are not applied to the correct number of arguments, because they are not linear or they are wrongly typed).

Error: Non exhaustive pattern matching.
The pattern matching is not exhaustive.

Error:
The elimination predicate term should be of arity *natural* (for non dependent case) or *natu*

The elimination predicate provided to match has not the expected arity.

Error: Unable to infer a match predicate
Error:
Either there is a type incompatibility or the problem involves dependencies.
There is a type mismatch between the different branches. The user should provide an elimination predicate.

2.2.4 Syntax extensions and notation scopes

In this chapter, we introduce advanced commands to modify the way Coq parses and prints objects, i.e. the translations between the concrete and internal representations of terms and commands.

The main commands to provide custom symbolic notations for terms are *Notation* and *Infix*; they will be described in the *next section*. There is also a variant of *Notation* which does not modify the parser; this provides a form of *abbreviation*. It is sometimes expected that the same symbolic notation has different meanings in different contexts; to achieve this form of overloading, Coq offers a notion of *notation scopes*. The main command to provide custom notations for tactics is *Tactic Notation*.

Notations

Basic notations

Command: **Notation** *string* := *one_term* (*syntax_modifier* , +) ? : *scope_name* ?

Defines a *notation*, an alternate syntax for entering or displaying a specific term or term pattern.

This command supports the *local* attribute, which limits its effect to the current module. If the command is inside a section, its effect is limited to the section.

Specifying *scope_name* associates the notation with that scope. Otherwise it is a *lonely notation*, that is, not associated with a scope.

For example, the following definition permits using the infix expression `A /\ B` to represent `(and A B)`:

```
Notation "A /\ B" := (and A B).
```

`"A /\ B"` is a *notation*, which tells how to represent the abbreviated term `(and A B)`.

Notations must be in double quotes, except when the abbreviation has the form of an ordinary applicative expression; see *Abbreviations*. The notation consists of *tokens* separated by spaces. Tokens which are identifiers (such as `A`, `x0'`, etc.) are the *parameters* of the notation. Each of them must occur at least once in the abbreviated term. The other elements of the string (such as `/\`) are the *symbols*.

Identifiers enclosed in single quotes are treated as symbols and thus lose their role of parameters. In the same vein, every symbol of at least 3 characters and starting with a simple quote must be quoted (then it starts with two single quotes). Here is an example.

```
Notation "'IF' c1 'then' c2 'else' c3" := (c1 /\ c2 \/ ~ c1 /\ c3) (at level 200,␣
↪right associativity).
```

A notation binds a syntactic expression to a term. Unless the parser and pretty-printer of Coq already know how to deal with the syntactic expression (such as through *Reserved Notation* or for notations that contain only literals), explicit precedences and associativity rules have to be given.

Note: The right-hand side of a notation is interpreted at the time the notation is given. In particular, disambiguation of constants, *implicit arguments* and other notations are resolved at the time of the declaration of the notation. The right-hand side is currently typed only at use time but this may change in the future.

Precedences and associativity

Mixing different symbolic notations in the same text may cause serious parsing ambiguity. To deal with the ambiguity of notations, Coq uses precedence levels ranging from 0 to 100 (plus one extra level numbered 200) and associativity rules.

Consider for example the new notation

```
Notation "A \/ B" := (or A B).
```

Clearly, an expression such as `forall A:Prop, True /\ A \/ A \/ False` is ambiguous. To tell the Coq parser how to interpret the expression, a priority between the symbols `/\` and `\/` has to be given. Assume for instance that we want conjunction to bind more than disjunction. This is expressed by assigning a precedence level to each notation, knowing that a lower level binds more than a higher level. Hence the level for disjunction must be higher than the level for conjunction.

Since connectives are not tight articulation points of a text, it is reasonable to choose levels not so far from the highest level which is 100, for example 85 for disjunction and 80 for conjunction[20].

Similarly, an associativity is needed to decide whether `True /\ False /\ False` defaults to `True /\ (False /\ False)` (right associativity) or to `(True /\ False) /\ False` (left associativity). We may even consider that the expression is not well-formed and that parentheses are mandatory (this is a "no associativity")[21]. We do not know of a special convention for the associativity of disjunction and conjunction, so let us apply right associativity (which is the choice of Coq).

Precedence levels and associativity rules of notations are specified with a list of parenthesized ***syntax_modifier***s. Here is how the previous examples refine:

```
Notation "A /\ B" := (and A B) (at level 80, right associativity).
Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

By default, a notation is considered nonassociative, but the precedence level is mandatory (except for special cases whose level is canonical). The level is either a number or the phrase `next level` whose meaning is obvious. Some *associativities are predefined* in the `Notations` module.

Complex notations

Notations can be made from arbitrarily complex symbols. One can for instance define prefix notations.

```
Notation "~ x" := (not x) (at level 75, right associativity).
```

One can also define notations for incomplete terms, with the hole expected to be inferred during type checking.

```
Notation "x = y" := (@eq _ x y) (at level 70, no associativity).
```

One can define *closed* notations whose both sides are symbols. In this case, the default precedence level for the inner sub-expression is 200, and the default level for the notation itself is 0.

```
Notation "( x , y )" := (@pair _ _ x y).
```

One can also define notations for binders.

```
Notation "{ x : A | P }" := (sig A (fun x => P)).
```

In the last case though, there is a conflict with the notation for type casts. The notation for type casts, as shown by the command *`Print Grammar`* `constr` is at level 100. To avoid `x : A` being parsed as a type cast, it is necessary to put `x` at a level below 100, typically 99. Hence, a correct definition is the following:

```
Notation "{ x : A | P }" := (sig A (fun x => P)) (x at level 99).
    Setting notation at level 0.
```

More generally, it is required that notations are explicitly factorized on the left. See the next section for more about factorization.

[20] which are the levels effectively chosen in the current implementation of Coq

[21] Coq accepts notations declared as nonassociative but the parser on which Coq is built, namely Camlp5, currently does not implement `no associativity` and replaces it with `left associativity`; hence it is the same for Coq: `no associativity` is in fact `left associativity` for the purposes of parsing

Simple factorization rules

Coq extensible parsing is performed by *Camlp5* which is essentially a LL1 parser: it decides which notation to parse by looking at tokens from left to right. Hence, some care has to be taken not to hide already existing rules by new rules. Some simple left factorization work has to be done. Here is an example.

```
Notation "x < y" := (lt x y) (at level 70).
Fail Notation "x < y < z" := (x < y /\ y < z) (at level 70).
    Toplevel input, characters 0-60:
    > Fail Notation "x < y < z" := (x < y /\ y < z) (at level 70).
    > ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Warning: Notation "_ < _ < _" was already defined with a different format.
    [notation-incompatible-format,parsing]
    The command has indeed failed with message:
    Notation "_ < _ < _" is already defined at level 70 with arguments constr
    at next level, constr at next level, constr at next level
    while it is now required to be at level 70 with arguments constr
    at next level, constr, constr at next level.
```

In order to factorize the left part of the rules, the subexpression referred to by `y` has to be at the same level in both rules. However the default behavior puts `y` at the next level below 70 in the first rule (`no associativity` is the default), and at level 200 in the second rule (`level 200` is the default for inner expressions). To fix this, we need to force the parsing level of `y`, as follows.

```
Notation "x < y" := (lt x y) (at level 70).
Notation "x < y < z" := (x < y /\ y < z) (at level 70, y at next level).
```

For the sake of factorization with Coq predefined rules, simple rules have to be observed for notations starting with a symbol, e.g., rules starting with "`{`" or "`(`" should be put at level 0. The list of Coq predefined notations can be found in the chapter on *The Coq library*.

Use of notations for printing

The command *Notation* has an effect both on the Coq parser and on the Coq printer. For example:

```
Check (and True True).
    True /\ True
         : Prop
```

However, printing, especially pretty-printing, also requires some care. We may want specific indentations, line breaks, alignment if on several lines, etc. For pretty-printing, Coq relies on OCaml formatting library, which provides indentation and automatic line breaks depending on page width by means of *formatting boxes*.

The default printing of notations is rudimentary. For printing a notation, a formatting box is opened in such a way that if the notation and its arguments cannot fit on a single line, a line break is inserted before the symbols of the notation and the arguments on the next lines are aligned with the argument on the first line.

A first, simple control that a user can have on the printing of a notation is the insertion of spaces at some places of the notation. This is performed by adding extra spaces between the symbols and parameters: each extra space (other than the single space needed to separate the components) is interpreted as a space to be inserted by the printer. Here is an example showing how to add spaces next to the curly braces.

```
Notation "{{  x : A | P  }}" := (sig (fun x : A => P)) (at level 0, x at level 99).
```

```
Check (sig (fun x : nat => x=x)).
    {{ x : nat | x = x }}
         : Set
```

The second, more powerful control on printing is by using ***syntax_modifier***s. Here is an example

```
Notation "'If' c1 'then' c2 'else' c3" := (IF_then_else c1 c2 c3)
(at level 200, right associativity, format
"'[v   ' 'If'  c1 '/' '[' 'then'  c2  ']' '/' '[' 'else'  c3 ']' ']'").
    Identifier 'If' now a keyword
    Toplevel input, characters 43-55:
    > Notation "'If' c1 'then' c2 'else' c3" := (IF_then_else c1 c2 c3) (at level 200,
 ↪ right associativity, format "'[v   ' 'If'  c1 '/' '[' 'then'  c2  ']' '/' '[' 'else
 ↪'  c3 ']' ']'").
    >                                             ^^^^^^^^^^^^
    Error: The reference IF_then_else was not found in the current environment.

Check
  (IF_then_else (IF_then_else True False True)
    (IF_then_else True False True)
    (IF_then_else True False True)).
    Toplevel input, characters 9-21:
    > Check   (IF_then_else (IF_then_else True False True)     (IF_then_else True␣
 ↪False True)     (IF_then_else True False True)).
    >         ^^^^^^^^^^^^
    Error: The reference IF_then_else was not found in the current environment.
```

A *format* is an extension of the string denoting the notation with the possible following elements delimited by single quotes:

- tokens of the form `'/ '` are translated into breaking points. If there is a line break, indents the number of spaces appearing after the "/" (no indentation in the example)
- tokens of the form `'//'` force writing on a new line
- well-bracketed pairs of tokens of the form `'[ '` and `']'` are translated into printing boxes; if there is a line break, an extra indentation of the number of spaces after the "[" is applied
- well-bracketed pairs of tokens of the form `'[hv '` and `']'` are translated into horizontal-or-else-vertical printing boxes; if the content of the box does not fit on a single line, then every breaking point forces a new line and an extra indentation of the number of spaces after the "[hv" is applied at the beginning of each new line
- well-bracketed pairs of tokens of the form `'[v '` and `']'` are translated into vertical printing boxes; every breaking point forces a new line, even if the line is large enough to display the whole content of the box, and an extra indentation of the number of spaces after the "[v" is applied at the beginning of each new line (3 spaces in the example)
- extra spaces in other tokens are preserved in the output

Notations disappear when a section is closed. No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the notation.

Note: The default for a notation is to be used both for parsing and printing. It is possible to declare a notation only for parsing by adding the option `only parsing` to the list of ***syntax_modifier***s of *Notation*. Symmetrically, the `only printing` ***syntax_modifier*** can be used to declare that a notation should only be used for printing.

If a notation to be used both for parsing and printing is overridden, both the parsing and printing are invalided, even if the overriding rule is only parsing.

If a given notation string occurs only in `only printing` rules, the parser is not modified at all.

To a given notation string and scope can be attached at most one notation with both parsing and printing or with only parsing. Contrastingly, an arbitrary number of `only printing` notations differing in their right-hand sides but only a unique right-hand side can be attached to a given string and scope. Obviously, expressions printed by means of such extra printing rules will not be reparsed to the same form.

Note: When several notations can be used to print a given term, the notations which capture the largest subterm of the term are used preferentially. Here is an example:

```
Notation "x < y" := (lt x y) (at level 70).
Notation "x < y < z" := (lt x y /\ lt y z) (at level 70, y at next level).

Check (0 < 1 /\ 1 < 2).
```

When several notations match the same subterm, or incomparable subterms of the term to print, the notation declared most recently is selected. Moreover, reimporting a library or module declares the notations of this library or module again. If the notation is in a scope (see *Notation scopes*), either the scope has to be opened or a delimiter has to exist in the scope for the notation to be usable.

The Infix command

The *Infix* command is a shortcut for declaring notations for infix symbols.

Command: **Infix** *string* **:=** *one_term* (*syntax_modifier* , +) ? : *scope_name* ?

This command is equivalent to

Notation "x *string* **y" := (***one_term* **x y)** (*syntax_modifier* , +) ? : *scope_name* ?

where x and y are fresh names and omitting the quotes around *string*. Here is an example:

```
Infix "/\" := and (at level 80, right associativity).
```

Reserving notations

Command: **Reserved Notation** *string* (*syntax_modifier* , +) ?

A given notation may be used in different contexts. Coq expects all uses of the notation to be defined at the same precedence and with the same associativity. To avoid giving the precedence and associativity every time, this command declares a parsing rule (*string*) in advance without giving its interpretation. Here is an example from the initial state of Coq.

```
Reserved Notation "x = y" (at level 70, no associativity).
```

Reserving a notation is also useful for simultaneously defining an inductive type or a recursive constant and a notation for it.

Note: The notations mentioned in the module *Notations* are reserved. Hence their precedence and associativity cannot be changed.

Command: **Reserved Infix** *string* (*syntax_modifier* , +) ?

This command declares an infix parsing rule without giving its interpretation.

When a format is attached to a reserved notation (with the `format` *syntax_modifier*), it is used by default by all subsequent interpretations of the corresponding notation. Individual interpretations can override the format.

Simultaneous definition of terms and notations

Thanks to reserved notations, inductive, co-inductive, record, recursive and corecursive definitions can use customized notations. To do this, insert a *decl_notations* clause after the definition of the (co)inductive type or (co)recursive term (or after the definition of each of them in case of mutual definitions). The exact syntax is given by ***decl_notation*** for inductive, co-inductive, recursive and corecursive definitions and in *Record types* for records. Note that only syntax modifiers that do not require adding or changing a parsing rule are accepted.

decl_notations ::= **where** *decl_notation* **and** *decl_notation* *

decl_notation ::= *string* **:=** *one_term* (*syntax_modifier* , +) ? **:** *scope_name* ?

Here are examples:

```
Reserved Notation "A & B" (at level 80).

Inductive and' (A B : Prop) : Prop := conj' : A -> B -> A & B
where "A & B" := (and' A B).

Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
    | O => m
    | S p => S (p+m)
end
where "n + m" := (plus n m).
```

Displaying information about notations

Flag: **Printing Notations**
This *flag* controls whether to use notations for printing terms wherever possible. Default is on.

Flag: **Printing Raw Literals**
This *flag* controls whether to use string and number notations for printing terms wherever possible (see *String notations*). Default is off.

Flag: **Printing Parentheses**
When this *flag* is on, parentheses are printed even if implied by associativity and precedence. Default is off.

See also:

Printing All to disable other elements in addition to notations.

Command: `Print Grammar` *ident*

Shows the grammar for the nonterminal *ident*, which must be one of the following:

- `constr` - for *term*s
- `pattern` - for *pattern*s
- `tactic` - for currently-defined tactic notations, *tactic*s and tacticals (corresponding to *ltac_expr* in the documentation).
- `vernac` - for *command*s
- `ltac2` - for Ltac2 notations (corresponding to *ltac2_expr*)

This command doesn't display all nonterminals of the grammar. For example, productions shown by `Print Grammar tactic` refer to nonterminals `tactic_then_locality` and `for_each_goal` which are not shown and can't be printed.

Most of the grammar in the documentation was updated in 8.12 to make it accurate and readable. This was done using a new developer tool that extracts the grammar from the source code, edits it and inserts it into the documentation files. While the edited grammar is equivalent to the original, for readability some nonterminals have been renamed and others have been eliminated by substituting the nonterminal definition where the nonterminal was referenced. This command shows the original grammar, so it won't exactly match the documentation.

The Coq parser is based on Camlp5. The documentation for Extensible grammars[14] is the most relevant but it assumes considerable knowledge. Here are the essentials:

Productions can contain the following elements:

- nonterminal names - identifiers in the form `[a-zA-Z0-9_]*`
- `"..."` - a literal string that becomes a keyword and cannot be used as an *ident*. The string doesn't have to be a valid identifier; frequently the string will contain only punctuation characters.
- `IDENT "..."` - a literal string that has the form of an *ident*
- `OPT element` - optionally include `element` (e.g. a nonterminal, IDENT "…" or "…")
- `LIST1 element` - a list of one or more `element`s
- `LIST0 element` - an optional list of `element`s
- `LIST1 element SEP sep` - a list of `element`s separated by `sep`
- `LIST0 element SEP sep` - an optional list of `element`s separated by `sep`
- `[ elements1 | elements2 | ... ]` - alternatives (either `elements1` or `elements2` or …)

Nonterminals can have multiple **levels** to specify precedence and associativity of its productions. This feature of grammars makes it simple to parse input such as `1+2*3` in the usual way as `1+(2*3)`. However, most nonterminals have a single level.

For example, this output from `Print Grammar tactic` shows the first 3 levels for `ltac_expr`, designated as "5", "4" and "3". Level 3 is right-associative, which applies to the productions within it, such as the `try` construct:

```
Entry ltac_expr is
[ "5" RIGHTA
  [ binder_tactic ]
| "4" LEFTA
  [ SELF; ";"; binder_tactic
  | SELF; ";"; SELF
  | SELF; ";"; tactic_then_locality; for_each_goal; "]" ]
| "3" RIGHTA
  [ IDENT "try"; SELF
  :
```

The interpretation of SELF depends on its position in the production and the associativity of the level:

- At the beginning of a production, SELF means the next level. In the fragment shown above, the next level for try is "2". (This is defined by the order of appearance in the grammar or output; the levels could just as well be named "foo" and "bar".)
- In the middle of a production, SELF means the top level ("5" in the fragment)
- At the end of a production, SELF means the next level within LEFTA levels and the current level within RIGHTA levels.

NEXT always means the next level. nonterminal LEVEL "..." is a reference to the specified level for nonterminal.

Associativity[15] explains SELF and NEXT in somewhat more detail.

The output for Print Grammar constr includes *Notation* definitions, which are dynamically added to the grammar at run time. For example, in the definition for term, the production on the second line shown here is defined by a *Reserved Notation* command in Notations.v:

```
| "50" LEFTA
  [ SELF; "||"; NEXT
```

Similarly, Print Grammar tactic includes *Tactic Notation*s, such as *dintuition*.

The file doc/tools/docgram/fullGrammar[16] in the source tree extracts the full grammar for Coq (not including notations and tactic notations defined in *.v files nor some optionally-loaded plugins) in a single file with minor changes to handle nonterminals using multiple levels (described in doc/tools/docgram/README.md[17]). This is complete and much easier to read than the grammar source files. doc/tools/docgram/orderedGrammar[18] has the edited grammar that's used in the documentation.

Developer documentation for parsing is in dev/doc/parsing.md[19].

Locating notations

To know to which notations a given symbol belongs to, use the *Locate* command. You can call it on any (composite) symbol surrounded by double quotes. To locate a particular notation, use a string where the variables of the notation are replaced by "_" and where possible single quotes inserted around identifiers or tokens starting with a single quote are dropped.

```
Locate "exists".
    Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))
      : type_scope (default interpretation)
    Notation "'exists' ! x .. y , p" :=
      (ex (unique (fun x => .. (ex (unique (fun y => p))) ..))) : type_scope
      (default interpretation)

Locate "exists _ .. _ , _".
    Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))
      : type_scope (default interpretation)
```

[14] http://camlp5.github.io/doc/htmlc/grammars.html
[15] http://camlp5.github.io/doc/htmlc/grammars.html#b:Associativity
[16] http://github.com/coq/coq/blob/master/doc/tools/docgram/fullGrammar
[17] http://github.com/coq/coq/blob/master/doc/tools/docgram/README.md
[18] http://github.com/coq/coq/blob/master/doc/tools/docgram/orderedGrammar
[19] http://github.com/coq/coq/blob/master/dev/doc/parsing.md

Inheritance of the properties of arguments of constants bound to a notation

If the right-hand side of a notation is a partially applied constant, the notation inherits the implicit arguments (see *Implicit arguments*) and notation scopes (see *Notation scopes*) of the constant. For instance:

```
Record R := {dom : Type; op : forall {A}, A -> dom}.
Notation "# x" := (@op x) (at level 8).

Check fun x:R => # x 3.
    fun x : R => # x 3
         : forall x : R, dom x
```

As an exception, if the right-hand side is just of the form **@*qualid***, this conventionally stops the inheritance of implicit arguments (but not of notation scopes).

Notations and binders

Notations can include binders. This section lists different ways to deal with binders. For further examples, see also *Notations with recursive patterns involving binders*.

Binders bound in the notation and parsed as identifiers

Here is the basic example of a notation using a binder:

```
Notation "'sigma' x : A , B" := (sigT (fun x : A => B))
  (at level 200, x name, A at level 200, right associativity).
```

The binding variables in the right-hand side that occur as a parameter of the notation (here `x`) dynamically bind all the occurrences in their respective binding scope after instantiation of the parameters of the notation. This means that the term bound to `B` can refer to the variable name bound to `x` as shown in the following application of the notation:

```
Check sigma z : nat, z = 0.
    sigma z : nat, z = 0
         : Set
```

Note the ***syntax_modifier*** **x name** in the declaration of the notation. It tells to parse `x` as a single identifier (or as the unnamed variable _).

Binders bound in the notation and parsed as patterns

In the same way as patterns can be used as binders, as in `fun '(x,y) => x+y` or `fun '(existT _ x _) => x`, notations can be defined so that any ***pattern*** can be used in place of the binder. Here is an example:

```
Notation "'subset' ' p , P " := (sig (fun p => P))
  (at level 200, p pattern, format "'subset'  ' p ,  P").

Check subset '(x,y), x+y=0.
    subset '(x, y), x + y = 0
         : Set
```

The ***syntax_modifier*** **p pattern** in the declaration of the notation tells to parse `p` as a pattern. Note that a single variable is both an identifier and a pattern, so, e.g., the following also works:

```
Check subset 'x, x=0.
    subset 'x, x = 0
         : Set
```

If one wants to prevent such a notation to be used for printing when the pattern is reduced to a single identifier, one has to use instead the ***syntax_modifier*** **p strict pattern**. For parsing, however, a `strict pattern` will continue to include the case of a variable. Here is an example showing the difference:

```
Notation "'subset_bis' ' p , P" := (sig (fun p => P))
  (at level 200, p strict pattern).
Notation "'subset_bis' p , P " := (sig (fun p => P))
  (at level 200, p name).

Check subset_bis 'x, x=0.
    subset_bis x, x = 0
         : Set
```

The default level for a `pattern` is 0. One can use a different level by using `pattern at level` n where the scale is the same as the one for terms (see *Notations*).

Binders bound in the notation and parsed as terms

Sometimes, for the sake of factorization of rules, a binder has to be parsed as a term. This is typically the case for a notation such as the following:

```
Notation "{ x : A | P }" := (sig (fun x : A => P))
    (at level 0, x at level 99 as name).
```

This is so because the grammar also contains rules starting with `{}` and followed by a term, such as the rule for the notation `{ A } + { B }` for the constant `sumbool` (see *Specification*).

Then, in the rule, `x name` is replaced by `x at level 99 as name` meaning that `x` is parsed as a term at level 99 (as done in the notation for `sumbool`), but that this term has actually to be a name, i.e. an identifier or _.

The notation `{ x | P }` is already defined in the standard library with the `as name` ***syntax_modifier***. We cannot redefine it but one can define an alternative notation, say `{ p such that P }`, using instead `as pattern`.

```
Notation "{ p 'such' 'that' P }" := (sig (fun p => P))
  (at level 0, p at level 99 as pattern).
```

Then, the following works:

```
Check {(x,y) such that x+y=0}.
    {(x, y) such that x + y = 0}
         : Set
```

To enforce that the pattern should not be used for printing when it is just a name, one could have said `p at level 99 as strict pattern`.

Note also that in the absence of a `as name`, `as strict pattern` or `as pattern` ***syntax_modifier***s, the default is to consider sub-expressions occurring in binding position and parsed as terms to be `as name`.

Binders bound in the notation and parsed as general binders

It is also possible to rely on Coq's syntax of binders using the `binder` modifier as follows:

```
Notation "'myforall' p , [ P , Q ] " := (forall p, P -> Q)
  (at level 200, p binder).
```

In this case, all of ***ident***, **{*ident*}**, **[*ident*]**, ***ident*:*type***, **{*ident*:*type*}**, **[*ident*:*type*]**, **'*pattern*** can be used in place of the corresponding notation variable. In particular, the binder can declare implicit arguments:

```
Check fun (f : myforall {a}, [a=0, Prop]) => f eq_refl.
    fun f : myforall a, [a = 0, Prop] => f 0 eq_refl
         : (myforall a, [a = 0, Prop]) -> Prop

Check myforall '((x,y):nat*nat), [ x = y, True ].
    myforall '(x, y), [x = y, True]
         : Prop
```

By using instead `closed binder`, the same list of binders is allowed except that ***ident*:*type*** requires parentheses around.

Binders not bound in the notation

We can also have binders in the right-hand side of a notation which are not themselves bound in the notation. In this case, the binders are considered up to renaming of the internal binder. E.g., for the notation

```
Notation "'exists_different' n" := (exists p:nat, p<>n) (at level 200).
```

the next command fails because p does not bind in the instance of n.

```
Fail Check (exists_different p).
    The command has indeed failed with message:
    The reference p was not found in the current environment.

Notation "[> a , .. , b <]" :=
  (cons a .. (cons b nil) .., cons b .. (cons a nil) ..).
```

Notations with expressions used both as binder and term

It is possible to use parameters of the notation both in term and binding position. Here is an example:

```
Definition force n (P:nat -> Prop) := forall n', n' >= n -> P n'.
Notation "□_ n P" := (force n (fun n => P))
  (at level 0, n name, P at level 9, format "□_ n  P").

Check exists p, □_p (p >= 1).
    exists p : nat, □_p (p >= 1)
         : Prop
```

More generally, the parameter can be a pattern, as in the following variant:

```
Definition force2 q (P:nat*nat -> Prop) :=
  (forall n', n' >= fst q -> forall p', p' >= snd q -> P q).
```

(continues on next page)

(continued from previous page)

```
Notation "□_ p P" := (force2 p (fun p => P))
  (at level 0, p pattern at level 0, P at level 9, format "□_ p  P").

Check exists x y, □_(x,y) (x >= 1 /\ y >= 2).
    exists x y : nat, □_(x, y) (x >= 1 /\ y >= 2)
         : Prop
```

This support is experimental. For instance, the notation is used for printing only if the occurrence of the parameter in term position comes in the right-hand side before the occurrence in binding position.

Notations with recursive patterns

A mechanism is provided for declaring elementary notations with recursive patterns. The basic example is:

```
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
    Setting notation at level 0.
```

On the right-hand side, an extra construction of the form `.. t ..` can be used. Notice that `..` is part of the Coq syntax and it must not be confused with the three-dots notation "..." used in this manual to denote a sequence of arbitrary size.

On the left-hand side, the part "`x s .. s y`" of the notation parses any number of times (but at least once) a sequence of expressions separated by the sequence of tokens `s` (in the example, `s` is just "`;`").

The right-hand side must contain a subterm of the form either `ψ(x, .. ψ(y,t) ..)` or `ψ(y, .. ψ(x,t) ..)` where $\varphi([\,]_E, [\,]_I)$, called the *iterator* of the recursive notation is an arbitrary expression with distinguished placeholders and where t is called the *terminating expression* of the recursive notation. In the example, we choose the names x and y but in practice they can of course be chosen arbitrarily. Note that the placeholder $[\,]_I$ has to occur only once but $[\,]_E$ can occur several times.

Parsing the notation produces a list of expressions which are used to fill the first placeholder of the iterating pattern which itself is repeatedly nested as many times as the length of the list, the second placeholder being the nesting point. In the innermost occurrence of the nested iterating pattern, the second placeholder is finally filled with the terminating expression.

In the example above, the iterator $\varphi([\,]_E, [\,]_I)$ is $cons[\,]_E\,[\,]_I$ and the terminating expression is `nil`.

Here is another example with the pattern associating on the left:

```
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) (at level 0).
```

Here is an example with more involved recursive patterns:

```
Notation "[| t * ( x , y , .. , z ) ; ( a , b , .. , c )  * u |]" :=
  (pair (pair .. (pair (pair t x) (pair t y)) .. (pair t z))
        (pair .. (pair (pair a u) (pair b u)) .. (pair c u)))
  (t at level 39).
```

To give a flavor of the extent and limits of the mechanism, here is an example showing a notation for a chain of equalities. It relies on an artificial expansion of the intended denotation so as to expose a `ψ(x, .. ψ(y,t) ..)` structure, with the drawback that if ever the beta-redexes are contracted, the notations stops to be used for printing. Support for notations defined in this way should be considered experimental.

```
Notation "x ⊡ y ⊡ .. ⊡ z ⊡ t" :=
  ((fun b A a => a <= b /\ A b) y .. ((fun b A a => a <= b /\ A b) z (fun b => b <=␣
↪t)) .. x)
  (at level 70, y at next level, z at next level, t at next level).
```

Note finally that notations with recursive patterns can be reserved like standard notations, they can also be declared within *notation scopes*.

Notations with recursive patterns involving binders

Recursive notations can also be used with binders. The basic example is:

```
Notation "'exists' x .. y , p" :=
  (ex (fun x => .. (ex (fun y => p)) ..))
  (at level 200, x binder, y binder, right associativity).
```

The principle is the same as in *Notations with recursive patterns* except that in the iterator $\varphi([\,]_E, [\,]_I)$, the placeholder $[\,]_E$ can also occur in position of the binding variable of a `fun` or a `forall`.

To specify that the part "`x .. y`" of the notation parses a sequence of binders, `x` and `y` must be marked as `binder` in the list of ***syntax_modifier***s of the notation. The binders of the parsed sequence are used to fill the occurrences of the first placeholder of the iterating pattern which is repeatedly nested as many times as the number of binders generated. If ever the generalization operator `'` (see *Implicit generalization*) is used in the binding list, the added binders are taken into account too.

There are two flavors of binder parsing. If `x` and `y` are marked as binder, then a sequence such as `a b c : T` will be accepted and interpreted as the sequence of binders `(a:T) (b:T) (c:T)`. For instance, in the notation above, the syntax `exists a b : nat, a = b` is valid.

The variables `x` and `y` can also be marked as closed binder in which case only well-bracketed binders of the form `(a b c:T)` or `{a b c:T}` etc. are accepted.

With closed binders, the recursive sequence in the left-hand side can be of the more general form `x s .. s y` where `s` is an arbitrary sequence of tokens. With open binders though, `s` has to be empty. Here is an example of recursive notation with closed binders:

```
Notation "'mylet' f x .. y :=  t 'in' u":=
  (let f := fun x => .. (fun y => t) .. in u)
  (at level 200, x closed binder, y closed binder, right associativity).
```

A recursive pattern for binders can be used in position of a recursive pattern for terms. Here is an example:

```
Notation "'FUNAPP' x .. y , f" :=
  (fun x => .. (fun y => (.. (f x) ..) y ) ..)
  (at level 200, x binder, y binder, right associativity).
```

If an occurrence of the $[\,]_E$ is not in position of a binding variable but of a term, it is the name used in the binding which is used. Here is an example:

```
Notation "'exists_non_null' x .. y  , P" :=
  (ex (fun x => x <> 0 /\ .. (ex (fun y => y <> 0 /\ P)) ..))
  (at level 200, x binder).
```

Predefined entries

By default, sub-expressions are parsed as terms and the corresponding grammar entry is called `constr`. However, one may sometimes want to restrict the syntax of terms in a notation. For instance, the following notation will accept to parse only global reference in position of `x`:

```
Notation "'apply' f a1 .. an" := (.. (f a1) .. an)
  (at level 10, f global, a1, an at level 9).
```

In addition to `global`, one can restrict the syntax of a sub-expression by using the entry names `ident`, `name` or `pattern` already seen in *Binders not bound in the notation*, even when the corresponding expression is not used as a binder in the right-hand side. E.g.:

```
Notation "'apply_id' f a1 .. an" := (.. (f a1) .. an)
  (at level 10, f ident, a1, an at level 9).
```

Note: As of version 8.13, the entry `ident` is a deprecated alias for `name`. In the future, it is planned to strictly parse an identifier (excluding _).

Custom entries

Command: `Declare Custom Entry` *`ident`*

Defines new grammar entries, called *custom entries*, that can later be referred to using the entry name **`custom`** ***`ident`***.

This command supports the *`local`* attribute, which limits the entry to the current module.

Non-local custom entries survive module closing and are declared when a file is Required.

Example

For instance, we may want to define an ad hoc parser for arithmetical operations and proceed as follows:

```
Inductive Expr :=
| One : Expr
| Mul : Expr -> Expr -> Expr
| Add : Expr -> Expr -> Expr.
    Expr is defined
    Expr_rect is defined
    Expr_ind is defined
    Expr_rec is defined
    Expr_sind is defined


Declare Custom Entry expr.
Notation "[ e ]" := e (e custom expr at level 2).
    Setting notation at level 0.

Notation "1" := One (in custom expr at level 0).
Notation "x y" := (Mul x y) (in custom expr at level 1, left associativity).
Notation "x + y" := (Add x y) (in custom expr at level 2, left associativity).
Notation "( x )" := x (in custom expr, x at level 2).
```

(continues on next page)

(continued from previous page)

```
    Setting notation at level 0.

Notation "{ x }" := x (in custom expr, x constr).
    Setting notation at level 0.

Notation "x" := x (in custom expr at level 0, x ident).

Axiom f : nat -> Expr.
    f is declared

Check fun x y z => [1 + y z + {f x}].
    fun (x : nat) (y z : Expr) => [1 + y z + {apply f x}]
         : nat -> Expr -> Expr -> Expr

Unset Printing Notations.
Check fun x y z => [1 + y z + {f x}].
    fun (x : nat) (y z : Expr) => Add (Add One (Mul y z)) (f x)
         : forall (_ : nat) (_ : Expr) (_ : Expr), Expr

Set Printing Notations.
Check fun e => match e with
| [1 + 1] => [1]
| [x y + z] => [x + y z]
| y => [y + e]
end.
    fun e : Expr =>
    match e with
    | [1 + 1] => [1]
    | [x y + z] => [x + y z]
    | _ => [e + e]
    end
         : Expr -> Expr
```

Custom entries have levels, like the main grammar of terms and grammar of patterns have. The lower level is 0 and this is the level used by default to put rules delimited with tokens on both ends. The level is left to be inferred by Coq when using **in custom** *ident*. The level is otherwise given explicitly by using the syntax **in custom** *ident* **at level** *natural*, where *natural* refers to the level.

Levels are cumulative: a notation at level `n` of which the left end is a term shall use rules at level less than `n` to parse this subterm. More precisely, it shall use rules at level strictly less than `n` if the rule is declared with `right associativity` and rules at level less or equal than `n` if the rule is declared with `left associativity`. Similarly, a notation at level `n` of which the right end is a term shall use by default rules at level strictly less than `n` to parse this subterm if the rule is declared left associative and rules at level less or equal than `n` if the rule is declared right associative. This is what happens for instance in the rule

```
Notation "x + y" := (Add x y) (in custom expr at level 2, left associativity).
```

where `x` is any expression parsed in entry `expr` at level less or equal than `2` (including, recursively, the given rule) and `y` is any expression parsed in entry `expr` at level strictly less than `2`.

Rules associated with an entry can refer different sub-entries. The grammar entry name `constr` can be used to refer to the main grammar of term as in the rule

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

which indicates that the subterm `x` should be parsed using the main grammar. If not indicated, the level is computed as

for notations in `constr`, e.g. using 200 as default level for inner sub-expressions. The level can otherwise be indicated explicitly by using `constr at level n` for some `n`, or `constr at next level`.

Conversely, custom entries can be used to parse sub-expressions of the main grammar, or from another custom entry as is the case in

```
Notation "[ e ]" := e (e custom expr at level 2).
```

to indicate that `e` has to be parsed at level `2` of the grammar associated with the custom entry `expr`. The level can be omitted, as in

```
Notation "[ e ]" := e (e custom expr).
```

in which case Coq infer it. If the sub-expression is at a border of the notation (as e.g. `x` and `y` in `x + y`), the level is determined by the associativity. If the sub-expression is not at the border of the notation (as e.g. `e` in `"[ e ]"`), the level is inferred to be the highest level used for the entry. In particular, this level depends on the highest level existing in the entry at the time of use of the notation.

In the absence of an explicit entry for parsing or printing a sub-expression of a notation in a custom entry, the default is to consider that this sub-expression is parsed or printed in the same custom entry where the notation is defined. In particular, if `x at level n` is used for a sub-expression of a notation defined in custom entry `foo`, it shall be understood the same as `x custom foo at level n`.

In general, rules are required to be *productive* on the right-hand side, i.e. that they are bound to an expression which is not reduced to a single variable. If the rule is not productive on the right-hand side, as it is the case above for

```
Notation "( x )" := x (in custom expr at level 0, x at level 2).
```

and

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

it is used as a *grammar coercion* which means that it is used to parse or print an expression which is not available in the current grammar at the current level of parsing or printing for this grammar but which is available in another grammar or in another level of the current grammar. For instance,

```
Notation "( x )" := x (in custom expr at level 0, x at level 2).
```

tells that parentheses can be inserted to parse or print an expression declared at level `2` of `expr` whenever this expression is expected to be used as a subterm at level 0 or 1. This allows for instance to parse and print `Add x y` as a subterm of `Mul (Add x y) z` using the syntax `(x + y) z`. Similarly,

```
Notation "{ x }" := x (in custom expr at level 0, x constr).
```

gives a way to let any arbitrary expression which is not handled by the custom entry `expr` be parsed or printed by the main grammar of term up to the insertion of a pair of curly brackets.

Another special situation is when parsing global references or identifiers. To indicate that a custom entry should parse identifiers, use the following form:

```
Notation "x" := x (in custom expr at level 0, x ident).
```

Similarly, to indicate that a custom entry should parse global references (i.e. qualified or non qualified identifiers), use the following form:

```
Notation "x" := x (in custom expr at level 0, x global).
```

Command: `Print Custom Grammar` *ident*

This displays the state of the grammar for terms associated with the custom entry *ident*.

Syntax

Here are the syntax elements used by the various notation commands.

syntax_modifier ::= **at level** *natural*
| **in custom** *ident* [**at level** *natural*]?
| *ident*+, [**at** *level* | **in scope** *ident*]
| *ident* **at** *level* [*binder_interp*]?
| *ident explicit_subentry*
| *ident binder_interp*
| **left associativity**
| **right associativity**
| **no associativity**
| **only parsing**
| **only printing**
| **format** *string* [*string*]?

explicit_subentry ::= **ident**
| **name**
| **global**
| **bigint**
| **strict pattern** [**at level** *natural*]?
| **binder**
| **closed binder**
| **constr** [**at** *level*]? [*binder_interp*]?
| **custom** *ident* [**at** *level*]? [*binder_interp*]?
| **pattern** [**at level** *natural*]?

binder_interp ::= **as ident**
| **as name**
| **as pattern**
| **as strict pattern**

level ::= **level** *natural*
| **next level**

Note: No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the notation.

Note: Some examples of Notation may be found in the files composing the initial state of Coq (see directory `$COQLIB/theories/Init`).

Note: The notation `"{ x }"` has a special status in the main grammars of terms and patterns so that complex notations of the form `"x + { y }"` or `"x * { y }"` can be nested with correct precedences. Especially, every notation involving a pattern of the form `"{ x }"` is parsed as a notation where the pattern `"{ x }"` has been simply replaced by `"x"` and the curly brackets are parsed separately. E.g. `"y + { z }"` is not parsed as a term of the given form but as a term of the form `"y + z"` where `z` has been parsed using the rule parsing `"{ x }"`. Especially, level and precedences for a rule including patterns of the form `"{ x }"` are relative not to the textual notation but to the notation

where the curly brackets have been removed (e.g. the level and the associativity given to some notation, say "`{ y } & { z }`" in fact applies to the underlying "`{ x }`"-free rule which is "`y & z`").

Note: Notations such as "`( p | q )`" (or starting with "`( x | `", more generally) are deprecated as they conflict with the syntax for nested disjunctive patterns (see *Extended pattern matching*), and are not honored in pattern expressions.

Warning:
Use of *string* Notation is deprecated as it is inconsistent with pattern syntax.

This warning is disabled by default to avoid spurious diagnostics due to legacy notation in the Coq standard library. It can be turned on with the `-w disj-pattern-notation` flag.

Note: As of version 8.13, the entry `ident` is a deprecated alias for `name`. In the future, it is planned to strictly parse an identifier (to the exclusion of _). If the intent was to use `ident` as an identifier (excluding _), just silence the warning with **`Set Warnings "-deprecated-ident-entry"`** and it should automatically get its intended meaning in version 8.15.

Similarly, `as ident` is a deprecated alias for `as name`, which will only accept an identifier in the future. If the intent was to use `as ident` as an identifier (excluding _), just silence the warning with **`Set Warnings "-deprecated-as-ident-kind"`**.

However, this deprecation does not apply to custom entries, where it already denotes an identifier, as expected.

Notation scopes

A *notation scope* is a set of notations for terms with their interpretations. Notation scopes provide a weak, purely syntactic form of notation overloading: a symbol may refer to different definitions depending on which notation scopes are currently open. For instance, the infix symbol + can be used to refer to distinct definitions of the addition operator, such as for natural numbers, integers or reals. Notation scopes can include an interpretation for numbers and strings with the *Number Notation* and *String Notation* commands.

scope	::=	*scope_name*
	|	*scope_key*
scope_name	::=	*ident*
scope_key	::=	*ident*

Each notation scope has a single *scope_name*, which by convention ends with the suffix "_scope", as in "nat_scope". One or more *scope_key*s (delimiting keys) may be associated with a notation scope with the *Delimit Scope* command. Most commands use *scope_name*; *scope_key*s are used within *term*s.

Command: `Declare Scope` *scope_name*

Declares a new notation scope. Note that the initial state of Coq declares the following notation scopes: `core_scope`, `type_scope`, `function_scope`, `nat_scope`, `bool_scope`, `list_scope`, `int_scope`, `uint_scope`.

Use commands such as *Notation* to add notations to the scope.

Global interpretation rules for notations

At any time, the interpretation of a notation for a term is done within a *stack* of notation scopes and lonely notations. If a notation is defined in multiple scopes, Coq uses the interpretation from the most recently opened notation scope or declared lonely notation.

Note that "stack" is a misleading name. Each scope or lonely notation can only appear in the stack once. New items are pushed onto the top of the stack, except that adding a item that's already in the stack moves it to the top of the stack instead. Scopes are removed by name (e.g. by `Close Scope`) wherever they are in the stack, rather than through "pop" operations.

Use the `Print Visibility` command to display the current notation scope stack.

Command: **`Open Scope`** ***`scope`***

Adds a scope to the notation scope stack. If the scope is already present, the command moves it to the top of the stack.

If the command appears in a section: By default, the scope is only added within the section. Specifying `global` marks the scope for export as part of the current module. Specifying `local` behaves like the default.

If the command does not appear in a section: By default, the scope marks the scope for export as part of the current module. Specifying `local` prevents exporting the scope. Specifying `global` behaves like the default.

Command: **`Close Scope`** ***`scope`***

Removes a scope from the notation scope stack.

If the command appears in a section: By default, the scope is only removed within the section. Specifying `global` marks the scope removal for export as part of the current module. Specifying `local` behaves like the default.

If the command does not appear in a section: By default, the scope marks the scope removal for export as part of the current module. Specifying `local` prevents exporting the removal. Specifying `global` behaves like the default.

Local interpretation rules for notations

In addition to the global rules of interpretation of notations, some ways to change the interpretation of subterms are available.

Opening a notation scope locally

term_scope `::=` *term0* **%** *scope_key*

The notation scope stack can be locally extended within a `term` with the syntax **`(`*`term`*`)%`*`scope_key`*** (or simply ***`term0`*`%`*`scope_key`*** for atomic terms).

In this case, ***`term`*** is interpreted in the scope stack extended with the scope bound to ***`scope_key`***.

Command: **`Delimit Scope`** ***`scope_name`*** **`with`** ***`scope_key`***

Binds the delimiting key `scope_key` to a scope.

Command: **`Undelimit Scope`** ***`scope_name`***

Removes the delimiting keys associated with a scope.

The arguments of an *abbreviation* can be interpreted in a scope stack locally extended with a given scope by using the modifier ***`ident`*** **`+`** **`,`** **`in scope`** ***`scope_name`***.s

Binding types or coercion classes to a notation scope

Command: **Bind Scope** *scope_name* **with** *class*

Binds the notation scope *scope_name* to the type or coercion class *class*. When bound, arguments of that type for any function will be interpreted in that scope by default. This default can be overridden for individual functions with the *Arguments* command. The association may be convenient when a notation scope is naturally associated with a *type* (e.g. nat and the natural numbers).

Whether the argument of a function has some type type is determined statically. For instance, if f is a polymorphic function of type forall X:Type, X -> X and type t is bound to a scope scope, then a of type t in f t a is not recognized as an argument to be interpreted in scope scope.

```
Parameter U : Set.
Declare Scope U_scope.
Bind Scope U_scope with U.
Parameter Uplus : U -> U -> U.
Parameter P : forall T:Set, T -> U -> Prop.
Parameter f : forall T:Set, T -> U.
Infix "+" := Uplus : U_scope.
Unset Printing Notations.
Open Scope nat_scope.

Check (fun x y1 y2 z t => P _ (x + t) ((f _ (y1 + y2) + z))).
    fun (x y1 y2 : nat) (z : U) (t : nat) =>
    P nat (Nat.add x t) (Uplus (f nat (Nat.add y1 y2)) z)
         : forall (_ : nat) (_ : nat) (_ : nat) (_ : U) (_ : nat), Prop
```

Note: When active, a bound scope has effect on all defined functions (even if they are defined after the *Bind Scope* directive), except if argument scopes were assigned explicitly using the *Arguments* command.

Note: The scopes type_scope and function_scope also have a local effect on interpretation. See the next section.

The type_scope notation scope

The scope type_scope has a special status. It is a primitive interpretation scope which is temporarily activated each time a subterm of an expression is expected to be a type. It is delimited by the key type, and bound to the coercion class Sortclass. It is also used in certain situations where an expression is statically known to be a type, including the conclusion and the type of hypotheses within an Ltac goal match (see *Pattern matching on goals and hypotheses: match goal*), the statement of a theorem, the type of a definition, the type of a binder, the domain and codomain of implication, the codomain of products, and more generally any type argument of a declared or defined constant.

The `function_scope` notation scope

The scope `function_scope` also has a special status. It is temporarily activated each time the argument of a global reference is recognized to be a `Funclass` instance, i.e., of type `forall x:A, B` or `A -> B`.

Notation scopes used in the standard library of Coq

We give an overview of the scopes used in the standard library of Coq. For a complete list of notations in each scope, use the commands *`Print Scopes`* or *`Print Scope`*.

`type_scope` This scope includes infix * for product types and infix + for sum types. It is delimited by the key `type`, and bound to the coercion class `Sortclass`, as described above.

`function_scope` This scope is delimited by the key `function`, and bound to the coercion class `Funclass`, as described above.

`nat_scope` This scope includes the standard arithmetical operators and relations on type nat. Positive integer numbers in this scope are mapped to their canonical representent built from `O` and `S`. The scope is delimited by the key `nat`, and bound to the type `nat` (see above).

`N_scope` This scope includes the standard arithmetical operators and relations on type `N` (binary natural numbers). It is delimited by the key `N` and comes with an interpretation for numbers as closed terms of type `N`.

`Z_scope` This scope includes the standard arithmetical operators and relations on type `Z` (binary integer numbers). It is delimited by the key `Z` and comes with an interpretation for numbers as closed terms of type `Z`.

`positive_scope` This scope includes the standard arithmetical operators and relations on type `positive` (binary strictly positive numbers). It is delimited by key `positive` and comes with an interpretation for numbers as closed terms of type `positive`.

`Q_scope` This scope includes the standard arithmetical operators and relations on type `Q` (rational numbers defined as fractions of an integer and a strictly positive integer modulo the equality of the numerator- denominator cross-product) and comes with an interpretation for numbers as closed terms of type `Q`.

`Qc_scope` This scope includes the standard arithmetical operators and relations on the type `Qc` of rational numbers defined as the type of irreducible fractions of an integer and a strictly positive integer.

`R_scope` This scope includes the standard arithmetical operators and relations on type `R` (axiomatic real numbers). It is delimited by the key `R` and comes with an interpretation for numbers using the `IZR` morphism from binary integer numbers to `R` and `Z.pow_pos` for potential exponent parts.

`bool_scope` This scope includes notations for the boolean operators. It is delimited by the key `bool`, and bound to the type `bool` (see above).

`list_scope` This scope includes notations for the list operators. It is delimited by the key `list`, and bound to the type `list` (see above).

`core_scope` This scope includes the notation for pairs. It is delimited by the key `core`.

`string_scope` This scope includes notation for strings as elements of the type string. Special characters and escaping follow Coq conventions on strings (see *Lexical conventions*). Especially, there is no convention to visualize non printable characters of a string. The file `String.v` shows an example that contains quotes, a newline and a beep (i.e. the ASCII character of code 7).

`char_scope` This scope includes interpretation for all strings of the form `"c"` where `c` is an ASCII character, or of the form `"nnn"` where nnn is a three-digit number (possibly with leading 0s), or of the form `""""`. Their respective denotations are the ASCII code of `c`, the decimal ASCII code `nnn`, or the ascii code of the character `"` (i.e. the ASCII code 34), all of them being represented in the type `ascii`.

Displaying information about scopes

Command: **Print Visibility** *scope_name*?

Displays the current notation scope stack. The top of the stack is displayed last. Notations in scopes whose interpretation is hidden by the same notation in a more recently opened scope are not displayed. Hence each notation is displayed only once.

If *scope_name* is specified, displays the current notation scope stack as if the scope *scope_name* is pushed on top of the stack. This is useful to see how a subterm occurring locally in the scope is interpreted.

Command: **Print Scopes**

Displays, for each existing notation scope, all accessible notations (whether or not currently in the notation scope stack), the most-recently defined delimiting key and the class the notation scope is bound to. The display also includes lonely notations.

Use the *Print Visibility* command to display the current notation scope stack.

Command: **Print Scope** *scope_name*

Displays all notations defined in the notation scope *scope_name*. It also displays the delimiting key and the class to which the scope is bound, if any.

Abbreviations

Command: **Notation** *ident* *ident*$_{parm}$* **:=** *one_term* (*syntax_modifier*+,)?

Defines an abbreviation *ident* with the parameters *ident*$_{parm}$.

This command supports the *local* attribute, which limits the notation to the current module.

An *abbreviation* is a name, possibly applied to arguments, that denotes a (presumably) more complex expression. Here are examples:

```
Notation Nlist := (list nat).

Check 1 :: 2 :: 3 :: nil.
    1 :: 2 :: 3 :: nil
         : Nlist

Notation reflexive R := (forall x, R x x).

Check forall A:Prop, A <-> A.
    reflexive iff
         : Prop

Check reflexive iff.
    reflexive iff
         : Prop

Notation Plus1 B := (Nat.add B 1).

Compute (Plus1 3).
    = 4
         : nat
```

An abbreviation expects no precedence nor associativity, since it is parsed as an usual application. Abbreviations are used as much as possible by the Coq printers unless the modifier `(only parsing)` is given.

An abbreviation is bound to an absolute name as an ordinary definition is and it also can be referred to by a qualified name.

Abbreviations are syntactic in the sense that they are bound to expressions which are not typed at the time of the definition of the abbreviation but at the time they are used. Especially, abbreviations can be bound to terms with holes (i.e. with "_"). For example:

```
Definition explicit_id (A:Set) (a:A) := a.

Notation id := (explicit_id _).

Check (id 0).
    id 0
         : nat
```

Abbreviations disappear when a section is closed. No typing of the denoted expression is performed at definition time. Type checking is done only at the time of use of the abbreviation.

Like for notations, if the right-hand side of an abbreviation is a partially applied constant, the abbreviation inherits the implicit arguments and notation scopes of the constant. As an exception, if the right-hand side is just of the form **@*qualid***, this conventionally stops the inheritance of implicit arguments.

Like for notations, it is possible to bind binders in abbreviations. Here is an example:

```
Definition force2 q (P:nat*nat -> Prop) :=
  (forall n', n' >= fst q -> forall p', p' >= snd q -> P q).

Notation F p P := (force2 p (fun p => P)).
Check exists x y, F (x,y) (x >= 1 /\ y >= 2).
```

Numbers and strings

primitive_notations	::=	*number*
	|	*string*

Numbers and strings have no predefined semantics in the calculus. They are merely notations that can be bound to objects through the notation mechanism. Initially, numbers are bound to Peano's representation of natural numbers (see *Datatypes*).

Note: Negative integers are not at the same level as ***natural***, for this would make precedence unnatural.

Number notations

Command:

Number Notation *qualid*$_{type}$ *qualid*$_{parse}$ *qualid*$_{print}$ (*number_modifier* , +) ? : *scope_name*

number_modifier ::= **warning after** *bignat*
| **abstract after** *bignat*
| *number_string_via*

number_string_via ::= **via** *qualid* **mapping** [*qualid* => *qualid* | [*qualid*] => *qualid* , +]

This command allows the user to customize the way number literals are parsed and printed.

qualid$_{type}$ the name of an inductive type, while *qualid*$_{parse}$ and *qualid*$_{print}$ should be the names of the parsing and printing functions, respectively. The parsing function *qualid*$_{parse}$ should have one of the following types:

- **Number.int ->** *qualid*$_{type}$
- **Number.int -> option** *qualid*$_{type}$
- **Number.uint ->** *qualid*$_{type}$
- **Number.uint -> option** *qualid*$_{type}$
- **Z ->** *qualid*$_{type}$
- **Z -> option** *qualid*$_{type}$
- **PrimInt63.int ->** *qualid*$_{type}$
- **PrimInt63.int -> option** *qualid*$_{type}$
- **Number.number ->** *qualid*$_{type}$
- **Number.number -> option** *qualid*$_{type}$

And the printing function *qualid*$_{print}$ should have one of the following types:

- *qualid*$_{type}$ **-> Number.int**
- *qualid*$_{type}$ **-> option Number.int**
- *qualid*$_{type}$ **-> Number.uint**
- *qualid*$_{type}$ **-> option Number.uint**
- *qualid*$_{type}$ **-> Z**
- *qualid*$_{type}$ **-> option Z**
- *qualid*$_{type}$ **-> PrimInt63.int**
- *qualid*$_{type}$ **-> option PrimInt63.int**
- *qualid*$_{type}$ **-> Number.number**
- *qualid*$_{type}$ **-> option Number.number**

When parsing, the application of the parsing function *qualid*$_{parse}$ to the number will be fully reduced, and universes of the resulting term will be refreshed.

Note that only fully-reduced ground terms (terms containing only function application, constructors, inductive type families, sorts, primitive integers, primitive floats, primitive arrays and type constants for primitive types) will be considered for printing.

Note: For example, *qualid*$_{type}$ can be `PrimInt63.int`, in which case *qualid*$_{print}$ takes `PrimInt63.int_wrapper` as input instead of `PrimInt63.int`. See below for an *example*.

---

via *qualid*$_{ind}$ **mapping [** *qualid*$_{constant}$ **=>** *qualid*$_{constructor}$ **+ ,** **]** When using this option, *qualid*$_{type}$ no longer needs to be an inductive type and is instead mapped to the inductive type *qualid*$_{ind}$ according to the provided list of pairs, whose first component *qualid*$_{constant}$ is a constant of type *qualid*$_{type}$ (or a function of type `_ ->` * *qualid*$_{type}$) and the second a constructor of type *qualid*$_{ind}$. The type *qualid*$_{type}$ is then replaced by *qualid*$_{ind}$ in the above parser and printer types.

When *qualid*$_{constant}$ is surrounded by square brackets, all the implicit arguments of *qualid*$_{constant}$ (whether maximally inserted or not) are ignored when translating to *qualid*$_{constructor}$ (i.e., before applying *qualid*$_{print}$) and replaced with implicit argument holes _ when translating from *qualid*$_{constructor}$ to *qualid*$_{constant}$ (after *qualid*$_{parse}$). See below for an *example*.

Note: The implicit status of the arguments is considered only at notation declaration time, any further modification of this status has no impact on the previously declared notations.

Note: In case of multiple implicit options (for instance `Arguments eq_refl {A}%type_scope {x}, [_] _`), an argument is considered implicit when it is implicit in any of the options.

Note: To use a `sort` as the target type *qualid*$_{type}$, use an *abbreviation* as in the *example below*.

---

warning after *bignat* displays a warning message about a possible stack overflow when calling *qualid*$_{parse}$ to parse a literal larger than *bignat*.

Warning:
Stack overflow or segmentation fault happens when working with large numbers in

When a `Number Notation` is registered in the current scope with **(warning after** *bignat***)**, this warning is emitted when parsing a number greater than or equal to `bignat`.

abstract after *bignat* returns **(***qualid*$_{parse}$ **m)** when parsing a literal **m** that's greater than *bignat* rather than reducing it to a normal form. Here `m` will be a `Number.int`, `Number.uint`, `Z` or `Number.number`, depending on the type of the parsing function *qualid*$_{parse}$. This allows for a more compact representation of literals in types such as `nat`, and limits parse failures due to stack overflow. Note that a warning will be emitted when an integer larger than `bignat` is parsed. Note that **(abstract after** *bignat***)** has no effect when *qualid*$_{parse}$ lands in an `option` type.

Warning: To avoid stack overflow, large numbers in *type* are interpreted as applications of *qualid*$_{parse}$.

When a *Number Notation* is registered in the current scope with **(abstract after *bignat*)**, this warning is emitted when parsing a number greater than or equal to *bignat*. Typically, this indicates that the fully computed representation of numbers can be so large that non-tail-recursive OCaml functions run out of stack space when trying to walk them.

Warning: The 'abstract after' directive has no effect when the parsing function (*qualid*$_{pa}$

As noted above, the **(abstract after *natural*)** directive has no effect when *qualid*$_{parse}$ lands in an `option` type.

Error: 'via' and 'abstract' cannot be used together.
With the **abstract after** option, the parser function *qualid*$_{parse}$ does not reduce large numbers to a normal form, which prevents doing the translation given in the **mapping** list.

Error: Cannot interpret this number as a value of type *type*
The number notation registered for *type* does not support the given number. This error is given when the interpretation function returns `None`, or if the interpretation is registered only for integers or non-negative integers, and the given number has a fractional or exponent part or is negative.

Error: overflow in int63 literal *bigint*
The constant's absolute value is too big to fit into a 63-bit integer **PrimInt63.int**.

Error: *qualid*$_{parse}$ should go from Number.int to *type* or (option *type*). Instead of Number.int, the types Number.uint or Z or PrimInt63. pos_neg_int63 or Number. number could be used (you may need to require BinNums or Number or PrimInt63 first).

The parsing function given to the *Number Notation* command is not of the right type.

Error: *qualid*$_{print}$ should go from *type* to Number.int or (option Number.int). Instead of Number.int, the types Number.uint or Z or PrimInt63. pos_neg_int63 or Number. number could be used (you may need to require BinNums or Number or PrimInt63 first).

The printing function given to the *Number Notation* command is not of the right type.

Error: Unexpected term *term* while parsing a number notation.
Parsing functions must always return ground terms, made up of function application, constructors, inductive type families, sorts and primitive integers. Parsing functions may not return terms containing axioms, bare (co)fixpoints, lambdas, etc.

Error: Unexpected non-option term *term* while parsing a number notation.
Parsing functions expected to return an `option` must always return a concrete `Some` or `None` when applied to a concrete number expressed as a (hexa)decimal. They may not return opaque constants.

Error: Multiple 'via' options.
At most one `via` option can be given.

Error: Multiple 'warning after' or 'abstract after' options.
At most one `warning after` or `abstract after` option can be given.

String notations

Command:

String Notation *qualid*$_{type}$ *qualid*$_{parse}$ *qualid*$_{print}$ (*number_string_via*)? **:** *scope_name*

Allows the user to customize how strings are parsed and printed.

qualid$_{type}$ the name of an inductive type, while *qualid*$_{parse}$ and *qualid*$_{print}$ should be the names of the parsing and printing functions, respectively. The parsing function *qualid*$_{parse}$ should have one of the following types:

- **Byte.byte ->** *qualid*$_{type}$
- **Byte.byte -> option** *qualid*$_{type}$
- **list Byte.byte ->** *qualid*$_{type}$
- **list Byte.byte -> option** *qualid*$_{type}$

The printing function *qualid*$_{print}$ should have one of the following types:

- *qualid*$_{type}$ **-> Byte.byte**
- *qualid*$_{type}$ **-> option Byte.byte**
- *qualid*$_{type}$ **-> list Byte.byte**
- *qualid*$_{type}$ **-> option (list Byte.byte)**

When parsing, the application of the parsing function *qualid*$_{parse}$ to the string will be fully reduced, and universes of the resulting term will be refreshed.

Note that only fully-reduced ground terms (terms containing only function application, constructors, inductive type families, sorts, primitive integers, primitive floats, primitive arrays and type constants for primitive types) will be considered for printing.

via *qualid*$_{ind}$ **mapping [** *qualid*$_{constant}$ **=>** *qualid*$_{constructor}$ **,**+ **]**

works as for *number notations above*.

Error: Cannot interpret this string as a value of type *type*

The string notation registered for `type` does not support the given string. This error is given when the interpretation function returns `None`.

Error: *qualid*$_{parse}$ **should go from Byte.byte or (list Byte.byte) to** *type* **or (option** *type***).**

The parsing function given to the `String Notation` command is not of the right type.

Error: *qualid*$_{print}$ **should go from** *type* **to Byte.byte or (option Byte.byte) or (list Byte.byte) or (option (list Byte.byte)).**

The printing function given to the `String Notation` command is not of the right type.

Error: Unexpected term *term* **while parsing a string notation.**

Parsing functions must always return ground terms, made up of function application, constructors, inductive type families, sorts and primitive integers. Parsing functions may not return terms containing axioms, bare (co)fixpoints, lambdas, etc.

Error: Unexpected non-option term *term* **while parsing a string notation.**

Parsing functions expected to return an `option` must always return a concrete `Some` or `None` when applied to a concrete string expressed as a decimal. They may not return opaque constants.

Note: Number or string notations for parameterized inductive types can be added by declaring an *abbreviation* for the inductive which instantiates all parameters. See *example below*.

The following errors apply to both string and number notations:

Error: ***type*** **is not an inductive type.**
String and number notations can only be declared for inductive types. Declare string or numeral notations for non-inductive types using ***number_string_via***.

Error:
qualid **was already mapped to** ***qualid*** **and cannot be remapped to** ***qualid***
Duplicates are not allowed in the **mapping** list.

Error: Missing mapping for constructor ***qualid***
A mapping should be provided for ***qualid*** in the **mapping** list.

Warning: ***type*** **was already mapped to** ***type*****,**
mapping it also to ***type*** **might yield ill typed terms when using the notation.**

Two pairs in the **mapping** list associate types that might be incompatible.

Warning: Type of ***qualid*** **seems incompatible with the type of** ***qualid*****.**
Expected type is: ***type*** **instead of** ***type*****.**
This might yield ill typed terms when using the notation.
A mapping given in the **mapping** list associates a constant with a seemingly incompatible constructor.

Error:
Cannot interpret in ***scope_name*** **because** ***qualid*** **could not be found in the current enviro**

The inductive type used to register the string or number notation is no longer available in the environment. Most likely, this is because the notation was declared inside a functor for an inductive type inside the functor. This use case is not currently supported.

Alternatively, you might be trying to use a primitive token notation from a plugin which forgot to specify which module you must `Require` for access to that notation.

Error:
Syntax error: [prim:reference] expected after 'Notation' (in [vernac:command]).

The type passed to *`String Notation`* or *`Number Notation`* must be a single qualified identifier.

Error:
Syntax error: [prim:reference] expected after [prim:reference] (in [vernac:command]).

Both functions passed to *`String Notation`* or *`Number Notation`* must be single qualified identifiers.

Error: ***qualid*** **is bound to a notation that does not denote a reference.**
Identifiers passed to *`String Notation`* or *`Number Notation`* must be global references, or notations which evaluate to single qualified identifiers.

Example: Number Notation for radix 3

The following example parses and prints natural numbers whose digits are `0`, `1` or `2` as terms of the following inductive type encoding radix 3 numbers.

```
Inductive radix3 : Set :=
  | x0 : radix3
  | x3 : radix3 -> radix3
  | x3p1 : radix3 -> radix3
  | x3p2 : radix3 -> radix3.
```

We first define a parsing function

```
Definition of_uint_dec (u : Decimal.uint) : option radix3 :=
  let fix f u := match u with
    | Decimal.Nil => Some x0
    | Decimal.D0 u => match f u with Some u => Some (x3 u) | None => None end
    | Decimal.D1 u => match f u with Some u => Some (x3p1 u) | None => None end
    | Decimal.D2 u => match f u with Some u => Some (x3p2 u) | None => None end
    | _ => None end in
  f (Decimal.rev u).
Definition of_uint (u : Number.uint) : option radix3 :=
  match u with Number.UIntDecimal u => of_uint_dec u | Number.UIntHexadecimal _ =>␣
 ↪None end.
```

and a printing function

```
Definition to_uint_dec (x : radix3) : Decimal.uint :=
  let fix f x := match x with
    | x0 => Decimal.Nil
    | x3 x => Decimal.D0 (f x)
    | x3p1 x => Decimal.D1 (f x)
    | x3p2 x => Decimal.D2 (f x) end in
  Decimal.rev (f x).
Definition to_uint (x : radix3) : Number.uint := Number.UIntDecimal (to_uint_dec x).
```

before declaring the notation

```
Declare Scope radix3_scope.
Open Scope radix3_scope.
Number Notation radix3 of_uint to_uint : radix3_scope.
```

We can check the printer

```
Check x3p2 (x3p1 x0).
    12
         : radix3
```

and the parser

```
Set Printing All.
Check 120.
    x3 (x3p2 (x3p1 x0))
         : radix3
```

Digits other than `0`, `1` and `2` are rejected.

```
Check 3.
    Toplevel input, characters 6-7:
    > Check 3.
    >       ^
    Error: Cannot interpret this number as a value of type radix3
```

Example: Number Notation for primitive integers

This shows the use of the primitive integers **PrimInt63.int** as *qualid*$_{type}$. It is the way parsing and printing of primitive integers are actually implemented in `PrimInt63.v`.

```
Require Import PrimInt63.
Definition parser (x : pos_neg_int63) : option int :=
  match x with Pos p => Some p | Neg _ => None end.
Definition printer (x : int_wrapper) : pos_neg_int63 := Pos (int_wrap x).
Number Notation int parser printer : uint63_scope.
```

Example: Number Notation for a non inductive type

The following example encodes the terms in the form `sum unit ( ... (sum unit unit) ... )` as the number of units in the term. For instance `sum unit (sum unit unit)` is encoded as `3` while `unit` is `1` and `0` stands for `Empty_set`. The inductive `I` will be used as *qualid*$_{ind}$.

```
Inductive I := Iempty : I | Iunit : I | Isum : I -> I -> I.
```

We then define *qualid*$_{parse}$ and *qualid*$_{print}$

```
Definition of_uint (x : Number.uint) : I :=
  let fix f n := match n with
    | O => Iempty | S O => Iunit
    | S n => Isum Iunit (f n) end in
  f (Nat.of_num_uint x).

Definition to_uint (x : I) : Number.uint :=
  let fix f i := match i with
    | Iempty => O | Iunit => 1
    | Isum i1 i2 => f i1 + f i2 end in
  Nat.to_num_uint (f x).

Inductive sum (A : Set) (B : Set) : Set := pair : A -> B -> sum A B.
```

the number notation itself

```
Notation nSet := Set (only parsing).
Number Notation nSet of_uint to_uint (via I
  mapping [Empty_set => Iempty, unit => Iunit, sum => Isum]) : type_scope.
```

and check the printer

```
Local Open Scope type_scope.
Check sum unit (sum unit unit).
    3
         : Set
```

and the parser

```
Set Printing All.
Check 3.
    sum unit (sum unit unit)
         : Set
```

Example: Number Notation with implicit arguments

The following example parses and prints natural numbers between `0` and `n-1` as terms of type `Fin.t n`.

```
Require Import Vector.
Print Fin.t.
    Inductive t : nat -> Set :=
        F1 : forall n : nat, Fin.t (S n)
      | FS : forall n : nat, Fin.t n -> Fin.t (S n).

    Arguments Fin.t _%nat_scope
    Arguments Fin.F1 {n}%nat_scope
    Arguments Fin.FS {n}%nat_scope _
```

Note the implicit arguments of `Fin.F1` and `Fin.FS`, which won't appear in the corresponding inductive type.

```
Inductive I := I1 : I | IS : I -> I.

Definition of_uint (x : Number.uint) : I :=
  let fix f n := match n with O => I1 | S n => IS (f n) end in
  f (Nat.of_num_uint x).

Definition to_uint (x : I) : Number.uint :=
  let fix f i := match i with I1 => O | IS n => S (f n) end in
  Nat.to_num_uint (f x).

Declare Scope fin_scope.
Delimit Scope fin_scope with fin.
Local Open Scope fin_scope.
Number Notation Fin.t of_uint to_uint (via I
  mapping [[Fin.F1] => I1, [Fin.FS] => IS]) : fin_scope.
```

Now `2` is parsed as `Fin.FS (Fin.FS Fin.F1)`, that is `@Fin.FS _ (@Fin.FS _ (@Fin.F1 _))`.

```
Check 2.
    2
         : Fin.t (S (S (S ?n)))
    where
    ?n : [ |- nat]
```

which can be of type `Fin.t 3` (numbers `0`, `1` and `2`)

```
Check 2 : Fin.t 3.
    2 : Fin.t 3
         : Fin.t 3
```

but cannot be of type `Fin.t 2` (only `0` and `1`)

```
Check 2 : Fin.t 2.
    Toplevel input, characters 6-7:
    > Check 2 : Fin.t 2.
    >       ^
    Error:
    The term "2" has type "Fin.t (S (S (S ?n)))"
    while it is expected to have type "Fin.t 2".
```

Example: String Notation with a parameterized inductive type

The parameter `Byte.byte` for the parameterized inductive type `list` is given through an *abbreviation*.

```
Notation string := (list Byte.byte) (only parsing).
Definition id_string := @id string.

String Notation string id_string id_string : list_scope.

Check "abc"%list.
    "abc"%list
         : list Byte.byte
```

Tactic Notations

Tactic notations allow customizing the syntax of tactics.

Command:

Tactic Notation (**at level** *natural*)? *ltac_production_item*+ **:=** *ltac_expr*

ltac_production_item ::= *string*

| *ident* (*ident* (, *string*)?)?

Defines a *tactic notation*, which extends the parsing and pretty-printing of tactics.

This command supports the *local* attribute, which limits the notation to the current module.

natural The parsing precedence to assign to the notation. This information is particularly relevant for notations for tacticals. Levels can be in the range 0 .. 5 (default is 5).

ltac_production_item+ The notation syntax. Notations for simple tactics should begin with a *string*. Note that `Tactic Notation foo := idtac` is not valid; it should be `Tactic Notation "foo" := idtac`.

string represents a literal value in the notation

ident is the name of a grammar nonterminal listed in the table below. In a few cases, to maintain backward compatibility, the name differs from the nonterminal name used elsewhere in the documentation.

($ident_{parm}$ (, $string_s$)?) $ident_{parm}$ is the parameter name associated with ***ident***. The $string_s$ is the separator string to use when ***ident*** specifies a list with separators (i.e. ***ident*** ends with `_list_sep`).

ltac_expr The tactic expression to substitute for the notation. $ident_{parm}$ tokens appearing in ***ltac_expr*** are substituted with the associated nonterminal value.

For example, the following command defines a notation with a single parameter `x`.

```
Tactic Notation "destruct_with_eqn" constr(x) := destruct x eqn:?.
```

For a complex example, examine the 16 `Tactic Notation "setoid_replace"`s defined in `$COQLIB/theories/Classes/SetoidTactics.v`, which are designed to accept any subset of 4 optional parameters.

The nonterminals that can specified in the tactic notation are:

Specified *ident*	Parsed as	Interpreted as	as in tactic
`ident`	*ident*	a user-given name	*intro*
`simple_intropattern`	*simple_intropattern*	an introduction pattern	*assert* `as`
`hyp`	*ident*	a hypothesis defined in context	*clear*
`reference`	*qualid*	a qualified identifier	name of an L_{tac}-defined tactic
`smart_global`	*reference*	a global reference of term	*unfold*, *with_strategy*
`constr`	*one_term*	a term	*exact*
`uconstr`	*one_term*	an untyped term	*refine*
`integer`	*integer*	an integer	
`int_or_var`	*int_or_var*	an integer	*do*
`strategy_level`	*strategy_level*	a strategy level	
`strategy_level_or_var`	*strategy_level_or_var*	a strategy level	*with_strategy*
`tactic`	*ltac_expr*	a tactic	
`tactic`*n* (*n* in 0..5)	*ltac_expr*n	a tactic at level *n*	
entry`_list`	**entry**`*`	a list of how *entry* is interpreted	
`ne_`*entry*`_list`	**entry**`+`	a list of how *entry* is interpreted	
entry`_list_sep`	**entry**`*` **s**	a list of how *entry* is interpreted	
`ne_`*entry*`_list_sep`	**entry**`+` **s**	a list of how *entry* is interpreted	

Note: In order to be bound in tactic definitions, each syntactic entry for argument type must include the case of a simple L_{tac} identifier as part of what it parses. This is naturally the case for `ident`, `simple_intropattern`, `reference`, `constr`, ... but not for `integer` nor for `strategy_level`. This is the reason for introducing special entries `int_or_var` and `strategy_level_or_var` which evaluate to integers or strategy levels only, respectively, but which syntactically includes identifiers in order to be usable in tactic definitions.

Note: The *entry*`_list*` and `ne_`*entry*`_list*` entries can be used in primitive tactics or in other notations at places where a list of the underlying entry can be used: entry is either `constr`, `hyp`, `integer`, `reference`, `strategy_level`, `strategy_level_or_var`, or `int_or_var`.

2.2.5 Setting properties of a function's arguments

Command:

Arguments *reference* *arg_specs** , *implicits_alt** : *args_modifier*+, ?

argument_spec ::= !? *name* % *scope_key* ?
arg_specs ::= *argument_spec*
| /
| &
| (*argument_spec*+) % *scope_key* ?
| [*argument_spec*+] % *scope_key* ?
| { *argument_spec*+ } % *scope_key* ?
implicits_alt ::= *name*
| [*name*+]
| { *name*+ }
args_modifier ::= **simpl nomatch**
| **simpl never**
| **default implicits**
| **clear implicits**
| **clear scopes**
| **clear bidirectionality hint**
| **rename**
| **assert**
| **extra scopes**
| **clear scopes and implicits**
| **clear implicits and scopes**

Specifies properties of the arguments of a function after the function has already been defined. It gives fine-grained control over the elaboration process (i.e. the translation of Gallina language extensions into the core language used by the kernel). The command's effects include:

- Making arguments implicit. Afterward, implicit arguments must be omitted in any expression that applies `reference`.
- Declaring that some arguments of a given function should be interpreted in a given scope.
- Affecting when the `simpl` and `cbn` tactics unfold the function. See *Effects of Arguments on unfolding*.
- Providing bidirectionality hints. See *Bidirectionality hints*.

This command supports the `local` and `global` attributes. Default behavior is to limit the effect to the current section but also to extend their effect outside the current module or library file. Applying `local` limits the effect of the command to the current module if it's not in a section. Applying `global` within a section extends the effect outside the current sections and current module in which the command appears.

/ the function will be unfolded only if it's applied to at least the arguments appearing before the `/`. See *Effects of Arguments on unfolding*.

Error: `The / modifier may only occur once.`

& tells the type checking algorithm to first type check the arguments before the `&` and then to propagate information from that typing context to type check the remaining arguments. See *Bidirectionality hints*.

Error: The & modifier may only occur once.

(...) **% *scope*** **?** **(*name*$_1$ *name*$_2$ ...)%*scope*** is shorthand for ***name*$_1$%*scope* *name*$_2$%*scope* ...**

[...] **% *scope*** **?** declares the enclosed names as implicit, non-maximally inserted. **[*name*$_1$ *name*$_2$ ...]%*scope*** is equivalent to **[*name*$_1$]%*scope* [*name*$_2$]%*scope* ...**

{ ... } **% *scope*** **?** declares the enclosed names as implicit, maximally inserted. **{*name*$_1$ *name*$_2$ ... }%*scope*** is equivalent to **{*name*$_1$}%*scope* {*name*$_2$}%*scope* ...**

! the function will be unfolded only if all the arguments marked with ! evaluate to constructors. See *Effects of Arguments on unfolding*.

name **% *scope*** **?** a *formal parameter* of the function ***reference*** (i.e. the parameter name used in the function definition). Unless `rename` is specified, the list of ***name***s must be a prefix of the formal parameters, including all implicit arguments. _ can be used to skip over a formal parameter.

The construct ***name*** **% *scope*** **?** declares ***name*** as non-implicit if `clear implicits` is specified or at least one other name is declared implicit in the same list of ***name***s. *scope* can be either a scope name or its delimiting key. See *Binding arguments to a scope*.

Error: To rename arguments the 'rename' flag must be specified.

Error: Flag 'rename' expected to rename *name* into *name*.

clear implicits makes all implicit arguments into explicit arguments

Error:
The 'clear implicits' flag must be omitted if implicit annotations are given.

default implicits automatically determine the implicit arguments of the object. See *Automatic declaration of implicit arguments*.

Error:
The 'default implicits' flag is incompatible with implicit annotations.

rename rename implicit arguments for the object. See the example *here*.

assert assert that the object has the expected number of arguments with the expected names. See the example here: *Renaming implicit arguments*.

Warning:
This command is just asserting the names of arguments of *qualid*.
If this is what you want, add ': assert' to silence the warning.
If you want to clear implicit arguments,
add ': clear implicits'. If you want to clear notation scopes,
add ': clear scopes'

clear scopes clears argument scopes of ***reference***

extra scopes defines extra argument scopes, to be used in case of coercion to `Funclass` (see *Implicit Coercions*) or with a computed type.

simpl nomatch prevents performing a simplification step for ***reference*** that would expose a match construct in the head position. See *Effects of Arguments on unfolding*.

simpl never prevents performing a simplification step for ***reference***. See *Effects of Arguments on unfolding*.

clear bidirectionality hint removes the bidirectionality hint, the `&`

implicits_alt use to specify alternative implicit argument declarations for functions that can only be applied to a fixed number of arguments (excluding, for instance, functions whose type is polymorphic). For parsing, the longest list of implicit arguments matching the function application is used to select which implicit arguments are inserted. For printing, the alternative with the most implicit arguments is used; the implict arguments will be omitted if `Printing Implicit` is not set. See the example *here*.

Use `About` to view the current implicit arguments setting for a `reference`.

Or use the `Print Implicit` command to see the implicit arguments of an object (see *Displaying implicit arguments*).

Manual declaration of implicit arguments

Example

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined


Check (cons nat 3 (nil nat)).
    cons nat 3 (nil nat)
         : list nat


Arguments cons [A] _ _.

Arguments nil {A}.

Check (cons 3 nil).
    cons 3 nil
         : list nat


Fixpoint map (A B : Type) (f : A -> B) (l : list A) : list B :=
  match l with nil => nil | cons a t => cons (f a) (map A B f t) end.
    map is defined
    map is recursively defined (guarded on 4th argument)


Fixpoint length (A : Type) (l : list A) : nat :=
  match l with nil => 0 | cons _ m => S (length A m) end.
    length is defined
    length is recursively defined (guarded on 2nd argument)


Arguments map [A B] f l.
```

(continues on next page)

(continued from previous page)

```
Arguments length {A} l. (* A has to be maximally inserted *)

Check (fun l:list (list nat) => map length l).
    fun l : list (list nat) => map length l
         : list (list nat) -> list nat
```

Example: Multiple alternatives with `implicits_alt`

```
Arguments map [A B] f l, [A] B f l, A B f l.

Check (fun l => map length l = map (list nat) nat length l).
    fun l : list (list nat) => map length l = map length l
         : list (list nat) -> Prop
```

Automatic declaration of implicit arguments

The "**`default implicits`**" *`args_modifier`* clause tells Coq to automatically determine the implicit arguments of the object.

Auto-detection is governed by flags specifying whether strict, contextual, or reversible-pattern implicit arguments must be considered or not (see *Controlling strict implicit arguments*, *Controlling contextual implicit arguments*, *Controlling reversible-pattern implicit arguments* and also *Controlling the insertion of implicit arguments not followed by explicit arguments*).

Example: Default implicits

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined


Arguments cons : default implicits.

Print Implicit cons.
    cons : forall [A : Set], A -> list A -> list A

    Argument A is implicit


Arguments nil : default implicits.

Print Implicit nil.
    nil : forall A : Set, list A


Set Contextual Implicit.
```

(continues on next page)

(continued from previous page)

```
Arguments nil : default implicits.

Print Implicit nil.
    nil : forall {A : Set}, list A

    Argument A is implicit and maximally inserted
```

The computation of implicit arguments takes account of the unfolding of *constants*. For instance, the variable p below has type (Transitivity R) which is reducible to forall x,y:U, R x y -> forall z:U, R y z -> R x z. As the variables x, y and z appear strictly in the *body* of the type, they are implicit.

```
Parameter X : Type.
    X is declared


Definition Relation := X -> X -> Prop.
    Relation is defined


Definition Transitivity (R:Relation) := forall x y:X, R x y -> forall z:X, R y z -> R␣
↪x z.
    Transitivity is defined


Parameters (R : Relation) (p : Transitivity R).
    R is declared
    p is declared


Arguments p : default implicits.

Print p.
    *** [ p : Transitivity R ]

    Expanded type for implicit arguments
    p : forall [x y : X], R x y -> forall z : X, R y z -> R x z

    Arguments p [x y] _ [z] _


Print Implicit p.
    p : forall [x y : X], R x y -> forall z : X, R y z -> R x z

    Arguments x, y, z are implicit


Parameters (a b c : X) (r1 : R a b) (r2 : R b c).
    a is declared
    b is declared
    c is declared
    r1 is declared
    r2 is declared
```

(continues on next page)

(continued from previous page)

```
Check (p r1 r2).
    p r1 r2
         : R a c
```

Renaming implicit arguments

Example: (continued) Renaming implicit arguments

```
Arguments p [s t] _ [u] _: rename.

Check (p r1 (u:=c)).
    p r1 (u:=c)
         : R b c -> R a c


Check (p (s:=a) (t:=b) r1 (u:=c) r2).
    p r1 r2
         : R a c


Fail Arguments p [s t] _ [w] _ : assert.
    The command has indeed failed with message:
    Flag "rename" expected to rename u into w.
```

Binding arguments to a scope

The following command declares that the first two arguments of `plus_fct` are in the *scope* delimited by the key `F` (`Rfun_scope`) and the third argument is in the scope delimited by the key `R` (`R_scope`).

```
Arguments plus_fct (f1 f2)%F x%R.
```

When interpreting a term, if some of the arguments of *reference* are built from a notation, then this notation is interpreted in the scope stack extended by the scope bound (if any) to this argument. The effect of the scope is limited to the argument itself. It does not propagate to subterms but the subterms that, after interpretation of the notation, turn to be themselves arguments of a reference are interpreted accordingly to the argument scopes bound to this reference.

Note: In notations, the subterms matching the identifiers of the notations are interpreted in the scope in which the identifiers occurred at the time of the declaration of the notation. Here is an example:

```
Parameter g : bool -> bool.
    g is declared

Declare Scope mybool_scope.

Notation "@@" := true (only parsing) : bool_scope.
    Setting notation at level 0.

Notation "@@" := false (only parsing): mybool_scope.
```

(continues on next page)

(continued from previous page)

```
Bind Scope bool_scope with bool.
Notation "# x #" := (g x) (at level 40).
Check # @@ #.
    # true #
         : bool

Arguments g _%mybool_scope.
Check # @@ #.
    # true #
         : bool

Delimit Scope mybool_scope with mybool.
Check # @@%mybool #.
    # false #
         : bool
```

Effects of `Arguments` on unfolding

- `simpl never` indicates that a *constant* should never be unfolded by *cbn*, *simpl* or *hnf*:

  Example

  ```
  Arguments minus n m : simpl never.
  ```

  After that command an expression like `(minus (S x) y)` is left untouched by the tactics *cbn* and *simpl*.

- A *constant* can be marked to be unfolded only if it's applied to at least the arguments appearing before the `/` in a *Arguments* command.

  Example

  ```
  Definition fcomp A B C f (g : A -> B) (x : A) : C := f (g x).
      fcomp is defined

  Arguments fcomp {A B C} f g x /.
  Notation "f \o g" := (fcomp f g) (at level 50).
  ```

  After that command the expression `(f \o g)` is left untouched by *simpl* while `((f \o g) t)` is reduced to `(f (g t))`. The same mechanism can be used to make a *constant* volatile, i.e. always unfolded.

  Example

  ```
  Definition volatile := fun x : nat => x.
      volatile is defined

  Arguments volatile / x.
  ```

- A *constant* can be marked to be unfolded only if an entire set of arguments evaluates to a constructor. The `!` symbol can be used to mark such arguments.

Example

```
Arguments minus !n !m.
```

After that command, the expression `(minus (S x) y)` is left untouched by *simpl*, while `(minus (S x) (S y))` is reduced to `(minus x y)`.

- `simpl nomatch` indicates that a *constant* should not be unfolded if it would expose a `match` construct in the head position. This affects the *cbn*, *simpl* and *hnf* tactics.

Example

```
Arguments minus n m : simpl nomatch.
```

In this case, `(minus (S (S x)) (S y))` is simplified to `(minus (S x) y)` even if an extra simplification is possible.

In detail: the tactic *simpl* first applies βι-reduction. Then, it expands transparent *constants* and tries to reduce further using βι-reduction. But, when no ι rule is applied after unfolding then δ-reductions are not applied. For instance trying to use *simpl* on `(plus n O) = n` changes nothing.

Bidirectionality hints

When type-checking an application, Coq normally does not use information from the context to infer the types of the arguments. It only checks after the fact that the type inferred for the application is coherent with the expected type. Bidirectionality hints make it possible to specify that after type-checking the first arguments of an application, typing information should be propagated from the context to help inferring the types of the remaining arguments.

An *Arguments* command containing ***arg_specs*$_1$ & *arg_specs*$_2$** provides bidirectionality hints. It tells the type-checking algorithm, when type checking applications of ***qualid***, to first type check the arguments in ***arg_specs*$_1$** and then propagate information from the typing context to type check the remaining arguments (in ***arg_specs*$_2$**).

Example: Bidirectionality hints

In a context where a coercion was declared from `bool` to `nat`:

```
Definition b2n (b : bool) := if b then 1 else 0.
Coercion b2n : bool >-> nat.
```

Coq cannot automatically coerce existential statements over `bool` to statements over `nat`, because the need for inserting a coercion is known only from the expected type of a subterm:

```
Fail Check (ex_intro _ true _ : exists n : nat, n > 0).
    The command has indeed failed with message:
    The term "ex_intro ?P true ?y" has type "exists y, ?P y"
    while it is expected to have type "exists n : nat, n > 0"
    (cannot unify "bool" and "nat").
```

However, a suitable bidirectionality hint makes the example work:

```
Arguments ex_intro _ _ & _ _.
Check (ex_intro _ true _ : exists n : nat, n > 0).
    ex_intro (fun n : nat => n > 0) true ?g : exists n : nat, n > 0
```

(continues on next page)

(continued from previous page)

```
     : exists n : nat, n > 0
where
?g : [ |- (fun n : nat => n > 0) true]
```

Coq will attempt to produce a term which uses the arguments you provided, but in some cases involving Program mode the arguments after the bidirectionality starts may be replaced by convertible but syntactically different terms.

2.2.6 Implicit Coercions

Author Amokrane Saïbi

General Presentation

This section describes the inheritance mechanism of Coq. In Coq with inheritance, we are not interested in adding any expressive power to our theory, but only convenience. Given a term, possibly not typable, we are interested in the problem of determining if it can be well typed modulo insertion of appropriate coercions. We allow to write:

- `f a` where `f:(forall x:A,B)` and `a:A'` when `A'` can be seen in some sense as a subtype of `A`.
- `x:A` when `A` is not a type, but can be seen in a certain sense as a type: set, group, category etc.
- `f a` when `f` is not a function, but can be seen in a certain sense as a function: bijection, functor, any structure morphism etc.

Classes

A class with n parameters is any defined name with a type **forall** (*ident*$_1$: *type*$_1$)..(*ident*$_n$:*type*$_n$), *sort*. Thus a class with parameters is considered as a single class and not as a family of classes. An object of a class is any term of type *class term*$_1$ .. *term*$_n$. In addition to these user-defined classes, we have two built-in classes:

- `Sortclass`, the class of sorts; its objects are the terms whose type is a sort (e.g. `Prop` or `Type`).
- `Funclass`, the class of functions; its objects are all the terms with a functional type, i.e. of form `forall x:A,B`.

Formally, the syntax of classes is defined as:

class	::=	**Funclass**
	|	**Sortclass**
	|	*reference*

Coercions

A name `f` can be declared as a coercion between a source user-defined class `C` with n parameters and a target class `D` if one of these conditions holds:

- `D` is a user-defined class, then the type of `f` must have the form `forall (x`$_1$`:A`$_1$`)..(x`$_n$`:A`$_n$`)(y:C x`$_1$`..x`$_n$`), D u`$_1$`..u`$_m$ where m is the number of parameters of `D`.
- `D` is `Funclass`, then the type of `f` must have the form `forall (x`$_1$`:A`$_1$`)..(x`$_n$`:A`$_n$`)(y:C x`$_1$`..x`$_n$`)(x:A), B`.

- `D` is `Sortclass`, then the type of `f` must have the form `forall (x₁:A₁)..(x?:A?)(y:C x₁..x?)`, `s` with `s` a sort.

We then write `f : C >-> D`. The restriction on the type of coercions is called *the uniform inheritance condition*.

Note: The built-in class `Sortclass` can be used as a source class, but the built-in class `Funclass` cannot.

To coerce an object `t:C t₁..t?` of `C` towards `D`, we have to apply the coercion `f` to it; the obtained term `f t₁..t? t` is then an object of `D`.

Identity Coercions

Identity coercions are special cases of coercions used to go around the uniform inheritance condition. Let `C` and `D` be two classes with respectively `n` and `m` parameters and `f:forall (x₁:T₁)..(x?:T?)(y:C u₁..u?), D v₁..v?` a function which does not verify the uniform inheritance condition. To declare `f` as coercion, one has first to declare a subclass `C'` of `C`:

```
C' := fun (x₁:T₁)..(x?:T?) => C u₁..u?
```

We then define an *identity coercion* between `C'` and `C`:

```
Id_C'_C := fun (x₁:T₁)..(x?:T?)(y:C' x₁..x?) => (y:C u₁..u?)
```

We can now declare `f` as coercion from `C'` to `D`, since we can "cast" its type as `forall (x₁:T₁)..(x?:T?)(y:C' x₁..x?),D v₁..v?`.

The identity coercions have a special status: to coerce an object `t:C' t₁..t?` of `C'` towards `C`, we do not have to insert explicitly `Id_C'_C` since `Id_C'_C t₁..t? t` is convertible with `t`. However we "rewrite" the type of `t` to become an object of `C`; in this case, it becomes `C u?'..u?'` where each `u?'` is the result of the substitution in `u?` of the variables `x?` by `t?`.

Inheritance Graph

Coercions form an inheritance graph with classes as nodes. We call *coercion path* an ordered list of coercions between two nodes of the graph. A class `C` is said to be a subclass of `D` if there is a coercion path in the graph from `C` to `D`; we also say that `C` inherits from `D`. Our mechanism supports multiple inheritance since a class may inherit from several classes, contrary to simple inheritance where a class inherits from at most one class. However there must be at most one path between two classes. If this is not the case, only the *oldest* one is valid and the others are ignored. So the order of declaration of coercions is important.

We extend notations for coercions to coercion paths. For instance `[f₁;..;f?] : C >-> D` is the coercion path composed by the coercions `f₁..f?`. The application of a coercion path to a term consists of the successive application of its coercions.

Declaring Coercions

Command: **`Coercion`** ***`reference`*** **`:`** ***`class`*** **`>->`** ***`class`***

Command: **`Coercion`** ***`ident`*** ***`univ_decl`***`?` ***`def_body`***

name Coercion; _

The first form declares the construction denoted by *`reference`* as a coercion between the two given classes. The second form defines *`ident`* just like *`Definition`* ***`ident`*** ***`univ_decl`***`?` ***`def_body`*** and then declares

ident as a coercion between it source and its target. Both forms support the *local* attribute, which makes the coercion local to the current section.

Error: *qualid* not declared.

Error: *qualid* is already a coercion.

Error: Funclass cannot be a source class.

Error: *qualid* is not a function.

Error: Cannot find the source class of *qualid*.

Error: Cannot recognize *class* as a source class of *qualid*.

Warning: *qualid* does not respect the uniform inheritance condition.

Error: Found target class ... instead of ...

Warning: New coercion path ... is ambiguous with existing ...
When the coercion *qualid* is added to the inheritance graph, new coercion paths which have the same classes as existing ones are ignored. The *Coercion* command tries to check the convertibility of new ones and existing ones. The paths for which this check fails are displayed by a warning in the form `[f₁;..;f?] : C >-> D`.

The convertibility checking procedure for coercion paths is complete for paths consisting of coercions satisfying the uniform inheritance condition, but some coercion paths could be reported as ambiguous even if they are convertible with existing ones when they have coercions that don't satisfy the uniform inheritance condition.

Warning: ... is not definitionally an identity function.
If a coercion path has the same source and target class, that is said to be circular. When a new circular coercion path is not convertible with the identity function, it will be reported as ambiguous.

Some objects can be declared as coercions when they are defined. This applies to *assumptions* and constructors of *inductive types and record fields*. Use **:>** instead of **:** before the ***type*** of the assumption to do so. See ***of_type***.

Command: Identity Coercion *ident* : *class* >-> *class*
If `C` is the source `class` and `D` the destination, we check that `C` is a *constant* with a *body* of the form `fun (x₁:T₁)..(x?:T?) => D t₁..t?` where `m` is the number of parameters of `D`. Then we define an identity function with type `forall (x₁:T₁)..(x?:T?)(y:C x₁..x?),D t₁..t?`, and we declare it as an identity coercion between `C` and `D`.

This command supports the *local* attribute, which makes the coercion local to the current section.

Error: *class* must be a transparent constant.

Command: SubClass *ident_decl def_body*
If ***type*** is a class ***ident'*** applied to some arguments then ***ident*** is defined and an identity coercion of name **Id_*ident*_*ident'*** is declared. Otherwise said, this is an abbreviation for

Definition *ident* := *type*. Identity Coercion Id_*ident*_*ident'* : *ident* >-> *ident'*.

This command supports the *local* attribute, which makes the coercion local to the current section.

Displaying Available Coercions

Command: **`Print Classes`**
Print the list of declared classes in the current context.

Command: **`Print Coercions`**
Print the list of declared coercions in the current context.

Command: **`Print Graph`**
Print the list of valid coercion paths in the current context.

Command: **`Print Coercion Paths`** ***`class class`***
Print the list of valid coercion paths between the two given classes.

Activating the Printing of Coercions

Flag: **`Printing Coercions`**
When on, this *flag* forces all the coercions to be printed. By default, coercions are not printed.

Table: **`Printing Coercion`** ***`qualid`***
This *table* specifies a set of qualids for which coercions are always displayed. Use the `Add` and `Remove` commands to update the set of qualids.

Classes as Records

Structures with Inheritance may be defined using the `Record` command.

Use > before the record name to declare the constructor name as a coercion from the class of the last field type to the record name (this may fail if the uniform inheritance condition is not satisfied). See `record_definition`.

Use `:>` in the field type to declare the field as a coercion from the record name to the class of the field type. See `of_type`.

Coercions and Sections

The inheritance mechanism is compatible with the section mechanism. The global classes and coercions defined inside a section are redefined after its closing, using their new value and new type. The classes and coercions which are local to the section are simply forgotten. Coercions with a local source class or a local target class, and coercions which do not verify the uniform inheritance condition any longer are also forgotten.

Coercions and Modules

The coercions present in a module are activated only when the module is explicitly imported.

Examples

There are three situations:

Coercion at function application

`f a` is ill-typed where `f:forall x:A,B` and `a:A'`. If there is a coercion path between `A'` and `A`, then `f a` is transformed into `f a'` where `a'` is the result of the application of this coercion path to `a`.

We first give an example of coercion between atomic inductive types

```
Definition bool_in_nat (b:bool) := if b then 0 else 1.
    bool_in_nat is defined

Coercion bool_in_nat : bool >-> nat.
    bool_in_nat is now a coercion

Check (0 = true).
    0 = true
         : Prop

Set Printing Coercions.
Check (0 = true).
    0 = bool_in_nat true
         : Prop

Unset Printing Coercions.
```

Warning: Note that `Check (true = 0)` would fail. This is "normal" behavior of coercions. To validate `true=0`, the coercion is searched from `nat` to `bool`. There is none.

We give an example of coercion between classes with parameters.

```
Parameters (C : nat -> Set) (D : nat -> bool -> Set) (E : bool -> Set).
    C is declared
    D is declared
    E is declared

Parameter f : forall n:nat, C n -> D (S n) true.
    f is declared

Coercion f : C >-> D.
    f is now a coercion

Parameter g : forall (n:nat) (b:bool), D n b -> E b.
    g is declared

Coercion g : D >-> E.
    g is now a coercion

Parameter c : C 0.
    c is declared

Parameter T : E true -> nat.
    T is declared

Check (T c).
    T c
         : nat
```

(continues on next page)

(continued from previous page)

```
Set Printing Coercions.
Check (T c).
    T (g 1 true (f 0 c))
         : nat

Unset Printing Coercions.
```

We give now an example using identity coercions.

```
Definition D' (b:bool) := D 1 b.
    D' is defined

Identity Coercion IdD'D : D' >-> D.
Print IdD'D.
    IdD'D =
    (fun (b : bool) (x : D' b) => x) : forall b : bool, D' b -> D 1 b
         : forall b : bool, D' b -> D 1 b

    Arguments IdD'D _%bool_scope _
    IdD'D is a coercion

Parameter d' : D' true.
    d' is declared

Check (T d').
    T d'
         : nat

Set Printing Coercions.
Check (T d').
    T (g 1 true d')
         : nat

Unset Printing Coercions.
```

In the case of functional arguments, we use the monotonic rule of sub-typing. To coerce `t : forall x : A, B` towards `forall x : A', B'`, we have to coerce `A'` towards `A` and `B` towards `B'`. An example is given below:

```
Parameters (A B : Set) (h : A -> B).
    A is declared
    B is declared
    h is declared

Coercion h : A >-> B.
    h is now a coercion

Parameter U : (A -> E true) -> nat.
    U is declared

Parameter t : B -> C 0.
    t is declared

Check (U t).
    U (fun x : A => t x)
         : nat

Set Printing Coercions.
```

(continues on next page)

(continued from previous page)

```
Check (U t).
    U (fun x : A => g 1 true (f 0 (t (h x))))
         : nat

Unset Printing Coercions.
```

Remark the changes in the result following the modification of the previous example.

```
Parameter U' : (C 0 -> B) -> nat.
    U' is declared

Parameter t' : E true -> A.
    t' is declared

Check (U' t').
    U' (fun x : C 0 => t' x)
         : nat

Set Printing Coercions.
Check (U' t').
    U' (fun x : C 0 => h (t' (g 1 true (f 0 x))))
         : nat

Unset Printing Coercions.
```

Coercion to a type

An assumption `x:A` when `A` is not a type, is ill-typed. It is replaced by `x:A'` where `A'` is the result of the application to `A` of the coercion path between the class of `A` and `Sortclass` if it exists. This case occurs in the abstraction `fun x:A => t`, universal quantification `forall x:A,B`, global variables and parameters of (co-)inductive definitions and functions. In `forall x:A,B`, such a coercion path may also be applied to `B` if necessary.

```
Parameter Graph : Type.
    Graph is declared

Parameter Node : Graph -> Type.
    Node is declared

Coercion Node : Graph >-> Sortclass.
    Node is now a coercion

Parameter G : Graph.
    G is declared

Parameter Arrows : G -> G -> Type.
    Arrows is declared

Check Arrows.
    Arrows
         : G -> G -> Type

Parameter fg : G -> G.
    fg is declared

Check fg.
```

(continues on next page)

(continued from previous page)

```
    fg
         : G -> G

Set Printing Coercions.
Check fg.
    fg
         : Node G -> Node G

Unset Printing Coercions.
```

Coercion to a function

`f a` is ill-typed because `f:A` is not a function. The term `f` is replaced by the term obtained by applying to `f` the coercion path between `A` and `Funclass` if it exists.

```
Parameter bij : Set -> Set -> Set.
    bij is declared

Parameter ap : forall A B:Set, bij A B -> A -> B.
    ap is declared

Coercion ap : bij >-> Funclass.
    ap is now a coercion

Parameter b : bij nat nat.
    b is declared

Check (b 0).
    b 0
         : nat

Set Printing Coercions.
Check (b 0).
    ap nat nat b 0
         : nat

Unset Printing Coercions.
```

Let us see the resulting graph after all these examples.

```
Print Graph.
    [h] : A >-> B
    [f] : C >-> D
    [f; g] : C >-> E
    [g] : D >-> E
    [IdD'D] : D' >-> D
    [IdD'D; g] : D' >-> E
    [Node] : Graph >-> Sortclass
    [ap] : bij >-> Funclass
    [bool_in_nat] : bool >-> nat
```

2.2.7 Typeclasses

This chapter presents a quick reference of the commands related to type classes. For an actual introduction to typeclasses, there is a description of the system [SO08] and the literature on type classes in Haskell which also applies.

Class and Instance declarations

The syntax for class and instance declarations is the same as the record syntax of Coq:

```
Class classname (p1 : t1) … (pn : tn) [: sort] := { f1 : u1 ; … ; fm : um }.

Instance instancename q1 … qm : classname p1 … pn := { f1 := t1 ; … ; fm := tm }.
```

The `pi : ti` variables are called the *parameters* of the class and the `fi : ti` are called the *methods*. Each class definition gives rise to a corresponding record declaration and each instance is a regular definition whose name is given by `instancename` and type is an instantiation of the record type.

We'll use the following example class in the rest of the chapter:

```
Class EqDec (A : Type) :=
  { eqb : A -> A -> bool ;
    eqb_leibniz : forall x y, eqb x y = true -> x = y }.
```

This class implements a boolean equality test which is compatible with Leibniz equality on some type. An example implementation is:

```
Instance unit_EqDec : EqDec unit :=
  { eqb x y := true ;
    eqb_leibniz x y H :=
      match x, y return x = y with
      | tt, tt => eq_refl tt
      end }.
```

Using the *refine* attribute, if the term is not sufficient to finish the definition (e.g. due to a missing field or non-inferable hole) it must be finished in proof mode. If it is sufficient a trivial proof mode with no open goals is started.

```
#[refine] Instance unit_EqDec' : EqDec unit := { eqb x y := true }.
Proof. intros [] [];reflexivity. Defined.
```

Note that if you finish the proof with *Qed* the entire instance will be opaque, including the fields given in the initial term.

Alternatively, in *Program Mode* if one does not give all the members in the Instance declaration, Coq generates obligations for the remaining fields, e.g.:

```
Require Import Program.Tactics.
Program Instance eq_bool : EqDec bool :=
  { eqb x y := if x then y else negb y }.

Next Obligation.
    1 goal

      x, y : bool
      H : (if x then y else negb y) = true
      ============================
      x = y

  destruct x ; destruct y ; (discriminate || reflexivity).
```

(continues on next page)

(continued from previous page)

```
    No more goals.

Defined.
```

One has to take care that the transparency of every field is determined by the transparency of the *Instance* proof. One can use alternatively the *program* attribute to get richer facilities for dealing with obligations.

Binding classes

Once a typeclass is declared, one can use it in class binders:

```
Definition neqb {A} {eqa : EqDec A} (x y : A) := negb (eqb x y).
    neqb is defined
```

When one calls a class method, a constraint is generated that is satisfied only in contexts where the appropriate instances can be found. In the example above, a constraint `EqDec A` is generated and satisfied by `eqa : EqDec A`. In case no satisfying constraint can be found, an error is raised:

```
Fail Definition neqb' (A : Type) (x y : A) := negb (eqb x y).
    The command has indeed failed with message:
    The following term contains unresolved implicit arguments:
      (fun (A : Type) (x y : A) => negb (eqb x y))
    More precisely:
    - ?EqDec: Cannot infer the implicit parameter EqDec of eqb whose type is
      "EqDec A" (no type class instance found) in environment:
      A : Type
      x, y : A
```

The algorithm used to solve constraints is a variant of the *eauto* tactic that does proof search with a set of lemmas (the instances). It will use local hypotheses as well as declared lemmas in the `typeclass_instances` database. Hence the example can also be written:

```
Definition neqb' A (eqa : EqDec A) (x y : A) := negb (eqb x y).
    neqb' is defined
```

However, the generalizing binders should be used instead as they have particular support for typeclasses:

- They automatically set the maximally implicit status for typeclass arguments, making derived functions as easy to use as class methods. In the example above, `A` and `eqa` should be set maximally implicit.
- They support implicit quantification on partially applied type classes (*Implicit generalization*). Any argument not given as part of a typeclass binder will be automatically generalized.
- They also support implicit quantification on *Superclasses*.

Following the previous example, one can write:

```
Generalizable Variables A B C.

Definition neqb_implicit `{eqa : EqDec A} (x y : A) := negb (eqb x y).
    neqb_implicit is defined
```

Here `A` is implicitly generalized, and the resulting function is equivalent to the one above.

Parameterized Instances

One can declare parameterized instances as in Haskell simply by giving the constraints as a binding context before the instance, e.g.:

```
Program Instance prod_eqb `(EA : EqDec A, EB : EqDec B) : EqDec (A * B) :=
  { eqb x y := match x, y with
               | (la, ra), (lb, rb) => andb (eqb la lb) (eqb ra rb)
               end }.
```

These instances are used just as well as lemmas in the instance hint database.

Sections and contexts

To ease developments parameterized by many instances, one can use the *Context* command to introduce the parameters into the *local context*, it works similarly to the command *Variable*, except it accepts any binding context as an argument, so variables can be implicit, and *Implicit generalization* can be used. For example:

```
Section EqDec_defs.

Context `{EA : EqDec A}.
    A is declared
    EA is declared

#[ global, program ] Instance option_eqb : EqDec (option A) :=
  { eqb x y := match x, y with
         | Some x, Some y => eqb x y
         | None, None => true
         | _, _ => false
         end }.
Admit Obligations.

End EqDec_defs.

About option_eqb.
    option_eqb : forall {A : Type}, EqDec A -> EqDec (option A)

    option_eqb is not universe polymorphic
    Arguments option_eqb {A}%type_scope {EA}
    option_eqb is transparent
    Expands to: Constant Top.option_eqb
```

Here the *global* attribute redeclares the instance at the end of the section, once it has been generalized by the context variables it uses.

See also:

Section *Section mechanism*

Building hierarchies

Superclasses

One can also parameterize classes by other classes, generating a hierarchy of classes and superclasses. In the same way, we give the superclasses as a binding context:

```
Class Ord `(E : EqDec A) := { le : A -> A -> bool }.
    Ord is defined
    le is defined
```

Contrary to Haskell, we have no special syntax for superclasses, but this declaration is equivalent to:

```
Class `(E : EqDec A) => Ord A :=
  { le : A -> A -> bool }.
```

This declaration means that any instance of the `Ord` class must have an instance of `EqDec`. The parameters of the subclass contain at least all the parameters of its superclasses in their order of appearance (here A is the only one). As we have seen, `Ord` is encoded as a record type with two parameters: a type `A` and an `E` of type `EqDec A`. However, one can still use it as if it had a single parameter inside generalizing binders: the generalization of superclasses will be done automatically.

```
Definition le_eqb `{Ord A} (x y : A) := andb (le x y) (le y x).
    le_eqb is defined
```

In some cases, to be able to specify sharing of structures, one may want to give explicitly the superclasses. It is is possible to do it directly in regular binders, and using the `!` modifier in class binders. For example:

```
Definition lt `{eqa : EqDec A, ! Ord eqa} (x y : A) := andb (le x y) (neqb x y).
    lt is defined
```

The `!` modifier switches the way a binder is parsed back to the usual interpretation of Coq. In particular, it uses the implicit arguments mechanism if available, as shown in the example.

Substructures

Substructures are components of a class which are instances of a class themselves. They often arise when using classes for logical properties, e.g.:

```
Class Reflexive (A : Type) (R : relation A) :=
  reflexivity : forall x, R x x.

Class Transitive (A : Type) (R : relation A) :=
  transitivity : forall x y z, R x y -> R y z -> R x z.
```

This declares singleton classes for reflexive and transitive relations, (see the *singleton class* variant for an explanation). These may be used as parts of other classes:

```
Class PreOrder (A : Type) (R : relation A) :=
  { PreOrder_Reflexive :> Reflexive A R ;
    PreOrder_Transitive :> Transitive A R }.
    PreOrder is defined
    PreOrder_Reflexive is defined
    PreOrder_Transitive is defined
```

The syntax `:>` indicates that each `PreOrder` can be seen as a `Reflexive` relation. So each time a reflexive relation is needed, a preorder can be used instead. This is very similar to the coercion mechanism of `Structure` declarations. The implementation simply declares each projection as an instance.

Warning: `Ignored instance declaration for "ident": "term" is not a class`
Using this `:>` syntax with a right-hand-side that is not itself a Class has no effect (apart from emitting this warning). In particular, is does not declare a coercion.

One can also declare existing objects or structure projections using the Existing Instance command to achieve the same effect.

Summary of the commands

Command: `Class` *`record_definition`*
Command: `Class` *`singleton_class_definition`*

singleton_class_definition `::=` `>`? *ident_decl* *binder** `: sort`? `:=` *constructor*

The first form declares a record and makes the record a typeclass with parameters *`binder`** and the listed record fields.

The second form declares a *singleton* class with a single method. This singleton class is a so-called definitional class, represented simply as a definition `ident binders := term` and whose instances are themselves objects of this type. Definitional classes are not wrapped inside records, and the trivial projection of an instance of such a class is convertible to the instance itself. This can be useful to make instances of existing objects easily and to reduce proof size by not inserting useless projections. The class *constant* itself is declared rigid during resolution so that the class abstraction is maintained.

Like any command declaring a record, this command supports the *`universes(polymorphic)`*, *`universes(template)`*, *`universes(cumulative)`*, and *`private(matching)`* attributes.

Command: `Existing Class` *`qualid`*
This variant declares a class from a previously declared *constant* or inductive definition. No methods or instances are defined.

Warning: *`ident`* `is already declared as a typeclass`
This command has no effect when used on a typeclass.

Command:

`Instance` *`ident_decl`* *`binder`**? `:` *`type`* *`hint_info`*? (`:= {` *`field_def`** `}` | `:=` *`term`*)?

Declares a typeclass instance named *`ident_decl`* of the class *`type`* with the specified parameters and with fields defined by *`field_def`*, where each field must be a declared field of the class.

Adds one or more *`binder`*s to declare a parameterized instance. *`hint_info`* may be used to specify the hint priority, where 0 is the highest priority as for *`auto`* hints. If the priority is not specified, the default is the number of non-dependent binders of the instance. If *`one_pattern`* is given, terms matching that pattern will trigger use of the instance. Otherwise, use is triggered based on the conclusion of the type.

This command supports the *`local`*, *`global`* and *`export`* locality attributes.

Deprecated since version 8.14: The default value for instance locality will change in a future release. Instances added outside of sections without an explicit locality are now deprecated. We recommend using *`export`* where possible.

Like *`Definition`*, it also supports the *`program`* attribute to switch the type checking to `Program` (chapter *Program*) and to use the obligation mechanism to manage missing fields.

Finally, it supports the lighter *refine* attribute:

Attribute: refine
This *attribute* can be used to leave holes or not provide all fields in the definition of an instance and open the tactic mode to fill them. It works exactly as if no *body* had been given and the *refine* tactic has been used first.

Command: Declare Instance *ident_decl* *binder** **:** *term* *hint_info*?
In a *Module Type*, declares that a corresponding concrete instance should exist in any implementation of this *Module Type*. This is similar to the distinction between *Parameter* vs. *Definition*, or between *Declare Module* and *Module*.

Command: Existing Instance *qualid* *hint_info*?

Command: Existing Instances *qualid*+ | *natural*?
Adds a *constant* whose type ends with an applied typeclass to the instance database with an optional priority *natural*. It can be used for redeclaring instances at the end of sections, or declaring structure projections as instances. This is equivalent to `Hint Resolve ident : typeclass_instances`, except it registers instances for *Print Instances*.

Flag: Instance Generalized Output
Deprecated since version 8.13.

Disabled by default, this *flag* provides compatibility with Coq version 8.12 and earlier.

When enabled, the type of the instance is implicitly generalized over unbound and *generalizable* variables as though surrounded by `\` { }`.

Command: Print Instances *reference*
Shows the list of instances associated with the typeclass *reference*.

Tactic: typeclasses eauto **bfs**? *nat_or_var*? **with** *ident*+?
This proof search tactic uses the resolution engine that is run implicitly during type checking. This tactic uses a different resolution engine than *eauto* and *auto*. The main differences are the following:

- Unlike *eauto* and *auto*, the resolution is done entirely in the proof engine, meaning that backtracking is available among dependent subgoals, and shelving goals is supported. `typeclasses eauto` is a multi-goal tactic. It analyses the dependencies between subgoals to avoid backtracking on subgoals that are entirely independent.
- The transparency information of databases is used consistently for all hints declared in them. It is always used when calling the unifier. When considering local hypotheses, we use the transparent state of the first hint database given. Using an empty database (created with *Create HintDb* for example) with unfoldable variables and *constants* as the first argument of `typeclasses eauto` hence makes resolution with the local hypotheses use full conversion during unification.
- The mode hints (see *Hint Mode*) associated with a class are taken into account by *typeclasses eauto*. When a goal does not match any of the declared modes for its head (if any), instead of failing like *eauto*, the goal is suspended and resolution proceeds on the remaining goals. If after one run of resolution, there remains suspended goals, resolution is launched against on them, until it reaches a fixed point when the set of remaining suspended goals does not change. Using `solve [typeclasses eauto]` can be used to ensure that no suspended goals remain.
- When considering local hypotheses, we use the union of all the modes declared in the given databases.
- Use the *Typeclasses eauto* command to customize the behavior of this tactic.

nat_or_var Specifies the maximum depth of the search.

> **Warning:** The semantics for the limit *nat_or_var* are different than for *auto*. By default, if no limit is given, the search is unbounded. Unlike *auto*, introduction steps count against the limit, which might result in larger limits being necessary when searching with *typeclasses eauto* than with *auto*.

with *ident* Runs resolution with the specified hint databases. It treats typeclass subgoals the same as other subgoals (no shelving of non-typeclass goals in particular), while allowing shelved goals to remain at any point during search.

When **with** is not specified, *typeclasses eauto* uses the `typeclass_instances` database by default (instead of `core`). Dependent subgoals are automatically shelved, and shelved goals can remain after resolution ends (following the behavior of Coq 8.5).

Note: `all:once (typeclasses eauto)` faithfully mimics what happens during typeclass resolution when it is called during refinement/type inference, except that *only* declared class subgoals are considered at the start of resolution during type inference, while `all` can select non-class subgoals as well. It might move to `all:typeclasses eauto` in future versions when the refinement engine will be able to backtrack.

Tactic: autoapply *one_term* with *ident*

The tactic `autoapply` applies *one_term* using the transparency information of the hint database *ident*, and does *no* typeclass resolution. This can be used in *Hint Extern*'s for typeclass instances (in the hint database `typeclass_instances`) to allow backtracking on the typeclass subgoals created by the lemma application, rather than doing typeclass resolution locally at the hint application time.

Typeclasses Transparent, Typeclasses Opaque

Command: Typeclasses Transparent *qualid*

Makes *qualid* transparent during typeclass resolution. A shortcut for *Hint Transparent* *qualid* : **typeclass_instances**

Command: Typeclasses Opaque *qualid*

Make *qualid* opaque for typeclass search. A shortcut for *Hint Opaque* *qualid* : **typeclass_instances**.

It is useful when some *constants* prevent some unifications and make resolution fail. It is also useful to declare constants which should never be unfolded during proof search, like fixpoints or anything which does not look like an abbreviation. This can additionally speed up proof search as the typeclass map can be indexed by such rigid constants (see *The hints databases for auto and eauto*).

By default, all *constants* and local variables are considered transparent. One should take care not to make opaque any constant that is used to abbreviate a type, like:

```
Definition relation A := A -> A -> Prop.
```

Settings

Flag: `Typeclasses Dependency Order`
This *flag* (off by default) respects the dependency order between subgoals, meaning that subgoals on which other subgoals depend come first, while the non-dependent subgoals were put before the dependent ones previously (Coq 8.5 and below). This can result in quite different performance behaviors of proof search.

Flag: `Typeclasses Filtered Unification`
This *flag*, which is off by default, switches the hint application procedure to a filter-then-unify strategy. To apply a hint, we first check that the goal *matches* syntactically the inferred or specified pattern of the hint, and only then try to *unify* the goal with the conclusion of the hint. This can drastically improve performance by calling unification less often, matching syntactic patterns being very quick. This also provides more control on the triggering of instances. For example, forcing a *constant* to explicitly appear in the pattern will make it never apply on a goal where there is a hole in that place.

Flag: `Typeclasses Limit Intros`
This *flag* (on by default) controls the ability to apply hints while avoiding (functional) eta-expansions in the generated proof term. It does so by allowing hints that conclude in a product to apply to a goal with a matching product directly, avoiding an introduction.

> **Warning:** This can be expensive as it requires rebuilding hint clauses dynamically, and does not benefit from the invertibility status of the product introduction rule, resulting in potentially more expensive proof search (i.e. more useless backtracking).

Flag: `Typeclass Resolution For Conversion`
This *flag* (on by default) controls the use of typeclass resolution when a unification problem cannot be solved during elaboration/type inference. With this flag on, when a unification fails, typeclass resolution is tried before launching unification once again.

Flag: `Typeclasses Strict Resolution`
Typeclass declarations introduced when this *flag* is set have a stricter resolution behavior (the flag is off by default). When looking for unifications of a goal with an instance of this class, we "freeze" all the existentials appearing in the goals, meaning that they are considered rigid during unification and cannot be instantiated.

Flag: `Typeclasses Unique Solutions`
When a typeclass resolution is launched we ensure that it has a single solution or fail. This *flag* ensures that the resolution is canonical, but can make proof search much more expensive.

Flag: `Typeclasses Unique Instances`
Typeclass declarations introduced when this *flag* is set have a more efficient resolution behavior (the flag is off by default). When a solution to the typeclass goal of this class is found, we never backtrack on it, assuming that it is canonical.

Flag: `Typeclasses Iterative Deepening`
When this *flag* is set, the proof search strategy is breadth-first search. Otherwise, the search strategy is depth-first search. The default is off. `Typeclasses eauto` is another way to set this flag.

Option: `Typeclasses Depth` *`natural`*
This *option* sets the maximum proof search depth. The default is unbounded. `Typeclasses eauto` is another way to set this option.

Flag: `Typeclasses Debug`
Controls whether typeclass resolution steps are shown during search. Setting this *flag* also sets `Typeclasses Debug Verbosity` to 1. `Typeclasses eauto` is another way to set this flag.

Option: `Typeclasses Debug Verbosity` *`natural`*
Determines how much information is shown for typeclass resolution steps during search. 1 is the default level. 2

shows additional information such as tried tactics and shelving of goals. Setting this *option* to 1 or 2 turns on the `Typeclasses Debug` flag; setting this option to 0 turns that flag off.

Typeclasses eauto

Command: `Typeclasses eauto := debug? ( bfs | dfs )? natural?`

Allows more global customization of the `typeclasses eauto` tactic. The options are:

`debug` Sets debug mode. In debug mode, a trace of successfully applied tactics is printed. Debug mode can also be set with `Typeclasses Debug`.

`dfs`, `bfs` Sets the search strategy to depth-first search (the default) or breadth-first search. The search strategy can also be set with `Typeclasses Iterative Deepening`.

`natural` Sets the depth limit for the search. The limit can also be set with `Typeclasses Depth`.

2.2.8 Canonical Structures

Authors Assia Mahboubi and Enrico Tassi

This chapter explains the basics of canonical structures and how they can be used to overload notations and build a hierarchy of algebraic structures. The examples are taken from [MT13]. We invite the interested reader to refer to this paper for all the details that are omitted here for brevity. The interested reader shall also find in [GZND11] a detailed description of another, complementary, use of canonical structures: advanced proof search. This latter papers also presents many techniques one can employ to tune the inference of canonical structures.

Declaration of canonical structures

A canonical structure is an instance of a record/structure type that can be used to solve unification problems involving a projection applied to an unknown structure instance (an implicit argument) and a value. The complete documentation of canonical structures can be found in *Canonical Structures*; here only a simple example is given.

Command: `Canonical Structure? reference`

Command: `Canonical Structure? ident_decl def_body`

The first form of this command declares an existing ***`reference`*** as a canonical instance of a structure (a record).

The second form defines a new *constant* as if the `Definition` command had been used, then declares it as a canonical instance as if the first form had been used on the defined object.

This command supports the `local` attribute. When used, the structure is canonical only within the `Section` containing it.

`qualid` (in `reference`) denotes an object **`(Build_struct c1 … cn)`** in the structure `struct` for which the fields are **$x_1, \ldots, x_n$**. Then, each time an equation of the form **$(x_i \ \_)$** $=_{\beta\delta\iota\zeta}$ **c_i** has to be solved during the type checking process, `qualid` is used as a solution. Otherwise said, `qualid` is canonically used to extend the field **x_i** into a complete structure built on **c_i** when **c_i** unifies with **$(x_i \ \_)$**.

The following kinds of terms are supported for the fields **c_i** of `qualid`:

- *Constants* and section variables of an active section, applied to zero or more arguments.
- `sort`s.
- Literal functions: `fun … => ….`

- Literal, (possibly dependent) function types: … `->` … and `forall …,` ….
- Variables bound in *qualid*.

Only the head symbol of an existing instance's field $\mathtt{c_i}$ is considered when searching for a canonical extension. We call this head symbol the *key* and we say "*qualid keys* the field $\mathtt{x_i}$ to $\mathtt{k}$" when $\mathtt{c_i}$'s head symbol is $\mathtt{k}$. Keys are the only piece of information that is used for canonical extension. The keys corresponding to the kinds of terms listed above are:

- For constants and section variables, potentially applied to arguments: the constant or variable itself, disregarding any arguments.
- For sorts: the sort itself.
- For literal functions: skip the abstractions and use the key of the body.
- For literal function types: a disembodied implication key denoted `forall _, _`, disregarding both its domain and codomain.
- For variables bound in *qualid*: a catch-all key denoted _.

This means that, for example, `(some_constant x1)` and `(some_constant (other_constant y1 y2) x2)` are not distinct keys.

Variables bound in *qualid* match any term for the purpose of canonical extension. This has two major consequences for a field $\mathtt{c_i}$ keyed to a variable of *qualid*:

1. Unless another key—and, thus, instance—matches $\mathtt{c_i}$, the instance will always be considered by unification.
2. $\mathtt{c_i}$ will be considered overlapping not distinct from any other canonical instance that keys $\mathtt{x_i}$ to one of its own variables.

A record field $\mathtt{x_i}$ can only be keyed once to each key. Coq prints a warning when *qualid* keys $\mathtt{x_i}$ to a term whose head symbol is already keyed by an existing canonical instance. In this case, Coq will not register that *qualid* as a canonical extension. (The remaining fields of the instance can still be used for canonical extension.)

Canonical structures are particularly useful when mixed with coercions and strict implicit arguments.

Example

Here is an example.

```
Require Import Relations.

Require Import EqNat.

Set Implicit Arguments.

Unset Strict Implicit.

Structure Setoid : Type := {Carrier :> Set; Equal : relation Carrier;
                            Prf_equiv : equivalence Carrier Equal}.
    Setoid is defined
    Carrier is defined
    Equal is defined
    Prf_equiv is defined


Definition is_law (A B:Setoid) (f:A -> B) := forall x y:A, Equal x y -> Equal (f␣
 ↪x) (f y).
    is_law is defined
```

(continues on next page)

(continued from previous page)

```
Axiom eq_nat_equiv : equivalence nat eq_nat.
    eq_nat_equiv is declared


Definition nat_setoid : Setoid := Build_Setoid eq_nat_equiv.
    nat_setoid is defined


Canonical nat_setoid.
```

Thanks to `nat_setoid` declared as canonical, the implicit arguments `A` and `B` can be synthesized in the next statement.

```
Lemma is_law_S : is_law S.
    1 goal

      ============================
      is_law (A:=nat_setoid) (B:=nat_setoid) S
```

Note: If a same field occurs in several canonical structures, then only the structure declared first as canonical is considered.

Attribute: `canonical` = `yes` | `no` ?

This *boolean attribute* can decorate a `Definition` or `Let` command. It is equivalent to having a `Canonical Structure` declaration just after the command.

To prevent a field from being involved in the inference of canonical instances, its declaration can be annotated with `canonical=no` (cf. the syntax of ***`record_field`***).

Example

For instance, when declaring the `Setoid` structure above, the `Prf_equiv` field declaration could be written as follows.

```
#[canonical=no] Prf_equiv : equivalence Carrier Equal
```

See *Hierarchy of structures* for a more realistic example.

Command: `Print Canonical Projections` *`reference`* *

This displays the list of global names that are components of some canonical structure. For each of them, the canonical structure of which it is a projection is indicated. If *constants* are given as its arguments, only the unification rules that involve or are synthesized from simultaneously all given constants will be shown.

Example

For instance, the above example gives the following output:

```
Print Canonical Projections.
    nat <- Carrier ( nat_setoid )
```

(continues on next page)

(continued from previous page)

```
    eq_nat <- Equal ( nat_setoid )
    eq_nat_equiv <- Prf_equiv ( nat_setoid )

Print Canonical Projections nat.
    nat <- Carrier ( nat_setoid )
```

Note: The last line in the first example would not show up if the corresponding projection (namely `Prf_equiv`) were annotated as not canonical, as described above.

Notation overloading

We build an infix notation == for a comparison predicate. Such notation will be overloaded, and its meaning will depend on the types of the terms that are compared.

```
Module EQ.
    Interactive Module EQ started

  Record class (T : Type) := Class { cmp : T -> T -> Prop }.
    class is defined
    cmp is defined

  Structure type := Pack { obj : Type; class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined

  Definition op (e : type) : obj e -> obj e -> Prop :=
    let 'Pack _ (Class _ the_cmp) := e in the_cmp.
    op is defined

  Check op.
    op
         : forall e : type, obj e -> obj e -> Prop

  Arguments op {e} x y : simpl never.
  Arguments Class {T} cmp.
  Module theory.
    Interactive Module theory started

    Notation "x == y" := (op x y) (at level 70).
  End theory.
    Module theory is defined

End EQ.
    Module EQ is defined
```

We use Coq modules as namespaces. This allows us to follow the same pattern and naming convention for the rest of the chapter. The base namespace contains the definitions of the algebraic structure. To keep the example small, the algebraic structure `EQ.type` we are defining is very simplistic, and characterizes terms on which a binary relation is defined, without requiring such relation to validate any property. The inner theory module contains the overloaded notation == and will eventually contain lemmas holding all the instances of the algebraic structure (in this case there are no lemmas).

Note that in practice the user may want to declare `EQ.obj` as a coercion, but we will not do that here.

The following line tests that, when we assume a type `e` that is in theEQ class, we can relate two of its objects with ==.

```
Import EQ.theory.
Check forall (e : EQ.type) (a b : EQ.obj e), a == b.
    forall (e : EQ.type) (a b : EQ.obj e), a == b
         : Prop
```

Still, no concrete type is in the `EQ` class.

```
Fail Check 3 == 3.
    The command has indeed failed with message:
    The term "3" has type "nat" while it is expected to have type "EQ.obj ?e".
```

We amend that by equipping `nat` with a comparison relation.

```
Definition nat_eq (x y : nat) := Nat.compare x y = Eq.
    nat_eq is defined

Definition nat_EQcl : EQ.class nat := EQ.Class nat_eq.
    nat_EQcl is defined

Canonical Structure nat_EQty : EQ.type := EQ.Pack nat nat_EQcl.
    nat_EQty is defined

Check 3 == 3.
    3 == 3
         : Prop

Eval compute in 3 == 4.
    = Lt = Eq
         : Prop
```

This last test shows that Coq is now not only able to type check `3 == 3`, but also that the infix relation was bound to the `nat_eq` relation. This relation is selected whenever == is used on terms of type nat. This can be read in the line declaring the canonical structure `nat_EQty`, where the first argument to `Pack` is the key and its second argument a group of canonical values associated with the key. In this case we associate with nat only one canonical value (since its class, `nat_EQcl` has just one member). The use of the projection `op` requires its argument to be in the class `EQ`, and uses such a member (function) to actually compare its arguments.

Similarly, we could equip any other type with a comparison relation, and use the == notation on terms of this type.

Derived Canonical Structures

We know how to use == on base types, like `nat`, `bool`, `Z`. Here we show how to deal with type constructors, i.e. how to make the following example work:

```
Fail Check forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b).
    The command has indeed failed with message:
    In environment
    e : EQ.type
    a : EQ.obj e
    b : EQ.obj e
    The term "(a, b)" has type "(EQ.obj e * EQ.obj e)%type"
    while it is expected to have type "EQ.obj ?e".
```

The error message is telling that Coq has no idea on how to compare pairs of objects. The following construction is telling Coq exactly how to do that.

```
Definition pair_eq (e1 e2 : EQ.type) (x y : EQ.obj e1 * EQ.obj e2) :=
  fst x == fst y /\ snd x == snd y.
    pair_eq is defined


Definition pair_EQcl e1 e2 := EQ.Class (pair_eq e1 e2).
    pair_EQcl is defined


Canonical Structure pair_EQty (e1 e2 : EQ.type) : EQ.type :=
    EQ.Pack (EQ.obj e1 * EQ.obj e2) (pair_EQcl e1 e2).
    pair_EQty is defined


Check forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b).
    forall (e : EQ.type) (a b : EQ.obj e), (a, b) == (a, b)
         : Prop


Check forall n m : nat, (3, 4) == (n, m).
    forall n m : nat, (3, 4) == (n, m)
         : Prop
```

Thanks to the `pair_EQty` declaration, Coq is able to build a comparison relation for pairs whenever it is able to build a comparison relation for each component of the pair. The declaration associates to the key `*` (the type constructor of pairs) the canonical comparison relation `pair_eq` whenever the type constructor `*` is applied to two types being themselves in the `EQ` class.

Hierarchy of structures

To get to an interesting example we need another base class to be available. We choose the class of types that are equipped with an order relation, to which we associate the infix <= notation.

```
Module LE.
    Interactive Module LE started


  Record class T := Class { cmp : T -> T -> Prop }.
    class is defined
    cmp is defined


  Structure type := Pack { obj : Type; class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined


  Definition op (e : type) : obj e -> obj e -> Prop :=
    let 'Pack _ (Class _ f) := e in f.
    op is defined


  Arguments op {_} x y : simpl never.

  Arguments Class {T} cmp.
```

(continues on next page)

(continued from previous page)

```
  Module theory.
    Interactive Module theory started


    Notation "x <= y" := (op x y) (at level 70).

  End theory.
    Module theory is defined


End LE.
    Module LE is defined
```

As before we register a canonical `LE` class for `nat`.

```
Import LE.theory.

Definition nat_le x y := Nat.compare x y <> Gt.
    nat_le is defined


Definition nat_LEcl : LE.class nat := LE.Class nat_le.
    nat_LEcl is defined


Canonical Structure nat_LEty : LE.type := LE.Pack nat nat_LEcl.
    nat_LEty is defined
```

And we enable Coq to relate pair of terms with <=.

```
Definition pair_le e1 e2 (x y : LE.obj e1 * LE.obj e2) :=
   fst x <= fst y /\ snd x <= snd y.
    pair_le is defined


Definition pair_LEcl e1 e2 := LE.Class (pair_le e1 e2).
    pair_LEcl is defined


Canonical Structure pair_LEty (e1 e2 : LE.type) : LE.type :=
   LE.Pack (LE.obj e1 * LE.obj e2) (pair_LEcl e1 e2).
    pair_LEty is defined


Check (3,4,5) <= (3,4,5).
    (3, 4, 5) <= (3, 4, 5)
         : Prop
```

At the current stage we can use == and <= on concrete types, like tuples of natural numbers, but we can't develop an algebraic theory over the types that are equipped with both relations.

```
Check 2 <= 3 /\ 2 == 2.
    2 <= 3 /\ 2 == 2
         : Prop
```

(continues on next page)

(continued from previous page)

```
Fail Check forall (e : EQ.type) (x y : EQ.obj e), x <= y -> y <= x -> x == y.
    The command has indeed failed with message:
    In environment
    e : EQ.type
    x : EQ.obj e
    y : EQ.obj e
    The term "x" has type "EQ.obj e" while it is expected to have type
     "LE.obj ?e".


Fail Check forall (e : LE.type) (x y : LE.obj e), x <= y -> y <= x -> x == y.
    The command has indeed failed with message:
    In environment
    e : LE.type
    x : LE.obj e
    y : LE.obj e
    The term "x" has type "LE.obj e" while it is expected to have type
     "EQ.obj ?e".
```

We need to define a new class that inherits from both `EQ` and `LE`.

```
Module LEQ.
    Interactive Module LEQ started


  Record mixin (e : EQ.type) (le : EQ.obj e -> EQ.obj e -> Prop) :=
    Mixin { compat : forall x y : EQ.obj e, le x y /\ le y x <-> x == y }.
    mixin is defined
    compat is defined


  Record class T := Class {
                      EQ_class : EQ.class T;
                      LE_class : LE.class T;
                      extra : mixin (EQ.Pack T EQ_class) (LE.cmp T LE_class) }.
    class is defined
    EQ_class is defined
    LE_class is defined
    extra is defined


  Structure type := _Pack { obj : Type; #[canonical=no] class_of : class obj }.
    type is defined
    obj is defined
    class_of is defined


  Arguments Mixin {e le} _.

  Arguments Class {T} _ _ _.
```

The mixin component of the `LEQ` class contains all the extra content we are adding to `EQ` and `LE`. In particular it contains the requirement that the two relations we are combining are compatible.

The `class_of` projection of the `type` structure is annotated as *not canonical*; it plays no role in the search for instances.

Unfortunately there is still an obstacle to developing the algebraic theory of this new class.

```
Module theory.
    Interactive Module theory started


Fail Check forall (le : type) (n m : obj le), n <= m -> n <= m -> n == m.
    The command has indeed failed with message:
    In environment
    le : type
    n : obj le
    m : obj le
    The term "n" has type "obj le" while it is expected to have type "LE.obj ?e".
```

The problem is that the two classes `LE` and `LEQ` are not yet related by a subclass relation. In other words Coq does not see that an object of the `LEQ` class is also an object of the `LE` class.

The following two constructions tell Coq how to canonically build the `LE.type` and `EQ.type` structure given an `LEQ.type` structure on the same type.

```
Definition to_EQ (e : type) : EQ.type :=
   EQ.Pack (obj e) (EQ_class _ (class_of e)).
    to_EQ is defined


Canonical Structure to_EQ.

Definition to_LE (e : type) : LE.type :=
   LE.Pack (obj e) (LE_class _ (class_of e)).
    to_LE is defined


Canonical Structure to_LE.
```

We can now formulate out first theorem on the objects of the `LEQ` structure.

```
Lemma lele_eq (e : type) (x y : obj e) : x <= y -> y <= x -> x == y.
    1 goal

      e : type
      x, y : obj e
      ============================
      x <= y -> y <= x -> x == y


   now intros; apply (compat _ _ (extra _ (class_of e)) x y); split.
    No more goals.


   Qed.

   Arguments lele_eq {e} x y _ _.

   End theory.
    Module theory is defined


End LEQ.
    Module LEQ is defined
```

(continues on next page)

(continued from previous page)

```
Import LEQ.theory.

Check lele_eq.
    lele_eq
         : forall x y : LEQ.obj ?e, x <= y -> y <= x -> x == y
    where
    ?e : [ |- LEQ.type]
```

Of course one would like to apply results proved in the algebraic setting to any concrete instate of the algebraic structure.

```
Example test_algebraic (n m : nat) : n <= m -> m <= n -> n == m.
    1 goal

      n, m : nat
      ============================
      n <= m -> m <= n -> n == m


Fail apply (lele_eq n m).
    The command has indeed failed with message:
    In environment
    n, m : nat
    The term "n" has type "nat" while it is expected to have type "LEQ.obj ?e".


Abort.

Example test_algebraic2 (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2) :
     n <= m -> m <= n -> n == m.
    1 goal

      l1, l2 : LEQ.type
      n, m : LEQ.obj l1 * LEQ.obj l2
      ============================
      n <= m -> m <= n -> n == m


Fail apply (lele_eq n m).
    The command has indeed failed with message:
    In environment
    l1, l2 : LEQ.type
    n, m : LEQ.obj l1 * LEQ.obj l2
    The term "n" has type "(LEQ.obj l1 * LEQ.obj l2)%type"
    while it is expected to have type "LEQ.obj ?e".


Abort.
```

Again one has to tell Coq that the type `nat` is in the `LEQ` class, and how the type constructor `*` interacts with the `LEQ` class. In the following proofs are omitted for brevity.

```
Lemma nat_LEQ_compat (n m : nat) : n <= m /\ m <= n <-> n == m.
    1 goal

      n, m : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      n <= m /\ m <= n <-> n == m


Admitted.
    nat_LEQ_compat is declared


Definition nat_LEQmx := LEQ.Mixin nat_LEQ_compat.
    nat_LEQmx is defined


Lemma pair_LEQ_compat (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2) :
   n <= m /\ m <= n <-> n == m.
    1 goal

      l1, l2 : LEQ.type
      n, m : LEQ.obj l1 * LEQ.obj l2
      ============================
      n <= m /\ m <= n <-> n == m


Admitted.
    pair_LEQ_compat is declared


Definition pair_LEQmx l1 l2 := LEQ.Mixin (pair_LEQ_compat l1 l2).
    pair_LEQmx is defined
```

The following script registers an `LEQ` class for `nat` and for the type constructor `*`. It also tests that they work as expected.

Unfortunately, these declarations are very verbose. In the following subsection we show how to make them more compact.

```
Module Add_instance_attempt.
    Interactive Module Add_instance_attempt started


  Canonical Structure nat_LEQty : LEQ.type :=
    LEQ._Pack nat (LEQ.Class nat_EQcl nat_LEcl nat_LEQmx).
    nat_LEQty is defined


  Canonical Structure pair_LEQty (l1 l2 : LEQ.type) : LEQ.type :=
    LEQ._Pack (LEQ.obj l1 * LEQ.obj l2)
      (LEQ.Class
         (EQ.class_of (pair_EQty (to_EQ l1) (to_EQ l2)))
         (LE.class_of (pair_LEty (to_LE l1) (to_LE l2)))
         (pair_LEQmx l1 l2)).
    pair_LEQty is defined


   Example test_algebraic (n m : nat) : n <= m -> m <= n -> n == m.
    1 goal

      n, m : nat
      ============================
      n <= m -> m <= n -> n == m
```

(continues on next page)

(continued from previous page)

```
  now apply (lele_eq n m).
   No more goals.


  Qed.

  Example test_algebraic2 (n m : nat * nat) : n <= m -> m <= n -> n == m.
   1 goal

     n, m : nat * nat
     ============================
     n <= m -> m <= n -> n == m


  now apply (lele_eq n m). Qed.
   No more goals.


End Add_instance_attempt.
    Module Add_instance_attempt is defined
```

Note that no direct proof of `n <= m -> m <= n -> n == m` is provided by the user for `n` and m of type `nat * nat`. What the user provides is a proof of this statement for `n` and `m` of type `nat` and a proof that the pair constructor preserves this property. The combination of these two facts is a simple form of proof search that Coq performs automatically while inferring canonical structures.

Compact declaration of Canonical Structures

We need some infrastructure for that.

```
Require Import Strings.String.
    [Loading ML file ring_plugin.cmxs ... done]


Module infrastructure.
    Interactive Module infrastructure started


  Inductive phantom {T : Type} (t : T) : Type := Phantom.
    phantom is defined
    phantom_rect is defined
    phantom_ind is defined
    phantom_rec is defined
    phantom_sind is defined


  Definition unify {T1 T2} (t1 : T1) (t2 : T2) (s : option string) :=
    phantom t1 -> phantom t2.
    unify is defined


  Definition id {T} {t : T} (x : phantom t) := x.
    id is defined
```

(continues on next page)

(continued from previous page)

```
  Notation "[find v | t1 ~ t2 ] p" := (fun v (_ : unify t1 t2 None) => p)
    (at level 50, v name, only parsing).

  Notation "[find v | t1 ~ t2 | s ] p" := (fun v (_ : unify t1 t2 (Some s)) => p)
    (at level 50, v name, only parsing).

  Notation "'Error : t : s" := (unify _ t (Some s))
    (at level 50, format "''Error' : t : s").

  Open Scope string_scope.

End infrastructure.
    Module infrastructure is defined
```

To explain the notation `[find v | t1 ~ t2]` let us pick one of its instances: `[find e | EQ.obj e ~ T | "is not an EQ.type" ]`. It should be read as: "find a class e such that its objects have type T or fail with message "T is not an EQ.type"".

The other utilities are used to ask Coq to solve a specific unification problem, that will in turn require the inference of some canonical structures. They are explained in more details in [MT13].

We now have all we need to create a compact "packager" to declare instances of the `LEQ` class.

```
Import infrastructure.

Definition packager T e0 le0 (m0 : LEQ.mixin e0 le0) :=
  [find e | EQ.obj e ~ T | "is not an EQ.type" ]
  [find o | LE.obj o ~ T | "is not an LE.type" ]
  [find ce | EQ.class_of e ~ ce ]
  [find co | LE.class_of o ~ co ]
  [find m | m ~ m0 | "is not the right mixin" ]
  LEQ._Pack T (LEQ.Class ce co m).
    packager is defined


 Notation Pack T m := (packager T _ _ m _ id _ id _ id _ id).
```

The object `Pack` takes a type `T` (the key) and a mixin `m`. It infers all the other pieces of the class `LEQ` and declares them as canonical values associated with the `T` key. All in all, the only new piece of information we add in the `LEQ` class is the mixin, all the rest is already canonical for `T` and hence can be inferred by Coq.

`Pack` is a notation, hence it is not type checked at the time of its declaration. It will be type checked when it is used, an in that case `T` is going to be a concrete type. The odd arguments _ and `id` we pass to the packager represent respectively the classes to be inferred (like `e`, `o`, etc) and a token (`id`) to force their inference. Again, for all the details the reader can refer to [MT13].

The declaration of canonical instances can now be way more compact:

```
Canonical Structure nat_LEQty := Eval hnf in Pack nat nat_LEQmx.
    nat_LEQty is defined


Canonical Structure pair_LEQty (l1 l2 : LEQ.type) :=
   Eval hnf in Pack (LEQ.obj l1 * LEQ.obj l2) (pair_LEQmx l1 l2).
    pair_LEQty is defined
```

Error messages are also quite intelligible (if one skips to the end of the message).

```
Fail Canonical Structure err := Eval hnf in Pack bool nat_LEQmx.
    The command has indeed failed with message:
    The term "id" has type "phantom (EQ.obj ?e) -> phantom (EQ.obj ?e)"
    while it is expected to have type "'Error:bool:"is not an EQ.type"".
```

2.2.9 Program

Author Matthieu Sozeau

We present here the **Program** tactic commands, used to build certified Coq programs, elaborating them from their algorithmic skeleton and a rich specification [Soz07]. It can be thought of as a dual of *Extraction*. The goal of **Program** is to program as in a regular functional programming language whilst using as rich a specification as desired and proving that the code meets the specification using the whole Coq proof apparatus. This is done using a technique originating from the "Predicate subtyping" mechanism of PVS [ROS98], which generates type checking conditions while typing a term constrained to a particular type. Here we insert existential variables in the term, which must be filled with proofs to get a complete Coq term. **Program** replaces the **Program** tactic by Catherine Parent [Par95] which had a similar goal but is no longer maintained.

The languages available as input are currently restricted to Coq's term language, but may be extended to OCaml, Haskell and others in the future. We use the same syntax as Coq and permit to use implicit arguments and the existing coercion mechanism. Input terms and types are typed in an extended system (Russell) and interpreted into Coq terms. The interpretation process may produce some proof obligations which need to be resolved to create the final term.

Elaborating programs

The main difference from Coq is that an object in a type `T : Set` can be considered as an object of type `{x : T | P}` for any well-formed `P : Prop`. If we go from `T` to the subset of `T` verifying property `P`, we must prove that the object under consideration verifies it. Russell will generate an obligation for every such coercion. In the other direction, Russell will automatically insert a projection.

Another distinction is the treatment of pattern matching. Apart from the following differences, it is equivalent to the standard match operation (see *Extended pattern matching*).

- Generation of equalities. A match expression is always generalized by the corresponding equality. As an example, the expression:

  ```
  match x with
  | 0 => t
  | S n => u
  end.
  ```

  will be first rewritten to:

  ```
  (match x as y return (x = y -> _) with
  | 0 => fun H : x = 0 -> t
  | S n => fun H : x = S n -> u
  end) (eq_refl x).
  ```

  This permits to get the proper equalities in the context of proof obligations inside clauses, without which reasoning is very limited.

- Generation of disequalities. If a pattern intersects with a previous one, a disequality is added in the context of the second branch. See for example the definition of div2 below, where the second branch is typed in a context where `∀ p, _ <> S (S p)`.

- Coercion. If the object being matched is coercible to an inductive type, the corresponding coercion will be automatically inserted. This also works with the previous mechanism.

There are flags to control the generation of equalities and coercions.

Flag: `Program Cases`
This *flag* controls the special treatment of pattern matching generating equalities and disequalities when using **Program** (it is on by default). All pattern-matches and let-patterns are handled using the standard algorithm of Coq (see *Extended pattern matching*) when this flag is deactivated.

Flag: `Program Generalized Coercion`
This *flag* controls the coercion of general inductive types when using **Program** (the flag is on by default). Coercion of subset types and pairs is still active in this case.

Flag: `Program Mode`
This *flag* enables the program mode, in which 1) typechecking allows subset coercions and 2) the elaboration of pattern matching of `Fixpoint` and `Definition` acts as if the `program` attribute has been used, generating obligations if there are unresolved holes after typechecking.

Attribute: `program` = `yes` | `no` ?
This *boolean attribute* allows using or disabling the Program mode on a specific definition. An alternative and commonly used syntax is to use the legacy `Program` prefix (cf. ***`legacy_attr`***) as it is elsewhere in this chapter.

Syntactic control over equalities

To give more control over the generation of equalities, the type checker will fall back directly to Coq's usual typing of dependent pattern matching if a `return` or `in` clause is specified. Likewise, the if construct is not treated specially by **Program** so boolean tests in the code are not automatically reflected in the obligations. One can use the `dec` combinator to get the correct hypotheses as in:

```
Require Import Program Arith.

Program Definition id (n : nat) : { x : nat | x = n } :=
  if dec (leb n 0) then 0
  else S (pred n).
    id has type-checked, generating 2 obligations
    Solving obligations automatically...
    2 obligations remaining
    Obligation 1 of id:
    (forall n : nat, (n <=? 0) = true -> (fun x : nat => x = n) 0).

    Obligation 2 of id:
    (forall n : nat,
     (n <=? 0) = false -> (fun x : nat => x = n) (S (Init.Nat.pred n))).
```

The `let` tupling construct `let (x1, ..., xn) := t in b` does not produce an equality, contrary to the let pattern construct `let '(x1,..., xn) := t in b`.

The next two commands are similar to their standard counterparts `Definition` and `Fixpoint` in that they define *constants*. However, they may require the user to prove some goals to construct the final definitions.

Program Definition

A *Definition* command with the *program* attribute types the value term in Russell and generates proof obligations. Once solved using the commands shown below, it binds the final Coq term to the name ***ident*** in the global environment.

Program Definition ***ident*** **:** ***type*** **:=** ***term***

Interprets the type ***type***, potentially generating proof obligations to be resolved. Once done with them, we have a Coq type $type_0$. It then elaborates the preterm ***term*** into a Coq term $term_0$, checking that the type of $term_0$ is coercible to $type_0$, and registers ***ident*** as being of type $type_0$ once the set of obligations generated during the interpretation of $term_0$ and the aforementioned coercion derivation are solved.

See also:

Sections *Controlling reduction strategies and the conversion algorithm*, *unfold*

Program Fixpoint

A *Fixpoint* command with the *program* attribute may also generate obligations. It works with mutually recursive definitions too. For example:

```
Require Import Program Arith.

Program Fixpoint div2 (n : nat) : { x : nat | n = 2 * x \/ n = 2 * x + 1 } :=
  match n with
  | S (S p) => S (div2 p)
  | _ => O
  end.
    Solving obligations automatically...
    4 obligations remaining
```

The *Fixpoint* command may include an optional ***fixannot*** annotation, which can be:

- `measure f R` where `f` is a value of type `X` computed on any subset of the arguments and the optional term `R` is a relation on `X`. `X` defaults to `nat` and `R` to `lt`.
- `wf R x` which is equivalent to `measure x R`.

Here we have one obligation for each branch (branches for `0` and `(S 0)` are automatically generated by the pattern matching compilation algorithm).

```
Obligation 1.
    1 goal

      p, x : nat
      o : p = x + (x + 0) \/ p = x + (x + 0) + 1
      ============================
      S (S p) = S (x + S (x + 0)) \/ S (S p) = S (x + S (x + 0) + 1)
```

One can use a well-founded order or a measure as termination orders using the syntax:

```
Program Fixpoint div2 (n : nat) {measure n} : { x : nat | n = 2 * x \/ n = 2 * x + 1 }
 ↪ :=
  match n with
  | S (S p) => S (div2 p)
  | _ => O
  end.
```

Caution: When defining structurally recursive functions, the generated obligations should have the prototype of the currently defined functional in their context. In this case, the obligations should be transparent (e.g. defined using `Defined`) so that the guardedness condition on recursive calls can be checked by the kernel's type- checker. There is an optimization in the generation of obligations which gets rid of the hypothesis corresponding to the functional when it is not necessary, so that the obligation can be declared opaque (e.g. using `Qed`). However, as soon as it appears in the context, the proof of the obligation is *required* to be declared transparent.

No such problems arise when using measures or well-founded recursion.

Program Lemma

A *Lemma* command with the *program* attribute uses the Russell language to type statements of logical properties. It generates obligations, tries to solve them automatically and fails if some unsolved obligations remain. In this case, one can first define the lemma's statement using *Definition* and use it as the goal afterwards. Otherwise the proof will be started with the elaborated version as a goal. The *program* attribute can similarly be used with *Variable*, *Hypothesis*, *Axiom* etc.

Solving obligations

The following commands are available to manipulate obligations. The optional identifier is used when multiple functions have unsolved obligations (e.g. when defining mutually recursive blocks). The optional tactic is replaced by the default one if not specified.

Command: **Obligation Tactic** **:=** *ltac_expr*

Sets the default obligation solving tactic applied to all obligations automatically, whether to solve them or when starting to prove one, e.g. using *Next Obligation*.

This command supports the *local* and *global* attributes. *local* makes the setting last only for the current module. *local* is the default inside sections while *global* otherwise.

Command: **Show Obligation Tactic**

Displays the current default tactic.

Command: **Obligations** **of** *ident* ?

Displays all remaining obligations.

Command: **Obligation** *natural* **of** *ident* ? **: type with** *ltac_expr* ? ?

Start the proof of obligation *natural*.

Command: **Next Obligation** **of** *ident* ? **with** *ltac_expr* ?

Start the proof of the next unsolved obligation.

Command: **Solve Obligations** **of** *ident* ? **with** *ltac_expr* ?

Tries to solve each obligation of *ident* using the given *ltac_expr* or the default one.

Command: **Solve All Obligations** **with** *ltac_expr* ?

Tries to solve each obligation of every program using the given tactic or the default one (useful for mutually recursive definitions).

Command: **Admit Obligations** **of** *ident* ?

Admits all obligations (of *ident*).

Note: Does not work with structurally recursive programs.

Command: **Preterm** **of** *ident*?
Shows the term that will be fed to the kernel once the obligations are solved. Useful for debugging.

Flag: **Transparent Obligations**
This *flag* controls whether all obligations should be declared as transparent (the default), or if the system should infer which obligations can be declared opaque.

The module `Coq.Program.Tactics` defines the default tactic for solving obligations called `program_simpl`. Importing `Coq.Program.Program` also adds some useful notations, as documented in the file itself.

Frequently Asked Questions

Error: **Ill-formed recursive definition.**
This error can happen when one tries to define a function by structural recursion on a subset object, which means the Coq function looks like:

```
Program Fixpoint f (x : A | P) := match x with A b => f b end.
```

Supposing `b : A`, the argument at the recursive call to `f` is not a direct subterm of `x` as `b` is wrapped inside an `exist` constructor to build an object of type `{x : A | P}`. Hence the definition is rejected by the guardedness condition checker. However one can use wellfounded recursion on subset objects like this:

```
Program Fixpoint f (x : A | P) { measure (size x) } :=
  match x with A b => f b end.
```

One will then just have to prove that the measure decreases at each recursive call. There are three drawbacks though:

1. A measure function has to be defined;
2. The reduction is a little more involved, although it works well using lazy evaluation;
3. Mutual recursion on the underlying inductive type isn't possible anymore, but nested mutual recursion is always possible.

2.2.10 Commands

Displaying

Command: **Print** **Term**? *reference* *univ_name_list*?

univ_name_list ::= @{ *name** }

Displays definitions of terms, including opaque terms, for the object ***reference***.

- **Term** - a syntactic marker to allow printing a term that is the same as one of the various **Print** commands. For example, *Print All* is a different command, while **Print Term All** shows information on the object whose name is "**All**".
- ***univ_name_list*** - locally renames the polymorphic universes of ***reference***. The name _ means the usual name is printed.

Error: *qualid* **not a defined object.**

Error: **Universe instance should have length** *natural*.

Error: **This object does not support universe names.**

Command: **Print All**
This command displays information about the current state of the environment, including sections and modules.

Command: **Inspect** *natural*
This command displays the *natural* last objects of the current environment, including sections and modules.

Command: **Print Section** *qualid*
Displays the objects defined since the beginning of the section named *qualid*.

Query commands

Unlike other commands, `query_command`s may be prefixed with a goal selector (*natural*`:`) to specify which goals it applies to. If no selector is provided, the command applies to the current goal. If no proof is open, then the command only applies to accessible objects. (see Section *Invocation of tactics*).

Eval and *Compute* are also *query_command*s, which are described elsewhere

Command: **About** *reference* *univ_name_list* ?
Displays information about the *reference* object, which, if a proof is open, may be a hypothesis of the selected goal, or an accessible theorem, axiom, etc.: its kind (module, constant, assumption, inductive, constructor, abbreviation, …), long name, type, implicit arguments and argument scopes (as set in the definition of *reference* or subsequently with the *Arguments* command). It does not print the body of definitions or proofs.

Command: **Check** *term*
Displays the type of *term*. When called in proof mode, the term is checked in the local context of the selected goal.

Command: **Search** *search_query* + (**inside** | **outside** *qualid* +) ?
This command can be used to filter the goal and the global context to retrieve objects whose name or type satisfies a number of conditions. Library files that were not loaded with *Require* are not considered. The *Search Blacklist* table can also be used to exclude some things from all calls to *Search*.

The output of the command is a list of qualified identifiers and their types. If the *Search Output Name Only* flag is on, the types are omitted.

search_query ::= *search_item*
| - *search_query*
| [*search_query* + |]

Multiple *search_item*s can be combined into a complex *search_query*:

- *search_query* Excludes the objects that would be filtered by *search_query*. See *this example*.

[*search_query* + | ... | *search_query* +] This is a disjunction of conjunctions of queries. A simple conjunction can be expressed by having a single disjunctive branch. For a conjunction at top-level, the surrounding brackets are not required.

search_item ::= head | hyp | concl | headhyp | headconcl ? : *string* ?
% *scope_key* ?
| head | hyp | concl | headhyp | headconcl ? : *one_pattern*
| is : *logical_kind*

Searched objects can be filtered by patterns, by the constants they contain (identified by their name or a notation) and by their names. The location of the pattern or constant within a term

one_pattern Search for objects whose type contains a subterm matching the pattern ***one_pattern***. Holes of the pattern are indicated by _ or **?*ident***. If the same **?*ident*** occurs more than once in the pattern, all occurrences in the subterm must be identical. See *this example*.

string % ***scope_key*** ?

- If ***string*** is a substring of a valid identifier and no % ***scope_key*** is provided, search for objects whose name contains ***string***. See *this example*.
- Otherwise, search for objects whose type contains the reference that this string, interpreted as a notation, is attached to (as described in ***reference***). See *this example*.

Note: To refer to a string used in a notation that is a substring of a valid identifier, put it between single quotes or explicitly provide a scope. See *this example*.

---

hyp: The provided pattern or reference is matched against any subterm of an hypothesis of the type of the objects. See *this example*.

headhyp: The provided pattern or reference is matched against the subterms in head position (any partial applicative subterm) of the hypotheses of the type of the objects. See *the previous example*.

concl: The provided pattern or reference is matched against any subterm of the conclusion of the type of the objects. See *this example*.

headconcl: The provided pattern or reference is matched against the subterms in head position (any partial applicative subterm) of the conclusion of the type of the objects. See *the previous example*.

head: This is simply the union between `headconcl:` and `headhyp:`.

is: ***logical_kind***

logical_kind ::= *thm_token* | *assumption_token*
| Definition | Example | Context | Primitive
| Coercion | Instance | Scheme | Canonical | SubClass
| Field | Method

Filters objects by the keyword that was used to define them (`Theorem`, `Lemma`, `Axiom`, `Variable`, `Context`, `Primitive`...) or its status (`Coercion`, `Instance`, `Scheme`, `Canonical`, `SubClass`, Field' for record fields, `Method` for class fields). Note that `Coercions`, `Canonical Structures`, Instance's and `Schemes` can be defined without using those keywords. See *this example*.

Additional clauses:

- **inside** ***qualid*** + - limit the search to the specified modules
- **outside** ***qualid*** + - exclude the specified modules from the search

Error: Module/section *qualid* not found.
There is no constant in the environment named *qualid*, where *qualid* is in an `inside` or `outside` clause.

Example: Searching for a pattern

We can repeat meta-variables to narrow down the search. Here, we are looking for commutativity lemmas.

```
Search (_ ?n ?m = _ ?m ?n).
    Nat.land_comm: forall a b : nat, Nat.land a b = Nat.land b a
    Nat.lor_comm: forall a b : nat, Nat.lor a b = Nat.lor b a
    Nat.lxor_comm: forall a b : nat, Nat.lxor a b = Nat.lxor b a
    Nat.lcm_comm: forall a b : nat, Nat.lcm a b = Nat.lcm b a
    Nat.min_comm: forall n m : nat, Nat.min n m = Nat.min m n
    Nat.gcd_comm: forall n m : nat, Nat.gcd n m = Nat.gcd m n
    Bool.xorb_comm: forall b b' : bool, xorb b b' = xorb b' b
    Nat.max_comm: forall n m : nat, Nat.max n m = Nat.max m n
    Nat.mul_comm: forall n m : nat, n * m = m * n
    Nat.add_comm: forall n m : nat, n + m = m + n
    Bool.orb_comm: forall b1 b2 : bool, (b1 || b2)%bool = (b2 || b1)%bool
    Bool.andb_comm: forall b1 b2 : bool, (b1 && b2)%bool = (b2 && b1)%bool
    Nat.eqb_sym: forall x y : nat, (x =? y) = (y =? x)
```

Example: Searching for part of an identifier

```
Search "_assoc".
    or_assoc: forall A B C : Prop, (A \/ B) \/ C <-> A \/ B \/ C
    and_assoc: forall A B C : Prop, (A /\ B) /\ C <-> A /\ B /\ C
    eq_trans_assoc:
      forall [A : Type] [x y z t : A] (e : x = y) (e' : y = z) (e'' : z = t),
      eq_trans e (eq_trans e' e'') = eq_trans (eq_trans e e') e''
```

Example: Searching for a reference by notation

```
Search "+".
    plus_O_n: forall n : nat, 0 + n = n
    plus_n_O: forall n : nat, n = n + 0
    plus_n_Sm: forall n m : nat, S (n + m) = n + S m
    plus_Sn_m: forall n m : nat, S n + m = S (n + m)
    mult_n_Sm: forall n m : nat, n * m + n = n * S m
    f_equal2_plus:
      forall x1 y1 x2 y2 : nat, x1 = y1 -> x2 = y2 -> x1 + x2 = y1 + y2
    nat_rect_plus:
      forall (n m : nat) {A : Type} (f : A -> A) (x : A),
      nat_rect (fun _ : nat => A) x (fun _ : nat => f) (n + m) =
      nat_rect (fun _ : nat => A)
        (nat_rect (fun _ : nat => A) x (fun _ : nat => f) m)
        (fun _ : nat => f) n
```

Example: Disambiguating between part of identifier and notation

In this example, we show two ways of searching for all the objects whose type contains `Nat.modulo` but which

do not contain the substring ”mod”.

```
Search "'mod'" -"mod".
    Nat.bit0_eqb: forall a : nat, Nat.testbit a 0 = (a mod 2 =? 1)
    Nat.land_ones: forall a n : nat, Nat.land a (Nat.ones n) = a mod 2 ^ n
    Nat.div_exact: forall a b : nat, b <> 0 -> a = b * (a / b) <-> a mod b = 0
    Nat.testbit_spec':
      forall a n : nat, Nat.b2n (Nat.testbit a n) = (a / 2 ^ n) mod 2
    Nat.pow_div_l:
      forall a b c : nat, b <> 0 -> a mod b = 0 -> (a / b) ^ c = a ^ c / b ^ c
    Nat.testbit_eqb: forall a n : nat, Nat.testbit a n = ((a / 2 ^ n) mod 2 =? 1)
    Nat.testbit_false:
      forall a n : nat, Nat.testbit a n = false <-> (a / 2 ^ n) mod 2 = 0
    Nat.testbit_true:
      forall a n : nat, Nat.testbit a n = true <-> (a / 2 ^ n) mod 2 = 1

Search "mod"%nat -"mod".
    Nat.bit0_eqb: forall a : nat, Nat.testbit a 0 = (a mod 2 =? 1)
    Nat.land_ones: forall a n : nat, Nat.land a (Nat.ones n) = a mod 2 ^ n
    Nat.div_exact: forall a b : nat, b <> 0 -> a = b * (a / b) <-> a mod b = 0
    Nat.testbit_spec':
      forall a n : nat, Nat.b2n (Nat.testbit a n) = (a / 2 ^ n) mod 2
    Nat.pow_div_l:
      forall a b c : nat, b <> 0 -> a mod b = 0 -> (a / b) ^ c = a ^ c / b ^ c
    Nat.testbit_eqb: forall a n : nat, Nat.testbit a n = ((a / 2 ^ n) mod 2 =? 1)
    Nat.testbit_false:
      forall a n : nat, Nat.testbit a n = false <-> (a / 2 ^ n) mod 2 = 0
    Nat.testbit_true:
      forall a n : nat, Nat.testbit a n = true <-> (a / 2 ^ n) mod 2 = 1
```

Example: Search in hypotheses

The following search shows the objects whose type contains `bool` in an hypothesis as a strict subterm only:

```
Search hyp:bool -headhyp:bool.
    Nat.bitwise: (bool -> bool -> bool) -> nat -> nat -> nat -> nat
    Byte.of_bits:
      bool * (bool * (bool * (bool * (bool * (bool * (bool * bool)))))) ->
      Byte.byte
    Byte.to_bits_of_bits:
      forall
        b : bool * (bool * (bool * (bool * (bool * (bool * (bool * bool)))))),
      Byte.to_bits (Byte.of_bits b) = b
```

Example: Search in conclusion

The following search shows the objects whose type contains `bool` in the conclusion as a strict subterm only:

```
Search concl:bool -headconcl:bool.
    Byte.to_bits:
      Byte.byte ->
      bool * (bool * (bool * (bool * (bool * (bool * (bool * bool))))))
    andb_prop: forall a b : bool, (a && b)%bool = true -> a = true /\ b = true
    andb_true_intro:
      forall [b1 b2 : bool], b1 = true /\ b2 = true -> (b1 && b2)%bool = true
```

(continues on next page)

(continued from previous page)

```
    Byte.to_bits_of_bits:
      forall
        b : bool * (bool * (bool * (bool * (bool * (bool * (bool * bool))))))),
      Byte.to_bits (Byte.of_bits b) = b
    bool_choice:
      forall [S : Set] [R1 R2 : S -> Prop],
      (forall x : S, {R1 x} + {R2 x}) ->
      {f : S -> bool | forall x : S, f x = true /\ R1 x \/ f x = false /\ R2 x}
```

Example: Search by keyword or status

The following search shows the definitions whose type is a `nat` or a function which returns a `nat` and the lemmas about +:

```
Search [ is:Definition headconcl:nat | is:Lemma (_ + _) ].
    Nat.two: nat
    Nat.zero: nat
    Nat.one: nat
    Nat.succ: nat -> nat
    Nat.log2: nat -> nat
    Nat.sqrt: nat -> nat
    Nat.square: nat -> nat
    Nat.double: nat -> nat
    Nat.pred: nat -> nat
    Nat.ldiff: nat -> nat -> nat
    Nat.tail_mul: nat -> nat -> nat
    Nat.land: nat -> nat -> nat
    Nat.div: nat -> nat -> nat
    Nat.modulo: nat -> nat -> nat
    Nat.lor: nat -> nat -> nat
    Nat.lxor: nat -> nat -> nat
    Nat.of_hex_uint: Hexadecimal.uint -> nat
    Nat.of_uint: Decimal.uint -> nat
    Nat.of_num_uint: Number.uint -> nat
    length: forall [A : Type], list A -> nat
    plus_n_O: forall n : nat, n = n + 0
    plus_O_n: forall n : nat, 0 + n = n
    plus_n_Sm: forall n m : nat, S (n + m) = n + S m
    plus_Sn_m: forall n m : nat, S n + m = S (n + m)
    mult_n_Sm: forall n m : nat, n * m + n = n * S m
```

The following search shows the instances whose type includes the classes `Reflexive` or `Symmetric`:

```
Search is:Instance [ Reflexive | Symmetric ].
    iff_Symmetric: Symmetric iff
    iff_Reflexive: Reflexive iff
    impl_Reflexive: Reflexive Basics.impl
    eq_Symmetric: forall {A : Type}, Symmetric eq
    eq_Reflexive: forall {A : Type}, Reflexive eq
    Equivalence_Symmetric:
      forall {A : Type} {R : Relation_Definitions.relation A},
      Equivalence R -> Symmetric R
    Equivalence_Reflexive:
      forall {A : Type} {R : Relation_Definitions.relation A},
      Equivalence R -> Reflexive R
```

(continues on next page)

(continued from previous page)

```
    PER_Symmetric:
      forall {A : Type} {R : Relation_Definitions.relation A},
      PER R -> Symmetric R
    PreOrder_Reflexive:
      forall {A : Type} {R : Relation_Definitions.relation A},
      PreOrder R -> Reflexive R
    reflexive_eq_dom_reflexive:
      forall {A B : Type} {R' : Relation_Definitions.relation B},
      Reflexive R' -> Reflexive (eq ==> R')%signature
```

Command: **SearchPattern** *one_pattern* [**inside** | **outside** *qualid*]+ ?

Displays the name and type of all hypotheses of the selected goal (if any) and theorems of the current context ending with [**forall** *binder* * ,]? [$\mathbf{P_i}$ **->**]* **C** that match the pattern *one_pattern*.

See *Search* for an explanation of the `inside`/`outside` clauses.

Example: **SearchPattern examples**

```
Require Import Arith.

SearchPattern (_ + _ = _ + _).
    Nat.add_comm: forall n m : nat, n + m = m + n
    plus_Snm_nSm: forall n m : nat, S n + m = n + S m
    Nat.add_succ_comm: forall n m : nat, S n + m = n + S m
    Nat.add_shuffle3: forall n m p : nat, n + (m + p) = m + (n + p)
    plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
    Nat.add_assoc: forall n m p : nat, n + (m + p) = n + m + p
    Nat.add_shuffle0: forall n m p : nat, n + m + p = n + p + m
    f_equal2_plus:
      forall x1 y1 x2 y2 : nat, x1 = y1 -> x2 = y2 -> x1 + x2 = y1 + y2
    Nat.add_shuffle2: forall n m p q : nat, n + m + (p + q) = n + q + (m + p)
    Nat.add_shuffle1: forall n m p q : nat, n + m + (p + q) = n + p + (m + q)

SearchPattern (nat -> bool).
    Nat.odd: nat -> bool
    Init.Nat.odd: nat -> bool
    Nat.even: nat -> bool
    Init.Nat.even: nat -> bool
    Init.Nat.testbit: nat -> nat -> bool
    Nat.leb: nat -> nat -> bool
    Nat.eqb: nat -> nat -> bool
    Init.Nat.eqb: nat -> nat -> bool
    Nat.ltb: nat -> nat -> bool
    Nat.testbit: nat -> nat -> bool
    Init.Nat.leb: nat -> nat -> bool
    Init.Nat.ltb: nat -> nat -> bool
    BinNat.N.testbit_nat: BinNums.N -> nat -> bool
    BinPosDef.Pos.testbit_nat: BinNums.positive -> nat -> bool
    BinPos.Pos.testbit_nat: BinNums.positive -> nat -> bool
    BinNatDef.N.testbit_nat: BinNums.N -> nat -> bool

SearchPattern (forall l : list _, _ l l).
```

(continues on next page)

(continued from previous page)

```
    List.incl_refl: forall [A : Type] (l : list A), List.incl l l
    List.lel_refl: forall [A : Type] (l : list A), List.lel l l

SearchPattern (?X1 + _ = _ + ?X1).
    Nat.add_comm: forall n m : nat, n + m = m + n
```

Command: **SearchRewrite** *one_pattern* [**inside** | **outside** *qualid*]+ ?

Displays the name and type of all hypotheses of the selected goal (if any) and theorems of the current context that have the form **forall** *binder* * , ? **P**$_i$ -> * **LHS = RHS** where *one_pattern* matches either `LHS` or `RHS`.

See *Search* for an explanation of the `inside`/`outside` clauses.

Example: **SearchRewrite examples**

```
Require Import Arith.

SearchRewrite (_ + _ + _).
    Nat.add_shuffle0: forall n m p : nat, n + m + p = n + p + m
    plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
    Nat.add_assoc: forall n m p : nat, n + (m + p) = n + m + p
    Nat.add_shuffle1: forall n m p q : nat, n + m + (p + q) = n + p + (m + q)
    Nat.add_shuffle2: forall n m p q : nat, n + m + (p + q) = n + q + (m + p)
    Nat.add_carry_div2:
      forall (a b : nat) (c0 : bool),
      (a + b + Nat.b2n c0) / 2 =
      a / 2 + b / 2 +
      Nat.b2n
        (Nat.testbit a 0 && Nat.testbit b 0
         || c0 && (Nat.testbit a 0 || Nat.testbit b 0))
```

Table: **Search Blacklist** *string*

This *table* specifies a set of strings used to exclude lemmas from the results of *Search*, *SearchPattern* and *SearchRewrite* queries. A lemma whose fully-qualified name contains any of the strings will be excluded from the search results. The default blacklisted substrings are `_subterm`, `_subproof` and `Private_`.

Use the *Add* and *Remove* commands to update the set of blacklisted strings.

Flag: **Search Output Name Only**

This *flag* restricts the output of search commands to identifier names; turning it on causes invocations of *Search*, *SearchPattern*, *SearchRewrite* etc. to omit types from their output, printing only identifiers.

Requests to the environment

Command: **`Print Assumptions`** ***reference***
Displays all the assumptions (axioms, parameters and variables) a theorem or definition depends on.

The message ”Closed under the global context” indicates that the theorem or definition has no dependencies.

Command: **`Print Opaque Dependencies`** ***reference***
Displays the assumptions and opaque constants that ***reference*** depends on.

Command: **`Print Transparent Dependencies`** ***reference***
Displays the assumptions and transparent constants that ***reference*** depends on.

Command: **`Print All Dependencies`** ***reference***
Displays all the assumptions and constants ***reference*** depends on.

Command: **`Locate`** ***reference***

reference ::= *qualid*
| *string* `%` *scope_key* ?

Displays the full name of objects from Coq's various qualified namespaces such as terms, modules and Ltac, thereby showing the module they are defined in. It also displays notation definitions.

Note that objects are reported only when the module containing them has been loaded, such as through a `Require` command. Notation definitions are reported only when the containing module has been imported (e.g. with `Require Import` or `Import`).

qualid refers to object names that end with ***qualid***.

string `%` ***scope_key*** ? refers to definitions of notations. ***string*** can be a single token in the notation such as ”->” or a pattern that matches the notation. See *Locating notations*.

`%` ***scope_key***, if present, limits the reference to the scope bound to the delimiting key ***scope_key***, such as, for example, **`%nat`**. (see Section *Local interpretation rules for notations*)

Command: **`Locate Term`** ***reference***
Like `Locate`, but limits the search to terms

Command: **`Locate Module`** ***qualid***
Like `Locate`, but limits the search to modules

Command: **`Locate Ltac`** ***qualid***
Like `Locate`, but limits the search to tactics

Command: **`Locate Library`** ***qualid***
Displays the full name, status and file system path of the module ***qualid***, whether loaded or not.

Command: **`Locate File`** ***string***
Displays the file system path of the file ending with ***string***. Typically, ***string*** has a suffix such as `.cmo` or `.vo` or `.v` file, such as **`Nat.v`**.

Example: Locate examples

```
Locate nat.
    Inductive Coq.Init.Datatypes.nat

Locate Datatypes.O.
```

(continues on next page)

(continued from previous page)

```
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

Locate Init.Datatypes.O.
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

Locate Coq.Init.Datatypes.O.
    Constructor Coq.Init.Datatypes.O
      (shorter name to refer to it in current context is O)

Locate I.Dont.Exist.
    No object of suffix I.Dont.Exist
```

Printing flags

Flag: `Fast Name Printing`
When this *flag* is turned on, Coq uses an asymptotically faster algorithm for the generation of unambiguous names of bound variables while printing terms. While faster, it is also less clever and results in a typically less elegant display, e.g. it will generate more names rather than reusing certain names across subterms. This flag is not enabled by default, because as Ltac observes bound names, turning it on can break existing proof scripts.

Loading files

Coq offers the possibility of loading different parts of a whole development stored in separate files. Their contents will be loaded as if they were entered from the keyboard. This means that the loaded files are text files containing sequences of commands for Coq's toplevel. This kind of file is called a *script* for Coq. The standard (and default) extension of Coq's script files is .v.

Command: `Load` `Verbose`? *string* | *ident*
Loads a file. If *ident* is specified, the command loads a file named ***ident*`.v`**, searching successively in each of the directories specified in the *loadpath*. (see Section *Libraries and filesystem*)

If *string* is specified, it must specify a complete filename. `~` and `..` abbreviations are allowed as well as shell variables. If no extension is specified, Coq will use the default extension `.v`.

Files loaded this way can't leave proofs open, nor can `Load` be used inside a proof.

We discourage the use of `Load`; use `Require` instead. `Require` loads `.vo` files that were previously compiled from `.v` files.

`Verbose` displays the Coq output for each command and tactic in the loaded file, as if the commands and tactics were entered interactively.

Error: `Can't find file` *ident* `on loadpath.`

Error: `Load is not supported inside proofs.`

Error: `Files processed by Load cannot leave open proofs.`

Compiled files

This section describes the commands used to load compiled files (see Chapter *The Coq commands* for documentation on how to compile a file). A compiled file is a particular case of a module called a *library file*.

Command: `Require` (`Import` | `Export`)? *qualid*+

Loads compiled modules into the Coq environment. For each *qualid*, which has the form *ident*$_{prefix}$ `.`* *ident*, the command searches for the logical name represented by the *ident*$_{prefix}$s and loads the compiled file *ident*`.vo` from the associated filesystem directory.

The process is applied recursively to all the loaded files; if they contain `Require` commands, those commands are executed as well. The compiled files must have been compiled with the same version of Coq. The compiled files are neither replayed nor rechecked.

- **`Import`** - additionally does an `Import` on the loaded module, making components defined in the module available by their short names
- **`Export`** - additionally does an `Export` on the loaded module, making components defined in the module available by their short names *and* marking them to be exported by the current module

If the required module has already been loaded, **`Import`** and **`Export`** make the command equivalent to `Import` or `Export`.

The loadpath must contain the same mapping used to compile the file (see Section *Libraries and filesystem*). If several files match, one of them is picked in an unspecified fashion. Therefore, library authors should use a unique name for each module and users are encouraged to use fully-qualified names or the `From … Require` command to load files.

Command: `From` *dirpath* `Require` (`Import` | `Export`)? *qualid*+

Works like `Require`, but loads, for each *qualid*, the library whose fully-qualified name matches *dirpath*`.`*ident* `.`* *qualid* for some *ident* `.`* . This is useful to ensure that the *qualid* library comes from a particular package.

Error: `Cannot load` *qualid*`: no physical path bound to` *dirpath*`.`

Error: `Cannot find library foo in loadpath.`

The command did not find the file foo.vo. Either foo.v exists but is not compiled or foo.vo is in a directory which is not in your LoadPath (see Section *Libraries and filesystem*).

Error: `Compiled library` *ident*`.vo makes inconsistent assumptions over library` *qualid*`.`

The command tried to load library file *ident*.vo that depends on some specific version of library *qualid* which is not the one already loaded in the current Coq session. Probably *ident*`.v` was not properly recompiled with the last version of the file containing module *qualid*.

Error: `Bad magic number.`

The file *ident*`.vo` was found but either it is not a Coq compiled module, or it was compiled with an incompatible version of Coq.

Error: `The file` *ident*`.vo contains library` *qualid*$_1$ `and not library` *qualid*$_2$`.`

The library *qualid*$_2$ is indirectly required by a `Require` or `From … Require` command. The loadpath maps *qualid*$_2$ to *ident*`.vo`, which was compiled using a loadpath that bound it to *qualid*$_1$. Usually the appropriate solution is to recompile *ident*`.v` using the correct loadpath. See *Libraries and filesystem*.

Warning: `Require inside a module is deprecated and strongly discouraged. You can Require a module at toplevel and optionally Import it inside another one.`

Note that the `Import` and `Export` commands can be used inside modules.

See also:

Chapter *The Coq commands*

Command: **`Print Libraries`**
This command displays the list of library files loaded in the current Coq session.

Command: **`Declare ML Module`** *`string`*`+`
This commands dynamically loads OCaml compiled code from a **`.mllib`** file. It is used to load plugins dynamically. The files must be accessible in the current OCaml loadpath (see *command line option* **`-I`** and command `Add ML Path`). The **`.mllib`** suffix may be omitted.

This command is reserved for plugin developers, who should provide a .v file containing the command. Users of the plugins will then generally load the .v file.

This command supports the `local` attribute. If present, the listed files are not exported, even if they're outside a section.

Error: **`File not found on loadpath:`** *`string`*`.`

Command: **`Print ML Modules`**
This prints the name of all OCaml modules loaded with `Declare ML Module`. To know from where these module were loaded, the user should use the command `Locate File`.

Loadpath

Loadpaths are preferably managed using Coq command line options (see Section *Libraries and filesystem*), but there are also commands to manage them within Coq. These commands are only meant to be issued in the toplevel, and using them in source files is discouraged.

Command: **`Pwd`**
This command displays the current working directory.

Command: **`Cd`** *`string`*`?`
If *`string`* is specified, changes the current directory according to `string` which can be any valid path. Otherwise, it displays the current directory.

Command: **`Add LoadPath`** *`string`* **`as`** *`dirpath`*

dirpath `::=` *`ident.`*`*` *ident*

This command is equivalent to the command line option **`-Q`** *`string dirpath`*. It adds a mapping to the loadpath from the logical name *`dirpath`* to the file system directory *`string`*.

- *`dirpath`* is a prefix of a module name. The module name hierarchy follows the file system hierarchy. On Linux, for example, the prefix `A.B.C` maps to the directory *`string`***`/B/C`**. Avoid using spaces after a **`.`** in the path because the parser will interpret that as the end of a command or tactic.

Command: **`Add Rec LoadPath`** *`string`* **`as`** *`dirpath`*
This command is equivalent to the command line option **`-R`** *`string dirpath`*. It adds the physical directory string and all its subdirectories to the current Coq loadpath.

Command: **`Remove LoadPath`** *`string`*
This command removes the path *`string`* from the current Coq loadpath.

Command: **`Print LoadPath`** *`dirpath`*`?`
This command displays the current Coq loadpath. If *`dirpath`* is specified, displays only the paths that extend that prefix.

Command: Add ML Path *string*
Equivalent to the *command line option* **-I *string***. Adds the path ***string*** to the current OCaml loadpath (cf. *Declare ML Module*). It is for convenience, such as for use in an interactive session, and it is not exported to compiled files. For separation of concerns with respect to the relocability of files, we recommend using **-I *string***.

Command: Print ML Path
Displays the current OCaml loadpath, as provided by the *command line option* **-I *string*** or by the command *Add ML Path* `@string` (cf. *Declare ML Module*).

Backtracking

The backtracking commands described in this section can only be used interactively, they cannot be part of a Coq file loaded via `Load` or compiled by `coqc`.

Command: Reset *ident*
This command removes all the objects in the environment since ***ident*** was introduced, including ***ident***. ***ident*** may be the name of a defined or declared object as well as the name of a section. One cannot reset over the name of a module or of an object inside a module.

Error: *ident*: no such entry.

Command: Reset Initial
Goes back to the initial state, just after the start of the interactive session.

Command: Back ***natural***?
Undoes all the effects of the last ***natural sentence***s. If ***natural*** is not specified, the command undoes one sentence. Sentences read from a `.v` file via a *Load* are considered a single sentence. While *Back* can undo tactics and commands executed within proof mode, once you exit proof mode, such as with *Qed*, all the statements executed within are thereafter considered a single sentence. *Back* immediately following *Qed* gets you back to the state just after the statement of the proof.

Error: Invalid backtrack.
The user wants to undo more commands than available in the history.

Command: BackTo *natural*
This command brings back the system to the state labeled ***natural***, forgetting the effect of all commands executed after this state. The state label is an integer which grows after each successful command. It is displayed in the prompt when in -emacs mode. Just as *Back* (see above), the *BackTo* command now handles proof states. For that, it may have to undo some extra commands and end on a state ***natural′ ≤ natural*** if necessary.

Quitting and debugging

Command: Quit
Causes Coq to exit. Valid only in coqtop.

Command: Drop
This command temporarily enters the OCaml toplevel. It is a debug facility used by Coq's implementers. Valid only in the bytecode version of coqtop. The OCaml command:

```
#use "include";;
```

adds the right loadpaths and loads some toplevel printers for all abstract types of Coq- section_path, identifiers, terms, judgments, ... You can also use the file base_include instead, that loads only the pretty-printers for section_paths and identifiers. You can return back to Coq with the command:

```
go();;
```

> **Warning:**
>
> 1. It only works with the bytecode version of Coq (i.e. `coqtop.byte`, see Section `interactive-use`).
> 2. You must have compiled Coq from the source package and set the environment variable COQTOP to the root of your copy of the sources (see Section `customization-by-environment-variables`).

Command: `Time` *`sentence`*
Executes ***`sentence`*** and displays the time needed to execute it.

Command: `Redirect` *`string sentence`*
Executes ***`sentence`***, redirecting its output to the file "***`string`***.out".

Command: `Timeout` *`natural sentence`*
Executes ***`sentence`***. If the operation has not terminated after ***`natural`*** seconds, then it is interrupted and an error message is displayed.

Option: `Default Timeout` *`natural`*
When this *option* is set, each ***`sentence`*** is treated as if it was prefixed with *`Timeout`* ***`natural`***, except for *`Timeout`* commands themselves. If unset, no timeout is applied.

Command: `Fail` *`sentence`*
For debugging scripts, sometimes it is desirable to know whether a command or a tactic fails. If ***`sentence`*** fails, then **`Fail`** ***`sentence`*** succeeds (except for critical errors, such as "stack overflow"), without changing the proof state. In interactive mode, the system prints a message confirming the failure.

Error: `The command has not failed!`
If the given ***`command`*** succeeds, then **`Fail`** ***`sentence`*** fails with this error message.

Note: *`Time`*, *`Redirect`*, *`Timeout`* and *`Fail`* are `control_commands`. For these commands, attributes and goal selectors, when specified, are part of the ***`sentence`*** argument, and thus come after the control command prefix and before the inner command or tactic. For example: `Time #[ local ] Definition foo := 0.` or `Fail Timeout 10 all: auto.`

Controlling display

Flag: `Silent`
This *flag* controls the normal displaying.

Option: `Warnings` "`[ - | + ? ] ident [ + , ]`"
This *option* configures the display of warnings. It is experimental, and expects, between quotes, a comma-separated list of warning names or categories. Adding - in front of a warning or category disables it, adding + makes it an error. It is possible to use the special categories all and default, the latter containing the warnings enabled by default. The flags are interpreted from left to right, so in case of an overlap, the flags on the right have higher priority, meaning that `A,-A` is equivalent to `-A`.

Option: `Debug` "`[ - ? ] ident [ + , ]`"
This *option* configures the display of debug messages. Each ***`ident`*** enables debug messages for that compo-

nent, while **-*ident*** disables messages for the component. `all` activates or deactivates all other components. `backtrace` controls printing of error backtraces.

`Test` `Debug` displays the list of components and their enabled/disabled state.

Option: `Printing Width` *`natural`*
This *option* sets which left-aligned part of the width of the screen is used for display. At the time of writing this documentation, the default value is 78.

Option: `Printing Depth` *`natural`*
This *option* controls the nesting depth of the formatter used for pretty- printing. Beyond this depth, display of subterms is replaced by dots. At the time of writing this documentation, the default value is 50.

Flag: `Printing Compact Contexts`
This *flag* controls the compact display mode for goals contexts. When on, the printer tries to reduce the vertical size of goals contexts by putting several variables (even if of different types) on the same line provided it does not exceed the printing width (see *`Printing Width`*). At the time of writing this documentation, it is off by default.

Flag: `Printing Unfocused`
This *flag* controls whether unfocused goals are displayed. Such goals are created by focusing other goals with bullets (see *Bullets* or *curly braces*). It is off by default.

Flag: `Printing Dependent Evars Line`
This *flag* controls the printing of the "(dependent evars: …)" information after each tactic. The information is used by the Prooftree tool in Proof General. (https://askra.de/software/prooftree)

Printing constructions in full

Flag: `Printing All`
Coercions, implicit arguments, the type of pattern matching, but also notations (see *Syntax extensions and notation scopes*) can obfuscate the behavior of some tactics (typically the tactics applying to occurrences of subterms are sensitive to the implicit arguments). Turning this *flag* on deactivates all high-level printing features such as coercions, implicit arguments, returned type of pattern matching, notations and various syntactic sugar for pattern matching or record projections. Otherwise said, *`Printing All`* includes the effects of the flags *`Printing Implicit`*, *`Printing Coercions`*, *`Printing Synth`*, *`Printing Projections`*, and *`Printing Notations`*. To reactivate the high-level printing features, use the command `Unset Printing All`.

Note: In some cases, setting *`Printing All`* may display terms that are so big they become very hard to read. One technique to work around this is use *`Undelimit Scope`* and/or *`Close Scope`* to turn off the printing of notations bound to particular scope(s). This can be useful when notations in a given scope are getting in the way of understanding a goal, but turning off all notations with *`Printing All`* would make the goal unreadable.

Controlling Typing Flags

Flag: `Guard Checking`
This *flag* can be used to enable/disable the guard checking of fixpoints. Warning: this can break the consistency of the system, use at your own risk. Decreasing argument can still be specified: the decrease is not checked anymore but it still affects the reduction of the term. Unchecked fixpoints are printed by *`Print Assumptions`*.

Attribute: `bypass_check(guard` `=` `yes` | `no` `)` ?
This *boolean attribute* is similar to the *`Guard Checking`* flag, but on a per-declaration basis. Disable guard checking locally with `bypass_check(guard)`.

Flag: `Positivity Checking`
This *flag* can be used to enable/disable the positivity checking of inductive types and the productivity checking of coinductive types. Warning: this can break the consistency of the system, use at your own risk. Unchecked (co)inductive types are printed by `Print Assumptions`.

Attribute: `bypass_check(positivity = yes | no ?)`
This *boolean attribute* is similar to the `Positivity Checking` flag, but on a per-declaration basis. Disable positivity checking locally with `bypass_check(positivity)`.

Flag: `Universe Checking`
This *flag* can be used to enable/disable the checking of universes, providing a form of "type in type". Warning: this breaks the consistency of the system, use at your own risk. Constants relying on "type in type" are printed by `Print Assumptions`. It has the same effect as `-type-in-type` command line argument (see *By command line options*).

Attribute: `bypass_check(universes = yes | no ?)`
This *boolean attribute* is similar to the `Universe Checking` flag, but on a per-declaration basis. Disable universe checking locally with `bypass_check(universes)`.

Command: `Print Typing Flags`
Print the status of the three typing flags: guard checking, positivity checking and universe checking.

See also `Cumulative StrictProp` in the SProp chapter.

Example

```
Unset Guard Checking.

Print Typing Flags.
    check_guarded: false
    check_positive: true
    check_universes: true
    cumulative sprop: false
    definitional uip: false


Fixpoint f (n : nat) : False
  := f n.
    f is defined
    f is recursively defined (guarded on 1st argument)


Fixpoint ackermann (m n : nat) {struct m} : nat :=
  match m with
  | 0 => S n
  | S m =>
    match n with
    | 0 => ackermann m 1
    | S n => ackermann m (ackermann (S m) n)
    end
  end.
    ackermann is defined
    ackermann is recursively defined (guarded on 1st argument)


Print Assumptions ackermann.
```

(continues on next page)

(continued from previous page)

```
Axioms:
ackermann is assumed to be guarded.
```

Note that the proper way to define the Ackermann function is to use an inner fixpoint:

```
Fixpoint ack m :=
  fix ackm n :=
  match m with
  | 0 => S n
  | S m' =>
    match n with
    | 0 => ack m' 1
    | S n' => ack m' (ackm n')
    end
  end.
ack is defined
ack is recursively defined (guarded on 1st argument)
```

Typing flags may not be changed while inside sections.

Internal registration commands

Due to their internal nature, the commands that are presented in this section are not for general use. They are meant to appear only in standard libraries and in support libraries of plug-ins.

Exposing constants to OCaml libraries

Command: **Register** *qualid*$_1$ **as** *qualid*$_2$

Makes the constant *qualid*$_1$ accessible to OCaml libraries under the name *qualid*$_2$. The constant can then be dynamically located in OCaml code by calling **Coqlib.lib_ref** "*qualid*$_2$". The OCaml code doesn't need to know where the constant is defined (what file, module, library, etc.).

As a special case, when the first segment of *qualid*$_2$ is `kernel`, the constant is exposed to the kernel. For instance, the `PrimInt63` module features the following declaration:

```
Register bool as kernel.ind_bool.
```

This makes the kernel aware of the `bool` type, which is used, for example, to define the return type of the `#int63_eq` primitive.

See also:

Primitive Integers

Inlining hints for the fast reduction machines

Command: **Register Inline** *qualid*

Gives a hint to the reduction machines (VM and native) that the body of the constant *qualid* should be inlined in the generated code.

Registering primitive operations

Command: **Primitive** *ident_decl* [**:** *term*]? **:=** **#***ident*

Makes the primitive type or primitive operator **#***ident* defined in OCaml accessible in Coq commands and tactics. For internal use by implementors of Coq's standard library or standard library replacements. No space is allowed after the `#`. Invalid values give a syntax error.

For example, the standard library files `PrimInt63.v` and `PrimFloat.v` use *Primitive* to support, respectively, the features described in *Primitive Integers* and *Primitive Floats*.

The types associated with an operator must be declared to the kernel before declaring operations that use the type. Do this with *Primitive* for primitive types and *Register* with the `kernel` prefix for other types. For example, in `PrimInt63.v`, `#int63_type` must be declared before the associated operations.

Error:
The type *ident* **must be registered before this construction can be typechecked.**

The type must be defined with *Primitive* command before this *Primitive* command (declaring an operation using the type) will succeed.

CHAPTER

THREE

PROOFS

3.1 Basic proof writing

Coq is an interactive theorem prover, or proof assistant, which means that proofs can be constructed interactively through a dialog between the user and the assistant. The building blocks for this dialog are tactics which the user will use to represent steps in the proof of a theorem.

The first section presents the proof mode (the core mechanism of the dialog between the user and the proof assistant). Then, several sections describe the available tactics. The last section covers the SSReflect proof language, which provides a consistent alternative set of tactics to the standard basic tactics.

Additional tactics are documented in the next chapter *Automatic solvers and programmable tactics*.

3.1.1 Proof mode

Proof mode is used to prove theorems. Coq enters proof mode when you begin a proof, such as with the `Theorem` command. It exits proof mode when you complete a proof, such as with the `Qed` command. Tactics, which are available only in proof mode, incrementally transform incomplete proofs to eventually generate a complete proof.

When you run Coq interactively, such as through CoqIDE, Proof General or coqtop, Coq shows the current proof state (the incomplete proof) as you enter tactics. This information isn't shown when you run Coq in batch mode with `coqc`.

Proof State

The proof state consists of one or more unproven goals. Each goal has a conclusion (the statement that is to be proven) and a local context, which contains named *hypotheses* (which are propositions), variables and local definitions that can be used in proving the conclusion. The proof may also use *constants* from the *global environment* such as definitions and proven theorems.

The term "goal" may refer to an entire goal or to the conclusion of a goal, depending on the context.

The conclusion appears below a line and the local context appears above the line. The conclusion is a type. Each item in the local context begins with a name and ends, after a colon, with an associated type. Local definitions are shown in the form `n := 0 : nat`, for example, in which `nat` is the type of `0`.

The local context of a goal contains items specific to the goal as well as section-local variables and hypotheses (see *Assumptions*) defined in the current *section*. The latter are included in the initial proof state. Items in the local context are ordered; an item can only refer to items that appear before it. (A more mathematical description of the *local context* is *here*.)

The global environment has definitions and proven theorems that are global in scope. (A more mathematical description of the *global environment* is *here*.)

When you begin proving a theorem, the proof state shows the statement of the theorem below the line and often nothing in the local context:

```
1 goal

  ============================
  forall n m : nat, n > m -> P 1 /\ P 2
```

After applying the *intros tactic*, we see hypotheses above the line. The names of variables (`n` and `m`) and hypotheses (`H`) appear before a colon, followed by the type they represent.

```
intros.
1 goal

  n, m : nat
  H : n > m
  ============================
  P 1 /\ P 2
```

Some tactics, such as *split*, create new goals, which may be referred to as subgoals for clarity. Goals are numbered from 1 to N at each step of the proof to permit applying a tactic to specific goals. The local context is only shown for the first goal.

```
split.
2 goals

  n, m : nat
  H : n > m
  ============================
  P 1

goal 2 is:
 P 2
```

"Variables" may refer specifically to local context items for which the type of their type is `Set` or `Type`, and "hypotheses" refers to items that are *propositions*, for which the type of their type is `Prop` or `SProp`, but these terms are also used interchangeably.

```
type of n : nat
type of nat : Set

type of H : (n > m)
type of (n > m) : Prop
```

A proof script, consisting of the tactics that are applied to prove a theorem, is often informally referred to as a "proof". The real proof, whether complete or incomplete, is a term, the proof term, which users may occasionally want to examine. (This is based on the *Curry-Howard isomorphism* [Bar81, GLT89, How80, Hue89], which is a correspondence between between proofs and terms and between propositions and types of λ-calculus. The isomorphism is also sometimes called the "propositions-as-types correspondence".)

The *Show Proof* command displays the incomplete proof term before you've completed the proof. For example, here's the proof term after using the *split* tactic above:

```
Show Proof.
    (fun (n m : nat) (H : n > m) => conj ?Goal ?Goal0)
```

The incomplete parts, the goals, are represented by *existential variables* with names that begin with `?Goal`. The *Show Existentials* command shows each existential with the hypotheses and conclusion for the associated goal.

```
Show Existentials.
    Existential 1 = ?Goal : [n : nat  m : nat  H : n > m |- P 1]
    Existential 2 = ?Goal0 : [n : nat  m : nat  H : n > m |- P 2]
```

Coq's kernel verifies the correctness of proof terms when it exits proof mode by checking that the proof term is *well-typed* and that its type is the same as the theorem statement.

After a proof is completed, *Print* <theorem_name> shows the proof term and its type. The type appears after the colon (`forall ...`), as for this theorem from Coq's standard library:

```
Print proj1.
    Fetching opaque proofs from disk for Coq.Init.Logic
    proj1 =
    fun (A B : Prop) (H : A /\ B) =>
    match H with
    | conj x x0 => (fun (H0 : A) (_ : B) => H0) x x0
    end
         : forall A B : Prop, A /\ B -> A

    Arguments proj1 [A B]%type_scope _
```

Entering and exiting proof mode

Coq enters *proof mode* when you begin a proof through commands such as *Theorem* or *Goal*. Coq user interfaces usually have a way to indicate that you're in proof mode.

Tactics are available only in proof mode (currently they give syntax errors outside of proof mode). Most *commands* can be used both in and out of proof mode, but some commands only work in or outside of proof mode.

When the proof is completed, you can exit proof mode with commands such as *Qed*, *Defined* and *Save*.

Command: `Goal` *type*
Asserts an unnamed proposition. This is intended for quick tests that a proposition is provable. If the proof is eventually completed and validated, you can assign a name with the *Save* or *Defined* commands. If no name is given, the name will be `Unnamed_thm` (or, if that name is already defined, a variant of that).

Command: `Qed`
Passes a completed *proof term* to Coq's kernel to check that the proof term is *well-typed* and to verify that its type matches the theorem statement. If it's verified, the proof term is added to the global environment as an *opaque* constant using the declared name from the original goal.

It's very rare for a proof term to fail verification. Generally this indicates a bug in a tactic you used or that you misused some unsafe tactics.

Error: `Attempt to save an incomplete proof.`

Error: `No focused proof (No proof-editing in progress).`
You tried to use a proof mode command such as *Qed* outside of proof mode.

Note: Sometimes an error occurs when building the proof term, because tactics do not enforce completely the term construction constraints.

The user should also be aware of the fact that since the proof term is completely rechecked at this point, one may have to wait a while when the proof is large. In some exceptional cases one may even incur a memory overflow.

Command: **Save** *ident*

Similar to *Qed*, except that the proof term is added to the global context with the name *ident*, which overrides any name provided by the *Theorem* command or its variants.

Command: **Defined** *ident*?

Similar to *Qed* and *Save*, except the proof is made *transparent*, which means that its content can be explicitly used for type checking and that it can be unfolded in conversion tactics (see *Applying conversion rules*, *Opaque*, *Transparent*). If *ident* is specified, the proof is defined with the given name, which overrides any name provided by the *Theorem* command or its variants.

Command: **Admitted**

This command is available in proof mode to give up the current proof and declare the initial goal as an axiom.

Command: **Abort** (**All** | *ident*)?

Cancels the current proof development, switching back to the previous proof development, or to the Coq toplevel if no other proof was being edited.

ident Aborts editing the proof named *ident* for use when you have nested proofs. See also *Nested Proofs Allowed*.

All Aborts all current proofs.

Error: **No focused proof (No proof-editing in progress).**

Command: **Proof** *term*

This command applies in proof mode. It is equivalent to **exact** *term*. **Qed.** That is, you have to give the full proof in one gulp, as a proof term (see Section *Applying theorems*).

> **Warning:** Use of this command is discouraged. In particular, it doesn't work in Proof General because it must immediately follow the command that opened proof mode, but Proof General inserts *Unset Silent* before it (see Proof General issue #498[22]).

Command: **Proof**

Is a no-op which is useful to delimit the sequence of tactic commands which start a proof, after a *Theorem* command. It is a good practice to use *Proof* as an opening parenthesis, closed in the script with a closing *Qed*.

See also:

Proof with

Command: **Proof using** *section_var_expr* (**with** *ltac_expr*)?

section_var_expr ::= *starred_ident_ref**
| (**-**)? *section_var_expr50*

section_var_expr50 ::= *section_var_expr0* **-** *section_var_expr0*
| *section_var_expr0* **+** *section_var_expr0*
| *section_var_expr0*

section_var_expr0 ::= *starred_ident_ref*
| **(** *section_var_expr* **)** *****?

starred_ident_ref ::= *ident* *****?
| **Type** *****?
| **All**

[22] https://github.com/ProofGeneral/PG/issues/498

Opens proof mode, declaring the set of section variables (see *Assumptions*) used by the proof. At `Qed` time, the system verifies that the set of section variables used in the proof is a subset of the declared one.

The set of declared variables is closed under type dependency. For example, if `T` is a variable and `a` is a variable of type `T`, then the commands `Proof using a` and `Proof using T a` are equivalent.

The set of declared variables always includes the variables used by the statement. In other words `Proof using e` is equivalent to `Proof using Type + e` for any declaration expression `e`.

`-` *`section_var_expr50`* Use all section variables except those specified by ***`section_var_expr50`***

`section_var_expr0`* `+` *`section_var_expr0` Use section variables from the union of both collections. See *Name a set of section hypotheses for Proof using* to see how to form a named collection.

`section_var_expr0`* `-` *`section_var_expr0` Use section variables which are in the first collection but not in the second one.

`*``?` Use the transitive closure of the specified collection.

`Type` Use only section variables occurring in the statement. Specifying **`*`** uses the forward transitive closure of all the section variables occurring in the statement. For example, if the variable `H` has type `p < 5` then `H` is in `p*` since `p` occurs in the type of `H`.

`All` Use all section variables.

See also:

Setting implicit automation tactics

Attribute: `using`

This *attribute* can be applied to the `Definition`, `Example`, `Fixpoint` and `CoFixpoint` commands as well as to `Lemma` and its variants. It takes a ***`section_var_expr`***, in quotes, as its value. This is equivalent to specifying the same ***`section_var_expr`*** in `Proof using`.

Example

```
Section Test.
Variable n : nat.
    n is declared

Hypothesis Hn : n <> 0.
    Hn is declared


#[using="Hn"]
Lemma example : 0 < n.
    1 goal

      n : nat
      Hn : n <> 0
      ============================
      0 < n

Abort.
End Test.
```

Proof using options

The following options modify the behavior of `Proof using`.

Option: `Default Proof Using "`*`section_var_expr`*`"`
Set this *option* to use *`section_var_expr`* as the default `Proof using` value. E.g. `Set Default Proof Using "a b"` will complete all `Proof` commands not followed by a `using` part with `using a b`.

Flag: `Suggest Proof Using`
When this *flag* is on, *`Qed`* suggests a `using` annotation if the user did not provide one.

Name a set of section hypotheses for `Proof using`

Command: `Collection` *`ident`* `:=` *`section_var_expr`*
This can be used to name a set of section hypotheses, with the purpose of making `Proof using` annotations more compact.

Example

Define the collection named `Some` containing `x`, `y` and `z`:

```
Collection Some := x y z.
```

Define the collection named `Fewer` containing only `x` and `y`:

```
Collection Fewer := Some - z
```

Define the collection named `Many` containing the set union or set difference of `Fewer` and `Some`:

```
Collection Many := Fewer + Some
Collection Many := Fewer - Some
```

Define the collection named `Many` containing the set difference of `Fewer` and the unnamed collection `x y`:

```
Collection Many := Fewer - (x y)
```

Command: `Existential` *`natural`* `: type?` `:=` *`term`*
This command instantiates an existential variable. *`natural`* is an index in the list of uninstantiated existential variables displayed by *`Show Existentials`*.

This command is intended to be used to instantiate existential variables when the proof is completed but some uninstantiated existential variables remain. To instantiate existential variables during proof edition, you should use the tactic *`instantiate`*.

Deprecated since version 8.13.

Command: `Grab Existential Variables`
This command can be run when a proof has no more goal to be solved but has remaining uninstantiated existential variables. It takes every uninstantiated existential variable and turns it into a goal.

Deprecated since version 8.13: Use *`Unshelve`* instead.

Proof modes

When entering proof mode through commands such as *Goal* and *Proof*, Coq picks by default the L_{tac} mode. Nonetheless, there exist other proof modes shipped in the standard Coq installation, and furthermore some plugins define their own proof modes. The default proof mode used when opening a proof can be changed using the following option.

Option: Default Proof Mode *string*

This *option* selects the proof mode to use when starting a proof. Depending on the proof mode, various syntactic constructs are allowed when writing a proof. All proof modes support commands; the proof mode determines which tactic language and set of tactic definitions are available. The possible option values are:

"Classic" Activates the L_{tac} language and the tactics with the syntax documented in this manual. Some tactics are not available until the associated plugin is loaded, such as `SSR` or `micromega`. This proof mode is set when the *prelude* is loaded.

"Noedit" No tactic language is activated at all. This is the default when the *prelude* is not loaded, e.g. through the `-noinit` option for `coqc`.

"Ltac2" Activates the Ltac2 language and the Ltac2-specific variants of the documented tactics. This value is only available after *Requiring* Ltac2. *Importing* Ltac2 sets this mode.

Some external plugins also define their own proof mode, which can be activated with this command.

Navigation in the proof tree

Command: Undo [**To** *natural*]?

Cancels the effect of the last *natural* commands or tactics. The **To** *natural* form goes back to the specified state number. If *natural* is not specified, the command goes back one command or tactic.

Command: Restart

Restores the proof to the original goal.

Error: No focused proof to restart.

Command: Focus *natural*?

Focuses the attention on the first goal to prove or, if *natural* is specified, the *natural*-th. The printing of the other goals is suspended until the focused goal is solved or unfocused.

Deprecated since version 8.8: Prefer the use of bullets or focusing brackets with a goal selector (see below).

Command: Unfocus

This command restores to focus the goal that were suspended by the last *Focus* command.

Deprecated since version 8.8.

Command: Unfocused

Succeeds if the proof is fully unfocused, fails if there are some goals out of focus.

Tactic: (*natural* | **[** *ident* **]**) **:**? **{**

Tactic: }

{ (without a terminating period) focuses on the first goal. The subproof can only be unfocused when it has been fully solved (*i.e.*, when there is no focused goal left). Unfocusing is then handled by } (again, without a terminating period). See also an example in the next section.

Note that when a focused goal is proved a message is displayed together with a suggestion about the right bullet or } to unfocus it or focus the next one.

***natural*:** Focuses on the *natural*-th goal to prove.

[*ident*]: { Focuses on the named goal *ident*.

Note: Goals are just existential variables and existential variables do not get a name by default. You can give a name to a goal by using **refine ?[*ident*]**. You may also wrap this in an Ltac-definition like:

```
Ltac name_goal name := refine ?[name].
```

See also:

Existential variables

Example

This first example uses the Ltac definition above, and the named goals only serve for documentation.

```
Goal forall n, n + 0 = n.
    1 goal

      ============================
      forall n : nat, n + 0 = n

Proof.
induction n; [ name_goal base | name_goal step ].
    2 goals

      ============================
      0 + 0 = 0

    goal 2 is:
     S n + 0 = S n

[base]: {
    1 goal

      ============================
      0 + 0 = 0

reflexivity.
    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    1 goal

    goal 1 is:
     S n + 0 = S n

}

[step]: {
    1 goal

      n : nat
      IHn : n + 0 = n
      ============================
      S n + 0 = S n
```

```
simpl.
    1 goal

      n : nat
      IHn : n + 0 = n
      ============================
      S (n + 0) = S n

f_equal.
    1 goal

      n : nat
      IHn : n + 0 = n
      ============================
      n + 0 = n

assumption.
    No more goals.

}
Qed.
    No more goals.
```

This can also be a way of focusing on a shelved goal, for instance:

```
Goal exists n : nat, n = n.
    1 goal

      ============================
      exists n : nat, n = n

eexists ?[x].
    1 focused goal
    (shelved: 1)

      ============================
      ?x = ?x

reflexivity.
    All the remaining goals are on the shelf.

    1 goal

    goal 1 is:
     nat

[x]: exact 0.
    No more goals.

Qed.
```

Error: `This proof is focused, but cannot be unfocused this way.`
You are trying to use `}` but the current subproof has not been fully solved.

Error: `No such goal (natural).`

Error: `No such goal (ident).`

Error: `Brackets do not support multi-goal selectors.`
Brackets are used to focus on a single goal given either by its position or by its name if it has one.

See also:

The error messages for bullets below.

Bullets

Alternatively, proofs can be structured with bullets instead of `{` and `}`. The use of a bullet `b` for the first time focuses on the first goal `g`, the same bullet cannot be used again until the proof of `g` is completed, then it is mandatory to focus the next goal with `b`. The consequence is that `g` and all goals present when `g` was focused are focused with the same bullet `b`. See the example below.

Different bullets can be used to nest levels. The scope of bullet does not go beyond enclosing `{` and `}`, so bullets can be reused as further nesting levels provided they are delimited by these. Bullets are made of repeated `-`, `+` or `*` symbols:

bullet ::=

Note again that when a focused goal is proved a message is displayed together with a suggestion about the right bullet or `}` to unfocus it or focus the next one.

Note: In Proof General (`Emacs` interface to Coq), you must use bullets with the priority ordering shown above to have a correct indentation. For example `-` must be the outer bullet and `**` the inner one in the example below.

The following example script illustrates all these features:

Example

```
Goal (((True /\ True) /\ True) /\ True) /\ True.
    1 goal

      ============================
      (((True /\ True) /\ True) /\ True) /\ True

Proof.
split.
    2 goals

      ============================
      ((True /\ True) /\ True) /\ True

    goal 2 is:
     True

- split.
    1 goal

      ============================
      ((True /\ True) /\ True) /\ True

    2 goals

      ============================
```

(continues on next page)

(continued from previous page)

```
      (True /\ True) /\ True

    goal 2 is:
     True

+ split.
    1 goal

      ============================
      (True /\ True) /\ True

    2 goals

      ============================
      True /\ True

    goal 2 is:
     True

** { split.
    1 goal

      ============================
      True /\ True

    1 goal

      ============================
      True /\ True

    2 goals

      ============================
      True

    goal 2 is:
     True

- trivial.
    1 goal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet -.

    4 goals

    goal 1 is:
     True
    goal 2 is:
     True
    goal 3 is:
     True
    goal 4 is:
     True
```

(continues on next page)

(continued from previous page)

```
- trivial.
    1 goal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    3 goals

    goal 1 is:
     True
    goal 2 is:
     True
    goal 3 is:
     True

}
** trivial.
    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet **.

    3 goals

    goal 1 is:
     True
    goal 2 is:
     True
    goal 3 is:
     True

    1 goal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet +.

    2 goals

    goal 1 is:
     True
    goal 2 is:
     True

+ trivial.
    1 goal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Focus next goal with bullet -.
```

(continues on next page)

(continued from previous page)

```
    1 goal

    goal 1 is:
     True

- assert True.
    1 goal

      ============================
      True

    2 goals

      ============================
      True

    goal 2 is:
     True

{ trivial. }
assumption.
    1 goal

      ============================
      True

    This subproof is complete, but there are some unfocused goals.
    Try unfocusing with "}".

    1 goal

    goal 1 is:
     True

    1 goal

      H : True
      ============================
      True

    No more goals.

Qed.
```

Error: Wrong bullet *bullet*$_1$: Current bullet *bullet*$_2$ is not finished.
Before using bullet *bullet*$_1$ again, you should first finish proving the current focused goal. Note that *bullet*$_1$ and *bullet*$_2$ may be the same.

Error: Wrong bullet *bullet*$_1$: Bullet *bullet*$_2$ is mandatory here.
You must put *bullet*$_2$ to focus on the next goal. No other bullet is allowed here.

Error: No such goal. Focus next goal with bullet *bullet*.
You tried to apply a tactic but no goals were under focus. Using *bullet* is mandatory here.

Error: No such goal. Try unfocusing with }.
You just finished a goal focused by `{`, you must unfocus it with `}`.

Mandatory Bullets

Using *Default Goal Selector* with the ! selector forces tactic scripts to keep focus to exactly one goal (e.g. using bullets) or use explicit goal selectors.

Set Bullet Behavior

Option: Bullet Behavior "None" | "Strict Subproofs"

This *option* controls the bullet behavior and can take two possible values:

- "None": this makes bullets inactive.
- "Strict Subproofs": this makes bullets active (this is the default behavior).

Modifying the order of goals

Tactic: cycle *int_or_var*

Reorders the selected goals so that the first *integer* goals appear after the other selected goals. If *integer* is negative, it puts the last *integer* goals at the beginning of the list. The tactic is only useful with a goal selector, most commonly `all:`. Note that other selectors reorder goals; `1,3: cycle 1` is not equivalent to `all: cycle 1`. See *… : … (goal selector)*.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 goal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 goals

      ============================
      P 1

    goal 2 is:
     P 2
    goal 3 is:
     P 3
    goal 4 is:
     P 4
    goal 5 is:
     P 5

all: cycle 2.
    5 goals

      ============================
      P 3

    goal 2 is:
     P 4
    goal 3 is:
```

(continues on next page)

(continued from previous page)

```
     P 5
    goal 4 is:
     P 1
    goal 5 is:
     P 2

all: cycle -3.
    5 goals

      ============================
      P 5

    goal 2 is:
     P 1
    goal 3 is:
     P 2
    goal 4 is:
     P 3
    goal 5 is:
     P 4
```

Tactic: swap *int_or_var int_or_var*

Exchanges the position of the specified goals. Negative values for ***integer*** indicate counting goals backward from the end of the list of selected goals. Goals are indexed from 1. The tactic is only useful with a goal selector, most commonly `all:`. Note that other selectors reorder goals; `1,3: swap 1 3` is not equivalent to `all: swap 1 3`. See *… : … (goal selector)*.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 goal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 goals

      ============================
      P 1

    goal 2 is:
     P 2
    goal 3 is:
     P 3
    goal 4 is:
     P 4
    goal 5 is:
     P 5

all: swap 1 3.
    5 goals

      ============================
      P 3
```

(continues on next page)

(continued from previous page)

```
    goal 2 is:
     P 2
    goal 3 is:
     P 1
    goal 4 is:
     P 4
    goal 5 is:
     P 5

all: swap 1 -1.
    5 goals

      ============================
      P 5

    goal 2 is:
     P 2
    goal 3 is:
     P 1
    goal 4 is:
     P 4
    goal 5 is:
     P 3
```

Tactic: `revgoals`

Reverses the order of the selected goals. The tactic is only useful with a goal selector, most commonly `all :`. Note that other selectors reorder goals; `1,3: revgoals` is not equivalent to `all: revgoals`. See *… : … (goal selector)*.

Example

```
Goal P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5.
    1 goal

      ============================
      P 1 /\ P 2 /\ P 3 /\ P 4 /\ P 5

repeat split.
    5 goals

      ============================
      P 1

    goal 2 is:
     P 2
    goal 3 is:
     P 3
    goal 4 is:
     P 4
    goal 5 is:
     P 5

all: revgoals.
    5 goals
```

(continues on next page)

(continued from previous page)

```
  ============================
  P 5

goal 2 is:
 P 4
goal 3 is:
 P 3
goal 4 is:
 P 2
goal 5 is:
 P 1
```

Postponing the proof of some goals

Goals can be shelved so they are no longer displayed in the proof state. They can then be unshelved to make them visible again.

Tactic: `shelve`
This tactic moves all goals under focus to a shelf. While on the shelf, goals will not be focused on. They can be solved by unification, or they can be called back into focus with the command *Unshelve*.

Tactic: `shelve_unifiable`
Shelves only the goals under focus that are mentioned in other goals. Goals that appear in the type of other goals can be solved by unification.

Example

```
Goal exists n, n=0.
    1 goal

      ============================
      exists n : nat, n = 0

refine (ex_intro _ _ _).
    1 focused goal
    (shelved: 1)

      ============================
      ?Goal = 0

all: shelve_unifiable.
reflexivity.
    No more goals.
```

Command: `Unshelve`
This command moves all the goals on the shelf (see *shelve*) from the shelf into focus, by appending them to the end of the current list of focused goals.

Tactic: `unshelve` *ltac_expr1*
Performs ***tactic***, then unshelves existential variables added to the shelf by the execution of ***tactic***, prepending them to the current goal.

Tactic: `give_up`
This tactic removes the focused goals from the proof. They are not solved, and cannot be solved later in the proof. As the goals are not solved, the proof cannot be closed.

The `give_up` tactic can be used while editing a proof, to choose to write the proof script in a non-sequential order.

Requesting information

Command: `Show` *ident* | *natural* ?
Displays the current goals.

natural Display only the *natural*-th goal.

ident Displays the named goal *ident*. This is useful in particular to display a shelved goal but only works if the corresponding existential variable has been named by the user (see *Existential variables*) as in the following example.

Example

```
Goal exists n, n = 0.
    1 goal

      ============================
      exists n : nat, n = 0

eexists ?[n].
    1 focused goal
    (shelved: 1)

      ============================
      ?n = 0

Show n.
    goal n is:

      ============================
      nat
```

Error: `No focused proof.`

Error: `No such goal.`

Command: `Show Proof` `Diffs` `removed` ? ?
Displays the proof term generated by the tactics that have been applied so far. If the proof is incomplete, the term will contain holes, which correspond to subterms which are still to be constructed. Each hole is an existential variable, which appears as a question mark followed by an identifier.

Specifying "Diffs" highlights the difference between the current and previous proof step. By default, the command shows the output once with additions highlighted. Including "removed" shows the output twice: once showing removals and once showing additions. It does not examine the *Diffs* option. See *"Show Proof" differences*.

Command: `Show Conjectures`
Prints the names of all the theorems that are currently being proved. As it is possible to start proving a previous lemma during the proof of a theorem, there may be multiple names.

Command: `Show Intro`
If the current goal begins by at least one product, prints the name of the first product as it would be generated by an anonymous *intro*. The aim of this command is to ease the writing of more robust scripts. For example, with an appropriate Proof General macro, it is possible to transform any anonymous *intro* into a qualified one such as `intro y13`. In the case of a non-product goal, it prints nothing.

Command: `Show Intros`
Similar to the previous command. Simulates the naming process of *intros*.

Command: `Show Existentials`
Displays all open goals / existential variables in the current proof along with the context and type of each variable.

Command: `Show Match` *qualid*
Displays a template of the Gallina *match* construct with a branch for each constructor of the type *qualid*. This is used internally by company-coq[23].

Example

```
Show Match nat.
    match # with
      | O =>
      | S x =>
     end
```

Error: `Unknown inductive type.`

Command: `Show Universes`
Displays the set of all universe constraints and its normalized form at the current stage of the proof, useful for debugging universe inconsistencies.

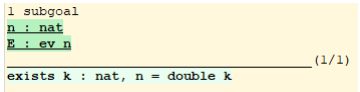
Command: `Show Goal` *natural* `at` *natural*
Available in coqtop. Displays a goal at a proof state using the goal ID number and the proof state ID number. It is primarily for use by tools such as Prooftree that need to fetch goal history in this way. Prooftree is a tool for visualizing a proof as a tree that runs in Proof General.

Command: `Guarded`
Some tactics (e.g. *refine*) allow to build proofs using fixpoint or co-fixpoint constructions. Due to the incremental nature of proof construction, the check of the termination (or guardedness) of the recursive calls in the fixpoint or cofixpoint constructions is postponed to the time of the completion of the proof.

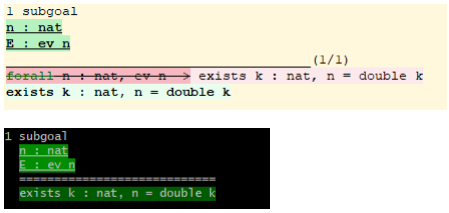
The command *Guarded* allows checking if the guard condition for fixpoint and cofixpoint is violated at some time of the construction of the proof without having to wait the completion of the proof.

Showing differences between proof steps

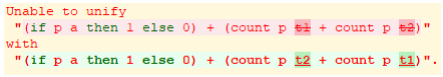
Coq can automatically highlight the differences between successive proof steps and between values in some error messages. Coq can also highlight differences in the proof term. For example, the following screenshots of CoqIDE and coqtop show the application of the same *intros* tactic. The tactic creates two new hypotheses, highlighted in green. The conclusion is entirely in pale green because although it's changed, no tokens were added to it. The second screenshot uses the "removed" option, so it shows the conclusion a second time with the old text, with deletions marked in red. Also, since the hypotheses are new, no line of old text is shown for them.



[23] https://github.com/cpitclaudel/company-coq



This image shows an error message with diff highlighting in CoqIDE:



How to enable diffs

Option: `Diffs` `"on"` | `"off"` | `"removed"`

This *option* is used to enable diffs. The "on" setting highlights added tokens in green, while the "removed" setting additionally reprints items with removed tokens in red. Unchanged tokens in modified items are shown with pale green or red. Diffs in error messages use red and green for the compared values; they appear regardless of the setting. (Colors are user-configurable.)

For coqtop, showing diffs can be enabled when starting coqtop with the `-diffs on|off|removed` command-line option or by setting the *`Diffs`* option within Coq. You will need to provide the `-color on|auto` command-line option when you start coqtop in either case.

Colors for coqtop can be configured by setting the `COQ_COLORS` environment variable. See section *By environment variables*. Diffs use the tags `diff.added`, `diff.added.bg`, `diff.removed` and `diff.removed.bg`.

In CoqIDE, diffs should be enabled from the `View` menu. Don't use the `Set Diffs` command in CoqIDE. You can change the background colors shown for diffs from the `Edit | Preferences | Tags` panel by changing the settings for the `diff.added`, `diff.added.bg`, `diff.removed` and `diff.removed.bg` tags. This panel also lets you control other attributes of the highlights, such as the foreground color, bold, italic, underline and strikeout.

Proof General can also display Coq-generated proof diffs automatically. Please see the PG documentation section "Showing Proof Diffs"[24]) for details.

How diffs are calculated

Diffs are calculated as follows:

1. Select the old proof state to compare to, which is the proof state before the last tactic that changed the proof. Changes that only affect the view of the proof, such as `all: swap 1 2`, are ignored.
2. For each goal in the new proof state, determine what old goal to compare it to—the one it is derived from or is the same as. Match the hypotheses by name (order is ignored), handling compacted items specially.
3. For each hypothesis and conclusion (the "items") in each goal, pass them as strings to the lexer to break them into tokens. Then apply the Myers diff algorithm [Mye86] on the tokens and add appropriate highlighting.

Notes:

- Aside from the highlights, output for the "on" option should be identical to the undiffed output.
- Goals completed in the last proof step will not be shown even with the "removed" setting.

This screen shot shows the result of applying a *`split`* tactic that replaces one goal with 2 goals. Notice that the goal `P 1` is not highlighted at all after the split because it has not changed.

[24] https://proofgeneral.github.io/doc/master/userman/Coq-Proof-General#Showing-Proof-Diffs

```
3 subgoals
_____________________________________(1/3)
P 1
_____________________________________(2/3)
P 2
_____________________________________(3/3)
P 3
```

Diffs may appear like this after applying a *intro* tactic that results in a compacted hypotheses:

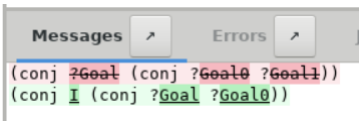
```
1 subgoal
n, m : nat
_____________________________________(1/1)
n + m = m + n
```

"Show Proof" differences

To show differences in the proof term:

- In coqtop and Proof General, use the *Show Proof* `Diffs` command.
- In CoqIDE, position the cursor on or just after a tactic to compare the proof term after the tactic with the proof term before the tactic, then select `View / Show Proof` from the menu or enter the associated key binding. Differences will be shown applying the current `Show Diffs` setting from the `View` menu. If the current setting is `Don't show diffs`, diffs will not be shown.

  Output with the "added and removed" option looks like this:



Controlling proof mode

Option: `Hyps Limit` *natural*
This *option* controls the maximum number of hypotheses displayed in goals after the application of a tactic. All the hypotheses remain usable in the proof development. When unset, it goes back to the default mode which is to print all available hypotheses.

Flag: `Nested Proofs Allowed`
When turned on (it is off by default), this *flag* enables support for nested proofs: a new assertion command can be inserted before the current proof is finished, in which case Coq will temporarily switch to the proof of this *nested lemma*. When the proof of the nested lemma is finished (with *Qed* or *Defined*), its statement will be made available (as if it had been proved before starting the previous proof) and Coq will switch back to the proof of the previous assertion.

Flag: `Printing Goal Names`
When this *flag* is turned on, the name of the goal is printed in proof mode, which can be useful in cases of cross references between goals.

Controlling memory usage

Command: `Print Debug GC`
Prints heap usage statistics, which are values from the `stat` type of the `Gc` module described here[25] in the OCaml documentation. The `live_words`, `heap_words` and `top_heap_words` values give the basic information. Words are 8 bytes or 4 bytes, respectively, for 64- and 32-bit executables.

When experiencing high memory usage the following commands can be used to force Coq to optimize some of its internal data structures.

Command: `Optimize Proof`
Shrink the data structure used to represent the current proof.

Command: `Optimize Heap`
Perform a heap compaction. This is generally an expensive operation. See: OCaml Gc.compact[26] There is also an analogous tactic *`optimize_heap`*.

Memory usage parameters can be set through the *OCAMLRUNPARAM* environment variable.

3.1.2 Tactics

Tactics specify how to transform the *proof state* of an incomplete proof to eventually generate a complete proof.

Proofs can be developed in two basic ways: In forward reasoning, the proof begins by proving simple statements that are then combined to prove the theorem statement as the last step of the proof. With forward reasoning, for example, the proof of `A /\ B` would begin with proofs of `A` and `B`, which are then used to prove `A /\ B`. Forward reasoning is probably the most common approach in human-generated proofs.

In backward reasoning, the proof begins with the theorem statement as the goal, which is then gradually transformed until every subgoal generated along the way has been proven. In this case, the proof of `A /\ B` begins with that formula as the goal. This can be transformed into two subgoals, `A` and `B`, followed by the proofs of `A` and `B`. Coq and its tactics use backward reasoning.

A tactic may fully prove a goal, in which case the goal is removed from the proof state. More commonly, a tactic replaces a goal with one or more *subgoals*. (We say that a tactic reduces a goal to its subgoals.)

Most tactics require specific elements or preconditions to reduce a goal; they display error messages if they can't be applied to the goal. A few tactics, such as *`auto`*, don't fail even if the proof state is unchanged.

Goals are identified by number. The current goal is number 1. Tactics are applied to the current goal by default. (The default can be changed with the *`Default Goal Selector`* option.) They can be applied to another goal or to multiple goals with a *goal selector* such as **`2:`** ***`tactic`***.

This chapter describes many of the most common built-in tactics. Built-in tactics can be combined to form tactic expressions, which are described in the *Ltac* chapter. Since tactic expressions can be used anywhere that a built-in tactic can be used, "tactic" may refer to both built-in tactics and tactic expressions.

[25] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html#TYPEstat
[26] http://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html#VALcompact

Common elements of tactics

Reserved keywords

The tactics described in this chapter reserve the following keywords:

```
by using
```

Thus, these keywords cannot be used as identifiers. It also declares the following character sequences as tokens:

```
** [= |-
```

Invocation of tactics

A tactic is applied as an ordinary command. It may be preceded by a goal selector (see Section *Goal selectors*). If no selector is specified, the default selector is used.

tactic_invocation ::= *toplevel_selector* **:** *tactic***.**
| *tactic***.**

Option: `Default Goal Selector "`*`toplevel_selector`*`"`

This *option* controls the default selector, used when no selector is specified when applying a tactic. The initial value is 1, hence the tactics are, by default, applied to the first goal.

Using value `all` will make it so that tactics are, by default, applied to every goal simultaneously. Then, to apply a tactic tac to the first goal only, you can write `1:tac`.

Using value `!` enforces that all tactics are used either on a single focused goal or with a local selector ("strict focusing mode").

Although other selectors are available, only `all`, `!` or a single natural number are valid default goal selectors.

Bindings

Tactics that take a term as an argument may also accept *bindings* to instantiate some parameters of the term by name or position. The general form of a term with *bindings* is ***term*$_{tac}$ `with` *bindings*** where *bindings* can take two different forms:

bindings ::= (*ident* | *natural* **:=** *term*) +
| *one_term* +

- In the first form, if an *ident* is specified, it must be bound in the type of ***term*** and provides the tactic with an instance for the parameter of this name. If a *natural* is specified, it refers to the `n`-th non dependent premise of ***term*$_{tac}$**.

  Error: `No such binder.`

- In the second form, the interpretation of the *one_term*s depend on which tactic they appear in. For *induction*, *destruct*, *elim* and *case*, the *one_term*s provide instances for all the dependent products in the type of ***term*$_{tac}$** while in the case of *apply*, or of *constructor* and its variants, only instances for the dependent products that are not bound in the conclusion of ***term*$_{tac}$** are required.

  Error: `Not the right number of missing arguments.`

Intro patterns

Intro patterns let you specify the name to assign to variables and hypotheses introduced by tactics. They also let you split an introduced hypothesis into multiple hypotheses or subgoals. Common tactics that accept intro patterns include `assert`, `intros` and `destruct`.

intropattern_list ::= *intropattern* *
intropattern ::= *****
| ******
| *simple_intropattern*
simple_intropattern ::= *simple_intropattern_closed* (**%** *term0*) *
simple_intropattern_closed ::= *naming_intropattern*
| **_**
| *or_and_intropattern*
| *rewriting_intropattern*
| *injection_intropattern*
naming_intropattern ::= *ident*
| **?**
| **?***ident*
or_and_intropattern ::= **[** *intropattern_list* * | **]**
| **(** *simple_intropattern* * **,** **)**
| **(** *simple_intropattern* * **&** **)**
rewriting_intropattern ::= **->**
| **<-**
injection_intropattern ::= **[=** *intropattern_list* **]**
or_and_intropattern_loc ::= *or_and_intropattern*
| **ident**

Note that the intro pattern syntax varies between tactics. Most tactics use ***simple_intropattern*** in the grammar. `destruct`, `edestruct`, `induction`, `einduction`, `case`, `ecase` and the various `inversion` tactics use ***or_and_intropattern_loc***, while `intros` and `eintros` use ***intropattern_list***. The **eqn:** construct in various tactics uses ***naming_intropattern***.

Naming patterns

Use these elementary patterns to specify a name:

- ***ident*** — use the specified name
- **?** — let Coq choose a name
- **?*ident*** — generate a name that begins with ***ident***
- **_** — discard the matched part (unless it is required for another hypothesis)
- if a disjunction pattern omits a name, such as `[|H2]`, Coq will choose a name

Splitting patterns

The most common splitting patterns are:

- split a hypothesis in the form **A /\ B** into two hypotheses `H1: A` and `H2: B` using the pattern `(H1 & H2)` or `(H1, H2)` or `[H1 H2]`. *Example*. This also works on **A <-> B**, which is just a notation representing **(A -> B) /\ (B -> A)**.

- split a hypothesis in the form `A \/ B` into two subgoals using the pattern `[H1|H2]`. The first subgoal will have the hypothesis `H1: A` and the second subgoal will have the hypothesis `H2: B`. *Example*
- split a hypothesis in either of the forms `A /\ B` or `A \/ B` using the pattern `[]`.

Patterns can be nested: **`[[Ha|Hb] H]`** can be used to split **`(A \/ B) /\ C`**.

Note that there is no equivalent to intro patterns for goals. For a goal `A /\ B`, use the *`split`* tactic to replace the current goal with subgoals `A` and `B`. For a goal `A \/ B`, use *`left`* to replace the current goal with `A`, or *`right`* to replace the current goal with `B`.

- **`(`** *`simple_intropattern`*`+ ,` **`)`** — matches a product over an inductive type with a *single constructor*. If the number of patterns equals the number of constructor arguments, then it applies the patterns only to the arguments, and **`(`** *`simple_intropattern`*`+ ,` **`)`** is equivalent to **`[`** *`simple_intropattern`*`+` **`]`**. If the number of patterns equals the number of constructor arguments plus the number of **`let-ins`**, the patterns are applied to the arguments and **`let-in`** variables.
- **`(`** *`simple_intropattern`*`+ &` **`)`** — matches a right-hand nested term that consists of one or more nested binary inductive types such as `a1 OP1 a2 OP2 ...` (where the `OPn` are right-associative). (If the `OPn` are left-associative, additional parentheses will be needed to make the term right-hand nested, such as `a1 OP1 (a2 OP2 ...)`.) The splitting pattern can have more than 2 names, for example `(H1 & H2 & H3)` matches `A /\ B /\ C`. The inductive types must have a *single constructor with two parameters*. *Example*
- **`[`** *`intropattern_list`*`+ |` **`]`** — splits an inductive type that has *multiple constructors* such as **`A \/ B`** into multiple subgoals. The number of *`intropattern_list`* must be the same as the number of constructors for the matched part.
- **`[`** *`intropattern`*`+` **`]`** — splits an inductive type that has a *single constructor with multiple parameters* such as **`A /\ B`** into multiple hypotheses. Use **`[H1 [H2 H3]]`** to match `A /\ B /\ C`.
- **`[]`** — splits an inductive type: If the inductive type has multiple constructors, such as **`A \/ B`**, create one subgoal for each constructor. If the inductive type has a single constructor with multiple parameters, such as **`A /\ B`**, split it into multiple hypotheses.

Equality patterns

These patterns can be used when the hypothesis is an equality:

- **`->`** — replaces the right-hand side of the hypothesis with the left-hand side of the hypothesis in the conclusion of the goal; the hypothesis is cleared; if the left-hand side of the hypothesis is a variable, it is substituted everywhere in the context and the variable is removed. *Example*
- **`<-`** — similar to **`->`**, but replaces the left-hand side of the hypothesis with the right-hand side of the hypothesis.
- **`[=`** *`intropattern`*`* ,` **`]`** — If the product is over an equality type, applies either *`injection`* or *`discriminate`*. If *`injection`* is applicable, the intropattern is used on the hypotheses generated by *`injection`*. If the number of patterns is smaller than the number of hypotheses generated, the pattern **`?`** is used to complete the list. *Example*

Other patterns

- **`*`** — introduces one or more quantified variables from the result until there are no more quantified variables. *Example*
- **`**`** — introduces one or more quantified variables or hypotheses from the result until there are no more quantified variables or implications (`->`). `intros **` is equivalent to `intros`. *Example*

- ***simple_intropattern_closed*** `% term`* — first applies each of the terms with the *apply … in* tactic on the hypothesis to be introduced, then it uses ***simple_intropattern_closed***. *Example*

Note: **A \/ B** and **A /\ B** use infix notation to refer to the inductive types **or** and **and**. **or** has multiple constructors (**or_introl** and **or_intror**), while **and** has a single constructor (**conj**) with multiple parameters (**A** and **B**). These are defined in `theories/Init/Logic.v`. The "where" clauses define the infix notation for "or" and "and".

```
Inductive or (A B:Prop) : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.

Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> A /\ B
where "A /\ B" := (and A B) : type_scope.
```

Note: **intros** `p`+ is not always equivalent to **intros p; ... ; intros p** if some of the **p** are _. In the first form, all erasures are done at once, while they're done sequentially for each tactic in the second form. If the second matched term depends on the first matched term and the pattern for both is _ (i.e., both will be erased), the first **intros** in the second form will fail because the second matched term still has the dependency on the first.

Examples:

Example: intro pattern for ∧

```
1 goal

  A, B : Prop
  H : A /\ B
  ============================
  True

destruct H as (HA & HB).
1 goal

  A, B : Prop
  HA : A
  HB : B
  ============================
  True
```

Example: intro pattern for ∨

```
1 goal

  A, B : Prop
  H : A \/ B
  ============================
  True
```

```
destruct H as [HA|HB]. all: swap 1 2.
    2 goals

      A, B : Prop
      HA : A
      ============================
      True

    goal 2 is:
     True

    2 goals

      A, B : Prop
      HB : B
      ============================
      True

    goal 2 is:
     True
```

Example: -> intro pattern

```
    1 goal

      x, y, z : nat
      H : x = y
      ============================
      y = z -> x = z

intros ->.
    1 goal

      x, z : nat
      H : x = z
      ============================
      x = z
```

Example: [=] intro pattern

The first **`intros [=]`** uses *`injection`* to strip **`(S ...)`** from both sides of the matched equality. The second uses *`discriminate`* on the contradiction **`1 = 2`** (internally represented as **`(S O) = (S (S O))`**) to complete the goal.

```
    1 goal

      n, m : nat
      ============================
      S n = S m -> 1 = 2 -> False

intros [= H].
    1 goal
```

(continues on next page)

(continued from previous page)

```
  n, m : nat
  H : n = m
  ============================
  1 = 2 -> False

intros [=].
    No more goals.
```

Example: (A & B & ...) intro pattern

```
    1 goal

  ============================
  A /\ (exists x : nat, B x /\ C) -> True

intros (a & x & b & c).
    1 goal

  a : A
  x : nat
  b : B x
  c : C
  ============================
  True
```

Example: * intro pattern

```
    1 goal

  ============================
  forall A B : Prop, A -> B

intros *.
    1 goal

  A, B : Prop
  ============================
  A -> B
```

Example: ** pattern ("intros **" is equivalent to "intros")

```
    1 goal

  ============================
  forall A B : Prop, A -> B

intros **.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      A, B : Prop
      H : A
      ============================
      B
```

Example: compound intro pattern

```
    1 goal

      ============================
      forall A B C : Prop, A \/ B /\ C -> (A -> C) -> C

intros * [a | (_,c)] f.
    2 goals

      A, B, C : Prop
      a : A
      f : A -> C
      ============================
      C

    goal 2 is:
     C

all: swap 1 2.
    2 goals

      A, B, C : Prop
      c : C
      f : A -> C
      ============================
      C

    goal 2 is:
     C
```

Example: combined intro pattern using [=] -> and %

```
    1 goal

      A : Type
      xs, ys : list A
      ============================
      S (length ys) = 1 -> xs ++ ys = xs

intros [=->%length_zero_iff_nil].
    1 goal

      A : Type
      xs : list A
      ============================
      xs ++ nil = xs
```

- `intros` would add `H : S (length ys) = 1`
- `intros [=]` would additionally apply *`injection`* to `H` to yield `H0 : length ys = 0`
- `intros [=->%length_zero_iff_nil]` applies the theorem, making H the equality `l=nil`, which is then applied as for `->`.

```
Theorem length_zero_iff_nil (l : list A):
   length l = 0 <-> l=nil.
```

The example is based on Tej Chajed's coq-tricks[27]

Occurrence clauses

An occurrence is a subterm of a goal or hypothesis that matches a pattern provided by a tactic. Occurrence clauses select a subset of the ocurrences in a goal and/or in one or more of its hypotheses.

occurrences ::= **at** *occs_nums*
| **in** *goal_occurrences*

simple_occurrences ::= *occurrences*

occs_nums ::= [**-**?] *nat_or_var* +

nat_or_var ::= *natural* | *ident*

goal_occurrences ::= *hyp_occs* + , [**|-** *concl_occs*?]?
| ***** **|-** *concl_occs*?
| **|-** *concl_occs*?
| *concl_occs*?

hyp_occs ::= *hypident* [**at** *occs_nums*]?

hypident ::= *ident*
| **(type of** *ident* **)**
| **(value of** *ident* **)**

concl_occs ::= ***** [**at** *occs_nums*]?

occurrences The first form of *`occurrences`* selects occurrences in the conclusion of the goal. The second form can select occurrences in the goal conclusion and in one or more hypotheses.

simple_occurrences A semantically restricted form of ***`occurrences`*** that doesn't allow the `at` clause anywhere within it.

[**-**?] ***`nat_or_var`*** + Selects the specified occurrences within a single goal or hypothesis. Occurrences are numbered starting with 1 following a depth-first traversal of the term's expression, including occurrences in *implicit arguments* and *coercions* that are not displayed by default. (Set the *`Printing All`* flag to show those in the printed term.)

For example, when matching the pattern `_ + _` in the term `(a + b) + c`, occurrence 1 is `(...) + c` and occurrence 2 is `(a + b)`. When matching that pattern with term `a + (b + c)`, occurrence 1 is `a + (...)` and occurrence 2 is `b + c`.

Specifying − includes all occurrences *except* the ones listed.

[27] https://github.com/tchajed/coq-tricks/blob/8e6efe4971ed828ac8bdb5512c1f615d7d62691e/src/IntroPatterns.v

hyp_occs [*|,] **|-** *concl_occs* ? Selects occurrences in the specified hypotheses and the specified occurrences in the conclusion.

*** |-** *concl_occs* ? Selects all occurrences in all hypotheses and the specified occurrences in the conclusion.

|- *concl_occs* ? Selects the specified occurrences in the conclusion.

goal_occurrences **::=** *concl_occs* ? Selects all occurrences in all hypotheses and in the specified occurrences in the conclusion.

hypident **at** *occs_nums* ? Omiting *occs_nums* selects all occurrences within the hypothesis.

hypident **::=** *ident* Selects the hypothesis named *ident*.

(type of *ident* **)** Selects the type part of the named hypothesis (e.g. `: nat`).

(value of *ident* **)** Selects the value part of the named hypothesis (e.g. `:= 1`).

concl_occs **::= * at** *occs_nums* ? Selects occurrences in the conclusion. '*' by itself selects all occurrences. *occs_nums* selects the specified occurrences.

Use `in *` to select all occurrences in all hypotheses and the conclusion, which is equivalent to `in * |- *`. Use `* |-` to select all occurrences in all hypotheses.

Tactics that select a specific hypothesis H to apply to other hypotheses, such as *rewrite* `H in * |-`, won't apply H to itself.

If multiple occurrences are given, such as in *rewrite* `H at 1 2 3`, the tactic must match at least one occurrence in order to succeed. The tactic will fail if no occurrences match. Occurrence numbers that are out of range (e.g. `at 1 3` when there are only 2 occurrences in the hypothesis or conclusion) are ignored.

Tactics that use occurrence clauses include *set*, *remember*, *induction* and *destruct*.

See also:

Managing the local context, *Case analysis and induction*, *Printing constructions in full*.

Applying theorems

Tactic: exact *term*
This tactic applies to any goal. It gives directly the exact proof term of the goal. Let `T` be our goal, let `p` be a term of type `U` then `exact p` succeeds iff `T` and `U` are convertible (see *Conversion rules*).

Error: Not an exact proof.

Variant: eexact *term*.
This tactic behaves like *exact* but is able to handle terms and goals with existential variables.

Tactic: assumption
This tactic looks in the local context for a hypothesis whose type is convertible to the goal. If it is the case, the subgoal is proved. Otherwise, it fails.

Error: No such assumption.

Variant: eassumption
This tactic behaves like *assumption* but is able to handle goals with existential variables.

Tactic: refine *term*

This tactic applies to any goal. It behaves like *exact* with a big difference: the user can leave some holes (denoted by _ or **(_ : *type*)**) in the term. *refine* will generate as many subgoals as there are remaining holes in the elaborated term. The type of holes must be either synthesized by the system or declared by an explicit cast like `(_ : nat -> Prop)`. Any subgoal that occurs in other subgoals is automatically shelved, as if calling *shelve_unifiable*. The produced subgoals (shelved or not) are *not* candidates for typeclass resolution, even if they have a type-class type as conclusion, letting the user control when and how typeclass resolution is launched on them. This low-level tactic can be useful to advanced users.

Example

```
Inductive Option : Set :=
| Fail : Option
| Ok : bool -> Option.
    Option is defined
    Option_rect is defined
    Option_ind is defined
    Option_rec is defined
    Option_sind is defined


Definition get : forall x:Option, x <> Fail -> bool.
    1 goal

      ============================
      forall x : Option, x <> Fail -> bool

  refine
    (fun x:Option =>
      match x return x <> Fail -> bool with
      | Fail => _
      | Ok b => fun _ => b
      end).
    1 goal

      x : Option
      ============================
      Fail <> Fail -> bool

  intros; absurd (Fail = Fail); trivial.
    No more goals.

Defined.
```

Error: Invalid argument.

The tactic *refine* does not know what to do with the term you gave.

Error: Refine passed ill-formed term.

The term you gave is not a valid proof (not easy to debug in general). This message may also occur in higher-level tactics that call *refine* internally.

Error: Cannot infer a term for this placeholder.

There is a hole in the term you gave whose type cannot be inferred. Put a cast around it.

Variant: simple refine *term*

This tactic behaves like refine, but it does not shelve any subgoal. It does not perform any beta-reduction either.

Variant: `notypeclasses refine` *term*
This tactic behaves like *refine* except it performs type checking without resolution of typeclasses.

Variant: `simple notypeclasses refine` *term*
This tactic behaves like the combination of *simple refine* and *notypeclasses refine*: it performs type checking without resolution of typeclasses, does not perform beta reductions or shelve the subgoals.

Debug `"unification"` enables printing traces of unification steps used during elaboration/typechecking and the *refine* tactic. `"ho-unification"` prints information about higher order heuristics.

Tactic: `apply` *term*
This tactic applies to any goal. The argument term is a term well-formed in the local context. The tactic *apply* tries to match the current goal against the conclusion of the type of *term*. If it succeeds, then the tactic returns as many subgoals as the number of non-dependent premises of the type of term. If the conclusion of the type of *term* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then each component of the tuple is recursively matched to the goal in the left-to-right order.

The tactic *apply* relies on first-order unification with dependent types unless the conclusion of the type of *term* is of the form **`P (t1 ... tn)`** with `P` to be instantiated. In the latter case, the behavior depends on the form of the goal. If the goal is of the form **`(fun x => Q) u1 ... un`** and the t_i and u_i unify, then `P` is taken to be `(fun x => Q)`. Otherwise, *apply* tries to define `P` by abstracting over `t_1 ... t__n` in the goal. See *pattern* to transform the goal so that it gets the form **`(fun x => Q) u1 ... un`**.

Error: `Unable to unify` *term* `with` *term*`.`
The *apply* tactic failed to match the conclusion of *term* and the current goal. You can help the *apply* tactic by transforming your goal with the *change* or *pattern* tactics.

Error: `Unable to find an instance for the variables` *ident*`+` `.`
This occurs when some instantiations of the premises of *term* are not deducible from the unification. This is the case, for instance, when you want to apply a transitivity property. In this case, you have to use one of the variants below:

Variant: `apply` *term* `with` *term*`+`
Provides apply with explicit instantiations for all dependent premises of the type of term that do not occur in the conclusion and consequently cannot be found by unification. Notice that the collection *term*`+` must be given according to the order of these dependent premises of the type of term.

Error: `Not the right number of missing arguments.`

Variant: `apply` *term* `with` *bindings*
This also provides apply with values for instantiating premises. Here, variables are referred by names and non-dependent products by increasing numbers (see *Bindings*).

Variant: `apply` *term*`+,`
This is a shortcut for **`apply` *term*$_1$`; [.. | ... ; [ .. | apply` *term*$_n$`] ... ]`**, i.e. for the successive applications of *term*$_{i+1}$ on the last subgoal generated by **`apply` *term*$_i$** , starting from the application of *term*$_1$.

Variant: `eapply` *term*
The tactic *eapply* behaves like *apply* but it does not fail when no instantiations are deducible for some variables in the premises. Rather, it turns these variables into existential variables which are variables still to instantiate (see *Existential variables*). The instantiation is intended to be found later in the proof.

Variant: `rapply` *term*
The tactic *rapply* behaves like *eapply* but it uses the proof engine of *refine* for dealing with existential variables, holes, and conversion problems. This may result in slightly different behavior regarding which conversion problems are solvable. However, like *apply* but unlike *eapply*, *rapply* will fail if there are any holes which remain in *term* itself after typechecking and typeclass resolution but before unification with

the goal. More technically, ***term*** is first parsed as a `constr` rather than as a `uconstr` or `open_constr` before being applied to the goal. Note that *rapply* prefers to instantiate as many hypotheses of ***term*** as possible. As a result, if it is possible to apply ***term*** to arbitrarily many arguments without getting a type error, *rapply* will loop.

Note that you need to **`Require Import Coq.Program.Tactics`** to make use of *rapply*.

Variant: `simple apply` *term*.

This behaves like *apply* but it reasons modulo conversion only on subterms that contain no variables to instantiate. For instance, the following example does not succeed because it would require the conversion of `id ?foo` and `O`.

Example

```
Definition id (x : nat) := x.
    id is defined

Parameter H : forall x y, id x = y.
    H is declared

Goal O = O.
    1 goal

      ============================
      0 = 0

Fail simple apply H.
    The command has indeed failed with message:
    Unable to unify "id ?M160 = ?M161" with "0 = 0".
```

Because it reasons modulo a limited amount of conversion, *simple apply* fails quicker than *apply* and it is then well-suited for uses in user-defined tactics that backtrack often. Moreover, it does not traverse tuples as *apply* does.

Variant: `simple` ? `apply` *term* `with` *bindings* ? + ,

Variant: `simple` ? `eapply` *term* `with` *bindings* ? + ,

This summarizes the different syntaxes for *apply* and *eapply*.

Variant: `lapply` *term*

This tactic applies to any goal, say `G`. The argument term has to be well-formed in the current context, its type being reducible to a non-dependent product `A -> B` with `B` possibly containing products. Then it generates two subgoals `B->G` and `A`. Applying `lapply H` (where `H` has type `A->B` and `B` does not start with a product) does the same as giving the sequence `cut B. 2:apply H.` where `cut` is described below.

Warning:
When *term* contains more than one non dependent product the tactic lapply only takes

---

Example

Assume we have a transitive relation `R` on `nat`:

```
Parameter R : nat -> nat -> Prop.

Axiom Rtrans : forall x y z:nat, R x y -> R y z -> R x z.

Parameters n m p : nat.

Axiom Rnm : R n m.

Axiom Rmp : R m p.
```

Consider the goal `(R n p)` provable using the transitivity of `R`:

```
Goal R n p.
```

The direct application of `Rtrans` with `apply` fails because no value for `y` in `Rtrans` is found by `apply`:

```
apply Rtrans.
    Toplevel input, characters 6-12:
    > apply Rtrans.
    >       ^^^^^^
    Error: Unable to find an instance for the variable y.
```

A solution is to `apply (Rtrans n m p)` or `(Rtrans n m)`.

```
apply (Rtrans n m p).
    2 goals

      ============================
      R n m

    goal 2 is:
     R m p
```

Note that `n` can be inferred from the goal, so the following would work too.

```
apply (Rtrans _ m).
```

More elegantly, `apply Rtrans with (y:=m)` allows only mentioning the unknown m:

```
apply Rtrans with (y := m).
```

Another solution is to mention the proof of `(R x y)` in `Rtrans`

```
apply Rtrans with (1 := Rnm).
    1 goal

      ============================
      R m p
```

... or the proof of `(R y z)`.

```
apply Rtrans with (2 := Rmp).
    1 goal

      ============================
      R n m
```

On the opposite, one can use `eapply` which postpones the problem of finding `m`. Then one can apply the hypotheses `Rnm` and `Rmp`. This instantiates the existential variable and completes the proof.

```
eapply Rtrans.
    2 focused goals
    (shelved: 1)

      ============================
      R n ?y

    goal 2 is:
     R ?y p


apply Rnm.
    1 goal

      ============================
      R m p


apply Rmp.
    No more goals.
```

Note: When the conclusion of the type of the term to `apply` is an inductive type isomorphic to a tuple type and `apply` looks recursively whether a component of the tuple matches the goal, it excludes components whose statement would result in applying an universal lemma of the form `forall A, ... -> A`. Excluding this kind of lemma can be avoided by setting the following flag:

Flag: `Universal Lemma Under Conjunction`
This *flag*, which preserves compatibility with versions of Coq prior to 8.4 is also available for **`apply`** *`term`* **`in`** *`ident`* (see *`apply … in`*).

Tactic: `apply` *`term`* `in` *`ident`*
This tactic applies to any goal. The argument *`term`* is a term well-formed in the local context and the argument *`ident`* is an hypothesis of the context. The tactic **`apply`** *`term`* **`in`** *`ident`* tries to match the conclusion of the type of *`ident`* against a non-dependent premise of the type of *`term`*, trying them from right to left. If it succeeds, the statement of hypothesis *`ident`* is replaced by the conclusion of the type of *`term`*. The tactic also returns as many subgoals as the number of other non-dependent premises in the type of *`term`* and of the non-dependent premises of the type of *`ident`*. If the conclusion of the type of *`term`* does not match the goal *and* the conclusion is an inductive type isomorphic to a tuple type, then the tuple is (recursively) decomposed and the first component of the tuple of which a non-dependent premise matches the conclusion of the type of *`ident`*. Tuples are decomposed in a width-first left-to-right order (for instance if the type of `H1` is `A <-> B` and the type of `H2` is `A` then `apply H1 in H2` transforms the type of `H2` into `B`). The tactic *`apply`* relies on first-order pattern matching with dependent types.

Error: `Statement without assumptions.`
This happens if the type of *`term`* has no non-dependent premise.

Error: `Unable to apply.`
This happens if the conclusion of *`ident`* does not match any of the non-dependent premises of the type of *`term`*.

Variant: `apply` *`term`*`+,` `in` *`ident`*`+,`
This applies each *`term`* in sequence in each hypothesis *`ident`*.

Variant: `apply` *`term`* `with` *`bindings`*`+,` `in` *`ident`*`+,`

This does the same but uses the bindings to instantiate parameters of *term* (see *Bindings*).

Variant: eapply *term* **with** *bindings*? , + **in** *ident* , +

This works as *apply … in* but turns unresolved bindings into existential variables, if any, instead of failing.

Variant: apply *term* **with** *bindings*? , + **in** *ident* **as** *simple_intropattern*? , +

This works as *apply … in* but applying an associated *simple_intropattern* to each hypothesis *ident* that comes with such clause.

Variant: simple apply *term* **in** *ident* , +

This behaves like *apply … in* but it reasons modulo conversion only on subterms that contain no variables to instantiate and does not traverse tuples. See *the corresponding example*.

Variant:

simple? **apply** *term* **with** *bindings*? , + **in** *ident* **as** *simple_intropattern*? , +

Variant:

simple? **eapply** *term* **with** *bindings*? , + **in** *ident* **as** *simple_intropattern*? , +

This summarizes the different syntactic variants of **apply** *term* **in** *ident* , + and **eapply** *term* **in** *ident* , +.

Debug `"tactic-unification"` enables printing traces of unification steps in tactic unification. Tactic unification is used in tactics such as *apply* and *rewrite*.

Managing the local context

Tactic: intro

This tactic applies to a goal that is either a product or starts with a let-binder. If the goal is a product, the tactic implements the "Lam" rule given in *Typing rules*[1]. If the goal starts with a let-binder, then the tactic implements a mix of the "Let" and "Conv".

If the current goal is a dependent product `forall x:T, U` (resp `let x:=t in U`) then *intro* puts `x:T` (resp `x:=t`) in the local context. The new subgoal is `U`.

If the goal is a non-dependent product $T \to U$, then it puts in the local context either `Hn:T` (if `T` is of type `Set` or `Prop`) or `Xn:T` (if the type of `T` is `Type`). The optional index `n` is such that `Hn` or `Xn` is a fresh identifier. In both cases, the new subgoal is `U`.

If the goal is an existential variable, *intro* forces the resolution of the existential variable into a dependent product $\forall$ `x:?X, ?Y`, puts `x:?X` in the local context and leaves `?Y` as a new subgoal allowed to depend on `x`.

The tactic *intro* applies the tactic *hnf* until *intro* can be applied or the goal is not head-reducible.

Error: No product even after head-reduction.

Variant: intro *ident*

This applies *intro* but forces *ident* to be the name of the introduced hypothesis.

Error: *ident* **is already used.**

Note: If a name used by intro hides the base name of a global constant then the latter can still be referred to by a qualified name (see *Qualified identifiers*).

Variant: `intros`
This repeats *intro* until it meets the head-constant. It never reduces head-constants and it never fails.

Variant: `intros` *ident*`+` `.`
This is equivalent to the composed tactic **`intro` *ident*`;` `...` `;` `intro` *ident***.

Variant: `intros until` *ident*
This repeats intro until it meets a premise of the goal having the form **`(`*ident* `:` *type*`)`** and discharges the variable named *ident* of the current goal.

Error: `No` `such` `hypothesis` `in` `current` `goal`.

Variant: `intros until` *natural*
This repeats *intro* until the *natural*-th non-dependent product.

Example

On the subgoal `forall x y : nat, x = y -> y = x` the tactic **`intros until 1`** is equivalent to **`intros x y H`**, as `x = y -> y = x` is the first non-dependent product.

On the subgoal `forall x y z : nat, x = y -> y = x` the tactic **`intros until 1`** is equivalent to **`intros x y z`** as the product on `z` can be rewritten as a non-dependent product: `forall x y : nat, nat -> x = y -> y = x`.

Error: `No` `such` `hypothesis` `in` `current` `goal`.
This happens when *natural* is 0 or is greater than the number of non-dependent products of the goal.

Variant: `intro` *ident*$_1$`?` `after` *ident*$_2$

Variant: `intro` *ident*$_1$`?` `before` *ident*$_2$

Variant: `intro` *ident*$_1$`?` `at top`

Variant: `intro` *ident*$_1$`?` `at bottom`

These tactics apply **`intro`** *ident*$_1$`?` and move the freshly introduced hypothesis respectively after the hypothesis *ident*$_2$, before the hypothesis *ident*$_2$, at the top of the local context, or at the bottom of the local context. All hypotheses on which the new hypothesis depends are moved too so as to respect the order of dependencies between hypotheses. It is equivalent to **`intro`** *ident*$_1$`?` followed by the appropriate call to *move*, *move … before …*, *move … at top*, or *move … at bottom*.

Note: **`intro at bottom`** is a synonym for **`intro`** with no argument.

Error: `No` `such` `hypothesis:` *ident*`.`

Tactic: `intros` *intropattern_list*
Introduces one or more variables or hypotheses from the goal by matching the intro patterns. See the description in *Intro patterns*.

[1] Actually, only the second subgoal will be generated since the other one can be automatically checked.

Tactic: **eintros** *intropattern_list*

Works just like *intros …* except that it creates existential variables for any unresolved variables rather than failing.

Tactic: **clear** *ident*

This tactic erases the hypothesis named *ident* in the local context of the current goal. As a consequence, *ident* is no more displayed and no more usable in the proof development.

Error: **No such hypothesis.**

Error: *ident* **is used in the conclusion.**

Error: *ident* **is used in the hypothesis** *ident*.

Variant: **clear** *ident* +

This is equivalent to **clear** *ident*. **...** **clear** *ident*.

Variant: **clear -** *ident* +

This variant clears all the hypotheses except the ones depending in the hypotheses named *ident* + and in the goal.

Variant: **clear**

This variants clears all the hypotheses except the ones the goal depends on.

Variant: **clear dependent** *ident*

This clears the hypothesis *ident* and all the hypotheses that depend on it.

Variant: **clearbody** *ident* +

This tactic expects *ident* + to be local definitions and clears their respective bodies. In other words, it turns the given definitions into assumptions.

Error: *ident* **is not a local definition.**

Tactic: **revert** *ident* +

This applies to any goal with variables *ident* +. It moves the hypotheses (possibly defined) to the goal, if this respects dependencies. This tactic is the inverse of *intro*.

Error: **No such hypothesis.**

Error: $ident_1$ **is used in the hypothesis** $ident_2$.

Variant: **revert dependent** *ident*

This moves to the goal the hypothesis *ident* and all the hypotheses that depend on it.

Tactic: **move** $ident_1$ **after** $ident_2$

This moves the hypothesis named $ident_1$ in the local context after the hypothesis named $ident_2$, where "after" is in reference to the direction of the move. The proof term is not changed.

If $ident_1$ comes before $ident_2$ in the order of dependencies, then all the hypotheses between $ident_1$ and $ident_2$ that (possibly indirectly) depend on $ident_1$ are moved too, and all of them are thus moved after $ident_2$ in the order of dependencies.

If $ident_1$ comes after $ident_2$ in the order of dependencies, then all the hypotheses between $ident_1$ and $ident_2$ that (possibly indirectly) occur in the type of $ident_1$ are moved too, and all of them are thus moved before $ident_2$ in the order of dependencies.

Variant: **move** $ident_1$ **before** $ident_2$

This moves $ident_1$ towards and just before the hypothesis named $ident_2$. As for *move*, dependencies

over *ident*$_1$ (when *ident*$_1$ comes before *ident*$_2$ in the order of dependencies) or in the type of *ident*$_1$ (when *ident*$_1$ comes after *ident*$_2$ in the order of dependencies) are moved too.

Variant: `move` *ident* `at top`
This moves *ident* at the top of the local context (at the beginning of the context).

Variant: `move` *ident* `at bottom`
This moves *ident* at the bottom of the local context (at the end of the context).

Error: `No such hypothesis.`

Error: `Cannot move` *ident*$_1$ `after` *ident*$_2$`: it occurs in the type of` *ident*$_2$`.`

Error: `Cannot move` *ident*$_1$ `after` *ident*$_2$`: it depends on` *ident*$_2$`.`

Example

```
Goal forall x :nat, x = 0 -> forall z y:nat, y=y-> 0=x.
    1 goal

      ============================
      forall x : nat, x = 0 -> nat -> forall y : nat, y = y -> 0 = x

  intros x H z y H0.
    1 goal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

  move x after H0.
    1 goal

      z, y : nat
      H0 : y = y
      x : nat
      H : x = 0
      ============================
      0 = x

  Undo.
    1 goal

      x : nat
      H : x = 0
      z, y : nat
      H0 : y = y
      ============================
      0 = x

  move x before H0.
    1 goal

      z, y, x : nat
      H : x = 0
      H0 : y = y
```

(continues on next page)

(continued from previous page)

```
    ============================
    0 = x

Undo.
  1 goal

    x : nat
    H : x = 0
    z, y : nat
    H0 : y = y
    ============================
    0 = x

move H0 after H.
  1 goal

    x, y : nat
    H0 : y = y
    H : x = 0
    z : nat
    ============================
    0 = x

Undo.
  1 goal

    x : nat
    H : x = 0
    z, y : nat
    H0 : y = y
    ============================
    0 = x

move H0 before H.
  1 goal

    x : nat
    H : x = 0
    y : nat
    H0 : y = y
    z : nat
    ============================
    0 = x
```

Tactic: `rename` *ident*$_1$ `into` *ident*$_2$

This renames hypothesis *ident*$_1$ into *ident*$_2$ in the current context. The name of the hypothesis in the proof-term, however, is left unchanged.

Variant: `rename` *ident*$_i$ `into` *ident*$_j$ (`+` `,`)

This renames the variables *ident*$_i$ into *ident*$_j$ in parallel. In particular, the target identifiers may contain identifiers that exist in the source context, as long as the latter are also renamed by the same tactic.

Error: `No such hypothesis.`

Error: *ident* `is already used.`

Tactic: **set** **(***ident* **:=** *term***)**
This replaces *term* by *ident* in the conclusion of the current goal and adds the new definition *ident* **:=** *term* to the local context.

If *term* has holes (i.e. subexpressions of the form "_"), the tactic first checks that all subterms matching the pattern are compatible before doing the replacement using the leftmost subterm matching the pattern.

Error: **The variable** *ident* **is already defined.**

Variant: **set** **(***ident* **:=** *term***)** **in** *goal_occurrences*
This notation allows specifying which occurrences of *term* have to be substituted in the context. The **in** *goal_occurrences* clause is an occurrence clause whose syntax and behavior are described in *goal occurrences*.

Variant: **set** **(***ident* *binder*`*` **:=** *term***)** **in** *goal_occurrences*`?`
This is equivalent to **set** **(***ident* **:=** **fun** *binder*`*` **=>** *term***)** **in** *goal_occurrences*`?`.

Variant: **set** *term* **in** *goal_occurrences*`?`
This behaves as **set** **(***ident* **:=** *term***)** **in** *goal_occurrences*`?` but *ident* is generated by Coq.

Variant: **eset** **(***ident* *binder*`*` **:=** *term***)** **in** *goal_occurrences*`?`

Variant: **eset** *term* **in** *goal_occurrences*`?`
While the different variants of *set* expect that no existential variables are generated by the tactic, *eset* removes this constraint. In practice, this is relevant only when *eset* is used as a synonym of *epose*, i.e. when the *term* does not occur in the goal.

Tactic: **remember** *term* **as** *ident*$_1$ **eqn:***naming_intropattern*`?`
This behaves as **set** **(***ident* **:=** *term***)** **in** *****, using a logical (Leibniz's) equality instead of a local definition. Use *naming_intropattern* to name or split up the new equation.

Variant:
remember *term* **as** *ident*$_1$ **eqn:***naming_intropattern*`?` **in** *goal_occurrences*
This is a more general form of *remember* that remembers the occurrences of *term* specified by an occurrence set.

Variant:
eremember *term* **as** *ident*$_1$ **eqn:***naming_intropattern*`?` **in** *goal_occurrences*`?`
While the different variants of *remember* expect that no existential variables are generated by the tactic, *eremember* removes this constraint.

Tactic: **pose** **(***ident* **:=** *term***)**
This adds the local definition *ident* **:=** *term* to the current context without performing any replacement in the goal or in the hypotheses. It is equivalent to **set** **(***ident* **:=** *term***)** **in** **|-**.

Variant: **pose** **(***ident* *binder*`*` **:=** *term***)**
This is equivalent to **pose** **(***ident* **:=** **fun** *binder*`*` **=>** *term***)**.

Variant: **pose** *term*
This behaves as **pose** **(***ident* **:=** *term***)** but *ident* is generated by Coq.

Variant: **epose** **(***ident* *binder*`*` **:=** *term***)**

Variant: epose *term*
While the different variants of *pose* expect that no existential variables are generated by the tactic, *epose* removes this constraint.

Tactic: decompose [*qualid*+] *term*
This tactic recursively decomposes a complex proposition in order to obtain atomic ones.

Example

```
Goal forall A B C:Prop, A /\ B /\ C \/ B /\ C \/ C /\ A -> C.
    1 goal

      ============================
      forall A B C : Prop, A /\ B /\ C \/ B /\ C \/ C /\ A -> C

  intros A B C H; decompose [and or] H.
    3 goals

      A, B, C : Prop
      H : A /\ B /\ C \/ B /\ C \/ C /\ A
      H1 : A
      H0 : B
      H3 : C
      ============================
      C

    goal 2 is:
     C
    goal 3 is:
     C

  all: assumption.
    No more goals.

Qed.
```

Note: *decompose* does not work on right-hand sides of implications or products.

Variant: decompose sum *term*
This decomposes sum types (like `or`).

Variant: decompose record *term*
This decomposes record types (inductive types with one constructor, like `and` and `exists` and those defined with the *Record* command.

Controlling the proof flow

Tactic: assert (*ident* : *type*)
This tactic applies to any goal. **assert (H : U)** adds a new hypothesis of name **H** asserting U to the current goal and opens a new subgoal U[2]. The subgoal U comes first in the list of subgoals remaining to prove.

Error: Not a proposition or a type.
Arises when the argument *type* is neither of type Prop, Set nor Type.

Variant: assert *type*
This behaves as **assert (*ident* : *type*)** but ***ident*** is generated by Coq.

Variant: assert *type* by *tactic*
This tactic behaves like *assert* but applies tactic to solve the subgoals generated by assert.

Error: Proof is not complete.

Variant: assert *type* as *simple_intropattern*
If **simple_intropattern** is an intro pattern (see *Intro patterns*), the hypothesis is named after this introduction pattern (in particular, if **simple_intropattern** is ***ident***, the tactic behaves like **assert (*ident* : *type*)**). If **simple_intropattern** is an action introduction pattern, the tactic behaves like **assert *type*** followed by the action done by this introduction pattern.

Variant: assert *type* as *simple_intropattern* by *tactic*
This combines the two previous variants of *assert*.

Variant: assert (*ident* := *term*)
This behaves as **assert (*ident* : *type*) by exact *term*** where *type* is the type of *term*. This is equivalent to using *pose proof*. If the head of term is *ident*, the tactic behaves as *specialize*.

Error: Variable *ident* is already declared.

Variant: eassert *type* as *simple_intropattern* by *tactic*
While the different variants of *assert* expect that no existential variables are generated by the tactic, *eassert* removes this constraint. This lets you avoid specifying the asserted statement completely before starting to prove it.

Variant: pose proof *term* as *simple_intropattern*?

This tactic behaves like **assert *type* as *simple_intropattern*? by exact *term*** where *type* is the type of *term*. In particular, **pose proof *term* as *ident*** behaves as **assert (*ident* := *term*)** and **pose proof *term* as *simple_intropattern*** is the same as applying the *simple_intropattern* to *term*.

Variant: epose proof *term* as *simple_intropattern*?
While *pose proof* expects that no existential variables are generated by the tactic, *epose proof* removes this constraint.

Variant: pose proof (*ident* := *term*)
This is an alternative syntax for **assert (*ident* := *term*)** and **pose proof *term* as *ident***, following the model of **pose (*ident* := *term*)** but dropping the value of *ident*.

Variant: epose proof (*ident* := *term*)
This is an alternative syntax for **eassert (*ident* := *term*)** and **epose proof *term* as *ident***, following the model of **epose (*ident* := *term*)** but dropping the value of *ident*.

Variant: enough (*ident* : *type*)
This adds a new hypothesis of name *ident* asserting *type* to the goal the tactic *enough* is applied to. A new subgoal stating *type* is inserted after the initial goal rather than before it as *assert* would do.

[2] This corresponds to the cut rule of sequent calculus.

Variant: enough *type*
This behaves like **enough (*ident* : *type*)** with the name *ident* of the hypothesis generated by Coq.

Variant: enough *type* as *simple_intropattern*
This behaves like **enough *type*** using *simple_intropattern* to name or destruct the new hypothesis.

Variant: enough (*ident* : *type*) by *tactic*

Variant: enough *type* [as *simple_intropattern*?] by *tactic*
This behaves as above but with *tactic* expected to solve the initial goal after the extra assumption *type* is added and possibly destructed. If the **as *simple_intropattern*** clause generates more than one subgoal, *tactic* is applied to all of them.

Variant: eenough *type* [as *simple_intropattern*?] [by *tactic*?]

Variant: eenough (*ident* : *type*) [by *tactic*?]
While the different variants of *enough* expect that no existential variables are generated by the tactic, *eenough* removes this constraint.

Variant: cut *type*
This tactic applies to any goal. It implements the non-dependent case of the "App" rule given in *Typing rules*. (This is Modus Ponens inference rule.) **cut U** transforms the current goal `T` into the two following subgoals: `U -> T` and `U`. The subgoal `U -> T` comes first in the list of remaining subgoal to prove.

Variant: specialize (*ident* [*term]) [as *simple_intropattern*?]**

Variant: specialize *ident* with *bindings* [as *simple_intropattern*?]
This tactic works on local hypothesis ***ident***. The premises of this hypothesis (either universal quantifications or non-dependent implications) are instantiated by concrete terms coming either from arguments [***term****] or from *Bindings*. In the first form the application to [***term****] can be partial. The first form is equivalent to **assert (*ident* := *ident* [*term**])**. In the second form, instantiation elements can also be partial. In this case the uninstantiated arguments are inferred by unification if possible or left quantified in the hypothesis otherwise. With the **as** clause, the local hypothesis ***ident*** is left unchanged and instead, the modified hypothesis is introduced as specified by the *simple_intropattern*. The name ***ident*** can also refer to a global lemma or hypothesis. In this case, for compatibility reasons, the behavior of *specialize* is close to that of *generalize*: the instantiated statement becomes an additional premise of the goal. The `as` clause is especially useful in this case to immediately introduce the instantiated statement as a local hypothesis.

Error: *ident* is used in hypothesis *ident*.

Error: *ident* is used in conclusion.

Tactic: generalize *term*
This tactic applies to any goal. It generalizes the conclusion with respect to some term.

Example

```
Show.
    1 goal

      x, y : nat
      ============================
      0 <= x + y + y

generalize (x + y + y).
    1 goal
```

(continues on next page)

(continued from previous page)

```
x, y : nat
============================
forall n : nat, 0 <= n
```

If the goal is `G` and `t` is a subterm of type `T` in the goal, then **`generalize t`** replaces the goal by `forall (x:T), G'` where `G'` is obtained from `G` by replacing all occurrences of `t` by `x`. The name of the variable (here `n`) is chosen based on `T`.

Variant: `generalize` *`term`* +

This is equivalent to **`generalize`** ***`term`***`; ... ;` **`generalize`** ***`term`***. Note that the sequence of term $_i$'s are processed from n to 1.

Variant: `generalize` *`term`* `at` *`natural`* +

This is equivalent to **`generalize`** ***`term`*** but it generalizes only over the specified occurrences of ***`term`*** (counting from left to right on the expression printed using the *`Printing All`* flag).

Variant: `generalize` *`term`* `as` *`ident`*

This is equivalent to **`generalize`** ***`term`*** but it uses ***`ident`*** to name the generalized hypothesis.

Variant: `generalize` *`term`* `at` *`natural`* + **`as` *`ident`*** + ,

This is the most general form of **`generalize`** that combines the previous behaviors.

Variant: `generalize dependent` *`term`*

This generalizes term but also *all* hypotheses that depend on ***`term`***. It clears the generalized hypotheses.

Tactic: `evar (`*`ident`*` : `*`term`*`)`

The **`evar`** tactic creates a new local definition named ***`ident`*** with type ***`term`*** in the context. The body of this binding is a fresh existential variable.

Tactic: `instantiate (`*`ident`*` := `*`term`*` )`

The instantiate tactic refines (see *`refine`*) an existential variable ***`ident`*** with the term ***`term`***. It is equivalent to **`only [ident]: refine`** ***`term`*** (preferred alternative).

Note: To be able to refer to an existential variable by name, the user must have given the name explicitly (see *Existential variables*).

Note: When you are referring to hypotheses which you did not name explicitly, be aware that Coq may make a different decision on how to name the variable in the current goal and in the context of the existential variable. This can lead to surprising behaviors.

Variant: `instantiate (`*`natural`*` := `*`term`*`)`

This variant selects an existential variable by its position. The ***`natural`*** argument is the position of the existential variable *from right to left* in the conclusion of the goal. (Use one of the variants below to select an existential variable in a hypothesis.) Counting starts at 1 and multiple occurrences of the same existential variable are counted multiple times. Because this variant is not robust to slight changes in the goal, its use is strongly discouraged.

Variant: `instantiate (` *`natural`*` := `*`term`*` ) in `*`ident`*
Variant: `instantiate (` *`natural`*` := `*`term`*` ) in ( value of `*`ident`*` )`
Variant: `instantiate (` *`natural`*` := `*`term`*` ) in ( type of `*`ident`*` )`

These allow to refer respectively to existential variables occurring in a hypothesis or in the body or the type of a local definition (named ***`ident`***).

Variant: `instantiate`
Without argument, the instantiate tactic tries to solve as many existential variables as possible, using information gathered from other tactics in the same tactical. This is automatically done after each complete tactic (i.e. after a dot in proof mode), but not, for example, between each tactic when they are sequenced by semicolons.

Tactic: `admit`
This tactic allows temporarily skipping a subgoal so as to progress further in the rest of the proof. A proof containing admitted goals cannot be closed with *`Qed`* but only with *`Admitted`*.

Variant: `give_up`
Synonym of *`admit`*.

Tactic: `absurd` *`term`*
This tactic applies to any goal. The argument term is any proposition `P` of type `Prop`. This tactic applies False elimination, that is it deduces the current goal from False, and generates as subgoals `~P` and `P`. It is very useful in proofs by cases, where some cases are impossible. In most cases, `P` or `~P` is one of the hypotheses of the local context.

Tactic: `contradiction`
This tactic applies to any goal. The contradiction tactic attempts to find in the current context (after all intros) a hypothesis that is equivalent to an empty inductive type (e.g. `False`), to the negation of a singleton inductive type (e.g. `True` or `x=x`), or two contradictory hypotheses.

Error: `No such assumption.`

Variant: `contradiction` *`ident`*
The proof of False is searched in the hypothesis named *`ident`*.

Tactic: `contradict` *`ident`*
This tactic allows manipulating negated hypothesis and goals. The name *`ident`* should correspond to a hypothesis. With **`contradict H`**, the current goal and context is transformed in the following way:

- H:¬A ⊢ B becomes ⊢ A
- H:¬A ⊢ ¬B becomes H: B ⊢ A
- H: A ⊢ B becomes ⊢ ¬A
- H: A ⊢ ¬B becomes H: B ⊢ ¬A

Tactic: `exfalso`
This tactic implements the "ex falso quodlibet" logical principle: an elimination of False is performed on the current goal, and the user is then required to prove that False is indeed provable in the current context. This tactic is a macro for **`elimtype False`**.

Checking properties of terms

Each of the following tactics acts as the identity if the check succeeds, and results in an error otherwise.

Tactic: `constr_eq` *`term term`*
This tactic checks whether its arguments are equal modulo alpha conversion, casts and universe constraints. It may unify universes.

Error: `Not equal.`

Error: `Not equal (due to universes).`

Tactic: `constr_eq_strict` *`term term`*
This tactic checks whether its arguments are equal modulo alpha conversion, casts and universe constraints. It does not add new constraints.

Error: `Not equal.`

Error: **Not equal (due to universes).**

Tactic: **unify** ***term term***
This tactic checks whether its arguments are unifiable, potentially instantiating existential variables.

Error: **Unable to unify** ***term*** **with** ***term*****.**

Variant: **unify** ***term term*** **with** ***ident***
Unification takes the transparency information defined in the hint database ***ident*** into account (see *the hints databases for auto and eauto*).

Tactic: **is_evar** ***term***
This tactic checks whether its argument is a current existential variable. Existential variables are uninstantiated variables generated by *eapply* and some other tactics.

Error: **Not an evar.**

Tactic: **has_evar** ***term***
This tactic checks whether its argument has an existential variable as a subterm. Unlike context patterns combined with is_evar, this tactic scans all subterms, including those under binders.

Error: **No evars.**

Tactic: **is_var** ***term***
This tactic checks whether its argument is a variable or hypothesis in the current local context.

Error: **Not a variable or hypothesis.**

Classical tactics

In order to ease the proving process, when the Classical module is loaded, a few more tactics are available. Make sure to load the module using the Require Import command.

Tactic: **classical_left**
Tactic: **classical_right**
These tactics are the analog of *left* and *right* but using classical logic. They can only be used for disjunctions. Use *classical_left* to prove the left part of the disjunction with the assumption that the negation of right part holds. Use *classical_right* to prove the right part of the disjunction with the assumption that the negation of left part holds.

Delaying solving unification constraints

Tactic: **solve_constraints**

Flag: **Solve Unification Constraints**
By default, after each tactic application, postponed typechecking unification problems are resolved using heuristics. Unsetting this *flag* disables this behavior, allowing tactics to leave unification constraints unsolved. Use the *solve_constraints* tactic at any point to solve the constraints.

Proof maintenance

Experimental. Many tactics, such as *intros*, can automatically generate names, such as "H0" or "H1" for a new hypothesis introduced from a goal. Subsequent proof steps may explicitly refer to these names. However, future versions of Coq may not assign names exactly the same way, which could cause the proof to fail because the new names don't match the explicit references in the proof.

The following "Mangle Names" settings let users find all the places where proofs rely on automatically generated names, which can then be named explicitly to avoid any incompatibility. These settings cause Coq to generate different names, producing errors for references to automatically generated names.

Flag: Mangle Names
When this *flag* is set (it is off by default), generated names use the prefix specified in the following option instead of the default prefix.

Option: Mangle Names Prefix *string*
This *option* specifies the prefix to use when generating names.

Performance-oriented tactic variants

Tactic: exact_no_check *term*
For advanced usage. Similar to *exact* ***term***, but as an optimization, it skips checking that ***term*** has the goal's type, relying on the kernel check instead. See *change_no_check* for more explanation.

Example

```
Goal False.
    1 goal

      ============================
      False

  exact_no_check I.
    No more goals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

Variant: vm_cast_no_check *term*
For advanced usage. Similar to *exact_no_check* ***term***, but additionally instructs the kernel to use *vm_compute* to compare the goal's type with the ***term***'s type.

Example

```
Goal False.
    1 goal

      ============================
      False

  vm_cast_no_check I.
    No more goals.
```

(continues on next page)

(continued from previous page)

```
Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

Variant: `native_cast_no_check` *term*

for advanced usage. similar to *`exact_no_check` **term***, but additionally instructs the kernel to use *`native_compute`* to compare the goal's type with the ***term***'s type.

Example

```
Goal False.
    1 goal

      ============================
      False

  native_cast_no_check I.
    No more goals.

Fail Qed.
    The command has indeed failed with message:
    The term "I" has type "True" while it is expected to have type "False".
```

3.1.3 Reasoning with equalities

There are multiple notions of equality in Coq:

- Leibniz equality is the standard way to define equality in Coq and the Calculus of Inductive Constructions, which is in terms of a binary relation, i.e. a binary function that returns a `Prop`. The standard library defines `eq` similar to this:

  ```
  Inductive eq {A : Type} (x : A) : A -> Prop := eq_refl : eq x x.
  ```

  The notation `x = y` represents the term `eq x y`. The notation `x = y :> A` gives the type of x and y explicitly.

- Setoid equality defines equality in terms of an equivalence relation. A setoid is a set that is equipped with an equivalence relation (see https://en.wikipedia.org/wiki/Setoid). These are needed to form a quotient set or quotient (see https://en.wikipedia.org/wiki/Equivalence_Class). In Coq, users generally work with setoids rather than constructing quotients, for which there is no specific support.

- Definitional equality is equality based on the *conversion rules*, which Coq can determine automatically. When two terms are definitionally equal, Coq knows it can replace one with the other, such as with *`change` `X with Y`*, among many other advantages. "*Convertible*" is another way of saying that two terms are definitionally equal.

Tactics for simple equalities

Tactic: `reflexivity`

For a goal with the form `forall open_binders ,? t = u`, verifies that `t` and `u` are *definitionally equal*, and if so, solves the goal (by applying `eq_refl`). If not, it fails.

The tactic may also be applied to goals with the form `forall open_binders ,? R term1 term2` where `R` is a reflexive relation registered with the `Equivalence` or `Reflexive` typeclasses. See *Class* and *Instance*.

Error: `The relation ident is not a declared reflexive relation. Maybe you need to require the Coq.Classes.RelationClasses library`

Tactic: `symmetry` *simple_occurrences*?

Changes a goal that has the form `forall open_binders ,? t = u` into `u = t`. *simple_occurrences* may be used to apply the change in the selected hypotheses and/or the conclusion.

The tactic may also be applied to goals with the form `forall open_binders ,? R term1 term2` where `R` is a symmetric relation registered with the `Equivalence` or `Symmetric` typeclasses. See *Class* and *Instance*.

Error: `The relation ident is not a declared symmetric relation. Maybe you need to require the Coq.Classes.RelationClasses library`

Tactic: `transitivity` *one_term*

Changes a goal that has the form `forall open_binders ,? t = u` into the two subgoals `t = one_term` and `one_term = u`.

The tactic may also be applied to goals with the form `forall open_binders ,? R term1 term2` where `R` is a transitive relation registered with the `Equivalence` or `Transitivity` typeclasses. See *Class* and *Instance*.

Tactic: `etransitivity`

This tactic behaves like *transitivity*, using a fresh evar instead of a concrete *one_term*.

Error: `The relation ident is not a declared transitive relation. Maybe you need to require the Coq.Classes.RelationClasses library`

Tactic: `f_equal`

For a goal with the form `f a1 ... an = g b1 ... bn`, creates subgoals `f = g` and `ai = bi` for the `n` arguments. Subgoals that can be proven by *reflexivity* or *congruence* are solved automatically.

Rewriting with Leibniz and setoid equality

Tactic: `rewrite` *oriented_rewriter* +, *occurrences*? `by` *ltac_expr3*?

oriented_rewriter ::= `->` | `<-`? *natural*? `?` | `!`? *one_term_with_bindings*

one_term_with_bindings ::= `>`? *one_term* with *bindings*?

Replaces subterms with other subterms that have been proven to be equal. The type of *one_term* must have the form:

forall *open_binders* **,** ? **EQ** *term*$_1$ *term*$_2$

where EQ is the Leibniz equality `eq` or a registered *setoid equality*. Note that **eq** *term*$_1$ *term*$_2$ is typically written with the infix notation *term*$_1$ **=** *term*$_2$. You must `Require Setoid` to use the tactic with a setoid equality or with *setoid rewriting*.

rewrite *one_term* finds subterms matching *term*$_1$ in the goal, and replaces them with *term*$_2$ (or the reverse if `<-` is given). Some of the variables x_i are solved by unification, and some of the types **A**$_1$ **, …, A**$_n$ may become new subgoals. *rewrite* won't find occurrences inside `forall` that refer to variables bound by the `forall`; use the more advanced *setoid_rewrite* if you want to find such occurrences.

oriented_rewriter + **,** The *oriented_rewriter*s are applied sequentially to the first goal generated by the previous *oriented_rewriter*. If any of them fail, the tactic fails.

-> | **<-** ? For `->` (the default), *term*$_1$ is rewritten into *term*$_2$. For `<-`, *term*$_2$ is rewritten into *term*$_1$.

natural ? | **?** | **!** ? *natural* is the number of rewrites to perform. If `?` is given, *natural* is the maximum number of rewrites to perform; otherwise *natural* is the exact number of rewrites to perform.

`?` (without *natural*) performs the rewrite as many times as possible (possibly zero times). This form never fails. `!` (without *natural*) performs the rewrite as many times as possible and at least once. The tactic fails if the requested number of rewrites can't be performed. *natural* **!** is equivalent to *natural*.

occurrences If *occurrences* specifies multiple occurrences, the tactic succeeds if any of them can be rewritten. If not specified, only the first occurrence in the conclusion is replaced.

Note: If **at** *occs_nums* is specified, rewriting is always done with *setoid rewriting*, even for Leibniz equality, which means that you must `Require Setoid` to use that form. However, note that *rewrite* (even when using setoid rewriting) and *setoid_rewrite* don't behave identically (as is noted above and below).

---

by *ltac_expr3* If specified, is used to resolve all side conditions generated by the tactic.

Note: For each selected hypothesis and/or the conclusion, *rewrite* finds the first matching subterm in depth-first search order. Only subterms identical to that first matched subterm are rewritten. If the `at` clause is specified, only these subterms are considered when counting occurrences. To select a different set of matching subterms, you can specify how some or all of the free variables are bound by using a `with` clause (see *one_term_with_bindings*).

For instance, if we want to rewrite the right-hand side in the following goal, this will not work:

```
1 goal

  x, y : nat
  ============================
  x + y = y + x
```

```
rewrite Nat.add_comm at 2.
    Toplevel input, characters 0-25:
    > rewrite Nat.add_comm at 2.
    > ^^^^^^^^^^^^^^^^^^^^^^^^^
    Error: Invalid occurrence number: 2.
```

One can explicitly specify how some variables are bound to match a different subterm:

```
rewrite Nat.add_comm with (m := x).
    1 goal

      x, y : nat
      ============================
      x + y = x + y
```

Note that the more advanced *setoid_rewrite* tactic behaves differently, and thus the number of occurrences available to rewrite may differ between the two tactics.

Error: Tactic failure: Setoid library not loaded.

Error: Cannot find a relation to rewrite.

Error: Tactic generated a subgoal identical to the original goal.

Error: Found no subterm matching *term* in *ident*.
Error: Found no subterm matching *term* in the current goal.
This happens if *term* does not occur in, respectively, the named hypothesis or the goal.

Tactic: erewrite *oriented_rewriter* [+ ,] *occurrences* [?] **by** *ltac_expr3* [?]

Works like *rewrite*, but turns unresolved bindings, if any, into existential variables instead of failing. It has the same parameters as *rewrite*.

Flag: Keyed Unification
This *flag* makes higher-order unification used by *rewrite* rely on a set of keys to drive unification. The subterms, considered as rewriting candidates, must start with the same key as the left- or right-hand side of the lemma given to rewrite, and the arguments are then unified up to full reduction.

Tactic:
rewrite * [**->** | **<-**] [?] *one_term* [**in** *ident*] [?] [**at** *rewrite_occs*] [?] [**by** *ltac_expr3*] [?]

Tactic: rewrite * [**->** | **<-**] [?] *one_term* **at** *rewrite_occs* **in** *ident* [**by** *ltac_expr3*] [?]

Tactic: rewrite_db *ident* [**in** *ident*] [?]

Tactic: replace one_term_{from} **with** one_term_{to} [*occurrences*] [?] [**by** *ltac_expr3*] [?]

Tactic: replace [**->** | **<-**] [?] one_term_{from} [*occurrences*] [?]
The first form replaces all free occurrences of one_term_{from} in the current goal with one_term_{to} and generates an equality one_term_{to} **=** one_term_{from} as a subgoal. (Note the generated equality is reversed with respect to the order of the two terms in the tactic syntax; see issue #13480[28].) This equality is automatically solved if it occurs among the hypotheses, or if its symmetric form occurs.

The second form, with `->` or no arrow, replaces one_term_{from} with $term_{to}$ using the first hypothesis whose type has the form one_term_{from} **=** $term_{to}$. If `<-` is given, the tactic uses the first hypothesis with the reverse form, i.e. $term_{to}$ **=** one_term_{from}.

occurrences The `type of` and `value of` forms are not supported. Note you must `Require Setoid` to use the `at` clause in *occurrences*.

by *ltac_expr3* Applies the *ltac_expr3* to solve the generated equality.

Error: Terms do not have convertible types.

Tactic: cutrewrite [**->** | **<-**] [?] *one_term* [**in** *ident*] [?]
Where *one_term* is an equality.

Deprecated since version 8.5: Use *replace* instead.

Tactic: **substitute** [-> | <-]? *one_term* **with** *bindings*?

Tactic: **subst** *ident**

For each *ident*, in order, for which there is a hypothesis in the form *ident* **=** *term* or *term* **=** *ident*, replaces *ident* with *term* everywhere in the hypotheses and the conclusion and clears *ident* and the hypothesis from the context. If there are multiple hypotheses that match the *ident*, the first one is used. If no *ident* is given, replacement is done for all hypotheses in the appropriate form in top to bottom order.

If *ident* is a local definition of the form *ident* **:=** *term*, it is also unfolded and cleared.

If *ident* is a section variable it must have no indirect occurrences in the goal, i.e. no global declarations implicitly depending on the section variable may be present in the goal.

Note: If the hypothesis is itself dependent in the goal, it is replaced by the proof of reflexivity of equality.

Flag: **Regular Subst Tactic**

This *flag* controls the behavior of *subst*. When it is activated (it is by default), *subst* also deals with the following corner cases:

- A context with ordered hypotheses *ident*$_1$ **=** *ident*$_2$ and *ident*$_1$ **=** **t**, or **t'** **=** *ident*$_1$ with `t'` not a variable, and no other hypotheses of the form *ident*$_2$ **=** **u** or **u** **=** *ident*$_2$; without the flag, a second call to subst would be necessary to replace *ident*$_2$ by `t` or `t'` respectively.
- The presence of a recursive equation which without the flag would be a cause of failure of *subst*.
- A context with cyclic dependencies as with hypotheses *ident*$_1$ **=** **f** *ident*$_2$ and *ident*$_2$ **=** **g** *ident*$_1$ which without the flag would be a cause of failure of *subst*.

Additionally, it prevents a local definition such as *ident* **:=** **t** from being unfolded which otherwise it would exceptionally unfold in configurations containing hypotheses of the form *ident* **=** **u**, or **u'** **=** *ident* with `u'` not a variable. Finally, it preserves the initial order of hypotheses, which without the flag it may break.

Error: **Cannot find any non-recursive equality over** *ident*.

Error:
Section variable *ident* **occurs implicitly in global declaration** *qualid* **present in hypot**

Error:
Section variable *ident* **occurs implicitly in global declaration** *qualid* **present in the c**

Raised when the variable is a section variable with indirect dependencies in the goal. If *ident* is a section variable, it must not have any indirect occurrences in the goal, i.e. no global declarations implicitly depending on the section variable may be present in the goal.

Tactic: **simple subst**

Tactic: **stepl** *one_term* **by** *ltac_expr*?

For chaining rewriting steps. It assumes a goal in the form **R** *term*$_1$ *term*$_2$ where `R` is a binary relation and relies on a database of lemmas of the form `forall x y z, R x y -> eq x z -> R z y` where `eq` is typically a setoid equality. The application of **stepl** *one_term* then replaces the goal by **R** *one_term* *term*$_2$ and adds a new goal stating **eq** *one_term* *term*$_1$.

If *ltac_expr* is specified, it is applied to the side condition.

[28] https://github.com/coq/coq/issues/13480

Command: **Declare Left Step** *one_term*
Adds *one_term* to the database used by *stepl*.

This tactic is especially useful for parametric setoids which are not accepted as regular setoids for *rewrite* and *setoid_replace* (see *Generalized rewriting*).

Tactic: **stepr** *one_term* **by** *ltac_expr*?
This behaves like *stepl* but on the right hand side of the binary relation. Lemmas are expected to be in the form `forall x y z, R x y -> eq y z -> R x z.`

Command: **Declare Right Step** *one_term*
Adds *term* to the database used by *stepr*.

Rewriting with definitional equality

Tactic: **change** *one_term*$_{from}$ **at** *occs_nums*? **with** *one_term*$_{to}$ *occurrences*?

Replaces terms with other *convertible* terms. If *one_term*$_{from}$ is not specified, then *one_term*$_{to}$ replaces the conclusion and/or the specified hypotheses. If *one_term*$_{from}$ is specified, the tactic replaces occurrences of *one_term*$_{to}$ within the conclusion and/or the specified hypotheses.

one_term$_{from}$ **at** *occs_nums*? **with** Replaces the occurrences of *one_term*$_{from}$ specified by *occs_nums* with *one_term*$_{to}$, provided that the two *one_term*s are convertible. *one_term*$_{from}$ may contain pattern variables such as `?x`, whose value which will substituted for `x` in *one_term*$_{to}$, such as in `change (f ?x ?y) with (g (x, y))` or `change (fun x => ?f x) with f.`

The `at … with …` form is deprecated in 8.14; use `with … at …` instead. For `at … with … in H |-`, use `with … in H at … |-`.

occurrences If `with` is not specified, *occurrences* must only specify entire hypotheses and/or the goal; it must not include any **at** *occs_nums* clauses.

Error: **Not convertible.**

Error: **Found an "at" clause without "with" clause**

Tactic: **now_show** *one_term*
A synonym for **change** *one_term*. It can be used to make some proof steps explicit when refactoring a proof script to make it readable.

See also:

Applying conversion rules

Tactic:

change_no_check *one_term*$_{from}$ **at** *occs_nums*? **with** *one_term*$_{to}$ *occurrences*?
For advanced usage. Similar to *change*, but as an optimization, it skips checking that *one_term*$_{to}$ is convertible with the goal or *one_term*$_{from}$.

Recall that the Coq kernel typechecks proofs again when they are concluded to ensure correctness. Hence, using *change* checks convertibility twice overall, while *change_no_check* can produce ill-typed terms, but checks convertibility only once. Hence, *change_no_check* can be useful to speed up certain proof scripts, especially if one knows by construction that the argument is indeed convertible to the goal.

In the following example, *change_no_check* replaces `False` with `True`, but *Qed* then rejects the proof, ensuring consistency.

Example

```
Goal False.
    1 goal

      ============================
      False

  change_no_check True.
    1 goal

      ============================
      True

  exact I.
    No more goals.

Qed.
    Toplevel input, characters 0-4:
    > Qed.
    > ^^^^
    Error:
    The term "I" has type "True" while it is expected to have type "False".
```

Example

```
Goal True -> False.
    1 goal

      ============================
      True -> False

  intro H.
    1 goal

      H : True
      ============================
      False

  change_no_check False in H.
    1 goal

      H : False
      ============================
      False

  exact H.
    No more goals.

Qed.
    Toplevel input, characters 0-4:
    > Qed.
    > ^^^^
    Error:
    The term "fun H : True => H" has type "True -> True"
```

(continues on next page)

(continued from previous page)

```
    while it is expected to have type "True -> False".
```

Applying conversion rules

These tactics apply reductions and expansions, replacing *convertible* subterms with others that are equal by definition in CIC. They implement different specialized uses of the `change` tactic. Other ways to apply these reductions are through the `Eval` command, the `Eval` clause in the `Definition`/`Example` command and the `eval` tactic.

Tactics described in this section include:

- `lazy` and `cbv`, which allow precise selection of which reduction rules to apply
- `simpl` and `cbn`, which are "clever" tactics meant to give the most readable result
- `hnf` and `red`, which apply reduction rules only to the head of the term
- `vm_compute` and `native_compute`, which are performance-oriented.

Conversion tactics, with two exceptions, only change the types and contexts of existential variables and leave the proof term unchanged. (The `vm_compute` and `native_compute` tactics change existential variables in a way similar to other conversions while also adding a single explicit cast to the proof term to tell the kernel which reduction engine to use. See *Type cast*.) For example:

```
Goal 3 + 4 = 7.
    1 goal

      ============================
      3 + 4 = 7

Show Proof.
    ?Goal

Show Existentials.
    Existential 1 = ?Goal : [ |- 3 + 4 = 7]

cbv.
    1 goal

      ============================
      7 = 7

Show Proof.
    ?Goal

Show Existentials.
    Existential 1 = ?Goal : [ |- 7 = 7]
```

Tactic: `lazy` *reductions*? *simple_occurrences*

Tactic: `cbv` *reductions*? *simple_occurrences*

reductions ::= *reduction*+
| *delta_reductions*
reduction ::= **beta**
| **delta** *delta_reductions*?
| **match**
| **fix**
| **cofix**
| **iota**
| **zeta**
delta_reductions ::= **-**? **[** *reference*+ **]**

Normalize the goal as specified by ***reductions***. If no reductions are specified by name, all reductions are applied. If any reductions are specified by name, then only the named reductions are applied. The reductions include:

beta *beta-reduction* of functional application

delta ***delta_reductions***? *delta-reduction*: unfolding of transparent constants, see *Controlling reduction strategies and the conversion algorithm*. The form in ***reductions*** without the keyword `delta` includes `beta`, `iota` and `zeta` reductions in addition to `delta` using the given ***delta_reductions***.

-? **[** ***reference***+ **]** without the −, limits delta unfolding to the listed constants. If the − is present, unfolding is applied to all constants that are not listed. Notice that the `delta` doesn't apply to variables bound by a let-in construction inside the term itself (use `zeta` to inline these). Opaque constants are never unfolded except by `vm_compute` and `native_compute` (see #4476[29] and *Controlling reduction strategies and the conversion algorithm*).

iota *iota-reduction* of pattern matching (`match`) over a constructed term and reduction of `fix` and `cofix` expressions. Shorthand for `match fix cofix`.

zeta *zeta-reduction*: reduction of *let-in definitions*

Normalization is done by first evaluating the head of the expression into weak-head normal form, i.e. until the evaluation is blocked by a variable, an opaque constant, an axiom, such as in **x** $\mathbf{u_1}$ … $\mathbf{u_n}$, `match x with … end`, `(fix f x {struct x} := …) x`, a constructed form (a λ-expression, constructor, cofixpoint, inductive type, product type or sort) or a redex for which flags prevent reduction of the redex. Once a weak-head normal form is obtained, subterms are recursively reduced using the same strategy.

There are two strategies for reduction to weak-head normal form: *lazy* (the `lazy` tactic), or *call-by-value* (the `cbv` tactic). The lazy strategy is a call by need[30] strategy, with sharing of reductions: the arguments of a function call are weakly evaluated only when necessary, and if an argument is used several times then it is weakly computed only once. This reduction is efficient for reducing expressions with dead code. For instance, the proofs of a proposition `exists x. P(x)` reduce to a pair of a witness `t` and a proof that `t` satisfies the predicate `P`. Most of the time, `t` may be computed without computing the proof of `P(t)`, thanks to the lazy strategy.

The call-by-value strategy is the one used in ML languages: the arguments of a function call are systematically weakly evaluated first. The lazy strategy is similar to how Haskell reduces terms. Although the lazy strategy always does fewer reductions than the call-by-value strategy, the latter is generally more efficient for evaluating purely computational expressions (i.e. with little dead code).

Tactic: compute ***delta_reductions***? ***simple_occurrences***
A variant form of `cbv`.

`Debug "Cbv"` makes `cbv` (and its derivative `compute`) print information about the constants it encounters and the unfolding decisions it makes.

Tactic:
simpl *delta_reductions*? *reference_occs* | *pattern_occs*? *simple_occurrences*

reference_occs ::= *reference* **at** *occs_nums*?

pattern_occs ::= *one_term* **at** *occs_nums*?

Reduces a term to something still readable instead of fully normalizing it. It performs a sort of strong normalization with two key differences:

- It unfolds constants only if they lead to an ι-reduction, i.e. reducing a match or unfolding a fixpoint.
- When reducing a constant unfolding to (co)fixpoints, the tactic uses the name of the constant the (co)fixpoint comes from instead of the (co)fixpoint definition in recursive calls.

simple_occurrences Permits selecting whether to reduce the conclusion and/or one or more hypotheses. While the `at` option of ***occurrences*** is not allowed here, ***reference_occs*** and ***pattern_occs*** have a somewhat less flexible `at` option for selecting specific occurrences.

simpl can unfold transparent constants whose name can be reused in recursive calls as well as those designated by *Arguments* ***reference*** **…** **/** commands. For instance, a constant `plus' := plus` may be unfolded and reused in recursive calls, but a constant such as `succ := plus (S O)` is not unfolded unless it was specifically designated in an *Arguments* command such as **Arguments succ /.**.

reference_occs | ***pattern_occs*** can limit the application of *simpl* to:

- applicative subterms whose *head* is the constant ***qualid*** or is the constant used in the notation ***string*** (see ***reference***)
- subterms matching a pattern ***one_term***

Tactic: cbn ***reductions***? ***simple_occurrences***

cbn was intended to be a more principled, faster and more predictable replacement for *simpl*.

The main difference between *cbn* and *simpl* is that *cbn* may unfold constants even when they cannot be reused in recursive calls: in the previous example, `succ t` is reduced to `S t`.

Debug `"RAKAM"` makes *cbn* print various debugging information. `RAKAM` is the Refolding Algebraic Krivine Abstract Machine.

Tactic: hnf ***simple_occurrences***

Replaces the current goal with its weak-head normal form according to the βδιζ-reduction rules, i.e. it reduces the *head* of the goal until it becomes a product or an irreducible term. All inner βι-redexes are also reduced. While *hnf* behaves similarly to *simpl* and *cbn*, unlike them, it does not recurse into subterms. The behavior of *hnf* can be tuned using the *Arguments* command.

Example: The term `fun n : nat => S n + S n` is not reduced by **hnf**.

Note: The δ rule only applies to transparent constants (see *Controlling reduction strategies and the conversion algorithm* on transparency and opacity).

Tactic: red ***simple_occurrences***

βιζ-reduces the constant at the head of `T` (which may be called the head constant; head means the beginning of the term), if possible, in the selected hypotheses and/or the goal, which must have the form:

[29] https://github.com/coq/coq/issues/4476
[30] https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_need

forall *open_binders*, **T**

(where T does not begin with a forall) to **c t_1 … t_n** where c is a constant. If c is transparent then it replaces c with its definition and reduces again until no further reduction is possible.

Error: No head constant to reduce.

Tactic: unfold *reference_occs* *occurrences*

Applies *delta-reduction* to the constants specified by each *reference_occs*. The selected hypotheses and/or goals are then reduced to βιζ-normal form. Use the general reduction tactics if you want to only apply the δ rule, for example *cbv* **delta [** *reference* **]**.

reference_occs If *reference* is a *qualid*, it must be a defined transparent constant or local definition (see *Top-level definitions* and *Controlling reduction strategies and the conversion algorithm*).

If *reference* is a *string* *scope_key* , the *string* is the discriminating symbol of a notation (e.g. "+") or an expression defining a notation (e.g. **"_ + _"**) and the notation is an application whose head symbol is an unfoldable constant, then the tactic unfolds it.

occurrences If *occurrences* is specified, the specified occurrences will be replaced in the selected hypotheses and/or goal. Otherwise every occurrence of the constants in the goal is replaced. If multiple *reference_occs* are given, any at clauses must be in the *reference_occs* rather than in *occurrences*.

Error: Cannot turn **inductive** | **constructor** **into an evaluable reference.**
Occurs when trying to unfold something that is defined as an inductive type (or constructor) and not as a definition.

Example

```
Goal 0 <= 1.
    1 goal

      ============================
      0 <= 1

unfold le.
    Toplevel input, characters 7-9:
    > unfold le.
    >        ^^
    Error: Cannot turn inductive le into an evaluable reference.
```

Error: *ident* **is opaque.**
Raised if you are trying to unfold a definition that has been marked opaque.

Example

```
Opaque Nat.add.
Goal 1 + 0 = 1.
    1 goal

      ============================
      1 + 0 = 1

unfold Nat.add.
```

(continues on next page)

(continued from previous page)

```
Toplevel input, characters 0-14:
> unfold Nat.add.
> ^^^^^^^^^^^^^^
Error: Nat.add is opaque.
```

Error: Bad occurrence number of *qualid*.

Error: *qualid* does not occur.

Tactic: `fold` *one_term*+ *simple_occurrences*

First, this tactic reduces each *one_term* using the *red* tactic. Then, every occurrence of the resulting terms in the selected hypotheses and/or goal will be replaced by its associated *one_term*. This tactic is particularly useful for reversing undesired unfoldings, which may make the goal very hard to read. The undesired unfoldings may be due to the limited capabilities of other reduction tactics. On the other hand, when an unfolded function applied to its argument has been reduced, the *fold* tactic doesn't do anything.

fold one_term_1 one_term_2 is equivalent to **fold** one_term_1**;** **fold** one_term_2.

Example: `fold` doesn't always undo `unfold`

```
Goal ~0=0.
    1 goal

      ============================
      0 <> 0

unfold not.
    1 goal

      ============================
      0 = 0 -> False
```

This *fold* doesn't undo the preceeding *unfold* (it makes no change):

```
fold not.
    1 goal

      ============================
      0 = 0 -> False
```

However, this *pattern* followed by *fold* does:

```
pattern (0 = 0).
    1 goal

      ============================
      (fun P : Prop => P -> False) (0 = 0)

fold not.
    1 goal

      ============================
      0 <> 0
```

Example: Use `fold` to reverse unfolding of `fold_right`

```
Goal forall x xs, fold_right and True (x::xs).
    1 goal

      ============================
      forall (x : Prop) (xs : list Prop), fold_right and True (x :: xs)

red.
    1 goal

      ============================
      forall (x : Prop) (xs : list Prop),
      x /\
      (fix fold_right (l : list Prop) : Prop :=
         match l with
         | nil => True
         | b :: t => b /\ fold_right t
         end) xs

fold (fold_right and True).
    1 goal

      ============================
      forall (x : Prop) (xs : list Prop), x /\ fold_right and True xs
```

Tactic: `pattern` *pattern_occs*`+,` *occurrences*`?`

Performs beta-expansion (the inverse of *beta-reduction*) for the selected hypotheses and/or goals. The *one_term*s in *pattern_occs* must be free subterms in the selected items. The expansion is done for each selected item `T` for a set of *one_term*s in the *pattern_occs* by:

- replacing all selected occurrences of the *one_term*s in `T` with fresh variables
- abstracting these variables
- applying the abstracted goal to the *one_term*s

For instance, if the current goal `T` is expressible as $\psi(t_1 \ldots t_n)$ where the notation captures all the instances of the t_i in φ, then *pattern* $t_1, \ldots, t_n$ generates the equivalent goal $(\mathtt{fun}\ (x_1:A_1) \ldots (x_n:A_n) \Rightarrow \psi(x_1 \ldots x_n))\ t_1 \ldots t_n$. If t_i occurs in one of the generated types A_j (for `j > i`), occurrences will also be considered and possibly abstracted.

This tactic can be used, for instance, when the tactic *apply* fails on matching or to better control the behavior of *rewrite*.

Fast reduction tactics: vm_compute and native_compute

vm_compute is a brute-force but efficient tactic that first normalizes the terms before comparing them. It is based on a bytecode representation of terms similar to the bytecode representation used in the ZINC virtual machine [Ler90]. It is especially useful for intensive computation of algebraic values, such as numbers, and for reflection-based tactics.

native_compute is based on on converting the Coq code to OCaml.

Note that both these tactics ignore *Opaque* markings (see issue #4776[31]), nor do they apply unfolding strategies such as from *Strategy*.

[31] https://github.com/coq/coq/issues/4776

native_compute is typically two to five times faster than *vm_compute* at applying conversion rules when Coq is running native code, but *native_compute* requires considerably more overhead. We recommend using *native_compute* when all of the following are true (otherwise use *vm_compute*):

- the running time in *vm_compute* at least 5-10 seconds
- the size of the input term is small (e.g. hand-generated code rather than automatically-generated code that may have nested destructs on inductives with dozens or hundreds of constructors)
- the output is small (e.g. you're returning a boolean, a natural number or an integer rather than a large abstract syntax tree)

These tactics change existential variables in a way similar to other conversions while also adding a single explicit cast (see *Type cast*) to the proof term to tell the kernel which reduction engine to use.

Tactic: vm_compute *reference_occs* | *pattern_occs*? *occurrences*?

Evaluates the goal using the optimized call-by-value evaluation bytecode-based virtual machine described in [GregoireL02]. This algorithm is dramatically more efficient than the algorithm used for the *cbv* tactic, but it cannot be fine-tuned. It is especially useful for full evaluation of algebraic objects. This includes the case of reflection-based tactics.

Tactic: native_compute *reference_occs* | *pattern_occs*? *occurrences*?

Evaluates the goal by compilation to OCaml as described in [BDenesGregoire11]. Depending on the configuration, this tactic can either default to *vm_compute*, recompile dependencies or fail due to some missing precompiled dependencies, see *the native-compiler option* for details.

Flag: NativeCompute Timing

This *flag* causes all calls to the native compiler to print timing information for the conversion to native code, compilation, execution, and reification phases of native compilation. Timing is printed in units of seconds of wall-clock time.

Flag: NativeCompute Profiling

On Linux, if you have the `perf` profiler installed, this *flag* makes it possible to profile *native_compute* evaluations.

Option: NativeCompute Profile Filename *string*

This *option* specifies the profile output; the default is `native_compute_profile.data`. The actual filename used will contain extra characters to avoid overwriting an existing file; that filename is reported to the user. That means you can individually profile multiple uses of *native_compute* in a script. From the Linux command line, run `perf report` on the profile file to see the results. Consult the `perf` documentation for more details.

Computing in a term: eval and Eval

Evaluation of a term can be performed with:

Tactic: `eval` *red_expr* `in` *term*

red_expr ::= **lazy** *reductions*?
| **cbv** *reductions*?
| **compute** *delta_reductions*?
| **vm_compute** *reference_occs* | *pattern_occs*?
| **native_compute** *reference_occs* | *pattern_occs*?
| **red**
| **hnf**
| **simpl** *delta_reductions*? *reference_occs* | *pattern_occs*?
| **cbn** *reductions*?
| **unfold** *reference_occs*+,
| **fold** *one_term*+
| **pattern** *pattern_occs*+,
| *ident*

eval is a *value_tactic*. It returns the result of applying the conversion rules specified by ***red_expr***. It does not change the proof state.

The ***red_expr*** alternatives that begin with a keyword correspond to the tactic with the same name, though in several cases with simpler syntax than the tactic. ***ident*** is a named reduction expression created with *Declare Reduction*.

See also:

Section *Applying conversion rules*.

Command: `Eval` *red_expr* `in` *term*

Performs the specified reduction on ***term*** and displays the resulting term with its type. If a proof is open, ***term*** may reference hypotheses of the selected goal. *Eval* is a *query_command*, so it may be prefixed with a goal selector.

Command: `Compute` *term*

Evaluates ***term*** using the bytecode-based virtual machine. It is a shortcut for *Eval* **`vm_compute in`** ***term***. *Compute* is a *query_command*, so it may be prefixed with a goal selector.

Command: `Declare Reduction` *ident* `:=` *red_expr*

Declares a short name for the reduction expression ***red_expr***, for instance `lazy beta delta [foo bar]`. This short name can then be used in **`Eval` *ident* `in`** or `eval` constructs. This command accepts the *local* attribute, which indicates that the reduction will be discarded at the end of the file or module. The name is not qualified. In particular declaring the same name in several modules or in several functor applications will be rejected if these declarations are not local. The name ***ident*** cannot be used directly as an Ltac tactic, but nothing prevents the user from also performing a **`Ltac` *ident* `:=` *red_expr***.

Controlling reduction strategies and the conversion algorithm

The commands to fine-tune the reduction strategies and the lazy conversion algorithm are described in this section. Also see *Effects of Arguments on unfolding*, which supports additional fine-tuning.

Command: **Opaque** *reference* +

Marks the specified constants as *opaque* so tactics won't *unfold* them with *delta-reduction*. "Constants" are items defined by commands such as `Definition`, `Let` (with an explicit body), `Fixpoint`, `CoFixpoint` and `Function`.

This command accepts the `global` attribute. By default, the scope of `Opaque` is limited to the current section or module.

`Opaque` also affects Coq's conversion algorithm, causing it to delay unfolding the specified constants as much as possible when it has to check that two distinct applied constants are convertible. See Section *Conversion rules*.

Command: **Transparent** *reference* +

The opposite of `Opaque`, it marks the specified constants as *transparent* so that tactics may unfold them. See `Opaque` above.

This command accepts the `global` attribute. By default, the scope of `Transparent` is limited to the current section or module.

Note that constants defined by proofs ending with `Qed` are irreversibly opaque; `Transparent` will not make them transparent. This is consistent with the usual mathematical practice of *proof irrelevance*: what matters in a mathematical development is the sequence of lemma statements, not their actual proofs. This distinguishes lemmas from the usual defined constants, whose actual values are of course relevant in general.

Error: **The reference** *qualid* **was not found in the current environment.**

There is no constant named *qualid* in the environment.

See also:

Applying conversion rules, `Qed` and `Defined`

Command: **Strategy** *strategy_level* **[** *reference* + **]** +

strategy_level ::= **opaque**
| *integer*
| **expand**
| **transparent**

Generalizes the behavior of the `Opaque` and `Transparent` commands. It is used to fine-tune the strategy for unfolding constants, both at the tactic level and at the kernel level. This command associates a *strategy_level* with the qualified names in the *reference* sequence. Whenever two expressions with two distinct head constants are compared (for example, typechecking `f x` where `f : A -> B` and `x : C` will result in converting `A` and `C`), the one with lower level is expanded first. In case of a tie, the second one (appearing in the cast type) is expanded.

This command accepts the `local` attribute, which limits its effect to the current section or module, in which case the section and module behavior is the same as `Opaque` and `Transparent` (without `global`).

Levels can be one of the following (higher to lower):

- `opaque` : level of opaque constants. They cannot be expanded by tactics (behaves like +∞, see next item).
- *integer* : levels indexed by an integer. Level 0 corresponds to the default behavior, which corresponds to transparent constants. This level can also be referred to as `transparent`. Negative levels correspond to constants to be expanded before normal transparent constants, while positive levels correspond to constants to be expanded after normal transparent constants.

- `expand` : level of constants that should be expanded first (behaves like $-\infty$)
- `transparent` : Equivalent to level 0

Command: `Print Strategy` *`reference`*
This command prints the strategy currently associated with ***`reference`***. It fails if ***`reference`*** is not an unfoldable reference, that is, neither a variable nor a constant.

Error: `The reference is not unfoldable.`

Command: `Print Strategies`
Print all the currently non-transparent strategies.

Tactic: `with_strategy` *`strategy_level_or_var`* `[` *`reference`*`+` `]` *`ltac_expr3`*

`strategy_level_or_var` ::= *`strategy_level`*
| *`ident`*

Executes *`ltac_expr3`*, applying the alternate unfolding behavior that the *`Strategy`* command controls, but only for *`ltac_expr3`*. This can be useful for guarding calls to reduction in tactic automation to ensure that certain constants are never unfolded by tactics like *`simpl`* and *`cbn`* or to ensure that unfolding does not fail.

Example

```
Opaque id.
Goal id 10 = 10.
    1 goal

      ============================
      id 10 = 10

Fail unfold id.
    The command has indeed failed with message:
    id is opaque.

with_strategy transparent [id] unfold id.
    1 goal

      ============================
      10 = 10
```

Warning: Use this tactic with care, as effects do not persist past the end of the proof script. Notably, this fine-tuning of the conversion strategy is not in effect during *`Qed`* nor *`Defined`*, so this tactic is most useful either in combination with *`abstract`*, which will check the proof early while the fine-tuning is still in effect, or to guard calls to conversion in tactic automation to ensure that, e.g., *`unfold`* does not fail just because the user made a constant *`Opaque`*.

This can be illustrated with the following example involving the factorial function.

```
Fixpoint fact (n : nat) : nat :=
  match n with
  | 0 => 1
  | S n' => n * fact n'
  end.
```

Suppose now that, for whatever reason, we want in general to unfold the `id` function very late during conversion:

```
Strategy 1000 [id].
```

If we try to prove `id (fact n) = fact n` by *reflexivity*, it will now take time proportional to $n!$, because Coq will keep unfolding `fact` and `*` and `+` before it unfolds `id`, resulting in a full computation of `fact n` (in unary, because we are using `nat`), which takes time $n!$. We can see this cross the relevant threshold at around $n = 9$:

```
Goal True.
    1 goal

      ============================
      True

Time assert (id (fact 8) = fact 8) by reflexivity.
    Finished transaction in 0.472 secs (0.319u,0.149s) (successful)
    1 goal

      H : id (fact 8) = fact 8
      ============================
      True

Time assert (id (fact 9) = fact 9) by reflexivity.
    Finished transaction in 2.699 secs (2.676u,0.s) (successful)
    1 goal

      H : id (fact 8) = fact 8
      H0 : id (fact 9) = fact 9
      ============================
      True
```

Note that behavior will be the same if you mark `id` as `Opaque` because while most reduction tactics refuse to unfold `Opaque` constants, conversion treats `Opaque` as merely a hint to unfold this constant last.

We can get around this issue by using *with_strategy*:

```
Goal True.
    1 goal

      ============================
      True

Fail Timeout 1 assert (id (fact 100) = fact 100) by reflexivity.
    The command has indeed failed with message:
    Timeout!

Time assert (id (fact 100) = fact 100) by with_strategy -1 [id] reflexivity.
    Finished transaction in 0.003 secs (0.003u,0.s) (successful)
    1 goal

      H : id (fact 100) = fact 100
      ============================
      True
```

However, when we go to close the proof, we will run into trouble, because the reduction strategy changes are local to the tactic passed to *with_strategy*.

```
exact I.
    No more goals.

Timeout 1 Defined.
    Toplevel input, characters 0-18:
    > Timeout 1 Defined.
    > ^^^^^^^^^^^^^^^^^^
    Error: Timeout!
```

We can fix this issue by using *abstract*:

```
Goal True.
    1 goal

      ============================
      True

Time assert (id (fact 100) = fact 100) by with_strategy -1 [id] abstract␣
↪reflexivity.
    Finished transaction in 0.005 secs (0.005u,0.s) (successful)
    1 goal

      H : id (fact 100) = fact 100
      ============================
      True

exact I.
    No more goals.

Time Defined.
    Finished transaction in 0.003 secs (0.003u,0.s) (successful)
```

On small examples this sort of behavior doesn't matter, but because Coq is a super-linear performance domain in so many places, unless great care is taken, tactic automation using *with_strategy* may not be robustly performant when scaling the size of the input.

Warning: In much the same way this tactic does not play well with *Qed* and *Defined* without using *abstract* as an intermediary, this tactic does not play well with `coqchk`, even when used with *abstract*, due to the inability of tactics to persist information about conversion hints in the proof term. See #12200[32] for more details.

3.1.4 Reasoning with inductive types

Applying constructors

The tactics presented here specialize *apply* (and *eapply*) to the case of constructors of inductive types.

Tactic: `constructor` *natural*

This tactic applies to a goal such that its conclusion is an inductive type (say `I`). The argument *natural* must be less or equal to the numbers of constructor(s) of `I`. Let $\mathbf{c_i}$ be the i-th constructor of `I`, then `constructor i` is equivalent to **`intros; apply`** $\mathbf{c_i}$.

[32] https://github.com/coq/coq/issues/12200

Error: **`Not an inductive product.`**

Error: **`Not enough constructors.`**

Variant: **`constructor`**
This tries `constructor 1` then `constructor 2`, ..., then `constructor n` where `n` is the number of constructors of the head of the goal.

Variant: **`constructor`** ***`natural`*** **`with`** ***`bindings`***
Let `c` be the i-th constructor of `I`, then **`constructor i with`** ***`bindings`*** is equivalent to **`intros; apply c with`** ***`bindings`***.

> **Warning:** The terms in *`bindings`* are checked in the context where constructor is executed and not in the context where *`apply`* is executed (the introductions are not taken into account).

Variant: **`split`** **`with`** ***`bindings`*** **?**
This applies only if `I` has a single constructor. It is then equivalent to **`constructor 1 with`** ***`bindings`*** **?**. It is typically used in the case of a conjunction $A \wedge B$.

Variant: **`exists`** ***`bindings`***
This applies only if `I` has a single constructor. It is then equivalent to **`intros; constructor 1 with`** ***`bindings`***. It is typically used in the case of an existential quantification $\exists x, P(x)$.

Variant: **`exists`** ***`bindings`*** **+ ,**
This iteratively applies **`exists`** ***`bindings`***.

Error: **`Not an inductive goal with 1 constructor.`**

Variant: **`left`** **`with`** ***`bindings`*** **?**

Variant: **`right`** **`with`** ***`bindings`*** **?**
These tactics apply only if `I` has two constructors, for instance in the case of a disjunction $A \vee B$. Then, they are respectively equivalent to **`constructor 1 with`** ***`bindings`*** **?** and **`constructor 2 with`** ***`bindings`*** **?**.

Error: **`Not an inductive goal with 2 constructors.`**

Variant: **`econstructor`**
Variant: **`eexists`**
Variant: **`esplit`**
Variant: **`eleft`**
Variant: **`eright`**
These tactics and their variants behave like *`constructor`*, *`exists`*, *`split`*, *`left`*, *`right`* and their variants but they introduce existential variables instead of failing when the instantiation of a variable cannot be found (cf. *`eapply`* and *`apply`*).

Case analysis and induction

The tactics presented in this section implement induction or case analysis on inductive or co-inductive objects (see *Theory of inductive definitions*).

Tactic: `destruct` *term*

This tactic applies to any goal. The argument *term* must be of inductive or co-inductive type and the tactic generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive or co-inductive type. Unlike *induction*, no induction hypothesis is generated by *destruct*.

Variant: `destruct` *ident*

If *ident* denotes a quantified variable of the conclusion of the goal, then **`destruct` *ident*** behaves as **`intros until` *ident*`;` `destruct` *ident***. If *ident* is not anymore dependent in the goal after application of *destruct*, it is erased (to avoid erasure, use parentheses, as in **`destruct (`*ident*`)`**).

If *ident* is a hypothesis of the context, and *ident* is not anymore dependent in the goal after application of *destruct*, it is erased (to avoid erasure, use parentheses, as in **`destruct (`*ident*`)`**).

Variant: `destruct` *natural*

`destruct` *natural* behaves as **`intros until` *natural*** followed by destruct applied to the last introduced hypothesis.

Note: For destruction of a number, use syntax **`destruct (`*natural*`)`** (not very interesting anyway).

Variant: `destruct` *pattern*

The argument of *destruct* can also be a pattern of which holes are denoted by "_". In this case, the tactic checks that all subterms matching the pattern in the conclusion and the hypotheses are compatible and performs case analysis using this subterm.

Variant: `destruct` *term*`+``,`

This is a shortcut for **`destruct` *term*`; ...;` `destruct` *term***.

Variant: `destruct` *term* `as` *or_and_intropattern_loc*

This behaves as **`destruct` *term*** but uses the names in *or_and_intropattern_loc* to name the variables introduced in the context. The *or_and_intropattern_loc* must have the form **`[p11 ... p1n | ... | pm1 ... pmn ]`** with `m` being the number of constructors of the type of *term*. Each variable introduced by *destruct* in the context of the `i`-th goal gets its name from the list **`pi1 ... pin`** in order. If there are not enough names, *destruct* invents names for the remaining variables to introduce. More generally, the **`pij`** can be any introduction pattern (see *intros*). This provides a concise notation for chaining destruction of a hypothesis.

Variant: `destruct` *term* `eqn:`*naming_intropattern*

This behaves as **`destruct` *term*** but adds an equation between *term* and the value that it takes in each of the possible cases. The name of the equation is specified by *naming_intropattern* (see *intros*), in particular `?` can be used to let Coq generate a fresh name.

Variant: `destruct` *term* `with` *bindings*

This behaves like **`destruct` *term*** providing explicit instances for the dependent premises of the type of *term*.

Variant: `edestruct` *term*

This tactic behaves like **`destruct` *term*** except that it does not fail if the instance of a dependent premises of the type of *term* is not inferable. Instead, the unresolved instances are left as existential variables to be inferred later, in the same way as *eapply* does.

Variant: **destruct** *term* **using** *term* **with** *bindings* ?

This is synonym of **induction** *term* **using** *term* **with** *bindings* ?.

Variant: **destruct** *term* **in** *goal_occurrences*

This syntax is used for selecting which occurrences of *term* the case analysis has to be done on. The **in** *goal_occurrences* clause is an occurrence clause whose syntax and behavior is described in *occurrences sets*.

Variant:

destruct *term* **with** *bindings* ? **as** *or_and_intropattern_loc* ? **eqn:***naming_intropattern* ?

Variant:

edestruct *term* **with** *bindings* ? **as** *or_and_intropattern_loc* ? **eqn:***naming_intropattern* ?

These are the general forms of *destruct* and *edestruct*. They combine the effects of the `with`, `as`, `eqn:`, `using`, and `in` clauses.

Tactic: **case** *term*

The tactic **case** is a more basic tactic to perform case analysis without recursion. It behaves as **elim** *term* but using a case-analysis elimination principle and not a recursive one.

Variant: **case** *term* **with** *bindings*

Analogous to **elim** *term* **with** *bindings* above.

Variant: **ecase** *term* **with** *bindings* ?

In case the type of *term* has dependent premises, or dependent premises whose values are not inferable from the **with** *bindings* clause, **ecase** turns them into existential variables to be resolved later on.

Variant: **simple destruct** *ident*

This tactic behaves as **intros until** *ident*; **case** *ident* when *ident* is a quantified variable of the goal.

Variant: **simple destruct** *natural*

This tactic behaves as **intros until** *natural*; **case** *ident* where *ident* is the name given by **intros until** *natural* to the *natural* -th non-dependent premise of the goal.

Variant: **case_eq** *term*

The tactic **case_eq** is a variant of the **case** tactic that allows to perform case analysis on a term without completely forgetting its original form. This is done by generating equalities between the original form of the term and the outcomes of the case analysis.

Tactic: **induction** *term*

This tactic applies to any goal. The argument *term* must be of inductive type and the tactic **induction** generates subgoals, one for each possible form of *term*, i.e. one for each constructor of the inductive type.

If the argument is dependent in either the conclusion or some hypotheses of the goal, the argument is replaced by the appropriate constructor form in each of the resulting subgoals and induction hypotheses are added to the local context using names whose prefix is **IH**.

There are particular cases:

- If term is an identifier *ident* denoting a quantified variable of the conclusion of the goal, then inductionident behaves as **intros until** *ident*; **induction** *ident*. If *ident* is not anymore dependent in the goal after application of **induction**, it is erased (to avoid erasure, use parentheses, as in **induction (***ident***)**).
- If *term* is a *natural*, then **induction** *natural* behaves as **intros until** *natural* followed by **induction** applied to the last introduced hypothesis.

Note: For simple induction on a number, use syntax induction (number) (not very interesting anyway).

- In case term is a hypothesis *ident* of the context, and *ident* is not anymore dependent in the goal after application of **induction**, it is erased (to avoid erasure, use parentheses, as in **induction (***ident***)**).
- The argument *term* can also be a pattern of which holes are denoted by "_". In this case, the tactic checks that all subterms matching the pattern in the conclusion and the hypotheses are compatible and performs induction using this subterm.

Example

```
Lemma induction_test : forall n:nat, n = n -> n <= n.
    1 goal

      ============================
      forall n : nat, n = n -> n <= n

intros n H.
    1 goal

      n : nat
      H : n = n
      ============================
      n <= n

induction n.
    2 goals

      H : 0 = 0
      ============================
      0 <= 0

    goal 2 is:
     S n <= S n

exact (le_n 0).
    1 goal

      n : nat
      H : S n = S n
      IHn : n = n -> n <= n
      ============================
      S n <= S n
```

Error: Not an inductive product.

Error: Unable to find an instance for the variables *ident* ... *ident*.
Use in this case the variant *elim … with* below.

Variant: induction *term* as *or_and_intropattern_loc*
This behaves as *induction* but uses the names in *or_and_intropattern_loc* to name the variables introduced in the context. The *or_and_intropattern_loc* must typically be of the form **[** $\mathbf{p}_{11}$ **...** $\mathbf{p}_{1n}$ **|** **...** **|** $\mathbf{p}_{m1}$ **...** $\mathbf{p}_{mn}$ **]** with **m** being the number of constructors of the type of *term*. Each variable introduced by induction in the context of the i-th goal gets its name from the list $\mathbf{p}_{i1}$ **...** $\mathbf{p}_{in}$ in order. If there are not enough names, induction invents names for the remaining variables to introduce. More generally, the $\mathbf{p}_{ij}$ can

be any disjunctive/conjunctive introduction pattern (see *intros …*). For instance, for an inductive type with one constructor, the pattern notation **(p$_1$, ... , p$_n$)** can be used instead of **[p$_1$ ... p$_n$]**.

Variant: induction *term* with *bindings*
This behaves like *induction* providing explicit instances for the premises of the type of **term** (see *Bindings*).

Variant: einduction *term*
This tactic behaves like *induction* except that it does not fail if some dependent premise of the type of ***term*** is not inferable. Instead, the unresolved premises are posed as existential variables to be inferred later, in the same way as *eapply* does.

Variant: induction *term* using *term*
This behaves as *induction* but using ***term*** as induction scheme. It does not expect the conclusion of the type of the first ***term*** to be inductive.

Variant: induction *term* using *term* with *bindings*
This behaves as *induction … using …* but also providing instances for the premises of the type of the second ***term***.

Variant: induction *term*$^{+,}$ using *qualid*
This syntax is used for the case ***qualid*** denotes an induction principle with complex predicates as the induction principles generated by `Function` or `Functional Scheme` may be.

Variant: induction *term* in *goal_occurrences*
This syntax is used for selecting which occurrences of ***term*** the induction has to be carried on. The **in *goal_occurrences*** clause is an occurrence clause whose syntax and behavior is described in *occurrences sets*. If variables or hypotheses not mentioning ***term*** in their type are listed in ***goal_occurrences***, those are generalized as well in the statement to prove.

Example

```
Lemma comm x y : x + y = y + x.
    1 goal

      x, y : nat
      ============================
      x + y = y + x

induction y in x |-   *.
    2 goals

      x : nat
      ============================
      x + 0 = 0 + x

    goal 2 is:
     x + S y = S y + x

Show 2.
    goal 2 is:

      x, y : nat
      IHy : forall x : nat, x + y = y + x
      ============================
      x + S y = S y + x
```

Variant:
induction *term* **with** *bindings* **as** *or_and_intropattern_loc* **using** *term* **with** *bindings* **in** *goal_o*
Variant:
einduction *term* **with** *bindings* **as** *or_and_intropattern_loc* **using** *term* **with** *bindings* **in** *goal_*
These are the most general forms of *induction* and *einduction*. It combines the effects of the with, as, using, and in clauses.

Variant: elim *term*
This is a more basic induction tactic. Again, the type of the argument *term* must be an inductive type. Then, according to the type of the goal, the tactic `elim` chooses the appropriate destructor and applies it as the tactic *apply* would do. For instance, if the local context contains `n:nat` and the current goal is `T` of type `Prop`, then **elim n** is equivalent to **apply nat_ind with (n:=n)**. The tactic `elim` does not modify the context of the goal, neither introduces the induction loading into the context of hypotheses. More generally, **elim** *term* also works when the type of *term* is a statement with premises and whose conclusion is inductive. In that case the tactic performs induction on the conclusion of the type of *term* and leaves the non-dependent premises of the type as subgoals. In the case of dependent products, the tactic tries to find an instance for which the elimination lemma applies and fails otherwise.

Variant: elim *term* **with** *bindings*
Allows to give explicit instances to the premises of the type of *term* (see *Bindings*).

Variant: eelim *term*
In case the type of *term* has dependent premises, this turns them into existential variables to be resolved later on.

Variant: elim *term* **using** *term*
Variant: elim *term* **using** *term* **with** *bindings*
Allows the user to give explicitly an induction principle *term* that is not the standard one for the underlying inductive type of *term*. The *bindings* clause allows instantiating premises of the type of *term*.

Variant: elim *term* **with** *bindings* **using** *term* **with** *bindings*
Variant: eelim *term* **with** *bindings* **using** *term* **with** *bindings*
These are the most general forms of *elim* and *eelim*. It combines the effects of the `using` clause and of the two uses of the `with` clause.

Variant: elimtype *type*
The argument *type* must be inductively defined. **elimtype I** is equivalent to **cut I. intro Hn; elim Hn; clear Hn.** Therefore the hypothesis `Hn` will not appear in the context(s) of the subgoal(s). Conversely, if `t` is a *term* of (inductive) type `I` that does not occur in the goal, then **elim t** is equivalent to **elimtype I; 2:exact t.**

Variant: simple induction *ident*
This tactic behaves as **intros until** *ident*; **elim** *ident* when *ident* is a quantified variable of the goal.

Variant: simple induction *natural*
This tactic behaves as **intros until** *natural*; **elim** *ident* where *ident* is the name given by **intros until** *natural* to the *natural*-th non-dependent premise of the goal.

Tactic: dependent induction *ident*
The *experimental* tactic dependent induction performs induction- inversion on an instantiated inductive predicate. One needs to first require the Coq.Program.Equality module to use this tactic. The tactic is based on the BasicElim tactic by Conor McBride [McB00] and the work of Cristina Cornes around inversion [CT95]. From an instantiated inductive predicate and a goal, it generates an equivalent goal where the hypothesis has been generalized over its indexes which are then constrained by equalities to be the right instances. This permits to state lemmas without resorting to manually adding these equalities and still get enough information in the proofs.

Example

```
Lemma lt_1_r : forall n:nat, n < 1 -> n = 0.
    1 goal

      ============================
      forall n : nat, n < 1 -> n = 0

intros n H ; induction H.
    2 goals

      n : nat
      ============================
      n = 0

    goal 2 is:
     n = 0
```

Here we did not get any information on the indexes to help fulfill this proof. The problem is that, when we use the `induction` tactic, we lose information on the hypothesis instance, notably that the second argument is 1 here. Dependent induction solves this problem by adding the corresponding equality to the context.

```
Require Import Coq.Program.Equality.
Lemma lt_1_r : forall n:nat, n < 1 -> n = 0.
    1 goal

      ============================
      forall n : nat, n < 1 -> n = 0

intros n H ; dependent induction H.
    2 goals

      ============================
      0 = 0

    goal 2 is:
     n = 0
```

The subgoal is cleaned up as the tactic tries to automatically simplify the subgoals with respect to the generated equalities. In this enriched context, it becomes possible to solve this subgoal.

```
reflexivity.
    1 goal

      n : nat
      H : S n <= 0
      IHle : 0 = 1 -> n = 0
      ============================
      n = 0
```

Now we are in a contradictory context and the proof can be solved.

```
inversion H.
    No more goals.
```

This technique works with any inductive predicate. In fact, the `dependent induction` tactic is just a wrapper around the `induction` tactic. One can make its own variant by just writing a new tactic based on the definition found in `Coq.Program.Equality`.

Variant: **dependent induction** *ident* **generalizing** *ident*+

This performs dependent induction on the hypothesis *ident* but first generalizes the goal by the given variables so that they are universally quantified in the goal. This is generally what one wants to do with the variables that are inside some constructors in the induction hypothesis. The other ones need not be further generalized.

Variant: **dependent destruction** *ident*

This performs the generalization of the instance *ident* but uses `destruct` instead of induction on the generalized hypothesis. This gives results equivalent to `inversion` or `dependent inversion` if the hypothesis is dependent.

See also the larger example of *dependent induction* and an explanation of the underlying technique.

See also:

functional induction

Tactic: **discriminate** *term*

This tactic proves any goal from an assumption stating that two structurally different *term*s of an inductive set are equal. For example, from `(S (S O))=(S O)` we can derive by absurdity any proposition.

The argument *term* is assumed to be a proof of a statement of conclusion *term* **=** *term* with the two terms being elements of an inductive set. To build the proof, the tactic traverses the normal forms[1] of the terms looking for a couple of subterms `u` and `w` (`u` subterm of the normal form of *term* and `w` subterm of the normal form of *term*), placed at the same positions and whose head symbols are two different constructors. If such a couple of subterms exists, then the proof of the current goal is completed, otherwise the tactic fails.

Note: The syntax **discriminate** *ident* can be used to refer to a hypothesis quantified in the goal. In this case, the quantified hypothesis whose name is *ident* is first introduced in the local context using **intros until** *ident*.

Error: **No primitive equality found.**

Error: **Not a discriminable equality.**

Variant: **discriminate** *natural*

This does the same thing as **intros until** *natural* followed by **discriminate** *ident* where *ident* is the identifier for the last introduced hypothesis.

Variant: **discriminate** *term* **with** *bindings*

This does the same thing as **discriminate** *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: **ediscriminate** *natural*

Variant: **ediscriminate** *term* **with** *bindings*?

This works the same as *discriminate* but if the type of *term*, or the type of the hypothesis referred to by *natural*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: **discriminate**

This behaves like **discriminate** *ident* if ident is the name of an hypothesis to which `discriminate` is applicable; if the current goal is of the form *term* **<>** *term*, this behaves as **intro** *ident*; **discriminate** *ident*.

Error: **No discriminable equalities.**

Tactic: **injection** *term*

The injection tactic exploits the property that constructors of inductive types are injective, i.e. that if `c` is a constructor of an inductive type and `c t1` and `c t2` are equal then `t1` and `t2` are equal too.

[1] Reminder: opaque constants will not be expanded by δ reductions.

If *term* is a proof of a statement of conclusion *term* = *term*, then *injection* applies the injectivity of constructors as deep as possible to derive the equality of all the subterms of *term* and *term* at positions where the terms start to differ. For example, from `(S p, S n) = (q, S (S m))` we may derive `S p = q` and `n = S m`. For this tactic to work, the terms should be typed with an inductive type and they should be neither convertible, nor having a different head constructor. If these conditions are satisfied, the tactic derives the equality of all the subterms at positions where they differ and adds them as antecedents to the conclusion of the current goal.

Example

Consider the following goal:

```
Inductive list : Set :=
| nil : list
| cons : nat -> list -> list.
Parameter P : list -> Prop.
Goal forall l n, P nil -> cons n l = cons 0 nil -> P l.

intros.
    1 goal

      l : list
      n : nat
      H : P nil
      H0 : cons n l = cons 0 nil
      ============================
      P l

injection H0.
    1 goal

      l : list
      n : nat
      H : P nil
      H0 : cons n l = cons 0 nil
      ============================
      l = nil -> n = 0 -> P l
```

Beware that injection yields an equality in a sigma type whenever the injected object has a dependent type `P` with its two instances in different types `(P` t_1 `...` t_n `)` and `(P` u_1 `...` u_n `)`. If t_1 and u_1 are the same and have for type an inductive type for which a decidable equality has been declared using *Scheme* **Equality ...** (see *Generation of induction principles with Scheme*), the use of a sigma type is avoided.

Note: If some quantified hypothesis of the goal is named *ident*, then **injection** *ident* first introduces the hypothesis in the local context using **intros until** *ident*.

Error: Nothing to do, it is an equality between convertible terms.

Error: Not a primitive equality.

Error: Nothing to inject.
This error is given when one side of the equality is not a constructor.

Variant: injection *natural*
This does the same thing as **intros until** *natural* followed by **injection** *ident* where *ident* is the identifier for the last introduced hypothesis.

Variant: **injection** *term* **with** *bindings*

This does the same as **injection** *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: **einjection** *natural*

Variant: **einjection** *term* **with** *bindings*?

This works the same as **injection** but if the type of *term*, or the type of the hypothesis referred to by *natural*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: **injection**

If the current goal is of the form *term* **<>** *term* , this behaves as **intro** *ident*; **injection** *ident*.

Error: **goal does not satisfy the expected preconditions.**

Variant: **injection** *term* **with** *bindings*? **as** *simple_intropattern*+

Variant: **injection** *natural* **as** *simple_intropattern*+

Variant: **injection as** *simple_intropattern*+

Variant: **einjection** *term* **with** *bindings*? **as** *simple_intropattern*+

Variant: **einjection** *natural* **as** *simple_intropattern*+

Variant: **einjection as** *simple_intropattern*+

These variants apply **intros** *simple_intropattern*+ after the call to *injection* or *einjection* so that all equalities generated are moved in the context of hypotheses. The number of *simple_intropattern* must not exceed the number of equalities newly generated. If it is smaller, fresh names are automatically generated to adjust the list of *simple_intropattern* to the number of new equalities. The original equality is erased if it corresponds to a hypothesis.

Variant: **injection** *term* **with** *bindings*? **as** *injection_intropattern*
Variant: **injection** *natural* **as** *injection_intropattern*
Variant: **injection as** *injection_intropattern*

Variant: **einjection** *term* **with** *bindings*? **as** *injection_intropattern*
Variant: **einjection** *natural* **as** *injection_intropattern*
Variant: **einjection as** *injection_intropattern*

These are equivalent to the previous variants but using instead the syntax *injection_intropattern* which *intros* uses. In particular **as** **[=** *simple_intropattern*+ **]** behaves the same as **as** *simple_intropattern*+ .

Flag: **Structural Injection**

This *flag* ensures that **injection** *term* erases the original hypothesis and leaves the generated equalities in the context rather than putting them as antecedents of the current goal, as if giving **injection** *term* **as** (with an empty list of names). This flag is off by default.

Flag: **Keep Proof Equalities**

By default, *injection* only creates new equalities between *term*s whose type is in sort `Type` or `Set`, thus implementing a special behavior for objects that are proofs of a statement in `Prop`. This *flag* controls this behavior.

Tactic: **inversion** *ident*

Let the type of *ident* in the local context be `(I t)`, where `I` is a (co)inductive predicate. Then, `inversion` applied to *ident* derives for each possible constructor `c i` of `(I t)`, all the necessary conditions that should

hold for the instance `(I t)` to be proved by `c i`.

Note: If *ident* does not denote a hypothesis in the local context but refers to a hypothesis quantified in the goal, then the latter is first introduced in the local context using **intros until** *ident*.

Note: As `inversion` proofs may be large in size, we recommend the user to stock the lemmas whenever the same instance needs to be inverted several times. See *Generation of inversion principles with Derive Inversion*.

Note: Part of the behavior of the `inversion` tactic is to generate equalities between expressions that appeared in the hypothesis that is being processed. By default, no equalities are generated if they relate two proofs (i.e. equalities between *term*s whose type is in sort `Prop`). This behavior can be turned off by using the *Keep Proof Equalities* setting.

Variant: inversion *natural*

This does the same thing as **intros until** *natural* then **inversion** *ident* where *ident* is the identifier for the last introduced hypothesis.

Variant: inversion_clear *ident*

This behaves as **inversion** and then erases *ident* from the context.

Variant: inversion *ident* **as** *or_and_intropattern_loc*

This generally behaves as inversion but using names in *or_and_intropattern_loc* for naming hypotheses. The *or_and_intropattern_loc* must have the form **[** $\mathbf{p}_{11}$ `...` $\mathbf{p}_{1n}$ `|` `...` `|` $\mathbf{p}_{m1}$ `...` $\mathbf{p}_{mn}$ **]** with `m` being the number of constructors of the type of *ident*. Be careful that the list must be of length `m` even if `inversion` discards some cases (which is precisely one of its roles): for the discarded cases, just use an empty list (i.e. `n = 0`).The arguments of the i-th constructor and the equalities that `inversion` introduces in the context of the goal corresponding to the i-th constructor, if it exists, get their names from the list $\mathbf{p}_{i1}$ `...` $\mathbf{p}_{in}$ in order. If there are not enough names, `inversion` invents names for the remaining variables to introduce. In case an equation splits into several equations (because `inversion` applies `injection` on the equalities it generates), the corresponding name $\mathbf{p}_{ij}$ in the list must be replaced by a sublist of the form **[** $\mathbf{p}_{ij1}$ `...` $\mathbf{p}_{ijq}$ **]** (or, equivalently, **(** $\mathbf{p}_{ij1}$ `,` `...,` $\mathbf{p}_{ijq}$ **)**) where `q` is the number of subequalities obtained from splitting the original equation. Here is an example. The `inversion ... as` variant of `inversion` generally behaves in a slightly more expectable way than `inversion` (no artificial duplication of some hypotheses referring to other hypotheses). To take benefit of these improvements, it is enough to use `inversion ... as []`, letting the names being finally chosen by Coq.

Example

```
Inductive contains0 : list nat -> Prop :=
| in_hd : forall l, contains0 (0 :: l)
| in_tl : forall l b, contains0 l -> contains0 (b :: l).
    contains0 is defined
    contains0_ind is defined
    contains0_sind is defined

Goal forall l:list nat, contains0 (1 :: l) -> contains0 l.
    1 goal

      ============================
      forall l : list nat, contains0 (1 :: l) -> contains0 l

intros l H; inversion H as [ | l' p Hl' [Heqp Heql'] ].
```

(continues on next page)

(continued from previous page)

```
1 goal

  l : list nat
  H : contains0 (1 :: l)
  l' : list nat
  p : nat
  Hl' : contains0 l
  Heqp : p = 1
  Heql' : l' = l
  ============================
  contains0 l
```

Variant: `inversion` *`natural`* `as` *`or_and_intropattern_loc`*
This allows naming the hypotheses introduced by **`inversion`** *`natural`* in the context.

Variant: `inversion_clear` *`ident`* `as` *`or_and_intropattern_loc`*
This allows naming the hypotheses introduced by `inversion_clear` in the context. Notice that hypothesis names can be provided as if `inversion` were called, even though the `inversion_clear` will eventually erase the hypotheses.

Variant: `inversion` *`ident`* `in` *`ident`*`+`

Let *`ident`*`+` be identifiers in the local context. This tactic behaves as generalizing *`ident`*`+`, and then performing `inversion`.

Variant: `inversion` *`ident`* `as` *`or_and_intropattern_loc`* `in` *`ident`*`+`

This allows naming the hypotheses introduced in the context by **`inversion`** *`ident`* **`in`** *`ident`*`+`.

Variant: `inversion_clear` *`ident`* `in` *`ident`*`+`

Let *`ident`*`+` be identifiers in the local context. This tactic behaves as generalizing *`ident`*`+`, and then performing `inversion_clear`.

Variant: `inversion_clear` *`ident`* `as` *`or_and_intropattern_loc`* `in` *`ident`*`+`

This allows naming the hypotheses introduced in the context by **`inversion_clear`** *`ident`* **`in`** *`ident`*`+`.

Variant: `dependent inversion` *`ident`*
That must be used when *`ident`* appears in the current goal. It acts like `inversion` and then substitutes *`ident`* for the corresponding **`@`***`term`* in the goal.

Variant: `dependent inversion` *`ident`* `as` *`or_and_intropattern_loc`*
This allows naming the hypotheses introduced in the context by **`dependent inversion`** *`ident`*.

Variant: `dependent inversion_clear` *`ident`*
Like `dependent inversion`, except that *`ident`* is cleared from the local context.

Variant: `dependent inversion_clear` *`ident`* `as` *`or_and_intropattern_loc`*
This allows naming the hypotheses introduced in the context by **`dependent inversion_clear`** *`ident`*.

Variant: `dependent inversion` *`ident`* `with` *`term`*
This variant allows you to specify the generalization of the goal. It is useful when the system fails to generalize the goal automatically. If *`ident`* has type `(I t)` and `I` has type `forall (x:T), s`, then *`term`* must be of type `I:forall (x:T), I x -> s'` where `s'` is the type of the goal.

Variant: `dependent inversion` *`ident`* `as` *`or_and_intropattern_loc`* `with` *`term`*
This allows naming the hypotheses introduced in the context by **`dependent inversion`** *`ident`* **`with`**

term.

Variant: dependent inversion_clear *ident* with *term*
Like *dependent inversion … with …* with but clears *ident* from the local context.

Variant: dependent inversion_clear *ident* as *or_and_intropattern_loc* with *term*
This allows naming the hypotheses introduced in the context by **dependent inversion_clear *ident* with *term***.

Variant: simple inversion *ident*
It is a very primitive inversion tactic that derives all the necessary equalities but it does not simplify the constraints as `inversion` does.

Variant: simple inversion *ident* as *or_and_intropattern_loc*
This allows naming the hypotheses introduced in the context by `simple inversion`.

Tactic: inversion *ident* using *ident*
Let *ident* have type `(I t)` (`I` an inductive predicate) in the local context, and *ident* be a (dependent) inversion lemma. Then, this tactic refines the current goal with the specified lemma.

Variant: inversion *ident* using *ident* in *ident* +

This tactic behaves as generalizing *ident* +, then doing **inversion *ident* using *ident***.

Variant: inversion_sigma *ident* **as** *simple_intropattern* ? ?
This tactic turns equalities of dependent pairs (e.g., `existT P x p = existT P y q`, frequently left over by inversion on a dependent type family) into pairs of equalities (e.g., a hypothesis `H : x = y` and a hypothesis of type `rew H in p = q`); these hypotheses can subsequently be simplified using *subst*, without ever invoking any kind of axiom asserting uniqueness of identity proofs. If you want to explicitly specify the hypothesis to be inverted, you can pass it as an argument to *inversion_sigma*. This tactic also works for `sig`, `sigT2`, `sig2`, `ex`, and `ex2` and there are similar `eq_sig ***_rect` induction lemmas.

Error:
Type of *ident* is not an equality of recognized Σ types: expected one of sig sig2 sigT s
When applied to a hypothesis, *inversion_sigma* can only handle equalities of the listed sigma types.

Error: *ident* is not an equality of Σ types
When applied to a hypothesis, *inversion_sigma* can only be called on hypotheses that are equalities using `Coq.Logic.Init.eq`.

Example

Non-dependent inversion.

Let us consider the relation `Le` over natural numbers:

```
Inductive Le : nat -> nat -> Set :=
| LeO : forall n:nat, Le 0 n
| LeS : forall n m:nat, Le n m -> Le (S n) (S m).
```

Let us consider the following goal:

```
1 goal

  P : nat -> nat -> Prop
  Q : forall n m : nat, Le n m -> Prop
  n, m : nat
  H : Le (S n) m
```

(continues on next page)

(continued from previous page)

```
  ============================
  P n m
```

To prove the goal, we may need to reason by cases on `H` and to derive that `m` is necessarily of the form `(S m0)` for certain `m0` and that `(Le n m0)`. Deriving these conditions corresponds to proving that the only possible constructor of `(Le (S n) m)` is `LeS` and that we can invert the arrow in the type of `LeS`. This inversion is possible because `Le` is the smallest set closed by the constructors `LeO` and `LeS`.

```
inversion_clear H.
    1 goal

  P : nat -> nat -> Prop
  Q : forall n m : nat, Le n m -> Prop
  n, m, m0 : nat
  H0 : Le n m0
  ============================
  P n (S m0)
```

Note that `m` has been substituted in the goal for `(S m0)` and that the hypothesis `(Le n m0)` has been added to the context.

Sometimes it is interesting to have the equality `m = (S m0)` in the context to use it after. In that case we can use *inversion* that does not clear the equalities:

```
inversion H.
    1 goal

  P : nat -> nat -> Prop
  Q : forall n m : nat, Le n m -> Prop
  n, m : nat
  H : Le (S n) m
  n0, m0 : nat
  H1 : Le n m0
  H0 : n0 = n
  H2 : S m0 = m
  ============================
  P n (S m0)
```

Example

Dependent inversion.

Let us consider the following goal:

```
    1 goal

  P : nat -> nat -> Prop
  Q : forall n m : nat, Le n m -> Prop
  n, m : nat
  H : Le (S n) m
  ============================
  Q (S n) m H
```

As `H` occurs in the goal, we may want to reason by cases on its structure and so, we would like inversion tactics to substitute `H` by the corresponding @term in constructor form. Neither *inversion* nor *inversion_clear* do such a substitution. To have such a behavior we use the dependent inversion tactics:

```
dependent inversion_clear H.
    1 goal

      P : nat -> nat -> Prop
      Q : forall n m : nat, Le n m -> Prop
      n, m, m0 : nat
      l : Le n m0
      ============================
      Q (S n) (S m0) (LeS n m0 l)
```

Note that `H` has been substituted by `(LeS n m0 l)` and `m` by `(S m0)`.

Example

Using inversion_sigma.

Let us consider the following inductive type of length-indexed lists, and a lemma about inverting equality of cons:

```
Require Import Coq.Logic.Eqdep_dec.

Inductive vec A : nat -> Type :=
| nil : vec A O
| cons {n} (x : A) (xs : vec A n) : vec A (S n).
    vec is defined
    vec_rect is defined
    vec_ind is defined
    vec_rec is defined
    vec_sind is defined


Lemma invert_cons : forall A n x xs y ys,
         @cons A n x xs = @cons A n y ys
         -> xs = ys.
    1 goal

      ============================
      forall (A : Type) (n : nat) (x : A) (xs : vec A n) (y : A) (ys : vec A n),
      cons A x xs = cons A y ys -> xs = ys


Proof.
intros A n x xs y ys H.
    1 goal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      ============================
      xs = ys
```

After performing inversion, we are left with an equality of existTs:

```
inversion H.
    1 goal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H2 : existT (fun n : nat => vec A n) n xs =
           existT (fun n : nat => vec A n) n ys
      ============================
      xs = ys
```

We can turn this equality into a usable form with inversion_sigma:

```
inversion_sigma.
    1 goal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H2_ : n = n
      H2_0 : eq_rect n (fun n : nat => vec A n) xs n H2_ = ys
      ============================
      xs = ys
```

To finish cleaning up the proof, we will need to use the fact that that all proofs of n = n for n a nat are eq_refl:

```
let H := match goal with H : n = n |- _ => H end in
pose proof (Eqdep_dec.UIP_refl_nat _ H); subst H.
    1 goal

      A : Type
      n : nat
      x : A
      xs : vec A n
      y : A
      ys : vec A n
      H : cons A x xs = cons A y ys
      H1 : x = y
      H2_0 : eq_rect n (fun n : nat => vec A n) xs n eq_refl = ys
      ============================
      xs = ys

simpl in *.
    1 goal

      A : Type
      n : nat
      x : A
```

(continues on next page)

(continued from previous page)

```
xs : vec A n
y : A
ys : vec A n
H : cons A x xs = cons A y ys
H1 : x = y
H2_0 : xs = ys
============================
xs = ys
```

Finally, we can finish the proof:

```
assumption.
    No more goals.

Qed.
```

See also:

functional inversion

Tactic: `fix` *ident natural*

This tactic is a primitive tactic to start a proof by induction. In general, it is easier to rely on higher-level induction tactics such as the ones described in *induction*.

In the syntax of the tactic, the identifier ***ident*** is the name given to the induction hypothesis. The natural number ***natural*** tells on which premise of the current goal the induction acts, starting from 1, counting both dependent and non dependent products, but skipping local definitions. Especially, the current lemma must be composed of at least ***natural*** products.

Like in a fix expression, the induction hypotheses have to be used on structurally smaller arguments. The verification that inductive proof arguments are correct is done only at the time of registering the lemma in the global environment. To know if the use of induction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section *Requesting information*).

Variant: `fix` *ident natural* `with` `(`*ident* *binder*`+` `[{struct` *ident*`}]` `:` *type*`)``+`

This starts a proof by mutual induction. The statements to be simultaneously proved are respectively `forall binder ... binder, type`. The identifiers ***ident*** are the names of the induction hypotheses. The identifiers ***ident*** are the respective names of the premises on which the induction is performed in the statements to be simultaneously proved (if not given, the system tries to guess itself what they are).

Tactic: `cofix` *ident*

This tactic starts a proof by coinduction. The identifier ***ident*** is the name given to the coinduction hypothesis. Like in a cofix expression, the use of induction hypotheses have to guarded by a constructor. The verification that the use of co-inductive hypotheses is correct is done only at the time of registering the lemma in the global environment. To know if the use of coinduction hypotheses is correct at some time of the interactive development of a proof, use the command `Guarded` (see Section *Requesting information*).

Variant: `cofix` *ident* `with` `(`*ident* *binder*`+` `:` *type*`)``+`

This starts a proof by mutual coinduction. The statements to be simultaneously proved are respectively `forall binder ... binder, type` The identifiers ***ident*** are the names of the coinduction hypotheses.

Equality and inductive sets

We describe in this section some special purpose tactics dealing with equality and inductive sets or types. These tactics use the equality `eq:forall (A:Type), A->A->Prop`, simply written with the infix symbol =.

Tactic: `decide equality`
This tactic solves a goal of the form `forall x y : R, {x = y} + {~ x = y}`, where `R` is an inductive type such that its constructors do not take proofs or functions as arguments, nor objects in dependent types. It solves goals of the form `{x = y} + {~ x = y}` as well.

Tactic: `compare` *term term*
This tactic compares two given objects *term* and *term* of an inductive datatype. If `G` is the current goal, it leaves the sub- goals *term* `=` *term* `-> G` and `~` *term* `=` *term* `-> G`. The type of *term* and *term* must satisfy the same restrictions as in the tactic `decide equality`.

Tactic: `simplify_eq` *term*
Let *term* be the proof of a statement of conclusion *term* `=` *term*. If *term* and *term* are structurally different (in the sense described for the tactic *discriminate*), then the tactic `simplify_eq` behaves as `discriminate` *term*, otherwise it behaves as `injection` *term*.

Note: If some quantified hypothesis of the goal is named *ident*, then `simplify_eq` *ident* first introduces the hypothesis in the local context using `intros until` *ident*.

Variant: `simplify_eq` *natural*
This does the same thing as `intros until` *natural* then `simplify_eq` *ident* where *ident* is the identifier for the last introduced hypothesis.

Variant: `simplify_eq` *term* `with` *bindings*
This does the same as `simplify_eq` *term* but using the given bindings to instantiate parameters or hypotheses of *term*.

Variant: `esimplify_eq` *natural*

Variant: `esimplify_eq` *term* `with` *bindings* ?
This works the same as *simplify_eq* but if the type of *term*, or the type of the hypothesis referred to by *natural*, has uninstantiated parameters, these parameters are left as existential variables.

Variant: `simplify_eq`
If the current goal has form `t1 <> t2`, it behaves as `intro` *ident*`;` `simplify_eq` *ident*.

Tactic: `dependent rewrite ->` *ident*
This tactic applies to any goal. If *ident* has type `(existT B a b)=(existT B a' b')` in the local context (i.e. each *term* of the equality has a sigma type `{ a:A & (B a)}`) this tactic rewrites `a` into `a'` and `b` into `b'` in the current goal. This tactic works even if `B` is also a sigma type. This kind of equalities between dependent pairs may be derived by the *injection* and *inversion* tactics.

Variant: `dependent rewrite <-` *ident*
Analogous to *dependent rewrite ->* but uses the equality from right to left.

Generation of induction principles with `Scheme`

Command: `Scheme` *ident* `:=`? *scheme_kind* `with` *ident* `:=`? *scheme_kind* *

scheme_kind ::= **Equality for** *reference*
| **Induction** | **Minimality** | **Elimination** | **Case** **for** *reference*
Sort *sort_family*

sort_family ::= **Set**
| **Prop**
| **SProp**
| **Type**

A high-level tool for automatically generating (possibly mutual) induction principles for given types and sorts. Each *reference* is a different inductive type identifier belonging to the same package of mutual inductive definitions. The command generates the *ident*s as mutually recursive definitions. Each term *ident* proves a general principle of mutual induction for objects in type *reference*.

ident The name of the scheme. If not provided, the scheme name will be determined automatically from the sorts involved.

Minimality for *reference* **Sort** *sort_family* Defines a non-dependent elimination principle more natural for inductively defined relations.

Equality for *reference* Tries to generate a Boolean equality and a proof of the decidability of the usual equality. If *reference* involves other inductive types, their equality has to be defined first.

Example

Induction scheme for tree and forest.

A mutual induction principle for tree and forest in sort `Set` can be defined using the command

```
Inductive tree : Set := node : A -> forest -> tree
with forest : Set :=
    leaf : B -> forest
  | cons : tree -> forest -> forest.
    tree, forest are defined
    tree_rect is defined
    tree_ind is defined
    tree_rec is defined
    tree_sind is defined
    forest_rect is defined
    forest_ind is defined
    forest_rec is defined
    forest_sind is defined


Scheme tree_forest_rec := Induction for tree Sort Set
  with forest_tree_rec := Induction for forest Sort Set.
    forest_tree_rec is defined
    tree_forest_rec is defined
    tree_forest_rec, forest_tree_rec are recursively defined
```

You may now look at the type of tree_forest_rec:

```
Check tree_forest_rec.
    tree_forest_rec
         : forall (P : tree -> Set) (P0 : forest -> Set),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           forall t : tree, P t
```

This principle involves two different predicates for trees andforests; it also has three premises each one corresponding to a constructor of one of the inductive definitions.

The principle `forest_tree_rec` shares exactly the same premises, only the conclusion now refers to the property of forests.

Example

Predicates odd and even on naturals.

Let odd and even be inductively defined as:

```
Inductive odd : nat -> Prop := oddS : forall n:nat, even n -> odd (S n)
with even : nat -> Prop :=
  | evenO : even 0
  | evenS : forall n:nat, odd n -> even (S n).
    odd, even are defined
    odd_ind is defined
    odd_sind is defined
    even_ind is defined
    even_sind is defined
```

The following command generates a powerful elimination principle:

```
Scheme odd_even := Minimality for odd Sort Prop
with even_odd := Minimality for even Sort Prop.
    even_odd is defined
    odd_even is defined
    odd_even, even_odd are recursively defined
```

The type of odd_even for instance will be:

```
Check odd_even.
    odd_even
         : forall P P0 : nat -> Prop,
           (forall n : nat, even n -> P0 n -> P (S n)) ->
           P0 0 ->
           (forall n : nat, odd n -> P n -> P0 (S n)) ->
           forall n : nat, odd n -> P n
```

The type of `even_odd` shares the same premises but the conclusion is `(n:nat)(even n)->(P0 n)`.

Automatic declaration of schemes

Flag: `Elimination Schemes`
This *flag* enables automatic declaration of induction principles when defining a new inductive type. Defaults to on.

Flag: `Nonrecursive Elimination Schemes`
This *flag* enables automatic declaration of induction principles for types declared with the `Variant` and `Record` commands. Defaults to off.

Flag: `Case Analysis Schemes`
This *flag* governs the generation of case analysis lemmas for inductive types, i.e. corresponding to the pattern matching term alone and without fixpoint.

Flag: `Boolean Equality Schemes`
Flag: `Decidable Equality Schemes`
These *flags* control the automatic declaration of those Boolean equalities (see the second variant of `Scheme`).

Warning: You have to be careful with these flags since Coq may now reject well-defined inductive types because it cannot compute a Boolean equality for them.

Flag: `Rewriting Schemes`
This *flag* governs generation of equality-related schemes such as congruence.

Combined Scheme

Command: `Combined Scheme` *ident*$_{def}$ `from` *ident*`+`
This command is a tool for combining induction principles generated by the `Scheme` command. Each ***ident*** is a different inductive principle that must belong to the same package of mutual inductive principle definitions. This command generates ***ident***$_{def}$ as the conjunction of the principles: it is built from the common premises of the principles and concluded by the conjunction of their conclusions. In the case where all the inductive principles used are in sort `Prop`, the propositional conjunction `and` is used, otherwise the simple product `prod` is used instead.

Example

We can define the induction principles for trees and forests using:

```
Scheme tree_forest_ind := Induction for tree Sort Prop
with forest_tree_ind := Induction for forest Sort Prop.
    forest_tree_ind is defined
    tree_forest_ind is defined
    tree_forest_ind, forest_tree_ind are recursively defined
```

Then we can build the combined induction principle which gives the conjunction of the conclusions of each individual principle:

```
Combined Scheme tree_forest_mutind from tree_forest_ind,forest_tree_ind.
    tree_forest_mutind is defined
    tree_forest_mutind is recursively defined
```

The type of tree_forest_mutind will be:

```
Check tree_forest_mutind.
    tree_forest_mutind
         : forall (P : tree -> Prop) (P0 : forest -> Prop),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           (forall t : tree, P t) /\ (forall f2 : forest, P0 f2)
```

Example

We can also combine schemes at sort `Type`:

```
Scheme tree_forest_rect := Induction for tree Sort Type
with forest_tree_rect := Induction for forest Sort Type.
    forest_tree_rect is defined
    tree_forest_rect is defined
    tree_forest_rect, forest_tree_rect are recursively defined

Combined Scheme tree_forest_mutrect from tree_forest_rect, forest_tree_rect.
    tree_forest_mutrect is defined
    tree_forest_mutrect is recursively defined

Check tree_forest_mutrect.
    tree_forest_mutrect
         : forall (P : tree -> Type) (P0 : forest -> Type),
           (forall (a : A) (f : forest), P0 f -> P (node a f)) ->
           (forall b : B, P0 (leaf b)) ->
           (forall t : tree, P t -> forall f1 : forest, P0 f1 -> P0 (cons t f1)) ->
           (forall t : tree, P t) * (forall f2 : forest, P0 f2)
```

See also:

Generation of induction principles with Functional Scheme

Generation of inversion principles with `Derive Inversion`

Command: **`Derive Inversion`** *`ident`* **`with`** *`one_term`* [**`Sort`** *`sort_family`*]?

Generates an inversion lemma for the *`inversion ... using ...`* tactic. *`ident`* is the name of the generated lemma. *`one_term`* should be in the form *`qualid`* or **`(forall`** [*`binder`*]+ **`,`** *`qualid term`***`)`** where *`qualid`* is the name of an inductive predicate and [*`binder`*]+ binds the variables occurring in the term *`term`*. The lemma is generated for the sort *`sort_family`* corresponding to *`one_term`*. Applying the lemma is equivalent to inverting the instance with the *`inversion`* tactic.

Command: **`Derive Inversion_clear`** *`ident`* **`with`** *`one_term`* [**`Sort`** *`sort_family`*]?

When applied, it is equivalent to having inverted the instance with the tactic inversion replaced by the tactic `inversion_clear`.

Command: **`Derive Dependent Inversion`** *`ident`* **`with`** *`one_term`* **`Sort`** *`sort_family`*

When applied, it is equivalent to having inverted the instance with the tactic `dependent inversion`.

Command: **`Derive Dependent Inversion_clear`** *`ident`* **`with`** *`one_term`* **`Sort`** *`sort_family`*

When applied, it is equivalent to having inverted the instance with the tactic `dependent inversion_clear`.

Example

Consider the relation `Le` over natural numbers and the following parameter `P`:

```
Inductive Le : nat -> nat -> Set :=
| LeO : forall n:nat, Le 0 n
| LeS : forall n m:nat, Le n m -> Le (S n) (S m).
    Le is defined
    Le_rect is defined
    Le_ind is defined
    Le_rec is defined
    Le_sind is defined


Parameter P : nat -> nat -> Prop.
    P is declared
```

To generate the inversion lemma for the instance `(Le (S n) m)` and the sort `Prop`, we do:

```
Derive Inversion_clear leminv with (forall n m:nat, Le (S n) m) Sort Prop.
    leminv is defined

Check leminv.
    leminv
         : forall (n m : nat) (P : nat -> nat -> Prop),
           (forall m0 : nat, Le n m0 -> P n (S m0)) -> Le (S n) m -> P n m
```

Then we can use the proven inversion lemma:

```
Show.
    1 goal

      n, m : nat
      H : Le (S n) m
      ============================
      P n m


inversion H using leminv.
    1 goal

      n, m : nat
      H : Le (S n) m
      ============================
      forall m0 : nat, Le n m0 -> P n (S m0)
```

Examples of dependent destruction / dependent induction

The tactics `dependent induction` and `dependent destruction` are another solution for inverting inductive predicate instances and potentially doing induction at the same time. It is based on the `BasicElim` tactic of Conor McBride which works by abstracting each argument of an inductive instance by a variable and constraining it by equalities afterwards. This way, the usual induction and destruct tactics can be applied to the abstracted instance and after simplification of the equalities we get the expected goals.

The abstracting tactic is called generalize_eqs and it takes as argument a hypothesis to generalize. It uses the JMeq datatype defined in Coq.Logic.JMeq, hence we need to require it before. For example, revisiting the first example of the inversion documentation:

```
Require Import Coq.Logic.JMeq.

Inductive Le : nat -> nat -> Set :=
     | LeO : forall n:nat, Le 0 n
     | LeS : forall n m:nat, Le n m -> Le (S n) (S m).

Parameter P : nat -> nat -> Prop.

Goal forall n m:nat, Le (S n) m -> P n m.

intros n m H.

generalize_eqs H.
    1 goal

      n, m, gen_x : nat
      H : Le gen_x m
      ============================
      gen_x = S n -> P n m
```

The index `S n` gets abstracted by a variable here, but a corresponding equality is added under the abstract instance so that no information is actually lost. The goal is now almost amenable to do induction or case analysis. One should indeed first move `n` into the goal to strengthen it before doing induction, or `n` will be fixed in the inductive hypotheses (this does not matter for case analysis). As a rule of thumb, all the variables that appear inside constructors in the indices of the hypothesis should be generalized. This is exactly what the `generalize_eqs_vars` variant does:

```
generalize_eqs_vars H.
induction H.
    2 goals

      n, n0 : nat
      ============================
      0 = S n -> P n n0

    goal 2 is:
     S n0 = S n -> P n (S m)
```

As the hypothesis itself did not appear in the goal, we did not need to use an heterogeneous equality to relate the new hypothesis to the old one (which just disappeared here). However, the tactic works just as well in this case, e.g.:

```
Parameter Q : forall (n m : nat), Le n m -> Prop.
Goal forall n m (p : Le (S n) m), Q (S n) m p.

intros n m p.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      n, m : nat
      p : Le (S n) m
      ============================
      Q (S n) m p

generalize_eqs_vars p.
    1 goal

      m, gen_x : nat
      p : Le gen_x m
      ============================
      forall (n : nat) (p0 : Le (S n) m), gen_x = S n -> p ~= p0 -> Q (S n) m p0
```

One drawback of this approach is that in the branches one will have to substitute the equalities back into the instance to get the right assumptions. Sometimes injection of constructors will also be needed to recover the needed equalities. Also, some subgoals should be directly solved because of inconsistent contexts arising from the constraints on indexes. The nice thing is that we can make a tactic based on discriminate, injection and variants of substitution to automatically do such simplifications (which may involve the axiom K). This is what the `simplify_dep_elim` tactic from `Coq.Program.Equality` does. For example, we might simplify the previous goals considerably:

```
induction p ; simplify_dep_elim.
    1 goal

      n, m : nat
      p : Le n m
      IHp : forall (n0 : nat) (p0 : Le (S n0) m),
            n = S n0 -> p ~= p0 -> Q (S n0) m p0
      ============================
      Q (S n) (S m) (LeS n m p)
```

The higher-order tactic `do_depind` defined in `Coq.Program.Equality` takes a tactic and combines the building blocks we have seen with it: generalizing by equalities calling the given tactic with the generalized induction hypothesis as argument and cleaning the subgoals with respect to equalities. Its most important instantiations are `dependent induction` and `dependent destruction` that do induction or simply case analysis on the generalized hypothesis. For example we can redo what we've done manually with dependent destruction:

```
Lemma ex : forall n m:nat, Le (S n) m -> P n m.

intros n m H.

dependent destruction H.
    1 goal

      n, m : nat
      H : Le n m
      ============================
      P n (S m)
```

This gives essentially the same result as inversion. Now if the destructed hypothesis actually appeared in the goal, the tactic would still be able to invert it, contrary to dependent inversion. Consider the following example on vectors:

```
Set Implicit Arguments.
```

```
Parameter A : Set.

Inductive vector : nat -> Type :=
         | vnil : vector 0
         | vcons : A -> forall n, vector n -> vector (S n).

Goal forall n, forall v : vector (S n),
         exists v' : vector n, exists a : A, v = vcons a v'.

intros n v.

dependent destruction v.
    1 goal

      n : nat
      a : A
      v : vector n
      ============================
      exists (v' : vector n) (a0 : A), vcons a v = vcons a0 v'
```

In this case, the `v` variable can be replaced in the goal by the generalized hypothesis only when it has a type of the form `vector (S n)`, that is only in the second case of the destruct. The first one is dismissed because `S n <> 0`.

A larger example

Let's see how the technique works with induction on inductive predicates on a real example. We will develop an example application to the theory of simply-typed lambda-calculus formalized in a dependently-typed style:

```
Inductive type : Type :=
         | base : type
         | arrow : type -> type -> type.

Notation " t --> t' " := (arrow t t') (at level 20, t' at next level).

Inductive ctx : Type :=
         | empty : ctx
         | snoc : ctx -> type -> ctx.

Notation " G , tau " := (snoc G tau) (at level 20, tau at next level).

Fixpoint conc (G D : ctx) : ctx :=
         match D with
         | empty => G
         | snoc D' x => snoc (conc G D') x
         end.

Notation " G ; D " := (conc G D) (at level 20).

Inductive term : ctx -> type -> Type :=
         | ax : forall G tau, term (G, tau) tau
         | weak : forall G tau,
                    term G tau -> forall tau', term (G, tau') tau
```

(continues on next page)

(continued from previous page)

```
| abs : forall G tau tau',
        term (G , tau) tau' -> term G (tau --> tau')
| app : forall G tau tau',
        term G (tau --> tau') -> term G tau -> term G tau'.
```

We have defined types and contexts which are snoc-lists of types. We also have a `conc` operation that concatenates two contexts. The `term` datatype represents in fact the possible typing derivations of the calculus, which are isomorphic to the well-typed terms, hence the name. A term is either an application of:

- the axiom rule to type a reference to the first variable in a context
- the weakening rule to type an object in a larger context
- the abstraction or lambda rule to type a function
- the application to type an application of a function to an argument

Once we have this datatype we want to do proofs on it, like weakening:

```
Lemma weakening : forall G D tau, term (G ; D) tau ->
                  forall tau', term (G , tau' ; D) tau.
```

The problem here is that we can't just use induction on the typing derivation because it will forget about the `G ; D` constraint appearing in the instance. A solution would be to rewrite the goal as:

```
Lemma weakening' : forall G' tau, term G' tau ->
                   forall G D, (G ; D) = G' ->
                   forall tau', term (G, tau' ; D) tau.
```

With this proper separation of the index from the instance and the right induction loading (putting `G` and `D` after the inducted-on hypothesis), the proof will go through, but it is a very tedious process. One is also forced to make a wrapper lemma to get back the more natural statement. The `dependent induction` tactic alleviates this trouble by doing all of this plumbing of generalizing and substituting back automatically. Indeed we can simply write:

```
Require Import Coq.Program.Tactics.
Require Import Coq.Program.Equality.

Lemma weakening : forall G D tau, term (G ; D) tau ->
                  forall tau', term (G , tau' ; D) tau.

Proof with simpl in * ; simpl_depind ; auto.

intros G D tau H. dependent induction H generalizing G D ; intros.
```

This call to dependent induction has an additional arguments which is a list of variables appearing in the instance that should be generalized in the goal, so that they can vary in the induction hypotheses. By default, all variables appearing inside constructors (except in a parameter position) of the instantiated hypothesis will be generalized automatically but one can always give the list explicitly.

```
Show.
    4 goals

      G0 : ctx
      tau : type
      G, D : ctx
      x : G0, tau = G; D
      tau' : type
```

(continues on next page)

(continued from previous page)

```
  ============================
  term ((G, tau'); D) tau

goal 2 is:
 term ((G, tau'0); D) tau
goal 3 is:
 term ((G, tau'0); D) (tau --> tau')
goal 4 is:
 term ((G, tau'0); D) tau'
```

The `simpl_depind` tactic includes an automatic tactic that tries to simplify equalities appearing at the beginning of induction hypotheses, generally using trivial applications of `reflexivity`. In cases where the equality is not between constructor forms though, one must help the automation by giving some arguments, using the `specialize` tactic for example.

```
destruct D... apply weak; apply ax. apply ax.

destruct D...

Show.
4 goals

  G0 : ctx
  tau : type
  H : term G0 tau
  tau' : type
  IHterm : forall G D : ctx,
           G0 = G; D -> forall tau' : type, term ((G, tau'); D) tau
  tau'0 : type
  ============================
  term ((G0, tau'), tau'0) tau

goal 2 is:
 term (((G, tau'0); D), t) tau
goal 3 is:
 term ((G, tau'0); D) (tau --> tau')
goal 4 is:
 term ((G, tau'0); D) tau'

specialize (IHterm G0 empty eq_refl).
4 goals

  G0 : ctx
  tau : type
  H : term G0 tau
  tau' : type
  IHterm : forall tau' : type, term ((G0, tau'); empty) tau
  tau'0 : type
  ============================
  term ((G0, tau'), tau'0) tau

goal 2 is:
 term (((G, tau'0); D), t) tau
goal 3 is:
 term ((G, tau'0); D) (tau --> tau')
goal 4 is:
```

(continues on next page)

(continued from previous page)

```
     term ((G, tau'0); D) tau'
```

Once the induction hypothesis has been narrowed to the right equality, it can be used directly.

```
apply weak, IHterm.
    3 goals

      tau : type
      G, D : ctx
      IHterm : forall G0 D0 : ctx,
               G; D = G0; D0 -> forall tau' : type, term ((G0, tau'); D0) tau
      H : term (G; D) tau
      t, tau'0 : type
      ============================
      term (((G, tau'0); D), t) tau

    goal 2 is:
     term ((G, tau'0); D) (tau --> tau')
    goal 3 is:
     term ((G, tau'0); D) tau'
```

Now concluding this subgoal is easy.

```
constructor; apply IHterm; reflexivity.
```

3.1.5 The SSReflect proof language

Authors Georges Gonthier, Assia Mahboubi, Enrico Tassi

Introduction

This chapter describes a set of tactics known as SSReflect originally designed to provide support for the so-called *small scale reflection* proof methodology. Despite the original purpose this set of tactic is of general interest and is available in Coq starting from version 8.7.

SSReflect was developed independently of the tactics described in Chapter *Tactics*. Indeed the scope of the tactics part of SSReflect largely overlaps with the standard set of tactics. Eventually the overlap will be reduced in future releases of Coq.

Proofs written in SSReflect typically look quite different from the ones written using only tactics as per Chapter *Tactics*. We try to summarise here the most "visible" ones in order to help the reader already accustomed to the tactics described in Chapter *Tactics* to read this chapter.

The first difference between the tactics described in this chapter and the tactics described in Chapter *Tactics* is the way hypotheses are managed (we call this *bookkeeping*). In Chapter *Tactics* the most common approach is to avoid moving explicitly hypotheses back and forth between the context and the conclusion of the goal. On the contrary in SSReflect all bookkeeping is performed on the conclusion of the goal, using for that purpose a couple of syntactic constructions behaving similar to tacticals (and often named as such in this chapter). The `:` tactical moves hypotheses from the context to the conclusion, while `=>` moves hypotheses from the conclusion to the context, and `in` moves back and forth a hypothesis from the context to the conclusion for the time of applying an action to it.

While naming hypotheses is commonly done by means of an `as` clause in the basic model of Chapter *Tactics*, it is here to `=>` that this task is devoted. Tactics frequently leave new assumptions in the conclusion, and are often followed by `=>` to explicitly name them. While generalizing the goal is normally not explicitly needed in Chapter *Tactics*, it is an explicit operation performed by `:`.

See also:

Bookkeeping

Beside the difference of bookkeeping model, this chapter includes specific tactics which have no explicit counterpart in Chapter *Tactics* such as tactics to mix forward steps and generalizations as *generally have* or *without loss*.

SSReflect adopts the point of view that rewriting, definition expansion and partial evaluation participate all to a same concept of rewriting a goal in a larger sense. As such, all these functionalities are provided by the *rewrite* tactic.

SSReflect includes a little language of patterns to select subterms in tactics or tacticals where it matters. Its most notable application is in the *rewrite* tactic, where patterns are used to specify where the rewriting step has to take place.

Finally, SSReflect supports so-called reflection steps, typically allowing to switch back and forth between the computational view and logical view of a concept.

To conclude it is worth mentioning that SSReflect tactics can be mixed with non SSReflect tactics in the same proof, or in the same Ltac expression. The few exceptions to this statement are described in section *Compatibility issues*.

Acknowledgments

The authors would like to thank Frédéric Blanqui, François Pottier and Laurence Rideau for their comments and suggestions.

Usage

Getting started

To be available, the tactics presented in this manual need the following minimal set of libraries to be loaded: `ssreflect.v`, `ssrfun.v` and `ssrbool.v`. Moreover, these tactics come with a methodology specific to the authors of SSReflect and which requires a few options to be set in a different way than in their default way. All in all, this corresponds to working in the following context:

```
From Coq Require Import ssreflect ssrfun ssrbool.
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

See also:

Implicit Arguments, *Strict Implicit*, *Printing Implicit Defensive*

Compatibility issues

Requiring the above modules creates an environment which is mostly compatible with the rest of Coq, up to a few discrepancies:

- New keywords (`is`) might clash with variable, constant, tactic or tactical names, or with quasi-keywords in tactic or notation commands.
- New tactic(al)s names (*last*, *done*, *have*, *suffices*, *suff*, *without loss*, *wlog*, *congr*, *unlock*) might clash with user tactic names.
- Identifiers with both leading and trailing _, such as `_x_`, are reserved by SSReflect and cannot appear in scripts.

- The extensions to the *rewrite* tactic are partly incompatible with those available in current versions of Coq; in particular: `rewrite .. in (type of k)` or `rewrite .. in *` or any other variant of *rewrite* will not work, and the SSReflect syntax and semantics for occurrence selection and rule chaining is different. Use an explicit rewrite direction (`rewrite <- ...` or `rewrite -> ...`) to access the Coq rewrite tactic.
- New symbols (`//`, `/=`, `//=`) might clash with adjacent existing symbols. This can be avoided by inserting white spaces.
- New constant and theorem names might clash with the user theory. This can be avoided by not importing all of SSReflect:

  ```
  From Coq Require ssreflect.
  Import ssreflect.SsrSyntax.
  ```

  Note that the full syntax of SSReflect's rewrite and reserved identifiers are enabled only if the ssreflect module has been required and if `SsrSyntax` has been imported. Thus a file that requires (without importing) `ssreflect` and imports `SsrSyntax`, can be required and imported without automatically enabling SSReflect's extended rewrite syntax and reserved identifiers.
- Some user notations (in particular, defining an infix `;`) might interfere with the "open term", parenthesis free, syntax of tactics such as have, set and pose.
- The generalization of if statements to non-Boolean conditions is turned off by SSReflect, because it is mostly subsumed by Coercion to `bool` of the `sumXXX` types (declared in `ssrfun.v`) and the **`if`** *`term`* **`is`** *`pattern`* **`then`** *`term`* **`else`** *`term`* construct (see *Pattern conditional*). To use the generalized form, turn off the SSReflect Boolean `if` notation using the command: `Close Scope boolean_if_scope`.
- The following flags can be unset to make SSReflect more compatible with parts of Coq:

Flag: `SsrRewrite`
Controls whether the incompatible rewrite syntax is enabled (the default). Disabling the *flag* makes the syntax compatible with other parts of Coq.

Flag: `SsrIdents`
Controls whether tactics can refer to SSReflect-generated variables that are in the form _xxx_. Scripts with explicit references to such variables are fragile; they are prone to failure if the proof is later modified or if the details of variable name generation change in future releases of Coq.

The default is on, which gives an error message when the user tries to create such identifiers. Disabling the *flag* generates a warning instead, increasing compatibility with other parts of Coq.

Gallina extensions

Small-scale reflection makes an extensive use of the programming subset of Gallina, Coq's logical specification language. This subset is quite suited to the description of functions on representations, because it closely follows the well-established design of the ML programming language. The SSReflect extension provides three additions to Gallina, for pattern assignment, pattern testing, and polymorphism; these mitigate minor but annoying discrepancies between Gallina and ML.

Pattern assignment

The SSReflect extension provides the following construct for irrefutable pattern matching, that is, destructuring assignment:

term += **let:** *pattern* **:=** *term* **in** *term*

Note the colon `:` after the `let` keyword, which avoids any ambiguity with a function definition or Coq's basic destructuring let. The let: construct differs from the latter in that

- The pattern can be nested (deep pattern matching), in particular, this allows expression of the form:

```
let: exist (x, y) p_xy := Hp in … .
```

- The destructured constructor is explicitly given in the pattern, and is used for type inference.

Example

```
Definition f u := let: (m, n) := u in m + n.
    f is defined

Check f.
    f
         : nat * nat -> nat
```

Using `let:` Coq infers a type for `f`, whereas with a usual `let` the same term requires an extra type annotation in order to type check.

```
Fail Definition f u := let (m, n) := u in m + n.
    The command has indeed failed with message:
    Cannot infer a type for this expression.
```

The `let:` construct is just (more legible) notation for the primitive Gallina expression **`match`** *`term`* **`with`** *`pattern`* **`=>`** *`term`* **`end`**.

The SSReflect destructuring assignment supports all the dependent match annotations; the full syntax is

term += **let:** *pattern* [**as** *ident*]? [**in** *pattern*]? **:=** *term* [**return** *term*]? **in** *term*

where the second *`pattern`* and the second *`term`* are *types*.

When the `as` and `return` keywords are both present, then *`ident`* is bound in both the second *`pattern`* and the second *`term`*; variables in the optional type *`pattern`* are bound only in the second term, and other variables in the first *`pattern`* are bound only in the third *`term`*, however.

Pattern conditional

The following construct can be used for a refutable pattern matching, that is, pattern testing:

term += **if** *term* **is** *pattern* **then** *term* **else** *term*

Although this construct is not strictly ML (it does exist in variants such as the pattern calculus or the ρ-calculus), it turns out to be very convenient for writing functions on representations, because most such functions manipulate simple data types such as Peano integers, options, lists, or binary trees, and the pattern conditional above is almost always the right construct for analyzing such simple types. For example, the null and all list function(al)s can be defined as follows:

Example

```
Variable d: Set.
    d is declared

Definition null (s : list d) :=
  if s is nil then true else false.
    null is defined

Variable a : d -> bool.
    a is declared

Fixpoint all (s : list d) : bool :=
  if s is cons x s' then a x && all s' else true.
    all is defined
    all is recursively defined (guarded on 1st argument)
```

The pattern conditional also provides a notation for destructuring assignment with a refutable pattern, adapted to the pure functional setting of Gallina, which lacks a `Match_Failure` exception.

Like `let:` above, the `if...is` construct is just (more legible) notation for the primitive Gallina expression **`match`** *`term`* **`with`** *`pattern`* **`=>`** *`term`* **`| _ =>`** *`term`* **`end`**.

Similarly, it will always be displayed as the expansion of this form in terms of primitive match expressions (where the default expression may be replicated).

Explicit pattern testing also largely subsumes the generalization of the `if` construct to all binary data types; compare **`if`** *`term`* **`is inl _ then`** *`term`* **`else`** *`term`* and **`if`** *`term`* **`then`** *`term`* **`else`** *`term`*.

The latter appears to be marginally shorter, but it is quite ambiguous, and indeed often requires an explicit annotation `(term : {_} + {_})` to type check, which evens the character count.

Therefore, SSReflect restricts by default the condition of a plain if construct to the standard `bool` type; this avoids spurious type annotations.

Example

```
Definition orb b1 b2 := if b1 then true else b2.
    orb is defined
```

As pointed out in section *Compatibility issues*, this restriction can be removed with the command:

```
Close Scope boolean_if_scope.
```

Like `let:` above, the `if-is-then-else` construct supports the dependent match annotations:

term += **if** *term* **is** *pattern* **as** *ident* **in** *pattern* **return** *term* **then** *term* **else** *term*

As in `let:` the variable *ident* (and those in the type pattern) are bound in the second *term*; *ident* is also bound in the third *term* (but not in the fourth *term*), while the variables in the first *pattern* are bound only in the third *term*.

Another variant allows to treat the `else` case first:

term += **if** *term* **isn't** *pattern* **then** *term* **else** *term*

Note that *pattern* eventually binds variables in the third *term* and not in the second *term*.

Parametric polymorphism

Unlike ML, polymorphism in core Gallina is explicit: the type parameters of polymorphic functions must be declared explicitly, and supplied at each point of use. However, Coq provides two features to suppress redundant parameters:

- Sections are used to provide (possibly implicit) parameters for a set of definitions.
- Implicit arguments declarations are used to tell Coq to use type inference to deduce some parameters from the context at each point of call.

The combination of these features provides a fairly good emulation of ML-style polymorphism, but unfortunately this emulation breaks down for higher-order programming. Implicit arguments are indeed not inferred at all points of use, but only at points of call, leading to expressions such as

---

Example

```
Definition all_null (s : list T) := all (@null T) s.
    all_null is defined
```

Unfortunately, such higher-order expressions are quite frequent in representation functions, especially those which use Coq's `Structures` to emulate Haskell typeclasses.

Therefore, SSReflect provides a variant of Coq's implicit argument declaration, which causes Coq to fill in some implicit parameters at each point of use, e.g., the above definition can be written:

Example

```
Prenex Implicits null.
Definition all_null (s : list T) := all null s.
    all_null is defined
```

Better yet, it can be omitted entirely, since `all_null s` isn't much of an improvement over `all null s`.

The syntax of the new declaration is

Command: `Prenex Implicits` *ident*$_i$ +

This command checks that each ***ident***$_i$ is the name of a functional constant, whose implicit arguments are prenex, i.e., the first $n_i > 0$ arguments of ***ident***$_i$ are implicit; then it assigns `Maximal Implicit` status to these arguments.

As these prenex implicit arguments are ubiquitous and have often large display strings, it is strongly recommended to change the default display settings of Coq so that they are not printed (except after a `Set Printing All` command). All SSReflect library files thus start with the incantation

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

Anonymous arguments

When in a definition, the type of a certain argument is mandatory, but not its name, one usually uses "arrow" abstractions for prenex arguments, or the `(_ : term)` syntax for inner arguments. In SSReflect, the latter can be replaced by the open syntax `of term` or (equivalently) `& term`, which are both syntactically equivalent to a `(_ : term)` expression. This feature almost behaves as the following extension of the binder syntax:

binder += **& *term*** | **of *term***

Caveat: `& T` and `of T` abbreviations have to appear at the end of a binder list. For instance, the usual two-constructor polymorphic type list, i.e. the one of the standard `List` library, can be defined by the following declaration:

Example

```
Inductive list (A : Type) : Type := nil | cons of A & list A.
    list is defined
    list_rect is defined
    list_ind is defined
    list_rec is defined
    list_sind is defined
```

Wildcards

The terms passed as arguments to SSReflect tactics can contain *holes*, materialized by wildcards _. Since SSReflect allows a more powerful form of type inference for these arguments, it enhances the possibilities of using such wildcards. These holes are in particular used as a convenient shorthand for abstractions, especially in local definitions or type expressions.

Wildcards may be interpreted as abstractions (see for example sections *Definitions* and *Structure*), or their content can be inferred from the whole context of the goal (see for example section *Abbreviations*).

Definitions

Tactic: `pose`

This tactic allows to add a defined constant to a proof context. SSReflect generalizes this tactic in several ways. In particular, the SSReflect pose tactic supports *open syntax*: the body of the definition does not need surrounding parentheses. For instance:

```
pose t := x + y.
```

is a valid tactic expression.

The pose tactic is also improved for the local definition of higher order terms. Local definitions of functions can use the same syntax as global ones. For example, the tactic `pose` supports parameters:

Example

```
Lemma test : True.
    1 goal

      ============================
      True

pose f x y := x + y.
    1 goal

      f := fun x y : nat => x + y : nat -> nat -> nat
      ============================
      True
```

The SSReflect pose tactic also supports (co)fixpoints, by providing the local counterpart of the `Fixpoint f := …` and `CoFixpoint f := …` constructs. For instance, the following tactic:

```
pose fix f (x y : nat) {struct x} : nat :=
  if x is S p then S (f p y) else 0.
```

defines a local fixpoint `f`, which mimics the standard plus operation on natural numbers.

Similarly, local cofixpoints can be defined by a tactic of the form:

```
pose cofix f (arg : T) := … .
```

The possibility to include wildcards in the body of the definitions offers a smooth way of defining local abstractions. The type of "holes" is guessed by type inference, and the holes are abstracted. For instance the tactic:

```
pose f := _ + 1.
```

is shorthand for:

```
pose f n := n + 1.
```

When the local definition of a function involves both arguments and holes, hole abstractions appear first. For instance, the tactic:

```
pose f x := x + _.
```

is shorthand for:

```
pose f n x := x + n.
```

The interaction of the pose tactic with the interpretation of implicit arguments results in a powerful and concise syntax for local definitions involving dependent types. For instance, the tactic:

```
pose f x y := (x, y).
```

adds to the context the local definition:

```
pose f (Tx Ty : Type) (x : Tx) (y : Ty) := (x, y).
```

The generalization of wildcards makes the use of the pose tactic resemble ML-like definitions of polymorphic functions.

Abbreviations

Tactic: **`set`** *ident* [*: term*]? **`:=`** [*occ_switch*]? *term*

The SSReflect `set` tactic performs abbreviations: it introduces a defined constant for a subterm appearing in the goal and/or in the context.

SSReflect extends the *set* tactic by supplying:

- an open syntax, similarly to the *pose* tactic;
- a more aggressive matching algorithm;
- an improved interpretation of wildcards, taking advantage of the matching algorithm;
- an improved occurrence selection mechanism allowing to abstract only selected occurrences of a term.

occ_switch **`::=`** **`{`** [**`+`** | **`-`**]? *natural** **`}`**

where:

- *ident* is a fresh identifier chosen by the user.
- term 1 is an optional type annotation. The type annotation term 1 can be given in open syntax (no surrounding parentheses). If no *occ_switch* (described hereafter) is present, it is also the case for the second *term*. On the other hand, in presence of *occ_switch*, parentheses surrounding the second *term* are mandatory.
- In the occurrence switch *occ_switch*, if the first element of the list is a natural, this element should be a number, and not an Ltac variable. The empty list `{}` is not interpreted as a valid occurrence switch, it is rather used as a flag to signal the intent of the user to clear the name following it (see *Occurrence switches and redex switches* and *Introduction in the context*)

The tactic:

Example

```
Lemma test x :  f x + f x = f x.
    1 goal

      x : nat
      ============================
      f x + f x = f x

set t := f _.
    1 goal

      x : nat
      t := f x : nat
      ============================
      t + t = t

set t := {2}(f _).
    1 goal

      x : nat
      t := f x : nat
      ============================
      f x + t = f x
```

The type annotation may contain wildcards, which will be filled with the appropriate value by the matching process.

The tactic first tries to find a subterm of the goal matching the second *term* (and its type), and stops at the first subterm it finds. Then the occurrences of this subterm selected by the optional *occ_switch* are replaced by *ident* and a definition ***ident*** **:=** ***term*** is added to the context. If no *occ_switch* is present, then all the occurrences are abstracted.

Matching

The matching algorithm compares a pattern *term* with a subterm of the goal by comparing their heads and then pairwise unifying their arguments (modulo conversion). Head symbols match under the following conditions:

- If the head of *term* is a constant, then it should be syntactically equal to the head symbol of the subterm.
- If this head is a projection of a canonical structure, then canonical structure equations are used for the matching.
- If the head of term is *not* a constant, the subterm should have the same structure (λ abstraction, let…in structure …).
- If the head of *term* is a hole, the subterm should have at least as many arguments as *term*.

Example

```
Lemma test (x y z : nat) :  x + y = z.
    1 goal

      x, y, z : nat
      ============================
      x + y = z

set t := _ x.
    1 goal

      x, y, z : nat
      t := Nat.add x : nat -> nat
      ============================
      t y = z
```

- In the special case where `term` is of the form `(let f := t0 in f) t1 … tn` , then the pattern `term` is treated as `(_ t1 … tn)`. For each subterm in the goal having the form `(A u1 … um)` with m ≥ n, the matching algorithm successively tries to find the largest partial application `(A u1 … uj)` convertible to the head `t0` of `term`.

  Example

  ```
  Lemma test : (let f x y z := x + y + z in f 1) 2 3 = 6.
      1 goal

        ============================
        (let f := fun x y z : nat => x + y + z in f 1) 2 3 = 6

  set t := (let g y z := S y + z in g) 2.
      1 goal
  ```

(continues on next page)

(continued from previous page)

```
      t := unkeyed (fun y z : nat => S y + z) 2 : nat -> nat
      ============================
      t 3 = 6
```

The notation `unkeyed` defined in `ssreflect.v` is a shorthand for the degenerate term `let x := … in x`.

Moreover:

- Multiple holes in `term` are treated as independent placeholders.

Example

```
Lemma test x y z : x + y = z.
    1 goal

      x, y, z : nat
      ============================
      x + y = z

set t := _ + _.
    1 goal

      x, y, z : nat
      t := x + y : nat
      ============================
      t = z
```

- The type of the subterm matched should fit the type (possibly casted by some type annotations) of the pattern `term`.
- The replacement of the subterm found by the instantiated pattern should not capture variables. In the example above `x` is bound and should not be captured.

Example

```
Lemma test : forall x : nat, x + 1 = 0.
    1 goal

      ============================
      forall x : nat, x + 1 = 0

Fail set t := _ + 1.
    The command has indeed failed with message:
    The pattern (_ + 1) did not match and has holes. Did you mean pose?
```

- Typeclass inference should fill in any residual hole, but matching should never assign a value to a global existential variable.

Occurrence selection

SSReflect provides a generic syntax for the selection of occurrences by their position indexes. These *occurrence switches* are shared by all SSReflect tactics which require control on subterm selection like rewriting, generalization, …

An *occurrence switch* can be:

- A list natural numbers `{+ n1 … nm}` of occurrences affected by the tactic.

Example

```
Lemma test : f 2 + f 8 = f 2 + f 2.
    1 goal

      ============================
      f 2 + f 8 = f 2 + f 2

set x := {+1 3}(f 2).
    1 goal

      x := f 2 : nat
      ============================
      x + f 8 = f 2 + x
```

Notice that some occurrences of a given term may be hidden to the user, for example because of a notation. Setting the *`Printing All`* flag causes these hidden occurrences to be shown when the term is displayed. This setting should be used to find the correct coding of the occurrences to be selected[33].

Example

```
Notation "a < b":= (le (S a) b).
Lemma test x y : x < y -> S x < S y.
    1 goal

      x, y : nat
      ============================
      x < y -> S x < S y

set t := S x.
    1 goal

      x, y : nat
      t := S x : nat
      ============================
      t <= y -> t < S y
```

- A list of natural numbers between `{n1 … nm}`. This is equivalent to the previous `{+ n1 … nm}` but the list should start with a number, and not with an Ltac variable.
- A list `{- n1 … nm}` of occurrences *not* to be affected by the tactic.

Example

[33] Unfortunately, even after a call to the Set Printing All command, some occurrences are still not displayed to the user, essentially the ones possibly hidden in the predicate of a dependent match structure.

```
Lemma test : f 2 + f 8 = f 2 + f 2.
    1 goal

      ============================
      f 2 + f 8 = f 2 + f 2

set x := {-2}(f 2).
    1 goal

      x := f 2 : nat
      ============================
      x + f 8 = f 2 + x
```

Note that, in this goal, it behaves like `set x := {1 3}(f 2).`

- In particular, the switch `{+}` selects *all* the occurrences. This switch is useful to turn off the default behavior of a tactic which automatically clears some assumptions (see section *Discharge* for instance).
- The switch `{-}` imposes that *no* occurrences of the term should be affected by the tactic. The tactic: `set x := {-}(f 2).` leaves the goal unchanged and adds the definition `x := f 2` to the context. This kind of tactic may be used to take advantage of the power of the matching algorithm in a local definition, instead of copying large terms by hand.

It is important to remember that matching *precedes* occurrence selection.

Example

```
Lemma test x y z : x + y = x + y + z.
    1 goal

      x, y, z : nat
      ============================
      x + y = x + y + z

set a := {2}(_ + _).
    1 goal

      x, y, z : nat
      a := x + y : nat
      ============================
      x + y = a + z
```

Hence, in the following goal, the same tactic fails since there is only one occurrence of the selected term.

Example

```
Lemma test x y z : (x + y) + (z + z) = z + z.
    1 goal

      x, y, z : nat
      ============================
      x + y + (z + z) = z + z

Fail set a := {2}(_ + _).
```

(continues on next page)

(continued from previous page)

```
The command has indeed failed with message:
Only 1 < 2 occurrence of (x + y + (z + z))
```

Basic localization

It is possible to define an abbreviation for a term appearing in the context of a goal thanks to the `in` tactical.

Variant: `set` *ident* `:=` *term* `in` *ident* +

This variant of *set* introduces a defined constant called *ident* in the context, and folds it in the context entries mentioned on the right hand side of `in`. The body of *ident* is the first subterm matching these context entries (taken in the given order).

Example

```
Lemma test x t (Hx : x = 3) : x + t = 4.
    1 goal

      x, t : nat
      Hx : x = 3
      ============================
      x + t = 4

set z := 3 in Hx.
    1 goal

      x, t : nat
      z := 3 : nat
      Hx : x = z
      ============================
      x + t = 4
```

Variant: `set` *ident* `:=` *term* `in` *ident* `*` +

This variant matches *term* and then folds *ident* similarly in all the given context entries but also folds *ident* in the goal.

Example

```
Lemma test x t (Hx : x = 3) : x + t = 4.
    1 goal

      x, t : nat
      Hx : x = 3
      ============================
      x + t = 4

set z := 3 in Hx * .
    1 goal

      x, t : nat
```

(continues on next page)

(continued from previous page)

```
z := 3 : nat
Hx : x = z
============================
x + t = S z
```

Indeed, remember that 4 is just a notation for (S 3).

The use of the `in` tactical is not limited to the localization of abbreviations: for a complete description of the in tactical, see section *Bookkeeping* and *Localization*.

Basic tactics

A sizable fraction of proof scripts consists of steps that do not ”prove” anything new, but instead perform menial bookkeeping tasks such as selecting the names of constants and assumptions or splitting conjuncts. Although they are logically trivial, bookkeeping steps are extremely important because they define the structure of the data-flow of a proof script. This is especially true for reflection-based proofs, which often involve large numbers of constants and assumptions. Good bookkeeping consists in always explicitly declaring (i.e., naming) all new constants and assumptions in the script, and systematically pruning irrelevant constants and assumptions in the context. This is essential in the context of an interactive development environment (IDE), because it facilitates navigating the proof, allowing to instantly ”jump back” to the point at which a questionable assumption was added, and to find relevant assumptions by browsing the pruned context. While novice or casual Coq users may find the automatic name selection feature convenient, the usage of such a feature severely undermines the readability and maintainability of proof scripts, much like automatic variable declaration in programming languages. The SSReflect tactics are therefore designed to support precise bookkeeping and to eliminate name generation heuristics. The bookkeeping features of SSReflect are implemented as tacticals (or pseudo-tacticals), shared across most SSReflect tactics, and thus form the foundation of the SSReflect proof language.

Bookkeeping

During the course of a proof Coq always present the user with a *sequent* whose general form is:

```
ci : Ti
…
dj := ej : Tj
…
Fk : Pk
…
=================
forall (xl : Tl) …,
let ym := bm in … in
Pn -> … -> C
```

The *goal* to be proved appears below the double line; above the line is the *context* of the sequent, a set of declarations of *constants* `ci` , *defined constants* `dj` , and *facts* `Fk` that can be used to prove the goal (usually, `Ti` , `Tj : Type` and `Pk : Prop`). The various kinds of declarations can come in any order. The top part of the context consists of declarations produced by the Section commands `Variable`, `Let`, and `Hypothesis`. This *section context* is never affected by the SSReflect tactics: they only operate on the lower part — the *proof context*. As in the figure above, the goal often decomposes into a series of (universally) quantified *variables* `(xl : Tl)`, local *definitions* `let ym := bm in`, and *assumptions* `P n ->`, and a *conclusion* `C` (as in the context, variables, definitions, and assumptions can appear in any order). The conclusion is what actually needs to be proved — the rest of the goal can be seen as a part of the proof context that happens to be “below the line”.

However, although they are logically equivalent, there are fundamental differences between constants and facts on the one hand, and variables and assumptions on the others. Constants and facts are *unordered*, but *named* explicitly in the proof

text; variables and assumptions are *ordered*, but *unnamed*: the display names of variables may change at any time because of α-conversion.

Similarly, basic deductive steps such as apply can only operate on the goal because the Gallina terms that control their action (e.g., the type of the lemma used by `apply`) only provide unnamed bound variables.[34] Since the proof script can only refer directly to the context, it must constantly shift declarations from the goal to the context and conversely in between deductive steps.

In SSReflect these moves are performed by two *tacticals* => and `:`, so that the bookkeeping required by a deductive step can be directly associated with that step, and that tactics in an SSReflect script correspond to actual logical steps in the proof rather than merely shuffle facts. Still, some isolated bookkeeping is unavoidable, such as naming variables and assumptions at the beginning of a proof. SSReflect provides a specific `move` tactic for this purpose.

Now `move` does essentially nothing: it is mostly a placeholder for => and `:`. The => tactical moves variables, local definitions, and assumptions to the context, while the `:` tactical moves facts and constants to the goal.

Example

For example, the proof of[35]

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
    1 goal

      ============================
      forall m n : nat, n <= m -> m - n + n = m
```

might start with

```
move=> m n le_n_m.
    1 goal

      m, n : nat
      le_n_m : n <= m
      ============================
      m - n + n = m
```

where move does nothing, but `=> m n le_m_n` changes the variables and assumption of the goal in the constants `m n : nat` and the fact `le_n_m : n <= m`, thus exposing the conclusion `m - n + n = m`.

The `:` tactical is the converse of =>, indeed it removes facts and constants from the context by turning them into variables and assumptions.

```
move: m le_n_m.
    1 goal

      n : nat
      ============================
      forall m : nat, n <= m -> m - n + n = m
```

turns back `m` and `le_m_n` into a variable and an assumption, removing them from the proof context, and changing the goal to `forall m, n <= m -> m - n + n = m` which can be proved by induction on `n` using `elim: n`.

Because they are tacticals, `:` and => can be combined, as in

[34] Thus scripts that depend on bound variable names, e.g., via intros or with, are inherently fragile.

[35] The name `subnK` reads as "right cancellation rule for nat subtraction".

```
move: m le_n_m => p le_n_p.
```

simultaneously renames `m` and `le_m_n` into `p` and `le_n_p`, respectively, by first turning them into unnamed variables, then turning these variables back into constants and facts.

Furthermore, SSReflect redefines the basic Coq tactics `case`, `elim`, and `apply` so that they can take better advantage of `:` and `=>`. In there SSReflect variants, these tactic operate on the first variable or constant of the goal and they do not use or change the proof context. The `:` tactical is used to operate on an element in the context.

Example

For instance the proof of `subnK` could continue with `elim: n`. Instead of `elim n` (note, no colon), this has the advantage of removing n from the context. Better yet, this `elim` can be combined with previous move and with the branching version of the `=>` tactical (described in *Introduction in the context*), to encapsulate the inductive step in a single command:

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
    1 goal

      ============================
      forall m n : nat, n <= m -> m - n + n = m

move=> m n le_n_m.
    1 goal

      m, n : nat
      le_n_m : n <= m
      ============================
      m - n + n = m

elim: n m le_n_m => [|n IHn] m => [_ | lt_n_m].
    2 goals

      m : nat
      ============================
      m - 0 + 0 = m

    goal 2 is:
     m - S n + S n = m
```

which breaks down the proof into two subgoals, the second one having in its context `lt_n_m : S n <= m` and `IHn : forall m, n <= m -> m - n + n = m`.

The `:` and `=>` tacticals can be explained very simply if one views the goal as a stack of variables and assumptions piled on a conclusion:

- `tactic : a b c` pushes the context constants `a`, `b`, `c` as goal variables *before* performing tactic.
- `tactic => a b c` pops the top three goal variables as context constants `a`, `b`, `c`, *after* tactic has been performed.

These pushes and pops do not need to balance out as in the examples above, so `move: m le_n_m => p` would rename `m` into `p`, but leave an extra assumption `n <= p` in the goal.

Basic tactics like apply and elim can also be used without the ':' tactical: for example we can directly start a proof of `subnK` by induction on the top variable `m` with

```
elim=> [|m IHm] n le_n.
```

The general form of the localization tactical in is also best explained in terms of the goal stack:

```
tactic in a H1 H2 *.
```

is basically equivalent to

```
move: a H1 H2; tactic => a H1 H2.
```

with two differences: the in tactical will preserve the body of an if a is a defined constant, and if the `*` is omitted it will use a temporary abbreviation to hide the statement of the goal from `tactic`.

The general form of the in tactical can be used directly with the `move`, `case` and `elim` tactics, so that one can write

```
elim: n => [|n IHn] in m le_n_m *.
```

instead of

```
elim: n m le_n_m => [|n IHn] m le_n_m.
```

This is quite useful for inductive proofs that involve many facts.

See section *Localization* for the general syntax and presentation of the in tactical.

The defective tactics

In this section we briefly present the three basic tactics performing context manipulations and the main backward chaining tool.

The move tactic.

Tactic: move

This tactic, in its defective form, behaves like the *hnf* tactic.

Example

```
Require Import ssreflect.
Goal not False.
    1 goal

      ============================
      ~ False

move.
    1 goal

      ============================
      False -> False
```

More precisely, the *move* tactic inspects the goal and does nothing (*idtac*) if an introduction step is possible, i.e. if the goal is a product or a `let ... in`, and performs *hnf* otherwise.

Of course this tactic is most often used in combination with the bookkeeping tacticals (see section *Introduction in the context* and *Discharge*). These combinations mostly subsume the `intros`, `generalize`, `revert`, `rename`, `clear` and `pattern` tactics.

The case tactic

Tactic: `case`

This tactic performs *primitive case analysis* on (co)inductive types; specifically, it destructs the top variable or assumption of the goal, exposing its constructor(s) and its arguments, as well as setting the value of its type family indices if it belongs to a type family (see section *Type families*).

The SSReflect case tactic has a special behavior on equalities. If the top assumption of the goal is an equality, the case tactic "destructs" it as a set of equalities between the constructor arguments of its left and right hand sides, as per the tactic injection. For example, `case` changes the goal:

```
(x, y) = (1, 2) -> G.
```

into:

```
x = 1 -> y = 2 -> G.
```

The `case` can generate the following warning:

Warning: `SSReflect: cannot obtain new equations out of ...`

The tactic was run on an equation that cannot generate simpler equations, for example `x = 1`.

The warning can be silenced or made fatal by using the `Warnings` option and the `spurious-ssr-injection` key.

Finally the `case` tactic of SSReflect performs `False` elimination, even if no branch is generated by this case operation. Hence the tactic `case` on a goal of the form `False -> G` will succeed and prove the goal.

The elim tactic

Tactic: `elim`

This tactic performs inductive elimination on inductive types. In its defective form, the tactic performs inductive elimination on a goal whose top assumption has an inductive type.

Example

```
Lemma test m : forall n : nat, m <= n.
    1 goal

      m : nat
      ============================
      forall n : nat, m <= n

elim.
    2 goals

      m : nat
      ============================
      m <= 0
```

(continues on next page)

(continued from previous page)

```
goal 2 is:
 forall n : nat, m <= n -> m <= S n
```

The apply tactic

Tactic: `apply` *term* ?

This is the main backward chaining tactic of the proof system. It takes as argument any *term* and applies it to the goal. Assumptions in the type of *term* that don't directly match the goal may generate one or more subgoals.

In its defective form, this tactic is a synonym for:

```
intro top; first [refine top | refine (top _) | refine (top _ _) | …]; clear top.
```

where `top` is a fresh name, and the sequence of *refine* tactics tries to catch the appropriate number of wildcards to be inserted. Note that this use of the *refine* tactic implies that the tactic tries to match the goal up to expansion of constants and evaluation of subterms.

apply has a special behavior on goals containing existential metavariables of sort `Prop`.

Example

```
Lemma test : forall y, 1 < y -> y < 2 -> exists x : { n | n < 3 }, 0 < proj1_sig x.
    1 goal

      ============================
      forall y : nat,
      1 < y -> y < 2 -> exists x : {n : nat | n < 3}, 0 < proj1_sig x

move=> y y_gt1 y_lt2; apply: (ex_intro _ (exist _ y _)).
    2 focused goals
    (shelved: 2)

      y : nat
      y_gt1 : 1 < y
      y_lt2 : y < 2
      ============================
      y < 3

    goal 2 is:
     forall Hyp0 : y < 3, 0 < proj1_sig (exist (fun n : nat => n < 3) y Hyp0)

  by apply: lt_trans y_lt2 _.
    1 focused goal
    (shelved: 1)

      y : nat
      y_gt1 : 1 < y
      y_lt2 : y < 2
      ============================
      forall Hyp0 : y < 3, 0 < proj1_sig (exist (fun n : nat => n < 3) y Hyp0)

by move=> y_lt3; apply: lt_trans y_gt1.
    No more goals.
```

Note that the last _ of the tactic `apply: (ex_intro _ (exist _ y _))` represents a proof that `y < 3`. Instead of generating the goal:

```
0 < proj1_sig (exist (fun n : nat => n < 3) y ?Goal).
```

the system tries to prove `y < 3` calling the trivial tactic. If it succeeds, let's say because the context contains `H : y < 3`, then the system generates the following goal:

```
0 < proj1_sig (exist (fun n => n < 3) y H).
```

Otherwise the missing proof is considered to be irrelevant, and is thus discharged generating the two goals shown above.

Last, the user can replace the trivial tactic by defining an Ltac expression named `ssrautoprop`.

Discharge

The general syntax of the discharging tactical `:` is:

Tactic: *tactic* *ident*? `:` *d_item*+ *clear_switch*?

d_item ::= *occ_switch* | *clear_switch*? *term*

clear_switch ::= { *ident*+ }

with the following requirements:

- *tactic* must be one of the four basic tactics described in *The defective tactics*, i.e., `move`, `case`, `elim` or `apply`, the `exact` tactic (section *Terminators*), the `congr` tactic (section *Congruence*), or the application of the *view* tactical '/' (section *Interpreting assumptions*) to one of move, case, or elim.
- The optional *ident* specifies *equation generation* (section *Generation of equations*), and is only allowed if tactic is `move`, `case` or `elim`, or the application of the view tactical '/' (section *Interpreting assumptions*) to `case` or `elim`.
- An *occ_switch* selects occurrences of *term*, as in *Abbreviations*; *occ_switch* is not allowed if *tactic* is `apply` or `exact`.
- A clear item *clear_switch* specifies facts and constants to be deleted from the proof context (as per the clear tactic).

The `:` tactical first *discharges* all the *d_item*, right to left, and then performs tactic, i.e., for each *d_item*, starting with the last one :

1. The SSReflect matching algorithm described in section *Abbreviations* is used to find occurrences of term in the goal, after filling any holes '_' in term; however if tactic is apply or exact a different matching algorithm, described below, is used[36].
2. These occurrences are replaced by a new variable; in particular, if term is a fact, this adds an assumption to the goal.
3. If term is *exactly* the name of a constant or fact in the proof context, it is deleted from the context, unless there is an *occ_switch*.

[36] Also, a slightly different variant may be used for the first *d_item* of case and elim; see section *Type families*.

Finally, tactic is performed just after the first *d_item* has been generalized — that is, between steps 2 and 3. The names listed in the final *clear_switch* (if it is present) are cleared first, before *d_item* n is discharged.

Switches affect the discharging of a *d_item* as follows:

- An *occ_switch* restricts generalization (step 2) to a specific subset of the occurrences of term, as per section *Abbreviations*, and prevents clearing (step 3).
- All the names specified by a *clear_switch* are deleted from the context in step 3, possibly in addition to term.

For example, the tactic:

```
move: n {2}n (refl_equal n).
```

- first generalizes `(refl_equal n : n = n)`;
- then generalizes the second occurrence of `n`.
- finally generalizes all the other occurrences of `n`, and clears `n` from the proof context (assuming n is a proof constant).

Therefore this tactic changes any goal `G` into

```
forall n n0 : nat, n = n0 -> G.
```

where the name `n0` is picked by the Coq display function, and assuming `n` appeared only in `G`.

Finally, note that a discharge operation generalizes defined constants as variables, and not as local definitions. To override this behavior, prefix the name of the local definition with a `@`, like in `move: @n`.

This is in contrast with the behavior of the in tactical (see section *Localization*), which preserves local definitions by default.

Clear rules

The clear step will fail if term is a proof constant that appears in other facts; in that case either the facts should be cleared explicitly with a *clear_switch*, or the clear step should be disabled. The latter can be done by adding an *occ_switch* or simply by putting parentheses around term: both `move: (n).` and `move: {+}n.` generalize `n` without clearing `n` from the proof context.

The clear step will also fail if the *clear_switch* contains a *ident* that is not in the *proof* context. Note that SSReflect never clears a section constant.

If tactic is `move` or `case` and an equation *ident* is given, then clear (step 3) for *d_item* is suppressed (see section *Generation of equations*).

Intro patterns (see section *Introduction in the context*) and the `rewrite` tactic (see section *Rewriting*) let one place a *clear_switch* in the middle of other items (namely identifiers, views and rewrite rules). This can trigger the addition of proof context items to the ones being explicitly cleared, and in turn this can result in clear errors (e.g. if the context item automatically added occurs in the goal). The relevant sections describe ways to avoid the unintended clear of context items.

Matching for apply and exact

The matching algorithm for *d_item* of the SSReflect `apply` and `exact` tactics exploits the type of the first *d_item* to interpret wildcards in the other *d_item* and to determine which occurrences of these should be generalized. Therefore, occur switches are not needed for apply and exact.

Indeed, the SSReflect tactic `apply: H x` is equivalent to `refine (@H _ … _ x); clear H x` with an appropriate number of wildcards between `H` and `x`.

Note that this means that matching for `apply` and `exact` has much more context to interpret wildcards; in particular it can accommodate the `_ d_item`, which would always be rejected after `move:`.

Example

```
Lemma test (Hfg : forall x, f x = g x) a b : f a = g b.
    1 goal

      Hfg : forall x : nat, f x = g x
      a, b : nat
      ============================
      f a = g b

apply: trans_equal (Hfg _) _.
    1 focused goal
    (shelved: 1)

      Hfg : forall x : nat, f x = g x
      a, b : nat
      ============================
      g a = g b
```

This tactic is equivalent (see section *Bookkeeping*) to: `refine (trans_equal (Hfg _) _).` and this is a common idiom for applying transitivity on the left hand side of an equation.

The abstract tactic

Tactic: `abstract:` *d_item* +

This tactic assigns an abstract constant previously introduced with the **`[: ident ]`** intro pattern (see section *Introduction in the context*).

In a goal like the following:

```
m : nat
abs : <hidden>
n : nat
=============
m < 5 + n
```

The tactic `abstract: abs n` first generalizes the goal with respect to `n` (that is not visible to the abstract constant abs) and then assigns abs. The resulting goal is:

```
m : nat
n : nat
=============
m < 5 + n
```

Once this subgoal is closed, all other goals having abs in their context see the type assigned to `abs`. In this case:

```
m : nat
abs : forall n, m < 5 + n
=============
...
```

For a more detailed example the reader should refer to section *Structure*.

Introduction in the context

The application of a tactic to a given goal can generate (quantified) variables, assumptions, or definitions, which the user may want to *introduce* as new facts, constants or defined constants, respectively. If the tactic splits the goal into several subgoals, each of them may require the introduction of different constants and facts. Furthermore it is very common to immediately decompose or rewrite with an assumption instead of adding it to the context, as the goal can often be simplified and even proved after this.

All these operations are performed by the introduction tactical =>, whose general syntax is

Tactic: *tactic* **=>** *i_item*+

i_item ::= *i_pattern* | *s_item* | *clear_switch* | *i_view* | *i_block*

s_item ::= **/=** | **//** | **//=**

i_view ::= {}? | */term* | **/ltac:(** *tactic* **)**

i_pattern ::= *ident* | **>** | **_** | **?** | ***** | **+** | *occ_switch*? **->** | **<-** | **[** *i_item*? | | **]** | **-** | **[:** *ident*+ **]**

i_block ::= **[^** *ident* **]** | **[^~** *ident* | *natural* **]**

The => tactical first executes *tactic*, then the *i_item*s, left to right. An *s_item* specifies a simplification operation; a *clear_switch* specifies context pruning as in *Discharge*. The *i_pattern*s can be seen as a variant of *intro patterns* (see *intros*:) each performs an introduction operation, i.e., pops some variables or assumptions from the goal.

Simplification items

An *s_item* can simplify the set of subgoals or the subgoals themselves:

- `//` removes all the "trivial" subgoals that can be resolved by the SSReflect tactic *done* described in *Terminators*, i.e., it executes `try done`.
- `/=` simplifies the goal by performing partial evaluation, as per the tactic *simpl*[37].
- `//=` combines both kinds of simplification; it is equivalent to `/= //`, i.e., `simpl; try done`.

When an *s_item* immediately precedes a *clear_switch*, then the *clear_switch* is executed *after* the *s_item*, e.g., `{IHn}//` will solve some subgoals, possibly using the fact `IHn`, and will erase `IHn` from the context of the remaining subgoals.

[37] Except /= does not expand the local definitions created by the SSReflect in tactical.

Views

The first entry in the *i_view* grammar rule, **/*term***, represents a view (see section *Views and reflection*). It interprets the top of the stack with the view *term*. It is equivalent to **move/*term***.

A *clear_switch* that immediately precedes an *i_view* is complemented with the name of the view if an only if the *i_view* is a simple proof context entry[42]. E.g. `{}/v` is equivalent to `/v{v}`. This behavior can be avoided by separating the *clear_switch* from the *i_view* with the − intro pattern or by putting parentheses around the view.

A *clear_switch* that immediately precedes an *i_view* is executed after the view application.

If the next *i_item* is a view, then the view is applied to the assumption in top position once all the previous *i_item* have been performed.

The second entry in the *i_view* grammar rule, `/ltac:(` *tactic* `)`, executes *tactic*. Notations can be used to name tactics, for example

```
Notation "'myop'" := (ltac:(my ltac code)) : ssripat_scope.
```

lets one write just `/myop` in the intro pattern. Note the scope annotation: views are interpreted opening the `ssripat` scope. We provide the following ltac views: `/[dup]` to duplicate the top of the stack, `/[swap]` to swap the two first elements and `/[apply]` to apply the top of the stack to the next.

Intro patterns

SSReflect supports the following *i_pattern*s:

ident pops the top variable, assumption, or local definition into a new constant, fact, or defined constant *ident*, respectively. Note that defined constants cannot be introduced when δ-expansion is required to expose the top variable or assumption. A *clear_switch* (even an empty one) immediately preceding an *ident* is complemented with that *ident* if and only if the identifier is a simple proof context entry[?]. As a consequence by prefixing the *ident* with `{}` one can *replace* a context entry. This behavior can be avoided by separating the *clear_switch* from the *ident* with the − intro pattern.

> pops every variable occurring in the rest of the stack. Type class instances are popped even if they don't occur in the rest of the stack. The tactic `move=> >` is equivalent to `move=> ? ?` on a goal such as:

```
forall x y, x < y -> G
```

A typical use if `move=>> H` to name `H` the first assumption, in the example above `x < y`.

? pops the top variable into an anonymous constant or fact, whose name is picked by the tactic interpreter. SSReflect only generates names that cannot appear later in the user script[38].

_ pops the top variable into an anonymous constant that will be deleted from the proof context of all the subgoals produced by the => tactical. They should thus never be displayed, except in an error message if the constant is still actually used in the goal or context after the last *i_item* has been executed (*s_item* can erase goals or terms where the constant appears).

***** pops all the remaining apparent variables/assumptions as anonymous constants/facts. Unlike `?` and `move` the `*` *i_item* does not expand definitions in the goal to expose quantifiers, so it may be useful to repeat a `move=> *` tactic, e.g., on the goal:

```
forall a b : bool, a <> b
```

[42] A simple proof context entry is a naked identifier (i.e. not between parentheses) designating a context entry that is not a section variable.
[38] SSReflect reserves all identifiers of the form "_x_", which is used for such generated names.

a first `move=> *` adds only `_a_ : bool` and `_b_ : bool` to the context; it takes a second `move=> *` to add `_Hyp_ : _a_ = _b_`.

+ temporarily introduces the top variable. It is discharged at the end of the intro pattern. For example `move=> + y` on a goal:

```
forall x y, P
```

is equivalent to `move=> _x_ y; move: _x_` that results in the goal:

```
forall x, P
```

occ_switch **?** **->** (resp. *occ_switch* **<-**) pops the top assumption (which should be a rewritable proposition) into an anonymous fact, rewrites (resp. rewrites right to left) the goal with this fact (using the SSReflect `rewrite` tactic described in section *Rewriting*, and honoring the optional occurrence selector), and finally deletes the anonymous fact from the context.

[***i_item*** *** | … | *i_item* *]** when it is the very *first* *i_pattern* after tactic `=>` tactical *and* tactic is not a move, is a *branching* *i_pattern*. It executes the sequence i_item_i on the i-th subgoal produced by tactic. The execution of tactic should thus generate exactly m subgoals, unless the `[…]` *i_pattern* comes after an initial `//` or `//=` *s_item* that closes some of the goals produced by `tactic`, in which case exactly m subgoals should remain after the *s_item*, or we have the trivial branching *i_pattern* [], which always does nothing, regardless of the number of remaining subgoals.

[***i_item*** *** | … | *i_item* *]** when it is *not* the first *i_pattern* or when tactic is a `move`, is a *destructing* *i_pattern*. It starts by destructing the top variable, using the SSReflect `case` tactic described in *The defective tactics*. It then behaves as the corresponding branching *i_pattern*, executing the sequence i_item_i in the i-th subgoal generated by the case analysis; unless we have the trivial destructing *i_pattern* `[]`, the latter should generate exactly m subgoals, i.e., the top variable should have an inductive type with exactly m constructors[39]. While it is good style to use the *i_item* i * to pop the variables and assumptions corresponding to each constructor, this is not enforced by SSReflect.

– does nothing, but counts as an intro pattern. It can also be used to force the interpretation of `[` *i_item* `* | … |` *i_item* `* ]` as a case analysis like in `move=> -[H1 H2]`. It can also be used to indicate explicitly the link between a view and a name like in `move=> /eqP-H1`. Last, it can serve as a separator between views. Section *Views and reflection*[41] explains in which respect the tactic `move=> /v1/v2` differs from the tactic `move=> /v1-/v2`.

[: ***ident*** **…]** introduces in the context an abstract constant for each *ident*. Its type has to be fixed later on by using the `abstract` tactic. Before then the type displayed is `<hidden>`.

Note that SSReflect does not support the syntax `(ipat, …, ipat)` for destructing intro patterns.

Clear switch

Clears are deferred until the end of the intro pattern.

Example

```
Lemma test x y : Nat.leb 0 x = true -> (Nat.leb 0 x) && (Nat.leb y 2) = true.
    1 goal

      x, y : nat
```

(continues on next page)

[39] More precisely, it should have a quantified inductive type with a assumptions and m – a constructors.
[41] The current state of the proof shall be displayed by the Show Proof command of Coq proof mode.

(continued from previous page)

```
      ============================
      Nat.leb 0 x = true -> Nat.leb 0 x && Nat.leb y 2 = true

move=> {x} ->.
    1 goal

      y : nat
      ============================
      true && Nat.leb y 2 = true
```

If the cleared names are reused in the same intro pattern, a renaming is performed behind the scenes.

Facts mentioned in a clear switch must be valid names in the proof context (excluding the section context).

Branching and destructuring

The rules for interpreting branching and destructing *i_pattern* are motivated by the fact that it would be pointless to have a branching pattern if tactic is a `move`, and in most of the remaining cases tactic is `case` or `elim`, which implies destruction. The rules above imply that:

- `move=> [a b].`
- `case=> [a b].`
- `case=> a b.`

are all equivalent, so which one to use is a matter of style; `move` should be used for casual decomposition, such as splitting a pair, and `case` should be used for actual decompositions, in particular for type families (see *Type families*) and proof by contradiction.

The trivial branching *i_pattern* can be used to force the branching interpretation, e.g.:

- `case=> [] [a b] c.`
- `move=> [[a b] c].`
- `case; case=> a b c.`

are all equivalent.

Block introduction

SSReflect supports the following *i_block*s:

[^ *ident*] *block destructing* *i_pattern*. It performs a case analysis on the top variable and introduces, in one go, all the variables coming from the case analysis. The names of these variables are obtained by taking the names used in the inductive type declaration and prefixing them with *ident*. If the intro pattern immediately follows a call to `elim` with a custom eliminator (see *Interpreting eliminations*) then the names are taken from the ones used in the type of the eliminator.

Example

```
Record r := { a : nat; b := (a, 3); _ : bool; }.
    r is defined
    a is defined
```

(continues on next page)

(continued from previous page)

```
    b is defined


Lemma test : r -> True.
    1 goal

      ============================
      r -> True

Proof. move => [^ x ].
    1 goal

      xa : nat
      xb := (xa, 3) : nat * nat
      _x?_ : bool
      ============================
      True
```

[^~ *ident*] *block destructing* using *ident* as a suffix.

[^~ *natural*] *block destructing* using *natural* as a suffix.

Only a *s_item* is allowed between the elimination tactic and the block destructing.

Generation of equations

The generation of named equations option stores the definition of a new constant as an equation. The tactic:

```
move En: (size l) => n.
```

where `l` is a list, replaces `size l` by `n` in the goal and adds the fact `En : size l = n` to the context. This is quite different from:

```
pose n := (size l).
```

which generates a definition `n := (size l)`. It is not possible to generalize or rewrite such a definition; on the other hand, it is automatically expanded during computation, whereas expanding the equation `En` requires explicit rewriting.

The use of this equation name generation option with a `case` or an `elim` tactic changes the status of the first *i_item*, in order to deal with the possible parameters of the constants introduced.

Example

```
Lemma test (a b :nat) : a <> b.
    1 goal

      a, b : nat
      ============================
      a <> b

case E : a => [|n].
    2 goals

      a, b : nat
```

(continues on next page)

(continued from previous page)

```
    E : a = 0
    ============================
    0 <> b

goal 2 is:
 S n <> b
```

If the user does not provide a branching *i_item* as first *i_item*, or if the *i_item* does not provide enough names for the arguments of a constructor, then the constants generated are introduced under fresh SSReflect names.

Example

```
Lemma test (a b :nat) : a <> b.
    1 goal

    a, b : nat
    ============================
    a <> b

case E : a => H.
    2 goals

    a, b : nat
    E : a = 0
    H : 0 = b
    ============================
    False

goal 2 is:
 False

Show 2.
    goal 2 is:

    a, b, _n_ : nat
    E : a = S _n_
    H : S _n_ = b
    ============================
    False
```

Combining the generation of named equations mechanism with the *case* tactic strengthens the power of a case analysis. On the other hand, when combined with the *elim* tactic, this feature is mostly useful for debug purposes, to trace the values of decomposed parameters and pinpoint failing branches.

Type families

When the top assumption of a goal has an inductive type, two specific operations are possible: the case analysis performed by the *case* tactic, and the application of an induction principle, performed by the *elim* tactic. When this top assumption has an inductive type, which is moreover an instance of a type family, Coq may need help from the user to specify which occurrences of the parameters of the type should be substituted.

Variant: `case`: *d_item*+ **/** *d_item*+

Variant: `elim`: *d_item*+ **/** *d_item*+

A specific / switch indicates the type family parameters of the type of a *d_item* immediately following this / switch. The *d_item* on the right side of the / switch are discharged as described in section *Discharge*. The case analysis or elimination will be done on the type of the top assumption after these discharge operations.

Every *d_item* preceding the / is interpreted as arguments of this type, which should be an instance of an inductive type family. These terms are not actually generalized, but rather selected for substitution. Occurrence switches can be used to restrict the substitution. If a term is left completely implicit (e.g. writing just _), then a pattern is inferred looking at the type of the top assumption. This allows for the compact syntax:

```
case: {2}_ / eqP.
```

where _ is interpreted as `(_ == _)` since `eqP T a b : reflect (a = b) (a == b)` and reflect is a type family with one index.

Moreover if the *d_item* list is too short, it is padded with an initial sequence of _ of the right length.

Example

Here is a small example on lists. We define first a function which adds an element at the end of a given list.

```
Require Import List.
Section LastCases.
Variable A : Type.
    A is declared

Implicit Type l : list A.
Fixpoint add_last a l : list A :=
  match l with
 | nil => a :: nil
 | hd :: tl => hd :: (add_last a tl) end.
    add_last is defined
    add_last is recursively defined (guarded on 2nd argument)
```

Then we define an inductive predicate for case analysis on lists according to their last element:

```
Inductive last_spec : list A -> Type :=
| LastSeq0 : last_spec nil
| LastAdd s x : last_spec (add_last x s).
    last_spec is defined
    last_spec_rect is defined
    last_spec_ind is defined
    last_spec_rec is defined
    last_spec_sind is defined


Theorem lastP : forall l : list A, last_spec l.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      A : Type
      ============================
      forall l, last_spec l

Admitted.
    lastP is declared
```

We are now ready to use `lastP` in conjunction with `case`.

```
Lemma test l : (length l) * 2 = length (l ++ l).
    1 goal

      A : Type
      l : list A
      ============================
      length l * 2 = length (l ++ l)

case: (lastP l).
    2 goals

      A : Type
      l : list A
      ============================
      length nil * 2 = length (nil ++ nil)

    goal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (add_last x s ++ add_last x s)
```

Applied to the same goal, the tactc `case: l / (lastP l)` generates the same subgoals but `l` has been cleared from both contexts:

```
case: l / (lastP l).
    2 goals

      A : Type
      ============================
      length nil * 2 = length (nil ++ nil)

    goal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (add_last x s ++ add_last x s)
```

Again applied to the same goal:

```
case: {1 3}l / (lastP l).
    2 goals

      A : Type
      l : list A
      ============================
      length nil * 2 = length (l ++ nil)

    goal 2 is:
     forall (s : list A) (x : A),
     length (add_last x s) * 2 = length (l ++ add_last x s)
```

Note that selected occurrences on the left of the / switch have been substituted with l instead of being affected by the case analysis.

The equation name generation feature combined with a type family / switch generates an equation for the *first* dependent *d_item* specified by the user. Again starting with the above goal, the command:

Example

```
Lemma test l : (length l) * 2 = length (l ++ l).
    1 goal

      A : Type
      l : list A
      ============================
      length l * 2 = length (l ++ l)

case E: {1 3}l / (lastP l) => [|s x].
    2 goals

      A : Type
      l : list A
      E : l = nil
      ============================
      length nil * 2 = length (l ++ nil)

    goal 2 is:
     length (add_last x s) * 2 = length (l ++ add_last x s)

Show 2.
    goal 2 is:

      A : Type
      l, s : list A
      x : A
      E : l = add_last x s
      ============================
      length (add_last x s) * 2 = length (l ++ add_last x s)
```

There must be at least one *d_item* to the left of the / switch; this prevents any confusion with the view feature. However, the *d_item* to the right of the / are optional, and if they are omitted the first assumption provides the instance of the type family.

The equation always refers to the first *d_item* in the actual tactic call, before any padding with initial _. Thus, if an inductive type has two family parameters, it is possible to have SSReflect generate an equation for the second one by omitting the pattern for the first; note however that this will fail if the type of the second parameter depends on the value of the first parameter.

Control flow

Indentation and bullets

A linear development of Coq scripts gives little information on the structure of the proof. In addition, replaying a proof after some changes in the statement to be proved will usually not display information to distinguish between the various branches of case analysis for instance.

To help the user in this organization of the proof script at development time, SSReflect provides some bullets to highlight the structure of branching proofs. The available bullets are `-`, `+` and `*`. Combined with tabulation, this lets us highlight four nested levels of branching; the most we have ever needed is three. Indeed, the use of "simpl and closing" switches, of terminators (see above section *Terminators*) and selectors (see section *Selectors*) is powerful enough to avoid most of the time more than two levels of indentation.

Here is a fragment of such a structured script:

```
case E1: (abezoutn _ _) => [[| k1] [| k2]].
- rewrite !muln0 !gexpn0 mulg1 => H1.
  move/eqP: (sym_equal F0); rewrite -H1 orderg1 eqn_mul1.
  by case/andP; move/eqP.
- rewrite muln0 gexpn0 mulg1 => H1.
  have F1: t %| t * S k2.+1 - 1.
    apply: (@dvdn_trans (orderg x)); first by rewrite F0; exact: dvdn_mull.
    rewrite orderg_dvd; apply/eqP; apply: (mulgI x).
    rewrite -{1}(gexpn1 x) mulg1 gexpn_add leq_add_sub //.
    by move: P1; case t.
  rewrite dvdn_subr in F1; last by exact: dvdn_mulr.
  + rewrite H1 F0 -{2}(muln1 (p ^ l)); congr (_ * _).
    by apply/eqP; rewrite -dvdn1.
  + by move: P1; case: (t) => [| [| s1]].
- rewrite muln0 gexpn0 mul1g => H1.
...
```

Terminators

To further structure scripts, SSReflect supplies *terminating* tacticals to explicitly close off tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closed tactic fails to prove its subgoal.

It is hence recommended practice that the proof of any subgoal should end with a tactic which *fails if it does not solve the current goal*, like `discriminate`, `contradiction` or `assumption`.

In fact, SSReflect provides a generic tactical which turns any tactic into a closing one (similar to `now`). Its general syntax is:

Tactic: `by` *`tactic`*

The Ltac expression **`by [`*`tactic`*` | `*`tactic`*` | …]`** is equivalent to **`do [done | by`** *`tactic`* **`| by`** *`tactic`* **`| …]`**, which corresponds to the standard Ltac expression **`first [done |`** *`tactic`***`; done |`** *`tactic`***`; done | …]`**.

In the script provided as example in section *Indentation and bullets*, the paragraph corresponding to each sub-case ends with a tactic line prefixed with a `by`, like in:

```
by apply/eqP; rewrite -dvdn1.
```

Tactic: `done`

The `by` tactical is implemented using the user-defined, and extensible `done` tactic. This `done` tactic tries to solve

the current goal by some trivial means and fails if it doesn't succeed. Indeed, the tactic expression **by** *tactic* is equivalent to *tactic*; **done**.

Conversely, the tactic `by [ ]` is equivalent to *done*.

The default implementation of the done tactic, in the `ssreflect.v` file, is:

```
Ltac done :=
  trivial; hnf; intros; solve
   [ do ![solve [trivial | apply: sym_equal; trivial]
         | discriminate | contradiction | split]
   | case not_locked_false_eq_true; assumption
   | match goal with H : ~ _ |- _ => solve [case H; trivial] end ].
```

The lemma `not_locked_false_eq_true` is needed to discriminate *locked* boolean predicates (see section *Locking, unlocking*). The iterator tactical do is presented in section *Iteration*. This tactic can be customized by the user, for instance to include an *auto* tactic.

A natural and common way of closing a goal is to apply a lemma which is the exact one needed for the goal to be solved. The defective form of the tactic:

```
exact.
```

is equivalent to:

```
do [done | by move=> top; apply top].
```

where `top` is a fresh name assigned to the top assumption of the goal. This applied form is supported by the `:` discharge tactical, and the tactic:

```
exact: MyLemma.
```

is equivalent to:

```
by apply: MyLemma.
```

(see section *Discharge* for the documentation of the apply: combination).

Warning: The list of tactics (possibly chained by semicolons) that follows the `by` keyword is considered to be a parenthesized block applied to the current goal. Hence for example if the tactic:

```
by rewrite my_lemma1.
```

succeeds, then the tactic:

```
by rewrite my_lemma1; apply my_lemma2.
```

usually fails since it is equivalent to:

```
by (rewrite my_lemma1; apply my_lemma2).
```

Selectors

Tactic: `last`
Tactic: `first`

When composing tactics, the two tacticals `first` and `last` let the user restrict the application of a tactic to only one of the subgoals generated by the previous tactic. This covers the frequent cases where a tactic generates two subgoals one of which can be easily disposed of.

This is another powerful way of linearization of scripts, since it happens very often that a trivial subgoal can be solved in a less than one line tactic. For instance, ***tactic* ; last by *tactic*** tries to solve the last subgoal generated by the first tactic using the given second tactic, and fails if it does not succeed. Its analogue ***tactic* ; first by *tactic*** tries to solve the first subgoal generated by the first tactic using the second given tactic, and fails if it does not succeed.

SSReflect also offers an extension of this facility, by supplying tactics to *permute* the subgoals generated by a tactic.

Variant: `last first`
Variant: `first last`

These two equivalent tactics invert the order of the subgoals in focus.

Variant: `last` *`natural`* `first`

If *`natural`*'s value is k, this tactic rotates the n subgoals G_1 , ..., G_n in focus. Subgoal G_{n+1-k} becomes the first, and the circular order of subgoals remains unchanged.

Tactic: `first` *`natural`* `last`

If *`natural`*'s value is k, this tactic rotates the n subgoals G_1 , ..., G_n in focus. Subgoal $G_{k+1 \bmod n}$ becomes the first, and the circular order of subgoals remains unchanged.

Finally, the tactics `last` and `first` combine with the branching syntax of Ltac: if the tactic generates n subgoals on a given goal, then the tactic

```
tactic ; last k [ tactic1 |…| tacticm ] || tacticn.
```

where natural denotes the integer k as above, applies tactic1 to the $n-k+1$-th goal, ... tacticm to the $n-k+2$-th goal and tacticn to the others.

Example

Here is a small example on lists. We define first a function which adds an element at the end of a given list.

```
Inductive test : nat -> Prop :=
| C1 n of n = 1 : test n
| C2 n of n = 2 : test n
| C3 n of n = 3 : test n
| C4 n of n = 4 : test n.
    test is defined
    test_ind is defined
    test_sind is defined


Lemma example n (t : test n) : True.
    1 goal

      n : nat
      t : test n
      ============================
      True
```

(continues on next page)

(continued from previous page)

```
case: t; last 2 [move=> k| move=> l]; idtac.
    4 goals

      n : nat
      ============================
      forall n0 : nat, n0 = 1 -> True

    goal 2 is:
     k = 2 -> True
    goal 3 is:
     l = 3 -> True
    goal 4 is:
     forall n0 : nat, n0 = 4 -> True
```

Iteration

Tactic: do *mult*? | *tactic* | [*tactic*+|]

This tactical offers an accurate control on the repetition of tactics. *mult* is a *multiplier*.

Brackets can only be omitted if a single tactic is given *and* a multiplier is present.

A tactic of the form:

```
do [ tactic 1 | … | tactic n ].
```

is equivalent to the standard Ltac expression:

```
first [ tactic 1 | … | tactic n ].
```

The optional multiplier *mult* specifies how many times the action of tactic should be repeated on the current subgoal.

There are four kinds of multipliers:

mult ::= *natural*! | ! | *natural*? | ?

Their meaning is:

- n! the step tactic is repeated exactly n times (where n is a positive integer argument).
- ! the step tactic is repeated as many times as possible, and done at least once.
- ? the step tactic is repeated as many times as possible, optionally.
- n? the step tactic is repeated up to n times, optionally.

For instance, the tactic:

```
tactic; do 1? rewrite mult_comm.
```

rewrites at most one time the lemma mult_comm in all the subgoals generated by tactic, whereas the tactic:

```
tactic; do 2! rewrite mult_comm.
```

rewrites exactly two times the lemma mult_comm in all the subgoals generated by tactic, and fails if this rewrite is not possible in some subgoal.

Note that the combination of multipliers and rewrite is so often used that multipliers are in fact integrated to the syntax of the SSReflect rewrite tactic, see section *Rewriting*.

Localization

In sections *Basic localization* and *Bookkeeping*, we have already presented the *localization* tactical in, whose general syntax is:

Tactic: ***tactic*** **in** ***ident*** [+] ***** [?]

where *ident* is a name in the context. On the left side of `in`, *tactic* can be `move`, `case`, `elim`, `rewrite`, `set`, or any tactic formed with the general iteration tactical `do` (see section *Iteration*).

The operation described by tactic is performed in the facts listed after `in` and in the goal if a `*` ends the list of names.

The `in` tactical successively:

- generalizes the selected hypotheses, possibly "protecting" the goal if `*` is not present,
- performs *tactic*, on the obtained goal,
- reintroduces the generalized facts, under the same names.

This defective form of the `do` tactical is useful to avoid clashes between standard Ltac in and the SSReflect tactical in.

Example

```
Ltac mytac H := rewrite H.
    mytac is defined


Lemma test x y (H1 : x = y) (H2 : y = 3) : x + y = 6.
    1 goal

      x, y : nat
      H1 : x = y
      H2 : y = 3
      ============================
      x + y = 6

do [mytac H2] in H1 *.
    1 goal

      x, y : nat
      H2 : y = 3
      H1 : x = 3
      ============================
      x + 3 = 6
```

the last tactic rewrites the hypothesis `H2 : y = 3` both in `H1 : x = y` and in the goal `x + y = 6`.

By default `in` keeps the body of local definitions. To erase the body of a local definition during the generalization phase, the name of the local definition must be written between parentheses, like in `rewrite H in H1 (def_n) H2.`

Variant:

tactic **in** (***clear_switch*** | [?] **@** ***ident*** | **(** ***ident*** **)** | [?] **@** ***ident*** **:=** ***c_pattern*** **)** [+] ***** [?]

This is the most general form of the `in` tactical. In its simplest form the last option lets one rename hypotheses

that can't be cleared (like section variables). For example, `(y := x)` generalizes over `x` and reintroduces the generalized variable under the name `y` (and does not clear `x`). For a more precise description of this form of localization refer to *Advanced generalization*.

Structure

Forward reasoning structures the script by explicitly specifying some assumptions to be added to the proof context. It is closely associated with the declarative style of proof, since an extensive use of these highlighted statements make the script closer to a (very detailed) textbook proof.

Forward chaining tactics allow to state an intermediate lemma and start a piece of script dedicated to the proof of this statement. The use of closing tactics (see section *Terminators*) and of indentation makes syntactically explicit the portion of the script building the proof of the intermediate statement.

The have tactic.

Tactic: `have : term`

This is the main SSReflect forward reasoning tactic. It can be used in two modes: one starts a new (sub)proof for an intermediate result in the main proof, and the other provides explicitly a proof term for this intermediate step.

This tactic supports open syntax for *term*. Applied to a goal `G`, it generates a first subgoal requiring a proof of *term* in the context of `G`. The second generated subgoal is of the form **`term -> G`**, where term becomes the new top assumption, instead of being introduced with a fresh name. At the proof-term level, the have tactic creates a β redex, and introduces the lemma under a fresh name, automatically chosen.

Like in the case of the **`pose <pose (ssreflect)>`** tactic (see section *Definitions*), the types of the holes are abstracted in term.

Example

```
Lemma test : True.
    1 goal

      ============================
      True

have: _ * 0 = 0.
    2 goals

      ============================
      forall n : nat, n * 0 = 0

    goal 2 is:
     (forall n : nat, n * 0 = 0) -> True
```

The invocation of `have` is equivalent to:

```
have: forall n : nat, n * 0 = 0.
    2 goals

      ============================
      forall n : nat, n * 0 = 0

    goal 2 is:
     (forall n : nat, n * 0 = 0) -> True
```

The have tactic also enjoys the same abstraction mechanism as the `pose` tactic for the non-inferred implicit arguments. For instance, the tactic:

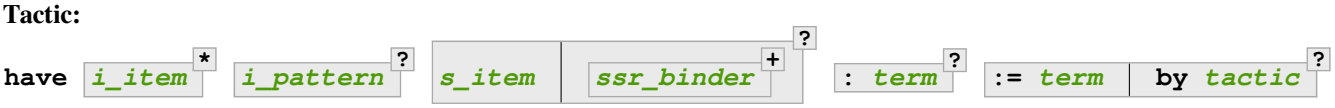
Example

```
have: forall x y, (x, y) = (x, y + 0).
    2 goals

      ============================
      forall (T : Type) (x : T) (y : nat), (x, y) = (x, y + 0)

    goal 2 is:
     (forall (T : Type) (x : T) (y : nat), (x, y) = (x, y + 0)) -> True
```

opens a new subgoal where the type of `x` is quantified.

The behavior of the defective have tactic makes it possible to generalize it in the following general construction:

Tactic:



Open syntax is supported for both *term*. For the description of *i_item* and *s_item* see section *Introduction in the context*. The first mode of the have tactic, which opens a sub-proof for an intermediate result, uses tactics of the form:

Variant: have *clear_switch i_item* : *term* by *tactic*

which behave like:

```
have: term ; first by tactic.
move=> clear_switch i_item.
```

Note that the *clear_switch* *precedes* the *i_item*, which allows to reuse a name of the context, possibly used by the proof of the assumption, to introduce the new assumption itself.

The `by` feature is especially convenient when the proof script of the statement is very short, basically when it fits in one line like in:

```
have H23 : 3 + 2 = 2 + 3 by rewrite addnC.
```

The possibility of using *i_item* supplies a very concise syntax for the further use of the intermediate step. For instance,

Example

```
Lemma test a : 3 * a - 1 = a.
    1 goal

      a : nat
      ============================
      3 * a - 1 = a

have -> : forall x, x * a = a.
    2 goals

      a : nat
```

(continues on next page)

(continued from previous page)

```
  ============================
  forall x : nat, x * a = a

goal 2 is:
 a - 1 = a
```

Note how the second goal was rewritten using the stated equality. Also note that in this last subgoal, the intermediate result does not appear in the context.

Thanks to the deferred execution of clears, the following idiom is also supported (assuming x occurs in the goal only):

```
have {x} -> : x = y.
```

Another frequent use of the intro patterns combined with `have` is the destruction of existential assumptions like in the tactic:

Example

```
Lemma test : True.
1 goal

  ============================
  True

have [x Px]: exists x : nat, x > 0; last first.
2 goals

  x : nat
  Px : x > 0
  ============================
  True

goal 2 is:
 exists x : nat, x > 0
```

An alternative use of the `have` tactic is to provide the explicit proof term for the intermediate lemma, using tactics of the form:

Variant: have *ident*? **:=** *term*

This tactic creates a new assumption of type the type of *term*. If the optional *ident* is present, this assumption is introduced under the name *ident*. Note that the body of the constant is lost for the user.

Again, non inferred implicit arguments and explicit holes are abstracted.

Example

```
Lemma test : True.
1 goal

  ============================
  True

have H := forall x, (x, x) = (x, x).
```

(continues on next page)

(continued from previous page)

```
    1 goal

      H : Type -> Prop
      ============================
      True
```

adds to the context `H : Type -> Prop.` This is a schematic example but the feature is specially useful when the proof term to give involves for instance a lemma with some hidden implicit arguments.

After the *i_pattern*, a list of binders is allowed.

Example

```
Lemma test : True.
    1 goal

      ============================
      True

have H x (y : nat) : 2 * x + y = x + x + y by lia.
    1 goal

      H : forall x y : nat, 2 * x + y = x + x + y
      ============================
      True
```

A proof term provided after `:=` can mention these bound variables (that are automatically introduced with the given names). Since the *i_pattern* can be omitted, to avoid ambiguity, bound variables can be surrounded with parentheses even if no type is specified:

```
have (x) : 2 * x = x + x by lia.
    1 goal

      ============================
      (forall x : nat, 2 * x = x + x) -> True
```

The *i_item* and *s_item* can be used to interpret the asserted hypothesis with views (see section *Views and reflection*) or simplify the resulting goals.

The *have* tactic also supports a `suff` modifier which allows for asserting that a given statement implies the current goal without copying the goal itself.

Example

```
have suff H : 2 + 2 = 3; last first.
    2 goals

      H : 2 + 2 = 3 -> True
      ============================
      True

    goal 2 is:
     2 + 2 = 3 -> True
```

Note that H is introduced in the second goal.

The `suff` modifier is not compatible with the presence of a list of binders.

Generating let in context entries with have

Since SSReflect 1.5 the *have* tactic supports a "transparent" modifier to generate let in context entries: the `@` symbol in front of the context entry name.

Example

```
Inductive Ord n := Sub x of x < n.
    Ord is defined
    Ord_rect is defined
    Ord_ind is defined
    Ord_rec is defined
    Ord_sind is defined

Notation "'I_ n" := (Ord n) (at level 8, n at level 2, format "''I_' n").
Arguments Sub {_} _ _.

Lemma test n m (H : m + 1 < n) : True.
    1 goal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have @i : 'I_n by apply: (Sub m); lia.
    1 goal

      n, m : nat
      H : m + 1 < n
      i := Sub m
             (ZifyClasses.rew_iff_rev (m < n) (Z.of_nat m < Z.of_nat n)%Z
                (ZifyClasses.mkrel nat Z lt Z.of_nat Z.lt Nat2Z.inj_lt m
                   (...) eq_refl n (...) eq_refl)
                (let H0 : ...%Z := ... in ... ...)) : 'I_n
      ============================
      True
```

Note that the subterm produced by *lia* is in general huge and uninteresting, and hence one may want to hide it. For this purpose the `[: name ]` intro pattern and the tactic `abstract` (see *The abstract tactic*) are provided.

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 goal

      n, m : nat
      H : m + 1 < n
      ============================
```

(continues on next page)

(continued from previous page)

```
      True

have [:pm] @i : 'I_n by apply: (Sub m); abstract: pm; lia.
    1 goal

      n, m : nat
      H : m + 1 < n
      pm : m < n (*1*)
      i := Sub m pm : 'I_n
      ============================
      True
```

The type of `pm` can be cleaned up by its annotation `(*1*)` by just simplifying it. The annotations are there for technical reasons only.

When intro patterns for abstract constants are used in conjunction with have and an explicit term, they must be used as follows:

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 goal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have [:pm] @i : 'I_n := Sub m pm.
    2 goals

      n, m : nat
      H : m + 1 < n
      ============================
      S m <= n

    goal 2 is:
     True

  by lia.
    1 goal

      n, m : nat
      H : m + 1 < n
      pm : S m <= n (*1*)
      i := (Sub m pm : 'I_n) : 'I_n
      ============================
      True
```

In this case the abstract constant `pm` is assigned by using it in the term that follows `:=` and its corresponding goal is left to be solved. Goals corresponding to intro patterns for abstract constants are opened in the order in which the abstract constants are declared (not in the "order" in which they are used in the term).

Note that abstract constants do respect scopes. Hence, if a variable is declared after their introduction, it has to be properly generalized (i.e. explicitly passed to the abstract constant when one makes use of it).

Example

```
Lemma test n m (H : m + 1 < n) : True.
    1 goal

      n, m : nat
      H : m + 1 < n
      ============================
      True

have [:pm] @i k : 'I_(n+k) by apply: (Sub m); abstract: pm k; lia.
    1 goal

      n, m : nat
      H : m + 1 < n
      pm : (forall k : nat, m < n + k)  (*1*)
      i := fun k : nat => Sub m (pm k) : forall k : nat, 'I_(n + k)
      ============================
      True
```

Last, notice that the use of intro patterns for abstract constants is orthogonal to the transparent flag `@` for have.

The have tactic and typeclass resolution

Since SSReflect 1.5 the `have` tactic behaves as follows with respect to typeclass inference.

```
have foo : ty.
    2 goals

      ============================
      ty

    goal 2 is:
     True
```

Full inference for `ty`. The first subgoal demands a proof of such instantiated statement.

```
have foo : ty := .
```

No inference for `ty`. Unresolved instances are quantified in `ty`. The first subgoal demands a proof of such quantified statement. Note that no proof term follows `:=`, hence two subgoals are generated.

```
have foo : ty := t.
    1 goal

      foo : ty
      ============================
      True
```

No inference for `ty` and `t`.

```
have foo := t.
    1 goal
```

(continues on next page)

(continued from previous page)

```
foo : ty
============================
True
```

No inference for `t`. Unresolved instances are quantified in the (inferred) type of `t` and abstracted in `t`.

Flag: `SsrHave NoTCResolution`
This *flag* restores the behavior of SSReflect 1.4 and below (never resolve typeclasses).

Variants: the suff and wlog tactics

As it is often the case in mathematical textbooks, forward reasoning may be used in slightly different variants. One of these variants is to show that the intermediate step L easily implies the initial goal G. By easily we mean here that the proof of L ⇒ G is shorter than the one of L itself. This kind of reasoning step usually starts with: "It suffices to show that …".

This is such a frequent way of reasoning that SSReflect has a variant of the `have` tactic called `suffices` (whose abridged name is `suff`). The `have` and `suff` tactics are equivalent and have the same syntax but:

- the order of the generated subgoals is inverted
- the optional clear item is still performed in the *second* branch. This means that the tactic:

  ```
  suff {H} H : forall x : nat, x >= 0.
  ```

  fails if the context of the current goal indeed contains an assumption named `H`.

The rationale of this clearing policy is to make possible "trivial" refinements of an assumption, without changing its name in the main branch of the reasoning.

The `have` modifier can follow the `suff` tactic.

Example

```
Lemma test : G.
    1 goal

      ============================
      G

suff have H : P.
    2 goals

      H : P
      ============================
      G

    goal 2 is:
     (P -> G) -> G
```

Note that, in contrast with `have suff`, the name H has been introduced in the first goal.

Another useful construct is reduction, showing that a particular case is in fact general enough to prove a general property. This kind of reasoning step usually starts with: "Without loss of generality, we can suppose that …". Formally, this corresponds to the proof of a goal `G` by introducing a cut `wlog_statement -> G`. Hence the user shall provide a proof for both `(wlog_statement -> G) -> G` and `wlog_statement -> G`. However, such cuts are usually

rather painful to perform by hand, because the statement `wlog_statement` is tedious to write by hand, and sometimes even to read.

SSReflect implements this kind of reasoning step through the *without loss* tactic, whose short name is *wlog*. It offers support to describe the shape of the cut statements, by providing the simplifying hypothesis and by pointing at the elements of the initial goals which should be generalized. The general syntax of without loss is:

Tactic: wlog **suff**? *clear_switch*? *i_item*? : *ident** / *term*

Tactic: without loss **suff**? *clear_switch*? *i_item*? : *ident** / *term*

where each *ident* is a constant in the context of the goal. Open syntax is supported for *term*.

In its defective form:

Variant: wlog: / *term*
Variant: without loss: / *term*

on a goal G, it creates two subgoals: a first one to prove the formula (term -> G) -> G and a second one to prove the formula term -> G.

If the optional list of *ident* is present on the left side of `/`, these constants are generalized in the premise (term -> G) of the first subgoal. By default bodies of local definitions are erased. This behavior can be inhibited by prefixing the name of the local definition with the `@` character.

In the second subgoal, the tactic:

```
move=> clear_switch i_item.
```

is performed if at least one of these optional switches is present in the *wlog* tactic.

The *wlog* tactic is specially useful when a symmetry argument simplifies a proof. Here is an example showing the beginning of the proof that quotient and reminder of natural number euclidean division are unique.

Example

```
Lemma quo_rem_unicity d q1 q2 r1 r2 :
  q1*d + r1 = q2*d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2).
    1 goal

      d, q1, q2, r1, r2 : nat
      ============================
      q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)

wlog: q1 q2 r1 r2 / q1 <= q2.
    2 goals

      d, q1, q2, r1, r2 : nat
      ============================
      (forall q3 q4 r3 r4 : nat,
       q3 <= q4 ->
       q3 * d + r3 = q4 * d + r4 -> r3 < d -> r4 < d -> (q3, r3) = (q4, r4)) ->
      q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)

    goal 2 is:
     q1 <= q2 ->
     q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)

  by case (le_gt_dec q1 q2)=> H; last symmetry; eauto with arith.
    1 goal
```

(continues on next page)

(continued from previous page)

```
  d, q1, q2, r1, r2 : nat
  ============================
  q1 <= q2 ->
  q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (q2, r2)
```

The `wlog suff` variant is simpler, since it cuts `wlog_statement` instead of `wlog_statement -> G`. It thus opens the goals `wlog_statement -> G` and `wlog_statement`.

In its simplest form the `generally have : ...` tactic is equivalent to `wlog suff : ...` followed by last first. When the `have` tactic is used with the `generally` (or `gen`) modifier it accepts an extra identifier followed by a comma before the usual intro pattern. The identifier will name the new hypothesis in its more general form, while the intro pattern will be used to process its instance.

Example

```
Lemma simple n (ngt0 : 0 < n ) : P n.
    1 goal

  n : nat
  ngt0 : 0 < n
  ============================
  P n

gen have ltnV, /andP[nge0 neq0] : n ngt0 / (0 <= n) && (n != 0); last first.
    2 goals

  n : nat
  ngt0 : 0 < n
  ltnV : forall n : nat, 0 < n -> (0 <= n) && (n != 0)
  nge0 : 0 <= n
  neq0 : n != 0
  ============================
  P n

  goal 2 is:
   (0 <= n) && (n != 0)
```

Advanced generalization

The complete syntax for the items on the left hand side of the `/` separator is the following one:

Variant: `wlog … :` *clear_switch* | `@`? *ident* | `(` `@`? *ident* `:=` *c_pattern*`)` `?` `/` *term*

Clear operations are intertwined with generalization operations. This helps in particular avoiding dependency issues while generalizing some facts.

If an *ident* is prefixed with the `@` mark, then a let-in redex is created, which keeps track if its body (if any). The syntax **(*ident* := *c_pattern*)** allows to generalize an arbitrary term using a given name. Note that its simplest form `(x := y)` is just a renaming of `y` into `x`. In particular, this can be useful in order to simulate the generalization of a section variable, otherwise not allowed. Indeed renaming does not require the original variable to be cleared.

The syntax `(@x := y)` generates a let-in abstraction but with the following caveat: `x` will not bind `y`, but its body, whenever `y` can be unfolded. This covers the case of both local and global definitions, as illustrated in the following example.

Example

```
Section Test.
Variable x : nat.
    x is declared

Definition addx z := z + x.
    addx is defined

Lemma test : x <= addx x.
    1 goal

      x : nat
      ============================
      x <= addx x

wlog H : (y := x) (@twoy := addx x) / twoy = 2 * y.
    2 goals

      x : nat
      ============================
      (forall y : nat, let twoy := y + y in twoy = 2 * y -> y <= twoy) ->
      x <= addx x

    goal 2 is:
     y <= twoy
```

To avoid unfolding the term captured by the pattern add x one can use the pattern `id (addx x)`, that would produce the following first subgoal

```
wlog H : (y := x) (@twoy := id (addx x)) / twoy = 2 * y.
    2 goals

      x : nat
      ============================
      (forall y : nat, let twoy := addx y in twoy = 2 * y -> y <= addx y) ->
      x <= addx x

    goal 2 is:
     y <= addx y
```

Rewriting

The generalized use of reflection implies that most of the intermediate results handled are properties of effectively computable functions. The most efficient mean of establishing such results are computation and simplification of expressions involving such functions, i.e., rewriting. SSReflect therefore includes an extended `rewrite` tactic, that unifies and combines most of the rewriting functionalities.

An extended rewrite tactic

The main features of the rewrite tactic are:

- It can perform an entire series of such operations in any subset of the goal and/or context;
- It allows to perform rewriting, simplifications, folding/unfolding of definitions, closing of goals;
- Several rewriting operations can be chained in a single tactic;
- Control over the occurrence at which rewriting is to be performed is significantly enhanced.

The general form of an SSReflect rewrite tactic is:

Tactic: `rewrite` *rstep*+

The combination of a rewrite tactic with the `in` tactical (see section *Localization*) performs rewriting in both the context and the goal.

A rewrite step *rstep* has the general form:

rstep ::= *r_prefix*? *r_item*

r_prefix ::= -? *mult*? *occ_switch* | *clear_switch*? [*r_pattern*]?

r_pattern ::= *term* | in *ident* in? *term* | *term* in | *term* as *ident* in *term*

r_item ::= /? *term* | *s_item*

An *r_prefix* contains annotations to qualify where and how the rewrite operation should be performed:

- The optional initial − indicates the direction of the rewriting of *r_item*: if present the direction is right-to-left and it is left-to-right otherwise.
- The multiplier *mult* (see section *Iteration*) specifies if and how the rewrite operation should be repeated.
- A rewrite operation matches the occurrences of a *rewrite pattern*, and replaces these occurrences by another term, according to the given *r_item*. The optional *redex switch* `[r_pattern]`, which should always be surrounded by brackets, gives explicitly this rewrite pattern. In its simplest form, it is a regular term. If no explicit redex switch is present the rewrite pattern to be matched is inferred from the *r_item*.
- This optional term, or the *r_item*, may be preceded by an *occ_switch* (see section *Selectors*) or a *clear_switch* (see section *Discharge*), these two possibilities being exclusive.

  An occurrence switch selects the occurrences of the rewrite pattern which should be affected by the rewrite operation.

  A clear switch, even an empty one, is performed *after* the *r_item* is actually processed and is complemented with the name of the rewrite rule if an only if it is a simple proof context entry[7]. As a consequence one can write

`rewrite {}H` to rewrite with `H` and dispose `H` immediately afterwards. This behavior can be avoided by putting parentheses around the rewrite rule.

An *r_item* can be:

- A *simplification* *r_item*, represented by a *s_item* (see section *Introduction in the context*). Simplification operations are intertwined with the possible other rewrite operations specified by the list of *r_item*.
- A *folding/unfolding* *r_item*. The tactic: `rewrite /term` unfolds the head constant of term in every occurrence of the first matching of term in the goal. In particular, if `my_def` is a (local or global) defined constant, the tactic: `rewrite /my_def.` is analogous to: `unfold my_def`. Conversely: `rewrite -/my_def.` is equivalent to: `fold my_def`. When an unfold *r_item* is combined with a redex pattern, a conversion operation is performed. A tactic of the form: `rewrite -[term1]/term2.` is equivalent to: `change term1 with term2.` If `term2` is a single constant and `term1` head symbol is not `term2`, then the head symbol of `term1` is repeatedly unfolded until `term2` appears.
- **A *term*, which can be:**
  - A term whose type has the form: `forall (x1 : A1 )…(xn : An ), eq term1 term2` where `eq` is the Leibniz equality or a registered setoid equality.
  - A list of terms `(t1 ,…,tn)`, each `ti` having a type above. The tactic: `rewrite r_prefix (t1 ,…,tn ).` is equivalent to: `do [rewrite r_prefix t1 | … | rewrite r_prefix tn ].`
  - An anonymous rewrite lemma `(_ : term)`, where term has a type as above.

Example

```
Definition double x := x + x.
    double is defined

Definition ddouble x := double (double x).
    ddouble is defined

Lemma test x : ddouble x = 4 * x.
    1 goal

      x : nat
      ============================
      ddouble x = 4 * x

rewrite [ddouble _]/double.
    1 goal

      x : nat
      ============================
      double x + double x = 4 * x
```

Warning: The SSReflect terms containing holes are *not* typed as abstractions in this context. Hence the following script fails.

```
Definition f := fun x y => x + y.
    f is defined

Lemma test x y : x + y = f y x.
    1 goal
```

```
      x, y : nat
      ============================
      x + y = f y x

rewrite -[f y](y + _).
    Toplevel input, characters 0-22:
    > rewrite -[f y](y + _).
    > ^^^^^^^^^^^^^^^^^^^^^^
    Error: fold pattern (y + _) does not match redex (f y)
```

but the following script succeeds

```
rewrite -[f y x](y + _).
    1 goal

      x, y : nat
      ============================
      x + y = y + x
```

Flag: `SsrOldRewriteGoalsOrder`
Controls the order in which generated subgoals (side conditions) are added to the proof context. The *flag* is off by default, which puts subgoals generated by conditional rules first, followed by the main goal. When it is on, the main goal appears first. If your proofs are organized to complete proving the main goal before side conditions, turning the flag on will save you from having to add `last first` tactics that would be needed to keep the main goal as the currently focused goal.

Remarks and examples

Rewrite redex selection

The general strategy of SSReflect is to grasp as many redexes as possible and to let the user select the ones to be rewritten thanks to the improved syntax for the control of rewriting.

This may be a source of incompatibilities between the two rewrite tactics.

In a rewrite tactic of the form:

```
rewrite occ_switch [term1]term2.
```

`term1` is the explicit rewrite redex and `term2` is the rewrite rule. This execution of this tactic unfolds as follows:

- First `term1` and `term2` are βι normalized. Then `term2` is put in head normal form if the Leibniz equality constructor `eq` is not the head symbol. This may involve ζ reductions.
- Then, the matching algorithm (see section *Abbreviations*) determines the first subterm of the goal matching the rewrite pattern. The rewrite pattern is given by `term1`, if an explicit redex pattern switch is provided, or by the type of `term2` otherwise. However, matching skips over matches that would lead to trivial rewrites. All the occurrences of this subterm in the goal are candidates for rewriting.
- Then only the occurrences coded by `occ_switch` (see again section *Abbreviations*) are finally selected for rewriting.
- The left hand side of `term2` is unified with the subterm found by the matching algorithm, and if this succeeds, all the selected occurrences in the goal are replaced by the right hand side of `term2`.
- Finally the goal is βι normalized.

In the case `term2` is a list of terms, the first top-down (in the goal) left-to-right (in the list) matching rule gets selected.

Chained rewrite steps

The possibility to chain rewrite operations in a single tactic makes scripts more compact and gathers in a single command line a bunch of surgical operations which would be described by a one sentence in a pen and paper proof.

Performing rewrite and simplification operations in a single tactic enhances significantly the concision of scripts. For instance the tactic:

```
rewrite /my_def {2}[f _]/= my_eq //=.
```

unfolds `my_def` in the goal, simplifies the second occurrence of the first subterm matching pattern `[f _]`, rewrites `my_eq`, simplifies the goals and closes trivial goals.

Here are some concrete examples of chained rewrite operations, in the proof of basic results on natural numbers arithmetic.

Example

```
Axiom addn0 : forall m, m + 0 = m.
    addn0 is declared

Axiom addnS : forall m n, m + S n = S (m + n).
    addnS is declared

Axiom addSnnS : forall m n, S m + n = m + S n.
    addSnnS is declared


Lemma addnCA m n p : m + (n + p) = n + (m + p).
    1 goal

      m, n, p : nat
      ============================
      m + (n + p) = n + (m + p)

by elim: m p => [ | m Hrec] p; rewrite ?addSnnS -?addnS.
    No more goals.

Qed.

Lemma addnC n m : m + n = n + m.
    1 goal

      n, m : nat
      ============================
      m + n = n + m

by rewrite -{1}[n]addn0 addnCA addn0.
    No more goals.

Qed.
```

Note the use of the `?` switch for parallel rewrite operations in the proof of `addnCA`.

Explicit redex switches are matched first

If an *r_prefix* involves a *redex switch*, the first step is to find a subterm matching this redex pattern, independently from the left hand side of the equality the user wants to rewrite.

Example

```
Lemma test (H : forall t u, t + u = u + t) x y : x + y = y + x.
    1 goal

      H : forall t u : nat, t + u = u + t
      x, y : nat
      ============================
      x + y = y + x

rewrite [y + _]H.
    1 goal

      H : forall t u : nat, t + u = u + t
      x, y : nat
      ============================
      x + y = x + y
```

Note that if this first pattern matching is not compatible with the *r_item*, the rewrite fails, even if the goal contains a correct redex matching both the redex switch and the left hand side of the equality.

Example

```
Lemma test (H : forall t u, t + u * 0 = t) x y : x + y * 4 + 2 * 0 = x + 2 * 0.
    1 goal

      H : forall t u : nat, t + u * 0 = t
      x, y : nat
      ============================
      x + y * 4 + 2 * 0 = x + 2 * 0

Fail rewrite [x + _]H.
    The command has indeed failed with message:
    pattern (x + y * 4) does not match LHS of H
```

Indeed the left hand side of `H` does not match the redex identified by the pattern `x + y * 4`.

Occurrence switches and redex switches

Example

```
Lemma test x y : x + y + 0 = x + y + y + 0 + 0 + (x + y + 0).
    1 goal

      x, y : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
      x + y + 0 = x + y + y + 0 + 0 + (x + y + 0)

rewrite {2}[_ + y + 0](_: forall z, z + 0 = z).
    2 goals

      x, y : nat
      ============================
      forall z : nat, z + 0 = z

    goal 2 is:
     x + y + 0 = x + y + y + 0 + 0 + (x + y)
```

The second subgoal is generated by the use of an anonymous lemma in the rewrite tactic. The effect of the tactic on the initial goal is to rewrite this lemma at the second occurrence of the first matching `x + y + 0` of the explicit rewrite redex `_ + y + 0`.

Occurrence selection and repetition

Occurrence selection has priority over repetition switches. This means the repetition of a rewrite tactic specified by a multiplier will perform matching each time an elementary rewrite operation is performed. Repeated rewrite tactics apply to every subgoal generated by the previous tactic, including the previous instances of the repetition.

Example

```
Lemma test x y (z : nat) : x + 1 = x + y + 1.
    1 goal

      x, y, z : nat
      ============================
      x + 1 = x + y + 1

rewrite 2!(_ : _ + 1 = z).
    4 goals

      x, y, z : nat
      ============================
      x + 1 = z

    goal 2 is:
     z = z
    goal 3 is:
     x + y + 1 = z
    goal 4 is:
     z = z
```

This last tactic generates *three* subgoals because the second rewrite operation specified with the `2!` multiplier applies to the two subgoals generated by the first rewrite.

Multi-rule rewriting

The rewrite tactic can be provided a *tuple* of rewrite rules, or more generally a tree of such rules, since this tuple can feature arbitrary inner parentheses. We call *multirule* such a generalized rewrite rule. This feature is of special interest when it is combined with multiplier switches, which makes the rewrite tactic iterate the rewrite operations prescribed by the rules on the current goal.

Example

```
Variables (a b c : nat).
    a is declared
    b is declared
    c is declared

Hypothesis eqab : a = b.
    eqab is declared

Hypothesis eqac : a = c.
    eqac is declared


Lemma test : a = a.
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      ============================
      a = a

rewrite (eqab, eqac).
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      ============================
      b = b
```

Indeed rule `eqab` is the first to apply among the ones gathered in the tuple passed to the rewrite tactic. This multirule `(eqab, eqac)` is actually a Coq term and we can name it with a definition:

```
Definition multi1 := (eqab, eqac).
    multi1 is defined
```

In this case, the tactic `rewrite multi1` is a synonym for `rewrite (eqab, eqac).`

More precisely, a multirule rewrites the first subterm to which one of the rules applies in a left-to-right traversal of the goal, with the first rule from the multirule tree in left-to-right order. Matching is performed according to the algorithm described in Section *Abbreviations*, but literal matches have priority.

Example

```
Definition d := a.
    d is defined
```

(continues on next page)

(continued from previous page)

```
Hypotheses eqd0 : d = 0.
    eqd0 is declared

Definition multi2 := (eqab, eqd0).
    multi2 is defined


Lemma test : d = b.
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      ============================
      d = b

rewrite multi2.
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      ============================
      0 = b
```

Indeed rule `eqd0` applies without unfolding the definition of `d`.

For repeated rewrites the selection process is repeated anew.

Example

```
Hypothesis eq_adda_b : forall x, x + a = b.
    eq_adda_b is declared

Hypothesis eq_adda_c : forall x, x + a = c.
    eq_adda_c is declared

Hypothesis eqb0 : b = 0.
    eqb0 is declared

Definition multi3 := (eq_adda_b, eq_adda_c, eqb0).
    multi3 is defined


Lemma test : 1 + a = 12 + a.
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      eq_adda_b : forall x : nat, x + a = b
```

(continues on next page)

(continued from previous page)

```
      eq_adda_c : forall x : nat, x + a = c
      eqb0 : b = 0
      ============================
      1 + a = 12 + a

rewrite 2!multi3.
    1 goal

      a, b, c : nat
      eqab : a = b
      eqac : a = c
      eqd0 : d = 0
      eq_adda_b : forall x : nat, x + a = b
      eq_adda_c : forall x : nat, x + a = c
      eqb0 : b = 0
      ============================
      0 = 12 + a
```

It uses `eq_adda_b` then `eqb0` on the left-hand side only. Without the bound `2` one would obtain `0 = 0`.

The grouping of rules inside a multirule does not affect the selection strategy but can make it easier to include one rule set in another or to (universally) quantify over the parameters of a subset of rules (as there is special code that will omit unnecessary quantifiers for rules that can be syntactically extracted). It is also possible to reverse the direction of a rule subset, using a special dedicated syntax: the tactic rewrite `(=~ multi1)` is equivalent to `rewrite multi1_rev`.

Example

```
Hypothesis eqba : b = a.
    eqba is declared

Hypothesis eqca : c = a.
    eqca is declared

Definition multi1_rev := (eqba, eqca).
    multi1_rev is defined
```

---

except that the constants `eqba`, `eqab`, `mult1_rev` have not been created.

Rewriting with multirules is useful to implement simplification or transformation procedures, to be applied on terms of small to medium size. For instance the library `ssrnat` (Mathematical Components library) provides two implementations for arithmetic operations on natural numbers: an elementary one and a tail recursive version, less inefficient but also less convenient for reasoning purposes. The library also provides one lemma per such operation, stating that both versions return the same values when applied to the same arguments:

```
Lemma addE : add =2 addn.
Lemma doubleE : double =1 doublen.
Lemma add_mulE n m s : add_mul n m s = addn (muln n m) s.
Lemma mulE : mul =2 muln.
Lemma mul_expE m n p : mul_exp m n p = muln (expn m n) p.
Lemma expE : exp =2 expn.
Lemma oddE : odd =1 oddn.
```

The operation on the left hand side of each lemma is the efficient version, and the corresponding naive implementation is on the right hand side. In order to reason conveniently on expressions involving the efficient operations, we gather all these rules in the definition `trecE`:

```
Definition trecE := (addE, (doubleE, oddE), (mulE, add_mulE, (expE, mul_expE))).
```

The tactic: `rewrite !trecE.` restores the naive versions of each operation in a goal involving the efficient ones, e.g. for the purpose of a correctness proof.

Wildcards vs abstractions

The rewrite tactic supports *r_item*s containing holes. For example, in the tactic `rewrite (_ : _ * 0 = 0).` the term `_ * 0 = 0` is interpreted as `forall n : nat, n * 0 = 0.` Anyway this tactic is *not* equivalent to `rewrite (_ : forall x, x * 0 = 0).`.

Example

```
Lemma test y z : y * 0 + y * (z * 0) = 0.
    1 goal

      y, z : nat
      ============================
      y * 0 + y * (z * 0) = 0

rewrite (_ : _ * 0 = 0).
    2 goals

      y, z : nat
      ============================
      y * 0 = 0

    goal 2 is:
     0 + y * (z * 0) = 0
```

while the other tactic results in

```
rewrite (_ : forall x, x * 0 = 0).
    2 goals

      y, z : nat
      ============================
      forall x : nat, x * 0 = 0

    goal 2 is:
     0 + y * (z * 0) = 0
```

The first tactic requires you to prove the instance of the (missing) lemma that was used, while the latter requires you prove the quantified form.

When SSReflect rewrite fails on standard Coq licit rewrite

In a few cases, the SSReflect rewrite tactic fails rewriting some redexes which standard Coq successfully rewrites. There are two main cases:

- SSReflect never accepts to rewrite indeterminate patterns like:

```
Lemma foo (x : unit) : x = tt.
```

  SSReflect will however accept the ηζ expansion of this rule:

```
Lemma fubar (x : unit) : (let u := x in u) = tt.
```

- The standard rewrite tactic provided by Coq uses a different algorithm to find instances of the rewrite rule.

Example

```
Variable g : nat -> nat.
    g is declared

Definition f := g.
    f is defined

Axiom H : forall x, g x = 0.
    H is declared


Lemma test : f 3 + f 3 = f 6.
    1 goal

      g : nat -> nat
      ============================
      f 3 + f 3 = f 6

(* we call the standard rewrite tactic here *)
rewrite -> H.
    1 goal

      g : nat -> nat
      ============================
      0 + 0 = f 6
```

This rewriting is not possible in SSReflect because there is no occurrence of the head symbol `f` of the rewrite rule in the goal.

```
rewrite H.
    Toplevel input, characters 0-9:
    > rewrite H.
    > ^^^^^^^^^
    Error: The LHS of H
        (g _)
    does not match any subterm of the goal
```

Rewriting with `H` first requires unfolding the occurrences of `f` where the substitution is to be performed (here there is a single such occurrence), using tactic `rewrite /f` (for a global replacement of f by g) or `rewrite pattern/f`, for a finer selection.

```
rewrite /f H.
    1 goal

      g : nat -> nat
      ============================
      0 + 0 = g 6
```

alternatively one can override the pattern inferred from `H`

```
rewrite [f _]H.
    1 goal

      g : nat -> nat
      ============================
      0 + 0 = f 6
```

Existential metavariables and rewriting

The rewrite tactic will not instantiate existing existential metavariables when matching a redex pattern.

If a rewrite rule generates a goal with new existential metavariables in the `Prop` sort, these will be generalized as for `apply` (see *The apply tactic*) and corresponding new goals will be generated.

Example

```
Axiom leq : nat -> nat -> bool.
    leq is declared

Notation "m <= n" := (leq m n) : nat_scope.
Notation "m < n"  := (S m <= n) : nat_scope.
Inductive Ord n := Sub x of x < n.
    Ord is defined
    Ord_rect is defined
    Ord_ind is defined
    Ord_rec is defined
    Ord_sind is defined

Notation "'I_ n" := (Ord n) (at level 8, n at level 2, format "''I_' n").
Arguments Sub {_} _ _.
Definition val n (i : 'I_n) := let: Sub a _ := i in a.
    val is defined

Definition insub n x :=
  if @idP (x < n) is ReflectT _ Px then Some (Sub x Px) else None.
    insub is defined

Axiom insubT : forall n x Px, insub n x = Some (Sub x Px).
    insubT is declared


Lemma test (x : 'I_2) y : Some x = insub 2 y.
    1 goal

      x : 'I_2
```

(continues on next page)

(continued from previous page)

```
      y : nat
      ============================
      Some x = insub 2 y

rewrite insubT.
    2 goals

      x : 'I_2
      y : nat
      ============================
      forall Hyp0 : y < 2, Some x = Some (Sub y Hyp0)

    goal 2 is:
     y < 2
```

Since the argument corresponding to Px is not supplied by the user, the resulting goal should be `Some x = Some (Sub y ?Goal)`. Instead, SSReflect `rewrite` tactic hides the existential variable.

As in *The apply tactic*, the `ssrautoprop` tactic is used to try to solve the existential variable.

```
Lemma test (x : 'I_2) y (H : y < 2) : Some x = insub 2 y.
    1 goal

      x : 'I_2
      y : nat
      H : y < 2
      ============================
      Some x = insub 2 y

rewrite insubT.
    1 goal

      x : 'I_2
      y : nat
      H : y < 2
      ============================
      Some x = Some (Sub y H)
```

As a temporary limitation, this behavior is available only if the rewriting rule is stated using Leibniz equality (as opposed to setoid relations). It will be extended to other rewriting relations in the future.

Rewriting under binders

Goals involving objects defined with higher-order functions often require "rewriting under binders". While setoid rewriting is a possible approach in this case, it is common to use regular rewriting along with dedicated extensionality lemmas. This may cause some practical issues during the development of the corresponding scripts, notably as we might be forced to provide the rewrite tactic with complete terms, as shown by the simple example below.

Example

```
Axiom subnn : forall n : nat, n - n = 0.
Parameter map : (nat -> nat) -> list nat -> list nat.
Parameter sumlist : list nat -> nat.
```

(continues on next page)

(continued from previous page)

```
Axiom eq_map :
  forall F1 F2 : nat -> nat,
  (forall n : nat, F1 n = F2 n) ->
  forall l : list nat, map F1 l = map F2 l.

Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 goal

      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0
```

In this context, one cannot directly use `eq_map`:

```
rewrite eq_map.
    Toplevel input, characters 0-14:
    > rewrite eq_map.
    > ^^^^^^^^^^^^^^
    Error: Unable to find an instance for the variable F2.
    Rule's type:
    (forall F1 F2 : nat -> nat,
     (forall n : nat, F1 n = F2 n) -> forall l : list nat, map F1 l = map F2 l)
```

as we need to explicitly provide the non-inferable argument `F2`, which corresponds here to the term we want to obtain *after* the rewriting step. In order to perform the rewrite step one has to provide the term by hand as follows:

```
rewrite (@eq_map _ (fun _ : nat => 0)).
    2 goals

      l : list nat
      ============================
      forall n : nat, n - n = 0

    goal 2 is:
     sumlist (map (fun _ : nat => 0) l) = 0

  by move=> m; rewrite subnn.
    1 goal

      l : list nat
      ============================
      sumlist (map (fun _ : nat => 0) l) = 0
```

The *under* tactic lets one perform the same operation in a more convenient way:

```
Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 goal

      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0

under eq_map => m do rewrite subnn.
    1 goal

      l : list nat
```

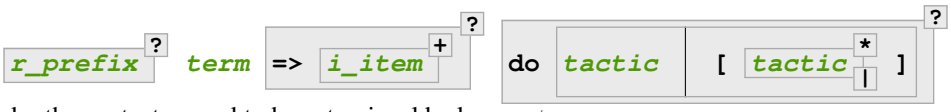
(continues on next page)

(continued from previous page)

```
  ============================
  sumlist (map (fun _ : nat => 0) l) = 0
```

The under tactic

The convenience *under* tactic supports the following syntax:



Operate under the context proved to be extensional by lemma *term*.

Error: Incorrect number of tactics (expected N tactics, was given M).
This error can occur when using the version with a `do` clause.

The multiplier part of *r_prefix* is not supported.

We distinguish two modes, *interactive mode* without a `do` clause, and *one-liner mode* with a `do` clause, which are explained in more detail below.

Interactive mode

Let us redo the running example in interactive mode.

Example

```
Lemma example_map l : sumlist (map (fun m => m - m) l) = 0.
    1 goal

      l : list nat
      ============================
      sumlist (map (fun m : nat => m - m) l) = 0

under eq_map => m.
    2 focused goals
    (shelved: 2)

      l : list nat
      m : nat
      ============================
      'Under[ m - m ]

    goal 2 is:
     sumlist (map ?Goal l) = 0

  rewrite subnn.
    2 focused goals
    (shelved: 2)

      l : list nat
      m : nat
      ============================
```

(continues on next page)

(continued from previous page)

```
    'Under[ 0 ]

  goal 2 is:
   sumlist (map ?Goal l) = 0

over.
  1 focused goal
  (shelved: 1)

    l : list nat
    ============================
    sumlist (map (fun _ : nat => 0) l) = 0
```

The execution of the Ltac expression:

under *term* **=> [** i_item_1 **|** … **|** i_item_n **].**

involves the following steps:

1. It performs a **rewrite** *term* without failing like in the first example with `rewrite eq_map.`, but creating evars (see *evar*). If **term** is prefixed by a pattern or an occurrence selector, then the modifiers are honoured.
2. As a n-branches intro pattern is provided *under* checks that n+1 subgoals have been created. The last one is the main subgoal, while the other ones correspond to premises of the rewrite rule (such as `forall n, F1 n = F2 n` for `eq_map`).
3. If so *under* puts these n goals in head normal form (using the defective form of the tactic *move*), then executes the corresponding intro pattern $i\_pattern_i$ in each goal.
4. Then *under* checks that the first n subgoals are (quantified) Leibniz equalities, double implications or registered relations (w.r.t. Class `RewriteRelation`) between a term and an evar, e.g. `m - m = ?F2 m` in the running example. (This support for setoid-like relations is enabled as soon as we do both `Require Import ssreflect.` and `Require Setoid.`)
5. If so *under* protects these n goals against an accidental instantiation of the evar. These protected goals are displayed using the `'Under[ … ]` notation (e.g. `'Under[ m - m ]` in the running example).
6. The expression inside the `'Under[ … ]` notation can be proved equivalent to the desired expression by using a regular *rewrite* tactic.
7. Interactive editing of the first n goals has to be signalled by using the *over* tactic or rewrite rule (see below), which requires that the underlying relation is reflexive. (The running example deals with Leibniz equality, but `PreOrder` relations are also supported, for example.)
8. Finally, a post-processing step is performed in the main goal to keep the name(s) for the bound variables chosen by the user in the intro pattern for the first branch.

The over tactic

Two equivalent facilities (a terminator and a lemma) are provided to close intermediate subgoals generated by *under* (i.e. goals displayed as `'Under[ … ]`):

Tactic: over
: This terminator tactic allows one to close goals of the form `'Under[ … ]`.

Variant: by rewrite over
: This is a variant of *over* in order to close `'Under[ … ]` goals, relying on the `over` rewrite rule.

Note that a rewrite rule `UnderE` is available as well, if one wants to "unprotect" the evar, without closing the goal automatically (e.g., to instantiate it manually with another rule than reflexivity).

One-liner mode

The Ltac expression:

under *term* **=>** **[** *i_item*$_1$ **|** … **|** *i_item*$_n$ **]** **do** **[** *tactic*$_1$ **|** … **|** *tactic*$_n$ **].**

can be seen as a shorter form for the following expression:

(under *term***) => [** *i_item*$_1$ **|** … **|** *i_item*$_n$ **|]; [** *tactic*$_1$**; over |** … **|** *tactic*$_n$**; over | cbv beta iota].**

Notes:

- The `beta-iota` reduction here is useful to get rid of the beta redexes that could be introduced after the substitution of the evars by the *under* tactic.
- Note that the provided tactics can as well involve other *under* tactics. See below for a typical example involving the `bigop` theory from the Mathematical Components library.
- If there is only one tactic, the brackets can be omitted, e.g.: **under** *term* **=> i do** *tactic***.** and that shorter form should be preferred.
- If the `do` clause is provided and the intro pattern is omitted, then the default *i_item* `*` is applied to each branch. E.g., the Ltac expression: **under** *term* **do [** *tactic*$_1$ **|** … **|** *tactic*$_n$ **]** is equivalent to: **under** *term* **=> [* |** … **| *] do [** *tactic*$_1$ **|** … **|** *tactic*$_n$ **]** (and it can be noted here that the *under* tactic performs a `move.` before processing the intro patterns `=> [ * | … | * ]`).

Example

```
Parameter addnC : forall m n : nat, m + n = n + m.
Parameter muln1 : forall n : nat, n * 1 = n.

Check eq_bigr.
    eq_bigr
         : forall (n m : nat) (P : nat -> bool) (F1 F2 : nat -> nat),
           (forall x : nat, P x -> F1 x = F2 x) ->
           \sum_(n <= i < m | P i) F1 i = \sum_(n <= i < m | P i) F2 i

Check eq_big.
    eq_big
         : forall (n m : nat) (P1 P2 : nat -> bool) (F1 F2 : nat -> nat),
           (forall x : nat, P1 x = P2 x) ->
           (forall i : nat, P1 i -> F1 i = F2 i) ->
           \sum_(n <= i < m | P1 i) F1 i = \sum_(n <= i < m | P2 i) F2 i


Lemma test_big_nested (m n : nat) :
  \sum_(0 <= a < m | prime a) \sum_(0 <= j < n | odd (j * 1)) (a + j) =
  \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i).
    1 goal

      m, n : nat
      ============================
      \sum_(0 <= a < m | prime a) \sum_(0 <= j < n | odd (j * 1)) (a + j) =
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i)
```

(continues on next page)

(continued from previous page)

```
under eq_bigr => i prime_i do
  under eq_big => [ j | j odd_j ] do
    [ rewrite (muln1 j) | rewrite (addnC i j) ].
    1 goal

      m, n : nat
      ============================
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i) =
      \sum_(0 <= i < m | prime i) \sum_(0 <= j < n | odd j) (j + i)
```

Remark how the final goal uses the name `i` (the name given in the intro pattern) rather than `a` in the binder of the first summation.

Locking, unlocking

As program proofs tend to generate large goals, it is important to be able to control the partial evaluation performed by the simplification operations that are performed by the tactics. These evaluations can for example come from a `/=` simplification switch, or from rewrite steps which may expand large terms while performing conversion. We definitely want to avoid repeating large subterms of the goal in the proof script. We do this by "clamping down" selected function symbols in the goal, which prevents them from being considered in simplification or rewriting steps. This clamping is accomplished by using the occurrence switches (see section *Abbreviations*) together with "term tagging" operations.

SSReflect provides two levels of tagging.

The first one uses auxiliary definitions to introduce a provably equal copy of any term t. However this copy is (on purpose) *not convertible* to t in the Coq system[40]. The job is done by the following construction:

```
Lemma master_key : unit. Proof. exact tt. Qed.
Definition locked A := let: tt := master_key in fun x : A => x.
Lemma lock : forall A x, x = locked x :> A.
```

Note that the definition of *master_key* is explicitly opaque. The equation `t = locked t` given by the `lock` lemma can be used for selective rewriting, blocking on the fly the reduction in the term `t`.

Example

```
Variable A : Type.
    A is declared

Fixpoint has (p : A -> bool) (l : list A) : bool :=
  if l is cons x l then p x || (has p l) else false.
    has is defined
    has is recursively defined (guarded on 2nd argument)


Lemma test p x y l (H : p x = true) : has p ( x :: y :: l) = true.
    1 goal

      A : Type
      p : A -> bool
      x, y : A
```

(continues on next page)

[40] This is an implementation feature: there is no such obstruction in the metatheory

(continued from previous page)

```
      l : list A
      H : p x = true
      ============================
      has p (x :: y :: l) = true

rewrite {2}[cons]lock /= -lock.
    1 goal

      A : Type
      p : A -> bool
      x, y : A
      l : list A
      H : p x = true
      ============================
      p x || has p (y :: l) = true
```

It is sometimes desirable to globally prevent a definition from being expanded by simplification; this is done by adding locked in the definition.

Example

```
Definition lid := locked (fun x : nat => x).
    lid is defined


Lemma test : lid 3 = 3.
    1 goal

      ============================
      lid 3 = 3

rewrite /=.
    1 goal

      ============================
      lid 3 = 3

unlock lid.
    1 goal

      ============================
      3 = 3
```

Tactic: `unlock` *occ_switch*? *ident*

This tactic unfolds such definitions while removing "locks", i.e. it replaces the occurrence(s) of *ident* coded by the *occ_switch* with the corresponding body.

We found that it was usually preferable to prevent the expansion of some functions by the partial evaluation switch `/=`, unless this allowed the evaluation of a condition. This is possible thanks to another mechanism of term tagging, resting on the following *Notation*:

```
Notation "'nosimpl' t" := (let: tt := tt in t).
```

The term `(nosimpl t)` simplifies to `t` *except* in a definition. More precisely, given:

```
Definition foo := (nosimpl bar).
```

the term `foo` (or `(foo t')`) will *not* be expanded by the *simpl* tactic unless it is in a forcing context (e.g., in `match foo t' with … end`, `foo t'` will be reduced if this allows `match` to be reduced). Note that `nosimpl bar` is simply notation for a term that reduces to `bar`; hence `unfold foo` will replace `foo` by `bar`, and `fold foo` will replace `bar` by `foo`.

Warning: The `nosimpl` trick only works if no reduction is apparent in `t`; in particular, the declaration:

```
Definition foo x := nosimpl (bar x).
```

will usually not work. Anyway, the common practice is to tag only the function, and to use the following definition, which blocks the reduction as expected:

```
Definition foo x := nosimpl bar x.
```

A standard example making this technique shine is the case of arithmetic operations. We define for instance:

```
Definition addn := nosimpl plus.
```

The operation `addn` behaves exactly like `plus`, except that `(addn (S n) m)` will not simplify spontaneously to `(S (addn n m))` (the two terms, however, are convertible). In addition, the unfolding step: `rewrite /addn` will replace `addn` directly with `plus`, so the `nosimpl` form is essentially invisible.

Congruence

Because of the way matching interferes with parameters of type families, the tactic:

```
apply: my_congr_property.
```

will generally fail to perform congruence simplification, even on rather simple cases. We therefore provide a more robust alternative in which the function is supplied:

Tactic: `congr` *natural*? *term*

This tactic:

- checks that the goal is a Leibniz equality;
- matches both sides of this equality with "term applied to some arguments", inferring the right number of arguments from the goal and the type of term. This may expand some definitions or fixpoints;
- generates the subgoals corresponding to pairwise equalities of the arguments present in the goal.

The goal can be a non dependent product `P -> Q`. In that case, the system asserts the equation `P = Q`, uses it to solve the goal, and calls the `congr` tactic on the remaining goal `P = Q`. This can be useful for instance to perform a transitivity step, like in the following situation.

Example

```
Lemma test (x y z : nat) (H : x = y) : x = z.
    1 goal

      x, y, z : nat
      H : x = y
```

(continues on next page)

(continued from previous page)

```
      ============================
      x = z

congr (_ = _) : H.
    1 focused goal
    (shelved: 1)

      x, y, z : nat
      ============================
      y = z

Abort.

Lemma test (x y z : nat) : x = y -> x = z.
    1 goal

      x, y, z : nat
      ============================
      x = y -> x = z

congr (_ = _).
    1 focused goal
    (shelved: 1)

      x, y, z : nat
      ============================
      y = z
```

The optional *natural* forces the number of arguments for which the tactic should generate equality proof obligations.

This tactic supports equalities between applications with dependent arguments. Yet dependent arguments should have exactly the same parameters on both sides, and these parameters should appear as first arguments.

Example

```
Definition f n :=
  if n is 0 then plus else mult.
    f is defined

Definition g (n m : nat) := plus.
    g is defined


Lemma test x y : f 0 x y = g 1 1 x y.
    1 goal

      x, y : nat
      ============================
      f 0 x y = g 1 1 x y

congr plus.
    No more goals.
```

This script shows that the `congr` tactic matches `plus` with `f 0` on the left hand side and `g 1 1` on the right

hand side, and solves the goal.

Example

```
Lemma test n m (Hnm : m <= n) : S m + (S n - S m) = S n.
    1 goal

      n, m : nat
      Hnm : m <= n
      ============================
      S m + (S n - S m) = S n

congr S; rewrite -/plus.
    1 goal

      n, m : nat
      Hnm : m <= n
      ============================
      m + (S n - S m) = n
```

The tactic `rewrite -/plus` folds back the expansion of plus which was necessary for matching both sides of the equality with an application of `S`.

Like most SSReflect arguments, *term* can contain wildcards.

Example

```
Lemma test x y : x + (y * (y + x - x)) = x * 1 + (y + 0) * y.
    1 goal

      x, y : nat
      ============================
      x + y * (y + x - x) = x * 1 + (y + 0) * y

congr ( _ + (_ * _)).
    3 focused goals
    (shelved: 3)

      x, y : nat
      ============================
      x = x * 1

    goal 2 is:
     y = y + 0
    goal 3 is:
     y + x - x = y
```

Contextual patterns

The simple form of patterns used so far, terms possibly containing wild cards, often require an additional *occ_switch* to be specified. While this may work pretty fine for small goals, the use of polymorphic functions and dependent types may lead to an invisible duplication of function arguments. These copies usually end up in types hidden by the implicit arguments machinery or by user-defined notations. In these situations computing the right occurrence numbers is very tedious because they must be counted on the goal as printed after setting the *Printing All* flag. Moreover the resulting script is not really informative for the reader, since it refers to occurrence numbers he cannot easily see.

Contextual patterns mitigate these issues allowing to specify occurrences according to the context they occur in.

Syntax

The following table summarizes the full syntax of *c_pattern* and the corresponding subterm(s) identified by the pattern. In the third column we use s.m.r. for "the subterms matching the redex" specified in the second column.

c_pattern	redex	subterms affected
`term`	`term`	all occurrences of `term`
`ident in term`	subterm of `term` selected by `ident`	all the subterms identified by `ident` in all the occurrences of `term`
`term1 in ident in term2`	`term1` in all s.m.r.	in all the subterms identified by `ident` in all the occurrences of `term2`
`term1 as ident in term2`	`term 1`	in all the subterms identified by `ident` in all the occurrences of `term2[term 1 /ident]`

The rewrite tactic supports two more patterns obtained prefixing the first two with in. The intended meaning is that the pattern identifies all subterms of the specified context. The `rewrite` tactic will infer a pattern for the redex looking at the rule used for rewriting.

r_pattern	redex	subterms affected
`in term`	inferred from rule	in all s.m.r. in all occurrences of `term`
`in ident in term`	inferred from rule	in all s.m.r. in all the subterms identified by `ident` in all the occurrences of `term`

The first *c_pattern* is the simplest form matching any context but selecting a specific redex and has been described in the previous sections. We have seen so far that the possibility of selecting a redex using a term with holes is already a powerful means of redex selection. Similarly, any terms provided by the user in the more complex forms of *c_pattern*s presented in the tables above can contain holes.

For a quick glance at what can be expressed with the last *r_pattern* consider the goal `a = b` and the tactic

```
rewrite [in X in _ = X]rule.
```

It rewrites all occurrences of the left hand side of `rule` inside `b` only (`a`, and the hidden type of the equality, are ignored). Note that the variant `rewrite [X in _ = X]rule` would have rewritten `b` exactly (i.e., it would only work if `b` and the left hand side of rule can be unified).

Matching contextual patterns

The *c_pattern* and *r_pattern* involving terms with holes are matched against the goal in order to find a closed instantiation. This matching proceeds as follows:

c_pattern	instantiation order and place for term_i and redex
term	term is matched against the goal, redex is unified with the instantiation of term
ident in term	term is matched against the goal, redex is unified with the subterm of the instantiation of term identified by ident
term1 in ident in term2	term2 is matched against the goal, term1 is matched against the subterm of the instantiation of term1 identified by ident, redex is unified with the instantiation of term1
term1 as ident in term2	term2[term1/ident] is matched against the goal, redex is unified with the instantiation of term1

In the following patterns, the redex is intended to be inferred from the rewrite rule.

r_pattern	instantiation order and place for term_i and redex
in ident in term	term is matched against the goal, the redex is matched against the subterm of the instantiation of term identified by ident
in term	term is matched against the goal, redex is matched against the instantiation of term

Examples

Contextual pattern in set and the : tactical

As already mentioned in section *Abbreviations* the set tactic takes as an argument a term in open syntax. This term is interpreted as the simplest form of *c_pattern*. To avoid confusion in the grammar, open syntax is supported only for the simplest form of patterns, while parentheses are required around more complex patterns.

Example

```
Lemma test a b : a + b + 1 = b + (a + 1).
    1 goal

      a, b : nat
      ============================
      a + b + 1 = b + (a + 1)

set t := (X in _ = X).
    1 goal

      a, b : nat
      t := b + (a + 1) : nat
      ============================
      a + b + 1 = t

rewrite {}/t.
    1 goal

      a, b : nat
```

(continues on next page)

(continued from previous page)

```
      ============================
      a + b + 1 = b + (a + 1)

set t := (a + _ in X in _ = X).
    1 goal

      a, b : nat
      t := a + 1 : nat
      ============================
      a + b + 1 = b + t
```

Since the user may define an infix notation for `in` the result of the former tactic may be ambiguous. The disambiguation rule implemented is to prefer patterns over simple terms, but to interpret a pattern with double parentheses as a simple term. For example, the following tactic would capture any occurrence of the term `a in A`.

```
set t := ((a in A)).
```

Contextual patterns can also be used as arguments of the `:` tactical. For example:

```
elim: n (n in _ = n) (refl_equal n).
```

Contextual patterns in rewrite

Example

```
Notation "n .+1" := (Datatypes.S n) (at level 2, left associativity,
                    format "n .+1") : nat_scope.

Axiom addSn : forall m n, m.+1 + n = (m + n).+1.
    addSn is declared

Axiom addn0 : forall m, m + 0 = m.
    addn0 is declared

Axiom addnC : forall m n, m + n = n + m.
    addnC is declared


Lemma test x y z f : (x.+1 + y) + f (x.+1 + y) (z + (x + y).+1) = 0.
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      x.+1 + y + f (x.+1 + y) (z + (x + y).+1) = 0

rewrite [in f _ _]addSn.
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      x.+1 + y + f (x + y).+1 (z + (x + y).+1) = 0
```

Note: the simplification rule addSn is applied only under the f symbol. Then we simplify also the first addition and expand 0 into 0 + 0.

```
rewrite addSn -[X in _ = X]addn0.
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (x + y).+1) = 0 + 0
```

Note that the right hand side of addn0 is undetermined, but the rewrite pattern specifies the redex explicitly. The right hand side of addn0 is unified with the term identified by X, here 0.

The following pattern does not specify a redex, since it identifies an entire region, hence the rewrite rule has to be instantiated explicitly. Thus the tactic:

```
rewrite -{2}[in X in _ = X](addn0 0).
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (x + y).+1) = 0 + (0 + 0)
```

The following tactic is quite tricky:

```
rewrite [_.+1 in X in f _ X](addnC x.+1).
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (x + y).+1 (z + (y + x.+1)) = 0 + (0 + 0)
```

The explicit redex _.+1 is important since its head constant S differs from the head constant inferred from (addnC x.+1) (that is +). Moreover, the pattern f _ X is important to rule out the first occurrence of (x + y).+1. Last, only the subterms of f _ X identified by X are rewritten, thus the first argument of f is skipped too. Also note the pattern _.+1 is interpreted in the context identified by X, thus it gets instantiated to (y + x).+1 and not (x + y).+1.

The last rewrite pattern allows to specify exactly the shape of the term identified by X, that is thus unified with the left hand side of the rewrite rule.

```
rewrite [x.+1 + y as X in f X _]addnC.
    1 goal

      x, y, z : nat
      f : nat -> nat -> nat
      ============================
      (x + y).+1 + f (y + x.+1) (z + (y + x.+1)) = 0 + (0 + 0)
```

Patterns for recurrent contexts

The user can define shortcuts for recurrent contexts corresponding to the `ident in term` part. The notation scope identified with `%pattern` provides a special notation `(X in t)` the user must adopt in order to define context shortcuts.

The following example is taken from `ssreflect.v` where the `LHS` and `RHS` shortcuts are defined.

```
Notation RHS := (X in _ = X)%pattern.
Notation LHS := (X in X = _)%pattern.
```

Shortcuts defined this way can be freely used in place of the trailing `ident in term` part of any contextual pattern. Some examples follow:

```
set rhs := RHS.
rewrite [in RHS]rule.
case: (a + _ in RHS).
```

Views and reflection

The bookkeeping facilities presented in section *Basic tactics* are crafted to ease simultaneous introductions and generalizations of facts and operations of casing, naming etc. It also a common practice to make a stack operation immediately followed by an *interpretation* of the fact being pushed, that is, to apply a lemma to this fact before passing it to a tactic for decomposition, application and so on.

SSReflect provides a convenient, unified syntax to combine these interpretation operations with the proof stack operations. This *view mechanism* relies on the combination of the `/` view switch with bookkeeping tactics and tacticals.

Interpreting eliminations

The view syntax combined with the `elim` tactic specifies an elimination scheme to be used instead of the default, generated, one. Hence the SSReflect tactic:

```
elim/V.
```

is a synonym for:

```
intro top; elim top using V; clear top.
```

where top is a fresh name and V any second-order lemma.

Since an elimination view supports the two bookkeeping tacticals of discharge and introduction (see section *Basic tactics*), the SSReflect tactic:

```
elim/V: x => y.
```

is a synonym for:

```
elim x using V; clear x; intro y.
```

where `x` is a variable in the context, `y` a fresh name and `V` any second order lemma; SSReflect relaxes the syntactic restrictions of the Coq `elim`. The first pattern following `:` can be a _ wildcard if the conclusion of the view `V` specifies a pattern for its last argument (e.g., if `V` is a functional induction lemma generated by the `Function` command).

The elimination view mechanism is compatible with the equation name generation (see section *Generation of equations*).

Example

The following script illustrates a toy example of this feature. Let us define a function adding an element at the end of a list:

```
Variable d : Type.
    d is declared

Fixpoint add_last (s : list d) (z : d) {struct s} : list d :=
  if s is cons x s' then cons x (add_last s' z) else z :: nil.
    add_last is defined
    add_last is recursively defined (guarded on 1st argument)
```

One can define an alternative, reversed, induction principle on inductively defined lists, by proving the following lemma:

```
Axiom last_ind_list : forall P : list d -> Prop,
  P nil -> (forall s (x : d), P s -> P (add_last s x)) ->
    forall s : list d, P s.
    last_ind_list is declared
```

Then the combination of elimination views with equation names result in a concise syntax for reasoning inductively using the user-defined elimination scheme.

```
Lemma test (x : d) (l : list d): l = l.
    1 goal

      d : Type
      x : d
      l : list d
      ============================
      l = l

elim/last_ind_list E : l=> [| u v]; last first.
    2 goals

      d : Type
      x : d
      u : list d
      v : d
      l : list d
      E : l = add_last u v
      ============================
      u = u -> add_last u v = add_last u v

    goal 2 is:
     nil = nil
```

User-provided eliminators (potentially generated with Coq's `Function` command) can be combined with the type family switches described in section *Type families*. Consider an eliminator `foo_ind` of type:

```
foo_ind : forall …, forall x : T, P p1 … pm.
```

and consider the tactic:

```
elim/foo_ind: e1 … / en.
```

The `elim/` tactic distinguishes two cases:

truncated eliminator when `x` does not occur in `P p1 … pm` and the type of `en` unifies with `T` and `en` is not `_`. In that case, `en` is passed to the eliminator as the last argument (`x` in `foo_ind`) and `en-1 …`

`e1` are used as patterns to select in the goal the occurrences that will be bound by the predicate `P`, thus it must be possible to unify the subterm of the goal matched by `en-1` with `pm` , the one matched by `en-2` with `pm-1` and so on.

regular eliminator in all the other cases. Here it must be possible to unify the term matched by `en` with `pm` , the one matched by `en-1` with `pm-1` and so on. Note that standard eliminators have the shape `…forall x, P … x`, thus `en` is the pattern identifying the eliminated term, as expected.

As explained in section *Type families*, the initial prefix of `ei` can be omitted.

Here is an example of a regular, but nontrivial, eliminator.

Example

Here is a toy example illustrating this feature.

```
Function plus (m n : nat) {struct n} : nat :=
  if n is S p then S (plus m p) else m.
    plus is defined
    plus is recursively defined (guarded on 2nd argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
    plus_rec is defined
    R_plus_correct is defined
    R_plus_complete is defined


About plus_ind.
    plus_ind :
    forall [m : nat] [P : nat -> nat -> Prop],
    (forall n p : nat, n = S p -> P p (plus m p) -> P (S p) (S (plus m p))) ->
    (forall n _x : nat,
     n = _x -> match _x with
               | 0 => True
               | S _ => False
               end -> P _x m) -> forall n : nat, P n (plus m n)

    plus_ind is not universe polymorphic
    Arguments plus_ind [m]%nat_scope [P]%function_scope (_ _)%function_scope
      _%nat_scope
    plus_ind is transparent
    Expands to: Constant Top.Test.plus_ind


Lemma test x y z : plus (plus x y) z = plus x (plus y z).
    1 goal

      x, y, z : nat
      ============================
      plus (plus x y) z = plus x (plus y z)
```

The following tactics are all valid and perform the same elimination on this goal.

```
elim/plus_ind: z / (plus _ z).
elim/plus_ind: {z}(plus _ z).
elim/plus_ind: {z}_.
elim/plus_ind: z / _.
```

```
elim/plus_ind: z / _.
    2 goals

      x, y : nat
      ============================
      forall n p : nat,
      n = S p ->
      plus (plus x y) p = plus x (plus y p) ->
      S (plus (plus x y) p) = plus x (plus y (S p))

    goal 2 is:
     forall n _x : nat,
     n = _x ->
     match _x with
     | 0 => True
     | S _ => False
     end -> plus x y = plus x (plus y _x)
```

The two latter examples feature a wildcard pattern: in this case, the resulting pattern is inferred from the type of the eliminator. In both these examples, it is `(plus _ _)`, which matches the subterm `plus (plus x y) z` thus instantiating the last _ with `z`. Note that the tactic:

```
Fail elim/plus_ind: y / _.
    The command has indeed failed with message:
    The given pattern matches the term y while the inferred pattern z doesn't
```

triggers an error: in the conclusion of the `plus_ind` eliminator, the first argument of the predicate `P` should be the same as the second argument of `plus`, in the second argument of `P`, but `y` and `z` do no unify.

Here is an example of a truncated eliminator:

Example

Consider the goal:

```
Lemma test p n (n_gt0 : 0 < n) (pr_p : prime p) :
  p %| \prod_(i <- prime_decomp n | i \in prime_decomp n) i.1 ^ i.2 ->
    exists2 x : nat * nat, x \in prime_decomp n & p = x.1.
Proof.
elim/big_prop: _ => [| u v IHu IHv | [q e] /=].
```

where the type of the `big_prop` eliminator is

```
big_prop: forall (R : Type) (Pb : R -> Type)
  (idx : R) (op1 : R -> R -> R), Pb idx ->
  (forall x y : R, Pb x -> Pb y -> Pb (op1 x y)) ->
  forall (I : Type) (r : seq I) (P : pred I) (F : I -> R),
  (forall i : I, P i -> Pb (F i)) ->
    Pb (\big[op1/idx]_(i <- r | P i) F i).
```

Since the pattern for the argument of Pb is not specified, the inferred one is used instead: `big[_/_]_(i <- _ | _ i) _ i`, and after the introductions, the following goals are generated:

```
subgoal 1 is:
  p %| 1 -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 2 is:
```

(continues on next page)

(continued from previous page)

```
  p %| u * v -> exists2 x : nat * nat, x \in prime_decomp n & p = x.1
subgoal 3 is:
  (q, e) \in prime_decomp n -> p %| q ^ e ->
    exists2 x : nat * nat, x \in prime_decomp n & p = x.1.
```

Note that the pattern matching algorithm instantiated all the variables occurring in the pattern.

Interpreting assumptions

Interpreting an assumption in the context of a proof consists in applying to it a lemma before generalizing, and/or decomposing this assumption. For instance, with the extensive use of boolean reflection (see section *Views and reflection*), it is quite frequent to need to decompose the logical interpretation of (the boolean expression of) a fact, rather than the fact itself. This can be achieved by a combination of `move : _ => _` switches, like in the following example, where `||` is a notation for the boolean disjunction.

Example

```
Variables P Q : bool -> Prop.
    P is declared
    Q is declared

Hypothesis P2Q : forall a b, P (a || b) -> Q a.
    P2Q is declared


Lemma test a : P (a || a) -> True.
    1 goal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
      ============================
      P (a || a) -> True

move=> HPa; move: {HPa}(P2Q HPa) => HQa.
    1 goal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
      HQa : Q a
      ============================
      True
```

which transforms the hypothesis `HPa : P a` which has been introduced from the initial statement into `HQa : Q a`. This operation is so common that the tactic shell has specific syntax for it. The following scripts:

```
move=> HPa; move/P2Q: HPa => HQa.
    1 goal

      P, Q : bool -> Prop
      P2Q : forall a b : bool, P (a || b) -> Q a
      a : bool
```

(continues on next page)

(continued from previous page)

```
  HQa : Q a
  ============================
  True
```

or more directly:

```
move/P2Q=> HQa.
    1 goal

  P, Q : bool -> Prop
  P2Q : forall a b : bool, P (a || b) -> Q a
  a : bool
  HQa : Q a
  ============================
  True
```

are equivalent to the former one. The former script shows how to interpret a fact (already in the context), thanks to the discharge tactical (see section *Discharge*) and the latter, how to interpret the top assumption of a goal. Note that the number of wildcards to be inserted to find the correct application of the view lemma to the hypothesis has been automatically inferred.

The view mechanism is compatible with the `case` tactic and with the equation name generation mechanism (see section *Generation of equations*):

Example

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis Q2P : forall a b, Q (a || b) -> P a \/ P b.
    Q2P is declared


Lemma test a b : Q (a || b) -> True.
    1 goal

  P, Q : bool -> Prop
  Q2P : forall a b : bool, Q (a || b) -> P a \/ P b
  a, b : bool
  ============================
  Q (a || b) -> True

case/Q2P=> [HPa | HPb].
    2 goals

  P, Q : bool -> Prop
  Q2P : forall a b : bool, Q (a || b) -> P a \/ P b
  a, b : bool
  HPa : P a
  ============================
  True

goal 2 is:
 True
```

This view tactic performs:

```
move=> HQ; case: {HQ}(Q2P HQ) => [HPa | HPb].
```

The term on the right of the / view switch is called a *view lemma*. Any SSReflect term coercing to a product type can be used as a view lemma.

The examples we have given so far explicitly provide the direction of the translation to be performed. In fact, view lemmas need not to be oriented. The view mechanism is able to detect which application is relevant for the current goal.

Example

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.
    PQequiv is declared


Lemma test a b : P (a || b) -> True.
    1 goal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a, b : bool
      ============================
      P (a || b) -> True

move/PQequiv=> HQab.
    1 goal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a, b : bool
      HQab : Q a
      ============================
      True
```

has the same behavior as the first example above.

The view mechanism can insert automatically a *view hint* to transform the double implication into the expected simple implication. The last script is in fact equivalent to:

```
Lemma test a b : P (a || b) -> True.
move/(iffLR (PQequiv _ _)).
```

where:

```
Lemma iffLR P Q : (P <-> Q) -> P -> Q.
```

Specializing assumptions

The special case when the *head symbol* of the view lemma is a wildcard is used to interpret an assumption by *specializing* it. The view mechanism hence offers the possibility to apply a higher-order assumption to some given arguments.

Example

```
Lemma test z : (forall x y, x + y = z -> z = x) -> z = 0.
    1 goal

      z : nat
      ============================
      (forall x y : nat, x + y = z -> z = x) -> z = 0

move/(_ 0 z).
    1 goal

      z : nat
      ============================
      (0 + z = z -> z = 0) -> z = 0
```

Interpreting goals

In a similar way, it is also often convenient to changing a goal by turning it into an equivalent proposition. The view mechanism of SSReflect has a special syntax `apply/` for combining in a single tactic simultaneous goal interpretation operations and bookkeeping steps.

Example

The following example use the ~~ prenex notation for boolean negation:

```
Variables P Q: bool -> Prop.
    P is declared
    Q is declared

Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.
    PQequiv is declared


Lemma test a : P ((~~ a) || a).
    1 goal

      P, Q : bool -> Prop
      PQequiv : forall a b : bool, P (a || b) <-> Q a
      a : bool
      ============================
      P (~~ a || a)

apply/PQequiv.
    1 focused goal
    (shelved: 1)

      P, Q : bool -> Prop
```

(continues on next page)

(continued from previous page)

```
PQequiv : forall a b : bool, P (a || b) <-> Q a
a : bool
============================
Q (~~ a)
```

thus in this case, the tactic `apply/PQequiv` is equivalent to `apply: (iffRL (PQequiv _ _))`, where `iffRL` is the analogue of `iffRL` for the converse implication.

Any SSReflect term whose type coerces to a double implication can be used as a view for goal interpretation.

Note that the goal interpretation view mechanism supports both `apply` and `exact` tactics. As expected, a goal interpretation view command exact/term should solve the current goal or it will fail.

Warning: Goal interpretation view tactics are *not* compatible with the bookkeeping tactical => since this would be redundant with the `apply: term => _` construction.

Boolean reflection

In the Calculus of Inductive Constructions, there is an obvious distinction between logical propositions and boolean values. On the one hand, logical propositions are objects of *sort* `Prop` which is the carrier of intuitionistic reasoning. Logical connectives in `Prop` are *types*, which give precise information on the structure of their proofs; this information is automatically exploited by Coq tactics. For example, Coq knows that a proof of `A \/ B` is either a proof of `A` or a proof of `B`. The tactics `left` and `right` change the goal `A \/ B` to `A` and `B`, respectively; dually, the tactic `case` reduces the goal `A \/ B => G` to two subgoals `A => G` and `B => G`.

On the other hand, bool is an inductive *datatype* with two constructors true and false. Logical connectives on bool are *computable functions*, defined by their truth tables, using case analysis:

Example

```
Definition orb (b1 b2 : bool) := if b1 then true else b2.
    orb is defined
```

Properties of such connectives are also established using case analysis

---

Example

```
Lemma test b : b || ~~ b = true.
    1 goal

      b : bool
      ============================
      b || ~~ b = true

by case: b.
    No more goals.
```

Once `b` is replaced by `true` in the first goal and by `false` in the second one, the goals reduce by computations to the trivial `true = true`.

Thus, `Prop` and `bool` are truly complementary: the former supports robust natural deduction, the latter allows brute-force evaluation. SSReflect supplies a generic mechanism to have the best of the two worlds and move freely from a propositional version of a decidable predicate to its boolean version.

First, booleans are injected into propositions using the coercion mechanism:

```
Coercion is_true (b : bool) := b = true.
```

This allows any boolean formula `b` to be used in a context where Coq would expect a proposition, e.g., after `Lemma … :`. It is then interpreted as `(is_true b)`, i.e., the proposition `b = true`. Coercions are elided by the pretty-printer, so they are essentially transparent to the user.

The reflect predicate

To get all the benefits of the boolean reflection, it is in fact convenient to introduce the following inductive predicate `reflect` to relate propositions and booleans:

```
Inductive reflect (P: Prop): bool -> Type :=
| Reflect_true : P -> reflect P true
| Reflect_false : ~P -> reflect P false.
```

The statement `(reflect P b)` asserts that `(is_true b)` and `P` are logically equivalent propositions.

For instance, the following lemma:

```
Lemma andP: forall b1 b2, reflect (b1 /\ b2) (b1 && b2).
```

relates the boolean conjunction to the logical one `/\`. Note that in `andP`, `b1` and `b2` are two boolean variables and the proposition `b1 /\ b2` hides two coercions. The conjunction of `b1` and `b2` can then be viewed as `b1 /\ b2` or as `b1 && b2`.

Expressing logical equivalences through this family of inductive types makes possible to take benefit from *rewritable equations* associated to the case analysis of Coq's inductive types.

Since the equivalence predicate is defined in Coq as:

```
Definition iff (A B:Prop) := (A -> B) /\ (B -> A).
```

where `/\` is a notation for `and`:

```
Inductive and (A B:Prop) : Prop := conj : A -> B -> and A B.
```

This make case analysis very different according to the way an equivalence property has been defined.

```
Lemma andE (b1 b2 : bool) : (b1 /\ b2) <-> (b1 && b2).
```

Let us compare the respective behaviors of `andE` and `andP`.

Example

```
Lemma test (b1 b2 : bool) : if (b1 && b2) then b1 else ~~(b1||b2).
    1 goal

      b1, b2 : bool
      ============================
      if b1 && b2 then b1 else ~~ (b1 || b2)
```

```
case: (@andE b1 b2).
    1 goal

      b1, b2 : bool
      ============================
      (b1 /\ b2 -> b1 && b2) ->
      (b1 && b2 -> b1 /\ b2) -> if b1 && b2 then b1 else ~~ (b1 || b2)

case: (@andP b1 b2).
    2 goals

      b1, b2 : bool
      ============================
      b1 /\ b2 -> b1

    goal 2 is:
     ~ (b1 /\ b2) -> ~~ (b1 || b2)
```

Expressing reflection relation through the `reflect` predicate is hence a very convenient way to deal with classical reasoning, by case analysis. Using the `reflect` predicate allows moreover to program rich specifications inside its two constructors, which will be automatically taken into account during destruction. This formalisation style gives far more efficient specifications than quantified (double) implications.

A naming convention in SSReflect is to postfix the name of view lemmas with `P`. For example, `orP` relates `||` and `\/`, `negP` relates `~~` and `~`.

The view mechanism is compatible with reflect predicates.

Example

```
Lemma test (a b : bool) (Ha : a) (Hb : b) : a /\ b.
    1 goal

      a, b : bool
      Ha : a
      Hb : b
      ============================
      a /\ b

apply/andP.
    1 focused goal
    (shelved: 1)

      a, b : bool
      Ha : a
      Hb : b
      ============================
      a && b
```

Conversely

```
Lemma test (a b : bool) : a /\ b -> a.
    1 goal

      a, b : bool
      ============================
```

(continues on next page)

(continued from previous page)

```
      a /\ b -> a

move/andP.
    1 goal

      a, b : bool
      ============================
      a && b -> a
```

The same tactics can also be used to perform the converse operation, changing a boolean conjunction into a logical one. The view mechanism guesses the direction of the transformation to be used i.e., the constructor of the reflect predicate which should be chosen.

General mechanism for interpreting goals and assumptions

Specializing assumptions

The SSReflect tactic:

```
move/(_ term1 … termn).
```

is equivalent to the tactic:

```
intro top; generalize (top term1 … termn); clear top.
```

where `top` is a fresh name for introducing the top assumption of the current goal.

Interpreting assumptions

The general form of an assumption view tactic is:

Variant: `move | case` **/** *`term`*

The term , called the *view lemma* can be:

- a (term coercible to a) function;
- a (possibly quantified) implication;
- a (possibly quantified) double implication;
- a (possibly quantified) instance of the reflect predicate (see section *Views and reflection*).

Let `top` be the top assumption in the goal.

There are three steps in the behavior of an assumption view tactic:

- It first introduces `top`.
- If the type of *`term`* is neither a double implication nor an instance of the reflect predicate, then the tactic automatically generalises a term of the form: `term term1 … termn` where the terms `term1 … termn` instantiate the possible quantified variables of `term` , in order for `(term term1 … termn top)` to be well typed.
- If the type of `term` is an equivalence, or an instance of the reflect predicate, it generalises a term of the form: `(termvh (term term1 … termn ))` where the term `termvh` inserted is called an *assumption interpretation view hint*.

- It finally clears top.

For a `case/term` tactic, the generalisation step is replaced by a case analysis step.

View hints are declared by the user (see section *Views and reflection*) and are stored in the Hint View database. The proof engine automatically detects from the shape of the top assumption `top` and of the view lemma `term` provided to the tactic the appropriate view hint in the database to be inserted.

If `term` is a double implication, then the view hint will be one of the defined view hints for implication. These hints are by default the ones present in the file `ssreflect.v`:

```
Lemma iffLR : forall P Q, (P <-> Q) -> P -> Q.
```

which transforms a double implication into the left-to-right one, or:

```
Lemma iffRL : forall P Q, (P <-> Q) -> Q -> P.
```

which produces the converse implication. In both cases, the two first Prop arguments are implicit.

If `term` is an instance of the `reflect` predicate, then `A` will be one of the defined view hints for the `reflect` predicate, which are by default the ones present in the file `ssrbool.v`. These hints are not only used for choosing the appropriate direction of the translation, but they also allow complex transformation, involving negations.

Example

```
Check introN.
    introN
         : forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b

Lemma test (a b : bool) (Ha : a) (Hb : b) : ~~ (a && b).
    1 goal

      a, b : bool
      Ha : a
      Hb : b
      ============================
      ~~ (a && b)

apply/andP.
    1 focused goal
    (shelved: 1)

      a, b : bool
      Ha : a
      Hb : b
      ============================
      ~ (a /\ b)
```

In fact this last script does not exactly use the hint `introN`, but the more general hint:

```
Check introNTF.
    introNTF
         : forall (P : Prop) (b c : bool),
           reflect P b -> (if c then ~ P else P) -> ~~ b = c
```

The lemma `introN` is an instantiation of `introNF` using `c := true`.

Note that views, being part of *i_pattern*, can be used to interpret assertions too. For example the following script asserts `a && b` but actually uses its propositional interpretation.

Example

```
Lemma test (a b : bool) (pab : b && a) : b.
    1 goal

      a, b : bool
      pab : b && a
      ============================
      b

have /andP [pa ->] : (a && b) by rewrite andbC.
    1 goal

      a, b : bool
      pab : b && a
      pa : a
      ============================
      true
```

Interpreting goals

A goal interpretation view tactic of the form:

Variant: `apply/`*`term`*

applied to a goal `top` is interpreted in the following way:

- If the type of `term` is not an instance of the `reflect` predicate, nor an equivalence, then the term `term` is applied to the current goal `top`, possibly inserting implicit arguments.
- If the type of `term` is an instance of the reflect predicate or an equivalence, then a *goal interpretation view hint* can possibly be inserted, which corresponds to the application of a term `(termvh (term _ … _))` to the current goal, possibly inserting implicit arguments.

Like assumption interpretation view hints, goal interpretation ones are user-defined lemmas stored (see section *Views and reflection*) in the `Hint View` database bridging the possible gap between the type of `term` and the type of the goal.

Interpreting equivalences

Equivalent boolean propositions are simply *equal* boolean terms. A special construction helps the user to prove boolean equalities by considering them as logical double implications (between their coerced versions), while performing at the same time logical operations on both sides.

The syntax of double views is:

Variant: `apply/`*`term`*`/`*`term`*

The first term is the view lemma applied to the left hand side of the equality, while the second term is the one applied to the right hand side.

In this context, the identity view can be used when no view has to be applied:

```
Lemma idP : reflect b1 b1.
```

Example

```
Lemma test (b1 b2 b3 : bool) : ~~ (b1 || b2) = b3.
    1 goal

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) = b3

apply/idP/idP.
    2 focused goals
    (shelved: 2)

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) -> b3

    goal 2 is:
     b3 -> ~~ (b1 || b2)
```

The same goal can be decomposed in several ways, and the user may choose the most convenient interpretation.

```
Lemma test (b1 b2 b3 : bool) : ~~ (b1 || b2) = b3.
    1 goal

      b1, b2, b3 : bool
      ============================
      ~~ (b1 || b2) = b3

apply/norP/idP.
    2 focused goals
    (shelved: 2)

      b1, b2, b3 : bool
      ============================
      ~~ b1 /\ ~~ b2 -> b3

    goal 2 is:
     b3 -> ~~ b1 /\ ~~ b2
```

Declaring new Hint Views

Command: **Hint View for move /** *ident* [| *natural*]?

Command: **Hint View for apply /** *ident* [| *natural*]?

This command can be used to extend the database of hints for the view mechanism.

As library `ssrbool.v` already declares a corpus of hints, this feature is probably useful only for users who define their own logical connectives.

The *ident* is the name of the lemma to be declared as a hint. If `move` is used as tactic, the hint is declared for assumption interpretation tactics, `apply` declares hints for goal interpretations. Goal interpretation view hints are declared for both simple views and left hand side views. The optional natural number is the number of implicit arguments to be considered for the declared hint view lemma.

Variant: `Hint View for apply//`*ident* [`|` *natural*] ?
This variant with a double slash `//`, declares hint views for right hand sides of double views.

See the files `ssreflect.v` and `ssrbool.v` for examples.

Multiple views

The hypotheses and the goal can be interpreted by applying multiple views in sequence. Both move and apply can be followed by an arbitrary number of `/term`. The main difference between the following two tactics

```
apply/v1/v2/v3.
apply/v1; apply/v2; apply/v3.
```

is that the former applies all the views to the principal goal. Applying a view with hypotheses generates new goals, and the second line would apply the view `v2` to all the goals generated by `apply/v1`.

Note that the NO-OP intro pattern `-` can be used to separate two views, making the two following examples equivalent:

```
move=> /v1; move=> /v2.
move=> /v1 - /v2.
```

The tactic `move` can be used together with the `in` tactical to pass a given hypothesis to a lemma.

Example

```
Variable P2Q : P -> Q.
    P2Q is declared

Variable Q2R : Q -> R.
    Q2R is declared


Lemma test (p : P) : True.
    1 goal

      P, Q, R : Prop
      P2Q : P -> Q
      Q2R : Q -> R
      p : P
      ============================
      True

move/P2Q/Q2R in p.
    1 goal

      P, Q, R : Prop
      P2Q : P -> Q
      Q2R : Q -> R
      p : R
      ============================
      True
```

If the list of views is of length two, `Hint Views` for interpreting equivalences are indeed taken into account, otherwise only single `Hint Views` are used.

SSReflect searching tool

Command:

Search *pattern*? (−? *string* | *pattern* | % *ident*?)* in (−? *qualid*+)?

Changed in version 8.12: This command is only available when loading a separate plugin (`ssrsearch`).

Deprecated since version 8.12: This command is deprecated since all the additional features it provides have been integrated in the standard *Search* command.

This is the SSReflect extension of the Search command. *qualid* is the name of an open module. This command returns the list of lemmas:

- whose *conclusion* contains a subterm matching the optional first pattern. A − reverses the test, producing the list of lemmas whose conclusion does not contain any subterm matching the pattern;
- whose name contains the given string. A − prefix reverses the test, producing the list of lemmas whose name does not contain the string. A string that contains symbols or is followed by a scope key, is interpreted as the constant whose notation involves that string (e.g., + for `addn`), if this is unambiguous; otherwise the diagnostic includes the output of the *Locate* command.
- whose statement, including assumptions and types, contains a subterm matching the next patterns. If a pattern is prefixed by −, the test is reversed;
- contained in the given list of modules, except the ones in the modules prefixed by a −.

Note:

- As for regular terms, patterns can feature scope indications. For instance, the command: `Search _ (_ + _) %N.` lists all the lemmas whose statement (conclusion or hypotheses) involves an application of the binary operation denoted by the infix + symbol in the `N` scope (which is SSReflect scope for natural numbers).
- Patterns with holes should be surrounded by parentheses.
- Search always volunteers the expansion of the notation, avoiding the need to execute Locate independently. Moreover, a string fragment looks for any notation that contains fragment as a substring. If the `ssrbool.v` library is imported, the command: `Search "~~".` answers :

```
Require Import ssrsearch.

Search "~~".
    Toplevel input, characters 0-12:
    > Search "~~".
    > ^^^^^^^^^^^^
    Warning: SSReflect's Search command is deprecated.
    [deprecated-ssr-search,deprecated]
    "~~" is part of notation ("~~ _")
    In bool_scope, ("~~ b") denotes negb b
    Toplevel input, characters 0-12:
    > Search "~~".
    > ^^^^^^^^^^^^
    Warning: Listing only lemmas with conclusion matching (~~ ?b)
    negbT: forall b : bool, b = false -> ~~ b

    contra: forall c b : bool, (c -> b) -> ~~ b -> ~~ c

    contraNN: forall c b : bool, (c -> b) -> ~~ b -> ~~ c
```

(continues on next page)

(continued from previous page)

```
contraL: forall c b : bool, (c -> ~~ b) -> b -> ~~ c

contraTN: forall c b : bool, (c -> ~~ b) -> b -> ~~ c

contraFN: forall c b : bool, (c -> b) -> b = false -> ~~ c

contra_notN: forall (P : Prop) (b : bool), (b -> P) -> ~ P -> ~~ b

contraPN: forall (P : Prop) (b : bool), (b -> ~ P) -> P -> ~~ b

introN: forall (P : Prop) (b : bool), reflect P b -> ~ P -> ~~ b
```

- A diagnostic is issued if there are different matching notations; it is an error if all matches are partial.
- Similarly, a diagnostic warns about multiple interpretations, and signals an error if there is no default one.
- The command `Search in M.` is a way of obtaining the complete signature of the module `M`.
- Strings and pattern indications can be interleaved, but the first indication has a special status if it is a pattern, and only filters the conclusion of lemmas:
  - The command : `Search (_ =1 _) "bij".` lists all the lemmas whose conclusion features a `=1` and whose name contains the string `bij`.
  - The command : `Search "bij" (_ =1 _).` lists all the lemmas whose statement, including hypotheses, features a `=1` and whose name contains the string `bij`.

Synopsis and Index

Parameters

SSReflect tactics

d_tactic ::= **elim** | **case** | **congr** | **apply** | **exact** | **move**

Notation scope

key ::= *ident*

Module name

modname ::= *qualid*

Natural number

nat_or_ident ::= *natural* | *ident*

where *ident* is an Ltac variable denoting a standard Coq number (should not be the name of a tactic which can be followed by a bracket `[`, like `do`, `have`,...)

Items and switches

ssr_binder : : = *ident* | (*ident* [: *term*]?)

binder see *Abbreviations*.

clear_switch : : = { *ident*+ }

clear switch see *Discharge*

c_pattern : : = *term* **in** | *term* [**as** *ident*]? **in** *term*

context pattern see *Contextual patterns*

d_item : : = *occ_switch* | *clear_switch*? | *term* | (*c_pattern*)?

discharge item see *Discharge*

gen_item : : = [@? *ident*] | (*ident*) | ([@? *ident*] **:=** *c_pattern*)

generalization item see *Structure*

i_pattern : : = *ident* | **>** | **_** | **?** | ***** | **+** | *occ_switch*? **->** | **<-** | [*i_item*? |] | **-** | [: *ident*+]

intro pattern *Introduction in the context*

i_item : : = *clear_switch* | *s_item* | *i_pattern* | *i_view* | *i_block*

view *Introduction in the context*

i_view : : = {}? | */term* | **/ltac:(** *tactic*)

intro block *Introduction in the context*

i_block : : = [^ *ident*] | [^~ *ident* | *natural*]

intro item see *Introduction in the context*

int_mult : : = *natural*? *mult_mark*

multiplier see *Iteration*

occ_switch : : = { **+** | **-**? *natural** }

occur. switch see *Occurrence selection*

mult : : = *natural*? *mult_mark*

multiplier see *Iteration*

mult_mark ::= ? | !

multiplier mark see *Iteration*

r_item ::= / ? *term* | *s_item*

rewrite item see *Rewriting*

r_prefix ::= - ? *int_mult* ? *occ_switch* | *clear_switch* ? [*r_pattern*] ?

rewrite prefix see *Rewriting*

r_pattern ::= *term* | *c_pattern* | in *ident* ? in *term*

rewrite pattern see *Rewriting*

r_step ::= *r_prefix* ? *r_item*

rewrite step see *Rewriting*

s_item ::= /= | // | //=

simplify switch see *Introduction in the context*

Tactics

Note: `without loss` and `suffices` are synonyms for `wlog` and `suff` respectively.

Tactic: `move`
`idtac` or `hnf` (see *Bookkeeping*)

Tactic: `apply`
Tactic: `exact`
application (see *The defective tactics*)

Variant: `abstract` : *d_item* +
see *The abstract tactic* and *Generating let in context entries with have*

Variant: `elim`
induction (see *The defective tactics*)

Variant: `case`
case analysis (see *The defective tactics*)

Variant: `rewrite` *r_step* +
rewrite (see *Rewriting*)

Tactic: `under` *r_prefix* ? *term* => *i_item* + ? do *tactic* | [*tactic* * |] ?
under (see *Rewriting under binders*)

Tactic: `over`
over (see *The over tactic*)

Tactic: `have` *i_item*`*` *i_pattern*`?` (*s_item* | *ssr_binder*`+`)`?` `: term`? `:=` *term*

Tactic: `have` *i_item*`*` *i_pattern*`?` (*s_item* | *ssr_binder*`+`)`?` `: term` `by` *tactic*`?`

Tactic: `have suff` *clear_switch*`?` *i_pattern*`?` `: term`? `:=` *term*

Tactic: `have suff` *clear_switch*`?` *i_pattern*`?` `: term` `by` *tactic*`?`

Tactic: `gen have` *ident* `,`? *i_pattern*`?` `:` *gen_item*`+` `/` *term* `by` *tactic*`?`

Tactic: `generally have` *ident* `,`? *i_pattern*`?` `:` *gen_item*`+` `/` *term* `by` *tactic*`?`
forward chaining (see *Structure*)

Tactic: `wlog` `suff`? *i_item*`?` `:` *gen_item* | *clear_switch*`*` `/` *term*
specializing (see *Structure*)

Tactic: `suff` *i_item*`*` *i_pattern*`?` *ssr_binder*`+` `: term` `by` *tactic*`?`

Tactic: `suffices` *i_item*`*` *i_pattern*`?` *ssr_binder*`+` `: term` `by` *tactic*`?`

Tactic: `suff` `have`? *clear_switch*`?` *i_pattern*`?` `: term` `by` *tactic*`?`

Tactic: `suffices` `have`? *clear_switch*`?` *i_pattern*`?` `: term` `by` *tactic*`?`
backchaining (see *Structure*)

Variant: `pose` *ident* `:=` *term*
local definition (see *Definitions*)

Variant: `pose` *ident* *ssr_binder*`+` `:=` *term*
local function definition

Variant: `pose fix` *fix_decl*
local fix definition

Variant: `pose cofix` *fix_decl*
local cofix definition

Tactic: `set` *ident* `: term`? `:=` *occ_switch*`?` *term* | `(` *c_pattern*`)`
abbreviation (see *Abbreviations*)

Tactic: `unlock` (*r_prefix*`?` *ident*)`*`
unlock (see *Locking, unlocking*)

Tactic: `congr` *natural*`?` *term*
congruence (see *Congruence*)

Tacticals

tactic += *d_tactic* *ident*? : *d_item*+ *clear_switch*?

discharge *Discharge*

tactic += *tactic* => *i_item*+

introduction see *Introduction in the context*

tactic += *tactic* **in** *gen_item* | *clear_switch*+ *?

localization see *Localization*

tactic += **do** *mult*? *tactic* | [*tactic*+|]

iteration see *Iteration*

tactic += *tactic* ; **first** | **last** *natural*? *tactic* | [*tactic*+|]

selector see *Selectors*

tactic += *tactic* ; **first** | **last** *natural*?

rotation see *Selectors*

tactic += **by** *tactic* | [*tactic**|]

closing see *Terminators*

Commands

Command: `Hint View for` `move` | `apply` `/` *ident* | *natural*?
view hint declaration (see *Declaring new Hint Views*)

Command: `Hint View for apply //` *ident* *natural*?
right hand side double , view hint declaration (see *Declaring new Hint Views*)

Command: `Prenex Implicits` *ident*+
prenex implicits declaration (see *Parametric polymorphism*)

Settings

Flag: `Debug Ssreflect`
Developer only. Print debug information on reflect.

Flag: `Debug SsrMatching`
Developer only. Print debug information on SSR matching.

3.2 Automatic solvers and programmable tactics

Some tactics are largely automated and are able to solve complex goals. This chapter presents both built-in solvers that can be used on specific categories of goals and programmable tactics that the user can instrument to handle complex goals in new domains.

3.2.1 Solvers for logic and equality

Tactic: `tauto`
This tactic implements a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhoff [Dyc92]. Note that *tauto* succeeds on any instance of an intuitionistic tautological proposition. *tauto* unfolds negations and logical equivalence but does not unfold any other definition.

Example

The following goal can be proved by *tauto* whereas *auto* would fail:

```
Goal forall (x:nat) (P:nat -> Prop), x = 0 \/ P x -> x <> 0 -> P x.
    1 goal

      ============================
      forall (x : nat) (P : nat -> Prop), x = 0 \/ P x -> x <> 0 -> P x

intros.
    1 goal

      x : nat
      P : nat -> Prop
      H : x = 0 \/ P x
      H0 : x <> 0
      ============================
      P x

tauto.
    No more goals.
```

Moreover, if it has nothing else to do, *tauto* performs introductions. Therefore, the use of *intros* in the previous proof is unnecessary. *tauto* can for instance for:

Example

```
Goal forall (A:Prop) (P:nat -> Prop), A \/ (forall x:nat, ~ A -> P x) -> forall␣
↪x:nat, ~ A -> P x.
    1 goal

      ============================
      forall (A : Prop) (P : nat -> Prop),
      A \/ (forall x : nat, ~ A -> P x) -> forall x : nat, ~ A -> P x

tauto.
    No more goals.
```

Note: In contrast, *tauto* cannot solve the following goal `Goal forall (A:Prop) (P:nat -> Prop), A \/ (forall x:nat, ~ A -> P x) -> forall x:nat, ~ ~ (A \/ P x).` because `(forall x:nat, ~ A -> P x)` cannot be treated as atomic and an instantiation of `x` is necessary.

Tactic: dtauto

While *tauto* recognizes inductively defined connectives isomorphic to the standard connectives `and`, `prod`, `or`, `sum`, `False`, `Empty_set`, `unit` and `True`, *dtauto* also recognizes all inductive types with one constructor and no indices, i.e. record-style connectives.

Tactic: intuition *ltac_expr*?

Uses the search tree built by the decision procedure for *tauto* to generate a set of subgoals equivalent to the original one (but simpler than it) and applies *ltac_expr* to them [Mun94]. If *ltac_expr* is not specified, it defaults to **auto with *** If *ltac_expr* fails on some goals then *intuition* fails. In fact, *tauto* is simply `intuition fail`.

intuition recognizes inductively defined connectives isomorphic to the standard connectives `and`, `prod`, `or`, `sum`, `False`, `Empty_set`, `unit` and `True`.

Example

For instance, the tactic `intuition auto` applied to the goal:

```
(forall (x:nat), P x) /\ B -> (forall (y:nat), P y) /\ P O \/ B /\ P O
```

internally replaces it by the equivalent one:

```
(forall (x:nat), P x), B |- P O
```

and then uses *auto* which completes the proof.

Tactic: dintuition *ltac_expr*?

In addition to the inductively defined connectives recognized by *intuition*, *dintuition* also recognizes all inductive types with one constructor and no indices, i.e. record-style connectives.

Flag: Intuition Negation Unfolding

This *flag* controls whether *intuition* unfolds inner negations which do not need to be unfolded. It is on by default.

Tactic: rtauto

Solves propositional tautologies similarly to *tauto*, but the proof term is built using a reflection scheme applied to a sequent calculus proof of the goal. The search procedure is also implemented using a different technique.

Users should be aware that this difference may result in faster proof search but slower proof checking, and *rtauto* might not solve goals that *tauto* would be able to solve (e.g. goals involving universal quantifiers).

Note that this tactic is only available after a `Require Import Rtauto.`

Tactic: firstorder *ltac_expr*? **using** *qualid*+, ? **with** *ident*+ ?

An experimental extension of *tauto* to first-order reasoning. It is not restricted to usual logical connectives but instead can reason about any first-order class inductive definition.

ltac_expr Tries to solve the goal with *ltac_expr* when no logical rule applies. If unspecified, the tactic uses the default from the *Firstorder Solver* option.

using *qualid*+, Adds the lemmas *qualid*+, to the proof search environment. If *qualid* refers to an inductive type, its constructors are added to the proof search environment.

with *ident*+ Adds lemmas from *auto* hint bases *ident*+ to the proof search environment.

Option: Firstorder Solver *ltac_expr*
The default tactic used by *firstorder* when no rule applies in `auto with core`. It can be set locally or globally using this *option*.

Command: Print Firstorder Solver
Prints the default tactic used by *firstorder* when no rule applies.

Option: Firstorder Depth *natural*
This *option* controls the proof search depth bound.

Tactic: congruence *natural*? **with** *one_term*+ ?

natural Specifies the maximum number of hypotheses stating quantified equalities that may be added to the problem in order to solve it. The default is 1000.

with *one_term*+ ? Adds *one_term*+ to the pool of terms used by *congruence*. This helps in case you have partially applied constructors in your goal.

Implements the standard Nelson and Oppen congruence closure algorithm, which is a decision procedure for ground equalities with uninterpreted symbols. It also includes constructor theory (see *injection* and *discriminate*). If the goal is a non-quantified equality, congruence tries to prove it with non-quantified equalities in the context. Otherwise it tries to infer a discriminable equality from those in the context. Alternatively, congruence tries to prove that a hypothesis is equal to the goal or to the negation of another hypothesis.

congruence is also able to take advantage of hypotheses stating quantified equalities, but you have to provide a bound for the number of extra equalities generated that way. Please note that one of the sides of the equality must contain all the quantified variables in order for congruence to match against it.

Increasing the maximum number of hypotheses may solve problems that would have failed with a smaller value. It will make failures slower but it won't make successes found with the smaller value any slower. You may want to use *assert* to add some lemmas as hypotheses so that *congruence* can use them.

Example

```
Theorem T (A:Type) (f:A -> A) (g: A -> A -> A) a b: a=(f a) -> (g b (f a))=(f (f␣
↪a)) -> (g a b)=(f (g b a)) -> (g a b)=a.
    1 goal
```

(continues on next page)

(continued from previous page)

```
  A : Type
  f : A -> A
  g : A -> A -> A
  a, b : A
  ============================
  a = f a -> g b (f a) = f (f a) -> g a b = f (g b a) -> g a b = a

intros.
  1 goal

  A : Type
  f : A -> A
  g : A -> A -> A
  a, b : A
  H : a = f a
  H0 : g b (f a) = f (f a)
  H1 : g a b = f (g b a)
  ============================
  g a b = a

congruence.
  No more goals.

Qed.

Theorem inj (A:Type) (f:A -> A * A) (a c d: A) : f = pair a -> Some (f c) = Some␣
↪(f d) -> c=d.
  1 goal

  A : Type
  f : A -> A * A
  a, c, d : A
  ============================
  f = pair a -> Some (f c) = Some (f d) -> c = d

intros.
  1 goal

  A : Type
  f : A -> A * A
  a, c, d : A
  H : f = pair a
  H0 : Some (f c) = Some (f d)
  ============================
  c = d

congruence.
  No more goals.

Qed.
```

Error: `I don't know how to handle dependent equality.`

The decision procedure managed to find a proof of the goal or of a discriminable equality but this proof could not be built in Coq because of dependently-typed functions.

Error: `Goal is solvable by congruence but some arguments are missing.`

`Try congruence with` *`term`*`+, replacing metavariables by arbitrary terms.`
The decision procedure could solve the goal with the provision that additional arguments are supplied for some partially applied constructors. Any term of an appropriate type will allow the tactic to successfully solve the goal. Those additional arguments can be given to congruence by filling in the holes in the terms given in the error message, using the `with` clause.

`Debug` `"congruence"` makes *`congruence`* print debug information.

Tactic: `btauto`
The tactic *`btauto`* implements a reflexive solver for boolean tautologies. It solves goals of the form `t = u` where `t` and `u` are constructed over the following grammar:

btauto_term	`::=`	*ident*
	|	**true**
	|	**false**
	|	**orb** *btauto_term btauto_term*
	|	**andb** *btauto_term btauto_term*
	|	**xorb** *btauto_term btauto_term*
	|	**negb** *btauto_term*
	|	**if** *btauto_term* **then** *btauto_term* **else** *btauto_term*

Whenever the formula supplied is not a tautology, it also provides a counter-example.

Internally, it uses a system very similar to the one of the ring tactic.

Note that this tactic is only available after a `Require Import Btauto.`

Error: `Cannot recognize a boolean equality.`
The goal is not of the form `t = u`. Especially note that *`btauto`* doesn't introduce variables into the context on its own.

3.2.2 Micromega: solvers for arithmetic goals over ordered rings

Authors Frédéric Besson and Evgeny Makarov

Short description of the tactics

The Psatz module (`Require Import Psatz.`) gives access to several tactics for solving arithmetic goals over $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{Z}$ but also `nat` and `N`. It also possible to get the tactics for integers by a `Require Import Lia`, rationals `Require Import Lqa` and reals `Require Import Lra`.

- *`lia`* is a decision procedure for linear integer arithmetic;
- *`nia`* is an incomplete proof procedure for integer non-linear arithmetic;
- *`lra`* is a decision procedure for linear (real or rational) arithmetic;
- *`nra`* is an incomplete proof procedure for non-linear (real or rational) arithmetic;
- *`psatz`* `D n` where `D` is $\mathbb{Z}$ or $\mathbb{Q}$ or $\mathbb{R}$, and `n` is an optional integer limiting the proof search depth, is an incomplete proof procedure for non-linear arithmetic. It is based on John Harrison's HOL Light driver to the external prover `csdp`[43]. Note that the `csdp` driver generates a *proof cache* which makes it possible to rerun scripts even without `csdp`.

[43] Sources and binaries can be found at https://projects.coin-or.org/Csdp

Flag: `Simplex`
Deprecated since version 8.14.

This *flag* (set by default) instructs the decision procedures to use the Simplex method for solving linear goals instead of the deprecated Fourier elimination.

Option: `Dump Arith`
This *option* (unset by default) may be set to a file path where debug info will be written.

Command: `Show Lia Profile`
This command prints some statistics about the amount of pivoting operations needed by `lia` and may be useful to detect inefficiencies (only meaningful if flag `Simplex` is set).

Flag: `Lia Cache`
This *flag* (set by default) instructs `lia` to cache its results in the file `.lia.cache`

Flag: `Nia Cache`
This *flag* (set by default) instructs `nia` to cache its results in the file `.nia.cache`

Flag: `Nra Cache`
This *flag* (set by default) instructs `nra` to cache its results in the file `.nra.cache`

The tactics solve propositional formulas parameterized by atomic arithmetic expressions interpreted over a domain $D \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. The syntax for formulas over $\mathbb{Z}$ is:

F	::=	*A*	**P**	**True**	**False**	*F* /\\ *F*	*F* \\/ *F*	*F* <-> *F*	*F* -> *F*	~ *F*	*F* = *F*
A	::=	*p* = *p*	*p* > *p*	*p* < *p*	*p* >= *p*	*p* <= *p*					
p	::=	**c**	**x**	−*p*	*p* − *p*	*p* + *p*	*p* * *p*	*p* ^ **n**			

where

- `F` is interpreted over either `Prop` or `bool`
- **`P`** is an arbitrary proposition
- **`c`** is a numeric constant of D
- **`x`** $\in D$ is a numeric variable
- **`−`**, **`+`** and **`*`** are respectively subtraction, addition and product
- **`p ^ n`** is exponentiation by a constant n

When F is interpreted over `bool`, the boolean operators are `&&`, `||`, `Bool.eqb`, `Bool.implb`, `Bool.negb` and the comparisons in A are also interpreted over the booleans (e.g., for $\mathbb{Z}$, we have `Z.eqb`, `Z.gtb`, `Z.ltb`, `Z.geb`, `Z.leb`).

For $\mathbb{Q}$, use the equality of rationals == rather than Leibniz equality =.

For $\mathbb{Z}$ (resp. $\mathbb{Q}$), c ranges over integer constants (resp. rational constants). For $\mathbb{R}$, the tactic recognizes as real constants the following expressions:

```
c ::= R0 | R1 | Rmul(c,c) | Rplus(c,c) | Rminus(c,c) | IZR z | IQR q | Rdiv(c,c) |␣
↪Rinv c
```

where z is a constant in $\mathbb{Z}$ and q is a constant in $\mathbb{Q}$. This includes integer constants written using the decimal notation, *i.e.*, `c%R`.

Positivstellensatz refutations

The name `psatz` is an abbreviation for *positivstellensatz* – literally "positivity theorem" – which generalizes Hilbert's *nullstellensatz*. It relies on the notion of Cone. Given a (finite) set of polynomials S, *Cone*(S) is inductively defined as the smallest set of polynomials closed under the following rules:

$$\frac{p \in S}{p \in \mathit{Cone}(S)} \quad \frac{}{p^2 \in \mathit{Cone}(S)} \quad \frac{p_1 \in \mathit{Cone}(S) \quad p_2 \in \mathit{Cone}(S) \qquad \in \{+, *\}}{p_1 \;\; p_2 \in \mathit{Cone}(S)}$$

The following theorem provides a proof principle for checking that a set of polynomial inequalities does not have solutions[44].

Theorem (Psatz). Let S be a set of polynomials. If -1 belongs to *Cone*(S), then the conjunction $\bigwedge_{p \in S} p \geq 0$ is unsatisfiable. A proof based on this theorem is called a *positivstellensatz* refutation. The tactics work as follows. Formulas are normalized into conjunctive normal form $\bigwedge_i C_i$ where C_i has the general form $(\bigwedge_{j \in S_i} p_j \;\; 0) \rightarrow \mathit{False}$ and $\in \{>, \geq, =\}$ for $D \in \{\mathbb{Q}, \mathbb{R}\}$ and $\in \{\geq, =\}$ for $\mathbb{Z}$.

For each conjunct C_i, the tactic calls an oracle which searches for -1 within the cone. Upon success, the oracle returns a *cone expression* that is normalized by the `ring` tactic (see *ring and field: solvers for polynomial and rational equations*) and checked to be -1.

`lra`: a decision procedure for linear real and rational arithmetic

Tactic: `lra`

This tactic is searching for *linear* refutations. As a result, this tactic explores a subset of the *Cone* defined as

$$\mathit{LinCone}(S) = \left\{ \sum_{p \in S} \alpha_p \times p \;\middle|\; \alpha_p \text{ are positive constants} \right\}$$

The deductive power of `lra` overlaps with the one of `field` tactic *e.g.*, $x = 10 * x/10$ is solved by `lra`.

`lia`: a tactic for linear integer arithmetic

Tactic: `lia`

This tactic solves linear goals over `Z` by searching for *linear* refutations and cutting planes. `lia` provides support for `Z`, `nat`, `positive` and `N` by pre-processing via the `zify` tactic.

High level view of `lia`

Over $\mathbb{R}$, *positivstellensatz* refutations are a complete proof principle[45]. However, this is not the case over $\mathbb{Z}$. Actually, *positivstellensatz* refutations are not even sufficient to decide linear *integer* arithmetic. The canonical example is $2 * x = 1 \rightarrow$ `False` which is a theorem of $\mathbb{Z}$ but not a theorem of $\mathbb{R}$. To remedy this weakness, the `lia` tactic is using recursively a combination of:

- linear *positivstellensatz* refutations;
- cutting plane proofs;
- case split.

[44] Variants deal with equalities and strict inequalities.
[45] In practice, the oracle might fail to produce such a refutation.

Cutting plane proofs

are a way to take into account the discreteness of $\mathbb{Z}$ by rounding up (rational) constants up-to the closest integer.

Theorem: `Bound on the ceiling function`
Let p be an integer and c a rational constant. Then $p \geq c \rightarrow p \geq \lceil c \rceil$.

For instance, from $2x = 1$ we can deduce

- $x \geq 1/2$ whose cut plane is $x \geq \lceil 1/2 \rceil = 1$;
- $x \leq 1/2$ whose cut plane is $x \leq \lfloor 1/2 \rfloor = 0$.

By combining these two facts (in normal form) $x - 1 \geq 0$ and $-x \geq 0$, we conclude by exhibiting a *positivstellensatz* refutation: $-1 \equiv x - 1 + -x \in \mathit{Cone}(x - 1, x)$.

Cutting plane proofs and linear *positivstellensatz* refutations are a complete proof principle for integer linear arithmetic.

Case split

enumerates over the possible values of an expression.

Theorem. Let p be an integer and c_1 and c_2 integer constants. Then:

$$c_1 \leq p \leq c_2 \Rightarrow \bigvee_{x \in [c_1, c_2]} p = x$$

Our current oracle tries to find an expression e with a small range $[c_1, c_2]$. We generate $c_2 - c_1$ subgoals which contexts are enriched with an equation $e = i$ for $i \in [c_1, c_2]$ and recursively search for a proof.

`nra`: a proof procedure for non-linear arithmetic

Tactic: `nra`
This tactic is an *experimental* proof procedure for non-linear arithmetic. The tactic performs a limited amount of non-linear reasoning before running the linear prover of `lra`. This pre-processing does the following:

- If the context contains an arithmetic expression of the form $e[x^2]$ where x is a monomial, the context is enriched with $x^2 \geq 0$;
- For all pairs of hypotheses $e_1 \geq 0$, $e_2 \geq 0$, the context is enriched with $e_1 \times e_2 \geq 0$.

After this pre-processing, the linear prover of `lra` searches for a proof by abstracting monomials by variables.

`nia`: a proof procedure for non-linear integer arithmetic

Tactic: `nia`
This tactic is a proof procedure for non-linear integer arithmetic. It performs a pre-processing similar to `nra`. The obtained goal is solved using the linear integer prover `lia`.

`psatz`: a proof procedure for non-linear arithmetic

Tactic: `psatz` *one_term* *nat_or_var* ?

This tactic explores the *Cone* by increasing degrees – hence the depth parameter *nat_or_var*. In theory, such a proof search is complete – if the goal is provable the search eventually stops. Unfortunately, the external oracle is using numeric (approximate) optimization techniques that might miss a refutation.

To illustrate the working of the tactic, consider we wish to prove the following Coq goal:

```
Require Import ZArith Psatz.
Open Scope Z_scope.
Goal forall x, -x^2 >= 0 -> x - 1 >= 0 -> False.
intro x.
psatz Z 2.
```

As shown, such a goal is solved by `intro x. psatz Z 2.`. The oracle returns the cone expression $2 \times (x - 1) + (\mathbf{x - 1}) \times (\mathbf{x - 1}) + -x^2$ (polynomial hypotheses are printed in bold). By construction, this expression belongs to $Cone(-x^2, x - 1)$. Moreover, by running *ring* we obtain -1. By Theorem *Psatz*, the goal is valid.

`zify`: pre-processing of arithmetic goals

Tactic: `zify`

This tactic is internally called by *lia* to support additional types, e.g., `nat`, `positive` and `N`. Additional support is provided by the following modules:

- For boolean operators (e.g., `Nat.leb`), require the module `ZifyBool`.
- For comparison operators (e.g., `Z.compare`), require the module `ZifyComparison`.
- For native unsigned 63 bit integers, require the module `ZifyUint63`.
- For native signed 63 bit integers, require the module `ZifySint63`.
- For operators `Nat.div`, `Nat.mod`, and `Nat.pow`, require the module `ZifyNat`.
- For operators `N.div`, `N.mod`, and `N.pow`, require the module `ZifyN`.

zify can also be extended by rebinding the tactics `Zify.zify_pre_hook` and `Zify.zify_post_hook` that are respectively run in the first and the last steps of *zify*.

- To support `Z.div` and `Z.modulo`: `Ltac Zify.zify_post_hook ::= Z.div_mod_to_equations`.
- To support `Z.quot` and `Z.rem`: `Ltac Zify.zify_post_hook ::= Z.quot_rem_to_equations`.
- To support `Z.div`, `Z.modulo`, `Z.quot` and `Z.rem`: either `Ltac Zify.zify_post_hook ::= Z.to_euclidean_division_equations` or `Ltac Zify.zify_convert_to_euclidean_division_equations_flag ::= constr:(true)`.

The *zify* tactic can be extended with new types and operators by declaring and registering new typeclass instances using the following commands. The typeclass declarations can be found in the module `ZifyClasses` and the default instances can be found in the module `ZifyInst`.

Command: `Add Zify` *add_zify qualid*

add_zify	::=	**InjTyp**	**BinOp**	**UnOp**	**CstOp**	**BinRel**	**UnOpSpec**	**BinOpSpec**
	|	**PropOp**	**PropBinOp**	**PropUOp**	**Saturate**			

Registers an instance of the specified typeclass. The typeclass type (e.g. `BinOp Z.mul` or `BinRel (@eq Z)`) has the additional constraint that the non-implicit argument (here, `Z.mul` or `(@eq Z)`) is either a ***reference*** (here, `Z.mul`) or the application of a ***reference*** (here, `@eq`) to a sequence of ***one_term***.

Command: **`Show Zify`** ***show_zify***

show_zify	::=	**InjTyp**	**BinOp**	**UnOp**	**CstOp**	**BinRel**	**UnOpSpec**	**BinOpSpec**	**Spec**

Prints instances for the specified typeclass. For instance, *Show Zify* `InjTyp` prints the list of types that supported by *zify* i.e., `Z`, `nat`, `positive` and `N`.

3.2.3 ring and field: solvers for polynomial and rational equations

Author Bruno Barras, Benjamin Grégoire, Assia Mahboubi, Laurent Théry[46]

This chapter presents the tactics dedicated to dealing with ring and field equations.

What does this tactic do?

`ring` does associative-commutative rewriting in ring and semiring structures. Assume you have two binary functions $\oplus$ and $\otimes$ that are associative and commutative, with $\oplus$ distributive on $\otimes$, and two constants 0 and 1 that are unities for $\oplus$ and $\otimes$. A polynomial is an expression built on variables V_0, V_1, ... and constants by application of $\oplus$ and $\otimes$.

Let an ordered product be a product of variables $V_{i_1} \otimes \cdots \otimes V_{i_n}$ verifying $i_1 \leq i_2 \leq \cdots \leq i_n$. Let a monomial be the product of a constant and an ordered product. We can order the monomials by the lexicographic order on products of variables. Let a canonical sum be an ordered sum of monomials that are all different, i.e. each monomial in the sum is strictly less than the following monomial according to the lexicographic order. It is an easy theorem to show that every polynomial is equivalent (modulo the ring properties) to exactly one canonical sum. This canonical sum is called the normal form of the polynomial. In fact, the actual representation shares monomials with same prefixes. So what does the `ring` tactic do? It normalizes polynomials over any ring or semiring structure. The basic use of `ring` is to simplify ring expressions, so that the user does not have to deal manually with the theorems of associativity and commutativity.

Example

In the ring of integers, the normal form of

$$x(3 + yx + 25(1 - z)) + zx$$

is

$$28x + (-24)xz + xxy.$$

`ring` is also able to compute a normal form modulo monomial equalities. For example, under the hypothesis that $2x^2 = yz + 1$, the normal form of $2(x + 1)x - x - zy$ is $x + 1$.

[46] based on previous work from Patrick Loiseleur and Samuel Boutin

The variables map

It is frequent to have an expression built with $+$ and $\times$, but rarely on variables only. Let us associate a number to each subterm of a ring expression in the Gallina language. For example, consider this expression in the semiring `nat`:

```
(plus (mult (plus (f (5)) x) x)
      (mult (if b then (4) else (f (3))) (2)))
```

As a ring expression, it has 3 subterms. Give each subterm a number in an arbitrary order:

0	$\mapsto$	if b then (4) else (f (3))
1	$\mapsto$	(f (5))
2	$\mapsto$	x

Then normalize the "abstract" polynomial $((V_1 \oplus V_2) \otimes V_2) \oplus (V_0 \otimes 2)$ In our example the normal form is: $(2 \otimes V_0) \oplus (V_1 \otimes V_2) \oplus (V_2 \otimes V_2)$. Then substitute the variables by their values in the variables map to get the concrete normal polynomial:

```
(plus (mult (2) (if b then (4) else (f (3))))
      (plus (mult (f (5)) x) (mult x x)))
```

Is it automatic?

Yes, building the variables map and doing the substitution after normalizing is automatically done by the tactic. So you can just forget this paragraph and use the tactic according to your intuition.

Concrete usage in Coq

Tactic: `ring` [*one_term*+]?

Solves polynomical equations of a ring (or semiring) structure. It proceeds by normalizing both sides of the equation (w.r.t. associativity, commutativity and distributivity, constant propagation, rewriting of monomials) and syntactically comparing the results.

[*one_term*+] If specified, the tactic decides the equality of two terms modulo ring operations and the equalities defined by the *one_term*s. Each *one_term* has to be a proof of some equality `m = p`, where `m` is a monomial (after "abstraction"), `p` a polynomial and = is the corresponding equality of the ring structure.

Tactic: `ring_simplify` [*one_term*+]? *one_term*+ **in** *ident*?

Applies the normalization procedure described above to the given *one_term*s. The tactic then replaces all occurrences of the *one_term*s given in the conclusion of the goal by their normal forms. If no *one_term* is given, then the conclusion should be an equation and both sides are normalized. The tactic can also be applied in a hypothesis.

in *ident* If specified, the tactic performs the simplification in the hypothesis named *ident*.

Note: **ring_simplify** one_term_1**;** **ring_simplify** one_term_2 is not equivalent to **ring_simplify** one_term_1 one_term_2.

In the latter case the variables map is shared between the two *one_term*s, and common subterm `t` of ***one_term*** $_1$ and ***one_term*** $_2$ will have the same associated variable number. So the first alternative should be avoided for *one_term*s belonging to the same ring theory.

The tactic must be loaded by `Require Import Ring`. The ring structures must be declared with the `Add Ring` command (see below). The ring of booleans is predefined; if one wants to use the tactic on `nat` one must first require the module `ArithRing` exported by `Arith`); for `Z`, do `Require Import ZArithRing` or simply `Require Import ZArith`; for `N`, do `Require Import NArithRing` or `Require Import NArith`.

All declared field structures can be printed with the *Print Rings* command.

Command: `Print Rings`

Example

```
Require Import ZArith.
    [Loading ML file ring_plugin.cmxs ... done]
    [Loading ML file zify_plugin.cmxs ... done]
    [Loading ML file micromega_plugin.cmxs ... done]

Open Scope Z_scope.
Goal forall a b c:Z,
    (a + b + c) ^ 2 =
     a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c.
    1 goal

      ============================
      forall a b c : Z,
      (a + b + c) ^ 2 = a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c

intros; ring.
    No more goals.

Abort.
Goal forall a b:Z,
     2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30.
    1 goal

      ============================
      forall a b : Z, 2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30

intros a b H; ring [H].
    No more goals.

Abort.
```

Error messages:

Error: `Not a valid ring equation.`
The conclusion of the goal is not provable in the corresponding ring theory.

Error: `Arguments of ring_simplify do not have all the same type.`
ring_simplify cannot simplify terms of several rings at the same time. Invoke the tactic once per ring structure.

Error: `Cannot find a declared ring structure over` *term*.

No ring has been declared for the type of the terms to be simplified. Use *Add Ring* first.

Error: Cannot find a declared ring structure for equality *term*.
Same as above in the case of the *ring* tactic.

Adding a ring structure

Declaring a new ring consists in proving that a ring signature (a carrier set, an equality, and ring operations: `Ring_theory.ring_theory` and `Ring_theory.semi_ring_theory`) satisfies the ring axioms. Semi- rings (rings without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see *Tactics enabled on user provided relations*). The definitions of ring and semiring (see module `Ring_theory`) are:

```
Record ring_theory : Prop := mk_rt {
  Radd_0_l    : forall x, 0 + x == x;
  Radd_sym    : forall x y, x + y == y + x;
  Radd_assoc  : forall x y z, x + (y + z) == (x + y) + z;
  Rmul_1_l    : forall x, 1 * x == x;
  Rmul_sym    : forall x y, x * y == y * x;
  Rmul_assoc  : forall x y z, x * (y * z) == (x * y) * z;
  Rdistr_l    : forall x y z, (x + y) * z == (x * z) + (y * z);
  Rsub_def    : forall x y, x - y == x + -y;
  Ropp_def    : forall x, x + (- x) == 0
}.

Record semi_ring_theory : Prop := mk_srt {
  SRadd_0_l   : forall n, 0 + n == n;
  SRadd_sym   : forall n m, n + m == m + n ;
  SRadd_assoc : forall n m p, n + (m + p) == (n + m) + p;
  SRmul_1_l   : forall n, 1*n == n;
  SRmul_0_l   : forall n, 0*n == 0;
  SRmul_sym   : forall n m, n*m == m*n;
  SRmul_assoc : forall n m p, n*(m*p) == (n*m)*p;
  SRdistr_l   : forall n m p, (n + m)*p == n*p + m*p
}.
```

This implementation of `ring` also features a notion of constant that can be parameterized. This can be used to improve the handling of closed expressions when operations are effective. It consists in introducing a type of *coefficients* and an implementation of the ring operations, and a morphism from the coefficient type to the ring carrier type. The morphism needs not be injective, nor surjective.

As an example, one can consider the real numbers. The set of coefficients could be the rational numbers, upon which the ring operations can be implemented. The fact that there exists a morphism is defined by the following properties:

```
Record ring_morph : Prop := mkmorph {
  morph0    : [c0] == 0;
  morph1    : [cI] == 1;
  morph_add : forall x y, [x +! y] == [x]+[y];
  morph_sub : forall x y, [x -! y] == [x]-[y];
  morph_mul : forall x y, [x *! y] == [x]*[y];
  morph_opp : forall x, [-!x] == -[x];
  morph_eq  : forall x y, x?=!y = true -> [x] == [y]
}.

Record semi_morph : Prop := mkRmorph {
  Smorph0 : [c0] == 0;
  Smorph1 : [cI] == 1;
```

(continues on next page)

(continued from previous page)

```
  Smorph_add : forall x y, [x +! y] == [x]+[y];
  Smorph_mul : forall x y, [x *! y] == [x]*[y];
  Smorph_eq  : forall x y, x?=!y = true -> [x] == [y]
}.
```

where `c0` and `cI` denote the 0 and 1 of the coefficient set, `+!`, `*!`, `-!` are the implementations of the ring operations, `==` is the equality of the coefficients, `?+!` is an implementation of this equality, and `[x]` is a notation for the image of `x` by the ring morphism.

Since `Z` is an initial ring (and `N` is an initial semiring), it can always be considered as a set of coefficients. There are basically three kinds of (semi-)rings:

abstract rings to be used when operations are not effective. The set of coefficients is `Z` (or `N` for semirings).

computational rings to be used when operations are effective. The set of coefficients is the ring itself. The user only has to provide an implementation for the equality.

customized ring for other cases. The user has to provide the coefficient set and the morphism.

This implementation of ring can also recognize simple power expressions as ring expressions. A power function is specified by the following property:

```
Require Import Reals.
Section POWER.
  Variable Cpow : Set.
  Variable Cp_phi : N -> Cpow.
  Variable rpow : R -> Cpow -> R.

  Record power_theory : Prop := mkpow_th {
    rpow_pow_N : forall r n, rpow r (Cp_phi n) = pow_N 1%R Rmult r n
  }.

End POWER.
```

The syntax for adding a new ring is

Command: `Add Ring` *ident* : *one_term* (*ring_mod* , … , *ring_mod*)?

ring_mod ::=
- **decidable** *one_term*
- **abstract**
- **morphism** *one_term*
- **constants** [*ltac_expr*]
- **preprocess** [*ltac_expr*]
- **postprocess** [*ltac_expr*]
- **setoid** *one_term one_term*
- **sign** *one_term*
- **power** *one_term* [*qualid* … *qualid*]
- **power_tac** *one_term* [*ltac_expr*]
- **div** *one_term*
- **closed** [*qualid* … *qualid*]

The *ident* is used only for error messages. The *one_term* is a proof that the ring signature satisfies the (semi-)ring axioms. The optional list of modifiers is used to tailor the behavior of the tactic. Here are their effects:

`abstract` declares the ring as abstract. This is the default.

decidable *one_term* declares the ring as computational. The expression *one_term* is the correctness proof of an equality test `?=!` (which should be evaluable). Its type should be of the form `forall x y, x ?=! y = true → x == y`.

morphism *one_term* declares the ring as a customized one. The expression *one_term* is a proof that there exists a morphism between a set of coefficient and the ring carrier (see `Ring_theory.ring_morph` and `Ring_theory.semi_morph`).

setoid *one_term one_term* forces the use of given setoid. The first *one_term* is a proof that the equality is indeed a setoid (see `Setoid.Setoid_Theory`), and the second a proof that the ring operations are morphisms (see `Ring_theory.ring_eq_ext` and `Ring_theory.sring_eq_ext`). This modifier needs not be used if the setoid and morphisms have been declared.

constants [*ltac_expr* **]** specifies a tactic expression *ltac_expr* that, given a term, returns either an object of the coefficient set that is mapped to the expression via the morphism, or returns `InitialRing.NotConstant`. The default behavior is to map only 0 and 1 to their counterpart in the coefficient set. This is generally not desirable for non trivial computational rings.

preprocess [*ltac_expr* **]** specifies a tactic *ltac_expr* that is applied as a preliminary step for *ring* and *ring_simplify*. It can be used to transform a goal so that it is better recognized. For instance, `S n` can be changed to `plus 1 n`.

postprocess [*ltac_expr* **]** specifies a tactic *ltac_expr* that is applied as a final step for *ring_simplify*. For instance, it can be used to undo modifications of the preprocessor.

power *one_term* **[** *qualid*+ **]** to be documented

power_tac *one_term ltac_expr* **]** allows *ring* and *ring_simplify* to recognize power expressions with a constant positive integer exponent (example: x^2). The term *one_term* is a proof that a given power function satisfies the specification of a power function (term has to be a proof of `Ring_theory.power_theory`) and *tactic* specifies a tactic expression that, given a term, "abstracts" it into an object of type `N` whose interpretation via `Cp_phi` (the evaluation function of power coefficient) is the original term, or returns `InitialRing.NotConstant` if not a constant coefficient (i.e. L_{tac} is the inverse function of `Cp_phi`). See files `plugins/ring/ZArithRing.v` and `plugins/ring/RealField.v` for examples. By default the tactic does not recognize power expressions as ring expressions.

sign *one_term* allows *ring_simplify* to use a minus operation when outputting its normal form, i.e writing `x - y` instead of `x + (- y)`. The term *term* is a proof that a given sign function indicates expressions that are signed (*term* has to be a proof of `Ring_theory.get_sign`). See `plugins/ring/InitialRing.v` for examples of sign function.

div *one_term* allows *ring* and *ring_simplify* to use monomials with coefficients other than 1 in the rewriting. The term *one_term* is a proof that a given division function satisfies the specification of an euclidean division function (*one_term* has to be a proof of `Ring_theory.div_theory`). For example, this function is called when trying to rewrite $7x$ by $2x = z$ to tell that $7 = 3 \times 2 + 1$. See `plugins/ring/InitialRing.v` for examples of div function.

closed [*qualid*+ **]** to be documented

Error messages:

Error: Bad ring structure.
The proof of the ring structure provided is not of the expected type.

Error: Bad lemma for decidability of equality.
The equality function provided in the case of a computational ring has not the expected type.

Error: Ring operation should be declared as a morphism.
A setoid associated with the carrier of the ring structure has been found, but the ring operation should be declared as morphism. See *Tactics enabled on user provided relations*.

How does it work?

The code of `ring` is a good example of a tactic written using *reflection*. What is reflection? Basically, using it means that a part of a tactic is written in Gallina, Coq's language of terms, rather than $\mathtt{L}_{\mathtt{tac}}$ or OCaml. From the philosophical point of view, reflection is using the ability of the Calculus of Constructions to speak and reason about itself. For the `ring` tactic we used Coq as a programming language and also as a proof environment to build a tactic and to prove its correctness.

The interested reader is strongly advised to have a look at the file `Ring_polynom.v`. Here a type for polynomials is defined:

```
Inductive PExpr : Type :=
  | PEc : C -> PExpr
  | PEX : positive -> PExpr
  | PEadd : PExpr -> PExpr -> PExpr
  | PEsub : PExpr -> PExpr -> PExpr
  | PEmul : PExpr -> PExpr -> PExpr
  | PEopp : PExpr -> PExpr
  | PEpow : PExpr -> N -> PExpr.
```

Polynomials in normal form are defined as:

```
Inductive Pol : Type :=
  | Pc : C -> Pol
  | Pinj : positive -> Pol -> Pol
  | PX : Pol -> positive -> Pol -> Pol.
```

where `Pinj n P` denotes `P` in which V_i is replaced by V_{i+n} , and `PX P n Q` denotes $P \otimes V_1^n \oplus Q'$, `Q'` being `Q` where V_i is replaced by V_{i+1}.

Variable maps are represented by lists of ring elements, and two interpretation functions, one that maps a variables map and a polynomial to an element of the concrete ring, and the second one that does the same for normal forms:

```
Definition PEeval : list R -> PExpr -> R := [...].
Definition Pphi_dev : list R -> Pol -> R := [...].
```

A function to normalize polynomials is defined, and the big theorem is its correctness w.r.t interpretation, that is:

```
Definition norm : PExpr -> Pol := [...].
Lemma Pphi_dev_ok :
   forall l pe npe, norm pe = npe -> PEeval l pe == Pphi_dev l npe.
```

So now, what is the scheme for a normalization proof? Let p be the polynomial expression that the user wants to normalize. First a little piece of ML code guesses the type of `p`, the ring theory `T` to use, an abstract polynomial `ap` and a variables map `v` such that `p` is βδι- equivalent to `(PEeval v ap)`. Then we replace it by `(Pphi_dev v (norm ap))`, using the main correctness theorem and we reduce it to a concrete expression `p'`, which is the concrete normal form of `p`. This is summarized in this diagram:

`p`	$\rightarrow_{\beta\delta\iota}$	`(PEeval v ap)`
	$=_{\text{(by the main correctness theorem)}}$	
`p'`	$\leftarrow_{\beta\delta\iota}$	`(Pphi_dev v (norm ap))`

The user does not see the right part of the diagram. From outside, the tactic behaves like a βδι simplification extended with rewriting rules for associativity and commutativity. Basically, the proof is only the application of the main correctness theorem to well-chosen arguments.

Dealing with fields

Tactic: field [*one_term*+]?

An extension of the *ring* tactic that deals with rational expressions. Given a rational expression $F = 0$. It first reduces the expression `F` to a common denominator $N/D = 0$ where `N` and `D` are two ring expressions. For example, if we take $F = (1-1/x)x - x + 1$, this gives $N = (x-1)x - x^2 + x$ and $D = x$. It then calls ring to solve $N = 0$.

[*one_term*+] If specified, the tactic decides the equality of two terms modulo field operations and the equalities defined by the *one_term*s. Each *one_term* has to be a proof of some equality `m = p`, where `m` is a monomial (after "abstraction"), `p` a polynomial and `=` the corresponding equality of the field structure.

Note:

Rewriting works with the equality `m = p` only if `p` is a polynomial since rewriting is handled by the underlying ring tactic.

Note that **field** also generates nonzero conditions for all the denominators it encounters in the reduction. In our example, it generates the condition $x \neq 0$. These conditions appear as one subgoal which is a conjunction if there are several denominators. Nonzero conditions are always polynomial expressions. For example when reducing the expression $1/(1 + 1/x)$, two side conditions are generated: $x \neq 0$ and $x + 1 \neq 0$. Factorized expressions are broken since a field is an integral domain, and when the equality test on coefficients is complete w.r.t. the equality of the target field, constants can be proven different from zero automatically.

The tactic must be loaded by `Require Import Field`. New field structures can be declared to the system with the `Add Field` command (see below). The field of real numbers is defined in module `RealField` (in `plugins/ring`). It is exported by module `Rbase`, so that requiring `Rbase` or `Reals` is enough to use the field tactics on real numbers. Rational numbers in canonical form are also declared as a field in the module `Qcanon`.

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x,
       x <> 0 -> (1 - 1 / x) * x - x + 1 = 0.
    1 goal

      ============================
      forall x : R, x <> 0 -> (1 - 1 / x) * x - x + 1 = 0

intros; field; auto.
    No more goals.

Abort.
Goal forall x y,
       y <> 0 -> y = x -> x / y = 1.
    1 goal

      ============================
      forall x y : R, y <> 0 -> y = x -> x / y = 1

intros x y H H1; field [H1]; auto.
```

(continues on next page)

(continued from previous page)

```
    No more goals.

Abort.
```

Example: `field` that generates side goals

```
Require Import Reals.
Goal forall x y:R,
(x * y > 0)%R ->
(x * (1 / x + x / (x + y)))%R =
((- 1 / y) * y * (- x * (x / (x + y)) - 1))%R.
    1 goal

  ============================
  forall x y : R,
  (x * y > 0)%R ->
  (x * (1 / x + x / (x + y)))%R = (-1 / y * y * (- x * (x / (x + y)) - 1))%R


intros; field.
    1 goal

  x, y : R
  H : (x * y > 0)%R
  ============================
  (x + y)%R <> 0%R /\ y <> 0%R /\ x <> 0%R
```

Tactic: `field_simplify` [*one_term*$_{eq}$]? [*one_term*]+ **`in`** *ident*?

Performs the simplification in the conclusion of the goal, $F_1 = F_2$ becomes $N_1/D_1 = N_2/D_2$. A normalization step (the same as the one for rings) is then applied to N_1, D_1, N_2 and D_2. This way, polynomials remain in factorized form during fraction simplification. This yields smaller expressions when reducing to the same denominator since common factors can be canceled.

[*one_term*$_{eq}$+ **]** Do simplification in the conclusion of the goal using the equalities defined by these *one_term*s.

one_term+ Terms to simplify in the conclusion.

`in` *ident* If specified, substitute in the hypothesis *ident* instead of the conclusion.

Tactic: `field_simplify_eq` [*one_term*+]? **`in`** *ident*?

Performs the simplification in the conclusion of the goal, removing the denominator. $F_1 = F_2$ becomes $N_1 D_2 = N_2 D_1$.

[*one_term*+ **]** Do simplification in the conclusion of the goal using the equalities defined by these *one_term*s.

`in` *ident* If specified, simplify in the hypothesis *ident* instead of the conclusion.

Adding a new field structure

Declaring a new field consists in proving that a field signature (a carrier set, an equality, and field operations: `Field_theory.field_theory` and `Field_theory.semi_field_theory`) satisfies the field axioms. Semi-fields (fields without + inverse) are also supported. The equality can be either Leibniz equality, or any relation declared as a setoid (see *Tactics enabled on user provided relations*). The definition of fields and semifields is:

```
Record field_theory : Prop := mk_field {
  F_R : ring_theory rO rI radd rmul rsub ropp req;
  F_1_neq_0 : ~ 1 == 0;
  Fdiv_def : forall p q, p / q == p * / q;
  Finv_l : forall p, ~ p == 0 ->  / p * p == 1
}.

Record semi_field_theory : Prop := mk_sfield {
  SF_SR : semi_ring_theory rO rI radd rmul req;
  SF_1_neq_0 : ~ 1 == 0;
  SFdiv_def : forall p q, p / q == p * / q;
  SFinv_l : forall p, ~ p == 0 ->  / p * p == 1
}.
```

The result of the normalization process is a fraction represented by the following type:

```
Record linear : Type := mk_linear {
  num : PExpr C;
  denum : PExpr C;
  condition : list (PExpr C)
}.
```

where `num` and `denum` are the numerator and denominator; `condition` is a list of expressions that have appeared as a denominator during the normalization process. These expressions must be proven different from zero for the correctness of the algorithm.

The syntax for adding a new field is

Command: Add Field *ident* : *one_term* (*field_mod* , ... , *field_mod*)?

field_mod ::= *ring_mod*
| **completeness** *one_term*

The *ident* is used only for error messages. *one_term* is a proof that the field signature satisfies the (semi-)field axioms. The optional list of modifiers is used to tailor the behavior of the tactic.

Since field tactics are built upon `ring` tactics, all modifiers of *Add Ring* apply. There is only one specific modifier:

completeness *one_term* allows the field tactic to prove automatically that the image of nonzero coefficients are mapped to nonzero elements of the field. *one_term* is a proof of `forall x y, [x] == [y] -> x ?=! y = true`, which is the completeness of equality on coefficients w.r.t. the field equality.

History of ring

First Samuel Boutin designed the tactic `ACDSimpl`. This tactic did lot of rewriting. But the proofs terms generated by rewriting were too big for Coq's type checker. Let us see why:

```
Require Import ZArith.
Open Scope Z_scope.
Goal forall x y z : Z,
       x + 3 + y + y * z = x + 3 + y + z * y.
    1 goal

      ============================
      forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y

intros; rewrite (Zmult_comm y z); reflexivity.
    No more goals.

Save foo.
Print foo.
    foo =
    fun x y z : Z =>
    eq_ind_r (fun z0 : Z => x + 3 + y + z0 = x + 3 + y + z * y) eq_refl
      (Z.mul_comm y z)
         : forall x y z : Z, x + 3 + y + y * z = x + 3 + y + z * y

    Arguments foo (_ _ _)%Z_scope
```

At each step of rewriting, the whole context is duplicated in the proof term. Then, a tactic that does hundreds of rewriting generates huge proof terms. Since `ACDSimpl` was too slow, Samuel Boutin rewrote it using reflection (see [Bou97]). Later, it was rewritten by Patrick Loiseleur: the new tactic does not any more require `ACDSimpl` to compile and it makes use of βδι-reduction not only to replace the rewriting steps, but also to achieve the interleaving of computation and reasoning (see *Discussion*). He also wrote some ML code for the `Add Ring` command that allows registering new rings dynamically.

Proofs terms generated by ring are quite small, they are linear in the number of $\oplus$ and $\otimes$ operations in the normalized terms. Type checking those terms requires some time because it makes a large use of the conversion rule, but memory requirements are much smaller.

Discussion

Efficiency is not the only motivation to use reflection here. `ring` also deals with constants, it rewrites for example the expression `34 + 2 * x - x + 12` to the expected result `x + 46`. For the tactic `ACDSimpl`, the only constants were 0 and 1. So the expression `34 + 2 * (x - 1) + 12` is interpreted as $V_0 \oplus V_1 \otimes (V_2 \ominus 1) \oplus V_3$, with the variables mapping $\{V_0 \mapsto 34; V_1 \mapsto 2; V_2 \mapsto x; V_3 \mapsto 12\}$. Then it is rewritten to `34 - x + 2 * x + 12`, very far from the expected result. Here rewriting is not sufficient: you have to do some kind of reduction (some kind of computation) to achieve the normalization.

The tactic `ring` is not only faster than the old one: by using reflection, we get for free the integration of computation and reasoning that would be very difficult to implement without it.

Is it the ultimate way to write tactics? The answer is: yes and no. The `ring` tactic intensively uses the conversion rules of the Calculus of Inductive Constructions, i.e. it replaces proofs by computations as much as possible. It can be useful in all situations where a classical tactic generates huge proof terms, like symbolic processing and tautologies. But there are also tactics like `auto` or `linear` that do many complex computations, using side-effects and backtracking, and generate a small proof term. Clearly, it would be significantly less efficient to replace them by tactics using reflection.

Another idea suggested by Benjamin Werner: reflection could be used to couple an external tool (a rewriting program or a model checker) with Coq. We define (in Coq) a type of terms, a type of *traces*, and prove a correctness theorem that states that *replaying traces* is safe with respect to some interpretation. Then we let the external tool do every computation (using side-effects, backtracking, exception, or others features that are not available in pure lambda calculus) to produce the trace. Now we can check in Coq that the trace has the expected semantics by applying the correctness theorem.

3.2.4 Nsatz: a solver for equalities in integral domains

Author Loïc Pottier

To use the tactics described in this section, load the `Nsatz` module with the command `Require Import Nsatz`. Alternatively, if you prefer not to transitively depend on the files that declare the axioms used to define the real numbers, you can `Require Import NsatzTactic` instead; this will still allow *nsatz* to solve goals defined about $\mathbb{Z}$, $\mathbb{Q}$ and any user-registered rings.

Tactic:

nsatz **with radicalmax :=** *one_term* **strategy :=** *one_term* **parameters :=** *one_term* **variables :=**

This tactic is for solving goals of the form

$$\forall X_1, \ldots, X_n \in A,$$
$$P_1(X_1, \ldots, X_n) = Q_1(X_1, \ldots, X_n), \ldots, P_s(X_1, \ldots, X_n) = Q_s(X_1, \ldots, X_n)$$
$$\vdash P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)$$

where $P, Q, P_1, Q_1, \ldots, P_s, Q_s$ are polynomials and A is an integral domain, i.e. a commutative ring with no zero divisors. For example, A can be $\mathbb{R}$, $\mathbb{Z}$, or $\mathbb{Q}$. Note that the equality $=$ used in these goals can be any setoid equality (see *Tactics enabled on user provided relations*) , not only Leibniz equality.

It also proves formulas

$$\forall X_1, \ldots, X_n \in A,$$
$$P_1(X_1, \ldots, X_n) = Q_1(X_1, \ldots, X_n) \wedge \ldots \wedge P_s(X_1, \ldots, X_n) = Q_s(X_1, \ldots, X_n)$$
$$\rightarrow P(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n)$$

doing automatic introductions.

radicalmax bound when searching for r such that $c(P-Q)r = \sum_{i=1..s} S_i(Pi - Qi)$. This argument must be of type `N` (binary natural numbers).

strategy gives the order on variables $X_1, \ldots, X_n$ and the strategy used in Buchberger algorithm (see [GMN+91] for details):

- `strategy := 0%Z`: reverse lexicographic order and newest s-polynomial.
- `strategy := 1%Z`: reverse lexicographic order and sugar strategy.
- `strategy := 2%Z`: pure lexicographic order and newest s-polynomial.
- `strategy := 3%Z`: pure lexicographic order and sugar strategy.

parameters a list of parameters of type `R`, containing the variables $X_{i_1}, \ldots, X_{i_k}$ among $X_1, \ldots, X_n$. Computation will be performed with rational fractions in these parameters, i.e. polynomials have coefficients in $R(X_{i_1}, \ldots, X_{i_k})$. In this case, the coefficient c can be a nonconstant polynomial in $X_{i_1}, \ldots, X_{i_k}$, and the tactic produces a goal which states that c is not zero.

variables a list of variables of type `R` in the decreasing order in which they will be used in the Buchberger algorithm. If the list is empty, then `lvar` is replaced by all the variables which are not in `parameters`.

See the file Nsatz.v[47] for examples, especially in geometry.

[47] https://github.com/coq/coq/blob/master/test-suite/success/Nsatz.v

More about `nsatz`

Hilbert's Nullstellensatz theorem shows how to reduce proofs of equalities on polynomials on a commutative ring A with no zero divisors to algebraic computations: it is easy to see that if a polynomial P in $A[X_1, \ldots, X_n]$ verifies $cP^r = \sum_{i=1}^{s} S_i P_i$, with $c \in A$, $c \neq 0$, r a positive integer, and the S_i s in $A[X_1, \ldots, X_n]$, then P is zero whenever polynomials $P_1, \ldots, P_s$ are zero (the converse is also true when A is an algebraically closed field: the method is complete).

So, solving our initial problem reduces to finding $S_1, \ldots, S_s$, c and r such that $c(P-Q)^r = \sum_i S_i(P_i - Q_i)$, which will be proved by the tactic ring.

This is achieved by the computation of a Gröbner basis of the ideal generated by $P_1 - Q_1, ..., P_s - Q_s$, with an adapted version of the Buchberger algorithm.

This computation is done after a step of *reification*, which is performed using *Typeclasses*.

3.2.5 Programmable proof search

Tactic: `auto` *nat_or_var*? *auto_using*? *hintbases*?

auto_using ::= **using** *one_term*, +

hintbases ::= **with** *

| **with** *ident* +

Implements a Prolog-like resolution procedure to solve the current goal. It first tries to solve the goal using the `assumption` tactic, then it reduces the goal to an atomic one using `intros` and introduces the newly generated hypotheses as hints. Then it looks at the list of tactics associated with the head symbol of the goal and tries to apply one of them. Lower cost tactics are tried before higher-cost tactics. This process is recursively applied to the generated subgoals.

nat_or_var Specifies the maximum search depth. The default is 5.

`using` *one_term*, +

Uses lemmas *one_term*, + in addition to hints. If *one_term* is an inductive type, the collection of its constructors are added as hints.

Note that hints passed through the `using` clause are used in the same way as if they were passed through a hint database. Consequently, they use a weaker version of `apply` and **`auto using`** *one_term* may fail where **`apply`** *one_term* succeeds.

`with *` Use all existing hint databases. Using this variant is highly discouraged in finished scripts since it is both slower and less robust than explicitly selecting the required databases.

`with` *ident* + Use the hint databases *ident* + in addition to the database `core`. Use the fake database `nocore` to omit `core`.

If no `with` clause is given, `auto` only uses the hypotheses of the current goal and the hints of the database named `core`.

`auto` generally either completely solves the goal or leaves it unchanged. Use `solve [ auto ]` if you want a failure when they don't solve the goal. `auto` will fail if `fail` or `gfail` are invoked directly or indirectly, in which case setting the `Ltac Debug` may help you debug the failure.

Warning: *auto* uses a weaker version of *apply* that is closer to *simple apply* so it is expected that sometimes *auto* will fail even if applying manually one of the hints would succeed.

See also:

The hints databases for auto and eauto for the list of pre-defined databases and the way to create or extend a database.

Tactic: info_auto *nat_or_var*? *auto_using*? *hintbases*?
Behaves like *auto* but shows the tactics it uses to solve the goal. This variant is very useful for getting a better understanding of automation, or to know what lemmas/assumptions were used.

Tactic: debug auto *nat_or_var*? *auto_using*? *hintbases*?
Behaves like *auto* but shows the tactics it tries to solve the goal, including failing paths.

Tactic: trivial *auto_using*? *hintbases*?

Tactic: debug trivial *auto_using*? *hintbases*?

Tactic: info_trivial *auto_using*? *hintbases*?
Like *auto*, but is not recursive and only tries hints with zero cost. Typically used to solve goals for which a lemma is already available in the specified **hintbases**.

Flag: Info Auto
Flag: Debug Auto
Flag: Info Trivial
Flag: Debug Trivial
These *flags* enable printing of informative or debug information for the *auto* and *trivial* tactics.

Tactic: eauto *nat_or_var*? *auto_using*? *hintbases*?
Generalizes *auto*. While *auto* does not try resolution hints which would leave existential variables in the goal, *eauto* does try them (informally speaking, it internally uses a tactic close to *simple eapply* instead of a tactic close to *simple apply* in the case of *auto*). As a consequence, *eauto* can solve such a goal:

Example

```
Hint Resolve ex_intro : core.
    The hint ex_intro will only be used by eauto, because applying ex_intro would
    leave variable x as unresolved existential variable.

Goal forall P:nat -> Prop, P 0 -> exists n, P n.
    1 goal

      ============================
      forall P : nat -> Prop, P 0 -> exists n : nat, P n

eauto.
    No more goals.
```

`ex_intro` is declared as a hint so the proof succeeds.

See also:

The hints databases for auto and eauto

Tactic: `info_eauto` *nat_or_var*? *auto_using*? *hintbases*?
The various options for `info_eauto` are the same as for `info_auto`.

`eauto` also obeys the following flags:

Flag: `Info Eauto`
Flag: `Debug Eauto`

Tactic: `debug eauto` *nat_or_var*? *auto_using*? *hintbases*?
Behaves like `eauto` but shows the tactics it tries to solve the goal, including failing paths.

Tactic: `bfs eauto` *nat_or_var*? *auto_using*? *hintbases*?
Like `eauto`, but uses a breadth-first search[48].

Tactic: `autounfold` *hintbases*? *simple_occurrences*?
Unfolds constants that were declared through a `Hint Unfold` in the given databases.

simple_occurrences Performs the unfolding in the specified occurrences.

Tactic: `autorewrite` `*`? `with` *ident*+ *occurrences*? `using` *ltac_expr*?

`*` If present, rewrite all occurrences whose side conditions are solved.

`with` *ident*+ Specifies the rewriting rule bases to use.

occurrences Performs rewriting in the specified occurrences. Note: the `at` clause is currently not supported.

Error: `The "at" syntax isn't available yet for the autorewrite tactic.`
Appears when there is an `at` clause on the conclusion.

`using` *ltac_expr* `ltac_expr` is applied to the main subgoal after each rewriting step.

Applies rewritings according to the rewriting rule bases *ident*+.

For each rule base, applies each rewriting to the main subgoal until it fails. Once all the rules have been processed, if the main subgoal has changed then the rules of this base are processed again. If the main subgoal has not changed then the next base is processed. For the bases, the behavior is very similar to the processing of the rewriting rules.

The rewriting rule bases are built with the `Hint Rewrite` command.

Warning: This tactic may loop if you build non terminating rewriting systems.

See also:

`Hint Rewrite` for feeding the database of lemmas used by `autorewrite` and `autorewrite` for examples showing the use of this tactic. Also see *Strategies for rewriting*.

Here are two examples of `autorewrite` use. The first one (*Ackermann function*) shows actually a quite basic use where there is no conditional rewriting. The second one (*Mac Carthy function*) involves conditional rewritings and shows how to deal with them using the optional tactic of the `Hint Rewrite` command.

Example: Ackermann function

```
Require Import Arith.
```

[48] https://en.wikipedia.org/wiki/Breadth-first_search

```
Parameter Ack : nat -> nat -> nat.

Axiom Ack0 : forall m:nat, Ack 0 m = S m.
Axiom Ack1 : forall n:nat, Ack (S n) 0 = Ack n 1.
Axiom Ack2 : forall n m:nat, Ack (S n) (S m) = Ack n (Ack (S n) m).

Global Hint Rewrite Ack0 Ack1 Ack2 : base0.

Lemma ResAck0 : Ack 3 2 = 29.
    1 goal

      ============================
      Ack 3 2 = 29

autorewrite with base0 using try reflexivity.
    No more goals.
```

Example: MacCarthy function

```
Require Import Lia.

Parameter g : nat -> nat -> nat.

Axiom g0 : forall m:nat, g 0 m = m.
Axiom g1 : forall n m:nat, (n > 0) -> (m > 100) -> g n m = g (pred n) (m - 10).
Axiom g2 : forall n m:nat, (n > 0) -> (m <= 100) -> g n m = g (S n) (m + 11).

Global Hint Rewrite g0 g1 g2 using lia : base1.

Lemma Resg0 : g 1 110 = 100.

    1 goal

      ============================
      g 1 110 = 100

autorewrite with base1 using reflexivity || simpl.
    No more goals.

Lemma Resg1 : g 1 95 = 91.
    1 goal

      ============================
      g 1 95 = 91

autorewrite with base1 using reflexivity || simpl.
    No more goals.
```

Tactic: `easy`

This tactic tries to solve the current goal by a number of standard closing steps. In particular, it tries to close the current goal using the closing tactics *trivial*, *reflexivity*, *symmetry*, *contradiction* and

inversion of hypothesis. If this fails, it tries introducing variables and splitting and-hypotheses, using the closing tactics afterwards, and splitting the goal using *split* and recursing.

This tactic solves goals that belong to many common classes; in particular, many cases of unsatisfiable hypotheses, and simple equality goals are usually solved by this tactic.

Tactic: now *ltac_expr*
Run ***tactic*** followed by *easy*. This is a notation for ***tactic*****; easy**.

The hints databases for auto and eauto

The hints for *auto* and *eauto* are stored in databases. Each database maps head symbols to a list of hints. Use the *Print Hint* command to view the database.

Each hint has a cost that is a nonnegative integer and an optional pattern. Hints with lower costs are tried first. *auto* tries a hint when the conclusion of the current goal matches its pattern or when the hint has no pattern.

Creating Hint databases

Hint databases can be created with the *Create HintDb* command or implicitly by adding a hint to an unknown database. We recommend you always use *Create HintDb* and then imediately use *Hint Constants* and *Hint Variables* to make those settings explicit.

Note that the default transparency settings differ between these two methods of creation. Databases created with *Create HintDb* have the default setting `Transparent` for both `Variables` and `Constants`, while implicitly created databases have the `Opaque` setting.

Command: Create HintDb *ident* discriminated ?
Creates a new hint database named ***ident***. The database is implemented by a Discrimination Tree (DT) that serves as an index of all the lemmas. The DT can use transparency information to decide if a constant should be indexed or not making the retrieval more efficient. The legacy implementation (the default one for new databases) uses the DT only on goals without existentials (i.e., *auto* goals), for non-Immediate hints and does not make use of transparency hints, putting more work on the unification that is run after retrieval (it keeps a list of the lemmas in case the DT is not used). The new implementation enabled by the discriminated option makes use of DTs in all cases and takes transparency information into account. However, the order in which hints are retrieved from the DT may differ from the order in which they were inserted, making this implementation observationally different from the legacy one.

Creating Hints

The various `Hint` commands share these elements:

: ***ident*** + ? specifies the hint database(s) to add to. *(Deprecated since version 8.10:* If no *ident*s are given, the hint is added to the `core` database.)

Outside of sections, these commands support the *local*, *export* and *global* attributes. *global* is the default. Inside sections, only the *local* attribute is supported because hints are local to sections.

- *local* hints are never visible from other modules, even if they *Import* or *Require* the current module.
- *export* hints are visible from other modules when they *Import* the current module, but not when they only *Require* it.
- *global* hints are visible from other modules when they *Import* or *Require* the current module.

New in version 8.14: The *Hint Rewrite* now supports locality attributes like other Hint commands.

Deprecated since version 8.13: The default value for hint locality will change in a future release. Hints added outside of sections without an explicit locality are now deprecated. We recommend using *export* where possible.

The Hint commands are:

Command: Hint Resolve *qualid* | *one_term* [+] *hint_info* [?] [: *ident* [+]] [?]

Command: Hint Resolve -> | <- *qualid* [+] *natural* [?] [: *ident* [+]] [?]

hint_info ::= | *natural* [?] *one_pattern* [?]
one_pattern ::= *one_term*

The first form adds each ***qualid*** as a hint with the head symbol of the type of ***qualid*** to the specified hint databases (***ident***s). The cost of the hint is the number of subgoals generated by *simple apply* ***qualid*** or, if specified, ***natural***. The associated pattern is inferred from the conclusion of the type of ***qualid*** or, if specified, the given ***one_pattern***.

If the inferred type of ***qualid*** does not start with a product, *exact* ***qualid*** is added to the hint list. If the type can be reduced to a type starting with a product, *simple apply* ***qualid*** is also added to the hints list.

If the inferred type of ***qualid*** contains a dependent quantification on a variable which occurs only in the premises of the type and not in its conclusion, no instance could be inferred for the variable by unification with the goal. In this case, the hint is only used by *eauto* / *typeclasses eauto*, but not by *auto*. A typical hint that would only be used by *eauto* is a transitivity lemma.

-> | **<-** The second form adds the left-to-right (->) or right-ot-left implication (<-) of an equivalence as a hint (informally the hint will be used as, respectively, *apply* **-> *qualid*** or *apply* **<- *qualid***, although as mentioned before, the tactic actually used is a restricted version of *apply*).

one_term Permits declaring a hint without declaring a new constant first, but this is not recommended.

> **Warning: Declaring arbitrary terms as hints is fragile; it is recommended to declare a toplevel constant instead**

Error: *qualid* cannot be used as a hint
The head symbol of the type of ***qualid*** is a bound variable such that this tactic cannot be associated with a constant.

Command: Hint Immediate *qualid* | *one_term* [+] [: *ident* [+]] [?]
For each specified ***qualid***, adds the tactic *simple apply* ***qualid***; *solve* **[** *trivial* **]** to the hint list associated with the head symbol of the type of ***qualid***. This tactic will fail if all the subgoals generated by *simple apply* ***qualid*** are not solved immediately by the *trivial* tactic (which only tries tactics with cost 0). This command is useful for theorems such as the symmetry of equality or `n+1=m+1 -> n=m` that we may want to introduce with limited use in order to avoid useless proof search. The cost of this tactic (which never generates subgoals) is always 1, so that it is not used by *trivial* itself.

Command: Hint Constructors *qualid* [+] [: *ident* [+]] [?]
For each ***qualid*** that is an inductive type, adds all its constructors as hints of type Resolve. Then,

when the conclusion of current goal has the form **(*qualid* ...)**, *auto* will try to apply each constructor.

Error: *qualid* is not an inductive type

Command: Hint Unfold *qualid*+ [: *ident*+]?

For each ***qualid***, adds the tactic *unfold* ***qualid*** to the hint list that will only be used when the head constant of the goal is *qualid*. Its cost is 4.

Command: Hint **Transparent** | **Opaque** *qualid*+ [: *ident*+]?

Adds transparency hints to the database, making each ***qualid*** a transparent or opaque constant during resolution. This information is used during unification of the goal with any lemma in the database and inside the discrimination network to relax or constrain it in the case of discriminated databases.

Error: Cannot coerce *qualid* to an evaluable reference.

Command:

Hint **Constants** | **Variables** | **Transparent** | **Opaque** [: *ident*+]?

Sets the transparency flag for constants or variables for the specified hint databases. These flags affect the unification of hints in the database. We advise using this just after a *Create HintDb* command.

Command: Hint Extern *natural* [*one_pattern*]? **=>** *ltac_expr* [: *ident*+]?

Extends *auto* with tactics other than *apply* and *unfold*. ***natural*** is the cost, ***one_term*** is the pattern to match and ***ltac_expr*** is the action to apply.

Note: Use a *Hint Extern* with no pattern to do pattern matching on hypotheses using `match goal with` inside the tactic.

Example

```
Hint Extern 4 (~(_ = _)) => discriminate : core.
```

Now, when the head of the goal is a disequality, `auto` will try discriminate if it does not manage to solve the goal with hints with a cost less than 4.

One can even use some sub-patterns of the pattern in the tactic script. A sub-pattern is a question mark followed by an identifier, like `?X1` or `?X2`. Here is an example:

Example

```
Require Import List.
Hint Extern 5 ({?X1 = ?X2} + {?X1 <> ?X2}) =>
  generalize  X1, X2; decide equality : eqdec.
Goal forall a b:list (nat * nat), {a = b} + {a <> b}.
    1 goal

      ============================
      forall a b : list (nat * nat), {a = b} + {a <> b}

info_auto.
```

(continues on next page)

(continued from previous page)

```
(* info auto: *)
idtac.
```

Command: `Hint Cut [ hints_regexp ] : ident+ ?`

```
hints_regexp ::= qualid+                    (hint or instance identifier)
               | _                          (any hint)
               | hints_regexp |             (disjunction)
                 hints_regexp
               | hints_regexp hints_regexp  (sequence)
               | hints_regexp *             (Kleene star)
               | emp                        (empty)
               | eps                        (epsilon)
               | ( hints_regexp )
```

Used to cut the proof search tree according to a regular expression that matches the paths to be cut.

During proof search, the path of successive successful hints on a search branch is recorded as a list of identifiers for the hints (note that *Hint Extern*s do not have an associated identifier). For each hint ***qualid*** in the hint database, the current path `p` extended with ***qualid*** is matched against the current cut expression `c` associated with the hint database. If the match succeeds the hint is *not* applied.

`Hint Cut` ***hints_regexp*** sets the cut expression to **`c |`** ***hints_regexp***. The initial cut expression is `emp`.

The output of *Print HintDb* shows the cut expression.

> **Warning:** The regexp matches the entire path. Most hints will start with a leading `( _* )` to match the tail of the path. (Note that `(_*)` misparses since `*)` would end a comment.)

> **Warning:** There is no operator precedence during parsing, one can check with *Print HintDb* to verify the current cut expression.

> **Warning:** These hints currently only apply to typeclass proof search and the *typeclasses eauto* tactic.

Command: `Hint Mode` ***qualid*** `+ | ! | -`+ `: ident`+ ?

Sets an optional mode of use for the identifier ***qualid***. When proof search has a goal that ends in an application of ***qualid*** to arguments ***arg*** `...` ***arg***, the mode tells if the hints associated with ***qualid*** can be applied or not. A mode specification is a list of `+`, `!` or `-` items that specify if an argument of the identifier is to be treated as an input (`+`), if its head only is an input (`!`) or an output (`-`) of the identifier. Mode `-` matches any term, mode `+` matches a term if and only if it does not contain existential variables, while mode `!` matches a term if and only if the *head* of the term is not an existential variable. The head of a term is understood here as the applicative head, recursively, ignoring casts.

Hint Mode is especially useful for typeclasses, when one does not want to support default instances and wants to avoid ambiguity in general. Setting a parameter of a class as an input forces proof search to be driven by that index of the class, with ! allowing existentials to appear in the index but not at its head.

Note:

- Multiple modes can be declared for a single identifier. In that case only one mode needs to match the arguments for the hints to be applied.
- If you want to add hints such as *Hint Transparent*, *Hint Cut*, or *Hint Mode*, for typeclass resolution, do not forget to put them in the `typeclass_instances` hint database.

Command: **Hint Rewrite** [-> | <-]? *one_term*+ [**using** *ltac_expr*]? [: *ident**]?

[**using** *ltac_expr*]? If specified, *ltac_expr* is applied to the generated subgoals, except for the main subgoal.

[-> | <-] Arrows specify the orientation; left to right (**->**) or right to left (**<-**). If no arrow is given, the default orientation is left to right (**->**).

Adds the terms *one_term*+ (their types must be equalities) to the rewriting bases *ident** . Note that the rewriting bases are distinct from the *auto* hint bases and that *auto* does not take them into account.

Command: **Print Rewrite HintDb** *ident*

This command displays all rewrite hints contained in *ident*.

Command: **Remove Hints** *qualid*+ [: *ident*+]?

Removes the hints associated with the *qualid*+ in databases *ident*+ . Note: hints created with *Hint Extern* currently can't be removed. The best workaround for this is to make the hints non global and carefully select which modules you import.

Command: **Print Hint** [* | *reference*]?

***** Display all declared hints.

reference Display all hints associated with the head symbol *reference*.

Displays tactics from the hints list. The default is to show hints that apply to the conclusion of the current goal. The other forms with ***** and *reference* can be used even if no proof is open.

Each hint has a cost that is a nonnegative integer and an optional pattern. The hints with lower cost are tried first.

Command: **Print HintDb** *ident*

This command displays all hints from database *ident*.

Hint databases defined in the Coq standard library

Several hint databases are defined in the Coq standard library. The actual content of a database is the collection of hints declared to belong to this database in each of the various modules currently loaded. Especially, requiring new modules may extend the database. At Coq startup, only the core database is nonempty and can be used.

core This special database is automatically used by `auto`, except when pseudo-database `nocore` is given to `auto`. The core database contains only basic lemmas about negation, conjunction, and so on. Most of the hints in this database come from the Init and Logic directories.

arith This database contains all lemmas about Peano's arithmetic proved in the directories Init and Arith.

zarith contains lemmas about binary signed integers from the directories theories/ZArith. The database also contains high-cost hints that call *`lia`* on equations and inequalities in `nat` or `Z`.

bool contains lemmas about booleans, mostly from directory theories/Bool.

datatypes is for lemmas about lists, streams and so on that are mainly proved in the Lists subdirectory.

sets contains lemmas about sets and relations from the directories Sets and Relations.

typeclass_instances contains all the typeclass instances declared in the environment, including those used for `setoid_rewrite`, from the Classes directory.

fset internal database for the implementation of the `FSets` library.

ordered_type lemmas about ordered types (as defined in the legacy `OrderedType` module), mainly used in the `FSets` and `FMaps` libraries.

You are advised not to put your own hints in the core database, but use one or several databases specific to your development.

Hint locality

Hints provided by the `Hint` commands are erased when closing a section. Conversely, all hints of a module `A` that are not defined inside a section (and not defined with option `Local`) become available when the module `A` is required (using e.g. `Require A.`).

As of today, hints only have a binary behavior regarding locality, as described above: either they disappear at the end of a section scope, or they remain global forever. This causes a scalability issue, because hints coming from an unrelated part of the code may badly influence another development. It can be mitigated to some extent thanks to the *`Remove Hints`* command, but this is a mere workaround and has some limitations (for instance, external hints cannot be removed).

A proper way to fix this issue is to bind the hints to their module scope, as for most of the other objects Coq uses. Hints should only be made available when the module they are defined in is imported, not just required. It is very difficult to change the historical behavior, as it would break a lot of scripts. We propose a smooth transitional path by providing the *`Loose Hint Behavior`* option which accepts three flags allowing for a fine-grained handling of non-imported hints.

Option: `Loose Hint Behavior` **`"Lax"`** | **`"Warn"`** | **`"Strict"`**

This *option* accepts three values, which control the behavior of hints w.r.t. *`Import`*:

- "Lax": this is the default, and corresponds to the historical behavior, that is, hints defined outside of a section have a global scope.
- "Warn": outputs a warning when a non-imported hint is used. Note that this is an over-approximation, because a hint may be triggered by a run that will eventually fail and backtrack, resulting in the hint not being actually useful for the proof.
- "Strict": changes the behavior of an unloaded hint to a immediate fail tactic, allowing to emulate an import-scoped hint mechanism.

Setting implicit automation tactics

Command: `Proof with` *`ltac_expr`* `using` *`section_var_expr`* ?

Starts a proof in which *`ltac_expr`* is applied to the active goals after each tactic that ends with `...` instead of the usual single period. "*`tactic`*`...`" is equivalent to "*`tactic`*`; `*`ltac_expr`*`.".

See also:

`Proof` in *Entering and exiting proof mode*.

3.2.6 Generalized rewriting

Author Matthieu Sozeau

This chapter presents the extension of several equality related tactics to work over user-defined structures (called setoids) that are equipped with ad-hoc equivalence relations meant to behave as equalities. Actually, the tactics have also been generalized to relations weaker than equivalences (e.g. rewriting systems). The toolbox also extends the automatic rewriting capabilities of the system, allowing the specification of custom strategies for rewriting.

This documentation is adapted from the previous setoid documentation by Claudio Sacerdoti Coen (based on previous work by Clément Renard). The new implementation is a drop-in replacement for the old one[49], hence most of the documentation still applies.

The work is a complete rewrite of the previous implementation, based on the typeclass infrastructure. It also improves on and generalizes the previous implementation in several ways:

- User-extensible algorithm. The algorithm is separated into two parts: generation of the rewriting constraints (written in ML) and solving these constraints using typeclass resolution. As typeclass resolution is extensible using tactics, this allows users to define general ways to solve morphism constraints.
- Subrelations. An example extension to the base algorithm is the ability to define one relation as a subrelation of another so that morphism declarations on one relation can be used automatically for the other. This is done purely using tactics and typeclass search.
- Rewriting under binders. It is possible to rewrite under binders in the new implementation, if one provides the proper morphisms. Again, most of the work is handled in the tactics.
- First-class morphisms and signatures. Signatures and morphisms are ordinary Coq terms, hence they can be manipulated inside Coq, put inside structures and lemmas about them can be proved inside the system. Higher-order morphisms are also allowed.
- Performance. The implementation is based on a depth-first search for the first solution to a set of constraints which can be as fast as linear in the size of the term, and the size of the proof term is linear in the size of the original term. Besides, the extensibility allows the user to customize the proof search if necessary.

Introduction to generalized rewriting

Relations and morphisms

A parametric *relation* `R` is any term of type `forall (x1 : T1) ... (xn : Tn), relation A`. The expression `A`, which depends on `x1 ... xn`, is called the *carrier* of the relation and `R` is said to be a relation over `A`; the list `x1,...,xn` is the (possibly empty) list of parameters of the relation.

Example: Parametric relation

[49] Nicolas Tabareau helped with the gluing.

It is possible to implement finite sets of elements of type `A` as unordered lists of elements of type `A`. The function `set_eq: forall (A : Type), relation (list A)` satisfied by two lists with the same elements is a parametric relation over `(list A)` with one parameter `A`. The type of `set_eq` is convertible with `forall (A : Type), list A -> list A -> Prop.`

An *instance* of a parametric relation `R` with n parameters is any term `(R t1 ... tn)`.

Let `R` be a relation over `A` with `n` parameters. A term is a parametric proof of reflexivity for `R` if it has type `forall (x1 : T1) ... (xn : Tn), reflexive (R x1 ... xn)`. Similar definitions are given for parametric proofs of symmetry and transitivity.

Example: Parametric relation (continued)

The `set_eq` relation of the previous example can be proved to be reflexive, symmetric and transitive. A parametric unary function `f` of type `forall (x1 : T1) ... (xn : Tn), A1 -> A2` covariantly respects two parametric relation instances `R1` and `R2` if, whenever `x`, `y` satisfy `R1 x y`, their images `(f x)` and `(f y)` satisfy `R2 (f x) (f y)`. An `f` that respects its input and output relations will be called a unary covariant *morphism*. We can also say that `f` is a monotone function with respect to `R1` and `R2` . The sequence `x1 ... xn` represents the parameters of the morphism.

Let `R1` and `R2` be two parametric relations. The *signature* of a parametric morphism of type `forall (x1 : T1) ... (xn : Tn), A1 -> A2` that covariantly respects two instances I_{R_1} and I_{R_2} of `R1` and `R2` is written $I_{R_1} ++> I_{R_2}$. Notice that the special arrow ++>, which reminds the reader of covariance, is placed between the two relation instances, not between the two carriers. The signature relation instances and morphism will be typed in a context introducing variables for the parameters.

The previous definitions are extended straightforwardly to n-ary morphisms, that are required to be simultaneously monotone on every argument.

Morphisms can also be contravariant in one or more of their arguments. A morphism is contravariant on an argument associated with the relation instance R if it is covariant on the same argument when the inverse relation R^{-1} (`inverse R` in Coq) is considered. The special arrow `-->` is used in signatures for contravariant morphisms.

Functions having arguments related by symmetric relations instances are both covariant and contravariant in those arguments. The special arrow ==> is used in signatures for morphisms that are both covariant and contravariant.

An instance of a parametric morphism f with n parameters is any term $f\, t_1 \ldots t_n$.

Example: Morphisms

Continuing the previous example, let `union: forall (A : Type), list A -> list A -> list A` perform the union of two sets by appending one list to the other. `union` is a binary morphism parametric over `A` that respects the relation instance `(set_eq A)`. The latter condition is proved by showing:

```
forall (A: Type) (S1 S1' S2 S2': list A),
  set_eq A S1 S1' ->
  set_eq A S2 S2' ->
  set_eq A (union A S1 S2) (union A S1' S2').
```

The signature of the function `union A` is `set_eq A ==> set_eq A ==> set_eq A` for all `A`.

Example: Contravariant morphisms

The division function `Rdiv : R -> R -> R` is a morphism of signature `le ++> le --> le` where `le` is the usual order relation over real numbers. Notice that division is covariant in its first argument and contravariant in its second

argument.

Leibniz equality is a relation and every function is a morphism that respects Leibniz equality. Unfortunately, Leibniz equality is not always the intended equality for a given structure.

In the next section we will describe the commands to register terms as parametric relations and morphisms. Several tactics that deal with equality in Coq can also work with the registered relations. The exact list of tactics will be given *in this section*. For instance, the tactic reflexivity can be used to solve a goal `R n n` whenever `R` is an instance of a registered reflexive relation. However, the tactics that replace in a context `C[]` one term with another one related by `R` must verify that `C[]` is a morphism that respects the intended relation. Currently the verification consists of checking whether `C[]` is a syntactic composition of morphism instances that respects some obvious compatibility constraints.

Example: Rewriting

Continuing the previous examples, suppose that the user must prove `set_eq int (union int (union int S1 S2) S2) (f S1 S2)` under the hypothesis `H : set_eq int S2 (@nil int)`. It is possible to use the `rewrite` tactic to replace the first two occurrences of `S2` with `@nil int` in the goal since the context `set_eq int (union int (union int S1 nil) nil) (f S1 S2)`, being a composition of morphisms instances, is a morphism. However the tactic will fail replacing the third occurrence of `S2` unless `f` has also been declared as a morphism.

Adding new relations and morphisms

Command:

Add Parametric Relation *binder** **:** *one_term*$_A$ *one_term*$_{Aeq}$ **reflexivity proved by** *one_term*? ...

Declares a parametric relation of *one_term*$_A$, which is a `Type`, say `T`, with *one_term*$_{Aeq}$, which is a relation on `T`, i.e. of type `(T -> T -> Prop)`. Thus, if *one_term*$_A$ is **A: forall** a_1 … a_n**, Type** then *one_term*$_{Aeq}$ is **Aeq: forall** a_1 … a_n**, (A** a_1 … a_n**) -> (A** a_1 … a_n**) -> Prop**, or equivalently, **Aeq: forall** a_1 … a_n**, relation (A** a_1 … a_n**)**.

one_term$_A$ and *one_term*$_{Aeq}$ must be typeable under the context *binder*s. In practice, the *binder*s usually correspond to the **a**s

The final *ident* gives a unique name to the morphism and it is used by the command to generate fresh names for automatically provided lemmas used internally.

Notice that the carrier and relation parameters may refer to the context of variables introduced at the beginning of the declaration, but the instances need not be made only of variables. Also notice that `A` is *not* required to be a term having the same parameters as `Aeq`, although that is often the case in practice (this departs from the previous implementation).

To use this command, you need to first import the module `Setoid` using the command `Require Import Setoid`.

Command:

Add Relation *one_term one_term* **reflexivity proved by** *one_term*? **symmetry proved by** *one_ter*...

If the carrier and relations are not parametric, use this command instead, whose syntax is the same except there is no local context.

The proofs of reflexivity, symmetry and transitivity can be omitted if the relation is not an equivalence relation. The proofs must be instances of the corresponding relation definitions: e.g. the proof of reflexivity must have a type convertible to `reflexive (A t1 … tn) (Aeq t' 1 … t' n)`. Each proof may refer to the introduced variables as well.

Example: Parametric relation

For Leibniz equality, we may declare:

```
Add Parametric Relation (A : Type) : A (@eq A)
  [reflexivity proved by @refl_equal A]
...
```

Some tactics (*reflexivity*, *symmetry*, *transitivity*) work only on relations that respect the expected properties. The remaining tactics (*replace*, *rewrite* and derived tactics such as *autorewrite*) do not require any properties over the relation. However, they are able to replace terms with related ones only in contexts that are syntactic compositions of parametric morphism instances declared with the following command.

Command:

Add Parametric Morphism *binder** **:** *one_term* **with signature** *term* **as** *ident*

Declares a parametric morphism *one_term* of signature *term*. The final identifier *ident* gives a unique name to the morphism and it is used as the base name of the typeclass instance definition and as the name of the lemma that proves the well-definedness of the morphism. The parameters of the morphism as well as the signature may refer to the context of variables. The command asks the user to prove interactively that the function denoted by the first *ident* respects the relations identified from the signature.

Example

We start the example by assuming a small theory over homogeneous sets and we declare set equality as a parametric equivalence relation and union of two sets as a parametric morphism.

```
Require Export Setoid.
Require Export Relation_Definitions.

Set Implicit Arguments.

Parameter set : Type -> Type.
Parameter empty : forall A, set A.
Parameter eq_set : forall A, set A -> set A -> Prop.
Parameter union : forall A, set A -> set A -> set A.

Axiom eq_set_refl : forall A, reflexive _ (eq_set (A:=A)).
Axiom eq_set_sym : forall A, symmetric _ (eq_set (A:=A)).
Axiom eq_set_trans : forall A, transitive _ (eq_set (A:=A)).
Axiom empty_neutral : forall A (S : set A), eq_set (union S (empty A)) S.

Axiom union_compat :
  forall (A : Type),
    forall x x' : set A, eq_set x x' ->
    forall y y' : set A, eq_set y y' ->
      eq_set (union x y) (union x' y').

Add Parametric Relation A : (set A) (@eq_set A)
  reflexivity proved by (eq_set_refl (A:=A))
  symmetry proved by (eq_set_sym (A:=A))
  transitivity proved by (eq_set_trans (A:=A))
  as eq_set_rel.

Add Parametric Morphism A : (@union A)
  with signature (@eq_set A) ==> (@eq_set A) ==> (@eq_set A) as union_mor.
```

(continues on next page)

(continued from previous page)

```
Proof.
  exact (@union_compat A).
Qed.
```

It is possible to reduce the burden of specifying parameters using (maximally inserted) implicit arguments. If `A` is always set as maximally implicit in the previous example, one can write:

```
Add Parametric Relation A : (set A) eq_set
  reflexivity proved by eq_set_refl
  symmetry proved by eq_set_sym
  transitivity proved by eq_set_trans
  as eq_set_rel.

Add Parametric Morphism A : (@union A) with
  signature eq_set ==> eq_set ==> eq_set as union_mor.
Proof. exact (@union_compat A). Qed.
```

We proceed now by proving a simple lemma performing a rewrite step and then applying reflexivity, as we would do working with Leibniz equality. Both tactic applications are accepted since the required properties over `eq_set` and `union` can be established from the two declarations above.

```
Goal forall (S : set nat),
  eq_set (union (union S (empty nat)) S) (union S S).

Proof. intros. rewrite empty_neutral. reflexivity. Qed.
```

The tables of relations and morphisms are managed by the typeclass instance mechanism. The behavior on section close is to generalize the instances by the variables of the section (and possibly hypotheses used in the proofs of instance declarations) but not to export them in the rest of the development for proof search. One can use the cmd:`Existing Instance` command to do so outside the section, using the name of the declared morphism suffixed by `_Morphism`, or use the `Global` modifier for the corresponding class instance declaration (see *First Class Setoids and Morphisms*) at definition time. When loading a compiled file or importing a module, all the declarations of this module will be loaded.

Rewriting and non reflexive relations

To replace only one argument of an n-ary morphism it is necessary to prove that all the other arguments are related to themselves by the respective relation instances.

Example

To replace `(union S empty)` with `S` in `(union (union S empty) S) (union S S)` the rewrite tactic must exploit the monotony of `union` (axiom `union_compat` in the previous example). Applying `union_compat` by hand we are left with the goal `eq_set (union S S) (union S S)`.

When the relations associated with some arguments are not reflexive, the tactic cannot automatically prove the reflexivity goals, that are left to the user.

Setoids whose relations are partial equivalence relations (PER) are useful for dealing with partial functions. Let `R` be a PER. We say that an element `x` is defined if `R x x`. A partial function whose domain comprises all the defined elements is declared as a morphism that respects `R`. Every time a rewriting step is performed the user must prove that the argument of the morphism is defined.

Example

Let `eq0` be `fun x y => x = y /\ x <> 0` (the smallest PER over nonzero elements). Division can be declared as a morphism of signature `eq ==> eq0 ==> eq`. Replacing `x` with `y` in `div x n = div y n` opens an additional goal `eq0 n n` which is equivalent to `n = n /\ n <> 0`.

Rewriting and non symmetric relations

When the user works up to relations that are not symmetric, it is no longer the case that any covariant morphism argument is also contravariant. As a result it is no longer possible to replace a term with a related one in every context, since the obtained goal implies the previous one if and only if the replacement has been performed in a contravariant position. In a similar way, replacement in an hypothesis can be performed only if the replaced term occurs in a covariant position.

Example: Covariance and contravariance

Suppose that division over real numbers has been defined as a morphism of signature `Z.div : Z.lt ++> Z.lt --> Z.lt` (i.e. `Z.div` is increasing in its first argument, but decreasing on the second one). Let < denote `Z.lt`. Under the hypothesis `H : x < y` we have `k < x / y -> k < x / x`, but not `k < y / x -> k < x / x`. Dually, under the same hypothesis `k < x / y -> k < y / y` holds, but `k < y / x -> k < y / y` does not. Thus, if the current goal is `k < x / x`, it is possible to replace only the second occurrence of `x` (in contravariant position) with `y` since the obtained goal must imply the current one. On the contrary, if `k < x / x` is an hypothesis, it is possible to replace only the first occurrence of `x` (in covariant position) with `y` since the current hypothesis must imply the obtained one.

Contrary to the previous implementation, no specific error message will be raised when trying to replace a term that occurs in the wrong position. It will only fail because the rewriting constraints are not satisfiable. However it is possible to use the at modifier to specify which occurrences should be rewritten.

As expected, composing morphisms together propagates the variance annotations by switching the variance every time a contravariant position is traversed.

Example

Let us continue the previous example and let us consider the goal `x / (x / x) < k`. The first and third occurrences of `x` are in a contravariant position, while the second one is in covariant position. More in detail, the second occurrence of `x` occurs covariantly in `(x / x)` (since division is covariant in its first argument), and thus contravariantly in `x / (x / x)` (since division is contravariant in its second argument), and finally covariantly in `x / (x / x) < k` (since <, as every transitive relation, is contravariant in its first argument with respect to the relation itself).

Rewriting in ambiguous setoid contexts

One function can respect several different relations and thus it can be declared as a morphism having multiple signatures.

Example

Union over homogeneous lists can be given all the following signatures: `eq ==> eq ==> eq` (`eq` being the equality over ordered lists) `set_eq ==> set_eq ==> set_eq` (`set_eq` being the equality over unordered lists up to

duplicates), `multiset_eq ==> multiset_eq ==> multiset_eq` (`multiset_eq` being the equality over unordered lists).

To declare multiple signatures for a morphism, repeat the *Add Morphism* command.

When morphisms have multiple signatures it can be the case that a rewrite request is ambiguous, since it is unclear what relations should be used to perform the rewriting. Contrary to the previous implementation, the tactic will always choose the first possible solution to the set of constraints generated by a rewrite and will not try to find *all* the possible solutions to warn the user about them.

Commands and tactics

First class setoids and morphisms

The implementation is based on a first-class representation of properties of relations and morphisms as typeclasses. That is, the various combinations of properties on relations and morphisms are represented as records and instances of these classes are put in a hint database. For example, the declaration:

```
Add Parametric Relation (x1 : T1) ... (xn : Tn) : (A t1 ... tn) (Aeq t'1 ... t'm)
  [reflexivity proved by refl]
  [symmetry proved by sym]
  [transitivity proved by trans]
  as id.
```

is equivalent to an instance declaration:

```
Instance (x1 : T1) ... (xn : Tn) => id : @Equivalence (A t1 ... tn) (Aeq t'1 ... t'm)␣
↪:=
  [Equivalence_Reflexive := refl]
  [Equivalence_Symmetric := sym]
  [Equivalence_Transitive := trans].
```

The declaration itself amounts to the definition of an object of the record type `Coq.Classes.RelationClasses.Equivalence` and a hint added to the `typeclass_instances` hint database. Morphism declarations are also instances of a typeclass defined in `Classes.Morphisms`. See the documentation on *Typeclasses* and the theories files in Classes for further explanations.

One can inform the rewrite tactic about morphisms and relations just by using the typeclass mechanism to declare them using the *Instance* and *Context* commands. Any object of type Proper (the type of morphism declarations) in the local context will also be automatically used by the rewriting tactic to solve constraints.

Other representations of first class setoids and morphisms can also be handled by encoding them as records. In the following example, the projections of the setoid relation and of the morphism function can be registered as parametric relations and morphisms.

Example: First class setoids

```
Require Import Relation_Definitions Setoid.

Record Setoid : Type :=
{ car: Type;
  eq: car -> car -> Prop;
  refl: reflexive _ eq;
  sym: symmetric _ eq;
  trans: transitive _ eq
```

(continues on next page)

(continued from previous page)

```
}.

Add Parametric Relation (s : Setoid) : (@car s) (@eq s)
  reflexivity proved by (refl s)
  symmetry proved by (sym s)
  transitivity proved by (trans s) as eq_rel.

Record Morphism (S1 S2 : Setoid) : Type :=
{ f: car S1 -> car S2;
  compat: forall (x1 x2 : car S1), eq S1 x1 x2 -> eq S2 (f x1) (f x2)
}.

Add Parametric Morphism (S1 S2 : Setoid) (M : Morphism S1 S2) :
  (@f S1 S2 M) with signature (@eq S1 ==> @eq S2) as apply_mor.
Proof. apply (compat S1 S2 M). Qed.

Lemma test : forall (S1 S2 : Setoid) (m : Morphism S1 S2)
  (x y : car S1), eq S1 x y -> eq S2 (f _ _ m x) (f _ _ m y).
Proof. intros. rewrite H. reflexivity. Qed.
```

Tactics enabled on user provided relations

The following tactics, all prefixed by `setoid_`, deal with arbitrary registered relations and morphisms. Moreover, all the corresponding unprefixed tactics (i.e. *reflexivity*, *symmetry*, *transitivity*, *replace*, *rewrite*) have been extended to fall back to their prefixed counterparts when the relation involved is not Leibniz equality. Notice, however, that using the prefixed tactics it is possible to pass additional arguments such as `using relation`.

Tactic: setoid_reflexivity

Tactic: setoid_symmetry [**in** *ident*]?
Tactic: setoid_transitivity *one_term*
Tactic:
setoid_rewrite [**->** | **<-**]? *one_term* [**with** *bindings*]? [**at** *rewrite_occs*]? [**in** *ident*]?
Tactic:
setoid_rewrite [**->** | **<-**]? *one_term* [**with** *bindings*]? **in** *ident* **at** *rewrite_occs*
Tactic:
setoid_replace *one_term* **with** *one_term* [**using relation** *one_term*]? [**in** *ident*]? [**at** *int_or_var*

rewrite_occs ::= [*integer*]+
| *ident*

The `using relation` arguments cannot be passed to the unprefixed form. The latter argument tells the tactic what parametric relation should be used to replace the first tactic argument with the second one. If omitted, it defaults to the `DefaultRelation` instance on the type of the objects. By default, it means the most recent `Equivalence` instance in the global environment, but it can be customized by declaring new `DefaultRelation` instances. As Leibniz equality is a declared equivalence, it will fall back to it if no other relation is declared on a given type.

Every derived tactic that is based on the unprefixed forms of the tactics considered above will also work up to user defined relations. For instance, it is possible to register hints for *autorewrite* that are not proofs of Leibniz equalities. In

particular it is possible to exploit *autorewrite* to simulate normalization in a term rewriting system up to user defined equalities.

Printing relations and morphisms

Use the *Print Instances* command with the class names Reflexive, Symmetric or Transitive to print registered reflexive, symmetric or transitive relations and with the class name Proper to print morphisms.

When rewriting tactics refuse to replace a term in a context because the latter is not a composition of morphisms, this command can be useful to understand what additional morphisms should be registered.

Deprecated syntax and backward incompatibilities

Command: Add Setoid *one_term*$_{carrier}$ *one_term*$_{congruence}$ *one_term*$_{proofs}$ **as** *ident*

This command for declaring setoids and morphisms is also accepted due to backward compatibility reasons.

Here *one_term*$_{congruence}$ is a congruence relation without parameters, *one_term*$_{carrier}$ is its carrier and *one_term*$_{proofs}$ is an object of type (**Setoid_Theory** *one_term*$_{carrier}$ *one_term*$_{congruence}$) (i.e. a record packing together the reflexivity, symmetry and transitivity lemmas). Notice that the syntax is not completely backward compatible since the identifier was not required.

Command: Add Morphism *one_term* **:** *ident*

Command: Add Morphism *one_term* **with signature** *term* **as** *ident*

This command is restricted to the declaration of morphisms without parameters. It is not fully backward compatible since the property the user is asked to prove is slightly different: for n-ary morphisms the hypotheses of the property are permuted; moreover, when the morphism returns a proposition, the property is now stated using a bi-implication in place of a simple implication. In practice, porting an old development to the new semantics is usually quite simple.

Command: Declare Morphism *one_term* **:** *ident*

Declares a parameter in a module type that is a morphism.

Notice that several limitations of the old implementation have been lifted. In particular, it is now possible to declare several relations with the same carrier and several signatures for the same morphism. Moreover, it is now also possible to declare several morphisms having the same signature. Finally, the *replace* and *rewrite* tactics can be used to replace terms in contexts that were refused by the old implementation. As discussed in the next section, the semantics of the new *setoid_rewrite* tactic differs slightly from the old one and *rewrite*.

Extensions

Rewriting under binders

Warning: Due to compatibility issues, this feature is enabled only when calling the *setoid_rewrite* tactic directly and not *rewrite*.

To be able to rewrite under binding constructs, one must declare morphisms with respect to pointwise (setoid) equivalence of functions. Example of such morphisms are the standard all and ex combinators for universal and existential quantification respectively. They are declared as morphisms in the Classes.Morphisms_Prop module. For example, to declare that universal quantification is a morphism for logical equivalence:

```
Instance all_iff_morphism (A : Type) :
         Proper (pointwise_relation A iff ==> iff) (@all A).
```

```
Proof. simpl_relation.
    1 goal

      A : Type
      x, y : A -> Prop
      H : pointwise_relation A iff x y
      ============================
      all x <-> all y
```

One then has to show that if two predicates are equivalent at every point, their universal quantifications are equivalent. Once we have declared such a morphism, it will be used by the setoid rewriting tactic each time we try to rewrite under an `all` application (products in `Prop` are implicitly translated to such applications).

Indeed, when rewriting under a lambda, binding variable `x`, say from `P x` to `Q x` using the relation iff, the tactic will generate a proof of `pointwise_relation A iff (fun x => P x) (fun x => Q x)` from the proof of `iff (P x) (Q x)` and a constraint of the form `Proper (pointwise_relation A iff ==> ?) m` will be generated for the surrounding morphism `m`.

Hence, one can add higher-order combinators as morphisms by providing signatures using pointwise extension for the relations on the functional arguments (or whatever subrelation of the pointwise extension). For example, one could declare the `map` combinator on lists as a morphism:

```
Instance map_morphism `{Equivalence A eqA, Equivalence B eqB} :
         Proper ((eqA ==> eqB) ==> list_equiv eqA ==> list_equiv eqB) (@map A B).
```

where `list_equiv` implements an equivalence on lists parameterized by an equivalence on the elements.

Note that when one does rewriting with a lemma under a binder using *`setoid_rewrite`*, the application of the lemma may capture the bound variable, as the semantics are different from rewrite where the lemma is first matched on the whole term. With the new *`setoid_rewrite`*, matching is done on each subterm separately and in its local context, and all matches are rewritten *simultaneously* by default. The semantics of the previous *`setoid_rewrite`* implementation can almost be recovered using the `at 1` modifier.

Subrelations

Subrelations can be used to specify that one relation is included in another, so that morphism signatures for one can be used for the other. If a signature mentions a relation `R` on the left of an arrow `==>`, then the signature also applies for any relation `S` that is smaller than `R`, and the inverse applies on the right of an arrow. One can then declare only a few morphisms instances that generate the complete set of signatures for a particular *constant*. By default, the only declared subrelation is `iff`, which is a subrelation of `impl` and `inverse impl` (the dual of implication). That's why we can declare only two morphisms for conjunction: `Proper (impl ==> impl ==> impl) and` and `Proper (iff ==> iff ==> iff) and`. This is sufficient to satisfy any rewriting constraints arising from a rewrite using `iff`, `impl` or `inverse impl` through `and`.

Subrelations are implemented in `Classes.Morphisms` and are a prime example of a mostly user-space extension of the algorithm.

Constant unfolding

The resolution tactic is based on typeclasses and hence regards user-defined *constants* as transparent by default. This may slow down the resolution due to a lot of unifications (all the declared `Proper` instances are tried at each node of the search tree). To speed it up, declare your constant as rigid for proof search using the command *`Typeclasses Opaque`*.

Strategies for rewriting

Usage

Tactic: **`rewrite_strat`** ***`rewstrategy`*** **`in`** ***`ident`*** **`?`**

Rewrite using ***`rewstrategy`*** in the conclusion or in the hypothesis ***`ident`***.

Error: **`Nothing to rewrite.`**

The strategy didn't find any matches.

Error: **`No progress made.`**

If the strategy succeeded but made no progress.

Error: **`Unable to satisfy the rewriting constraints.`**

If the strategy succeeded and made progress but the corresponding rewriting constraints are not satisfied.

`setoid_rewrite` ***`one_term`*** is basically equivalent to **`rewrite_strat outermost`** ***`one_term`***.

Definitions

The generalized rewriting tactic is based on a set of strategies that can be combined to create custom rewriting procedures. Its set of strategies is based on the programmable rewriting strategies with generic traversals by Visser et al. [LV97] [VBT98], which formed the core of the Stratego transformation language [Vis01]. Rewriting strategies are applied using the tactic **`rewrite_strat`** ***`rewstrategy`***.

rewstrategy ::= *one_term*
| **<-** *one_term*
| **fail**
| **id**
| **refl**
| **progress** *rewstrategy*
| **try** *rewstrategy*
| *rewstrategy* **;** *rewstrategy*
| **choice** *rewstrategy rewstrategy*
| **repeat** *rewstrategy*
| **any** *rewstrategy*
| **subterm** *rewstrategy*
| **subterms** *rewstrategy*
| **innermost** *rewstrategy*
| **outermost** *rewstrategy*
| **bottomup** *rewstrategy*
| **topdown** *rewstrategy*
| **hints** *ident*
| **terms** *one_term* *
| **eval** *red_expr*
| **fold** *one_term*
| **(** *rewstrategy* **)**
| **old_hints** *ident*

one_term lemma, left to right

<- ***one_term*** lemma, right to left

fail failure

id identity

refl reflexivity

progress ***rewstrategy*** progress

try ***rewstrategy*** try catch

rewstrategy **;** ***rewstrategy*** composition

choice ***rewstrategy*** ***rewstrategy*** left_biased_choice

repeat ***rewstrategy*** one or more

any ***rewstrategy*** zero or more

subterm ***rewstrategy*** one subterm

subterms ***rewstrategy*** all subterms

innermost ***rewstrategy*** Innermost first. When there are multiple nested matches in a subterm, the innermost subterm is rewritten. For *example*, rewriting **(a + b) + c** with Nat.add_comm gives **(b + a) + c**.

outermost ***rewstrategy*** Outermost first. When there are multiple nested matches in a subterm, the outermost subterm is rewritten. For *example*, rewriting **(a + b) + c** with Nat.add_comm gives **c + (a + b)**.

bottomup ***rewstrategy*** bottom-up

topdown ***rewstrategy*** top-down

hints ***ident*** apply hints from hint database

terms *one_term*[*] any of the terms

eval *red_expr* apply reduction

fold *term* unify

(*rewstrategy* **)** to be documented

old_hints *ident* to be documented

Conceptually, a few of these are defined in terms of the others using a primitive fixpoint operator `fix`, which the tactic doesn't currently support:

- **try** *rewstrategy* **:= choice** *rewstrategy* **id**
- **any** *rewstrategy* **:= fix** *ident*. **try (***rewstrategy* **;** *ident***)**
- **repeat** *rewstrategy* **:=** *rewstrategy*; **any** *rewstrategy*
- **bottomup** *rewstrategy* **:= fix** *ident*. **(choice (progress (subterms** *ident***))** *rewstrategy***) ; try** *ident*
- **topdown** *rewstrategy* **:= fix** *ident*. **(choice** *rewstrategy* **(progress (subterms** *ident***))) ; try** *ident*
- **innermost** *rewstrategy* **:= fix** *ident*. **(choice (subterm** *ident***)** *rewstrategy***)**
- **outermost** *rewstrategy* **:= fix** *ident*. **(choice** *rewstrategy* **(subterm** *ident***))**

The basic control strategy semantics are straightforward: strategies are applied to subterms of the term to rewrite, starting from the root of the term. The lemma strategies unify the left-hand-side of the lemma with the current subterm and on success rewrite it to the right- hand-side. Composition can be used to continue rewriting on the current subterm. The `fail` strategy always fails while the identity strategy succeeds without making progress. The reflexivity strategy succeeds, making progress using a reflexivity proof of rewriting. `progress` tests progress of the argument ***rewstrategy*** and fails if no progress was made, while `try` always succeeds, catching failures. `choice` is left-biased: it will launch the first strategy and fall back on the second one in case of failure. One can iterate a strategy at least 1 time using `repeat` and at least 0 times using `any`.

The `subterm` and `subterms` strategies apply their argument ***rewstrategy*** to respectively one or all subterms of the current term under consideration, left-to-right. `subterm` stops at the first subterm for which ***rewstrategy*** made progress. The composite strategies `innermost` and `outermost` perform a single innermost or outermost rewrite using their argument ***rewstrategy***. Their counterparts `bottomup` and `topdown` perform as many rewritings as possible, starting from the bottom or the top of the term.

Hint databases created for *autorewrite* can also be used by *rewrite_strat* using the `hints` strategy that applies any of the lemmas at the current subterm. The `terms` strategy takes the lemma names directly as arguments. The `eval` strategy expects a reduction expression (see *Applying conversion rules*) and succeeds if it reduces the subterm under consideration. The `fold` strategy takes a *term* and tries to *unify* it to the current subterm, converting it to *term* on success. It is stronger than the tactic `fold`.

Example: `innermost` and `outermost`

The type of `Nat.add_comm` is `forall n m : nat, n + m = m + n`.

```
Require Import Coq.Arith.Arith.
    [Loading ML file ring_plugin.cmxs ... done]

Set Printing Parentheses.
Goal forall a b c: nat, a + b + c = 0.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      ============================
      forall a b c : nat, ((a + b) + c) = 0

rewrite_strat innermost Nat.add_comm.
    1 goal

      ============================
      forall a b c : nat, ((b + a) + c) = 0
```

Using **outermost** instead gives this result:

```
rewrite_strat outermost Nat.add_comm.
    1 goal

      ============================
      forall a b c : nat, (c + (a + b)) = 0
```

3.3 Creating new tactics

The languages presented in this chapter allow one to build complex tactics by combining existing ones with constructs such as conditionals and looping. While *Ltac* was initially thought of as a language for doing some basic combinations, it has been used successfully to build highly complex tactics as well, but this has also highlighted its limits and fragility. The experimental language *Ltac2* is a typed and more principled variant which is more adapted to building complex tactics.

There are other solutions beyond these two tactic languages to write new tactics:

- Mtac2[50] is an external plugin which provides another typed tactic language. While Ltac2 belongs to the ML language family, Mtac2 reuses the language of Coq itself as the language to build Coq tactics.
- The most traditional way of building new complex tactics is to write a Coq plugin in OCaml. Beware that this also requires much more effort and commitment. A tutorial for writing Coq plugins is available in the Coq repository in doc/plugin_tutorial[51].

3.3.1 Ltac

This chapter documents the tactic language $\mathtt{L}_{\mathrm{tac}}$.

We start by giving the syntax followed by the informal semantics. To learn more about the language and especially about its foundations, please refer to [Del00]. (Note the examples in the paper won't work as-is; Coq has evolved since the paper was written.)

Example: Basic tactic macros

Here are some examples of simple tactic macros you can create with $\mathtt{L}_{\mathrm{tac}}$:

```
Ltac reduce_and_try_to_solve := simpl; intros; auto.

Ltac destruct_bool_and_rewrite b H1 H2 :=
  destruct b; [ rewrite H1; eauto | rewrite H2; eauto ].
```

[50] https://github.com/Mtac2/Mtac2
[51] https://github.com/coq/coq/tree/master/doc/plugin_tutorial

See Section *Examples of using Ltac* for more advanced examples.

Syntax

The syntax of the tactic language is given below.

The main entry of the grammar is ***ltac_expr***, which is used in proof mode as well as to define new tactics with the *Ltac* command.

The grammar uses multiple **ltac_expr*** nonterminals to express how subexpressions are grouped when they're not fully parenthesized. For example, in many programming languages, `a*b+c` is interpreted as `(a*b)+c` because `*` has higher precedence than `+`. Usually `a/b/c` is given the left associative interpretation `(a/b)/c` rather than the right associative interpretation `a/(b/c)`.

In Coq, the expression **try repeat** ***tactic$_1$*** **||** ***tactic$_2$*****;** ***tactic$_3$*****;** ***tactic$_4$*** is interpreted as **(try (repeat (***tactic$_1$* **||** *tactic$_2$***)); ***tactic$_3$***); ***tactic$_4$*** because `||` is part of *ltac_expr2*, which has higher precedence than *try* and *repeat* (at the level of *ltac_expr3*), which in turn have higher precedence than **;**, which is part of *ltac_expr4*. (A *lower* number in the nonterminal name means *higher* precedence in this grammar.)

The constructs in *ltac_expr* are *left associative*.

ltac_expr	::=	*ltac_expr4* | *binder_tactic*
ltac_expr4	::=	*ltac_expr3* **;** *ltac_expr3* | *binder_tactic*
	|	*ltac_expr3* **;** [*for_each_goal*]
	|	*ltac_expr3*
ltac_expr3	::=	*l3_tactic*
	|	*ltac_expr2*
ltac_expr2	::=	*ltac_expr1* **+** *ltac_expr2* | *binder_tactic*
	|	*ltac_expr1* **||** *ltac_expr2* | *binder_tactic*
	|	*l2_tactic*
	|	*ltac_expr1*
ltac_expr1	::=	*tactic_value*
	|	*qualid* *tactic_arg*+
	|	*l1_tactic*
	|	*ltac_expr0*
tactic_value	::=	*value_tactic* | *syn_value*
tactic_arg	::=	*tactic_value*
	|	*term*
	|	()
ltac_expr0	::=	(*ltac_expr*)
	|	[> *for_each_goal*]
	|	*tactic_atom*
tactic_atom	::=	*integer*
	|	*qualid*
	|	()

Note: Tactics described in other chapters of the documentation are `simple_tactics`, which only modify the proof state. L_{tac} provides additional constructs that can generally be used wherever a *simple_tactic* can appear, even though they don't modify the proof state and that syntactically they're at varying levels in *ltac_expr*. For simplicity of presentation, the L_{tac} constructs are documented as tactics. Tactics are grouped as follows:

- `binder_tactics` are: *fun* and *let*

- `l3_tactics` include $\mathrm{L}_{\mathrm{tac}}$ tactics: *try*, *do*, *repeat*, *timeout*, *time*, *progress*, *once*, *exactly_once*, *only* and *abstract*
- `l2_tactics` are: *tryif*
- `l1_tactics` are the *simple_tactic*s, *first*, *solve*, *idtac*, *fail* and *gfail* as well as *match*, *match goal* and their **lazymatch** and **multimatch** variants.
- `value_tactics`, which return values rather than change the proof state. They are: *eval*, *context*, *numgoals*, *fresh*, *type of* and *type_term*.

The documentation for these $\mathrm{L}_{\mathrm{tac}}$ constructs mentions which group they belong to.

The difference is only relevant in some compound tactics where extra parentheses may be needed. For example, parenthesees are required in **idtac + (once idtac)** because *once* is an *l3_tactic*, which the production ***ltac_expr2*** **::=** ***ltac_expr1*** **+** | ***ltac_expr2*** | ***binder_tactic*** | doesn't accept after the +.

Note:

- The grammar reserves the token `||`.

Semantics

Types of values

An $\mathrm{L}_{\mathrm{tac}}$ value can be a tactic, integer, string, unit (written as "`()`") or syntactic value. Syntactic values correspond to certain nonterminal symbols in the grammar, each of which is a distinct type of value. Most commonly, the value of an $\mathrm{L}_{\mathrm{tac}}$ expression is a tactic that can be executed.

While there are a number of constructs that let you combine multiple tactics into compound tactics, there are no operations for combining most other types of values. For example, there's no function to add two integers. Syntactic values are entered with the *syn_value* construct. Values of all types can be assigned to toplevel symbols with the *Ltac* command or to local symbols with the *let* tactic. $\mathrm{L}_{\mathrm{tac}}$ *functions* can return values of any type.

Syntactic values

syn_value **::=** ***ident*** **:** **(** ***nonterminal*** **)**

Provides a way to use the syntax and semantics of a grammar nonterminal as a value in an *ltac_expr*. The table below describes the most useful of these. You can see the others by running "*Print Grammar* `tactic`" and examining the part at the end under "Entry tactic:tactic_value".

ident name of a grammar nonterminal listed in the table

nonterminal represents syntax described by *nonterminal*.

Specified *ident*	Parsed as	Interpreted as	as in tactic
ident	*ident*	a user-specified name	*intro*
string	*string*	a string	
integer	*integer*	an integer	
reference	*qualid*	a qualified identifier	
uconstr	*term*	an untyped term	*refine*
constr	*term*	a term	*exact*
ltac	*ltac_expr*	a tactic	

ltac:(*ltac_expr*) can be used to indicate that the parenthesized item should be interpreted as a tactic and not as a term. The constructs can also be used to pass parameters to tactics written in OCaml. (While all of the *syn_value*s can appear at the beginning of an *ltac_expr*, the others are not useful because they will not evaluate to tactics.)

uconstr:(*term*) can be used to build untyped terms. Terms built in $\mathrm{L}_{\mathrm{tac}}$ are well-typed by default. Building large terms in recursive $\mathrm{L}_{\mathrm{tac}}$ functions may give very slow behavior because terms must be fully type checked at each step. In this case, using an untyped term may avoid most of the repetitive type checking for the term, improving performance.

Untyped terms built using **uconstr:(…)** can be used as arguments to the *refine* tactic, for example. In that case the untyped term is type checked against the conclusion of the goal, and the holes which are not solved by the typing procedure are turned into new subgoals.

Tactics in terms

term_ltac ::= **ltac :** (*ltac_expr*)

Allows including an *ltac_expr* within a term. Semantically, it's the same as the *syn_value* for ltac, but these are distinct in the grammar.

Substitution

*name*s within $\mathrm{L}_{\mathrm{tac}}$ expressions are used to represent both terms and $\mathrm{L}_{\mathrm{tac}}$ variables. If the *name* corresponds to an $\mathrm{L}_{\mathrm{tac}}$ variable or tactic name, $\mathrm{L}_{\mathrm{tac}}$ substitutes the value before applying the expression. Generally it's best to choose distinctive names for $\mathrm{L}_{\mathrm{tac}}$ variables that won't clash with term names. You can use **ltac:(*name*)** or **(*name*)** to control whether a *name* is interpreted as, respectively, an $\mathrm{L}_{\mathrm{tac}}$ variable or a term.

Note that values from toplevel symbols, unlike locally-defined symbols, are substituted only when they appear at the beginning of an *ltac_expr* or as a *tactic_arg*. Local symbols are also substituted into tactics:

Example: Substitution of global and local symbols

```
Ltac n := 1.
    n is defined

let n2 := n in idtac n2.
    1

Fail idtac n.
    The command has indeed failed with message:
    n not found.
```

Sequence: ;

A sequence is an expression of the following form:

Tactic: *ltac_expr3*$_1$ **;** *ltac_expr3*$_2$ | *binder_tactic*

The expression *ltac_expr3*$_1$ is evaluated to $\mathbf{v_1}$, which must be a tactic value. The tactic $\mathbf{v_1}$ is applied to the current goals, possibly producing more goals. Then the right-hand side is evaluated to produce $\mathbf{v_2}$, which must be a tactic value. The tactic $\mathbf{v_2}$ is applied to all the goals produced by the prior application. Sequence is associative.

Note:

- If you want *tactic*$_2$**;** *tactic*$_3$ to be fully applied to the first subgoal generated by *tactic*$_1$ before applying it to the other subgoals, then you should write:
  - *tactic*$_1$**; [>** *tactic*$_2$**;** *tactic*$_3$ **..]** rather than
  - *tactic*$_1$**; (**_tactic_$_2$**;** _tactic_$_3$**)**.

Local application of tactics: [> ...]

Tactic: **[>** *for_each_goal* **]**

for_each_goal ::= *goal_tactics*

| *goal_tactics* | ? *ltac_expr* ? .. | *goal_tactics* ?

goal_tactics ::= (*ltac_expr* ? |)*

Applies a different *ltac_expr*? to each of the focused goals. In the first form of *for_each_goal* (without **..**), the construct fails if the number of specified *ltac_expr*? is not the same as the number of focused goals. Omitting an *ltac_expr* leaves the corresponding goal unchanged.

In the second form (with *ltac_expr*? **..**), the left and right *goal_tactics* are applied respectively to a prefix or suffix of the list of focused goals. The *ltac_expr*? before the **..** is applied to any focused goals in the middle (possibly none) that are not covered by the *goal_tactics*. The number of *ltac_expr*? in the *goal_tactics* must be no more than the number of focused goals.

In particular:

goal_tactics **|** *goal_tactics* **..** **|** *goal_tactics* The goals not covered by the two *goal_tactics* are left unchanged.

[> *ltac_expr* **..** **]** *ltac_expr* is applied independently to each of the goals, rather than globally. In particular, if there are no goals, the tactic is not run at all. A tactic which expects multiple goals, such as *swap*, would act as if a single goal is focused.

Note that *ltac_expr3* **;** **[** (*ltac_expr* |)* **]** is a convenient idiom to process the goals generated by applying *ltac_expr3*.

Tactic: *ltac_expr3* **;** **[** *for_each_goal* **]**

ltac_expr3 **;** **[** **...** **]** is equivalent to **[>** *ltac_expr3* **;** **[>** **...** **]** **..** **]**.

Goal selectors

By default, tactic expressions are applied only to the first goal. Goal selectors provide a way to apply a tactic expression to another goal or multiple goals. (The *Default Goal Selector* option can be used to change the default behavior.)

Tactic: ***toplevel_selector*** **:** ***ltac_expr***

toplevel_selector	::=	*selector*
	|	**all**
	|	**!**
	|	**par**

Reorders the goals and applies *ltac_expr* to the selected goals. It can only be used at the top level of a tactic expression; it cannot be used within a tactic expression. The selected goals are reordered so they appear after the lowest-numbered selected goal, ordered by goal number. *Example*. If the selector applies to a single goal or to all goals, the reordering will not be apparent. The order of the goals in the *selector* is irrelevant. (This may not be what you expect; see #8481[52].)

all Selects all focused goals.

! If exactly one goal is in focus, apply *ltac_expr* to it. Otherwise the tactic fails.

par Applies ***ltac_expr*** to all focused goals in parallel. The number of workers can be controlled via the command line option **-async-proofs-tac-j** ***natural*** to specify the desired number of workers. Limitations: `par:` only works on goals that don't contain existential variables. ***ltac_expr*** must either solve the goal completely or do nothing (i.e. it cannot make some progress).

Selectors can also be used nested within a tactic expression with the *only* tactic:

Tactic: **only** ***selector*** **:** ***ltac_expr3***

selector	::=	*range_selector* (+ ,)
	|	**[** *ident* **]**
range_selector	::=	*natural* **-** *natural*
	|	*natural*

Applies *ltac_expr3* to the selected goals.

only is an *l3_tactic*.

range_selector (+ ,) The selected goals are the union of the specified *range_selector*s.

[***ident*** **]** Limits the application of *ltac_expr3* to the goal previously named *ident* by the user (see *Existential variables*).

$natural_1$ **-** $natural_2$ Selects the goals $natural_1$ through $natural_2$, inclusive.

natural Selects a single goal.

Error: **No such goal.**

Example: Selector reordering goals

```
Goal 1=0 /\ 2=0 /\ 3=0.
```

[52] https://github.com/coq/coq/issues/8481

```
repeat split.
    3 goals

      ============================
      1 = 0

    goal 2 is:
     2 = 0
    goal 3 is:
     3 = 0

1,3: idtac.
    3 goals

      ============================
      1 = 0

    goal 2 is:
     3 = 0
    goal 3 is:
     2 = 0
```

Processing multiple goals

When presented with multiple focused goals, most L_{tac} constructs process each goal separately. They succeed only if there is a success for each goal. For example:

Example: Multiple focused goals

This tactic fails because there no match for the second goal (`False`).

```
    2 goals

      ============================
      True

    goal 2 is:
     False

Fail all: let n := numgoals in idtac "numgoals =" n;
match goal with
| |- True => idtac
end.
    numgoals = 2
    The command has indeed failed with message:
    No matching clauses for match.
```

Do loop

Tactic: `do` *`nat_or_var ltac_expr3`*

The do loop repeats a tactic *nat_or_var* times:

ltac_expr is evaluated to v, which must be a tactic value. This tactic value v is applied *nat_or_var* times. If *nat_or_var* > 1, after the first application of v, v is applied, at least once, to the generated subgoals and so on. It fails if the application of v fails before *nat_or_var* applications have been completed.

do is an *l3_tactic*.

Repeat loop

Tactic: `repeat` *`ltac_expr3`*

The repeat loop repeats a tactic until it fails.

ltac_expr is evaluated to v. If v denotes a tactic, this tactic is applied to each focused goal independently. If the application succeeds, the tactic is applied recursively to all the generated subgoals until it eventually fails. The recursion stops in a subgoal when the tactic has failed *to make progress*. The tactic *repeat* ***ltac_expr*** itself never fails.

repeat is an *l3_tactic*.

Catching errors: try

We can catch the tactic errors with:

Tactic: `try` *`ltac_expr3`*

ltac_expr is evaluated to v which must be a tactic value. The tactic value v is applied to each focused goal independently. If the application of v fails in a goal, it catches the error and leaves the goal unchanged. If the level of the exception is positive, then the exception is re-raised with its level decremented.

try is an *l3_tactic*.

Detecting progress

We can check if a tactic made progress with:

Tactic: `progress` *`ltac_expr3`*

ltac_expr is evaluated to v which must be a tactic value. The tactic value v is applied to each focused subgoal independently. If the application of v to one of the focused subgoal produced subgoals equal to the initial goals (up to syntactical equality), then an error of level 0 is raised.

progress is an *l3_tactic*.

Error: `Failed to progress.`

Branching and backtracking

L_{tac} provides several branching tactics that permit trying multiple alternative tactics for a proof step. For example, *first*, which tries several alternatives and selects the first that succeeds, or *tryif*, which tests whether a given tactic would succeed or fail if it was applied and then, depending on the result, applies one of two alternative tactics. There are also looping constructs *do* and *repeat*. The order in which the subparts of these tactics are evaluated is generally similar to structured programming constructs in many languages.

The *+*, *multimatch* and *multimatch goal* tactics provide more complex capability. Rather than applying a single successful tactic, these tactics generate a series of successful tactic alternatives that are tried sequentially when subsequent tactics outside these constructs fail. For example:

Example: Backtracking

```
Fail multimatch True with
| True => idtac "branch 1"
| _ => idtac "branch 2"
end ;
idtac "branch A"; fail.
    branch 1
    branch A
    branch 2
    branch A
    The command has indeed failed with message:
    Tactic failure.
```

These constructs are evaluated using backtracking. Each creates a backtracking point. When a subsequent tactic fails, evaluation continues from the nearest prior backtracking point with the next successful alternative and repeats the tactics after the backtracking point. When a backtracking point has no more successful alternatives, evaluation continues from the next prior backtracking point. If there are no more prior backtracking points, the overall tactic fails.

Thus, backtracking tactics can have multiple successes. Non-backtracking constructs that appear after a backtracking point are reprocessed after backtracking, as in the example above, in which the *;* construct is reprocessed after backtracking. When a backtracking construct is within a non-backtracking construct, the latter uses the first success. Backtracking to a point within a non-backtracking construct won't change the branch that was selected by the non-backtracking construct.

The *once* tactic stops further backtracking to backtracking points within that tactic.

Branching with backtracking: +

We can branch with backtracking with the following structure:

Tactic: ***ltac_expr1*** **+** ***ltac_expr2*** | ***binder_tactic***

Evaluates and applies ***ltac_expr1*** to each focused goal independently. If it fails (i.e. there is no initial success), then evaluates and applies the right-hand side. If the right-hand side fails, the construct fails.

If **ltac_expr1** has an initial success and a subsequent tactic (outside the + construct) fails, L_{tac} backtracks and selects the next success for **ltac_expr1**. If there are no more successes, then + similarly evaluates and applies (and backtracks in) the right-hand side. To prevent evaluation of further alternatives after an initial success for a tactic, use *first* instead.

+ is left-associative.

In all cases, **(***$ltac\_expr_1$* **+** *$ltac\_expr_2$***);** *$ltac\_expr_3$* is equivalent to **(***$ltac\_expr_1$***;** *$ltac\_expr_3$***) + (***$ltac\_expr_2$***;** *$ltac\_expr_3$***)**.

Additionally, in most cases, **(*ltac_expr*$_1$ + *ltac_expr*$_2$) + *ltac_expr*$_3$** is equivalent to **(*ltac_expr*$_1$ + (*ltac_expr*$_2$ + *ltac_expr*$_3$))**. Here's an example where the behavior differs slightly:

```
Fail (fail 2 + idtac) + idtac.
    The command has indeed failed with message:
    Tactic failure.

Fail fail 2 + (idtac + idtac).
    The command has indeed failed with message:
    Tactic failure (level 1).
```

Example: Backtracking branching with +

In the first tactic, `idtac "2"` is not executed. In the second, the subsequent `fail` causes backtracking and the execution of `idtac "B"`.

```
idtac "1" + idtac "2".
    1

assert_fails ((idtac "A" + idtac "B"); fail).
    A
    B
```

First tactic to succeed

In some cases backtracking may be too expensive.

Tactic: first [*ltac_expr |]**

For each focused goal, independently apply the first *ltac_expr* that succeeds. The ***ltac_expr***s must evaluate to tactic values. Failures in tactics after the *first* won't cause backtracking. (To allow backtracking, use the *+* construct above instead.)

If the *first* contains a tactic that can backtrack, "success" means the first success of that tactic. Consider the following:

Example: Backtracking inside a non-backtracking construct

The *fail* doesn't trigger the second *idtac*:

```
assert_fails (first [ idtac "1" | idtac "2" ]; fail).
    1
```

This backtracks within `(idtac "1A" + idtac "1B" + fail)` but *first* won't consider the `idtac "2"` alternative:

```
assert_fails (first [ (idtac "1A" + idtac "1B" + fail) | idtac "2" ]; fail).
    1A
    1B
```

---

first is an *l1_tactic*.

Error: No applicable tactic.

Variant: `first` *ltac_expr*

This is an L_{tac} alias that gives a primitive access to the first tactical as an L_{tac} definition without going through a parsing rule. It expects to be given a list of tactics through a *Tactic Notation* command, permitting notations with the following form to be written:

Example

```
Tactic Notation "foo" tactic_list(tacs) := first tacs.
```

Solving

Selects and applies the first tactic that solves each goal (i.e. leaves no subgoal) in a series of alternative tactics:

Tactic: `solve [` *ltac_expr*$_i$ `*` `|` `]`

For each current subgoal: evaluates and applies each ***ltac_expr*** in order until one is found that solves the subgoal.

If any of the subgoals are not solved, then the overall *solve* fails.

Note: In *solve* and *first*, ***ltac_expr***s that don't evaluate to tactic values are ignored. So *solve* `[ () | 1 |` *constructor* `]` is equivalent to *solve* `[` *constructor* `]`. This may make it harder to debug scripts that inadvertently include non-tactic values.

---

solve is an *l1_tactic*.

Variant: `solve` *ltac_expr*

This is an L_{tac} alias that gives a primitive access to the *solve* tactic. See the *first* tactic for more information.

First tactic to make progress: ||

Yet another way of branching without backtracking is the following structure:

Tactic: *ltac_expr1* `||` *ltac_expr2* | *binder_tactic*

ltac_expr1* `||` *ltac_expr2 is equivalent to **`first [ progress` *ltac_expr1* `|` *ltac_expr2* `]`**, except that if it fails, it fails like ***ltac_expr2*****`.`** `` `|| `` is left-associative.

ltac_exprs that don't evaluate to tactic values are ignored. See the note at *solve*.

Conditional branching: tryif

Tactic: `tryif` *ltac_expr*$_{test}$ `then` *ltac_expr*$_{then}$ `else` *ltac_expr2*$_{else}$

For each focused goal, independently: Evaluate and apply ***ltac_expr***$_{test}$. If ***ltac_expr***$_{test}$ succeeds at least once, evaluate and apply ***ltac_expr***$_{then}$ to all the subgoals generated by ***ltac_expr***$_{test}$. Otherwise, evaluate and apply ***ltac_expr2***$_{else}$ to all the subgoals generated by ***ltac_expr***$_{test}$.

tryif is an *l2_tactic*.

Soft cut: once

Another way of restricting backtracking is to restrict a tactic to a single success:

Tactic: once *ltac_expr3*

ltac_expr3 is evaluated to v which must be a tactic value. The tactic value v is applied but only its first success is used. If v fails, *once ltac_expr3* fails like v. If v has at least one success, *once ltac_expr3* succeeds once, but cannot produce more successes.

once is an *l3_tactic*.

Checking for a single success: exactly_once

Coq provides an experimental way to check that a tactic has *exactly one* success:

Tactic: exactly_once *ltac_expr3*

ltac_expr3 is evaluated to v which must be a tactic value. The tactic value v is applied if it has at most one success. If v fails, *exactly_once ltac_expr3* fails like v. If v has a exactly one success, *exactly_once ltac_expr3* succeeds like v. If v has two or more successes, *exactly_once ltac_expr3* fails.

exactly_once is an *l3_tactic*.

> **Warning:** The experimental status of this tactic pertains to the fact if v has side effects, they may occur in an unpredictable way. Indeed, normally v would only be executed up to the first success until backtracking is needed, however *exactly_once* needs to look ahead to see whether a second success exists, and may run further effects immediately.

Error: This tactic has more than one success.

Checking for failure: assert_fails

Coq defines an $\mathrm{L}_{\mathrm{tac}}$ tactic in `Init.Tactics` to check that a tactic *fails*:

Tactic: assert_fails *ltac_expr3*

If *ltac_expr3* fails, the proof state is unchanged and no message is printed. If *ltac_expr3* unexpectedly has at least one success, the tactic performs a *gfail* **0**, printing the following message:

Error: Tactic failure: <tactic closure> succeeds.

Note: *assert_fails* and *assert_succeeds* work as described when *ltac_expr3* is a *simple_tactic*. In some more complex expressions, they may report an error from within *ltac_expr3* when they shouldn't. This is due to the order in which parts of the *ltac_expr3* are evaluated and executed. For example:

```
assert_fails match True with _ => fail end.
    Toplevel input, characters 0-43:
    > assert_fails match True with _ => fail end.
    > ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Error: Tactic failure.
```

should not show any message. The issue is that *assert_fails* is an $\mathrm{L}_{\mathrm{tac}}$-defined tactic. That makes it a function that's processed in the evaluation phase, causing the *match* to find its first success earlier. One workaround is to prefix *ltac_expr3* with "`idtac;`".

```
assert_fails (idtac; match True with _ => fail end).
```

Alternatively, substituting the *match* into the definition of *assert_fails* works as expected:

```
tryif (once match True with _ => fail end) then gfail 0 (* tac *) "succeeds" else␣
↪idtac.
```

Checking for success: assert_succeeds

Coq defines an L_{tac} tactic in `Init.Tactics` to check that a tactic has *at least one* success:

Tactic: assert_succeeds *ltac_expr3*

If *ltac_expr3* has at least one success, the proof state is unchanged and no message is printed. If *ltac_expr3* fails, the tactic performs a *gfail* **0**, printing the following message:

Error: Tactic failure: <tactic closure> fails.

Print/identity tactic: idtac

Tactic: idtac *ident* | *string* | *natural* *****

Leaves the proof unchanged and prints the given tokens. *String*s and *natural*s are printed literally. If *ident* is an L_{tac} variable, its contents are printed; if not, it is an error.

idtac is an *l1_tactic*.

Failing

Tactic: **fail** | **gfail** *nat_or_var* **?** *ident* | *string* | *natural* *****

fail is the always-failing tactic: it does not solve any goal. It is useful for defining other tactics since it can be caught by *try*, *repeat*, *match goal*, or the branching tacticals.

gfail fails even when used after **;** and there are no goals left. Similarly, *gfail* fails even when used after `all:` and there are no goals left.

fail and *gfail* are *l1_tactic*s.

See the example for a comparison of the two constructs.

Note that if Coq terms have to be printed as part of the failure, term construction always forces the tactic into the goals, meaning that if there are no goals when it is evaluated, a tactic call like *let* **x := H in** *fail* `0 x` will succeed.

nat_or_var The failure level. If no level is specified, it defaults to 0. The level is used by *try*, *repeat*, *match goal* and the branching tacticals. If 0, it makes *match goal* consider the next clause (backtracking). If nonzero, the current *match goal* block, *try*, *repeat*, or branching command is aborted and the level is decremented. In the case of **+**, a nonzero level skips the first backtrack point, even if the call to *fail* ***natural*** is not enclosed in a **+** construct, respecting the algebraic identity.

ident | *string* | *natural* ***** The given tokens are used for printing the failure message. If *ident* is an L_{tac} variable, its contents are printed; if not, it is an error.

Error: Tactic failure.

Error: `Tactic failure (level natural).`

Error: `No such goal.`

Example

```
Goal True.
    1 goal

      ============================
      True

Proof. fail. Abort.
    Toplevel input, characters 7-12:
    > Proof. fail.
    >        ^^^^^
    Error: Tactic failure.


Goal True.
    1 goal

      ============================
      True

Proof. trivial; fail. Qed.
    No more goals.


Goal True.
    1 goal

      ============================
      True

Proof. trivial. fail. Abort.
    No more goals.

    Toplevel input, characters 16-21:
    > Proof. trivial. fail.
    >                 ^^^^^
    Error: No such goal.


Goal True.
    1 goal

      ============================
      True

Proof. trivial. all: fail. Qed.
    No more goals.


Goal True.
    1 goal

      ============================
```

(continues on next page)

(continued from previous page)

```
      True

Proof. gfail. Abort.
    Toplevel input, characters 7-13:
    > Proof. gfail.
    >        ^^^^^^
    Error: Tactic failure.


Goal True.
    1 goal

      ============================
      True

Proof. trivial; gfail. Abort.
    Toplevel input, characters 7-22:
    > Proof. trivial; gfail.
    >        ^^^^^^^^^^^^^^^
    Error: Tactic failure.


Goal True.
    1 goal

      ============================
      True

Proof. trivial. gfail. Abort.
    No more goals.

    Toplevel input, characters 16-22:
    > Proof. trivial. gfail.
    >                 ^^^^^^
    Error: No such goal.


Goal True.
    1 goal

      ============================
      True

Proof. trivial. all: gfail. Abort.
    No more goals.

    Toplevel input, characters 16-27:
    > Proof. trivial. all: gfail.
    >                 ^^^^^^^^^^^
    Error: Tactic failure.
```

Timeout

We can force a tactic to stop if it has not finished after a certain amount of time:

Tactic: `timeout` *nat_or_var ltac_expr3*
ltac_expr3 is evaluated to v which must be a tactic value. The tactic value v is applied normally, except that it is interrupted after *nat_or_var* seconds if it is still running. In this case the outcome is a failure.

timeout is an *l3_tactic*.

> **Warning:** For the moment, timeout is based on elapsed time in seconds, which is very machine-dependent: a script that works on a quick machine may fail on a slow one. The converse is even possible if you combine a timeout with some other tacticals. This tactical is hence proposed only for convenience during debugging or other development phases, we strongly advise you to not leave any timeout in final scripts. Note also that this tactical isn't available on the native Windows port of Coq.

Timing a tactic

A tactic execution can be timed:

Tactic: `time` *string*? *ltac_expr3*
evaluates *ltac_expr3* and displays the running time of the tactic expression, whether it fails or succeeds. In case of several successes, the time for each successive run is displayed. Time is in seconds and is machine-dependent. The *string* argument is optional. When provided, it is used to identify this particular occurrence of *time*.

time is an *l3_tactic*.

Timing a tactic that evaluates to a term: time_constr

Tactic expressions that produce terms can be timed with the experimental tactic

Tactic: `time_constr` *ltac_expr*
which evaluates *ltac_expr* `()` and displays the time the tactic expression evaluated, assuming successful evaluation. Time is in seconds and is machine-dependent.

This tactic currently does not support nesting, and will report times based on the innermost execution. This is due to the fact that it is implemented using the following internal tactics:

Tactic: `restart_timer` *string*?
Reset a timer

Tactic: `finish_timing` (*string*)? *string*?
Display an optionally named timer. The parenthesized string argument is also optional, and determines the label associated with the timer for printing.

By copying the definition of *time_constr* from the standard library, users can achieve support for a fixed pattern of nesting by passing different *string* parameters to *restart_timer* and *finish_timing* at each level of nesting.

Example

```
Ltac time_constr1 tac :=
  let eval_early := match goal with _ => restart_timer "(depth 1)" end in
  let ret := tac () in
```

(continues on next page)

(continued from previous page)

```
  let eval_early := match goal with _ => finish_timing ( "Tactic evaluation" )
↪"(depth 1)" end in
  ret.
    time_constr1 is defined


Goal True.
    1 goal

      ============================
      True

  let v := time_constr
       ltac:(fun _ =>
               let x := time_constr1 ltac:(fun _ => constr:(10 * 10)) in
               let y := time_constr1 ltac:(fun _ => eval compute in x) in
               y) in
  pose v.
    Tactic evaluation (depth 1) ran for 0. secs (0.u,0.s)
    Tactic evaluation (depth 1) ran for 0. secs (0.u,0.s)
    Tactic evaluation ran for 0.001 secs (0.u,0.s)
    1 goal

      n := 100 : nat
      ============================
      True
```

Local definitions: let

Tactic: **let** **rec**? *let_clause* **with** *let_clause** **in** *ltac_expr*

let_clause ::= *name* **:=** *ltac_expr*
| *ident* *name*+ **:=** *ltac_expr*

Binds symbols within *ltac_expr*. *let* evaluates each ***let_clause***, substitutes the bound variables into ***ltac_expr*** and then evaluates ***ltac_expr***. There are no dependencies between the ***let_clause***s.

Use *let* rec to create recursive or mutually recursive bindings, which causes the definitions to be evaluated lazily.

let is a *binder_tactic*.

Function construction and application

A parameterized tactic can be built anonymously (without resorting to local definitions) with:

Tactic: **fun** *name*+ **=>** ***ltac_expr***

Indeed, local definitions of functions are syntactic sugar for binding a **fun** tactic to an identifier.

fun is a *binder_tactic*.

Functions can return values of any type.

A function application is an expression of the form:

Tactic: *qualid* *tactic_arg* +

qualid must be bound to a L_{tac} function with at least as many arguments as the provided *tactic_arg*s. The *tactic_arg*s are evaluated before the function is applied or partially applied.

Functions may be defined with the *fun* and *let* tactics and with the *Ltac* command.

Pattern matching on terms: match

Tactic: *match_key* *ltac_expr*$_{term}$ **with** | ? *match_pattern* => *ltac_expr* + | **end**

match_key	::=	**lazymatch**
	|	**match**
	|	**multimatch**
match_pattern	::=	*cpattern*
	|	**context** *ident* ? [*cpattern*]
cpattern	::=	*term*

lazymatch, *match* and *multimatch* are *ltac_expr1*s.

Evaluates *ltac_expr*$_{term}$, which must yield a term, and matches it sequentially with the *match_pattern*s, which may have metavariables. When a match is found, metavariable values are substituted into *ltac_expr*, which is then applied.

Matching may continue depending on whether `lazymatch`, `match` or `multimatch` is specified.

In the *match_pattern*s, metavariables have the form **?***ident*, whereas in the *ltac_expr*s, the question mark is omitted. Choose your metavariable names with care to avoid name conflicts. For example, if you use the metavariable `S`, then the *ltac_expr* can't use `S` to refer to the constructor of `nat` without qualifying the constructor as `Datatypes.S`.

Matching is non-linear: if a metavariable occurs more than once, each occurrence must match the same expression. Expressions match if they are syntactically equal or are *α-convertible*. Matching is first-order except on variables of the form **@?***ident* that occur in the head position of an application. For these variables, matching is second-order and returns a functional term.

lazymatch Causes the match to commit to the first matching branch rather than trying a new match if *ltac_expr* fails. *Example*.

match If *ltac_expr* fails, continue matching with the next branch. Failures in subsequent tactics (after the `match`) will not cause selection of a new branch. Examples *here* and *here*.

multimatch If *ltac_expr* fails, continue matching with the next branch. When an *ltac_expr* succeeds for a branch, subsequent failures (after the `multimatch`) causing consumption of all the successes of *ltac_expr* trigger selection of a new matching branch. *Example*.

match **…** is, in fact, shorthand for *once multimatch* ….

cpattern The syntax of *cpattern* is the same as that of *term*s, but it can contain pattern matching metavariables in the form **?***ident*. _ can be used to match irrelevant terms. *Example*.

When a metavariable in the form **?id** occurs under binders, say $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and the expression matches, the metavariable is instantiated by a term which can then be used in any context which also binds the variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with same types. This provides with a primitive form of matching under context which does not require manipulating a functional term.

There is also a special notation for second-order pattern matching: in an applicative pattern of the form **@?*ident* *ident*$_1$ … *ident*$_n$**, the variable *ident* matches any complex expression with (possible) dependencies in the variables ***ident***$_i$ and returns a functional term of the form **fun *ident*$_1$ … *ident*$_n$ => *term***.

context *ident*? **[** *cpattern* **]** Matches any term with a subterm matching *cpattern*. If there is a match and ***ident*** is present, it is assigned the "matched context", i.e. the initial term where the matched subterm is replaced by a hole. Note that `context` (with very similar syntax) appearing after the `=>` is the *context* tactic.

For *match* and *multimatch*, if the evaluation of the *ltac_expr* fails, the next matching subterm is tried. If no further subterm matches, the next branch is tried. Matching subterms are considered from top to bottom and from left to right (with respect to the raw printing obtained by setting the *Printing All* flag). *Example*.

ltac_expr The tactic to apply if the construct matches. Metavariable values from the pattern match are substituted into ***ltac_expr*** before it's applied. Note that metavariables are not prefixed with the question mark as they are in *cpattern*.

If *ltac_expr* evaluates to a tactic, then it is applied. If the tactic succeeds, the result of the match expression is *idtac*. If *ltac_expr* does not evaluate to a tactic, that value is the result of the match expression.

If ***ltac_expr*** is a tactic with backtracking points, then subsequent failures after a *lazymatch* or *multimatch* (but not *match*) can cause backtracking into ***ltac_expr*** to select its next success. (*match* … is equivalent to *once multimatch* …. The *once* prevents backtracking into the *match* after it has succeeded.)

Note: Each L_{tac} construct is processed in two phases: an evaluation phase and an execution phase. In most cases, tactics that may change the proof state are applied in the second phase. (Tactics that generate integer, string or syntactic values, such as *fresh*, are processed during the evaluation phase.)

Unlike other tactics, `*match*` tactics get their first success (applying tactics to do so) as part of the evaluation phase. Among other things, this can affect how early failures are processed in *assert_fails*. Please see the note in *assert_fails*.

Error: `Expression does not evaluate to a tactic.`
ltac_expr must evaluate to a tactic.

Error: `No matching clauses for match.`
For at least one of the focused goals, there is no branch that matches its pattern *and* gets at least one success for ***ltac_expr***.

Error: `Argument of match does not evaluate to a term.`
This happens when ***ltac_expr***$_{term}$ does not denote a term.

Example: Comparison of lazymatch and match

In *lazymatch*, if *ltac_expr* fails, the *lazymatch* fails; it doesn't look for further matches. In *match*, if *ltac_expr* fails in a matching branch, it will try to match on subsequent branches.

```
Fail lazymatch True with
| True => idtac "branch 1"; fail
| _ => idtac "branch 2"
end.
    branch 1
```

(continues on next page)

(continued from previous page)

```
    The command has indeed failed with message:
    Tactic failure.

match True with
| True => idtac "branch 1"; fail
| _ => idtac "branch 2"
end.
    branch 1
    branch 2
```

Example: Comparison of match and multimatch

match tactics are only evaluated once, whereas *multimatch* tactics may be evaluated more than once if the following constructs trigger backtracking:

```
Fail match True with
| True => idtac "branch 1"
| _ => idtac "branch 2"
end ;
idtac "branch A"; fail.
    branch 1
    branch A
    The command has indeed failed with message:
    Tactic failure.

Fail multimatch True with
| True => idtac "branch 1"
| _ => idtac "branch 2"
end ;
idtac "branch A"; fail.
    branch 1
    branch A
    branch 2
    branch A
    The command has indeed failed with message:
    Tactic failure.
```

Example: Matching a pattern with holes

Notice the *idtac* prints `(z + 1)` while the *pose* substitutes `(x + 1)`.

```
Goal True.

match constr:(fun x => (x + 1) * 3) with
| fun z => ?y * 3 => idtac "y =" y; pose (fun z: nat => y * 5)
end.
    y = (z + 1)
    1 goal

      n := fun x : nat => (x + 1) * 5 : nat -> nat
      ============================
      True
```

Example: Multiple matches for a "context" pattern.

Internally "x <> y" is represented as "(~ (x = y))", which produces the first match.

```
Ltac f t := match t with
            | context [ (~ ?t) ] => idtac "?t = " t; fail
            | _ => idtac
            end.
Goal True.

f ((~ True) <> (~ False)).
    ?t =  ((~ True) = (~ False))
    ?t =  True
    ?t =  False
```

Pattern matching on goals and hypotheses: match goal

Tactic: *match_key* **reverse**? **goal with** (| *goal_pattern* **=>** *ltac_expr*)+ **end**

goal_pattern ::= (*match_hyp*, *)? |- *match_pattern*
| [(*match_hyp*, *)? |- *match_pattern*]
| _

match_hyp ::= *name* **:** *match_pattern*
| *name* **:=** *match_pattern*
| *name* **:=** [*match_pattern*] **:** *match_pattern*

lazymatch goal, *match goal* and *multimatch goal* are *ltac1_tactic*s.

Use this form to match hypotheses and/or goals in the local context. These patterns have zero or more subpatterns to match hypotheses followed by a subpattern to match the conclusion. Except for the differences noted below, this works the same as the corresponding ***match_key ltac_expr*** construct (see *match*). Each current goal is processed independently.

Matching is non-linear: if a metavariable occurs more than once, each occurrence must match the same expression. Within a single term, expressions match if they are syntactically equal or *α-convertible*. When a metavariable is used across multiple hypotheses or across a hypothesis and the current goal, the expressions match if they are *convertible*.

(***match_hyp***, *)? Patterns to match with hypotheses. Each pattern must match a distinct hypothesis in order for the branch to match.

Hypotheses have the form ***name*** (**:=** ***term***$_{binder}$)? **:** ***type***. Patterns bind each of these nonterminals separately:

Pattern syntax	Example pattern
name `:` ***match_pattern***$_{type}$	`n : ?t`
name `:=` ***match_pattern***$_{binder}$	`n := ?b`
name `:=` ***term***$_{binder}$ `:` ***type***	`n := ?b : ?t`
name `:= [` ***match_pattern***$_{binder}$ `] :` ***match_pattern***$_{type}$	`n := [ ?b ] : ?t`

name can't have a `?`. Note that the last two forms are equivalent except that:

- if the `:` in the third form has been bound to something else in a notation, you must use the fourth form. Note that cmd:`Require Import ssreflect` loads a notation that does this.
- a ***term***$_{binder}$ such as `[ ?l ]` (e.g., denoting a singleton list after *Import* `ListNotations`) must be parenthesized or, for the fourth form, use double brackets: `[ [ ?l ] ]`.

term$_{binder}$s in the form `[?x ; ?y]` for a list are not parsed correctly. The workaround is to add parentheses or to use the underlying term instead of the notation, i.e. `(cons ?x ?y)`.

If there are multiple *match_hyp*s in a branch, there may be multiple ways to match them to hypotheses. For *match goal* and *multimatch goal*, if the evaluation of the *ltac_expr* fails, matching will continue with the next hypothesis combination. When those are exhausted, the next alternative from any `context` constructs in the *match_pattern*s is tried and then, when the context alternatives are exhausted, the next branch is tried. *Example*.

reverse Hypothesis matching for *match_hyp*s normally begins by matching them from left to right, to hypotheses, last to first. Specifying `reverse` begins matching in the reverse order, from first to last. *Normal* and *reverse* examples.

|- ***match_pattern*** A pattern to match with the current goal

goal_pattern **with [...]** The square brackets don't affect the semantics. They are permitted for aesthetics.

Error: No matching clauses for match goal.
No clause succeeds, i.e. all matching patterns, if any, fail at the application of the *ltac_expr*.

Examples:

Example: Matching hypotheses

Hypotheses are matched from the last hypothesis (which is by default the newest hypothesis) to the first until the *apply* succeeds.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros.
    1 goal

      A, B : Prop
      H : A
      H0 : B
      H1 : A
```

(continues on next page)

(continued from previous page)

```
      ============================
      B

match goal with
| H : _ |- _ => idtac "apply " H; apply H
end.
    apply  H1
    apply  H0
    No more goals.
```

Example: Matching hypotheses with reverse

Hypotheses are matched from the first hypothesis to the last until the *apply* succeeds.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros.
    1 goal

      A, B : Prop
      H : A
      H0 : B
      H1 : A
      ============================
      B

match reverse goal with
| H : _ |- _ => idtac "apply " H; apply H
end.
    apply  A
    apply  B
    apply  H
    apply  H0
    No more goals.
```

Example: Multiple ways to match hypotheses

Every possible match for the hypotheses is evaluated until the right-hand side succeeds. Note that `H1` and `H2` are never matched to the same hypothesis. Observe that the number of permutations can grow as the factorial of the number of hypotheses and hypothesis patterns.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros A B H.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      A, B : Prop
      H : A
      ============================
      B -> A -> B

match goal with
| H1 : _, H2 : _ |- _ => idtac "match " H1 H2; fail
| _ => idtac
end.
    match  B H
    match  A H
    match  H B
    match  A B
    match  H A
    match  B A
```

Filling a term context

The following expression is not a tactic in the sense that it does not produce subgoals but generates a term to be used in tactic expressions:

Tactic: `context` *`ident`* `[` *`term`* `]`
Returns the term matched with the `context` pattern (described *here*) substituting *`term`* for the hole created by the pattern.

`context` is a *`value_tactic`*.

Error: `Not a context variable.`

Error: `Unbound context identifier` *`ident`*`.`

Example: Substituting a matched context

```
Goal True /\ True.
    1 goal

      ============================
      True /\ True

match goal with
| |- context G [True] => let x := context G [False] in idtac x
end.
    (False /\ True)
```

Generating fresh hypothesis names

Tactics sometimes need to generate new names for hypothesis. Letting Coq choose a name with the intro tactic is not so good since it is very awkward to retrieve that name. The following expression returns an identifier:

Tactic: `fresh` *string* | *qualid* *

Returns a fresh identifier name (i.e. one that is not already used in the local context and not previously returned by *fresh* in the current *ltac_expr*). The fresh identifier is formed by concatenating the final *ident* of each *qualid* (dropping any qualified components) and each specified *string*. If the resulting name is already used, a number is appended to make it fresh. If no arguments are given, the name is a fresh derivative of the name `H`.

Note: We recommend generating the fresh identifier immediately before adding it to the local context. Using *fresh* in a local function may not work as you expect:

Successive calls to *fresh* give distinct names even if the names haven't yet been added to the local context:

```
1 goal

  x : True
  ============================
  True
```

```
let a := fresh "x" in
let b := fresh "x" in
idtac a b.
    x0 x1
```

When applying *fresh* in a function, the name is chosen based on the tactic context at the point where the function was defined:

```
let a := fresh "x" in
let f := fun _ => fresh "x" in
let c := f () in
let d := f () in
idtac a c d.
    x0 x1 x1
```

---

fresh is a *value_tactic*.

Computing in a term: eval

Evaluation of a term can be performed with:

`eval` *red_expr* `in` *term*

See *eval*. *eval* is a *value_tactic*.

Getting the type of a term

Tactic: `type of` *term*

This tactic returns the type of *term*.

type of is a *value_tactic*.

Manipulating untyped terms: type_term

The **`uconstr : ( term )`** construct can be used to build an untyped term. See *syn_value*.

Tactic: `type_term` *one_term*

In $\mathrm{L}_{\mathrm{tac}}$, an untyped term can contain references to hypotheses or to $\mathrm{L}_{\mathrm{tac}}$ variables containing typed or untyped terms. An untyped term can be type checked with *type_term* whose argument is parsed as an untyped term and returns a well-typed term which can be used in tactics.

type_term is a *value_tactic*.

Counting goals: numgoals

Tactic: `numgoals`

The number of goals under focus can be recovered using the **`numgoals`** function. Combined with the *guard* tactic below, it can be used to branch over the number of goals produced by previous tactics.

numgoals is a *value_tactic*.

Example

```
Ltac pr_numgoals := let n := numgoals in idtac "There are" n "goals".

Goal True /\ True /\ True.
split;[|split].

all:pr_numgoals.
    There are 3 goals
```

Testing boolean expressions: guard

Tactic: `guard` *int_or_var comparison int_or_var*

int_or_var	::=	*integer* | *ident*
comparison	::=	`=`
	|	`<`
	|	`<=`
	|	`>`
	|	`>=`

Tests a boolean expression. If the expression evaluates to true, it succeeds without affecting the proof. The tactic fails if the expression is false.

The accepted tests are simple integer comparisons.

Example

```
Goal True /\ True /\ True.
split;[|split].

all:let n:= numgoals in guard n<4.
Fail all:let n:= numgoals in guard n=2.
    The command has indeed failed with message:
    Condition not satisfied: 3=2
```

Error: Condition not satisfied.

Proving a subgoal as a separate lemma: abstract

Tactic: abstract *ltac_expr2* **using** *ident*$_{name}$?

Does a *solve* **[** *ltac_expr2* **]** and saves the subproof as an auxiliary lemma. if *ident*$_{name}$ is specified, the lemma is saved with that name; otherwise the lemma is saved with the name *ident*_subproof*natural* ? where *ident* is the name of the current goal (e.g. the theorem name) and *natural* is chosen to get a fresh name. If the proof is closed with *Qed*, the auxiliary lemma is inlined in the final proof term.

This is useful with tactics such as *discriminate* that generate huge proof terms with many intermediate goals. It can significantly reduce peak memory use. In most cases it doesn't have a significant impact on run time. One case in which it can reduce run time is when a tactic `foo` is known to always pass type checking when it succeeds, such as in reflective proofs. In this case, the idiom "*abstract exact_no_check* `foo`" will save half the type checking type time compared to "*exact* `foo`".

abstract is an *l3_tactic*.

Warning: The abstract tactic, while very useful, still has some known limitations. See #9146[53] for more details. We recommend caution when using it in some "non-standard" contexts. In particular, `abstract` doesn't work properly when used inside quotations `ltac:(...)`. If used as part of typeclass resolution, it may produce incorrect terms when in polymorphic universe mode.

Warning: Provide *ident*$_{name}$ at your own risk; explicitly named and reused subterms don't play well with asynchronous proofs.

Tactic: transparent_abstract *ltac_expr3* **using** *ident* ?

Like *abstract*, but save the subproof in a transparent lemma with a name in the form *ident*_subterm*natural* ? .

Warning: Use this feature at your own risk; building computationally relevant terms with tactics is fragile, and explicitly named and reused subterms don't play well with asynchronous proofs.

Error: Proof is not complete.

[53] https://github.com/coq/coq/issues/9146

Tactic toplevel definitions

Defining $\mathrm{L}_{\mathrm{tac}}$ symbols

$\mathrm{L}_{\mathrm{tac}}$ toplevel definitions are made as follows:

Command: **Ltac** *tacdef_body* **with** *tacdef_body* *

tacdef_body ::= *qualid* *name* * := | ::= *ltac_expr*

Defines or redefines an $\mathrm{L}_{\mathrm{tac}}$ symbol.

If the *local* attribute is specified, the definition will not be exported outside the current module.

qualid Name of the symbol being defined or redefined. For definitions, *qualid* must be a simple *ident*.

name * If specified, the symbol defines a function with the given parameter names. If no names are specified, *qualid* is assigned the value of *ltac_expr*.

:= Defines a user-defined symbol, but gives an error if the symbol has already been defined.

Error: There is already an Ltac named *qualid*

::= Redefines an existing user-defined symbol, but gives an error if the symbol doesn't exist. Note that *Tactic Notation*s do not count as user-defined tactics for ::=. If *local* is not specified, the redefinition applies across module boundaries.

Error: There is no Ltac named *qualid*

with *tacdef_body* * Permits definition of mutually recursive tactics.

Note: The following definitions are equivalent:

- **Ltac *qualid* *name*** + **:= *ltac_expr***
- **Ltac *qualid* := fun *name*** + **=> *ltac_expr***

Printing $\mathrm{L}_{\mathrm{tac}}$ tactics

Command: **Print Ltac *qualid***
Defined $\mathrm{L}_{\mathrm{tac}}$ functions can be displayed using this command.

Command: **Print Ltac Signatures**
This command displays a list of all user-defined tactics, with their arguments.

Examples of using Ltac

Proof that the natural numbers have at least two elements

Example: Proof that the natural numbers have at least two elements

The first example shows how to use pattern matching over the proof context to prove that natural numbers have at least two elements. This can be done as follows:

```
Lemma card_nat :
  ~ exists x y : nat, forall z:nat, x = z \/ y = z.
    1 goal

      ============================
      ~ (exists x y : nat, forall z : nat, x = z \/ y = z)

Proof.
intros (x & y & Hz).
    1 goal

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      ============================
      False

destruct (Hz 0), (Hz 1), (Hz 2).
    8 goals

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      H : x = 0
      H0 : x = 1
      H1 : x = 2
      ============================
      False

    goal 2 is:
     False
    goal 3 is:
     False
    goal 4 is:
     False
    goal 5 is:
     False
    goal 6 is:
     False
    goal 7 is:
     False
    goal 8 is:
     False
```

At this point, the *congruence* tactic would finish the job:

```
all: congruence.
    No more goals.
```

But for the purpose of the example, let's craft our own custom tactic to solve this:

```
all: match goal with
     | _ : ?a = ?b, _ : ?a = ?c |- _ => assert (b = c) by now transitivity a
     end.
    8 goals

      x, y : nat
      Hz : forall z : nat, x = z \/ y = z
      H : x = 0
      H0 : x = 1
      H1 : x = 2
      H2 : 1 = 2
      ============================
      False

    goal 2 is:
     False
    goal 3 is:
     False
    goal 4 is:
     False
    goal 5 is:
     False
    goal 6 is:
     False
    goal 7 is:
     False
    goal 8 is:
     False

all: discriminate.
    No more goals.
```

Notice that all the (very similar) cases coming from the three eliminations (with three distinct natural numbers) are successfully solved by a `match goal` structure and, in particular, with only one pattern (use of non-linear matching).

Proving that a list is a permutation of a second list

Example: Proving that a list is a permutation of a second list

Let's first define the permutation predicate:

```
Section Sort.

  Variable A : Set.

  Inductive perm : list A -> list A -> Prop :=
  | perm_refl : forall l, perm l l
  | perm_cons : forall a l0 l1, perm l0 l1 -> perm (a :: l0) (a :: l1)
  | perm_append : forall a l, perm (a :: l) (l ++ a :: nil)
  | perm_trans : forall l0 l1 l2, perm l0 l1 -> perm l1 l2 -> perm l0 l2.

End Sort.
```

Next we define an auxiliary tactic `perm_aux` which takes an argument used to control the recursion depth. This tactic works as follows: If the lists are identical (i.e. convertible), it completes the proof. Otherwise, if the lists have identical

heads, it looks at their tails. Finally, if the lists have different heads, it rotates the first list by putting its head at the end.

Every time we perform a rotation, we decrement `n`. When `n` drops down to `1`, we stop performing rotations and we fail. The idea is to give the length of the list as the initial value of `n`. This way of counting the number of rotations will avoid going back to a head that had been considered before.

From Section *Syntax* we know that Ltac has a primitive notion of integers, but they are only used as arguments for primitive tactics and we cannot make computations with them. Thus, instead, we use Coq's natural number type `nat`.

```
Ltac perm_aux n :=
  match goal with
  | |- (perm _ ?l ?l) => apply perm_refl
  | |- (perm _ (?a :: ?l1) (?a :: ?l2)) =>
     let newn := eval compute in (length l1) in
         (apply perm_cons; perm_aux newn)
  | |- (perm ?A (?a :: ?l1) ?l2) =>
     match eval compute in n with
     | 1 => fail
     | _ =>
         let l1' := constr:(l1 ++ a :: nil) in
         (apply (perm_trans A (a :: l1) l1' l2);
         [ apply perm_append | compute; perm_aux (pred n) ])
     end
  end.
```

The main tactic is `solve_perm`. It computes the lengths of the two lists and uses them as arguments to call `perm_aux` if the lengths are equal. (If they aren't, the lists cannot be permutations of each other.)

```
Ltac solve_perm :=
  match goal with
  | |- (perm _ ?l1 ?l2) =>
     match eval compute in (length l1 = length l2) with
     | (?n = ?n) => perm_aux n
     end
  end.
```

And now, here is how we can use the tactic `solve_perm`:

```
    1 goal

      ============================
      perm nat (1 :: 2 :: 3 :: nil) (3 :: 2 :: 1 :: nil)

solve_perm.
    No more goals.

    1 goal

      ============================
      perm nat (0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: 9 :: nil)
        (0 :: 2 :: 4 :: 6 :: 8 :: 9 :: 7 :: 5 :: 3 :: 1 :: nil)

solve_perm.
    No more goals.
```

Deciding intuitionistic propositional logic

Pattern matching on goals allows powerful backtracking when returning tactic values. An interesting application is the problem of deciding intuitionistic propositional logic. Considering the contraction-free sequent calculi LJT* of Roy Dyckhoff [Dyc92], it is quite natural to code such a tactic using the tactic language as shown below.

```
Ltac basic :=
match goal with
    | |- True => trivial
    | _ : False |- _ => contradiction
    | _ : ?A |- ?A => assumption
end.

Ltac simplify :=
repeat (intros;
    match goal with
        | H : ~ _ |- _ => red in H
        | H : _ /\ _ |- _ =>
            elim H; do 2 intro; clear H
        | H : _ \/ _ |- _ =>
            elim H; intro; clear H
        | H : ?A /\ ?B -> ?C |- _ =>
            cut (A -> B -> C);
                [ intro | intros; apply H; split; assumption ]
        | H: ?A \/ ?B -> ?C |- _ =>
            cut (B -> C);
                [ cut (A -> C);
                    [ intros; clear H
                    | intro; apply H; left; assumption ]
                | intro; apply H; right; assumption ]
        | H0 : ?A -> ?B, H1 : ?A |- _ =>
            cut B; [ intro; clear H0 | apply H0; assumption ]
        | |- _ /\ _ => split
        | |- ~ _ => red
    end).

Ltac my_tauto :=
  simplify; basic ||
  match goal with
      | H : (?A -> ?B) -> ?C |- _ =>
          cut (B -> C);
              [ intro; cut (A -> B);
                  [ intro; cut C;
                      [ intro; clear H | apply H; assumption ]
                  | clear H ]
              | intro; apply H; intro; assumption ]; my_tauto
      | H : ~ ?A -> ?B |- _ =>
          cut (False -> B);
              [ intro; cut (A -> False);
                  [ intro; cut B;
                      [ intro; clear H | apply H; assumption ]
                  | clear H ]
              | intro; apply H; red; intro; assumption ]; my_tauto
      | |- _ \/ _ => (left; my_tauto) || (right; my_tauto)
  end.
```

The tactic `basic` tries to reason using simple rules involving truth, falsity and available assumptions. The tactic

`simplify` applies all the reversible rules of Dyckhoff's system. Finally, the tactic `my_tauto` (the main tactic to be called) simplifies with `simplify`, tries to conclude with `basic` and tries several paths using the backtracking rules (one of the four Dyckhoff's rules for the left implication to get rid of the contraction and the right `or`).

Having defined `my_tauto`, we can prove tautologies like these:

```
Lemma my_tauto_ex1 :
  forall A B : Prop, A /\ B -> A \/ B.
Proof. my_tauto. Qed.

Lemma my_tauto_ex2 :
  forall A B : Prop, (~ ~ B -> B) -> (A -> B) -> ~ ~ A -> B.
Proof. my_tauto. Qed.
```

Deciding type isomorphisms

A trickier problem is to decide equalities between types modulo isomorphisms. Here, we choose to use the isomorphisms of the simply typed λ-calculus with Cartesian product and unit type (see, for example, [dC95]). The axioms of this λ-calculus are given below.

```
Open Scope type_scope.

Section Iso_axioms.

Variables A B C : Set.

Axiom Com : A * B = B * A.

Axiom Ass : A * (B * C) = A * B * C.

Axiom Cur : (A * B -> C) = (A -> B -> C).

Axiom Dis : (A -> B * C) = (A -> B) * (A -> C).

Axiom P_unit : A * unit = A.

Axiom AR_unit : (A -> unit) = unit.

Axiom AL_unit : (unit -> A) = A.

Lemma Cons : B = C -> A * B = A * C.

Proof.

intro Heq; rewrite Heq; reflexivity.

Qed.

End Iso_axioms.

Ltac simplify_type ty :=
match ty with
    | ?A * ?B * ?C =>
        rewrite <- (Ass A B C); try simplify_type_eq
```

(continues on next page)

(continued from previous page)

```
    | ?A * ?B -> ?C =>
      rewrite (Cur A B C); try simplify_type_eq
    | ?A -> ?B * ?C =>
      rewrite (Dis A B C); try simplify_type_eq
    | ?A * unit =>
      rewrite (P_unit A); try simplify_type_eq
    | unit * ?B =>
      rewrite (Com unit B); try simplify_type_eq
    | ?A -> unit =>
      rewrite (AR_unit A); try simplify_type_eq
    | unit -> ?B =>
      rewrite (AL_unit B); try simplify_type_eq
    | ?A * ?B =>
      (simplify_type A; try simplify_type_eq) ||
      (simplify_type B; try simplify_type_eq)
    | ?A -> ?B =>
      (simplify_type A; try simplify_type_eq) ||
      (simplify_type B; try simplify_type_eq)
end
with simplify_type_eq :=
match goal with
    | |- ?A = ?B => try simplify_type A; try simplify_type B
end.

Ltac len trm :=
match trm with
    | _ * ?B => let succ := len B in constr:(S succ)
    | _ => constr:(1)
end.

Ltac assoc := repeat rewrite <- Ass.

Ltac solve_type_eq n :=
match goal with
    | |- ?A = ?A => reflexivity
    | |- ?A * ?B = ?A * ?C =>
      apply Cons; let newn := len B in solve_type_eq newn
    | |- ?A * ?B = ?C =>
      match eval compute in n with
          | 1 => fail
          | _ =>
            pattern (A * B) at 1; rewrite Com; assoc; solve_type_eq (pred n)
      end
end.

Ltac compare_structure :=
match goal with
    | |- ?A = ?B =>
      let l1 := len A
      with l2 := len B in
          match eval compute in (l1 = l2) with
              | ?n = ?n => solve_type_eq n
          end
end.
```

```
Ltac solve_iso := simplify_type_eq; compare_structure.
```

The tactic to judge equalities modulo this axiomatization is shown above. The algorithm is quite simple. First types are simplified using axioms that can be oriented (this is done by `simplify_type` and `simplify_type_eq`). The normal forms are sequences of Cartesian products without a Cartesian product in the left component. These normal forms are then compared modulo permutation of the components by the tactic `compare_structure`. If they have the same length, the tactic `solve_type_eq` attempts to prove that the types are equal. The main tactic that puts all these components together is `solve_iso`.

Here are examples of what can be solved by `solve_iso`.

```
Lemma solve_iso_ex1 :
  forall A B : Set, A * unit * B = B * (unit * A).
Proof.
  intros; solve_iso.
Qed.

Lemma solve_iso_ex2 :
  forall A B C : Set,
    (A * unit -> B * (C * unit)) =
    (A * unit -> (C -> unit) * C) * (unit -> A -> B).
Proof.
  intros; solve_iso.
Qed.
```

Debugging L_{tac} tactics

Backtraces

Flag: `Ltac Backtrace`
Setting this *flag* displays a backtrace on Ltac failures that can be useful to find out what went wrong. It is disabled by default for performance reasons.

Tracing execution

Command: `Info` *`natural ltac_expr`*
Applies *`ltac_expr`* and prints a trace of the tactics that were successfully applied, discarding branches that failed. *`idtac`* tactics appear in the trace as comments containing the output.

This command is valid only in proof mode. It accepts *Goal selectors*.

The number ***`natural`*** is the unfolding level of tactics in the trace. At level 0, the trace contains a sequence of tactics in the actual script, at level 1, the trace will be the concatenation of the traces of these tactics, etc…

Example

```
Ltac t x := exists x; reflexivity.
Goal exists n, n=0.

Info 0 t 1||t 0.
    exists  with 0;<ltac_plugin::reflexivity@0>
    No more goals.
```

```
Undo.

Info 1 t 1||t 0.
    <ltac_plugin::exists@1>  with 0;simple refine ?X11
    No more goals.
```

The trace produced by *Info* tries its best to be a reparsable $\mathrm{L}_{\mathrm{tac}}$ script, but this goal is not achievable in all generality. So some of the output traces will contain oddities.

As an additional help for debugging, the trace produced by *Info* contains (in comments) the messages produced by the *idtac* tactical at the right position in the script. In particular, the calls to idtac in branches which failed are not printed.

Option: Info Level *natural*

This *option* is an alternative to the *Info* command.

This will automatically print the same trace as **Info *natural*** at each tactic call. The unfolding level can be overridden by a call to the *Info* command.

Interactive debugger

Flag: Ltac Debug

This *flag* governs the step-by-step debugger that comes with the $\mathrm{L}_{\mathrm{tac}}$ interpreter.

When the debugger is activated, it stops at every step of the evaluation of the current $\mathrm{L}_{\mathrm{tac}}$ expression and prints information on what it is doing. The debugger stops, prompting for a command which can be one of the following:

newline	go to the next step
h	get help
r n	advance n steps further
r string	advance up to the next call to “idtac string”
s	continue current evaluation without stopping
x	exit current evaluation

Error: Debug mode not available in the IDE

A non-interactive mode for the debugger is available via the flag:

Flag: Ltac Batch Debug

This *flag* has the effect of presenting a newline at every prompt, when the debugger is on. The debug log thus created, which does not require user input to generate when this flag is set, can then be run through external tools such as diff.

Profiling $\mathrm{L}_{\mathrm{tac}}$ tactics

It is possible to measure the time spent in invocations of primitive tactics as well as tactics defined in $\mathrm{L}_{\mathrm{tac}}$ and their inner invocations. The primary use is the development of complex tactics, which can sometimes be so slow as to impede interactive usage. The reasons for the performance degradation can be intricate, like a slowly performing $\mathrm{L}_{\mathrm{tac}}$ match or a sub-tactic whose performance only degrades in certain situations. The profiler generates a call tree and indicates the time spent in a tactic depending on its calling context. Thus it allows to locate the part of a tactic definition that contains the performance issue.

Flag: `Ltac Profiling`
This *flag* enables and disables the profiler.

Command: `Show Ltac Profile` `CutOff` *integer* | *string* ?
Prints the profile.

`CutOff` *integer* By default, tactics that account for less than 2% of the total time are not displayed. `CutOff` lets you specify a different percentage.

string

Limits the profile to all tactics that start with ***string***. Append a period (.) to the string if you only want exactly that name.

Command: `Reset Ltac Profile`
Resets the profile, that is, deletes all accumulated information.

> **Warning:** Backtracking across a *Reset Ltac Profile* will not restore the information.

```
Require Import Lia.

Ltac mytauto := tauto.
Ltac tac := intros; repeat split; lia || mytauto.

Notation max x y := (x + (y - x)) (only parsing).

Goal forall x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z,
    max x (max y z) = max (max x y) z /\ max x (max y z) = max (max x y) z
    /\
    (A /\ B /\ C /\ D /\ E /\ F /\ G /\ H /\ I /\ J /\ K /\ L /\ M /\
     N /\ O /\ P /\ Q /\ R /\ S /\ T /\ U /\ V /\ W /\ X /\ Y /\ Z
     ->
     Z /\ Y /\ X /\ W /\ V /\ U /\ T /\ S /\ R /\ Q /\ P /\ O /\ N /\
     M /\ L /\ K /\ J /\ I /\ H /\ G /\ F /\ E /\ D /\ C /\ B /\ A).
Proof.

Set Ltac Profiling.
tac.
    No more goals.

Show Ltac Profile.
    total time:      8.319s

     tactic                                        local  total   calls       max
    ──────────────────────────────────────────────┴──────┴──────┴───────┴──────────┘
    ─tac ----------------------------------       0.1% 100.0%       1    8.319s
    ─<Coq.Init.Tauto.with_uniform_flags> ---      0.0%  96.3%      26    0.564s
    ─<Coq.Init.Tauto.tauto_gen> ------------      0.0%  96.3%      26    0.564s
    ─<Coq.Init.Tauto.tauto_intuitionistic> -      0.0%  96.3%      26    0.564s
    ─t_tauto_intuit ------------------------      0.1%  96.2%      26    0.563s
    ─<Coq.Init.Tauto.simplif> --------------     67.2%  92.6%      26    0.552s
    ─<Coq.Init.Tauto.is_conj> --------------     19.2%  19.2%   28756    0.035s
    ─<Coq.Init.Tauto.axioms> ---------------      2.6%   3.6%       0    0.016s
    ─elim id -------------------------------      3.5%   3.5%     650    0.056s
    ─lia ------------------------------------      0.1%   3.3%      28    0.217s
    ─xlia (tactic) -------------------------      2.5%   2.7%      28    0.209s
```

(continues on next page)

(continued from previous page)

```
 tactic                                    local  total   calls       max
──────────────────────────────────────────┴──────┴──────┴───────┴──────────┘
─tac ------------------------------------   0.1% 100.0%       1    8.319s
 ├─<Coq.Init.Tauto.with_uniform_flags> -   0.0%  96.3%      26    0.564s
 │└<Coq.Init.Tauto.tauto_gen> ----------   0.0%  96.3%      26    0.564s
 │└<Coq.Init.Tauto.tauto_intuitionistic>   0.0%  96.3%      26    0.564s
 │└t_tauto_intuit ----------------------   0.1%  96.2%      26    0.563s
 │ ├─<Coq.Init.Tauto.simplif> ----------  67.2%  92.6%      26    0.552s
 │ │ ├─<Coq.Init.Tauto.is_conj> --------  19.2%  19.2%   28756    0.035s
 │ │ └─elim id --------------------------   3.5%   3.5%     650    0.056s
 │ └─<Coq.Init.Tauto.axioms> ----------   2.6%   3.6%       0    0.016s
 └─lia -----------------------------------   0.1%   3.3%      28    0.217s
  └xlia (tactic) ----------------------   2.5%   2.7%      28    0.209s

Show Ltac Profile "lia".
    total time:      8.319s

 tactic                                    local  total   calls       max
──────────────────────────────────────────┴──────┴──────┴───────┴──────────┘
─lia ------------------------------------   0.1%   3.3%      28    0.217s

 tactic                                    local  total   calls       max
──────────────────────────────────────────┴──────┴──────┴───────┴──────────┘

Abort.
Unset Ltac Profiling.
```

Tactic: `start ltac profiling`
This tactic behaves like *idtac* but enables the profiler.

Tactic: `stop ltac profiling`
Similarly to *start ltac profiling*, this tactic behaves like *idtac*. Together, they allow you to exclude parts of a proof script from profiling.

Tactic: `reset ltac profile`
Equivalent to the *Reset Ltac Profile* command, which allows resetting the profile from tactic scripts for benchmarking purposes.

Tactic: `show ltac profile` `cutoff` *integer* | *string* ?
Equivalent to the *Show Ltac Profile* command, which allows displaying the profile from tactic scripts for benchmarking purposes.

Warning: `Ltac Profiler encountered an invalid stack (no self node). This can happen if you reset the profile during tactic execution`
Currently, *reset ltac profile* is not very well-supported, as it clears all profiling information about all tactics, including ones above the current tactic. As a result, the profiler has trouble understanding where it is in tactic execution. This mixes especially poorly with backtracking into multi-success tactics. In general, non-top-level calls to *reset ltac profile* should be avoided.

You can also pass the `-profile-ltac` command line option to `coqc`, which turns the *Ltac Profiling* flag on at the beginning of each document, and performs a *Show Ltac Profile* at the end.

Run-time optimization tactic

Tactic: `optimize_heap`
This tactic behaves like *`idtac`*, except that running it compacts the heap in the OCaml run-time system. It is analogous to the *`Optimize Heap`* command.

Tactic: `infoH` *`ltac_expr3`*
Used internally by Proof General. See #12423[54] for some background.

`infoH` is an *`l3_tactic`*.

3.3.2 Ltac2

The $\mathrm{L}_{\mathrm{tac}}$ tactic language is probably one of the ingredients of the success of Coq, yet it is at the same time its Achilles' heel. Indeed, $\mathrm{L}_{\mathrm{tac}}$:

- has often unclear semantics
- is very non-uniform due to organic growth
- lacks expressivity (data structures, combinators, types, ...)
- is slow
- is error-prone and fragile
- has an intricate implementation

Following the need of users who are developing huge projects relying critically on Ltac, we believe that we should offer a proper modern language that features at least the following:

- at least informal, predictable semantics
- a type system
- standard programming facilities (e.g., datatypes)

This new language, called Ltac2, is described in this chapter. It is still experimental but we nonetheless encourage users to start testing it, especially wherever an advanced tactic language is needed. The previous implementation of Ltac, described in the previous chapter, will be referred to as Ltac1.

Current limitations include:

- There are a number of tactics that are not yet supported in Ltac2 because the interface OCaml and/or Ltac2 notations haven't been written. See *Defining tactics*.
- Missing usability features such as:
  - Printing functions are limited and awkward to use. Only a few data types are printable.
  - Deep pattern matching and matching on tuples don't work.
  - A convenient way to build terms with casts through the low-level API. Because the cast type is opaque, building terms with casts currently requires an awkward construction like the following, which also incurs extra overhead to repeat typechecking for each call to `get_vm_cast`:

    ```
    Constr.Unsafe.make (Constr.Unsafe.Cast 'I (get_vm_cast ()) 'True)
    ```

    with:

[54] https://github.com/coq/coq/issues/12423

```
Ltac2 get_vm_cast () :=
  match Constr.Unsafe.kind '(I <: True) with
  | Constr.Unsafe.Cast _ cst _ => cst
  | _ => Control.throw Not_found
  end.
```

- Missing low-level primitives that are convenient for writing automation, such as:
  - An easy way to get the number of constructors of an inductive type. Currently only way to do this is to destruct a variable of the inductive type and count the number of goals that result.
- Error messages may be cryptic.

General design

There are various alternatives to Ltac1, such as Mtac or Rtac for instance. While those alternatives can be quite different from Ltac1, we designed Ltac2 to be as close as reasonably possible to Ltac1, while fixing the aforementioned defects.

In particular, Ltac2 is:

- a member of the ML family of languages, i.e.
  - a call-by-value functional language
  - with effects
  - together with the Hindley-Milner type system
- a language featuring meta-programming facilities for the manipulation of Coq-side terms
- a language featuring notation facilities to help write palatable scripts

We describe these in more detail in the remainder of this document.

ML component

Overview

Ltac2 is a member of the ML family of languages, in the sense that it is an effectful call-by-value functional language, with static typing à la Hindley-Milner (see [DM82]). It is commonly accepted that ML constitutes a sweet spot in PL design, as it is relatively expressive while not being either too lax (unlike dynamic typing) nor too strict (unlike, say, dependent types).

The main goal of Ltac2 is to serve as a meta-language for Coq. As such, it naturally fits in the ML lineage, just as the historical ML was designed as the tactic language for the LCF prover. It can also be seen as a general-purpose language, by simply forgetting about the Coq-specific features.

Sticking to a standard ML type system can be considered somewhat weak for a meta-language designed to manipulate Coq terms. In particular, there is no way to statically guarantee that a Coq term resulting from an Ltac2 computation will be well-typed. This is actually a design choice, motivated by backward compatibility with Ltac1. Instead, well-typedness is deferred to dynamic checks, allowing many primitive functions to fail whenever they are provided with an ill-typed term.

The language is naturally effectful as it manipulates the global state of the proof engine. This allows to think of proof-modifying primitives as effects in a straightforward way. Semantically, proof manipulation lives in a monad, which allows to ensure that Ltac2 satisfies the same equations as a generic ML with unspecified effects would do, e.g. function reduction is substitution by a value.

Use the following command to import Ltac2:

```
From Ltac2 Require Import Ltac2.
```

Type Syntax

At the level of terms, we simply elaborate on Ltac1 syntax, which is quite close to OCaml. Types follow the simply-typed syntax of OCaml.

ltac2_type ::= *ltac2_type2* **->** *ltac2_type*
| *ltac2_type2*

ltac2_type2 ::= *ltac2_type1* ***** *ltac2_type1* (+ *)*
| *ltac2_type1*

ltac2_type1 ::= *ltac2_type0 qualid*
| *ltac2_type0*

ltac2_type0 ::= **(** *ltac2_type* (+ **,**) **)** *qualid* (?)
| *ltac2_typevar*
| **_**
| *qualid*

ltac2_typevar ::= **'** *ident*

The set of base types can be extended thanks to the usual ML type declarations such as algebraic datatypes and records.

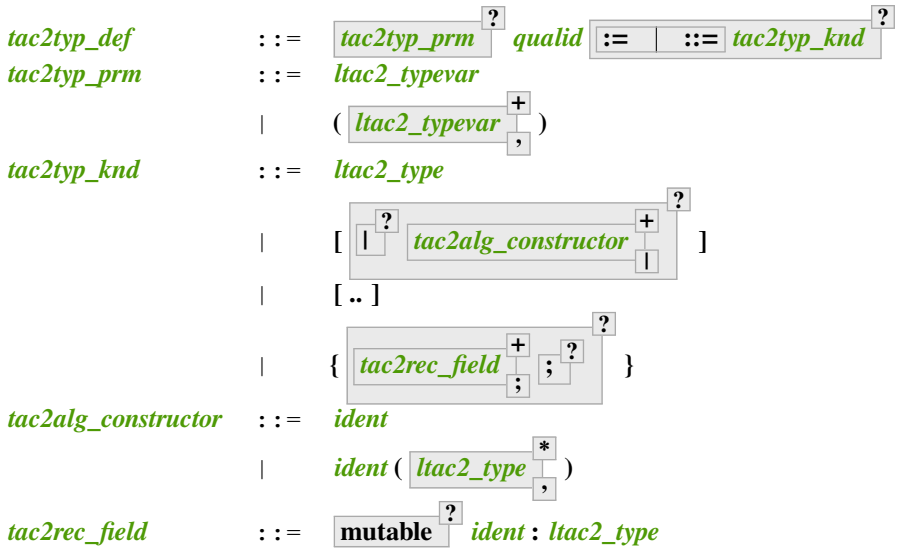
Built-in types include:

- `int`, machine integers (size not specified, in practice inherited from OCaml)
- `string`, mutable strings
- `'a array`, mutable arrays
- `exn`, exceptions
- `constr`, kernel-side terms
- `pattern`, term patterns
- `ident`, well-formed identifiers

Type declarations

One can define new types with the following commands.

Command: **Ltac2 Type** **rec** (?) *tac2typ_def* **with** *tac2typ_def* (*)



`:=` Defines a type with with an explicit set of constructors

`::=` Extends an existing open variant type, a special kind of variant type whose constructors are not statically defined, but can instead be extended dynamically. A typical example is the standard `exn` type for exceptions. Pattern matching on open variants must always include a catch-all clause. They can be extended with this form, in which case *tac2typ_knd* should be in the form `[` `]`.

Without `:=` | `::=` Defines an abstract type for use representing data from OCaml. Not for end users.

with *tac2typ_def* Permits definition of mutually recursive type definitions.

Each production of *tac2typ_knd* defines one of four possible kinds of definitions, respectively: alias, variant, open variant and record types.

Aliases are names for a given type expression and are transparently unfoldable to that expression. They cannot be recursive.

Variants are sum types defined by constructors and eliminated by pattern-matching. They can be recursive, but the `rec` flag must be explicitly set. Pattern matching must be exhaustive.

Open variants can be extended with additional constructors using the `::=` form.

Records are product types with named fields and eliminated by projection. Likewise they can be recursive if the `rec` flag is set.

Command: Ltac2 @ external *ident* **:** *ltac2_type* **:=** *string string*

Declares abstract terms. Frequently, these declare OCaml functions defined in Coq and give their type information. They can also declare data structures from OCaml. This command has no use for the end user.

This command supports the *deprecated* attribute.

APIs

Ltac2 provides over 150 API functions that provide various capabilities. These are declared with *Ltac2 external* in **lib/coq/user-contrib/Ltac2/*.v**. For example, `Message.print` defined in `Message.v` is used to print messages:

```
Message.print (Message.of_string "fully qualified calls").
    fully qualified calls

From Ltac2 Require Import Message.
print (of_string "unqualified calls").
    unqualified calls
```

Term Syntax

The syntax of the functional fragment is very close to that of Ltac1, except that it adds a true pattern-matching feature, as well as a few standard constructs from ML.

In practice, there is some additional syntactic sugar that allows the user to bind a variable and match on it at the same time, in the usual ML style.

There is dedicated syntax for list and array literals.

```
ltac2_expr          ::= ltac2_expr5 ; ltac2_expr
                    |   ltac2_expr5
ltac2_expr5         ::= fun tac2pat0+ (: ltac2_type)? => ltac2_expr
                    |   let rec? ltac2_let_clause with ltac2_let_clause* in ltac2_expr
                    |   ltac2_expr3
ltac2_let_clause    ::= tac2pat0+ (: ltac2_type)? := ltac2_expr
ltac2_expr3         ::= ltac2_expr2 +,
ltac2_expr2         ::= ltac2_expr1 :: ltac2_expr2
                    |   ltac2_expr1
ltac2_expr1         ::= ltac2_expr0 ltac2_expr0+
                    |   ltac2_expr0 .( qualid )
                    |   ltac2_expr0 .( qualid ) := ltac2_expr5
                    |   ltac2_expr0
tac2rec_fieldexpr   ::= qualid := ltac2_expr1
ltac2_expr0         ::= ( ltac2_expr )
                    |   ( ltac2_expr : ltac2_type )
                    |   ()
                    |   [ ltac2_expr5 *; ]
                    |   { (tac2rec_fieldexpr + ;)? }
                    |   ltac2_tactic_atom
ltac2_tactic_atom   ::= integer
                    |   string
                    |   qualid
                    |   @ ident
                    |   & lident
                    |   ' term
                    |   ltac2_quotations
```

The non-terminal `lident` designates identifiers starting with a lowercase letter.

'*term* is equivalent to **open_constr:(***term***)**.

Ltac2 Definitions

Command: Ltac2 **mutable**? **rec**? *tac2def_body* **with** *tac2def_body**

```
tac2def_body    ::= _ | ident tac2pat0* (: ltac2_type)? := ltac2_expr
```

This command defines a new global Ltac2 value. If one or more *tac2pat0* are specified, the new value is a function. This is a shortcut for one of the *ltac2_expr5* productions. For example: **Ltac2 foo a b := …** is equivalent to **Ltac2 foo := fun a b => …**.

The body of an Ltac2 definition is required to be a syntactical value that is, a function, a constant, a pure constructor recursively applied to values or a (non-recursive) let binding of a value in a value.

If `rec` is set, the tactic is expanded into a recursive binding.

If `mutable` is set, the definition can be redefined at a later stage (see below).

This command supports the *deprecated* attribute.

Command: **Ltac2 Set** *qualid* **as** *ident*? **:=** *ltac2_expr*

This command redefines a previous `mutable` definition. Mutable definitions act like dynamic binding, i.e. at runtime, the last defined value for this entry is chosen. This is useful for global flags and the like. The previous value of the binding can be optionally accessed using the `as` binding syntax.

Example: Dynamic nature of mutable cells

```
Ltac2 mutable x := true.
Ltac2 y := x.
Ltac2 Eval y.
    - : bool = true

Ltac2 Set x := false.
Ltac2 Eval y.
    - : bool = false
```

Example: Interaction with recursive calls

```
Ltac2 mutable rec f b := if b then 0 else f true.
Ltac2 Set f := fun b => if b then 1 else f true.
Ltac2 Eval (f false).
    - : int = 1

Ltac2 Set f as oldf := fun b => if b then  2 else oldf false.
Ltac2 Eval (f false).
    - : int = 2
```

In the definition, the `f` in the body is resolved statically because the definition is marked recursive. In the first re-definition, the `f` in the body is resolved dynamically. This is witnessed by the second re-definition.

Reduction

We use the usual ML call-by-value reduction, with an otherwise unspecified evaluation order. This is a design choice making it compatible with OCaml, if ever we implement native compilation. The expected equations are as follows:

```
(fun x => t) V ≡ t{x := V} (βv)

let x := V in t ≡ t{x := V} (let)

match C V₀ ... V□ with ... | C x₀ ... x□ => t | ... end ≡ t {x□ := V□} (ι)

(t any term, V values, C constructor)
```

Note that call-by-value reduction is already a departure from Ltac1 which uses heuristics to decide when to evaluate an expression. For instance, the following expressions do not evaluate the same way in Ltac1.

`foo (idtac; let x := 0 in bar)`

`foo (let x := 0 in bar)`

Instead of relying on the **`idtac`** idiom, we would now require an explicit thunk to not compute the argument, and **`foo`** would have e.g. type **`(unit -> unit) -> unit`**.

```
foo (fun () => let x := 0 in bar)
```

Typing

Typing is strict and follows the Hindley-Milner system. Unlike Ltac1, there are no type casts at runtime, and one has to resort to conversion functions. See notations though to make things more palatable.

In this setting, all the usual argument-free tactics have type **`unit -> unit`**, but one can return a value of type **`t`** thanks to terms of type **`unit -> t`**, or take additional arguments.

Effects

Effects in Ltac2 are straightforward, except that instead of using the standard IO monad as the ambient effectful world, Ltac2 is has a tactic monad.

Note that the order of evaluation of application is *not* specified and is implementation-dependent, as in OCaml.

We recall that the `Proofview.tactic` monad is essentially a IO monad together with backtracking state representing the proof state.

Intuitively a thunk of type **`unit -> 'a`** can do the following:

- It can perform non-backtracking IO like printing and setting mutable variables
- It can fail in a non-recoverable way
- It can use first-class backtracking. One way to think about this is that thunks are isomorphic to this type: **`(unit -> 'a) ~ (unit -> exn + ('a * (exn -> 'a)))`** i.e. thunks can produce a lazy list of results where each tail is waiting for a continuation exception.
- It can access a backtracking proof state, consisting among other things of the current evar assignment and the list of goals under focus.

We now describe more thoroughly the various effects in Ltac2.

Standard IO

The Ltac2 language features non-backtracking IO, notably mutable data and printing operations.

Mutable fields of records can be modified using the set syntax. Likewise, built-in types like `string` and `array` feature imperative assignment. See modules `String` and `Array` respectively.

A few printing primitives are provided in the `Message` module for displaying information to the user.

Fatal errors

The Ltac2 language provides non-backtracking exceptions, also known as *panics*, through the following primitive in module `Control`:

```
val throw : exn -> 'a
```

Unlike backtracking exceptions from the next section, this kind of error is never caught by backtracking primitives, that is, throwing an exception destroys the stack. This is codified by the following equation, where `E` is an evaluation context:

```
E[throw e] ≡ throw e

(e value)
```

There is currently no way to catch such an exception, which is a deliberate design choice. Eventually there might be a way to catch it and destroy all backtrack and return values.

Backtracking

In Ltac2, we have the following backtracking primitives, defined in the `Control` module:

```
Ltac2 Type 'a result := [ Val ('a) | Err (exn) ].

val zero : exn -> 'a
val plus : (unit -> 'a) -> (exn -> 'a) -> 'a
val case : (unit -> 'a) -> ('a * (exn -> 'a)) result
```

If one views thunks as lazy lists, then `zero` is the empty list and `plus` is list concatenation, while `case` is pattern-matching.

The backtracking is first-class, i.e. one can write **`plus (fun () => "x") (fun _ => "y") : string`** producing a backtracking string.

These operations are expected to satisfy a few equations, most notably that they form a monoid compatible with sequentialization.:

```
plus t zero ≡ t ()
plus (fun () => zero e) f ≡ f e
plus (plus t f) g ≡ plus t (fun e => plus (f e) g)

case (fun () => zero e) ≡ Err e
case (fun () => plus (fun () => t) f) ≡ Val (t,f)

let x := zero e in u ≡ zero e
let x := plus t f in u ≡ plus (fun () => let x := t in u) (fun e => let x := f e in u)

(t, u, f, g, e values)
```

Goals

A goal is given by the data of its conclusion and hypotheses, i.e. it can be represented as `[Γ ⊢ A]`.

The tactic monad naturally operates over the whole proofview, which may represent several goals, including none. Thus, there is no such thing as *the current goal*. Goals are naturally ordered, though.

It is natural to do the same in Ltac2, but we must provide a way to get access to a given goal. This is the role of the `enter` primitive, which applies a tactic to each currently focused goal in turn:

```
val enter : (unit -> unit) -> unit
```

It is guaranteed that when evaluating `enter f`, `f` is called with exactly one goal under focus. Note that `f` may be called several times, or never, depending on the number of goals under focus before the call to `enter`.

Accessing the goal data is then implicit in the Ltac2 primitives, and may panic if the invariants are not respected. The two essential functions for observing goals are given below.:

```
val hyp : ident -> constr
val goal : unit -> constr
```

The two above functions panic if there is not exactly one goal under focus. In addition, `hyp` may also fail if there is no hypothesis with the corresponding name.

Meta-programming

Overview

One of the major implementation issues of Ltac1 is the fact that it is never clear whether an object refers to the object world or the meta-world. This is an incredible source of slowness, as the interpretation must be aware of bound variables and must use heuristics to decide whether a variable is a proper one or referring to something in the Ltac context.

Likewise, in Ltac1, constr parsing is implicit, so that `foo 0` is not `foo` applied to the Ltac integer expression `0` (L_{tac} does have a notion of integers, though it is not first-class), but rather the Coq term `Datatypes.O`.

The implicit parsing is confusing to users and often gives unexpected results. Ltac2 makes these explicit using quoting and unquoting notation, although there are notations to do it in a short and elegant way so as not to be too cumbersome to the user.

Quotations

Built-in quotations

ltac2_quotations ::=
- **ident :** (*lident*)
- | **constr :** (*term*)
- | **open_constr :** (*term*)
- | **pat :** (*cpattern*)
- | **reference :** (**&** *ident* | *qualid*)
- | **ltac1 :** (*ltac1_expr_in_env*)
- | **ltac1val :** (*ltac1_expr_in_env*)

ltac1_expr_in_env ::=
- *ltac_expr*
- | *ident* * **|-** *ltac_expr*

The current implementation recognizes the following built-in quotations:

- `ident`, which parses identifiers (type `Init.ident`).
- `constr`, which parses Coq terms and produces an-evar free term at runtime (type `Init.constr`).
- `open_constr`, which parses Coq terms and produces a term potentially with holes at runtime (type `Init.constr` as well).
- `pat`, which parses Coq patterns and produces a pattern used for term matching (type `Init.pattern`).
- `reference` Qualified names are globalized at internalization into the corresponding global reference, while `&id` is turned into `Std.VarRef id`. This produces at runtime a `Std.reference`.
- `ltac1`, for calling Ltac1 code, described in *Simple API*.
- `ltac1val`, for manipulating Ltac1 values, described in *Low-level API*.

The following syntactic sugar is provided for two common cases:

- `@id` is the same as `ident:(id)`

- **'***term* is the same as **open_constr:(***term***)**

Strict vs. non-strict mode

Depending on the context, quotation-producing terms (i.e. `constr` or `open_constr`) are not internalized in the same way. There are two possible modes, the *strict* and the *non-strict* mode.

- In strict mode, all simple identifiers appearing in a term quotation are required to be resolvable statically. That is, they must be the short name of a declaration which is defined globally, excluding section variables and hypotheses. If this doesn't hold, internalization will fail. To work around this error, one has to specifically use the `&` notation.
- In non-strict mode, any simple identifier appearing in a term quotation which is not bound in the global environment is turned into a dynamic reference to a hypothesis. That is to say, internalization will succeed, but the evaluation of the term at runtime will fail if there is no such variable in the dynamic context.

Strict mode is enforced by default, such as for all Ltac2 definitions. Non-strict mode is only set when evaluating Ltac2 snippets in interactive proof mode. The rationale is that it is cumbersome to explicitly add `&` interactively, while it is expected that global tactics enforce more invariants on their code.

Term Antiquotations

Syntax

One can also insert Ltac2 code into Coq terms, similar to what is possible in Ltac1.

term += **ltac2:(** *ltac2_expr* **)**

Antiquoted terms are expected to have type `unit`, as they are only evaluated for their side-effects.

Semantics

A quoted Coq term is interpreted in two phases, internalization and evaluation.

- Internalization is part of the static semantics, that is, it is done at Ltac2 typing time.
- Evaluation is part of the dynamic semantics, that is, it is done when a term gets effectively computed by Ltac2.

Note that typing of Coq terms is a *dynamic* process occurring at Ltac2 evaluation time, and not at Ltac2 typing time.

Static semantics

During internalization, Coq variables are resolved and antiquotations are type checked as Ltac2 terms, effectively producing a `glob_constr` in Coq implementation terminology. Note that although it went through the type checking of **Ltac2**, the resulting term has not been fully computed and is potentially ill-typed as a runtime **Coq** term.

Example

The following term is valid (with type `unit -> constr`), but will fail at runtime:

```
Ltac2 myconstr () := constr:(nat -> 0).
```

Term antiquotations are type checked in the enclosing Ltac2 typing context of the corresponding term expression.

Example

The following will type check, with type `constr`.

```
let x := '0 in constr:(1 + ltac2:(exact x))
```

Beware that the typing environment of antiquotations is **not** expanded by the Coq binders from the term.

Example

The following Ltac2 expression will **not** type check:

```
`constr:(fun x : nat => ltac2:(exact x))`
`(* Error: Unbound variable 'x' *)`
```

There is a simple reason for that, which is that the following expression would not make sense in general.

```
constr:(fun x : nat => ltac2:(clear @x; exact x))
```

Indeed, a hypothesis can suddenly disappear from the runtime context if some other tactic pulls the rug from under you.

Rather, the tactic writer has to resort to the **dynamic** goal environment, and must write instead explicitly that she is accessing a hypothesis, typically as follows.

```
constr:(fun x : nat => ltac2:(exact (hyp @x)))
```

This pattern is so common that we provide dedicated Ltac2 and Coq term notations for it.

- `&x` as an Ltac2 expression expands to `hyp @x`.
- `&x` as a Coq constr expression expands to `ltac2:(Control.refine (fun () => hyp @x))`.

In the special case where Ltac2 antiquotations appear inside a Coq term notation, the notation variables are systematically bound in the body of the tactic expression with type `Ltac2.Init.preterm`. Such a type represents untyped syntactic Coq expressions, which can by typed in the current context using the `Ltac2.Constr.pretype` function.

Example

The following notation is essentially the identity.

```
Notation "[ x ]" := ltac2:(let x := Ltac2.Constr.pretype x in exact $x) (only parsing).
 ↪
```

Dynamic semantics

During evaluation, a quoted term is fully evaluated to a kernel term, and is in particular type checked in the current environment.

Evaluation of a quoted term goes as follows.

- The quoted term is first evaluated by the pretyper.
- Antiquotations are then evaluated in a context where there is exactly one goal under focus, with the hypotheses coming from the current environment extended with the bound variables of the term, and the resulting term is fed into the quoted term.

Relative orders of evaluation of antiquotations and quoted term are not specified.

For instance, in the following example, `tac` will be evaluated in a context with exactly one goal under focus, whose last hypothesis is `H : nat`. The whole expression will thus evaluate to the term `fun H : nat => H`.

```
let tac () := hyp @H in constr:(fun H : nat => ltac2:(tac ()))
```

Many standard tactics perform type checking of their argument before going further. It is your duty to ensure that terms are well-typed when calling such tactics. Failure to do so will result in non-recoverable exceptions.

Trivial Term Antiquotations

It is possible to refer to a variable of type `constr` in the Ltac2 environment through a specific syntax consistent with the antiquotations presented in the notation section.

term += **$***lident*

In a Coq term, writing `$x` is semantically equivalent to `ltac2:(Control.refine (fun () => x))`, up to re-typechecking. It allows to insert in a concise way an Ltac2 variable of type **`constr`** into a Coq term.

Match over terms

Ltac2 features a construction similar to Ltac1 *match* over terms, although in a less hard-wired way.

Tactic: ***ltac2_match_key ltac2_expr*$_{term}$ with *ltac2_match_list* end**

ltac2_match_key ::= **lazy_match!**
| **match!**
| **multi_match!**

ltac2_match_list ::= [**|**]? *ltac2_match_rule* (**|** *ltac2_match_rule*)+

ltac2_match_rule ::= *ltac2_match_pattern* **=>** *ltac2_expr*

ltac2_match_pattern ::= *cpattern*
| **context** *ident*? **[** *cpattern* **]**

Evaluates ***ltac2_expr***$_{term}$, which must yield a term, and matches it sequentially with the *ltac2_match_pattern*s, which may contain metavariables. When a match is found, metavariable values are substituted into ***ltac2_expr***, which is then applied.

Matching may continue depending on whether `lazy_match!`, `match!` or `multi_match!` is specified.

In the *ltac2_match_pattern*s, metavariables have the form **`?`*****ident***, whereas in the ***ltac2_expr***s, the question mark is omitted.

Matching is non-linear: if a metavariable occurs more than once, each occurrence must match the same expression. Expressions match if they are syntactically equal or are *α-convertible*. Matching is first-order except on variables of

the form **@?*ident*** that occur in the head position of an application. For these variables, matching is second-order and returns a functional term.

lazy_match! Causes the match to commit to the first matching branch rather than trying a new match if ***ltac2_expr*** fails. *Example*.

match! If ***ltac2_expr*** fails, continue matching with the next branch. Failures in subsequent tactics (after the `match!`) will not cause selection of a new branch. Examples *here* and *here*.

multi_match! If ***ltac2_expr*** fails, continue matching with the next branch. When a ***ltac2_expr*** succeeds for a branch, subsequent failures (after the `multi_match!`) causing consumption of all the successes of ***ltac2_expr*** trigger selection of a new matching branch. *Example*.

cpattern The syntax of *cpattern* is the same as that of *term*s, but it can contain pattern matching metavariables in the form **?*ident*** and **@?*ident***. _ can be used to match irrelevant terms.

Unlike Ltac1, Ltac2 **?id** metavariables only match closed terms.

There is also a special notation for second-order pattern matching: in an applicative pattern of the form **@?*ident*** ***ident*$_1$** … ***ident*$_n$**, the variable *ident* matches any complex expression with (possible) dependencies in the variables ***ident*$_i$** and returns a functional term of the form **fun** ***ident*$_1$** … ***ident*$_n$** **=>** ***term***.

context ***ident*** **?** **[** ***cpattern*** **]** Matches any term with a subterm matching *cpattern*. If there is a match and ***ident*** is present, it is assigned the "matched context", i.e. the initial term where the matched subterm is replaced by a hole. This hole in the matched context can be filled with the expression **Pattern.instantiate** ***ident*** ***cpattern***.

For *match!* and *multi_match!*, if the evaluation of the *ltac2_expr* fails, the next matching subterm is tried. If no further subterm matches, the next branch is tried. Matching subterms are considered from top to bottom and from left to right (with respect to the raw printing obtained by setting the *Printing All* flag). *Example*.

ltac2_expr The tactic to apply if the construct matches. Metavariable values from the pattern match are statically bound as Ltac2 variables in ***ltac2_expr*** before it is applied.

If ***ltac2_expr*** is a tactic with backtracking points, then subsequent failures after a *lazy_match!* or *multi_match!* (but not *match!*) can cause backtracking into ***ltac2_expr*** to select its next success.

Variables from the ***tac2pat1*** are statically bound in the body of the branch. Variables from the ***term*** pattern have values of type `constr`. Variables from the ***ident*** in the `context` construct have values of type `Pattern.context` (defined in `Pattern.v`).

Note that unlike Ltac1, only lowercase identifiers are valid as Ltac2 bindings. Ltac2 will report an error if one of the bound variables starts with an uppercase character.

The semantics of this construction are otherwise the same as the corresponding one from Ltac1, except that it requires the goal to be focused.

Example: Ltac2 Comparison of lazy_match! and match!

(Equivalent to this *Ltac1 example*.)

These lines define a `msg` tactic that's used in several examples as a more-succinct alternative to `print (to_string "...")`:

```
From Ltac2 Require Import Message.
Ltac2 msg x := print (of_string x).
```

In *lazy_match!*, if *ltac2_expr* fails, the *lazy_match!* fails; it doesn't look for further matches. In *match!*, if *ltac2_expr* fails in a matching branch, it will try to match on subsequent branches. Note that **'*term*** below is equivalent to **open_constr:(*term*)**.

```
Fail lazy_match! 'True with
| True => msg "branch 1"; fail
| _ => msg "branch 2"
end.
    branch 1
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)


match! 'True with
| True => msg "branch 1"; fail
| _ => msg "branch 2"
end.
    branch 1
    branch 2
```

Example: Ltac2 Comparison of match! and multi_match!

(Equivalent to this *Ltac1 example*.)

match! tactics are only evaluated once, whereas *multi_match!* tactics may be evaluated more than once if the following constructs trigger backtracking:

```
Fail match! 'True with
| True => msg "branch 1"
| _ => msg "branch 2"
end ;
msg "branch A"; fail.
    branch 1
    branch A
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)

Fail multi_match! 'True with
| True => msg "branch 1"
| _ => msg "branch 2"
end ;
msg "branch A"; fail.
    branch 1
    branch A
    branch 2
    branch A
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Match_failure
```

Example: Ltac2 Multiple matches for a "context" pattern.

(Equivalent to this *Ltac1 example*.)

Internally "x <> y" is represented as "(~ (x = y))", which produces the first match.

```
Ltac2 f2 t := match! t with
              | context [ (~ ?t) ] => print (of_constr t); fail
              | _ => ()
              end.

f2 constr:((~ True) <> (~ False)).
    ((~ True) = (~ False))
    True
    False
```

Match over goals

Tactic: *ltac2_match_key* **reverse**? **goal with** *goal_match_list* **end**

goal_match_list ::= (| *gmatch_rule*)+

gmatch_rule ::= *gmatch_pattern* => *ltac2_expr*

gmatch_pattern ::= [(*gmatch_hyp_pattern*),* |- *ltac2_match_pattern*]

gmatch_hyp_pattern ::= *name* : *ltac2_match_pattern*

Matches over goals, similar to Ltac1 `match goal`. Use this form to match hypotheses and/or goals in the local context. These patterns have zero or more subpatterns to match hypotheses followed by a subpattern to match the conclusion. Except for the differences noted below, this works the same as the corresponding *ltac2_match_key ltac2_expr* construct (see `match!`). Each current goal is processed independently.

Matching is non-linear: if a metavariable occurs more than once, each occurrence must match the same expression. Within a single term, expressions match if they are syntactically equal or *α-convertible*. When a metavariable is used across multiple hypotheses or across a hypothesis and the current goal, the expressions match if they are *convertible*.

gmatch_pattern,* Patterns to match with hypotheses. Each pattern must match a distinct hypothesis in order for the branch to match.

Hypotheses have the form *name* (:= *term*$_{binder}$)? : *type*. Currently Ltac2 doesn't allow matching on or capturing the value of *term*$_{binder}$. It only supports matching on the `name` and the `type`, for example `n : ?t`.

If there are multiple `gmatch_hyp_pattern`s in a branch, there may be multiple ways to match them to hypotheses. For `match! goal` and `multi_match! goal`, if the evaluation of the `ltac2_expr` fails, matching will continue with the next hypothesis combination. When those are exhausted, the next alternative from any `context` construct in the `ltac2_match_pattern`s is tried and then, when the context alternatives are exhausted, the next branch is tried. *Example*.

reverse Hypothesis matching for `gmatch_hyp_pattern`s normally begins by matching them from left to right, to hypotheses, last to first. Specifying `reverse` begins matching in the reverse order, from first to last. *Normal* and *reverse* examples.

|- *ltac2_match_pattern* A pattern to match with the current goal

Note that unlike Ltac1, only lowercase identifiers are valid as Ltac2 bindings. Ltac2 will report an error if you try to use a bound variable that starts with an uppercase character.

Variables from *gmatch_hyp_pattern* and *ltac2_match_pattern* are bound in the body of the branch. Their types are:

- `constr` for pattern variables appearing in a ***term***
- `Pattern.context` for variables binding a context
- `ident` for variables binding a hypothesis name.

The same identifier caveat as in the case of matching over constr applies, and this feature has the same semantics as in Ltac1.

Example: Ltac2 Matching hypotheses

(Equivalent to this *Ltac1 example*.)

Hypotheses are matched from the last hypothesis (which is by default the newest hypothesis) to the first until the *apply* succeeds.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros.
    1 goal

      A, B : Prop
      H : A
      H0 : B
      H1 : A
      ============================
      B

match! goal with
| [ h : _ |- _ ] => let h := Control.hyp h in print (of_constr h); apply $h
end.
    H1
    H0
    No more goals.
```

Example: Matching hypotheses with reverse

(Equivalent to this *Ltac1 example*.)

Hypotheses are matched from the first hypothesis to the last until the *apply* succeeds.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros.
    1 goal

      A, B : Prop
      H : A
      H0 : B
      H1 : A
```

(continues on next page)

(continued from previous page)

```
      ============================
      B

match! reverse goal with
| [ h : _ |- _ ] => let h := Control.hyp h in print (of_constr h); apply $h
end.
    A
    B
    H
    H0
    No more goals.
```

Example: Multiple ways to match a hypotheses

(Equivalent to this *Ltac1 example*.)

Every possible match for the hypotheses is evaluated until the right-hand side succeeds. Note that `h1` and `h2` are never matched to the same hypothesis. Observe that the number of permutations can grow as the factorial of the number of hypotheses and hypothesis patterns.

```
Goal forall A B : Prop, A -> B -> (A->B).
    1 goal

      ============================
      forall A B : Prop, A -> B -> A -> B

intros A B H.
    1 goal

      A, B : Prop
      H : A
      ============================
      B -> A -> B

match! goal with
| [ h1 : _, h2 : _ |- _ ] =>
   print (concat (of_string "match ")
         (concat (of_constr (Control.hyp h1))
         (concat (of_string " ")
         (of_constr (Control.hyp h2)))));
   fail
| [ |- _ ] => ()
end.
    match B H
    match A H
    match H B
    match A B
    match H A
    match B A
```

Match on values

Tactic: **match** *ltac2_expr5* **with** *ltac2_branches*? **end**
Matches a value, akin to the OCaml `match` construct. By itself, it doesn't cause backtracking as do the `*match*!` and `*match*! goal` constructs.

ltac2_branches ::= (|)? *tac2pat1* => *ltac2_expr* (|)+

tac2pat1 ::= *qualid* *tac2pat0*+
| *qualid*
| [ ]
| *tac2pat0* :: *tac2pat0*
| *tac2pat0*

tac2pat0 ::= _
| ()
| *qualid*
| (*atomic_tac2pat*?)

atomic_tac2pat ::= *tac2pat1* : *ltac2_type*
| *tac2pat1* , *tac2pat1* (,)*
| *tac2pat1*

Tactic: **if** $ltac2\_expr5_{test}$ **then** $ltac2\_expr5_{then}$ **else** $ltac2\_expr5_{else}$
Equivalent to a *match* on a boolean value. If the $ltac2\_expr5_{test}$ evaluates to true, $ltac2\_expr5_{then}$ is evaluated. Otherwise $ltac2\_expr5_{else}$ is evaluated.

Note: For now, deep pattern matching is not implemented.

Notations

Command: **Ltac2 Notation** *ltac2_scope*+ (: *natural*)? := *ltac2_expr*
Ltac2 Notation provides a way to extend the syntax of Ltac2 tactics. The left-hand side (before the `:=`) defines the syntax to recognize and gives formal parameter names for the syntactic values. ***integer*** is the level of the notation. When the notation is used, the values are substituted into the right-hand side. The right-hand side is typechecked when the notation is used, not when it is defined. In the following example, `x` is the formal parameter name and `constr` is its *syntactic class*. `print` and `of_constr` are functions provided by Coq through `Message.v`.

Example: Printing a `term`

```
From Ltac2 Require Import Message.
Ltac2 Notation "ex1" x(constr) := print (of_constr x).
ex1 (1 + 2).
    (1 + 2)
```

You can also print terms with a regular Ltac2 definition, but then the ***term*** must be in the quotation `constr:( … )`:

```
Ltac2 ex2 x := print (of_constr x).
ex2 constr:(1+2).
    (1 + 2)
```

There are also metasyntactic classes described *here* that combine other items. For example, `list1(constr, ",")` recognizes a comma-separated list of one or more *term*s.

Example: Parsing a list of `terms`

```
Ltac2 rec print_list x := match x with
| a :: t => print (of_constr a); print_list t
| [] => ()
end.
Ltac2 Notation "ex2" x(list1(constr, ",")) := print_list x.
ex2 1, 2, 3.
    1
    2
    3
```

An Ltac2 notation adds a parsing rule to the Ltac2 grammar, which is expanded to the provided body where every token from the notation is let-bound to the corresponding generated expression.

Example

Assume we perform:

```
Ltac2 Notation "foo" c(thunk(constr)) ids(list0(ident)) := Bar.f c ids.
```

Then the following expression

```
let y := @X in foo (nat -> nat) x $y
```

will expand at parsing time to

```
let y := @X in  let c := fun () => constr:(nat -> nat) with ids := [@x; y]
in Bar.f c ids
```

Beware that the order of evaluation of multiple let-bindings is not specified, so that you may have to resort to thunking to ensure that side-effects are performed at the right time.

This command supports the *deprecated* attribute.

Error: `Notation levels must range between 0 and 6.`
The level of a notation must be an integer between 0 and 6 inclusive.

Abbreviations

Command: **Ltac2 Notation** *string* | *lident* **:=** *ltac2_expr*

Introduces a special kind of notation, called an abbreviation, that does not add any parsing rules. It is similar in spirit to Coq abbreviations (see *Notation (abbreviation)*, insofar as its main purpose is to give an absolute name to a piece of pure syntax, which can be transparently referred to by this name as if it were a proper definition.

The abbreviation can then be manipulated just like a normal Ltac2 definition, except that it is expanded at internalization time into the given expression. Furthermore, in order to make this kind of construction useful in practice in an effectful language such as Ltac2, any syntactic argument to an abbreviation is thunked on-the-fly during its expansion.

For instance, suppose that we define the following.

```
Ltac2 Notation foo := fun x => x ().
```

Then we have the following expansion at internalization time.

```
foo 0 ↦ (fun x => x ()) (fun _ => 0)
```

Note that abbreviations are not type checked at all, and may result in typing errors after expansion.

This command supports the *deprecated* attribute.

Defining tactics

Built-in tactics (those defined in OCaml code in the Coq executable) and Ltac1 tactics, which are defined in `.v` files, must be defined through notations. (Ltac2 tactics can be defined with *Ltac2*.

Notations for many but not all built-in tactics are defined in `Notations.v`, which is automatically loaded with Ltac2. The Ltac2 syntax for these tactics is often identical or very similar to the tactic syntax described in other chapters of this documentation. These notations rely on tactic functions declared in `Std.v`. Functions corresponding to some built-in tactics may not yet be defined in the Coq executable or declared in `Std.v`. Adding them may require code changes to Coq or defining workarounds through Ltac1 (described below).

Two examples of syntax differences:

- There is no notation defined that's equivalent to **intros until** *ident* | *natural*. There is, however, already an `intros_until` tactic function defined `Std.v`, so it may be possible for a user to add the necessary notation.
- The built-in `simpl` tactic in Ltac1 supports the use of scope keys in delta flags, e.g. **simpl ["+"%nat]** which is not accepted by Ltac2. This is because Ltac2 uses a different definition for *delta_reductions*; compare it to *ltac2_delta_reductions*. This also affects *compute*.

Ltac1 tactics are not automatically available in Ltac2. (Note that some of the tactics described in the documentation are defined with Ltac1.) You can make them accessible in Ltac2 with commands similar to the following:

```
From Coq Require Import Lia.
Local Ltac2 lia_ltac1 () := ltac1:(lia).
Ltac2 Notation "lia" := lia_ltac1 ().
```

A similar approach can be used to access missing built-in tactics. See *Simple API* for an example that passes two parameters to a missing build-in tactic.

Syntactic classes

The simplest syntactic classes in Ltac2 notations represent individual nonterminals from the Coq grammar. Only a few selected nonterminals are available as syntactic classes. In addition, there are metasyntactic operations for describing more complex syntax, such as making an item optional or representing a list of items. When parsing, each syntactic class expression returns a value that's bound to a name in the notation definition.

Syntactic classes are described with a form of S-expression:

ltac2_scope ::= *string*
| *integer*
| *name*
| *name* (*ltac2_scope* [+ ,])

Metasyntactic operations that can be applied to other syntactic classes are:

opt(*ltac2_scope***)** Parses an optional *ltac2_scope*. The associated value is either **None** or enclosed in **Some**

list1(*ltac2_scope* [**, *string***]?**)** Parses a list of one or more *ltac2_scope*s. If *string* is specified, items must be separated by *string*.

list0(*ltac2_scope* [**, *string***]?**)** Parses a list of zero or more *ltac2_scope*s. If *string* is specified, items must be separated by *string*. For zero items, the associated value is an empty list.

seq(*ltac2_scope* [+ ,] **)** Parses the *ltac2_scope*s in order. The associated value is a tuple, omitting *ltac2_scope*s that are *string*s. `self` and `next` are not permitted within `seq`.

The following classes represent nonterminals with some special handling. The table further down lists the classes that that are handled plainly.

constr [**(** *scope_key* [+ ,] **)**]? Parses a *term*. If specified, the *scope_key*s are used to interpret the term (as described in *Local interpretation rules for notations*). The last *scope_key* is the top of the scope stack that's applied to the *term*.

open_constr [**(** *scope_key* [+ ,] **)**]? Parses an open *term*. Like **constr** above, this class accepts a list of notation scopes with the same effects.

ident Parses *ident* or **$***ident*. The first form returns **ident:(***ident***)**, while the latter form returns the variable ***ident***.

string Accepts the specified string that is not a keyword, returning a value of `()`.

keyword(*string***)** Accepts the specified string that is a keyword, returning a value of `()`.

terminal(*string***)** Accepts the specified string whether it's a keyword or not, returning a value of `()`.

tactic [**(***integer***)**]? Parses an *ltac2_expr*. If *integer* is specified, the construct parses a **ltac2_expr***integer*, for example `tactic(5)` parses *ltac2_expr5*. `tactic(6)` parses *ltac2_expr*. *integer* must be in the range `0 .. 6`.

You can also use `tactic` to accept an *integer* or a *string*, but there's no syntactic class that accepts *only* an *integer* or a *string*.

self parses an Ltac2 expression at the current level and returns it as is.

next parses an Ltac2 expression at the next level and returns it as is.

thunk(*ltac2_scope***)** Used for semantic effect only, parses the same as *ltac2_scope*. If **e** is the parsed expression for *ltac2_scope*, thunk returns **fun () => e**.

pattern parses a *cpattern*

A few syntactic classes contain antiquotation features. For the sake of uniformity, all antiquotations are introduced by the syntax **$***lident*.

A few other specific syntactic classes exist to handle Ltac1-like syntax, but their use is discouraged and they are thus not documented.

For now there is no way to declare new syntactic classes from the Ltac2 side, but this is planned.

Other nonterminals that have syntactic classes are listed here.

Syntactic class name	Nonterminal	Similar non-Ltac2 syntax
intropatterns	*ltac2_intropatterns*	*intropattern_list*
intropattern	*ltac2_simple_intropattern*	*simple_intropattern*
ident	*ident_or_anti*	*ident*
destruction_arg	*ltac2_destruction_arg*	destruction_arg
with_bindings	*q_with_bindings*	**with** ***bindings*** ?
bindings	*ltac2_bindings*	*bindings*
reductions	*ltac2_reductions*	*reductions*
reference	*refglobal*	*reference*
clause	*ltac2_clause*	*occurrences*
occurrences	*q_occurrences*	**at** ***occs_nums*** ?
induction_clause	*ltac2_induction_clause*	induction_clause
conversion	*ltac2_conversion*	
rewriting	*ltac2_oriented_rewriter*	*oriented_rewriter*
dispatch	*ltac2_for_each_goal*	*for_each_goal*
hintdb	*hintdb*	*hintbases*
move_location	*move_location*	where
pose	*pose*	bindings_with_parameters
assert	*assertion*	**(** ***ident*** **:=** ***term*** **)**
constr_matching	*ltac2_match_list*	See *match*
goal_matching	*goal_match_list*	See *match goal*

Here is the syntax for the **q_*** nonterminals:

ltac2_intropatterns ::= *nonsimple_intropattern* *

nonsimple_intropattern ::= *****
| ******
| *ltac2_simple_intropattern*

ltac2_simple_intropattern ::= *ltac2_naming_intropattern*
| _
| *ltac2_or_and_intropattern*
| *ltac2_equality_intropattern*

ltac2_or_and_intropattern ::= [*ltac2_intropatterns* + |]
| ()
| (*ltac2_simple_intropattern* + ,)
| (*ltac2_simple_intropattern* + &)

ltac2_equality_intropattern ::= ->
| <-
| [= *ltac2_intropatterns*]

ltac2_naming_intropattern ::= ? *lident*
| ?$ *lident*
| ?
| *ident_or_anti*

ident_or_anti ::= *lident*
| $ *ident*

ltac2_destruction_arg ::= *natural*
| *lident*
| *ltac2_constr_with_bindings*

ltac2_constr_with_bindings ::= *term* with *ltac2_bindings* ?

q_with_bindings ::= with *ltac2_bindings* ?

ltac2_bindings ::= *ltac2_simple_binding* +
| *term* +

ltac2_simple_binding ::= (*qhyp* := *term*)

qhyp ::= $ *ident*
| *natural*
| *lident*

ltac2_reductions ::= *ltac2_red_flag* +
| *ltac2_delta_reductions* ?

ltac2_red_flag ::= beta
| iota
| match
| fix
| cofix
| zeta
| delta *ltac2_delta_reductions* ?

ltac2_delta_reductions ::= - ? [*refglobal* +]

refglobal ::= & *ident*
| *qualid*
| $ *ident*

ltac2_clause : := **in** *ltac2_in_clause*
| **at** *ltac2_occs_nums*

ltac2_in_clause : := * *ltac2_occs*?
| * |- *ltac2_concl_occ*?
| *ltac2_hypident_occ* (,)* |- *ltac2_concl_occ*?

q_occurrences : := *ltac2_occs*?
ltac2_occs : := **at** *ltac2_occs_nums*
ltac2_occs_nums : := -? *natural* | $ *ident* +
ltac2_concl_occ : := * *ltac2_occs*?
ltac2_hypident_occ : := *ltac2_hypident* *ltac2_occs*?
ltac2_hypident : := *ident_or_anti*
| **(type of** *ident_or_anti* **)**
| **(value of** *ident_or_anti* **)**

ltac2_induction_clause : := *ltac2_destruction_arg* *ltac2_as_or_and_ipat*? *ltac2_eqn_ipat*?
ltac2_clause?
ltac2_as_or_and_ipat : := **as** *ltac2_or_and_intropattern*
ltac2_eqn_ipat : := **eqn :** *ltac2_naming_intropattern*

ltac2_conversion : := *term*
| *term* **with** *term*

ltac2_oriented_rewriter : := (-> | <-)? *ltac2_rewriter*
ltac2_rewriter : := *natural*? (? | !)? *ltac2_constr_with_bindings*

ltac2_for_each_goal : := *ltac2_goal_tactics*
| *ltac2_goal_tactics*? | *ltac2_expr*? .. | *ltac2_goal_tactics*?
ltac2_goal_tactics : := (*ltac2_expr*? |)*

hintdb : := *
| *ident_or_anti* +

move_location : := **at top**
| **at bottom**
| **after** *ident_or_anti*
| **before** *ident_or_anti*

pose : := **(** *ident_or_anti* **:=** *term* **)**
| *term* *ltac2_as_name*?
ltac2_as_name : := **as** *ident_or_anti*

assertion ::= (*ident_or_anti* **:=** *term*)
| (*ident_or_anti* **:** *term*) *ltac2_by_tactic*?
| *term* *ltac2_as_ipat*? *ltac2_by_tactic*?
ltac2_as_ipat ::= **as** *ltac2_simple_intropattern*
ltac2_by_tactic ::= **by** *ltac2_expr*

Evaluation

Ltac2 features a toplevel loop that can be used to evaluate expressions.

Command: `Ltac2 Eval` *`ltac2_expr`*
This command evaluates the term in the current proof if there is one, or in the global environment otherwise, and displays the resulting value to the user together with its type. This command is pure in the sense that it does not modify the state of the proof, and in particular all side-effects are discarded.

Debug

Flag: `Ltac2 Backtrace`
When this *flag* is set, toplevel failures will be printed with a backtrace.

Compatibility layer with Ltac1

Ltac1 from Ltac2

Simple API

One can call Ltac1 code from Ltac2 by using the **`ltac1:(`*`ltac1_expr_in_env`*`)`** quotation. See *Built-in quotations*. It parses a Ltac1 expression, and semantics of this quotation is the evaluation of the corresponding code for its side effects. In particular, it cannot return values, and the quotation has type **`unit`**.

Ltac1 **cannot** implicitly access variables from the Ltac2 scope, but this can be done with an explicit annotation on the **`ltac1:(`***`ident`***`*`** **`|-`** *`ltac_expr`*`)` quotation. See *Built-in quotations*. For example:

```
Local Ltac2 replace_with (lhs: constr) (rhs: constr) :=
  ltac1:(lhs rhs |- replace lhs with rhs) (Ltac1.of_constr lhs) (Ltac1.of_constr rhs).
Ltac2 Notation "replace" lhs(constr) "with" rhs(constr) := replace_with lhs rhs.
```

The return type of this expression is a function of the same arity as the number of identifiers, with arguments of type `Ltac2.Ltac1.t` (see below). This syntax will bind the variables in the quoted Ltac1 code as if they had been bound from Ltac1 itself. Similarly, the arguments applied to the quotation will be passed at runtime to the Ltac1 code.

Low-level API

There exists a lower-level FFI into Ltac1 that is not recommended for daily use, which is available in the `Ltac2.Ltac1` module. This API allows to directly manipulate dynamically-typed Ltac1 values, either through the function calls, or using the `ltac1val` quotation. The latter parses the same as `ltac1`, but has type `Ltac2.Ltac1.t` instead of `unit`, and dynamically behaves as an Ltac1 thunk, i.e. `ltac1val:(foo)` corresponds to the tactic closure that Ltac1 would generate from `idtac; foo`.

Due to intricate dynamic semantics, understanding when Ltac1 value quotations focus is very hard. This is why some functions return a continuation-passing style value, as it can dispatch dynamically between focused and unfocused behaviour.

The same mechanism for explicit binding of variables as described in the previous section applies.

Ltac2 from Ltac1

Same as above by switching Ltac1 by Ltac2 and using the `ltac2` quotation instead.

ltac_expr += **ltac2 :** (*ltac2_expr*)

| **ltac2 :** (*ident* + **|-** *ltac2_expr*)

The typing rules are dual, that is, the optional identifiers are bound with type `Ltac2.Ltac1.t` in the Ltac2 expression, which is expected to have type unit. The value returned by this quotation is an Ltac1 function with the same arity as the number of bound variables.

Note that when no variables are bound, the inner tactic expression is evaluated eagerly, if one wants to use it as an argument to a Ltac1 function, one has to resort to the good old **`idtac; ltac2:(foo)`** trick. For instance, the code below will fail immediately and won't print anything.

```
From Ltac2 Require Import Ltac2.
Set Default Proof Mode "Classic".

Ltac mytac tac := idtac "I am being evaluated"; tac.
    mytac is defined


Goal True.
    1 goal

      ============================
      True

Proof.
(* Doesn't print anything *)
Fail mytac ltac2:(fail).
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)

(* Prints and fails *)
Fail mytac ltac:(idtac; ltac2:(fail)).
    I am being evaluated
    The command has indeed failed with message:
    Uncaught Ltac2 exception: Tactic_failure (None)
```

In any case, the value returned by the fully applied quotation is an unspecified dummy Ltac1 closure and should not be further used.

Switching between Ltac languages

We recommend using the `Default Proof Mode` option to switch between tactic languages with a proof-based granularity. This allows to incrementally port the proof scripts.

Transition from Ltac1

Owing to the use of a lot of notations, the transition should not be too difficult. In particular, it should be possible to do it incrementally. That said, we do *not* guarantee it will be a blissful walk either. Hopefully, owing to the fact Ltac2 is typed, the interactive dialogue with Coq will help you.

We list the major changes and the transition strategies hereafter.

Syntax changes

Due to conflicts, a few syntactic rules have changed.

- The dispatch tactical **`tac; [foo|bar]`** is now written **`tac > [foo|bar]`**.
- Levels of a few operators have been revised. Some tacticals now parse as if they were normal functions. Parentheses are now required around complex arguments, such as abstractions. The tacticals affected are: **`try`**, **`repeat`**, **`do`**, **`once`**, **`progress`**, **`time`**, **`abstract`**.
- **`idtac`** is no more. Either use **`()`** if you expect nothing to happen, **`(fun () => ())`** if you want a thunk (see next section), or use printing primitives from the **`Message`** module if you want to display something.

Tactic delay

Tactics are not magically delayed anymore, neither as functions nor as arguments. It is your responsibility to thunk them beforehand and apply them at the call site.

A typical example of a delayed function:

```
Ltac foo := blah.
```

becomes

```
Ltac2 foo () := blah.
```

All subsequent calls to `foo` must be applied to perform the same effect as before.

Likewise, for arguments:

```
Ltac bar tac := tac; tac; tac.
```

becomes

```
Ltac2 bar tac := tac (); tac (); tac ().
```

We recommend the use of syntactic notations to ease the transition. For instance, the first example can alternatively be written as:

```
Ltac2 foo0 () := blah. Ltac2 Notation foo := foo0 ().
```

This allows to keep the subsequent calls to the tactic as-is, as the expression `foo` will be implicitly expanded everywhere into `foo0 ()`. Such a trick also works for arguments, as arguments of syntactic notations are implicitly thunked. The second example could thus be written as follows.

```
Ltac2 bar0 tac := tac (); tac (); tac (). Ltac2 Notation bar := bar0.
```

Variable binding

Ltac1 relies on complex dynamic trickery to be able to tell apart bound variables from terms, hypotheses, etc. There is no such thing in Ltac2, as variables are recognized statically and other constructions do not live in the same syntactic world. Due to the abuse of quotations, it can sometimes be complicated to know what a mere identifier represents in a tactic expression. We recommend tracking the context and letting the compiler print typing errors to understand what is going on.

We list below the typical changes one has to perform depending on the static errors produced by the typechecker.

In Ltac expressions

Error: `Unbound value | constructor X`

- if `X` is meant to be a term from the current static environment, replace the problematic use by `'X`.
- if `X` is meant to be a hypothesis from the local context, replace the problematic use by `&X`.

In quotations

Error: `The reference X was not found in the current environment`

- if `X` is meant to be a tactic expression bound by a Ltac2 let or function, replace the problematic use by `$X`.
- if `X` is meant to be a hypothesis from the local context, replace the problematic use by `&X`.

Exception catching

Ltac2 features a proper exception-catching mechanism. For this reason, the Ltac1 mechanism relying on `fail` taking integers, and tacticals decreasing it, has been removed. Now exceptions are preserved by all tacticals, and it is your duty to catch them and re-raise them as needed.

CHAPTER

FOUR

USING COQ

4.1 Libraries and plugins

Coq is distributed with a standard library and a set of internal plugins (most of which provide tactics that have already been presented in *Basic proof writing*). This chapter presents this standard library and some of these internal plugins which provide features that are not tactics.

In addition, Coq has a rich ecosystem of external libraries and plugins. These libraries and plugins can be browsed online through the Coq Package Index[55] and installed with the opam package manager[56].

4.1.1 The Coq library

The Coq library has two parts:

- The prelude: definitions and theorems for the most commonly used elementary logical notions and data types. Coq normally loads these files automatically when it starts.
- The standard library: general-purpose libraries with definitions and theorems for sets, lists, sorting, arithmetic, etc. To use these files, users must load them explicitly with the `Require` command (see *Compiled files*)

There are also many libraries provided by Coq users' community. These libraries and developments are available for download at http://coq.inria.fr (see *Users' contributions*).

This chapter briefly reviews the Coq libraries whose contents can also be browsed at http://coq.inria.fr/stdlib/.

The prelude

This section lists the basic notions and results which are directly available in the standard Coq system. Most of these constructions are defined in the `Prelude` module in directory `theories/Init` in the Coq root directory; this includes the modules `Notations`, `Logic`, `Datatypes`, `Specif`, `Peano`, `Wf` and `Tactics`. Module `Logic_Type` also makes it in the initial state.

[55] https://coq.inria.fr/opam/www/
[56] https://coq.inria.fr/opam-using.html

Notations

This module defines the parsing and pretty-printing of many symbols (infixes, prefixes, etc.). However, it does not assign a meaning to these notations. The purpose of this is to define and fix once for all the precedence and associativity of very common notations. The main notations fixed in the initial state are :

Notation	Precedence	Associativity
`_ -> _`	99	right
`_ <-> _`	95	no
`_ \/ _`	85	right
`_ /\ _`	80	right
`~ _`	75	right
`_ = _`	70	no
`_ = _ = _`	70	no
`_ = _ :> _`	70	no
`_ <> _`	70	no
`_ <> _ :> _`	70	no
`_ < _`	70	no
`_ > _`	70	no
`_ <= _`	70	no
`_ >= _`	70	no
`_ < _ < _`	70	no
`_ < _ <= _`	70	no
`_ <= _ < _`	70	no
`_ <= _ <= _`	70	no
`_ + _`	50	left
`_ || _`	50	left
`_ - _`	50	left
`_ * _`	40	left
`_ _`	40	left
`_ / _`	40	left
`- _`	35	right
`/ _`	35	right
`_ ^ _`	30	right

Logic

The basic library of Coq comes with the definitions of standard (intuitionistic) logical connectives (they are defined as inductive constructions). They are equipped with an appealing syntax enriching the subclass *form* of the syntactic class *term*. The constructs for `form` are:

True	True
False	False
~ *form*	not
form **/** *form*	and
form **\/** *form*	or
form **->** *form*	primitive implication
form **<->** *form*	iff
forall *ident* **:** *type*, *form*	primitive for all
exists *ident* *specif*? **,** *form*	ex
exists2 *ident* *specif*? **,** *form* **&** *form*	ex2
term **=** *term*	eq
term **=** *term* **:>** *specif*	eq

Note: Implication is not defined but primitive (it is a non-dependent product of a proposition over another proposition). There is also a primitive universal quantification (it is a dependent product over a proposition). The primitive universal quantification allows both first-order and higher-order quantification.

Propositional Connectives

First, we find propositional calculus connectives:

```
Inductive True : Prop := I.
Inductive False :  Prop := .
Definition not (A: Prop) := A -> False.
Inductive and (A B:Prop) : Prop := conj (_:A) (_:B).
Section Projections.
 Variables A B : Prop.
 Theorem proj1 : A /\ B -> A.
 Theorem proj2 : A /\ B -> B.
End Projections.
Inductive or (A B:Prop) : Prop :=
| or_introl (_:A)
| or_intror (_:B).
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).
Definition IF_then_else (P Q R:Prop) := P /\ Q \/ ~ P /\ R.
```

Quantifiers

Then we find first-order quantifiers:

```
Definition all (A:Set) (P:A -> Prop) := forall x:A, P x.
Inductive ex (A: Set) (P:A -> Prop) : Prop :=
 ex_intro (x:A) (_:P x).
Inductive ex2 (A:Set) (P Q:A -> Prop) : Prop :=
 ex_intro2 (x:A) (_:P x) (_:Q x).
```

The following abbreviations are allowed:

`exists x:A, P`	`ex A (fun x:A => P)`
`exists x, P`	`ex _ (fun x => P)`
`exists2 x:A, P & Q`	`ex2 A (fun x:A => P) (fun x:A => Q)`
`exists2 x, P & Q`	`ex2 _ (fun x => P) (fun x => Q)`

The type annotation `:A` can be omitted when `A` can be synthesized by the system.

Equality

Then, we find equality, defined as an inductive relation. That is, given a type `A` and an `x` of type `A`, the predicate `(eq A x)` is the smallest one which contains `x`. This definition, due to Christine Paulin-Mohring, is equivalent to define `eq` as the smallest reflexive relation, and it is also equivalent to Leibniz' equality.

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
  eq_refl : eq A x x.
```

Lemmas

Finally, a few easy lemmas are provided.

```
Theorem absurd : forall A C:Prop, A -> ~ A -> C.
Section equality.
Variables A B : Type.
Variable f : A -> B.
Variables x y z : A.
Theorem eq_sym : x = y -> y = x.
Theorem eq_trans : x = y -> y = z -> x = z.
Theorem f_equal : x = y -> f x = f y.
Theorem not_eq_sym : x <> y -> y <> x.
End equality.
Definition eq_ind_r :
 forall (A:Type) (x:A) (P:A->Prop), P x -> forall y:A, y = x -> P y.
Definition eq_rec_r :
 forall (A:Type) (x:A) (P:A->Set), P x -> forall y:A, y = x -> P y.
Definition eq_rect_r :
 forall (A:Type) (x:A) (P:A->Type), P x -> forall y:A, y = x -> P y.
Hint Immediate eq_sym not_eq_sym : core.
```

The theorem `f_equal` is extended to functions with two to five arguments. The theorem are names `f_equal2`, `f_equal3`, `f_equal4` and `f_equal5`. For instance `f_equal3` is defined the following way.

```
Theorem f_equal3 :
 forall (A1 A2 A3 B:Type) (f:A1 -> A2 -> A3 -> B)
   (x1 y1:A1) (x2 y2:A2) (x3 y3:A3),
   x1 = y1 -> x2 = y2 -> x3 = y3 -> f x1 x2 x3 = f y1 y2 y3.
```

Datatypes

In the basic library, we find in `Datatypes.v` the definition of the basic data-types of programming, defined as inductive constructions over the sort `Set`. Some of them come with a special syntax shown below (this syntax table is common with the next section *Specification*). The constructs for `specif` are:

specif* * *specif	prod
specif* + *specif	sum
***specif* + { *specif* }**	sumor
{ *specif* } + { *specif* }	sumbool
{ *ident* : *specif* | *form* }	sig
{ *ident* : *specif* | *form* & *form* }	sig2
{ *ident* : *specif* & *specif* }	sigT
{ *ident* : *specif* & *specif* & *specif* }	sigT2

The notation for pairs (elements of type prod) is: **(*term*, *term*)**

Programming

```
Inductive unit : Set := tt.
Inductive bool : Set := true | false.
Inductive nat : Set := O | S (n:nat).
Inductive option (A:Set) : Set := Some (_:A) | None.
Inductive identity (A:Type) (a:A) : A -> Type :=
  refl_identity : identity A a a.
```

Note that zero is the letter `O`, and *not* the numeral `0`.

The predicate `identity` is logically equivalent to equality but it lives in sort `Type`. It is mainly maintained for compatibility.

We then define the disjoint sum of `A+B` of two sets `A` and `B`, and their product `A*B`.

```
Inductive sum (A B:Set) : Set := inl (_:A) | inr (_:B).
Inductive prod (A B:Set) : Set := pair (_:A) (_:B).
Section projections.
Variables A B : Set.
Definition fst (H: prod A B) := match H with
                                | pair _ _ x y => x
                                end.
Definition snd (H: prod A B) := match H with
                                | pair _ _ x y => y
                                end.
End projections.
```

Some operations on `bool` are also provided: `andb` (with infix notation `&&`), `orb` (with infix notation `||`), `xorb`, `implb` and `negb`.

Specification

The following notions defined in module `Specif.v` allow to build new data-types and specifications. They are available with the syntax shown in the previous section *Datatypes*.

For instance, given `A:Type` and `P:A->Prop`, the construct `{x:A | P x}` (in abstract syntax `(sig A P)`) is a `Type`. We may build elements of this set as `(exist x p)` whenever we have a witness `x:A` with its justification `p:P x`.

From such a `(exist x p)` we may in turn extract its witness `x:A` (using an elimination construct such as `match`) but *not* its justification, which stays hidden, like in an abstract data-type. In technical terms, one says that `sig` is a *weak (dependent) sum*. A variant `sig2` with two predicates is also provided.

```
Inductive sig (A:Set) (P:A -> Prop) : Set := exist (x:A) (_:P x).
Inductive sig2 (A:Set) (P Q:A -> Prop) : Set :=
  exist2 (x:A) (_:P x) (_:Q x).
```

A *strong (dependent) sum* `{x:A & P x}` may be also defined, when the predicate `P` is now defined as a constructor of types in `Type`.

```
Inductive sigT (A:Type) (P:A -> Type) : Type := existT (x:A) (_:P x).
Section Projections2.
Variable A : Type.
Variable P : A -> Type.
Definition projT1 (H:sigT A P) := let (x, h) := H in x.
Definition projT2 (H:sigT A P) :=
 match H return P (projT1 H) with
  existT _ _ x h => h
 end.
End Projections2.
Inductive sigT2 (A: Type) (P Q:A -> Type) : Type :=
  existT2 (x:A) (_:P x) (_:Q x).
```

A related non-dependent construct is the constructive sum `{A}+{B}` of two propositions `A` and `B`.

```
Inductive sumbool (A B:Prop) : Set := left (_:A) | right (_:B).
```

This `sumbool` construct may be used as a kind of indexed boolean data-type. An intermediate between `sumbool` and `sum` is the mixed `sumor` which combines `A:Set` and `B:Prop` in the construction `A+{B}` in `Set`.

```
Inductive sumor (A:Set) (B:Prop) : Set :=
| inleft (_:A)
| inright (_:B).
```

We may define variants of the axiom of choice, like in Martin-Löf's Intuitionistic Type Theory.

```
Lemma Choice :
 forall (S S':Set) (R:S -> S' -> Prop),
  (forall x:S, {y : S' | R x y}) ->
  {f : S -> S' | forall z:S, R z (f z)}.
Lemma Choice2 :
 forall (S S':Set) (R:S -> S' -> Set),
  (forall x:S, {y : S' &  R x y}) ->
   {f : S -> S' &  forall z:S, R z (f z)}.
Lemma bool_choice :
 forall (S:Set) (R1 R2:S -> Prop),
  (forall x:S, {R1 x} + {R2 x}) ->
```

(continues on next page)

(continued from previous page)

```
  {f : S -> bool |
   forall x:S, f x = true /\ R1 x \/ f x = false /\ R2 x}.
```

The next construct builds a sum between a data-type `A:Type` and an exceptional value encoding errors:

```
Definition Exc := option.
Definition value := Some.
Definition error := None.
```

This module ends with theorems, relating the sorts `Set` or `Type` and `Prop` in a way which is consistent with the realizability interpretation.

```
Definition except := False_rec.
Theorem absurd_set : forall (A:Prop) (C:Set), A -> ~ A -> C.
Theorem and_rect2 :
 forall (A B:Prop) (P:Type), (A -> B -> P) -> A /\ B -> P.
```

Basic Arithmetic

The basic library includes a few elementary properties of natural numbers, together with the definitions of predecessor, addition and multiplication, in module `Peano.v`. It also provides a scope `nat_scope` gathering standard notations for common operations (`+`, `*`) and a decimal notation for numbers, allowing for instance to write `3` for `S (S (S O))`. This also works on the left hand side of a `match` expression (see for example section *refine*). This scope is opened by default.

Example

The following example is not part of the standard library, but it shows the usage of the notations:

```
Fixpoint even (n:nat) : bool :=
 match n with
 | 0 => true
 | 1 => false
 | S (S n) => even n
 end.
```

Now comes the content of module `Peano`:

```
Theorem eq_S : forall x y:nat, x = y -> S x = S y.
Definition pred (n:nat) : nat :=
 match n with
 | 0 => 0
 | S u => u
 end.
Theorem pred_Sn : forall m:nat, m = pred (S m).
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Hint Immediate eq_add_S : core.
Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Definition IsSucc (n:nat) : Prop :=
 match n with
 | 0 => False
 | S p => True
 end.
```

(continues on next page)

(continued from previous page)

```
Theorem O_S : forall n:nat, 0 <> S n.
Theorem n_Sn : forall n:nat, n <> S n.
Fixpoint plus (n m:nat) {struct n} : nat :=
 match n with
 | 0 => m
 | S p => S (p + m)
 end
where "n + m" := (plus n m) : nat_scope.
Lemma plus_n_O : forall n:nat, n = n + 0.
Lemma plus_n_Sm : forall n m:nat, S (n + m) = n + S m.
Fixpoint mult (n m:nat) {struct n} : nat :=
 match n with
 | 0 => 0
 | S p => m + p * m
 end
where "n * m" := (mult n m) : nat_scope.
Lemma mult_n_O : forall n:nat, 0 = n * 0.
Lemma mult_n_Sm : forall n m:nat, n * m + n = n * (S m).
```

Finally, it gives the definition of the usual orderings `le`, `lt`, `ge` and `gt`.

```
Inductive le (n:nat) : nat -> Prop :=
| le_n : le n n
| le_S : forall m:nat, n <= m -> n <= (S m)
where "n <= m" := (le n m) : nat_scope.
Definition lt (n m:nat) := S n <= m.
Definition ge (n m:nat) := m <= n.
Definition gt (n m:nat) := m < n.
```

Properties of these relations are not initially known, but may be required by the user from modules `Le` and `Lt`. Finally, `Peano` gives some lemmas allowing pattern matching, and a double induction principle.

```
Theorem nat_case :
 forall (n:nat) (P:nat -> Prop),
 P 0 -> (forall m:nat, P (S m)) -> P n.
Theorem nat_double_ind :
 forall R:nat -> nat -> Prop,
  (forall n:nat, R 0 n) ->
  (forall n:nat, R (S n) 0) ->
  (forall n m:nat, R n m -> R (S n) (S m)) -> forall n m:nat, R n m.
```

Well-founded recursion

The basic library contains the basics of well-founded recursion and well-founded induction, in module `Wf.v`.

```
Section Well_founded.
Variable A : Type.
Variable R : A -> A -> Prop.
Inductive Acc (x:A) : Prop :=
  Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
Lemma Acc_inv x : Acc x -> forall y:A, R y x -> Acc y.
Definition well_founded := forall a:A, Acc a.
Hypothesis Rwf : well_founded.
Theorem well_founded_induction :
 forall P:A -> Set,
```

(continues on next page)

(continued from previous page)

```
  (forall x:A, (forall y:A, R y x -> P y) -> P x) -> forall a:A, P a.
Theorem well_founded_ind :
 forall P:A -> Prop,
  (forall x:A, (forall y:A, R y x -> P y) -> P x) -> forall a:A, P a.
```

The automatically generated scheme `Acc_rect` can be used to define functions by fixpoints using well-founded relations to justify termination. Assuming extensionality of the functional used for the recursive call, the fixpoint equation can be proved.

```
Section FixPoint.
Variable P : A -> Type.
Variable F : forall x:A, (forall y:A, R y x -> P y) -> P x.
Fixpoint Fix_F (x:A) (r:Acc x) {struct r} : P x :=
  F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)).
Definition Fix (x:A) := Fix_F x (Rwf x).
Hypothesis F_ext :
  forall (x:A) (f g:forall y:A, R y x -> P y),
    (forall (y:A) (p:R y x), f y p = g y p) -> F x f = F x g.
Lemma Fix_F_eq :
 forall (x:A) (r:Acc x),
   F x (fun (y:A) (p:R y x) => Fix_F y (Acc_inv x r y p)) = Fix_F x r.
Lemma Fix_F_inv : forall (x:A) (r s:Acc x), Fix_F x r = Fix_F x s.
Lemma Fix_eq : forall x:A, Fix x = F x (fun (y:A) (p:R y x) => Fix y).
End FixPoint.
End Well_founded.
```

Accessing the Type level

The standard library includes `Type` level definitions of counterparts of some logic concepts and basic lemmas about them.

The module `Datatypes` defines `identity`, which is the `Type` level counterpart of equality:

```
Inductive identity (A:Type) (a:A) : A -> Type :=
  identity_refl : identity A a a.
```

Some properties of `identity` are proved in the module `Logic_Type`, which also provides the definition of `Type` level negation:

```
Definition notT (A:Type) := A -> False.
```

Tactics

A few tactics defined at the user level are provided in the initial state, in module `Tactics.v`. They are listed at http://coq.inria.fr/stdlib, in paragraph `Init`, link `Tactics`.

The standard library

Survey

The rest of the standard library is structured into the following subdirectories:

- **Logic** : Classical logic and dependent equality
- **Arith** : Basic Peano arithmetic
- **PArith** : Basic positive integer arithmetic
- **NArith** : Basic binary natural number arithmetic
- **ZArith** : Basic relative integer arithmetic
- **Numbers** : Various approaches to natural, integer and cyclic numbers (currently axiomatically and on top of 2^31 binary words)
- **Bool** : Booleans (basic functions and results)
- **Lists** : Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)
- **Sets** : Sets (classical, constructive, finite, infinite, power set, etc.)
- **FSets** : Specification and implementations of finite sets and finite maps (by lists and by AVL trees)
- **Reals** : Axiomatization of real numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,...)
- **Floats** : Machine implementation of floating-point arithmetic (for the binary64 format)
- **Relations** : Relations (definitions and basic results)
- **Sorting** : Sorted list (basic definitions and heapsort correctness)
- **Strings** : 8-bits characters and strings
- **Wellfounded** : Well-founded relations (basic results)

These directories belong to the initial load path of the system, and the modules they provide are compiled at installation time. So they are directly accessible with the command `Require` (see Section *Compiled files*).

The different modules of the Coq standard library are documented online at https://coq.inria.fr/stdlib.

Peano's arithmetic (nat)

While in the initial state, many operations and predicates of Peano's arithmetic are defined, further operations and results belong to other modules. For instance, the decidability of the basic predicates are defined here. This is provided by requiring the module `Arith`.

The following table describes the notations available in scope `nat_scope` :

Notation	Interpretation
_ < _	lt
_ <= _	le
_ > _	gt
_ >= _	ge
x < y < z	x < y /\ y < z
x < y <= z	x < y /\ y <= z
x <= y < z	x <= y /\ y < z
x <= y <= z	x <= y /\ y <= z
_ + _	plus
_ - _	minus
_ * _	mult

Notations for integer arithmetic

The following table describes the syntax of expressions for integer arithmetic. It is provided by requiring and opening the module ZArith and opening scope Z_scope. It specifies how notations are interpreted and, when not already reserved, the precedence and associativity.

Notation	Interpretation	Precedence	Associativity
_ < _	Z.lt		
_ <= _	Z.le		
_ > _	Z.gt		
_ >= _	Z.ge		
x < y < z	x < y /\ y < z		
x < y <= z	x < y /\ y <= z		
x <= y < z	x <= y /\ y < z		
x <= y <= z	x <= y /\ y <= z		
_ ?= _	Z.compare	70	no
_ + _	Z.add		
_ - _	Z.sub		
_ * _	Z.mul		
_ / _	Z.div		
_ mod _	Z.modulo	40	no
- _	Z.opp		
_ ^ _	Z.pow		

Example

```
Require Import ZArith.
    [Loading ML file ring_plugin.cmxs ... done]
    [Loading ML file zify_plugin.cmxs ... done]
    [Loading ML file micromega_plugin.cmxs ... done]

Check (2 + 3)%Z.
    (2 + 3)%Z
         : Z

Open Scope Z_scope.
Check 2 + 3.
```

(continues on next page)

(continued from previous page)

```
2 + 3
     : Z
```

Real numbers library

Notations for real numbers

This is provided by requiring and opening the module `Reals` and opening scope `R_scope`. This set of notations is very similar to the notation for integer arithmetic. The inverse function was added.

Notation	Interpretation
`_ < _`	`Rlt`
`_ <= _`	`Rle`
`_ > _`	`Rgt`
`_ >= _`	`Rge`
`x < y < z`	`x < y /\ y < z`
`x < y <= z`	`x < y /\ y <= z`
`x <= y < z`	`x <= y /\ y < z`
`x <= y <= z`	`x <= y /\ y <= z`
`_ + _`	`Rplus`
`_ - _`	`Rminus`
`_ * _`	`Rmult`
`_ / _`	`Rdiv`
`- _`	`Ropp`
`/ _`	`Rinv`
`_ ^ _`	`pow`

Example

```
Require Import Reals.
Check  (2 + 3)%R.
    (2 + 3)%R
     : R

Open Scope R_scope.
Check 2 + 3.
    2 + 3
     : R
```

Some tactics for real numbers

In addition to the powerful `ring`, `field` and `lra` tactics (see Chapter *Tactics*), there are also:

Tactic: `discrR`
Proves that two real integer constants are different.

Example

```
Require Import DiscrR.
Open Scope R_scope.
Goal 5 <> 0.
    1 goal

      ============================
      5 <> 0

discrR.
```

Tactic: `split_Rabs`
Allows unfolding the `Rabs` constant and splits corresponding conjunctions.

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x:R, x <= Rabs x.
    1 goal

      ============================
      forall x : R, x <= Rabs x

intro; split_Rabs.
    2 goals

      x : R
      Hlt : x < 0
      ============================
      x <= - x

    goal 2 is:
     x <= x
```

Tactic: `split_Rmult`
Splits a condition that a product is non null into subgoals corresponding to the condition on each operand of the product.

Example

```
Require Import Reals.
Open Scope R_scope.
Goal forall x y z:R, x * y * z <> 0.
    1 goal
```

(continues on next page)

(continued from previous page)

```
      ============================
      forall x y z : R, x * y * z <> 0

intros; split_Rmult.
    3 goals

      x, y, z : R
      ============================
      x <> 0

    goal 2 is:
     y <> 0
    goal 3 is:
     z <> 0
```

These tactics has been written with the tactic language L_{tac} described in Chapter *Ltac*.

List library

Some elementary operations on polymorphic lists are defined here. They can be accessed by requiring module `List`.

It defines the following notions:

- `length`
- `head` : first element (with default)
- `tail` : all but first element
- `app` : concatenation
- `rev` : reverse
- `nth` : accessing n-th element (with default)
- `map` : applying a function
- `flat_map` : applying a function returning lists
- `fold_left` : iterator (from head to tail)
- `fold_right` : iterator (from tail to head)

The following table shows notations available when opening scope `list_scope`.

Notation	Interpretation	Precedence	Associativity
`_ ++ _`	`app`	60	right
`_ :: _`	`cons`	60	right

Floats library

The library of primitive floating-point arithmetic can be loaded by requiring module `Floats`:

```
Require Import Floats.
```

It exports the module `PrimFloat` that provides a primitive type named `float`, defined in the kernel (see section *Primitive Floats*), as well as two variant types `float_comparison` and `float_class`:

```
Print float.
    *** [ float : Set ]

Print float_comparison.
    Variant float_comparison : Set :=
        FEq : float_comparison
      | FLt : float_comparison
      | FGt : float_comparison
      | FNotComparable : float_comparison.

Print float_class.
    Variant float_class : Set :=
        PNormal : float_class
      | NNormal : float_class
      | PSubn : float_class
      | NSubn : float_class
      | PZero : float_class
      | NZero : float_class
      | PInf : float_class
      | NInf : float_class
      | NaN : float_class.
```

It then defines the primitive operators below, using the processor floating-point operators for binary64 in rounding-to-nearest even:

- `abs`
- `opp`
- `sub`
- `add`
- `mul`
- `div`
- `sqrt`
- `compare` : compare two floats and return a `float_comparison`
- `classify` : analyze a float and return a `float_class`
- `of_int63` : round a primitive integer and convert it into a float
- `normfr_mantissa` : take a float in `[0.5; 1.0)` and return its mantissa
- `frshiftexp` : convert a float to fractional part in `[0.5; 1.0)` and integer part
- `ldshiftexp` : multiply a float by an integral power of `2`
- `next_up` : return the next float towards positive infinity
- `next_down` : return the next float towards negative infinity

For special floating-point values, the following constants are also defined:

- `zero`
- `neg_zero`
- `one`
- `two`
- `infinity`
- `neg_infinity`
- `nan` : Not a Number (assumed to be unique: the ”payload” of NaNs is ignored)

The following table shows the notations available when opening scope `float_scope`.

Notation	Interpretation
`- _`	`opp`
`_ - _`	`sub`
`_ + _`	`add`
`_ * _`	`mul`
`_ / _`	`div`
`_ =? _`	`eqb`
`_ <? _`	`ltb`
`_ <=? _`	`leb`
`_ ?= _`	`compare`

Floating-point constants are parsed and pretty-printed as (17-digit) decimal constants. This ensures that the composition parse ∘ print amounts to the identity.

Warning: `The constant` *`number`* `is not a binary64 floating-point value. A closest value` *`number`* `will be used and unambiguously printed` *`number`*`. [inexact-float,parsing]`

Not all decimal constants are floating-point values. This warning is generated when parsing such a constant (for instance `0.1`).

Flag: `Printing Float`

Turn this *flag* off (it is on by default) to deactivate decimal printing of floating-point constants. They will then be printed with an hexadecimal representation.

Example

```
Open Scope float_scope.
Eval compute in 1 + 0.5.
    = 1.5
         : float

Eval compute in 1 / 0.
    = infinity
         : float

Eval compute in 1 / -0.
    = neg_infinity
         : float

Eval compute in 0 / 0.
```

(continues on next page)

(continued from previous page)

```
    = nan
         : float

Eval compute in 0 ?= -0.
    = FEq
         : float_comparison

Eval compute in nan ?= nan.
    = FNotComparable
         : float_comparison

Eval compute in next_down (-1).
    = -1.0000000000000002
         : float
```

The primitive operators are specified with respect to their Gallina counterpart, using the variant type `spec_float`, and the injection `Prim2SF`:

```
Print spec_float.
    Variant spec_float : Set :=
        S754_zero : bool -> spec_float
      | S754_infinity : bool -> spec_float
      | S754_nan : spec_float
      | S754_finite : bool -> positive -> Z -> spec_float.

    Arguments S754_zero _%bool_scope
    Arguments S754_infinity _%bool_scope
    Arguments S754_finite _%bool_scope _%positive_scope _%Z_scope

Check Prim2SF.
    Prim2SF
         : float -> spec_float

Check mul_spec.
    mul_spec
         : forall x y : float, Prim2SF (x * y) = SF64mul (Prim2SF x) (Prim2SF y)
```

For more details on the available definitions and lemmas, see the online documentation of the `Floats` library.

Users' contributions

Numerous users' contributions have been collected and are available at URL http://coq.inria.fr/opam/www/. On this web page, you have a list of all contributions with informations (author, institution, quick description, etc.) and the possibility to download them one by one. You will also find informations on how to submit a new contribution.

4.1.2 Program extraction

Authors Jean-Christophe Filliâtre and Pierre Letouzey

We present here the Coq extraction commands, used to build certified and relatively efficient functional programs, extracting them from either Coq functions or Coq proofs of specifications. The functional languages available as output are currently OCaml, Haskell and Scheme. In the following, "ML" will be used (abusively) to refer to any of the three.

Before using any of the commands or options described in this chapter, the extraction framework should first be loaded explicitly via `Require Extraction`, or via the more robust `From Coq Require Extraction`. Note that in earlier versions of Coq, these commands and options were directly available without any preliminary `Require`.

```
Require Extraction.
```

Generating ML Code

Note: In the following, a qualified identifier *qualid* can be used to refer to any kind of Coq global "object" : *constant*, inductive type, inductive constructor or module name.

The next two commands are meant to be used for rapid preview of extraction. They both display extracted term(s) inside Coq.

Command: `Extraction` *qualid*
Extraction of the mentioned object in the Coq toplevel.

Command: `Recursive Extraction` *qualid*+
Recursive extraction of all the mentioned objects and all their dependencies in the Coq toplevel.

All the following commands produce real ML files. User can choose to produce one monolithic file or one file per Coq library.

Command: `Extraction` *string* *qualid*+
Recursive extraction of all the mentioned objects and all their dependencies in one monolithic file *string*. Global and local identifiers are renamed according to the chosen ML language to fulfill its syntactic conventions, keeping original names as much as possible.

Command: `Extraction Library` *ident*
Extraction of the whole Coq library ***ident*`.v`** to an ML module ***ident*`.ml`**. In case of name clash, identifiers are here renamed using prefixes `coq_` or `Coq_` to ensure a session-independent renaming.

Command: `Recursive Extraction Library` *ident*
Extraction of the Coq library ***ident*`.v`** and all other modules ***ident*`.v`** depends on.

Command: `Separate Extraction` *qualid*+
Recursive extraction of all the mentioned objects and all their dependencies, just as **`Extraction` *string* *qualid*+**, but instead of producing one monolithic file, this command splits the produced code in separate ML files, one per corresponding Coq `.v` file. This command is hence quite similar to *Recursive Extraction Library*, except that only the needed parts of Coq libraries are extracted instead of the whole. The naming convention in case of name clash is the same one as *Extraction Library*: identifiers are here renamed using prefixes `coq_` or `Coq_`.

The following command is meant to help automatic testing of the extraction, see for instance the `test-suite` directory in the Coq sources.

Command: **`Extraction TestCompile`** *`qualid`* +

All the mentioned objects and all their dependencies are extracted to a temporary OCaml file, just as in `Extraction "file"`. Then this temporary file and its signature are compiled with the same OCaml compiler used to built Coq. This command succeeds only if the extraction and the OCaml compilation succeed. It fails if the current target language of the extraction is not OCaml.

Extraction Options

Setting the target language

Command: **`Extraction Language`** *`language`*

language ::= **OCaml**
| **Haskell**
| **Scheme**
| **JSON**

The ability to fix target language is the first and most important of the extraction options. Default is `OCaml`.

The JSON output is mostly for development or debugging: it contains the raw ML term produced as an intermediary target.

Inlining and optimizations

Since OCaml is a strict language, the extracted code has to be optimized in order to be efficient (for instance, when using induction principles we do not want to compute all the recursive calls but only the needed ones). So the extraction mechanism provides an automatic optimization routine that will be called each time the user wants to generate an OCaml program. The optimizations can be split in two groups: the type-preserving ones (essentially constant inlining and reductions) and the non type-preserving ones (some function abstractions of dummy types are removed when it is deemed safe in order to have more elegant types). Therefore some *constants* may not appear in the resulting monolithic OCaml program. In the case of modular extraction, even if some inlining is done, the inlined constants are nevertheless printed, to ensure session-independent programs.

Concerning Haskell, type-preserving optimizations are less useful because of laziness. We still make some optimizations, for example in order to produce more readable code.

The type-preserving optimizations are controlled by the following Coq flags and commands:

Flag: **`Extraction Optimize`**

Default is on. This *flag* controls all type-preserving optimizations made on the ML terms (mostly reduction of dummy beta/iota redexes, but also simplifications on Cases, etc). Turn this flag off if you want a ML term as close as possible to the Coq term.

Flag: **`Extraction Conservative Types`**

Default is off. This *flag* controls the non type-preserving optimizations made on ML terms (which try to avoid function abstraction of dummy types). Turn this flag on to make sure that `e:t` implies that `e':t'` where `e'` and `t'` are the extracted code of `e` and `t` respectively.

Flag: **`Extraction KeepSingleton`**

Default is off. Normally, when the extraction of an inductive type produces a singleton type (i.e. a type with only one constructor, and only one argument to this constructor), the inductive structure is removed and this type is seen as an alias to the inner type. The typical example is `sig`. This *flag* allows disabling this optimization when one wishes to preserve the inductive structure of types.

Flag: `Extraction AutoInline`
Default is on. The extraction mechanism inlines the *bodies* of some defined *constants*, according to some heuristics like size of bodies, uselessness of some arguments, etc. Those heuristics are not always perfect; if you want to disable this feature, turn this *flag* off.

Command: `Extraction Inline` *qualid*+
In addition to the automatic inline feature, the *constants* mentioned by this command will always be inlined during extraction.

Command: `Extraction NoInline` *qualid*+
Conversely, the constants mentioned by this command will never be inlined during extraction.

Command: `Print Extraction Inline`
Prints the current state of the table recording the custom inlinings declared by the two previous commands.

Command: `Reset Extraction Inline`
Empties the table recording the custom inlinings (see the previous commands).

Inlining and printing of a constant declaration:

The user can explicitly ask for a *constant* to be extracted by two means:

- by mentioning it on the extraction command line
- by extracting the whole Coq module of this *constant*.

In both cases, the declaration of this *constant* will be present in the produced file. But this same *constant* may or may not be inlined in the following terms, depending on the automatic/custom inlining mechanism.

For the *constants* non-explicitly required but needed for dependency reasons, there are two cases:

- If an inlining decision is taken, whether automatically or not, all occurrences of this *constant* are replaced by its extracted *body*, and this *constant* is not declared in the generated file.
- If no inlining decision is taken, the *constant* is normally declared in the produced file.

Extra elimination of useless arguments

The following command provides some extra manual control on the code elimination performed during extraction, in a way which is independent but complementary to the main elimination principles of extraction (logical parts and types).

Command: `Extraction Implicit` *qualid* `[` (*ident* | *integer*)* `]`
Declares some arguments of *qualid* as implicit, meaning that they are useless in extracted code. The extracted code will omit these arguments. Here *qualid* can be any function or inductive constructor, and the *ident*s are the names of the useless arguments. Arguments can can also be identified positionally by *integer*s starting from 1.

When an actual extraction takes place, an error is normally raised if the *Extraction Implicit* declarations cannot be honored, that is if any of the implicit arguments still occurs in the final code. This behavior can be relaxed via the following flag:

Flag: `Extraction SafeImplicits`
Default is on. When this *flag* is off, a warning is emitted instead of an error if some implicit arguments still occur in the final code of an extraction. This way, the extracted code may be obtained nonetheless and reviewed manually to locate the source of the issue (in the code, some comments mark the location of these remaining implicit arguments). Note that this extracted code might not compile or run properly, depending of the use of these remaining implicit arguments.

Realizing axioms

Extraction will fail if it encounters an informative axiom not realized. A warning will be issued if it encounters a logical axiom, to remind the user that inconsistent logical axioms may lead to incorrect or non-terminating extracted terms.

It is possible to assume some axioms while developing a proof. Since these axioms can be any kind of proposition or object or type, they may perfectly well have some computational content. But a program must be a closed term, and of course the system cannot guess the program which realizes an axiom. Therefore, it is possible to tell the system what ML term corresponds to a given axiom.

Command: **Extract Constant** *qualid* *string*$_{tv}$ * **=>** *ident* | *string*

Give an ML extraction for the given *constant*.

string$_{tv}$ If the type scheme axiom is an arity (a sequence of products followed by a sort), then some type variables have to be given (as quoted strings).

The number of type variables is checked by the system. For example:

```
Axiom Y : Set -> Set -> Set.
Extract Constant Y "'a" "'b" => " 'a * 'b ".
```

Command: **Extract Inlined Constant** *qualid* **=>** *ident* | *string*

Same as the previous one, except that the given ML terms will be inlined everywhere instead of being declared via a `let`.

Note: This command is sugar for an *Extract Constant* followed by a *Extraction Inline*. Hence a *Reset Extraction Inline* will have an effect on the realized and inlined axiom.

Caution: It is the responsibility of the user to ensure that the ML terms given to realize the axioms do have the expected types. In fact, the strings containing realizing code are just copied to the extracted files. The extraction recognizes whether the realized axiom should become a ML type constant or a ML object declaration. For example:

```
Axiom X:Set.
Axiom x:X.
Extract Constant X => "int".
Extract Constant x => "0".
```

Realizing an axiom via *Extract Constant* is only useful in the case of an informative axiom (of sort `Type` or `Set`). A logical axiom has no computational content and hence will not appear in extracted terms. But a warning is nonetheless issued if extraction encounters a logical axiom. This warning reminds user that inconsistent logical axioms may lead to incorrect or non-terminating extracted terms.

If an informative axiom has not been realized before an extraction, a warning is also issued and the definition of the axiom is filled with an exception labeled `AXIOM TO BE REALIZED`. The user must then search these exceptions inside the extracted file and replace them by real code.

Realizing inductive types

The system also provides a mechanism to specify ML terms for inductive types and constructors. For instance, the user may want to use the ML native boolean type instead of the Coq one. The syntax is the following:

Command:

Extract Inductive *qualid* **=>** *ident* | *string* **[** *ident* | *string* **]*** *string*$_{match}$**?**

Give an ML extraction for the given inductive type. You must specify extractions for the type itself (the initial *ident* | *string*) and all its constructors (the **[** *ident* | *string* **]***). In this form, the ML extraction must be an ML inductive datatype, and the native pattern matching of the language will be used.

When the initial *ident* | *string* matches the name of the type of characters or strings (`char` and `string` for OCaml, `Prelude.Char` and `Prelude.String` for Haskell), extraction of literals is handled in a specialized way, so as to generate literals in the target language. This feature requires the type designated by *qualid* to be registered as the standard char or string type, using the *Register* command.

string$_{match}$ Indicates how to perform pattern matching over this inductive type. In this form, the ML extraction could be an arbitrary type. For an inductive type with k constructors, the function used to emulate the pattern matching should expect $k+1$ arguments, first the k branches in functional form, and then the inductive element to destruct. For instance, the match branch `| S n => foo` gives the functional form `(fun n -> foo)`. Note that a constructor with no arguments is considered to have one unit argument, in order to block early evaluation of the branch: `| O => bar` leads to the functional form `(fun () -> bar)`. For instance, when extracting `nat` into OCaml `int`, the code to be provided has type: `(unit->'a)->(int->'a)->int->'a`.

Caution: As for *Extract Constant*, this command should be used with care:

- The ML code provided by the user is currently **not** checked at all by extraction, even for syntax errors.
- Extracting an inductive type to a pre-existing ML inductive type is quite sound. But extracting to a general type (by providing an ad-hoc pattern matching) will often **not** be fully rigorously correct. For instance, when extracting `nat` to OCaml `int`, it is theoretically possible to build `nat` values that are larger than OCaml `max_int`. It is the user's responsibility to be sure that no overflow or other bad events occur in practice.
- Translating an inductive type to an arbitrary ML type does **not** magically improve the asymptotic complexity of functions, even if the ML type is an efficient representation. For instance, when extracting `nat` to OCaml `int`, the function `Nat.mul` stays quadratic. It might be interesting to associate this translation with some specific *Extract Constant* when primitive counterparts exist.

Typical examples are the following:

```
Extract Inductive unit => "unit" [ "()" ].
Extract Inductive bool => "bool" [ "true" "false" ].
Extract Inductive sumbool => "bool" [ "true" "false" ].
```

Note: When extracting to OCaml, if an inductive constructor or type has arity 2 and the corresponding string is enclosed by parentheses, and the string meets OCaml's lexical criteria for an infix symbol, then the rest of the string is used as an infix constructor or type.

```
Extract Inductive list => "list" [ "[]" "(::)" ].
Extract Inductive prod => "(*)"  [ "(,)" ].
```

As an example of translation to a non-inductive datatype, let's turn `nat` into OCaml `int` (see caveat above):

```
Extract Inductive nat => int [ "0" "succ" ] "(fun fO fS n -> if n=0 then fO () else␣
↪fS (n-1))".
```

Avoiding conflicts with existing filenames

When using *Extraction Library*, the names of the extracted files directly depend on the names of the Coq files. It may happen that these filenames are in conflict with already existing files, either in the standard library of the target language or in other code that is meant to be linked with the extracted code. For instance the module `List` exists both in Coq and in OCaml. It is possible to instruct the extraction not to use particular filenames.

Command: Extraction Blacklist *ident* +
Instruct the extraction to avoid using these names as filenames for extracted code.

Command: Print Extraction Blacklist
Show the current list of filenames the extraction should avoid.

Command: Reset Extraction Blacklist
Allow the extraction to use any filename.

For OCaml, a typical use of these commands is `Extraction Blacklist String List`.

Additional settings

Option: Extraction File Comment *string*
This *option* provides a comment that is included at the beginning of the output files.

Option: Extraction Flag *natural*
This *option* controls which optimizations are used during extraction, providing a finer-grained control than *Extraction Optimize*. The bits of *natural* are used as a bit mask. Keeping an option off keeps the extracted ML more similar to the Coq term. Values are:

Bit	Value	Optimization (default is on unless noted otherwise)
0	1	Remove local dummy variables
1	2	Use special treatment for fixpoints
2	4	Simplify case with iota-redux
3	8	Factor case branches as functions
4	16	(not available, default false)
5	32	Simplify case as function of one argument
6	64	Simplify case by swapping case and lambda
7	128	Some case optimization
8	256	Push arguments inside a letin
9	512	Use linear let reduction (default false)
10	1024	Use linear beta reduction (default false)

Flag: Extraction TypeExpand
If this *flag* is set, fully expand Coq types in ML. See the Coq source code to learn more.

Differences between Coq and ML type systems

Due to differences between Coq and ML type systems, some extracted programs are not directly typable in ML. We now solve this problem (at least in OCaml) by adding when needed some unsafe casting `Obj.magic`, which give a generic type `'a` to any term.

First, if some part of the program is *very* polymorphic, there may be no ML type for it. In that case the extraction to ML works alright but the generated code may be refused by the ML type checker. A very well known example is the `distr_pair` function:

```
Definition dp {A B:Type}(x:A)(y:B)(f:forall C:Type, C->C) := (f A x, f B y).
```

In OCaml, for instance, the direct extracted term would be:

```
let dp x y f = Pair((f () x),(f () y))
```

and would have type:

```
dp : 'a -> 'a -> (unit -> 'a -> 'b) -> ('b,'b) prod
```

which is not its original type, but a restriction.

We now produce the following correct version:

```
let dp x y f = Pair ((Obj.magic f () x), (Obj.magic f () y))
```

Secondly, some Coq definitions may have no counterpart in ML. This happens when there is a quantification over types inside the type of a constructor; for example:

```
Inductive anything : Type := dummy : forall A:Set, A -> anything.
```

which corresponds to the definition of an ML dynamic type. In OCaml, we must cast any argument of the constructor dummy (no GADT are produced yet by the extraction).

Even with those unsafe castings, you should never get error like `segmentation fault`. In fact even if your program may seem ill-typed to the OCaml type checker, it can't go wrong : it comes from a Coq well-typed terms, so for example inductive types will always have the correct number of arguments, etc. Of course, when launching manually some extracted function, you should apply it to arguments of the right shape (from the Coq point-of-view).

More details about the correctness of the extracted programs can be found in [Let02].

We have to say, though, that in most "realistic" programs, these problems do not occur. For example all the programs of Coq library are accepted by the OCaml type checker without any `Obj.magic` (see examples below).

Some examples

We present here two examples of extraction, taken from the Coq Standard Library. We choose OCaml as the target language, but everything, with slight modifications, can also be done in the other languages supported by extraction. We then indicate where to find other examples and tests of extraction.

A detailed example: Euclidean division

The file `Euclid` contains the proof of Euclidean division. The natural numbers used here are unary, represented by the type `nat`, which is defined by two constructors `O` and `S`. This module contains a theorem `eucl_dev`, whose type is:

```
forall b:nat, b > 0 -> forall a:nat, diveucl a b
```

where `diveucl` is a type for the pair of the quotient and the modulo, plus some logical assertions that disappear during extraction. We can now extract this program to OCaml:

```
Require Extraction.
Require Import Euclid Wf_nat.
Extraction Inline gt_wf_rec lt_wf_rec induction_ltof2.
Recursive Extraction eucl_dev.
    type nat =
    | O
    | S of nat

    type sumbool =
    | Left
    | Right

    (** val sub : nat -> nat -> nat **)

    let rec sub n m =
      match n with
      | O -> n
      | S k -> (match m with
                | O -> n
                | S l -> sub k l)

    (** val le_lt_dec : nat -> nat -> sumbool **)

    let rec le_lt_dec n m =
      match n with
      | O -> Left
      | S n0 -> (match m with
                 | O -> Right
                 | S n1 -> le_lt_dec n0 n1)

    (** val le_gt_dec : nat -> nat -> sumbool **)

    let le_gt_dec =
      le_lt_dec

    type diveucl =
    | Divex of nat * nat

    (** val eucl_dev : nat -> nat -> diveucl **)

    let rec eucl_dev n m =
      let s = le_gt_dec n m in
      (match s with
       | Left ->
         let d = let y = sub m n in eucl_dev n y in
         let Divex (q, r) = d in Divex ((S q), r)
       | Right -> Divex (O, m))
```

The inlining of `gt_wf_rec` and others is not mandatory. It only enhances readability of extracted code. You can then copy-paste the output to a file `euclid.ml` or let Coq do it for you with the following command:

```
Extraction "euclid" eucl_dev.
```

Let us play the resulting program (in an OCaml toplevel):

```
#use "euclid.ml";;
type nat = O | S of nat
type sumbool = Left | Right
val sub : nat -> nat -> nat = <fun>
val le_lt_dec : nat -> nat -> sumbool = <fun>
val le_gt_dec : nat -> nat -> sumbool = <fun>
type diveucl = Divex of nat * nat
val eucl_dev : nat -> nat -> diveucl = <fun>

# eucl_dev (S (S O)) (S (S (S (S (S O)))));;
- : diveucl = Divex (S (S O), S O)
```

It is easier to test on OCaml integers:

```
# let rec nat_of_int = function 0 -> O | n -> S (nat_of_int (n-1));;
val nat_of_int : int -> nat = <fun>

# let rec int_of_nat = function O -> 0 | S p -> 1+(int_of_nat p);;
val int_of_nat : nat -> int = <fun>

# let div a b =
  let Divex (q,r) = eucl_dev (nat_of_int b) (nat_of_int a)
  in (int_of_nat q, int_of_nat r);;
val div : int -> int -> int * int = <fun>

# div 173 15;;
- : int * int = (11, 8)
```

Note that these `nat_of_int` and `int_of_nat` are now available via a mere `Require Import ExtrOcamlIntConv` and then adding these functions to the list of functions to extract. This file `ExtrOcamlIntConv.v` and some others in `plugins/extraction/` are meant to help building concrete program via extraction.

Extraction's horror museum

Some pathological examples of extraction are grouped in the file `test-suite/success/extraction.v` of the sources of Coq.

Users' Contributions

Several of the Coq Users' Contributions use extraction to produce certified programs. In particular the following ones have an automatic extraction test:

- `additions` : https://github.com/coq-contribs/additions
- `bdds` : https://github.com/coq-contribs/bdds
- `canon-bdds` : https://github.com/coq-contribs/canon-bdds
- `chinese` : https://github.com/coq-contribs/chinese

- `continuations` : https://github.com/coq-contribs/continuations
- `coq-in-coq` : https://github.com/coq-contribs/coq-in-coq
- `exceptions` : https://github.com/coq-contribs/exceptions
- `firing-squad` : https://github.com/coq-contribs/firing-squad
- `founify` : https://github.com/coq-contribs/founify
- `graphs` : https://github.com/coq-contribs/graphs
- `higman-cf` : https://github.com/coq-contribs/higman-cf
- `higman-nw` : https://github.com/coq-contribs/higman-nw
- `hardware` : https://github.com/coq-contribs/hardware
- `multiplier` : https://github.com/coq-contribs/multiplier
- `search-trees` : https://github.com/coq-contribs/search-trees
- `stalmarck` : https://github.com/coq-contribs/stalmarck

Note that `continuations` and `multiplier` are a bit particular. They are examples of developments where `Obj.magic` is needed. This is probably due to a heavy use of impredicativity. After compilation, those two examples run nonetheless, thanks to the correction of the extraction [Let02].

4.1.3 Program derivation

Coq comes with an extension called `Derive`, which supports program derivation. Typically in the style of Bird and Meertens or derivations of program refinements. To use the Derive extension it must first be required with `Require Coq.derive.Derive`. When the extension is loaded, it provides the following command:

Command: `Derive` *ident$_1$* `SuchThat` *one_term* `As` *ident$_2$*

ident$_1$ can appear in *one_term*. This command opens a new proof presenting the user with a goal for *one_term* in which the name *ident$_1$* is bound to an existential variable `?x` (formally, there are other goals standing for the existential variables but they are shelved, as described in *shelve*).

When the proof ends two *constants* are defined:

- The first one is named *ident$_1$* and is defined as the proof of the shelved goal (which is also the value of `?x`). It is always transparent.
- The second one is named *ident$_2$*. It has type *type*, and its *body* is the proof of the initially visible goal. It is opaque if the proof ends with *Qed*, and transparent if the proof ends with *Defined*.

Example

```
Require Coq.derive.Derive.
    [Loading ML file derive_plugin.cmxs ... done]

Require Import Coq.Numbers.Natural.Peano.NPeano.

Section P.

Variables (n m k:nat).
    n is declared
    m is declared
    k is declared
```

(continues on next page)

(continued from previous page)

```
Derive p SuchThat ((k*n)+(k*m) = p) As h.
    1 focused goal
    (shelved: 1)

      n, m, k : nat
      p := ?Goal : nat
      ============================
      k * n + k * m = p

Proof.
rewrite <- Nat.mul_add_distr_l.
    1 focused goal
    (shelved: 1)

      n, m, k : nat
      p := ?Goal : nat
      ============================
      k * (n + m) = p

subst p.
    1 focused goal
    (shelved: 1)

      n, m, k : nat
      ============================
      k * (n + m) = ?Goal

reflexivity.
    No more goals.

Qed.

End P.

Print p.
    p = fun n m k : nat => k * (n + m)
          : nat -> nat -> nat -> nat

    Arguments p (_ _ _)%nat_scope

Check h.
    h
          : forall n m k : nat, k * n + k * m = p n m k
```

Any property can be used as `term`, not only an equation. In particular, it could be an order relation specifying some form of program refinement or a non-executable property from which deriving a program is convenient.

4.1.4 Functional induction

Advanced recursive functions

The following command is available when the `FunInd` library has been loaded via `Require Import FunInd`:

Command: **`Function`** *`fix_definition`* **`with`** *`fix_definition`*`*`

This command is a generalization of *`Fixpoint`*. It is a wrapper for several ways of defining a function *and* other useful related objects, namely: an induction principle that reflects the recursive structure of the function (see *`functional induction`*) and its fixpoint equality. This defines a function similar to those defined by *`Fixpoint`*. As in *`Fixpoint`*, the decreasing argument must be given (unless the function is not recursive), but it might not necessarily be *structurally* decreasing. Use the ***`fixannot`*** clause to name the decreasing argument *and* to describe which kind of decreasing criteria to use to ensure termination of recursive calls.

`Function` also supports the **`with`** clause to create mutually recursive definitions, however this feature is limited to structurally recursive functions (i.e. when ***`fixannot`*** is a **`struct`** clause).

See *`functional induction`* and *`Functional Scheme`* for how to use the induction principle to reason easily about the function.

The form of the ***`fixannot`*** clause determines which definition mechanism *`Function`* uses. (Note that references to **`ident`** below refer to the name of the function being defined.):

- If ***`fixannot`*** is not specified, *`Function`* defines the nonrecursive function *`ident`* as if it was declared with *`Definition`*. In addition, the following are defined:
  - *`ident`*`_rect`, *`ident`*`_rec` and *`ident`*`_ind`, which reflect the pattern matching structure of *`term`* (see *`Inductive`*);
  - The inductive **`R_`***`ident`* corresponding to the graph of *`ident`* (silently);
  - *`ident`*`_complete` and *`ident`*`_correct` which are inversion information linking the function and its graph.
- If **`{ struct ... }`** is specified, *`Function`* defines the structural recursive function *`ident`* as if it was declared with *`Fixpoint`*. In addition, the following are defined:
  - The same objects as above;
  - The fixpoint equation of *`ident`*: ***`ident`***`_equation`.
- If **`{ measure ... }`** or **`{ wf ... }`** are specified, *`Function`* defines a recursive function by well-founded recursion. The module `Recdef` of the standard library must be loaded for this feature.
  - **`{measure`** *`one_term`*$_1$ *`ident`*`?` *`one_term`*$_2$`?` **`}`**: where *`ident`* is the decreasing argument and *`one_term`*$_1$ is a function from the type of *`ident`* to `nat` for which the decreasing argument decreases (for the `lt` order on `nat`) for each recursive call of the function. The parameters of the function are bound in *`one_term`*$_1$.
  - **`{wf`** *`one_term ident`* **`}`**: where *`ident`* is the decreasing argument and *`one_term`* is an ordering relation on the type of *`ident`* (i.e. of type $\mathtt{T}_{\mathtt{ident}} \to \mathtt{T}_{\mathtt{ident}} \to \mathtt{Prop}$) for which the decreasing argument decreases for each recursive call of the function. The order must be well-founded. The parameters of the function are bound in *`one_term`*.

  If the clause is `measure` or `wf`, the user is left with some proof obligations that will be used to define the function. These proofs are: proofs that each recursive call is actually decreasing with respect to the given criteria, and (if the criteria is `wf`) a proof that the ordering relation is well-founded. Once proof obligations are discharged, the following objects are defined:
  - The same objects as with the `struct` clause;

- The lemma *ident*_tcc which collects all proof obligations in one property;
- The lemmas *ident*_terminate and *ident*_F which will be inlined during extraction of *ident*.

The way this recursive function is defined is the subject of several papers by Yves Bertot and Antonia Balaa on the one hand, and Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu on the other hand.

Note: To obtain the right principle, it is better to put rigid parameters of the function as first arguments. For example it is better to define plus like this:

```
Function plus (m n : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus m p)
end.
    plus is defined
    plus is recursively defined (guarded on 2nd argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
    plus_rec is defined
    R_plus_correct is defined
    R_plus_complete is defined
```

than like this:

```
Function plus (n m : nat) {struct n} : nat :=
match n with
| 0 => m
| S p => S (plus p m)
end.
    plus is defined
    plus is recursively defined (guarded on 1st argument)
    plus_equation is defined
    plus_rect is defined
    plus_ind is defined
    plus_rec is defined
    R_plus_correct is defined
    R_plus_complete is defined
```

Limitations

term must be built as a *pure pattern matching tree* (`match ... with`) with applications only *at the end* of each branch.

Function does not support partial application of the function being defined. Thus, the following example cannot be accepted due to the presence of partial application of `wrong` in the body of `wrong`:

```
Function wrong (C:nat) : nat :=
  List.hd 0 (List.map wrong (C::nil)).
    Toplevel input, characters 0-70:
    > Function wrong (C:nat) : nat :=   List.hd 0 (List.map wrong (C::nil)).
    > ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Error:
    Recursive definition of wrong is ill-formed.
    In environment
    wrong : nat -> nat
    C : nat
```

(continues on next page)

(continued from previous page)

```
Recursive call to wrong has principal argument equal to
"C" instead of a subterm of "C".
Recursive definition is: "fun C : nat => List.hd 0 (List.map wrong [C])".
```

For now, dependent cases are not treated for non structurally terminating functions.

Error: The recursive argument must be specified.

Error: No argument name *ident*.

Error: Cannot use mutual definition with well-founded recursion or measure.

Warning: Cannot define graph for *ident*.
The generation of the graph relation (**R_*ident***) used to compute the induction scheme of ident raised a typing error. Only *ident* is defined; the induction scheme will not be generated. This error happens generally when:

- the definition uses pattern matching on dependent types, which *Function* cannot deal with yet.
- the definition is not a *pattern matching tree* as explained above.

Warning: Cannot define principle(s) for *ident*.
The generation of the graph relation (**R_*ident***) succeeded but the induction principle could not be built. Only *ident* is defined. Please report.

Warning: Cannot build functional inversion principle.
functional inversion will not be available for the function.

Tactics

Tactic:

functional induction *term* **using** *one_term*? **with** *bindings*? **as** *simple_intropattern*?
Performs case analysis and induction following the definition of a function *qualid*, which must be fully applied to its arguments as part of *term*. It uses a principle generated by *Function* or *Functional Scheme*. Note that this tactic is only available after a `Require Import FunInd`. See the *Function* command.

using *one_term* Specifies the induction principle (aka elimination scheme).

with *bindings* Specifies the arguments of the induction principle.

as *simple_intropattern* Provides names for the introduced variables.

Example

```
Require Import FunInd.
Functional Scheme minus_ind := Induction for minus Sort Prop.
    sub_equation is defined
    minus_ind is defined

Check minus_ind.
    minus_ind
         : forall P : nat -> nat -> nat -> Prop,
           (forall n m : nat, n = 0 -> P 0 m n) ->
           (forall n m k : nat, n = S k -> m = 0 -> P (S k) 0 n) ->
           (forall n m k : nat,
            n = S k ->
            forall l : nat, m = S l -> P k l (k - l) -> P (S k) (S l) (k - l)) ->
           forall n m : nat, P n m (n - m)
```

(continues on next page)

(continued from previous page)

```
Lemma le_minus (n m:nat) : n - m <= n.
    1 goal

      n, m : nat
      ============================
      n - m <= n

functional induction (minus n m) using minus_ind; simpl; auto.
    No more goals.

Qed.
```

Note: **functional induction (f x1 x2 x3)** is actually a wrapper for **induction x1, x2, x3, (f x1 x2 x3) using** *qualid* followed by a cleaning phase, where *qualid* is the induction principle registered for `f` (by the *Function* or *Functional Scheme* command) corresponding to the sort of the goal. Therefore *functional induction* may fail if the induction scheme *qualid* is not defined.

Note: There is a difference between obtaining an induction scheme for a function by using *Function* and by using *Functional Scheme* after a normal definition using *Fixpoint* or *Definition*.

Error: **Cannot find induction information on** *qualid*.

Error: **Not the right number of induction arguments.**

Tactic: **functional inversion** *ident* | *natural* *qualid*?

Performs inversion on hypothesis *ident* of the form *qualid* *term*+ **=** *term* or *term* **=** *qualid* *term*+ when *qualid* is defined using *Function*. Note that this tactic is only available after a `Require Import FunInd`.

natural Does the same thing as **intros until** *natural* followed by **functional inversion** *ident* where *ident* is the identifier for the last introduced hypothesis.

qualid If the hypothesis *ident* (or *natural*) has a type of the form $qualid_1$ $term_i$+ **=** $qualid_2$ $term_j$+ where $qualid_1$ and $qualid_2$ are valid candidates to functional inversion, this variant allows choosing which *qualid* is inverted.

Error: **Hypothesis** *ident* **must contain at least one Function.**

Error: **Cannot find inversion information for hypothesis** *ident*.
This error may be raised when some inversion lemma failed to be generated by Function.

Generation of induction principles with `Functional Scheme`

Command: **`Functional Scheme`** *`func_scheme_def`* **`with`** *`func_scheme_def`*`*`

func_scheme_def **::=** *ident* **:= Induction for** *qualid* **Sort** *sort_family*

An experimental high-level tool that automatically generates induction principles corresponding to functions that may be mutually recursive. The command generates an induction principle named *`ident`* for each given function named *`qualid`*. The *`qualid`*s must be given in the same order as when they were defined.

Note the command must be made available via *`Require Import`* `FunInd`.

Warning: There is a difference between induction schemes generated by the command *`Functional Scheme`* and these generated by the *`Function`*. Indeed, *`Function`* generally produces smaller principles that are closer to how a user would implement them. See *Advanced recursive functions* for details.

Example

Induction scheme for div2.

We define the function div2 as follows:

```
Require Import FunInd.
Require Import Arith.
    [Loading ML file ring_plugin.cmxs ... done]


Fixpoint div2 (n:nat) : nat :=
match n with
| O => 0
| S O => 0
| S (S n') => S (div2 n')
end.
    div2 is defined
    div2 is recursively defined (guarded on 1st argument)
```

The definition of a principle of induction corresponding to the recursive structure of `div2` is defined by the command:

```
Functional Scheme div2_ind := Induction for div2 Sort Prop.
    div2_equation is defined
    div2_ind is defined
```

You may now look at the type of div2_ind:

```
Check div2_ind.
    div2_ind
         : forall P : nat -> nat -> Prop,
           (forall n : nat, n = 0 -> P 0 0) ->
           (forall n n0 : nat, n = S n0 -> n0 = 0 -> P 1 0) ->
           (forall n n0 : nat,
            n = S n0 ->
            forall n' : nat,
            n0 = S n' -> P n' (div2 n') -> P (S (S n')) (S (div2 n'))) ->
           forall n : nat, P n (div2 n)
```

We can now prove the following lemma using this principle:

```
Lemma div2_le' : forall n:nat, div2 n <= n.
    1 goal

      ============================
      forall n : nat, div2 n <= n

intro n.
    1 goal

      n : nat
      ============================
      div2 n <= n

pattern n, (div2 n).
    1 goal

      n : nat
      ============================
      (fun n0 n1 : nat => n1 <= n0) n (div2 n)

apply div2_ind; intros.
    3 goals

      n, n0 : nat
      e : n0 = 0
      ============================
      0 <= 0

    goal 2 is:
     0 <= 1
    goal 3 is:
     S (div2 n') <= S (S n')

auto with arith.
    2 goals

      n, n0, n1 : nat
      e : n0 = S n1
      e0 : n1 = 0
      ============================
      0 <= 1

    goal 2 is:
     S (div2 n') <= S (S n')

auto with arith.
    1 goal

      n, n0, n1 : nat
      e : n0 = S n1
      n' : nat
      e0 : n1 = S n'
      H : div2 n' <= n'
      ============================
      S (div2 n') <= S (S n')

simpl; auto with arith.
```

(continues on next page)

(continued from previous page)

```
    No more goals.

Qed.
```

We can use directly the functional induction (*functional induction*) tactic instead of the pattern/apply trick:

```
Reset div2_le'.

Lemma div2_le : forall n:nat, div2 n <= n.
    1 goal

      ============================
      forall n : nat, div2 n <= n

intro n.
    1 goal

      n : nat
      ============================
      div2 n <= n

functional induction (div2 n).
    3 goals

      ============================
      0 <= 0

    goal 2 is:
     0 <= 1
    goal 3 is:
     S (div2 n') <= S (S n')

auto with arith.
    2 goals

      ============================
      0 <= 1

    goal 2 is:
     S (div2 n') <= S (S n')

auto with arith.
    1 goal

      n' : nat
      IHn0 : div2 n' <= n'
      ============================
      S (div2 n') <= S (S n')

auto with arith.
    No more goals.

Qed.
```

Example

Induction scheme for tree_size.

We define trees by the following mutual inductive type:

```
Axiom A : Set.
    A is declared


Inductive tree : Set :=
node : A -> forest -> tree
with forest : Set :=
| empty : forest
| cons : tree -> forest -> forest.
    tree, forest are defined
    tree_rect is defined
    tree_ind is defined
    tree_rec is defined
    tree_sind is defined
    forest_rect is defined
    forest_ind is defined
    forest_rec is defined
    forest_sind is defined
```

We define the function tree_size that computes the size of a tree or a forest. Note that we use `Function` which generally produces better principles.

```
Require Import FunInd.

Function tree_size (t:tree) : nat :=
match t with
| node A f => S (forest_size f)
end
with forest_size (f:forest) : nat :=
match f with
| empty => 0
| cons t f' => (tree_size t + forest_size f')
end.
    tree_size is defined
    forest_size is defined
    tree_size, forest_size are recursively defined (guarded respectively on 1st,
    1st arguments)
    tree_size_equation is defined
    tree_size_rect is defined
    tree_size_ind is defined
    tree_size_rec is defined
    forest_size_equation is defined
    forest_size_rect is defined
    forest_size_ind is defined
    forest_size_rec is defined
    R_tree_size_correct is defined
    R_forest_size_correct is defined
    R_tree_size_complete is defined
    R_forest_size_complete is defined
```

Notice that the induction principles `tree_size_ind` and `forest_size_ind` generated by `Function` are not mutual.

```
Check tree_size_ind.
```

(continues on next page)

(continued from previous page)

```
    tree_size_ind
         : forall P : tree -> nat -> Prop,
           (forall (t : tree) (A : A) (f : forest),
            t = node A f -> P (node A f) (S (forest_size f))) ->
           forall t : tree, P t (tree_size t)
```

Mutual induction principles following the recursive structure of `tree_size` and `forest_size` can be generated by the following command:

```
Functional Scheme tree_size_ind2 := Induction for tree_size Sort Prop
with forest_size_ind2 := Induction for forest_size Sort Prop.
    tree_size_ind2 is defined
    forest_size_ind2 is defined
```

You may now look at the type of `tree_size_ind2`:

```
Check tree_size_ind2.
    tree_size_ind2
         : forall (P : tree -> nat -> Prop) (P0 : forest -> nat -> Prop),
           (forall (t : tree) (A : A) (f : forest),
            t = node A f ->
            P0 f (forest_size f) -> P (node A f) (S (forest_size f))) ->
           (forall f0 : forest, f0 = empty -> P0 empty 0) ->
           (forall (f1 : forest) (t : tree) (f' : forest),
            f1 = cons t f' ->
            P t (tree_size t) ->
            P0 f' (forest_size f') ->
            P0 (cons t f') (tree_size t + forest_size f')) ->
           forall t : tree, P t (tree_size t)
```

4.1.5 Writing Coq libraries and plugins

This section presents the part of the Coq language that is useful only to library and plugin authors. A tutorial for writing Coq plugins is available in the Coq repository in doc/plugin_tutorial[57].

Deprecating library objects or tactics

You may use the following *attribute* to deprecate a notation or tactic. When renaming a definition or theorem, you can introduce a deprecated compatibility alias using *Notation (abbreviation)* (see *the example below*).

Attribute: **deprecated** **(** **since** **=** *string* **,**? **note** **=** *string*? **)**

At least one of **since** or **note** must be present. If both are present, either one may appear first and they must be separated by a comma.

This attribute is supported by the following commands: *Ltac*, *Tactic Notation*, *Notation*, *Infix*, *Ltac2*, *Ltac2 Notation*, *Ltac2 external*.

It can trigger the following warnings:

Warning: **Tactic** *qualid* **is deprecated since** $string_{since}$**.** $string_{note}$**.**
Warning: **Tactic Notation** *qualid* **is deprecated since** $string_{since}$**.** $string_{note}$**.**
Warning: **Notation** *string* **is deprecated since** $string_{since}$**.** $string_{note}$**.**

[57] https://github.com/coq/coq/tree/master/doc/plugin_tutorial

Warning: Ltac2 definition *qualid* is deprecated since *string*$_{since}$. *string*$_{note}$.
Warning: Ltac2 alias *qualid* is deprecated since *string*$_{since}$. *string*$_{note}$.
Warning:
Ltac2 notation *ltac2_scope*+ is deprecated since *string*$_{since}$. *string*$_{note}$.
qualid or *string* is the notation, *string*$_{since}$ is the version number, *string*$_{note}$ is the note (usually explains the replacement).

Example: Deprecating a tactic.

```
#[deprecated(since="0.9", note="Use idtac instead.")]
Ltac foo := idtac.
    foo is defined

Goal True.
    1 goal

      ============================
      True

Proof.
now foo.
    Toplevel input, characters 4-7:
    > now foo.
    >     ^^^
    Warning: Tactic foo is deprecated since 0.9. Use idtac instead.
    [deprecated-tactic,deprecated]
    No more goals.
```

Example: Introducing a compatibility alias

Let's say your library initially contained:

```
Definition foo x := S x.
```

and you want to rename `foo` into `bar`, but you want to avoid breaking your users' code without advanced notice. To do so, replace the previous code by the following:

```
Definition bar x := S x.
#[deprecated(since="1.2", note="Use bar instead.")]
Notation foo := bar (only parsing).
```

Then, the following code still works, but emits a warning:

```
Check (foo 0).
    Toplevel input, characters 7-10:
    > Check (foo 0).
    >       ^^^
    Warning: Notation foo is deprecated since 1.2. Use bar instead.
    [deprecated-syntactic-definition,deprecated]
    bar 0
         : nat
```

4.2 Command-line and graphical tools

This chapter presents the command-line tools that users will need to build their Coq project, the documentation of the CoqIDE standalone user interface and the documentation of the parallel proof processing feature that is supported by CoqIDE and several other user interfaces. A list of available user interfaces to interact with Coq is available on the Coq website[58].

4.2.1 The Coq commands

There are three Coq commands:

- `coqtop`: the Coq toplevel (interactive mode);
- `coqc`: the Coq compiler (batch compilation);
- `coqchk`: the Coq checker (validation of compiled libraries).

The options are (basically) the same for the first two commands, and roughly described below. You can also look at the `man` pages of `coqtop` and `coqc` for more details.

Interactive use (coqtop)

In the interactive mode, also known as the Coq toplevel, the user can develop his theories and proofs step by step. The Coq toplevel is run by the command `coqtop`.

There are two different binary images of Coq: the byte-code one and the native-code one (if OCaml provides a native-code compiler for your platform, which is supposed in the following). By default, `coqtop` executes the native-code version; run `coqtop.byte` to get the byte-code version.

The byte-code toplevel is based on an OCaml toplevel (to allow dynamic linking of tactics). You can switch to the OCaml toplevel with the command `Drop.`, and come back to the Coq toplevel with the command `Coqloop.loop();;`.

Flag: `Coqtop Exit On Error`
This *flag*, off by default, causes coqtop to exit with status code `1` if a command produces an error instead of recovering from it.

Batch compilation (coqc)

The `coqc` command compiles a Coq proof script file with a ".v" suffix to create a compiled file with a ".vo" suffix. (See *Compiled files*.) The last component of the filename must be a valid Coq identifier as described in *Lexical conventions*; it should contain only letters, digits or underscores (_) with a ".v" suffix on the final component. For example `/bar/foo/toto.v` is valid, but `/bar/foo/to-to.v` is not.

We recommend specifying a logical directory name (which is also the module name) with the `-R` or the `-Q` options. Generally we recommend using utilities such as `make` (using `coq_makefile` to generate the `Makefile`) or `dune` to build Coq projects. See *Building a Coq project*.

Example: Compiling and loading a single file

If `foo.v` is in Coq's current directory, you can use `coqc foo.v` to compile it and then `Require foo.` in your script. But this doesn't scale well for larger projects.

Generally it's' better to define a new module: To compile `foo.v` as part of a module `Mod1` that is rooted at `.` (i.e. the directory containing `foo.v`), run `coqc -Q . Mod1 foo.v`.

[58] https://coq.inria.fr/user-interfaces.html

To make the module available in CoqIDE, include the following line in the _CoqProject file (see *Building a Coq project with coq_makefile*) in the directory from which you start CoqIDE. *<PATH>* is the pathname of the directory containing the module, which can be an absolute path or relative to Coq's current directory. For now, you must close and reload a named script file for CoqIDE to pick up the change, or restart CoqIDE. The project file name is configurable in Edit / Preferences / Project.

```
-R <PATH> Mod1
```

It's also possible to load a module within coqtop or coqide with commands like these. The drawback of this is that it adds environment-specific information (the PATH) to your script, making it non-portable, so we discourage using this approach.

```
Add LoadPath "PATH" as Mod1.
Require Mod1.foo.
```

in which PATH is the pathname of the directory containing foo.v, which can be absolute or relative to Coq's current directory. The *Add LoadPath* is not needed if you provide the mapping from the logical directory (module name) to the physical directory by including the -Q . Mod1 as command-line arguments to coqtop or coqide.

Customization at launch time

By resource file

When Coq is launched, with either coqtop or coqc, the resource file $XDG_CONFIG_HOME/coq/coqrc.xxx, if it exists, will be implicitly prepended to any document read by Coq, whether it is an interactive session or a file to compile. Here, $XDG_CONFIG_HOME is the configuration directory of the user (by default it's ~/.config) and xxx is the version number (e.g. 8.8). If this file is not found, then the file $XDG_CONFIG_HOME/coqrc is searched. If not found, it is the file ~/.coqrc.xxx which is searched, and, if still not found, the file ~/.coqrc. If the latter is also absent, no resource file is loaded. You can also specify an arbitrary name for the resource file (see option -init-file below).

The resource file may contain, for instance, Add LoadPath commands to add directories to the load path of Coq. It is possible to skip the loading of the resource file with the option -q.

By environment variables

$COQPATH can be used to specify the load path. It is a list of directories separated by : (; on Windows). Coq will also honor $XDG_DATA_HOME and $XDG_DATA_DIRS (see Section *Libraries and filesystem*).

Some Coq commands call other Coq commands. In this case, they look for the commands in directory specified by $COQBIN. If this variable is not set, they look for the commands in the executable path.

$COQ_COLORS can be used to specify the set of colors used by coqtop to highlight its output. It uses the same syntax as the $LS_COLORS variable from GNU's ls, that is, a colon-separated list of assignments of the form **name=attr**[*] where name is the name of the corresponding highlight tag and each attr is an ANSI escape code. The list of highlight tags can be retrieved with the -list-tags command-line option of coqtop.

The string uses ANSI escape codes to represent attributes. For example:

```
export COQ_COLORS="diff.added=4;48;2;0;0;240:diff.removed=41"
```

sets the highlights for added text in diffs to underlined (the 4) with a background RGB color (0, 0, 240) and for removed text in diffs to a red background. Note that if you specify COQ_COLORS, the predefined attributes are ignored.

`$OCAMLRUNPARAM`, described here[59], can be used to specify certain runtime and memory usage parameters. In most cases, experimenting with these settings will likely not cause a significant performance difference and should be harmless.

If the variable is not set, Coq uses the default values[60], except that `space_overhead` is set to 120 and `minor_heap_size` is set to 32Mwords (256MB with 64-bit executables or 128MB with 32-bit executables).

By command line options

The following command-line options are recognized by the commands `coqc` and `coqtop`, unless stated otherwise:

-I *directory*, -include *directory* Add physical path *directory* to the OCaml loadpath.

See also:

Names of libraries and the command Declare ML Module Section *Compiled files*.

-Q *directory dirpath* Add physical path *directory* to the list of directories where Coq looks for a file and bind it to the logical directory *dirpath*. The subdirectory structure of *directory* is recursively available from Coq using absolute names (extending the ***dirpath*** prefix) (see Section *Qualified identifiers*). Note that only those subdirectories and files which obey the lexical conventions of what is an ***ident*** are taken into account. Conversely, the underlying file systems or operating systems may be more restrictive than Coq. While Linux's ext4 file system supports any Coq recursive layout (within the limit of 255 bytes per filename), the default on NTFS (Windows) or HFS+ (MacOS X) file systems is on the contrary to disallow two files differing only in the case in the same directory.

See also:

Section *Names of libraries*.

-R *directory dirpath* Do as `-Q` *directory dirpath* but make the subdirectory structure of *directory* recursively visible so that the recursive contents of physical *directory* is available from Coq using short or partially qualified names.

See also:

Section *Names of libraries*.

-top *dirpath* Set the logical module name to ***dirpath*** for the `coqtop` interactive session. If no module name is specified, `coqtop` will default to `Top`. `coqc` does not accept this option because the logical module name is inferred from the name of the input file and the corresponding `-R` / `-Q` options.

-exclude-dir *directory* Exclude any subdirectory named *directory* while processing options such as -R and -Q. By default, only the conventional version control management directories named CVS and_darcs are excluded.

-nois, -noinit Start from an empty state instead of loading the `Init.Prelude` module.

-init-file *file* Load *file* as the resource file instead of loading the default resource file from the standard configuration directories.

-q Do not to load the default resource file.

-l *file*, -load-vernac-source *file* Load and execute the Coq script from *file.v*.

-lv *file*, -load-vernac-source-verbose *file* Load and execute the Coq script from *file.v*. Write its contents to the standard output as it is executed.

-load-vernac-object *qualid* Load Coq compiled library ***qualid***. This is equivalent to running `Require` ***qualid***.

[59] https://caml.inria.fr/pub/docs/manual-ocaml/runtime.html#s:ocamlrun-options
[60] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html#TYPEcontrol

Note: Note that the relative order of this command-line option and its variants (`-rfrom`, `-ri`, `-re`, etc.) and of the `-set` and `-unset` options matters since the various *Require*, *Require Import*, *Require Export*, *Set* and *Unset* commands will be executed in the order specified on the command-line.

-rfrom *dirpath qualid* Load Coq compiled library ***qualid***. This is equivalent to running *From* ***dirpath*** *Require* ***qualid***. See the *note above* regarding the order of command-line options.

-ri *qualid*, -require-import *qualid* Load Coq compiled library ***qualid*** and import it. This is equivalent to running *Require Import* ***qualid***. See the *note above* regarding the order of command-line options.

-re *qualid*, -require-export *qualid* Load Coq compiled library ***qualid*** and transitively import it. This is equivalent to running *Require Export* ***qualid***. See the *note above* regarding the order of command-line options.

-rifrom *dirpath qualid*, -require-import-from *dirpath qualid* Load Coq compiled library ***qualid*** and import it. This is equivalent to running *From* ***dirpath*** *Require Import* ***qualid***. See the *note above* regarding the order of command-line options.

-refrom *dirpath qualid*, -require-export-from *dirpath qualid* Load Coq compiled library ***qualid*** and transitively import it. This is equivalent to running *From* ***dirpath*** *Require Export* ***qualid***. See the *note above* regarding the order of command-line options.

-batch Exit just after argument parsing. Available for `coqtop` only.

-verbose Output the content of the input file as it is compiled. This option is available for `coqc` only.

-native-compiler (yes|no|ondemand) Enable the *native_compute* reduction machine and precompilation to `.cmxs` files for future use by *native_compute*. Setting `yes` enables *native_compute*; it also causes Coq to precompile the native code for future use; all dependencies need to have been precompiled beforehand. Setting `no` disables *native_compute* which defaults back to *vm_compute*; no files are precompiled. Setting `ondemand` enables *native_compute* but disables precompilation; all missing dependencies will be recompiled every time *native_compute* is called.

Deprecated since version 8.14: This flag has been deprecated in favor of the *Split compilation of native computation files* binary. The toolchain has been adapted to transparently rely on the latter, so if you use *Building a Coq project with coq_makefile* there is nothing to do. Otherwise you should substitute calls to `coqc -native-compiler yes` to calls to `coqc` followed by `coqnative` on the resulting `vo` file.

Changed in version 8.13: The default value is set at configure time, `-config` can be used to retrieve it. All this can be summarized in the following table:

configure	coqc	native_compute	outcome	requirements
yes	yes (default)	native_compute	`.cmxs`	`.cmxs` of deps
yes	no	vm_compute	none	none
yes	ondemand	native_compute	none	none
no	yes, no, ondemand	vm_compute	none	none
ondemand	yes	native_compute	`.cmxs`	`.cmxs` of deps
ondemand	no	vm_compute	none	none
ondemand	ondemand (default)	native_compute	none	none

-native-output-dir Set the directory in which to put the aforementioned `.cmxs` for *native_compute*. Defaults to `.coq-native`.

-vos Indicate Coq to skip the processing of opaque proofs (i.e., proofs ending with `Qed` or `Admitted`), output a `.vos` files instead of a `.vo` file, and to load `.vos` files instead of `.vo` files when interpreting `Require` commands.

-vok Indicate Coq to check a file completely, to load `.vos` files instead of `.vo` files when interpreting `Require` commands, and to output an empty `.vok` files upon success instead of writing a `.vo` file.

-w (all|none|w_1,…,w_n) Configure the display of warnings. This option expects all, none or a comma-separated list of warning names or categories (see Section *Controlling display*).

-color (on|off|auto) *Coqtop only*. Enable or disable color output. Default is auto, meaning color is shown only if the output channel supports ANSI escape sequences.

-diffs (on|off|removed) *Coqtop only*. Controls highlighting of differences between proof steps. `on` highlights added tokens, `removed` highlights both added and removed tokens. Requires that `-color` is enabled. (see Section *Showing differences between proof steps*).

-beautify Pretty-print each command to *file.beautified* when compiling *file.v*, in order to get old-fashioned syntax/definitions/notations.

-emacs, -ide-slave Start a special toplevel to communicate with a specific IDE.

-impredicative-set Change the logical theory of Coq by declaring the sort `Set` impredicative.

> **Warning:** This is known to be inconsistent with some standard axioms of classical mathematics such as the functional axiom of choice or the principle of description.

-type-in-type Collapse the universe hierarchy of Coq.

> **Warning:** This makes the logic inconsistent.

-mangle-names *ident* *Experimental.* Do not depend on this option. Replace Coq's auto-generated name scheme with names of the form *ident0*, *ident1*, etc. Within Coq, the `Mangle Names` flag turns this behavior on, and the `Mangle Names Prefix` option sets the prefix to use. This feature is intended to be used as a linter for developments that want to be robust to changes in the auto-generated name scheme. The options are provided to facilitate tracking down problems.

-set *string* Enable flags and set options. *string* should be ***setting_name*=value**, the value is interpreted according to the type of the option. For flags ***setting_name*** is equivalent to ***setting_name*=true**. For instance `-set "Universe Polymorphism"` will enable `Universe Polymorphism`. Note that the quotes are shell syntax, Coq does not see them. See the *note above* regarding the order of command-line options.

-unset *string* As `-set` but used to disable options and flags. *string* must be **"*setting_name*"**. See the *note above* regarding the order of command-line options.

-compat *version* Load a file that sets a few options to maintain partial backward-compatibility with a previous version. This is equivalent to `Require Import Coq.Compat.CoqXXX` with `XXX` one of the last three released versions (including the current version). Note that the *explanations above* regarding the order of command-line options apply, and this could be relevant if you are resetting some of the compatibility options.

-dump-glob *file* Dump references for global names in file *file* (to be used by coqdoc, see *Documenting Coq files with coqdoc*). By default, if *file.v* is being compiled, *file.glob* is used.

-no-glob Disable the dumping of references for global names.

-image *file* Set the binary image to be used by `coqc` to be *file* instead of the standard one. Not of general use.

-bindir *directory* Set the directory containing Coq binaries to be used by `coqc`. It is equivalent to doing export COQBIN= *directory* before launching `coqc`.

-where Print the location of Coq's standard library and exit.

-config Print the locations of Coq's binaries, dependencies, and libraries, then exit.

-filteropts Print the list of command line arguments that `coqtop` has recognized as options and exit.

-v Print Coq's version and exit.

-list-tags Print the highlight tags known by Coq as well as their currently associated color and exit.

-h, --help Print a short usage and exit.

Compiled interfaces (produced using `-vos`)

Compiled interfaces help saving time while developing Coq formalizations, by compiling the formal statements exported by a library independently of the proofs that it contains.

> **Warning:** Compiled interfaces should only be used for development purposes. At the end of the day, one still needs to proof check all files by producing standard `.vo` files. (Technically, when using `-vos`, fewer universe constraints are collected.) Moreover, this feature is still experimental, it may be subject to change without prior notice.

Principle.

The compilation using `coqc -vos foo.v` produces a file called `foo.vos`, which is similar to `foo.vo` except that all opaque proofs are skipped in the compilation process.

The compilation using `coqc -vok foo.v` checks that the file `foo.v` correctly compiles, including all its opaque proofs. If the compilation succeeds, then the output is a file called `foo.vok`, with empty contents. This file is only a placeholder indicating that `foo.v` has been successfully compiled. (This placeholder is useful for build systems such as `make`.)

When compiling a file `bar.v` that depends on `foo.v` (for example via a `Require Foo.` command), if the compilation command is `coqc -vos bar.v` or `coqc -vok bar.v`, then the file `foo.vos` gets loaded (instead of `foo.vo`). A special case is if file `foo.vos` exists and has empty contents, and `foo.vo` exists, then `foo.vo` is loaded.

Appart from the aforementioned case where `foo.vo` can be loaded in place of `foo.vos`, in general the `.vos` and `.vok` files live totally independently from the `.vo` files.

Dependencies generated by "coq_makefile".

The files `foo.vos` and `foo.vok` both depend on `foo.v`.

Furthermore, if a file `foo.v` requires `bar.v`, then `foo.vos` and `foo.vok` also depend on `bar.vos`.

Note, however, that `foo.vok` does not depend on `bar.vok`. Hence, as detailed further, parallel compilation of proofs is possible.

In addition, `coq_makefile` generates for a file `foo.v` a target `foo.required_vos` which depends on the list of `.vos` files that `foo.vos` depends upon (excluding `foo.vos` itself). As explained next, the purpose of this target is to be able to request the minimal working state for editing interactively the file `foo.v`.

Warning: When writing a custom build system, be aware that `coqdep` only produces dependencies related to `.vos` and `.vok` if the `-vos` command line flag is passed. This is to maintain compatibility with dune (see ocaml/dune#2642 on github[61]).

Typical compilation of a set of file using a build system.

Assume a file `foo.v` that depends on two files `f1.v` and `f2.v`. The command `make foo.required_vos` will compile `f1.v` and `f2.v` using the option `-vos` to skip the proofs, producing `f1.vos` and `f2.vos`. At this point, one is ready to work interactively on the file `foo.v`, even though it was never needed to compile the proofs involved in the files `f1.v` and `f2.v`.

Assume a set of files `f1.v ... fn.v` with linear dependencies. The command `make vos` enables compiling the statements (i.e. excluding the proofs) in all the files. Next, `make -j vok` enables compiling all the proofs in parallel. Thus, calling `make -j vok` directly enables taking advantage of a maximal amount of parallelism during the compilation of the set of files.

Note that this comes at the cost of parsing and typechecking all definitions twice, once for the `.vos` file and once for the `.vok` file. However, if files contain nontrivial proofs, or if the files have many linear chains of dependencies, or if one has many cores available, compilation should be faster overall.

Need for "Proof using"

When a theorem is part of a section, typechecking the statement of this theorem might be insufficient for deducing the type of this statement as of at the end of the section. Indeed, the proof of the theorem could make use of section variables or section hypotheses that are not mentioned in the statement of the theorem.

For this reason, proofs inside section should begin with *`Proof using`* instead of *`Proof`*, where after the `using` clause one should provide the list of the names of the section variables that are required for the proof but are not involved in the typechecking of the statement. Note that it is safe to write `Proof using.` instead of `Proof.` also for proofs that are not within a section.

Warning: `You should use the "Proof using [...]." syntax instead of "Proof. " to enable skipping this proof which is located inside a section. Give as argument to "Proof using" the list of section variables that are not needed to type`

If Coq is invoked using the `-vos` option, whenever it finds the command `Proof.` inside a section, it will compile the proof, that is, refuse to skip it, and it will raise a warning. To disable the warning, one may pass the flag `-w -proof-without-using-in-section`.

Interaction with standard compilation

When compiling a file `foo.v` using `coqc` in the standard way (i.e., without `-vos` nor `-vok`), an empty file `foo.vos` and an empty file `foo.vok` are created in addition to the regular output file `foo.vo`. If `coqc` is subsequently invoked on some other file `bar.v` using option `-vos` or `-vok`, and that `bar.v` requires `foo.v`, if Coq finds an empty file `foo.vos`, then it will load `foo.vo` instead of `foo.vos`.

The purpose of this feature is to allow users to benefit from the `-vos` option even if they depend on libraries that were compiled in the traditional manner (i.e., never compiled using the `-vos` option).

[61] https://github.com/ocaml/dune/issues/2842

Compiled libraries checker (coqchk)

The `coqchk` command takes a list of library paths as argument, described either by their logical name or by their physical filename, which must end in `.vo`. The corresponding compiled libraries (`.vo` files) are searched in the path, recursively processing the libraries they depend on. The content of all these libraries is then type checked. The effect of `coqchk` is only to return with normal exit code in case of success, and with positive exit code if an error has been found. Error messages are not deemed to help the user understand what is wrong. In the current version, it does not modify the compiled libraries to mark them as successfully checked.

Note that non-logical information is not checked. By logical information, we mean the type and optional *body* associated with names. It excludes for instance anything related to the concrete syntax of objects (customized syntax rules, association between short and long names), implicit arguments, etc.

This tool can be used for several purposes. One is to check that a compiled library provided by a third-party has not been forged and that loading it cannot introduce inconsistencies[62]. Another point is to get an even higher level of security. Since `coqtop` can be extended with custom tactics, possibly ill-typed code, it cannot be guaranteed that the produced compiled libraries are correct. `coqchk` is a standalone verifier, and thus it cannot be tainted by such malicious code.

Command-line options `-Q`, `-R`, `-where` and `-impredicative-set` are supported by `coqchk` and have the same meaning as for `coqtop`. As there is no notion of relative paths in object files `-Q` and `-R` have exactly the same meaning.

-norec *module* Check *module* but do not check its dependencies.

-admit *module* Do not check *module* and any of its dependencies, unless explicitly required.

-o At exit, print a summary about the context. List the names of all assumptions and variables (constants without a *body*).

-silent Do not write progress information to the standard output.

Environment variable `$COQLIB` can be set to override the location of the standard library.

The algorithm for deciding which modules are checked or admitted is the following: assuming that `coqchk` is called with argument `M`, option `-norec N`, and `-admit A`. Let us write $\overline{S}$ for the set of reflexive transitive dependencies of set S. Then:

- Modules $C = \overline{M} \backslash \overline{A} \cup M \cup N$ are loaded and type checked before being added to the context.
- And $M \cup N \backslash C$ is the set of modules that are loaded and added to the context without type checking. Basic integrity checks (checksums) are nonetheless performed.

As a rule of thumb, -admit can be used to tell Coq that some libraries have already been checked. So `coqchk A B` can be split in `coqchk A` && `coqchk B -admit A` without type checking any definition twice. Of course, the latter is slightly slower since it makes more disk access. It is also less secure since an attacker might have replaced the compiled library `A` after it has been read by the first command, but before it has been read by the second command.

4.2.2 Utilities

The distribution provides utilities to simplify some tedious works beside proof development, tactics writing or documentation.

[62] Ill-formed non-logical information might for instance bind Coq.Init.Logic.True to short name False, so apparently False is inhabited, but using fully qualified names, Coq.Init.Logic.False will always refer to the absurd proposition, what we guarantee is that there is no proof of this latter constant.

Using Coq as a library

In previous versions, `coqmktop` was used to build custom toplevels - for example for better debugging or custom static linking. Nowadays, the preferred method is to use `ocamlfind`.

The most basic custom toplevel is built using:

```
% ocamlfind ocamlopt -thread -rectypes -linkall -linkpkg \
              -package coq.toplevel \
              topbin/coqtop_bin.ml -o my_toplevel.native
```

For example, to statically link $\mathtt{L}_{\mathtt{tac}}$, you can just do:

```
% ocamlfind ocamlopt -thread -rectypes -linkall -linkpkg \
              -package coq.toplevel,coq.plugins.ltac \
              topbin/coqtop_bin.ml -o my_toplevel.native
```

and similarly for other plugins.

Building a Coq project

As of today it is possible to build Coq projects using two tools:

- coq_makefile, which is distributed by Coq and is based on generating a makefile,
- Dune, the standard OCaml build tool, which, since version 1.9, supports building Coq libraries.

Building a Coq project with coq_makefile

The majority of Coq projects are very similar: a collection of `.v` files and eventually some `.ml` ones (a Coq plugin). The main piece of metadata needed in order to build the project are the command line options to `coqc` (e.g. `-R`, `-Q`, `-I`, see *command line options*). Collecting the list of files and options is the job of the `_CoqProject` file.

A simple example of a `_CoqProject` file follows:

```
-R theories/ MyCode
-arg -w
-arg all
theories/foo.v
theories/bar.v
-I src/
src/baz.mlg
src/bazaux.ml
src/qux_plugin.mlpack
```

where options `-R`, `-Q`, `-I` and `-native-compiler` are natively recognized, as well as file names. The lines of the form `-arg foo` are used in order to tell to literally pass an argument `foo` to `coqc`: in the example, this allows to pass the two-word option `-w all` (see *command line options*).

Note that it is mandatory to specify a `-R/-Q` flag for your project, so its modules are properly qualified. Omitting it will generate object files that are not usable except for expert cases.

The `-native-compiler` option given in the `_CoqProject` file will override the global one passed at configure time.

CoqIDE, Proof-General and VSCoq all understand `_CoqProject` files and can be used to invoke Coq with the desired options.

The `coq_makefile` utility can be used to set up a build infrastructure for the Coq project based on makefiles. The recommended way of invoking `coq_makefile` is the following one:

```
coq_makefile -f _CoqProject -o CoqMakefile
```

Such command generates the following files:

CoqMakefile is a makefile for `GNU Make` with targets to build the project (e.g. generate .vo or .html files from .v or compile .ml* files) and install it in the `user-contrib` directory where the Coq library is installed.

CoqMakefile.conf contains make variables assignments that reflect the contents of the `_CoqProject` file as well as the path relevant to Coq.

Run `coq_makefile --help` for a description of command line options.

The recommended approach is to invoke `CoqMakefile` from a standard `Makefile` of the following form:

Example

```
# KNOWNTARGETS will not be passed along to CoqMakefile
KNOWNTARGETS := CoqMakefile extra-stuff extra-stuff2
# KNOWNFILES will not get implicit targets from the final rule, and so
# depending on them won't invoke the submake
# Warning: These files get declared as PHONY, so any targets depending
# on them always get rebuilt
KNOWNFILES   := Makefile _CoqProject

.DEFAULT_GOAL := invoke-coqmakefile

CoqMakefile: Makefile _CoqProject
        $(COQBIN)coq_makefile -f _CoqProject -o CoqMakefile

invoke-coqmakefile: CoqMakefile
        $(MAKE) --no-print-directory -f CoqMakefile $(filter-out $(KNOWNTARGETS),
 ↪$(MAKECMDGOALS))

.PHONY: invoke-coqmakefile $(KNOWNFILES)

####################################################################
##                      Your targets here                         ##
####################################################################

# This should be the last rule, to handle any targets not declared above
%: invoke-coqmakefile
        @true
```

The advantage of a wrapper, compared to directly calling the generated `Makefile`, is that it provides a target independent of the version of Coq to regenerate a `Makefile` specific to the current version of Coq. Additionally, the master `Makefile` can be extended with targets not specific to Coq. Including the generated makefile with an include directive is discouraged, since the contents of this file, including variable names and status of rules, may change in the future.

Use the optional file `CoqMakefile.local` to extend `CoqMakefile`. In particular, you can declare custom actions to run before or after the build process. Similarly you can customize the install target or even provide new targets. See *CoqMakefile.local* for extension-point documentation. Although you can use all variables defined in `CoqMakefile` in the *recipes* of rules that you write and in the definitions of any variables that you assign with =, many variables are not available for use if you assign variable values with := nor to define the *targets* of rules nor in top-level conditionals such as

`ifeq`. Additionally, you must use secondary expansion[63] to make use of such variables in the prerequisites of rules. To access variables defined in `CoqMakefile` in rule target computation, top-level conditionals, and `:=` variable assignment, for example to add new dependencies to compiled outputs, use the optional file `CoqMakefile.local-late`. See *CoqMakefile.local-late* for a non-exhaustive list of variables.

The extensions of files listed in `_CoqProject` determine how they are built. In particular:

- Coq files must use the `.v` extension
- OCaml files must use the `.ml` or `.mli` extension
- OCaml files that require pre processing for syntax extensions (like `VERNAC EXTEND`) must use the `.mlg` extension
- In order to generate a plugin one has to list all OCaml modules (i.e. `Baz` for `baz.ml`) in a `.mlpack` file (or `.mllib` file).

The use of `.mlpack` files has to be preferred over `.mllib` files, since it results in a "packed" plugin: All auxiliary modules (as `Baz` and `Bazaux`) are hidden inside the plugin's "namespace" (`Qux_plugin`). This reduces the chances of begin unable to load two distinct plugins because of a clash in their auxiliary module names.

CoqMakefile.local

The optional file `CoqMakefile.local` is included by the generated file `CoqMakefile`. It can contain two kinds of directives.

Variable assignment

The variable must belong to the variables listed in the `Parameters` section of the generated makefile. Use `CoqMakefile.local-late` instead to access more variables. These include:

CAMLPKGS can be used to specify third party findlib packages, and is passed to the OCaml compiler on building or linking of modules. Eg: `-package yojson`.

CAMLFLAGS can be used to specify additional flags to the OCaml compiler, like `-bin-annot` or `-w`....

OCAMLWARN it contains a default of `-warn-error +a-3`, useful to modify this setting; beware this is not recommended for projects in Coq's CI.

COQC, COQDEP, COQDOC can be set in order to use alternative binaries (e.g. wrappers)

COQ_SRC_SUBDIRS can be extended by including other paths in which `*.cm*` files are searched. For example `COQ_SRC_SUBDIRS+=user-contrib/Unicoq` lets you build a plugin containing OCaml code that depends on the OCaml code of `Unicoq`

COQFLAGS override the flags passed to `coqc`. By default `-q`.

COQEXTRAFLAGS extend the flags passed to `coqc`

COQCHKFLAGS override the flags passed to `coqchk`. By default `-silent -o`.

COQCHKEXTRAFLAGS extend the flags passed to `coqchk`

COQDOCFLAGS override the flags passed to `coqdoc`. By default `-interpolate -utf8`.

COQDOCEXTRAFLAGS extend the flags passed to `coqdoc`

COQLIBINSTALL, COQDOCINSTALL specify where the Coq libraries and documentation will be installed. By default a combination of `$(DESTDIR)` (if defined) with `$(COQLIB)/user-contrib` and `$(DOCDIR)/user-contrib`.

[63] https://www.gnu.org/software/make/manual/html_node/Secondary-Expansion.html

Rule extension

The following makefile rules can be extended.

Example

```
pre-all::
        echo "This line is print before making the all target"
install-extra::
        cp ThisExtraFile /there/it/goes
```

pre-all:: run before the `all` target. One can use this to configure the project, or initialize sub modules or check dependencies are met.

post-all:: run after the `all` target. One can use this to run a test suite, or compile extracted code.

install-extra:: run after `install`. One can use this to install extra files.

install-doc:: One can use this to install extra doc.

uninstall::

uninstall-doc::

clean::

cleanall::

archclean::

merlin-hook:: One can append lines to the generated `.merlin` file extending this target.

CoqMakefile.local-late

The optional file `CoqMakefile.local-late` is included at the end of the generated file `CoqMakefile`. The following is a partial list of accessible variables:

COQ_VERSION the version of `coqc` being used, which can be used to provide different behavior depending on the Coq version

COQMAKEFILE_VERSION the version of Coq used to generate the Makefile, which can be used to detect version mismatches

ALLDFILES the list of generated dependency files, which can be used, for example, to cause `make` to recompute dependencies when files change by writing `$(ALLDFILES): myfiles` or to indicate that files must be generated before dependencies can be computed by writing `$(ALLDFILES): | mygeneratedfiles`

VOFILES, GLOBFILES, CMOFILES, CMXFILES, OFILES, CMAFILES, CMXAFILES, CMIFILES, CMXSFILES lists of files that are generated by various invocations of the compilers

In addition, the following variables may be useful for deciding what targets to present via `$(shell ...)`; these variables are already accessible in recipes for rules added in `CoqMakefile.local`, but are only accessible from top-level `$(shell ...)` invocations in `CoqMakefile.local-late`:

COQC, COQDEP, COQDOC, CAMLC, CAMLOPTC compiler binaries

COQFLAGS, CAMLFLAGS, COQLIBS, COQDEBUG, OCAMLLIBS flags passed to the Coq or OCaml compilers

Timing targets and performance testing

The generated `Makefile` supports the generation of two kinds of timing data: per-file build-times, and per-line times for an individual file.

The following targets and Makefile variables allow collection of per- file timing data:

- **`TIMED=1`** passing this variable will cause `make` to emit a line describing the user-space build-time and peak memory usage for each file built.

  Note: On `Mac OS`, this works best if you've installed `gnu-time`.

  Example

  For example, the output of `make TIMED=1` may look like this:

  ```
  COQDEP Fast.v
  COQDEP Slow.v
  COQC Slow.v
  Slow.vo (user: 0.34 mem: 395448 ko)
  COQC Fast.v
  Fast.vo (user: 0.01 mem: 45184 ko)
  ```

- **`pretty-timed`** this target stores the output of `make TIMED=1` into `time-of-build.log`, and displays a table of the times and peak memory usages, sorted from slowest to fastest, which is also stored in `time-of-build-pretty.log`. If you want to construct the `log` for targets other than the default one, you can pass them via the variable `TGTS`, e.g., `make pretty-timed TGTS="a.vo b.vo"`.

  Note: This target requires `python` to build the table.

  Note: This target will *append* to the timing log; if you want a fresh start, you must remove the file `time-of-build.log` or `run make cleanall`.

  Note: By default the table displays user times. If the build log contains real times (which it does by default), passing `TIMING_REAL=1` to `make pretty-timed` will use real times rather than user times in the table.

  Note: Passing `TIMING_INCLUDE_MEM=0` to `make` will result in the tables not including peak memory usage information. Passing `TIMING_SORT_BY_MEM=1` to `make` will result in the tables be sorted by peak memory usage rather than by the time taken.

  Example

  For example, the output of `make pretty-timed` may look like this:

```
COQDEP VFILES
COQC Slow.v
Slow.vo (real: 0.52, user: 0.39, sys: 0.12, mem: 394648 ko)
COQC Fast.v
Fast.vo (real: 0.06, user: 0.02, sys: 0.03, mem: 56980 ko)
    Time |  Peak Mem | File Name
-----------------------------------------
0m00.41s | 394648 ko | Total Time / Peak Mem
-----------------------------------------
0m00.39s | 394648 ko | Slow.vo
0m00.02s |  56980 ko | Fast.vo
```

- **print-pretty-timed-diff** this target builds a table of timing changes between two compilations; run `make make-pretty-timed-before` to build the log of the "before" times, and run `make make-pretty-timed-after` to build the log of the "after" times. The table is printed on the command line, and stored in `time-of-build-both.log`. This target is most useful for profiling the difference between two commits in a repository.

  Note: This target requires `python` to build the table.

  Note: The `make-pretty-timed-before` and `make-pretty-timed-after` targets will *append* to the timing log; if you want a fresh start, you must remove the files `time-of-build-before.log` and `time-of-build-after.log` or run `make cleanall` *before* building either the "before" or "after" targets.

  Note: The table will be sorted first by absolute time differences rounded towards zero to a whole-number of seconds, then by times in the "after" column, and finally lexicographically by file name. This will put the biggest changes in either direction first, and will prefer sorting by build-time over subsecond changes in build time (which are frequently noise); lexicographic sorting forces an order on files which take effectively no time to compile.

  If you prefer a different sorting order, you can pass `TIMING_SORT_BY=absolute` to sort by the total time taken, or `TIMING_SORT_BY=diff` to sort by the signed difference in time.

  Note: Just like `pretty-timed`, this table defaults to using user times. Pass `TIMING_REAL=1` to `make` on the command line to show real times instead.

  Note: Just like `pretty-timed`, passing `TIMING_INCLUDE_MEM=0` to `make` will result in the tables not including peak memory usage information. Passing `TIMING_SORT_BY_MEM=1` to `make` will result in the tables be sorted by peak memory usage rather than by the time taken.

  Example

  For example, the output table from `make print-pretty-timed-diff` may look like this:

```
   After |  Peak Mem | File Name              |   Before |  Peak Mem ||    ␣
 ↪Change || Change (mem) |  % Change | % Change (mem)
-------------------------------------------------------------------------------
 ↪-----------------------------------------------
0m00.43s | 394700 ko | Total Time / Peak Mem | 0m00.41s | 394648 ko || +0m00.
 ↪01s ||        52 ko |    +4.87% |         +0.01%
-------------------------------------------------------------------------------
 ↪-----------------------------------------------
0m00.39s | 394700 ko | Fast.vo                | 0m00.02s |  56980 ko || +0m00.
 ↪37s ||    337720 ko | +1850.00% |       +592.69%
0m00.04s |  56772 ko | Slow.vo                | 0m00.39s | 394648 ko || -0m00.
 ↪35s ||   -337876 ko |   -89.74% |        -85.61%
```

The following targets and `Makefile` variables allow collection of per- line timing data:

- **`TIMING=1`** passing this variable will cause `make` to use `coqc -time` to write to a `.v.timing` file for each `.v` file compiled, which contains line-by-line timing information.

  Example

  For example, running `make all TIMING=1` may result in a file like this:

  ```
  Chars 0 - 26 [Require~Coq.ZArith.BinInt.] 0.157 secs (0.128u,0.028s)
  Chars 27 - 68 [Declare~Reduction~comp~:=~vm_c...] 0. secs (0.u,0.s)
  Chars 69 - 162 [Definition~foo0~:=~Eval~comp~i...] 0.153 secs (0.136u,0.019s)
  Chars 163 - 208 [Definition~foo1~:=~Eval~comp~i...] 0.239 secs (0.236u,0.s)
  ```

- `print-pretty-single-time-diff`

  ```
  print-pretty-single-time-diff AFTER=path/to/file.v.after-timing␣
   ↪BEFORE=path/to/file.v.before-timing
  ```

  this target will make a sorted table of the per-line timing differences between the timing logs in the `BEFORE` and `AFTER` files, display it, and save it to the file specified by the `TIME_OF_PRETTY_BUILD_FILE` variable, which defaults to `time-of-build-pretty.log`. To generate the `.v.before-timing` or `.v.after-timing` files, you should pass `TIMING=before` or `TIMING=after` rather than `TIMING=1`.

  Note: The sorting used here is the same as in the `print-pretty-timed-diff` target.

  Note: This target requires python to build the table.

  Note: This target follows the same sorting order as the `print-pretty-timed-diff` target, and supports the same options for the `TIMING_SORT_BY` variable.

  Note: By default, two lines are only considered the same if the character offsets and initial code strings are identical. Passing `TIMING_FUZZ=N` relaxes this constraint by allowing the character locations to differ by up to `N`, as long as the total number of characters and initial code strings continue to match.

This is useful when there are small changes to a file, and you want to match later lines that have not changed even though the character offsets have changed.

Note: By default the table picks up real times, under the assumption that when comparing line-by-line, the real time is a more accurate representation as it includes disk time and time spent in the native compiler. Passing `TIMING_REAL=0` to `make` will use user times rather than real times in the table.

Example

For example, running `print-pretty-single-time-diff` might give a table like this:

```
After     | Code                                                  | Before ␣
↪  || Change    | % Change
---------------------------------------------------------------------------
↪-------------------------
0m00.50s  | Total                                                 | 0m04.
↪17s  || -0m03.66s | -87.96%
---------------------------------------------------------------------------
↪-------------------------
0m00.145s | Chars 069 - 162 [Definition~foo0~:=~Eval~comp~i...] | 0m00.
↪192s || -0m00.04s | -24.47%
0m00.126s | Chars 000 - 026 [Require~Coq.ZArith.BinInt.]        | 0m00.
↪143s || -0m00.01s | -11.88%
  N/A    | Chars 027 - 068 [Declare~Reduction~comp~:=~nati...] | 0m00.s ␣
↪  || +0m00.00s | N/A
0m00.s    | Chars 027 - 068 [Declare~Reduction~comp~:=~vm_c...] |    N/A ␣
↪  || +0m00.00s | N/A
0m00.231s | Chars 163 - 208 [Definition~foo1~:=~Eval~comp~i...] | 0m03.
↪836s || -0m03.60s | -93.97%
```

- **`all.timing.diff`, `path/to/file.v.timing.diff`** The `path/to/file.v.timing.diff` target will make a `.v.timing.diff` file for the corresponding `.v` file, with a table as would be generated by the `print-pretty-single-time-diff` target; it depends on having already made the corresponding `.v.before-timing` and `.v.after-timing` files, which can be made by passing `TIMING=before` and `TIMING=after`. The `all.timing.diff` target will make such timing difference files for all of the `.v` files that the `Makefile` knows about. It will fail if some `.v.before-timing` or `.v.after-timing` files don't exist.

  Note: This target requires python to build the table.

Building a subset of the targets with `-j`

To build, say, two targets foo.vo and bar.vo in parallel one can use `make only TGTS="foo.vo bar.vo" -j`.

Note: `make foo.vo bar.vo -j` has a different meaning for the make utility, in particular it may build a shared prerequisite twice.

Note: For users of coq_makefile with version < 8.7

- Support for "subdirectory" is deprecated. To perform actions before or after the build (like invoking `make` on a subdirectory) one can hook in pre-all and post-all extension points.
- `-extra-phony` and `-extra` are deprecated. To provide additional target (`.PHONY` or not) please use `CoqMakefile.local`.

Note: Due to limitations with the compilation chain, makefiles generated by `coq_makefile` won't correctly compile OCaml plugins with OCaml < 4.07.0 when using more than one job (`-j N` for `N > 1`).

Precompiling for `native_compute`

To compile files for `native_compute`, one can use the `-native-compiler yes` option of Coq, for instance by putting the following in a *CoqMakefile.local* file:

```
COQEXTRAFLAGS += -native-compiler yes
```

The generated `CoqMakefile` installation target will then take care of installing the extra `.coq-native` directories.

Note: As an alternative to modifying any file, one can set the environment variable when calling `make`:

```
COQEXTRAFLAGS="-native-compiler yes" make
```

This can be useful when files cannot be modified, for instance when installing via OPAM a package built with `coq_makefile`:

```
COQEXTRAFLAGS="-native-compiler yes" opam install coq-package
```

Note: This requires all dependencies to be themselves compiled with `-native-compiler yes`.

Building a Coq project with Dune

Note: Dune's Coq support is still experimental; we strongly recommend using Dune 2.3 or later.

Note: The canonical documentation for the Coq Dune extension is maintained upstream; please refer to the Dune manual[64] for up-to-date information. This documentation is up to date for Dune 2.3.

Building a Coq project with Dune requires setting up a Dune project for your files. This involves adding a `dune-project` and `pkg.opam` file to the root (`pkg.opam` can be empty or generated by Dune itself), and then providing `dune` files in the directories your `.v` files are placed. For the experimental version "0.1" of the Coq Dune language, Coq library stanzas look like:

[64] https://dune.readthedocs.io/

```
(coq.theory
 (name <module_prefix>)
 (package <opam_package>)
 (synopsis <text>)
 (modules <ordered_set_lang>)
 (libraries <ocaml_libraries>)
 (flags <coq_flags>))
```

This stanza will build all `.v` files in the given directory, wrapping the library under `<module_prefix>`. If you declare an `<opam_package>`, an `.install` file for the library will be generated; the optional `(modules <ordered_set_lang>)` field allows you to filter the list of modules, and `(libraries <ocaml_libraries>)` allows the Coq theory depend on ML plugins. For the moment, Dune relies on Coq's standard mechanisms (such as `COQPATH`) to locate installed Coq libraries.

By default Dune will skip `.v` files present in subdirectories. In order to enable the usual recursive organization of Coq projects add

```
(include_subdirs qualified)
```

to you `dune` file.

Once your project is set up, `dune build` will generate the `pkg.install` files and all the files necessary for the installation of your project.

Example

A typical stanza for a Coq plugin is split into two parts. An OCaml build directive, which is standard Dune:

```
(library
 (name equations_plugin)
 (public_name equations.plugin)
 (flags :standard -warn-error -3-9-27-32-33-50)
 (libraries coq.plugins.cc coq.plugins.extraction))

(coq.pp (modules g_equations))
```

And a Coq-specific part that depends on it via the `libraries` field:

```
(coq.theory
 (name Equations) ; -R flag
 (package equations)
 (synopsis "Equations Plugin")
 (libraries coq.plugins.extraction equations.plugin)
 (modules :standard \ IdDec NoCycle)) ; exclude some modules that don't build

(include_subdirs qualified)
```

Computing Module dependencies

In order to compute module dependencies (to be used by `make` or `dune`), Coq provides the `coqdep` tool.

`coqdep` computes inter-module dependencies for Coq programs, and prints the dependencies on the standard output in a format readable by make. When a directory is given as argument, it is recursively looked at.

Dependencies of Coq modules are computed by looking at `Require` commands (`Require`, `Require Export`, `Require Import`), but also at the command `Declare ML Module`.

See the man page of `coqdep` for more details and options.

Both Dune and `coq_makefile` use `coqdep` to compute the dependencies among the files part of a Coq project.

Split compilation of native computation files

Coq features a *`native_compute`* tactic to provide fast computation in the kernel. This process performs compilation of Coq terms to OCaml programs using the OCaml compiler, which may cause an important overhead. Hence native compilation is an opt-in configure flag.

When native compilation is activated, Coq generates the compiled files upfront, i.e. during the `coqc` invocation on the corresponding `.v` file. This is impractical because it means one must chose in advance whether they will use a native-capable Coq installation. In particular, activating native compilation forces the recompilation of the whole Coq installation. See *command line options* for more details.

Starting from Coq 8.14, a new binary `coqnative` is available. It allows performing split native compilation by generating the native compute files out of the compiled `.vo` file rather than out of the source `.v` file.

The `coqnative` command takes a name *file.vo* as argument and tries to perform native compilation on it. It assumes that the Coq libraries on which *file.vo* depends have been first compiled to their native files, and will fail otherwise. It accepts the `-R`, `-Q`, `-I` and `-nI` arguments with the same semantics as if the native compilation process had been performed through `coqc`. In particular, it means that:

- `-R` and `-Q` are equivalent
- `-I` is a no-op that is accepted only for scripting convenience

Embedded Coq phrases inside LaTeX documents

When writing documentation about a proof development, one may want to insert Coq phrases inside a LaTeX document, possibly together with the corresponding answers of the system. We provide a mechanical way to process such Coq phrases embedded in LaTeX files: the `coq-tex` filter. This filter extracts Coq phrases embedded in LaTeX files, evaluates them, and insert the outcome of the evaluation after each phrase.

Starting with a file `file.tex` containing Coq phrases, the `coq-tex` filter produces a file named `file.v.tex` with the Coq outcome.

There are options to produce the Coq parts in smaller font, italic, between horizontal rules, etc. See the man page of `coq-tex` for more details.

Man pages

There are man pages for the commands `coqdep` and `coq-tex`. Man pages are installed at installation time (see installation instructions in file `INSTALL`, step 6).

4.2.3 Documenting Coq files with coqdoc

coqdoc is a documentation tool for the proof assistant Coq, similar to `javadoc` or `ocamldoc`. The task of coqdoc is

1. to produce a nice LaTeX and/or HTML document from Coq source files, readable for a human and not only for the proof assistant;
2. to help the user navigate his own (or third-party) sources.

Principles

Documentation is inserted into Coq files as *special comments*. Thus your files will compile as usual, whether you use coqdoc or not. coqdoc presupposes that the given Coq files are well-formed (at least lexically). Documentation starts with `(**`, followed by a space, and ends with `*)`. The documentation format is inspired by Todd A. Coram's *Almost Free Text (AFT)* tool: it is mainly `ASCII` text with some syntax-light controls, described below. coqdoc is robust: it shouldn't fail, whatever the input is. But remember: "garbage in, garbage out".

Coq material inside documentation.

Coq material is quoted between the delimiters `[` and `]`. Square brackets may be nested, the inner ones being understood as being part of the quoted code (thus you can quote a term like `let id := fun [T : Type] (x : t) => x in id 0` by writing `[let id := fun [T : Type] (x : t) => x in id 0]`). Inside quotations, the code is pretty-printed the same way as in code parts.

Preformatted vernacular is enclosed by `[[` and `]]`. The former must be followed by a newline and the latter must follow a newline.

Pretty-printing.

coqdoc uses different faces for identifiers and keywords. The pretty- printing of Coq tokens (identifiers or symbols) can be controlled using one of the following commands:

```
(** printing  *token* %...LATEX...% #...html...# *)
```

or

```
(** printing  *token* $...LATEX math...$ #...html...# *)
```

It gives the LaTeX and HTML texts to be produced for the given Coq token. Either the LaTeX or the HTML rule may be omitted, causing the default pretty-printing to be used for this token.

The printing for one token can be removed with

```
(** remove printing  *token* *)
```

Initially, the pretty-printing table contains the following mapping:

->	→		<-	←		*	×
<=	≤		>=	≥		=>	⇒
<>	≠		<->	↔		|-	⊢
\/	∨		/\	∧		~	¬

Any of these can be overwritten or suppressed using the printing commands.

Note: The recognition of tokens is done by a (`ocaml`) lex automaton and thus applies the longest-match rule. For instance, `->~` is recognized as a single token, where Coq sees two tokens. It is the responsibility of the user to insert space between tokens *or* to give pretty-printing rules for the possible combinations, e.g.

```
(** printing ->~ %\ensuremath{\rightarrow\lnot}% *)
```

Sections

Sections are introduced by 1 to 4 asterisks at the beginning of a line followed by a space and the title of the section. One asterisk is a section, two a subsection, etc.

Example

```
(** * Well-founded relations

    In this section, we introduce...  *)
```

Lists.

List items are introduced by a leading dash. coqdoc uses whitespace to determine the depth of a new list item and which text belongs in which list items. A list ends when a line of text starts at or before the level of indenting of the list's dash. A list item's dash must always be the first non-space character on its line (so, in particular, a list can not begin on the first line of a comment - start it on the second line instead).

Example

```
We go by induction on [n]:
- If [n] is 0...
- If [n] is [S n'] we require...

  two paragraphs of reasoning, and two subcases:

  - In the first case...
  - In the second case...

So the theorem holds.
```

Rules.

More than 4 leading dashes produce a horizontal rule.

Emphasis.

Text can be italicized by enclosing it in underscores. A non-identifier character must precede the leading underscore and follow the trailing underscore, so that uses of underscores in names aren't mistaken for emphasis. Usually, these are spaces or punctuation.

```
This sentence contains some _emphasized text_.
```

Escaping to LaTeX and HTML.

Pure LaTeX or HTML material can be inserted using the following escape sequences:

- `$...LATEX stuff...$` inserts some LaTeX material in math mode. Simply discarded in HTML output.
- `%...LATEX stuff...%` inserts some LaTeX material. Simply discarded in HTML output.
- `#...HTML stuff...#` inserts some HTML material. Simply discarded in LaTeX output.

Note: to simply output the characters `$`, `%` and `#` and escaping their escaping role, these characters must be doubled.

Verbatim

Verbatim material is introduced by a leading << and closed by >> at the beginning of a line.

Example

```
Here is the corresponding caml code:
<<
  let rec fact n =
    if n <= 1 then 1 else n * fact (n-1)
>>
```

Verbatim material on a single line is also possible (assuming that >> is not part of the text to be presented as verbatim).

Example

```
Here is the corresponding caml expression: << fact (n-1) >>
```

Hyperlinks

Hyperlinks can be inserted into the HTML output, so that any identifier is linked to the place of its definition.

`coqc file.v` automatically dumps localization information in `file.glob` or appends it to a file specified using the option `--dump-glob file`. Take care of erasing this global file, if any, when starting the whole compilation process.

Then invoke coqdoc or `coqdoc --glob-from file` to tell coqdoc to look for name resolutions in the file `file` (it will look in `file.glob` by default).

Identifiers from the Coq standard library are linked to the Coq website http://coq.inria.fr/library/. This behavior can be changed using command line options `--no-externals` and `--coqlib`; see below.

Hiding / Showing parts of the source

Some parts of the source can be hidden using command line options `-g` and `-l` (see below), or using such comments:

```
(* begin hide *)
 *some Coq material*
(* end hide *)
```

Conversely, some parts of the source which would be hidden can be shown using such comments:

```
(* begin show *)
 *some Coq material*
(* end show *)
```

The latter cannot be used around some inner parts of a proof, but can be used around a whole proof.

Lastly, it is possible to adopt a middle-ground approach when the desired output is HTML, where a given snippet of Coq material is hidden by default, but can be made visible with user interaction.

```
(* begin details *)
 *some Coq material*
(* end details *)
```

There is also an alternative syntax available.

```
(* begin details : Some summary describing the snippet *)
 *some Coq material*
(* end details *)
```

Usage

coqdoc is invoked on a shell command line as follows: `coqdoc <options and files>`. Any command line argument which is not an option is considered to be a file (even if it starts with a `-`). Coq files are identified by the suffixes `.v` and `.g` and LaTeX files by the suffix `.tex`.

HTML output This is the default output format. One HTML file is created for each Coq file given on the command line, together with a file `index.html` (unless `option-no-index is passed`). The HTML pages use a style sheet named `style.css`. Such a file is distributed with coqdoc.

LaTeX output A single LaTeX file is created, on standard output. It can be redirected to a file using the option `-o`. The order of files on the command line is kept in the final document. LaTeX files given on the command line are copied 'as is' in the final document . DVI and PostScript can be produced directly with the options `-dvi` and `-ps` respectively.

TEXmacs output To translate the input files to TEXmacs format, to be used by the TEXmacs Coq interface.

Command line options

Overall options

--HTML Select a HTML output.

--LaTeX Select a LaTeX output.

--dvi Select a DVI output.

--ps Select a PostScript output.

--texmacs Select a TEXmacs output.

--stdout Write output to stdout.

-o file, --output file Redirect the output into the file 'file' (meaningless with `-html`).

-d dir, --directory dir Output files into directory 'dir' instead of the current directory (option `-d` does not change the filename specified with the option `-o`, if any).

--body-only Suppress the header and trailer of the final document. Thus, you can insert the resulting document into a larger one.

-p string, --preamble string Insert some material in the LaTeX preamble, right before `\begin{document}` (meaningless with `-html`).

--vernac-file file,--tex-file file Considers the file 'file' respectively as a `.v` (or `.g`) file or a `.tex` file.

--files-from file Read filenames to be processed from the file 'file' as if they were given on the command line. Useful for program sources split up into several directories.

-q, --quiet Be quiet. Do not print anything except errors.

-h, --help Give a short summary of the options and exit.

-v, --version Print the version and exit.

Index options

The default behavior is to build an index, for the HTML output only, into `index.html`.

--no-index Do not output the index.

--multi-index Generate one page for each category and each letter in the index, together with a top page `index.html`.

--index string Make the filename of the index string instead of "index". Useful since "index.html" is special.

Table of contents option

-toc, --table-of-contents Insert a table of contents. For a LaTeX output, it inserts a `\tableofcontents` at the beginning of the document. For a HTML output, it builds a table of contents into `toc.html`.

--toc-depth int Only include headers up to depth `int` in the table of contents.

Hyperlink options

--glob-from file Make references using Coq globalizations from file file. (Such globalizations are obtained with Coq option `-dump-glob`).

--no-externals Do not insert links to the Coq standard library.

--external url coqdir Use given URL for linking references whose name starts with prefix `coqdir`.

--coqlib url Set base URL for the Coq standard library (default is http://coq.inria.fr/library/). This is equivalent to `--external url Coq`.

-R dir coqdir Recursively map physical directory dir to Coq logical directory `coqdir` (similarly to Coq option `-R`).

-Q dir coqdir Map physical directory dir to Coq logical directory `coqdir` (similarly to Coq option `-Q`).

Note: options `-R` and `-Q` only have effect on the files *following* them on the command line, so you will probably need to put this option first.

Title options

-s , --short Do not insert titles for the files. The default behavior is to insert a title like "Library Foo" for each file.

--lib-name string Print "string Foo" instead of "Library Foo" in titles. For example "Chapter" and "Module" are reasonable choices.

--no-lib-name Print just "Foo" instead of "Library Foo" in titles.

--lib-subtitles Look for library subtitles. When enabled, the beginning of each file is checked for a comment of the form:

```
(** * ModuleName : text *)
```

where `ModuleName` must be the name of the file. If it is present, the text is used as a subtitle for the module in appropriate places.

-t string, --title string Set the document title.

Contents options

-g, --gallina Do not print proofs.

-l, --light Light mode. Suppress proofs (as with `-g`) and the following commands:

- [Recursive] Tactic Definition
- Hint / Hints
- Require
- Transparent / Opaque
- Implicit Argument / Implicits
- Section / Variable / Hypothesis / End

The behavior of options `-g` and `-l` can be locally overridden using the `(* begin show *)` ... `(* end show *)` environment (see above).

There are a few options that control the parsing of comments:

--parse-comments Parse regular comments delimited by `(*` and `*)` as well. They are typeset inline.

--plain-comments Do not interpret comments, simply copy them as plain-text.

--interpolate Use the globalization information to typeset identifiers appearing in Coq escapings inside comments.

Language options

The default behavior is to assume ASCII 7 bit input files.

-latin1, --latin1 Select ISO-8859-1 input files. It is equivalent to --inputenc latin1 --charset iso-8859-1.

-utf8, --utf8 Set --inputenc utf8x for LaTeX output and--charset utf-8 for HTML output. Also use Unicode replacements for a couple of standard plain ASCII notations such as → for `->` and ∀ for `forall`. LaTeX UTF-8 support can be found at http://www.ctan.org/pkg/unicode. For the interpretation of Unicode characters by LaTeX, extra packages which coqdoc does not provide by default might be required, such as textgreek for some Greek letters or `stmaryrd` for some mathematical symbols. If a Unicode character is missing an interpretation in the utf8x input encoding, add `\DeclareUnicodeCharacter{code}{LATEX-interpretation}`. Packages and declarations can be added with option `-p`.

--inputenc string Give a LaTeX input encoding, as an option to LaTeX package `inputenc`.

--charset string Specify the HTML character set, to be inserted in the HTML header.

The coqdoc LaTeX style file

In case you choose to produce a document without the default LaTeX preamble (by using option `--no-preamble`), then you must insert into your own preamble the command

```
\usepackage{coqdoc}
```

The package optionally takes the argument `[color]` to typeset identifiers with colors (this requires the `xcolor` package).

Then you may alter the rendering of the document by redefining some macros:

coqdockw, coqdocid, … The one-argument macros for typesetting keywords and identifiers. Defaults are sans-serif for keywords and italic for identifiers.For example, if you would like a slanted font for keywords, you may insert

```
\renewcommand{\coqdockw}[1]{\textsl{#1}}
```

anywhere between `\usepackage{coqdoc}` and `\begin{document}`.

coqdocmodule One-argument macro for typesetting the title of a `.v` file. Default is

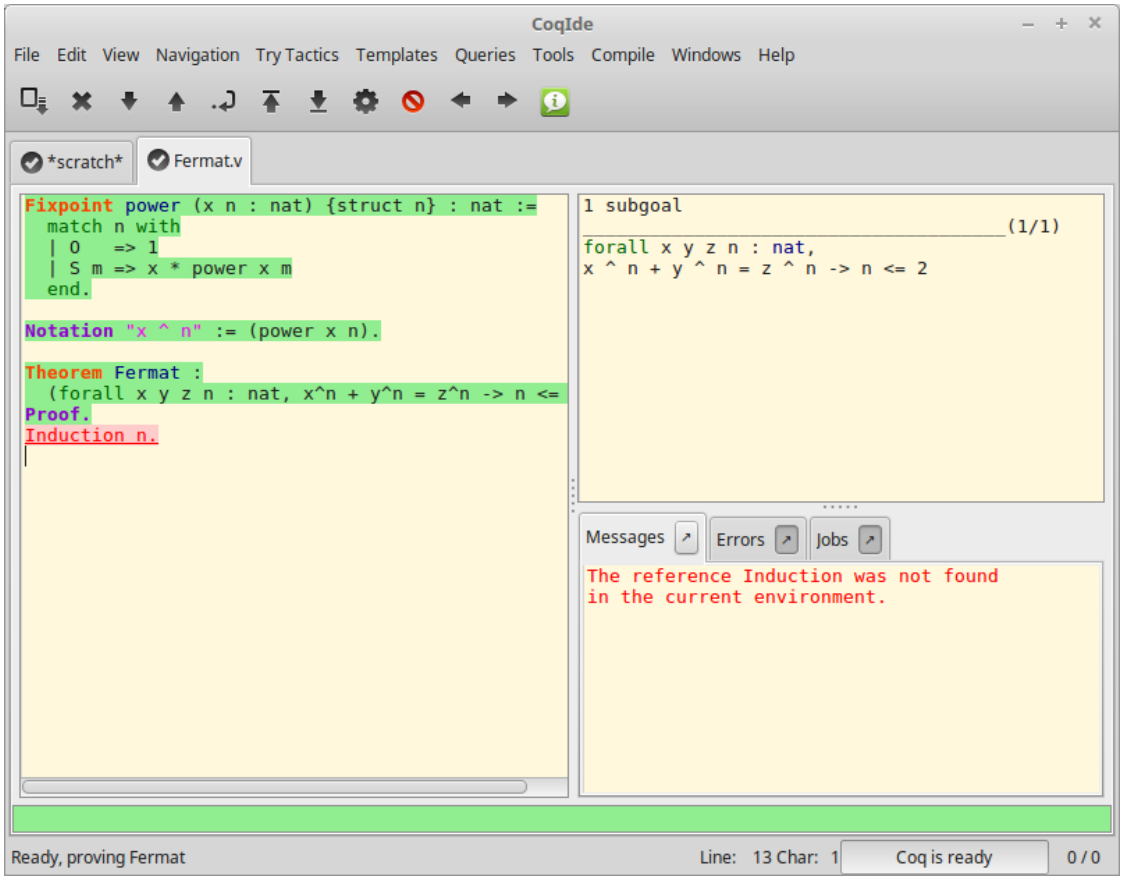
```
\newcommand{\coqdocmodule}[1]{\section*{Module #1}}
```

and you may redefine it using `\renewcommand`.

4.2.4 Coq Integrated Development Environment

The Coq Integrated Development Environment is a graphical tool, to be used as a user-friendly replacement to `coqtop`. Its main purpose is to allow the user to navigate forward and backward into a Coq file, executing corresponding commands or undoing them respectively.

CoqIDE is run by typing the command `coqide` on the command line. Without argument, the main screen is displayed with an "unnamed buffer", and with a filename as argument, another buffer displaying the contents of that file. Additionally, `coqide` accepts the same options as `coqtop`, given in *The Coq commands*, the ones having obviously no meaning for CoqIDE being ignored.



A sample CoqIDE main screen, while navigating into a file `Fermat.v`, is shown in the figure *CoqIDE main screen*. At the top is a menu bar, and a tool bar below it. The large window on the left is displaying the various *script buffers*. The upper right window is the *goal window*, where goals to be proven are displayed. The lower right window is the *message window*, where various messages resulting from commands are displayed. At the bottom is the status bar.

Managing files and buffers, basic editing

In the script window, you may open arbitrarily many buffers to edit. The *File* menu allows you to open files or create some, save them, print or export them into various formats. Among all these buffers, there is always one which is the current *running buffer*, whose name is displayed on a background in the *processed* color (green by default), which is the one where Coq commands are currently executed.

Buffers may be edited as in any text editor, and classical basic editing commands (Copy/Paste, ...) are available in the *Edit* menu. CoqIDE offers only basic editing commands, so if you need more complex editing commands, you may launch your favorite text editor on the current buffer, using the *Edit/External Editor* menu.

Interactive navigation into Coq scripts

The running buffer is the one where navigation takes place. The toolbar offers five basic commands for this. The first one, represented by a down arrow icon, is for going forward executing one command. If that command is successful, the part of the script that has been executed is displayed on a background with the processed color. If that command fails, the error message is displayed in the message window, and the location of the error is emphasized by an underline in the error foreground color (red by default).

In the figure *CoqIDE main screen*, the running buffer is `Fermat.v`, all commands until the `Theorem` have been already executed, and the user tried to go forward executing `Induction n`. That command failed because no such tactic exists (names of standard tactics are written in lowercase), and the failing command is underlined.

Notice that the processed part of the running buffer is not editable. If you ever want to modify something you have to go backward using the up arrow tool, or even better, put the cursor where you want to go back and use the goto button. Unlike with `coqtop`, you should never use `Undo` to go backward.

There are two additional buttons for navigation within the running buffer. The ”down” button with a line goes directly to the end; the ”up” button with a line goes back to the beginning. The handling of errors when using the go-to-the-end button depends on whether Coq is running in asynchronous mode or not (see Chapter *Asynchronous and Parallel Proof Processing*). If it is not running in that mode, execution stops as soon as an error is found. Otherwise, execution continues, and the error is marked with an underline in the error foreground color, with a background in the error background color (pink by default). The same characterization of error-handling applies when running several commands using the ”goto” button.

If you ever try to execute a command that runs for a long time and would like to abort it before it terminates, you may use the interrupt button (the white cross on a red circle).

There are other buttons on the CoqIDE toolbar: a button to save the running buffer; a button to close the current buffer (an ”X”); buttons to switch among buffers (left and right arrows); an ”information” button; and a ”gears” button.

The ”gears” button submits proof terms to the Coq kernel for type checking. When Coq uses asynchronous processing (see Chapter *Asynchronous and Parallel Proof Processing*), proofs may have been completed without kernel-checking of generated proof terms. The presence of unchecked proof terms is indicated by `Qed` statements that have a subdued *being-processed* color (light blue by default), rather than the processed color, though their preceding proofs have the processed color.

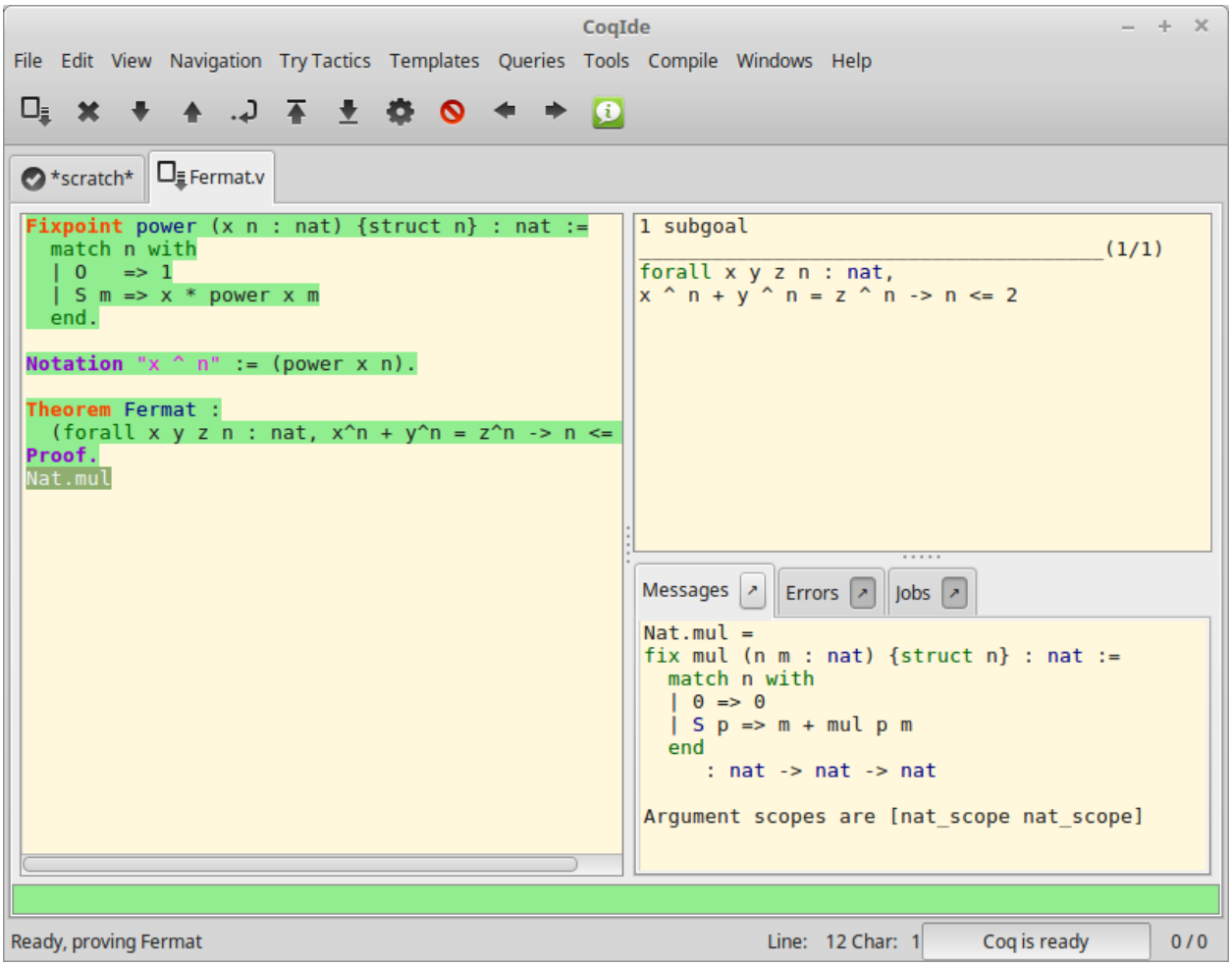
Notice that for all these buttons, except for the ”gears” button, their operations are also available in the menu, where their keyboard shortcuts are given.

Commands and templates

The Templates menu allows using shortcuts to insert commands. This is a nice way to proceed if you are not sure of the syntax of the command you want.

Moreover, from this menu you can automatically insert templates of complex commands like `Fixpoint` that you can conveniently fill afterwards.

Queries



We call *query* any command that does not change the current state, such as `Check`, `Search`, etc. To run such commands interactively, without writing them in scripts, CoqIDE offers a *query pane*. The query pane can be displayed on demand by using the `View` menu, or using the shortcut `F1`. Queries can also be performed by selecting a particular phrase, then choosing an item from the `Queries` menu. The response then appears in the message window. The image above shows the result after selecting of the phrase `Nat.mul` in the script window, and choosing `Print` from the `Queries` menu.

Compilation

The `Compile` menu offers direct commands to:

- compile the current buffer
- run a compilation using `make`
- go to the last compilation error
- create a `Makefile` using `coq_makefile`.

Customizations

You may customize your environment using the menu Edit/Preferences. A new window will be displayed, with several customization sections presented as a notebook.

The first section is for selecting the text font used for scripts, goal and message windows.

The second and third sections are for controlling colors and style of the three main buffers. A predefined Coq highlighting style as well as standard GtkSourceView styles are available. Other styles can be added e.g. in `$HOME/.local/share/gtksourceview-3.0/styles/` (see the general documentation about GtkSourceView for the various possibilities). Note that the style of the rest of graphical part of CoqIDE is not under the control of GtkSourceView but of GTK+ and governed by files such as `settings.ini` and `gtk.css` in `$XDG_CONFIG_HOME/gtk-3.0` or files in `$HOME/.themes/NameOfTheme/gtk-3.0`, as well as the environment variable `GTK_THEME` (search on internet for the various possibilities).

The fourth section is for customizing the editor. It includes in particular the ability to activate an Emacs mode named micro-Proof-General (use the Help menu to know more about the available bindings).

The next section is devoted to file management: you may configure automatic saving of files, by periodically saving the contents into files named `#f#` for each opened file `f`. You may also activate the *revert* feature: in case a opened file is modified on the disk by a third party, CoqIDE may read it again for you. Note that in the case you edited that same file, you will be prompted to choose to either discard your changes or not. The File charset encoding choice is described below in *Character encoding for saved files*.

The `Externals` section allows customizing the external commands for compilation, printing, web browsing. In the browser command, you may use `%s` to denote the URL to open, for example: `firefox -remote "OpenURL(%s)"`.

Notice that these settings are saved in the file `coqiderc` in the `coq` subdirectory of the user configuration directory which is the value of `$XDG_CONFIG_HOME` if this environment variable is set and which otherwise is `$HOME/.config/`.

A GTK+ accelerator keymap is saved under the name `coqide.keys` in the same `coq` subdirectory of the user configuration directory. It is not recommended to edit this file manually: to modify a given menu shortcut, go to the corresponding menu item without releasing the mouse button, press the key you want for the new shortcut, and release the mouse button afterwards. If your system does not allow it, you may still edit this configuration file by hand, but this is more involved.

Using Unicode symbols

CoqIDE is based on GTK+ and inherits from it support for Unicode in its text windows. Consequently a large set of symbols is available for notations. Furthermore, CoqIDE conveniently provides a simple way to input Unicode characters.

Displaying Unicode symbols

You just need to define suitable notations as described in the chapter *Syntax extensions and notation scopes*. For example, to use the mathematical symbols ∀ and ∃, you may define:

```
Notation "∀ x .. y , P" := (forall x, .. (forall y, P) ..)
  (at level 200, x binder, y binder, right associativity)
  : type_scope.
Notation "∃ x .. y , P" := (exists x, .. (exists y, P) ..)
  (at level 200, x binder, y binder, right associativity)
  : type_scope.
```

There exists a small set of such notations already defined, in the file `utf8.v` of Coq library, so you may enable them just by `Require Import Unicode.Utf8` inside CoqIDE, or equivalently, by starting CoqIDE with `coqide -l utf8`.

However, there are some issues when using such Unicode symbols: you of course need to use a character font which supports them. In the Fonts section of the preferences, the Preview line displays some Unicode symbols, so you could figure out if the selected font is OK. Related to this, one thing you may need to do is choosing whether GTK+ should use antialiased fonts or not, by setting the environment variable `GDK_USE_XFT` to 1 or 0 respectively.

Bindings for input of Unicode symbols

CoqIDE supports a builtin mechanism to input non-ASCII symbols. For example, to input π, it suffices to type `\pi` then press the combination of key `Shift+Space` (default key binding). Often, it suffices to type a prefix of the latex token, e.g. typing `\p` then `Shift+Space` suffices to insert a π.

For several symbols, ASCII art is also recognized, e.g. `\->` for a right arrow, or `\>=` for a greater than or equal sign.

A larger number of latex tokens are supported by default. The full list is available here: https://github.com/coq/coq/blob/master/ide/coqide/default_bindings_src.ml

Custom bindings may be added, as explained further on.

Note: It remains possible to input non-ASCII symbols using system-wide approaches independent of CoqIDE.

Adding custom bindings

To extend the default set of bindings, create a file named `coqide.bindings` and place it in the same folder as `coqide.keys`. This would be the folder `$XDG_CONFIG_HOME/coq`, defaulting to `~/.config/coq` if `XDG_CONFIG_HOME` is unset. The file `coqide.bindings` should contain one binding per line, in the form `\key value`, followed by an optional priority integer. (The key and value should not contain any space character.)

Example

Here is an example configuration file:

```
\par ||
\pi π 1
\le ≤ 1
\lambda λ 2
\lambdas λs
```

Above, the priority number 1 on `\pi` indicates that the prefix `\p` should resolve to `\pi`, and not to something else (e.g. `\par`). Similarly, the above settings ensure than `\l` resolves to `\le`, and that `\la` resolves to `\lambda`.

It can be useful to work with per-project binding files. For this purpose CoqIDE accepts a command line argument of the form `-unicode-bindings file1,file2,...,fileN`. Each of the file tokens provided may consists of one of:

- a path to a custom bindings file,
- the token `default`, which resolves to the default bindings file,
- the token `local`, which resolves to the `coqide.bindings` file stored in the user configuration directory.

Warning: If a filename other than the first one includes a "~" to refer to the home directory, it won't be expanded properly. To work around that issue, one should not use comas but instead repeat the flag, in the form: `-unicode-bindings file1 .. -unicode-bindings fileN`.

Note: If two bindings for a same token both have the same priority value (or both have no priority value set), then the binding considered is the one from the file that comes first on the command line.

Character encoding for saved files

In the Files section of the preferences, the encoding option is related to the way files are saved.

If you have no need to exchange files with non UTF-8 aware applications, it is better to choose the UTF-8 encoding, since it guarantees that your files will be read again without problems. (This is because when CoqIDE reads a file, it tries to automatically detect its character encoding.)

If you choose something else than UTF-8, then missing characters will be written encoded by `x{....}` or `x{........}` where each dot is an hexadecimal digit: the number between braces is the hexadecimal Unicode index for the missing character.

4.2.5 Asynchronous and Parallel Proof Processing

Author Enrico Tassi

This chapter explains how proofs can be asynchronously processed by Coq. This feature improves the reactivity of the system when used in interactive mode via CoqIDE. In addition, it allows Coq to take advantage of parallel hardware when used as a batch compiler by decoupling the checking of statements and definitions from the construction and checking of proofs objects.

This feature is designed to help dealing with huge libraries of theorems characterized by long proofs. In the current state, it may not be beneficial on small sets of short files.

This feature has some technical limitations that may make it unsuitable for some use cases.

For example, in interactive mode, some errors coming from the kernel of Coq are signaled late. The type of errors belonging to this category are universe inconsistencies.

At the time of writing, only opaque proofs (ending with `Qed` or `Admitted`) can be processed asynchronously.

Finally, asynchronous processing is disabled when running CoqIDE in Windows. The current implementation of the feature is not stable on Windows. It can be enabled, as described below at *Interactive mode*, though doing so is not recommended.

Proof annotations

To process a proof asynchronously Coq needs to know the precise statement of the theorem without looking at the proof. This requires some annotations if the theorem is proved inside a Section (see Section *Section mechanism*).

When a section ends, Coq looks at the proof object to decide which section variables are actually used and hence have to be quantified in the statement of the theorem. To avoid making the construction of proofs mandatory when ending a section, one can start each proof with the `Proof using` command (Section *Entering and exiting proof mode*) that declares which section variables the theorem uses.

The presence of `Proof` using is needed to process proofs asynchronously in interactive mode.

It is not strictly mandatory in batch mode if it is not the first time the file is compiled and if the file itself did not change. When the proof does not begin with Proof using, the system records in an auxiliary file, produced along with the `.vo` file, the list of section variables used.

Automatic suggestion of proof annotations

The *`Suggest Proof Using`* flag makes Coq suggest, when a `Qed` command is processed, a correct proof annotation. It is up to the user to modify the proof script accordingly.

Proof blocks and error resilience

Coq 8.6 introduced a mechanism for error resilience: in interactive mode Coq is able to completely check a document containing errors instead of bailing out at the first failure.

Two kind of errors are supported: errors occurring in commands and errors occurring in proofs.

To properly recover from a failing tactic, Coq needs to recognize the structure of the proof in order to confine the error to a sub proof. Proof block detection is performed by looking at the syntax of the proof script (i.e. also looking at indentation). Coq comes with four kind of proof blocks, and an ML API to add new ones.

curly blocks are delimited by { and }, see Chapter *Proof mode*

par blocks are atomic, i.e. just one tactic introduced by the `par:` goal selector

indent blocks end with a tactic indented less than the previous one

bullet blocks are delimited by two equal bullet signs at the same indentation level

Caveats

When a command fails the subsequent error messages may be bogus, i.e. caused by the first error. Error resilience for commands can be switched off by passing `-async-proofs-command-error-resilience off` to CoqIDE.

An incorrect proof block detection can result into an incorrect error recovery and hence in bogus errors. Proof block detection cannot be precise for bullets or any other non well parenthesized proof structure. Error resilience can be turned off or selectively activated for any set of block kind passing to CoqIDE one of the following options:

- `-async-proofs-tactic-error-resilience off`
- `-async-proofs-tactic-error-resilience all`
- `-async-proofs-tactic-error-resilience` **blocktype** *

Valid proof block types are: “curly”, “par”, “indent”, and “bullet”.

Interactive mode

At the time of writing the only user interface supporting asynchronous proof processing is CoqIDE.

When CoqIDE is started, two Coq processes are created. The master one follows the user, giving feedback as soon as possible by skipping proofs, which are delegated to the worker process. The worker process, whose state can be seen by clicking on the button in the lower right corner of the main CoqIDE window, asynchronously processes the proofs. If a proof contains an error, it is reported in red in the label of the very same button, that can also be used to see the list of errors and jump to the corresponding line.

If a proof is processed asynchronously the corresponding Qed command is colored using a lighter color than usual. This signals that the proof has been delegated to a worker process (or will be processed lazily if the `-async-proofs lazy` option is used). Once finished, the worker process will provide the proof object, but this will not be automatically checked by the kernel of the main process. To force the kernel to check all the proof objects, one has to click the button with the gears (Fully check the document) on the top bar. Only then all the universe constraints are checked.

Caveats

The number of worker processes can be increased by passing CoqIDE the `-async-proofs-j n` flag. Note that the memory consumption increases too, since each worker requires the same amount of memory as the master process. Also note that increasing the number of workers may reduce the reactivity of the master process to user commands.

To disable this feature, one can pass the `-async-proofs off` flag to CoqIDE. Conversely, on Windows, where the feature is disabled by default, pass the `-async-proofs on` flag to enable it.

Proofs that are known to take little time to process are not delegated to a worker process. The threshold can be configured with `-async-proofs-delegation-threshold`. Default is 0.03 seconds.

Batch mode

> **Warning:** The `-vio` flag is subsumed, for most practical usage, by the the more recent `-vos` flag. See *Compiled interfaces (produced using -vos)*.

> **Warning:** When working with `.vio` files, do not use the `-vos` option at the same time, otherwise stale files might get loaded when executing a `Require`. Indeed, the loading of a nonempty `.vos` file is assigned higher priority than the loading of a `.vio` file.

When Coq is used as a batch compiler by running `coqc`, it produces a `.vo` file for each `.v` file. A `.vo` file contains, among other things, theorem statements and proofs. Hence to produce a .vo Coq need to process all the proofs of the `.v` file.

The asynchronous processing of proofs can decouple the generation of a compiled file (like the `.vo` one) that can be loaded by `Require` from the generation and checking of the proof objects. The `-vio` flag can be passed to `coqc` to produce, quickly, `.vio` files. Alternatively, when using a Makefile produced by `coq_makefile`, the `vio` target can be used to compile all files using the `-vio` flag.

A `.vio` file can be loaded using `Require` exactly as a `.vo` file but proofs will not be available (the Print command produces an error). Moreover, some universe constraints might be missing, so universes inconsistencies might go unnoticed. A `.vio` file does not contain proof objects, but proof tasks, i.e. what a worker process can transform into a proof object.

Compiling a set of files with the `-vio` flag allows one to work, interactively, on any file without waiting for all the proofs to be checked.

When working interactively, one can fully check all the `.v` files by running `coqc` as usual.

Alternatively one can turn each `.vio` into the corresponding `.vo`. All .vio files can be processed in parallel, hence this alternative might be faster. The command `coqc -schedule-vio2vo 2 a b c` can be used to obtain a good scheduling for two workers to produce `a.vo`, `b.vo`, and `c.vo`. When using a Makefile produced by `coq_makefile`, the `vio2vo` target can be used for that purpose. Variable `J` should be set to the number of workers, e.g. `make vio2vo J=2`. The only caveat is that, while the .vo files obtained from `.vio` files are complete (they contain all proof terms and universe constraints), the satisfiability of all universe constraints has not been checked globally (they are checked to be consistent for every single proof). Constraints will be checked when these `.vo` files are (recursively) loaded with `Require`.

There is an extra, possibly even faster, alternative: just check the proof tasks stored in `.vio` files without producing the `.vo` files. This is possibly faster because all the proof tasks are independent, hence one can further partition the job to be done between workers. The `coqc -schedule-vio-checking 6 a b c` command can be used to obtain a good scheduling for 6 workers to check all the proof tasks of `a.vio`, `b.vio`, and `c.vio`. Auxiliary files are used to predict how long a proof task will take, assuming it will take the same amount of time it took last time. When using a Makefile produced by coq_makefile, the `checkproofs` target can be used to check all `.vio` files. Variable `J` should be set to the number of workers, e.g. `make checkproofs J=6`. As when converting `.vio` files to `.vo` files, universe constraints are not checked to be globally consistent. Hence this compilation mode is only useful for quick regression testing and on developments not making heavy use of the `Type` hierarchy.

Limiting the number of parallel workers

Many Coq processes may run on the same computer, and each of them may start many additional worker processes. The `coqworkmgr` utility lets one limit the number of workers, globally.

The utility accepts the `-j` argument to specify the maximum number of workers (defaults to 2). `coqworkmgr` automatically starts in the background and prints an environment variable assignment like `COQWORKMGR_SOCK=localhost:45634`. The user must set this variable in all the shells from which Coq processes will be started. If one uses just one terminal running the bash shell, then `export 'coqworkmgr -j 4'` will do the job.

After that, all Coq processes, e.g. `coqide` and `coqc`, will respect the limit, globally.

CHAPTER

FIVE

APPENDIX

5.1 History and recent changes

This chapter is divided in two parts. The first one is about the *early history of Coq* and is presented in chronological order. The second one provides *release notes about recent versions of Coq* and is presented in reverse chronological order. When updating your copy of Coq to a new version (especially a new major version), it is strongly recommended that you read the corresponding release notes. They may contain advice that will help you understand the differences with the previous version and upgrade your projects.

5.1.1 Early history of Coq

Historical roots

Coq is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification. It is the result of about ten years[65] of research of the Coq project. We shall briefly survey here three main aspects: the *logical language* in which we write our axiomatizations and specifications, the *proof assistant* which allows the development of verified mathematical proofs, and the *program extractor* which synthesizes computer programs obeying their formal specifications, written as logical assertions in the language.

The logical language used by Coq is a variety of type theory, called the *Calculus of Inductive Constructions*. Without going back to Leibniz and Boole, we can date the creation of what is now called mathematical logic to the work of Frege and Peano at the turn of the century. The discovery of antinomies in the free use of predicates or comprehension principles prompted Russell to restrict predicate calculus with a stratification of *types*. This effort culminated with *Principia Mathematica*, the first systematic attempt at a formal foundation of mathematics. A simplification of this system along the lines of simply typed λ-calculus occurred with Church's *Simple Theory of Types*. The λ-calculus notation, originally used for expressing functionality, could also be used as an encoding of natural deduction proofs. This Curry-Howard isomorphism was used by N. de Bruijn in the *Automath* project, the first full-scale attempt to develop and mechanically verify mathematical proofs. This effort culminated with Jutting's verification of Landau's *Grundlagen* in the 1970's. Exploiting this Curry-Howard isomorphism, notable achievements in proof theory saw the emergence of two type-theoretic frameworks; the first one, Martin-Löf's *Intuitionistic Theory of Types*, attempts a new foundation of mathematics on constructive principles. The second one, Girard's polymorphic λ-calculus F_ω, is a very strong functional system in which we may represent higher-order logic proof structures. Combining both systems in a higher-order extension of the Automath language, T. Coquand presented in 1985 the first version of the *Calculus of Constructions*, CoC. This strong logical system allowed powerful axiomatizations, but direct inductive definitions were not possible, and inductive notions had to be defined indirectly through functional encodings, which introduced inefficiencies and awkwardness. The formalism was extended in 1989 by T. Coquand and C. Paulin with primitive inductive definitions, leading to the current *Calculus of Inductive Constructions*. This extended formalism is not rigorously defined here. Rather, numerous concrete examples are discussed. We refer the interested reader to relevant research papers for more information about the formalism, its meta-theoretic properties, and semantics. However, it should not be necessary to understand this theoretical material

[65] At the time of writing, i.e. 1995.

in order to write specifications. It is possible to understand the Calculus of Inductive Constructions at a higher level, as a mixture of predicate calculus, inductive predicate definitions presented as typed PROLOG, and recursive function definitions close to the language ML.

Automated theorem-proving was pioneered in the 1960's by Davis and Putnam in propositional calculus. A complete mechanization (in the sense of a semidecision procedure) of classical first-order logic was proposed in 1965 by J.A. Robinson, with a single uniform inference rule called *resolution*. Resolution relies on solving equations in free algebras (i.e. term structures), using the *unification algorithm*. Many refinements of resolution were studied in the 1970's, but few convincing implementations were realized, except of course that PROLOG is in some sense issued from this effort. A less ambitious approach to proof development is computer-aided proof-checking. The most notable proof-checkers developed in the 1970's were LCF, designed by R. Milner and his colleagues at U. Edinburgh, specialized in proving properties about denotational semantics recursion equations, and the Boyer and Moore theorem-prover, an automation of primitive recursion over inductive data types. While the Boyer-Moore theorem-prover attempted to synthesize proofs by a combination of automated methods, LCF constructed its proofs through the programming of *tactics*, written in a high-level functional meta-language, ML.

The salient feature which clearly distinguishes our proof assistant from say LCF or Boyer and Moore's, is its possibility to extract programs from the constructive contents of proofs. This computational interpretation of proof objects, in the tradition of Bishop's constructive mathematics, is based on a realizability interpretation, in the sense of Kleene, due to C. Paulin. The user must just mark his intention by separating in the logical statements the assertions stating the existence of a computational object from the logical assertions which specify its properties, but which may be considered as just comments in the corresponding program. Given this information, the system automatically extracts a functional term from a consistency proof of its specifications. This functional term may be in turn compiled into an actual computer program. This methodology of extracting programs from proofs is a revolutionary paradigm for software engineering. Program synthesis has long been a theme of research in artificial intelligence, pioneered by R. Waldinger. The Tablog system of Z. Manna and R. Waldinger allows the deductive synthesis of functional programs from proofs in tableau form of their specifications, written in a variety of first-order logic. Development of a systematic *programming logic*, based on extensions of Martin-Löf's type theory, was undertaken at Cornell U. by the Nuprl team, headed by R. Constable. The first actual program extractor, PX, was designed and implemented around 1985 by S. Hayashi from Kyoto University. It allows the extraction of a LISP program from a proof in a logical system inspired by the logical formalisms of S. Feferman. Interest in this methodology is growing in the theoretical computer science community. We can foresee the day when actual computer systems used in applications will contain certified modules, automatically generated from a consistency proof of their formal specifications. We are however still far from being able to use this methodology in a smooth interaction with the standard tools from software engineering, i.e. compilers, linkers, run-time systems taking advantage of special hardware, debuggers, and the like. We hope that Coq can be of use to researchers interested in experimenting with this new methodology.

Versions 1 to 5

Note: This summary was written in 1995 together with the previous section and formed the initial version of the Credits chapter.

A more comprehensive description of these early versions is available in the following subsections, which come from a document written in September 2015 by Gérard Huet, Thierry Coquand and Christine Paulin.

A first implementation of CoC was started in 1984 by G. Huet and T. Coquand. Its implementation language was CAML, a functional programming language from the ML family designed at INRIA in Rocquencourt. The core of this system was a proof-checker for CoC seen as a typed λ-calculus, called the *Constructive Engine*. This engine was operated through a high-level notation permitting the declaration of axioms and parameters, the definition of mathematical types and objects, and the explicit construction of proof objects encoded as λ-terms. A section mechanism, designed and implemented by G. Dowek, allowed hierarchical developments of mathematical theories. This high-level language was called the *Mathematical Vernacular*. Furthermore, an interactive *Theorem Prover* permitted the incremental construction of proof trees in a top-down manner, subgoaling recursively and backtracking from dead-ends. The theorem prover executed

tactics written in CAML, in the LCF fashion. A basic set of tactics was predefined, which the user could extend by his own specific tactics. This system (Version 4.10) was released in 1989. Then, the system was extended to deal with the new calculus with inductive types by C. Paulin, with corresponding new tactics for proofs by induction. A new standard set of tactics was streamlined, and the vernacular extended for tactics execution. A package to compile programs extracted from proofs to actual computer programs in CAML or some other functional language was designed and implemented by B. Werner. A new user-interface, relying on a CAML-X interface by D. de Rauglaudre, was designed and implemented by A. Felty. It allowed operation of the theorem-prover through the manipulation of windows, menus, mouse-sensitive buttons, and other widgets. This system (Version 5.6) was released in 1991.

Coq was ported to the new implementation Caml-light of X. Leroy and D. Doligez by D. de Rauglaudre (Version 5.7) in 1992. A new version of Coq was then coordinated by C. Murthy, with new tools designed by C. Parent to prove properties of ML programs (this methodology is dual to program extraction) and a new user-interaction loop. This system (Version 5.8) was released in May 1993. A Centaur interface CTCoq was then developed by Y. Bertot from the Croap project from INRIA-Sophia-Antipolis.

In parallel, G. Dowek and H. Herbelin developed a new proof engine, allowing the general manipulation of existential variables consistently with dependent types in an experimental version of Coq (V5.9).

The version V5.10 of Coq is based on a generic system for manipulating terms with binding operators due to Chet Murthy. A new proof engine allows the parallel development of partial proofs for independent subgoals. The structure of these proof trees is a mixed representation of derivation trees for the Calculus of Inductive Constructions with abstract syntax trees for the tactics scripts, allowing the navigation in a proof at various levels of details. The proof engine allows generic environment items managed in an object-oriented way. This new architecture, due to C. Murthy, supports several new facilities which make the system easier to extend and to scale up:

- User-programmable tactics are allowed
- It is possible to separately verify development modules, and to load their compiled images without verifying them again - a quick relocation process allows their fast loading
- A generic parsing scheme allows user-definable notations, with a symmetric table-driven pretty-printer
- Syntactic definitions allow convenient abbreviations
- A limited facility of meta-variables allows the automatic synthesis of certain type expressions, allowing generic notations for e.g. equality, pairing, and existential quantification.

In the Fall of 1994, C. Paulin-Mohring replaced the structure of inductively defined types and families by a new structure, allowing the mutually recursive definitions. P. Manoury implemented a translation of recursive definitions into the primitive recursive style imposed by the internal recursion operators, in the style of the ProPre system. C. Muñoz implemented a decision procedure for intuitionistic propositional logic, based on results of R. Dyckhoff. J.C. Filliâtre implemented a decision procedure for first-order logic without contraction, based on results of J. Ketonen and R. Weyhrauch. Finally C. Murthy implemented a library of inversion tactics, relieving the user from tedious definitions of “inversion predicates”.

Rocquencourt, Feb. 1st 1995
Gérard Huet

Version 1

This software is a prototype type checker for a higher-order logical formalism known as the Theory of Constructions, presented in his PhD thesis by Thierry Coquand, with influences from Girard's system F and de Bruijn's Automath. The metamathematical analysis of the system is the PhD work of Thierry Coquand. The software is mostly the work of Gérard Huet. Most of the mathematical examples verified with the software are due to Thierry Coquand.

The programming language of the CONSTR software (as it was called at the time) was a version of ML adapted from the Edinburgh LCF system and running on a LISP backend. The main improvements from the original LCF ML were that ML was compiled rather than interpreted (Gérard Huet building on the original translator by Lockwood Morris), and that it was enriched by recursively defined types (work of Guy Cousineau). This ancestor of CAML was used and improved by Larry Paulson for his implementation of Cambridge LCF.

Software developments of this prototype occurred from late 1983 to early 1985.

Version 1.10 was frozen on December 22nd 1984. It is the version used for the examples in Thierry Coquand's thesis, defended on January 31st 1985. There was a unique binding operator, used both for universal quantification (dependent product) at the level of types and functional abstraction (λ) at the level of terms/proofs, in the manner of Automath. Substitution (λ-reduction) was implemented using de Bruijn's indexes.

Version 1.11 was frozen on February 19th, 1985. It is the version used for the examples in the paper: T. Coquand, G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics* [CH85].

Christine Paulin joined the team at this point, for her DEA research internship. In her DEA memoir (August 1985) she presents developments for the *lambo* function – lambo$(f)(n)$ computes the minimal m such that $f(m)$ is greater than n, for f an increasing integer function, a challenge for constructive mathematics. She also encoded the majority voting algorithm of Boyer and Moore.

Version 2

The formal system, now renamed as the *Calculus of Constructions*, was presented with a proof of consistency and comparisons with proof systems of Per Martin Löf, Girard, and the Automath family of N. de Bruijn, in the paper: T. Coquand and G. Huet. *The Calculus of Constructions* [CH86b].

An abstraction of the software design, in the form of an abstract machine for proof checking, and a fuller sequence of mathematical developments was presented in: T. Coquand, G. Huet. *Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions* [CH86a].

Version 2.8 was frozen on December 16th, 1985, and served for developing the examples in the above papers.

This calculus was then enriched in version 2.9 with a cumulative hierarchy of universes. Universe levels were initially explicit natural numbers. Another improvement was the possibility of automatic synthesis of implicit type arguments, relieving the user of tedious redundant declarations.

Christine Paulin wrote an article *Algorithm development in the Calculus of Constructions* [Moh86]. Besides *lambo* and *majority*, she presents *quicksort* and a text formatting algorithm.

Version 2.13 of the Calculus of Constructions with universes was frozen on June 25th, 1986.

A synthetic presentation of type theory along constructive lines with ML algorithms was given by Gérard Huet in his May 1986 CMU course notes *Formal Structures for Computation and Deduction*. Its chapter *Induction and Recursion in the Theory of Constructions* was presented as an invited paper at the Joint Conference on Theory and Practice of Software Development TAPSOFT'87 at Pise in March 1987, and published as *Induction Principles Formalized in the Calculus of Constructions* [Hue88].

Version 3

This version saw the beginning of proof automation, with a search algorithm inspired from PROLOG and the applicative logic programming programs of the course notes *Formal structures for computation and deduction*. The search algorithm was implemented in ML by Thierry Coquand. The proof system could thus be used in two modes: proof verification and proof synthesis, with tactics such as `AUTO`.

The implementation language was now called CAML, for Categorical Abstract Machine Language. It used as backend the LLM3 virtual machine of Le Lisp by Jérôme Chailloux. The main developers of CAML were Michel Mauny, Ascander Suarez and Pierre Weis.

V3.1 was started in the summer of 1986, V3.2 was frozen at the end of November 1986. V3.4 was developed in the first half of 1987.

Thierry Coquand held a post-doctoral position in Cambridge University in 1986-87, where he developed a variant implementation in SML, with which he wrote some developments on fixpoints in Scott's domains.

Version 4

This version saw the beginning of program extraction from proofs, with two varieties of the type `Prop` of propositions, indicating constructive intent. The proof extraction algorithms were implemented by Christine Paulin-Mohring.

V4.1 was frozen on July 24th, 1987. It had a first identified library of mathematical developments (directory `exemples`), with libraries `Logic` (containing impredicative encodings of intuitionistic logic and algebraic primitives for booleans, natural numbers and list), `Peano` developing second-order Peano arithmetic, `Arith` defining addition, multiplication, euclidean division and factorial. Typical developments were the Knaster-Tarski theorem and Newman's lemma from rewriting theory.

V4.2 was a joint development of a team consisting of Thierry Coquand, Gérard Huet and Christine Paulin-Mohring. A file V4.2.log records the log of changes. It was frozen on September 1987 as the last version implemented in CAML 2.3, and V4.3 followed on CAML 2.5, a more stable development system.

V4.3 saw the first top-level of the system. Instead of evaluating explicit quotations, the user could develop his mathematics in a high-level language called the mathematical vernacular (following Automath terminology). The user could develop files in the vernacular notation (with `.v` extension) which were now separate from the `ml` sources of the implementation. Gilles Dowek joined the team to develop the vernacular language as his DEA internship research.

A notion of sticky constant was introduced, in order to keep names of lemmas when local hypotheses of proofs were discharged. This gave a notion of global mathematical environment with local sections.

Another significant practical change was that the system, originally developed on the VAX central computer of our lab, was transferred on SUN personal workstations, allowing a level of distributed development. The extraction algorithm was modified, with three annotations `Pos`, `Null` and `Typ` decorating the sorts `Prop` and `Type`.

Version 4.3 was frozen at the end of November 1987, and was distributed to an early community of users (among those were Hugo Herbelin and Loic Colson).

V4.4 saw the first version of (encoded) inductive types. Now natural numbers could be defined as:

```
[source, coq]
Inductive NAT : Prop = O : NAT | Succ : NAT->NAT.
```

These inductive types were encoded impredicatively in the calculus, using a subsystem *rec* due to Christine Paulin. V4.4 was frozen on March 6th 1988.

Version 4.5 was the first one to support inductive types and program extraction. Its banner was *Calcul des Constructions avec Réalisations et Synthèse*. The vernacular language was enriched to accommodate extraction commands.

The verification engine design was presented as: G. Huet. *The Constructive Engine*. Version 4.5. Invited Conference, 2nd European Symposium on Programming, Nancy, March 88. The final paper, describing the V4.9 implementation, appeared in: A perspective in Theoretical Computer Science, Commemorative Volume in memory of Gift Siromoney, Ed. R. Narasimhan, World Scientific Publishing, 1989.

Version 4.5 was demonstrated in June 1988 at the YoP Institute on Logical Foundations of Functional Programming organized by Gérard Huet at Austin, Texas.

Version 4.6 was started during the summer of 1988. Its main improvement was the complete rehaul of the proof synthesis engine by Thierry Coquand, with a tree structure of goals.

Its source code was communicated to Randy Pollack on September 2nd 1988. It evolved progressively into LEGO, proof system for Luo's formalism of Extended Calculus of Constructions.

The discharge tactic was modified by Gérard Huet to allow for inter-dependencies in discharged lemmas. Christine Paulin improved the inductive definition scheme in order to accommodate predicates of any arity.

Version 4.7 was started on September 6th, 1988.

This version starts exploiting the CAML notion of module in order to improve the modularity of the implementation. Now the term verifier is identified as a proper module Machine, which the structure of its internal data structures being hidden and thus accessible only through the legitimate operations. This machine (the constructive engine) was the trusted core of the implementation. The proof synthesis mechanism was a separate proof term generator. Once a complete proof term was synthesized with the help of tactics, it was entirely re-checked by the engine. Thus there was no need to certify the tactics, and the system took advantage of this fact by having tactics ignore the universe levels, universe consistency check being relegated to the final type checking pass. This induced a certain puzzlement in early users who saw, after a successful proof search, their `QED` followed by silence, followed by a failure message due to a universe inconsistency…

The set of examples comprise set theory experiments by Hugo Herbelin, and notably the Schroeder-Bernstein theorem.

Version 4.8, started on October 8th, 1988, saw a major re-implementation of the abstract syntax type `constr`, separating variables of the formalism and metavariables denoting incomplete terms managed by the search mechanism. A notion of level (with three values `TYPE`, `OBJECT` and `PROOF`) is made explicit and a type judgement clarifies the constructions, whose implementation is now fully explicit. Structural equality is speeded up by using pointer equality, yielding spectacular improvements. Thierry Coquand adapts the proof synthesis to the new representation, and simplifies pattern matching to first-order predicate calculus matching, with important performance gain.

A new representation of the universe hierarchy is then defined by Gérard Huet. Universe levels are now implemented implicitly, through a hidden graph of abstract levels constrained with an order relation. Checking acyclicity of the graph insures well-foundedness of the ordering, and thus consistency. This was documented in a memo *Adding Type:Type to the Calculus of Constructions* which was never published.

The development version is released as a stable 4.8 at the end of 1988.

Version 4.9 is released on March 1st 1989, with the new "elastic" universe hierarchy.

The spring of 1989 saw the first attempt at documenting the system usage, with a number of papers describing the formalism:

- *Metamathematical Investigations of a Calculus of Constructions*, by Thierry Coquand [Coq89],
- *Inductive definitions in the Calculus of Constructions*, by Christine Paulin-Mohrin,
- *Extracting Fω's programs from proofs in the Calculus of Constructions*, by Christine Paulin-Mohring* [PM89],
- *The Constructive Engine*, by Gérard Huet [Hue89],

as well as a number of user guides:

- *A short user's guide for the Constructions*, Version 4.10, by Gérard Huet
- *A Vernacular Syllabus*, by Gilles Dowek.
- *The Tactics Theorem Prover, User's guide*, Version 4.10, by Thierry Coquand.

Stable V4.10, released on May 1st, 1989, was then a mature system, distributed with CAML V2.6.

In the mean time, Thierry Coquand and Christine Paulin-Mohring had been investigating how to add native inductive types to the Calculus of Constructions, in the manner of Per Martin-Löf's Intuitionistic Type Theory. The impredicative encoding had already been presented in: F. Pfenning and C. Paulin-Mohring. *Inductively defined types in the Calculus of Constructions* [PPM89]. An extension of the calculus with primitive inductive types appeared in: T. Coquand and C. Paulin-Mohring. *Inductively defined types* [CP90].

This led to the Calculus of Inductive Constructions, logical formalism implemented in Versions 5 upward of the system, and documented in: C. Paulin-Mohring. *Inductive Definitions in the System Coq - Rules and Properties* [PM93b].

The last version of CONSTR is Version 4.11, which was last distributed in the spring of 1990. It was demonstrated at the first workshop of the European Basic Research Action Logical Frameworks In Sophia Antipolis in May 1990.

Version 5

At the end of 1989, Version 5.1 was started, and renamed as the system Coq for the Calculus of Inductive Constructions. It was then ported to the new stand-alone implementation of ML called Caml-light.

In 1990 many changes occurred. Thierry Coquand left for Chalmers University in Göteborg. Christine Paulin-Mohring took a CNRS researcher position at the LIP laboratory of École Normale Supérieure de Lyon. Project Formel was terminated, and gave rise to two teams: Cristal at INRIA-Roquencourt, that continued developments in functional programming with Caml-light then OCaml, and Coq, continuing the type theory research, with a joint team headed by Gérard Huet at INRIA-Rocquencourt and Christine Paulin-Mohring at the LIP laboratory of CNRS-ENS Lyon.

Chetan Murthy joined the team in 1991 and became the main software architect of Version 5. He completely rehauled the implementation for efficiency. Versions 5.6 and 5.8 were major distributed versions, with complete documentation and a library of users' developments. The use of the RCS revision control system, and systematic ChangeLog files, allow a more precise tracking of the software developments.

September 2015 +
Thierry Coquand, Gérard Huet and Christine Paulin-Mohring.

Versions 6

Version 6.1

The present version 6.1 of Coq is based on the V5.10 architecture. It was ported to the new language Objective Caml by Bruno Barras. The underlying framework has slightly changed and allows more conversions between sorts.

The new version provides powerful tools for easier developments.

Cristina Cornes designed an extension of the Coq syntax to allow definition of terms using a powerful pattern matching analysis in the style of ML programs.

Amokrane Saïbi wrote a mechanism to simulate inheritance between types families extending a proposal by Peter Aczel. He also developed a mechanism to automatically compute which arguments of a constant may be inferred by the system and consequently do not need to be explicitly written.

Yann Coscoy designed a command which explains a proof term using natural language. Pierre Crégut built a new tactic which solves problems in quantifier-free Presburger Arithmetic. Both functionalities have been integrated to the Coq system by Hugo Herbelin.

Samuel Boutin designed a tactic for simplification of commutative rings using a canonical set of rewriting rules and equality modulo associativity and commutativity.

Finally the organisation of the Coq distribution has been supervised by Jean-Christophe Filliâtre with the help of Judicaël Courant and Bruno Barras.

Lyon, Nov. 18th 1996
Christine Paulin

Version 6.2

In version 6.2 of Coq, the parsing is done using camlp4, a preprocessor and pretty-printer for CAML designed by Daniel de Rauglaudre at INRIA. Daniel de Rauglaudre made the first adaptation of Coq for camlp4, this work was continued by Bruno Barras who also changed the structure of Coq abstract syntax trees and the primitives to manipulate them. The result of these changes is a faster parsing procedure with greatly improved syntax-error messages. The user-interface to introduce grammar or pretty-printing rules has also changed.

Eduardo Giménez redesigned the internal tactic libraries, giving uniform names to Caml functions corresponding to Coq tactic names.

Bruno Barras wrote new, more efficient reduction functions.

Hugo Herbelin introduced more uniform notations in the Coq specification language: the definitions by fixpoints and pattern matching have a more readable syntax. Patrick Loiseleur introduced user-friendly notations for arithmetic expressions.

New tactics were introduced: Eduardo Giménez improved the mechanism to introduce macros for tactics, and designed special tactics for (co)inductive definitions; Patrick Loiseleur designed a tactic to simplify polynomial expressions in an arbitrary commutative ring which generalizes the previous tactic implemented by Samuel Boutin. Jean-Christophe Filliâtre introduced a tactic for refining a goal, using a proof term with holes as a proof scheme.

David Delahaye designed the tool to search an object in the library given its type (up to isomorphism).

Henri Laulhère produced the Coq distribution for the Windows environment.

Finally, Hugo Herbelin was the main coordinator of the Coq documentation with principal contributions by Bruno Barras, David Delahaye, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin and Patrick Loiseleur.

Orsay, May 4th 1998
Christine Paulin

Version 6.3

The main changes in version V6.3 were the introduction of a few new tactics and the extension of the guard condition for fixpoint definitions.

B. Barras extended the unification algorithm to complete partial terms and fixed various tricky bugs related to universes.

D. Delahaye developed the `AutoRewrite` tactic. He also designed the new behavior of `Intro` and provided the tacticals `First` and `Solve`.

J.-C. Filliâtre developed the `Correctness` tactic.

E. Giménez extended the guard condition in fixpoints.

H. Herbelin designed the new syntax for definitions and extended the `Induction` tactic.

P. Loiseleur developed the `Quote` tactic and the new design of the `Auto` tactic, he also introduced the index of errors in the documentation.

C. Paulin wrote the `Focus` command and introduced the reduction functions in definitions, this last feature was proposed by J.-F. Monin from CNET Lannion.

Orsay, Dec. 1999
Christine Paulin

Versions 7

Summary of changes

The version V7 is a new implementation started in September 1999 by Jean-Christophe Filliâtre. This is a major revision with respect to the internal architecture of the system. The Coq version 7.0 was distributed in March 2001, version 7.1 in September 2001, version 7.2 in January 2002, version 7.3 in May 2002 and version 7.4 in February 2003.

Jean-Christophe Filliâtre designed the architecture of the new system. He introduced a new representation for environments and wrote a new kernel for type checking terms. His approach was to use functional data-structures in order to get more sharing, to prepare the addition of modules and also to get closer to a certified kernel.

Hugo Herbelin introduced a new structure of terms with local definitions. He introduced "qualified" names, wrote a new pattern matching compilation algorithm and designed a more compact algorithm for checking the logical consistency of universes. He contributed to the simplification of Coq internal structures and the optimisation of the system. He added basic tactics for forward reasoning and coercions in patterns.

David Delahaye introduced a new language for tactics. General tactics using pattern matching on goals and context can directly be written from the Coq toplevel. He also provided primitives for the design of user-defined tactics in Caml.

Micaela Mayero contributed the library on real numbers. Olivier Desmettre extended this library with axiomatic trigonometric functions, square, square roots, finite sums, Chasles property and basic plane geometry.

Jean-Christophe Filliâtre and Pierre Letouzey redesigned a new extraction procedure from Coq terms to Caml or Haskell programs. This new extraction procedure, unlike the one implemented in previous version of Coq is able to handle all terms in the Calculus of Inductive Constructions, even involving universes and strong elimination. P. Letouzey adapted user contributions to extract ML programs when it was sensible. Jean-Christophe Filliâtre wrote `coqdoc`, a documentation tool for Coq libraries usable from version 7.2.

Bruno Barras improved the efficiency of the reduction algorithm and the confidence level in the correctness of Coq critical type checking algorithm.

Yves Bertot designed the `SearchPattern` and `SearchRewrite` tools and the support for the pcoq interface (http://www-sop.inria.fr/lemme/pcoq/).

Micaela Mayero and David Delahaye introduced Field, a decision tactic for commutative fields.

Christine Paulin changed the elimination rules for empty and singleton propositional inductive types.

Loïc Pottier developed Fourier, a tactic solving linear inequalities on real numbers.

Pierre Crégut developed a new, reflection-based version of the Omega decision procedure.

Claudio Sacerdoti Coen designed an XML output for the Coq modules to be used in the Hypertextual Electronic Library of Mathematics (HELM cf http://www.cs.unibo.it/helm).

A library for efficient representation of finite maps using binary trees contributed by Jean Goubault was integrated in the basic theories.

Pierre Courtieu developed a command and a tactic to reason on the inductive structure of recursively defined functions.

Jacek Chrząszcz designed and implemented the module system of Coq whose foundations are in Judicaël Courant's PhD thesis.

The development was coordinated by C. Paulin.

Many discussions within the Démons team and the LogiCal project influenced significantly the design of Coq especially with J. Courant, J. Duprat, J. Goubault, A. Miquel, C. Marché, B. Monate and B. Werner.

Intensive users suggested improvements of the system : Y. Bertot, L. Pottier, L. Théry, P. Zimmerman from INRIA, C. Alvarado, P. Crégut, J.-F. Monin from France Telecom R & D.

Orsay, May. 2002
Hugo Herbelin & Christine Paulin

Details of changes in 7.0 and 7.1

Notes:

- items followed by (**) are important sources of incompatibilities
- items followed by (*) may exceptionally be sources of incompatibilities
- items followed by (+) have been introduced in version 7.0

Main novelties

References are to Coq 7.1 reference manual

- New primitive let-in construct (see sections 1.2.8 and)
- Long names (see sections 2.6 and 2.7)
- New high-level tactic language (see chapter 10)
- Improved search facilities (see section 5.2)
- New extraction algorithm managing the Type level (see chapter 17)
- New rewriting tactic for arbitrary equalities (see chapter 19)
- New tactic Field to decide equalities on commutative fields (see 7.11)
- New tactic Fourier to solve linear inequalities on reals numbers (see 7.11)
- New tactics for induction/case analysis in "natural" style (see 7.7)
- Deep restructuration of the code (safer, simpler and more efficient)
- Export of theories to XML for publishing and rendering purposes (see http://www.cs.unibo.it/helm)

Details of changes

Language: new ”let-in” construction

- New construction for local definitions (let-in) with syntax [x:=u]t (*)(+)
- Local definitions allowed in Record (a.k.a. record à la Randy Pollack)

Language: long names

- Each construction has a unique absolute names built from a base name, the name of the module in which they are defined (Top if in coqtop), and possibly an arbitrary long sequence of directory (e.g. ”Coq.Lists.PolyList.flat_map” where ”Coq” means that ”flat_map” is part of Coq standard library, ”Lists” means it is defined in the Lists library and ”PolyList” means it is in the file Polylist) (+)
- Constructions can be referred by their base name, or, in case of conflict, by a ”qualified” name, where the base name is prefixed by the module name (and possibly by a directory name, and so on). A fully qualified name is an absolute name which always refer to the construction it denotes (to preserve the visibility of all constructions, no conflict is allowed for an absolute name) (+)
- Long names are available for modules with the possibility of using the directory name as a component of the module full name (with option -R to coqtop and coqc, or command Add LoadPath) (+)
- Improved conflict resolution strategy (the Unix PATH model), allowing more constructions to be referred just by their base name

Language: miscellaneous

- The names of variables for Record projections _and_ for induction principles (e.g. sum_ind) is now based on the first letter of their type (main source of incompatibility) (**)(+)
- Most typing errors have now a precise location in the source (+)
- Slightly different mechanism to solve ”?” (*)(+)
- More arguments may be considered implicit at section closing (*)(+)
- Bug with identifiers ended by a number greater than 2^30 fixed (+)
- New visibility discipline for Remark, Fact and Local: Remark’s and Fact’s now survive at the end of section, but are only accessible using a qualified names as soon as their strength expires; Local’s disappear and are moved into local definitions for each construction persistent at section closing

Language: Cases

- Cases no longer considers aliases inferable from dependencies in types (*)(+)
- A redundant clause in Cases is now an error (*)

Reduction

- New reduction flags ”Zeta” and ”Evar” in Eval Compute, for inlining of local definitions and instantiation of existential variables
- Delta reduction flag does not perform Zeta and Evar reduction any more (*)
- Constants declared as opaque (using Qed) can no longer become transparent (a constant intended to be alternatively opaque and transparent must be declared as transparent (using Defined)); a risk exists (until next Coq version) that Simpl and Hnf reduces opaque constants (*)

New tactics

- New set of tactics to deal with types equipped with specific equalities (a.k.a. Setoids, e.g. nat equipped with eq_nat) [by C. Renard]
- New tactic Assert, similar to Cut but expected to be more user-friendly
- New tactic NewDestruct and NewInduction intended to replace Elim and Induction, Case and Destruct in a more user-friendly way (see restrictions in the reference manual)
- New tactic ROmega: an experimental alternative (based on reflexion) to Omega [by P. Crégut]
- New tactic language Ltac (see reference manual) (+)
- New versions of Tauto and Intuition, fully rewritten in the new Ltac language; they run faster and produce more compact proofs; Tauto is fully compatible but, in exchange of a better uniformity, Intuition is slightly weaker (then use Tauto instead) (**)(+)
- New tactic Field to decide equalities on commutative fields (as a special case, it works on real numbers) (+)
- New tactic Fourier to solve linear inequalities on reals numbers [by L. Pottier] (+)
- New tactics dedicated to real numbers: DiscrR, SplitRmult, SplitAbsolu (+)

Changes in existing tactics

- Reduction tactics in local definitions apply only to the body
- New syntax of the form ”Compute in Type of H.” to require a reduction on the types of local definitions
- Inversion, Injection, Discriminate, ... apply also on the quantified premises of a goal (using the ”Intros until” syntax)
- Decompose has been fixed but hypotheses may get different names (*)(+)
- Tauto now manages uniformly hypotheses and conclusions of the form `t=t` which all are considered equivalent to `True`. Especially, Tauto now solves goals of the form `H : ~ t = t |- A`.
- The ”Let” tactic has been renamed ”LetTac” and is now based on the primitive ”let-in” (+)
- Elim can no longer be used with an elimination schema different from the one defined at definition time of the inductive type. To overload an elimination schema, use ”Elim <hyp> using <name of the new schema>” (*)(+)
- Simpl no longer unfolds the recursive calls of a mutually defined fixpoint (*)(+)
- Intro now fails if the hypothesis name already exists (*)(+)
- ”Require Prolog” is no longer needed (i.e. it is available by default) (*)(+)
- Unfold now fails on a non unfoldable identifier (*)(+)
- Unfold also applies on definitions of the local context

- AutoRewrite now deals only with the main goal and it is the purpose of Hint Rewrite to deal with generated subgoals (+)
- Redundant or incompatible instantiations in Apply ... with ... are now correctly managed (+)

Efficiency

- Excessive memory uses specific to V7.0 fixed
- Sizes of .vo files vary a lot compared to V6.3 (from -30% to +300% depending on the developments)
- An improved reduction strategy for lazy evaluation
- A more economical mechanism to ensure logical consistency at the Type level; warning: this is experimental and may produce ”universes” anomalies (please report)

Concrete syntax of constructions

- Only identifiers starting with ”_” or a letter, and followed by letters, digits, ”_” or ”’” are allowed (e.g. ”$” and ”@” are no longer allowed) (*)
- A multiple binder like (a:A)(a,b:(P a))(Q a) is no longer parsed as (a:A)(a0:(P a))(b:(P a))(Q a0) but as (a:A)(a0:(P a))(b:(P a0))(Q a0) (*)(+)
- A dedicated syntax has been introduced for Reals (e.g `3+1/x`) (+)
- Pretty-printing of Infix notations fixed. (+)

Parsing and grammar extension

- More constraints when writing ast
  - ”{...}” and the macros $LIST, $VAR, etc. now expect a metavariable (an identifier starting with $) (*)
  - **identifiers should starts with a letter or ”_” and be followed** by letters, digits, ”_” or ”’” (other characters are still supported but it is not advised to use them) (*)(+)
- Entry ”command” in ”Grammar” and quotations («...» stuff) is renamed ”constr” as in ”Syntax” (+)
- New syntax ”[” sentence_1 ... sentence_n”].” to group sentences (useful for Time and to write grammar rules abbreviating several commands) (+)
- The default parser for actions in the grammar rules (and for patterns in the pretty-printing rules) is now the one associated with the grammar (i.e. vernac, tactic or constr); no need then for quotations as in <:vernac:<...»; to return an ”ast”, the grammar must be explicitly typed with tag ”: ast” or ”: ast list”, or if a syntax rule, by using «...» in the patterns (expression inside these angle brackets are parsed as ”ast”); for grammars other than vernac, tactic or constr, you may explicitly type the action with tags ”: constr”, ”: tactic”, or ”:vernac” (**)(+)
- Interpretation of names in Grammar rule is now based on long names, which allows to avoid problems (or sometimes tricks;) related to overloaded names (+)

New commands

- New commands ”Print XML All”, ”Show XML Proof”, ... to show or export theories to XML to be used with Helm’s publishing and rendering tools (see http://www.cs.unibo.it/helm) (by Claudio Sacerdoti Coen) (+)
- New commands to manually set implicit arguments (+)
  - ”Implicits ident.” to activate the implicit arguments mode just for ident
  - **”Implicits ident [num1 num2 ...].” to explicitly give which** arguments have to be considered as implicit
- New SearchPattern/SearchRewrite (by Yves Bertot) (+)
- New commands ”Debug on”/”Debug off” to activate/deactivate the tactic language debugger (+)
- New commands to map physical paths to logical paths (+) - Add LoadPath physical_dir as logical_dir - Add Rec LoadPath physical_dir as logical_dir

Changes in existing commands

- Generalization of the usage of qualified identifiers in tactics and commands about globals, e.g. Decompose, Eval Delta; Hints Unfold, Transparent, Require
- Require synchronous with Reset; Require’s scope stops at Section ending (*)
- For a module indirectly loaded by a ”Require” but not exported, the command ”Import module” turns the constructions defined in the module accessible by their short name, and activates the Grammar, Syntax, Hint, ... declared in the module (+)
- The scope of the ”Search” command can be restricted to some modules (+)
- Final dot in command (full stop/period) must be followed by a blank (newline, tabulation or whitespace) (+)
- Slight restriction of the syntax for Cbv Delta: if present, option [-myconst] must immediately follow the Delta keyword (*)(+)
- SearchIsos currently not supported
- Add ML Path is now implied by Add LoadPath (+)
- New names for the following commands (+)

  AddPath -> Add LoadPath Print LoadPath -> Print LoadPath DelPath -> Remove LoadPath AddRecPath -> Add Rec LoadPath Print Path -> Print Coercion Paths

  Implicit Arguments On -> Set Implicit Arguments Implicit Arguments Off -> Unset Implicit Arguments

  Begin Silent -> Set Silent End Silent -> Unset Silent.

Tools

- coqtop (+)
  - Two executables: coqtop.byte and coqtop.opt (if supported by the platform)
  - coqtop is a link to the more efficient executable (coqtop.opt if present)
  - option -full is obsolete (+)
- do_Makefile renamed into coq_makefile (+)
- New option -R to coqtop and coqc to map a physical directory to a logical one (+)

- coqc no longer needs to create a temporary file
- No more warning if no initialization file .coqrc exists

Extraction

- New algorithm for extraction able to deal with "Type" (+) (by J.-C. Filliâtre and P. Letouzey)

Standard library

- New library on maps on integers (IntMap, contributed by Jean Goubault)
- New lemmas about integer numbers [ZArith]
- New lemmas and a "natural" syntax for reals [Reals] (+)
- Exc/Error/Value renamed into Option/Some/None (*)

New user contributions

- Constructive complex analysis and the Fundamental Theorem of Algebra [FTA] (Herman Geuvers, Freek Wiedijk, Jan Zwanenburg, Randy Pollack, Henk Barendregt, Nijmegen)
- A new axiomatization of ZFC set theory [Functions_in_ZFC] (C. Simpson, Sophia-Antipolis)
- Basic notions of graph theory [GRAPHS-BASICS] (Jean Duprat, Lyon)
- A library for floating-point numbers [Float] (Laurent Théry, Sylvie Boldo, Sophia-Antipolis)
- Formalisation of CTL and TCTL temporal logic [CtlTctl] (Carlos Daniel Luna,Montevideo)
- Specification and verification of the Railroad Crossing Problem in CTL and TCTL [RailroadCrossing] (Carlos Daniel Luna,Montevideo)
- P-automaton and the ABR algorithm [PAutomata] (Christine Paulin, Emmanuel Freund, Orsay)
- Semantics of a subset of the C language [MiniC] (Eduardo Giménez, Emmanuel Ledinot, Suresnes)
- Correctness proofs of the following imperative algorithms: Bresenham line drawing algorithm [Bresenham], Marché's minimal edition distance algorithm [Diff] (Jean-Christophe Filliâtre, Orsay)
- Correctness proofs of Buchberger's algorithm [Buchberger] and RSA cryptographic algorithm [Rsa] (Laurent Théry, Sophia-Antipolis)
- Correctness proof of Stalmarck tautology checker algorithm [Stalmarck] (Laurent Théry, Pierre Letouzey, Sophia-Antipolis)

Details of changes in 7.2

Language

- Automatic insertion of patterns for local definitions in the type of the constructors of an inductive types (for compatibility with V6.3 let-in style)
- Coercions allowed in Cases patterns
- New declaration "Canonical Structure id = t : I" to help resolution of equations of the form (proj ?)=a; if proj(e)=a then a is canonically equipped with the remaining fields in e, i.e. ? is instantiated by e

Tactics

- New tactic ”ClearBody H” to clear the body of definitions in local context
- New tactic ”Assert H := c” for forward reasoning
- Slight improvement in naming strategy for NewInduction/NewDestruct
- Intuition/Tauto do not perform useless unfolding and work up to conversion

Extraction (details in plugins/extraction/CHANGES or documentation)

- Syntax changes: there are no more options inside the extraction commands. New commands for customization and options have been introduced instead.
- More optimizations on extracted code.
- Extraction tests are now embedded in 14 user contributions.

Standard library

- In [Relations], Rstar.v and Newman.v now axiom-free.
- In [Sets], Integers.v now based on nat
- In [Arith], more lemmas in Min.v, new file Max.v, tail-recursive plus and mult added to Plus.v and Mult.v respectively
- New directory [Sorting] with a proof of heapsort (dragged from 6.3.1 lib)
- In [Reals], more lemmas in Rbase.v, new lemmas on square, square root and trigonometric functions (R_sqr.v - Rtrigo.v); a complementary approach and new theorems about continuity and derivability in Ranalysis.v; some properties in plane geometry such as translation, rotation or similarity in Rgeom.v; finite sums and Chasles property in Rsigma.v

Bugs

- Confusion between implicit args of locals and globals of same base name fixed
- Various incompatibilities wrt inference of ”?” in V6.3.1 fixed
- Implicits in infix section variables bug fixed
- Known coercions bugs fixed
- Apply ”universe anomaly” bug fixed
- NatRing now working
- ”Discriminate 1”, ”Injection 1”, ”Simplify_eq 1” now working
- NewInduction bugs with let-in and recursively dependent hypotheses fixed
- Syntax [x:=t:T]u now allowed as mentioned in documentation
- Bug with recursive inductive types involving let-in fixed
- Known pattern-matching bugs fixed
- Known Cases elimination predicate bugs fixed
- Improved errors messages for pattern-matching and projections
- Better error messages for ill-typed Cases expressions

Incompatibilities

- New naming strategy for NewInduction/NewDestruct may affect 7.1 compatibility
- Extra parentheses may exceptionally be needed in tactic definitions.

- Coq extensions written in OCaml need to be updated (see dev/changements.txt for a description of the main changes in the interface files of V7.2)
- New behaviour of Intuition/Tauto may exceptionally lead to incompatibilities

Details of changes in 7.3

Language

- Slightly improved compilation of pattern-matching (slight source of incompatibilities)
- Record's now accept anonymous fields "_" which does not build projections
- Changes in the allowed elimination sorts for certain class of inductive definitions : an inductive definition without constructors of Sort Prop can be eliminated on sorts Set and Type A "singleton" inductive definition (one constructor with arguments in the sort Prop like conjunction of two propositions or equality) can be eliminated directly on sort Type (In V7.2, only the sorts Prop and Set were allowed)

Tactics

- New tactic "Rename x into y" for renaming hypotheses
- New tactics "Pose x:=u" and "Pose u" to add definitions to local context
- Pattern now working on partially applied subterms
- Ring no longer applies irreversible congruence laws of mult but better applies congruence laws of plus (slight source of incompatibilities).
- Field now accepts terms to be simplified as arguments (as for Ring). This extension has been also implemented using the toplevel tactic language.
- Intuition does no longer unfold constants except "<->" and "~". It can be parameterized by a tactic. It also can introduce dependent product if needed (source of incompatibilities)
- "Match Context" now matching more recent hypotheses first and failing only on user errors and Fail tactic (possible source of incompatibilities)
- Tactic Definition's without arguments now allowed in Coq states
- Better simplification and discrimination made by Inversion (source of incompatibilities)

Bugs

- "Intros H" now working like "Intro H" trying first to reduce if not a product
- Forward dependencies in Cases now taken into account
- Known bugs related to Inversion and let-in's fixed
- Bug unexpected Delta with let-in now fixed

Extraction (details in plugins/extraction/CHANGES or documentation)

- Signatures of extracted terms are now mostly expunged from dummy arguments.
- Haskell extraction is now operational (tested & debugged).

Standard library

- Some additions in [ZArith]: three files (Zcomplements.v, Zpower.v and Zlogarithms.v) moved from plugins/omega in order to be more visible, one Zsgn function, more induction principles (Wf_Z.v and tail of Zcomplements.v), one more general Euclid theorem
- Peano_dec.v and Compare_dec.v now part of Arith.v

Tools

- new option -dump-glob to coqtop to dump globalizations (to be used by the new documentation tool coqdoc; see http://www.lri.fr/~filliatr/coqdoc)

User Contributions

- CongruenceClosure (congruence closure decision procedure) [Pierre Corbineau, ENS Cachan]
- MapleMode (an interface to embed Maple simplification procedures over rational fractions in Coq) [David Delahaye, Micaela Mayero, Chalmers University]
- Presburger: A formalization of Presburger's algorithm [Laurent Thery, INRIA Sophia Antipolis]
- Chinese has been rewritten using Z from ZArith as datatype ZChinese is the new version, Chinese the obsolete one [Pierre Letouzey, LRI Orsay]

Incompatibilities

- Ring: exceptional incompatibilities (1 above 650 in submitted user contribs, leading to a simplification)
- Intuition: does not unfold any definition except "<->" and "~"
- Cases: removal of some extra Cases in configurations of the form "Cases ... of C _ => ... | _ D => ..." (effects on 2 definitions of submitted user contributions necessitating the removal of now superfluous proof steps in 3 different proofs)
- Match Context, in case of incompatibilities because of a now non trapped error (e.g. Not_found or Failure), use instead tactic Fail to force Match Context trying the next clause
- Inversion: better simplification and discrimination may occasionally lead to less subgoals and/or hypotheses and different naming of hypotheses
- Unification done by Apply/Elim has been changed and may exceptionally lead to incompatible instantiations
- Peano_dec.v and Compare_dec.v parts of Arith.v make Auto more powerful if these files were not already required (1 occurrence of this in submitted user contribs)

Changes in 7.3.1

Bug fixes

- Corrupted Field tactic and Match Context tactic construction fixed
- Checking of names already existing in Assert added (#1386)
- Invalid argument bug in Exact tactic solved (#1387)
- Colliding bound names bug fixed (#1412)
- Wrong non-recursivity test for Record fixed (#1394)
- Out of memory/seg fault bug related to parametric inductive fixed (#1404)
- Setoid_replace/Setoid_rewrite bug wrt "==" fixed

Misc

- Ocaml version >= 3.06 is needed to compile Coq from sources
- Simplification of fresh names creation strategy for Assert, Pose and LetTac (#1402)

Details of changes in 7.4

Symbolic notations

- Introduction of a notion of scope gathering notations in a consistent set; a notation sets has been developed for nat, Z and R (undocumented)
- New command "Notation" for declaring notations simultaneously for parsing and printing (see chap 10 of the reference manual)
- Declarations with only implicit arguments now handled (e.g. the argument of nil can be set implicit; use !nil to refer to nil without arguments)
- "Print Scope sc" and "Locate ntn" allows to know to what expression a notation is bound
- New defensive strategy for printing or not implicit arguments to ensure re-type-checkability of the printed term
- In Grammar command, the only predefined non-terminal entries are ident, global, constr and pattern (e.g. nvar, numarg disappears); the only allowed grammar types are constr and pattern; ast and ast list are no longer supported; some incompatibilities in Grammar: when a syntax is a initial segment of an other one, Grammar does not work, use Notation

Library

- Lemmas in Set from Compare_dec.v (le_lt_dec, ...) and Wf_nat.v (lt_wf_rec, ...) are now transparent. This may be source of incompatibilities.
- Syntactic Definitions Fst, Snd, Ex, All, Ex2, AllT, ExT, ExT2, ProjS1, ProjS2, Error, Value and Except are turned to notations. They now must be applied (incompatibilities only in unrealistic cases).
- More efficient versions of Zmult and times (30% faster)
- Reals: the library is now divided in 6 parts (Rbase, Rfunctions, SeqSeries, Rtrigo, Ranalysis, Integration). New tactics: Sup and RCompute. See Reals.v for details.

Modules

- Beta version, see doc chap 2.5 for commands and chap 5 for theory

Language

- Inductive definitions now accept ">" in constructor types to declare the corresponding constructor as a coercion.
- Idem for assumptions declarations and constants when the type is mentioned.
- The "Coercion" and "Canonical Structure" keywords now accept the same syntax as "Definition", i.e. "hyps :=c (:t)?" or "hyps :t".
- Theorem-like declaration now accepts the syntax "Theorem thm [x:t;...] : u".
- Remark's and Fact's now definitively behave as Theorem and Lemma: when sections are closed, the full name of a Remark or a Fact has no longer a section part (source of incompatibilities)
- Opaque Local's (i.e. built by tactics and ended by Qed), do not survive section closing any longer; as a side-effect, Opaque Local's now appear in the local context of proofs; their body is hidden though (source of incompatibilities); use one of Remark/Fact/Lemma/Theorem instead to simulate the old behaviour of Local (the section part of the name is not kept though)

ML tactics and commands

- "Grammar tactic" and "Grammar vernac" of type "ast" are no longer supported (only "Grammar tactic simple_tactic" of type "tactic" remains available).
- Concrete syntax for ML written commands and tactics is now declared at ML level using camlp4 macros TACTIC EXTEND et VERNAC COMMAND EXTEND.

- ”Check n c” now ”n:Check c”, ”Eval n ...” now ”n:Eval ...”
- `Proof with T` (no documentation)
- SearchAbout id - prints all theorems which contain id in their type

Tactic definitions

- Static globalisation of identifiers and global references (source of incompatibilities, especially, Recursive keyword is required for mutually recursive definitions).
- New evaluation semantics: no more partial evaluation at definition time; evaluation of all Tactic/Meta Definition, even producing terms, expect a proof context to be evaluated (especially ”()” is no longer needed).
- Debugger now shows the nesting level and the reasons of failure

Tactics

- Equality tactics (Rewrite, Reflexivity, Symmetry, Transitivity) now understand JM equality
- Simpl and Change now apply to subterms also
- ”Simpl f” reduces subterms whose head constant is f
- Double Induction now referring to hypotheses like ”Intros until”
- ”Inversion” now applies also on quantified hypotheses (naming as for Intros until)
- NewDestruct now accepts terms with missing hypotheses
- NewDestruct and NewInduction now accept user-provided elimination scheme
- NewDestruct and NewInduction now accept user-provided introduction names
- Omega could solve goals such as `~x<y |- x>=y` but failed when the hypothesis was unfolded to `x < y -> False`. This is fixed. In addition, it can also recognize ’False’ in the hypothesis and use it to solve the goal.
- Coercions now handled in ”with” bindings
- ”Subst x” replaces all occurrences of x by t in the goal and hypotheses when an hypothesis x=t or x:=t or t=x exists
- Fresh names for Assert and Pose now based on collision-avoiding Intro naming strategy (exceptional source of incompatibilities)
- LinearIntuition (no documentation)
- Unfold expects a correct evaluable argument
- Clear expects existing hypotheses

Extraction (See details in plugins/extraction/CHANGES and README):

- An experimental Scheme extraction is provided.
- Concerning OCaml, extracted code is now ensured to always type check, thanks to automatic inserting of Obj.magic.
- Experimental extraction of Coq new modules to Ocaml modules.

Proof rendering in natural language

- Export of theories to XML for publishing and rendering purposes now includes proof-trees (see http://www.cs.unibo.it/helm)

Miscellaneous

- Printing Coercion now used through the standard keywords Set/Add, Test, Print
- ”Print Term id” is an alias for ”Print id”

- New switch ”Unset/Set Printing Symbols” to control printing of symbolic notations
- Two new variants of implicit arguments are available
  - `Unset/Set Contextual Implicits` tells to consider implicit also the arguments inferable from the context (e.g. for nil or refl_eq)
  - `Unset/Set Strict Implicits` tells to consider implicit only the arguments that are inferable in any case (i.e. arguments that occurs as argument of rigid constants in the type of the remaining arguments; e.g. the witness of an existential is not strict since it can vanish when applied to a predicate which does not use its argument)

Incompatibilities

- ”Grammar tactic ... : ast” and ”Grammar vernac ... : ast” are no longer supported, use TACTIC EXTEND and VERNAC COMMAND EXTEND on the ML-side instead
- Transparency of le_lt_dec and co (leads to some simplification in proofs; in some cases, incompatibilites is solved by declaring locally opaque the relevant constant)
- Opaque Local do not now survive section closing (rename them into Remark/Lemma/... to get them still surviving the sections; this renaming allows also to solve incompatibilites related to now forbidden calls to the tactic Clear)
- Remark and Fact have no longer (very) long names (use Local instead in case of name conflict)

Bugs

- Improved localisation of errors in Syntactic Definitions
- Induction principle creation failure in presence of let-in fixed (#1459)
- Inversion bugs fixed (#1427 and #1437)
- Omega bug related to Set fixed (#1384)
- Type-checking inefficiency of nested destructuring let-in fixed (#1435)
- Improved handling of let-in during holes resolution phase (#1460)

Efficiency

- Implementation of a memory sharing strategy reducing memory requirements by an average ratio of 3.

5.1.2 Recent changes

Version 8.14

Summary of changes

Coq version 8.14 integrates many usability improvements, as well as an important change in the core language. The main changes include:

- The *internal representation* of `match` has changed to a more space-efficient and cleaner structure, allowing the fix of a completeness issue with cumulative inductive types in the type-checker. The internal representation is now closer to the user-level view of `match`, where the argument context of branches and the inductive binders `in` and `as` do not carry type annotations.
- A *new* `coqnative` binary performs separate native compilation of libraries, starting from a `.vo` file. It is supported by `coq_makefile`.
- *Improvements* to typeclasses and canonical structure resolution, allowing more terms to be considered as classes or keys.

- More control over *notations* declarations and support for primitive types in string and number notations.
- *Removal* of deprecated tactics, notably `omega`, which has been replaced by a greatly improved `lia`, along with many bug fixes.
- New *Ltac2* APIs for interaction with Ltac1, manipulation of inductive types and printing.
- Many *changes and additions* to the standard library in the numbers, vectors and lists libraries. A new signed primitive integers library `Sint63` is available in addition to the unsigned `Uint63` library.

See the *Changes in 8.14.0* section below for the detailed list of changes, including potentially breaking changes marked with **Changed**. Coq's reference manual[66], documentation of the standard library[67] and developer documentation of the ML API[68] are also available.

Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop and Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure.

Erik Martin-Dorel has maintained the Coq Docker images[69] that are used in many Coq projects for continuous integration.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Karl Palmskog, Matthieu Sozeau and Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The Coq Platform[70] has been maintained by Michael Soegtrop and Enrico Tassi.

Our current maintainers are Yves Bertot, Frédéric Besson, Ali Caglayan, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Jim Fehrle, Julien Forest, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Georges Gonthier, Benjamin Grégoire, Jason Gross, Hugo Herbelin, Vincent Laporte, Olivier Laurent, Assia Mahboubi, Kenji Maillard, Guillaume Melquiond, Pierre-Marie Pédrot, Clément Pit-Claudel, Pierre Roux, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-yao Xia and Théo Zimmermann. See the Coq Team face book[71] page for more details.

The 54 contributors to this version are Reynald Affeldt, Arthur Azevedo de Amorim, Yves Bertot, Frédéric Besson, Lasse Blaauwbroek, Ana Borges, Ali Caglayan, Cyril Cohen, Columbus240, Pierre Courtieu, Maxime Dénès, Andrej Dudenhefner, Jim Fehrle, Yannick Forster, Simon Friis Vindum, Gaëtan Gilbert, Jason Gross, Samuel Gruetter, Stefan Haan, Hugo Herbelin, Jasper Hugunin, Emilio Jesús Gallego Arias, Jacques-Henri Jourdan, Ralf Jung, Jan-Oliver Kaiser, Fabian Kunze, Vincent Laporte, Olivier Laurent, Yishuai Li, Barry M. Trager, Kenji Maillard, Erik Martin-Dorel, Guillaume Melquiond, Isaac Oscar Gariano, Pierre-Marie Pédrot, Rudy Peterson, Clément Pit-Claudel, Pierre Roux, Takafumi Saikawa, Kazuhiko Sakaguchi, Gabriel Scherer, Vincent Semeria, shenlebantongying, Avi Shinnar, slrnsc, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Hendrik Tews, Anton Trunov, Karolin Varner, Li-yao Xia, Beta Ziliani and Théo Zimmermann.

The Coq community at large helped improve the design of this new version via the GitHub issue and pull request system, the Coq development mailing list coqdev@inria.fr, the coq-club@inria.fr mailing list, the Discourse forum[72] and the Coq Zulip chat[73].

Version 8.14's development spanned 9 months from the release of Coq 8.13.0. Guillaume Melquiond is the release manager of Coq 8.14. This release is the result of 522 merged PRs, closing ~150 issues.

Nantes, September 2021,
Matthieu Sozeau for the Coq development team

[66] https://coq.github.io/doc/v8.14/refman
[67] https://coq.github.io/doc/v8.14/stdlib
[68] https://coq.github.io/doc/v8.14/api
[69] https://hub.docker.com/r/coqorg/coq
[70] https://github.com/coq/platform
[71] https://coq.inria.fr/coq-team.html
[72] https://coq.discourse.group/
[73] http://coq.zulipchat.com

Changes in 8.14.0

- *Kernel*
- *Specification language, type inference*
- *Notations*
- *Tactics*
- *Tactic language*
- *SSReflect*
- *Commands and options*
- *Command-line tools*
- *Native Compilation*
- *CoqIDE*
- *Standard library*
- *Infrastructure and dependencies*
- *Miscellaneous*

Kernel

- **Changed:** The term representation of pattern-matchings now uses a compact form that provides a few static guarantees such as eta-expansion of branches and return clauses and is usually more efficient. The most visible user change is that for the time being, the `destruct` tactic and its variants generate dummy cuts (β redexes) in the branches of the generated proof. This can also generate very uncommon backwards incompatibilities, such as a change of occurrence numbering for subterms, or breakage of unification in complex situations involving pattern-matchings whose underlying inductive type declares let-bindings in parameters, arity or constructor types. For ML plugin developers, an in-depth description of the new representation, as well as porting tips, can be found in dev/doc/case-repr.md (#13563[74], fixes #3166[75], by Pierre-Marie Pédrot).
- **Changed:** Linking of native-code libraries used by `native_compute` is now delayed until an actual call to the `native_compute` machinery is performed. This should make Coq more responsive on some systems (#13853[76], fixes #13849[77], by Guillaume Melquiond).
- **Removed:** The ability to change typing flags inside sections to prevent exploiting a weakness in `Print Assumptions` (#14395[78], fixes #14317[79], by Gaëtan Gilbert).

[74] https://github.com/coq/coq/pull/13563
[75] https://github.com/coq/coq/issues/3166
[76] https://github.com/coq/coq/pull/13853
[77] https://github.com/coq/coq/issues/13849
[78] https://github.com/coq/coq/pull/14395
[79] https://github.com/coq/coq/issues/14317

Specification language, type inference

- **Changed:** The hints mode `!` matches a term iff the applicative head is not an existential variable. It now also matches projections applied to any term or a `match` on any term. (#14392[80], by Matthieu Sozeau).
- **Removed:** The little used `:>` type cast, which was only interpreted in Program-mode (#13911[81], by Jim Fehrle and Théo Zimmermann).
- **Added:** Enable canonical `fun _ => _` projections, see *Canonical Structures* for details. (#14041[82], by Jan-Oliver Kaiser and Pierre Roux, reviewed by Cyril Cohen and Enrico Tassi).
- **Added:** *`Canonical Structure`* declarations now accept dependent function types `forall _, _` as keys. (#14386[83], by Jan-Oliver Kaiser and Kazuhiko Sakaguchi).
- **Added:** Ability to declare primitive projections as class, for dependent typeclass resolutions (#9711[84], fixes #12975[85], by Matthieu Sozeau).
- **Fixed:** Multiple printing of same warning about unused variables catching several cases (#14261[86], fixes #14207[87], by Hugo Herbelin).
- **Fixed:** Constants `id` and `not` were unduly set opaque in some parts of the unification algorithm (#14371[88], fixes #14374[89], by Hugo Herbelin).

Notations

- **Changed:** Flag *`Printing Notations`* no longer controls whether strings and numbers are printed raw (#13840[90], by Enrico Tassi).
- **Changed:** The error `Argument X was previously inferred to be in scope XXX_scope but is here used in YYY_scope.` is now the warning `[inconsistent-scopes,syntax]` and can be silenced by specifying the scope of the argument (#13965[91], by Enrico Tassi).
- **Removed:** Decimal-only number notations which were deprecated in 8.12. (#13842[92], by Pierre Roux).
- **Added:** *`Number Notation`* and *`String Notation`* now support parsing and printing of primitive floats, primitive arrays and type constants of primitive types. (#13519[93], fixes #13484[94] and #13517[95], by Fabian Kunze, with help of Jason Gross)
- **Added:** Flag *`Printing Raw Literals`* to control whether strings and numbers are printed raw. (#13840[96], by Enrico Tassi).

[80] https://github.com/coq/coq/pull/14392
[81] https://github.com/coq/coq/pull/13911
[82] https://github.com/coq/coq/pull/14041
[83] https://github.com/coq/coq/pull/14386
[84] https://github.com/coq/coq/pull/9711
[85] https://github.com/coq/coq/issues/12975
[86] https://github.com/coq/coq/pull/14261
[87] https://github.com/coq/coq/issues/14207
[88] https://github.com/coq/coq/pull/14371
[89] https://github.com/coq/coq/issues/14374
[90] https://github.com/coq/coq/pull/13840
[91] https://github.com/coq/coq/pull/13965
[92] https://github.com/coq/coq/pull/13842
[93] https://github.com/coq/coq/pull/13519
[94] https://github.com/coq/coq/issues/13484
[95] https://github.com/coq/coq/issues/13517
[96] https://github.com/coq/coq/pull/13840

- **Added:** Let the user specify a scope for abbreviation arguments, e.g. `Notation abbr X := t (X in scope my_scope).` (#13965[97], by Enrico Tassi).
- **Added:** Look-ahead of tokens is changed from sequential to tree-based, allowing more automatic rule factorizations in notations (#14070[98], by Hugo Herbelin).
- **Fixed:** Non-local custom entries survive module closing and are declared when a file is Required (#14183[99], fixes #13654[100], by Gaëtan Gilbert).
- **Fixed:** `ident` modifier in custom entry notations gave fatal errors at printing time (#14257[101], fixes #14211[102], by Hugo Herbelin).
- **Fixed:** Anomaly when overriding a notation with different applicability in `match` patterns (#14377[103], fixes #13966[104], by Hugo Herbelin).

Tactics

- **Changed:** More systematic checks that occurrences of an **`at`** clause are valid in tactics such as *`rewrite`* or *`pattern`* (#13568[105], fixes #13566[106], by Hugo Herbelin).
- **Removed:** *`fail`* and *`gfail`*, which formerly accepted negative values as a parameter, now give syntax errors for negative values (#13469[107], by Jim Fehrle).
- **Removed:** Deprecated flag `Bracketing Last Introduction Pattern` affecting the behavior of trailing disjunctive introduction patterns is definitively removed (#13509[108], by Hugo Herbelin).
- **Removed:** The `omega` tactic (deprecated in 8.12) and four `* Omega *` flags. Use `lia` instead. (#13741[109], by Jim Fehrle, who addressed the final details, building on much work by Frédéric Besson, who greatly improved *`lia`*, Maxime Dénès, Vincent Laporte and with the help of many package maintainers, among others).
- **Removed:** convert_concl_no_check. Use *`change_no_check`* instead (#13761[110], by Jim Fehrle).
- **Removed:** double induction tactic. Replace **`double induction`** ***`ident ident`*** with **`induction`** ***`ident`***`;` **`induction`** ***`ident`*** (or **`induction`** ***`ident`*** `;` **`destruct`** ***`ident`*** depending on the exact needs). Replace **`double induction`** $\mathbf{natural_1}$ $\mathbf{natural_2}$ with **`induction`** $\mathbf{natural_1}$`;` **`induction`** $\mathbf{natural_3}$ where $\mathbf{natural_3}$ is the result of $\mathbf{natural_2}$ **`-`** $\mathbf{natural_1}$ (#13762[111], by Jim Fehrle).
- **Deprecated:** In *`change`* and *`change_no_check`*, the `at ... with ...` form is deprecated. Use `with ... at ...` instead. For `at ... with ... in H |-`, use `with ... in H at ... |-`. (#13696[112], by Jim Fehrle).

[97] https://github.com/coq/coq/pull/13965
[98] https://github.com/coq/coq/pull/14070
[99] https://github.com/coq/coq/pull/14183
[100] https://github.com/coq/coq/issues/13654
[101] https://github.com/coq/coq/pull/14257
[102] https://github.com/coq/coq/issues/14211
[103] https://github.com/coq/coq/pull/14377
[104] https://github.com/coq/coq/issues/13966
[105] https://github.com/coq/coq/pull/13568
[106] https://github.com/coq/coq/issues/13566
[107] https://github.com/coq/coq/pull/13469
[108] https://github.com/coq/coq/pull/13509
[109] https://github.com/coq/coq/pull/13741
[110] https://github.com/coq/coq/pull/13761
[111] https://github.com/coq/coq/pull/13762
[112] https://github.com/coq/coq/pull/13696

- **Deprecated:** The micromega option *Simplex*, which is currently set by default (#13781[113], by Frédéric Besson).
- **Deprecated:** the undocumented `new auto` tactic (#14528[114], by Pierre-Marie Pédrot).
- **Added:** *lia* supports the boolean operator `Bool.implb` (#13715[115], by Frédéric Besson).
- **Added:** `zify` (lia/nia) support for `div`, `mod`, `pow` for `Nat` (via `ZifyNat` module) and `N` (via `ZifyN` module). The signature of `Z_div_mod_eq_full` has no assumptions. (#14037[116], fixes #11447[117], by Andrej Dudenhefner, Jason Gross, and Frédéric Besson).
- **Added:** Ltac2 now has a `unify` tactic (#14089[118], fixes #14083[119], by Samuel Gruetter).
- **Added:** *inversion_sigma* can now be applied to a specified hypothesis and additionally supports intropatterns, so it can be used much like *induction* and *inversion*. Additionally, *inversion_sigma* now supports the types **ex** (**exists x : A, P x**) and **ex2** (**exists2 x : A, P x & Q x**) in cases where the first argument **A** is a **Prop** (#14174[120], by Jason Gross).
- **Added:** `zify` (lia/nia) support for `Sint63`. (#14408[121], by Ana Borges, with help from Frédéric Besson).
- **Fixed:** Possible collision between a user-level name and an internal name when using the **%** introduction pattern (#13512[122], fixes #13413[123], by Hugo Herbelin).
- **Fixed:** *simpl* and *hnf* now reduce primitive functions on primitive integers, floats and arrays (#13699[124], fixes #13579[125], by Pierre Roux).
- **Fixed:** Setoid rewriting now remembers the (invisible) binder names of non-dependent product types. SSReflect's rewrite tactic expects these names to be retained when using `rewrite foo in H`. This also fixes SSR `rewrite foo in H *` erroneously reverting `H`. (#13882[126], fixes #12011[127], by Gaëtan Gilbert).
- **Fixed:** Properly expand projection parameters in hint discrimination nets. (#14033[128], fixes #9000[129], #14009[130], by Pierre-Marie Pédrot).
- **Fixed:** anomalies caused by empty strings in Ltac notations are now errors (#14378[131], fixes #14124[132], by Hugo Herbelin).
- **Fixed:** Print a message instead of a Diff_Failure anomaly when old and new goals can't be matched; show the goal without diff highlights (#14457[133], fixes #14425[134], by Jim Fehrle).
- **Fixed:** Anomaly of *destruct* on terms with dependent variables unused in goal (#15099[135], fixes #11504[136]

[113] https://github.com/coq/coq/pull/13781
[114] https://github.com/coq/coq/pull/14528
[115] https://github.com/coq/coq/pull/13715
[116] https://github.com/coq/coq/pull/14037
[117] https://github.com/coq/coq/issues/11447
[118] https://github.com/coq/coq/pull/14089
[119] https://github.com/coq/coq/issues/14083
[120] https://github.com/coq/coq/pull/14174
[121] https://github.com/coq/coq/pull/14408
[122] https://github.com/coq/coq/pull/13512
[123] https://github.com/coq/coq/issues/13413
[124] https://github.com/coq/coq/pull/13699
[125] https://github.com/coq/coq/issues/13579
[126] https://github.com/coq/coq/pull/13882
[127] https://github.com/coq/coq/issues/12011
[128] https://github.com/coq/coq/pull/14033
[129] https://github.com/coq/coq/issues/9000
[130] https://github.com/coq/coq/issues/14009
[131] https://github.com/coq/coq/pull/14378
[132] https://github.com/coq/coq/issues/14124
[133] https://github.com/coq/coq/pull/14457
[134] https://github.com/coq/coq/issues/14425
[135] https://github.com/coq/coq/pull/15099
[136] https://github.com/coq/coq/issues/11504

and #14090[137], by Lasse Blaauwbroek and Hugo Herbelin).

- **Fixed:** Correct convertibility of multiple terms selected by patterns in tactics such as `set` when these terms have subterms in `SProp` (#14610[138], fixes #14609[139], by Hugo Herbelin).

Tactic language

- **Changed:** Renamed Ltac2 `Bool.eq` into `Bool.equal` for uniformity. The old function is now a deprecated alias (#14128[140], by Pierre-Marie Pédrot).
- **Added:** A `printf` macro to Ltac2. It can be made accessible by importing the `Ltac2.Printf` module. See the documentation there for more information (#13236[141], fixes #10108[142], by Pierre-Marie Pédrot).
- **Added:** A function `Ltac1.lambda` allowing to embed Ltac2 functions into Ltac1 runtime values (#13442[143], fixes #12871[144], by Pierre-Marie Pédrot).
- **Added:** Ltac2 commands defining terms now accept the `deprecated` attribute (#13774[145], fixes #12317[146], by Pierre-Marie Pédrot).
- **Added:** Allow the presence of type casts for function return values, let bindings and global definitions in Ltac2 (#13914[147], by Pierre-Marie Pédrot).
- **Added:** The Ltac2 API `Ltac2.Ind` for manipulating inductive types (#13920[148], fixes #10095[149], by Pierre-Marie Pédrot).
- **Added:** Allow scope delimiters in Ltac2 `open_constr:(...)` quotation (#13939[150], fixes #12806[151], by Pierre-Marie Pédrot).
- **Added:** A FFI to convert between Ltac1 and Ltac2 identifiers (#13997[152], fixes #13996[153], by Pierre-Marie Pédrot).
- **Added:** Lazy evaluating boolean operators `lazy_and`, `lazy_or`, `lazy_impl` and infix notations `&&` and `||` to the Ltac2 `Bool.v` library l. (#14081[154], fixes #13964[155], by Michael Soegtrop).
- **Fixed:** Ltac2 notations now correctly take into account their assigned level (#14094[156], fixes #11866[157], by Pierre-Marie Pédrot).

[137] https://github.com/coq/coq/issues/14090
[138] https://github.com/coq/coq/pull/14610
[139] https://github.com/coq/coq/issues/14609
[140] https://github.com/coq/coq/pull/14128
[141] https://github.com/coq/coq/pull/13236
[142] https://github.com/coq/coq/issues/10108
[143] https://github.com/coq/coq/pull/13442
[144] https://github.com/coq/coq/issues/12871
[145] https://github.com/coq/coq/pull/13774
[146] https://github.com/coq/coq/issues/12317
[147] https://github.com/coq/coq/pull/13914
[148] https://github.com/coq/coq/pull/13920
[149] https://github.com/coq/coq/issues/10095
[150] https://github.com/coq/coq/pull/13939
[151] https://github.com/coq/coq/issues/12806
[152] https://github.com/coq/coq/pull/13997
[153] https://github.com/coq/coq/issues/13996
[154] https://github.com/coq/coq/pull/14081
[155] https://github.com/coq/coq/issues/13964
[156] https://github.com/coq/coq/pull/14094
[157] https://github.com/coq/coq/issues/11866

SSReflect

- **Added:** A test that the notations `{in _, _}` and `{pred _}` from `ssrbool.v` are displayed correctly. (#13473[158], by Cyril Cohen).
- **Added:** Lemmas about interaction between **`{in _, _}`**, **`{on _, _}`**, and **`sig`** have been backported from Mathematical Components 1.12.0 (#13490[159], by Kazuhiko Sakaguchi).

Commands and options

- **Changed:** *`Hint Rewrite`* now supports locality attributes (including *`export`*) like other *Hint* commands (#13725[160], fixes #13724[161], by Gaëtan Gilbert).
- **Changed:** In *`Record`*, alpha-rename the variable associated with the record to avoid alpha-renaming parameters of projections (#13852[162], fixes #13727[163], by Li-yao Xia).
- **Changed:** Improve the *`Coercion`* command to reduce the number of ambiguous paths to report. A pair of multiple inheritance paths that can be reduced to smaller adjoining pairs will not be reported as ambiguous paths anymore. (#13909[164], by Kazuhiko Sakaguchi).
- **Changed:** The printing order of *`Print Classes`* and *`Print Graph`*, due to the changes for the internal tables of coercion classes and coercion paths. (#13912[165], by Kazuhiko Sakaguchi).
- **Removed:** The Hide Obligations flag, deprecated in 8.12 (#13758[166], by Jim Fehrle).
- **Removed:** SearchHead command. Use the `headconcl:` clause of *`Search`* instead (#13763[167], by Jim Fehrle).
- **Removed:** `Show Zify Spec`, `Add InjTyp` and 11 similar `Add *` commands. For `Show Zify Spec`, use `Show Zify UnOpSpec` or `Show Zify BinOpSpec` instead. For `Add *`, `Use Add Zify *` intead of `Add *` (#13764[168], by Jim Fehrle).
- **Deprecated:** Like hints, typeclass instances added outside of sections without an explicit locality now generate a deprecation warning. See *Hint* (#14208[169], fixes #13562[170], by Pierre-Marie Pédrot).
- **Deprecated:** the *`Regular Subst Tactic`* flag (#14336[171], by Pierre-Marie Pédrot).
- **Added:** *`Debug`* to control debug messages, functioning similarly to the warning system (#13202[172], by Maxime Dénès and Gaëtan Gilbert). The following flags have been converted (such that `Set Flag` becomes `Set Debug "flag"`):
  - `Debug Unification` to `unification`
  - `Debug HO Unification` to `ho-unification`
  - `Debug Tactic Unification` to `tactic-unification`
  - `Congruence Verbose` to `congruence`

[158] https://github.com/coq/coq/pull/13473
[159] https://github.com/coq/coq/pull/13490
[160] https://github.com/coq/coq/pull/13725
[161] https://github.com/coq/coq/issues/13724
[162] https://github.com/coq/coq/pull/13852
[163] https://github.com/coq/coq/issues/13727
[164] https://github.com/coq/coq/pull/13909
[165] https://github.com/coq/coq/pull/13912
[166] https://github.com/coq/coq/pull/13758
[167] https://github.com/coq/coq/pull/13763
[168] https://github.com/coq/coq/pull/13764
[169] https://github.com/coq/coq/pull/14208
[170] https://github.com/coq/coq/issues/13562
[171] https://github.com/coq/coq/pull/14336
[172] https://github.com/coq/coq/pull/13202

  - `Debug Cbv` to `cbv`
  - `Debug RAKAM` to `RAKAM`
  - `Debug Ssreflect` to `ssreflect`

- **Added:** The Ltac2 grammar can now be printed using the Print Grammar ltac2 command (#14093[173], fixes #14092[174], by Pierre-Marie Pédrot).
- **Added:** *`Instance`* now accepts the *`export`* locality attribute (#14148[175], by Pierre-Marie Pédrot).
- **Fixed:** extraction failure of a parameterized type in `Prop` exported in an module interface as an assumption in `Type` (#14102[176], fixes #14100[177], by Hugo Herbelin).
- **Fixed:** Print Assumptions now treats delayed opaque proofs generated by vos compilation as if they were axioms (#14382[178], fixes #13589[179], by Pierre-Marie Pédrot).
- **Fixed:** Incorrect de Bruijn index handling in vernac class declaration, preventing users from marking existing instances of existing classes which are primitive projections (#14664[180], fixes #14652[181], by Ali Caglayan and Hugo Herbelin).

Command-line tools

- **Changed:** `coqc` now enforces that at most a single `.v` file can be passed in the command line. Support for multiple `.v` files in the form of `coqc f1.v f2.v` didn't properly work in 8.13, tho it was accepted. (#13876[182], by Emilio Jesus Gallego Arias).
- **Changed:** `coqdep` now reports an error if files specified on the command line don't exist or if it encounters unreadable files. Unknown options now generate a warning. Previously these conditions were ignored. (#14024[183], fixes #14023[184], by Hendrik Tews).
- **Changed:** Makefiles produced by `coq_makefile` now use `.DELETE_ON_ERROR` (#14238[185], by Gaëtan Gilbert).
- **Removed:** Previously deprecated command line options `-sprop-cumulative` and `-input-state` and its alias `-is` (#13822[186], by Gaëtan Gilbert).
- **Added:** `coq_makefile`-made `Makefile`s now support inclusion of a `.local-late` file at the end, allowing the user to access more variables (#12411[187], fixes #10912[188], by Jason Gross).
- **Fixed:** Failure of extraction in the presence of inductive types with local definitions in parameters (#13624[189], fixes #13581[190], by Hugo Herbelin).

[173] https://github.com/coq/coq/pull/14093
[174] https://github.com/coq/coq/issues/14092
[175] https://github.com/coq/coq/pull/14148
[176] https://github.com/coq/coq/pull/14102
[177] https://github.com/coq/coq/issues/14100
[178] https://github.com/coq/coq/pull/14382
[179] https://github.com/coq/coq/issues/13589
[180] https://github.com/coq/coq/pull/14664
[181] https://github.com/coq/coq/issues/14652
[182] https://github.com/coq/coq/pull/13876
[183] https://github.com/coq/coq/pull/14024
[184] https://github.com/coq/coq/issues/14023
[185] https://github.com/coq/coq/pull/14238
[186] https://github.com/coq/coq/pull/13822
[187] https://github.com/coq/coq/pull/12411
[188] https://github.com/coq/coq/issues/10912
[189] https://github.com/coq/coq/pull/13624
[190] https://github.com/coq/coq/issues/13581

- **Fixed:** File name was missing in coqdoc error position reporting (#14285[191], fixes #14283[192], by Arthur Charguéraud and Hugo Herbelin).

Native Compilation

- **Changed:** `coq_makefile` now uses the `coqnative` binary to generate native compilation files. Project files also understand directly the `-native-compiler` flag without having to wrap it with `-arg` (#14265[193], by Pierre-Marie Pédrot).
- **Deprecated:** the `-native-compiler` option for coqc. It is now recommended to use the *Split compilation of native computation files* binary instead to generate native compilation files ahead of time (#14309[194], by Pierre-Marie Pédrot).
- **Added:** A standalone `coqnative` binary that performs native compilation out of `vo` files, allowing to split library compilation from native compilation. See *Split compilation of native computation files*. The hybrid build system was adapted to perform a split compilation on the stdlib (#13287[195], by Pierre-Marie Pédrot).

CoqIDE

- **Added:** Ltac debugger support in CoqIDE (see `Ltac Debug`). Debugger output and prompts appear in the Messages panel (#13783[196], by Jim Fehrle and Emilio J. Gallego Arias).
- **Added:** Shift-return in the Find dialog now searches backwards (#13810[197], by slrnsc).

Standard library

- **Changed:** Minor Changes to `Rpower`: Generalizes `exp_ineq1` to hold for all non-zero numbers. Adds `exp_ineq1_le`, which holds for all reals (but is a <= instead of a <). (#13582[198], by Avi Shinnar and Barry Trager, with help from Laurent Théry).
- **Changed:** set `n mod 0 = n` uniformly for `nat`, `N`, `Z`, `int63`, `sint63`, `int31` such that `m = (m / n) * n + (m mod n)` holds (also for `n = 0`)

  > **Warning:** code that relies on `n mod 0 = 0` will break; for compatibility with both `n mod 0 = n` and `n mod 0 = 0` you can use `n mod 0 = ltac:(match eval hnf in (1 mod 0) with |0 => exact 0 |_ => exact n end)`

  (#14086[199], by Andrej Dudenhefner with help of Guillaume Melquiond, Jason Gross, and Kazuhiko Sakaguchi).

[191] https://github.com/coq/coq/pull/14285
[192] https://github.com/coq/coq/issues/14283
[193] https://github.com/coq/coq/pull/14265
[194] https://github.com/coq/coq/pull/14309
[195] https://github.com/coq/coq/pull/13287
[196] https://github.com/coq/coq/pull/13783
[197] https://github.com/coq/coq/pull/13810
[198] https://github.com/coq/coq/pull/13582
[199] https://github.com/coq/coq/pull/14086

- **Changed:** The standard library now contains a more complete theory of equality on types of the form `exists x : A, P x` and `exists2 x : A, P x & Q x` when we have `A : Prop`. To bring this theory more in line with the existing theory about sigma types, `eq_ex_uncurried`, `eq_ex2_uncurried`, `eq_ex`, `eq_ex2`, `eq_ex_hprop`, `eq_ex2_hprop` have been renamed into `eq_ex_intro_uncurried`, `eq_ex_intro2_uncurried`, `eq_ex_intro`, `eq_ex_intro2`, `eq_ex_intro_hprop`, `eq_ex_intro2_hprop` respectively and the implicit status of these lemmas has changed slightly (#14174[200], by Jason Gross).
- **Changed** Moved 39 lemmas and notations about the rationals `Q` from the constructive reals private file `theories/Reals/Cauchy/QExtra.v` to appropriate files in `theories/QArith`. The now public lemmas are mostly about compatibility of multiplication and power with relational operators and simple convenience lemmas e.g. for reduction of `Q` values. The following moved lemmas have been renamed: `Q_factorDenom` to `Qmult_frac_l`, `Q_reduce_fl` to `Qreduce_num_l`, `Qle_neq` to `Qlt_leneq`, `Qmult_lt_le_compat_nonneg` to `Qmult_le_lt_compat_pos`, `Qpower_pos_lt` to `Qpower_0_lt`, `Qpower_lt_1_increasing` to `Qpower_1_lt_pos`, `Qpower_lt_1_increasing'` to `Qpower_1_lt`, `Qpower_le_1_increasing` to `Qpower_1_le_pos`, `Qpower_le_1_increasing'` to `Qpower_1_le`, `Qzero_eq` to `Qreduce_zero`, `Qpower_lt_compat` to `Qpower_lt_compat_l`, `Qpower_le_compat` to `Qpower_le_compat_l`, `Qpower_lt_compat_inv` to `Qpower_lt_compat_l_inv`, `Qpower_le_compat_inv` to `Qpower_le_compat_l_inv`, `Qpower_decomp'` to `Qpower_decomp_pos` and `QarchimedeanExp2_Pos` to `Qarchimedean_power2_pos`. The following lemmas have been renamed and the sides of the equality swapped: `Qinv_swap_pos` to `Qinv_pos`, `Qinv_swap_neg` to `Qinv_neg` and. The following lemmas have been deleted: `Q_factorNum_l` and `Q_factorNum`. The lemma `Qopp_lt_compat` has been moved from `theories/QArith/Qround.v` to `theories/QArith/QArith_base.v`. About 10 additional lemmas have been added for similar cases as the moved lemmas. Compatibility notations are not provided because QExtra is considered internal (excluded from the library documentation). (#14293[201], by Michael Soegtrop).
- **Changed:** Importing `ZArith` no longer has the side-effect of closing `Z_scope`. (#14343[202], fixes #13307[203], by Ralf Jung).
- **Removed:** `IF_then_else` definition and corresponding `IF P then Q else R` notation (#13871[204], by Yishuai Li).
- **Removed:** from `List.v` deprecated/unexpected dependencies `Setoid`, `Le`, `Gt`, `Minus`, `Lt` (#13986[205], by Andrej Dudenhefner).
- **Deprecated:** Unsigned primitive integers are now named `uint63` instead of `int63`. The `Int63` module is replaced by `Uint63`. The full list of changes is described in the PR. (#13895[206], by Ana Borges).
- **Added:** `leb` and `ltb` functions for `ascii` (#13080[207], by Yishuai Li).
- **Added:** Library for signed primitive integers, Sint63. The following operations were added to the kernel: division, remainder, comparison functions, and arithmetic shift right. Everything else works the same for signed and unsigned ints. (#13559[208], fixes #12109[209], by Ana Borges, Guillaume Melquiond and Pierre Roux).
- **Added:** Lemmas about vectors related with `to_list`: `length_to_list`, `of_list_to_list_opp`, `to_list_nil`, `to_list_cons`, `to_list_hd`, `to_list_last`, `to_list_const`, `to_list_nth_order`, `to_list_tl`, `to_list_append`, `to_list_rev_append_tail`,

[200] https://github.com/coq/coq/pull/14174
[201] https://github.com/coq/coq/pull/14293
[202] https://github.com/coq/coq/pull/14343
[203] https://github.com/coq/coq/issues/13307
[204] https://github.com/coq/coq/pull/13871
[205] https://github.com/coq/coq/pull/13986
[206] https://github.com/coq/coq/pull/13895
[207] https://github.com/coq/coq/pull/13080
[208] https://github.com/coq/coq/pull/13559
[209] https://github.com/coq/coq/issues/12109

to_list_rev_append, to_list_rev, to_list_map, to_list_fold_left, to_list_fold_right, to_list_Forall, to_list_Exists, to_list_In, to_list_Forall2 (#13671[210], by Olivier Laurent).

- **Added:** Lemmas about count_occ: count_occ_app, count_occ_elt_eq, count_occ_elt_neq, count_occ_bound, count_occ_repeat_eq, count_occ_repeat_neq, count_occ_unique, count_occ_repeat_excl, count_occ_sgt, Permutation_count_occ (#13804[211], by Olivier Laurent with help of Jean-Christophe Léchenet).
- **Added:** Lemmas to List: Exists_map, Exists_concat, Exists_flat_map, Forall_map, Forall_concat, Forall_flat_map, nth_error_map, nth_repeat, nth_error_repeat (#13955[212], by Andrej Dudenhefner, with help from Olivier Laurent).
- **Added:** Cantor.v containing the Cantor pairing function and its inverse. Cantor.to_nat : nat * nat -> nat and Cantor.of_nat : nat -> nat * nat are the respective bijections between nat * nat and nat. (#14008[213], by Andrej Dudenhefner).
- **Added:** Lemmas to Q: Qeq_from_parts, Qden_cancel, Qnum_cancel, Qreduce_l, Qreduce_r, Qmult_inject_Z_l, Qmult_inject_Z_r QArith_base Reduction of rationals; establishing equality for Qden/Qnum separately. (#14087[214], by Karolin Varner).
- **Added:** Coq.Structures.OrdersEx.String_as_OT and Coq.Structures.OrdersEx.Ascii_as_OT to make strings and ascii ordered types (using lexical order). (#14096[215], by Jason Gross).
- **Added:** Lemmas app_eq_app, Forall_nil_iff, Forall_cons_iff to List.v (#14153[216], closes #1803[217], by Andrej Dudenhefner, with help from Olivier Laurent).
- **Added:** Z, positive and N constants can now be printed in hexadecimal by opening hex_Z_scope, hex_positive_scope, and hex_N_scope respectively (#14263[218], by Jason Gross).
- **Added:** Absolute value function for Sint63. (#14384[219], by Ana Borges).
- **Added:** Lemmas showing firstn and skipn commute with map. (#14406[220], by Rudy Peterson).
- **Fixed:** Haskell extraction is now compatible with GHC versions >= 9.0. Some #if statements have been added to extract unsafeCoerce to its new location in newer versions of GHC. (#14345[221], fixes #14256[222], by Jason Gross).

[210] https://github.com/coq/coq/pull/13671
[211] https://github.com/coq/coq/pull/13804
[212] https://github.com/coq/coq/pull/13955
[213] https://github.com/coq/coq/pull/14008
[214] https://github.com/coq/coq/pull/14087
[215] https://github.com/coq/coq/pull/14096
[216] https://github.com/coq/coq/pull/14153
[217] https://github.com/coq/coq/issues/1803
[218] https://github.com/coq/coq/pull/14263
[219] https://github.com/coq/coq/pull/14384
[220] https://github.com/coq/coq/pull/14406
[221] https://github.com/coq/coq/pull/14345
[222] https://github.com/coq/coq/issues/14256

Infrastructure and dependencies

- **Changed:** Coq's configure script now requires absolute paths for the `-prefix` option. (#12567[223], by Emilio Jesus Gallego Arias).
- **Changed:** The regular Coq package has been split in two: coq-core, with OCaml-based libraries and tools; and coq-stdlib, which contains the Gallina-based standard library. The package Coq now depends on both for compatiblity. (#12567[224], by Emilio Jesus Gallego Arias, review by Vincent Laporte, Guillaume Melquiond, Enrico Tassi, and Théo Zimmerman).
- **Changed:** Coq's OCaml parts and tools [coq-core] are now built using Dune. The main user-facing change is that Dune >= 2.5 is now required to build Coq, most recent Dune is usually recommended. For developers and plugin authors, see the entry in `dev/doc/changes.md`. (#13617[225], by Emilio Jesús Gallego Arias, Rudi Grinberg, and Théo Zimmerman; review and testing by Gaëtan Gilbert, Guillaume Melquiond, and Enrico Tassi)
- **Changed:** Undocumented variables `OLDROOT` and `COQPREFIXINSTALL` which added a prefix path to `make install` have been removed. Now, `make install` does support the more standard `DESTDIR` variable, akin to what `coq_makefile` does. (#14258[226], by Emilio Jesus Gallego Arias).
- **Added:** Support OCaml 4.12 (#13885[227], by Emilio Jesus Gallego Arias, review by Gaëtan Gilbert and Théo Zimmerman).

Miscellaneous

- **Changed:** The representation of micromega caches was slightly altered for efficiency purposes. As a consequence all stale caches must be cleaned up (#13405[228], by Pierre-Marie Pédrot).
- **Fixed:** Fix the timeout facility on Unix to allow for nested timeouts. Previous behavior on nested timeouts was that an "inner" timeout would replace an "outer" timeout, so that the outer timeout would no longer fire. With the new behavior, Unix and Windows implementations should be (approximately) equivalent. (#13586[229], by Lasse Blaauwbroek).

Changes in 8.14.1

Kernel

- **Fixed:** Fix the implementation of persistent arrays used by the VM and native compute so that it uses a uniform representation. Previously, storing primitive floats inside primitive arrays could cause memory corruption (#15081[230], closes #15070[231], by Pierre-Marie Pédrot).

[223] https://github.com/coq/coq/pull/12567
[224] https://github.com/coq/coq/pull/12567
[225] https://github.com/coq/coq/pull/13617
[226] https://github.com/coq/coq/pull/14258
[227] https://github.com/coq/coq/pull/13885
[228] https://github.com/coq/coq/pull/13405
[229] https://github.com/coq/coq/pull/13586
[230] https://github.com/coq/coq/pull/15081
[231] https://github.com/coq/coq/issues/15070

Specification language, type inference

- **Fixed:** Missing registration of universe constraints in *Module Type* elaboration (#14666[232], fixes #14505[233], by Hugo Herbelin).

Tactics

- **Fixed:** *abstract* more robust with respect to Ltac `constr` bindings containing existential variables (#14671[234], fixes #10796[235], by Hugo Herbelin).
- **Fixed:** correct support of trailing **`let`** by tactic *specialize* (#15046[236], fixes #15043[237], by Hugo Herbelin).

Commands and options

- **Fixed:** anomaly with *Extraction Conservative Types* when extracting pattern-matching on singleton types (#14669[238], fixes #3527[239], by Hugo Herbelin).
- **Fixed:** a regular error instead of an anomaly when calling *Separate Extraction* in a module (#14670[240], fixes #10796[241], by Hugo Herbelin).

Version 8.13

Summary of changes

Coq version 8.13 integrates many usability improvements, as well as extensions of the core language. The main changes include:

- *Introduction* of *primitive persistent arrays* in the core language, implemented using imperative persistent arrays.
- Introduction of *definitional proof irrelevance* for the equality type defined in the SProp sort.
- Cumulative record and inductive type declarations can now *specify* the variance of their universes.
- Various bugfixes and uniformization of behavior with respect to the use of implicit arguments and the handling of existential variables in declarations, unification and tactics.
- New warning for *unused variables* in catch-all match branches that match multiple distinct patterns.
- New *warning* for `Hint` commands outside sections without a locality attribute, whose goal is to eventually remove the fragile default behavior of importing hints only when using `Require`. The recommended fix is to declare hints as `export`, instead of the current default `global`, meaning that they are imported through `Require Import` only, not `Require`. See the following rationale and guidelines[242] for details.
- General support for *boolean attributes*.

[232] https://github.com/coq/coq/pull/14666
[233] https://github.com/coq/coq/issues/14505
[234] https://github.com/coq/coq/pull/14671
[235] https://github.com/coq/coq/issues/10796
[236] https://github.com/coq/coq/pull/15046
[237] https://github.com/coq/coq/issues/15043
[238] https://github.com/coq/coq/pull/14669
[239] https://github.com/coq/coq/issues/3527
[240] https://github.com/coq/coq/pull/14670
[241] https://github.com/coq/coq/issues/10796
[242] https://coq.discourse.group/t/change-of-default-locality-for-hint-commands-in-coq-8-13/1140

- Many improvements to the handling of *notations*, including number notations, recursive notations and notations with bindings. A new algorithm chooses the most precise notation available to print an expression, which might introduce changes in printing behavior.
- Tactic *improvements* in `lia` and its `zify` preprocessing step, now supporting reasoning on boolean operators such as `Z.leb` and supporting primitive integers `Int63`.
- Typing flags can now be specified *per-constant / inductive*.
- Improvements to the reference manual including updated syntax descriptions that match Coq's grammar in several chapters, and splitting parts of the tactics chapter to independent sections.

See the *Changes in 8.13+beta1* section and following sections for the detailed list of changes, including potentially breaking changes marked with **Changed**.

Coq's documentation is available at https://coq.github.io/doc/v8.13/refman (reference manual), and https://coq.github.io/doc/v8.13/stdlib (documentation of the standard library). Developer documentation of the ML API is available at https://coq.github.io/doc/v8.13/api.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop and Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure.

Erik Martin-Dorel has maintained the Coq Docker images[243] that are used in many Coq projects for continuous integration.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Karl Palmskog, Matthieu Sozeau and Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

Our current 32 maintainers are Yves Bertot, Frédéric Besson, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Jim Fehrle, Julien Forest, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Georges Gonthier, Benjamin Grégoire, Jason Gross, Hugo Herbelin, Vincent Laporte, Olivier Laurent, Assia Mahboubi, Kenji Maillard, Guillaume Melquiond, Pierre-Marie Pédrot, Clément Pit-Claudel, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-yao Xia and Théo Zimmermann.

The 51 contributors to this version are Reynald Affeldt, Tanaka Akira, Frédéric Besson, Lasse Blaauwbroek, Clément Blaudeau, Martin Bodin, Ali Caglayan, Tej Chajed, Cyril Cohen, Julien Coolen, Matthew Dempsky, Maxime Dénès, Andres Erbsen, Jim Fehrle, Emilio Jesús Gallego Arias, Paolo G. Giarrusso, Attila Gáspár, Gaëtan Gilbert, Jason Gross, Benjamin Grégoire, Hugo Herbelin, Wolf Honore, Jasper Hugunin, Ignat Insarov, Ralf Jung, Fabian Kunze, Vincent Laporte, Olivier Laurent, Larry D. Lee Jr, Thomas Letan, Yishuai Li, James Lottes, Jean-Christophe Léchenet, Kenji Maillard, Erik Martin-Dorel, Yusuke Matsushita, Guillaume Melquiond, Carl Patenaude-Poulin, Clément Pit-Claudel, Pierre-Marie Pédrot, Pierre Roux, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Anton Trunov, Edward Wang, Li-yao Xia, Beta Ziliani and Théo Zimmermann.

The Coq community at large helped improve the design of this new version via the GitHub issue and pull request system, the Coq development mailing list coqdev@inria.fr, the coq-club@inria.fr mailing list, the Discourse forum[244] and the Coq Zulip chat[245].

Version 8.13's development spanned 5 months from the release of Coq 8.12.0. Enrico Tassi and Maxime Dénès are the release managers of Coq 8.13. This release is the result of 400 merged PRs, closing ~100 issues.

Nantes, November 2020,
Matthieu Sozeau for the Coq development team

[243] https://hub.docker.com/r/coqorg/coq
[244] https://coq.discourse.group/
[245] http://coq.zulipchat.com

Changes in 8.13+beta1

- *Kernel*
- *Specification language, type inference*
- *Notations*
- *Tactics*
- *Tactic language*
- *SSReflect*
- *Commands and options*
- *Tools*
- *CoqIDE*
- *Standard library*
- *Infrastructure and dependencies*

Kernel

- **Added:** Definitional UIP, only when `Definitional UIP` is enabled. This models definitional uniqueness of identity proofs for the equality type in SProp. It is deactivated by default as it can lead to non-termination in combination with impredicativity. Use of this flag is also printed by `Print Assumptions`. See documentation of the flag for details. (#10390[246], by Gaëtan Gilbert).
- **Added:** Built-in support for persistent arrays, which expose a functional interface but are implemented using an imperative data structure, for better performance. (#11604[247], by Maxime Dénès and Benjamin Grégoire, with help from Gaëtan Gilbert).

  Primitive arrays are irrelevant in their single polymorphic universe (same as a polymorphic cumulative list inductive would be) (#13356[248], fixes #13354[249], by Gaëtan Gilbert).
- **Fixed:** A loss of definitional equality for declarations obtained through `Include` when entering the scope of a `Module` or `Module Type` was causing `Search` not to see the included declarations (#12537[250], fixes #12525[251] and #12647[252], by Hugo Herbelin).
- **Fixed:** Fix an incompleteness in the typechecking of `match` for cumulative inductive types. This could result in breaking subject reduction. (#13501[253], fixes #13495[254], by Matthieu Sozeau).

[246] https://github.com/coq/coq/pull/10390
[247] https://github.com/coq/coq/pull/11604
[248] https://github.com/coq/coq/pull/13356
[249] https://github.com/coq/coq/issues/13354
[250] https://github.com/coq/coq/pull/12537
[251] https://github.com/coq/coq/pull/12525
[252] https://github.com/coq/coq/pull/12647
[253] https://github.com/coq/coq/pull/13501
[254] https://github.com/coq/coq/issues/13495

Specification language, type inference

- **Changed:** *Boolean attributes* are now specified using key/value pairs, that is to say *ident*$_{attr}$ `= yes | no` ?. If the value is missing, the default is **yes**. The old syntax is still supported, but produces the `deprecated-attribute-syntax` warning.

  Deprecated attributes are *universes(monomorphic)*, *universes(notemplate)* and *universes(noncumulative)*, which are respectively replaced by *universes(polymorphic=no)*, *universes(template=no)* and *universes(cumulative=no)*. Attributes *program* and *canonical* are also affected, with the syntax *ident*$_{attr}$**(false)** being deprecated in favor of *ident*$_{attr}$**=no**. (#13312[255], by Emilio Jesus Gallego Arias).

- **Changed:** Heuristics for universe minimization to `Set`: also use constraints `Prop <= i` (#10331[256], by Gaëtan Gilbert with help from Maxime Dénès and Matthieu Sozeau, fixes #12414[257]).

- **Changed:** The type given to *Instance* is no longer automatically generalized over unbound and *generalizable* variables. Use `Instance : `{type}` instead of **Instance : *type*** to get the old behaviour, or enable the compatibility flag *Instance Generalized Output*. (#13188[258], fixes #6042[259], by Gaëtan Gilbert).

- **Changed:** Tweaked the algorithm giving default names to arguments. Should reduce the frequency that argument names get an unexpected suffix. Also makes *Mangle Names* not mess up argument names. (#12756[260], fixes #12001[261] and #6785[262], by Jasper Hugunin).

- **Removed:** Undocumented and experimental forward class hint feature `:>>`. Use `:>` (see ***of_type***) instead (#13106[263], by Pierre-Marie Pédrot).

- **Added:** Commands *Inductive*, *Record* and synonyms now support syntax `Inductive foo@{=i +j *k l}` to specify variance information for their universes (in *Cumulative* mode) (#12653[264], by Gaëtan Gilbert).

- **Added:** Warning on unused variables in pattern-matching branches of **match** serving as catch-all branches for at least two distinct patterns. (#12768[265], fixes #12762[266], by Hugo Herbelin).

- **Added:** Definition and (Co)Fixpoint now support the *using* attribute. It has the same effect as *Proof using*, which is only available in interactive mode. (#13183[267], by Enrico Tassi).

- **Added:** Typing flags can now be specified per-constant / inductive, this allows to fine-grain specify them from plugins or attributes. See *Controlling Typing Flags* for details on attribute syntax. (#12586[268], by Emilio Jesus Gallego Arias).

- **Added:** Inference of return predicate of a `match` by inversion takes sort elimination constraints into account (#13290[269], grants #13278[270], by Hugo Herbelin).

[255] https://github.com/coq/coq/pull/13312
[256] https://github.com/coq/coq/pull/10331
[257] https://github.com/coq/coq/issues/12414
[258] https://github.com/coq/coq/pull/13188
[259] https://github.com/coq/coq/issues/6042
[260] https://github.com/coq/coq/pull/12756
[261] https://github.com/coq/coq/issues/12001
[262] https://github.com/coq/coq/issues/6785
[263] https://github.com/coq/coq/pull/13106
[264] https://github.com/coq/coq/pull/12653
[265] https://github.com/coq/coq/pull/12768
[266] https://github.com/coq/coq/issues/12762
[267] https://github.com/coq/coq/pull/13183
[268] https://github.com/coq/coq/pull/12586
[269] https://github.com/coq/coq/pull/13290
[270] https://github.com/coq/coq/issues/13278

- **Fixed:** Implicit arguments taken into account in defined fields of a record type declaration (#13166[271], fixes #13165[272], by Hugo Herbelin).
- **Fixed:** Allow use of typeclass inference for the return predicate of a **`match`** (was deactivated in versions 8.10 to 8.12, #13217[273], fixes #13216[274], by Hugo Herbelin).
- **Fixed:** A case of unification raising an anomaly IllTypedInstance (#13376[275], fixes #13266[276], by Hugo Herbelin).
- **Fixed:** Using **`{wf ...}`** in local fixpoints is an error, not an anomaly (#13383[277], fixes #11816[278], by Hugo Herbelin).
- **Fixed:** Issue when two expressions involving different projections and one is primitive need to be unified (#13386[279], fixes #9971[280], by Hugo Herbelin).
- **Fixed:** A bug producing ill-typed instances of existential variables when let-ins interleaved with assumptions (#13387[281], fixes #12348[282], by Hugo Herbelin).

Notations

- **Changed:** In notations (except in custom entries), the misleading ***`syntax_modifier`*** ***`ident`*** **`ident`** (which accepted either an identifier or a _) is deprecated and should be replaced by ***`ident`*** **`name`**. If the intent was really to only parse identifiers, this will eventually become possible, but only as of Coq 8.15. In custom entries, the meaning of ***`ident`*** **`ident`** is silently changed from parsing identifiers or _ to parsing only identifiers without warning, but this presumably affects only rare, recent and relatively experimental code (#11841[283], fixes #9514[284], by Hugo Herbelin).
- **Changed:** Improved support for notations/abbreviations with mixed terms and patterns (such as the forcing modality) (#12099[285], by Hugo Herbelin).
- **Changed** Rational and real constants are parsed differently. The exponent is now encoded separately from the fractional part using `Z.pow_pos`. This way, parsing large exponents can no longer blow up and constants are printed in a form closer to the one in which they were parsed (i.e., `102e-2` is reprinted as such and not `1.02`). (#12218[286], by Pierre Roux).
- **Changed:** Scope information is propagated in indirect applications to a reference prefixed with `@`; this covers for instance the case `r.(@p) t` where scope information from `p` is now taken into account for interpreting `t` (#12685[287], by Hugo Herbelin).
- **Changed:** New model for `only parsing` and `only printing` notations with support for at most one parsing-and-printing or only-parsing notation per notation and scope, but an arbitrary number of only-printing

[271] https://github.com/coq/coq/pull/13166
[272] https://github.com/coq/coq/issues/13165
[273] https://github.com/coq/coq/pull/13217
[274] https://github.com/coq/coq/issues/13216
[275] https://github.com/coq/coq/pull/13376
[276] https://github.com/coq/coq/issues/13266
[277] https://github.com/coq/coq/pull/13383
[278] https://github.com/coq/coq/issues/11816
[279] https://github.com/coq/coq/pull/13386
[280] https://github.com/coq/coq/issues/9971
[281] https://github.com/coq/coq/pull/13387
[282] https://github.com/coq/coq/issues/13387
[283] https://github.com/coq/coq/pull/11841
[284] https://github.com/coq/coq/pull/9514
[285] https://github.com/coq/coq/pull/12099
[286] https://github.com/coq/coq/pull/12218
[287] https://github.com/coq/coq/pull/12685

notations (#12950[288], fixes #4738[289] and #9682[290] and part 2 of #12908[291], by Hugo Herbelin).

- **Changed:** Redeclaring a notation also reactivates its printing rule; in particular a second *Import* of the same module reactivates the printing rules declared in this module. In theory, this leads to changes in behavior for printing. However, this is mitigated in general by the adoption in #12986[292] of a priority given to notations which match a larger part of the term to print (#12984[293], fixes #7443[294] and #10824[295], by Hugo Herbelin).
- **Changed:** Use of notations for printing now gives preference to notations which match a larger part of the term to abbreviate (#12986[296], by Hugo Herbelin).
- **Removed** OCaml parser and printer for real constants have been removed. Real constants are now handled with proven Coq code. (#12218[297], by Pierre Roux).
- **Deprecated** `Numeral.v` is deprecated, please use `Number.v` instead. (#12218[298], by Pierre Roux).
- **Deprecated:** `Numeral Notation`, please use *Number Notation* instead (#12979[299], by Pierre Roux).
- **Added:** *Printing Float* flag to print primitive floats as hexadecimal instead of decimal values. This is included in the *Printing All* flag (#11986[300], by Pierre Roux).
- **Added:** *Number Notation* and *String Notation* commands now support parameterized inductive and non inductive types (#12218[301], fixes #12035[302], by Pierre Roux, review by Jason Gross and Jim Fehrle for the reference manual).
- **Added:** Added support for encoding notations of the form `x □ y □ .. □ z □ t`. This feature is considered experimental. (#12765[303], by Hugo Herbelin).
- **Added:** The ***binder*** entry of *Notation* can now be used in notations expecting a single (non-recursive) binder (#13265[304], by Hugo Herbelin, see section *Notations and binders* of the reference manual).
- **Fixed:** Issues in the presence of notations recursively referring to another applicative notations, such as missing scope propagation, or failure to use a notation for printing (#12960[305], fixes #9403[306] and #10803[307], by Hugo Herbelin).
- **Fixed:** Capture the names of global references by binders in the presence of notations for binders (#12965[308], fixes #9569[309], by Hugo Herbelin).
- **Fixed:** Preventing notations for constructors to involve binders (#13092[310], fixes #13078[311], by Hugo Herbelin).

[288] https://github.com/coq/coq/pull/12950
[289] https://github.com/coq/coq/issues/4738
[290] https://github.com/coq/coq/issues/9682
[291] https://github.com/coq/coq/issues/12908
[292] https://github.com/coq/coq/pull/12986
[293] https://github.com/coq/coq/pull/12984
[294] https://github.com/coq/coq/issues/7443
[295] https://github.com/coq/coq/issues/10824
[296] https://github.com/coq/coq/pull/12986
[297] https://github.com/coq/coq/pull/12218
[298] https://github.com/coq/coq/pull/12218
[299] https://github.com/coq/coq/pull/12979
[300] https://github.com/coq/coq/pull/11986
[301] https://github.com/coq/coq/pull/12218
[302] https://github.com/coq/coq/issues/12035
[303] https://github.com/coq/coq/pull/12765
[304] https://github.com/coq/coq/pull/13265
[305] https://github.com/coq/coq/pull/12960
[306] https://github.com/coq/coq/issues/9403
[307] https://github.com/coq/coq/issues/10803
[308] https://github.com/coq/coq/pull/12965
[309] https://github.com/coq/coq/issues/9569
[310] https://github.com/coq/coq/pull/13092
[311] https://github.com/coq/coq/issues/13078

- **Fixed:** Notations understand universe names without getting confused by different imported modules between declaration and use locations (#13415[312], fixes #13303[313], by Gaëtan Gilbert).

Tactics

- **Changed:** In *refine*, new existential variables unified with existing ones are no longer considered as fresh. The behavior of *simple refine* no longer depends on the orientation of evar-evar unification problems, and new existential variables are always turned into (unshelved) goals. This can break compatibility in some cases (#7825[314], by Matthieu Sozeau, with help from Maxime Dénès, review by Pierre-Marie Pédrot and Enrico Tassi, fixes #4095[315] and #4413[316]).
- **Changed:** Giving an empty list of occurrences after **in** in tactics is no longer permitted. Omitting the **in** gives the same behavior (#13237[317], fixes #13235[318], by Hugo Herbelin).
- **Removed:** **at** ***occs_nums*** clauses in tactics such as *unfold* no longer allow negative values. A "-" before the list (for set complement) is still supported. Ex: "at -1 -2" is no longer supported but "at -1 2" is. (#13403[319], by Jim Fehrle).
- **Removed:** A number of tactics that formerly accepted negative numbers as parameters now give syntax errors for negative values. These include {e}constructor, do, timeout, 9 {e}auto tactics and psatz*. (#13417[320], by Jim Fehrle).
- **Removed:** The deprecated and undocumented `prolog` tactic was removed (#12399[321], by Pierre-Marie Pédrot).
- **Removed:** `info` tactic that was deprecated in 8.5. (#12423[322], by Jim Fehrle).
- **Deprecated:** Undocumented **eauto** ***nat_or_var nat_or_var*** syntax in favor of new *bfs eauto*. Also deprecated 2-integer syntax for *debug eauto* and *info_eauto*. (Use *bfs eauto* with the *Info Eauto* or *Debug Eauto* flags instead.) (#13381[323], by Jim Fehrle).
- **Added:** *lia* is extended to deal with boolean operators e.g. `andb` or `Z.leb`. (As `lia` gets more powerful, this may break proof scripts relying on `lia` failure.) (#11906[324], by Frédéric Besson).
- **Added:** *apply … in* supports several hypotheses (#12246[325], by Hugo Herbelin; grants #9816[326]).
- **Added:** The *zify* tactic can now be extended by redefining the `zify_pre_hook` tactic. (#12552[327], by Kazuhiko Sakaguchi).
- **Added:** The *zify* tactic provides support for primitive integers (module `ZifyInt63`). (#12648[328], by Frédéric Besson).

[312] https://github.com/coq/coq/pull/13415
[313] https://github.com/coq/coq/issues/13303
[314] https://github.com/coq/coq/pull/7825
[315] https://github.com/coq/coq/issues/4095
[316] https://github.com/coq/coq/issues/4413
[317] https://github.com/coq/coq/pull/13236
[318] https://github.com/coq/coq/issues/13235
[319] https://github.com/coq/coq/pull/13403
[320] https://github.com/coq/coq/pull/13417
[321] https://github.com/coq/coq/pull/12399
[322] https://github.com/coq/coq/pull/12423
[323] https://github.com/coq/coq/pull/13381
[324] https://github.com/coq/coq/pull/11906
[325] https://github.com/coq/coq/pull/12246
[326] https://github.com/coq/coq/pull/9816
[327] https://github.com/coq/coq/pull/12552
[328] https://github.com/coq/coq/pull/12648

- **Fixed:** Avoid exposing an internal name of the form **_tmp** when applying the **_** introduction pattern which would break a dependency (#13337[329], fixes #13336[330], by Hugo Herbelin).
- **Fixed:** The case of tactics, such as *eapply*, producing existential variables under binders with an ill-formed instance (#13373[331], fixes #13363[332], by Hugo Herbelin).

Tactic language

- **Added:** An if-then-else syntax to Ltac2 (#13232[333], fixes #10110[334], by Pierre-Marie Pédrot).
- **Fixed:** Printing of the quotation qualifiers when printing `Ltac` functions (#13028[335], fixes #9716[336] and #13004[337], by Hugo Herbelin).

SSReflect

- **Added:** SSReflect intro pattern ltac views `/[dup]`, `/[swap]` and `/[apply]` (#13317[338], by Cyril Cohen).
- **Fixed:** Working around a bug of interaction between + and /(ltac:(...)) cf #13458[339] (#13459[340], by Cyril Cohen).

Commands and options

- **Changed:** Drop prefixes from grammar non-terminal names, e.g. "constr:global" -> "global", "Prim.name" -> "name". Visible in the output of *Print Grammar* and *Print Custom Grammar*. (#13096[341], by Jim Fehrle).
- **Changed:** When declaring arbitrary terms as hints, unsolved evars are not abstracted implicitly anymore and instead raise an error (#13139[342], by Pierre-Marie Pédrot).
- **Removed:** In the *Extraction Language* command, remove `Ocaml` as a valid value. Use `OCaml` instead. This was deprecated in Coq 8.8, #6261[343] (#13016[344], by Jim Fehrle).
- **Deprecated:** Hint locality currently defaults to *local* in a section and *global* otherwise, but this will change in a future release. Hints added outside of sections without an explicit locality now generate a deprecation warning. We recommend using *export* where possible (#13384[345], by Pierre-Marie Pédrot).
- **Deprecated:** *Grab Existential Variables* and *Existential* commands (#12516[346], by Maxime Dénès).

[329] https://github.com/coq/coq/pull/13337
[330] https://github.com/coq/coq/issues/13336
[331] https://github.com/coq/coq/pull/13373
[332] https://github.com/coq/coq/issues/13363
[333] https://github.com/coq/coq/pull/13232
[334] https://github.com/coq/coq/issues/10110
[335] https://github.com/coq/coq/pull/13028
[336] https://github.com/coq/coq/issues/9716
[337] https://github.com/coq/coq/issues/13004
[338] https://github.com/coq/coq/pull/13317
[339] https://github.com/coq/coq/issues/13458
[340] https://github.com/coq/coq/pull/13459
[341] https://github.com/coq/coq/pull/13096
[342] https://github.com/coq/coq/pull/13139
[343] https://github.com/coq/coq/pull/6261
[344] https://github.com/coq/coq/pull/13016
[345] https://github.com/coq/coq/pull/13384
[346] https://github.com/coq/coq/pull/12516

- **Added:** The *export* locality can now be used for all Hint commands, including *Hint Cut*, *Hint Mode*, *Hint Transparent* / *Opaque* and *Remove Hints* (#13388[347], by Pierre-Marie Pédrot).
- **Added:** Support for automatic insertion of coercions in *Search* patterns. Additionally, head patterns are now automatically interpreted as types (#13255[348], fixes #13244[349], by Hugo Herbelin).
- **Added:** The *Proof using* command can now be used without loading the Ltac plugin (`-noinit` mode) (#13339[350], by Théo Zimmermann).
- **Added:** Clarify in the documentation that *Add ML Path* is not exported to compiled files (#13345[351], fixes #13344[352], by Hugo Herbelin).

Tools

- **Changed:** Option `-native-compiler` of the configure script now impacts the default value of the `-native-compiler` option of coqc. The `-native-compiler` option of the configure script supports a new `ondemand` value, which becomes the default, thus preserving the previous default behavior. The stdlib is still precompiled when configuring with `-native-compiler yes`. It is not precompiled otherwise. This an implementation of point 2 of CEP #48[353] (#13352[354], by Pierre Roux).
- **Changed:** Added the ability for coq_makefile to directly set the installation folders, through the `COQLIBINSTALL` and `COQDOCINSTALL` variables. See *CoqMakefile.local*. (#12389[355], by Martin Bodin, review of Enrico Tassi).
- **Removed:** The option `-I` of coqchk was removed (it was deprecated in Coq 8.8) (#12613[356], by Gaëtan Gilbert).
- **Fixed:** `coqchk` no longer reports names from inner modules of opaque modules as axioms (#12862[357], fixes #12845[358], by Jason Gross).

CoqIDE

- **Added:** Support showing diffs for *Show Proof* in CoqIDE from the **View** menu. See *"Show Proof" differences*. (#12874[359], by Jim Fehrle and Enrico Tassi)
- **Added:** Support for flag *Printing Goal Names* in View menu (#13145[360], by Hugo Herbelin).

[347] https://github.com/coq/coq/pull/13388
[348] https://github.com/coq/coq/pull/13255
[349] https://github.com/coq/coq/issues/13244
[350] https://github.com/coq/coq/pull/13339
[351] https://github.com/coq/coq/pull/13345
[352] https://github.com/coq/coq/issues/13344
[353] https://github.com/coq/ceps/pull/48
[354] https://github.com/coq/coq/pull/13352
[355] https://github.com/coq/coq/pull/12389
[356] https://github.com/coq/coq/pull/12613
[357] https://github.com/coq/coq/pull/12862
[358] https://github.com/coq/coq/issues/12845
[359] https://github.com/coq/coq/pull/12874
[360] https://github.com/coq/coq/pull/13145

Standard library

- **Changed:** In the reals theory changed the epsilon in the definition of the modulus of convergence for CReal from 1/n (n in positive) to 2^z (z in Z) so that a precision coarser than one is possible. Also added an upper bound to CReal to enable more efficient computations. (#12186[361], by Michael Soegtrop).
- **Changed:** Int63 notations now match up with the rest of the standard library: `a \% m`, `m == n`, `m < n`, `m <= n`, and `m ≤ n` have been replaced with `a mod m`, `m =? n`, `m <? n`, `m <=? n`, and `m ≤? n`. The old notations are still available as deprecated notations. Additionally, there is now a `Coq.Numbers.Cyclic.Int63.Int63.Int63Notations` module that users can import to get the `Int63` notations without unqualifying the various primitives (#12479[362], fixes #12454[363], by Jason Gross).
- **Changed:** PrimFloat notations now match up with the rest of the standard library: `m == n`, `m < n`, and `m <= n` have been replaced with `m =? n`, `m <? n`, and `m <=? n`. The old notations are still available as deprecated notations. Additionally, there is now a `Coq.Floats.PrimFloat.PrimFloatNotations` module that users can import to get the `PrimFloat` notations without unqualifying the various primitives (#12556[364], fixes #12454[365], by Jason Gross).
- **Changed:** the sort of cyclic numbers from Type to Set. For backward compatibility, a dynamic sort was defined in the 3 packages bignums, coqprime and color. See for example commit 6f62bda in bignums. (#12801[366], by Vincent Semeria).
- **Changed:** `Require Import Coq.nsatz.NsatzTactic` now allows using *`nsatz`* with `Z` and `Q` without having to supply instances or using `Require Import Coq.nsatz.Nsatz`, which transitively requires unneeded files declaring axioms used in the reals (#12861[367], fixes #12860[368], by Jason Gross).
- **Deprecated:** `prod_curry` and `prod_uncurry`, in favor of `uncurry` and `curry` (#12716[369], by Yishuai Li).
- **Added:** New lemmas about `repeat` in `List` and `Permutation`: `repeat_app`, `repeat_eq_app`, `repeat_eq_cons`, `repeat_eq_elt`, `Forall_eq_repeat`, `Permutation_repeat` (#12799[370], by Olivier Laurent).
- **Added:** Extend some list lemmas to both directions: `app_inj_tail_iff`, `app_inv_head_iff`, `app_inv_tail_iff`. (#12094[371], fixes #12093[372], by Edward Wang).
- **Added:** `Decidable` instance for negation (#12420[373], by Yishuai Li).
- **Fixed:** `Coq.Program.Wf.Fix_F_inv` and `Coq.Program.Wf.Fix_eq` are now axiom-free. They no longer assume proof irrelevance. (#13365[374], by Li-yao Xia).

[361] https://github.com/coq/coq/pull/12186
[362] https://github.com/coq/coq/pull/12479
[363] https://github.com/coq/coq/issues/12454
[364] https://github.com/coq/coq/pull/12556
[365] https://github.com/coq/coq/issues/12454
[366] https://github.com/coq/coq/pull/12801
[367] https://github.com/coq/coq/pull/12861
[368] https://github.com/coq/coq/issues/12860
[369] https://github.com/coq/coq/pull/12716
[370] https://github.com/coq/coq/pull/12799
[371] https://github.com/coq/coq/pull/12094
[372] https://github.com/coq/coq/issues/12093
[373] https://github.com/coq/coq/pull/12420
[374] https://github.com/coq/coq/pull/13365

Infrastructure and dependencies

- **Changed:** When compiled with OCaml >= 4.10.0, Coq will use the new best-fit GC policy, which should provide some performance benefits. Coq's policy is optimized for speed, but could increase memory consumption in some cases. You are welcome to tune it using the `OCAMLRUNPARAM` variable and report back on good settings so we can improve the defaults. (#13040[375], fixes #11277[376], by Emilio Jesus Gallego Arias).
- **Changed:** Coq now uses the zarith[377] library, based on GNU's gmp instead of `num` which is deprecated upstream. The custom `bigint` module is no longer provided. (#11742[378], #13007[379], by Emilio Jesus Gallego Arias and Vicent Laporte, with help from Frédéric Besson).

Changes in 8.13.0

Commands and options

- **Changed:** The warning `custom-entry-overriden` has been renamed to `custom-entry-overridden` (with two d's). (#13556[380], by Simon Friis Vindum).

Changes in 8.13.1

Kernel

- **Fixed:** Fix arities of VM opcodes for some floating-point operations that could cause memory corruption (#13867[381], by Guillaume Melquiond).

CoqIDE

- **Added:** Option `-v` and `--version` to CoqIDE (#13870[382], by Guillaume Melquiond).

Changes in 8.13.2

Kernel

- **Fixed:** Crash when using *`vm_compute`* on an irreducible `PArray.set` (#14005[383], fixes #13998[384], by Guillaume Melquiond).
- **Fixed:** Never store persistent arrays as VM / native structured values. This could be used to make vo marshalling crash, and probably breaking some other invariants of the kernel (#14007[385], fixes #14006[386], by Pierre-Marie Pédrot).

[375] https://github.com/coq/coq/pull/13040
[376] https://github.com/coq/coq/issues/11277
[377] https://github.com/ocaml/Zarith
[378] https://github.com/coq/coq/pull/11742
[379] https://github.com/coq/coq/pull/13007
[380] https://github.com/coq/coq/pull/13556
[381] https://github.com/coq/coq/pull/13867
[382] https://github.com/coq/coq/pull/13870
[383] https://github.com/coq/coq/pull/14005
[384] https://github.com/coq/coq/issues/13998
[385] https://github.com/coq/coq/pull/14007
[386] https://github.com/coq/coq/issues/14006

Tactic language

- **Fixed:** Ltac2 `Array.init` no longer incurs exponential overhead when used recursively (#14012[387], fixes #14011[388], by Jason Gross).

Version 8.12

Summary of changes

Coq version 8.12 integrates many usability improvements, in particular with respect to notations, scopes and implicit arguments, along with many bug fixes and major improvements to the reference manual. The main changes include:

- New *binder notation* for non-maximal implicit arguments using `[ ]` allowing to set and see the implicit status of arguments immediately.
- New notation `Inductive I A | x : s := ...` to distinguish the *uniform* from the non-uniform parameters in inductive definitions.
- More robust and expressive treatment of *implicit inductive* parameters in inductive declarations.
- Improvements in the treatment of implicit arguments and partially applied constants in *notations*, parsing of hexadecimal number notation and better handling of scopes and coercions for printing.
- A correct and efficient *coercion coherence* checking algorithm, avoiding spurious or duplicate warnings.
- An improved *`Search` command* which accepts complex queries. Note that this takes precedence over the now deprecated *ssreflect search*.
- Many additions and improvements of the *standard library*.
- Improvements to the *reference manual* include a more logical organization of chapters along with updated syntax descriptions that match Coq's grammar in most but not all chapters.

Additionally, the `omega` tactic is deprecated in this version of Coq, and we recommend users to switch to *`lia`* in new proof scripts.

See the *Changes in 8.12+beta1* section and following sections for the detailed list of changes, including potentially breaking changes marked with **Changed**.

Coq's documentation is available at https://coq.github.io/doc/v8.12/refman (reference manual), and https://coq.github.io/doc/v8.12/stdlib (documentation of the standard library). Developer documentation of the ML API is available at https://coq.github.io/doc/v8.12/api.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop and Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure.

Erik Martin-Dorel has maintained the Coq Docker images[389] that are used in many Coq projects for continuous integration.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Karl Palmskog, Matthieu Sozeau and Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

Previously, most components of Coq had a single principal maintainer. This was changed in 8.12 (#11295[390]) so that every component now has a team of maintainers, who are in charge of reviewing and merging incoming pull requests. This gave us a chance to significantly expand the pool of maintainters and provide faster feedback to contributors. Special thanks to all our maintainers!

[387] https://github.com/coq/coq/pull/14012
[388] https://github.com/coq/coq/issues/14011
[389] https://hub.docker.com/r/coqorg/coq
[390] https://github.com/coq/coq/pull/11295

Our current 31 maintainers are Yves Bertot, Frédéric Besson, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Jim Fehrle, Julien Forest, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Georges Gonthier, Benjamin Grégoire, Jason Gross, Hugo Herbelin, Vincent Laporte, Assia Mahboubi, Kenji Maillard, Guillaume Melquiond, Pierre-Marie Pédrot, Clément Pit-Claudel, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-yao Xia, Théo Zimmermann

The 59 contributors to this version are Abhishek Anand, Yves Bertot, Frédéric Besson, Lasse Blaauwbroek, Simon Boulier, Quentin Carbonneaux, Tej Chajed, Arthur Charguéraud, Cyril Cohen, Pierre Courtieu, Matthew Dempsky, Maxime Dénès, Andres Erbsen, Erika (@rrika), Nikita Eshkeev, Jim Fehrle, @formalize, Emilio Jesús Gallego Arias, Paolo G. Giarrusso, Gaëtan Gilbert, Jason Gross, Samuel Gruetter, Attila Gáspár, Hugo Herbelin, Jan-Oliver Kaiser, Robbert Krebbers, Vincent Laporte, Olivier Laurent, Xavier Leroy, Thomas Letan, Yishuai Li, Kenji Maillard, Erik Martin-Dorel, Guillaume Melquiond, Ike Mulder, Guillaume Munch-Maccagnoni, Antonio Nikishaev, Karl Palmskog, Pierre-Marie Pédrot, Clément Pit-Claudel, Ramkumar Ramachandra, Lars Rasmusson, Daniel de Rauglaudre, Talia Ringer, Pierre Roux, Kazuhiko Sakaguchi, Vincent Semeria, @scinart, Kartik Singhal, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Ralf Treinen, Anton Trunov, Bernhard M. Wiedemann, Li-yao Xia, Nickolai Zeldovich and Théo Zimmermann.

Many power users helped to improve the design of this new version via the GitHub issue and pull request system, the Coq development mailing list coqdev@inria.fr, the coq-club@inria.fr mailing list, the Discourse forum[391] and the new Coq Zulip chat[392] (thanks to Cyril Cohen for organizing the move from Gitter).

Version 8.12's development spanned 6 months from the release of Coq 8.11.0. Emilio Jesus Gallego Arias and Théo Zimmermann are the release managers of Coq 8.12. This release is the result of ~500 PRs merged, closing ~100 issues.

Nantes, June 2020,
Matthieu Sozeau for the Coq development team

Changes in 8.12+beta1

- *Kernel*
- *Specification language, type inference*
- *Notations*
- *Tactics*
- *Tactic language*
- *SSReflect*
- *Flags, options and attributes*
- *Commands*
- *Tools*
- *CoqIDE*
- *Standard library*
- *Reals library*
- *Extraction*

[391] https://coq.discourse.group/
[392] http://coq.zulipchat.com

- *Reference manual*
- *Infrastructure and dependencies*

Kernel

- **Fixed:** Specification of **`PrimFloat.leb`** which made **`(x <= y)%float`** true for any non NaN **`x`** and **`y`**. (#12484[393], fixes #12483[394], by Pierre Roux).

Specification language, type inference

- **Changed:** The deprecation warning raised since Coq 8.10 when a trailing implicit is declared to be non-maximally inserted (with the command *`Arguments`*) has been turned into an error (#11368[395], by SimonBoulier).
- **Changed:** Typeclass resolution, accessible through *`typeclasses eauto`*, now suspends constraints according to their modes instead of failing. If a typeclass constraint does not match any of the declared modes for its class, the constraint is postponed, and the proof search continues on other goals. Proof search does a fixed point computation to try to solve them at a later stage of resolution. It does not fail if there remain only stuck constraints at the end of resolution. This makes typeclasses with declared modes more robust with respect to the order of resolution. (#10858[396], fixes #9058[397], by Matthieu Sozeau).
- **Added:** Warn when manual implicit arguments are used in unexpected positions of a term (e.g. in `Check id (forall {x}, x)`) or when an implicit argument name is shadowed (e.g. in `Check fun f : forall {x:nat} {x}, nat => f`) (#10202[398], by Hugo Herbelin).
- **Added:** *`Arguments`* now supports setting implicit an anonymous argument, as e.g. in `Arguments id {A} {_}` (#11098[399], by Hugo Herbelin, fixes #4696[400], #5173[401], #9098[402]).
- **Added:** Syntax for non-maximal implicit arguments in definitions and terms using square brackets. The syntax is `[x : A]`, `[x]`, `` `[A] `` to be consistent with the command *`Arguments`* (#11235[403], by Simon Boulier).
- **Added:** *`Implicit Types`* are now taken into account for printing. To inhibit it, unset the *`Printing Use Implicit Types`* flag (#11261[404], by Hugo Herbelin, granting #10366[405]).
- **Added:** New syntax *`Inductive`* ***`ident`*** ***`binder`***`*` **`|`** ***`binder`***`*` **`:= ...`** to specify which parameters of an inductive type are uniform. See *Parameterized inductive types* (#11600[406], by Gaëtan Gilbert).
- **Added:** Warn when using *`Fixpoint`* or *`CoFixpoint`* for definitions which are not recursive (#12121[407], by Hugo Herbelin).

[393] https://github.com/coq/coq/pull/12484
[394] https://github.com/coq/coq/issues/12483
[395] https://github.com/coq/coq/pull/11368
[396] https://github.com/coq/coq/pull/10858
[397] https://github.com/coq/coq/issues/9058
[398] https://github.com/coq/coq/pull/10202
[399] https://github.com/coq/coq/pull/11098
[400] https://github.com/coq/coq/pull/4696
[401] https://github.com/coq/coq/pull/5173
[402] https://github.com/coq/coq/pull/9098
[403] https://github.com/coq/coq/pull/11235
[404] https://github.com/coq/coq/pull/11261
[405] https://github.com/coq/coq/pull/10366
[406] https://github.com/coq/coq/pull/11600
[407] https://github.com/coq/coq/pull/12121

- **Fixed:** More robust and expressive treatment of implicit inductive parameters in inductive declarations (#11579[408], by Maxime Dénès, Gaëtan Gilbert and Jasper Hugunin; fixes #7253[409] and #11585[410]).
- **Fixed:** Anomaly which could be raised when printing binders with implicit types (#12323[411], by Hugo Herbelin; fixes #12322[412]).
- **Fixed:** Case of an anomaly in trying to infer the return clause of an ill-typed `match` (#12422[413], fixes #12418[414], by Hugo Herbelin).

Notations

- **Changed:** Notation scopes are now always inherited in notations binding a partially applied constant, including for notations binding an expression of the form **`@qualid`**. The latter was not the case beforehand (part of #11120[415]).
- **Changed:** The printing algorithm now interleaves search for notations and removal of coercions (#11172[416], by Hugo Herbelin).
- **Changed:** Nicer printing for decimal constants in R and Q. 1.5 is now printed 1.5 rather than 15e-1 (#11848[417], by Pierre Roux).
- **Removed:** deprecated `compat` modifier of *`Notation`* and *`Infix`* commands. Use the *`deprecated`* attribute instead (#11113[418], by Théo Zimmermann, with help from Jason Gross).
- **Deprecated:** Numeral Notation on `Decimal.uint`, `Decimal.int` and `Decimal.decimal` are replaced respectively by numeral notations on `Numeral.uint`, `Numeral.int` and `Numeral.numeral` (#11948[419], by Pierre Roux).
- **Added:** Notations declared with the `where` clause in the declaration of inductive types, coinductive types, record fields, fixpoints and cofixpoints now support the `only parsing` modifier (#11602[420], by Hugo Herbelin).
- **Added:** *`Printing Parentheses`* flag to print parentheses even when implied by associativity or precedence (#11650[421], by Hugo Herbelin and Abhishek Anand).
- **Added:** Numeral notations now parse hexadecimal constants such as `0x2a` or `0xb.2ap-2`. Parsers added for `nat`, `positive`, `Z`, `N`, `Q`, `R`, primitive integers and primitive floats (#11948[422], by Pierre Roux).
- **Added:** Abbreviations support arguments occurring both in term and binder position (#8808[423], by Hugo Herbelin).
- **Fixed:** Different interpretations in different scopes of the same notation string can now be associated with different printing formats (#10832[424], by Hugo Herbelin, fixes #6092[425] and #7766[426]).
- **Fixed:** Parsing and printing consistently handle inheritance of implicit arguments in notations. With the exception of notations of the form **`Notation`** ***`string`*** **`:= @`*****`qualid`*** and **`Notation`** ***`ident`*** **`:= @`*****`qualid`*** which

[408] https://github.com/coq/coq/pull/11579
[409] https://github.com/coq/coq/pull/7253
[410] https://github.com/coq/coq/pull/11585
[411] https://github.com/coq/coq/pull/12323
[412] https://github.com/coq/coq/pull/12322
[413] https://github.com/coq/coq/pull/12422
[414] https://github.com/coq/coq/pull/12418
[415] https://github.com/coq/coq/pull/11120
[416] https://github.com/coq/coq/pull/11172
[417] https://github.com/coq/coq/pull/11848
[418] https://github.com/coq/coq/pull/11113
[419] https://github.com/coq/coq/pull/11948
[420] https://github.com/coq/coq/pull/11602
[421] https://github.com/coq/coq/pull/11650
[422] https://github.com/coq/coq/pull/11948
[423] https://github.com/coq/coq/pull/8808
[424] https://github.com/coq/coq/pull/10832
[425] https://github.com/coq/coq/issues/6092
[426] https://github.com/coq/coq/issues/7766

inhibit implicit arguments, all notations binding a partially applied constant, as e.g. in **Notation** *string* **:= (***qualid* *arg*+**)**, or **Notation** *string* **:= (@***qualid* *arg*+**)**, or **Notation** *ident* **:= (***qualid* *arg*+**)**, or **Notation** *ident* **:= (@***qualid* *arg*+**)**, inherit the remaining implicit arguments (#11120[427], by Hugo Herbelin, fixing #4690[428] and #11091[429]).

- **Fixed:** Notations in `only printing` mode do not uselessly reserve parsing keywords (#11590[430], by Hugo Herbelin, fixes #9741[431]).
- **Fixed:** Numeral Notations now play better with multiple scopes for the same inductive type. Previously, when multiple numeral notations were defined for the same inductive, only the last one was considered for printing. Now, among the notations that are usable for printing and either have a scope delimiter or are open, the selection is made according to the order of open scopes, or according to the last defined notation if no appropriate scope is open (#12163[432], fixes #12159[433], by Pierre Roux, review by Hugo Herbelin and Jason Gross).

Tactics

- **Changed:** The *rapply* tactic in `Coq.Program.Tactics` now handles arbitrary numbers of underscores and takes in a `uconstr`. In rare cases where users were relying on *rapply* inserting exactly 15 underscores and no more, due to the lemma having a completely unspecified codomain (and thus allowing for any number of underscores), the tactic will now loop instead (#10760[434], by Jason Gross).
- **Changed:** The `auto with zarith` tactic and variations (including *intuition*) may now call *lia* instead of `omega` (when the `Omega` module is loaded); more goals may be automatically solved, fewer section variables will be captured spuriously (#11018[435], by Vincent Laporte).
- **Changed:** The new *NativeCompute Timing* flag causes calls to *native_compute* (as well as kernel calls to the native compiler) to emit separate timing information about conversion to native code, compilation, execution, and reification. It replaces the timing information previously emitted when the `-debug` command-line flag was set, and allows more fine-grained timing of the native compiler (#11025[436], by Jason Gross). Additionally, the timing information now uses real time rather than user time (fixes #11962[437], #11963[438], by Jason Gross)
- **Changed:** Improve the efficiency of `PreOmega.elim_let` using an iterator implemented in OCaml (#11370[439], by Frédéric Besson).
- **Changed:** Improve the efficiency of *zify* by rewritting the remaining Ltac code in OCaml (#11429[440], by Frédéric Besson).
- **Changed:** Backtrace information for tactics has been improved (#11755[441], by Emilio Jesus Gallego Arias).
- **Changed:** The default tactic used by `firstorder` is `auto with core` instead of `auto with *`; see *Solvers for logic and equality* for details; old behavior can be reset by using the `-compat 8.12` command-line

[427] https://github.com/coq/coq/pull/11120
[428] https://github.com/coq/coq/pull/4690
[429] https://github.com/coq/coq/pull/11091
[430] https://github.com/coq/coq/pull/11590
[431] https://github.com/coq/coq/pull/9741
[432] https://github.com/coq/coq/pull/12163
[433] https://github.com/coq/coq/pull/12159
[434] https://github.com/coq/coq/pull/10760
[435] https://github.com/coq/coq/pull/11018
[436] https://github.com/coq/coq/pull/11025
[437] https://github.com/coq/coq/issues/11962
[438] https://github.com/coq/coq/pull/11963
[439] https://github.com/coq/coq/pull/11370
[440] https://github.com/coq/coq/pull/11429
[441] https://github.com/coq/coq/pull/11755

flag; to ease the migration of legacy code, the default solver can be set to `debug auto with *` with `Set Firstorder Solver debug auto with *` (#11760[442], by Vincent Laporte).

- **Changed:** *autounfold* no longer fails when the *Opaque* command is used on constants in the hint databases (#11883[443], by Attila Gáspár).
- **Changed:** Tactics with qualified name of the form `Coq.Init.Notations` are now qualified with prefix `Coq.Init.Ltac`; users of the `-noinit` option should now import `Coq.Init.Ltac` if they want to use Ltac (#12023[444], by Hugo Herbelin; minor source of incompatibilities).
- **Changed:** Tactic *subst* ***ident*** now fails over a section variable which is indirectly dependent in the goal; the incompatibility can generally be fixed by first clearing the hypotheses causing an indirect dependency, as reported by the error message, or by using *rewrite* `...` **`in *`** instead; similarly, *subst* has no more effect on such variables (#12146[445], by Hugo Herbelin; fixes #10812[446] and #12139[447]).
- **Changed:** The check that *unfold* arguments were indeed unfoldable has been moved to runtime (#12256[448], by Pierre-Marie Pédrot; fixes #5764[449], #5159[450], #4925[451] and #11727[452]).
- **Changed** When the tactic *functional induction* $\mathtt{c_1\ c_2\ \ldots\ c_n}$ is used with no parenthesis around $\mathtt{c_1\ c_2\ \ldots\ c_n}$, $\mathtt{c_1\ c_2\ \ldots\ c_n}$ is now read as one single applicative term. In particular implicit arguments should be omitted. Rare source of incompatibility (#12326[453], by Pierre Courtieu).
- **Changed:** When using *exists* or *eexists* with multiple arguments, the evaluation of arguments and applications of constructors are now interleaved. This improves unification in some cases (#12366[454], fixes #12365[455], by Attila Gáspár).
- **Removed:** Undocumented `omega with`. Using *lia* is the recommended replacement, although the old semantics of `omega with *` can also be recovered with `zify; omega` (#11288[456], by Emilio Jesus Gallego Arias).
- **Removed:** Deprecated syntax `_eqn` for *destruct* and *remember*. Use `eqn:` syntax instead (#11877[457], by Hugo Herbelin).
- **Removed:** `at` clauses can no longer be used with *autounfold*. Since they had no effect, it is safe to remove them (#11883[458], by Attila Gáspár).
- **Deprecated:** The `omega` tactic is deprecated; use *lia* from the *Micromega* plugin instead (#11976[459], by Vincent Laporte).
- **Added:** The *zify* tactic is now aware of `Pos.pred_double`, `Pos.pred_N`, `Pos.of_nat`, `Pos.add_carry`, `Pos.pow`, `Pos.square`, `Z.pow`, `Z.double`, `Z.pred_double`, `Z.succ_double`, `Z.square`, `Z.div2`, and `Z.quot2`. Injections for internal definitions in module `ZifyBool` (`isZero` and

[442] https://github.com/coq/coq/pull/11760
[443] https://github.com/coq/coq/pull/11883
[444] https://github.com/coq/coq/pull/12023
[445] https://github.com/coq/coq/pull/12146
[446] https://github.com/coq/coq/pull/10812
[447] https://github.com/coq/coq/pull/12139
[448] https://github.com/coq/coq/pull/12256
[449] https://github.com/coq/coq/issues/5764
[450] https://github.com/coq/coq/issues/5159
[451] https://github.com/coq/coq/issues/4925
[452] https://github.com/coq/coq/issues/11727
[453] https://github.com/coq/coq/pull/12326
[454] https://github.com/coq/coq/pull/12366
[455] https://github.com/coq/coq/issues/12365
[456] https://github.com/coq/coq/pull/11288
[457] https://github.com/coq/coq/pull/11877
[458] https://github.com/coq/coq/pull/11883
[459] https://github.com/coq/coq/pull/11976

isLeZero) are also added to help users to declare new *zify* class instances using Micromega tactics (#10998[460], by Kazuhiko Sakaguchi).

- **Added:** *Show Lia Profile* prints some statistics about *lia* calls (#11474[461], by Frédéric Besson).
- **Added:** Syntax *pose proof* **(*ident*:=*term*)** as an alternative to *pose proof* ***term*** **as** ***ident***, following the model of *pose* **(*ident*:=*term*)** (#11522[462], by Hugo Herbelin).
- **Added:** New tactical *with_strategy* which behaves like the command *Strategy*, with effects local to the given tactic (#12129[463], by Jason Gross).
- **Added:** The *zify* tactic is now aware of `Nat.le`, `Nat.lt` and `Nat.eq` (#12213[464], by Frédéric Besson; fixes #12210[465]).
- **Fixed:** *zify* now handles `Z.pow_pos` by default. In Coq 8.11, this was the case only when loading module `ZifyPow` because this triggered a regression of *lia*. The regression is now fixed, and the module kept only for compatibility (#11362[466], fixes #11191[467], by Frédéric Besson).
- **Fixed:** Efficiency regression of *lia* (#11474[468], fixes #11436[469], by Frédéric Besson).
- **Fixed:** The behavior of *autounfold* no longer depends on the names of terms and modules (#11883[470], fixes #7812[471], by Attila Gáspár).
- **Fixed:** Wrong type error in tactic *functional induction* (#12326[472], by Pierre Courtieu, fixes #11761[473], reported by Lasse Blaauwbroek).

Tactic language

- **Changed:** The "reference" tactic generic argument now accepts arbitrary variables of the goal context (#12254[474], by Pierre-Marie Pédrot).
- **Added:** An array library for Ltac2 (as compatible as possible with OCaml standard library) (#10343[475], by Michael Soegtrop).
- **Added:** The Ltac2 rebinding command *Ltac2 Set* has been extended with the ability to give a name to the old value so as to be able to reuse it inside the new one (#11503[476], by Pierre-Marie Pédrot).
- **Added:** Ltac2 notations for *enough* and *eenough* (#11740[477], by Michael Soegtrop).
- **Added:** New Ltac2 function `Fresh.Free.of_goal` to return the list of names of declarations of the current goal; new Ltac2 function `Fresh.in_goal` to return a variable fresh in the current goal (#11882[478], by Hugo Herbelin).

[460] https://github.com/coq/coq/pull/10998
[461] https://github.com/coq/coq/pull/11474
[462] https://github.com/coq/coq/pull/11522
[463] https://github.com/coq/coq/pull/12129
[464] https://github.com/coq/coq/pull/12213
[465] https://github.com/coq/coq/issues/12210
[466] https://github.com/coq/coq/pull/11362
[467] https://github.com/coq/coq/issues/11191
[468] https://github.com/coq/coq/pull/11474
[469] https://github.com/coq/coq/issues/11436
[470] https://github.com/coq/coq/pull/11883
[471] https://github.com/coq/coq/issues/7812
[472] https://github.com/coq/coq/pull/12326
[473] https://github.com/coq/coq/issues/11761
[474] https://github.com/coq/coq/pull/12254
[475] https://github.com/coq/coq/pull/10343
[476] https://github.com/coq/coq/pull/11503
[477] https://github.com/coq/coq/pull/11740
[478] https://github.com/coq/coq/pull/11882

- **Added:** Ltac2 notations for reductions in terms: **`eval` *`red_expr`* `in` *`term`*** (#11981[479], by Michael Soegtrop).
- **Fixed:** The *`Ltac Profiling`* machinery now correctly handles backtracking into multi-success tactics. The call-counts of some tactics are unfortunately inflated by 1, as some tactics are implicitly implemented as `tac + fail`, which has two entry-points rather than one (fixes #12196[480], #12197[481], by Jason Gross).

SSReflect

- **Changed:** The *`Search (ssreflect)`* command that used to be available when loading the `ssreflect` plugin has been moved to a separate plugin that needs to be loaded separately: `ssrsearch` (part of #8855[482], fixes #12253[483], by Théo Zimmermann).
- **Deprecated:** *`Search (ssreflect)`* (available through `Require ssrsearch.`) in favor of the `headconcl:` clause of *`Search`* (part of #8855[484], by Théo Zimmermann).

Flags, options and attributes

- **Changed:** *Legacy attributes* can now be passed in any order (#11665[485], by Théo Zimmermann).
- **Removed:** `Typeclasses Axioms Are Instances` flag, deprecated since 8.10. Use *`Declare Instance`* for axioms which should be instances (#11185[486], by Théo Zimmermann).
- **Removed:** Deprecated unsound compatibility `Template Check` flag that was introduced in 8.10 to help users gradually move their template polymorphic inductive type definitions outside sections (#11546[487], by Pierre-Marie Pédrot).
- **Removed:** Deprecated `Shrink Obligations` flag (#11828[488], by Emilio Jesus Gallego Arias).
- **Removed:** Unqualified `polymorphic`, `monomorphic`, `template`, `notemplate` attributes (they were deprecated since Coq 8.10). Use *`universes(polymorphic)`*, `universes(monomorphic)`, *`universes(template)`* and `universes(notemplate)` instead (#11663[489], by Théo Zimmermann).
- **Deprecated:** `Hide Obligations` flag (#11828[490], by Emilio Jesus Gallego Arias).
- **Added:** Handle the *`local`* attribute in *`Canonical Structure`* declarations (#11162[491], by Enrico Tassi).
- **Added:** New attributes supported when defining an inductive type *`universes(cumulative)`*, `universes(noncumulative)` and *`private(matching)`*, which correspond to legacy attributes `Cumulative`, `NonCumulative`, and the previously undocumented `Private` (#11665[492], by Théo Zimmermann).

[479] https://github.com/coq/coq/pull/11981
[480] https://github.com/coq/coq/issues/12196
[481] https://github.com/coq/coq/pull/12197
[482] https://github.com/coq/coq/pull/8855
[483] https://github.com/coq/coq/issues/12253
[484] https://github.com/coq/coq/pull/8855
[485] https://github.com/coq/coq/pull/11665
[486] https://github.com/coq/coq/pull/11185
[487] https://github.com/coq/coq/pull/11546
[488] https://github.com/coq/coq/pull/11828
[489] https://github.com/coq/coq/pull/11663
[490] https://github.com/coq/coq/pull/11828
[491] https://github.com/coq/coq/pull/11162
[492] https://github.com/coq/coq/pull/11665

- **Added:** The *Hint* commands now accept the `export` locality as an attribute, allowing to make import-scoped hints (#11812[493], by Pierre-Marie Pédrot).
- **Added:** `Cumulative StrictProp` to control cumulativity of SProp (#12034[494], by Gaëtan Gilbert).

Commands

- **Changed:** The `Coercion` command has been improved to check the coherence of the inheritance graph. It checks whether a circular inheritance path of `C >-> C` is convertible with the identity function or not, then report it as an ambiguous path if it is not. The new mechanism does not report ambiguous paths that are redundant with others. For example, checking the ambiguity of `[f; g]` and `[f'; g]` is redundant with that of `[f]` and `[f']` thus will not be reported (#11258[495], by Kazuhiko Sakaguchi).
- **Changed:** Several commands (`Search`, `About`, ...) now print the implicit arguments in brackets when printing types (#11795[496], by Simon Boulier).
- **Changed:** The warning when using `Require` inside a section moved from the `deprecated` category to the `fragile` category, because there is no plan to remove the functionality at this time (#11972[497], by Gaëtan Gilbert).
- **Changed:** `Redirect` now obeys the `Printing Width` and `Printing Depth` options (#12358[498], by Emilio Jesus Gallego Arias).
- **Removed:** Recursive OCaml loadpaths are not supported anymore; the command `Add Rec ML Path` has been removed; `Add ML Path` is now the preferred one. We have also dropped support for the non-qualified version of the `Add LoadPath` command, that is to say, the `Add LoadPath dir` version; now, you must always specify a prefix now using `Add Loadpath dir as Prefix` (#11618[499], by Emilio Jesus Gallego Arias).
- **Removed:** undocumented `Chapter` command. Use `Section` instead (#11746[500], by Théo Zimmermann).
- **Removed:** `SearchAbout` command that was deprecated since 8.5. Use `Search` instead (#11944[501], by Jim Fehrle).
- **Deprecated:** Declaration of arbitrary terms as hints. Global references are now preferred (#7791[502], by Pierre-Marie Pédrot).
- **Deprecated:** `SearchHead` in favor of the new `headconcl:` clause of `Search` (part of #8855[503], by Théo Zimmermann).
- **Added:** `Print Canonical Projections` can now take constants as arguments and prints only the unification rules that involve or are synthesized from the given constants (#10747[504], by Kazuhiko Sakaguchi).
- **Added:** A section variable introduced with `Let` can be declared as a `Canonical Structure` (#11164[505], by Enrico Tassi).

[493] https://github.com/coq/coq/pull/11812
[494] https://github.com/coq/coq/pull/12034
[495] https://github.com/coq/coq/pull/11258
[496] https://github.com/coq/coq/pull/11795
[497] https://github.com/coq/coq/pull/11972
[498] https://github.com/coq/coq/pull/12358
[499] https://github.com/coq/coq/pull/11618
[500] https://github.com/coq/coq/pull/11746
[501] https://github.com/coq/coq/pull/11944
[502] https://github.com/coq/coq/pull/7791
[503] https://github.com/coq/coq/pull/8855
[504] https://github.com/coq/coq/pull/10747
[505] https://github.com/coq/coq/pull/11164

- **Added:** Support for universe bindings and universe contrainsts in `Let` definitions (#11534[506], by Théo Zimmermann).
- **Added:** Support for new clauses `hyp:`, `headhyp:`, `concl:`, `headconcl:`, `head:` and `is:` in `Search`. Support for complex search queries combining disjunctions, conjunctions and negations (#8855[507], by Hugo Herbelin, with ideas from Cyril Cohen and help from Théo Zimmermann).
- **Fixed:** A printing bug in the presence of elimination principles with local definitions (#12295[508], by Hugo Herbelin; fixes #12233[509]).
- **Fixed:** Anomalies with `Show Proof` (#12296[510], by Hugo Herbelin; fixes #12234[511]).

Tools

- **Changed:** Internal options and behavior of `coqdep`. `coqdep` no longer works as a replacement for `ocamldep`, thus `.ml` files are not supported as input. Also, several deprecated options have been removed: `-w`, `-D`, `-mldep`, `-prefix`, `-slash`, and `-dumpbox`. Passing `-boot` to `coqdep` will not load any path by default now, `-R/-Q` should be used instead (#11523[512] and #11589[513], by Emilio Jesus Gallego Arias).
- **Changed:** The order in which the require flags `-ri`, `-re`, `-rfrom`, etc. and the option flags `-set`, `-unset` are given now matters. In particular, it is now possible to interleave the loading of plugins and the setting of options by choosing the right order for these flags. The load flags `-l` and `-lv` are still processed afterward for now (#11851[514] and #12097[515], by Lasse Blaauwbroek).
- **Changed:** The `cleanall` target of a makefile generated by `coq_makefile` now erases `.lia.cache` and `.nia.cache` (#12006[516], by Olivier Laurent).
- **Changed:** The output of `make TIMED=1` (and therefore the timing targets such as `print-pretty-timed` and `print-pretty-timed-diff`) now displays the full name of the output file being built, rather than the stem of the rule (which was usually the filename without the extension, but in general could be anything for user-defined rules involving `%`) (#12126[517], by Jason Gross).
- **Changed:** When passing `TIMED=1` to `make` with either Coq's own makefile or a `coq_makefile`-made makefile, timing information is now printed for OCaml files as well (#12211[518], by Jason Gross).
- **Changed:** The pretty-timed scripts and targets now print a newline at the end of their tables, rather than creating text with no trailing newline (#12368[519], by Jason Gross).
- **Removed:** The `-load-ml-source` and `-load-ml-object` command-line options have been removed; their use was very limited, you can achieve the same adding additional object files in the linking step or using a plugin (#11409[520], by Emilio Jesus Gallego Arias).
- **Removed:** The confusingly-named `-require` command-line option, which was deprecated since 8.11. Use the equivalent `-require-import` / `-ri` options instead (#12005[521], by Théo Zimmermann).

[506] https://github.com/coq/coq/pull/11534
[507] https://github.com/coq/coq/pull/8855
[508] https://github.com/coq/coq/pull/12295
[509] https://github.com/coq/coq/pull/12233
[510] https://github.com/coq/coq/pull/12296
[511] https://github.com/coq/coq/pull/12234
[512] https://github.com/coq/coq/pull/11523
[513] https://github.com/coq/coq/pull/11589
[514] https://github.com/coq/coq/pull/11851
[515] https://github.com/coq/coq/pull/12097
[516] https://github.com/coq/coq/pull/12006
[517] https://github.com/coq/coq/pull/12126
[518] https://github.com/coq/coq/pull/12211
[519] https://github.com/coq/coq/pull/12368
[520] https://github.com/coq/coq/pull/11409
[521] https://github.com/coq/coq/pull/12005

- **Deprecated:** `-cumulative-sprop` command-line flag in favor of the new *`Cumulative StrictProp`* flag (#12034[522], by Gaëtan Gilbert).
- **Added:** A new documentation environment `details` to make certain portion of a Coq document foldable. See *Hiding / Showing parts of the source* (#10592[523], by Thomas Letan).
- **Added:** The `make-both-single-timing-files.py` script now accepts a `--fuzz=N` parameter on the command line which determines how many characters two lines may be offset in the "before" and "after" timing logs while still being considered the same line. When invoking this script via the `print-pretty-single-time-diff` target in a `Makefile` made by `coq_makefile`, you can set this argument by passing `TIMING_FUZZ=N` to `make` (#11302[524], by Jason Gross).
- **Added:** The `make-one-time-file.py` and `make-both-time-files.py` scripts now accept a `--real` parameter on the command line to print real times rather than user times in the tables. The `make-both-single-timing-files.py` script accepts a `--user` parameter to use user times. When invoking these scripts via the `print-pretty-timed` or `print-pretty-timed-diff` or `print-pretty-single-time-diff` targets in a `Makefile` made by `coq_makefile`, you can set this argument by passing `TIMING_REAL=1` (to pass `--real`) or `TIMING_REAL=0` (to pass `--user`) to `make` (#11302[525], by Jason Gross).
- **Added:** Coq's build system now supports both `TIMING_FUZZ`, `TIMING_SORT_BY`, and `TIMING_REAL` just like a `Makefile` made by `coq_makefile` (#11302[526], by Jason Gross).
- **Added:** The `make-one-time-file.py` and `make-both-time-files.py` scripts now include peak memory usage information in the tables (can be turned off by the `--no-include-mem` command-line parameter), and a `--sort-by-mem` parameter to sort the tables by memory rather than time. When invoking these scripts via the `print-pretty-timed` or `print-pretty-timed-diff` targets in a `Makefile` made by `coq_makefile`, you can set this argument by passing `TIMING_INCLUDE_MEM=0` (to pass `--no-include-mem`) and `TIMING_SORT_BY_MEM=1` (to pass `--sort-by-mem`) to `make` (#11606[527], by Jason Gross).
- **Added:** Coq's build system now supports both `TIMING_INCLUDE_MEM` and `TIMING_SORT_BY_MEM` just like a `Makefile` made by `coq_makefile` (#11606[528], by Jason Gross).
- **Added:** New `coqc` / `coqtop` option `-boot` that will not bind the `Coq` library prefix by default (#11617[529], by Emilio Jesus Gallego Arias).
- **Added:** Definitions in coqdoc link to themselves, giving access in html to their own url (#12026[530], by Hugo Herbelin; granting #7093[531]).
- **Added:** Hyperlinks on bound variables in coqdoc (#12033[532], by Hugo Herbelin; it incidentally fixes #7697[533]).
- **Added:** Highlighting of link targets in coqdoc (#12091[534], by Hugo Herbelin).
- **Fixed:** The various timing targets for Coq's standard library now correctly display and label the "before" and "after" columns, rather than mixing them up (#11302[535] fixes #11301[536], by Jason Gross).

[522] https://github.com/coq/coq/pull/12034
[523] https://github.com/coq/coq/pull/10592
[524] https://github.com/coq/coq/pull/11302
[525] https://github.com/coq/coq/pull/11302
[526] https://github.com/coq/coq/pull/11302
[527] https://github.com/coq/coq/pull/11606
[528] https://github.com/coq/coq/pull/11606
[529] https://github.com/coq/coq/pull/11617
[530] https://github.com/coq/coq/pull/12026
[531] https://github.com/coq/coq/pull/7093
[532] https://github.com/coq/coq/pull/12033
[533] https://github.com/coq/coq/pull/7697
[534] https://github.com/coq/coq/pull/12091
[535] https://github.com/coq/coq/pull/11302
[536] https://github.com/coq/coq/issues/11301

- **Fixed:** The sorting order of the timing script `make-both-time-files.py` and the target `print-pretty-timed-diff` is now deterministic even when the sorting order is `absolute` or `diff`; previously the relative ordering of two files with identical times was non-deterministic (#11606[537], by Jason Gross).
- **Fixed:** Fields of a record tuple now link in coqdoc to their definition (#12027[538], fixes #3415[539], by Hugo Herbelin).
- **Fixed:** `coqdoc` now reports the location of a mismatched opening `[[` instead of throwing an uninformative exception (#12037[540], fixes #9670[541], by Xia Li-yao).
- **Fixed:** coqchk incorrectly reporting names from opaque modules as axioms (#12076[542], by Pierre Roux; fixes #5030[543]).
- **Fixed:** coq_makefile-generated `Makefile`s `pretty-timed-diff` target no longer raises Python exceptions in the rare corner case where the log of times contains no files (#12388[544], fixes #12387[545], by Jason Gross).

CoqIDE

- **Removed:** "Tactic" menu from CoqIDE which had been unmaintained for a number of years (#11414[546], by Pierre-Marie Pédrot).
- **Removed:** "Revert all buffers" command from CoqIDE which had been broken for a long time (#11415[547], by Pierre-Marie Pédrot).

Standard library

- **Changed:** Notations **`[|`*`term`*`|]`** and **`[||`*`term`*`||]`** for morphisms from 63-bit integers to `Z` and `zn2z int` have been removed in favor of **`ψ(`*`term`*`)`** and **`Φ(`*`term`*`)`** respectively. These notations were breaking Ltac parsing (#11686[548], by Maxime Dénès).
- **Changed:** The names of `Sorted_sort` and `LocallySorted_sort` in `Coq.Sorting.MergeSort` have been swapped to appropriately reflect their meanings (#11885[549], by Lysxia).
- **Changed:** Notations `<=?` and `<?` from `Coq.Structures.Orders` and `Coq.Sorting.Mergesort.NatOrder` are now at level 70 rather than 35, so as to be compatible with the notations defined everywhere else in the standard library. This may require re-parenthesizing some expressions. These notations were breaking the ability to import modules from the standard library that were otherwise compatible (fixes #11890[550], #11891[551], by Jason Gross).
- **Changed:** The level of ≡ in `Coq.Numbers.Cyclic.Int63.Int63` is now 70, no associativity, in line with =. Note that this is a minor incompatibility with developments that declare their own ≡ notation and import `Int63`

[537] https://github.com/coq/coq/pull/11606
[538] https://github.com/coq/coq/pull/12027
[539] https://github.com/coq/coq/issues/3415
[540] https://github.com/coq/coq/pull/12037
[541] https://github.com/coq/coq/issues/9670
[542] https://github.com/coq/coq/pull/12076
[543] https://github.com/coq/coq/issues/5030
[544] https://github.com/coq/coq/pull/12388
[545] https://github.com/coq/coq/pull/12387
[546] https://github.com/coq/coq/pull/11414
[547] https://github.com/coq/coq/pull/11415
[548] https://github.com/coq/coq/pull/11686
[549] https://github.com/coq/coq/pull/11885
[550] https://github.com/coq/coq/issues/11890
[551] https://github.com/coq/coq/pull/11891

(fixes #11905[552], #11909[553], by Jason Gross).

- **Changed:** No longer re-export `ListNotations` from `Program` (`Program.Syntax`) (#11992[554], by Antonio Nikishaev).
- **Changed:** It is now possible to import the `nsatz` machinery without transitively depending on the axioms of the real numbers nor of classical logic by loading `Coq.nsatz.NsatzTactic` rather than `Coq.nsatz.Nsatz`. Note that some constants have changed kernel names, living in `Coq.nsatz.NsatzTactic` rather than `Coq.nsatz.Nsatz`; this might cause minor incompatibilities that can be fixed by actually running `Import Nsatz` rather than relying on absolute names (#12073[555], by Jason Gross; fixes #5445[556]).
- **Changed:** new lemma `NoDup_incl_NoDup` in `List.v` to remove useless hypothesis `NoDup l'` in `Sorting.Permutation.NoDup_Permutation_bis` (#12120[557], by Olivier Laurent).
- **Changed:** *`Fixpoints`* of the standard library without a recursive call turned into ordinary *`Definitions`* (#12121[558], by Hugo Herbelin; fixes #11903[559]).
- **Deprecated:** `Bool.leb` in favor of `Bool.le`. The definition of `Bool.le` is made local to avoid conflicts with `Nat.le`. As a consequence, previous calls to `leb` based on importing `Bool` should now be qualified into `Bool.le` even if `Bool` is imported (#12162[560], by Olivier Laurent).
- **Added:** Theorem `bezout_comm` for natural numbers (#11127[561], by Daniel de Rauglaudre).
- **Added** `rew dependent` notations for the dependent version of `rew` in `Coq.Init.Logic.EqNotations` to improve the display and parsing of `match` statements on `Logic.eq` (#11240[562], by Jason Gross).
- **Added:** Lemmas about lists:
  - properties of `In`: `in_elt`, `in_elt_inv`
  - properties of `nth`: `app_nth2_plus`, `nth_middle`, `nth_ext`
  - properties of `last`: `last_last`, `removelast_last`
  - properties of `remove`: `remove_cons`, `remove_app`, `notin_remove`, `in_remove`, `in_in_remove`, `remove_remove_comm`, `remove_remove_eq`, `remove_length_le`, `remove_length_lt`
  - properties of `concat`: `in_concat`, `remove_concat`
  - properties of `map` and `flat_map`: `map_last`, `map_eq_cons`, `map_eq_app`, `flat_map_app`, `flat_map_ext`, `nth_nth_nth_map`
  - properties of `incl`: `incl_nil_l`, `incl_l_nil`, `incl_cons_inv`, `incl_app_app`, `incl_app_inv`, `remove_incl`, `incl_map`, `incl_filter`, `incl_Forall_in_iff`
  - properties of `NoDup` and `nodup`: `NoDup_rev`, `NoDup_filter`, `nodup_incl`
  - properties of `Exists` and `Forall`: `Exists_nth`, `Exists_app`, `Exists_rev`, `Exists_fold_right`, `incl_Exists`, `Forall_nth`, `Forall_app`, `Forall_elt`, `Forall_rev`, `Forall_fold_right`, `incl_Forall`, `map_ext_Forall`, `Exists_or`, `Exists_or_inv`, `Forall_and`, `Forall_and_inv`, `exists_Forall`, `Forall_image`, `concat_nil_Forall`, `in_flat_map_Exists`, `notin_flat_map_Forall`

[552] https://github.com/coq/coq/issues/11905
[553] https://github.com/coq/coq/pull/11909
[554] https://github.com/coq/coq/pull/11992
[555] https://github.com/coq/coq/pull/12073
[556] https://github.com/coq/coq/issues/5445
[557] https://github.com/coq/coq/pull/12119
[558] https://github.com/coq/coq/pull/12121
[559] https://github.com/coq/coq/pull/11903
[560] https://github.com/coq/coq/pull/12162
[561] https://github.com/coq/coq/pull/11127
[562] https://github.com/coq/coq/pull/11240

  - properties of `repeat`: `repeat_cons`, `repeat_to_concat`
  - definitions and properties of `list_sum` and `list_max`: `list_sum_app`, `list_max_app`, `list_max_le`, `list_max_lt`
  - misc: `elt_eq_unit`, `last_length`, `rev_eq_app`, `removelast_firstn_len`, `cons_seq`, `seq_S`

  (#11249[563], #12237[564], by Olivier Laurent).

- **Added:** Well-founded induction principles for `nat`: `lt_wf_rect1`, `lt_wf_rect`, `gt_wf_rect`, `lt_wf_double_rect` (#11335[565], by Olivier Laurent).
- **Added:** `remove'` and `count_occ'` over lists, alternatives to `remove` and `count_occ` based on `filter` (#11350[566], by Yishuai Li).
- **Added:** Facts about `N.iter` and `Pos.iter`:
  - `N.iter_swap_gen`, `N.iter_swap`, `N.iter_succ`, `N.iter_succ_r`, `N.iter_add`, `N.iter_ind`, `N.iter_invariant`;
  - `Pos.iter_succ_r`, `Pos.iter_ind`.

  (#11880[567], by Lysxia).
- **Added:** Facts about `Permutation`:
  - structure: `Permutation_refl'`, `Permutation_morph_transp`
  - compatibilities: `Permutation_app_rot`, `Permutation_app_swap_app`, `Permutation_app_middle`, `Permutation_middle2`, `Permutation_elt`, `Permutation_Forall`, `Permutation_Exists`, `Permutation_Forall2`, `Permutation_flat_map`, `Permutation_list_sum`, `Permutation_list_max`
  - inversions: `Permutation_app_inv_m`, `Permutation_vs_elt_inv`, `Permutation_vs_cons_inv`, `Permutation_vs_cons_cons_inv`, `Permutation_map_inv`, `Permutation_image`, `Permutation_elt_map_inv`
  - length-preserving definition by means of transpositions `Permutation_transp` with associated properties: `Permutation_transp_sym`, `Permutation_transp_equiv`, `Permutation_transp_cons`, `Permutation_Permutation_transp`, `Permutation_ind_transp`

  (#11946[568], by Olivier Laurent).
- **Added:** Notations for sigma types: `{ x & P & Q }`, `{ ' pat & P }`, `{ ' pat & P & Q }` (#11957[569], by Olivier Laurent).
- **Added:** Order relations `lt` and `compare` added in `Bool.Bool`. Order properties for `bool` added in `Bool.BoolOrder` as well as two modules `Bool_as_OT` and `Bool_as_DT` in `Structures.OrdersEx` (#12008[570], by Olivier Laurent).
- **Added:** Properties of some operations on vectors:
  - `nth_order`: `nth_order_hd`, `nth_order_tl`, `nth_order_ext`

[563] https://github.com/coq/coq/pull/11249
[564] https://github.com/coq/coq/pull/12237
[565] https://github.com/coq/coq/pull/11335
[566] https://github.com/coq/coq/pull/11350
[567] https://github.com/coq/coq/pull/11880
[568] https://github.com/coq/coq/pull/11946
[569] https://github.com/coq/coq/pull/11957
[570] https://github.com/coq/coq/pull/12008

  - replace: nth_order_replace_eq, nth_order_replace_neq, replace_id, replace_replace_eq, replace_replace_neq
  - map: map_id, map_map, map_ext_in, map_ext
  - Forall and Forall2: Forall_impl, Forall_forall, Forall_nth_order, Forall2_nth_order

  (#12014[571], by Olivier Laurent).

- **Added:** Lemmas orb_negb_l, andb_negb_l, implb_true_iff, implb_false_iff, implb_true_r, implb_false_r, implb_true_l, implb_false_l, implb_same, implb_contrapositive, implb_negb, implb_curry, implb_andb_distrib_r, implb_orb_distrib_r, implb_orb_distrib_l in library Bool (#12018[572], by Hugo Herbelin).
- **Added:** Definition and properties of cyclic permutations / circular shifts: CPermutation (#12031[573], by Olivier Laurent).
- **Added:** Structures.OrderedTypeEx.Ascii_as_OT (#12044[574], by formalize.eth (formalize@protonmail.com)).
- **Fixed:** Rewrote Structures.OrderedTypeEx.String_as_OT.compare to avoid huge proof terms (#12044[575], by formalize.eth (formalize@protonmail.com); fixes #12015[576]).

Reals library

- **Changed:** Cleanup of names in the Reals theory: replaced tan_is_inj with tan_inj and replaced atan_right_inv with tan_atan - compatibility notations are provided. Moved various auxiliary lemmas from Ratan.v to more appropriate places (#9803[577], by Laurent Théry and Michael Soegtrop).
- **Changed:** Replace CRzero and CRone by CR_of_Q 0 and CR_of_Q 1 in ConstructiveReals. Use implicit arguments for ConstructiveReals. Move ConstructiveReals into new directory Abstract. Remove imports of implementations inside those Abstract files. Move implementation by means of Cauchy sequences in new directory Cauchy. Split files ConstructiveMinMax and ConstructivePower.

  Warning: The constructive reals modules are marked as experimental.

  (#11725[578], #12287[579] and #12288[580], by Vincent Semeria).

- **Removed:** Type RList has been removed. All uses have been replaced by list R. Functions from RList named In, Rlength, cons_Rlist, app_Rlist have also been removed as they are essentially the same as In, length, app, and map from List, modulo the following changes:
  - RList.In x (RList.cons a l) used to be convertible to (x = a) \/ RList.In x l, but List.In x (a :: l) is convertible to (a = x) \/ List.In l. The equality is reversed.
  - app_Rlist and List.map take arguments in different order.

[571] https://github.com/coq/coq/pull/12014
[572] https://github.com/coq/coq/pull/12018
[573] https://github.com/coq/coq/pull/12031
[574] https://github.com/coq/coq/pull/12044
[575] https://github.com/coq/coq/pull/12044
[576] https://github.com/coq/coq/issues/12015
[577] https://github.com/coq/coq/pull/9803
[578] https://github.com/coq/coq/pull/11725
[579] https://github.com/coq/coq/pull/12287
[580] https://github.com/coq/coq/pull/12288

(#11404[581], by Yves Bertot).

- **Added:** inverse trigonometric functions `asin` and `acos` with lemmas for the derivatives, bounds and special values of these functions; an extensive set of identities between trigonometric functions and their inverse functions; lemmas for the injectivity of sine and cosine; lemmas on the derivative of the inverse of decreasing functions and on the derivative of horizontally mirrored functions; various generic auxiliary lemmas and definitions for `Rsqr`, `sqrt`, `posreal` and others (#9803[582], by Laurent Théry and Michael Soegtrop).

Extraction

- **Added:** Support for better extraction of strings in OCaml and Haskell: `ExtOcamlNativeString` provides bindings from the Coq `String` type to the OCaml `string` type, and string literals can be extracted to literals, both in OCaml and Haskell (#10486[583], by Xavier Leroy, with help from Maxime Dénès, review by Hugo Herbelin).
- **Fixed:** In Haskell extraction with `ExtrHaskellString`, equality comparisons on strings and characters are now guaranteed to be uniquely well-typed, even in very polymorphic contexts under `unsafeCoerce`; this is achieved by adding type annotations to the extracted code, and by making `ExtrHaskellString` export `ExtrHaskellBasic` (#12263[584], by Jason Gross, fixes #12257[585] and #12258[586]).

Reference manual

- **Changed:** The reference manual has been restructured to get a more logical organization. In the new version, there are fewer top-level chapters, and, in the HTML format, chapters are split into smaller pages. This is still a work in progress and further restructuring is expected in the next versions of Coq (CEP#43[587], implemented in #11601[588], #11871[589], #11914[590], #12148[591], #12172[592], #12239[593] and #12330[594], effort inspired by Matthieu Sozeau, led by Théo Zimmermann, with help and reviews of Jim Fehrle, Clément Pit-Claudel and others).
- **Changed:** Most of the grammar is now presented using the notation mechanism that has been used to present commands and tactics since Coq 8.8 and which is documented in *Syntax conventions* (#11183[595], #11314[596], #11423[597], #11705[598], #11718[599], #11720[600], #11961[601] and #12103[602], by Jim Fehrle, reviewed by Théo Zimmermann).
- **Added:** A glossary of terms and an index of attributes (#11869[603], #12150[604] and #12224[605], by Jim Fehrle and Théo Zimmermann, reviewed by Clément Pit-Claudel)

[581] https://github.com/coq/coq/pull/11404
[582] https://github.com/coq/coq/pull/9803
[583] https://github.com/coq/coq/pull/10486
[584] https://github.com/coq/coq/pull/12263
[585] https://github.com/coq/coq/issues/12257
[586] https://github.com/coq/coq/issues/12258
[587] https://github.com/coq/ceps/pull/43
[588] https://github.com/coq/coq/pull/11601
[589] https://github.com/coq/coq/pull/11871
[590] https://github.com/coq/coq/pull/11914
[591] https://github.com/coq/coq/pull/12148
[592] https://github.com/coq/coq/pull/12172
[593] https://github.com/coq/coq/pull/12239
[594] https://github.com/coq/coq/pull/12330
[595] https://github.com/coq/coq/pull/11183
[596] https://github.com/coq/coq/pull/11314
[597] https://github.com/coq/coq/pull/11423
[598] https://github.com/coq/coq/pull/11705
[599] https://github.com/coq/coq/pull/11718
[600] https://github.com/coq/coq/pull/11720
[601] https://github.com/coq/coq/pull/11961
[602] https://github.com/coq/coq/pull/12103
[603] https://github.com/coq/coq/pull/11869
[604] https://github.com/coq/coq/pull/12150
[605] https://github.com/coq/coq/pull/12224

- **Added:** A selector that allows switching between versions of the reference manual (#12286[606], by Clément Pit-Claudel).
- **Fixed:** Most of the documented syntax has been thoroughly updated to make it accurate and easily understood. This was done using a semi-automated `doc_grammar` tool introduced for this purpose and through significant revisions to the text (#9884[607], #10614[608], #11314[609], #11423[610], #11705[611], #11718[612], #11720[613] #11797[614], #11913[615], #11958[616], #11960[617], #11961[618] and #12103[619], by Jim Fehrle, reviewed by Théo Zimmermann and Jason Gross).

Infrastructure and dependencies

- **Changed:** Minimal versions of dependencies for building the reference manual: now requires Sphinx >= 2.3.1 & < 3.0.0, sphinx_rtd_theme 0.4.3+ and sphinxcontrib-bibtex 0.4.2+.

  Warning: The reference manual is known not to build properly with Sphinx 3.

  (#12224[620], by Jim Fehrle and Théo Zimmermann).
- **Removed:** Python 2 is no longer required in any part of the codebase (#11245[621], by Emilio Jesus Gallego Arias).

Changes in 8.12.0

Notations

- **Added:** Simultaneous definition of terms and notations now support custom entries. Fixes #11121[622]. (#12523[623], by Maxime Dénès).
- **Fixed:** Printing bug with notations for n-ary applications used with applied references. (#12683[624], fixes #12682[625], by Hugo Herbelin).

Tactics

- **Fixed:** *`typeclasses eauto`* (and discriminated hint bases) now correctly classify local variables as being unfoldable (#12572[626], fixes #12571[627], by Pierre-Marie Pédrot).

[606] https://github.com/coq/coq/pull/12286
[607] https://github.com/coq/coq/pull/9884
[608] https://github.com/coq/coq/pull/10614
[609] https://github.com/coq/coq/pull/11314
[610] https://github.com/coq/coq/pull/11423
[611] https://github.com/coq/coq/pull/11705
[612] https://github.com/coq/coq/pull/11718
[613] https://github.com/coq/coq/pull/11720
[614] https://github.com/coq/coq/pull/11797
[615] https://github.com/coq/coq/pull/11913
[616] https://github.com/coq/coq/pull/11958
[617] https://github.com/coq/coq/pull/11960
[618] https://github.com/coq/coq/pull/11961
[619] https://github.com/coq/coq/pull/12103
[620] https://github.com/coq/coq/pull/12224
[621] https://github.com/coq/coq/pull/11245
[622] https://github.com/coq/coq/pull/11121
[623] https://github.com/coq/coq/pull/11523
[624] https://github.com/coq/coq/pull/12683
[625] https://github.com/coq/coq/pull/12682
[626] https://github.com/coq/coq/pull/12572
[627] https://github.com/coq/coq/issues/12571

Tactic language

- **Fixed:** Excluding occurrences was causing an anomaly in tactics (e.g., `pattern _ at L` where `L` is `-2`). (#12541[628], fixes #12228[629], by Pierre Roux).
- **Fixed:** Parsing of multi-parameters Ltac2 types (#12594[630], fixes #12595[631], by Pierre-Marie Pédrot).

SSReflect

- **Fixed:** Do not store the full environment inside ssr ast_closure_term (#12708[632], fixes #12707[633], by Pierre-Marie Pédrot).

Commands and options

- **Fixed:** Properly report the mismatched magic number of vo files (#12677[634], fixes #12513[635], by Pierre-Marie Pédrot).
- **Changed:** Arbitrary hints have been undeprecated, and their definition now triggers a standard warning instead (#12678[636], fixes #11970[637], by Pierre-Marie Pédrot).

CoqIDE

- **Fixed:** CoqIDE no longer exits when trying to open a file whose name is not a valid identifier (#12562[638], fixes #10988[639], by Vincent Laporte).

Infrastructure and dependencies

- **Fixed:** Running `make` in `test-suite/` twice (or more) in a row will no longer rebuild the `modules/` tests on subsequent runs, if they have not been modified in the meantime (#12583[640], fixes #12582[641], by Jason Gross).

Changes in 8.12.1

Kernel

- **Fixed:** Incompleteness of conversion checking on problems involving *η-expansion* and *cumulative universe polymorphic inductive types* (#12738[642], fixes #7015[643], by Gaëtan Gilbert).
- **Fixed:** Polymorphic side-effects inside monomorphic definitions were incorrectly handled as not inlined. This allowed deriving an inconsistency (#13331[644], fixes #13330[645], by Pierre-Marie Pédrot).

Notations

[628] https://github.com/coq/coq/pull/12541
[629] https://github.com/coq/coq/issues/12228
[630] https://github.com/coq/coq/pull/12594
[631] https://github.com/coq/coq/issues/12595
[632] https://github.com/coq/coq/pull/12708
[633] https://github.com/coq/coq/issues/12707
[634] https://github.com/coq/coq/pull/12677
[635] https://github.com/coq/coq/issues/12513
[636] https://github.com/coq/coq/pull/12678
[637] https://github.com/coq/coq/issues/11970
[638] https://github.com/coq/coq/pull/12562
[639] https://github.com/coq/coq/issues/10988
[640] https://github.com/coq/coq/pull/12583
[641] https://github.com/coq/coq/issues/12582
[642] https://github.com/coq/coq/pull/12738
[643] https://github.com/coq/coq/issues/7015
[644] https://github.com/coq/coq/pull/13331
[645] https://github.com/coq/coq/issues/13330

- **Fixed:** Undetected collision between a lonely notation and a notation in scope at printing time (#12946[646], fixes the first part of #12908[647], by Hugo Herbelin).
- **Fixed:** Printing of notations in custom entries with variables not mentioning an explicit level (#13026[648], fixes #12775[649] and #13018[650], by Hugo Herbelin).

Tactics

- **Added:** *replace* and *inversion* support registration of a `core.identity`-like equality in `Type`, such as HoTT's `path` (#12847[651], partially fixes #12846[652], by Hugo Herbelin).
- **Fixed:** Anomaly with *injection* involving artificial dependencies disappearing by reduction (#12816[653], fixes #12787[654], by Hugo Herbelin).

Tactic language

- **Fixed:** Miscellaneous issues with locating tactic errors (#13247[655], fixes #12773[656] and #12992[657], by Hugo Herbelin).

SSReflect

- **Fixed:** Regression in error reporting after *case*. A generic error message "Could not fill dependent hole in apply" was reported for any error following *case* or *elim* (#12857[658], fixes #12837[659], by Enrico Tassi).

Commands and options

- **Fixed:** Failures of *Search* in the presence of primitive projections (#13301[660], fixes #13298[661], by Hugo Herbelin).
- **Fixed:** *Search* supports filtering on parts of identifiers which are not proper identifiers themselves, such as **`"1"`** (#13351[662], fixes #13349[663], by Hugo Herbelin).

Tools

- **Fixed:** Special symbols now escaped in the index produced by coqdoc, avoiding collision with the syntax of the output format (#12754[664], fixes #12752[665], by Hugo Herbelin).
- **Fixed:** The `details` environment added in the 8.12 release can now be used as advertised in the reference manual (#12772[666], by Thomas Letan).
- **Fixed:** Targets such as `print-pretty-timed` in `coq_makefile`-made `Makefiles` no longer error in rare cases where `--output-sync` is not passed to make and the timing output gets interleaved in just the wrong

[646] https://github.com/coq/coq/pull/12946
[647] https://github.com/coq/coq/issues/12908
[648] https://github.com/coq/coq/pull/13026
[649] https://github.com/coq/coq/issues/12775
[650] https://github.com/coq/coq/issues/13018
[651] https://github.com/coq/coq/pull/12847
[652] https://github.com/coq/coq/issues/12846
[653] https://github.com/coq/coq/pull/12816
[654] https://github.com/coq/coq/issues/12787
[655] https://github.com/coq/coq/pull/13247
[656] https://github.com/coq/coq/issues/12773
[657] https://github.com/coq/coq/issues/12992
[658] https://github.com/coq/coq/pull/12857
[659] https://github.com/coq/coq/issues/12837
[660] https://github.com/coq/coq/pull/13301
[661] https://github.com/coq/coq/issues/13298
[662] https://github.com/coq/coq/pull/13351
[663] https://github.com/coq/coq/issues/13349
[664] https://github.com/coq/coq/pull/12754
[665] https://github.com/coq/coq/issues/12752
[666] https://github.com/coq/coq/pull/12772

way (#13063[667], fixes #13062[668], by Jason Gross).

CoqIDE

- **Fixed:** View menu "Display parentheses" (#12794[669] and #13067[670], fixes #12793[671], by Jean-Christophe Léchenet and Hugo Herbelin).

Infrastructure and dependencies

- **Added:** Coq is now tested against OCaml 4.11.1 (#12972[672], by Emilio Jesus Gallego Arias).
- **Fixed:** The reference manual can now build with Sphinx 3 (#13011[673], fixes #12332[674], by Théo Zimmermann and Jim Fehrle).

Changes in 8.12.2

Notations

- **Fixed:** 8.12 regression causing notations mentioning a coercion to be ignored (#13436[675], fixes #13432[676], by Hugo Herbelin).

Tactics

- **Fixed:** 8.12 regression: incomplete inference of implicit arguments in `exists` (#13468[677], fixes #13456[678], by Hugo Herbelin).

Version 8.11

Summary of changes

The main changes brought by Coq version 8.11 are:

- *Ltac2*, a new tactic language for writing more robust larger scale tactics, with built-in support for datatypes and the multi-goal tactic monad.
- *Primitive floats* are integrated in terms and follow the binary64 format of the IEEE 754 standard, as specified in the `Coq.Float.Floats` library.
- *Cleanups* of the section mechanism, delayed proofs and further restrictions of template polymorphism to fix soundness issues related to universes.
- New *unsafe flags* to disable locally guard, positivity and universe checking. Reliance on these flags is always printed by `Print Assumptions`.
- *Fixed bugs* of `Export` and `Import` that can have a significant impact on user developments (**common source of incompatibility!**).

[667] https://github.com/coq/coq/pull/13063
[668] https://github.com/coq/coq/issues/13062
[669] https://github.com/coq/coq/pull/12794
[670] https://github.com/coq/coq/pull/13067
[671] https://github.com/coq/coq/issues/12793
[672] https://github.com/coq/coq/pull/12972
[673] https://github.com/coq/coq/pull/13011
[674] https://github.com/coq/coq/issues/12332
[675] https://github.com/coq/coq/pull/13436
[676] https://github.com/coq/coq/issues/13432
[677] https://github.com/coq/coq/pull/13468
[678] https://github.com/coq/coq/issues/13456

- New interactive development method based on `vos` *interface files*, allowing to work on a file without recompiling the proof parts of their dependencies.
- New `Arguments` annotation for *bidirectional type inference* configuration for reference (e.g. constants, inductive) applications.
- New *refine attribute* for `Instance` can be used instead of the removed `Refine Instance Mode`.
- Generalization of the `under` and `over` *tactics* of SSReflect to arbitrary relations.
- *Revision* of the `Coq.Reals` library, its axiomatisation and instances of the constructive and classical real numbers.

Additionally, while the `omega` tactic is not yet deprecated in this version of Coq, it should soon be the case and we already recommend users to switch to `lia` in new proof scripts.

The `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases. See the *Changes in 8.11+beta1* section and following sections for the detailed list of changes, including potentially breaking changes marked with **Changed**.

Coq's documentation is available at https://coq.github.io/doc/v8.11/api (documentation of the ML API), https://coq.github.io/doc/v8.11/refman (reference manual), and https://coq.github.io/doc/v8.11/stdlib (documentation of the standard library).

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop and Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Karl Palmskog, Matthieu Sozeau and Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 61 contributors to this version are Michael D. Adams, Guillaume Allais, Helge Bahmann, Langston Barrett, Guillaume Bertholon, Frédéric Besson, Simon Boulier, Michele Caci, Tej Chajed, Arthur Charguéraud, Cyril Cohen, Frédéric Dabrowski, Arthur Azevedo de Amorim, Maxime Dénès, Nikita Eshkeev, Jim Fehrle, Emilio Jesús Gallego Arias, Paolo G. Giarrusso, Gaëtan Gilbert, Georges Gonthier, Jason Gross, Samuel Gruetter, Armaël Guéneau, Hugo Herbelin, Florent Hivert, Jasper Hugunin, Shachar Itzhaky, Jan-Oliver Kaiser, Robbert Krebbers, Vincent Laporte, Olivier Laurent, Samuel Lelièvre, Nicholas Lewycky, Yishuai Li, Jose Fernando Lopez Fernandez, Andreas Lynge, Kenji Maillard, Erik Martin-Dorel, Guillaume Melquiond, Alexandre Moine, Oliver Nash, Wojciech Nawrocki, Antonio Nikishaev, Pierre-Marie Pédrot, Clément Pit-Claudel, Lars Rasmusson, Robert Rand, Talia Ringer, JP Rodi, Pierre Roux, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Matthieu Sozeau, spanjel, Claude Stolze, Enrico Tassi, Laurent Théry, James R. Wilcox, Xia Li-yao, Théo Zimmermann

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list, the coq-club@inria.fr mailing list or the Discourse forum[679]. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.11 is the sixth release of Coq developed on a time-based development cycle. Its development spanned 3 months from the release of Coq 8.10. Pierre-Marie Pédrot is the release manager and maintainer of this release, assisted by Matthieu Sozeau. This release is the result of 2000+ commits and 300+ PRs merged, closing 75+ issues.

Paris, November 2019,
Matthieu Sozeau for the Coq development team

[679] https://coq.discourse.group/

Changes in 8.11+beta1

Kernel

- **Added:** A built-in support of floating-point arithmetic, allowing one to devise efficient reflection tactics involving numerical computation. Primitive floats are added in the language of terms, following the binary64 format of the IEEE 754 standard, and the related operations are implemented for the different reduction engines of Coq by using the corresponding processor operators in rounding-to-nearest-even. The properties of these operators are axiomatized in the theory `Coq.Floats.FloatAxioms` which is part of the library `Coq.Floats.Floats`. See Section *Primitive Floats* (#9867[680], closes #8276[681], by Guillaume Bertholon, Erik Martin-Dorel, Pierre Roux).
- **Changed:** Internal definitions generated by *`abstract`*-like tactics are now inlined inside universe *`Qed`*-terminated polymorphic definitions, similarly to what happens for their monomorphic counterparts, (#10439[682], by Pierre-Marie Pédrot).
- **Fixed:** Section data is now part of the kernel. Solves a soundness issue in interactive mode where global monomorphic universe constraints would be dropped when forcing a delayed opaque proof inside a polymorphic section. Also relaxes the nesting criterion for sections, as polymorphic sections can now appear inside a monomorphic one (#10664[683], by Pierre-Marie Pédrot).
- **Changed:** Using `SProp` is now allowed by default, without needing to pass `-allow-sprop` or use *`Allow StrictProp`* (#10811[684], by Gaëtan Gilbert).

Specification language, type inference

- **Added:** Annotation in `Arguments` for bidirectionality hints: it is now possible to tell type inference to use type information from the context once the `n` first arguments of an application are known. The syntax is: `Arguments foo x y & z`. See *Bidirectionality hints* (#10049[685], by Maxime Dénès with help from Enrico Tassi).
- **Added:** Record fields can be annotated to prevent them from being used as canonical projections; see *Canonical Structures* for details (#10076[686], by Vincent Laporte).
- **Changed:** Require parentheses around nested disjunctive patterns, so that pattern and term syntax are consistent; match branch patterns no longer require parentheses for notation at level 100 or more.

  Warning: Incompatibilities
  - In `match p with (_, (0|1)) => ...` parentheses may no longer be omitted around `0|1`.
  - Notation `(p | q)` now potentially clashes with core pattern syntax, and should be avoided. `-w disj-pattern-notation` flags such *`Notation`*.

  See *Extended pattern matching* for details (#10167[687], by Georges Gonthier).
- **Changed:** *`Function`* always opens a proof when used with a `measure` or `wf` annotation, see *Advanced recursive functions* for the updated documentation (#10215[688], by Enrico Tassi).

[680] https://github.com/coq/coq/pull/9867
[681] https://github.com/coq/coq/issues/8276
[682] https://github.com/coq/coq/pull/10439
[683] https://github.com/coq/coq/pull/10664
[684] https://github.com/coq/coq/pull/10811
[685] https://github.com/coq/coq/pull/10049
[686] https://github.com/coq/coq/pull/10076
[687] https://github.com/coq/coq/pull/10167
[688] https://github.com/coq/coq/pull/10215

- **Changed:** The legacy command *Add Morphism* always opens a proof and cannot be used inside a module type. In order to declare a module type parameter that happens to be a morphism, use *Declare Morphism*. See *Deprecated syntax and backward incompatibilities* for the updated documentation (#10215[689], by Enrico Tassi).
- **Changed:** The universe polymorphism setting now applies from the opening of a section. In particular, it is not possible anymore to mix polymorphic and monomorphic definitions in a section when there are no variables nor universe constraints defined in this section. This makes the behaviour consistent with the documentation. (#10441[690], by Pierre-Marie Pédrot)
- **Added:** The *Section* command now accepts the "universes" attribute. In addition to setting the section universe polymorphism, it also locally sets the universe polymorphic option inside the section. (#10441[691], by Pierre-Marie Pédrot)
- **Fixed:** `Program Fixpoint` now uses `ex` and `sig` to make telescopes involving `Prop` types (#10758[692], by Gaëtan Gilbert, fixing #10757[693] reported by Xavier Leroy).
- **Changed:** Output of the *Print* and *About* commands. Arguments meta-data is now displayed as the corresponding *Arguments* command instead of the human-targeted prose used in previous Coq versions. (#10985[694], by Gaëtan Gilbert).
- **Added:** *refine* attribute for *Instance*, a more predictable version of the old `Refine Instance Mode` which unconditionally opens a proof (#10996[695], by Gaëtan Gilbert).
- **Changed:** The unsupported attribute error is now an error-by-default warning, meaning it can be disabled (#10997[696], by Gaëtan Gilbert).
- **Fixed:** Bugs sometimes preventing to define valid (co)fixpoints with implicit arguments in the presence of local definitions, see #3282[697] (#11132[698], by Hugo Herbelin).

Example

The following features an implicit argument after a local definition. It was wrongly rejected.

```
Definition f := fix f (o := true) {n : nat} m {struct m} :=
  match m with 0 => 0 | S m' => f (n:=n+1) m' end.
```

Notations

- **Added:** Numeral Notations now support sorts in the input to printing functions (e.g., numeral notations can be defined for terms containing things like `@cons Set nat nil`). (#9883[699], by Jason Gross).
- **Added:** The *Notation* and *Infix* commands now support the `deprecated` attribute (#10180[700], by Maxime Dénès).
- **Deprecated:** The former `compat` annotation for notations is deprecated, and its semantics changed. It is now made equivalent to using a `deprecated` attribute, and is no longer connected with the `-compat` command-line flag (#10180[701], by Maxime Dénès).

[689] https://github.com/coq/coq/pull/10215
[690] https://github.com/coq/coq/pull/10441
[691] https://github.com/coq/coq/pull/10441
[692] https://github.com/coq/coq/pull/10758
[693] https://github.com/coq/coq/issues/10757
[694] https://github.com/coq/coq/pull/10985
[695] https://github.com/coq/coq/pull/10996
[696] https://github.com/coq/coq/pull/10997
[697] https://github.com/coq/coq/issues/3282
[698] https://github.com/coq/coq/pull/11132
[699] https://github.com/coq/coq/pull/9883
[700] https://github.com/coq/coq/pull/10180
[701] https://github.com/coq/coq/pull/10180

- **Changed:** A simplification of parsing rules could cause a slight change of parsing precedences for the very rare users who defined notations with `constr` at level strictly between 100 and 200 and used these notations on the right-hand side of a cast operator (`:`, `<:`, `<<:`) (#10963[702], by Théo Zimmermann, simplification initially noticed by Jim Fehrle).

Tactics

- **Added:** Syntax **`injection`** *`term`* **`as [=`** *`intropattern`*`+` **`]`** as an alternative to **`injection`** *`term`* **`as`** *`simple_intropattern`*`+` using the standard *`injection_intropattern`* syntax (#9288[703], by Hugo Herbelin).
- **Changed:** Reimplementation of the *`zify`* tactic. The tactic is more efficient and copes with dependent hypotheses. It can also be extended by redefining the tactic `zify_post_hook`. (#9856[704], fixes #8898[705], #7886[706], #9848[707] and #5155[708], by Frédéric Besson).
- **Changed:** The goal selector tactical `only` now checks that the goal range it is given is valid instead of ignoring goals out of the focus range (#10318[709], by Gaëtan Gilbert).
- **Added:** Flags *`Lia Cache`*, *`Nia Cache`* and *`Nra Cache`*. (#10765[710], by Frédéric Besson, see #10772[711] for use case).
- **Added:** The *`zify`* tactic is now aware of `Z.to_N`. (#10774[712], grants #9162[713], by Kazuhiko Sakaguchi).
- **Changed:** The *`assert_succeeds`* and *`assert_fails`* tactics now only run their tactic argument once, even if it has multiple successes. This prevents blow-up and looping from using multisuccess tactics with *`assert_succeeds`*. (#10966[714] fixes #10965[715], by Jason Gross).
- **Fixed:** The *`assert_succeeds`* and *`assert_fails`* tactics now behave correctly when their tactic fully solves the goal. (#10966[716] fixes #9114[717], by Jason Gross).

Tactic language

- **Added:** Ltac2, a new version of the tactic language Ltac, that doesn't preserve backward compatibility, has been integrated in the main Coq distribution. It is still experimental, but we already recommend users of advanced Ltac to start using it and report bugs or request enhancements. See its documentation in the *dedicated chapter* (#10002[718], plugin authored by Pierre-Marie Pédrot, with contributions by various users, integration by Maxime Dénès, help on integrating / improving the documentation by Théo Zimmermann and Jim Fehrle).
- **Added:** Ltac2 tactic notations with "constr" arguments can specify the notation scope for these arguments; see *Notations* for details (#10289[719], by Vincent Laporte).

[702] https://github.com/coq/coq/pull/10963
[703] https://github.com/coq/coq/pull/9288
[704] https://github.com/coq/coq/pull/9856
[705] https://github.com/coq/coq/issues/8898
[706] https://github.com/coq/coq/issues/7886
[707] https://github.com/coq/coq/issues/9848
[708] https://github.com/coq/coq/issues/5155
[709] https://github.com/coq/coq/pull/10318
[710] https://github.com/coq/coq/pull/10765
[711] https://github.com/coq/coq/issues/10772
[712] https://github.com/coq/coq/pull/10774
[713] https://github.com/coq/coq/issues/9162
[714] https://github.com/coq/coq/pull/10966
[715] https://github.com/coq/coq/issues/10965
[716] https://github.com/coq/coq/pull/10966
[717] https://github.com/coq/coq/issues/9114
[718] https://github.com/coq/coq/pull/10002
[719] https://github.com/coq/coq/pull/10289

- **Changed:** White spaces are forbidden in the **&*ident*** syntax for ltac2 references that are described in *Built-in quotations* (#10324[720], fixes #10088[721], authored by Pierre-Marie Pédrot).

SSReflect

- **Added:** Generalize tactics *under* and *over* for any registered relation. More precisely, assume the given context lemma has type `forall f1 f2, .. -> (forall i, R1 (f1 i) (f2 i)) -> R2 f1 f2`. The first step performed by *under* (since Coq 8.10) amounts to calling the tactic *rewrite*, which itself relies on *setoid_rewrite* if need be. So this step was already compatible with a double implication or setoid equality for the conclusion head symbol `R2`. But a further step consists in tagging the generated subgoal `R1 (f1 i) (?f2 i)` to protect it from unwanted evar instantiation, and get `Under_rel _ R1 (f1 i) (?f2 i)` that is displayed as `'Under[ f1 i ]`. In Coq 8.10, this second (convenience) step was only performed when `R1` was Leibniz' `eq` or `iff`. Now, it is also performed for any relation `R1` which has a `RewriteRelation` instance (a `RelationClasses.Reflexive` instance being also needed so *over* can discharge the `'Under[ _ ]` goal by instantiating the hidden evar.) This feature generalizing support for setoid-like relations is enabled as soon as we do both `Require Import ssreflect.` and `Require Setoid.` Finally, a rewrite rule `UnderE` has been added if one wants to "unprotect" the evar, and instantiate it manually with another rule than reflexivity (i.e., without using the *over* tactic nor the `over` rewrite rule). See also Section *Rewriting under binders* (#10022[722], by Erik Martin-Dorel, with suggestions and review by Enrico Tassi and Cyril Cohen).
- **Added:** A `void` notation for the standard library empty type (`Empty_set`) (#10932[723], by Arthur Azevedo de Amorim).
- **Added:** Lemma `inj_compr` to `ssr.ssrfun` (#11136[724], by Cyril Cohen).

Commands and options

- **Removed:** Deprecated flag `Refine Instance Mode` (#9530[725], fixes #3632[726], #3890[727] and #4638[728] by Maxime Dénès, review by Gaëtan Gilbert).
- **Changed:** *Fail* does not catch critical errors (including "stack overflow") anymore (#10173[729], by Gaëtan Gilbert).
- **Removed:** Undocumented **Instance : !*type*** syntax (#10185[730], by Gaëtan Gilbert).
- **Removed:** Deprecated `Show Script` command (#10277[731], by Gaëtan Gilbert).
- **Added:** Unsafe commands to enable/disable guard checking, positivity checking and universes checking (providing a local `-type-in-type`). See *Controlling Typing Flags* (#10291[732] by Simon Boulier).
- **Fixed:** Two bugs in *Export*. This can have an impact on the behavior of the *Import* command on libraries. `Import A` when `A` imports `B` which exports `C` was importing `C`, whereas *Import* is not transitive. Also, after `Import A B`, the import of `B` was sometimes incomplete (#10476[733], by Maxime Dénès).

[720] https://github.com/coq/coq/pull/10324
[721] https://github.com/coq/coq/issues/10088
[722] https://github.com/coq/coq/pull/10022
[723] https://github.com/coq/coq/pull/10932
[724] https://github.com/coq/coq/pull/11136
[725] https://github.com/coq/coq/pull/9530
[726] https://github.com/coq/coq/issues/3632
[727] https://github.com/coq/coq/issues/3890
[728] https://github.com/coq/coq/issues/4638
[729] https://github.com/coq/coq/pull/10173
[730] https://github.com/coq/coq/pull/10185
[731] https://github.com/coq/coq/pull/10277
[732] https://github.com/coq/coq/pull/10291
[733] https://github.com/coq/coq/pull/10476

> **Warning:** This is a common source of incompatibilities in projects migrating to Coq 8.11.

- **Changed:** Output generated by *Printing Dependent Evars Line* flag used by the Prooftree tool in Proof General. (#10489[734], closes #4504[735], #10399[736] and #10400[737], by Jim Fehrle).
- **Added:** Optionally highlight the differences between successive proof steps in the *Show Proof* command. Experimental; only available in coqtop and Proof General for now, may be supported in other IDEs in the future. (#10494[738], by Jim Fehrle).
- **Removed:** Legacy commands `AddPath`, `AddRecPath`, and `DelPath` which were undocumented, broken variants of *Add LoadPath*, *Add Rec LoadPath*, and *Remove LoadPath* (#11187[739], by Maxime Dénès and Théo Zimmermann).

Tools

- **Added:** `coqc` now provides the ability to generate compiled interfaces. Use `coqc -vos foo.v` to skip all opaque proofs during the compilation of `foo.v`, and output a file called `foo.vos`. This feature is experimental. It enables working on a Coq file without the need to first compile the proofs contained in its dependencies (#8642[740] by Arthur Charguéraud, review by Maxime Dénès and Emilio Gallego).
- **Added:** Command-line options `-require-import`, `-require-export`, `-require-import-from` and `-require-export-from`, as well as their shorthand, `-ri`, `-re`, `-refrom` and `-rifrom`. Deprecate confusing command line option `-require` (#10245[741] by Hugo Herbelin, review by Emilio Gallego).
- **Changed:** Renamed `VDFILE` from `.coqdeps.d` to `.<CoqMakefile>.d` in the `coq_makefile` utility, where `<CoqMakefile>` is the name of the output file given by the `-o` option. In this way two generated makefiles can coexist in the same directory. (#10947[742], by Kazuhiko Sakaguchi).
- **Fixed:** `coq_makefile` now supports environment variable `COQBIN` with no ending `/` character (#11068[743], by Gaëtan Gilbert).

Standard library

- **Changed:** Moved the *auto* hints of the `OrderedType` module into a new `ordered_type` database (#9772[744], by Vincent Laporte).
- **Removed:** Deprecated modules `Coq.ZArith.Zlogarithm` and `Coq.ZArith.Zsqrt_compat` (#9811[745], by Vincent Laporte).
- **Added:** Module `Reals.Cauchy.ConstructiveCauchyReals` defines constructive real numbers by Cauchy sequences of rational numbers (#10445[746], by Vincent Semeria, with the help and review of Guillaume Melquiond and Bas Spitters). This module is not meant to be imported directly, please import `Reals.Abstract.ConstructiveReals` instead.
- **Added:** New module `Reals.ClassicalDedekindReals` defines Dedekind real numbers as boolean-valued functions along with 3 logical axioms: limited principle of omniscience, excluded middle of negations, and

[734] https://github.com/coq/coq/pull/10489
[735] https://github.com/coq/coq/issues/4504
[736] https://github.com/coq/coq/issues/10399
[737] https://github.com/coq/coq/issues/10400
[738] https://github.com/coq/coq/pull/10494
[739] https://github.com/coq/coq/pull/11187
[740] https://github.com/coq/coq/pull/8642
[741] https://github.com/coq/coq/pull/10245
[742] https://github.com/coq/coq/pull/10947
[743] https://github.com/coq/coq/pull/11068
[744] https://github.com/coq/coq/pull/9772
[745] https://github.com/coq/coq/pull/9811
[746] https://github.com/coq/coq/pull/10445

functional extensionality. The exposed type `R` in module `Reals.Rdefinitions` now corresponds to these Dedekind reals, hidden behind an opaque module, which significantly reduces the number of axioms needed (see `Reals.Rdefinitions` and `Reals.Raxioms`), while preserving backward compatibility. Classical Dedekind reals are a quotient of constructive reals, which allows to transport many constructive proofs to the classical case (#10827[747], by Vincent Semeria, based on discussions with Guillaume Melquiond, Bas Spitters and Hugo Herbelin, code review by Hugo Herbelin).

- **Added:** New lemmas on `combine`, `filter`, `nodup`, `nth`, and `nth_error` functions on lists (#10651[748], and #10731[749], by Oliver Nash).
- **Changed:** The lemma `filter_app` was moved to the `List` module (#10651[750], by Oliver Nash).
- **Added:** Standard equivalence between weak excluded-middle and the classical instance of De Morgan's law, in module `ClassicalFacts` (#10895[751], by Hugo Herbelin).

Infrastructure and dependencies

- **Changed:** Coq now officially supports OCaml 4.08. See `INSTALL` file for details (#10471[752], by Emilio Jesús Gallego Arias).

Changes in 8.11.0

Kernel

- **Changed:** the native compilation (*`native_compute`*) now creates a directory to contain temporary files instead of putting them in the root of the system temporary directory (#11081[753], by Gaëtan Gilbert).
- **Fixed:** #11360[754]. Broken section closing when a template polymorphic inductive type depends on a section variable through its parameters (#11361[755], by Gaëtan Gilbert).
- **Fixed:** The type of `Set+1` would be computed to be itself, leading to a proof of False (#11422[756], by Gaëtan Gilbert).

Specification language, type inference

- **Changed:** Heuristics for universe minimization to `Set`: only minimize flexible universes (#10657[757], by Gaëtan Gilbert with help from Maxime Dénès and Matthieu Sozeau).
- **Fixed:** A dependency was missing when looking for default clauses in the algorithm for printing pattern matching clauses (#11233[758], by Hugo Herbelin, fixing #11231[759], reported by Barry Jay).

Notations

- **Fixed:** *`Print Visibility`* was failing in the presence of only-printing notations (#11276[760], by Hugo Herbelin, fixing #10750[761]).

[747] https://github.com/coq/coq/pull/10827
[748] https://github.com/coq/coq/pull/10651
[749] https://github.com/coq/coq/pull/10731
[750] https://github.com/coq/coq/pull/10651
[751] https://github.com/coq/coq/pull/10895
[752] https://github.com/coq/coq/pull/10471
[753] https://github.com/coq/coq/pull/11081
[754] https://github.com/issues/11360
[755] https://github.com/coq/coq/pull/11361
[756] https://github.com/coq/coq/pull/11422
[757] https://github.com/coq/coq/pull/10657
[758] https://github.com/coq/coq/pull/11233
[759] https://github.com/coq/coq/pull/11231
[760] https://github.com/coq/coq/pull/11276
[761] https://github.com/coq/coq/pull/10750

- **Fixed:** Recursive notations with custom entries were incorrectly parsing `constr` instead of custom grammars (#11311[762] by Maxime Dénès, fixes #9532[763], #9490[764]).

Tactics

- **Changed:** The tactics *`eapply`*, *`refine`* and variants no longer allow shelved goals to be solved by typeclass resolution (#10762[765], by Matthieu Sozeau).
- **Fixed:** The optional string argument to *`time`* is now properly quoted under *`Print Ltac`* (#11203[766], fixes #10971[767], by Jason Gross)
- **Fixed:** Efficiency regression of *`lia`* introduced in 8.10 by PR #9725[768] (#11263[769], fixes #11063[770], and #11242[771], and #11270[772], by Frédéric Besson).
- **Deprecated:** The undocumented `omega with` tactic variant has been deprecated. Using *`lia`* is the recommended replacement, though the old semantics of `omega with *` can be recovered with `zify; omega` (#11337[773], by Emilio Jesus Gallego Arias).
- **Fixed** For compatibility reasons, in 8.11, *`zify`* does not support `Z.pow_pos` by default. It can be enabled by explicitly loading the module `ZifyPow` (#11430[774] by Frédéric Besson fixes #11191[775]).

Tactic language

- **Fixed:** Syntax of tactic `cofix ... with ...` was broken since Coq 8.10 (#11241[776], by Hugo Herbelin).

Commands and options

- **Deprecated:** The `-load-ml-source` and `-load-ml-object` command line options have been deprecated; their use was very limited, you can achieve the same by adding object files in the linking step or by using a plugin (#11428[777], by Emilio Jesus Gallego Arias).

Tools

- **Fixed:** `coqtop --version` was broken when called in the middle of an installation process (#11255[778], by Hugo Herbelin, fixing #11254[779]).
- **Deprecated:** The `-quick` command is renamed to `-vio`, for consistency with the new `-vos` and `-vok` flags. Usage of `-quick` is now deprecated (#11280[780], by Arthur Charguéraud).
- **Fixed:** `coq_makefile` does not break when using the `CAMLPKGS` variable together with an unpacked (`mllib`) plugin (#11357[781], by Gaëtan Gilbert).

[762] https://github.com/coq/coq/pull/11311
[763] https://github.com/coq/coq/pull/9532
[764] https://github.com/coq/coq/pull/9490
[765] https://github.com/coq/coq/pull/10762
[766] https://github.com/coq/coq/pull/11203
[767] https://github.com/coq/coq/issues/10971
[768] https://github.com/coq/coq/pull/9725
[769] https://github.com/coq/coq/pull/11263
[770] https://github.com/coq/coq/issues/11063
[771] https://github.com/coq/coq/issues/11242
[772] https://github.com/coq/coq/issues/11270
[773] https://github.com/coq/coq/pull/11337
[774] https://github.com/coq/coq/pull/11430
[775] https://github.com/coq/coq/issues/11191
[776] https://github.com/coq/coq/pull/11241
[777] https://github.com/coq/coq/pull/11428
[778] https://github.com/coq/coq/pull/11255
[779] https://github.com/coq/coq/pull/11254
[780] https://github.com/coq/coq/pull/11280
[781] https://github.com/coq/coq/pull/11357

- **Fixed:** `coqdoc` with option `-g` (Gallina only) now correctly prints commands with attributes (#11394[782], fixes #11353[783], by Karl Palmskog).

CoqIDE

- **Changed:** CoqIDE now uses the GtkSourceView native implementation of the autocomplete mechanism (#11400[784], by Pierre-Marie Pédrot).

Standard library

- **Removed:** Export of module `RList` in `Ranalysis` and `Ranalysis_reg`. Module `RList` is still there but must be imported explicitly where required (#11396[785], by Michael Soegtrop).

Infrastructure and dependencies

- **Added:** Build date can now be overridden by setting the `SOURCE_DATE_EPOCH` environment variable (#11227[786], by Bernhard M. Wiedemann).

Changes in 8.11.1

Kernel

- **Fixed:** Allow more inductive types in `Unset Positivity Checking` mode (#11811[787], by SimonBoulier).

Notations

- **Fixed:** Bugs in dealing with precedences of notations in custom entries (#11530[788], by Hugo Herbelin, fixing in particular #9517[789], #9519[790], #9521[791], #11331[792]).
- **Added:** In primitive floats, print a warning when parsing a decimal value that is not exactly a binary64 floating-point number. For instance, parsing 0.1 will print a warning whereas parsing 0.5 won't. (#11859[793], by Pierre Roux).

CoqIDE

- **Fixed:** Compiling file paths containing spaces (#10008[794], by snyke7, fixing #11595[795]).

Infrastructure and dependencies

- **Added:** Bump official OCaml support and CI testing to 4.10.0 (#11131[796], #11123[797], #11102[798], by Emilio Jesus Gallego Arias, Jacques-Henri Jourdan, Guillaume Melquiond, and Guillaume Munch-Maccagnoni).

Miscellaneous

[782] https://github.com/coq/coq/pull/11394
[783] https://github.com/coq/coq/issues/11353
[784] https://github.com/coq/coq/pull/11400
[785] https://github.com/coq/coq/pull/11396
[786] https://github.com/coq/coq/pull/11227
[787] https://github.com/coq/coq/pull/11811
[788] https://github.com/coq/coq/pull/11530
[789] https://github.com/coq/coq/pull/9517
[790] https://github.com/coq/coq/pull/9519
[791] https://github.com/coq/coq/pull/9521
[792] https://github.com/coq/coq/pull/11331
[793] https://github.com/coq/coq/pull/11859
[794] https://github.com/coq/coq/pull/10008
[795] https://github.com/coq/coq/pull/11595
[796] https://github.com/coq/coq/pull/11131
[797] https://github.com/coq/coq/pull/11123
[798] https://github.com/coq/coq/pull/11102

- **Fixed:** `Extraction Implicit` on the constructor of a record was leading to an anomaly (#11329[799], by Hugo Herbelin, fixes #11114[800]).

Changes in 8.11.2

Kernel

- **Fixed:** Using `Require` inside a section caused an anomaly when closing the section. (#11972[801], by Gaëtan Gilbert, fixing #11783[802], reported by Attila Boros).

Tactics

- **Fixed:** Anomaly with induction schemes whose conclusion is not normalized (#12116[803], by Hugo Herbelin; fixes #12045[804])
- **Fixed:** Loss of location of some tactic errors (#12223[805], by Hugo Herbelin; fixes #12152[806] and #12255[807]).

Commands and options

- **Changed:** Ignore -native-compiler option when built without native compute support. (#12070[808], by Pierre Roux).

CoqIDE

- **Changed:** CoqIDE now uses native window frames by default on Windows. The GTK window frames can be restored by setting the `GTK_CSD` environment variable to `1` (#12060[809], fixes #11080[810], by Attila Gáspár).
- **Fixed:** New patch presumably fixing the random Coq 8.11 segfault issue with CoqIDE completion (#12068[811], by Hugo Herbelin, presumably fixing #11943[812]).
- **Fixed:** Highlighting style consistently applied to all three buffers of CoqIDE (#12106[813], by Hugo Herbelin; fixes #11506[814]).

Version 8.10

Summary of changes

Coq version 8.10 contains two major new features: support for a native fixed-precision integer type and a new sort SProp of strict propositions. It is also the result of refinements and stabilization of previous features, deprecations or removals of deprecated features, cleanups of the internals of the system and API, and many documentation improvements. This release includes many user-visible changes, including deprecations that are documented in the next subsection, and new features that are documented in the reference manual. Here are the most important user-visible changes:

[799] https://github.com/coq/coq/pull/11329
[800] https://github.com/coq/coq/pull/11114
[801] https://github.com/coq/coq/pull/11972
[802] https://github.com/coq/coq/issues/11783
[803] https://github.com/coq/coq/pull/12116
[804] https://github.com/coq/coq/pull/12045
[805] https://github.com/coq/coq/pull/12223
[806] https://github.com/coq/coq/pull/12152
[807] https://github.com/coq/coq/pull/12255
[808] https://github.com/coq/coq/pull/12070
[809] https://github.com/coq/coq/pull/12060
[810] https://github.com/coq/coq/issues/11080
[811] https://github.com/coq/coq/pull/12068
[812] https://github.com/coq/coq/pull/11943
[813] https://github.com/coq/coq/pull/12106
[814] https://github.com/coq/coq/pull/11506

- Kernel:
  - A notion of primitive object was added to the calculus. Its first instance is primitive cyclic unsigned integers, axiomatized in module `UInt63`. See Section *Primitive Integers*. The `Coq.Numbers.Cyclic.Int31` library is deprecated (#6914[815], by Maxime Dénès, Benjamin Grégoire and Vincent Laporte, with help and reviews from many others).
  - The SProp sort of definitionally proof-irrelevant propositions was introduced. SProp allows to mark proof terms as irrelevant for conversion, and is treated like Prop during extraction. It is enabled using the `-allow-sprop` command-line flag or the *`Allow StrictProp`* flag. See Chapter *SProp (proof irrelevant propositions)* (#8817[816], by Gaëtan Gilbert).
  - The unfolding heuristic in termination checking was made more complete, allowing more constants to be unfolded to discover valid recursive calls. Performance regression may occur in Fixpoint declarations without an explicit `{struct}` annotation, since guessing the decreasing argument can now be more expensive (#9602[817], by Enrico Tassi).
- Universes:
  - Added Subgraph variant to *`Print Universes`*. Try for instance `Print Universes Subgraph(sigT2.u1 sigT_of_sigT2.u1 projT3_eq.u1).` (#8451[818], by Gaëtan Gilbert).
  - Added private universes for opaque polymorphic constants, see the documentation for the *`Private Polymorphic Universes`* flag, and unset it to get the previous behaviour (#8850[819], by Gaëtan Gilbert).
- Notations:
  - New command *`String Notation`* to register string syntax for custom inductive types (#8965[820], by Jason Gross).
  - Experimental: *Number Notations* now parse decimal constants such as `1.02e+01` or `10.2`. Parsers added for `Q` and `R`. In the rare case when such numeral notations were used in a development along with `Q` or `R`, they may have to be removed or disambiguated through explicit scope annotations (#8764[821], by Pierre Roux).
- Ltac backtraces can be turned on using the *`Ltac Backtrace`* flag, which is off by default (#9142[822], fixes #7769[823] and #7385[824], by Pierre-Marie Pédrot).
- The tactics *`lia`*, *`nia`*, *`lra`*, *`nra`* are now using a novel Simplex-based proof engine. In case of regression, unset *`Simplex`* to get the venerable Fourier-based engine (#8457[825], by Fréderic Besson).
- SSReflect:
  - New intro patterns:
    - temporary introduction: `=> +`
    - block introduction: `=> [^ prefix ] [^~ suffix ]`
    - fast introduction: `=> >`
    - tactics as views: `=> /ltac:mytac`

[815] https://github.com/coq/coq/pull/6914
[816] https://github.com/coq/coq/pull/8817
[817] https://github.com/coq/coq/pull/9602
[818] https://github.com/coq/coq/pull/8451
[819] https://github.com/coq/coq/pull/8850
[820] https://github.com/coq/coq/pull/8965
[821] https://github.com/coq/coq/pull/8764
[822] https://github.com/coq/coq/pull/9142
[823] https://github.com/coq/coq/issues/7769
[824] https://github.com/coq/coq/issues/7385
[825] https://github.com/coq/coq/pull/8457

* replace hypothesis: `=> {}H`

  See Section *Introduction in the context* (#6705[826], by Enrico Tassi, with help from Maxime Dénès, ideas coming from various users).

– New tactic `under` to rewrite under binders, given an extensionality lemma:

  * interactive mode: **`under term`**, associated terminator: `over`
  * one-liner mode: **`under term do [tactic | ...]`**

  It can take occurrence switches, contextual patterns, and intro patterns: `under {2}[in RHS]eq_big => [i|i ?]` (#9651[827], by Erik Martin-Dorel and Enrico Tassi).

- `Combined Scheme` now works when inductive schemes are generated in sort Type. It used to be limited to sort `Prop` (#7634[828], by Théo Winterhalter).
- A new registration mechanism for reference from ML code to Coq constructs has been added (#186[829], by Emilio Jesús Gallego Arias, Maxime Dénès and Vincent Laporte).
- CoqIDE:
  – CoqIDE now depends on gtk+3 and lablgtk3 instead of gtk+2 and lablgtk2. The INSTALL file available in the Coq sources has been updated to list the new dependencies (#9279[830], by Hugo Herbelin, with help from Jacques Garrigue, Emilio Jesús Gallego Arias, Michael Sogetrop and Vincent Laporte).
  – Smart input for Unicode characters. For example, typing `\alpha` then `Shift+Space` will insert the greek letter alpha. A larger number of default bindings are provided, following the latex naming convention. Bindings can be customized, either globally, or on a per-project basis. See Section *Bindings for input of Unicode symbols* for details (#8560[831], by Arthur Charguéraud).
- Infrastructure and dependencies:
  – Coq 8.10 requires OCaml >= 4.05.0, bumped from 4.02.3 See the `INSTALL` file for more information on dependencies (#7522[832], by Emilio Jesús Gallego Arías).
  – Coq 8.10 doesn't need Camlp5 to build anymore. It now includes a fork of the core parsing library that Coq uses, which is a small subset of the whole Camlp5 distribution. In particular, this subset doesn't depend on the OCaml AST, allowing easier compilation and testing on experimental OCaml versions. Coq also ships a new parser `coqpp` that plugin authors must switch to (#7902[833], #7979[834], #8161[835], #8667[836], and #8945[837], by Pierre-Marie Pédrot and Emilio Jesús Gallego Arias).

    The Coq developers would like to thank Daniel de Rauglaudre for many years of continued support.

  – Coq now supports building with Dune, in addition to the traditional Makefile which is scheduled for deprecation (#6857[838], by Emilio Jesús Gallego Arias, with help from Rudi Grinberg).

    Experimental support for building Coq projects has been integrated in Dune at the same time, providing an improved experience[839] for plugin developers. We thank the Dune team for their work supporting Coq.

[826] https://github.com/coq/coq/pull/6705
[827] https://github.com/coq/coq/pull/9651
[828] https://github.com/coq/coq/pull/7634
[829] https://github.com/coq/coq/pull/186
[830] https://github.com/coq/coq/pull/9279
[831] https://github.com/coq/coq/pull/8560
[832] https://github.com/coq/coq/pull/7522
[833] https://github.com/coq/coq/pull/7902
[834] https://github.com/coq/coq/pull/7979
[835] https://github.com/coq/coq/pull/8161
[836] https://github.com/coq/coq/pull/8667
[837] https://github.com/coq/coq/pull/8945
[838] https://github.com/coq/coq/pull/6857
[839] https://coq.discourse.group/t/a-guide-to-building-your-coq-libraries-and-plugins-with-dune/

Version 8.10 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system, including many additions to the standard library (see the next subsection for details).

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version and a plugin development tutorial originally made by Yves Bertot is now in `doc/plugin_tutorial`. The `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, and Maxime Dénès.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop, Théo Zimmermann worked on maintaining and improving the continuous integration system and package building infrastructure. Coq is now continuously tested against the OCaml trunk, in addition to the oldest supported and latest OCaml releases.

Coq's documentation for the development branch is now deployed continuously at https://coq.github.io/doc/master/api (documentation of the ML API), https://coq.github.io/doc/master/refman (reference manual), and https://coq.github.io/doc/master/stdlib (documentation of the standard library). Similar links exist for the `v8.10` branch.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi (who migrated it to opam 2) with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 61 contributors to this version are Tanaka Akira, Benjamin Barenblat, Yves Bertot, Frédéric Besson, Lasse Blaauwbroek, Martin Bodin, Joachim Breitner, Tej Chajed, Frédéric Chapoton, Arthur Charguéraud, Cyril Cohen, Lukasz Czajka, David A. Dalrymple, Christian Doczkal, Maxime Dénès, Andres Erbsen, Jim Fehrle, Emilio Jesus Gallego Arias, Gaëtan Gilbert, Matěj Grabovský, Simon Gregersen, Jason Gross, Samuel Gruetter, Hugo Herbelin, Jasper Hugunin, Mirai Ikebuchi, Chantal Keller, Matej Košík, Sam Pablo Kuper, Vincent Laporte, Olivier Laurent, Larry Darryl Lee Jr, Nick Lewycky, Yao Li, Yishuai Li, Assia Mahboubi, Simon Marechal, Erik Martin-Dorel, Thierry Martinez, Guillaume Melquiond, Kayla Ngan, Karl Palmskog, Pierre-Marie Pédrot, Clément Pit-Claudel, Pierre Roux, Kazuhiko Sakaguchi, Ryan Scott, Vincent Semeria, Gan Shen, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Kamil Trzciński, whitequark, Théo Winterhalter, Xia Li-yao, Beta Ziliani and Théo Zimmermann.

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list, the coq-club@inria.fr mailing list or the new Discourse forum. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.10 is the fifth release of Coq developed on a time-based development cycle. Its development spanned 6 months from the release of Coq 8.9. Vincent Laporte is the release manager and maintainer of this release. This release is the result of ~2500 commits and ~650 PRs merged, closing 150+ issues.

Santiago de Chile, April 2019,
Matthieu Sozeau for the Coq development team

Other changes in 8.10+beta1

- Command-line tools and options:
  - The use of `coqtop` as a compiler has been deprecated, in favor of `coqc`. Consequently option `-compile` will stop to be accepted in the next release. `coqtop` is now reserved to interactive use (#9095[840], by Emilio Jesús Gallego Arias).
  - New option `-topfile filename`, which will set the current module name (*à la* `-top`) based on the filename passed, taking into account the proper `-R`/`-Q` options. For example, given `-R Foo foolib`

[840] https://github.com/coq/coq/pull/9095

using `-topfile foolib/bar.v` will set the module name to `Foo.Bar`. CoqIDE now properly sets the module name for a given file based on its path (#8991[841], closes #8989[842], by Gaëtan Gilbert).

– Experimental: Coq flags and options can now be set on the command-line, e.g. `-set "Universe Polymorphism=true"` (#9876[843], by Gaëtan Gilbert).

– The `-native-compiler` flag of `coqc` and `coqtop` now takes an argument which can have three values:

  * `no` disables native_compute
  * `yes` enables native_compute and precompiles `.v` files to native code
  * `ondemand` enables native_compute but compiles code only when `native_compute` is called

  The default value is `ondemand`. Note that this flag now has priority over the configure flag of the same name.

  A new `-bytecode-compiler` flag for `coqc` and `coqtop` controls whether conversion can use the VM. The default value is `yes`.

  (#8870[844], by Maxime Dénès)

– The pretty timing diff scripts (flag `TIMING=1` to a `coq_makefile`-made `Makefile`, also `tools/make-both-single-timing-files.py`, `tools/make-both-time-files.py`, and `tools/make-one-time-file.py`) now correctly support non-UTF-8 characters in the output of `coqc` / `make` as well as printing to stdout, on both python2 and python3 (#9872[845], closes #9767[846] and #9705[847], by Jason Gross)

– coq_makefile's install target now errors if any file to install is missing (#9906[848], by Gaëtan Gilbert).

– Preferences from `coqide.keys` are no longer overridden by modifiers preferences in `coqiderc` (#10014[849], by Hugo Herbelin).

- Specification language, type inference:

  – Fixing a missing check in interpreting instances of existential variables that are bound to local definitions. Might exceptionally induce an overhead if the cost of checking the conversion of the corresponding definitions is additionally high (#8217[850], closes #8215[851], by Hugo Herbelin).

  – A few improvements in inference of the return clause of `match` that can exceptionally introduce incompatibilities. This can be solved by writing an explicit `return` clause, sometimes even simply an explicit `return _` clause (#262[852], by Hugo Herbelin).

  – Using non-projection values with the projection syntax is not allowed. For instance `0.(S)` is not a valid way to write `S 0`. Projections from non-primitive (emulated) records are allowed with warning "nonprimitive-projection-syntax" (#8829[853], by Gaëtan Gilbert).

  – An option and attributes to control the automatic decision to declare an inductive type as template polymorphic were added. Warning "auto-template" (off by default) can trigger when an inductive is automatically declared template polymorphic without the attribute.

    Inductive types declared by Funind will never be template polymorphic.

[841] https://github.com/coq/coq/pull/8991
[842] https://github.com/coq/coq/issues/8989
[843] https://github.com/coq/coq/pull/9876
[844] https://github.com/coq/coq/pull/8870
[845] https://github.com/coq/coq/pull/9872
[846] https://github.com/coq/coq/issues/9767
[847] https://github.com/coq/coq/issues/9705
[848] https://github.com/coq/coq/pull/9906
[849] https://github.com/coq/coq/pull/10014
[850] https://github.com/coq/coq/pull/8217
[851] https://github.com/coq/coq/issues/8215
[852] https://github.com/coq/coq/pull/262
[853] https://github.com/coq/coq/pull/8829

(#8488[854], by Gaëtan Gilbert)

- Notations:
  - New command *Declare Scope* to explicitly declare a scope name before any use of it. Implicit declaration of a scope at the time of *Bind Scope*, *Delimit Scope*, *Undelimit Scope*, or *Notation* is deprecated (#7135[855], by Hugo Herbelin).
  - Various bugs have been fixed (e.g. #9214[856] on removing spurious parentheses on abbreviations shortening a strict prefix of an application, by Hugo Herbelin).
  - *Number Notation* now support inductive types in the input to printing functions (e.g., numeral notations can be defined for terms containing things like `@cons nat O O`), and parsing functions now fully normalize terms including parameters of constructors (so that, e.g., a numeral notation whose parsing function outputs a proof of `Nat.gcd x y = 1` will no longer fail to parse due to containing the constant `Nat.gcd` in the parameter-argument of `eq_refl`) (#9874[857], closes #9840[858] and #9844[859], by Jason Gross).
  - Deprecated compatibility notations have actually been removed. Uses of these notations are generally easy to fix thanks to the hint contained in the deprecation warning emitted by Coq 8.8 and 8.9. For projects that require more than a handful of such fixes, there is a script[860] that will do it automatically, using the output of `coqc` (#8638[861], by Jason Gross).
  - Allow inspecting custom grammar entries by *Print Custom Grammar* (#10061[862], fixes #9681[863], by Jasper Hugunin, review by Pierre-Marie Pédrot and Hugo Herbelin).
- The quote plugin[864] was removed. If some users are interested in maintaining this plugin externally, the Coq development team can provide assistance for extracting the plugin and setting up a new repository (#7894[865], by Maxime Dénès).
- Ltac:
  - Tactic names are no longer allowed to clash, even if they are not defined in the same section. For example, the following is no longer accepted: `Ltac foo := idtac. Section S. Ltac foo := fail. End S.` (#8555[866], by Maxime Dénès).
  - Names of existential variables occurring in Ltac functions (e.g. `?[n]` or `?n` in terms - not in patterns) are now interpreted the same way as other variable names occurring in Ltac functions (#7309[867], by Hugo Herbelin).
- Tactics:
  - Removed the deprecated `romega` tactic (#8419[868], by Maxime Dénès and Vincent Laporte).
  - Hint declaration and removal should now specify a database (e.g. `Hint Resolve foo : database`). When the database name is omitted, the hint is added to the `core` database (as previously), but a deprecation warning is emitted (#8987[869], by Maxime Dénès).

[854] https://github.com/coq/coq/pull/8488
[855] https://github.com/coq/coq/pull/7135
[856] https://github.com/coq/coq/pull/9214
[857] https://github.com/coq/coq/pull/9874
[858] https://github.com/coq/coq/issues/9840
[859] https://github.com/coq/coq/issues/9844
[860] https://gist.github.com/JasonGross/9770653967de3679d131c59d42de6d17#file-replace-notations-py
[861] https://github.com/coq/coq/pull/8638
[862] https://github.com/coq/coq/pull/10061
[863] https://github.com/coq/coq/pull/9681
[864] https://coq.inria.fr/distrib/V8.9.0/refman/proof-engine/detailed-tactic-examples.html#quote
[865] https://github.com/coq/coq/pull/7894
[866] https://github.com/coq/coq/pull/8555
[867] https://github.com/coq/coq/pull/7309
[868] https://github.com/coq/coq/pull/8419
[869] https://github.com/coq/coq/pull/8987

  - There are now tactics in `PreOmega.v` called `Z.div_mod_to_equations`, `Z.quot_rem_to_equations`, and `Z.to_euclidean_division_equations` (which combines the `div_mod` and `quot_rem` variants) which allow *`lia`*, *`nia`*, etc to support `Z.div` and `Z.modulo` (`Z.quot` and `Z.rem`, respectively), by posing the specifying equation for `Z.div` and `Z.modulo` before replacing them with atoms (#8062[870], by Jason Gross).
  - The syntax of the *`autoapply`* tactic was fixed to conform with preexisting documentation: it now takes a `with` clause instead of a `using` clause (#9524[871], closes #7632[872], by Théo Zimmermann).
  - Modes are now taken into account by *`typeclasses eauto`* for local hypotheses (#9996[873], fixes #5752[874], by Maxime Dénès, review by Pierre-Marie Pédrot).
  - New variant *`change_no_check`* of *`change`*, usable as a documented replacement of `convert_concl_no_check` (#10012[875], #10017[876], #10053[877], and #10059[878], by Hugo Herbelin and Paolo G. Giarrusso).
  - The simplified value returned by *`field_simplify`* is not always a fraction anymore. When the denominator is `1`, it returns `x` while previously it was returning `x/1`. This change could break codes that were post-processing application of *`field_simplify`* to get rid of these `x/1` (#9854[879], by Laurent Théry, with help from Michael Soegtrop, Maxime Dénès, and Vincent Laporte).

- SSReflect:
  - Clear discipline made consistent across the entire proof language. Whenever a clear switch `{x..}` comes immediately before an existing proof context entry (used as a view, as a rewrite rule or as name for a new context entry) then such entry is cleared too.

    E.g. The following sentences are elaborated as follows (when H is an existing proof context entry):

    ```
    * => {x..} H -> => {x..H} H
    * => {x..} /H -> => /v {x..H}
    * rewrite {x..} H -> rewrite E {x..H}
    ```

    (#9341[880], by Enrico Tassi).
  - `inE` now expands `y in r x` when `r` is a `simpl_rel`. New `{pred T}` notation for a `pred T` alias in the `pred_sort` coercion class, simplified `predType` interface: `pred_class` and `mkPredType` deprecated, `{pred T}` and `PredType` should be used instead. `if c return t then ...` now expects `c` to be a variable bound in `t`. New `nonPropType` interface matching types that do _not_ have sort `Prop`. New `relpre R f` definition for the preimage of a relation R under f (#9995[881], by Georges Gonthier).

- Commands:
  - Binders for an *`Instance`* now act more like binders for a *`Theorem`*. Names may not be repeated, and may not overlap with section variable names (#8820[882], closes #8791[883], by Jasper Hugunin).

[870] https://github.com/coq/coq/pull/8062
[871] https://github.com/coq/coq/pull/9524
[872] https://github.com/coq/coq/issues/7632
[873] https://github.com/coq/coq/pull/9996
[874] https://github.com/coq/coq/issues/5752
[875] https://github.com/coq/coq/pull/10012
[876] https://github.com/coq/coq/pull/10017
[877] https://github.com/coq/coq/pull/10053
[878] https://github.com/coq/coq/pull/10059
[879] https://github.com/coq/coq/pull/9854
[880] https://github.com/coq/coq/pull/9341
[881] https://github.com/coq/coq/pull/9995
[882] https://github.com/coq/coq/pull/8820
[883] https://github.com/coq/coq/issues/8791

- Removed the deprecated `Implicit Tactic` family of commands (#8779[884], by Pierre-Marie Pédrot).
- The `Automatic Introduction` option has been removed and is now the default (#9001[885], by Emilio Jesús Gallego Arias).
- `Arguments` now accepts names for arguments provided with `extra_scopes` (#9117[886], by Maxime Dénès).
- The naming scheme for anonymous binders in a `Theorem` has changed to avoid conflicts with explicitly named binders (#9160[887], closes #8819[888], by Jasper Hugunin).
- Computation of implicit arguments now properly handles local definitions in the binders for an `Instance`, and can be mixed with implicit binders `{x : T}` (#9307[889], closes #9300[890], by Jasper Hugunin).
- *`Declare Instance`* now requires an instance name.

  The flag `Refine Instance Mode` has been turned off by default, meaning that *`Instance`* no longer opens a proof when a body is provided. The flag has been deprecated and will be removed in the next version.

  (#9270[891], and #9825[892], by Maxime Dénès)
- Command *`Instance`*, when no body is provided, now always opens a proof. This is a breaking change, as instance of **`Instance`** ***`ident`*$_1$** **`:`** ***`ident`*$_2$** **`.`** where ***`ident`*$_2$** is a trivial class will have to be changed into **`Instance`** ***`ident`*$_1$** **`:`** ***`ident`*$_2$** **`:= {}.`** or **`Instance`** ***`ident`*$_1$** **`:`** ***`ident`*$_2$** **`. Proof. Qed.`** (#9274[893], by Maxime Dénès).
- The flag *`Program Mode`* now means that the `Program` attribute is enabled for all commands that support it. In particular, it does not have any effect on tactics anymore. May cause some incompatibilities (#9410[894], by Maxime Dénès).
- The algorithm computing implicit arguments now behaves uniformly for primitive projection and application nodes (#9509[895], closes #9508[896], by Pierre-Marie Pédrot).
- *`Hypotheses`* and *`Variables`* can now take implicit binders inside sections (#9364[897], closes #9363[898], by Jasper Hugunin).
- Removed deprecated option `Automatic Coercions Import` (#8094[899], by Maxime Dénès).
- The `Show Script` command has been deprecated (#9829[900], by Vincent Laporte).
- *`Coercion`* does not warn ambiguous paths which are obviously convertible with existing ones. The ambiguous paths messages have been turned to warnings, thus now they could appear in the output of `coqc`. The convertibility checking procedure for coercion paths is complete for paths consisting of coercions satisfying the uniform inheritance condition, but some coercion paths could be reported as ambiguous even if they are

[884] https://github.com/coq/coq/pull/8779
[885] https://github.com/coq/coq/pull/9001
[886] https://github.com/coq/coq/pull/9117
[887] https://github.com/coq/coq/pull/9160
[888] https://github.com/coq/coq/issues/8819
[889] https://github.com/coq/coq/pull/9307
[890] https://github.com/coq/coq/issues/9300
[891] https://github.com/coq/coq/pull/9270
[892] https://github.com/coq/coq/pull/9825
[893] https://github.com/coq/coq/pull/9274
[894] https://github.com/coq/coq/pull/9410
[895] https://github.com/coq/coq/pull/9509
[896] https://github.com/coq/coq/issues/9508
[897] https://github.com/coq/coq/pull/9364
[898] https://github.com/coq/coq/issues/9363
[899] https://github.com/coq/coq/pull/8094
[900] https://github.com/coq/coq/pull/9829

convertible with existing ones when they have coercions that don't satisfy the uniform inheritance condition (#9743[901], closes #3219[902], by Kazuhiko Sakaguchi).

- A new flag *Fast Name Printing* has been introduced. It changes the algorithm used for allocating bound variable names for a faster but less clever one (#9078[903], by Pierre-Marie Pédrot).
- Option `Typeclasses Axioms Are Instances` (compatibility option introduced in the previous version) is deprecated. Use *Declare Instance* for axioms which should be instances (#8920[904], by Gaëtan Gilbert).
- Removed option `Printing Primitive Projection Compatibility` (#9306[905], by Gaëtan Gilbert).

- Standard Library:
  - Added `Bvector.BVeq` that decides whether two `Bvectors` are equal. Added notations for `BVxor`, `BVand`, `BVor`, `BVeq` and `BVneg` (#8171[906], by Yishuai Li).
  - Added `ByteVector` type that can convert to and from `string` (#8365[907], by Yishuai Li).
  - Added lemmas about monotonicity of `N.double` and `N.succ_double`, and about the upper bound of number represented by a vector. Allowed implicit vector length argument in `Ndigits.Bv2N` (#8815[908], by Yishuai Li).
  - The prelude used to be automatically Exported and is now only Imported. This should be relevant only when importing files which don't use `-noinit` into files which do (#9013[909], by Gaëtan Gilbert).
  - Added `Coq.Structures.OrderedTypeEx.String_as_OT` to make strings an ordered type, using lexical order (#7221[910], by Li Yao).
  - Added lemmas about `Z.testbit`, `Z.ones`, and `Z.modulo` (#9425[911], by Andres Erbsen).
  - Moved the `auto` hints of the `FSet` library into a new `fset` database (#9725[912], by Frédéric Besson).
  - Added `Coq.Structures.EqualitiesFacts.PairUsualDecidableTypeFull` (#9984[913], by Jean-Christophe Léchenet and Oliver Nash).
- Some error messages that show problems with a pair of non-matching values will now highlight the differences (#8669[914], by Jim Fehrle).
- Changelog has been moved from a specific file `CHANGES.md` to the reference manual; former Credits chapter of the reference manual has been split in two parts: a History chapter which was enriched with additional historical information about Coq versions 1 to 5, and a Changes chapter which was enriched with the content formerly in `CHANGES.md` and `COMPATIBILITY` (#9133[915], #9668[916], #9939[917], #9964[918], and #10085[919], by Théo

[901] https://github.com/coq/coq/pull/9743
[902] https://github.com/coq/coq/issues/3219
[903] https://github.com/coq/coq/pull/9078
[904] https://github.com/coq/coq/pull/8920
[905] https://github.com/coq/coq/pull/9306
[906] https://github.com/coq/coq/pull/8171
[907] https://github.com/coq/coq/pull/8365
[908] https://github.com/coq/coq/pull/8815
[909] https://github.com/coq/coq/pull/9013
[910] https://github.com/coq/coq/pull/7221
[911] https://github.com/coq/coq/pull/9425
[912] https://github.com/coq/coq/pull/9725
[913] https://github.com/coq/coq/pull/9984
[914] https://github.com/coq/coq/pull/8669
[915] https://github.com/coq/coq/pull/9133
[916] https://github.com/coq/coq/pull/9668
[917] https://github.com/coq/coq/pull/9939
[918] https://github.com/coq/coq/pull/9964
[919] https://github.com/coq/coq/pull/10085

Zimmermann, with help and ideas from Emilio Jesús Gallego Arias, Gaëtan Gilbert, Clément Pit-Claudel, Matthieu Sozeau, and Enrico Tassi).

Changes in 8.10+beta2

Many bug fixes and documentation improvements, in particular:

Tactics

- Make the `discriminate` tactic work together with `Universe Polymorphism` and equality in `Type`. This, in particular, makes `discriminate` compatible with the HoTT library https://github.com/HoTT/HoTT (#10205[920], by Andreas Lynge, review by Pierre-Marie Pédrot and Matthieu Sozeau).

SSReflect

- Make the `case E: t` tactic work together with `Universe Polymorphism` and equality in `Type`. This makes `case` compatible with the HoTT library https://github.com/HoTT/HoTT (#10302[921], fixes #10301[922], by Andreas Lynge, review by Enrico Tassi)
- Make the `rewrite /t` tactic work together with `Universe Polymorphism`. This makes `rewrite` compatible with the HoTT library https://github.com/HoTT/HoTT (#10305[923], fixes #9336[924], by Andreas Lynge, review by Enrico Tassi)

CoqIDE

- Fix CoqIDE instability on Windows after the update to gtk3 (#10360[925], by Michael Soegtrop, closes #9885[926]).

Miscellaneous

- Proof General can now display Coq-generated diffs between proof steps in color (#10019[927] and (in Proof General) #421[928], by Jim Fehrle).

Changes in 8.10+beta3

Kernel

- Fix soundness issue with template polymorphism (#9294[929]).

  Declarations of template-polymorphic inductive types ignored the provenance of the universes they were abstracting on and did not detect if they should be greater or equal to Set in general. Previous universes and universes introduced by the inductive definition could have constraints that prevented their instantiation with e.g. Prop, resulting in unsound instantiations later. The implemented fix only allows abstraction over universes introduced by the inductive declaration, and properly records all their constraints by making them by default only $>=$ Prop. It is also checked that a template polymorphic inductive actually is polymorphic on at least one universe.

  This prevents inductive declarations in sections to be universe polymorphic over section parameters. For a backward compatible fix, simply hoist the inductive definition out of the section. An alternative is to declare the inductive as universe-polymorphic and cumulative in a universe-polymorphic section: all universes and constraints will be

[920] https://github.com/coq/coq/pull/10205
[921] https://github.com/coq/coq/pull/10302
[922] https://github.com/coq/coq/issues/10301
[923] https://github.com/coq/coq/pull/10305
[924] https://github.com/coq/coq/issues/9336
[925] https://github.com/coq/coq/pull/10360
[926] https://github.com/coq/coq/issues/9885
[927] https://github.com/coq/coq/pull/10019
[928] https://github.com/ProofGeneral/PG/pull/421
[929] https://github.com/coq/coq/issues/9294

properly gathered in this case. See *Template polymorphism* for a detailed exposition of the rules governing template-polymorphic types.

To help users incrementally fix this issue, a command line option `-no-template-check` and a global flag `Template Check` are available to selectively disable the new check. Use at your own risk.

(#9918[930], by Matthieu Sozeau and Maxime Dénès).

User messages

- Improve the ambiguous paths warning to indicate which path is ambiguous with new one (#10336[931], closes #3219[932], by Kazuhiko Sakaguchi).

Extraction

- Fix extraction to OCaml of primitive machine integers; see *Primitive Integers* (#10430[933], fixes #10361[934], by Vincent Laporte).
- Fix a printing bug of OCaml extraction on dependent record projections, which produced improper `assert false`. This change makes the OCaml extractor internally inline record projections by default; thus the monolithic OCaml extraction (`Extraction` and `Recursive Extraction`) does not produce record projection constants anymore except for record projections explicitly instructed to extract, and records declared in opaque modules (#10577[935], fixes #7348[936], by Kazuhiko Sakaguchi).

Standard library

- Added `splitat` function and lemmas about `splitat` and `uncons` (#9379[937], by Yishuai Li, with help of Konstantinos Kallas, follow-up of #8365[938], which added `uncons` in 8.10+beta1).

Changes in 8.10.0

- Micromega tactics (`lia`, `nia`, etc) are no longer confused by primitive projections (#10806[939], fixes #9512[940] by Vincent Laporte).

Changes in 8.10.1

A few bug fixes and documentation improvements, in particular:

Kernel

- Fix proof of False when using SProp (incorrect De Bruijn handling when inferring the relevance mark of a function) (#10904[941], by Pierre-Marie Pédrot).

Tactics

- Fix an anomaly when unsolved evar in `Add Ring` (#10891[942], fixes #9851[943], by Gaëtan Gilbert).

[930] https://github.com/coq/coq/pull/9918
[931] https://github.com/coq/coq/pull/10336
[932] https://github.com/coq/coq/issues/3219
[933] https://github.com/coq/coq/pull/10430
[934] https://github.com/coq/coq/issues/10361
[935] https://github.com/coq/coq/pull/10577
[936] https://github.com/coq/coq/issues/7348
[937] https://github.com/coq/coq/pull/9379
[938] https://github.com/coq/coq/pull/8365
[939] https://github.com/coq/coq/pull/10806
[940] https://github.com/coq/coq/issues/9512
[941] https://github.com/coq/coq/pull/10904
[942] https://github.com/coq/coq/pull/10891
[943] https://github.com/coq/coq/issues/9851

Tactic language

- Fix Ltac regression in binding free names in uconstr (#10899[944], fixes #10894[945], by Hugo Herbelin).

CoqIDE

- Fix handling of unicode input before space (#10852[946], fixes #10842[947], by Arthur Charguéraud).

Extraction

- Fix custom extraction of inductives to JSON (#10897[948], fixes #4741[949], by Helge Bahmann).

Changes in 8.10.2

Kernel

- Fixed a critical bug of template polymorphism and nonlinear universes (#11128[950], fixes #11039[951], by Gaëtan Gilbert).
- Fixed an anomaly "Uncaught exception Constr.DestKO" on `Inductive` (#11052[952], fixes #11048[953], by Gaëtan Gilbert).
- Fixed an anomaly "not enough abstractions in fix body" (#11014[954], fixes #8459[955], by Gaëtan Gilbert).

Notations

- Fixed an 8.10 regression related to the printing of coercions associated with notations (#11090[956], fixes #11033[957], by Hugo Herbelin).

CoqIDE

- Fixed uneven dimensions of CoqIDE panels when window has been resized (#11070[958], fixes 8.10-regression #10956[959], by Guillaume Melquiond).
- Do not include final stops in queries (#11069[960], fixes 8.10-regression #11058[961], by Guillaume Melquiond).

Infrastructure and dependencies

- Enable building of executables when they are running (#11000[962], fixes 8.9-regression #10728[963], by Gaëtan Gilbert).

[944] https://github.com/coq/coq/pull/10899
[945] https://github.com/coq/coq/issues/10894
[946] https://github.com/coq/coq/pull/10852
[947] https://github.com/coq/coq/issues/10842
[948] https://github.com/coq/coq/pull/10897
[949] https://github.com/coq/coq/issues/4741
[950] https://github.com/coq/coq/pull/11128
[951] https://github.com/coq/coq/issues/11039
[952] https://github.com/coq/coq/pull/11052
[953] https://github.com/coq/coq/issues/11048
[954] https://github.com/coq/coq/pull/11014
[955] https://github.com/coq/coq/issues/8459
[956] https://github.com/coq/coq/pull/11090
[957] https://github.com/coq/coq/issues/11033
[958] https://github.com/coq/coq/pull/11070
[959] https://github.com/coq/coq/issues/10956
[960] https://github.com/coq/coq/pull/11069
[961] https://github.com/coq/coq/issues/11058
[962] https://github.com/coq/coq/pull/11000
[963] https://github.com/coq/coq/issues/10728

Version 8.9

Summary of changes

Coq version 8.9 contains the result of refinements and stabilization of features and deprecations or removals of deprecated features, cleanups of the internals of the system and API along with a few new features. This release includes many user-visible changes, including deprecations that are documented in the next subsection and new features that are documented in the reference manual. Here are the most important changes:

- Kernel: mutually recursive records are now supported, by Pierre-Marie Pédrot.
- Notations:
  - Support for autonomous grammars of terms called "custom entries", by Hugo Herbelin (see Section *Custom entries* of the reference manual).
  - Deprecated notations of the standard library will be removed in the next version of Coq, see the next subsection for a script to ease porting, by Jason Gross and Jean-Christophe Léchenet.
  - Added the `Number Notation` command for registering decimal numeral notations for custom types, by Daniel de Rauglaudre, Pierre Letouzey and Jason Gross.
- Tactics: Introduction tactics `intro`/`intros` on a goal that is an existential variable now force a refinement of the goal into a dependent product rather than failing, by Hugo Herbelin.
- Decision procedures: deprecation of tactic `romega` in favor of `lia` and removal of `fourier`, replaced by `lra` which subsumes it, by Frédéric Besson, Maxime Dénès, Vincent Laporte and Laurent Théry.
- Proof language: focusing bracket `{` now supports named *goals*, e.g. `[x]:{` will focus on a goal (existential variable) named `x`, by Théo Zimmermann.
- SSReflect: the implementation of delayed clear was simplified by Enrico Tassi: the variables are always renamed using inaccessible names when the clear switch is processed and finally cleared at the end of the intro pattern. In addition to that, the use-and-discard flag `{}` typical of rewrite rules can now be also applied to views, e.g. `=> {}/v` applies `v` and then clears `v`. See Section *Introduction in the context*.
- Vernacular:
  - Experimental support for *attributes* on commands, by Vincent Laporte, as in `#[local] Lemma foo : bar.` Tactics and tactic notations now support the `deprecated` attribute.
  - Removed deprecated commands `Arguments Scope` and `Implicit Arguments` in favor of `Arguments`, with the help of Jasper Hugunin.
  - New flag `Uniform Inductive Parameters` by Jasper Hugunin to avoid repeating uniform parameters in constructor declarations.
  - New commands `Hint Variables` and `Hint Constants`, by Matthieu Sozeau, for controlling the opacity status of variables and constants in hint databases. It is recommended to always use these commands after creating a hint database with `Create HintDb`.
  - Multiple sections with the same name are now allowed, by Jasper Hugunin.
- Library: additions and changes in the `VectorDef`, `Ascii`, and `String` libraries. Syntax notations are now available only when using `Import` of libraries and not merely `Require`, by various contributors (source of incompatibility, see the next subsection for details).
- Toplevels: `coqtop` and `coqide` can now display diffs between proof steps in color, using the `Diffs` option, by Jim Fehrle.
- Documentation: we integrated a large number of fixes to the new Sphinx documentation by various contributors, coordinated by Clément Pit-Claudel and Théo Zimmermann.

- Tools: removed the `gallina` utility and the homebrewed `Emacs` mode.
- Packaging: as in Coq 8.8.2, the Windows installer now includes many more external packages that can be individually selected for installation, by Michael Soegtrop.

Version 8.9 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. Most important ones are documented in the next subsection file.

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version and a plugin development tutorial kept in sync with Coq was introduced by Yves Bertot http://github.com/ybertot/plugin_tutorials. The new `dev/doc/critical-bugs` file documents the known critical bugs of Coq and affected releases.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, and Maxime Dénès.

Maxime Dénès, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Michael Soegtrop, Théo Zimmermann worked on maintaining and improving the continuous integration system.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 54 contributors for this version are Léo Andrès, Rin Arakaki, Benjamin Barenblat, Langston Barrett, Siddharth Bhat, Martin Bodin, Simon Boulier, Timothy Bourke, Joachim Breitner, Tej Chajed, Arthur Charguéraud, Pierre Courtieu, Maxime Dénès, Andres Erbsen, Jim Fehrle, Julien Forest, Emilio Jesus Gallego Arias, Gaëtan Gilbert, Matěj Grabovský, Jason Gross, Samuel Gruetter, Armaël Guéneau, Hugo Herbelin, Jasper Hugunin, Ralf Jung, Sam Pablo Kuper, Ambroise Lafont, Leonidas Lampropoulos, Vincent Laporte, Peter LeFanu Lumsdaine, Pierre Letouzey, Jean-Christophe Léchenet, Nick Lewycky, Yishuai Li, Sven M. Hallberg, Assia Mahboubi, Cyprien Mangin, Guillaume Melquiond, Perry E. Metzger, Clément Pit-Claudel, Pierre-Marie Pédrot, Daniel R. Grayson, Kazuhiko Sakaguchi, Michael Soegtrop, Matthieu Sozeau, Paul Steckler, Enrico Tassi, Laurent Théry, Anton Trunov, whitequark, Théo Winterhalter, Zeimer, Beta Ziliani, Théo Zimmermann.

Many power users helped to improve the design of the new features via the issue and pull request system, the Coq development mailing list or the coq-club@inria.fr mailing list. It would be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.9 is the fourth release of Coq developed on a time-based development cycle. Its development spanned 7 months from the release of Coq 8.8. The development moved to a decentralized merging process during this cycle. Guillaume Melquiond was in charge of the release process and is the maintainer of this release. This release is the result of ~2,000 commits and ~500 PRs merged, closing 75+ issues.

The Coq development team welcomed Vincent Laporte, a new Coq engineer working with Maxime Dénès in the Coq consortium.

Paris, November 2018,
Matthieu Sozeau for the Coq development team

Details of changes in 8.9+beta1

Kernel

- Mutually defined records are now supported.

Notations

- New support for autonomous grammars of terms, called "custom entries" (see chapter "Syntax extensions" of the reference manual).
- Deprecated compatibility notations will actually be removed in the next version of Coq. Uses of these notations are generally easy to fix thanks to the hint contained in the deprecation warnings. For projects that require more than a handful of such fixes, there is a script[964] that will do it automatically, using the output of `coqc`. The script contains documentation on its usage in a comment at the top.

Tactics

- Added toplevel goal selector `!` which expects a single focused goal. Use with `Set Default Goal Selector` to force focusing before tactics are called.
- The undocumented "nameless" forms `fix N`, `cofix` that were deprecated in 8.8 have been removed from Ltac's syntax; please use `fix ident N`/`cofix ident` to explicitly name the (co)fixpoint hypothesis to be introduced.
- Introduction tactics `intro`/`intros` on a goal that is an existential variable now force a refinement of the goal into a dependent product rather than failing.
- Support for `fix`/`cofix` added in Ltac `match` and `lazymatch`.
- Ltac backtraces now include trace information about tactics called by OCaml-defined tactics.
- Option `Ltac Debug` now applies also to terms built using Ltac functions.
- Deprecated the `Implicit Tactic` family of commands.
- The default program obligation tactic uses a bounded proof search instead of an unbounded and potentially non-terminating one now (source of incompatibility).
- The `simple apply` tactic now respects the `Opaque` flag when called from Ltac (`auto` still does not respect it).
- Tactic `constr_eq` now adds universe constraints needed for the identity to the context (it used to ignore them). New tactic `constr_eq_strict` checks that the required constraints already hold without adding new ones. Preexisting tactic `constr_eq_nounivs` can still be used if you really want to ignore universe constraints.
- Tactics and tactic notations now understand the `deprecated` attribute.
- The `fourier` tactic has been removed. Please now use `lra` instead. You may need to add `Require Import Lra` to your developments. For compatibility, we now define `fourier` as a deprecated alias of `lra`.
- The `romega` tactics have been deprecated; please use `lia` instead.

Focusing

- Focusing bracket `{` now supports named goal selectors, e.g. `[x]: {` will focus on a goal (existential variable) named `x`. As usual, unfocus with `}` once the subgoal is fully solved.

Specification language

- A fix to unification (which was sensitive to the ascii name of variables) may occasionally change type inference in incompatible ways, especially regarding the inference of the return clause of `match`.

Standard Library

[964] https://gist.github.com/JasonGross/9770653967de3679d131c59d42de6d17#file-replace-notations-py

- Added `Ascii.eqb` and `String.eqb` and the `=?` notation for them, and proved some lemmas about them. Note that this might cause incompatibilities if you have, e.g., `string_scope` and `Z_scope` both open with `string_scope` on top, and expect `=?` to refer to `Z.eqb`. Solution: wrap `_ =? _` in `(_ =? _)%Z` (or whichever scope you want).
- Added `Ndigits.N2Bv_sized`, and proved some lemmas about it. Deprecated `Ndigits.N2Bv_gen`.
- The scopes `int_scope` and `uint_scope` have been renamed to `dec_int_scope` and `dec_uint_scope`, to clash less with ssreflect and other packages. They are still delimited by `%int` and `%uint`.
- Syntax notations for `string`, `ascii`, `Z`, `positive`, `N`, `R`, and `int31` are no longer available merely by *Require*ing the files that define the inductives. You must *Import* `Coq.Strings.String.StringSyntax` (after `Require Coq.Strings.String`), `Coq.Strings.Ascii.AsciiSyntax` (after `Require Coq.Strings.Ascii`), `Coq.ZArith.BinIntDef`, `Coq.PArith.BinPosDef`, `Coq.NArith.BinNatDef`, `Coq.Reals.Rdefinitions`, and `Coq.Numbers.Cyclic.Int31.Int31`, respectively, to be able to use these notations. Note that passing `-compat 8.8` or issuing `Require Import Coq.Compat.Coq88` will make these notations available. Users wishing to port their developments automatically may download `fix.py` from https://gist.github.com/JasonGross/5d4558edf8f5c2c548a3d96c17820169 and run a command like `while true; do make -Okj 2>&1 | /path/to/fix.py; done` and get a cup of coffee. (This command must be manually interrupted once the build finishes all the way though. Note also that this method is not fail-proof; you may have to adjust some scopes if you were relying on string notations not being available even when `string_scope` was open.)
- Numeral syntax for `nat` is no longer available without loading the entire prelude (`Require Import Coq.Init.Prelude`). This only impacts users running Coq without the init library (`-nois` or `-noinit`) and also issuing `Require Import Coq.Init.Datatypes`.

Tools

- Coq_makefile lets one override or extend the following variables from the command line: `COQFLAGS`, `COQCHKFLAGS`, `COQDOCFLAGS`. `COQFLAGS` is now entirely separate from `COQLIBS`, so in custom Makefiles `$(COQFLAGS)` should be replaced by `$(COQFLAGS) $(COQLIBS)`.
- Removed the `gallina` utility (extracts specification from Coq vernacular files). If you would like to maintain this tool externally, please contact us.
- Removed the Emacs modes distributed with Coq. You are advised to use Proof-General[965] (and optionally Company-Coq[966]) instead. If your use case is not covered by these alternative Emacs modes, please open an issue. We can help set up external maintenance as part of Proof-General, or independently as part of coq-community.

Commands

- Removed deprecated commands `Arguments Scope` and `Implicit Arguments` (not the option). Use the `Arguments` command instead.
- Nested proofs may be enabled through the option `Nested Proofs Allowed`. By default, they are disabled and produce an error. The deprecation warning which used to occur when using nested proofs has been removed.
- Added option `Uniform Inductive Parameters` which abstracts over parameters before typechecking constructors, allowing to write for example `Inductive list (A : Type) := nil : list | cons : A -> list -> list.`
- New `Set Hint Variables/Constants Opaque/Transparent` commands for setting globally the opacity flag of variables and constants in hint databases, overriding the opacity setting of the hint database.
- Added generic syntax for "attributes", as in: `#[local] Lemma foo : bar.`
- Added the `Numeral Notation` command for registering decimal numeral notations for custom types

[965] https://proofgeneral.github.io/
[966] https://github.com/cpitclaudel/company-coq

- The `Set SsrHave NoTCResolution` command no longer has special global scope. If you want the previous behavior, use `Global Set SsrHave NoTCResolution`.
- Multiple sections with the same name are allowed.

Coq binaries and process model

- Before 8.9, Coq distributed a single `coqtop` binary and a set of dynamically loadable plugins that used to take over the main loop for tasks such as IDE language server or parallel proof checking.

  These plugins have been turned into full-fledged binaries so each different process has associated a particular binary now, in particular `coqidetop` is the CoqIDE language server, and `coq{proof,tactic,query}worker` are in charge of task-specific and parallel proof checking.

SSReflect

- The implementation of delayed clear switches in intro patterns is now simpler to explain:

  1. The immediate effect of a clear switch like `{x}` is to rename the variable `x` to `_x_` (i.e. a reserved identifier that cannot be mentioned explicitly)
  2. The delayed effect of `{x}` is that `_x_` is cleared at the end of the intro pattern
  3. A clear switch immediately before a view application like `{x}/v` is translated to `/v{x}`.

  In particular, the third rule lets one write `{x}/v` even if `v` uses the variable `x`: indeed the view is executed before the renaming.
- An empty clear switch is now accepted in intro patterns before a view application whenever the view is a variable. One can now write `{}/v` to mean `{v}/v`. Remark that `{}/x` is very similar to the idiom `{}e` for the rewrite tactic (the equation `e` is used for rewriting and then discarded).

Standard Library

- There are now conversions between `string` and `positive`, `Z`, `nat`, and `N` in binary, octal, and hex.

Display diffs between proof steps

- `coqtop` and `coqide` can now highlight the differences between proof steps in color. This can be enabled from the command line or the `Set Diffs "on"/"off"/"removed"` command. Please see the documentation for details. Showing diffs in Proof General requires small changes to PG (under discussion).

Notations

- Added `++` infix for `VectorDef.append`. Note that this might cause incompatibilities if you have, e.g., `list_scope` and `vector_scope` both open with `vector_scope` on top, and expect `++` to refer to `app`. Solution: wrap `_ ++ _` in `(_ ++ _)%list` (or whichever scope you want).

Changes in 8.8.0

Various bug fixes.

Changes in 8.8.1

- Some quality-of-life fixes.
- Numerous improvements to the documentation.
- Fix a critical bug related to primitive projections and *native_compute*.
- Ship several additional Coq libraries with the Windows installer.

Version 8.8

Summary of changes

Coq version 8.8 contains the result of refinements and stabilization of features and deprecations, cleanups of the internals of the system along with a few new features. The main user visible changes are:

- Kernel: fix a subject reduction failure due to allowing fixpoints on non-recursive values, by Matthieu Sozeau. Handling of evars in the VM (the kernel still does not accept evars) by Pierre-Marie Pédrot.
- Notations: many improvements on recursive notations and support for destructuring patterns in the syntax of notations by Hugo Herbelin.
- Proof language: tacticals for profiling, timing and checking success or failure of tactics by Jason Gross. The focusing bracket `{` supports single-numbered goal selectors, e.g. `2:{`, by Théo Zimmermann.
- Vernacular: deprecation of commands and more uniform handling of the `Local` flag, by Vincent Laporte and Maxime Dénès, part of a larger attribute system overhaul. Experimental `Show Extraction` command by Pierre Letouzey. Coercion now accepts `Prop` or `Type` as a source by Arthur Charguéraud. `Export` modifier for options allowing to export the option to modules that `Import` and not only `Require` a module, by Pierre-Marie Pédrot.
- Universes: many user-level and API level enhancements: qualified naming and printing, variance annotations for cumulative inductive types, more general constraints and enhancements of the minimization heuristics, interaction with modules by Gaëtan Gilbert, Pierre-Marie Pédrot and Matthieu Sozeau.
- Library: Decimal Numbers library by Pierre Letouzey and various small improvements.
- Documentation: a large community effort resulted in the migration of the reference manual to the Sphinx documentation tool. The result is this manual. The new documentation infrastructure (based on Sphinx) is by Clément Pit-Claudel. The migration was coordinated by Maxime Dénès and Paul Steckler, with some help of Théo Zimmermann during the final integration phase. The 14 people who ported the manual are Calvin Beck, Heiko Becker, Yves Bertot, Maxime Dénès, Richard Ford, Pierre Letouzey, Assia Mahboubi, Clément Pit-Claudel, Laurence Rideau, Matthieu Sozeau, Paul Steckler, Enrico Tassi, Laurent Théry, Nikita Zyuzin.
- Tools: experimental `-mangle-names` option to `coqtop`/`coqc` for linting proof scripts, by Jasper Hugunin.

On the implementation side, the `dev/doc/changes.md` file documents the numerous changes to the implementation and improvements of interfaces. The file provides guidelines on porting a plugin to the new version.

Version 8.8 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. Most important ones are documented in the next subsection file.

The efficiency of the whole system has seen improvements thanks to contributions from Gaëtan Gilbert, Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

The official wiki and the bugtracker of Coq migrated to the GitHub platform, thanks to the work of Pierre Letouzey and Théo Zimmermann. Gaëtan Gilbert, Emilio Jesús Gallego Arias worked on maintaining and improving the continuous integration system.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

The 44 contributors for this version are Yves Bertot, Joachim Breitner, Tej Chajed, Arthur Charguéraud, Jacques-Pascal Deplaix, Maxime Dénès, Jim Fehrle, Julien Forest, Yannick Forster, Gaëtan Gilbert, Jason Gross, Samuel Gruetter, Thomas Hebb, Hugo Herbelin, Jasper Hugunin, Emilio Jesus Gallego Arias, Ralf Jung, Johannes Kloos, Matej Košík, Robbert Krebbers, Tony Beta Lambda, Vincent Laporte, Peter LeFanu Lumsdaine, Pierre Letouzey, Farzon Lotfi, Cyprien Mangin, Guillaume Melquiond, Raphaël Monat, Carl Patenaude Poulin, Pierre-Marie Pédrot, Clément Pit-Claudel, Matthew Ryan, Matt Quinn, Sigurd Schneider, Bernhard Schommer, Michael Soegtrop, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, Enrico Tassi, Anton Trunov, Martin Vassor, Vadim Zaliva and Théo Zimmermann.

Version 8.8 is the third release of Coq developed on a time-based development cycle. Its development spanned 6 months from the release of Coq 8.7 and was based on a public roadmap. The development process was coordinated by Matthieu Sozeau. Maxime Dénès was in charge of the release process. Théo Zimmermann is the maintainer of this release.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the coq-club@inria.fr mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Jason Gross, Ralf Jung, Robbert Krebbers and Amin Timany. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

The Coq consortium, an organization directed towards users and supporters of the system, is now running and employs Maxime Dénès. The contacts of the Coq Consortium are Yves Bertot and Maxime Dénès.

Santiago de Chile, March 2018,
Matthieu Sozeau for the Coq development team

Details of changes in 8.8+beta1

Kernel

- Support for template polymorphism for definitions was removed. May trigger more ”universe inconsistency” errors in rare occasions.
- Fixpoints are no longer allowed on non-recursive inductive types.

Notations

- Recursive notations with the recursive pattern repeating on the right (e.g. ”(x ; .. ; y ; z)”) now supported.
- Notations with a specific level for the leftmost nonterminal, when printing-only, are supported.
- Notations can now refer to the syntactic category of patterns (as in ”fun ’pat =>” or ”match p with pat => ... end”). Two variants are available, depending on whether a single variable is considered as a pattern or not.
- Recursive notations now support ”..” patterns with several occurrences of the recursive term or binder, possibly mixing terms and binders, possibly in reverse left-to-right order.
- ”Locate” now working also on notations of the form ”x + y” (rather than ”_ + _”).

Specification language

- When printing clauses of a ”match”, clauses with same right-hand side are factorized and the last most factorized clause with no variables, if it exists, is turned into a default clause. Use ”Unset Printing Allow Default Clause” do deactivate printing of a default clause. Use ”Unset Printing Factorizable Match Patterns” to deactivate factorization of clauses with same right-hand side.

Tactics

- On Linux, ”native_compute” calls can be profiled using the ”perf” utility. The command ”Set NativeCompute Profiling” enables profiling, and ”Set NativeCompute Profile Filename” customizes the profile filename.
- The tactic ”omega” is now aware of the bodies of context variables such as ”x := 5 : Z” (see #1362). This could be disabled via Unset Omega UseLocalDefs.
- The tactic ”romega” is also aware now of the bodies of context variables.
- The tactic ”zify” resp. ”omega with N” is now aware of N.pred.
- Tactic ”decide equality” now able to manage constructors which contain proofs.
- Added tactics reset ltac profile, show ltac profile (and variants)
- Added tactics restart_timer, finish_timing, and time_constr as an experimental way of timing Ltac's evaluation phase
- Added tactic optimize_heap, analogous to the Vernacular Optimize Heap, which performs a major garbage collection and heap compaction in the OCaml run-time system.
- The tactics ”dtauto”, ”dintuition”, ”firstorder” now handle inductive types with let bindings in the parameters.
- The tactic `dtauto` now handles some inductives such as `@sigT A (fun _ => B)` as non-dependent conjunctions.
- A bug fixed in `rewrite H in *` and `rewrite H in * |-` may cause a few rare incompatibilities (it was unintendedly recursively rewriting in the side conditions generated by H).
- Added tactics ”assert_succeeds tac” and ”assert_fails tac” to ensure properties of the execution of a tactic without keeping the effect of the execution.
- `vm_compute` now supports existential variables.
- Calls to `shelve` and `give_up` within calls to tactic `refine` now working.
- Deprecated tactic `appcontext` was removed.

Focusing

- Focusing bracket `{` now supports single-numbered goal selector, e.g. `2: {` will focus on the second subgoal. As usual, unfocus with `}` once the subgoal is fully solved. The `Focus` and `Unfocus` commands are now deprecated.

Commands

- Proofs ending in ”Qed exporting ident, .., ident” are not supported anymore. Constants generated during `abstract` are kept private to the local environment.
- The deprecated Coercion Local, Open Local Scope, Notation Local syntax was removed. Use Local as a prefix instead.
- For the Extraction Language command, ”OCaml” is spelled correctly. The older ”Ocaml” is still accepted, but deprecated.
- Using “Require” inside a section is deprecated.
- An experimental command ”Show Extraction” allows to extract the content of the current ongoing proof (grant wish #4129).
- Coercion now accepts the type of its argument to be ”Prop” or ”Type”.
- The ”Export” modifier can now be used when setting and unsetting options, and will result in performing the same change when the module corresponding the command is imported.
- The `Axiom` command does not automatically declare axioms as instances when their type is a class. Previous behavior can be restored using `Set Typeclasses Axioms Are Instances`.

Universes

- Qualified naming of global universes now works like other namespaced objects (e.g. constants), with a separate namespace, inside and across module and library boundaries. Global universe names introduced in an inductive / constant / Let declaration get qualified with the name of the declaration.
- Universe cumulativity for inductive types is now specified as a variance for each polymorphic universe. See the reference manual for more information.
- Inference of universe constraints with cumulative inductive types produces more general constraints. Unsetting new option Cumulativity Weak Constraints produces even more general constraints (but may produce too many universes to be practical).
- Fix #5726: Notations that start with `Type` now support universe instances with `@{u}`.
- `with Definition` now understands universe declarations (like `@{u| Set < u}`).

Tools

- Coq can now be run with the option -mangle-names to change the auto-generated name scheme. This is intended to function as a linter for developments that want to be robust to changes in auto-generated names. This feature is experimental, and may change or disappear without warning.
- GeoProof support was removed.

Checker

- The checker now accepts filenames in addition to logical paths.

CoqIDE

- Find and Replace All report the number of occurrences found; Find indicates when it wraps.

coqdep

- Learned to read -I, -Q, -R and filenames from _CoqProject files. This is used by coq_makefile when generating dependencies for .v files (but not other files).

Documentation

- The Coq FAQ, formerly located at https://coq.inria.fr/faq, has been moved to the GitHub wiki section of this repository; the main entry page is https://github.com/coq/coq/wiki/The-Coq-FAQ.
- Documentation: a large community effort resulted in the migration of the reference manual to the Sphinx documentation tool. The result is partially integrated in this version.

Standard Library

- New libraries Coq.Init.Decimal, Coq.Numbers.DecimalFacts, Coq.Numbers.DecimalNat, Coq.Numbers.DecimalPos, Coq.Numbers.DecimalN, Coq.Numbers.DecimalZ, Coq.Numbers.DecimalString providing a type of decimal numbers, some facts about them, and conversions between decimal numbers and nat, positive, N, Z, and string.
- Added [Coq.Strings.String.concat] to concatenate a list of strings inserting a separator between each item
- Notation `'` for Zpos in QArith was removed.
- Some deprecated aliases are now emitting warnings when used.

Compatibility support

- Support for compatibility with versions before 8.6 was dropped.

Options

- The following deprecated options have been removed:
  - `Refolding Reduction`

- Standard Proposition Elimination
- Dependent Propositions Elimination
- Discriminate Introduction
- Shrink Abstract
- Tactic Pattern Unification
- Intuition Iff Unfolding
- Injection L2R Pattern Order
- Record Elimination Schemes
- Match Strict
- Tactic Compat Context
- Typeclasses Legacy Resolution
- Typeclasses Module Eta
- Typeclass Resolution After Apply

Details of changes in 8.8.0

Tools

- Asynchronous proof delegation policy was fixed. Since version 8.7 Coq was ignoring previous runs and the `-async-proofs-delegation-threshold` option did not have the expected behavior.

Tactic language

- The undocumented "nameless" forms `fix N`, `cofix` have been deprecated; please use `fix ident N` / `cofix ident` to explicitly name the (co)fixpoint hypothesis to be introduced.

Documentation

- The reference manual is now fully ported to Sphinx.

Other small deprecations and bug fixes.

Details of changes in 8.8.1

Kernel

- Fix a critical bug with cofixpoints and `vm_compute`/`native_compute` (#7333).
- Fix a critical bug with modules and algebraic universes (#7695)
- Fix a critical bug with inlining of polymorphic constants (#7615).
- Fix a critical bug with universe polymorphism and `vm_compute` (#7723). Was present since 8.5.

Notations

- Fixed unexpected collision between only-parsing and only-printing notations (issue #7462).

Windows installer

- The Windows installer now includes external packages Ltac2 and Equations (it included the Bignums package since 8.8+beta1).

Many other bug fixes, documentation improvements (including fixes of regressions due to the Sphinx migration), and user message improvements (for details, see the 8.8.1 milestone at https://github.com/coq/coq/milestone/13?closed=1).

Details of changes in 8.8.2

Documentation

- A PDF version of the reference manual is available once again.

Tools

- The coq-makefile targets `print-pretty-timed`, `print-pretty-timed-diff`, and `print-pretty-single-time-diff` now correctly label the "before" and "after" columns, rather than swapping them.

Kernel

- The kernel does not tolerate capture of global universes by polymorphic universe binders, fixing a soundness break (triggered only through custom plugins)

Windows installer

- The Windows installer now includes many more external packages that can be individually selected for installation.

Many other bug fixes and lots of documentation improvements (for details, see the 8.8.2 milestone at https://github.com/coq/coq/milestone/15?closed=1).

Version 8.7

Summary of changes

Coq version 8.7 contains the result of refinements, stabilization of features and cleanups of the internals of the system along with a few new features. The main user visible changes are:

- New tactics: variants of tactics supporting existential variables *`eassert`*, *`eenough`*, etc... by Hugo Herbelin. Tactics `extensionality in H` and *`inversion_sigma`* by Jason Gross, `specialize with ...` accepting partial bindings by Pierre Courtieu.
- `Cumulative Polymorphic Inductive` types, allowing cumulativity of universes to go through applied inductive types, by Amin Timany and Matthieu Sozeau.
- Integration of the SSReflect plugin and its documentation in the reference manual, by Enrico Tassi, Assia Mahboubi and Maxime Dénès.
- The `coq_makefile` tool was completely redesigned to improve its maintainability and the extensibility of generated Makefiles, and to make `_CoqProject` files more palatable to IDEs by Enrico Tassi.

Coq 8.7 involved a large amount of work on cleaning and speeding up the code base, notably the work of Pierre-Marie Pédrot on making the tactic-level system insensitive to existential variable expansion, providing a safer API to plugin writers and making the code more robust. The `dev/doc/changes.txt` file documents the numerous changes to the implementation and improvements of interfaces. An effort to provide an official, streamlined API to plugin writers is in progress, thanks to the work of Matej Košík.

Version 8.7 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

Thomas Sibut-Pinote and Hugo Herbelin added support for side effect hooks in cbv, cbn and simpl. The side effects are provided via a plugin available at https://github.com/herbelin/reduction-effects/.

The BigN, BigZ, BigQ libraries are no longer part of the Coq standard library, they are now provided by a separate repository https://github.com/coq/bignums, maintained by Pierre Letouzey.

In the Reals library, `IZR` has been changed to produce a compact representation of integers and real constants are now represented using `IZR` (work by Guillaume Melquiond).

Standard library additions and improvements by Jason Gross, Pierre Letouzey and others, documented in the next subsection file.

The mathematical proof language/declarative mode plugin was removed from the archive.

The OPAM repository for Coq packages has been maintained by Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi with contributions from many users. A list of packages is available at https://coq.inria.fr/opam/www/.

Packaging tools and software development kits were prepared by Michael Soegtrop with the help of Maxime Dénès and Enrico Tassi for Windows, and Maxime Dénès for MacOS X. Packages are regularly built on the Travis continuous integration server.

The contributors for this version are Abhishek Anand, C.J. Bell, Yves Bertot, Frédéric Besson, Tej Chajed, Pierre Courtieu, Maxime Dénès, Julien Forest, Gaëtan Gilbert, Jason Gross, Hugo Herbelin, Emilio Jesús Gallego Arias, Ralf Jung, Matej Košík, Xavier Leroy, Pierre Letouzey, Assia Mahboubi, Cyprien Mangin, Erik Martin-Dorel, Olivier Marty, Guillaume Melquiond, Sam Pablo Kuper, Benjamin Pierce, Pierre-Marie Pédrot, Lars Rasmusson, Lionel Rieg, Valentin Robert, Yann Régis-Gianas, Thomas Sibut-Pinote, Michael Soegtrop, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, George Stelle, Pierre-Yves Strub, Enrico Tassi, Hendrik Tews, Amin Timany, Laurent Théry, Vadim Zaliva and Théo Zimmermann.

The development process was coordinated by Matthieu Sozeau with the help of Maxime Dénès, who was also in charge of the release process. Théo Zimmermann is the maintainer of this release.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the Coq-Club mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Jason Gross, Ralf Jung, Robbert Krebbers, Xavier Leroy, Clément Pit–Claudel and Gabriel Scherer. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.7 is the second release of Coq developed on a time-based development cycle. Its development spanned 9 months from the release of Coq 8.6 and was based on a public road-map. It attracted many external contributions. Code reviews and continuous integration testing were systematically used before integration of new features, with an important focus given to compatibility and performance issues, resulting in a hopefully more robust release than Coq 8.6 while maintaining compatibility.

Coq Enhancement Proposals (CEPs for short) and open pull request discussions were used to discuss publicly the new features.

The Coq consortium, an organization directed towards users and supporters of the system, is now upcoming and will rely on Inria's newly created Foundation.

Paris, August 2017,
Matthieu Sozeau and the Coq development team

Potential compatibility issues

- Extra superfluous names in introduction patterns may now raise an error rather than a warning when the superfluous name is already in use. The easy fix is to remove the superfluous name.

Details of changes in 8.7+beta1

Tactics

- New tactic "extensionality in H" which applies (possibly dependent) functional extensionality in H supposed to be a quantified equality until giving a bare equality.
- New tactic `inversion_sigma` which turns equalities of dependent pairs (e.g., `existT P x p = existT P y q`, frequently left over by `inversion` on a dependent type family) into pairs of equalities (e.g., a hypothesis `H : x = y` and a hypothesis of type `rew H in p = q`); these hypotheses can subsequently be simplified using `subst`, without ever invoking any kind of axiom asserting uniqueness of identity proofs. If you want to explicitly specify the hypothesis to be inverted, or name the generated hypotheses, you can invoke `induction H as [H1 H2] using eq_sigT_rect`. The tactic also works for `sig`, `sigT2`, and `sig2`, and there are similar `eq_sig*_rect` induction lemmas.
- Tactic "specialize with ..." now accepts any partial bindings. Missing bindings are either solved by unification or left quantified in the hypothesis.
- New representation of terms that statically ensure stability by evar-expansion. This has several consequences.
  - In terms of performance, this adds a cost to every term destructuration, but at the same time most eager evar normalizations were removed, which couterbalances this drawback and even sometimes outperforms the old implementation. For instance, many operations that would require O(n) normalization of the term are now O(1) in tactics. YMMV.
  - This triggers small changes in unification, which was not evar-insensitive. Most notably, the new implementation recognizes Miller patterns that were missed before because of a missing normalization step. Hopefully this should be fairly uncommon.
- Tactic "auto with real" can now discharge comparisons of literals.
- The types of variables in patterns of "match" are now beta-iota-reduced after type checking. This has an impact on the type of the variables that the tactic "refine" introduces in the context, producing types that should be closer to the expectations.
- In "Tactic Notation" or "TACTIC EXTEND", entry "constr_with_bindings" now uses type classes and rejects terms with unresolved holes, like entry "constr" does. To get the former behavior use "open_constr_with_bindings" (possible source of incompatibility).
- New e-variants eassert, eenough, epose proof, eset, eremember, epose which behave like the corresponding variants with no "e" but turn unresolved implicit arguments into existential variables, on the shelf, rather than failing.
- Tactic injection has become more powerful (closes bug #4890) and its documentation has been updated.
- New variants of the `first` and `solve` tacticals that do not rely on parsing rules, meant to define tactic notations.
- Added support for side effects hooks in `cbv`, `cbn` and `simpl`. The side effects are provided via a plugin: https://github.com/herbelin/reduction-effects/
- It is now possible to take hint database names as parameters in a Ltac definition or a Tactic Notation.
- New option `Set Ltac Batch Debug` on top of `Set Ltac Debug` for non-interactive Ltac debug output.

Gallina

- Now supporting all kinds of binders, including 'pat, in syntax of record fields.

Commands

- Goals context can be printed in a more compact way when `Set Printing Compact Contexts` is activated.
- Unfocused goals can be printed with the `Set Printing Unfocused` option.
- `Print` now shows the types of let-bindings.
- The compatibility options for printing primitive projections (`Set Printing Primitive Projection Parameters` and `Set Printing Primitive Projection Compatibility`) are now off by default.
- Possibility to unset the printing of notations in a more fine grained fashion than `Unset Printing Notations` is provided without any user-syntax. The goal is that someone creates a plugin to experiment such a user-syntax, to be later integrated in Coq when stabilized.
- `About` now tells if a reference is a coercion.
- The deprecated `Save` vernacular and its form `Save Theorem id` to close proofs have been removed from the syntax. Please use `Qed`.
- `Search` now sorts results by relevance (the relevance metric is a weighted sum of number of distinct symbols and size of the term).

Standard Library

- New file PropExtensionality.v to explicitly work in the axiomatic context of propositional extensionality.
- New file SetoidChoice.v axiomatically providing choice over setoids, and, consequently, choice of representatives in equivalence classes. Various proof-theoretic characterizations of choice over setoids in file ChoiceFacts.v.
- New lemmas about iff and about orders on positive and Z.
- New lemmas on powerRZ.
- Strengthened statement of JMeq_eq_dep (closes bug #4912).
- The BigN, BigZ, BigZ libraries are no longer part of the Coq standard library, they are now provided by a separate repository https://github.com/coq/bignums The split has been done just after the Int31 library.
- IZR (Reals) has been changed to produce a compact representation of integers. As a consequence, IZR is no longer convertible to INR and lemmas such as INR_IZR_INZ should be used instead.
- Real constants are now represented using IZR rather than R0 and R1; this might cause rewriting rules to fail to apply to constants.
- Added new notation {x & P} for sigT (without a type for x)

Plugins

- The Ssreflect plugin is now distributed with Coq. Its documentation has been integrated as a chapter of the reference manual. This chapter is work in progress so feedback is welcome.
- The mathematical proof language (also known as declarative mode) was removed.
- A new command Extraction TestCompile has been introduced, not meant for the general user but instead for Coq's test-suite.
- The extraction plugin is no longer loaded by default. It must be explicitly loaded with [Require Extraction], which is backwards compatible.
- The functional induction plugin (which provides the [Function] vernacular) is no longer loaded by default. It must be explicitly loaded with [Require FunInd], which is backwards compatible.

Dependencies

- Support for camlp4 has been removed.

Tools

- coq_makefile was completely redesigned to improve its maintainability and the extensibility of generated Makefiles, and to make _CoqProject files more palatable to IDEs. Overview:
  - _CoqProject files contain only Coq specific data (i.e. the list of files, -R options, ...)
  - coq_makefile translates _CoqProject to Makefile.conf and copies in the desired location a standard Makefile (that reads Makefile.conf)
  - Makefile extensions can be implemented in a Makefile.local file (read by the main Makefile) by installing a hook in the extension points provided by the standard Makefile

  The current version contains code for retro compatibility that prints warnings when a deprecated feature is used. Please upgrade your _CoqProject accordingly.
  - Additionally, coq_makefile-made Makefiles now support experimental timing targets `pretty-timed`, `pretty-timed-before`, `pretty-timed-after`, `print-pretty-timed-diff`, `print-pretty-single-time-diff`, `all.timing.diff`, and the variable `TIMING=1` (or `TIMING=before` or `TIMING=after`); see the documentation for more details.

Build Infrastructure

- Note that 'make world' does not build the bytecode binaries anymore. For that, you can use 'make byte' (and 'make install-byte' afterwards). Warning: native and byte compilations should *not* be mixed in the same instance of 'make -j', otherwise both ocamlc and ocamlopt might race for access to the same .cmi files. In short, use "make -j && make -j byte" instead of "make -j world byte".

Universes

- Cumulative inductive types. see prefixes "Cumulative", "NonCumulative" for inductive definitions and the option "Set Polymorphic Inductive Cumulativity" in the reference manual.
- New syntax `foo@{_}` to instantiate a polymorphic definition with anonymous universes (can also be used with `Type`).

XML Protocol and internal changes

See dev/doc/changes.txt

Many bugfixes including #1859, #2884, #3613, #3943, #3994, #4250, #4709, #4720, #4824, #4844, #4911, #5026, #5233, #5275, #5315, #5336, #5360, #5390, #5414, #5417, #5420, #5439, #5449, #5475, #5476, #5482, #5501, #5507, #5520, #5523, #5524, #5553, #5577, #5578, #5589, #5597, #5598, #5607, #5618, #5619, #5620, #5641, #5648, #5651, #5671.

Many bugfixes on OS X and Windows (now the test-suite passes on these platforms too).

Many optimizations.

Many documentation improvements.

Details of changes in 8.7+beta2

Tools

- In CoqIDE, the "Compile Buffer" command takes account of flags in _CoqProject or other project file.

Improvements around some error messages.

Many bug fixes including two important ones:

- Bug #5730: CoqIDE becomes unresponsive on file open.

- coq_makefile: make sure compile flags for Coq and coq_makefile are in sync (in particular, make sure the `-safe-string` option is used to compile plugins).

Details of changes in 8.7.0

OCaml

- Users can pass specific flags to the OCaml optimizing compiler by -using the flambda-opts configure-time option.

  Beware that compiling Coq with a flambda-enabled compiler is experimental and may require large amounts of RAM and CPU, see INSTALL for more details.

Details of changes in 8.7.1

Compatibility with OCaml 4.06.0.

Many bug fixes, documentation improvements, and user message improvements (for details see the 8.7.1 milestone at https://github.com/coq/coq/milestone/10?closed=1).

Details of changes in 8.7.2

Fixed a critical bug in the VM handling of universes (#6677). This bug affected all releases since 8.5.

Improved support for building with OCaml 4.06.0 and external num package.

Many other bug fixes, documentation improvements, and user message improvements (for details, see the 8.7.2 milestone at https://github.com/coq/coq/milestone/11?closed=1).

Version 8.6

Summary of changes

Coq version 8.6 contains the result of refinements, stabilization of 8.5's features and cleanups of the internals of the system. Over the year of (now time-based) development, about 450 bugs were resolved and over 100 contributions integrated. The main user visible changes are:

- A new, faster state-of-the-art universe constraint checker, by Jacques-Henri Jourdan.
- In CoqIDE and other asynchronous interfaces, more fine-grained asynchronous processing and error reporting by Enrico Tassi, making Coq capable of recovering from errors and continue processing the document.
- More access to the proof engine features from Ltac: goal management primitives, range selectors and a `typeclasses eauto` engine handling multiple goals and multiple successes, by Cyprien Mangin, Matthieu Sozeau and Arnaud Spiwack.
- Tactic behavior uniformization and specification, generalization of intro-patterns by Hugo Herbelin and others.
- A brand new warning system allowing to control warnings, turn them into errors or ignore them selectively by Maxime Dénès, Guillaume Melquiond, Pierre-Marie Pédrot and others.
- Irrefutable patterns in abstractions, by Daniel de Rauglaudre.
- The ssreflect subterm selection algorithm by Georges Gonthier and Enrico Tassi is now accessible to tactic writers through the ssrmatching plugin.
- Integration of LtacProf, a profiler for Ltac by Jason Gross, Paul Steckler, Enrico Tassi and Tobias Tebbi.

Coq 8.6 also comes with a bunch of smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

The iota reduction flag is now a shorthand for match, fix and cofix flags controlling the corresponding reduction rules (by Hugo Herbelin and Maxime Dénès).

Maxime Dénès maintained the native compilation machinery.

Pierre-Marie Pédrot separated the Ltac code from general purpose tactics, and generalized and rationalized the handling of generic arguments, allowing to create new versions of Ltac more easily in the future.

In patterns and terms, @, abbreviations and notations are now interpreted the same way, by Hugo Herbelin.

Name handling for universes has been improved by Pierre-Marie Pédrot and Matthieu Sozeau. The minimization algorithm has been improved by Matthieu Sozeau.

The unifier has been improved by Hugo Herbelin and Matthieu Sozeau, fixing some incompatibilities introduced in Coq 8.5. Unification constraints can now be left floating around and be seen by the user thanks to a new option. The Keyed Unification mode has been improved by Matthieu Sozeau.

The typeclass resolution engine and associated proof search tactic have been reimplemented on top of the proof-engine monad, providing better integration in tactics, and new options have been introduced to control it, by Matthieu Sozeau with help from Théo Zimmermann.

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot, Maxime Dénès and Matthieu Sozeau and performance issue tracking by Jason Gross and Paul Steckler.

Standard library improvements by Jason Gross, Sébastien Hinderer, Pierre Letouzey and others.

Emilio Jesús Gallego Arias contributed many cleanups and refactorings of the pretty-printing and user interface communication components.

Frédéric Besson maintained the micromega tactic.

The OPAM repository for Coq packages has been maintained by Guillaume Claret, Guillaume Melquiond, Matthieu Sozeau, Enrico Tassi and others. A list of packages is now available at https://coq.inria.fr/opam/www/.

Packaging tools and software development kits were prepared by Michael Soegtrop with the help of Maxime Dénès and Enrico Tassi for Windows, and Maxime Dénès and Matthieu Sozeau for MacOS X. Packages are now regularly built on the continuous integration server. Coq now comes with a META file usable with ocamlfind, contributed by Emilio Jesús Gallego Arias, Gregory Malecha, and Matthieu Sozeau.

Matej Košík maintained and greatly improved the continuous integration setup and the testing of Coq contributions. He also contributed many API improvements and code cleanups throughout the system.

The contributors for this version are Bruno Barras, C.J. Bell, Yves Bertot, Frédéric Besson, Pierre Boutillier, Tej Chajed, Guillaume Claret, Xavier Clerc, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Ricky Elrod, Emilio Jesús Gallego Arias, Jason Gross, Hugo Herbelin, Sébastien Hinderer, Jacques-Henri Jourdan, Matej Košík, Xavier Leroy, Pierre Letouzey, Gregory Malecha, Cyprien Mangin, Erik Martin-Dorel, Guillaume Melquiond, Clément Pit–Claudel, Pierre-Marie Pédrot, Daniel de Rauglaudre, Lionel Rieg, Gabriel Scherer, Thomas Sibut-Pinote, Matthieu Sozeau, Arnaud Spiwack, Paul Steckler, Enrico Tassi, Laurent Théry, Nickolai Zeldovich and Théo Zimmermann. The development process was coordinated by Hugo Herbelin and Matthieu Sozeau with the help of Maxime Dénès, who was also in charge of the release process.

Many power users helped to improve the design of the new features via the bug tracker, the pull request system, the Coq development mailing list or the Coq-Club mailing list. Special thanks to the users who contributed patches and intensive brain-storming and code reviews, starting with Cyril Cohen, Jason Gross, Robbert Krebbers, Jonathan Leivent, Xavier Leroy, Gregory Malecha, Clément Pit–Claudel, Gabriel Scherer and Beta Ziliani. It would however be impossible to mention exhaustively the names of everybody who to some extent influenced the development.

Version 8.6 is the first release of Coq developed on a time-based development cycle. Its development spanned 10 months from the release of Coq 8.5 and was based on a public roadmap. To date, it contains more external contributions than

any previous Coq system. Code reviews were systematically done before integration of new features, with an important focus given to compatibility and performance issues, resulting in a hopefully more robust release than Coq 8.5.

Coq Enhancement Proposals (CEPs for short) were introduced by Enrico Tassi to provide more visibility and a discussion period on new features, they are publicly available https://github.com/coq/ceps.

Started during this period, an effort is led by Yves Bertot and Maxime Dénès to put together a Coq consortium.

Paris, November 2016,
Matthieu Sozeau and the Coq development team

Potential sources of incompatibilities

- Symptom: An obligation generated by Program or an abstracted subproof has different arguments.

  Cause: Set Shrink Abstract and Set Shrink Obligations are on by default and the subproof does not use the argument.

  Remedy:

  - Adapt the script.
  - Write an explicit lemma to prove the obligation/subproof and use it instead (compatible with 8.4).
  - Unset the option for the program/proof the obligation/subproof originates from.

- Symptom: In a goal, order of hypotheses, or absence of an equality of the form ”x = t” or ”t = x”, or no unfolding of a local definition.

  Cause: This might be connected to a number of fixes in the tactic ”subst”. The former behavior can be reactivated by issuing ”Unset Regular Subst Tactic”.

Details of changes in 8.6beta1

Kernel

- A new, faster state-of-the-art universe constraint checker.

Specification language

- Giving implicit arguments explicitly to a constant with multiple choices of implicit arguments does not break any more insertion of further maximal implicit arguments.
- Ability to put any pattern in binders, prefixed by quote, e.g. ”fun ’(a,b) => ...”, ”λ ’(a,(b,c)), ...”, ”Definition foo ’(x,y) := ...”. It expands into a ”let ’pattern := ...”

Tactics

- Flag ”Bracketing Last Introduction Pattern” is now on by default.
- Flag ”Regular Subst Tactic” is now on by default: it respects the initial order of hypothesis, it contracts cycles, it unfolds no local definitions (common source of incompatibilities, fixable by ”Unset Regular Subst Tactic”).
- New flag ”Refolding Reduction”, now disabled by default, which turns on refolding of constants/fixpoints (as in cbn) during the reductions done during type inference and tactic retyping. Can be extremely expensive. When set off, this recovers the 8.4 behaviour of unification and type inference. Potential source of incompatibility with 8.5 developments (the option is set on in Compat/Coq85.v).

- New flag "Shrink Abstract" that minimalizes proofs generated by the abstract tactical w.r.t. variables appearing in the body of the proof. On by default and deprecated. Minor source of incompatibility for code relying on the precise arguments of abstracted proofs.
- Serious bugs are fixed in tactic "double induction" (source of incompatibilities as soon as the inductive types have dependencies in the type of their constructors; "double induction" remains however deprecated).
- In introduction patterns of the form (pat1,...,patn), n should match the exact number of hypotheses introduced (except for local definitions for which pattern can be omitted, as in regular pattern-matching).
- Tactic scopes in Ltac like constr: and ltac: now require parentheses around their argument.
- Every generic argument type declares a tactic scope of the form "name:(...)" where name is the name of the argument. This generalizes the constr: and ltac: instances.
- When in strict mode (i.e. in a Ltac definition), if the "intro" tactic is given a free identifier, it is not bound in subsequent tactics anymore. In order to introduce a binding, use e.g. the "fresh" primitive instead (potential source of incompatibilities).
- New tactics is_ind, is_const, is_proj, is_constructor for use in Ltac.
- New goal selectors. Sets of goals can be selected by listing integers ranges. Example: "1,4-7,24: tac" focuses "tac" on goals 1,4,5,6,7,24.
- For uniformity with "destruct"/"induction" and for a more natural behavior, "injection" can now work in place by activating option "Structural Injection". In this case, hypotheses are also put in the context in the natural left-to-right order and the hypothesis on which injection applies is cleared.
- Tactic "contradiction" (hence "easy") now also solve goals with hypotheses of the form "~True" or "t<>t" (possible source of incompatibilities because of more successes in automation, but generally a more intuitive strategy).
- Option "Injection On Proofs" was renamed "Keep Proof Equalities". When enabled, injection and inversion do not drop equalities between objects in Prop. Still disabled by default.
- New tactics "notypeclasses refine" and "simple notypeclasses refine" that disallow typeclass resolution when typechecking their argument, for use in typeclass hints.
- Integration of LtacProf, a profiler for Ltac.
- Reduction tactics now accept more fine-grained flags: iota is now a shorthand for the new flags match, fix and cofix.
- The ssreflect subterm selection algorithm is now accessible to tactic writers through the ssrmatching plugin.
- When used as an argument of an ltac function, "auto" without "with" nor "using" clause now correctly uses only the core hint database by default.

Hints

- Revised the syntax of [Hint Cut] to follow standard notation for regexps.
- Hint Mode now accepts "!" which means that the mode matches only if the argument's head is not an evar (it goes under applications, casts, and scrutinees of matches and projections).
- Hints can now take an optional user-given pattern, used only by [typeclasses eauto] with the [Filtered Unification] option on.

Typeclasses

- Many new options and new engine based on the proof monad. The [typeclasses eauto] tactic is now a multi-goal, multi-success tactic. See reference manual for more information. It is planned to replace auto and eauto in the following version. The 8.5 resolution engine is still available to help solve compatibility issues.

Program

- The "Shrink Obligations" flag now applies to all obligations, not only those solved by the automatic tactic.

- ”Shrink Obligations” is on by default and deprecated. Minor source of incompatibility for code relying on the precise arguments of obligations.

Notations

- ”Bind Scope” can once again bind ”Funclass” and ”Sortclass”.

General infrastructure

- New configurable warning system which can be controlled with the vernacular command ”Set Warnings”, or, under coqc/coqtop, with the flag ”-w”. In particular, the default is now that warnings are printed by coqc.
- In asynchronous mode, Coq is now capable of recovering from errors and continue processing the document.

Tools

- coqc accepts a -o option to specify the output file name
- coqtop accepts --print-version to print Coq and OCaml versions in easy to parse format
- Setting [Printing Dependent Evars Line] can be unset to disable the computation associated with printing the ”dependent evars: ” line in -emacs mode
- Removed the -verbose-compat-notations flag and the corresponding Set Verbose Compat vernacular, since these warnings can now be silenced or turned into errors using ”-w”.

XML protocol

- message format has changed, see dev/doc/changes.txt for more details.

Many bug fixes, minor changes and documentation improvements are not mentioned here.

Details of changes in 8.6

Kernel

- Fixed critical bug #5248 in VM long multiplication on 32-bit architectures. Was there only since 8.6beta1, so no stable release impacted.

Other bug fixes in universes, type class shelving,...

Details of changes in 8.6.1

- Fix #5380: Default colors for CoqIDE are actually applied.
- Fix plugin warnings
- Document named evars (including Show ident)
- Fix Bug #5574, document function scope
- Adding a test case as requested in bug 5205.
- Fix Bug #5568, no dup notation warnings on repeated module imports
- Fix documentation of Typeclasses eauto :=
- Refactor documentation of records.
- Protecting from warnings while compiling 8.6
- Fixing an inconsistency between configure and configure.ml
- Add test-suite checks for coqchk with constraints

- Fix bug #5019 (looping zify on dependent types)
- Fix bug 5550: ”typeclasses eauto with” does not work with section variables.
- Bug 5546, qualify datatype constructors when needed in Show Match
- Bug #5535, test for Show with -emacs
- Fix bug #5486, don’t reverse ids in tuples
- Fixing #5522 (anomaly with free vars of pat)
- Fix bug #5526, don’t check for nonlinearity in notation if printing only
- Fix bug #5255
- Fix bug #3659: -time should understand multibyte encodings.
- FIx bug #5300: Anomaly: Uncaught exception Not_found” in ”Print Assumptions”.
- Fix outdated description in RefMan.
- Repairing `Set Rewriting Schemes`
- Fixing #5487 (v8.5 regression on ltac-matching expressions with evars).
- Fix description of command-line arguments for Add (Rec) LoadPath
- Fix bug #5377: @? patterns broken.
- add XML protocol doc
- Fix anomaly when doing [all:Check _.] during a proof.
- Correction of bug #4306
- Fix #5435: [Eval native_compute in] raises anomaly.
- Instances should obey universe binders even when defined by tactics.
- Intern names bound in match patterns
- funind: Ignore missing info for current function
- Do not typecheck twice the type of opaque constants.
- show unused intro pattern warning
- [future] Be eager when ”chaining” already resolved future values.
- Opaque side effects
- Fix #5132: coq_makefile generates incorrect install goal
- Run non-tactic comands without resilient_command
- Univs: fix bug #5365, generation of u+k <= v constraints
- make `emit` tail recursive
- Don’t require printing-only notation to be productive
- Fix the way setoid_rewrite handles bindings.
- Fix for bug 5244 - set printing width ignored when given enough space
- Fix bug 4969, autoapply was not tagging shelved subgoals correctly

Version 8.5

Summary of changes

Coq version 8.5 contains the result of five specific long-term projects:

- A new asynchronous evaluation and compilation mode by Enrico Tassi with help from Bruno Barras and Carst Tankink.
- Full integration of the new proof engine by Arnaud Spiwack helped by Pierre-Marie Pédrot,
- Addition of conversion and reduction based on native compilation by Maxime Dénès and Benjamin Grégoire.
- Full universe polymorphism for definitions and inductive types by Matthieu Sozeau.
- An implementation of primitive projections with η-conversion bringing significant performance improvements when using records by Matthieu Sozeau.

The full integration of the proof engine, by Arnaud Spiwack and Pierre-Marie Pédrot, brings to primitive tactics and the user level Ltac language dependent subgoals, deep backtracking and multiple goal handling, along with miscellaneous features and an improved potential for future modifications. Dependent subgoals allow statements in a goal to mention the proof of another. Proofs of unsolved subgoals appear as existential variables. Primitive backtracking makes it possible to write a tactic with several possible outcomes which are tried successively when subsequent tactics fail. Primitives are also available to control the backtracking behavior of tactics. Multiple goal handling paves the way for smarter automation tactics. It is currently used for simple goal manipulation such as goal reordering.

The way Coq processes a document in batch and interactive mode has been redesigned by Enrico Tassi with help from Bruno Barras. Opaque proofs, the text between Proof and Qed, can be processed asynchronously, decoupling the checking of definitions and statements from the checking of proofs. It improves the responsiveness of interactive development, since proofs can be processed in the background. Similarly, compilation of a file can be split into two phases: the first one checking only definitions and statements and the second one checking proofs. A file resulting from the first phase – with the .vio extension – can be already Required. All .vio files can be turned into complete .vo files in parallel. The same infrastructure also allows terminating tactics to be run in parallel on a set of goals via the `par:` goal selector.

CoqIDE was modified to cope with asynchronous checking of the document. Its source code was also made separate from that of Coq, so that CoqIDE no longer has a special status among user interfaces, paving the way for decoupling its release cycle from that of Coq in the future.

Carst Tankink developed a Coq back-end for user interfaces built on Makarius Wenzel's Prover IDE framework (PIDE), like PIDE/jEdit (with help from Makarius Wenzel) or PIDE/Coqoon (with help from Alexander Faithfull and Jesper Bengtson). The development of such features was funded by the Paral-ITP French ANR project.

The full universe polymorphism extension was designed by Matthieu Sozeau. It conservatively extends the universes system and core calculus with definitions and inductive declarations parameterized by universes and constraints. It is based on a modification of the kernel architecture to handle constraint checking only, leaving the generation of constraints to the refinement/type inference engine. Accordingly, tactics are now fully universe aware, resulting in more localized error messages in case of inconsistencies and allowing higher-level algorithms like unification to be entirely type safe. The internal representation of universes has been modified but this is invisible to the user.

The underlying logic has been extended with η-conversion for records defined with primitive projections by Matthieu Sozeau. This additional form of η-conversion is justified using the same principle than the previously added η-conversion for function types, based on formulations of the Calculus of Inductive Constructions with typed equality. Primitive projections, which do not carry the parameters of the record and are rigid names (not defined as a pattern matching construct), make working with nested records more manageable in terms of time and space consumption. This extension and universe polymorphism were carried out partly while Matthieu Sozeau was working at the IAS in Princeton.

The guard condition has been made compliant with extensional equality principles such as propositional extensionality and univalence, thanks to Maxime Dénès and Bruno Barras. To ensure compatibility with the univalence axiom, a new flag `-indices-matter` has been implemented, taking into account the universe levels of indices when computing the

levels of inductive types. This supports using Coq as a tool to explore the relations between homotopy theory and type theory.

Maxime Dénès and Benjamin Grégoire developed an implementation of conversion test and normal form computation using the OCaml native compiler. It complements the virtual machine conversion offering much faster computation for expensive functions.

Coq 8.5 also comes with a bunch of many various smaller-scale changes and improvements regarding the different components of the system. We shall only list a few of them.

Pierre Boutillier developed an improved tactic for simplification of expressions called `cbn`.

Maxime Dénès maintained the bytecode-based reduction machine. Pierre Letouzey maintained the extraction mechanism.

Pierre-Marie Pédrot has extended the syntax of terms to, experimentally, allow holes in terms to be solved by a locally specified tactic.

Existential variables are referred to by identifiers rather than mere numbers, thanks to Hugo Herbelin who also improved the tactic language here and there.

Error messages for universe inconsistencies have been improved by Matthieu Sozeau. Error messages for unification and type inference failures have been improved by Hugo Herbelin, Pierre-Marie Pédrot and Arnaud Spiwack.

Pierre Courtieu contributed new features for using Coq through Proof General and for better interactive experience (bullets, Search, etc).

The efficiency of the whole system has been significantly improved thanks to contributions from Pierre-Marie Pédrot.

A distribution channel for Coq packages using the OPAM tool has been initiated by Thomas Braibant and developed by Guillaume Claret, with contributions by Enrico Tassi and feedback from Hugo Herbelin.

Packaging tools were provided by Pierre Letouzey and Enrico Tassi (Windows), Pierre Boutillier, Matthieu Sozeau and Maxime Dénès (MacOS X). Maxime Dénès improved significantly the testing and benchmarking support.

Many power users helped to improve the design of the new features via the bug tracker, the coq development mailing list or the Coq-Club mailing list. Special thanks are going to the users who contributed patches and intensive brain-storming, starting with Jason Gross, Jonathan Leivent, Greg Malecha, Clément Pit-Claudel, Marc Lasson, Lionel Rieg. It would however be impossible to mention with precision all names of people who to some extent influenced the development.

Version 8.5 is one of the most important releases of Coq. Its development spanned over about 3 years and a half with about one year of beta-testing. General maintenance during part or whole of this period has been done by Pierre Boutillier, Pierre Courtieu, Maxime Dénès, Hugo Herbelin, Pierre Letouzey, Guillaume Melquiond, Pierre-Marie Pédrot, Matthieu Sozeau, Arnaud Spiwack, Enrico Tassi as well as Bruno Barras, Yves Bertot, Frédéric Besson, Xavier Clerc, Pierre Corbineau, Jean-Christophe Filliâtre, Julien Forest, Sébastien Hinderer, Assia Mahboubi, Jean-Marc Notin, Yann Régis-Gianas, François Ripault, Carst Tankink. Maxime Dénès coordinated the release process.

Paris, January 2015, revised December 2015,
Hugo Herbelin, Matthieu Sozeau and the Coq development team

Potential sources of incompatibilities

List of typical changes to be done to adapt files from Coq 8.4 to Coq 8.5 when not using compatibility option `-compat 8.4`.

- Symptom: ”The reference omega was not found in the current environment”.

  Cause: ”Require Omega” does not import the tactic ”omega” any more

  Possible solutions:

  - use ”Require Import OmegaTactic” (not compatible with 8.4)
  - use ”Require Import Omega” (compatible with 8.4)
  - add definition ”Ltac omega := Coq.omega.Omega.omega.”

- Symptom: ”intuition” cannot solve a goal (not working anymore on non standard connective)

  Cause: ”intuition” had an accidental non uniform behavior fixed on non standard connectives

  Possible solutions:

  - use ”dintuition” instead; it is stronger than ”intuition” and works uniformly on non standard connectives, such as n-ary conjunctions or disjunctions (not compatible with 8.4)
  - do the script differently

- Symptom: The constructor foo (in type bar) expects n arguments.

  Cause: parameters must now be given in patterns

  Possible solutions:

  - use option ”Set Asymmetric Patterns” (compatible with 8.4)
  - add ”_” for the parameters (not compatible with 8.4)
  - turn the parameters into implicit arguments (compatible with 8.4)

- Symptom: ”NPeano.Nat.foo” not existing anymore

  Possible solutions:

  - use ”Nat.foo” instead

  Symptom: typing problems with proj1_sig or similar

  Cause: coercion from sig to sigT and similar coercions have been removed so as to make the initial state easier to understand for beginners

  Solution: change proj1_sig into projT1 and similarly (compatible with 8.4)

Other detailed changes

- options for *coq* compilation (see below for ocaml).

  - [-I foo] is now deprecated and will not add directory foo to the coq load path (only for ocaml, see below). Just replace [-I foo] by [-Q foo ””] in your project file and re-generate makefile. Or perform the same operation directly in your makefile if you edit it by hand.
  - Option -R Foo bar is the same in v8.5 than in v8.4 concerning coq load path.
  - Option [-I foo -as bar] is unchanged but discouraged unless you compile ocaml code. Use -Q foo bar instead.

  for more details: see section ”Customization at launch time” of the reference manual.

- Command line options for ocaml Compilation of ocaml code (plugins)

  - [-I foo] is *not* deprecated to add foo to the ocaml load path.
  - [-I foo -as bar] adds foo to the ocaml load path *and* adds foo to the coq load path with logical name bar (shortcut for -I foo -Q foo bar).

  for more details: section "Customization at launch time" of the reference manual.

- Universe Polymorphism.
- Refinement, unification and tactics are now aware of universes, resulting in more localized errors. Universe inconsistencies should no more get raised at Qed time but during the proof. Unification *always* produces well-typed substitutions, hence some rare cases of unifications that succeeded while producing ill-typed terms before will now fail.
- The [change p with c] tactic semantics changed, now typechecking [c] at each matching occurrence [t] of the pattern [p], and converting [t] with [c].
- Template polymorphic inductive types: the partial application of a template polymorphic type (e.g. list) is not polymorphic. An explicit parameter application (e.g [fun A => list A]) or [apply (list _)] will result in a polymorphic instance.
- The type inference algorithm now takes opacity of constants into account. This may have effects on tactics using type inference (e.g. induction). Extra "Transparent" might have to be added to revert opacity of constants.

Type classes.

- When writing an `Instance foo : Class A := {| proj := t |}` (note the vertical bars), support for typechecking the projections using the type information and switching to proof mode is no longer available. Use `{ }` (without the vertical bars) instead.

Tactic abstract.

- Auxiliary lemmas generated by the abstract tactic are removed from the global environment and inlined in the proof term when a proof is ended with Qed. The behavior of 8.4 can be obtained by ending proofs with "Qed exporting" or "Qed exporting ident, .., ident".

Details of changes in 8.5beta1

Logic

- Primitive projections for records allow for a compact representation of projections, without parameters and avoid the behavior of defined projections that can unfold to a case expression. To turn the use of native projections on, use [Set Primitive Projections]. Record, Class and Structure types defined while this option is set will be defined with primitive projections instead of the usual encoding as a case expression. For compatibility, when p is a primitive projection, @p can be used to refer to the projection with explicit parameters, i.e. [@p] is definitionally equal to [λ params r. r.(p)]. Records with primitive projections have eta-conversion, the canonical form being [mkR pars (p1 t) ... (pn t)].
- New universe polymorphism (see reference manual)
- New option -type-in-type to collapse the universe hierarchy (this makes the logic inconsistent).
- The guard condition for fixpoints is now a bit stricter. Propagation of subterm value through pattern matching is restricted according to the return predicate. Restores compatibility of Coq's logic with the propositional extensionality axiom. May create incompatibilities in recursive programs heavily using dependent types.
- Trivial inductive types are no longer defined in Type but in Prop, which leads to a non-dependent induction principle being generated in place of the dependent one. To recover the old behavior, explicitly define your inductive types in Set.

Commands

- A command "Variant" allows to define non-recursive variant types.
- The command "Record foo ..." does not generate induction principles (foo_rect, foo_rec, foo_ind) anymore by default (feature wish #2693). The command "Variant foo ..." does not either. A flag "Set/Unset Nonrecursive Elimination Schemes" allows changing this. The tactic "induction" on a "Record" or a "Variant" is now actually doing "destruct".
- The "Open Scope" command can now be given also a delimiter (e.g. Z).
- The "Definition" command now allows the "Local" modifier, allowing for non-importable definitions. The same goes for "Axiom" and "Parameter".
- Section-specific commands such as "Let" (resp. "Variable", "Hypothesis") used out of a section now behave like the corresponding "Local" command, i.e. "Local Definition" (resp. "Local Parameter", "Local Axiom"). (potential source of rare incompatibilities).
- The "Let" command can now define local (co)fixpoints.
- Command "Search" has been renamed into "SearchHead". The command name "Search" now behaves like former "SearchAbout". The latter name is deprecated.
- "Search", "About", "SearchHead", "SearchRewrite" and "SearchPattern" now search for hypothesis (of the current goal by default) first. They now also support the goal selector prefix to specify another goal to search: e.g. "n:Search id". This is also true for SearchAbout although it is deprecated.
- The coq/user-contrib directory and the XDG directories are no longer recursively added to the load path, so files from installed libraries now need to be fully qualified for the "Require" command to find them. The tools/update-require script can be used to convert a development.
- A new Print Strategies command allows visualizing the opacity status of the whole engine.
- The "Locate" command now searches through all sorts of qualified namespaces of Coq: terms, modules, tactics, etc. The old behavior of the command can be retrieved using the "Locate Term" command.
- New "Derive" command to help writing program by derivation.
- New "Refine Instance Mode" option that allows to deactivate the generation of obligations in incomplete typeclass instances, raising an error instead.
- "Collection" command to name sets of section hypotheses. Named collections can be used in the syntax of "Proof using" to assert which section variables are used in a proof.
- The "Optimize Proof" command can be placed in the middle of a proof to force the compaction of the data structure used to represent the ongoing proof (evar map). This may result in a lower memory footprint and speed up the execution of the following tactics.
- "Optimize Heap" command to tell the OCaml runtime to perform a major garbage collection step and heap compaction.
- `Instance` no longer treats the `{|...|}` syntax specially; it handles it in the same way as other commands, e.g. "Definition". Use the `{...}` syntax (no pipe symbols) to recover the old behavior.

Specification Language

- Slight changes in unification error messages.
- Added a syntax $(...)$ that allows putting tactics in terms (may break user notations using "$(", fixable by inserting a space or rewriting the notation).
- Constructors in pattern-matching patterns now respect the same rules regarding implicit arguments as in applicative position. The old behavior can be recovered by the command "Set Asymmetric Patterns". As a side effect, notations for constructors explicitly mentioning non-implicit parameters can now be used in patterns. Considering that the pattern language is already rich enough, binding local definitions is however now forbidden in patterns (source of incompatibilities for local definitions that delta-reduce to a constructor).

- Type inference algorithm now granting opacity of constants. This might also affect behavior of tactics (source of incompatibilities, solvable by re-declaring transparent constants which were set opaque).
- Existential variables are now referred to by an identifier and the relevant part of their instance is displayed by default. They can be reparsed. The naming policy is yet unstable and subject to changes in future releases.

Tactics

- New tactic engine allowing dependent subgoals, fully backtracking (also known as multiple success) tactics, as well as tactics which can consider multiple goals together. In the new tactic engine, instantiation information of existential variables is always propagated to tactics, removing the need to manually use the "instantiate" tactics to mark propagation points.
  - New tactical (a+b) inserts a backtracking point. When (a+b);c fails during the execution of c, it can backtrack and try b instead of a.
  - New tactical (once a) removes all the backtracking points from a (i.e. it selects the first success of a).
  - Tactic "constructor" is now fully backtracking. In case of incompatibilities (e.g. combinatoric explosion), the former behavior of "constructor" can be retrieved by using instead "[> once constructor ..]". Thanks to backtracking, undocumented "constructor <tac>" syntax is now equivalent to "[> once (constructor; tac) ..]".
  - New "multimatch" variant of "match" tactic which backtracks to new branches in case of a later failure. The "match" tactic is equivalent to "once multimatch".
  - New selector "all:" such that "all:tac" applies tactic "tac" to all the focused goals, instead of just the first one as is the default.
  - A corresponding new option Set Default Goal Selector "all" makes the tactics in scripts be applied to all the focused goal by default
  - New selector "par:" such that "par:tac" applies the (terminating) tactic "tac" to all the focused goal in parallel. The number of worker can be selected with -async-proofs-tac-j and also limited using the coqworkmgr utility.
  - New tactics "revgoals", "cycle" and "swap" to reorder goals.
  - The semantics of recursive tactics (introduced with "Ltac t := ..." or "let rec t := ... in ...") changed slightly as t is now applied to every goal, not each goal independently. In particular it may be applied when no goals are left. This may cause tactics such as "let rec t := constructor;t" to loop indefinitely. The simple fix is to rewrite the recursive calls as follows: "let rec t := constructor;[t..]" which recovers the earlier behavior (source of rare incompatibilities).
  - New tactic language feature "numgoals" to count number of goals. It is accompanied by a "guard" tactic which fails if a Boolean test over integers does not pass.
  - New tactical "[> ...]" to apply tactics to individual goals.
  - New tactic "gfail" which works like "fail" except it will also fail if every goal has been solved.
  - The refine tactic is changed not to use an ad hoc typing algorithm to generate subgoals. It also uses the dependent subgoal feature to generate goals to materialize every existential variable which is introduced by the refinement (source of incompatibilities).
  - A tactic shelve is introduced to manage the subgoals which may be solved by unification: shelve removes every goal it is applied to from focus. These goals can later be called back into focus by the Unshelve command.
  - A variant shelve_unifiable only removes those goals which appear as existential variables in other goals. To emulate the old refine, use "refine c;shelve_unifiable". This can still cause incompatibilities in rare occasions.
  - New "give_up" tactic to skip over a goal. A proof containing given up goals cannot be closed with "Qed", but only with "Admitted".
- The implementation of the admit tactic has changed: no axiom is generated for the admitted sub proof. "admit" is now an alias for "give_up". Code relying on this specific behavior of "admit" can be made to work by:

  - Adding an "Axiom" for each admitted subproof.
  - Adding a single "Axiom proof_admitted : False." and the Ltac definition "Ltac admit := case proof_admitted.".

- Matching using "lazymatch" was fundamentally modified. It now behaves like "match" (immediate execution of the matching branch) but without the backtracking mechanism in case of failure.
- New "tryif t then u else v" tactical which executes "u" in case of success of "t" and "v" in case of failure.
- New conversion tactic "native_compute": evaluates the goal (or an hypothesis) with a call-by-value strategy, using the OCaml native compiler. Useful on very intensive computations.
- New "cbn" tactic, a well-behaved simpl.
- Repeated identical calls to omega should now produce identical proof terms.
- Tactics btauto, a reflexive Boolean tautology solver.
- Tactic "tauto" was exceptionally able to destruct other connectives than the binary connectives "and", "or", "prod", "sum", "iff". This non-uniform behavior has been fixed (bug #2680) and tauto is slightly weaker (possible source of incompatibilities). On the opposite side, new tactic "dtauto" is able to destruct any record-like inductive types, superseding the old version of "tauto".
- Similarly, "intuition" has been made more uniform and, where it now fails, "dintuition" can be used (possible source of incompatibilities).
- New option "Unset Intuition Negation Unfolding" for deactivating automatic unfolding of "not" in intuition.
- Tactic notations can now be defined locally to a module (use "Local" prefix).
- Tactic "red" now reduces head beta-iota redexes (potential source of rare incompatibilities).
- Tactic "hnf" now reduces inner beta-iota redexes (potential source of rare incompatibilities).
- Tactic "intro H" now reduces beta-iota redexes if these hide a product (potential source of rare incompatibilities).
- In Ltac matching on patterns of the form "_ pat1 ... patn" now behaves like if matching on "?X pat1 ... patn", i.e. accepting "_" to be instantiated by an applicative term (experimental at this stage, potential source of incompatibilities).
- In Ltac matching on goal, types of hypotheses are now interpreted in the %type scope (possible source of incompatibilities).
- "change ... in ..." and "simpl ... in ..." now properly consider nested occurrences (possible source of incompatibilities since this alters the numbering of occurrences), but do not support nested occurrences.
- Tactics simpl, vm_compute and native_compute can be given a notation string to a constant as argument.
- When given a reference as argument, simpl, vm_compute and native_compute now strictly interpret it as the head of a pattern starting with this reference.
- The "change p with c" tactic semantics changed, now type checking "c" at each matching occurrence "t" of the pattern "p", and converting "t" with "c".
- Now "appcontext" and "context" behave the same. The old buggy behavior of "context" can be retrieved at parse time by setting the "Tactic Compat Context" flag (possible source of incompatibilities).
- New introduction pattern p/c which applies lemma c on the fly on the hypothesis under consideration before continuing with introduction pattern p.
- New introduction pattern [= x1 .. xn] applies "injection as [x1 .. xn]" on the fly if injection is applicable to the hypothesis under consideration (idea borrowed from Georges Gonthier). Introduction pattern [=] applies "discriminate" if a discriminable equality.

- New introduction patterns * and ** to respectively introduce all forthcoming dependent variables and all variables/hypotheses dependent or not.
- Tactic "injection c as ipats" now clears c if c refers to an hypothesis and moves the resulting equations in the hypotheses independently of the number of ipats, which has itself to be less than the number of new hypotheses (possible source of incompatibilities; former behavior obtainable by "Unset Injection L2R Pattern Order").
- Tactic "injection" now automatically simplifies subgoals "existT n p = existT n p'" into "p = p'" when "n" is in an inductive type for which a decidable equality scheme has been generated with "Scheme Equality" (possible source of incompatibilities).
- New tactic "rewrite_strat" for generalized rewriting with user-defined strategies, subsuming autorewrite.
- Injection can now also deduce equality of arguments of sort Prop, by using the option "Set Injection On Proofs" (disabled by default). Also improved the error messages.
- Tactic "subst id" now supports id occurring in dependent local definitions.
- Bugs fixed about intro-pattern "*" might lead to some rare incompatibilities.
- New tactical "time" to display time spent executing its argument.
- Tactics referring or using a constant dependent in a section variable which has been cleared or renamed in the current goal context now fail (possible source of incompatibilities solvable by avoiding clearing the relevant hypotheses).
- New construct "uconstr:c" and "type_term c" to build untyped terms.
- Binders in terms defined in Ltac (either "constr" or "uconstr") can now take their names from identifiers defined in Ltac. As a consequence, a name cannot be used in a binder "constr:(fun x => ...)" if an Ltac variable of that name already exists and does not contain an identifier. Source of occasional incompatibilities.
- The "refine" tactic now accepts untyped terms built with "uconstr" so that terms with holes can be constructed piecewise in Ltac.
- New bullets --, ++, **+, ---, +++, ***, ... made available.
- More informative messages when wrong bullet is used.
- Bullet suggestion when a subgoal is solved.
- New tactic "enough", symmetric to "assert", but with subgoals swapped, as a more friendly replacement of "cut".
- In destruct/induction, experimental modifier "!" prefixing the hypothesis name to tell not erasing the hypothesis.
- Bug fixes in "inversion as" may occasionally lead to incompatibilities.
- Behavior of introduction patterns -> and <- made more uniform (hypothesis is cleared, rewrite in hypotheses and conclusion and erasing the variable when rewriting a variable).
- New experimental option "Set Standard Proposition Elimination Names" so that case analysis or induction on schemes in Type containing propositions now produces "H"-based names.
- Tactics from plugins are now active only when the corresponding module is imported (source of incompatibilities, solvable by adding an "Import"; in the particular case of Omega, use "Require Import OmegaTactic").
- Semantics of destruct/induction has been made more regular in some edge cases, possibly leading to incompatibilities:
  - new goals are now opened when the term does not match a subterm of the goal and has unresolved holes, while in 8.4 these holes were turned into existential variables
  - when no "at" option is given, the historical semantics which selects all subterms syntactically identical to the first subterm matching the given pattern is used
  - non-dependent destruct/induction on an hypothesis with premises in an inductive type with indices is fixed

  - residual local definitions are now correctly removed.

- The rename tactic may now replace variables in parallel.
- A new ”Info” command replaces the ”info” tactical discontinued in v8.4. It still gives informative results in many cases.
- The ”info_auto” tactic is known to be broken and does not print a trace anymore. Use ”Info 1 auto” instead. The same goes for ”info_trivial”. On the other hand ”info_eauto” still works fine, while ”Info 1 eauto” prints a trivial trace.
- When using a lemma of the prototypical form ”forall A, {a:A & P a}”, ”apply” and ”apply in” do not instantiate anymore ”A” with the current goal and use ”a” as the proof, as they were sometimes doing, now considering that it is a too powerful decision.

Program

- ”Solve Obligations using” changed to ”Solve Obligations with”, consistent with ”Proof with”.
- Program Lemma, Definition now respect automatic introduction.
- Program Lemma, Definition, etc.. now interpret ”->” like Lemma and Definition as a non-dependent arrow (potential source of incompatibility).
- Add/document ”Set Hide Obligations” (to hide obligations in the final term inside an implicit argument) and ”Set Shrink Obligations” (to minimize dependencies of obligations defined by tactics).

Notations

- The syntax ”x -> y” is now declared at level 99. In particular, it has now a lower priority than ”<->”: ”A -> B <-> C” is now ”A -> (B <-> C)” (possible source of incompatibilities)
- Notations accept term-providing tactics using the $(...)$ syntax.
- ”Bind Scope” can no longer bind ”Funclass” and ”Sortclass”.
- A notation can be given a (compat ”8.x”) annotation, making it behave like a ”only parsing” notation, but the annotation may lead to eventually issue warnings or errors in further versions when this notation is used.
- More systematic insertion of spaces as a default for printing notations (”format” still available to override the default).
- In notations, a level modifier referring to a non-existent variable is now considered an error rather than silently ignored.

Tools

- Option -I now only adds directories to the ml path.
- Option -Q behaves as -R, except that the logical path of any loaded file has to be fully qualified.
- Option -R no longer adds recursively to the ml path; only the root directory is added. (Behavior with respect to the load path is unchanged.)
- Option -nois prevents coq/theories and coq/plugins to be recursively added to the load path. (Same behavior as with coq/user-contrib.)
- coqdep accepts a -dumpgraph option generating a dot file.
- Makefiles generated through coq_makefile have three new targets ”quick” ”checkproofs” and ”vio2vo”, allowing respectively to asynchronously compile the files without playing the proof scripts, asynchronously checking that the quickly generated proofs are correct and generating the object files from the quickly generated proofs.
- The XML plugin was discontinued and removed from the source.

- A new utility called coqworkmgr can be used to limit the number of concurrent workers started by independent processes, like make and CoqIDE. This is of interest for users of the par: goal selector.

Interfaces

- CoqIDE supports asynchronous edition of the document, ongoing tasks and errors are reported in the bottom right window. The number of workers taking care of processing proofs can be selected with -async-proofs-j.
- CoqIDE highlights in yellow ”unsafe” commands such as axiom declarations, and tactics like ”give_up”.
- CoqIDE supports Proof General like key bindings; to activate the PG mode go to Edit -> Preferences -> Editor. For the documentation see Help -> Help for PG mode.
- CoqIDE automatically retracts the locked area when one edits the locked text.
- CoqIDE search and replace got regular expressions power. See the documentation of OCaml's Str module for the supported syntax.
- Many CoqIDE windows, including the query one, are now detachable to improve usability on multi screen work stations.
- Coqtop/coqc outputs highlighted syntax. Colors can be configured thanks to the COQ_COLORS environment variable, and their current state can be displayed with the -list-tags command line option.
- Third party user interfaces can install their main loop in $COQLIB/toploop and call coqtop with the -toploop flag to select it.

Internal Infrastructure

- Many reorganizations in the ocaml source files. For instance, many internal a.s.t. of Coq are now placed in mli files in a new directory intf/, for instance constrexpr.mli or glob_term.mli. More details in dev/doc/changes.
- The file states/initial.coq does not exist anymore. Instead, coqtop initially does a ”Require” of Prelude.vo (or nothing when given the options -noinit or -nois).
- The format of vo files has slightly changed: cf final comments in checker/cic.mli.
- The build system does not produce anymore programs named coqtop.opt and a symbolic link to coqtop. Instead, coqtop is now directly an executable compiled with the best OCaml compiler available. The bytecode program coqtop.byte is still produced. Same for other utilities.
- Some options of the ./configure script slightly changed:
  - The -coqrunbyteflags and its blank-separated argument is replaced by option -vmbyteflags which expects a comma-separated argument.
  - The -coqtoolsbyteflags option is discontinued, see -no-custom instead.

Miscellaneous

- ML plugins now require a ”DECLARE PLUGIN ”foo”” statement. The ”foo” name must be exactly the name of the ML module that will be loaded through a ”Declare ML ”foo”” command.

Details of changes in 8.5beta2

Logic

- The VM now supports inductive types with up to 8388851 non-constant constructors and up to 8388607 constant ones.

Specification language

- Syntax ”$(tactic)$” changed to ”ltac: tactic”.

Tactics

- A script using the admit tactic can no longer be concluded by either Qed or Defined. In the first case, Admitted can be used instead. In the second case, a subproof should be used.
- The easy tactic and the now tactical now have a more predictable behavior, but they might now discharge some previously unsolved goals.

Extraction

- Definitions extracted to Haskell GHC should no longer randomly segfault when some Coq types cannot be represented by Haskell types.
- Definitions can now be extracted to Json for post-processing.

Tools

- Option -I -as has been removed, and option -R -as has been deprecated. In both cases, option -R can be used instead.
- coq_makefile now generates double-colon rules for rules such as clean.

API

- The interface of [change] has changed to take a [change_arg], which can be built from a [constr] using [make_change_arg].

Details of changes in 8.5beta3

Commands

- New command ”Redirect” to redirect the output of a command to a file.
- New command ”Undelimit Scope” to remove the delimiter of a scope.
- New option ”Strict Universe Declaration”, set by default. It enforces the declaration of all polymorphic universes appearing in a definition when introducing it.
- New command ”Show id” to show goal named id.
- Option ”Virtual Machine” removed.

Tactics

- New flag ”Regular Subst Tactic” which fixes ”subst” in situations where it failed to substitute all substitutable equations or failed to simplify cycles, or accidentally unfolded local definitions (flag is off by default).
- New flag ”Loose Hint Behavior” to handle hints loaded but not imported in a special way. It accepts three distinct flags: * ”Lax”, which is the default one, sets the old behavior, i.e. a non-imported hint behaves the same as an imported one. * ”Warn” outputs a warning when a non-imported hint is used. Note that this is an over-approximation, because a hint may be triggered by an eauto run that will eventually fail and backtrack. * ”Strict” changes the behavior of an unloaded hint to the one of the fail tactic, allowing to emulate the hopefully future import-scoped hint mechanism.

- New compatibility flag "Universal Lemma Under Conjunction" which let tactics working under conjunctions apply sublemmas of the form "forall A, ... -> A".
- New compatibility flag "Bracketing Last Introduction Pattern" which can be set so that the last disjunctive-conjunctive introduction pattern given to "intros" automatically complete the introduction of its subcomponents, as the the disjunctive-conjunctive introduction patterns in non-terminal position already do.
- New flag "Shrink Abstract" that minimalizes proofs generated by the abstract tactical w.r.t. variables appearing in the body of the proof.

Program

- The "Shrink Obligations" flag now applies to all obligations, not only those solved by the automatic tactic.
- Importing Program no longer overrides the "exists" tactic (potential source of incompatibilities).
- Hints costs are now correctly taken into account (potential source of incompatibilities).
- Documented the Hint Cut command that allows control of the proof search during typeclass resolution (see reference manual).

API

- Some functions from pretyping/typing.ml and their derivatives were potential source of evarmap leaks, as they dropped their resulting evarmap. The situation was clarified by renaming them according to a `unsafe_*` scheme. Their sound variant is likewise renamed to their old name. The following renamings were made.
  - `Typing.type_of` -> `unsafe_type_of`
  - `Typing.e_type_of` -> `type_of`
  - A new `e_type_of` function that matches the `e_` prefix policy
  - `Tacmach.pf_type_of` -> `pf_unsafe_type_of`
  - A new safe `pf_type_of` function.

  All uses of `unsafe_*` functions should be eventually eliminated.

Tools

- Added an option -w to control the output of coqtop warnings.
- Configure now takes an optional -native-compiler (yes|no) flag replacing -no-native-compiler. The new flag is set to no by default under Windows.
- Flag -no-native-compiler was removed and became the default for coqc. If precompilation of files for native conversion test is desired, use -native-compiler.
- The -compile command-line option now takes the full path of the considered file, including the ".v" extension, and outputs a warning if such an extension is lacking.
- The -require and -load-vernac-object command-line options now take a logical path of a given library rather than a physical path, thus they behave like Require [Import] path.
- The -vm command-line option has been removed.

Standard Library

- There is now a Coq.Compat.Coq84 library, which sets the various compatibility options and does a few redefinitions to make Coq behave more like Coq v8.4. The standard way of putting Coq in v8.4 compatibility mode is to pass the command line flags "-require Coq.Compat.Coq84 -compat 8.4".

Details of changes in 8.5

Tools

- Flag ”-compat 8.4” now loads Coq.Compat.Coq84. The standard way of putting Coq in v8.4 compatibility mode is to pass the command line flag ”-compat 8.4”. It can be followed by ”-require Coq.Compat.AdmitAxiom” if the 8.4 behavior of admit is needed, in which case it uses an axiom.

Specification language

- Syntax ”$(tactic)$” changed to ”ltac:(tactic)”.

Tactics

- Syntax ”destruct !hyp” changed to ”destruct (hyp)”, and similarly for induction (rare source of incompatibilities easily solvable by removing parentheses around ”hyp” when not for the purpose of keeping the hypothesis).
- Syntax ”p/c” for on-the-fly application of a lemma c before introducing along pattern p changed to p%c1..%cn. The feature and syntax are in experimental stage.
- ”Proof using” does not clear unused section variables.
- Tactic ”refine” has been changed back to the 8.4 behavior of shelving subgoals that occur in other subgoals. The ”refine” tactic of 8.5beta3 has been renamed ”simple refine”; it does not shelve any subgoal.
- New tactical ”unshelve tac” which grab existential variables put on the tactic shelve by the execution of ”tac”.

Details of changes in 8.5pl1

Critical bugfix

- The subterm relation for the guard condition was incorrectly defined on primitive projections (#4588)

Plugin development tools

- add a .merlin target to the makefile

Various performance improvements (time, space used by .vo files)

Other bugfixes

- Fix order of arguments to Big.compare_case in ExtrOcamlZBigInt.v
- Added compatibility coercions from Specif.v which were present in Coq 8.4.
- Fixing a source of inefficiency and an artificial dependency in the printer in the congruence tactic.
- Allow to unset the refinement mode of Instance in ML
- Fixing an incorrect use of prod_appvect on a term which was not a product in setoid_rewrite.
- Add -compat 8.4 econstructor tactics, and tests
- Add compatibility Nonrecursive Elimination Schemes
- Fixing the ”No applicable tactic” non informative error message regression on apply.
- Univs: fix get_current_context (bug #4603, part I)
- Fix a bug in Program coercion code
- Fix handling of arity of definitional classes.
- #4630: Some tactics are 20x slower in 8.5 than 8.4.
- #4627: records with no declared arity can be template polymorphic.

- #4623: set tactic too weak with universes (regression)
- Fix incorrect behavior of CS resolution
- #4591: Uncaught exception in directory browsing.
- CoqIDE is more resilient to initialization errors.
- #4614: ”Fully check the document” is uninterruptible.
- Try eta-expansion of records only on non-recursive ones
- Fix bug when a sort is ascribed to a Record
- Primitive projections: protect kernel from erroneous definitions.
- Fixed bug #4533 with previous Keyed Unification commit
- Win: kill unreliable hence do not waitpid after kill -9 (Close #4369)
- Fix strategy of Keyed Unification
- #4608: Anomaly ”output_value: abstract value (outside heap)”.
- #4607: do not read native code files if native compiler was disabled.
- #4105: poor escaping in the protocol between CoqIDE and coqtop.
- #4596: [rewrite] broke in the past few weeks.
- #4533 (partial): respect declared global transparency of projections in unification.ml
- #4544: Backtrack on using full betaiota reduction during keyed unification.
- #4540: CoqIDE bottom progress bar does not update.
- Fix regression from 8.4 in reflexivity
- #4580: [Set Refine Instance Mode] also used for Program Instance.
- #4582: cannot override notation [x]. MAY CREATE INCOMPATIBILITIES, see #4683.
- STM: Print/Extraction have to be skipped if -quick
- #4542: CoqIDE: STOP button also stops workers
- STM: classify some variants of Instance as regular “ `Fork ` nodes.
- #4574: Anomaly: Uncaught exception Invalid_argument(”splay_arity”).
- Do not give a name to anonymous evars anymore. See bug #4547.
- STM: always stock in vio files the first node (state) of a proof
- STM: not delegate proofs that contain Vernac(Module|Require|Import), #4530
- Don’t fail fatally if PATH is not set.
- #4537: Coq 8.5 is slower in typeclass resolution.
- #4522: Incorrect ”Warning...” on windows.
- #4373: coqdep does not know about .vio files.
- #3826: ”Incompatible module types” is uninformative.
- #4495: Failed assertion in metasyntax.ml.
- #4511: evar tactic can create non-typed evars.
- #4503: mixing universe polymorphic and monomorphic variables and definitions in sections is unsupported.

- #4519: oops, global shadowed local universe level bindings.
- #4506: Anomaly: File "pretyping/indrec.ml", line 169, characters 14-20: Assertion failed.
- #4548: CoqIDE crashes when going back one command

Details of changes in 8.5pl2

Critical bugfix

- Checksums of .vo files dependencies were not correctly checked.
- Unicode-to-ASCII translation was not injective, leading in a soundness bug in the native compiler.

Other bugfixes

- #4097: more efficient occur-check in presence of primitive projections
- #4398: type_scope used consistently in "match goal".
- #4450: eauto does not work with polymorphic lemmas
- #4677: fix alpha-conversion in notations needing eta-expansion.
- Fully preserve initial order of hypotheses in "Regular Subst Tactic" mode.
- #4644: a regression in unification.
- #4725: Function (Error: Conversion test raised an anomaly) and Program (Error: Cannot infer this placeholder of type)
- #4747: Problem building Coq 8.5pl1 with OCaml 4.03.0: Fatal warnings
- #4752: CoqIDE crash on files not ended by ".v".
- #4777: printing inefficiency with implicit arguments
- #4818: "Admitted" fails due to undefined universe anomaly after calling "destruct"
- #4823: remote counter: avoid thread race on sockets
- #4841: -verbose flag changed semantics in 8.5, is much harder to use
- #4851: [nsatz] cannot handle duplicated hypotheses
- #4858: Anomaly: Uncaught exception Failure("hd"). Please report. in variant of nsatz
- #4880: [nsatz_compute] generates invalid certificates if given redundant hypotheses
- #4881: synchronizing "Declare Implicit Tactic" with backtrack.
- #4882: anomaly with Declare Implicit Tactic on hole of type with evars
- Fix use of "Declare Implicit Tactic" in refine. triggered by CoqIDE
- #4069, #4718: congruence fails when universes are involved.

Universes

- Disallow silently dropping universe instances applied to variables (forward compatible)
- Allow explicit universe instances on notations, when they can apply to the head reference of their expansion.

Build infrastructure

- New update on how to find camlp5 binary and library at configure time.

Details of changes in 8.5pl3

Critical bugfix

- #4876: Guard checker incompleteness when using primitive projections

Other bugfixes

- #4780: Induction with universe polymorphism on was creating ill-typed terms.
- #4673: regression in setoid_rewrite, unfolding let-ins for type unification.
- #4754: Regression in setoid_rewrite, allow postponed unification problems to remain.
- #4769: Anomaly with universe polymorphic schemes defined inside sections.
- #3886: Program: duplicate obligations of mutual fixpoints.
- #4994: Documentation typo.
- #5008: Use the ”md5” command on OpenBSD.
- #5007: Do not assume the ”TERM” environment variable is always set.
- #4606: Output a break before a list only if there was an empty line.
- #5001: metas not cleaned properly in clenv_refine_in.
- #2336: incorrect glob data for module symbols (bug #2336).
- #4832: Remove extraneous dot in error message.
- Anomaly in printing a unification error message.
- #4947: Options which take string arguments are not backwards compatible.
- #4156: micromega cache files are now hidden files.
- #4871: interrupting par:abstract kills coqtop.
- #5043: [Admitted] lemmas pick up section variables.
- Fix name of internal refine (”simple refine”).
- #5062: probably a typo in Strict Proofs mode.
- #5065: Anomaly: Not a proof by induction.
- Restore native compiler optimizations, they were disabled since 8.5!
- #5077: failure on typing a fixpoint with evars in its type.
- Fix recursive notation bug.
- #5095: non relevant too strict test in let-in abstraction.
- Ensuring that the evar name is preserved by ”rename”.
- #4887: confusion between using and with in documentation of firstorder.
- Bug in subst with let-ins.
- #4762: eauto weaker than auto.
- Remove if_then_else (was buggy). Use tryif instead.
- #4970: confusion between special ”{” and non special ”{{” in notations.
- #4529: primitive projections unfolding.
- #4416: Incorrect ”Error: Incorrect number of goals”.

- #4863: abstract in typeclass hint fails.
- #5123: unshelve can impact typeclass resolution
- Fix a collision about the meta-variable ".." in recursive notations.
- Fix printing of info_auto.
- #3209: Not_found due to an occur-check cycle.
- #5097: status of evars refined by "clear" in ltac: closed wrt evars.
- #5150: Missing dependency of the test-suite subsystems in prerequisite.
- Fix a bug in error printing of unif constraints
- #3941: Do not stop propagation of signals when Coq is busy.
- #4822: Incorrect assertion in cbn.
- #3479 parsing of "{" and "}" when a keyword starts with "{" or "}".
- #5127: Memory corruption with the VM.
- #5102: bullets parsing broken by calls to parse_entry.

Various documentation improvements

Version 8.4

Summary of changes

Coq version 8.4 contains the result of three long-term projects: a new modular library of arithmetic by Pierre Letouzey, a new proof engine by Arnaud Spiwack and a new communication protocol for CoqIDE by Vincent Gross.

The new modular library of arithmetic extends, generalizes and unifies the existing libraries on Peano arithmetic (types nat, N and BigN), positive arithmetic (type positive), integer arithmetic (Z and BigZ) and machine word arithmetic (type Int31). It provides with unified notations (e.g. systematic use of add and mul for denoting the addition and multiplication operators), systematic and generic development of operators and properties of these operators for all the types mentioned above, including gcd, pcm, power, square root, base 2 logarithm, division, modulo, bitwise operations, logical shifts, comparisons, iterators, ...

The most visible feature of the new proof engine is the support for structured scripts (bullets and proof brackets) but, even if yet not user-available, the new engine also provides the basis for refining existential variables using tactics, for applying tactics to several goals simultaneously, for reordering goals, all features which are planned for the next release. The new proof engine forced Pierre Letouzey to reimplement info and Show Script differently.

Before version 8.4, CoqIDE was linked to Coq with the graphical interface living in a separate thread. From version 8.4, CoqIDE is a separate process communicating with Coq through a textual channel. This allows for a more robust interfacing, the ability to interrupt Coq without interrupting the interface, and the ability to manage several sessions in parallel. Relying on the infrastructure work made by Vincent Gross, Pierre Letouzey, Pierre Boutillier and Pierre-Marie Pédrot contributed many various refinements of CoqIDE.

Coq 8.4 also comes with a bunch of various smaller-scale changes and improvements regarding the different components of the system.

The underlying logic has been extended with η-conversion thanks to Hugo Herbelin, Stéphane Glondu and Benjamin Grégoire. The addition of η-conversion is justified by the confidence that the formulation of the Calculus of Inductive Constructions based on typed equality (such as the one considered in Lee and Werner to build a set-theoretic model of CIC [LW11]) is applicable to the concrete implementation of Coq.

The underlying logic benefited also from a refinement of the guard condition for fixpoints by Pierre Boutillier, the point being that it is safe to propagate the information about structurally smaller arguments through β-redexes that are blocked by the "match" construction (blocked commutative cuts).

Relying on the added permissiveness of the guard condition, Hugo Herbelin could extend the pattern matching compilation algorithm so that matching over a sequence of terms involving dependencies of a term or of the indices of the type of a term in the type of other terms is systematically supported.

Regarding the high-level specification language, Pierre Boutillier introduced the ability to give implicit arguments to anonymous functions, Hugo Herbelin introduced the ability to define notations with several binders (e.g. `exists x y z, P`), Matthieu Sozeau made the typeclass inference mechanism more robust and predictable, Enrico Tassi introduced a command Arguments that generalizes Implicit Arguments and Arguments Scope for assigning various properties to arguments of constants. Various improvements in the type inference algorithm were provided by Matthieu Sozeau and Hugo Herbelin with contributions from Enrico Tassi.

Regarding tactics, Hugo Herbelin introduced support for referring to expressions occurring in the goal by pattern in tactics such as set or destruct. Hugo Herbelin also relied on ideas from Chung-Kil Hur's Heq plugin to introduce automatic computation of occurrences to generalize when using destruct and induction on types with indices. Stéphane Glondu introduced new tactics *constr_eq*, *is_evar*, and *has_evar*, to be used when writing complex tactics. Enrico Tassi added support to fine-tuning the behavior of *simpl*. Enrico Tassi added the ability to specify over which variables of a section a lemma has to be exactly generalized. Pierre Letouzey added a tactic timeout and the interruptibility of *vm_compute*. Bug fixes and miscellaneous improvements of the tactic language came from Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau.

Regarding decision tactics, Loïc Pottier maintained nsatz, moving in particular to a typeclass based reification of goals while Frédéric Besson maintained Micromega, adding in particular support for division.

Regarding commands, Stéphane Glondu provided new commands to analyze the structure of type universes.

Regarding libraries, a new library about lists of a given length (called vectors) has been provided by Pierre Boutillier. A new instance of finite sets based on Red-Black trees and provided by Andrew Appel has been adapted for the standard library by Pierre Letouzey. In the library of real analysis, Yves Bertot changed the definition of π and provided a proof of the long-standing fact yet remaining unproved in this library, namely that $sin\frac{\pi}{2} = 1$.

Pierre Corbineau maintained the Mathematical Proof Language (C-zar).

Bruno Barras and Benjamin Grégoire maintained the call-by-value reduction machines.

The extraction mechanism benefited from several improvements provided by Pierre Letouzey.

Pierre Letouzey maintained the module system, with contributions from Élie Soubiran.

Julien Forest maintained the Function command.

Matthieu Sozeau maintained the setoid rewriting mechanism.

Coq related tools have been upgraded too. In particular, coq_makefile has been largely revised by Pierre Boutillier. Also, patches from Adam Chlipala for coqdoc have been integrated by Pierre Boutillier.

Bruno Barras and Pierre Letouzey maintained the `coqchk` checker.

Pierre Courtieu and Arnaud Spiwack contributed new features for using Coq through Proof General.

The Dp plugin has been removed. Use the plugin provided with Why 3 instead (http://why3.lri.fr/).

Under the hood, the Coq architecture benefited from improvements in terms of efficiency and robustness, especially regarding universes management and existential variables management, thanks to Pierre Letouzey and Yann Régis-Gianas with contributions from Stéphane Glondu and Matthias Puech. The build system is maintained by Pierre Letouzey with contributions from Stéphane Glondu and Pierre Boutillier.

A new backtracking mechanism simplifying the task of external interfaces has been designed by Pierre Letouzey.

The general maintenance was done by Pierre Letouzey, Hugo Herbelin, Pierre Boutillier, Matthieu Sozeau and Stéphane Glondu with also specific contributions from Guillaume Melquiond, Julien Narboux and Pierre-Marie Pédrot.

Packaging tools were provided by Pierre Letouzey (Windows), Pierre Boutillier (MacOS), Stéphane Glondu (Debian). Releasing, testing and benchmarking support was provided by Jean-Marc Notin.

Many suggestions for improvements were motivated by feedback from users, on either the bug tracker or the Coq-Club mailing list. Special thanks are going to the users who contributed patches, starting with Tom Prince. Other patch contributors include Cédric Auger, David Baelde, Dan Grayson, Paolo Herms, Robbert Krebbers, Marc Lasson, Hendrik Tews and Eelis van der Weegen.

Paris, December 2011
Hugo Herbelin

Potential sources of incompatibilities

The main known incompatibilities between 8.3 and 8.4 are consequences of the following changes:

- The reorganization of the library of numbers:

  Several definitions have new names or are defined in modules of different names, but a special care has been taken to have this renaming transparent for the user thanks to compatibility notations.

  However some definitions have changed, what might require some adaptations. The most noticeable examples are:

  - The "?=" notation which now bind to Pos.compare rather than former Pcompare (now Pos.compare_cont).
  - Changes in names may induce different automatically generated names in proof scripts (e.g. when issuing "destruct Z_le_gt_dec").
  - Z.add has a new definition, hence, applying "simpl" on subterms of its body might give different results than before.
  - BigN.shiftl and BigN.shiftr have reversed arguments order, the power function in BigN now takes two BigN.
- Other changes in libraries:
  - The definition of functions over "vectors" (list of fixed length) have changed.
  - TheoryList.v has been removed.
- Slight changes in tactics:
  - Less unfolding of fixpoints when applying destruct or inversion on a fixpoint hiding an inductive type (add an extra call to simpl to preserve compatibility).
  - Less unexpected local definitions when applying "destruct" (incompatibilities solvable by adapting name hypotheses).
  - Tactic "apply" might succeed more often, e.g. by now solving pattern-matching of the form ?f x y = g(x,y) (compatibility ensured by using "Unset Tactic Pattern Unification"), but also because it supports (full) betaiota (using "simple apply" might then help).
  - Tactic autorewrite does no longer instantiate pre-existing existential variables.
  - Tactic "info" is now available only for auto, eauto and trivial.
- Miscellaneous changes:
  - The command "Load" is now atomic for backtracking (use "Unset Atomic Load" for compatibility).

Details of changes in 8.4beta

Logic

- Standard eta-conversion now supported (dependent product only).
- Guard condition improvement: subterm property is propagated through beta-redex blocked by pattern-matching, as in "(match v with C .. => fun x => u end) x"; this allows for instance to use "rewrite ... in ..." without breaking the guard condition.

Specification language and notations

- Maximal implicit arguments can now be set locally by { }. The registration traverses fixpoints and lambdas. Because there is conversion in types, maximal implicit arguments are not taken into account in partial applications (use eta expanded form with explicit { } instead).
- Added support for recursive notations with binders (allows for instance to write "exists x y z, P").
- Structure/Record printing can be disable by "Unset Printing Records". In addition, it can be controlled on type by type basis using "Add Printing Record" or "Add Printing Constructor".
- Pattern-matching compilation algorithm: in "match x, y with ... end", possible dependencies of x (or of the indices of its type) in the type of y are now taken into account.

Tactics

- New proof engine.
- Scripts can now be structured thanks to bullets - * + and to subgoal delimitation via { }. Note: for use with Proof General, a cvs version of Proof General no older than mid-July 2011 is currently required.
- Support for tactical "info" is suspended.
- Support for command "Show Script" is suspended.
- New tactics constr_eq, is_evar and has_evar for use in Ltac (DOC TODO).
- Removed the two-argument variant of "decide equality".
- New experimental tactical "timeout <n> <tac>". Since <n> is a time in second for the moment, this feature should rather be avoided in scripts meant to be machine-independent.
- Fix in "destruct": removal of unexpected local definitions in context might result in some rare incompatibilities (solvable by adapting name hypotheses).
- Introduction pattern "_" made more robust.
- Tactic (and Eval command) vm_compute can now be interrupted via Ctrl-C.
- Unification in "apply" supports unification of patterns of the form ?f x y = g(x,y) (compatibility ensured by using "Unset Tactic Pattern Unification"). It also supports (full) betaiota.
- Tactic autorewrite does no longer instantiate pre-existing existential variables (theoretical source of possible incompatibilities).
- Tactic "dependent rewrite" now supports equality in "sig".
- Tactic omega now understands Zpred (wish #1912) and can prove any goal from a context containing an arithmetical contradiction (wish #2236).
- Using "auto with nocore" disables the use of the "core" database (wish #2188). This pseudo-database "nocore" can also be used with trivial and eauto.
- Tactics "set", "destruct" and "induction" accepts incomplete terms and use the goal to complete the pattern assuming it is non ambiguous.

- When used on arguments with a dependent type, tactics such as ”destruct”, ”induction”, ”case”, ”elim”, etc. now try to abstract automatically the dependencies over the arguments of the types (based on initial ideas from Chung-Kil Hur, extension to nested dependencies suggested by Dan Grayson)
- Tactic ”injection” now failing on an equality showing no constructors while it was formerly generalizing again the goal over the given equality.
- In Ltac, the ”context [...]” syntax has now a variant ”appcontext [...]” allowing to match partial applications in larger applications.
- When applying destruct or inversion on a fixpoint hiding an inductive type, recursive calls to the fixpoint now remain folded by default (rare source of incompatibility generally solvable by adding a call to simpl).
- In an ltac pattern containing a ”match”, a final ”| _ => _” branch could be used now instead of enumerating all remaining constructors. Moreover, the pattern ”match _ with _ => _ end” now allows to match any ”match”. A ”in” annotation can also be added to restrict to a precise inductive type.
- The behavior of ”simpl” can be tuned using the ”Arguments” vernacular. In particular constants can be marked so that they are always/never unfolded by ”simpl”, or unfolded only when a set of arguments evaluates to a constructor. Last one can mark a constant so that it is unfolded only if the simplified term does not expose a match in head position.

Commands

- It is now mandatory to have a space (or tabulation or newline or end-of-file) after a ”.” ending a sentence.
- In SearchAbout, the [ ] delimiters are now optional.
- New command ”Add/Remove Search Blacklist <substring> ...”: a Search or SearchAbout or similar query will never mention lemmas whose qualified names contain any of the declared substrings. The default blacklisted substrings are `_subproof`, `Private_`.
- When the output file of ”Print Universes” ends in ”.dot” or ”.gv”, the universe graph is printed in the DOT language, and can be processed by Graphviz tools.
- New command ”Print Sorted Universes”.
- The undocumented and obsolete option ”Set/Unset Boxed Definitions” has been removed, as well as syntaxes like ”Boxed Fixpoint foo”.
- A new option ”Set Default Timeout n / Unset Default Timeout”.
- Qed now uses information from the reduction tactics used in proof script to avoid conversion at Qed time to go into a very long computation.
- New command ”Show Goal ident” to display the statement of a goal, even a closed one (available from Proof General).
- Command ”Proof” accept a new modifier ”using” to force generalization over a given list of section variables at section ending (DOC TODO).
- New command ”Arguments” generalizing ”Implicit Arguments” and ”Arguments Scope” and that also allows to rename the parameters of a definition and to tune the behavior of the tactic ”simpl”.

Module System

- During subtyping checks, an opaque constant in a module type could now be implemented by anything of the right type, even if bodies differ. Said otherwise, with respect to subtyping, an opaque constant behaves just as a parameter. Coqchk was already implementing this, but not coqtop.
- The inlining done during application of functors can now be controlled more precisely, by the annotations (no inline) or (inline at level XX). With the latter annotation, only functor parameters whose levels are lower or equal than XX will be inlined. The level of a parameter can be fixed by ”Parameter Inline(30) foo”. When levels aren’t given, the default value is 100. One can also use the flag ”Set Inline Level ...” to set a level (DOC TODO).

- Print Assumptions should now handle correctly opaque modules (#2168).
- Print Module (Type) now tries to print more details, such as types and bodies of the module elements. Note that Print Module Type could be used on a module to display only its interface. The option "Set Short Module Printing" could be used to switch back to the earlier behavior were only field names were displayed.

Libraries

- Extension of the abstract part of Numbers, which now provide axiomatizations and results about many more integer functions, such as pow, gcd, lcm, sqrt, log2 and bitwise functions. These functions are implemented for nat, N, BigN, Z, BigZ. See in particular file NPeano for new functions about nat.
- The definition of types positive, N, Z is now in file BinNums.v
- Major reorganization of ZArith. The initial file ZArith/BinInt.v now contains an internal module Z implementing the Numbers interface for integers. This module Z regroups:
  - all functions over type Z : Z.add, Z.mul, ...
  - the minimal proofs of specifications for these functions : Z.add_0_l, ...
  - an instantiation of all derived properties proved generically in Numbers : Z.add_comm, Z.add_assoc, ...

  A large part of ZArith is now simply compatibility notations, for instance Zplus_comm is an alias for Z.add_comm. The direct use of module Z is now recommended instead of relying on these compatibility notations.
- Similar major reorganization of NArith, via a module N in NArith/BinNat.v
- Concerning the positive datatype, BinPos.v is now in a specific directory PArith, and contains an internal submodule Pos. We regroup there functions such as Pos.add Pos.mul etc as well as many results about them. These results are here proved directly (no Number interface for strictly positive numbers).
- Note that in spite of the compatibility layers, all these reorganizations may induce some marginal incompatibilies in scripts. In particular:
  - the "?=" notation for positive now refers to a binary function Pos.compare, instead of the infamous ternary Pcompare (now Pos.compare_cont).
  - some hypothesis names generated by the system may changed (typically for a "destruct Z_le_gt_dec") since naming is done after the short name of the head predicate (here now "le" in module Z instead of "Zle", etc).
  - the internals of Z.add has changed, now relying of Z.pos_sub.
- Also note these new notations:
  - "<?" "<=?" "=?" for boolean tests such as Z.ltb Z.leb Z.eqb.
  - "÷" for the alternative integer division Z.quot implementing the Truncate convention (former ZOdiv), while the notation for the Coq usual division Z.div implementing the Flooring convention remains "/". Their corresponding modulo functions are Z.rem (no notations) for Z.quot and Z.modulo (infix "mod" notation) for Z.div.
- Lemmas about conversions between these datatypes are also organized in modules, see for instance modules Z2Nat, N2Z, etc.
- When creating BigN, the macro-generated part NMake_gen is much smaller. The generic part NMake has been reworked and improved. Some changes may introduce incompatibilities. In particular, the order of the arguments for BigN.shiftl and BigN.shiftr is now reversed: the number to shift now comes first. By default, the power function now takes two BigN.
- Creation of Vector, an independent library for lists indexed by their length. Vectors' names override lists' one so you should not "Import" the library. All old names changed: function names follow the ocaml ones and, for example, Vcons becomes Vector.cons. You can get [..;..;..]-style notations by importing Vector.VectorNotations.
- Removal of TheoryList. Requiring List instead should work most of the time.

- New syntax ”rew Heq in H” and ”rew <- Heq in H” for eq_rect and eq_rect_r (available by importing module EqNotations).
- Wf.iter_nat is now Peano.nat_iter (with an implicit type argument).

Internal infrastructure

- Opaque proofs are now loaded lazily by default. This allows to be almost as fast as -dont-load-proofs, while being safer (no creation of axioms) and avoiding feature restrictions (Print and Print Assumptions work ok).
- Revised hash-consing code allowing more sharing of memory
- Experimental support added for camlp4 (the one provided alongside ocaml), simply pass option -usecamlp4 to ./configure. By default camlp5 is used.
- Revised build system: no more stages in Makefile thanks to some recursive aspect of recent gnu make, use of vo.itarget files containing .v to compile for both make and ocamlbuild, etc.
- Support of cross-compilation via mingw from unix toward Windows, contact P. Letouzey for more informations.
- New Makefile rules mli-doc to make html of mli in dev/doc/html and full-stdlib to get a (huge) pdf reflecting the whole standard library.

Extraction

- By default, opaque terms are now truly considered opaque by extraction: instead of accessing their body, they are now considered as axioms. The previous behaviour can be reactivated via the option ”Set Extraction AccessOpaque”.
- The pretty-printer for Haskell now produces layout-independent code
- A new command ”Separate Extraction cst1 cst2 ...” that mixes a minimal extracted environment a la ”Recursive Extraction” and the production of several files (one per coq source) a la ”Extraction Library” (DOC TODO).
- New option ”Set/Unset Extraction KeepSingleton” for preventing the extraction to optimize singleton container types (DOC TODO).
- The extraction now identifies and properly rejects a particular case of universe polymorphism it cannot handle yet (the pair (I,I) being Prop).
- Support of anonymous fields in record (#2555).

CoqIDE

- CoqIDE now runs coqtop as separated process, making it more robust: coqtop subprocess can be interrupted, or even killed and relaunched (cf button ”Restart Coq”, ex-”Go to Start”). For allowing such interrupts, the Windows version of coqide now requires Windows >= XP SP1.
- The communication between CoqIDE and coqtop is now done via a dialect of XML (DOC TODO).
- The backtrack engine of CoqIDE has been reworked, it now uses the ”Backtrack” command similarly to Proof General.
- The CoqIDE parsing of sentences has be reworked and now supports tactic delimitation via { }.
- CoqIDE now accepts the Abort command (wish #2357).
- CoqIDE can read coq_makefile files as ”project file” and use it to set automatically options to send to coqtop.
- Preference files have moved to $XDG_CONFIG_HOME/coq and accelerators are not stored as a list anymore.

Tools

- Coq now searches directories specified in COQPATH, $XDG_DATA_HOME/coq, $XDG_DATA_DIRS/coq, and user-contribs before the standard library.
- Coq rc file has moved to $XDG_CONFIG_HOME/coq.

- Major changes to coq_makefile:
  - mli/mlpack/mllib taken into account, ml not preproccessed anymore, ml4 work;
  - mlihtml generates doc of mli, install-doc install the html doc in DOCDIR with the same policy as vo in COQLIB;
  - More variables are given by coqtop -config, others are defined only if the users doesn't have defined them elsewhere. Consequently, generated makefile should work directly on any architecture;
  - Packagers can take advantage of $(DSTROOT) introduction. Installation can be made in $XDG_DATA_HOME/coq;
  - -arg option allows to send option as argument to coqc.

Details of changes in 8.4beta2

Commands

- Commands "Back" and "BackTo" are now handling the proof states. They may perform some extra steps of backtrack to avoid states where the proof state is unavailable (typically a closed proof).
- The commands "Suspend" and "Resume" have been removed.
- A basic Show Script has been reintroduced (no indentation).
- New command "Set Parsing Explicit" for deactivating parsing (and printing) of implicit arguments (useful for teaching).
- New command "Grab Existential Variables" to transform the unresolved evars at the end of a proof into goals.

Tactics

- Still no general "info" tactical, but new specific tactics info_auto, info_eauto, info_trivial which provides information on the proofs found by auto/eauto/trivial. Display of these details could also be activated by "Set Info Auto"/"Set Info Eauto"/"Set Info Trivial".
- Details on everything tried by auto/eauto/trivial during a proof search could be obtained by "debug auto", "debug eauto", "debug trivial" or by a global "Set Debug Auto"/"Set Debug Eauto"/"Set Debug Trivial".
- New command "r string" in Ltac debugger that interprets "idtac string" in Ltac code as a breakpoint and jumps to its next use.
- Tactics from the Dp plugin (simplify, ergo, yices, cvc3, z3, cvcl, harvey, zenon, gwhy) have been removed, since Why2 has not been maintained for the last few years. The Why3 plugin should be a suitable replacement in most cases.

Libraries

- MSetRBT: a new implementation of MSets via Red-Black trees (initial contribution by Andrew Appel).
- MSetAVL: for maximal sharing with the new MSetRBT, the argument order of Node has changed (this should be transparent to regular MSets users).

Module System

- The names of modules (and module types) are now in a fully separated namespace from ordinary definitions: "Definition E:=0. Module E. End E." is now accepted.

CoqIDE

- CoqIDE now supports the "Restart" command, and "Undo" (with a warning). Better support for "Abort".

Details of changes in 8.4

Commands

- The ”Reset” command is now supported again in files given to coqc or Load.
- ”Show Script” now indents again the displayed scripts. It can also work correctly across Load’ed files if the option ”Unset Atomic Load” is used.
- ”Open Scope” can now be given the delimiter (e.g. Z) instead of the full scope name (e.g. Z_scope).

Notations

- Most compatibility notations of the standard library are now tagged as (compat xyz), where xyz is a former Coq version, for instance ”8.3”. These notations behave as (only parsing) notations, except that they may triggers warnings (or errors) when used while Coq is not in a corresponding -compat mode.
- To activate these compatibility warnings, use ”Set Verbose Compat Notations” or the command-line flag -verbose-compat-notations.
- For a strict mode without these compatibility notations, use ”Unset Compat Notations” or the command-line flag -no-compat-notations.

Tactics

- An annotation ”eqn:H” or ”eqn:?” can be added to a ”destruct” or ”induction” to make it generate equations in the spirit of ”case_eq”. The former syntax ”_eqn” is discontinued.
- The name of the hypothesis introduced by tactic ”remember” can be set via the new syntax ”remember t as x eqn:H” (wish #2489).

Libraries

- Reals: changed definition of PI, no more axiom about sin(PI/2).
- SetoidPermutation: a notion of permutation for lists modulo a setoid equality.
- BigN: fixed the ocaml code doing the parsing/printing of big numbers.
- List: a couple of lemmas added especially about no-duplication, partitions.
- Init: Removal of the coercions between variants of sigma-types and subset types (possible source of incompatibility).

Version 8.3

Summary of changes

Coq version 8.3 is before all a transition version with refinements or extensions of the existing features and libraries and a new tactic nsatz based on Hilbert’s Nullstellensatz for deciding systems of equations over rings.

With respect to libraries, the main evolutions are due to Pierre Letouzey with a rewriting of the library of finite sets FSets and a new round of evolutions in the modular development of arithmetic (library Numbers). The reason for making FSets evolve is that the computational and logical contents were quite intertwined in the original implementation, leading in some cases to longer computations than expected and this problem is solved in the new MSets implementation. As for the modular arithmetic library, it was only dealing with the basic arithmetic operators in the former version and its current extension adds the standard theory of the division, min and max functions, all made available for free to any implementation of $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$.

The main other evolutions of the library are due to Hugo Herbelin who made a revision of the sorting library (including a certified merge-sort) and to Guillaume Melquiond who slightly revised and cleaned up the library of reals.

The module system evolved significantly. Besides the resolution of some efficiency issues and a more flexible construction of module types, Élie Soubiran brought a new model of name equivalence, the Δ-equivalence, which respects as much as possible the names given by the users. He also designed with Pierre Letouzey a new, convenient operator <+ for nesting functor application that provides a light notation for inheriting the properties of cascading modules.

The new tactic nsatz is due to Loïc Pottier. It works by computing Gröbner bases. Regarding the existing tactics, various improvements have been done by Matthieu Sozeau, Hugo Herbelin and Pierre Letouzey.

Matthieu Sozeau extended and refined the typeclasses and Program features (the Russell language). Pierre Letouzey maintained and improved the extraction mechanism. Bruno Barras and Élie Soubiran maintained the Coq checker, Julien Forest maintained the Function mechanism for reasoning over recursively defined functions. Matthieu Sozeau, Hugo Herbelin and Jean-Marc Notin maintained coqdoc. Frédéric Besson maintained the Micromega platform for deciding systems of inequalities. Pierre Courtieu maintained the support for the Proof General Emacs interface. Claude Marché maintained the plugin for calling external provers (dp). Yves Bertot made some improvements to the libraries of lists and integers. Matthias Puech improved the search functions. Guillaume Melquiond usefully contributed here and there. Yann Régis-Gianas grounded the support for Unicode on a more standard and more robust basis.

Though invisible from outside, Arnaud Spiwack improved the general process of management of existential variables. Pierre Letouzey and Stéphane Glondu improved the compilation scheme of the Coq archive. Vincent Gross provided support to CoqIDE. Jean-Marc Notin provided support for benchmarking and archiving.

Many users helped by reporting problems, providing patches, suggesting improvements or making useful comments, either on the bug tracker or on the Coq-Club mailing list. This includes but not exhaustively Cédric Auger, Arthur Charguéraud, François Garillot, Georges Gonthier, Robin Green, Stéphane Lescuyer, Eelis van der Weegen, ...

Though not directly related to the implementation, special thanks are going to Yves Bertot, Pierre Castéran, Adam Chlipala, and Benjamin Pierce for the excellent teaching materials they provided.

Paris, April 2010
Hugo Herbelin

Details of changes

Rewriting tactics

- Tactic ”rewrite” now supports rewriting on ad hoc equalities such as eq_true.
- ”Hint Rewrite” now checks that the lemma looks like an equation.
- New tactic ”etransitivity”.
- Support for heterogeneous equality (JMeq) in ”injection” and ”discriminate”.
- Tactic ”subst” now supports heterogeneous equality and equality proofs that are dependent (use ”simple subst” for preserving compatibility).
- Added support for Leibniz-rewriting of dependent hypotheses.
- Renamed ”Morphism” into ”Proper” and ”respect” into ”proper_prf” (possible source of incompatibility). A partial fix is to define ”Notation Morphism R f := (Proper (R%signature) f).”
- New tactic variants ”rewrite* by” and ”autorewrite*” that rewrite respectively the first and all matches whose side-conditions are solved.
- ”Require Import Setoid” does not export all of ”Morphisms” and ”RelationClasses” anymore (possible source of incompatibility, fixed by importing ”Morphisms” too).

- Support added for using Chung-Kil Hur's Heq library for rewriting over heterogeneous equality (courtesy of the library's author).
- Tactic "replace" supports matching terms with holes.

Automation tactics

- Tactic `intuition` now preserves inner `iff` and `not` (exceptional source of incompatibilities solvable by redefining `intuition` as `unfold iff, not in *; intuition`, or, for iff only, by using `Set Intuition Iff Unfolding`.)
- Tactic `tauto` now proves classical tautologies as soon as classical logic (i.e. library `Classical_Prop` or `Classical`) is loaded.
- Tactic `gappa` has been removed from the Dp plugin.
- Tactic `firstorder` now supports the combination of its `using` and `with` options.
- New `Hint Resolve ->` (or `<-`) for declaring iff's as oriented hints (wish #2104).
- An inductive type as argument of the `using` option of `auto` / `eauto` / `firstorder` is interpreted as using the collection of its constructors.
- New decision tactic "nsatz" to prove polynomial equations by computation of Groebner bases.

Other tactics

- Tactic "discriminate" now performs intros before trying to discriminate an hypothesis of the goal (previously it applied intro only if the goal had the form t1<>t2) (exceptional source of incompatibilities - former behavior can be obtained by "Unset Discriminate Introduction").
- Tactic "quote" now supports quotation of arbitrary terms (not just the goal).
- Tactic "idtac" now displays its "list" arguments.
- New introduction patterns "*" for introducing the next block of dependent variables and "**" for introducing all quantified variables and hypotheses.
- Pattern Unification for existential variables activated in tactics and new option "Unset Tactic Evars Pattern Unification" to deactivate it.
- Resolution of canonical structure is now part of the tactic's unification algorithm.
- New tactic "decide lemma with hyp" for rewriting decidability lemmas when one knows which side is true.
- Improved support of dependent goals over objects in dependent types for "destruct" (rare source of incompatibility that can be avoided by unsetting option "Dependent Propositions Elimination").
- Tactic "exists", "eexists", "destruct" and "edestruct" supports iteration using comma-separated arguments.
- Tactic names "case" and "elim" now support clauses "as" and "in" and become then synonymous of "destruct" and "induction" respectively.
- A new tactic name "exfalso" for the use of 'ex-falso quodlibet' principle. This tactic is simply a shortcut for "elimtype False".
- Made quantified hypotheses get the name they would have if introduced in the context (possible but rare source of incompatibilities).
- When applying a component of a conjunctive lemma, "apply in" (and sequences of "apply in") now leave the side conditions of the lemmas uniformly after the main goal (possible source of rare incompatibilities).
- In "simpl c" and "change c with d", c can be a pattern.
- Tactic "revert" now preserves let-in's making it the exact inverse of "intro".

- New tactics ”clear dependent H” and ”revert dependent H” that clears (resp. reverts) H and all the hypotheses that depend on H.
- Ltac’s pattern-matching now supports matching metavariables that depend on variables bound upwards in the pattern.

Tactic definitions

- Ltac definitions support Local option for non-export outside modules.
- Support for parsing non-empty lists with separators in tactic notations.
- New command ”Locate Ltac” to get the full name of an Ltac definition.

Notations

- Record syntax `{|x=...; y=...|}` now works inside patterns too.
- Abbreviations from non-imported module now invisible at printing time.
- Abbreviations now use implicit arguments and arguments scopes for printing.
- Abbreviations to pure names now strictly behave like the name they refer to (make redirections of qualified names easier).
- Abbreviations for applied constant now propagate the implicit arguments and arguments scope of the underlying reference (possible source of incompatibilities generally solvable by changing such abbreviations from e.g. `Notation foo' := (foo x)` to `Notation foo' y := (foo x (y:=y))`).
- The ”where” clause now supports multiple notations per defined object.
- Recursive notations automatically expand one step on the left for better factorization; recursion notations inner separators now ensured being tokens.
- Added ”Reserved Infix” as a specific shortcut of the corresponding ”Reserved Notation”.
- Open/Close Scope command supports Global option in sections.

Specification language

- New support for local binders in the syntax of Record/Structure fields.
- Fixpoint/CoFixpoint now support building part or all of bodies using tactics.
- Binders given before ”:” in lemmas and in definitions built by tactics are now automatically introduced (possible source of incompatibility that can be resolved by invoking ”Unset Automatic Introduction”).
- New support for multiple implicit arguments signatures per reference.

Module system

- Include Type is now deprecated since Include now accepts both modules and module types.
- Declare ML Module supports Local option.
- The sharing between non-logical object and the management of the name-space has been improved by the new ”Delta-equivalence” on qualified name.
- The include operator has been extended to high-order structures
- Sequences of Include can be abbreviated via new syntax ”<+”.
- A module (or module type) can be given several ”<:” signatures.
- Interactive proofs are now permitted in module type. Functors can hence be declared as Module Type and be used later to type themselves.

- A functor application can be prefixed by a "!" to make it ignore any "Inline" annotation in the type of its argument(s) (for examples of use of the new features, see libraries Structures and Numbers).
- Coercions are now active only when modules are imported (use "Set Automatic Coercions Import" to get the behavior of the previous versions of Coq).

Extraction

- When using (Recursive) Extraction Library, the filenames are directly the Coq ones with new appropriate extensions : we do not force anymore uncapital first letters for Ocaml and capital ones for Haskell.
- The extraction now tries harder to avoid code transformations that can be dangerous for the complexity. In particular many eta-expansions at the top of functions body are now avoided, clever partial applications will likely be preserved, let-ins are almost always kept, etc.
- In the same spirit, auto-inlining is now disabled by default, except for induction principles, since this feature was producing more frequently weird code than clear gain. The previous behavior can be restored via "Set Extraction AutoInline".
- Unicode characters in identifiers are now transformed into ascii strings that are legal in Ocaml and other languages.
- Harsh support of module extraction to Haskell and Scheme: module hierarchy is flattened, module abbreviations and functor applications are expanded, module types and unapplied functors are discarded.
- Less unsupported situations when extracting modules to Ocaml. In particular module parameters might be alpha-renamed if a name clash is detected.
- Extract Inductive is now possible toward non-inductive types (e.g. nat => int)
- Extraction Implicit: this new experimental command allows to mark some arguments of a function or constructor for removed during extraction, even if these arguments don't fit the usual elimination principles of extraction, for instance the length n of a vector.
- Files ExtrOcaml*.v in plugins/extraction try to provide a library of common extraction commands: mapping of basics types toward Ocaml's counterparts, conversions from/to int and big_int, or even complete mapping of nat,Z,N to int or big_int, or mapping of ascii to char and string to char list (in this case recognition of ascii constants is hard-wired in the extraction).

Program

- Streamlined definitions using well-founded recursion and measures so that they can work on any subset of the arguments directly (uses currying).
- Try to automatically clear structural fixpoint prototypes in obligations to avoid issues with opacity.
- Use return type clause inference in pattern-matching as in the standard typing algorithm.
- Support [Local Obligation Tactic] and [Next Obligation with tactic].
- Use [Show Obligation Tactic] to print the current default tactic.
- [fst] and [snd] have maximal implicit arguments in Program now (possible source of incompatibility).

Type classes

- Declaring axiomatic type class instances in Module Type should be now done via new command "Declare Instance", while the syntax "Instance" now always provides a concrete instance, both in and out of Module Type.
- Use [Existing Class foo] to declare a preexisting object [foo] as a class. [foo] can be an inductive type or a constant definition. No projections or instances are defined.
- Various bug fixes and improvements: support for defined fields, anonymous instances, declarations giving terms, better handling of sections and [Context].

Commands

- New command "Timeout <n> <command>." interprets a command and a timeout interrupts the execution after <n> seconds.
- New command "Compute <expr>." is a shortcut for "Eval vm_compute in <expr>".
- New command "Fail <command>." interprets a command and is successful iff the command fails on an error (but not an anomaly). Handy for tests and illustration of wrong commands.
- Most commands referring to constant (e.g. Print or About) now support referring to the constant by a notation string.
- New option "Boolean Equality Schemes" to make generation of boolean equality automatic for datatypes (together with option "Decidable Equality Schemes", this replaces deprecated option "Equality Scheme").
- Made support for automatic generation of case analysis schemes available to user (governed by option "Set Case Analysis Schemes").
- New command `Global?` **`Generalizable`** `All | No | Variable | Variables` *`ident`*`*` to declare which identifiers are generalizable in " `{ }` ` and " `( )` ` binders.
- New command "Print Opaque Dependencies" to display opaque constants in addition to all variables, parameters or axioms a theorem or definition relies on.
- New command "Declare Reduction <id> := <conv_expr>", allowing to write later "Eval <id> in ...". This command accepts a Local variant.
- Syntax of Implicit Type now supports more than one block of variables of a given type.
- Command "Canonical Structure" now warns when it has no effects.
- Commands of the form "Set X" or "Unset X" now support "Local" and "Global" prefixes.

Library

- Use "standard" Coq names for the properties of eq and identity (e.g. refl_equal is now eq_refl). Support for compatibility is provided.
- The function Compare_dec.nat_compare is now defined directly, instead of relying on lt_eq_lt_dec. The earlier version is still available under the name nat_compare_alt.
- Lemmas in library Relations and Reals have been homogenized a bit.
- The implicit argument of Logic.eq is now maximally inserted, allowing to simply write "eq" instead of "@eq _" in morphism signatures.
- Wrongly named lemmas (Zlt_gt_succ and Zlt_succ_gt) fixed (potential source of incompatibilities)
- List library:
  - Definitions of list, length and app are now in Init/Datatypes. Support for compatibility is provided.
  - Definition of Permutation is now in Sorting/Permtation.v
  - Some other light revisions and extensions (possible source of incompatibilities solvable by qualifying names accordingly).
- In ListSet, set_map has been fixed (source of incompatibilities if used).
- Sorting library:
  - new mergesort of worst-case complexity O(n*ln(n)) made available in Mergesort.v;
  - former notion of permutation up to setoid from Permutation.v is deprecated and moved to PermutSetoid.v;
  - heapsort from Heap.v of worst-case complexity O(n*n) is deprecated;
  - new file Sorted.v for some definitions of being sorted.

- Structure library. This new library is meant to contain generic structures such as types with equalities or orders, either in Module version (for now) or Type Classes (still to do):
  - DecidableType.v and OrderedType.v: initial notions for FSets/FMaps, left for compatibility but considered as deprecated.
  - Equalities.v and Orders.v: evolutions of the previous files, with fine-grain Module architecture, many variants, use of Equivalence and other relevant Type Classes notions.
  - OrdersTac.v: a generic tactic for solving chains of (in)equalities over variables. See {Nat,N,Z,P}OrderedType.v for concrete instances.
  - GenericMinMax.v: any ordered type can be equipped with min and max. We derived here all the generic properties of these functions.
- MSets library: an important evolution of the FSets library. "MSets" stands for Modular (Finite) Sets, by contrast with a forthcoming library of Class (Finite) Sets contributed by S. Lescuyer which will be integrated with the next release of Coq. The main features of MSets are:
  - The use of Equivalence, Proper and other Type Classes features easing the handling of setoid equalities.
  - The interfaces are now stated in iff-style. Old specifications are now derived properties.
  - The compare functions are now pure, and return a "comparison" value. Thanks to the CompSpec inductive type, reasoning on them remains easy.
  - Sets structures requiring invariants (i.e. sorted lists) are built first as "Raw" sets (pure objects and separate proofs) and attached with their proofs thanks to a generic functor. "Raw" sets have now a proper interface and can be manipulated directly.

  Note: No Maps yet in MSets. The FSets library is still provided for compatibility, but will probably be considered as deprecated in the next release of Coq.
- Numbers library:
  - The abstract layer (NatInt, Natural/Abstract, Integer/Abstract) has been simplified and enhance thanks to new features of the module system such as Include (see above). It has been extended to Euclidean division (three flavors for integers: Trunc, Floor and Math).
  - The arbitrary-large efficient numbers (BigN, BigZ, BigQ) has also been reworked. They benefit from the abstract layer improvements (especially for div and mod). Note that some specifications have slightly changed (compare, div, mod, shift{r,l}). Ring/Field should work better (true recognition of constants).

Tools

- Option -R now supports binding Coq root read-only.
- New coqtop/coqc option -beautify to reformat .v files (usable e.g. to globally update notations).
- New tool beautify-archive to beautify a full archive of developments.
- New coqtop/coqc option -compat X.Y to simulate the general behavior of previous versions of Coq (provides e.g. support for 8.2 compatibility).

Coqdoc

- List have been revamped. List depth and scope is now determined by an "offside" whitespace rule.
- Text may be italicized by placing it in _underscores_.
- The "--index <string>" flag changes the filename of the index.
- The "--toc-depth <int>" flag limits the depth of headers which are included in the table of contents.
- The "--lib-name <string>" flag prints "<string> Foo" instead of "Library Foo" where library titles are called for. The "--no-lib-name" flag eliminates the extra title.

- New option "--parse-comments" to allow parsing of regular `(* *)` comments.
- New option "--plain-comments" to disable interpretation inside comments.
- New option "--interpolate" to try and typeset identifiers in Coq escapings using the available globalization information.
- New option "--external url root" to refer to external libraries.
- Links to section variables and notations now supported.

Internal infrastructure

- To avoid confusion with the repository of user's contributions, the subdirectory "contrib" has been renamed into "plugins". On platforms supporting ocaml native dynlink, code located there is built as loadable plugins for coqtop.
- An experimental build mechanism via ocamlbuild is provided. From the top of the archive, run ./configure as usual, and then ./build. Feedback about this build mechanism is most welcome. Compiling Coq on platforms such as Windows might be simpler this way, but this remains to be tested.
- The Makefile system has been simplified and factorized with the ocamlbuild system. In particular "make" takes advantage of .mllib files for building .cma/.cmxa. The .vo files to compile are now listed in several vo.itarget files.

Version 8.2

Summary of changes

Coq version 8.2 adds new features, new libraries and improves on many various aspects.

Regarding the language of Coq, the main novelty is the introduction by Matthieu Sozeau of a package of commands providing Haskell-style typeclasses. Typeclasses, which come with a few convenient features such as type-based resolution of implicit arguments, play a new landmark role in the architecture of Coq with respect to automation. For instance, thanks to typeclass support, Matthieu Sozeau could implement a new resolution-based version of the tactics dedicated to rewriting on arbitrary transitive relations.

Another major improvement of Coq 8.2 is the evolution of the arithmetic libraries and of the tools associated with them. Benjamin Grégoire and Laurent Théry contributed a modular library for building arbitrarily large integers from bounded integers while Evgeny Makarov contributed a modular library of abstract natural and integer arithmetic together with a few convenient tactics. On his side, Pierre Letouzey made numerous extensions to the arithmetic libraries on $\mathbb{Z}$ and $\mathbb{Q}$, including extra support for automation in presence of various number-theory concepts.

Frédéric Besson contributed a reflective tactic based on Krivine-Stengle Positivstellensatz (the easy way) for validating provability of systems of inequalities. The platform is flexible enough to support the validation of any algorithm able to produce a "certificate" for the Positivstellensatz and this covers the case of Fourier-Motzkin (for linear systems in $\mathbb{Q}$ and $\mathbb{R}$), Fourier-Motzkin with cutting planes (for linear systems in $\mathbb{Z}$) and sum-of-squares (for non-linear systems). Evgeny Makarov made the platform generic over arbitrary ordered rings.

Arnaud Spiwack developed a library of 31-bits machine integers and, relying on Benjamin Grégoire and Laurent Théry's library, delivered a library of unbounded integers in base 2^{31}. As importantly, he developed a notion of "retro-knowledge" so as to safely extend the kernel-located bytecode-based efficient evaluation algorithm of Coq version 8.1 to use 31-bits machine arithmetic for efficiently computing with the library of integers he developed.

Beside the libraries, various improvements were contributed to provide a more comfortable end-user language and more expressive tactic language. Hugo Herbelin and Matthieu Sozeau improved the pattern matching compilation algorithm (detection of impossible clauses in pattern matching, automatic inference of the return type). Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau contributed various new convenient syntactic constructs and new tactics or tactic features: more inference of redundant information, better unification, better support for proof or definition by fixpoint, more expressive rewriting tactics, better support for meta-variables, more convenient notations...

Élie Soubiran improved the module system, adding new features (such as an "include" command) and making it more flexible and more general. He and Pierre Letouzey improved the support for modules in the extraction mechanism.

Matthieu Sozeau extended the Russell language, ending in an convenient way to write programs of given specifications, Pierre Corbineau extended the Mathematical Proof Language and the automation tools that accompany it, Pierre Letouzey supervised and extended various parts of the standard library, Stéphane Glondu contributed a few tactics and improvements, Jean-Marc Notin provided help in debugging, general maintenance and coqdoc support, Vincent Siles contributed extensions of the Scheme command and of injection.

Bruno Barras implemented the `coqchk` tool: this is a stand-alone type checker that can be used to certify .vo files. Especially, as this verifier runs in a separate process, it is granted not to be "hijacked" by virtually malicious extensions added to Coq.

Yves Bertot, Jean-Christophe Filliâtre, Pierre Courtieu and Julien Forest acted as maintainers of features they implemented in previous versions of Coq.

Julien Narboux contributed to CoqIDE. Nicolas Tabareau made the adaptation of the interface of the old "setoid rewrite" tactic to the new version. Lionel Mamane worked on the interaction between Coq and its external interfaces. With Samuel Mimram, he also helped making Coq compatible with recent software tools. Russell O'Connor, Cezary Kaliszyk, Milad Niqui contributed to improve the libraries of integers, rational, and real numbers. We also thank many users and partners for suggestions and feedback, in particular Pierre Castéran and Arthur Charguéraud, the INRIA Marelle team, Georges Gonthier and the INRIA-Microsoft Mathematical Components team, the Foundations group at Radboud university in Nijmegen, reporters of bugs and participants to the Coq-Club mailing list.

Palaiseau, June 2008
Hugo Herbelin

Details of changes

Language

- If a fixpoint is not written with an explicit { struct ... }, then all arguments are tried successively (from left to right) until one is found that satisfies the structural decreasing condition.
- New experimental typeclass system giving ad-hoc polymorphism and overloading based on dependent records and implicit arguments.
- New syntax "let 'pat := b in c" for let-binding using irrefutable patterns.
- New syntax "forall {A}, T" for specifying maximally inserted implicit arguments in terms.
- Sort of Record/Structure, Inductive and CoInductive defaults to Type if omitted.
- (Co)Inductive types can be defined as records (e.g. "CoInductive stream := { hd : nat; tl : stream }.")
- New syntax "Theorem id1:t1 ... with idn:tn" for proving mutually dependent statements.
- Support for sort-polymorphism on constants denoting inductive types.
- Several evolutions of the module system (handling of module aliases, functorial module types, an Include feature, etc).
- Prop now a subtype of Set (predicative and impredicative forms).
- Recursive inductive types in Prop with a single constructor of which all arguments are in Prop is now considered to be a singleton type. It consequently supports all eliminations to Prop, Set and Type. As a consequence, Acc_rect has now a more direct proof [possible source of easily fixed incompatibility in case of manual definition of a recursor in a recursive singleton inductive type].

Commands

- Added option Global to "Arguments Scope" for section surviving.
- Added option "Unset Elimination Schemes" to deactivate the automatic generation of elimination schemes.
- Modification of the Scheme command so you can ask for the name to be automatically computed (e.g. Scheme Induction for nat Sort Set).
- New command "Combined Scheme" to build combined mutual induction principles from existing mutual induction principles.
- New command "Scheme Equality" to build a decidable (boolean) equality for simple inductive datatypes and a decision property over this equality (e.g. Scheme Equality for nat).
- Added option "Set Equality Scheme" to make automatic the declaration of the boolean equality when possible.
- Source of universe inconsistencies now printed when option "Set Printing Universes" is activated.
- New option "Set Printing Existential Instances" for making the display of existential variable instances explicit.
- Support for option "[id1 ... idn]", and "-[id1 ... idn]", for the "compute"/"cbv" reduction strategy, respectively meaning reduce only, or everything but, the constants id1 ... idn. "lazy" alone or followed by "[id1 ... idn]", and "-[id1 ... idn]" also supported, meaning apply all of beta-iota-zeta-delta, possibly restricting delta.
- New command "Strategy" to control the expansion of constants during conversion tests. It generalizes commands Opaque and Transparent by introducing a range of levels. Lower levels are assigned to constants that should be expanded first.
- New options Global and Local to Opaque and Transparent.
- New command "Print Assumptions" to display all variables, parameters or axioms a theorem or definition relies on.
- "Add Rec LoadPath" now provides references to libraries using partially qualified names (this holds also for coqtop/coqc option -R).
- SearchAbout supports negated search criteria, reference to logical objects by their notation, and more generally search of subterms.
- "Declare ML Module" now allows to import .cmxs files when Coq is compiled in native code with a version of OCaml that supports native Dynlink (>= 3.11).
- Specific sort constraints on Record now taken into account.
- "Print LoadPath" supports a path argument to filter the display.

Libraries

- Several parts of the libraries are now in Type, in particular FSets, SetoidList, ListSet, Sorting, Zmisc. This may induce a few incompatibilities. In case of trouble while fixing existing development, it may help to simply declare Set as an alias for Type (see file SetIsType).
- New arithmetical library in theories/Numbers. It contains:
  - an abstract modular development of natural and integer arithmetics in Numbers/Natural/Abstract and Numbers/Integer/Abstract
  - an implementation of efficient computational bounded and unbounded integers that can be mapped to processor native arithmetics. See Numbers/Cyclic/Int31 for 31-bit integers and Numbers/Natural/BigN for unbounded natural numbers and Numbers/Integer/BigZ for unbounded integers.
  - some proofs that both older libraries Arith, ZArith and NArith and newer BigN and BigZ implement the abstract modular development. This allows in particular BigN and BigZ to already come with a large database of basic lemmas and some generic tactics (ring),

This library has still an experimental status, as well as the processor-acceleration mechanism, but both its abstract and its concrete parts are already quite usable and could challenge the use of nat, N and Z in actual developments. Moreover, an extension of this framework to rational numbers is ongoing, and an efficient Q structure is already provided (see Numbers/Rational/BigQ), but this part is currently incomplete (no abstract layer and generic lemmas).

- Many changes in FSets/FMaps. In practice, compatibility with earlier version should be fairly good, but some adaptations may be required.
  - Interfaces of unordered ("weak") and ordered sets have been factorized thanks to new features of Coq modules (in particular Include), see FSetInterface. Same for maps. Hints in these interfaces have been reworked (they are now placed in a "set" database).
  - To allow full subtyping between weak and ordered sets, a field "eq_dec" has been added to OrderedType. The old version of OrderedType is now called MiniOrderedType and functor MOT_to_OT allow to convert to the new version. The interfaces and implementations of sets now contain also such a "eq_dec" field.
  - FSetDecide, contributed by Aaron Bohannon, contains a decision procedure allowing to solve basic set-related goals (for instance, is a point in a particular set ?). See FSetProperties for examples.
  - Functors of properties have been improved, especially the ones about maps, that now propose some induction principles. Some properties of fold need less hypothesis.
  - More uniformity in implementations of sets and maps: they all use implicit arguments, and no longer export unnecessary scopes (see bug #1347)
  - Internal parts of the implementations based on AVL have evolved a lot. The main files FSetAVL and FMapAVL are now much more lightweight now. In particular, minor changes in some functions has allowed to fully separate the proofs of operational correctness from the proofs of well-balancing: well-balancing is critical for efficiency, but not anymore for proving that these trees implement our interfaces, hence we have moved these proofs into appendix files FSetFullAVL and FMapFullAVL. Moreover, a few functions like union and compare have been modified in order to be structural yet efficient. The appendix files also contains alternative versions of these few functions, much closer to the initial Ocaml code and written via the Function framework.
- Library IntMap, subsumed by FSets/FMaps, has been removed from Coq Standard Library and moved into a user contribution Cachan/IntMap
- Better computational behavior of some constants (eq_nat_dec and le_lt_dec more efficient, Z_lt_le_dec and Positive_as_OT.compare transparent, ...) (exceptional source of incompatibilities).
- Boolean operators moved from module Bool to module Datatypes (may need to rename qualified references in script and force notations || and && to be at levels 50 and 40 respectively).
- The constructors xI and xO of type positive now have postfix notations "~1" and "~0", allowing to write numbers in binary form easily, for instance 6 is 1~1~0 and 4*p is p~0~0 (see BinPos.v).
- Improvements to NArith (Nminus, Nmin, Nmax), and to QArith (in particular a better power function).
- Changes in ZArith: several additional lemmas (used in theories/Numbers), especially in Zdiv, Znumtheory, Zpower. Moreover, many results in Zdiv have been generalized: the divisor may simply be non-null instead of strictly positive (see lemmas with name ending by "_full"). An alternative file ZOdiv proposes a different behavior (the one of Ocaml) when dividing by negative numbers.
- Changes in Arith: EqNat and Wf_nat now exported from Arith, some constructions on nat that were outside Arith are now in (e.g. iter_nat).
- In SetoidList, eqlistA now expresses that two lists have similar elements at the same position, while the predicate previously called eqlistA is now equivlistA (this one only states that the lists contain the same elements, nothing more).
- Changes in Reals:

  - Most statement in "sigT" (including the completeness axiom) are now in "sig" (in case of incompatibility, use proj1_sig instead of projT1, sig instead of sigT, etc).
  - More uniform naming scheme (identifiers in French moved to English, consistent use of 0 -- zero -- instead of O -- letter O --, etc).
  - Lemma on prod_f_SO is now on prod_f_R0.
  - Useless hypothesis of ln_exists1 dropped.
  - New Rlogic.v states a few logical properties about R axioms.
  - RIneq.v extended and made cleaner.

- Slight restructuration of the Logic library regarding choice and classical logic. Addition of files providing intuitionistic axiomatizations of descriptions: Epsilon.v, Description.v and IndefiniteDescription.v.
- Definition of pred and minus made compatible with the structural decreasing criterion for use in fixpoints.
- Files Relations/Rstar.v and Relations/Newman.v moved out to the user contribution repository (contribution CoC_History). New lemmas about transitive closure added and some bound variables renamed (exceptional risk of incompatibilities).
- Syntax for binders in terms (e.g. for "exists") supports anonymous names.

Notations, coercions, implicit arguments and type inference

- More automation in the inference of the return clause of dependent pattern-matching problems.
- Experimental allowance for omission of the clauses easily detectable as impossible in pattern-matching problems.
- Improved inference of implicit arguments.
- New options "Set Maximal Implicit Insertion", "Set Reversible Pattern Implicit", "Set Strongly Strict Implicit" and "Set Printing Implicit Defensive" for controlling inference and use of implicit arguments.
- New modifier in "Implicit Arguments" to force an implicit argument to be maximally inserted.
- New modifier of "Implicit Arguments" to enrich the set of implicit arguments.
- New options Global and Local to "Implicit Arguments" for section surviving or non export outside module.
- Level "constr" moved from 9 to 8.
- Structure/Record now printed as Record (unless option Printing All is set).
- Support for parametric notations defining constants.
- Insertion of coercions below product types refrains to unfold constants (possible source of incompatibility).
- New support for fix/cofix in notations.

Tactic Language

- Second-order pattern-matching now working in Ltac "match" clauses (syntax for second-order unification variable is "@?X").
- Support for matching on let bindings in match context using syntax "H := body" or "H := body : type".
- Ltac accepts integer arguments (syntax is "ltac:nnn" for nnn an integer).
- The general sequence tactical "expr_0 ; [expr_1 | ... | expr_n]" is extended so that at most one expr_i may have the form "expr .." or just "..". Also, n can be different from the number of subgoals generated by expr_0. In this case, the value of expr (or idtac in case of just "..") is applied to the intermediate subgoals to make the number of tactics equal to the number of subgoals.

- A name used as the name of the parameter of a lemma (like f in ”apply f_equal with (f:=t)”) is now interpreted as a ltac variable if such a variable exists (this is a possible source of incompatibility and it can be fixed by renaming the variables of a ltac function into names that do not clash with the lemmas parameter names used in the tactic).
- New syntax ”Ltac tac ::= ...” to rebind a tactic to a new expression.
- ”let rec ... in ... ” now supported for expressions without explicit parameters; interpretation is lazy to the contrary of ”let ... in ...”; hence, the ”rec” keyword can be used to turn the argument of a ”let ... in ...” into a lazy one.
- Patterns for hypotheses types in ”match goal” are now interpreted in type_scope.
- A bound variable whose name is not used elsewhere now serves as metavariable in ”match” and it gets instantiated by an identifier (allow e.g. to extract the name of a statement like ”exists x, P x”).
- New printing of Ltac call trace for better debugging.

Tactics

- New tactics ”apply -> term”, ”apply <- term”, ”apply -> term in ident”, ”apply <- term in ident” for applying equivalences (iff).
- Slight improvement of the hnf and simpl tactics when applied on expressions with explicit occurrences of match or fix.
- New tactics ”eapply in”, ”erewrite”, ”erewrite in”.
- New tactics ”ediscriminate”, ”einjection”, ”esimplify_eq”.
- Tactics ”discriminate”, ”injection”, ”simplify_eq” now support any term as argument. Clause ”with” is also supported.
- Unfoldable references can be given by notation’s string rather than by name in unfold.
- The ”with” arguments are now typed using informations from the current goal: allows support for coercions and more inference of implicit arguments.
- Application of ”f_equal”-style lemmas works better.
- Tactics elim, case, destruct and induction now support variants eelim, ecase, edestruct and einduction.
- Tactics destruct and induction now support the ”with” option and the ”in” clause option. If the option ”in” is used, an equality is added to remember the term to which the induction or case analysis applied (possible source of parsing incompatibilities when destruct or induction is part of a let-in expression in Ltac; extra parentheses are then required).
- New support for ”as” clause in tactics ”apply in” and ”eapply in”.
- Some new intro patterns:
  - intro pattern ”?A” genererates a fresh name based on A. Caveat about a slight loss of compatibility: Some intro patterns don’t need space between them. In particular intros ?a?b used to be legal and equivalent to intros ? a ? b. Now it is still legal but equivalent to intros ?a ?b.
  - intro pattern ”(A & ... & Y & Z)” synonym to ”(A,....,(Y,Z)))))” for right-associative constructs like /or exists.
- Several syntax extensions concerning ”rewrite”:
  - ”rewrite A,B,C” can be used to rewrite A, then B, then C. These rewrites occur only on the first subgoal: in particular, side-conditions of the ”rewrite A” are not concerned by the ”rewrite B,C”.
  - ”rewrite A by tac” allows to apply tac on all side-conditions generated by the ”rewrite A”.
  - ”rewrite A at n” allows to select occurrences to rewrite: rewrite only happen at the n-th exact occurrence of the first successful matching of A in the goal.
  - ”rewrite 3 A” or ”rewrite 3!A” is equivalent to ”rewrite A,A,A”.

  - ”rewrite !A” means rewriting A as long as possible (and at least once).
  - ”rewrite 3?A” means rewriting A at most three times.
  - ”rewrite ?A” means rewriting A as long as possible (possibly never).
  - many of the above extensions can be combined with each other.
- Introduction patterns better respect the structure of context in presence of missing or extra names in nested disjunction-conjunction patterns [possible source of rare incompatibilities].
- New syntax ”rename a into b, c into d” for ”rename a into b; rename c into d”
- New tactics ”dependent induction/destruction H [generalizing id_1 .. id_n]” to do induction-inversion on instantiated inductive families à la BasicElim.
- Tactics ”apply” and ”apply in” now able to reason modulo unfolding of constants (possible source of incompatibility in situations where apply may fail, e.g. as argument of a try or a repeat and in a ltac function); versions that do not unfold are renamed into ”simple apply” and ”simple apply in” (usable for compatibility or for automation).
- Tactics ”apply” and ”apply in” now able to traverse conjunctions and to select the first matching lemma among the components of the conjunction; tactic ”apply” also able to apply lemmas of conclusion an empty type.
- Tactic ”apply” now supports application of several lemmas in a row.
- Tactics ”set” and ”pose” can set functions using notation ”(f x1..xn := c)”.
- New tactic ”instantiate” (without argument).
- Tactic firstorder ”with” and ”using” options have their meaning swapped for consistency with auto/eauto (source of incompatibility).
- Tactic ”generalize” now supports ”at” options to specify occurrences and ”as” options to name the quantified hypotheses.
- New tactic ”specialize H with a” or ”specialize (H a)” allows to transform in-place a universally-quantified hypothesis (H : forall x, T x) into its instantiated form (H : T a). Nota: ”specialize” was in fact there in earlier versions of Coq, but was undocumented, and had a slightly different behavior.
- New tactic ”contradict H” can be used to solve any kind of goal as long as the user can provide afterwards a proof of the negation of the hypothesis H. If H is already a negation, say ~T, then a proof of T is asked. If the current goal is a negation, say ~U, then U is saved in H afterwards, hence this new tactic ”contradict” extends earlier tactic ”swap”, which is now obsolete.
- Tactics f_equal is now done in ML instead of Ltac: it now works on any equality of functions, regardless of the arity of the function.
- New options ”before id”, ”at top”, ”at bottom” for tactics ”move”/”intro”.
- Some more debug of reflexive omega (`romega`), and internal clarifications. Moreover, romega now has a variant `romega with *` that can be also used on non-Z goals (nat, N, positive) via a call to a translation tactic named zify (its purpose is to Z-ify your goal...). This zify may also be used independently of romega.
- Tactic ”remember” now supports an ”in” clause to remember only selected occurrences of a term.
- Tactic ”pose proof” supports name overriding in case of specialization of an hypothesis.
- Semi-decision tactic ”jp” for first-order intuitionistic logic moved to user contributions (subsumed by ”firstorder”).

Program

- Moved useful tactics in theories/Program and documented them.
- Add Program.Basics which contains standard definitions for functional programming (id, apply, flip...)
- More robust obligation handling, dependent pattern-matching and well-founded definitions.

- New syntax ” dest term as pat in term ” for destructing objects using an irrefutable pattern while keeping equalities (use this instead of ”let” in Programs).
- Program CoFixpoint is accepted, Program Fixpoint uses the new way to infer which argument decreases structurally.
- Program Lemma, Axiom etc... now permit to have obligations in the statement iff they can be automatically solved by the default tactic.
- Renamed ”Obligations Tactic” command to ”Obligation Tactic”.
- New command ”Preterm [of id]” to see the actual term fed to Coq for debugging purposes.
- New option ”Transparent Obligations” to control the declaration of obligations as transparent or opaque. All obligations are now transparent by default, otherwise the system declares them opaque if possible.
- Changed the notations ”left” and ”right” to ”in_left” and ”in_right” to hide the proofs in standard disjunctions, to avoid breaking existing scripts when importing Program. Also, put them in program_scope.

Type Classes

- New ”Class”, ”Instance” and ”Program Instance” commands to define classes and instances documented in the reference manual.
- New binding construct ” [Class_1 param_1 .. param_n, Class_2 ...] ” for binding type classes, usable everywhere.
- New command ” Print Classes ” and ” Print Instances some_class ” to print tables for typeclasses.
- New default eauto hint database ”typeclass_instances” used by the default typeclass instance search tactic.
- New theories directory ”theories/Classes” for standard typeclasses declarations. Module Classes.RelationClasses is a typeclass port of Relation_Definitions plus a generic development of algebra on n-ary heterogeneous predicates.

Setoid rewriting

- Complete (and still experimental) rewrite of the tactic based on typeclasses. The old interface and semantics are almost entirely respected, except:
  - Import Setoid is now mandatory to be able to call setoid_replace and declare morphisms.
  - ”-->”, ”++>” and ”==>” are now right associative notations declared at level 55 in scope signature_scope. Their introduction may break existing scripts that defined them as notations with different levels.
  - One needs to use [Typeclasses unfold [cst]] if [cst] is used as an abbreviation hiding products in types of morphisms, e.g. if ones redefines [relation] and declares morphisms whose type mentions [relation].
  - The [setoid_rewrite]’s semantics change when rewriting with a lemma: it can rewrite two different instantiations of the lemma at once. Use [setoid_rewrite H at 1] for (almost) the usual semantics. [setoid_rewrite] will also try to rewrite under binders now, and can succeed on different terms than before. In particular, it will unify under let-bound variables. When called through [rewrite], the semantics are unchanged though.
  - [Add Morphism term : id] has different semantics when used with parametric morphism: it will try to find a relation on the parameters too. The behavior has also changed with respect to default relations: the most recently declared Setoid/Relation will be used, the documentation explains how to customize this behavior.
  - Parametric Relation and Morphism are declared differently, using the new [Add Parametric] commands, documented in the manual.
  - Setoid_Theory is now an alias to Equivalence, scripts building objects of type Setoid_Theory need to unfold (or ”red”) the definitions of Reflexive, Symmetric and Transitive in order to get the same goals as before. Scripts which introduced variables explicitly will not break.
  - The order of subgoals when doing [setoid_rewrite] with side-conditions is always the same: first the new goal, then the conditions.

- New standard library modules `Classes.Morphisms` declares standard morphisms on `refl` / `sym` / `trans` relations. `Classes.Morphisms_Prop` declares morphisms on propositional connectives and `Classes.Morphisms_Relations` on generalized predicate connectives. `Classes.Equivalence` declares notations and tactics related to equivalences and `Classes.SetoidTactics` defines the setoid_replace tactics and some support for the `Add *` interface, notably the tactic applied automatically before each `Add Morphism` proof.
- User-defined subrelations are supported, as well as higher-order morphisms and rewriting under binders. The tactic is also extensible entirely in Ltac. The documentation has been updated to cover these features.
- [setoid_rewrite] and [rewrite] now support the [at] modifier to select occurrences to rewrite, and both use the [setoid_rewrite] code, even when rewriting with leibniz equality if occurrences are specified.

Extraction

- Improved behavior of the Caml extraction of modules: name clashes should not happen anymore.
- The command Extract Inductive has now a syntax for infix notations. This allows in particular to map Coq lists and pairs onto OCaml ones:
  - Extract Inductive list => list ["[]" "(::)"].
  - Extract Inductive prod => "(*)" ["(,)"].
- In pattern matchings, a default pattern "| _ -> ..." is now used whenever possible if several branches are identical. For instance, functions corresponding to decidability of equalities are now linear instead of quadratic.
- A new instruction Extraction Blacklist id1 .. idn allows to prevent filename conflits with existing code, for instance when extracting module List to Ocaml.

CoqIDE

- CoqIDE font defaults to monospace so as indentation to be meaningful.
- CoqIDE supports nested goals and any other kind of declaration in the middle of a proof.
- Undoing non-tactic commands in CoqIDE works faster.
- New CoqIDE menu for activating display of various implicit informations.
- Added the possibility to choose the location of tabs in coqide: (in Edit->Preferences->Misc)
- New Open and Save As dialogs in CoqIDE which filter `*.v` files.

Tools

- New stand-alone .vo files verifier "coqchk".
- Extended -I coqtop/coqc option to specify a logical dir: "-I dir -as coqdir".
- New coqtop/coqc option -exclude-dir to exclude subdirs for option -R.
- The binary "parser" has been renamed to "coq-parser".
- Improved coqdoc and dump of globalization information to give more meta-information on identifiers. All categories of Coq definitions are supported, which makes typesetting trivial in the generated documentation. Support for hyperlinking and indexing developments in the tex output has been implemented as well.

Miscellaneous

- Coq installation provides enough files so that Ocaml's extensions need not the Coq sources to be compiled (this assumes O'Caml 3.10 and Camlp5).
- New commands "Set Whelp Server" and "Set Whelp Getter" to customize the Whelp search tool.
- Syntax of "Test Printing Let ref" and "Test Printing If ref" changed into "Test Printing Let for ref" and "Test Printing If for ref".

- An overhauled build system (new Makefiles); see dev/doc/build-system.txt.
- Add -browser option to configure script.
- Build a shared library for the C part of Coq, and use it by default on non-(Windows or MacOS) systems. Bytecode executables are now pure. The behaviour is configurable with -coqrunbyteflags, -coqtoolsbyteflags and -custom configure options.
- Complexity tests can be skipped by setting the environment variable COQTEST_SKIPCOMPLEXITY.

Version 8.1

Summary of changes

Coq version 8.1 adds various new functionalities.

Benjamin Grégoire implemented an alternative algorithm to check the convertibility of terms in the Coq type checker. This alternative algorithm works by compilation to an efficient bytecode that is interpreted in an abstract machine similar to Xavier Leroy's ZINC machine. Convertibility is performed by comparing the normal forms. This alternative algorithm is specifically interesting for proofs by reflection. More generally, it is convenient in case of intensive computations.

Christine Paulin implemented an extension of inductive types allowing recursively non uniform parameters. Hugo Herbelin implemented sort-polymorphism for inductive types (now called template polymorphism).

Claudio Sacerdoti Coen improved the tactics for rewriting on arbitrary compatible equivalence relations. He also generalized rewriting to arbitrary transition systems.

Claudio Sacerdoti Coen added new features to the module system.

Benjamin Grégoire, Assia Mahboubi and Bruno Barras developed a new, more efficient and more general simplification algorithm for rings and semirings.

Laurent Théry and Bruno Barras developed a new, significantly more efficient simplification algorithm for fields.

Hugo Herbelin, Pierre Letouzey, Julien Forest, Julien Narboux and Claudio Sacerdoti Coen added new tactic features.

Hugo Herbelin implemented matching on disjunctive patterns.

New mechanisms made easier the communication between Coq and external provers. Nicolas Ayache and Jean-Christophe Filliâtre implemented connections with the provers cvcl, Simplify and zenon. Hugo Herbelin implemented an experimental protocol for calling external tools from the tactic language.

Matthieu Sozeau developed Russell, an experimental language to specify the behavior of programs with subtypes.

A mechanism to automatically use some specific tactic to solve unresolved implicit has been implemented by Hugo Herbelin.

Laurent Théry's contribution on strings and Pierre Letouzey and Jean-Christophe Filliâtre's contribution on finite maps have been integrated to the Coq standard library. Pierre Letouzey developed a library about finite sets "à la Objective Caml". With Jean-Marc Notin, he extended the library on lists. Pierre Letouzey's contribution on rational numbers has been integrated and extended.

Pierre Corbineau extended his tactic for solving first-order statements. He wrote a reflection-based intuitionistic tautology solver.

Pierre Courtieu, Julien Forest and Yves Bertot added extra support to reason on the inductive structure of recursively defined functions.

Jean-Marc Notin significantly contributed to the general maintenance of the system. He also took care of `coqdoc`.

Pierre Castéran contributed to the documentation of (co-)inductive types and suggested improvements to the libraries.

Pierre Corbineau implemented a declarative mathematical proof language, usable in combination with the tactic-based style of proof.

Finally, many users suggested improvements of the system through the Coq-Club mailing list and bug-tracker systems, especially user groups from INRIA Rocquencourt, Radboud University, University of Pennsylvania and Yale University.

Palaiseau, July 2006
Hugo Herbelin

Details of changes in 8.1beta

Logic

- Added sort-polymorphism on inductive families
- Allowance for recursively non uniform parameters in inductive types

Syntax

- No more support for version 7 syntax and for translation to version 8 syntax.
- In fixpoints, the { struct ... } annotation is not mandatory any more when only one of the arguments has an inductive type
- Added disjunctive patterns in match-with patterns
- Support for primitive interpretation of string literals
- Extended support for Unicode ranges

Commands

- Added ”Print Ltac qualid” to print a user defined tactic.
- Added ”Print Rewrite HintDb” to print the content of a DB used by autorewrite.
- Added ”Print Canonical Projections”.
- Added ”Example” as synonym of ”Definition”.
- Added ”Proposition” and ”Corollary” as extra synonyms of ”Lemma”.
- New command ”Whelp” to send requests to the Helm database of proofs formalized in the Calculus of Inductive Constructions.
- Command ”functional induction” has been re-implemented from the new ”Function” command.

Ltac and tactic syntactic extensions

- New primitive ”external” for communication with tool external to Coq
- New semantics for ”match t with”: if a clause returns a tactic, it is now applied to the current goal. If it fails, the next clause or next matching subterm is tried (i.e. it behaves as ”match goal with” does). The keyword ”lazymatch” can be used to delay the evaluation of tactics occurring in matching clauses.
- Hint base names can be parametric in auto and trivial.
- Occurrence values can be parametric in unfold, pattern, etc.
- Added entry constr_may_eval for tactic extensions.
- Low-priority term printer made available in ML-written tactic extensions.

- ”Tactic Notation” extended to allow notations of tacticals.

Tactics

- New implementation and generalization of `setoid_*` (`setoid_rewrite`, `setoid_symmetry`, `setoid_transitivity`, `setoid_reflexivity` and `autorewite`). New syntax for declaring relations and morphisms (old syntax still working with minor modifications, but deprecated).
- New implementation (still experimental) of the ring tactic with a built-in notion of coefficients and a better usage of setoids.
- New conversion tactic ”vm_compute”: evaluates the goal (or an hypothesis) with a call-by-value strategy, using the compiled version of terms.
- When rewriting H where H is not directly a Coq equality, search first H for a registered setoid equality before starting to reduce in H. This is unlikely to break any script. Should this happen nonetheless, one can insert manually some ”unfold ... in H” before rewriting.
- Fixed various bugs about (setoid) rewrite ... in ... (in particular bug #5941)
- ”rewrite ... in” now accepts a clause as place where to rewrite instead of just a simple hypothesis name. For instance: `rewrite H in H1,H2 |- *` means `rewrite H in H1; rewrite H in H2; rewrite H rewrite H in * |-` will do try `rewrite H in Hi` for all hypothesis Hi <> H.
- Added ”dependent rewrite term” and ”dependent rewrite term in hyp”.
- Added ”autorewrite with ... in hyp [using ...]”.
- Tactic ”replace” now accepts a ”by” tactic clause.
- Added ”clear - id” to clear all hypotheses except the ones depending in id.
- The argument of Declare Left Step and Declare Right Step is now a term (it used to be a reference).
- Omega now handles arbitrary precision integers.
- Several bug fixes in Reflexive Omega (romega).
- Idtac can now be left implicit in a [...|...] construct: for instance, [foo | | bar] stands for [foo | idtac | bar].
- Fixed a ”fold” bug (non critical but possible source of incompatibilities).
- Added classical_left and classical_right which transforms `|- A \/ B` into `~B |- A` and `~A |- B` respectively.
- Added command ”Declare Implicit Tactic” to set up a default tactic to be used to solve unresolved subterms of term arguments of tactics.
- Better support for coercions to Sortclass in tactics expecting type arguments.
- Tactic ”assert” now accepts ”as” intro patterns and ”by” tactic clauses.
- New tactic ”pose proof” that generalizes ”assert (id:=p)” with intro patterns.
- New introduction pattern ”?” for letting Coq choose a name.
- Introduction patterns now support side hypotheses (e.g. intros [|] on ”(nat -> nat) -> nat” works).
- New introduction patterns ”->” and ”<-” for immediate rewriting of introduced hypotheses.
- Introduction patterns coming after non trivial introduction patterns now force full introduction of the first pattern (e.g. `intros [[|] p]` on `nat->nat->nat` now behaves like `intros [[|?] p]`)
- Added ”eassumption”.
- Added option ’using lemmas’ to auto, trivial and eauto.

- Tactic ”congruence” is now complete for its intended scope (ground equalities and inequalities with constructors). Furthermore, it tries to equates goal and hypotheses.
- New tactic ”rtauto” solves pure propositional logic and gives a reflective version of the available proof.
- Numbering of ”pattern”, ”unfold”, ”simpl”, ... occurrences in ”match with” made consistent with the printing of the return clause after the term to match in the ”match-with” construct (use ”Set Printing All” to see hidden occurrences).
- Generalization of induction ”induction x1...xn using scheme” where scheme is an induction principle with complex predicates (like the ones generated by function induction).
- Some small Ltac tactics has been added to the standard library (file Tactics.v):
  - f_equal : instead of using the different f_equalX lemmas
  - case_eq : a ”case” without loss of information. An equality stating the current situation is generated in every sub-cases.
  - swap : for a negated goal ~B and a negated hypothesis H:~A, swap H asks you to prove A from hypothesis B
  - revert : revert H is generalize H; clear H.

Extraction

- All type parts should now disappear instead of sometimes producing _ (for instance in Map.empty).
- Haskell extraction: types of functions are now printed, better unsafeCoerce mechanism, both for hugs and ghc.
- Scheme extraction improved, see http://www.pps.jussieu.fr/~letouzey/scheme.
- Many bug fixes.

Modules

- Added ”Locate Module qualid” to get the full path of a module.
- Module/Declare Module syntax made more uniform.
- Added syntactic sugar ”Declare Module Export/Import” and ”Module Export/Import”.
- Added syntactic sugar ”Module M(Export/Import X Y: T)” and ”Module Type M(Export/Import X Y: T)” (only for interactive definitions)
- Construct ”with” generalized to module paths: T with (Definition|Module) M1.M2....Mn.l := l’.

Notations

- Option ”format” aware of recursive notations.
- Added insertion of spaces by default in recursive notations w/o separators.
- No more automatic printing box in case of user-provided printing ”format”.
- New notation ”exists! x:A, P” for unique existence.
- Notations for specific numerals now compatible with generic notations of numerals (e.g. ”1” can be used to denote the unit of a group without hiding 1%nat)

Libraries

- New library on String and Ascii characters (contributed by L. Thery).
- New library FSets+FMaps of finite sets and maps.
- New library QArith on rational numbers.
- Small extension of Zmin.V, new Zmax.v, new Zminmax.v.

- Reworking and extension of the files on classical logic and description principles (possible incompatibilities)
- Few other improvements in ZArith potentially exceptionally breaking the compatibility (useless hypothesys of Zgt_square_simpl and Zlt_square_simpl removed; fixed names mentioning letter O instead of digit 0; weaken premises in Z_lt_induction).
- Restructuration of Eqdep_dec.v and Eqdep.v: more lemmas in Type.
- Znumtheory now contains a gcd function that can compute within Coq.
- More lemmas stated on Type in Wf.v, removal of redundant Acc_iter and Acc_iter2.
- Change of the internal names of lemmas in OmegaLemmas.
- Acc in Wf.v and clos_refl_trans in Relation_Operators.v now rely on the allowance for recursively non uniform parameters (possible source of incompatibilities: explicit pattern-matching on these types may require to remove the occurrence associated with their recursively non uniform parameter).
- Coq.List.In_dec has been set transparent (this may exceptionally break proof scripts, set it locally opaque for compatibility).
- More on permutations of lists in List.v and Permutation.v.
- List.v has been much expanded.
- New file SetoidList.v now contains results about lists seen with respect to a setoid equality.
- Library NArith has been expanded, mostly with results coming from Intmap (for instance a bitwise xor), plus also a bridge between N and Bitvector.
- Intmap has been reorganized. In particular its address type "addr" is now N. User contributions known to use Intmap have been adapted accordingly. If you're using this library please contact us. A wrapper FMapIntMap now presents Intmap as a particular implementation of FMaps. New developments are strongly encouraged to use either this wrapper or any other implementations of FMap instead of using directly this obsolete Intmap.

Tools

- New semantics for coqtop options ("-batch" expects option "-top dir" for loading vernac file that contains definitions).
- Tool coq_makefile now removes custom targets that are file names in "make clean"
- New environment variable COQREMOTEBROWSER to set the command invoked to start the remote browser both in Coq and CoqIDE. Standard syntax: "%s" is the placeholder for the URL.

Details of changes in 8.1gamma

Syntax

- changed parsing precedence of let/in and fun constructions of Ltac: let x := t in e1; e2 is now parsed as let x := t in (e1;e2).

Language and commands

- Added sort-polymorphism for definitions in Type (but finally abandoned).
- Support for implicit arguments in the types of parameters in (co-)fixpoints and (co-)inductive declarations.
- Improved type inference: use as much of possible general information. before applying irreversible unification heuristics (allow e.g. to infer the predicate in "(exist _ 0 (refl_equal 0) : {n:nat | n=0 })").
- Support for Miller-Pfenning's patterns unification in type synthesis (e.g. can infer P such that P x y = phi(x,y)).
- Support for "where" clause in cofixpoint definitions.

- New option ”Set Printing Universes” for making Type levels explicit.

Tactics

- Improved implementation of the ring and field tactics. For compatibility reasons, the previous tactics are renamed as legacy ring and legacy field, but should be considered as deprecated.
- New declarative mathematical proof language.
- Support for argument lists of arbitrary length in Tactic Notation.
- `rewrite ... in H` now fails if `H` is used either in an hypothesis or in the goal.
- The semantics of `rewrite ... in *` has been slightly modified (see doc).
- Support for `as` clause in tactic injection.
- New forward-reasoning tactic ”apply in”.
- Ltac fresh operator now builds names from a concatenation of its arguments.
- New ltac tactic ”remember” to abstract over a subterm and keep an equality
- Support for Miller-Pfenning's patterns unification in apply/rewrite/... (may lead to few incompatibilities - generally now useless tactic calls).

Bug fixes

- Fix for notations involving basic ”match” expressions.
- Numerous other bugs solved (a few fixes may lead to incompatibilities).

Details of changes in 8.1

Bug fixes

- Many bugs have been fixed (cf coq-bugs web page)

Tactics

- New tactics ring, ring_simplify and new tactic field now able to manage power to a positive integer constant. Tactic ring on Z and R, and field on R manage power (may lead to incompatibilities with V8.1gamma).
- Tactic field_simplify now applicable in hypotheses.
- New field_simplify_eq for simplifying field equations into ring equations.
- Tactics ring, ring_simplify, field, field_simplify and field_simplify_eq all able to apply user-given equations to rewrite monoms on the fly (see documentation).

Libraries

- New file ConstructiveEpsilon.v defining an epsilon operator and proving the axiom of choice constructively for a countable domain and a decidable predicate.

Version 8.0

Summary of changes

Coq version 8 is a major revision of the Coq proof assistant. First, the underlying logic is slightly different. The so-called *impredicativity* of the sort Set has been dropped. The main reason is that it is inconsistent with the principle of description which is quite a useful principle for formalizing mathematics within classical logic. Moreover, even in an constructive setting, the impredicativity of Set does not add so much in practice and is even subject of criticism from a large part of the intuitionistic mathematician community. Nevertheless, the impredicativity of Set remains optional for users interested in investigating mathematical developments which rely on it.

Secondly, the concrete syntax of terms has been completely revised. The main motivations were

- a more uniform, purified style: all constructions are now lowercase, with a functional programming perfume (e.g. abstraction is now written fun), and more directly accessible to the novice (e.g. dependent product is now written forall and allows omission of types). Also, parentheses are no longer mandatory for function application.
- extensibility: some standard notations (e.g. "<" and ">") were incompatible with the previous syntax. Now all standard arithmetic notations (=, +, *, /, <, <=, ... and more) are directly part of the syntax.

Together with the revision of the concrete syntax, a new mechanism of *notation scopes* permits to reuse the same symbols (typically +, -, *, /, <, <=) in various mathematical theories without any ambiguities for Coq, leading to a largely improved readability of Coq scripts. New commands to easily add new symbols are also provided.

Coming with the new syntax of terms, a slight reform of the tactic language and of the language of commands has been carried out. The purpose here is a better uniformity making the tactics and commands easier to use and to remember.

Thirdly, a restructuring and uniformization of the standard library of Coq has been performed. There is now just one Leibniz equality usable for all the different kinds of Coq objects. Also, the set of real numbers now lies at the same level as the sets of natural and integer numbers. Finally, the names of the standard properties of numbers now follow a standard pattern and the symbolic notations for the standard definitions as well.

The fourth point is the release of CoqIDE, a new graphical gtk2-based interface fully integrated with Coq. Close in style to the Proof General Emacs interface, it is faster and its integration with Coq makes interactive developments more friendly. All mathematical Unicode symbols are usable within CoqIDE.

Finally, the module system of Coq completes the picture of Coq version 8.0. Though released with an experimental status in the previous version 7.4, it should be considered as a salient feature of the new version.

Besides, Coq comes with its load of novelties and improvements: new or improved tactics (including a new tactic for solving first-order statements), new management commands, extended libraries.

Bruno Barras and Hugo Herbelin have been the main contributors of the reflection and the implementation of the new syntax. The smart automatic translator from old to new syntax released with Coq is also their work with contributions by Olivier Desmettre.

Hugo Herbelin is the main designer and implementer of the notion of notation scopes and of the commands for easily adding new notations.

Hugo Herbelin is the main implementer of the restructured standard library.

Pierre Corbineau is the main designer and implementer of the new tactic for solving first-order statements in presence of inductive types. He is also the maintainer of the non-domain specific automation tactics.

Benjamin Monate is the developer of the CoqIDE graphical interface with contributions by Jean-Christophe Filliâtre, Pierre Letouzey, Claude Marché and Bruno Barras.

Claude Marché coordinated the edition of the Reference Manual for Coq V8.0.

Pierre Letouzey and Jacek Chrząszcz respectively maintained the extraction tool and module system of Coq.

Jean-Christophe Filliâtre, Pierre Letouzey, Hugo Herbelin and other contributors from Sophia-Antipolis and Nijmegen participated in extending the library.

Julien Narboux built a NSIS-based automatic Coq installation tool for the Windows platform.

Hugo Herbelin and Christine Paulin coordinated the development which was under the responsibility of Christine Paulin.

Palaiseau & Orsay, Apr. 2004
Hugo Herbelin & Christine Paulin
(updated Apr. 2006)

Details of changes in 8.0beta old syntax

Logic

- Set now predicative by default
- New option -impredicative-set to set Set impredicative
- The standard library doesn't need impredicativity of Set and is compatible with the classical axioms which contradict Set impredicativity

Syntax for arithmetic

- Notation "=" and "<>" in Z and R are no longer implicitly in Z or R (with possible introduction of a coercion), use <Z>...=... or <Z>...<>... instead
- Locate applied to a simple string (e.g. "+") searches for all notations containing this string

Commands

- "Declare ML Module" now allows to import .cma files. This avoids to use a bunch of "Declare ML Module" statements when using several ML files.
- "Set Printing Width n" added, allows to change the size of width printing.
- "Implicit Variables Type x,y:t" (new syntax: "Implicit Types x y:t") assigns default types for binding variables.
- Declarations of Hints and Notation now accept a "Local" flag not to be exported outside the current file even if not in section
- "Print Scopes" prints all notations
- New command "About name" for light printing of type, implicit arguments, etc.
- New command "Admitted" to declare incompletely proven statement as axioms
- New keyword "Conjecture" to declare an axiom intended to be provable
- SearchAbout can now search for lemmas referring to more than one constant and on substrings of the name of the lemma
- "Print Implicit" displays the implicit arguments of a constant
- Locate now searches for all names having a given suffix
- New command "Functional Scheme" for building an induction principle from a function defined by case analysis and fix.

Commands

- new coqtop/coqc option -dont-load-proofs not to load opaque proofs in memory

Implicit arguments

- Inductive in sections declared with implicits now ”discharged” with implicits (like constants and variables)
- Implicit Arguments flags are now synchronous with reset
- New switch ”Unset/Set Printing Implicits” (new syntax: ”Unset/Set Printing Implicit”) to globally control printing of implicits

Grammar extensions

- Many newly supported UTF-8 encoded unicode blocks - Greek letters (0380-03FF), Hebrew letters (U05D0-05EF), letter-like symbols (2100-214F, that includes double N,Z,Q,R), prime signs (from 2080-2089) and characters from many written languages are valid in identifiers - mathematical operators (2200-22FF), supplemental mathematical operators (2A00-2AFF), miscellaneous technical (2300-23FF that includes sqrt symbol), miscellaneous symbols (2600-26FF), arrows (2190-21FF and 2900-297F), invisible mathematical operators (from 2080-2089), ... are valid symbols

Library

- New file about the factorial function in Arith
- An additional elimination Acc_iter for Acc, simpler than Acc_rect. This new elimination principle is used for definition well_founded_induction.
- New library NArith on binary natural numbers
- R is now of type Set
- Restructuration in ZArith library
  - ”true_sub” used in Zplus now a definition, not a local one (source of incompatibilities in proof referring to true_sub, may need extra Unfold)
  - Some lemmas about minus moved from fast_integer to Arith/Minus.v (le_minus, lt_mult_left) (theoretical source of incompatibilities)
  - Several lemmas moved from auxiliary.v and zarith_aux.v to fast_integer.v (theoretical source of incompatibilities)
  - Variables names of iff_trans changed (source of incompatibilities)
  - ZArith lemmas named `OMEGA` something or `fast_` something, and lemma `new_var` are now out of ZArith (except `OMEGA2`)
  - Redundant ZArith lemmas have been renamed: for the following pairs, use the second name (Zle_Zmult_right2, Zle_mult_simpl), (OMEGA2, Zle_0_plus), (Zplus_assoc_l, Zplus_assoc), (Zmult_one, Zmult_1_n), (Zmult_assoc_l, Zmult_assoc), (Zmult_minus_distr, Zmult_Zminus_distr_l) (add_un_double_moins_un_xO, is_double_moins_un), (Rlt_monotony_rev,Rlt_monotony_contra) (source of incompatibilities)
- Few minor changes (no more implicit arguments in Zmult_Zminus_distr_l and Zmult_Zminus_distr_r, lemmas moved from Zcomplements to other files) (rare source of incompatibilities)
- New lemmas provided by users added

Tactic language

- Fail tactic now accepts a failure message
- Idtac tactic now accepts a message
- New primitive tactic ”FreshId” (new syntax: ”fresh”) to generate new names
- Debugger prints levels of calls

Tactics

- Replace can now replace proofs also
- Fail levels are now decremented at ”Match Context” blocks only and if the right-hand-side of ”Match term With” are tactics, these tactics are never evaluated immediately and do not induce backtracking (in contrast with ”Match Context”)
- Quantified names now avoid global names of the current module (like Intro names did) [source of rare incompatibilities: 2 changes in the set of user contribs]
- NewDestruct/NewInduction accepts intro patterns as introduction names
- NewDestruct/NewInduction now work for non-inductive type using option ”using”
- A NewInduction naming bug for inductive types with functional arguments (e.g. the accessibility predicate) has been fixed (source of incompatibilities)
- Symmetry now applies to hypotheses too
- Inversion now accept option ”as [...]” to name the hypotheses
- Contradiction now looks also for contradictory hypotheses stating ~A and A (source of incompatibility)
- ”Contradiction c” try to find an hypothesis in context which contradicts the type of c
- Ring applies to new library NArith (require file NArithRing)
- Field now works on types in Set
- Auto with reals now try to replace le by ge (Rge_le is no longer an immediate hint), resulting in shorter proofs
- Instantiate now works in hyps (syntax : Instantiate in ...)
- Some new tactics : EConstructor, ELeft, Eright, ESplit, EExists
- New tactic ”functional induction” to perform case analysis and induction following the definition of a function.
- Clear now fails when trying to remove a local definition used by a constant appearing in the current goal

Extraction (See details in plugins/extraction/CHANGES)

- The old commands: (Recursive) Extraction Module M. are now: (Recursive) Extraction Library M. To use these commands, M should come from a library M.v
- The other syntax Extraction & Recursive Extraction now accept module names as arguments.

Bugs

- see coq-bugs server for the complete list of fixed bugs

Miscellaneous

- Implicit parameters of inductive types definition now taken into account for inferring other implicit arguments

Incompatibilities

- Persistence of true_sub (4 incompatibilities in Coq user contributions)
- Variable names of some constants changed for a better uniformity (2 changes in Coq user contributions)
- Naming of quantified names in goal now avoid global names (2 occurrences)
- NewInduction naming for inductive types with functional arguments (no incompatibility in Coq user contributions)
- Contradiction now solve more goals (source of 2 incompatibilities)
- Merge of eq and eqT may exceptionally result in subgoals now solved automatically

- Redundant pairs of ZArith lemmas may have different names: it may cause ”Apply/Rewrite with” to fail if using the first name of a pair of redundant lemmas (this is solved by renaming the variables bound by ”with”; 3 incompatibilities in Coq user contribs)
- ML programs referring to constants from fast_integer.v must use ”Coqlib.gen_constant_modules Coqlib.zarith_base_modules” instead

Details of changes in 8.0beta new syntax

New concrete syntax

- A completely new syntax for terms
- A more uniform syntax for tactics and the tactic language
- A few syntactic changes for commands
- A smart automatic translator translating V8.0 files in old syntax to files valid for V8.0

Syntax extensions

- ”Grammar” for terms disappears
- ”Grammar” for tactics becomes ”Tactic Notation”
- ”Syntax” disappears
- Introduction of a notion of notation scope allowing to use the same notations in various contexts without using specific delimiters (e.g the same expression ”4<=3+x” is interpreted either in ”nat”, ”positive”, ”N” (previously ”entier”), ”Z”, ”R”, depending on which Notation scope is currently open) [see documentation for details]
- Notation now requires a precedence and associativity (default was to set precedence to 1 and associativity to none)

Revision of the standard library

- Many lemmas and definitions names have been made more uniform mostly in Arith, NArith, ZArith and Reals (e.g : ”times” -> ”Pmult”, ”times_sym” -> ”Pmult_comm”, ”Zle_Zmult_pos_right” -> ”Zmult_le_compat_r”, ”SUPERIEUR” -> ”Gt”, ”ZERO” -> ”Z0”)
- Order and names of arguments of basic lemmas on nat, Z, positive and R have been made uniform.
- Notions of Coq initial state are declared with (strict) implicit arguments
- eq merged with eqT: old eq disappear, new eq (written =) is old eqT and new eqT is syntactic sugar for new eq (notation == is an alias for = and is written as it, exceptional source of incompatibilities)
- Similarly, ex, ex2, all, identity are merged with exT, exT2, allT, identityT
- Arithmetical notations for nat, positive, N, Z, R, without needing any backquote or double-backquotes delimiters.
- In Lists: new concrete notations; argument of nil is now implicit
- All changes in the library are taken in charge by the translator

Semantical changes during translation

- Recursive keyword set by default (and no longer needed) in Tactic Definition
- Set Implicit Arguments is strict by default in new syntax
- reductions in hypotheses of the form ”... in H” now apply to the type also if H is a local definition
- etc

Gallina

- New syntax of the form ”Inductive bool : Set := true, false : bool.” for enumerated types
- Experimental syntax of the form p.(fst) for record projections (activable with option ”Set Printing Projections” which is recognized by the translator)

Known problems of the automatic translation

- iso-latin-1 characters are no longer supported: move your files to 7-bits ASCII or unicode before translation (switch to unicode is automatically done if a file is loaded and saved again by coqide)
- Renaming in ZArith: incompatibilities in Coq user contribs due to merging names INZ, from Reals, and inject_nat.
- Renaming and new lemmas in ZArith: may clash with names used by users
- Restructuration of ZArith: replace requirement of specific modules in ZArith by ”Require Import ZArith_base” or ”Require Import ZArith”
- Some implicit arguments must be made explicit before translation: typically for ”length nil”, the implicit argument of length must be made explicit
- Grammar rules, Infix notations and V7.4 Notations must be updated wrt the new scheme for syntactic extensions (see translator documentation)
- Unsafe for annotation Cases when constructors coercions are used or when annotations are eta-reduced predicates

Details of changes in 8.0

Commands

- New option ”Set Printing All” to deactivate all high-level forms of printing (implicit arguments, coercions, destructing let, if-then-else, notations, projections)
- ”Functional Scheme” and ”Functional Induction” extended to polymorphic types and dependent types
- Notation now allows recursive patterns, hence recovering parts of the functionalities of pre-V8 Grammar/Syntax commands
- Command ”Print.” discontinued.
- Redundant syntax ”Implicit Arguments On/Off” discontinued

New syntax

- Semantics change of the if-then-else construction in new syntax: ”if c then t1 else t2” now stands for ”match c with c1 _ ... _ => t1 | c2 _ ... _ => t2 end” with no dependency of t1 and t2 in the arguments of the constructors; this may cause incompatibilities for files translated using coq 8.0beta

Notation scopes

- Delimiting key %bool for bool_scope added
- Import no more needed to activate argument scopes from a module

Tactics and the tactic Language

- Semantics of ”assert” is now consistent with the reference manual
- New tactics stepl and stepr for chaining transitivity steps
- Tactic ”replace ... with ... in” added
- Intro patterns now supported in Ltac (parsed with prefix ”ipattern:”)

Executables and tools

- Added option -top to change the name of the toplevel module ”Top”

- Coqdoc updated to new syntax and now part of Coq sources
- XML exportation tool now exports the structure of vernacular files (cf chapter 13 in the reference manual)

User contributions

- User contributions have been updated to the new syntax

Bug fixes

- Many bugs have been fixed (cf coq-bugs web page)

BIBLIOGRAPHY

[AC19] Andreas Abel and Thierry Coquand. Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. Technical Report, Chalmers and Gothenburg University, 2019.

[Bar81] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

[BDenesGregoire11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of Lecture Notes in Computer Science, 362–377. Springer, 2011. URL: http://dx.doi.org/10.1007/978-3-642-25379-9_26, doi:10.1007/978-3-642-25379-9_26[967].

[Bou97] S. Boutin. Using reflection to build efficient and certified decision procedure s. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281 of Lecture Notes in Computer Science. Springer-Verlag, 1997.

[CF07] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *ACM SIGPLAN Workshop on ML*, 37–45. Freiburg, Germany, October 2007. ACM Press. URL: https://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf.

[Coq89] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989. URL: https://hal.inria.fr/inria-00075471.

[CH86a] T. Coquand and Gérard Huet. Concepts mathematiques et informatiques formalises dans le calcul des constructions. Technical Report RR-0515, INRIA, April 1986. URL: https://hal.inria.fr/inria-00076039.

[CH86b] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986. URL: https://hal.inria.fr/inria-00076024.

[Coq85] Th. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[Coq86] Th. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*. Cambridge, MA, 1986. IEEE Computer Society Press.

[Coq92] Th. Coquand. Pattern Matching with Dependent Types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. 1992.

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra*, 151–184. Springer Berlin Heidelberg, 1985. URL: http://dx.doi.org/10.1007/3-540-15983-5_13, doi:10.1007/3-540-15983-5_13[968].

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, 50–66. Springer Berlin Heidelberg, 1990. URL: http://dx.doi.org/10.1007/3-540-52335-9_47, doi:10.1007/3-540-52335-9_47[969].

[967] https://doi.org/10.1007/978-3-642-25379-9_26
[968] https://doi.org/10.1007/3-540-15983-5_13
[969] https://doi.org/10.1007/3-540-52335-9_47

[CT95] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In *TYPES*, 85–104. 1995.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*. Volume 1. North-Holland, 1958. §9E.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, 207–212. New York, NY, USA, 1982. ACM. URL: http://doi.acm.org/10.1145/582153.582176, doi:10.1145/582153.582176[970].

[dB72] N.J. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Math.*, 1972.

[Del00] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of Lecture Notes in Computer Science, 85–95. Springer-Verlag, November 2000. URL: http://www.lirmm.fr/%7Edelahaye/papers/ltac%20(LPAR%2700).pdf.

[dC95] R. di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, September 1992.

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof Irrelevance Without K. *Proc. ACM Program. Lang.*, 3(POPL):3:1–3:28, 2019. URL: http://doi.acm.org/10.1145/3290316.

[Gimenez94] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types'94 : Types for Proofs and Programs*, volume 996 of Lecture Notes in Computer Science. Springer-Verlag, 1994. Extended version in LIP research report 95-07, ENS Lyon.

[Gimenez95] E. Giménez. An application of co-inductive types in coq: verification of the alternating bit protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, 135–152. Springer-Verlag, 1995.

[Gimenez98] E. Giménez. A tutorial on recursive types in coq. Technical Report, INRIA, March 1998.

[GimenezCasteran05] E. Giménez and P. Castéran. A tutorial on [co-]inductive types in coq. available at http://coq.inria.fr/doc, January 2005.

[GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "one sugar cube, please" or selection strategies in the buchberger algorithm. In *Proceedings of the ISSAC'91, ACM Press*, 5–4. 1991.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[GZND11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *SIGPLAN Not.*, 46(9):163–175, September 2011. URL: http://doi.acm.org/10.1145/2034574.2034798, doi:10.1145/2034574.2034798[971].

[GregoireL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, 235–246. ACM, 2002. URL: http://doi.acm.org/10.1145/581478.581501, doi:10.1145/581478.581501[972].

[How80] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[970] https://doi.org/10.1145/582153.582176
[971] https://doi.org/10.1145/2034574.2034798
[972] https://doi.org/10.1145/581478.581501

[Hue89] G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney*. World Scientific Publishing, 1989.

[Hue88] Gérard Huet. Induction principles formalized in the calculus of constructions. In *Programming of Future Generation Computers. Elsevier Science*. Springer Berlin Heidelberg, 1988. URL: http://dx.doi.org/10.1007/3-540-17660-8_62, doi:10.1007/3-540-17660-8_62[973].

[LW11] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 2011.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[Let02] P. Letouzey. A new extraction for coq. In *TYPES*. 2002. URL: http://www.irif.fr/~letouzey/download/extraction2002.pdf.

[LV97] Sebastiaan P. Luttik and Eelco Visser. Specification of rewriting strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing*. Springer-Verlag, 1997.

[MT13] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of LNCS, 19–34. Rennes, France, 2013. Springer. URL: http://hal.inria.fr/hal-00816703, doi:10.1007/978-3-642-39634-2_5[974].

[McB00] Conor McBride. Elimination with a motive. In *TYPES*, 197–216. 2000.

[Moh86] Christine Mohring. Algorithm development in the calculus of constructions. In *LICS*, 84–91. 1986.

[Mun94] C. Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Master's thesis, DEA d'Informatique Fondamentale, Université Paris 7, September 1994.

[Mye86] Eugene Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1986. URL: http://www.xmailserver.org/diff2.pdf.

[Par95] C. Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics of Program Construction'95*, volume 947 of LNCS. Springer-Verlag, 1995.

[PM93a] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS. Springer-Verlag, 1993. Also LIP research report 92-49, ENS Lyon.

[PM89] Christine Paulin-Mohring. Extracting ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 89–104. ACM Press, 1989. URL: http://dx.doi.org/10.1145/75277.75285, doi:10.1145/75277.75285[975].

[PM93b] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, 328–345. Springer-Verlag, 1993. URL: http://dx.doi.org/10.1007/bfb0037116, doi:10.1007/bfb0037116[976].

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, 209–228. Springer-Verlag, 1989. URL: http://dx.doi.org/10.1007/bfb0040259, doi:10.1007/bfb0040259[977].

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[Soz07] Matthieu Sozeau. Subset coercions in Coq. In *TYPES'06*, volume 4502 of LNCS, 237–252. Springer, 2007.

[973] https://doi.org/10.1007/3-540-17660-8_62
[974] https://doi.org/10.1007/978-3-642-39634-2_5
[975] https://doi.org/10.1145/75277.75285
[976] https://doi.org/10.1007/bfb0037116
[977] https://doi.org/10.1007/bfb0040259

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs'08*. 2008.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA*, volume 2051 of LNCS, 357–362. 2001.

[VBT98] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, 13–26. 1998.

[Wer94] B. Werner. *Une théorie des constructions inductives*. Thèse de Doctorat, Université Paris 7, 1994.

GLOSSARY

a

algebraic universe, 11
alpha-convertible, 17
attribute, 8

b

backtracking, 463
backtracking point, 463
backward reasoning, 240
beta-redex, 17
beta-reduction, 17
body, 14
boolean attribute, 8
branching, 463

c

Calculus of Inductive Constructions, 22
command, 7
conclusion, 219
constant, 14
contraction, 17
convertible, 21

d

de Bruijn criterion, 3
definitional equality, 268
delta-reduction, 18

e

equality, 268
eta-expansion, 18
existential variable, 99
expansion, 17

f

first success, 463
flag, 9
forward reasoning, 240

g

global environment, 219
goal, 219

h

head, 277
head constant, 277
hypothesis, 220

i

inhabited, 7
iota-reduction, 18

l

left associative, 456
Leibniz equality, 268
local context, 219

o

occurrence, 248
opaque, 18
option, 9

p

prelude, 522
proof mode, 219
proof state, 219
proof term, 220
proposition, 10

q

quotient, 268
quotient set, 268

r

reduction, 16
right associative, 456

s

sentence, 7
setoid, 268
setoid equality, 268
shelved, 235
sort, 10
standard library, 522
strict proposition, 11

subgoal, 220

t

table, 9
tactic, 7
term, 6
transparent, 18
type, 6

u

unfold, 18
unshelved, 235

w

weak-head normal form, 276
well-typed, 6
witness, 7

z

zeta-reduction, 18

COMMAND INDEX

a

Abort, 222
About, 202
Add, 9
Add Field, 429
Add LoadPath, 212
Add ML Path, 212
Add Morphism, 450
Add Parametric Morphism, 445
Add Parametric Relation, 444
Add Rec LoadPath, 212
Add Relation, 444
Add Ring, 424
Add Setoid, 450
Add Zify, 419
Admit Obligations, 200
Admitted, 222
Arguments, 160
Axiom, 13
Axioms, 13

b

Back, 213
BackTo, 213
Bind Scope, 146

c

Canonical Structure, 184
Cd, 212
Check, 202
Class, 180
Close Scope, 145
Coercion, 169
CoFixpoint, 63
CoInductive, 61
Collection, 224
Combined Scheme, 307
Comments, 7
Compute, 282
Conjecture, 13
Conjectures, 13
Constraint, 89
Context, 65
Corollary, 15
Create HintDb, 436

d

Declare Custom Entry, 140
Declare Instance, 181
Declare Left Step, 272
Declare ML Module, 212
Declare Module, 68
Declare Morphism, 450
Declare Reduction, 282
Declare Right Step, 273
Declare Scope, 144
Defined, 222
Definition, 14
Delimit Scope, 145
Derive, 548
Derive Dependent Inversion, 308
Derive Dependent Inversion_clear, 308
Derive Inversion, 308
Derive Inversion_clear, 308
Drop, 213

e

End, 64
Eval, 282
Example, 14
Existential, 224
Existing Class, 180
Existing Instance, 181
Existing Instances, 181
Export, 70
Extract Constant, 542
Extract Inductive, 543
Extract Inlined Constant, 542
Extraction, 539
Extraction Blacklist, 544
Extraction Implicit, 541
Extraction Inline, 541
Extraction Language, 540
Extraction Library, 539

Extraction NoInline, 541
Extraction TestCompile, 539

f

Fact, 15
Fail, 214
Fixpoint, 42
Focus, 225
From … Require, 211
Function, 550
Functional Scheme, 554

g

Generalizable, 112
Goal, 221
Grab Existential Variables, 224
Guarded, 237

h

Hint Constants, 438
Hint Constructors, 437
Hint Cut, 439
Hint Extern, 438
Hint Immediate, 437
Hint Mode, 439
Hint Opaque, 438
Hint Resolve, 437
Hint Rewrite, 440
Hint Transparent, 438
Hint Unfold, 438
Hint Variables, 438
Hint View for, 410
Hint View for apply, 403
Hint View for move, 403
Hypotheses, 13
Hypothesis, 13

i

Identity Coercion, 170
Implicit Type, 110
Implicit Types, 110
Import, 69
Include, 68
Include Type, 68
Inductive, 36
Infix, 131
Info, 489
Inspect, 202
Instance, 180

l

Lemma, 15
Let, 65
Let CoFixpoint, 65
Let Fixpoint, 65
Load, 210
Locate, 209
Locate File, 209
Locate Library, 209
Locate Ltac, 209
Locate Module, 209
Locate Term, 209
Ltac, 482
Ltac2, 498
Ltac2 Eval, 518
Ltac2 external, 496
Ltac2 Notation, 511
Ltac2 Notation (abbreviation), 513
Ltac2 Set, 499
Ltac2 Type, 495

m

Module, 67
Module Type, 67

n

Next Obligation, 200
Notation, 127
Notation (abbreviation), 148
Number Notation, 150

o

Obligation, 200
Obligation Tactic, 200
Obligations, 200
Opaque, 283
Open Scope, 145
Optimize Heap, 240
Optimize Proof, 240

p

Parameter, 13
Parameters, 13
Prenex Implicits, 320
Preterm, 201
Primitive, 218
Print, 201
Print All, 202
Print All Dependencies, 209
Print Assumptions, 209
Print Canonical Projections, 186
Print Classes, 171
Print Coercion Paths, 171
Print Coercions, 171
Print Custom Grammar, 142
Print Debug GC, 240
Print Extraction Blacklist, 544
Print Extraction Inline, 541

Print Firstorder Solver, 413
Print Grammar, 132
Print Graph, 171
Print Hint, 440
Print HintDb, 440
Print Implicit, 108
Print Instances, 181
Print Libraries, 212
Print LoadPath, 212
Print Ltac, 482
Print Ltac Signatures, 482
Print ML Modules, 212
Print ML Path, 213
Print Module, 70
Print Module Type, 70
Print Opaque Dependencies, 209
Print Options, 10
Print Rewrite HintDb, 440
Print Rings, 422
Print Scope, 148
Print Scopes, 148
Print Section, 202
Print Strategies, 284
Print Strategy, 284
Print Table, 10
Print Tables, 10
Print Transparent Dependencies, 209
Print Typing Flags, 216
Print Universes, 90
Print Visibility, 148
Proof, 222
Proof `term`, 222
Proof using, 222
Proof with, 442
Property, 15
Proposition, 15
Pwd, 212

q

Qed, 221
Quit, 213

r

Record, 31
Recursive Extraction, 539
Recursive Extraction Library, 539
Redirect, 214
Register, 217
Register Inline, 218
Remark, 15
Remove, 9
Remove Hints, 440
Remove LoadPath, 212
Require, 211
Require Export, 211
Require Import, 211
Reserved Infix, 132
Reserved Notation, 131
Reset, 213
Reset Extraction Blacklist, 544
Reset Extraction Inline, 541
Reset Initial, 213
Reset Ltac Profile, 491
Restart, 225

s

Save, 221
Scheme, 305
Search, 202
Search (ssreflect), 405
SearchPattern, 207
SearchRewrite, 208
Section, 64
Separate Extraction, 539
Set, 9
Show, 236
Show Conjectures, 236
Show Existentials, 237
Show Goal, 237
Show Intro, 236
Show Intros, 237
Show Lia Profile, 416
Show Ltac Profile, 491
Show Match, 237
Show Obligation Tactic, 200
Show Proof, 236
Show Universes, 237
Show Zify, 420
Solve All Obligations, 200
Solve Obligations, 200
Strategy, 283
String Notation, 153
Structure, 31
SubClass, 170

t

Tactic Notation, 158
Test, 9
Theorem, 15
Time, 214
Timeout, 214
Transparent, 283
Typeclasses eauto, 184
Typeclasses Opaque, 182
Typeclasses Transparent, 182

u

Undelimit Scope, 145

Undo, 225
Unfocus, 225
Unfocused, 225
Universe, 89
Unset, 9
Unshelve, 235

V

Variable, 13
Variables, 13
Variant, 28

TACTIC INDEX

+

+ (backtracking branching), 463

=

=>, 338

[

[… | … | …] (dispatch), 459
[> … | … | …] (dispatch), 459

a

abstract, 481
abstract (ssreflect), 337
absurd, 265
admit, 265
apply, 251
apply (ssreflect), 334
apply … in, 254
apply … in … as, 255
assert, 262
assert_fails, 466
assert_succeeds, 467
assumption, 249
auto, 432
autoapply, 182
autorewrite, 434
autounfold, 434

b

bfs eauto, 434
btauto, 415
by, 347

c

case, 289
case (ssreflect), 333
cbn, 277
cbv, 275
change, 273
change_no_check, 273
classical_left, 266
classical_right, 266
clear, 257
clearbody, 257
cofix, 303
compare, 304
compute, 276
congr, 382
congruence, 413
constr_eq, 265
constr_eq_strict, 265
constructor, 286
context, 478
contradict, 265
contradiction, 265
cut, 263
cutrewrite, 271
cycle, 232

d

debug auto, 433
debug eauto, 434
debug trivial, 433
decide equality, 304
decompose, 261
dependent destruction, 294
dependent induction, 292
dependent inversion, 298
dependent inversion … with …, 298
dependent rewrite ->, 304
dependent rewrite <-, 304
destruct, 288
destruct … eqn:, 288
dintuition, 412
discriminate, 294
discrR, 534
do, 462
do (ssreflect), 350
done, 347
dtauto, 412

e

eapply, 251
eassert, 262

eassumption, 249
easy, 435
eauto, 433
ecase, 289
econstructor, 287
edestruct, 288
ediscriminate, 294
eelim, 292
eenough, 263
eexact, 249
eexists, 287
einduction, 291
einjection, 296
eintros, 256
eleft, 287
elim, 292
elim (ssreflect), 333
elim … with, 292
elimtype, 292
enough, 262
epose, 260
epose proof, 262
eremember, 260
erewrite, 271
eright, 287
eset, 260
esimplify_eq, 304
esplit, 287
etransitivity, 269
eval, 282
evar, 264
exact, 249
exact (ssreflect), 408
exact_no_check, 267
exactly_once, 466
exfalso, 265
exists, 287

f

f_equal, 269
fail, 467
field, 427
field_simplify, 428
field_simplify_eq, 428
finish_timing, 470
first, 464
first (ssreflect), 349
first last, 349
firstorder, 413
fix, 303
fold, 279
fresh, 479
fun, 471
functional induction, 552
functional inversion, 553

g

generalize, 263
generally have, 409
gfail, 467
give_up, 235
guard, 480

h

has_evar, 266
have, 352
hnf, 277

i

idtac, 467
if-then-else (Ltac2), 511
in, 351
induction, 289
induction … using …, 291
info_auto, 433
info_eauto, 433
info_trivial, 433
infoH, 493
injection, 294
instantiate, 264
intro, 255
intros, 256
intros …, 256
intuition, 412
inversion, 296
inversion ... using ..., 299
inversion_clear, 297
inversion_sigma, 299
is_evar, 266
is_var, 266

l

lapply, 252
last, 349
last first, 349
lazy, 275
lazy_match!, 505
lazy_match! goal, 508
lazymatch, 472
lazymatch goal, 475
left, 287
let, 471
lia, 417
lra, 417
ltac-seq, 459

m

match, 472

match (Ltac2), 511
match goal, 475
match!, 505
match! goal, 508
move, 257
move (ssreflect), 332
move … at bottom, 258
move … at top, 258
move … before …, 257
multi_match!, 505
multi_match! goal, 508
multimatch, 472
multimatch goal, 475

n

native_cast_no_check, 268
native_compute, 281
nia, 418
notypeclasses refine, 250
now, 436
now_show, 273
nra, 418
nsatz, 431
numgoals, 480

o

once, 466
only, 460
optimize_heap, 493
over, 378

p

pattern, 280
pose, 260
pose (ssreflect), 321
pose proof, 262
progress, 462
psatz, 419

r

rapply, 251
red, 277
refine, 249
reflexivity, 269
remember, 260
rename, 259
repeat, 462
replace, 271
reset ltac profile, 492
restart_timer, 470
revert, 257
revert dependent, 257
revgoals, 234
rewrite, 269
rewrite (ssreflect), 363
rewrite *, 271
rewrite_db, 271
rewrite_strat, 452
right, 287
ring, 421
ring_simplify, 421
rtauto, 412

s

set, 259
set (ssreflect), 323
setoid_reflexivity, 449
setoid_replace, 449
setoid_rewrite, 449
setoid_symmetry, 449
setoid_transitivity, 449
shelve, 235
shelve_unifiable, 235
show ltac profile, 492
simpl, 277
simple apply, 252
simple destruct, 289
simple eapply, 252
simple induction, 292
simple inversion, 299
simple notypeclasses refine, 251
simple refine, 250
simple subst, 272
simplify_eq, 304
solve, 465
solve_constraints, 266
specialize, 263
split, 287
split_Rabs, 534
split_Rmult, 534
start ltac profiling, 492
stepl, 272
stepr, 273
stop ltac profiling, 492
subst, 272
substitute, 272
suff, 409
suffices, 409
swap, 233
symmetry, 269

t

tauto, 411
time, 470
time_constr, 470
timeout, 470
transitivity, 269
transparent_abstract, 481

trivial, 433
try, 462
tryif, 465
type of, 480
type_term, 480
typeclasses eauto, 181

U

under, 377
unfold, 278
unify, 266
unlock, 381
unshelve, 235

V

vm_cast_no_check, 267
vm_compute, 281

W

with_strategy, 284
without loss, 360
wlog, 360

Z

zify, 419

{

{, 225

|

|| (first tactic making progress), 465

}

}, 225

…

… : … (goal selector), 460
… : … (ssreflect), 335

FLAGS, OPTIONS AND TABLES INDEX

a

Allow StrictProp, 94
Asymmetric Patterns, 121
Auto Template Polymorphism, 52

b

Boolean Equality Schemes, 307
Bullet Behavior, 232

c

Case Analysis Schemes, 307
Contextual Implicit, 106
Coqtop Exit On Error, 560
Cumulative StrictProp, 99
Cumulativity Weak Constraints, 88

d

Debug, 214
Debug Auto, 433
Debug Eauto, 434
Debug Ssreflect, 411
Debug SsrMatching, 411
Debug Trivial, 433
Decidable Equality Schemes, 307
Default Goal Selector, 241
Default Proof Mode, 225
Default Proof Using, 224
Default Timeout, 214
Definitional UIP, 97
Diffs, 238
Dump Arith, 416

e

Elaboration StrictProp Cumulativity, 98
Elimination Schemes, 307
Extraction AutoInline, 540
Extraction Conservative Types, 540
Extraction File Comment, 544
Extraction Flag, 544
Extraction KeepSingleton, 540
Extraction Optimize, 540
Extraction SafeImplicits, 541
Extraction TypeExpand, 544

f

Fast Name Printing, 210
Firstorder Depth, 413
Firstorder Solver, 413

g

Guard Checking, 215

h

Hyps Limit, 239

i

Implicit Arguments, 106
Info Auto, 433
Info Eauto, 434
Info Level, 490
Info Trivial, 433
Instance Generalized Output, 181
Intuition Negation Unfolding, 412

k

Keep Proof Equalities, 296
Keyed Unification, 271

l

Lia Cache, 416
Loose Hint Behavior, 441
Ltac Backtrace, 489
Ltac Batch Debug, 490
Ltac Debug, 490
Ltac Profiling, 490
Ltac2 Backtrace, 518

m

Mangle Names, 267
Mangle Names Prefix, 267
Maximal Implicit Insertion, 106

n

NativeCompute Profile Filename, 281
NativeCompute Profiling, 281
NativeCompute Timing, 281
Nested Proofs Allowed, 239
Nia Cache, 416
Nonrecursive Elimination Schemes, 307
Nra Cache, 416

p

Parsing Explicit, 109
Polymorphic Inductive Cumulativity, 86
Positivity Checking, 215
Primitive Projections, 35
Printing All, 215
Printing Allow Match Default Clause, 116
Printing Coercion, 171
Printing Coercions, 171
Printing Compact Contexts, 215
Printing Constructor, 33
Printing Dependent Evars Line, 215
Printing Depth, 215
Printing Existential Instances, 100
Printing Factorizable Match Patterns, 115
Printing Float, 537
Printing Goal Names, 239
Printing If, 116
Printing Implicit, 108
Printing Implicit Defensive, 108
Printing Let, 116
Printing Matching, 115
Printing Notations, 132
Printing Parentheses, 132
Printing Primitive Projection Parameters, 35
Printing Projections, 33
Printing Raw Literals, 132
Printing Record, 33
Printing Records, 33
Printing Synth, 116
Printing Unfocused, 215
Printing Universes, 90
Printing Use Implicit Types, 110
Printing Width, 215
Printing Wildcard, 116
Private Polymorphic Universes, 91
Program Cases, 198
Program Generalized Coercion, 198
Program Mode, 198

r

Regular Subst Tactic, 272
Reversible Pattern Implicit, 106
Rewriting Schemes, 307

s

Search Blacklist, 208
Search Output Name Only, 208
Short Module Printing, 70
Silent, 214
Simplex, 415
Solve Unification Constraints, 266
SsrHave NoTCResolution, 359
SsrIdents, 317
SsrOldRewriteGoalsOrder, 365
SsrRewrite, 317
Strict Implicit, 106
Strict Universe Declaration, 91
Strongly Strict Implicit, 106
Structural Injection, 296
Suggest Proof Using, 224

t

Transparent Obligations, 201
Typeclass Resolution For Conversion, 183
Typeclasses Debug, 183
Typeclasses Debug Verbosity, 183
Typeclasses Dependency Order, 183
Typeclasses Depth, 183
Typeclasses Filtered Unification, 183
Typeclasses Iterative Deepening, 183
Typeclasses Limit Intros, 183
Typeclasses Strict Resolution, 183
Typeclasses Unique Instances, 183
Typeclasses Unique Solutions, 183

u

Uniform Inductive Parameters, 40
Universal Lemma Under Conjunction, 254
Universe Checking, 216
Universe Minimization ToSet, 89
Universe Polymorphism, 85

w

Warnings, 214

GALLINA INDEX

b

Bound on the ceiling function, 418

ERRORS AND WARNINGS INDEX

,

'via' and 'abstract' cannot be used together, 152

.

... is not definitionally an identity function, 170

a

Argument of match does not evaluate to a term, 473
Argument ‘name’ is a trailing implicit, so it can't be declared non maximal. Please use { } instead of [ ], 103
Arguments of ring_simplify do not have all the same type, 422
Attempt to save an incomplete proof, 221
Automatically declaring ‘ident’ as template polymorphic, 52

b

Bad lemma for decidability of equality, 425
Bad magic number, 211
Bad occurrence number of ‘qualid’, 279
Bad relevance, 99
Bad ring structure, 425
Brackets do not support multi-goal selectors, 227

c

Cannot build functional inversion principle, 552
Cannot coerce ‘qualid’ to an evaluable reference, 438
Cannot define graph for ‘ident’, 552
Cannot define principle(s) for ‘ident’, 552
Cannot find a declared ring structure for equality ‘term’, 423
Cannot find a declared ring structure over ‘term’, 422
Cannot find a relation to rewrite, 271
Cannot find any non-recursive equality over ‘ident’, 272
Cannot find induction information on ‘qualid’, 553
Cannot find inversion information for hypothesis ‘ident’, 553
Cannot find library foo in loadpath, 211
Cannot find the source class of ‘qualid’, 170
Cannot handle mutually (co)inductive records, 34
Cannot infer a term for this placeholder. (Casual use of implicit arguments), 103
Cannot infer a term for this placeholder. (refine), 250
Cannot interpret in ‘scope_name’ because ‘qualid’ could not be found in the current environment, 154
Cannot interpret this number as a value of type ‘type’, 152
Cannot interpret this string as a value of type ‘type’, 153
Cannot load ‘qualid’: no physical path bound to ‘dirpath’, 211
Cannot move ‘ident’ after ‘ident’: it depends on ‘ident’, 258
Cannot move ‘ident’ after ‘ident’: it occurs in the type of ‘ident’, 258
Cannot recognize a boolean equality, 415
Cannot recognize ‘class’ as a source class of ‘qualid’, 170
Cannot turn [inductive|constructor] into an evaluable reference, 278
Cannot use mutual definition with well-founded recursion or measure, 552

Can't find file 'ident' on loadpath, 210
Casts are not supported in this pattern, 29
Compiled library 'ident'.vo makes inconsistent assumptions over library 'qualid', 211
Condition not satisfied, 481

d

Debug mode not available in the IDE, 490
Declaring arbitrary terms as hints is fragile; it is recommended to declare a toplevel constant instead, 437

e

Either there is a type incompatibility or the problem involves dependencies, 126
Expression does not evaluate to a tactic, 473

f

Failed to progress, 462
File not found on loadpath: 'string', 212
Files processed by Load cannot leave open proofs, 210
Flag 'rename' expected to rename 'name' into 'name', 161
Found a constructor of inductive type term while a constructor of term is expected, 126
Found an "at" clause without "with" clause, 273
Found no subterm matching 'term' in the current goal, 271
Found no subterm matching 'term' in 'ident', 271
Found target class ... instead of ..., 170
Funclass cannot be a source class, 170

g

goal does not satisfy the expected preconditions, 296
Goal is solvable by congruence but some arguments are missing. Try congruence with 'term'…'term', replacing metavariables by arbitrary terms, 414

h

Hypothesis 'ident' must contain at least one Function, 553

i

I don't know how to handle dependent equality, 414
Ignored instance declaration for "'ident'": "'term'" is not a class, 180
Ignoring implicit binder declaration in unexpected position, 105
Ill-formed recursive definition, 201
Ill-formed template inductive declaration: not polymorphic on any universe, 52
Incorrect number of tactics (expected N tactics, was given M), 377
Invalid argument, 250
Invalid backtrack, 213

l

Last block to end has name 'ident', 65
Load is not supported inside proofs, 210
Ltac Profiler encountered an invalid stack (no self node). This can happen if you reset the profile during tactic execution, 492
Ltac2 alias 'qualid' is deprecated since 'string'. 'string', 559
Ltac2 definition 'qualid' is deprecated since 'string'. 'string', 558
Ltac2 notation 'ltac2_scope'…'ltac2_scope' is deprecated since 'string'. 'string', 559

m

Making shadowed name of implicit argument accessible by position, 105
Missing mapping for constructor 'qualid', 154
Module/section 'qualid' not found, 203
Multiple 'via' options, 152
Multiple 'warning after' or 'abstract after' options, 152

n

Nested proofs are discouraged and not allowed by default. This error probably means that you forgot to close the last "Proof." with "Qed." or "Defined.". If you really intended to use nested proofs, you can do so by turning the "Nested Proofs Allowed" flag on, 16

New coercion path ... is ambiguous with existing ..., 170
No applicable tactic, 464
No argument name 'ident', 552
No discriminable equalities, 294
No evars, 266
No focused proof, 236
No focused proof (No proof-editing in progress), 221
No focused proof to restart, 225
No head constant to reduce, 278
No matching clauses for match, 473
No matching clauses for match goal, 476
No primitive equality found, 294
No product even after head-reduction, 255
No progress made, 452
No such assumption, 249
No such binder, 241
No such goal, 236
No such goal ('ident'), 227
No such goal ('natural'), 227
No such goal. (fail), 468
No such goal. (Goal selector), 460
No such goal. Focus next goal with bullet 'bullet', 231
No such hypothesis, 257
No such hypothesis in current goal, 256
No such hypothesis: 'ident', 256
No such label 'ident', 68
Non exhaustive pattern matching, 126
Non strictly positive occurrence of 'ident' in 'type', 36
Not a context variable, 478
Not a discriminable equality, 294
Not a primitive equality, 295
Not a proposition or a type, 262
Not a valid ring equation, 422
Not a variable or hypothesis, 266
Not an evar, 266
Not an exact proof, 249
Not an inductive goal with 1 constructor, 287
Not an inductive goal with 2 constructors, 287
Not an inductive product, 286
Not convertible, 273
Not enough constructors, 287
Not equal, 265
Not equal (due to universes), 265
Not the right number of induction arguments, 553
Not the right number of missing arguments, 241
Notation levels must range between 0 and 6, 512
Notation 'string' is deprecated since 'string'. 'string', 558
Nothing to do, it is an equality between convertible terms, 295
Nothing to inject, 295
Nothing to rewrite, 452

o

overflow in int63 literal 'bigint', 152

p

Polymorphic universe constraints can only be declared inside sections, use Monomorphic Constraint instead, 89
Polymorphic universes can only be declared inside sections, use Monomorphic Universe instead, 89
Proof is not complete. (abstract), 481
Proof is not complete. (assert), 262

r

Records declared with the keyword Record or Structure cannot be recursive, 34
Refine passed ill-formed term, 250
Require inside a module is deprecated and strongly discouraged. You can Require a module at toplevel and optionally Import it inside another one, 211
Ring operation should be declared as a morphism, 425

s

Section variable 'ident' occurs implicitly in global declaration 'qualid' present in hypothesis 'ident', 272
Section variable 'ident' occurs implicitly in global declaration 'qualid' present in the conclusion, 272
Signature components for label 'ident' do not match, 68
SProp is disallowed because the "Allow StrictProp" flag is off, 94
SSReflect: cannot obtain new equations out of ..., 333
Stack overflow or segmentation fault happens when working with large numbers in 'type' (threshold

may vary depending on your system limits and on the command executed), 151
Statement without assumptions, 254
Syntax error: [prim:reference] expected after 'Notation' (in [vernac:command]), 154
Syntax error: [prim:reference] expected after [prim:reference] (in [vernac:command]), 154

t

Tactic failure, 467
Tactic failure (level 'natural'), 467
Tactic failure: <tactic closure> fails, 467
Tactic failure: <tactic closure> succeeds, 466
Tactic failure: Setoid library not loaded, 271
Tactic generated a subgoal identical to the original goal, 271
Tactic Notation 'qualid' is deprecated since 'string'. 'string', 558
Tactic 'qualid' is deprecated since 'string'. 'string', 558
template and polymorphism not compatible, 52
Terms do not have convertible types, 271
The "at" syntax isn't available yet for the autorewrite tactic, 434
The & modifier may only occur once, 160
The 'abstract after' directive has no effect when the parsing function ('qualid') targets an option type, 152
The 'clear implicits' flag must be omitted if implicit annotations are given, 161
The 'default implicits' flag is incompatible with implicit annotations, 161
The / modifier may only occur once, 160
The command has not failed!, 214
The conclusion of 'type' is not valid; it must be built from 'ident', 37
The constant 'number' is not a binary64 floating-point value. A closest value 'number' will be used and unambiguously printed 'number'. [inexact-float,parsing], 537
The constructor 'ident' expects 'natural' arguments, 126
The cumulative attribute can only be used in a polymorphic context, 86
The elimination predicate term should be of arity 'natural' (for non dependent case) or 'natural' (for dependent case), 126
The field 'ident' is missing in 'qualid', 68
The file 'ident'.vo contains library 'qualid' and not library 'qualid', 211
The recursive argument must be specified, 552
The reference is not unfoldable, 284
The reference X was not found in the current environment, 521
The reference 'qualid' was not found in the current environment, 283
The relation 'ident' is not a declared reflexive relation. Maybe you need to require the Coq.Classes.RelationClasses library, 269
The relation 'ident' is not a declared symmetric relation. Maybe you need to require the Coq.Classes.RelationClasses library, 269
The relation 'ident' is not a declared transitive relation. Maybe you need to require the Coq.Classes.RelationClasses library, 269
The term 'term' has type 'type' which should be Set, Prop or Type, 16
The term 'term' has type 'type' while it is expected to have type 'type'', 15
The type 'ident' must be registered before this construction can be typechecked, 218
The variable ident is bound several times in pattern term, 126
The variable 'ident' is already defined, 260
The 'natural' th argument of 'ident' must be 'ident' in 'type', 28
There is already an Ltac named 'qualid', 482
There is no flag or option with this name: "'setting_name'", 9
There is no flag, option or table with this name: "'setting_name'", 9
There is no Ltac named 'qualid', 482

There is no qualid-valued table with this name: "'setting_name'", 10
There is no string-valued table with this name: "'setting_name'", 10
There is nothing to end, 65
This command does not support this attribute, 8
This command is just asserting the names of arguments of 'qualid'. If this is what you want, add ': assert' to silence the warning. If you want to clear implicit arguments, add ': clear implicits'. If you want to clear notation scopes, add ': clear scopes', 161
This object does not support universe names, 202
This proof is focused, but cannot be unfocused this way, 227
This tactic has more than one success, 466
To avoid stack overflow, large numbers in 'type' are interpreted as applications of 'qualid', 151
To rename arguments the 'rename' flag must be specified, 161
Trying to mask the absolute name 'qualid'!, 70
Type of 'ident' is not an equality of recognized Σ types: expected one of sig sig2 sigT sigT2 sigT2 ex or ex2 but got 'term', 299
Type of 'qualid' seems incompatible with the type of 'qualid'. Expected type is: 'type' instead of 'type'. This might yield ill typed terms when using the notation, 154

U

Unable to apply, 254
Unable to find an instance for the variables 'ident' ... 'ident', 290
Unable to find an instance for the variables 'ident'…'ident', 251
Unable to infer a match predicate, 126
Unable to satisfy the rewriting constraints, 452
Unable to unify 'term' with 'term', 251
Unbound [value|constructor] X, 521
Unbound context identifier 'ident', 478
Undeclared universe 'ident', 89
Unexpected non-option term 'term' while parsing a number notation, 152
Unexpected non-option term 'term' while parsing a string notation, 153
Unexpected term 'term' while parsing a number notation, 152
Unexpected term 'term' while parsing a string notation, 153
Universe inconsistency, 89
Universe instance should have length 'natural', 202
Unknown inductive type, 237
Unused variable 'ident' catches more than one case, 117
Use of 'string' Notation is deprecated as it is inconsistent with pattern syntax, 144

V

Variable 'ident' is already declared, 262

W

When 'term' contains more than one non dependent product the tactic lapply only takes into account the first product, 252
Wrong bullet 'bullet': Bullet 'bullet' is mandatory here, 231
Wrong bullet 'bullet': Current bullet 'bullet' is not finished, 231

Y

You should use the "Proof using [...]." syntax instead of "Proof." to enable skipping this proof which is located inside a section. Give as argument to "Proof using" the list of section variables that are not needed to typecheck the statement but that are required by the proof, 566

'

'class' must be a transparent constant, 170
'ident' already exists. (Axiom), 13
'ident' already exists. (Definition), 15
'ident' already exists. (Theorem), 16
'ident' cannot be defined, 34

‘ident’ is already declared as a typeclass, 180
‘ident’ is already used, 255
‘ident’ is declared as a local axiom, 13
‘ident’ is not a local definition, 257
‘ident’ is not an equality of Σ types, 299
‘ident’ is opaque, 278
‘ident’ is used in conclusion, 263
‘ident’ is used in hypothesis ‘ident’, 263
‘ident’ is used in the conclusion, 257
‘ident’ is used in the hypothesis ‘ident’, 257
‘ident’: no such entry, 213
‘qualid’ cannot be used as a hint, 437
‘qualid’ does not occur, 279
‘qualid’ does not respect the uniform inheritance condition, 170
‘qualid’ is already a coercion, 170
‘qualid’ is bound to a notation that does not denote a reference, 154
‘qualid’ is not a function, 170
‘qualid’ is not a module, 70
‘qualid’ is not an inductive type, 438
‘qualid’ not a defined object, 201
‘qualid’ not declared, 170
‘qualid’ should go from Byte.byte or (list Byte.byte) to ‘type’ or (option ‘type’), 153
‘qualid’ should go from Number.int to ‘type’ or (option ‘type’). Instead of Number.int, the types Number.uint or Z or PrimInt63.pos_neg_int63 or Number.number could be used (you may need to require BinNums or Number or PrimInt63 first), 152
‘qualid’ should go from ‘type’ to Byte.byte or (option Byte.byte) or (list Byte.byte) or (option (list Byte.byte)), 153
‘qualid’ should go from ‘type’ to Number.int or (option Number.int). Instead of Number.int, the types Number.uint or Z or PrimInt63.pos_neg_int63 or Number.number could be used (you may need to require BinNums or Number or PrimInt63 first), 152
‘qualid’ was already mapped to ‘qualid’ and cannot be remapped to ‘qualid’, 154
‘type’ is not an inductive type, 154
‘type’ was already mapped to ‘type’, mapping it also to ‘type’ might yield ill typed terms when using the notation, 154

ATTRIBUTE INDEX

b

bypass_check(guard), 215
bypass_check(positivity), 216
bypass_check(universes), 216

c

canonical, 186
Cumulative, 85

d

deprecated, 558

e

export, 80

g

global, 80

l

local, 80

m

Monomorphic, 85

n

NonCumulative, 85

p

Polymorphic, 85
Private, 28
private(matching), 28
program, 198
Program, 198

r

refine, 181

u

universes(cumulative), 85
universes(monomorphic), 85
universes(noncumulative), 86
universes(notemplate), 52
universes(polymorphic), 85
universes(template), 52
using, 223

INDEX

Symbols

'via' and 'abstract' cannot be used together (*error*), 152
* (*term*), 526, 532
+ (backtracking branching) (*tactic*), 463
+ (*term*), 526, 532
... : ... (*type cast*), 14
... <: ..., 14
... <<: ..., 14
... is not definitionally an identity function (*warning*), 170
::=, 482
:> (*coercion*), 171
:> (*substructure*), 179
< (*term*), 532
<= (*term*), 532
=> (*tactic*), 338
> (*term*), 532
>= (*term*), 532
?= (*term*), 532
_, 100
– (*term*), 532
{ (*tactic*), 225
{A}+{B} (*term*), 527
{x:A & P x} (*term*), 527
{x:A | P x} (*term*), 527
} (*tactic*), 225
|| (first tactic making progress) (*tactic*), 465
`( ), 111
`(!), 111
`{ }, 111
`{! }, 111
`[ ], 111
`[!], 111
[… | … | …] (dispatch) (*tactic*), 459
[> … | … | …] (dispatch) (*tactic*), 459

A

A*B (*term*), 526
A+{B} (*term*), 527
A+B (*term*), 526
Abort (*command*), 222
About (*command*), 202
abstract (ssreflect) (*tactic*), 337
abstract (*tactic*), 481
absurd (*tactic*), 265
absurd (*term*), 525
absurd_set (*term*), 528
Acc (*term*), 529
Acc_inv (*term*), 529
Acc_rect (*term*), 529
Add (*command*), 9
Add Field (*command*), 429
Add LoadPath (*command*), 212
Add ML Path (*command*), 212
Add Morphism (*command*), 450
Add Parametric Morphism (*command*), 445
Add Parametric Relation (*command*), 444
Add Rec LoadPath (*command*), 212
Add Relation (*command*), 444
Add Ring (*command*), 424
Add Setoid (*command*), 450
Add Zify (*command*), 419
admit (*tactic*), 265
Admit Obligations (*command*), 200
Admitted (*command*), 222
all (*term*), 524
Allow StrictProp (*flag*), 94
and (*term*), 524
and_rect (*term*), 528
app (*term*), 535
apply (ssreflect) (*tactic*), 334, 408
apply (*tactic*), 251
apply … in (*tactic*), 254
apply … in … as (*tactic variant*), 255
Argument of match does not evaluate to a term (*error*), 473
Argument ‘name’ is a trailing implicit, so it can't be declared non maximal. Please use { } instead of [ ] (*error*), 103
Arguments (*command*), 160
Arguments of ring_simplify do not have

all the same type (*error*), 422
Arithmetical notations, 532
assert (*tactic*), 262
assert_fails (*tactic*), 466
assert_succeeds (*tactic*), 467
assumption (*tactic*), 249
Asymmetric Patterns (*flag*), 121
Attempt to save an incomplete proof (*error*), 221
auto (*tactic*), 432
Auto Template Polymorphism (*flag*), 52
autoapply (*tactic*), 182
Automatically declaring 'ident' as template polymorphic (*warning*), 52
autorewrite (*tactic*), 434
autounfold (*tactic*), 434
Axiom (*command*), 13
Axioms (*command*), 13

B

Back (*command*), 213
BackTo (*command*), 213
Bad lemma for decidability of equality (*error*), 425
Bad magic number (*error*), 211
Bad occurrence number of 'qualid' (*error*), 279
Bad relevance (*warning*), 99
Bad ring structure (*error*), 425
bfs eauto (*tactic*), 434
Bind Scope (*command*), 146
bool (*term*), 526
bool_choice (*term*), 527
Boolean Equality Schemes (*flag*), 307
Bound on the ceiling function (*theorem*), 418
Brackets do not support multi-goal selectors (*error*), 227
btauto (*tactic*), 415
Bullet Behavior (*option*), 232
by (*tactic*), 347
bypass_check(guard) (*attribute*), 215
bypass_check(positivity) (*attribute*), 216
bypass_check(universes) (*attribute*), 216

C

Cannot build functional inversion principle (*warning*), 552
Cannot coerce 'qualid' to an evaluable reference (*error*), 438
Cannot define graph for 'ident' (*warning*), 552
Cannot define principle(s) for 'ident' (*warning*), 552
Cannot find a declared ring structure for equality 'term' (*error*), 423
Cannot find a declared ring structure over 'term' (*error*), 422
Cannot find a relation to rewrite (*error*), 271
Cannot find any non-recursive equality over 'ident' (*error*), 272
Cannot find induction information on 'qualid' (*error*), 553
Cannot find inversion information for hypothesis 'ident' (*error*), 553
Cannot find library foo in loadpath (*error*), 211
Cannot find the source class of 'qualid' (*error*), 170
Cannot handle mutually (co)inductive records (*error*), 34
Cannot infer a term for this placeholder. (Casual use of implicit arguments) (*error*), 103
Cannot infer a term for this placeholder. (refine) (*error*), 250
Cannot interpret in 'scope_name' because 'qualid' could not be found in the current environment (*error*), 154
Cannot interpret this number as a value of type 'type' (*error*), 152
Cannot interpret this string as a value of type 'type' (*error*), 153
Cannot load 'qualid': no physical path bound to 'dirpath' (*error*), 211
Cannot move 'ident' after 'ident': it depends on 'ident' (*error*), 258
Cannot move 'ident' after 'ident': it occurs in the type of 'ident' (*error*), 258
Cannot recognize a boolean equality (*error*), 415
Cannot recognize 'class' as a source class of 'qualid' (*error*), 170
Cannot turn [inductive|constructor] into an evaluable reference (*error*), 278
Cannot use mutual definition with well-founded recursion or measure (*error*), 552
canonical (*attribute*), 186
Canonical Structure (*command*), 184
Can't find file 'ident' on loadpath (*error*), 210
case (ssreflect) (*tactic*), 333
case (*tactic*), 289

Case Analysis Schemes (*flag*), 307
Casts are not supported in this pattern (*error*), 29
cbn (*tactic*), 277
cbv (*tactic*), 275
Cd (*command*), 212
change (*tactic*), 273
change_no_check (*tactic*), 273
Check (*command*), 202
Choice (*term*), 527
Choice2 (*term*), 527
Class (*command*), 180
classical_left (*tactic*), 266
classical_right (*tactic*), 266
clear (*tactic*), 257
clearbody (*tactic variant*), 257
Close Scope (*command*), 145
Coercion (*command*), 169
cofix, 63
cofix (*tactic*), 303
CoFixpoint (*command*), 63
CoInductive (*command*), 61
Collection (*command*), 224
Combined Scheme (*command*), 307
command, **7**
Comments (*command*), 7
compare (*tactic*), 304
Compiled library 'ident'.vo makes inconsistent assumptions over library 'qualid' (*error*), 211
Compute (*command*), 282
compute (*tactic*), 276
Condition not satisfied (*error*), 481
congr (*tactic*), 382, 409
congruence (*tactic*), 413
conj (*term*), 524
Conjecture (*command*), 13
Conjectures (*command*), 13
Connectives, 524
constr_eq (*tactic*), 265
constr_eq_strict (*tactic*), 265
Constraint (*command*), 89
constructor (*tactic*), 286
Context (*command*), 65
context (*tactic*), 478
Contextual Implicit (*flag*), 106
contradict (*tactic*), 265
contradiction (*tactic*), 265
coqdoc, 579
Coqtop Exit On Error (*flag*), 560
Corollary (*command*), 15
Create HintDb (*command*), 436
Cumulative (*attribute*), 85
Cumulative StrictProp (*flag*), 99
Cumulativity Weak Constraints (*flag*), 88
cut (*tactic variant*), 263
cutrewrite (*tactic*), 271
cycle (*tactic*), 232

D

Datatypes, 526
Debug (*option*), 214
Debug Auto (*flag*), 433
debug auto (*tactic*), 433
Debug Eauto (*flag*), 434
debug eauto (*tactic*), 434
Debug mode not available in the IDE (*error*), 490
Debug Ssreflect (*flag*), 411
Debug SsrMatching (*flag*), 411
Debug Trivial (*flag*), 433
debug trivial (*tactic*), 433
Decidable Equality Schemes (*flag*), 307
decide equality (*tactic*), 304
Declare Custom Entry (*command*), 140
Declare Instance (*command*), 181
Declare Left Step (*command*), 272
Declare ML Module (*command*), 212
Declare Module (*command*), 68
Declare Morphism (*command*), 450
Declare Reduction (*command*), 282
Declare Right Step (*command*), 273
Declare Scope (*command*), 144
Declaring arbitrary terms as hints is fragile
it is recommended to declare a toplevel constant instead (*warning*), 437
decompose (*tactic*), 261
Default Goal Selector (*option*), 241
Default Proof Mode (*option*), 225
Default Proof Using (*option*), 224
Default Timeout (*option*), 214
Defined (*command*), 222
Definition (*command*), 14
Definitional UIP (*flag*), 97
Delimit Scope (*command*), 145
dependent destruction (*tactic variant*), 294
dependent induction (*tactic*), 292
dependent inversion (*tactic variant*), 298
dependent inversion … with … (*tactic variant*), 298
dependent rewrite <- (*tactic variant*), 304
dependent rewrite -> (*tactic*), 304
deprecated (*attribute*), 558
Derive (*command*), 548
Derive Dependent Inversion (*command*), 308

Derive Dependent Inversion_clear (*command*), 308
Derive Inversion (*command*), 308
Derive Inversion_clear (*command*), 308
destruct (*tactic*), 288
destruct … eqn: (*tactic variant*), 288
Diffs (*option*), 238
dintuition (*tactic*), 412
discriminate (*tactic*), 294
discrR (*tactic*), 534
do (ssreflect) (*tactic*), 350
do (*tactic*), 462
done (*tactic*), 347
Drop (*command*), 213
dtauto (*tactic*), 412
Dump Arith (*option*), 416

E

eapply (*tactic variant*), 251
eassert (*tactic variant*), 262
eassumption (*tactic variant*), 249
easy (*tactic*), 435
eauto (*tactic*), 433
ecase (*tactic variant*), 289
econstructor (*tactic variant*), 287
edestruct (*tactic variant*), 288
ediscriminate (*tactic variant*), 294
eelim (*tactic variant*), 292
eenough (*tactic variant*), 263
eexact (*tactic variant*), 249
eexists (*tactic variant*), 287
einduction (*tactic variant*), 291
einjection (*tactic variant*), 296
eintros (*tactic*), 256
Either there is a type incompatibility or the problem involves dependencies (*error*), 126
Elaboration StrictProp Cumulativity (*flag*), 98
eleft (*tactic variant*), 287
elim (ssreflect) (*tactic*), 333
elim (*tactic variant*), 292
elim … with (*tactic variant*), 292
Elimination Schemes (*flag*), 307
elimtype (*tactic variant*), 292
End (*command*), 64
enough (*tactic variant*), 262
epose (*tactic variant*), 260
epose proof (*tactic variant*), 262
eq (*term*), 525
eq_add_S (*term*), 528
eq_ind_r (*term*), 525
eq_rec_r (*term*), 525
eq_rect (*term*), 525, 528
eq_rect_r (*term*), 525
eq_refl (*term*), 525
eq_S (*term*), 528
eq_sym (*term*), 525
eq_trans (*term*), 525
Equality, 525
eremember (*tactic variant*), 260
erewrite (*tactic*), 271
eright (*tactic variant*), 287
error (*term*), 528
eset (*tactic variant*), 260
esimplify_eq (*tactic variant*), 304
esplit (*tactic variant*), 287
etransitivity (*tactic*), 269
Eval (*command*), 282
eval (*tactic*), 282
evar (*tactic*), 264
ex (*term*), 524
ex2 (*term*), 524
ex_intro (*term*), 524
ex_intro2 (*term*), 524
exact (ssreflect) (*tactic*), 408
exact (*tactic*), 249
exact_no_check (*tactic*), 267
exactly_once (*tactic*), 466
Example (*command*), 14
Exc (*term*), 528
exfalso (*tactic*), 265
exist (*term*), 527
exist2 (*term*), 527
Existential (*command*), 224
Existing Class (*command*), 180
Existing Instance (*command*), 181
Existing Instances (*command*), 181
exists (*tactic variant*), 287
exists (*term*), 524
exists2 (*term*), 524
existT (*term*), 527
existT2 (*term*), 527
export (*attribute*), 80
Export (*command*), 70
Expression does not evaluate to a tactic (*error*), 473
Extract Constant (*command*), 542
Extract Inductive (*command*), 543
Extract Inlined Constant (*command*), 542
Extraction (*command*), 539
Extraction AutoInline (*flag*), 540
Extraction Blacklist (*command*), 544
Extraction Conservative Types (*flag*), 540
Extraction File Comment (*option*), 544
Extraction Flag (*option*), 544
Extraction Implicit (*command*), 541
Extraction Inline (*command*), 541

Extraction KeepSingleton (*flag*), 540
Extraction Language (*command*), 540
Extraction Library (*command*), 539
Extraction NoInline (*command*), 541
Extraction Optimize (*flag*), 540
Extraction SafeImplicits (*flag*), 541
Extraction TestCompile (*command*), 539
Extraction TypeExpand (*flag*), 544

F

f_equal (*tactic*), 269
f_equal (*term*), 525
f_equal2 ... f_equal5 (*term*), 525
Fact (*command*), 15
Fail (*command*), 214
fail (*tactic*), 467
Failed to progress (*error*), 462
False (*term*), 524
false (*term*), 526
False_rec (*term*), 528
False_rect (*term*), 528
Fast Name Printing (*flag*), 210
field (*tactic*), 427
field_simplify (*tactic*), 428
field_simplify_eq (*tactic*), 428
File not found on loadpath: 'string' (*error*), 212
Files processed by Load cannot leave open proofs (*error*), 210
finish_timing (*tactic*), 470
first (ssreflect) (*tactic*), 349
first (*tactic*), 464
first last (*tactic variant*), 349
firstorder (*tactic*), 413
Firstorder Depth (*option*), 413
Firstorder Solver (*option*), 413
fix, 41
fix (*tactic*), 303
Fix_eq (*term*), 530
Fix_F (*term*), 530
Fix_F_eq (*term*), 530
Fix_F_inv (*term*), 530
Fixpoint (*command*), 42
Flag 'rename' expected to rename 'name' into 'name' (*error*), 161
flat_map (*term*), 535
Focus (*command*), 225
fold (*tactic*), 279
fold_left (*term*), 535
fold_right (*term*), 535
forall, 12
Found a constructor of inductive type term while a constructor of term is expected (*error*), 126
Found an "at" clause without "with" clause (*error*), 273
Found no subterm matching 'term' in the current goal (*error*), 271
Found no subterm matching 'term' in 'ident' (*error*), 271
Found target class ... instead of ... (*error*), 170
fresh (*tactic*), 479
From … Require (*command*), 211
fst (*term*), 526
fun, 12
fun (*tactic*), 471
Funclass cannot be a source class (*error*), 170
Function (*command*), 550
function_scope, 147
functional induction (*tactic*), 552
functional inversion (*tactic*), 553
Functional Scheme (*command*), 554

G

ge (*term*), 529
Generalizable (*command*), 112
generalize (*tactic*), 263
generally have (*tactic*), 408
gfail (*tactic*), 467
give_up (*tactic*), 235
global (*attribute*), 80
Goal (*command*), 221
goal does not satisfy the expected preconditions (*error*), 296
Goal is solvable by congruence but some arguments are missing. Try congruence with 'term'…'term', replacing metavariables by arbitrary terms (*error*), 414
Grab Existential Variables (*command*), 224
gt (*term*), 529
guard (*tactic*), 480
Guard Checking (*flag*), 215
Guarded (*command*), 237

H

has_evar (*tactic*), 266
have (*tactic*), 352, 353
head (*term*), 535
Hint Constants (*command*), 438
Hint Constructors (*command*), 437
Hint Cut (*command*), 439
Hint Extern (*command*), 438
Hint Immediate (*command*), 437
Hint Mode (*command*), 439
Hint Opaque (*command*), 438

Hint Resolve (*command*), 437
Hint Rewrite (*command*), 440
Hint Transparent (*command*), 438
Hint Unfold (*command*), 438
Hint Variables (*command*), 438
Hint View for (*command*), 410
Hint View for apply (*command*), 403, 410
Hint View for move (*command*), 403
hnf (*tactic*), 277
Hypotheses (*command*), 13
Hypothesis (*command*), 13
Hypothesis 'ident' must contain at least one Function (*error*), 553
Hyps Limit (*option*), 239

I

I (*term*), 524
I don't know how to handle dependent equality (*error*), 414
identity (*term*), 526, 530
Identity Coercion (*command*), 170
idtac (*tactic*), 467
IF_then_else (*term*), 524
if-then-else (Ltac2) (*tactic*), 511
iff (*term*), 524
Ignored instance declaration for "'ident'": "'term'" is not a class (*warning*), 180
Ignoring implicit binder declaration in unexpected position (*warning*), 105
Ill-formed recursive definition (*error*), 201
Ill-formed template inductive declaration: not polymorphic on any universe (*error*), 52
Implicit Arguments (*flag*), 106
Implicit Type (*command*), 110
Implicit Types (*command*), 110
Import (*command*), 69
in (*tactic*), 351
Include (*command*), 68
Include Type (*command*), 68
Incorrect number of tactics (expected N tactics, was given M) (*error*), 377
induction (*tactic*), 289
induction … using … (*tactic variant*), 291
Inductive (*command*), 36
Infix (*command*), 131
Info (*command*), 489
Info Auto (*flag*), 433
Info Eauto (*flag*), 434
Info Level (*option*), 490
Info Trivial (*flag*), 433
info_auto (*tactic*), 433
info_eauto (*tactic*), 433
info_trivial (*tactic*), 433
infoH (*tactic*), 493
injection (*tactic*), 294
inl (*term*), 526
inleft (*term*), 527
inr (*term*), 526
inright (*term*), 527
Inspect (*command*), 202
Instance (*command*), 180
Instance Generalized Output (*flag*), 181
instantiate (*tactic*), 264
intro (*tactic*), 255
intros (*tactic variant*), 256
intros … (*tactic*), 256
intuition (*tactic*), 412
Intuition Negation Unfolding (*flag*), 412
Invalid argument (*error*), 250
Invalid backtrack (*error*), 213
inversion (*tactic*), 296
inversion ... using ... (*tactic*), 299
inversion_clear (*tactic variant*), 297
inversion_sigma (*tactic variant*), 299
is_evar (*tactic*), 266
is_var (*tactic*), 266
IsSucc (*term*), 528

K

Keep Proof Equalities (*flag*), 296
Keyed Unification (*flag*), 271

L

lapply (*tactic variant*), 252
last (*tactic*), 349
Last block to end has name 'ident' (*error*), 65
last first (*tactic variant*), 349
lazy (*tactic*), 275
lazy_match! (*tactic*), 505
lazy_match! goal (*tactic*), 508
lazymatch (*tactic*), 472
lazymatch goal (*tactic*), 475
le (*term*), 529
le_n (*term*), 529
le_S (*term*), 529
left (*tactic variant*), 287
left (*term*), 527
Lemma (*command*), 15
length (*term*), 535
Let (*command*), 65
let (*tactic*), 471
let ... := ... (*term*), 14
Let CoFixpoint (*command*), 65

Let Fixpoint (*command*), 65
lia (*tactic*), 417
Lia Cache (*flag*), 416
Load (*command*), 210
Load is not supported inside proofs (*error*), 210
local (*attribute*), 80
Locate (*command*), 209
Locate File (*command*), 209
Locate Library (*command*), 209
Locate Ltac (*command*), 209
Locate Module (*command*), 209
Locate Term (*command*), 209
Loose Hint Behavior (*option*), 441
lra (*tactic*), 417
lt (*term*), 529
Ltac (*command*), 482
Ltac Backtrace (*flag*), 489
Ltac Batch Debug (*flag*), 490
Ltac Debug (*flag*), 490
Ltac Profiler encountered an invalid stack (no self node). This can happen if you reset the profile during tactic execution (*warning*), 492
Ltac Profiling (*flag*), 490
Ltac2 (*command*), 498
Ltac2 alias 'qualid' is deprecated since 'string'. 'string' (*warning*), 558
Ltac2 Backtrace (*flag*), 518
Ltac2 definition 'qualid' is deprecated since 'string'. 'string' (*warning*), 558
Ltac2 Eval (*command*), 518
Ltac2 external (*command*), 496
Ltac2 Notation (abbreviation) (*command*), 513
Ltac2 Notation (*command*), 511
Ltac2 notation 'ltac2_scope'…'ltac2_scope' is deprecated since 'string'. 'string' (*warning*), 558
Ltac2 Set (*command*), 499
Ltac2 Type (*command*), 495
ltac-seq (*tactic*), 459

M

Making shadowed name of implicit argument accessible by position (*warning*), 105
Mangle Names (*flag*), 267
Mangle Names Prefix (*option*), 267
map (*term*), 535
match (Ltac2) (*tactic*), 511
match (*tactic*), 472
match ... with ..., 29
match goal (*tactic*), 475
match! (*tactic*), 505
match! goal (*tactic*), 508
Maximal Implicit Insertion (*flag*), 106
Missing mapping for constructor 'qualid' (*error*), 154
mod (*term*), 532
Module (*command*), 67
Module Type (*command*), 67
Module/section 'qualid' not found (*error*), 203
Monomorphic (*attribute*), 85
move (ssreflect) (*tactic*), 332, 408
move (*tactic*), 257
move … at bottom (*tactic variant*), 258
move … at top (*tactic variant*), 258
move … before … (*tactic variant*), 257
mult (*term*), 528
mult_n_O (*term*), 528
mult_n_Sm (*term*), 528
multi_match! (*tactic*), 505
multi_match! goal (*tactic*), 508
multimatch (*tactic*), 472
multimatch goal (*tactic*), 475
Multiple 'via' options (*error*), 152
Multiple 'warning after' or 'abstract after' options (*error*), 152

N

n_Sn (*term*), 528
nat (*term*), 526
nat_case (*term*), 529
nat_double_ind (*term*), 529
nat_scope, 531
native_cast_no_check (*tactic variant*), 268
native_compute (*tactic*), 281
NativeCompute Profile Filename (*option*), 281
NativeCompute Profiling (*flag*), 281
NativeCompute Timing (*flag*), 281
Nested Proofs Allowed (*flag*), 239
Nested proofs are discouraged and not allowed by default. This error probably means that you forgot to close the last "Proof." with "Qed." or "Defined.". If you really intended to use nested proofs, you can do so by turning the "Nested Proofs Allowed" flag on (*error*), 16
New coercion path ... is ambiguous with existing ... (*warning*), 170

Next Obligation (*command*), 200
nia (*tactic*), 418
Nia Cache (*flag*), 416
No applicable tactic (*error*), 464
No argument name 'ident' (*error*), 552
No discriminable equalities (*error*), 294
No evars (*error*), 266
No focused proof (*error*), 236
No focused proof (No proof-editing in progress) (*error*), 221, 222
No focused proof to restart (*error*), 225
No head constant to reduce (*error*), 278
No matching clauses for match (*error*), 473
No matching clauses for match goal (*error*), 476
No primitive equality found (*error*), 294
No product even after head-reduction (*error*), 255
No progress made (*error*), 452
No such assumption (*error*), 249, 265
No such binder (*error*), 241
No such goal (*error*), 236
No such goal ('ident') (*error*), 227
No such goal ('natural') (*error*), 227
No such goal. (fail) (*error*), 468
No such goal. (Goal selector) (*error*), 460
No such goal. Focus next goal with bullet 'bullet' (*error*), 231
No such hypothesis (*error*), 257–259
No such hypothesis in current goal (*error*), 256
No such hypothesis: 'ident' (*error*), 256
No such label 'ident' (*error*), 68
Non exhaustive pattern matching (*error*), 126
Non strictly positive occurrence of 'ident' in 'type' (*error*), 36
NonCumulative (*attribute*), 85
None (*term*), 526
Nonrecursive Elimination Schemes (*flag*), 307
not (*term*), 524
Not a context variable (*error*), 478
Not a discriminable equality (*error*), 294
Not a primitive equality (*error*), 295
Not a proposition or a type (*error*), 262
Not a valid ring equation (*error*), 422
Not a variable or hypothesis (*error*), 266
Not an evar (*error*), 266
Not an exact proof (*error*), 249
Not an inductive goal with 1 constructor (*error*), 287
Not an inductive goal with 2 constructors (*error*), 287
Not an inductive product (*error*), 286, 290
Not convertible (*error*), 273
Not enough constructors (*error*), 287
Not equal (due to universes) (*error*), 265, 266
Not equal (*error*), 265
Not the right number of induction arguments (*error*), 553
Not the right number of missing arguments (*error*), 241, 251
not_eq_S (*term*), 528
Notation (abbreviation) (*command*), 148
Notation (*command*), 127
Notation levels must range between 0 and 6 (*error*), 512
Notation 'string' is deprecated since 'string'. 'string' (*warning*), 558
Notations for lists, 535
Nothing to do, it is an equality between convertible terms (*error*), 295
Nothing to inject (*error*), 295
Nothing to rewrite (*error*), 452
notT (*term*), 530
notypeclasses refine (*tactic variant*), 250
now (*tactic*), 436
now_show (*tactic*), 273
nra (*tactic*), 418
Nra Cache (*flag*), 416
nsatz (*tactic*), 431
nth (*term*), 535
Number Notation (*command*), 150
numgoals (*tactic*), 480

O

O (*term*), 526
O_S (*term*), 528
Obligation (*command*), 200
Obligation Tactic (*command*), 200
Obligations (*command*), 200
once (*tactic*), 466
only (*tactic*), 460
Opaque (*command*), 283
Open Scope (*command*), 145
Optimize Heap (*command*), 240
Optimize Proof (*command*), 240
optimize_heap (*tactic*), 493
option (*term*), 526
or (*term*), 524
or_introl (*term*), 524
or_intror (*term*), 524
over (*tactic*), 378, 408
overflow in int63 literal 'bigint' (*error*), 152

P

pair (*term*), 526
Parameter (*command*), 13
Parameters (*command*), 13
Parsing Explicit (*flag*), 109
pattern (*tactic*), 280
Peano's arithmetic, 531
plus (*term*), 528
plus_n_O (*term*), 528
plus_n_Sm (*term*), 528
Polymorphic (*attribute*), 85
Polymorphic Inductive Cumulativity (*flag*), 86
Polymorphic universe constraints can only be declared inside sections, use Monomorphic Constraint instead (*error*), 89
Polymorphic universes can only be declared inside sections, use Monomorphic Universe instead (*error*), 89
pose (ssreflect) (*tactic*), 321
pose (*tactic*), 260
pose proof (*tactic variant*), 262
Positivity Checking (*flag*), 215
pred (*term*), 528
pred_Sn (*term*), 528
Prenex Implicits (*command*), 320, 410
Preterm (*command*), 201
Primitive (*command*), 218
Primitive Projections (*flag*), 35
Print (*command*), 201
Print All (*command*), 202
Print All Dependencies (*command*), 209
Print Assumptions (*command*), 209
Print Canonical Projections (*command*), 186
Print Classes (*command*), 171
Print Coercion Paths (*command*), 171
Print Coercions (*command*), 171
Print Custom Grammar (*command*), 142
Print Debug GC (*command*), 240
Print Extraction Blacklist (*command*), 544
Print Extraction Inline (*command*), 541
Print Firstorder Solver (*command*), 413
Print Grammar (*command*), 132
Print Graph (*command*), 171
Print Hint (*command*), 440
Print HintDb (*command*), 440
Print Implicit (*command*), 108
Print Instances (*command*), 181
Print Libraries (*command*), 212
Print LoadPath (*command*), 212
Print Ltac (*command*), 482
Print Ltac Signatures (*command*), 482
Print ML Modules (*command*), 212
Print ML Path (*command*), 213
Print Module (*command*), 70
Print Module Type (*command*), 70
Print Opaque Dependencies (*command*), 209
Print Options (*command*), 10
Print Rewrite HintDb (*command*), 440
Print Rings (*command*), 422
Print Scope (*command*), 148
Print Scopes (*command*), 148
Print Section (*command*), 202
Print Strategies (*command*), 284
Print Strategy (*command*), 284
Print Table (*command*), 10
Print Tables (*command*), 10
Print Transparent Dependencies (*command*), 209
Print Typing Flags (*command*), 216
Print Universes (*command*), 90
Print Visibility (*command*), 148
Printing All (*flag*), 215
Printing Allow Match Default Clause (*flag*), 116
Printing Coercion (*table*), 171
Printing Coercions (*flag*), 171
Printing Compact Contexts (*flag*), 215
Printing Constructor (*table*), 33
Printing Dependent Evars Line (*flag*), 215
Printing Depth (*option*), 215
Printing Existential Instances (*flag*), 100
Printing Factorizable Match Patterns (*flag*), 115
Printing Float (*flag*), 537
Printing Goal Names (*flag*), 239
Printing If (*table*), 116
Printing Implicit (*flag*), 108
Printing Implicit Defensive (*flag*), 108
Printing Let (*table*), 116
Printing Matching (*flag*), 115
Printing Notations (*flag*), 132
Printing Parentheses (*flag*), 132
Printing Primitive Projection Parameters (*flag*), 35
Printing Projections (*flag*), 33
Printing Raw Literals (*flag*), 132
Printing Record (*table*), 33
Printing Records (*flag*), 33
Printing Synth (*flag*), 116
Printing Unfocused (*flag*), 215
Printing Universes (*flag*), 90
Printing Use Implicit Types (*flag*), 110
Printing Width (*option*), 215
Printing Wildcard (*flag*), 116
Private (*attribute*), 28

Private Polymorphic Universes (*flag*), 91
private(matching) (*attribute*), 28
prod (*term*), 526
Program (*attribute*), 198
program (*attribute*), 198
Program Cases (*flag*), 198
Program Generalized Coercion (*flag*), 198
Program Mode (*flag*), 198
Programming, 526
progress (*tactic*), 462
proj1 (*term*), 524
proj2 (*term*), 524
projT1 (*term*), 527
projT2 (*term*), 527
Proof (*command*), 222
Proof `term` (*command*), 222
Proof is not complete. (abstract) (*error*), 481
Proof is not complete. (assert) (*error*), 262
Proof using (*command*), 222
Proof with (*command*), 442
Prop, 10
Property (*command*), 15
Proposition (*command*), 15
psatz (*tactic*), 419
Pwd (*command*), 212

Q

Qed (*command*), 221
Quantifiers, 524
Quit (*command*), 213

R

rapply (*tactic variant*), 251
Record (*command*), 31
Records declared with the keyword Record or Structure cannot be recursive (*error*), 34
Recursion, 529
Recursive Extraction (*command*), 539
Recursive Extraction Library (*command*), 539
red (*tactic*), 277
Redirect (*command*), 214
refine (*attribute*), 181
refine (*tactic*), 249
Refine passed ill-formed term (*error*), 250
refl_identity (*term*), 526
reflexivity (*tactic*), 269
Register (*command*), 217
Register Inline (*command*), 218
Regular Subst Tactic (*flag*), 272
Remark (*command*), 15
remember (*tactic*), 260
Remove (*command*), 9
Remove Hints (*command*), 440
Remove LoadPath (*command*), 212
rename (*tactic*), 259
repeat (*tactic*), 462
replace (*tactic*), 271
Require (*command*), 211
Require Export (*command*), 211
Require Import (*command*), 211
Require inside a module is deprecated and strongly discouraged. You can Require a module at toplevel and optionally Import it inside another one (*warning*), 211
Reserved Infix (*command*), 132
Reserved Notation (*command*), 131
Reset (*command*), 213
Reset Extraction Blacklist (*command*), 544
Reset Extraction Inline (*command*), 541
Reset Initial (*command*), 213
Reset Ltac Profile (*command*), 491
reset ltac profile (*tactic*), 492
Restart (*command*), 225
restart_timer (*tactic*), 470
rev (*term*), 535
Reversible Pattern Implicit (*flag*), 106
revert (*tactic*), 257
revert dependent (*tactic variant*), 257
revgoals (*tactic*), 234
rewrite (ssreflect) (*tactic*), 363
rewrite (*tactic*), 269
rewrite * (*tactic*), 271
rewrite_db (*tactic*), 271
rewrite_strat (*tactic*), 452
Rewriting Schemes (*flag*), 307
right (*tactic variant*), 287
right (*term*), 527
ring (*tactic*), 421
Ring operation should be declared as a morphism (*error*), 425
ring_simplify (*tactic*), 421
rtauto (*tactic*), 412

S

S (*term*), 526
Save (*command*), 221
Scheme (*command*), 305
Search (*command*), 202
Search (ssreflect) (*command*), 405
Search Blacklist (*table*), 208
Search Output Name Only (*flag*), 208
SearchPattern (*command*), 207
SearchRewrite (*command*), 208

Section (*command*), 64
Section variable ‘ident’ occurs implicitly in global declaration ‘qualid’ present in hypothesis ‘ident’ (*error*), 272
Section variable ‘ident’ occurs implicitly in global declaration ‘qualid’ present in the conclusion (*error*), 272
sentence, **7**
Separate Extraction (*command*), 539
Set (*command*), 9
Set (*sort*), 10
set (ssreflect) (*tactic*), 323, 409
set (*tactic*), 259
setoid_reflexivity (*tactic*), 449
setoid_replace (*tactic*), 449
setoid_rewrite (*tactic*), 449
setoid_symmetry (*tactic*), 449
setoid_transitivity (*tactic*), 449
shelve (*tactic*), 235
shelve_unifiable (*tactic*), 235
Short Module Printing (*flag*), 70
Show (*command*), 236
Show Conjectures (*command*), 236
Show Existentials (*command*), 237
Show Goal (*command*), 237
Show Intro (*command*), 236
Show Intros (*command*), 237
Show Lia Profile (*command*), 416
Show Ltac Profile (*command*), 491
show ltac profile (*tactic*), 492
Show Match (*command*), 237
Show Obligation Tactic (*command*), 200
Show Proof (*command*), 236
Show Universes (*command*), 237
Show Zify (*command*), 420
sig (*term*), 527
sig2 (*term*), 527
Signature components for label ‘ident’ do not match (*error*), 68
sigT (*term*), 527
sigT2 (*term*), 527
Silent (*flag*), 214
simpl (*tactic*), 277
simple apply (*tactic variant*), 252
simple destruct (*tactic variant*), 289
simple eapply (*tactic variant*), 252
simple induction (*tactic variant*), 292
simple inversion (*tactic variant*), 299
simple notypeclasses refine (*tactic variant*), 251
simple refine (*tactic variant*), 250
simple subst (*tactic*), 272
Simplex (*flag*), 415
simplify_eq (*tactic*), 304
singel: / (*term*), 532
snd (*term*), 526
solve (*tactic*), 465
Solve All Obligations (*command*), 200
Solve Obligations (*command*), 200
Solve Unification Constraints (*flag*), 266
solve_constraints (*tactic*), 266
Some (*term*), 526
specialize (*tactic variant*), 263
split (*tactic variant*), 287
split_Rabs (*tactic*), 534
split_Rmult (*tactic*), 534
SProp, 10
SProp is disallowed because the "Allow StrictProp" flag is off (*error*), 94
SSReflect: cannot obtain new equations out of ... (*warning*), 333
SsrHave NoTCResolution (*flag*), 359
SsrIdents (*flag*), 317
SsrOldRewriteGoalsOrder (*flag*), 365
SsrRewrite (*flag*), 317
Stack overflow or segmentation fault happens when working with large numbers in ‘type’ (threshold may vary depending on your system limits and on the command executed) (*warning*), 151
start ltac profiling (*tactic*), 492
Statement without assumptions (*error*), 254
stepl (*tactic*), 272
stepr (*tactic*), 273
stop ltac profiling (*tactic*), 492
Strategy (*command*), 283
Strict Implicit (*flag*), 106
Strict Universe Declaration (*flag*), 91
String Notation (*command*), 153
Strongly Strict Implicit (*flag*), 106
Structural Injection (*flag*), 296
Structure (*command*), 31
SubClass (*command*), 170
subst (*tactic*), 272
substitute (*tactic*), 272
suff (*tactic*), 409
suffices (*tactic*), 409
Suggest Proof Using (*flag*), 224
sum (*term*), 526
sumbool (*term*), 527
sumor (*term*), 527
swap (*tactic*), 233
sym_not_eq (*term*), 525
symmetry (*tactic*), 269
Syntax error: [prim:reference]

expected after 'Notation' (in [vernac:command]) (*error*), 154
Syntax error: [prim:reference] expected after [prim:reference] (in [vernac:command]) (*error*), 154

T

tactic, **7**
Tactic failure (*error*), 467
Tactic failure (level 'natural') (*error*), 467
Tactic failure: <tactic closure> fails (*error*), 467
Tactic failure: <tactic closure> succeeds (*error*), 466
Tactic failure: Setoid library not loaded (*error*), 271
Tactic generated a subgoal identical to the original goal (*error*), 271
Tactic Notation (*command*), 158
Tactic Notation 'qualid' is deprecated since 'string'. 'string' (*warning*), 558
Tactic 'qualid' is deprecated since 'string'. 'string' (*warning*), 558
tail (*term*), 535
tauto (*tactic*), 411
template and polymorphism not compatible (*error*), 52
term, **6**
Terms do not have convertible types (*error*), 271
Test (*command*), 9
The "at" syntax isn't available yet for the autorewrite tactic (*error*), 434
The 'abstract after' directive has no effect when the parsing function ('qualid') targets an option type (*warning*), 152
The 'clear implicits' flag must be omitted if implicit annotations are given (*error*), 161
The 'default implicits' flag is incompatible with implicit annotations (*error*), 161
The / modifier may only occur once (*error*), 160
The & modifier may only occur once (*error*), 160
The command has not failed! (*error*), 214
The conclusion of 'type' is not valid it must be built from 'ident' (*error*), 37
The constant 'number' is not a binary64 floating-point value. A closest value 'number' will be used and unambiguously printed 'number'. [inexact-float,parsing] (*warning*), 537
The constructor 'ident' expects 'natural' arguments (*error*), 126
The cumulative attribute can only be used in a polymorphic context (*error*), 86
The elimination predicate term should be of arity 'natural' (for non dependent case) or 'natural' (for dependent case) (*error*), 126
The field 'ident' is missing in 'qualid' (*error*), 68
The file 'ident'.vo contains library 'qualid' and not library 'qualid' (*error*), 211
The recursive argument must be specified (*error*), 552
The reference is not unfoldable (*error*), 284
The reference X was not found in the current environment (*error*), 521
The reference 'qualid' was not found in the current environment (*error*), 283
The relation 'ident' is not a declared reflexive relation. Maybe you need to require the Coq.Classes.RelationClasses library (*error*), 269
The relation 'ident' is not a declared symmetric relation. Maybe you need to require the Coq.Classes.RelationClasses library (*error*), 269
The relation 'ident' is not a declared transitive relation. Maybe you need to require the Coq.Classes.RelationClasses library (*error*), 269
The term 'term' has type 'type' which should be Set, Prop or Type (*error*), 16
The term 'term' has type 'type' while it is expected to have type 'type'' (*error*), 15
The type 'ident' must be registered before this construction can be typechecked (*error*), 218

The variable ident is bound several times in pattern term (*error*), 126
The variable 'ident' is already defined (*error*), 260
The 'natural' th argument of 'ident' must be 'ident' in 'type' (*error*), 28
Theorem (*command*), 15
Theories, 522
There is already an Ltac named 'qualid' (*error*), 482
There is no flag or option with this name: "'setting_name'" (*warning*), 9
There is no flag, option or table with this name: "'setting_name'" (*error*), 9
There is no Ltac named 'qualid' (*error*), 482
There is no qualid-valued table with this name: "'setting_name'" (*error*), 10
There is no string-valued table with this name: "'setting_name'" (*error*), 10
There is nothing to end (*error*), 65
This command does not support this attribute (*warning*), 8
This command is just asserting the names of arguments of 'qualid'. If this is what you want, add ': assert' to silence the warning. If you want to clear implicit arguments, add ': clear implicits'. If you want to clear notation scopes, add ': clear scopes' (*warning*), 161
This object does not support universe names (*error*), 202
This proof is focused, but cannot be unfocused this way (*error*), 227
This tactic has more than one success (*error*), 466
Time (*command*), 214
time (*tactic*), 470
time_constr (*tactic*), 470
Timeout (*command*), 214
timeout (*tactic*), 470
To avoid stack overflow, large numbers in 'type' are interpreted as applications of 'qualid' (*warning*), 151
To rename arguments the 'rename' flag must be specified (*error*), 161
transitivity (*tactic*), 269
Transparent (*command*), 283
Transparent Obligations (*flag*), 201
transparent_abstract (*tactic*), 481
trivial (*tactic*), 433
True (*term*), 524
true (*term*), 526
try (*tactic*), 462
tryif (*tactic*), 465
Trying to mask the absolute name 'qualid'! (*warning*), 70
tt (*term*), 526
Type, 10
type, **6**
type of (*tactic*), 480
Type of 'ident' is not an equality of recognized Σ types: expected one of sig sig2 sigT sigT2 sigT2 ex or ex2 but got 'term' (*error*), 299
Type of 'qualid' seems incompatible with the type of 'qualid'. Expected type is: 'type' instead of 'type'. This might yield ill typed terms when using the notation (*warning*), 154
type_scope, 146
type_term (*tactic*), 480
Typeclass Resolution For Conversion (*flag*), 183
Typeclasses Debug (*flag*), 183
Typeclasses Debug Verbosity (*option*), 183
Typeclasses Dependency Order (*flag*), 183
Typeclasses Depth (*option*), 183
Typeclasses eauto (*command*), 184
typeclasses eauto (*tactic*), 181
Typeclasses Filtered Unification (*flag*), 183
Typeclasses Iterative Deepening (*flag*), 183
Typeclasses Limit Intros (*flag*), 183
Typeclasses Opaque (*command*), 182
Typeclasses Strict Resolution (*flag*), 183
Typeclasses Transparent (*command*), 182
Typeclasses Unique Instances (*flag*), 183
Typeclasses Unique Solutions (*flag*), 183

U

Unable to apply (*error*), 254
Unable to find an instance for the variables 'ident' ... 'ident' (*error*), 290
Unable to find an instance for the variables 'ident'…'ident' (*error*), 251

Unable to infer a match predicate (*error*), 126
Unable to satisfy the rewriting constraints (*error*), 452
Unable to unify 'term' with 'term' (*error*), 251, 266
Unbound context identifier 'ident' (*error*), 478
Unbound [value|constructor] X (*error*), 521
Undeclared universe 'ident' (*error*), 89
Undelimit Scope (*command*), 145
under (*tactic*), 377, 408
Undo (*command*), 225
Unexpected non-option term 'term' while parsing a number notation (*error*), 152
Unexpected non-option term 'term' while parsing a string notation (*error*), 153
Unexpected term 'term' while parsing a number notation (*error*), 152
Unexpected term 'term' while parsing a string notation (*error*), 153
Unfocus (*command*), 225
Unfocused (*command*), 225
unfold (*tactic*), 278
Uniform Inductive Parameters (*flag*), 40
unify (*tactic*), 266
unit (*term*), 526
Universal Lemma Under Conjunction (*flag*), 254
Universe (*command*), 89
Universe Checking (*flag*), 216
Universe inconsistency (*error*), 89
Universe instance should have length 'natural' (*error*), 202
Universe Minimization ToSet (*flag*), 89
Universe Polymorphism (*flag*), 85
universes(cumulative) (*attribute*), 85
universes(monomorphic) (*attribute*), 85
universes(noncumulative) (*attribute*), 86
universes(notemplate) (*attribute*), 52
universes(polymorphic) (*attribute*), 85
universes(template) (*attribute*), 52
Unknown inductive type (*error*), 237
unlock (*tactic*), 381, 409
Unset (*command*), 9
Unshelve (*command*), 235
unshelve (*tactic*), 235
Unused variable 'ident' catches more than one case (*warning*), 117
Use of 'string' Notation is deprecated as it is inconsistent with pattern syntax (*warning*), 144
using (*attribute*), 223

V

value (*term*), 528
Variable (*command*), 13
Variable 'ident' is already declared (*error*), 262
Variables (*command*), 13
Variant (*command*), 28
vm_cast_no_check (*tactic variant*), 267
vm_compute (*tactic*), 281

W

Warnings (*option*), 214
Well founded induction, 529
Well foundedness, 529
well_founded (*term*), 529
When 'term' contains more than one non dependent product the tactic lapply only takes into account the first product (*warning*), 252
with_strategy (*tactic*), 284
without loss (*tactic*), 360
wlog (*tactic*), 360, 409
Wrong bullet 'bullet': Bullet 'bullet' is mandatory here (*error*), 231
Wrong bullet 'bullet': Current bullet 'bullet' is not finished (*error*), 231

Y

You should use the "Proof using [...]." syntax instead of "Proof." to enable skipping this proof which is located inside a section. Give as argument to "Proof using" the list of section variables that are not needed to typecheck the statement but that are required by the proof (*warning*), 566

Z

zify (*tactic*), 419

'class' must be a transparent constant (*error*), 170
'ident' already exists. (Axiom) (*error*), 13
'ident' already exists. (Definition) (*error*), 15
'ident' already exists. (Theorem) (*error*), 16

‘ident’ cannot be defined (*warning*), 34
‘ident’ is already declared as a typeclass (*warning*), 180
‘ident’ is already used (*error*), 255, 259
‘ident’ is declared as a local axiom (*warning*), 13
‘ident’ is not a local definition (*error*), 257
‘ident’ is not an equality of Σ types (*error*), 299
‘ident’ is opaque (*error*), 278
‘ident’ is used in conclusion (*error*), 263
‘ident’ is used in hypothesis ‘ident’ (*error*), 263
‘ident’ is used in the conclusion (*error*), 257
‘ident’ is used in the hypothesis ‘ident’ (*error*), 257
‘ident’: no such entry (*error*), 213
‘qualid’ cannot be used as a hint (*error*), 437
‘qualid’ does not occur (*error*), 279
‘qualid’ does not respect the uniform inheritance condition (*warning*), 170
‘qualid’ is already a coercion (*error*), 170
‘qualid’ is bound to a notation that does not denote a reference (*error*), 154
‘qualid’ is not a function (*error*), 170
‘qualid’ is not a module (*error*), 70
‘qualid’ is not an inductive type (*error*), 438
‘qualid’ not a defined object (*error*), 201
‘qualid’ not declared (*error*), 170
‘qualid’ should go from Byte.byte or (list Byte.byte) to ‘type’ or (option ‘type’) (*error*), 153
‘qualid’ should go from Number.int to ‘type’ or (option ‘type’). Instead of Number.int, the types Number.uint or Z or PrimInt63.pos_neg_int63 or Number.number could be used (you may need to require BinNums or Number or PrimInt63 first) (*error*), 152
‘qualid’ should go from ‘type’ to Byte.byte or (option Byte.byte) or (list Byte.byte) or (option (list Byte.byte)) (*error*), 153
‘qualid’ should go from ‘type’ to Number.int or (option Number.int). Instead of Number.int, the types Number.uint or Z or PrimInt63.pos_neg_int63 or Number.number could be used (you may need to require BinNums or Number or PrimInt63 first) (*error*), 152
‘qualid’ was already mapped to ‘qualid’ and cannot be remapped to ‘qualid’ (*error*), 154
‘type’ is not an inductive type (*error*), 154
‘type’ was already mapped to ‘type’, mapping it also to ‘type’ might yield ill typed terms when using the notation (*warning*), 154
… : … (goal selector) (*tactic*), 460
… : … (ssreflect) (*tactic*), 335