The Coq Proof Assistant
The standard library

January 18, 2021

Version 8.13.0[1]

πr^2 **Project (formerly LogiCal, then TypiCal)**

[1]This research was partly supported by IST working group "Types"

V8.13.0, January 18, 2021

Contents

This document is a short description of the Coq standard library. This library comes with the system as a complement of the core library (the **Init** library ; see the Reference Manual for a description of this library). It provides a set of modules directly available through the `Require` command.

The standard library is composed of the following subdirectories:

Logic Classical logic and dependent equality

Bool Booleans (basic functions and results)

Arith Basic Peano arithmetic

ZArith Basic integer arithmetic

Reals Classical Real Numbers and Analysis

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.)

Relations Relations (definitions and basic results).

Sorting Sorted list (basic definitions and heapsort correctness).

Wellfounded Well-founded relations (basic results).

Program Tactics to deal with dependently-typed programs and their proofs.

Classes Standard type class instances on relations and Coq part of the setoid rewriting tactic.

Each of these subdirectories contains a set of modules, whose specifications (Gallina files) have been roughly, and automatically, pasted in the following pages. There is also a version of this document in HTML format on the WWW, which you can access from the Coq home page at `http://coq.inria.fr/library`.

Chapter 1

Library Coq.Init.Hexadecimal

1.1 Hexadecimal numbers

These numbers coded in base 16 will be used for parsing and printing other Coq numeral datatypes in an human-readable way. See the *Number* `Notation` command. We represent numbers in base 16 as lists of hexadecimal digits, in big-endian order (most significant digit comes first).

```
Require Import Datatypes Specif Decimal.
```

Unsigned integers are just lists of digits. For instance, sixteen is (D1 (D0 Nil))

```
Inductive uint :=
 | Nil
 | D0 (_:uint)
 | D1 (_:uint)
 | D2 (_:uint)
 | D3 (_:uint)
 | D4 (_:uint)
 | D5 (_:uint)
 | D6 (_:uint)
 | D7 (_:uint)
 | D8 (_:uint)
 | D9 (_:uint)
 | Da (_:uint)
 | Db (_:uint)
 | Dc (_:uint)
 | Dd (_:uint)
 | De (_:uint)
 | Df (_:uint).
```

Nil is the number terminator. Taken alone, it behaves as zero, but rather use *D0 Nil* instead, since this form will be denoted as 0, while *Nil* will be printed as *Nil*.

```
Notation zero := (D0 Nil).
```

For signed integers, we use two constructors *Pos* and *Neg*.

```
Variant int := Pos (d:uint) | Neg (d:uint).
```

For decimal numbers, we use two constructors *Hexadecimal* and *HexadecimalExp*, depending on whether or not they are given with an exponent (e.g., 0x1.a2p+01). i is the integral part while f is the fractional part (beware that leading zeroes do matter).

```
Variant hexadecimal :=
 | Hexadecimal (i:int) (f:uint)
 | HexadecimalExp (i:int) (f:uint) (e:Decimal.int).

Scheme Equality for uint.
Scheme Equality for int.
Scheme Equality for hexadecimal.

Delimit Scope hex_uint_scope with huint.

Delimit Scope hex_int_scope with hint.

Fixpoint nb_digits d :=
  match d with
  | Nil ⇒ O
  | D0 d | D1 d | D2 d | D3 d | D4 d | D5 d | D6 d | D7 d | D8 d | D9 d
  | Da d | Db d | Dc d | Dd d | De d | Df d ⇒
    S (nb_digits d)
  end.
```

This representation favors simplicity over canonicity. For normalizing numbers, we need to remove head zero digits, and choose our canonical representation of 0 (here *D0 Nil* for unsigned numbers and *Pos* (*D0 Nil*) for signed numbers).

nzhead removes all head zero digits

```
Fixpoint nzhead d :=
  match d with
  | D0 d ⇒ nzhead d
  | _ ⇒ d
  end.
```

unorm : normalization of unsigned integers

```
Definition unorm d :=
  match nzhead d with
  | Nil ⇒ zero
  | d ⇒ d
  end.
```

norm : normalization of signed integers

```
Definition norm d :=
  match d with
  | Pos d ⇒ Pos (unorm d)
  | Neg d ⇒
    match nzhead d with
    | Nil ⇒ Pos zero
    | d ⇒ Neg d
    end
```

```
end.
```

A few easy operations. For more advanced computations, use the conversions with other Coq numeral datatypes (e.g. Z) and the operations on them.

```
Definition opp (d:int) :=
  match d with
  | Pos d ⇒ Neg d
  | Neg d ⇒ Pos d
  end.

Definition abs (d:int) : uint :=
  match d with
  | Pos d ⇒ d
  | Neg d ⇒ d
  end.
```

For conversions with binary numbers, it is easier to operate on little-endian numbers.

```
Fixpoint revapp (d d' : uint) :=
  match d with
  | Nil ⇒ d'
  | D0 d ⇒ revapp d (D0 d')
  | D1 d ⇒ revapp d (D1 d')
  | D2 d ⇒ revapp d (D2 d')
  | D3 d ⇒ revapp d (D3 d')
  | D4 d ⇒ revapp d (D4 d')
  | D5 d ⇒ revapp d (D5 d')
  | D6 d ⇒ revapp d (D6 d')
  | D7 d ⇒ revapp d (D7 d')
  | D8 d ⇒ revapp d (D8 d')
  | D9 d ⇒ revapp d (D9 d')
  | Da d ⇒ revapp d (Da d')
  | Db d ⇒ revapp d (Db d')
  | Dc d ⇒ revapp d (Dc d')
  | Dd d ⇒ revapp d (Dd d')
  | De d ⇒ revapp d (De d')
  | Df d ⇒ revapp d (Df d')
  end.

Definition rev d := revapp d Nil.

Definition app d d' := revapp (rev d) d'.

Definition app_int d1 d2 :=
  match d1 with Pos d1 ⇒ Pos (app d1 d2) | Neg d1 ⇒ Neg (app d1 d2) end.
```

nztail removes all trailing zero digits and return both the result and the number of removed digits.

```
Definition nztail d :=
  let fix aux d_rev :=
    match d_rev with
```

```
    | D0 d_rev ⇒ let (r, n) := aux d_rev in pair r (S n)
    | _ ⇒ pair d_rev O
    end in
  let (r, n) := aux (rev d) in pair (rev r) n.
```

```
Definition nztail_int d :=
  match d with
  | Pos d ⇒ let (r, n) := nztail d in pair (Pos r) n
  | Neg d ⇒ let (r, n) := nztail d in pair (Neg r) n
  end.
```

del_head n d removes n digits at beginning of d or returns *zero* if d has less than n digits.

```
Fixpoint del_head n d :=
  match n with
  | O ⇒ d
  | S n ⇒
    match d with
    | Nil ⇒ zero
    | D0 d | D1 d | D2 d | D3 d | D4 d | D5 d | D6 d | D7 d | D8 d | D9 d
    | Da d | Db d | Dc d | Dd d | De d | Df d ⇒
      del_head n d
    end
  end.
```

```
Definition del_head_int n d :=
  match d with
  | Pos d ⇒ del_head n d
  | Neg d ⇒ del_head n d
  end.
```

del_tail n d removes n digits at end of d or returns *zero* if d has less than n digits.

```
Definition del_tail n d := rev (del_head n (rev d)).
```

```
Definition del_tail_int n d :=
  match d with
  | Pos d ⇒ Pos (del_tail n d)
  | Neg d ⇒ Neg (del_tail n d)
  end.
```

Module LITTLE.

Successor of little-endian numbers

```
Fixpoint succ d :=
  match d with
  | Nil ⇒ D1 Nil
  | D0 d ⇒ D1 d
  | D1 d ⇒ D2 d
  | D2 d ⇒ D3 d
  | D3 d ⇒ D4 d
  | D4 d ⇒ D5 d
```

```
  | D5 d ⇒ D6 d
  | D6 d ⇒ D7 d
  | D7 d ⇒ D8 d
  | D8 d ⇒ D9 d
  | D9 d ⇒ Da d
  | Da d ⇒ Db d
  | Db d ⇒ Dc d
  | Dc d ⇒ Dd d
  | Dd d ⇒ De d
  | De d ⇒ Df d
  | Df d ⇒ D0 (succ d)
  end.
```

Doubling little-endian numbers

```
Fixpoint double d :=
  match d with
  | Nil ⇒ Nil
  | D0 d ⇒ D0 (double d)
  | D1 d ⇒ D2 (double d)
  | D2 d ⇒ D4 (double d)
  | D3 d ⇒ D6 (double d)
  | D4 d ⇒ D8 (double d)
  | D5 d ⇒ Da (double d)
  | D6 d ⇒ Dc (double d)
  | D7 d ⇒ De (double d)
  | D8 d ⇒ D0 (succ_double d)
  | D9 d ⇒ D2 (succ_double d)
  | Da d ⇒ D4 (succ_double d)
  | Db d ⇒ D6 (succ_double d)
  | Dc d ⇒ D8 (succ_double d)
  | Dd d ⇒ Da (succ_double d)
  | De d ⇒ Dc (succ_double d)
  | Df d ⇒ De (succ_double d)
  end

with succ_double d :=
  match d with
  | Nil ⇒ D1 Nil
  | D0 d ⇒ D1 (double d)
  | D1 d ⇒ D3 (double d)
  | D2 d ⇒ D5 (double d)
  | D3 d ⇒ D7 (double d)
  | D4 d ⇒ D9 (double d)
  | D5 d ⇒ Db (double d)
  | D6 d ⇒ Dd (double d)
  | D7 d ⇒ Df (double d)
```

```
| D8 d ⇒ D1 (succ_double d)
| D9 d ⇒ D3 (succ_double d)
| Da d ⇒ D5 (succ_double d)
| Db d ⇒ D7 (succ_double d)
| Dc d ⇒ D9 (succ_double d)
| Dd d ⇒ Db (succ_double d)
| De d ⇒ Dd (succ_double d)
| Df d ⇒ Df (succ_double d)
end.
End LITTLE.
```

Chapter 2

Library Coq.Init.Ltac

`Export Set` *Default* `Proof` *Mode* "Classic".

Chapter 3

Library Coq.Init.Prelude

```
Require Export Notations.
Require Export Logic.
Require Export Logic_Type.
Require Export Datatypes.
Require Export Specif.
Require Coq.Init.Byte.
Require Coq.Init.Decimal.
Require Coq.Init.Hexadecimal.
Require Coq.Init.Numeral.
Require Coq.Init.Number.
Require Coq.Init.Nat.
Require Export Peano.
Require Export Coq.Init.Wf.
Require Export Coq.Init.Ltac.
Require Export Coq.Init.Tactics.
Require Export Coq.Init.Tauto.
```

```
Export Byte.ByteSyntaxNotations.
```

```
Add Search Blacklist "_subproof" "_subterm" "Private_".
```

Chapter 4

Library Coq.Init.Wf

4.1 This module proves the validity of

- well-founded recursion (also known as course of values)
- well-founded induction

from a well-founded ordering on a given set

```
Set Implicit Arguments.

Require Import Notations.
Require Import Ltac.
Require Import Logic.
Require Import Datatypes.
```

Well-founded induction principle on Prop

```
Section Well_founded.
 Variable A : Type.
 Variable R : A → A → Prop.
```

The accessibility predicate is defined to be non-informative (Acc_rect is automatically defined because Acc is a singleton type)

```
 Inductive Acc (x: A) : Prop :=
     Acc_intro : (∀ y:A, R y x → Acc y) → Acc x.

 Lemma Acc_inv : ∀ x:A, Acc x → ∀ y:A, R y x → Acc y.
```

A relation is well-founded if every element is accessible

```
 Definition well_founded := ∀ a:A, Acc a.
```

Well-founded induction on Set and Prop

```
 Hypothesis Rwf : well_founded.

 Theorem well_founded_induction_type :
  ∀ P:A → Type,
    (∀ x:A, (∀ y:A, R y x → P y) → P x) → ∀ a:A, P a.
```

```
Theorem well_founded_induction :
 ∀ P:A → Set,
   (∀ x:A, (∀ y:A, R y x → P y) → P x) → ∀ a:A, P a.
Theorem well_founded_ind :
 ∀ P:A → Prop,
   (∀ x:A, (∀ y:A, R y x → P y) → P x) → ∀ a:A, P a.
```

Well-founded fixpoints

```
Section FixPoint.
 Variable P : A → Type.
 Variable F : ∀ x:A, (∀ y:A, R y x → P y) → P x.
 Fixpoint Fix_F (x:A) (a:Acc x) : P x :=
   F (fun (y:A) (h:R y x) ⇒ Fix_F (Acc_inv a h)).
 Scheme Acc_inv_dep := Induction for Acc Sort Prop.
 Lemma Fix_F_eq (x:A) (r:Acc x) :
    F (fun (y:A) (p:R y x) ⇒ Fix_F (x:=y) (Acc_inv r p)) = Fix_F (x:=x) r.
 Definition Fix (x:A) := Fix_F (Rwf x).
```

Proof that *well_founded_induction* satisfies the fixpoint equation. It requires an extra property of the functional

```
 Hypothesis
   F_ext :
     ∀ (x:A) (f g:∀ y:A, R y x → P y),
       (∀ (y:A) (p:R y x), f y p = g y p) → F f = F g.
 Lemma Fix_F_inv : ∀ (x:A) (r s:Acc x), Fix_F r = Fix_F s.
 Lemma Fix_eq : ∀ x:A, Fix x = F (fun (y:A) (p:R y x) ⇒ Fix y).
End FixPoint.
End Well_founded.
```

Well-founded fixpoints over pairs

```
Section Well_founded_2.
 Variables A B : Type.
 Variable R : A × B → A × B → Prop.
 Variable P : A → B → Type.
 Section FixPoint_2.
 Variable
   F :
     ∀ (x:A) (x':B),
       (∀ (y:A) (y':B), R (y, y') (x, x') → P y y') → P x x'.
 Fixpoint Fix_F_2 (x:A) (x':B) (a:Acc R (x, x')) : P x x' :=
   F
     (fun (y:A) (y':B) (h:R (y, y') (x, x')) ⇒
```

```
        Fix_F_2 (x:=y) (x':=y') (Acc_inv a (y,y') h)).
  End FixPoint_2.
  Hypothesis Rwf : well_founded R.
  Theorem well_founded_induction_type_2 :
   (∀ (x:A) (x':B),
      (∀ (y:A) (y':B), R (y, y') (x, x') → P y y') → P x x') →
   ∀ (a:A) (b:B), P a b.
End Well_founded_2.
Notation Acc_iter := Fix_F (only parsing). Notation Acc_iter_2 := Fix_F_2 (only parsing).
Section Acc_generator.
  Variable A : Type.
  Variable R : A → A → Prop.
  Fixpoint Acc_intro_generator n (wf : well_founded R) :=
    match n with
        | O ⇒ wf
        | S n ⇒ fun x ⇒ Acc_intro x (fun y _ ⇒ Acc_intro_generator n (Acc_intro_generator n wf)
y)
    end.
End Acc_generator.
```

Chapter 5

Library Coq.Init.Number

5.1 Decimal or Hexadecimal numbers

```
Require Import Decimal Hexadecimal.

Variant uint := UIntDecimal (u:Decimal.uint) | UIntHexadecimal (u:Hexadecimal.uint).
#[deprecated(since="8.13",note="Use UintDecimal instead.")]
Notation UIntDec := UIntDecimal (only parsing).
#[deprecated(since="8.13",note="Use UintHexadecimal instead.")]
Notation UIntHex := UIntHexadecimal (only parsing).

Variant int := IntDecimal (i:Decimal.int) | IntHexadecimal (i:Hexadecimal.int).
#[deprecated(since="8.13",note="Use IntDecimal instead.")]
Notation IntDec := IntDecimal (only parsing).
#[deprecated(since="8.13",note="Use IntHexadecimal instead.")]
Notation IntHex := IntHexadecimal (only parsing).

Variant number := Decimal (d:Decimal.decimal) | Hexadecimal (h:Hexadecimal.hexadecimal).
#[deprecated(since="8.13",note="Use Decimal instead.")]
Notation Dec := Decimal (only parsing).
#[deprecated(since="8.13",note="Use Hexadecimal instead.")]
Notation Hex := Hexadecimal (only parsing).

Scheme Equality for uint.
Scheme Equality for int.
Scheme Equality for number.
```

Pseudo-conversion functions used when declaring Number Notations on *uint* and *int*.

```
Definition uint_of_uint (i:uint) := i.
Definition int_of_int (i:int) := i.
```

Chapter 6

Library Coq.Init.Tactics

```
Require Import Notations.
Require Import Ltac.
Require Import Logic.
Require Import Specif.
```

6.1 Useful tactics

Ex falso quodlibet : a tactic for proving False instead of the current goal. This is just a nicer name for tactics such as `elimtype` *False* and other `cut` *False*.

```
Ltac exfalso := elimtype False.
```

A tactic for proof by contradiction. With contradict H,

- H:~A |- B gives |- A
- H:~A |- ~B gives H: B |- A
- H: A |- B gives |- ~A
- H: A |- ~B gives H: B |- ~A
- H:False leads to a resolved subgoal.

Moreover, negations may be in unfolded forms, and A or B may live in Type

```
Ltac contradict H :=
  let save tac H := let x:=fresh in intro x; tac H; rename x into H
  in
  let negpos H := case H; clear H
  in
  let negneg H := save negpos H
  in
  let pospos H :=
    let A := type of H in (exfalso; revert H; try fold (¬A))
  in
```

```
  let posneg H := save pospos H
  in
  let neg H := match goal with
   | ⊢ (¬_) ⇒ negneg H
   | ⊢ (_→False) ⇒ negneg H
   | ⊢ _ ⇒ negpos H
  end in
  let pos H := match goal with
   | ⊢ (¬_) ⇒ posneg H
   | ⊢ (_→False) ⇒ posneg H
   | ⊢ _ ⇒ pospos H
  end in
  match type of H with
   | (¬_) ⇒ neg H
   | (_→False) ⇒ neg H
   | _ ⇒ (elim H;fail) || pos H
  end.

Ltac absurd_hyp H :=
  idtac "absurd_hyp is OBSOLETE: use contradict instead.";
  let T := type of H in
  absurd T.

Ltac false_hyp H G :=
  let T := type of H in absurd T; [ apply G | assumption ].

Ltac case_eq x := generalize (eq_refl x); pattern x at -1; case x.

Ltac destr_eq H := discriminate H || (try (injection H as [= H])).

Tactic Notation "destruct_with_eqn" constr(x) :=
  destruct x eqn:?.
Tactic Notation "destruct_with_eqn" ident(n) :=
  try intros until n; destruct n eqn:?.
Tactic Notation "destruct_with_eqn" ":" ident(H) constr(x) :=
  destruct x eqn:H.
Tactic Notation "destruct_with_eqn" ":" ident(H) ident(n) :=
  try intros until n; destruct n eqn:H.
```

Break every hypothesis of a certain type

```
Ltac destruct_all t :=
 match goal with
  | x : t ⊢ _ ⇒ destruct x; destruct_all t
  | _ ⇒ idtac
 end.

Tactic Notation "rewrite_all" constr(eq) := repeat rewrite eq in *.
Tactic Notation "rewrite_all" "<-" constr(eq) := repeat rewrite ← eq in *.
```

Tactics for applying equivalences.

The following code provides tactics “apply -> t”, “apply <- t”, “apply -> t in H” and “apply <- t in H”. Here t is a term whose type consists of nested dependent and nondependent products with an equivalence A <-> B as the conclusion. The tactics with “->” in their names apply A -> B while those with “<-” in the name apply B -> A.

```
Ltac find_equiv H :=
let T := type of H in
lazymatch T with
| ?A → ?B ⇒
  let H1 := fresh in
  let H2 := fresh in
  cut A;
  [intro H1; pose proof (H H1) as H2; clear H H1;
   rename H2 into H; find_equiv H |
   clear H]
| ∀ x : ?t, _ ⇒
  let a := fresh "a" in
  let H1 := fresh "H" in
    evar (a : t); pose proof (H a) as H1; unfold a in H1;
    clear a; clear H; rename H1 into H; find_equiv H
| ?A ↔ ?B ⇒ idtac
| _ ⇒ fail "The given statement does not seem to end with an equivalence."
end.
```

```
Ltac bapply lemma todo :=
let H := fresh in
  pose proof lemma as H;
  find_equiv H; [todo H; clear H | .. ].
```

```
Tactic Notation "apply" "->" constr(lemma) :=
bapply lemma ltac:(fun H ⇒ destruct H as [H _]; apply H).
```

```
Tactic Notation "apply" "<-" constr(lemma) :=
bapply lemma ltac:(fun H ⇒ destruct H as [_ H]; apply H).
```

```
Tactic Notation "apply" "->" constr(lemma) "in" hyp(J) :=
bapply lemma ltac:(fun H ⇒ destruct H as [H _]; apply H in J).
```

```
Tactic Notation "apply" "<-" constr(lemma) "in" hyp(J) :=
bapply lemma ltac:(fun H ⇒ destruct H as [_ H]; apply H in J).
```

An experimental tactic simpler than auto that is useful for ending proofs “in one step”

```
Ltac easy :=
  let rec use_hyp H :=
    match type of H with
    | _ ∧ _ ⇒ exact H || destruct_hyp H
    | _ ⇒ try solve [inversion H]
    end
  with do_intro := let H := fresh in intro H; use_hyp H
```

```
  with destruct_hyp H := case H; clear H; do_intro; do_intro in
  let rec use_hyps :=
    match goal with
    | H : _ ∧ _ ⊢ _ ⇒ exact H || (destruct_hyp H; use_hyps)
    | H : _ ⊢ _ ⇒ solve [inversion H]
    | _ ⇒ idtac
    end in
  let do_atom :=
    solve [ trivial with eq_true | reflexivity | symmetry; trivial | contradiction ] in
  let rec do_ccl :=
    try do_atom;
    repeat (do_intro; try do_atom);
    solve [ split; do_ccl ] in
  solve [ do_atom | use_hyps; do_ccl ] ||
  fail "Cannot solve this goal".
Tactic Notation "now" tactic(t) := t; easy.
```

Slightly more than *easy*

```
Ltac easy' := repeat split; simpl; easy || now destruct 1.
```

A tactic to document or check what is proved at some point of a script

```
Ltac now_show c := change c.
```

Support for rewriting decidability statements

```
Set Implicit Arguments.

Lemma decide_left : ∀ (C:Prop) (decide:{C}+{¬C}),
  C → ∀ P:{C}+{¬C}→Prop, (∀ H:C, P (left _ H)) → P decide.

Lemma decide_right : ∀ (C:Prop) (decide:{C}+{¬C}),
  ¬C → ∀ P:{C}+{¬C}→Prop, (∀ H:¬C, P (right _ H)) → P decide.

Tactic Notation "decide" constr(lemma) "with" constr(H) :=
  let try_to_merge_hyps H :=
     try (clear H; intro H) ||
     (let H' := fresh H "bis" in intro H'; try clear H') ||
     (let H' := fresh in intro H'; try clear H') in
  match type of H with
  | ¬ ?C ⇒ apply (decide_right lemma H); try_to_merge_hyps H
  | ?C → False ⇒ apply (decide_right lemma H); try_to_merge_hyps H
  | _ ⇒ apply (decide_left lemma H); try_to_merge_hyps H
  end.
```

Clear an hypothesis and its dependencies

```
Tactic Notation "clear" "dependent" hyp(h) :=
 let rec depclear h :=
  clear h ||
  match goal with
   | H : context [ h ] ⊢ _ ⇒ depclear H; depclear h
```

```
  | H := context [ h ] ⊢ _ ⇒ depclear H; depclear h
  end ||
  fail "hypothesis to clear is used in the conclusion (maybe indirectly)"
 in depclear h.
```

Revert an hypothesis and its dependencies : this is actually generalize dependent...

```
Tactic Notation "revert" "dependent" hyp(h) :=
 generalize dependent h.
```

Provide an error message for dependent induction/dependent destruction that reports an import is required to use it. Importing Coq.Program.Equality will shadow this notation with the actual tactics.

```
Tactic Notation "dependent" "induction" ident(H) :=
  fail "To use dependent induction, first [Require Import Coq.Program.Equality.]".
Tactic Notation "dependent" "destruction" ident(H) :=
  fail "To use dependent destruction, first [Require Import Coq.Program.Equality.]".
```

inversion_sigma

The built-in `inversion` will frequently leave equalities of dependent pairs. When the first type in the pair is an hProp or otherwise simplifies, *inversion_sigma* is useful; it will replace the equality of pairs with a pair of equalities, one involving a term casted along the other. This might also prove useful for writing a version of `inversion` / `dependent destruction` which does not lose information, i.e., does not turn a goal which is provable into one which requires axiom K / UIP.

```
Ltac simpl_proj_exist_in H :=
  repeat match type of H with
           | context G[proj1_sig (exist _ ?x ?p)]
             ⇒ let G' := context G[x] in change G' in H
           | context G[proj2_sig (exist _ ?x ?p)]
             ⇒ let G' := context G[p] in change G' in H
           | context G[projT1 (existT _ ?x ?p)]
             ⇒ let G' := context G[x] in change G' in H
           | context G[projT2 (existT _ ?x ?p)]
             ⇒ let G' := context G[p] in change G' in H
           | context G[proj3_sig (exist2 _ _ ?x ?p ?q)]
             ⇒ let G' := context G[q] in change G' in H
           | context G[projT3 (existT2 _ _ ?x ?p ?q)]
             ⇒ let G' := context G[q] in change G' in H
           | context G[sig_of_sig2 (@exist2 ?A ?P ?Q ?x ?p ?q)]
             ⇒ let G' := context G[@exist A P x p] in change G' in H
           | context G[sigT_of_sigT2 (@existT2 ?A ?P ?Q ?x ?p ?q)]
             ⇒ let G' := context G[@existT A P x p] in change G' in H
           end.
Ltac induction_sigma_in_using H rect :=
  let H0 := fresh H in
  let H1 := fresh H in
```

```
  induction H as [H0 H1] using (rect _ _ _ _);
  simpl_proj_exist_in H0;
  simpl_proj_exist_in H1.
Ltac induction_sigma2_in_using H rect :=
  let H0 := fresh H in
  let H1 := fresh H in
  let H2 := fresh H in
  induction H as [H0 H1 H2] using (rect _ _ _ _ _);
  simpl_proj_exist_in H0;
  simpl_proj_exist_in H1;
  simpl_proj_exist_in H2.
Ltac inversion_sigma_step :=
  match goal with
  | [ H : _ = exist _ _ _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sig_rect
  | [ H : _ = existT _ _ _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sigT_rect
  | [ H : exist _ _ _ = _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sig_rect
  | [ H : existT _ _ _ = _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sigT_rect
  | [ H : _ = exist2 _ _ _ _ _ ⊢ _ ]
    ⇒ induction_sigma2_in_using H @eq_sig2_rect
  | [ H : _ = existT2 _ _ _ _ _ ⊢ _ ]
    ⇒ induction_sigma2_in_using H @eq_sigT2_rect
  | [ H : exist2 _ _ _ _ _ = _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sig2_rect
  | [ H : existT2 _ _ _ _ _ = _ ⊢ _ ]
    ⇒ induction_sigma_in_using H @eq_sigT2_rect
  end.
Ltac inversion_sigma := repeat inversion_sigma_step.
```

A version of *time* that works for constrs

```
Ltac time_constr tac :=
  let eval_early := match goal with _ ⇒ restart_timer end in
  let ret := tac () in
  let eval_early := match goal with _ ⇒ finish_timing ( "Tactic evaluation" ) end in
  ret.
```

Useful combinators

```
Ltac assert_fails tac :=
  tryif (once tac) then gfail 0 tac "succeeds" else idtac.
Ltac assert_succeeds tac :=
  tryif (assert_fails tac) then gfail 0 tac "fails" else idtac.
Tactic Notation "assert_succeeds" tactic3(tac) :=
  assert_succeeds tac.
```

```
Tactic Notation "assert_fails" tactic3(tac) :=
  assert_fails tac.
#[global]
Hint Variables Opaque : rewrite.
```

Chapter 7

Library Coq.Init.Nat

```
Require Import Notations Logic Datatypes.
Require Decimal Hexadecimal Number.
Local Open Scope nat_scope.
```

7.1 Peano natural numbers, definitions of operations

This file is meant to be used as a whole module, without importing it, leading to qualified definitions (e.g. Nat.pred)

```
Definition t := nat.
```

7.1.1 Constants

```
Definition zero := 0.
Definition one := 1.
Definition two := 2.
```

7.1.2 Basic operations

```
Definition succ := S.

Definition pred n :=
  match n with
    | 0 ⇒ n
    | S u ⇒ u
  end.

Fixpoint add n m :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (p + m)
  end
```

```
where "n + m" := (add n m) : nat_scope.

Definition double n := n + n.

Fixpoint mul n m :=
  match n with
  | 0 ⇒ 0
  | S p ⇒ m + p × m
  end

where "n * m" := (mul n m) : nat_scope.
```

Truncated subtraction: n-m is 0 if $n \leq m$

```
Fixpoint sub n m :=
  match n, m with
  | S k, S l ⇒ k - l
  | _, _ ⇒ n
  end

where "n - m" := (sub n m) : nat_scope.
```

7.1.3 Comparisons

```
Fixpoint eqb n m : bool :=
  match n, m with
    | 0, 0 ⇒ true
    | 0, S _ ⇒ false
    | S _, 0 ⇒ false
    | S n', S m' ⇒ eqb n' m'
  end.

Fixpoint leb n m : bool :=
  match n, m with
    | 0, _ ⇒ true
    | _, 0 ⇒ false
    | S n', S m' ⇒ leb n' m'
  end.

Definition ltb n m := leb (S n) m.

Infix "=?" := eqb (at level 70) : nat_scope.
Infix "<=?" := leb (at level 70) : nat_scope.
Infix "<?" := ltb (at level 70) : nat_scope.

Fixpoint compare n m : comparison :=
  match n, m with
   | 0, 0 ⇒ Eq
   | 0, S _ ⇒ Lt
   | S _, 0 ⇒ Gt
```

```
    | S n', S m' ⇒ compare n' m'
  end.
Infix "?=" := compare (at level 70) : nat_scope.
```

7.1.4 Minimum, maximum

```
Fixpoint max n m :=
  match n, m with
    | 0, _ ⇒ m
    | S n', 0 ⇒ n
    | S n', S m' ⇒ S (max n' m')
  end.
Fixpoint min n m :=
  match n, m with
    | 0, _ ⇒ 0
    | S n', 0 ⇒ 0
    | S n', S m' ⇒ S (min n' m')
  end.
```

7.1.5 Parity tests

```
Fixpoint even n : bool :=
  match n with
    | 0 ⇒ true
    | 1 ⇒ false
    | S (S n') ⇒ even n'
  end.
Definition odd n := negb (even n).
```

7.1.6 Power

```
Fixpoint pow n m :=
  match m with
    | 0 ⇒ 1
    | S m ⇒ n × (n^m)
  end

where "n ^ m" := (pow n m) : nat_scope.
```

7.1.7 Tail-recursive versions of *add* and *mul*

```
Fixpoint tail_add n m :=
  match n with
    | O ⇒ m
    | S n ⇒ tail_add n (S m)
```

```
end.
```

tail_addmul r n m is $r + n \times m$.

```
Fixpoint tail_addmul r n m :=
  match n with
    | O ⇒ r
    | S n ⇒ tail_addmul (tail_add m r) n m
  end.

Definition tail_mul n m := tail_addmul 0 n m.
```

7.1.8 Conversion with a decimal representation for printing/parsing

```
Fixpoint of_uint_acc (d:Decimal.uint)(acc:nat) :=
  match d with
  | Decimal.Nil ⇒ acc
  | Decimal.D0 d ⇒ of_uint_acc d (tail_mul ten acc)
  | Decimal.D1 d ⇒ of_uint_acc d (S (tail_mul ten acc))
  | Decimal.D2 d ⇒ of_uint_acc d (S (S (tail_mul ten acc)))
  | Decimal.D3 d ⇒ of_uint_acc d (S (S (S (tail_mul ten acc))))
  | Decimal.D4 d ⇒ of_uint_acc d (S (S (S (S (tail_mul ten acc)))))
  | Decimal.D5 d ⇒ of_uint_acc d (S (S (S (S (S (tail_mul ten acc))))))
  | Decimal.D6 d ⇒ of_uint_acc d (S (S (S (S (S (S (tail_mul ten acc)))))))
  | Decimal.D7 d ⇒ of_uint_acc d (S (S (S (S (S (S (S (tail_mul ten acc))))))))
  | Decimal.D8 d ⇒ of_uint_acc d (S (S (S (S (S (S (S (S (tail_mul ten acc)))))))))
  | Decimal.D9 d ⇒ of_uint_acc d (S (S (S (S (S (S (S (S (S (tail_mul ten acc))))))))))
  end.

Definition of_uint (d:Decimal.uint) := of_uint_acc d O.

Fixpoint of_hex_uint_acc (d:Hexadecimal.uint)(acc:nat) :=
  match d with
  | Hexadecimal.Nil ⇒ acc
  | Hexadecimal.D0 d ⇒ of_hex_uint_acc d (tail_mul sixteen acc)
  | Hexadecimal.D1 d ⇒ of_hex_uint_acc d (S (tail_mul sixteen acc))
  | Hexadecimal.D2 d ⇒ of_hex_uint_acc d (S (S (tail_mul sixteen acc)))
  | Hexadecimal.D3 d ⇒ of_hex_uint_acc d (S (S (S (tail_mul sixteen acc))))
  | Hexadecimal.D4 d ⇒ of_hex_uint_acc d (S (S (S (S (tail_mul sixteen acc)))))
  | Hexadecimal.D5 d ⇒ of_hex_uint_acc d (S (S (S (S (S (tail_mul sixteen acc))))))
  | Hexadecimal.D6 d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (tail_mul sixteen acc)))))))
  | Hexadecimal.D7 d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (tail_mul sixteen acc))))))))
  | Hexadecimal.D8 d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (tail_mul sixteen acc)))))))))
  | Hexadecimal.D9 d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc))))))))))
  | Hexadecimal.Da d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc)))))))))))
  | Hexadecimal.Db d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc))))))))))))
  | Hexadecimal.Dc d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc)))))))))))))
```

```
  | Hexadecimal.Dd d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc))))))))))))))
  | Hexadecimal.De d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc)))))))))))))))
  | Hexadecimal.Df d ⇒ of_hex_uint_acc d (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (tail_mul sixteen acc))))))))))))))))
  end.

Definition of_hex_uint (d:Hexadecimal.uint) := of_hex_uint_acc d O.

Definition of_num_uint (d:Number.uint) :=
  match d with
  | Number.UIntDecimal d ⇒ of_uint d
  | Number.UIntHexadecimal d ⇒ of_hex_uint d
  end.

Fixpoint to_little_uint n acc :=
  match n with
  | O ⇒ acc
  | S n ⇒ to_little_uint n (Decimal.Little.succ acc)
  end.

Definition to_uint n :=
  Decimal.rev (to_little_uint n Decimal.zero).

Fixpoint to_little_hex_uint n acc :=
  match n with
  | O ⇒ acc
  | S n ⇒ to_little_hex_uint n (Hexadecimal.Little.succ acc)
  end.

Definition to_hex_uint n :=
  Hexadecimal.rev (to_little_hex_uint n Hexadecimal.zero).

Definition to_num_uint n := Number.UIntDecimal (to_uint n).

Definition to_num_hex_uint n := Number.UIntHexadecimal (to_hex_uint n).

Definition of_int (d:Decimal.int) : option nat :=
  match Decimal.norm d with
    | Decimal.Pos u ⇒ Some (of_uint u)
    | _ ⇒ None
  end.

Definition of_hex_int (d:Hexadecimal.int) : option nat :=
  match Hexadecimal.norm d with
    | Hexadecimal.Pos u ⇒ Some (of_hex_uint u)
    | _ ⇒ None
  end.

Definition of_num_int (d:Number.int) : option nat :=
  match d with
  | Number.IntDecimal d ⇒ of_int d
```

```
  | Number.IntHexadecimal d ⇒ of_hex_int d
  end.
Definition to_int n := Decimal.Pos (to_uint n).
Definition to_hex_int n := Hexadecimal.Pos (to_hex_uint n).
Definition to_num_int n := Number.IntDecimal (to_int n).
```

7.1.9 Euclidean division

This division is linear and tail-recursive. In *divmod*, y is the predecessor of the actual divisor, and u is y minus the real remainder

```
Fixpoint divmod x y q u :=
  match x with
    | 0 ⇒ (q,u)
    | S x' ⇒ match u with
                 | 0 ⇒ divmod x' y (S q) y
                 | S u' ⇒ divmod x' y q u'
               end
  end.
Definition div x y :=
  match y with
    | 0 ⇒ y
    | S y' ⇒ fst (divmod x y' 0 y')
  end.
Definition modulo x y :=
  match y with
    | 0 ⇒ y
    | S y' ⇒ y' - snd (divmod x y' 0 y')
  end.
Infix "/" := div : nat_scope.
Infix "mod" := modulo (at level 40, no associativity) : nat_scope.
```

7.1.10 Greatest common divisor

We use Euclid algorithm, which is normally not structural, but Coq is now clever enough to accept this (behind modulo there is a subtraction, which now preserves being a subterm)

```
Fixpoint gcd a b :=
  match a with
   | O ⇒ b
   | S a' ⇒ gcd (b mod (S a')) (S a')
  end.
```

7.1.11 Square

```
Definition square n := n × n.
```

7.1.12 Square root

The following square root function is linear (and tail-recursive). With Peano representation, we can't do better. For faster algorithm, see Psqrt/Zsqrt/Nsqrt...

We search the square root of n = k + p^2 + (q - r) with q = 2p and 0<=r<=q. We start with p=q=r=0, hence looking for the square root of n = k. Then we progressively decrease k and r. When k = S k' and r=0, it means we can use (S p) as new sqrt candidate, since (S k')+p^2+2p = k'+(S p)^2. When k reaches 0, we have found the biggest p^2 square contained in n, hence the square root of n is p.

```
Fixpoint sqrt_iter k p q r :=
  match k with
    | O ⇒ p
    | S k' ⇒ match r with
                | O ⇒ sqrt_iter k' (S p) (S (S q)) (S (S q))
                | S r' ⇒ sqrt_iter k' p q r'
              end
  end.
Definition sqrt n := sqrt_iter n 0 0 0.
```

7.1.13 Log2

This base-2 logarithm is linear and tail-recursive.

In *log2_iter*, we maintain the logarithm p of the counter q, while r is the distance between q and the next power of 2, more precisely $q + S\ r = 2\hat{}(S\ p)$ and $r<2\hat{}p$. At each recursive call, q goes up while r goes down. When r is 0, we know that q has almost reached a power of 2, and we increase p at the next call, while resetting r to q.

Graphically (numbers are q, stars are r) :

```
                    10
                  9
                8
              7   *
            6       *
          5           ...
        4
      3   *
    2       *
  1   *       *
0   *   *       *
```

We stop when k, the global downward counter reaches 0. At that moment, q is the number we're considering (since $k+q$ is invariant), and p its logarithm.

```
Fixpoint log2_iter k p q r :=
  match k with
    | O ⇒ p
    | S k' ⇒ match r with
```

```
          | O ⇒ log2_iter k' (S p) (S q) q
          | S r' ⇒ log2_iter k' p (S q) r'
        end
  end.
Definition log2 n := log2_iter (pred n) 0 1 0.
```

Iterator on natural numbers

```
Definition iter (n:nat) {A} (f:A→A) (x:A) : A :=
 nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) n.
```

Bitwise operations

We provide here some bitwise operations for unary numbers. Some might be really naive, they are just there for fulfilling the same interface as other for natural representations. As soon as binary representations such as NArith are available, it is clearly better to convert to/from them and use their ops.

```
Fixpoint div2 n :=
  match n with
  | 0 ⇒ 0
  | S 0 ⇒ 0
  | S (S n') ⇒ S (div2 n')
  end.

Fixpoint testbit a n : bool :=
 match n with
   | 0 ⇒ odd a
   | S n ⇒ testbit (div2 a) n
 end.

Definition shiftl a := nat_rect _ a (fun _ ⇒ double).
Definition shiftr a := nat_rect _ a (fun _ ⇒ div2).

Fixpoint bitwise (op:bool→bool→bool) n a b :=
 match n with
  | 0 ⇒ 0
  | S n' ⇒
    (if op (odd a) (odd b) then 1 else 0) +
    2*(bitwise op n' (div2 a) (div2 b))
 end.

Definition land a b := bitwise andb a a b.
Definition lor a b := bitwise orb (max a b) a b.
Definition ldiff a b := bitwise (fun b b' ⇒ andb b (negb b')) a a b.
Definition lxor a b := bitwise xorb (max a b) a b.
```

Chapter 8

Library Coq.Init.Logic

```
Set Implicit Arguments.
Require Export Notations.
Require Import Ltac.
Notation "A -> B" := (∀ (_ : A), B) : type_scope.
```

8.1 Propositional connectives

True is the always true proposition

```
Inductive True : Prop :=
  I : True.
```

False is the always false proposition `Inductive False : Prop :=.`

not A, written $\neg A$, is the negation of A `Definition not (A:Prop) := A → False.`

```
Notation "~ x" := (not x) : type_scope.
```

Create the "core" hint database, and set its transparent state for variables and constants explicitly.

```
#[global]
Hint Variables Opaque : core.
#[global]
Hint Constants Opaque : core.
#[global]
Hint Unfold not: core.
```

and A B, written $A \wedge B$, is the conjunction of A and B
conj p q is a proof of $A \wedge B$ as soon as p is a proof of A and q a proof of B
proj1 and *proj2* are first and second projections of a conjunction

```
Inductive and (A B:Prop) : Prop :=
  conj : A → B → A ∧ B
```

```
where "A /\ B" := (and A B) : type_scope.

Section Conjunction.

  Variables A B : Prop.

  Theorem proj1 : A /\ B -> A.

  Theorem proj2 : A /\ B -> B.

End Conjunction.
```

or A B, written $A \vee B$, is the disjunction of A and B

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

where "A \/ B" := (or A B) : type_scope.
```

iff A B, written $A \leftrightarrow B$, expresses the equivalence of A and B

```
Definition iff (A B:Prop) := (A -> B) /\ (B -> A).

Notation "A <-> B" := (iff A B) : type_scope.

Section Equivalence.

Theorem iff_refl : forall A:Prop, A <-> A.

Theorem iff_trans : forall A B C:Prop, (A <-> B) -> (B <-> C) -> (A <-> C).

Theorem iff_sym : forall A B:Prop, (A <-> B) -> (B <-> A).

End Equivalence.

#[global]
Hint Unfold iff: extcore.
```

Backward direction of the equivalences above does not need assumptions

```
Theorem and_iff_compat_l : forall A B C : Prop,
  (B <-> C) -> (A /\ B <-> A /\ C).

Theorem and_iff_compat_r : forall A B C : Prop,
  (B <-> C) -> (B /\ A <-> C /\ A).

Theorem or_iff_compat_l : forall A B C : Prop,
  (B <-> C) -> (A \/ B <-> A \/ C).

Theorem or_iff_compat_r : forall A B C : Prop,
  (B <-> C) -> (B \/ A <-> C \/ A).

Theorem imp_iff_compat_l : forall A B C : Prop,
  (B <-> C) -> ((A -> B) <-> (A -> C)).

Theorem imp_iff_compat_r : forall A B C : Prop,
  (B <-> C) -> ((B -> A) <-> (C -> A)).

Theorem not_iff_compat : forall A B : Prop,
  (A <-> B) -> (~ A <-> ~B).
```

Some equivalences

Theorem neg_false : ∀ A : Prop, ¬ $A \leftrightarrow (A \leftrightarrow$ **False**).

Theorem and_cancel_l : ∀ $A\ B\ C$: Prop,
$(B \to A) \to (C \to A) \to ((A \land B \leftrightarrow A \land C) \leftrightarrow (B \leftrightarrow C))$.

Theorem and_cancel_r : ∀ $A\ B\ C$: Prop,
$(B \to A) \to (C \to A) \to ((B \land A \leftrightarrow C \land A) \leftrightarrow (B \leftrightarrow C))$.

Theorem and_comm : ∀ $A\ B$: Prop, $A \land B \leftrightarrow B \land A$.

Theorem and_assoc : ∀ $A\ B\ C$: Prop, $(A \land B) \land C \leftrightarrow A \land B \land C$.

Theorem or_cancel_l : ∀ $A\ B\ C$: Prop,
$(B \to \neg A) \to (C \to \neg A) \to ((A \lor B \leftrightarrow A \lor C) \leftrightarrow (B \leftrightarrow C))$.
Theorem or_cancel_r : ∀ $A\ B\ C$: Prop,
$(B \to \neg A) \to (C \to \neg A) \to ((B \lor A \leftrightarrow C \lor A) \leftrightarrow (B \leftrightarrow C))$.
Theorem or_comm : ∀ $A\ B$: Prop, $(A \lor B) \leftrightarrow (B \lor A)$.

Theorem or_assoc : ∀ $A\ B\ C$: Prop, $(A \lor B) \lor C \leftrightarrow A \lor B \lor C$.
Lemma iff_and : ∀ $A\ B$: Prop, $(A \leftrightarrow B) \to (A \to B) \land (B \to A)$.

Lemma iff_to_and : ∀ $A\ B$: Prop, $(A \leftrightarrow B) \leftrightarrow (A \to B) \land (B \to A)$.

(*IF_then_else* $P\ Q\ R$), written *IF* P then Q else R denotes either P and Q, or $\neg P$ and R

Definition IF_then_else ($P\ Q\ R$:Prop) := $P \land Q \lor \neg P \land R$.

Notation "'IF' c1 'then' c2 'else' c3" := (IF_then_else *c1 c2 c3*)
(at level 200, right associativity) : *type_scope*.

8.2 First-order quantifiers

ex P, or simply ∃ x, $P\ x$, or also ∃ x:A, $P\ x$, expresses the existence of an x of some type A in Set which satisfies the predicate P. This is existential quantification.

ex2 P Q, or simply *exists2* x, $P\ x$ & $Q\ x$, or also *exists2* x:A, $P\ x$ & $Q\ x$, expresses the existence of an x of type A which satisfies both predicates P and Q.

Universal quantification is primitively written ∀ x:A, Q. By symmetry with existential quantification, the construction *all P* is provided too.

```
Inductive ex (A:Type) (P:A → Prop) : Prop :=
  ex_intro : ∀ x:A, P x → ex (A:=A) P.

Section Projections.
  Variables (A:Prop) (P:A→Prop).
  Definition ex_proj1 (x:ex P) : A :=
    match x with ex_intro _ a _ ⇒ a end.
  Definition ex_proj2 (x:ex P) : P (ex_proj1 x) :=
    match x with ex_intro _ _ b ⇒ b end.
End Projections.

Inductive ex2 (A:Type) (P Q:A → Prop) : Prop :=
```

```
ex_intro2 : ∀ x:A, P x → Q x → ex2 (A:=A) P Q.
Definition all (A:Type) (P:A → Prop) := ∀ x:A, P x.

Notation "'exists' x .. y , p" := (ex (fun x ⇒ .. (ex (fun y ⇒ p)) ..))
  (at level 200, x binder, right associativity,
   format "'[' 'exists' '/ ' x .. y , '/ ' p ']'")
  : type_scope.

Notation "'exists2' x , p & q" := (ex2 (fun x ⇒ p) (fun x ⇒ q))
  (at level 200, x name, p at level 200, right associativity) : type_scope.
Notation "'exists2' x : A , p & q" := (ex2 (A:=A) (fun x ⇒ p) (fun x ⇒ q))
  (at level 200, x name, A at level 200, p at level 200, right associativity,
    format "'[' 'exists2' '/ ' x : A , '/ ' '[' p & '/' q ']' ']'")
  : type_scope.

Notation "'exists2' ' x , p & q" := (ex2 (fun x ⇒ p) (fun x ⇒ q))
  (at level 200, x strict pattern, p at level 200, right associativity) : type_scope.
Notation "'exists2' ' x : A , p & q" := (ex2 (A:=A) (fun x ⇒ p) (fun x ⇒ q))
  (at level 200, x strict pattern, A at level 200, p at level 200, right associativity,
    format "'[' 'exists2' '/ ' ' x : A , '/ ' '[' p & '/' q ']' ']'")
  : type_scope.
```

Derived rules for universal quantification

```
Section universal_quantification.

  Variable A : Type.
  Variable P : A → Prop.

  Theorem inst : ∀ x:A, all (fun x ⇒ P x) → P x.

  Theorem gen : ∀ (B:Prop) (f:∀ y:A, B → P y), B → all P.

End universal_quantification.
```

8.3 Equality

eq x y, or simply $x{=}y$ expresses the equality of x and y. Both x and y must belong to the same type A. The definition is inductive and states the reflexivity of the equality. The others properties (symmetry, transitivity, replacement of equals by equals) are proved below. The type of x and y can be made explicit using the notation $x = y$:> A. This is Leibniz equality as it expresses that x and y are equal iff every property on A which is true of x is also true of y

```
Inductive eq (A:Type) (x:A) : A → Prop :=
    eq_refl : x = x :>A

where "x = y :> A" := (@eq A x y) : type_scope.

Notation "x = y" := (eq x y) : type_scope.
Notation "x <> y :> T" := (¬ x = y :>T) : type_scope.
Notation "x <> y" := (¬ (x = y)) : type_scope.
```

```
#[global]
Hint Resolve I conj or_introl or_intror : core.
#[global]
Hint Resolve eq_refl: core.
#[global]
Hint Resolve ex_intro ex_intro2: core.

Section Logic_lemmas.
  Theorem absurd : ∀ 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 eq_trans_r : x = y → z = y → x = z.
    Theorem f_equal : x = y → f x = f y.

    Theorem not_eq_sym : x ≠ y → y ≠ x.
  End equality.
  Definition eq_sind_r :
    ∀ (A:Type) (x:A) (P:A → SProp), P x → ∀ y:A, y = x → P y.
  Definition eq_ind_r :
    ∀ (A:Type) (x:A) (P:A → Prop), P x → ∀ y:A, y = x → P y.
  Defined.

  Definition eq_rec_r :
    ∀ (A:Type) (x:A) (P:A → Set), P x → ∀ y:A, y = x → P y.
  Defined.
  Definition eq_rect_r :
    ∀ (A:Type) (x:A) (P:A → Type), P x → ∀ y:A, y = x → P y.
  Defined.
End Logic_lemmas.
Module EqNotations.
  Notation "'rew' H 'in' H'" := (eq_rect _ _ H' _ H)
    (at level 10, H' at level 10,
     format "'[' 'rew' H in '/' H' ']'").
  Notation "'rew' [ P ] H 'in' H'" := (eq_rect _ P H' _ H)
    (at level 10, H' at level 10,
     format "'[' 'rew' [ P ] '/ ' H in '/' H' ']'").
  Notation "'rew' <- H 'in' H'" := (eq_rect_r _ H' H)
    (at level 10, H' at level 10,
     format "'[' 'rew' <- H in '/' H' ']'").
```

```
Notation "'rew' <- [ P ] H 'in' H'" := (eq_rect_r P H' H)
  (at level 10, H' at level 10,
   format "'[' 'rew' <- [ P ] '/ ' H in '/' H' ']'").
Notation "'rew' -> H 'in' H'" := (eq_rect _ _ H' _ H)
  (at level 10, H' at level 10, only parsing).
Notation "'rew' -> [ P ] H 'in' H'" := (eq_rect _ P H' _ H)
  (at level 10, H' at level 10, only parsing).

Notation "'rew' 'dependent' H 'in' H'"
  := (match H with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10,
        format "'[' 'rew' 'dependent' '/ ' H in '/' H' ']'").
Notation "'rew' 'dependent' -> H 'in' H'"
  := (match H with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10, only parsing).
Notation "'rew' 'dependent' <- H 'in' H'"
  := (match eq_sym H with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10,
        format "'[' 'rew' 'dependent' <- '/ ' H in '/' H' ']'").
Notation "'rew' 'dependent' [ 'fun' y p => P ] H 'in' H'"
  := (match H as p in (_ = y) return P with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10, y name, p name,
        format "'[' 'rew' 'dependent' [ 'fun' y p => P ] '/ ' H in '/' H' ']'").
Notation "'rew' 'dependent' -> [ 'fun' y p => P ] H 'in' H'"
  := (match H as p in (_ = y) return P with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10, y name, p name, only parsing).
Notation "'rew' 'dependent' <- [ 'fun' y p => P ] H 'in' H'"
  := (match eq_sym H as p in (_ = y) return P with
      | eq_refl ⇒ H'
      end)
       (at level 10, H' at level 10, y name, p name,
        format "'[' 'rew' 'dependent' <- [ 'fun' y p => P ] '/ ' H in '/' H' ']'").
Notation "'rew' 'dependent' [ P ] H 'in' H'"
  := (match H as p in (_ = y) return P y p with
      | eq_refl ⇒ H'
      end)
```

```
        (at level 10, H' at level 10,
         format "'[' 'rew' 'dependent' [ P ] '/ ' H in '/' H' ']'").
  Notation "'rew' 'dependent' -> [ P ] H 'in' H'"
    := (match H as p in (_ = y) return P y p with
        | eq_refl ⇒ H'
        end)
        (at level 10, H' at level 10,
         only parsing).
  Notation "'rew' 'dependent' <- [ P ] H 'in' H'"
    := (match eq_sym H as p in (_ = y) return P y p with
        | eq_refl ⇒ H'
        end)
        (at level 10, H' at level 10,
         format "'[' 'rew' 'dependent' <- [ P ] '/ ' H in '/' H' ']'").
End EqNotations.

Import EqNotations.

Section equality_dep.
  Variable A : Type.
  Variable B : A → Type.
  Variable f : ∀ x, B x.
  Variables x y : A.

  Theorem f_equal_dep (H: x = y) : rew H in f x = f y.

End equality_dep.

Lemma f_equal_dep2 {A A' B B'} (f : A → A') (g : ∀ a:A, B a → B' (f a))
  {x1 x2 : A} {y1 : B x1} {y2 : B x2} (H : x1 = x2) :
  rew H in y1 = y2 → rew f_equal f H in g x1 y1 = g x2 y2.

Lemma rew_opp_r A (P:A→Type) (x y:A) (H:x=y) (a:P y) : rew H in rew ← H in a = a.

Lemma rew_opp_l A (P:A→Type) (x y:A) (H:x=y) (a:P x) : rew ← H in rew H in a = a.

Theorem f_equal2 :
  ∀ (A1 A2 B:Type) (f:A1 → A2 → B) (x1 y1:A1)
    (x2 y2:A2), x1 = y1 → x2 = y2 → f x1 x2 = f y1 y2.

Theorem f_equal3 :
  ∀ (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.

Theorem f_equal4 :
  ∀ (A1 A2 A3 A4 B:Type) (f:A1 → A2 → A3 → A4 → B)
    (x1 y1:A1) (x2 y2:A2) (x3 y3:A3) (x4 y4:A4),
    x1 = y1 → x2 = y2 → x3 = y3 → x4 = y4 → f x1 x2 x3 x4 = f y1 y2 y3 y4.

Theorem f_equal5 :
  ∀ (A1 A2 A3 A4 A5 B:Type) (f:A1 → A2 → A3 → A4 → A5 → B)
    (x1 y1:A1) (x2 y2:A2) (x3 y3:A3) (x4 y4:A4) (x5 y5:A5),
```

$x1$ = $y1$ →
$x2$ = $y2$ →
$x3$ = $y3$ → $x4$ = $y4$ → $x5$ = $y5$ → f $x1$ $x2$ $x3$ $x4$ $x5$ = f $y1$ $y2$ $y3$ $y4$ $y5$.

Theorem f_equal_compose A B C (a b:A) (f:A→B) (g:B→C) (e:a=b) :
f_equal g (f_equal f e) = f_equal (fun a ⇒ g (f a)) e.

The groupoid structure of equality

Theorem eq_trans_refl_l A (x y:A) (e:x=y) : eq_trans eq_refl e = e.

Theorem eq_trans_refl_r A (x y:A) (e:x=y) : eq_trans e eq_refl = e.

Theorem eq_sym_involutive A (x y:A) (e:x=y) : eq_sym (eq_sym e) = e.

Theorem eq_trans_sym_inv_l A (x y:A) (e:x=y) : eq_trans (eq_sym e) e = eq_refl.

Theorem eq_trans_sym_inv_r A (x y:A) (e:x=y) : eq_trans e (eq_sym e) = eq_refl.

Theorem eq_trans_assoc A (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''.

Theorem rew_map A B (P:B→Type) (f:A→B) $x1$ $x2$ (H:$x1$=$x2$) (y:P (f $x1$)) :
rew [fun x ⇒ P (f x)] H in y = rew f_equal f H in y.

Theorem eq_trans_map {A B} {$x1$ $x2$ $x3$:A} {$y1$:B $x1$} {$y2$:B $x2$} {$y3$:B $x3$}
($H1$:$x1$=$x2$) ($H2$:$x2$=$x3$) ($H1$': rew $H1$ in $y1$ = $y2$) ($H2$': rew $H2$ in $y2$ = $y3$) :
rew eq_trans $H1$ $H2$ in $y1$ = $y3$.

Lemma map_subst {A} {P Q:A→Type} (f : ∀ x, P x → Q x) {x y} (H:x=y) (z:P x) :
rew H in f x z = f y (rew H in z).

Lemma map_subst_map {A B} {P:A→Type} {Q:B→Type} (f:A→B) (g : ∀ x, P x → Q (f x))
{x y} (H:x=y) (z:P x) :
rew f_equal f H in g x z = g y (rew H in z).

Lemma rew_swap A (P:A→Type) $x1$ $x2$ (H:$x1$=$x2$) ($y1$:P $x1$) ($y2$:P $x2$) : rew H in $y1$ = $y2$ → $y1$ = rew ← H in $y2$.

Lemma rew_compose A (P:A→Type) $x1$ $x2$ $x3$ ($H1$:$x1$=$x2$) ($H2$:$x2$=$x3$) (y:P $x1$) :
rew $H2$ in rew $H1$ in y = rew (eq_trans $H1$ $H2$) in y.

Extra properties of equality

Theorem eq_id_comm_l A (f:A→A) (Hf:∀ a, a = f a) a : f_equal f (Hf a) = Hf (f a).

Theorem eq_id_comm_r A (f:A→A) (Hf:∀ a, f a = a) a : f_equal f (Hf a) = Hf (f a).

Lemma eq_refl_map_distr A B x (f:A→B) : f_equal f (eq_refl x) = eq_refl (f x).

Lemma eq_trans_map_distr A B x y z (f:A→B) (e:x=y) (e':y=z) : f_equal f (eq_trans e e') = eq_trans (f_equal f e) (f_equal f e').

Lemma eq_sym_map_distr A B (x y:A) (f:A→B) (e:x=y) : eq_sym (f_equal f e) = f_equal f (eq_sym e).

Lemma eq_trans_sym_distr A (x y z:A) (e:x=y) (e':y=z) : eq_sym (eq_trans e e') = eq_trans (eq_sym e') (eq_sym e).

Lemma eq_trans_rew_distr A (P:A → Type) (x y z:A) (e:x=y) (e':y=z) (k:P x) :
rew (eq_trans e e') in k = rew e' in rew e in k.

```
Lemma rew_const A P (x y:A) (e:x=y) (k:P) :
    rew [fun _ ⇒ P] e in k = k.

Notation sym_eq := eq_sym (only parsing).
Notation trans_eq := eq_trans (only parsing).
Notation sym_not_eq := not_eq_sym (only parsing).

Notation refl_equal := eq_refl (only parsing).
Notation sym_equal := eq_sym (only parsing).
Notation trans_equal := eq_trans (only parsing).
Notation sym_not_equal := not_eq_sym (only parsing).

#[global]
Hint Immediate eq_sym not_eq_sym: core.
```

Basic definitions about relations and properties

```
Definition subrelation (A B : Type) (R R' : A→B→Prop) :=
  ∀ x y, R x y → R' x y.

Definition unique (A : Type) (P : A→Prop) (x:A) :=
  P x ∧ ∀ (x':A), P x' → x=x'.

Definition uniqueness (A:Type) (P:A→Prop) := ∀ x y, P x → P y → x = y.
```

Unique existence

```
Notation "'exists' ! x .. y , p" :=
  (ex (unique (fun x ⇒ .. (ex (unique (fun y ⇒ p))) ..)))
  (at level 200, x binder, right associativity,
   format "'[' 'exists' ! '/ ' x .. y , '/ ' p ']'")
  : type_scope.

Lemma unique_existence : ∀ (A:Type) (P:A→Prop),
  ((∃ x, P x) ∧ uniqueness P) ↔ (∃! x, P x).

Lemma forall_exists_unique_domain_coincide :
  ∀ A (P:A→Prop), (∃! x, P x) →
  ∀ Q:A→Prop, (∀ x, P x → Q x) ↔ (∃ x, P x ∧ Q x).

Lemma forall_exists_coincide_unique_domain :
  ∀ A (P:A→Prop),
  (∀ Q:A→Prop, (∀ x, P x → Q x) ↔ (∃ x, P x ∧ Q x))
  → (∃! x, P x).
```

8.4 Being inhabited

The predicate *inhabited* can be used in different contexts. If A is thought as a type, *inhabited* A states that A is inhabited. If A is thought as a computationally relevant proposition, then *inhabited* A weakens A so as to hide its computational meaning. The so-weakened proof remains computationally relevant but only in a propositional context.

```
Inductive inhabited (A:Type) : Prop := inhabits : A → inhabited A.

#[global]
```

```
Hint Resolve inhabits: core.

Lemma exists_inhabited : ∀ (A:Type) (P:A→Prop),
  (∃ x, P x) → inhabited A.

Lemma inhabited_covariant (A B : Type) : (A → B) → inhabited A → inhabited B.
```

Declaration of stepl and stepr for eq and iff

```
Lemma eq_stepl : ∀ (A : Type) (x y z : A), x = y → x = z → z = y.

Declare Left Step eq_stepl.
Declare Right Step eq_trans.

Lemma iff_stepl : ∀ A B C : Prop, (A ↔ B) → (A ↔ C) → (C ↔ B).

Declare Left Step iff_stepl.
Declare Right Step iff_trans.
```

Equality for *ex* Section ex.

```
  Local Unset Implicit Arguments.
  Definition eq_ex_uncurried {A : Type} (P : A → Prop) {u1 v1 : A} {u2 : P u1} {v2 : P v1}
             (pq : ∃ p : u1 = v1, rew p in u2 = v2)
  : ex_intro P u1 u2 = ex_intro P v1 v2.

  Definition eq_ex {A : Type} {P : A → Prop} (u1 v1 : A) (u2 : P u1) (v2 : P v1)
             (p : u1 = v1) (q : rew p in u2 = v2)
  : ex_intro P u1 u2 = ex_intro P v1 v2
    := eq_ex_uncurried P (ex_intro _ p q).

  Definition eq_ex_hprop {A} {P : A → Prop} (P_hprop : ∀ (x : A) (p q : P x), p = q)
             (u1 v1 : A) (u2 : P u1) (v2 : P v1)
             (p : u1 = v1)
    : ex_intro P u1 u2 = ex_intro P v1 v2
    := eq_ex u1 v1 u2 v2 p (P_hprop _ _ _).

  Lemma rew_ex {A x} {P : A → Type} (Q : ∀ a, P a → Prop) (u : ∃ p, Q x p) {y} (H : x = y)
  : rew [fun a ⇒ ∃ p, Q a p] H in u
    = match u with
        | ex_intro _ u1 u2
          ⇒ ex_intro
               (Q y)
               (rew H in u1)
               (rew dependent H in u2)
      end.
End ex.
```

Equality for *ex2* Section ex2.

```
  Local Unset Implicit Arguments.

  Definition eq_ex2_uncurried {A : Type} (P Q : A → Prop) {u1 v1 : A}
             {u2 : P u1} {v2 : P v1}
             {u3 : Q u1} {v3 : Q v1}
             (pq : exists2 p : u1 = v1, rew p in u2 = v2 & rew p in u3 = v3)
  : ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3.
```

```
Definition eq_ex2 {A : Type} {P Q : A → Prop}
           (u1 v1 : A)
           (u2 : P u1) (v2 : P v1)
           (u3 : Q u1) (v3 : Q v1)
           (p : u1 = v1) (q : rew p in u2 = v2) (r : rew p in u3 = v3)
  : ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3
    := eq_ex2_uncurried P Q (ex_intro2 _ _ p q r).
Definition eq_ex2_hprop {A} {P Q : A → Prop}
           (P_hprop : ∀ (x : A) (p q : P x), p = q)
           (Q_hprop : ∀ (x : A) (p q : Q x), p = q)
           (u1 v1 : A) (u2 : P u1) (v2 : P v1) (u3 : Q u1) (v3 : Q v1)
           (p : u1 = v1)
    : ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3
    := eq_ex2 u1 v1 u2 v2 u3 v3 p (P_hprop _ _ _) (Q_hprop _ _ _).
Lemma rew_ex2 {A x} {P : A → Type}
      (Q : ∀ a, P a → Prop)
      (R : ∀ a, P a → Prop)
      (u : exists2 p, Q x p & R x p) {y} (H : x = y)
  : rew [fun a ⇒ exists2 p, Q a p & R a p] H in u
    = match u with
        | ex_intro2 _ _ u1 u2 u3
          ⇒ ex_intro2
               (Q y)
               (R y)
               (rew H in u1)
               (rew dependent H in u2)
               (rew dependent H in u3)
      end.
End ex2.
```

Chapter 9

Library Coq.Init.Logic_Type

This module defines type constructors for types in `Type` (*Datatypes.v* and *Logic.v* defined them for types in `Set`)

```
Set Implicit Arguments.

Require Import Ltac.
Require Import Datatypes.
Require Export Logic.
```

Negation of a type in `Type`

```
Definition notT (A:Type) := A → False.
```

Properties of *identity*

```
Section identity_is_a_congruence.

 Variables A B : Type.
 Variable f : A → B.

 Variables x y z : A.

 Lemma identity_sym : identity x y → identity y x.

 Lemma identity_trans : identity x y → identity y z → identity x z.

 Lemma identity_congr : identity x y → identity (f x) (f y).

 Lemma not_identity_sym : notT (identity x y) → notT (identity y x).

End identity_is_a_congruence.

Definition identity_ind_r :
  ∀ (A:Type) (a:A) (P:A → Prop), P a → ∀ y:A, identity y a → P y.
Defined.

Definition identity_rec_r :
  ∀ (A:Type) (a:A) (P:A → Set), P a → ∀ y:A, identity y a → P y.
Defined.

Definition identity_rect_r :
  ∀ (A:Type) (a:A) (P:A → Type), P a → ∀ y:A, identity y a → P y.
Defined.
```

```
#[global]
Hint Immediate identity_sym not_identity_sym: core.

Notation refl_id := identity_refl (only parsing).
Notation sym_id := identity_sym (only parsing).
Notation trans_id := identity_trans (only parsing).
Notation sym_not_id := not_identity_sym (only parsing).
```

Chapter 10

Library Coq.Init.Peano

The type *nat* of Peano natural numbers (built from O and S) is defined in *Datatypes.v*

This module defines the following operations on natural numbers :

- predecessor *pred*
- addition *plus*
- multiplication *mult*
- less or equal order *le*
- less *lt*
- greater or equal *ge*
- greater *gt*

It states various lemmas and theorems about natural numbers, including Peano's axioms of arithmetic (in Coq, these are provable). Case analysis on *nat* and induction on *nat* $\times$ *nat* are provided too

```
Require Import Notations.
Require Import Ltac.
Require Import Datatypes.
Require Import Logic.
Require Coq.Init.Nat.
```

```
Open Scope nat_scope.
```

```
Definition eq_S := f_equal S.
Definition f_equal_nat := f_equal (A:=nat).
```

```
#[global]
Hint Resolve f_equal_nat: core.
```

The predecessor function

```
Notation pred := Nat.pred (only parsing).
```

```
Definition f_equal_pred := f_equal pred.

Theorem pred_Sn : ∀ n:nat, n = pred (S n).
```

Injectivity of successor

```
Definition eq_add_S n m (H: S n = S m): n = m := f_equal pred H.
#[global]
Hint Immediate eq_add_S: core.

Theorem not_eq_S : ∀ n m:nat, n ≠ m → S n ≠ S m.
#[global]
Hint Resolve not_eq_S: core.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | O ⇒ False
  | S p ⇒ True
  end.
```

Zero is not the successor of a number

```
Theorem O_S : ∀ n:nat, 0 ≠ S n.
#[global]
Hint Resolve O_S: core.

Theorem n_Sn : ∀ n:nat, n ≠ S n.
#[global]
Hint Resolve n_Sn: core.
```

Addition

```
Notation plus := Nat.add (only parsing).
Infix "+" := Nat.add : nat_scope.

Definition f_equal2_plus := f_equal2 plus.
Definition f_equal2_nat := f_equal2 (A1:=nat) (A2:=nat).
#[global]
Hint Resolve f_equal2_nat: core.

Lemma plus_n_O : ∀ n:nat, n = n + 0.

#[global]
Remove Hints eq_refl : core.
#[global]
Hint Resolve plus_n_O eq_refl: core.

Lemma plus_O_n : ∀ n:nat, 0 + n = n.

Lemma plus_n_Sm : ∀ n m:nat, S (n + m) = n + S m.
#[global]
Hint Resolve plus_n_Sm: core.

Lemma plus_Sn_m : ∀ n m:nat, S n + m = S (n + m).
```

Standard associated names

```
Notation plus_0_r_reverse := plus_n_O (only parsing).
```

Notation plus_succ_r_reverse := plus_n_Sm (*only parsing*).

Multiplication

Notation mult := Nat.mul (*only parsing*).
Infix "×" := Nat.mul : *nat_scope*.

Definition f_equal2_mult := f_equal2 mult.
#[*global*]
Hint Resolve *f_equal2_mult*: *core*.

Lemma mult_n_O : ∀ *n*:**nat**, 0 = *n* × 0.
#[*global*]
Hint Resolve *mult_n_O*: *core*.

Lemma mult_n_Sm : ∀ *n m*:**nat**, *n* × *m* + *n* = *n* × S *m*.
#[*global*]
Hint Resolve *mult_n_Sm*: *core*.

Standard associated names

Notation mult_0_r_reverse := mult_n_O (*only parsing*).
Notation mult_succ_r_reverse := mult_n_Sm (*only parsing*).

Truncated subtraction: *m*-*n* is 0 if *n*≥*m*

Notation minus := Nat.sub (*only parsing*).
Infix "-" := Nat.sub : *nat_scope*.

Definition of the usual orders, the basic properties of *le* and *lt* can be found in files Le and Lt

Inductive **le** (*n*:**nat**) : **nat** → Prop :=
  | le_n : *n* ≤ *n*
  | le_S : ∀ *m*:**nat**, *n* ≤ *m* → *n* ≤ S *m*

where "n <= m" := (**le** *n m*) : *nat_scope*.

#[*global*]
Hint Constructors **le**: *core*.

Definition lt (*n m*:**nat**) := S *n* ≤ *m*.
#[*global*]
Hint Unfold lt: *core*.

Infix "<" := lt : *nat_scope*.

Definition ge (*n m*:**nat**) := *m* ≤ *n*.
#[*global*]
Hint Unfold ge: *core*.

Infix "≥" := ge : *nat_scope*.

Definition gt (*n m*:**nat**) := *m* < *n*.
#[*global*]
Hint Unfold gt: *core*.

Infix ">" := gt : *nat_scope*.

```
Notation "x <= y <= z" := (x ≤ y ∧ y ≤ z) : nat_scope.
Notation "x <= y < z" := (x ≤ y ∧ y < z) : nat_scope.
Notation "x < y < z" := (x < y ∧ y < z) : nat_scope.
Notation "x < y <= z" := (x < y ∧ y ≤ z) : nat_scope.

Theorem le_pred : ∀ n m, n ≤ m → pred n ≤ pred m.

Theorem le_S_n : ∀ n m, S n ≤ S m → n ≤ m.

Theorem le_0_n : ∀ n, 0 ≤ n.

Theorem le_n_S : ∀ n m, n ≤ m → S n ≤ S m.
```

Case analysis

```
Theorem nat_case :
 ∀ (n:nat) (P:nat → Prop), P 0 → (∀ m:nat, P (S m)) → P n.
```

Principle of double induction

```
Theorem nat_double_ind :
 ∀ R:nat → nat → Prop,
   (∀ n:nat, R 0 n) →
   (∀ n:nat, R (S n) 0) →
   (∀ n m:nat, R n m → R (S n) (S m)) → ∀ n m:nat, R n m.
```

Maximum and minimum : definitions and specifications

```
Notation max := Nat.max (only parsing).
Notation min := Nat.min (only parsing).

Lemma max_l n m : m ≤ n → Nat.max n m = n.

Lemma max_r n m : n ≤ m → Nat.max n m = m.

Lemma min_l n m : n ≤ m → Nat.min n m = n.

Lemma min_r n m : m ≤ n → Nat.min n m = m.

Lemma nat_rect_succ_r {A} (f: A → A) (x:A) n :
  nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) (S n) = nat_rect (fun _ ⇒ A) (f x) (fun _ ⇒ f) n.

Theorem nat_rect_plus :
  ∀ (n m:nat) {A} (f:A → A) (x:A),
    nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) (n + m) =
      nat_rect (fun _ ⇒ A) (nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) m) (fun _ ⇒ f) n.
```

Chapter 11

Library Coq.Init.Notations

These are the notations whose level and associativity are imposed by Coq

Notations for propositional connectives

```
Reserved Notation "x -> y" (at level 99, right associativity, y at level 200).
Reserved Notation "x <-> y" (at level 95, no associativity).
Reserved Notation "x /\ y" (at level 80, right associativity).
Reserved Notation "x \/ y" (at level 85, right associativity).
Reserved Notation "~ x" (at level 75, right associativity).
```

Notations for equality and inequalities

```
Reserved Notation "x = y :> T"
(at level 70, y at next level, no associativity).
Reserved Notation "x = y" (at level 70, no associativity).
Reserved Notation "x = y = z"
(at level 70, no associativity, y at next level).

Reserved Notation "x <> y :> T"
(at level 70, y at next level, no associativity).
Reserved Notation "x <> y" (at level 70, no associativity).

Reserved Notation "x <= y" (at level 70, no associativity).
Reserved Notation "x < y" (at level 70, no associativity).
Reserved Notation "x >= y" (at level 70, no associativity).
Reserved Notation "x > y" (at level 70, no associativity).

Reserved Notation "x <= y <= z" (at level 70, y at next level).
Reserved Notation "x <= y < z" (at level 70, y at next level).
Reserved Notation "x < y < z" (at level 70, y at next level).
Reserved Notation "x < y <= z" (at level 70, y at next level).
```

Arithmetical notations (also used for type constructors)

```
Reserved Notation "x + y" (at level 50, left associativity).
Reserved Notation "x - y" (at level 50, left associativity).
Reserved Notation "x * y" (at level 40, left associativity).
Reserved Notation "x / y" (at level 40, left associativity).
Reserved Notation "- x" (at level 35, right associativity).
```

```
Reserved Notation "/ x" (at level 35, right associativity).
Reserved Notation "x ^ y" (at level 30, right associativity).
```

Notations for booleans

```
Reserved Notation "x || y" (at level 50, left associativity).
Reserved Notation "x && y" (at level 40, left associativity).
```

Notations for pairs

```
Reserved Notation "( x , y , .. , z )" (at level 0).
```

Notation "{ x }" is reserved and has a special status as component of other notations such as "{ A } + { B }" and "A + { B }" (which are at the same level as "x + y"); "{ x }" is at level 0 to factor with "{ x : A | P }"

```
Reserved Notation "{ x }" (at level 0, x at level 99).
```

Notations for sigma-types or subsets

```
Reserved Notation "{ A } + { B }" (at level 50, left associativity).
Reserved Notation "A + { B }" (at level 50, left associativity).

Reserved Notation "{ x | P }" (at level 0, x at level 99).
Reserved Notation "{ x | P & Q }" (at level 0, x at level 99).

Reserved Notation "{ x : A | P }" (at level 0, x at level 99).
Reserved Notation "{ x : A | P & Q }" (at level 0, x at level 99).

Reserved Notation "{ x & P }" (at level 0, x at level 99).
Reserved Notation "{ x & P & Q }" (at level 0, x at level 99).

Reserved Notation "{ x : A & P }" (at level 0, x at level 99).
Reserved Notation "{ x : A & P & Q }" (at level 0, x at level 99).

Reserved Notation "{ ' pat | P }"
  (at level 0, pat strict pattern, format "{ ' pat | P }").
Reserved Notation "{ ' pat | P & Q }"
  (at level 0, pat strict pattern, format "{ ' pat | P & Q }").

Reserved Notation "{ ' pat : A | P }"
  (at level 0, pat strict pattern, format "{ ' pat : A | P }").
Reserved Notation "{ ' pat : A | P & Q }"
  (at level 0, pat strict pattern, format "{ ' pat : A | P & Q }").

Reserved Notation "{ ' pat & P }"
  (at level 0, pat strict pattern, format "{ ' pat & P }").
Reserved Notation "{ ' pat & P & Q }"
  (at level 0, pat strict pattern, format "{ ' pat & P & Q }").

Reserved Notation "{ ' pat : A & P }"
  (at level 0, pat strict pattern, format "{ ' pat : A & P }").
Reserved Notation "{ ' pat : A & P & Q }"
  (at level 0, pat strict pattern, format "{ ' pat : A & P & Q }").
```

Support for Gonthier-Ssreflect's "if c is pat then u else v"

```
Module IfNotations.
```

```
Notation "'if' c 'is' p 'then' u 'else' v" :=
  (match c with p ⇒ u | _ ⇒ v end)
  (at level 200, p pattern at level 100).

End IfNotations.
```

Scopes

```
Delimit Scope core_scope with core.

Delimit Scope function_scope with function.

Delimit Scope type_scope with type.

Open Scope core_scope.
Open Scope function_scope.
Open Scope type_scope.
```

Chapter 12

Library Coq.Init.Numeral

12.1 Deprecated: use Number.v instead

```
Require Import Decimal Hexadecimal Number.
#[deprecated(since="8.13",note="Use Number.uint instead.")]
Notation uint := uint (only parsing).
#[deprecated(since="8.13",note="Use Number.UintDecimal instead.")]
Notation UIntDec := UIntDecimal (only parsing).
#[deprecated(since="8.13",note="Use Number.UintHexadecimal instead.")]
Notation UIntHex := UIntHexadecimal (only parsing).

#[deprecated(since="8.13",note="Use Number.int instead.")]
Notation int := int (only parsing).
#[deprecated(since="8.13",note="Use Number.IntDecimal instead.")]
Notation IntDec := IntDecimal (only parsing).
#[deprecated(since="8.13",note="Use Number.IntHexadecimal instead.")]
Notation IntHex := IntHexadecimal (only parsing).

#[deprecated(since="8.13",note="Use Number.numeral instead.")]
Notation numeral := number (only parsing).
#[deprecated(since="8.13",note="Use Number.Decimal instead.")]
Notation Dec := Decimal (only parsing).
#[deprecated(since="8.13",note="Use Number.Hexadecimal instead.")]
Notation Hex := Hexadecimal (only parsing).

#[deprecated(since="8.13",note="Use Number.uint_beq instead.")]
Notation uint_beq := uint_beq (only parsing).
#[deprecated(since="8.13",note="Use Number.uint_eq_dec instead.")]
Notation uint_eq_dec := uint_eq_dec (only parsing).
#[deprecated(since="8.13",note="Use Number.int_beq instead.")]
Notation int_beq := int_beq (only parsing).
#[deprecated(since="8.13",note="Use Number.int_eq_dec instead.")]
Notation int_eq_dec := int_eq_dec (only parsing).
#[deprecated(since="8.13",note="Use Number.numeral_beq instead.")]
Notation numeral_beq := number_beq (only parsing).
```

#[*deprecated*(*since*="8.13",*note*="Use Number.numeral_eq_dec instead.")]
`Notation` numeral_eq_dec := number_eq_dec (*only parsing*).

#[*deprecated*(*since*="8.13",*note*="Use Number.uint_of_uint instead.")]
`Notation` uint_of_uint := uint_of_uint (*only parsing*).
#[*deprecated*(*since*="8.13",*note*="Use Number.int_of_int instead.")]
`Notation` int_of_int := int_of_int (*only parsing*).

Chapter 13

Library Coq.Init.Tauto

13.1 The tauto and intuition tactics

```
Require Import Notations.
Require Import Ltac.
Require Import Datatypes.
Require Import Logic.

Local Ltac not_dep_intros :=
  repeat match goal with
  | ⊢ (∀ (_: ?X1), ?X2) ⇒ intro
  | ⊢ (Coq.Init.Logic.not _) ⇒ unfold Coq.Init.Logic.not at 1; intro
  end.

Local Ltac axioms flags :=
  match reverse goal with
    | ⊢ ?X1 ⇒ is_unit_or_eq flags X1; constructor 1
    | _:?X1 ⊢ _ ⇒ is_empty flags X1; elimtype X1; assumption
    | _:?X1 ⊢ ?X1 ⇒ assumption
  end.

Local Ltac simplif flags :=
  not_dep_intros;
  repeat
     (match reverse goal with
      | id: ?X1 ⊢ _ ⇒ is_conj flags X1; elim id; do 2 intro; clear id
      | id: (Coq.Init.Logic.iff _ _) ⊢ _ ⇒ elim id; do 2 intro; clear id
      | id: (Coq.Init.Logic.not _) ⊢ _ ⇒ red in id
      | id: ?X1 ⊢ _ ⇒ is_disj flags X1; elim id; intro; clear id
      | id0: (∀ (_: ?X1), ?X2), id1: ?X1⊢ _ ⇒

    assert X2; [exact (id0 id1) | clear id0]
      | id: ∀ (_ : ?X1), ?X2⊢ _ ⇒
         is_unit_or_eq flags X1; cut X2;
    [ intro; clear id
```

```
      |
        cut X1; [exact id| constructor 1; fail]
      ]
        | id: ∀ (_ : ?X1), ?X2⊢ _ ⇒
          flatten_contravariant_conj flags X1 X2 id

        | id: ∀ (_: Coq.Init.Logic.iff ?X1 ?X2), ?X3⊢ _ ⇒
          assert (∀ (_: ∀ _:X1, X2), ∀ (_: ∀ _: X2, X1), X3)
      by (do 2 intro; apply id; split; assumption);
            clear id
        | id: ∀ (_:?X1), ?X2⊢ _ ⇒
          flatten_contravariant_disj flags X1 X2 id

        | ⊢ ?X1 ⇒ is_conj flags X1; split
        | ⊢ (Coq.Init.Logic.iff _ _) ⇒ split
        | ⊢ (Coq.Init.Logic.not _) ⇒ red
        end;
        not_dep_intros).
Local Ltac tauto_intuit flags t_reduce t_solver :=
 let rec t_tauto_intuit :=
 (simplif flags; axioms flags
 || match reverse goal with
      | id:∀(_: ∀ (_: ?X1), ?X2), ?X3⊢ _ ⇒
   cut X3;
      [ intro; clear id; t_tauto_intuit
      | cut (∀ (_: X1), X2);
          [ exact id
          | generalize (fun y:X2 ⇒ id (fun x:X1 ⇒ y)); intro; clear id;
            solve [ t_tauto_intuit ]]]
    | id:∀ (_:not ?X1), ?X3⊢ _ ⇒
   cut X3;
      [ intro; clear id; t_tauto_intuit
      | cut (not X1); [ exact id | clear id; intro; solve [t_tauto_intuit ]]]
    | ⊢ ?X1 ⇒
          is_disj flags X1; solve [left;t_tauto_intuit | right;t_tauto_intuit]
      end
   ||

   match goal with | ⊢ ∀ (_ : _), _ ⇒ intro; t_tauto_intuit
   | ⊢ _ ⇒ t_reduce;t_solver
   end
   ||
   t_solver
 ) in t_tauto_intuit.
Local Ltac intuition_gen flags solver := tauto_intuit flags reduction_not_iff solver.
```

```
Local Ltac tauto_intuitionistic flags := intuition_gen flags fail || fail "tauto failed".
Local Ltac tauto_classical flags :=
  (apply_nnpp || fail "tauto failed"); (tauto_intuitionistic flags || fail "Classical tauto failed").
Local Ltac tauto_gen flags := tauto_intuitionistic flags || tauto_classical flags.

Ltac tauto := with_uniform_flags ltac:(fun flags ⇒ tauto_gen flags).
Ltac dtauto := with_power_flags ltac:(fun flags ⇒ tauto_gen flags).

Ltac intuition := with_uniform_flags ltac:(fun flags ⇒ intuition_gen flags ltac:(auto with *)).
Local Ltac intuition_then tac := with_uniform_flags ltac:(fun flags ⇒ intuition_gen flags tac).

Ltac dintuition := with_power_flags ltac:(fun flags ⇒ intuition_gen flags ltac:(auto with *)).
Local Ltac dintuition_then tac := with_power_flags ltac:(fun flags ⇒ intuition_gen flags tac).

Tactic Notation "intuition" := intuition.
Tactic Notation "intuition" tactic(t) := intuition_then t.

Tactic Notation "dintuition" := dintuition.
Tactic Notation "dintuition" tactic(t) := dintuition_then t.
```

Chapter 14

Library Coq.Init.Specif

Basic specifications : sets that may contain logical information

```
Set Implicit Arguments.

Require Import Notations.
Require Import Ltac.
Require Import Datatypes.
Require Import Logic.
```

Subsets and Sigma-types

(*sig A P*), or more suggestively {*x*:*A* | *P x*}, denotes the subset of elements of the type *A* which satisfy the predicate *P*. Similarly (*sig2 A P Q*), or {*x*:*A* | *P x* & *Q x*}, denotes the subset of elements of the type *A* which satisfy both *P* and *Q*.

```
#[universes(template)]
Inductive sig (A:Type) (P:A → Prop) : Type :=
    exist : ∀ x:A, P x → sig P.

#[universes(template)]
Inductive sig2 (A:Type) (P Q:A → Prop) : Type :=
    exist2 : ∀ x:A, P x → Q x → sig2 P Q.
```

(*sigT A P*), or more suggestively {*x*:*A* & (*P x*)} is a Sigma-type. Similarly for (*sigT2 A P Q*), also written {*x*:*A* & (*P x*) & (*Q x*)}.

```
#[universes(template)]
Inductive sigT (A:Type) (P:A → Type) : Type :=
    existT : ∀ x:A, P x → sigT P.

#[universes(template)]
Inductive sigT2 (A:Type) (P Q:A → Type) : Type :=
    existT2 : ∀ x:A, P x → Q x → sigT2 P Q.

Notation "{ x | P }" := (sig (fun x ⇒ P)) : type_scope.
Notation "{ x | P & Q }" := (sig2 (fun x ⇒ P) (fun x ⇒ Q)) : type_scope.
Notation "{ x : A | P }" := (sig (A:=A) (fun x ⇒ P)) : type_scope.
Notation "{ x : A | P & Q }" := (sig2 (A:=A) (fun x ⇒ P) (fun x ⇒ Q)) :
```

type_scope.

```
Notation "{ x & P }" := (sigT (fun x ⇒ P)) : type_scope.
Notation "{ x & P & Q }" := (sigT2 (fun x ⇒ P) (fun x ⇒ Q)) : type_scope.
Notation "{ x : A & P }" := (sigT (A:=A) (fun x ⇒ P)) : type_scope.
Notation "{ x : A & P & Q }" := (sigT2 (A:=A) (fun x ⇒ P) (fun x ⇒ Q)) :
  type_scope.

Notation "{ ' pat | P }" := (sig (fun pat ⇒ P)) : type_scope.
Notation "{ ' pat | P & Q }" := (sig2 (fun pat ⇒ P) (fun pat ⇒ Q)) : type_scope.
Notation "{ ' pat : A | P }" := (sig (A:=A) (fun pat ⇒ P)) : type_scope.
Notation "{ ' pat : A | P & Q }" := (sig2 (A:=A) (fun pat ⇒ P) (fun pat ⇒ Q)) :
  type_scope.
Notation "{ ' pat & P }" := (sigT (fun pat ⇒ P)) : type_scope.
Notation "{ ' pat & P & Q }" := (sigT2 (fun pat ⇒ P) (fun pat ⇒ Q)) : type_scope.
Notation "{ ' pat : A & P }" := (sigT (A:=A) (fun pat ⇒ P)) : type_scope.
Notation "{ ' pat : A & P & Q }" := (sigT2 (A:=A) (fun pat ⇒ P) (fun pat ⇒ Q)) :
  type_scope.

Add Printing Let sig.
Add Printing Let sig2.
Add Printing Let sigT.
Add Printing Let sigT2.
```

Projections of *sig*

An element y of a subset $\{x{:}A \mid (P\ x)\}$ is the pair of an a of type A and of a proof h that a satisfies P. Then (*proj1_sig* y) is the witness a and (*proj2_sig* y) is the proof of $(P\ a)$

```
Section Subset_projections.

  Variable A : Type.
  Variable P : A → Prop.

  Definition proj1_sig (e:sig P) := match e with
                                    | exist _ a b ⇒ a
                                    end.

  Definition proj2_sig (e:sig P) :=
    match e return P (proj1_sig e) with
    | exist _ a b ⇒ b
    end.

End Subset_projections.
```

sig2 of a predicate can be projected to a *sig*.

This allows *proj1_sig* and *proj2_sig* to be usable with *sig2*.

The `let` statements occur in the body of the *exist* so that *proj1_sig* of a coerced X : *sig2* P Q will unify with `let` (a, _, _) := X `in` a

```
Definition sig_of_sig2 (A : Type) (P Q : A → Prop) (X : sig2 P Q) : sig P
  := exist P
          (let (a, _, _) := X in a)
          (let (x, p, _) as s return (P (let (a, _, _) := s in a)) := X in p).
```

Projections of *sig2*

An element y of a subset $\{x{:}A \mid (P\ x)\ \&\ (Q\ x)\}$ is the triple of an a of type A, a of a proof h that a satisfies P, and a proof h' that a satisfies Q. Then (*proj1_sig* (*sig_of_sig2* y)) is the witness a, (*proj2_sig* (*sig_of_sig2* y)) is the proof of ($P\ a$), and (*proj3_sig* y) is the proof of ($Q\ a$).

```
Section Subset_projections2.

  Variable A : Type.
  Variables P Q : A → Prop.

  Definition proj3_sig (e : sig2 P Q) :=
    let (a, b, c) return Q (proj1_sig (sig_of_sig2 e)) := e in c.

End Subset_projections2.
```

Projections of *sigT*

An element x of a sigma-type $\{y{:}A\ \&\ P\ y\}$ is a dependent pair made of an a of type A and an h of type $P\ a$. Then, (*projT1* x) is the first projection and (*projT2* x) is the second projection, the type of which depends on the *projT1*.

```
Section Projections.

  Variable A : Type.
  Variable P : A → Type.

  Definition projT1 (x:sigT P) : A := match x with
                                      | existT _ a _ ⇒ a
                                      end.

  Definition projT2 (x:sigT P) : P (projT1 x) :=
    match x return P (projT1 x) with
    | existT _ _ h ⇒ h
    end.

End Projections.
```

sigT2 of a predicate can be projected to a *sigT*.

This allows *projT1* and *projT2* to be usable with *sigT2*.

The `let` statements occur in the body of the *existT* so that *projT1* of a coerced X : *sigT2* $P\ Q$ will unify with `let` $(a, \_, \_) := X$ `in` a

```
Definition sigT_of_sigT2 (A : Type) (P Q : A → Type) (X : sigT2 P Q) : sigT P
  := existT P
            (let (a, _, _) := X in a)
            (let (x, p, _) as s return (P (let (a, _, _) := s in a)) := X in p).
```

Projections of *sigT2*

An element x of a sigma-type $\{y{:}A\ \&\ P\ y\ \&\ Q\ y\}$ is a dependent pair made of an a of type A, an h of type $P\ a$, and an h' of type $Q\ a$. Then, (*projT1* (*sigT_of_sigT2* x)) is the first projection, (*projT2* (*sigT_of_sigT2* x)) is the second projection, and (*projT3* x) is the third projection, the types of which depends on the *projT1*.

```
Section Projections2.

  Variable A : Type.
```

Variables P Q : $A \to$ Type.

Definition projT3 (e : **sigT2** P Q) :=
  let (a, b, c) return Q (projT1 (sigT_of_sigT2 e)) := e in c.

End Projections2.

sigT of a predicate is equivalent to *sig*

Definition sig_of_sigT (A : Type) (P : $A \to$ Prop) (X : **sigT** P) : **sig** P
  := exist P (projT1 X) (projT2 X).

Definition sigT_of_sig (A : Type) (P : $A \to$ Prop) (X : **sig** P) : **sigT** P
  := existT P (proj1_sig X) (proj2_sig X).

sigT2 of a predicate is equivalent to *sig2*

Definition sig2_of_sigT2 (A : Type) (P Q : $A \to$ Prop) (X : **sigT2** P Q) : **sig2** P Q
  := exist2 P Q (projT1 (sigT_of_sigT2 X)) (projT2 (sigT_of_sigT2 X)) (projT3 X).

Definition sigT2_of_sig2 (A : Type) (P Q : $A \to$ Prop) (X : **sig2** P Q) : **sigT2** P Q
  := existT2 P Q (proj1_sig (sig_of_sig2 X)) (proj2_sig (sig_of_sig2 X)) (proj3_sig X).

η Principles Definition sigT_eta {A P} (p : { a : A & P a })
  : p = existT _ (projT1 p) (projT2 p).

Definition sig_eta {A P} (p : { a : A | P a })
  : p = exist _ (proj1_sig p) (proj2_sig p).

Definition sigT2_eta {A P Q} (p : { a : A & P a & Q a })
  : p = existT2 _ _ (projT1 (sigT_of_sigT2 p)) (projT2 (sigT_of_sigT2 p)) (projT3 p).

Definition sig2_eta {A P Q} (p : { a : A | P a & Q a })
  : p = exist2 _ _ (proj1_sig (sig_of_sig2 p)) (proj2_sig (sig_of_sig2 p)) (proj3_sig p).

$\exists\ x : A, B$ is equivalent to *inhabited* $\{x : A \mid B\}$ Lemma exists_to_inhabited_sig {A P} : ($\exists$ x : A, P x) $\to$ **inhabited** {x : A | P x}.

Lemma inhabited_sig_to_exists {A P} : **inhabited** {x : A | P x} $\to$ $\exists$ x : A, P x.

Equality of sigma types

Import *EqNotations*.

Equality for *sigT* Section sigT.
  Local Unset Implicit Arguments.
Projecting an equality of a pair to equality of the first components Definition projT1_eq {A} {P : $A \to$ Type} {u v : { a : A & P a }} (p : u = v)
  : u.1 = v.1
  := f_equal (fun $x \Rightarrow x$.1) p.

Projecting an equality of a pair to equality of the second components Definition projT2_eq {A} {P : $A \to$ Type} {u v : { a : A & P a }} (p : u = v)
  : rew projT1_eq p in u.2 = v.2
  := rew dependent p in eq_refl.

Equality of *sigT* is itself a *sigT* (forwards-reasoning version) Definition eq_existT_uncurried {A : Type} {P : $A \to$ Type} {$u1$ $v1$: A} {$u2$: P $u1$} {$v2$: P $v1$}
  (pq : { p : $u1$ = $v1$ & rew p in $u2$ = $v2$ })

: (*u1* ; *u2*) = (*v1* ; *v2*).

Equality of *sigT* is itself a *sigT* (backwards-reasoning version) `Definition` eq_sigT_uncurried {*A* : `Type`} {*P* : *A* → `Type`} (*u v* : { *a* : *A* & *P a* })
(*pq* : { *p* : *u*.1 = *v*.1 & `rew` *p* `in` *u*.2 = *v*.2 })
: *u* = *v*.

`Lemma` eq_existT_curried {*A* : `Type`} {*P* : *A* → `Type`} {*u1 v1* : *A*} {*u2* : *P u1*} {*v2* : *P v1*}
(*p* : *u1* = *v1*) (*q* : `rew` *p* `in` *u2* = *v2*) : (*u1* ; *u2*) = (*v1* ; *v2*).

`Lemma` eq_existT_curried_map {*A A' P P'*} (*f*:*A* → *A'*) (*g*:∀ *u*:*A*, *P u* → *P'* (*f u*))
{*u1 v1* : *A*} {*u2* : *P u1*} {*v2* : *P v1*} (*p* : *u1* = *v1*) (*q* : `rew` *p* `in` *u2* = *v2*) :
f_equal (fun *x* ⇒ (*f x*.1; *g x*.1 *x*.2)) (= *p*; *q*) =
(= f_equal *f p*; f_equal_dep2 *f g p q*).

`Lemma` eq_existT_curried_trans {*A P*} {*u1 v1 w1* : *A*} {*u2* : *P u1*} {*v2* : *P v1*} {*w2* : *P w1*}
(*p* : *u1* = *v1*) (*q* : `rew` *p* `in` *u2* = *v2*)
(*p'* : *v1* = *w1*) (*q'*: `rew` *p'* `in` *v2* = *w2*) :
eq_trans (= *p*; *q*) (= *p'*; *q'*) =
(= eq_trans *p p'*; eq_trans_map *p p' q q'*).

`Theorem` eq_existT_curried_congr {*A P*} {*u1 v1* : *A*} {*u2* : *P u1*} {*v2* : *P v1*}
{*p p'* : *u1* = *v1*} {*q* : `rew` *p* `in` *u2* = *v2*} {*q'*: `rew` *p'* `in` *u2* = *v2*}
(*r* : *p* = *p'*) : `rew [fun` *H* ⇒ `rew` *H* `in` *u2* = *v2*`]` *r* `in` *q* = *q'* → (= *p*; *q*) = (= *p'*; *q'*).

Curried version of proving equality of sigma types `Definition` eq_sigT {*A* : `Type`} {*P* : *A* → `Type`} (*u v* : { *a* : *A* & *P a* })
(*p* : *u*.1 = *v*.1) (*q* : `rew` *p* `in` *u*.2 = *v*.2)
: *u* = *v*
:= eq_sigT_uncurried *u v* (existT _ *p q*).

Equality of *sigT* when the property is an hProp `Definition` eq_sigT_hprop {*A P*} (*P_hprop* : ∀ (*x* : *A*) (*p q* : *P x*), *p* = *q*)
(*u v* : { *a* : *A* & *P a* })
(*p* : *u*.1 = *v*.1)
: *u* = *v*
:= eq_sigT *u v p* (*P_hprop* _ _ _).

Equivalence of equality of *sigT* with a *sigT* of equality We could actually prove an isomorphism here, and not just ↔, but for simplicity, we don't. `Definition` eq_sigT_uncurried_iff {*A P*}
(*u v* : { *a* : *A* & *P a* })
: *u* = *v* ↔ { *p* : *u*.1 = *v*.1 & `rew` *p* `in` *u*.2 = *v*.2 }.

Induction principle for @*eq* (*sigT* _) `Definition` eq_sigT_rect {*A P*} {*u v* : { *a* : *A* & *P a* }} (*Q* : *u* = *v* → `Type`)
(*f* : ∀ *p q*, *Q* (eq_sigT *u v p q*))
: ∀ *p*, *Q p*.

`Definition` eq_sigT_rec {*A P u v*} (*Q* : *u* = *v* :> { *a* : *A* & *P a* } → `Set`) := eq_sigT_rect *Q*.

`Definition` eq_sigT_ind {*A P u v*} (*Q* : *u* = *v* :> { *a* : *A* & *P a* } → `Prop`) := eq_sigT_rec *Q*.

Equivalence of equality of *sigT* involving hProps with equality of the first components `Definition` eq_sigT_hprop_iff {*A P*} (*P_hprop* : ∀ (*x* : *A*) (*p q* : *P x*), *p* = *q*)

$(u\ v : \{\ a : A\ \&\ P\ a\ \})$
: $u = v \leftrightarrow (u.1 = v.1)$
:= conj (fun $p \Rightarrow$ f_equal (@projT1 _ _) p) (eq_sigT_hprop P_hprop u v).

Non-dependent classification of equality of *sigT* Definition eq_sigT_nondep $\{A\ B$: Type$\}$ $(u\ v : \{\ a : A\ \&\ B\ \})$
$(p : u.1 = v.1)$ $(q : u.2 = v.2)$
: $u = v$
:= @eq_sigT _ _ u v p (eq_trans (rew_const _ _) q).

Classification of transporting across an equality of *sigT*s Lemma rew_sigT $\{A\ x\}$ $\{P : A \to$ Type$\}$ $(Q : \forall\ a,\ P\ a \to$ Prop$)$ $(u : \{\ p : P\ x\ \&\ Q\ x\ p\ \})$ $\{y\}$ $(H : x = y)$
: rew [fun $a \Rightarrow \{\ p : P\ a\ \&\ Q\ a\ p\ \}$] H in u
= existT
$(Q\ y)$
(rew H in $u.1$)
(rew dependent H in $(u.2)$).
End sigT.

Equality for *sig* Section sig.
Local Unset Implicit Arguments.
Projecting an equality of a pair to equality of the first components Definition proj1_sig_eq $\{A\}$ $\{P : A \to$ Prop$\}$ $\{u\ v : \{\ a : A\ |\ P\ a\ \}\}$ $(p : u = v)$
: proj1_sig u = proj1_sig v
:= f_equal (@proj1_sig _ _) p.

Projecting an equality of a pair to equality of the second components Definition proj2_sig_eq $\{A\}$ $\{P : A \to$ Prop$\}$ $\{u\ v : \{\ a : A\ |\ P\ a\ \}\}$ $(p : u = v)$
: rew proj1_sig_eq p in proj2_sig u = proj2_sig v
:= rew dependent p in eq_refl.

Equality of *sig* is itself a *sig* (forwards-reasoning version) Definition eq_exist_uncurried $\{A$: Type$\}$ $\{P : A \to$ Prop$\}$ $\{u1\ v1 : A\}$ $\{u2 : P\ u1\}$ $\{v2 : P\ v1\}$
$(pq : \{\ p : u1 = v1\ |$ rew p in $u2 = v2\ \})$
: exist _ $u1$ $u2$ = exist _ $v1$ $v2$.

Equality of *sig* is itself a *sig* (backwards-reasoning version) Definition eq_sig_uncurried $\{A$: Type$\}$ $\{P : A \to$ Prop$\}$ $(u\ v : \{\ a : A\ |\ P\ a\ \})$
$(pq : \{\ p :$ proj1_sig u = proj1_sig v | rew p in proj2_sig u = proj2_sig v $\})$
: $u = v$.

Curried version of proving equality of sigma types Definition eq_sig $\{A$: Type$\}$ $\{P : A \to$ Prop$\}$ $(u\ v : \{\ a : A\ |\ P\ a\ \})$
$(p :$ proj1_sig u = proj1_sig $v)$ $(q :$ rew p in proj2_sig u = proj2_sig $v)$
: $u = v$
:= eq_sig_uncurried u v (exist _ p q).

Induction principle for @*eq* (*sig* _) Definition eq_sig_rect $\{A\ P\}$ $\{u\ v : \{\ a : A\ |\ P\ a\ \}\}$ $(Q : u = v \to$ Type$)$
$(f : \forall\ p\ q,\ Q$ (eq_sig u v p q)$)$
: $\forall\ p,\ Q\ p$.

Definition eq_sig_rec $\{A\ P\ u\ v\}$ $(Q : u = v$:> { a : A | $P\ a$ } $\rightarrow$ Set) := eq_sig_rect Q.
Definition eq_sig_ind $\{A\ P\ u\ v\}$ $(Q : u = v$:> { a : A | $P\ a$ } $\rightarrow$ Prop) := eq_sig_rec Q.

Equality of *sig* when the property is an hProp Definition eq_sig_hprop $\{A\}$ $\{P : A \rightarrow$ Prop$\}$ (P_hprop : $\forall$ $(x : A)$ $(p\ q : P\ x)$, $p = q$)
($u\ v$: { a : A | $P\ a$ })
(p : proj1_sig u = proj1_sig v)
: $u = v$
:= eq_sig $u\ v\ p$ (P_hprop _ _ _).

Equivalence of equality of *sig* with a *sig* of equality We could actually prove an isomorphism here, and not just $\leftrightarrow$, but for simplicity, we don't. Definition eq_sig_uncurried_iff $\{A\}$ $\{P : A \rightarrow$ Prop$\}$
($u\ v$: { a : A | $P\ a$ })
: $u = v \leftrightarrow$ { p : proj1_sig u = proj1_sig v | rew p in proj2_sig u = proj2_sig v }.

Equivalence of equality of *sig* involving hProps with equality of the first components Definition eq_sig_hprop_iff $\{A\}$ $\{P : A \rightarrow$ Prop$\}$ (P_hprop : $\forall$ $(x : A)$ $(p\ q : P\ x)$, $p = q$)
($u\ v$: { a : A | $P\ a$ })
: $u = v \leftrightarrow$ (proj1_sig u = proj1_sig v)
:= conj (fun $p \Rightarrow$ f_equal (@proj1_sig _ _) p) (eq_sig_hprop P_hprop u v).

Lemma rew_sig $\{A\ x\}$ $\{P : A \rightarrow$ Type$\}$ $(Q : \forall\ a,\ P\ a \rightarrow$ Prop$)$ (u : { p : $P\ x$ | $Q\ x\ p$ }) $\{y\}$ (H : $x = y$)
: rew [fun $a \Rightarrow$ { p : $P\ a$ | $Q\ a\ p$ }] H in u
= exist
($Q\ y$)
(rew H in proj1_sig u)
(rew dependent H in proj2_sig u).
End sig.

Equality for *sigT* Section sigT2.
Local Unset Implicit Arguments.
Projecting an equality of a pair to equality of the first components Definition sigT_of_sigT2_eq $\{A\}$ $\{P\ Q : A \rightarrow$ Type$\}$ $\{u\ v$: { a : A & $P\ a$ & $Q\ a$ }$\}$ (p : $u = v$)
: $u = v$:> { a : A & $P\ a$ }
:= f_equal _ p.
Definition projT1_of_sigT2_eq $\{A\}$ $\{P\ Q : A \rightarrow$ Type$\}$ $\{u\ v$: { a : A & $P\ a$ & $Q\ a$ }$\}$ (p : u = v)
: u.1 = v.1
:= projT1_eq (sigT_of_sigT2_eq p).

Projecting an equality of a pair to equality of the second components Definition projT2_of_sigT2_eq $\{A\}$ $\{P\ Q : A \rightarrow$ Type$\}$ $\{u\ v$: { a : A & $P\ a$ & $Q\ a$ }$\}$ (p : $u = v$)
: rew projT1_of_sigT2_eq p in u.2 = v.2
:= rew dependent p in eq_refl.

Projecting an equality of a pair to equality of the third components Definition projT3_eq $\{A\}$ $\{P\ Q : A \rightarrow$ Type$\}$ $\{u\ v$: { a : A & $P\ a$ & $Q\ a$ }$\}$ (p : $u = v$)
: rew projT1_of_sigT2_eq p in projT3 u = projT3 v

$:=$ `rew dependent` p `in` eq_refl.

Equality of *sigT2* is itself a *sigT2* (forwards-reasoning version) `Definition` eq_existT2_uncurried $\{A : \texttt{Type}\}$ $\{P\ Q : A \rightarrow \texttt{Type}\}$

$\{u1\ v1 : A\}$ $\{u2 : P\ u1\}$ $\{v2 : P\ v1\}$ $\{u3 : Q\ u1\}$ $\{v3 : Q\ v1\}$
$(pqr$: `{` p : $u1$ = $v1$
& `rew` p `in` $u2$ = $v2$ & `rew` p `in` $u3$ = $v3$ `}`)
: existT2 _ _ $u1\ u2\ u3$ = existT2 _ _ $v1\ v2\ v3$.

Equality of *sigT2* is itself a *sigT2* (backwards-reasoning version) `Definition` eq_sigT2_uncurried $\{A : \texttt{Type}\}$ $\{P\ Q : A \rightarrow \texttt{Type}\}$ $(u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`)

$(pqr$: `{` p : u.1 = v.1
& `rew` p `in` u.2 = v.2 & `rew` p `in` projT3 u = projT3 v `}`)
: u = v.

Curried version of proving equality of sigma types `Definition` eq_sigT2 $\{A : \texttt{Type}\}$ $\{P\ Q : A \rightarrow \texttt{Type}\}$ $(u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`)

$(p$: u.1 = v.1)
$(q$: `rew` p `in` u.2 = v.2)
$(r$: `rew` p `in` projT3 u = projT3 v)
: u = v
$:=$ eq_sigT2_uncurried $u\ v$ (existT2 _ _ $p\ q\ r$).

Equality of *sigT2* when the second property is an hProp `Definition` eq_sigT2_hprop $\{A\ P\ Q\}$ $(Q\_hprop : \forall\ (x : A)\ (p\ q : Q\ x),\ p = q)$

$(u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`)
$(p$: u = v :> `{` a : A & $P\ a$ `}`)
: u = v
$:=$ eq_sigT2 $u\ v$ (projT1_eq p) (projT2_eq p) $(Q\_hprop$ _ _ _).

Equivalence of equality of *sigT2* with a *sigT2* of equality We could actually prove an isomorphism here, and not just $\leftrightarrow$, but for simplicity, we don't. `Definition` eq_sigT2_uncurried_iff $\{A\ P\ Q\}$

$(u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`)
: u = v
$\leftrightarrow$ `{` p : u.1 = v.1
& `rew` p `in` u.2 = v.2 & `rew` p `in` projT3 u = projT3 v `}`.

Induction principle for @*eq* (*sigT2* _ _) `Definition` eq_sigT2_rect $\{A\ P\ Q\}$ $\{u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`$\}$ $(R : u = v \rightarrow \texttt{Type})$

$(f : \forall\ p\ q\ r,\ R$ (eq_sigT2 $u\ v\ p\ q\ r$))
: $\forall\ p,\ R\ p$.

`Definition` eq_sigT2_rec $\{A\ P\ Q\ u\ v\}$ $(R : u = v$:> `{` a : A & $P\ a$ & $Q\ a$ `}` $\rightarrow$ `Set`) $:=$ eq_sigT2_rect R.

`Definition` eq_sigT2_ind $\{A\ P\ Q\ u\ v\}$ $(R : u = v$:> `{` a : A & $P\ a$ & $Q\ a$ `}` $\rightarrow$ `Prop`) $:=$ eq_sigT2_rec R.

Equivalence of equality of *sigT2* involving hProps with equality of the first components `Definition` eq_sigT2_hprop_iff $\{A\ P\ Q\}$ $(Q\_hprop : \forall\ (x : A)\ (p\ q : Q\ x),\ p = q)$

$(u\ v$: `{` a : A & $P\ a$ & $Q\ a$ `}`)
: u = v $\leftrightarrow$ (u = v :> `{` a : A & $P\ a$ `}`)

:= conj (fun $p \Rightarrow$ f_equal (@sigT_of_sigT2 _ _ _) p) (eq_sigT2_hprop Q_hprop u v).

Non-dependent classification of equality of *sigT* Definition eq_sigT2_nondep {A B C : Type} (u v : { a : A & B & C })
(p : u.1 = v.1) (q : u.2 = v.2) (r : projT3 u = projT3 v)
: u = v
:= @eq_sigT2 _ _ _ u v p (eq_trans (rew_const _ _) q) (eq_trans (rew_const _ _) r).

Classification of transporting across an equality of *sigT2*s Lemma rew_sigT2 {A x} {P : $A \rightarrow$ Type} (Q R : $\forall$ a, P $a \rightarrow$ Prop)
(u : { p : P x & Q x p & R x p })
{y} (H : x = y)
: rew [fun $a \Rightarrow$ { p : P a & Q a p & R a p }] H in u
= existT2
(Q y)
(R y)
(rew H in u.1)
(rew dependent H in u.2)
(rew dependent H in projT3 u).
End sigT2.

Equality for *sig2* Section sig2.
Local Unset Implicit Arguments.

Projecting an equality of a pair to equality of the first components Definition sig_of_sig2_eq {A} {P Q : $A \rightarrow$ Prop} {u v : { a : A | P a & Q a }} (p : u = v)
: u = v :> { a : A | P a }
:= f_equal _ p.
Definition proj1_sig_of_sig2_eq {A} {P Q : $A \rightarrow$ Prop} {u v : { a : A | P a & Q a }} (p : u = v)
: proj1_sig u = proj1_sig v
:= proj1_sig_eq (sig_of_sig2_eq p).

Projecting an equality of a pair to equality of the second components Definition proj2_sig_of_sig2_eq {A} {P Q : $A \rightarrow$ Prop} {u v : { a : A | P a & Q a }} (p : u = v)
: rew proj1_sig_of_sig2_eq p in proj2_sig u = proj2_sig v
:= rew dependent p in eq_refl.

Projecting an equality of a pair to equality of the third components Definition proj3_sig_eq {A} {P Q : $A \rightarrow$ Prop} {u v : { a : A | P a & Q a }} (p : u = v)
: rew proj1_sig_of_sig2_eq p in proj3_sig u = proj3_sig v
:= rew dependent p in eq_refl.

Equality of *sig2* is itself a *sig2* (fowards-reasoning version) Definition eq_exist2_uncurried {A} {P Q : $A \rightarrow$ Prop}
{$u1$ $v1$: A} {$u2$: P $u1$} {$v2$: P $v1$} {$u3$: Q $u1$} {$v3$: Q $v1$}
(pqr : { p : $u1$ = $v1$
| rew p in $u2$ = $v2$ & rew p in $u3$ = $v3$ })
: exist2 _ _ $u1$ $u2$ $u3$ = exist2 _ _ $v1$ $v2$ $v3$.

Equality of *sig2* is itself a *sig2* (backwards-reasoning version) Definition eq_sig2_uncurried {A} {P Q : $A \rightarrow$ Prop} (u v : { a : A | P a & Q a })

(pqr : { p : proj1_sig u = proj1_sig v
| rew p in proj2_sig u = proj2_sig v & rew p in proj3_sig u = proj3_sig v })
: u = v.

Curried version of proving equality of sigma types Definition eq_sig2 {A} {P Q : A → Prop} (u v : { a : A | P a & Q a })
(p : proj1_sig u = proj1_sig v)
(q : rew p in proj2_sig u = proj2_sig v)
(r : rew p in proj3_sig u = proj3_sig v)
: u = v
:= eq_sig2_uncurried u v (exist2 _ _ p q r).

Equality of *sig2* when the second property is an hProp Definition eq_sig2_hprop {A} {P Q : A → Prop} (*Q_hprop* : ∀ (x : A) (p q : Q x), p = q)
(u v : { a : A | P a & Q a })
(p : u = v :> { a : A | P a })
: u = v
:= eq_sig2 u v (proj1_sig_eq p) (proj2_sig_eq p) (*Q_hprop* _ _ _).

Equivalence of equality of *sig2* with a *sig2* of equality We could actually prove an isomorphism here, and not just ↔, but for simplicity, we don't. Definition eq_sig2_uncurried_iff {A P Q}
(u v : { a : A | P a & Q a })
: u = v
↔ { p : proj1_sig u = proj1_sig v
| rew p in proj2_sig u = proj2_sig v & rew p in proj3_sig u = proj3_sig v }.

Induction principle for @*eq* (*sig2* _ _) Definition eq_sig2_rect {A P Q} {u v : { a : A | P a & Q a }} (R : u = v → Type)
(f : ∀ p q r, R (eq_sig2 u v p q r))
: ∀ p, R p.

Definition eq_sig2_rec {A P Q u v} (R : u = v :> { a : A | P a & Q a } → Set) := eq_sig2_rect R.

Definition eq_sig2_ind {A P Q u v} (R : u = v :> { a : A | P a & Q a } → Prop) := eq_sig2_rec R.

Equivalence of equality of *sig2* involving hProps with equality of the first components Definition eq_sig2_hprop_iff {A} {P Q : A → Prop} (*Q_hprop* : ∀ (x : A) (p q : Q x), p = q)
(u v : { a : A | P a & Q a })
: u = v ↔ (u = v :> { a : A | P a })
:= conj (fun p ⇒ f_equal (@sig_of_sig2 _ _ _) p) (eq_sig2_hprop *Q_hprop* u v).

Non-dependent classification of equality of *sig* Definition eq_sig2_nondep {A} {B C : Prop} (u v : @**sig2** A (fun _ ⇒ B) (fun _ ⇒ C))
(p : proj1_sig u = proj1_sig v) (q : proj2_sig u = proj2_sig v) (r : proj3_sig u = proj3_sig v)
: u = v
:= @eq_sig2 _ _ _ u v p (eq_trans (rew_const _ _) q) (eq_trans (rew_const _ _) r).

Classification of transporting across an equality of *sig2*s Lemma rew_sig2 {A x} {P : A → Type} (Q R : ∀ a, P a → Prop)

```
      (u : { p : P x | Q x p & R x p })
      {y} (H : x = y)
  : rew [fun a ⇒ { p : P a | Q a p & R a p }] H in u
    = exist2
        (Q y)
        (R y)
        (rew H in proj1_sig u)
        (rew dependent H in proj2_sig u)
        (rew dependent H in proj3_sig u).
End sig2.
```

sumbool is a boolean type equipped with the justification of their value

```
Inductive sumbool (A B:Prop) : Set :=
  | left : A → {A} + {B}
  | right : B → {A} + {B}
 where "{ A } + { B }" := (sumbool A B) : type_scope.

Add Printing If sumbool.
```

sumor is an option type equipped with the justification of why it may not be a regular value

```
#[universes(template)]
Inductive sumor (A:Type) (B:Prop) : Type :=
  | inleft : A → A + {B}
  | inright : B → A + {B}
 where "A + { B }" := (sumor A B) : type_scope.

Add Printing If sumor.
```

Various forms of the axiom of choice for specifications

```
Section Choice_lemmas.

  Variables S S' : Set.
  Variable R : S → S' → Prop.
  Variable R' : S → S' → Set.
  Variables R1 R2 : S → Prop.

  Lemma Choice :
   (∀ x:S, {y:S' | R x y}) → {f:S → S' | ∀ z:S, R z (f z)}.

  Lemma Choice2 :
   (∀ x:S, {y:S' & R' x y}) → {f:S → S' & ∀ z:S, R' z (f z)}.

  Lemma bool_choice :
   (∀ x:S, {R1 x} + {R2 x}) →
     {f:S → bool | ∀ x:S, f x = true ∧ R1 x ∨ f x = false ∧ R2 x}.

End Choice_lemmas.

Section Dependent_choice_lemmas.

  Variable X : Set.
  Variable R : X → X → Prop.
```

```
  Lemma dependent_choice :
    (∀ x:X, {y | R x y}) →
    ∀ x0, {f : nat → X | f O = x0 ∧ ∀ n, R (f n) (f (S n))}.
End Dependent_choice_lemmas.
```

A result of type (*Exc A*) is either a normal value of type *A* or an *error* :

`Inductive Exc [A:Type] : Type := value : A->(Exc A) | error : (Exc A).`

It is implemented using the option type.

```
Section Exc.
  Variable A : Type.

  Definition Exc := option A.
  Definition value := @Some A.
  Definition error := @None A.
End Exc.

Definition except := False_rec.

Theorem absurd_set : ∀ (A:Prop) (C:Set), A → ¬ A → C.

#[global]
Hint Resolve left right inleft inright: core.
#[global]
Hint Resolve exist exist2 existT existT2: core.
```

Chapter 15

Library Coq.Init.Datatypes

```
Set Implicit Arguments.
Require Import Notations.
Require Import Ltac.
Require Import Logic.
```

15.1 Datatypes with zero and one element

Empty_set is a datatype with no inhabitant

```
Inductive Empty_set : Set :=.
```

unit is a singleton datatype with sole inhabitant *tt*

```
Inductive unit : Set :=
    tt : unit.
```

15.2 The boolean datatype

bool is the datatype of the boolean values *true* and *false*

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
Add Printing If bool.
Delimit Scope bool_scope with bool.
```

Basic boolean operators

```
Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.
Definition orb (b1 b2:bool) : bool := if b1 then true else b2.
Definition implb (b1 b2:bool) : bool := if b1 then b2 else true.
Definition xorb (b1 b2:bool) : bool :=
  match b1, b2 with
```

```
    | true, true ⇒ false
    | true, false ⇒ true
    | false, true ⇒ true
    | false, false ⇒ false
  end.
```

```
Definition negb (b:bool) := if b then false else true.
```

```
Infix "||" := orb : bool_scope.
Infix "&&" := andb : bool_scope.
```

Basic properties of *andb*

```
Lemma andb_prop (a b:bool) : andb a b = true → a = true ∧ b = true.
#[global]
Hint Resolve andb_prop: bool.
```

```
Lemma andb_true_intro (b1 b2:bool) :
  b1 = true ∧ b2 = true → andb b1 b2 = true.
#[global]
Hint Resolve andb_true_intro: bool.
```

Interpretation of booleans as propositions

```
Inductive eq_true : bool → Prop := is_eq_true : eq_true true.
```

```
#[global]
Hint Constructors eq_true : eq_true.
```

Another way of interpreting booleans as propositions

```
Definition is_true b := b = true.
```

is_true can be activated as a coercion by (`Local`) `Coercion` *is_true* : *bool* >-> *Sortclass*.
Additional rewriting lemmas about *eq_true*

```
Lemma eq_true_ind_r :
  ∀ (P : bool → Prop) (b : bool), P b → eq_true b → P true.
```

```
Lemma eq_true_rec_r :
  ∀ (P : bool → Set) (b : bool), P b → eq_true b → P true.
```

```
Lemma eq_true_rect_r :
  ∀ (P : bool → Type) (b : bool), P b → eq_true b → P true.
```

The *BoolSpec* inductive will be used to relate a *boolean* value and two propositions corresponding respectively to the *true* case and the *false* case. Interest: *BoolSpec* behave nicely with `case` and `destruct`. See also *Bool.reflect* when $Q = \neg P$.

```
Inductive BoolSpec (P Q : Prop) : bool → Prop :=
  | BoolSpecT : P → BoolSpec P Q true
  | BoolSpecF : Q → BoolSpec P Q false.
#[global]
Hint Constructors BoolSpec : core.
```

15.3 Peano natural numbers

nat is the datatype of natural numbers built from O and successor S; note that the constructor name is the letter O. Numbers in *nat* can be denoted using a decimal notation; e.g. $3\%nat$ abbreviates S $(S\ (S\ O))$

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

```
Delimit Scope hex_nat_scope with xnat.
```

```
Delimit Scope nat_scope with nat.
```

15.4 Container datatypes

option A is the extension of *A* with an extra element *None*

```
#[universes(template)]
Inductive option (A:Type) : Type :=
  | Some : A → option A
  | None : option A.
```

```
Definition option_map (A B:Type) (f:A→B) (o : option A) : option B :=
  match o with
    | Some a ⇒ @Some B (f a)
    | None ⇒ @None B
  end.
```

sum A B, written $A + B$, is the disjoint sum of A and B

```
#[universes(template)]
Inductive sum (A B:Type) : Type :=
  | inl : A → sum A B
  | inr : B → sum A B.
```

```
Notation "x + y" := (sum x y) : type_scope.
```

prod A B, written $A \times B$, is the product of A and B; the pair *pair A B a b* of a and b is abbreviated (a,b)

```
#[universes(template)]
Inductive prod (A B:Type) : Type :=
  pair : A → B → A × B

where "x * y" := (prod x y) : type_scope.
```

```
Add Printing Let prod.
```

```
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) : core_scope.
```

```
Section projections.
```

```
  Context {A : Type} {B : Type}.

  Definition fst (p:A × B) := match p with (x, y) ⇒ x end.
  Definition snd (p:A × B) := match p with (x, y) ⇒ y end.

End projections.

#[global]
Hint Resolve pair inl inr: core.

Lemma surjective_pairing (A B:Type) (p:A × B) : p = (fst p, snd p).

Lemma injective_projections (A B:Type) (p1 p2:A × B) :
    fst p1 = fst p2 → snd p1 = snd p2 → p1 = p2.

Lemma pair_equal_spec (A B : Type) (a1 a2 : A) (b1 b2 : B) :
    (a1, b1) = (a2, b2) ↔ a1 = a2 ∧ b1 = b2.

Definition curry {A B C:Type} (f:A × B → C)
  (x:A) (y:B) : C := f (x,y).

Definition uncurry {A B C:Type} (f:A → B → C)
  (p:A × B) : C := match p with (x, y) ⇒ f x y end.

#[deprecated(since = "8.13", note = "Use curry instead.")]
Definition prod_uncurry (A B C:Type) : (A × B → C) → A → B → C := curry.

#[deprecated(since = "8.13", note = "Use uncurry instead.")]
Definition prod_curry (A B C:Type) : (A → B → C) → A × B → C := uncurry.

Import EqNotations.

Lemma rew_pair A (P Q : A→Type) x1 x2 (y1:P x1) (y2:Q x1) (H:x1=x2) :
  (rew H in y1, rew H in y2) = rew [fun x ⇒ (P x × Q x)%type] H in (y1,y2).
```

Polymorphic lists and some operations

```
#[universes(template)]
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.

Delimit Scope list_scope with list.

Infix "::" := cons (at level 60, right associativity) : list_scope.

Local Open Scope list_scope.

Definition length (A : Type) : list A → nat :=
  fix length l :=
  match l with
   | nil ⇒ O
   | _ :: l' ⇒ S (length l')
  end.
```

Concatenation of two lists

```
Definition app (A : Type) : list A → list A → list A :=
```

```
  fix app l m :=
  match l with
   | nil ⇒ m
   | a :: l1 ⇒ a :: app l1 m
  end.
Infix "++" := app (right associativity, at level 60) : list_scope.
```

15.5 The comparison datatype

```
Inductive comparison : Set :=
  | Eq : comparison
  | Lt : comparison
  | Gt : comparison.

Lemma comparison_eq_stable (c c' : comparison) : ~~ c = c' → c = c'.

Definition CompOpp (r:comparison) :=
  match r with
    | Eq ⇒ Eq
    | Lt ⇒ Gt
    | Gt ⇒ Lt
  end.

Lemma CompOpp_involutive c : CompOpp (CompOpp c) = c.

Lemma CompOpp_inj c c' : CompOpp c = CompOpp c' → c = c'.

Lemma CompOpp_iff : ∀ c c', CompOpp c = c' ↔ c = CompOpp c'.
```

The *CompareSpec* inductive relates a *comparison* value with three propositions, one for each possible case. Typically, it can be used to specify a comparison function via some equality and order predicates. Interest: *CompareSpec* behave nicely with `case` and `destruct`.

```
Inductive CompareSpec (Peq Plt Pgt : Prop) : comparison → Prop :=
 | CompEq : Peq → CompareSpec Peq Plt Pgt Eq
 | CompLt : Plt → CompareSpec Peq Plt Pgt Lt
 | CompGt : Pgt → CompareSpec Peq Plt Pgt Gt.
#[global]
Hint Constructors CompareSpec : core.
```

For having clean interfaces after extraction, *CompareSpec* is declared in Prop. For some situations, it is nonetheless useful to have a version in Type. Interestingly, these two versions are equivalent.

```
Inductive CompareSpecT (Peq Plt Pgt : Prop) : comparison → Type :=
 | CompEqT : Peq → CompareSpecT Peq Plt Pgt Eq
 | CompLtT : Plt → CompareSpecT Peq Plt Pgt Lt
 | CompGtT : Pgt → CompareSpecT Peq Plt Pgt Gt.
#[global]
Hint Constructors CompareSpecT : core.
```

```
Lemma CompareSpec2Type Peq Plt Pgt c :
 CompareSpec Peq Plt Pgt c → CompareSpecT Peq Plt Pgt c.
```

As an alternate formulation, one may also directly refer to predicates *eq* and *lt* for specifying a comparison, rather that fully-applied propositions. This *CompSpec* is now a particular case of *CompareSpec*.

```
Definition CompSpec {A} (eq lt : A→A→Prop)(x y:A) : comparison → Prop :=
 CompareSpec (eq x y) (lt x y) (lt y x).

Definition CompSpecT {A} (eq lt : A→A→Prop)(x y:A) : comparison → Type :=
 CompareSpecT (eq x y) (lt x y) (lt y x).
#[global]
Hint Unfold CompSpec CompSpecT : core.

Lemma CompSpec2Type : ∀ A (eq lt:A→A→Prop) x y c,
 CompSpec eq lt x y c → CompSpecT eq lt x y c.
```

15.6 Misc Other Datatypes

identity A a is the family of datatypes on *A* whose sole non-empty member is the singleton datatype *identity A a a* whose sole inhabitant is denoted *identity_refl A a* Beware: this inductive actually falls into Prop, as the sole constructor has no arguments and -*indices*-*matter* is not activated in the standard library.

```
Inductive identity (A:Type) (a:A) : A → Type :=
  identity_refl : identity a a.
#[global]
Hint Resolve identity_refl: core.
```

Identity type

```
Definition ID := ∀ A:Type, A → A.
Definition id : ID := fun A x ⇒ x.

Definition IDProp := ∀ A:Prop, A → A.
Definition idProp : IDProp := fun A x ⇒ x.
```

Chapter 16

Library Coq.Init.Byte

16.1 Bytes

Require Import Coq.Init.Ltac.
Require Import Coq.Init.Datatypes.
Require Import Coq.Init.Logic.
Require Import Coq.Init.Specif.
Require Coq.Init.Nat.

We define an inductive for use with the *String* Notation command which contains all ascii characters. We use 256 constructors for efficiency and ease of conversion.

Delimit Scope *byte_scope* with *byte*.

Inductive **byte** :=
| x00
| x01
| x02
| x03
| x04
| x05
| x06
| x07
| x08
| x09
| x0a
| x0b
| x0c
| x0d
| x0e
| x0f
| x10
| x11
| x12

```
| x13
| x14
| x15
| x16
| x17
| x18
| x19
| x1a
| x1b
| x1c
| x1d
| x1e
| x1f
| x20
| x21
| x22
| x23
| x24
| x25
| x26
| x27
| x28
| x29
| x2a
| x2b
| x2c
| x2d
| x2e
| x2f
| x30
| x31
| x32
| x33
| x34
| x35
| x36
| x37
| x38
| x39
| x3a
| x3b
| x3c
| x3d
| x3e
| x3f
```

```
| x40
| x41
| x42
| x43
| x44
| x45
| x46
| x47
| x48
| x49
| x4a
| x4b
| x4c
| x4d
| x4e
| x4f
| x50
| x51
| x52
| x53
| x54
| x55
| x56
| x57
| x58
| x59
| x5a
| x5b
| x5c
| x5d
| x5e
| x5f
| x60
| x61
| x62
| x63
| x64
| x65
| x66
| x67
| x68
| x69
| x6a
| x6b
| x6c
```

| x6d
| x6e
| x6f
| x70
| x71
| x72
| x73
| x74
| x75
| x76
| x77
| x78
| x79
| x7a
| x7b
| x7c
| x7d
| x7e
| x7f
| x80
| x81
| x82
| x83
| x84
| x85
| x86
| x87
| x88
| x89
| x8a
| x8b
| x8c
| x8d
| x8e
| x8f
| x90
| x91
| x92
| x93
| x94
| x95
| x96
| x97
| x98
| x99

| x9a
| x9b
| x9c
| x9d
| x9e
| x9f
| xa0
| xa1
| xa2
| xa3
| xa4
| xa5
| xa6
| xa7
| xa8
| xa9
| xaa
| xab
| xac
| xad
| xae
| xaf
| xb0
| xb1
| xb2
| xb3
| xb4
| xb5
| xb6
| xb7
| xb8
| xb9
| xba
| xbb
| xbc
| xbd
| xbe
| xbf
| xc0
| xc1
| xc2
| xc3
| xc4
| xc5
| xc6

```
| xc7
| xc8
| xc9
| xca
| xcb
| xcc
| xcd
| xce
| xcf
| xd0
| xd1
| xd2
| xd3
| xd4
| xd5
| xd6
| xd7
| xd8
| xd9
| xda
| xdb
| xdc
| xdd
| xde
| xdf
| xe0
| xe1
| xe2
| xe3
| xe4
| xe5
| xe6
| xe7
| xe8
| xe9
| xea
| xeb
| xec
| xed
| xee
| xef
| xf0
| xf1
| xf2
| xf3
```

| xf4
| xf5
| xf6
| xf7
| xf8
| xf9
| xfa
| xfb
| xfc
| xfd
| xfe
| xff
.

We pick a definition that matches with *Ascii.ascii* `Definition` of_bits (b : **bool** × (**bool** × (**bool** × (**bool** × (**bool** × (**bool** × (**bool** × **bool**))))))) : **byte**
:= `match` b `with`
| (0,(0,(0,(0,(0,(0,(0,0))))))) ⇒ x00
| (1,(0,(0,(0,(0,(0,(0,0))))))) ⇒ x01
| (0,(1,(0,(0,(0,(0,(0,0))))))) ⇒ x02
| (1,(1,(0,(0,(0,(0,(0,0))))))) ⇒ x03
| (0,(0,(1,(0,(0,(0,(0,0))))))) ⇒ x04
| (1,(0,(1,(0,(0,(0,(0,0))))))) ⇒ x05
| (0,(1,(1,(0,(0,(0,(0,0))))))) ⇒ x06
| (1,(1,(1,(0,(0,(0,(0,0))))))) ⇒ x07
| (0,(0,(0,(1,(0,(0,(0,0))))))) ⇒ x08
| (1,(0,(0,(1,(0,(0,(0,0))))))) ⇒ x09
| (0,(1,(0,(1,(0,(0,(0,0))))))) ⇒ x0a
| (1,(1,(0,(1,(0,(0,(0,0))))))) ⇒ x0b
| (0,(0,(1,(1,(0,(0,(0,0))))))) ⇒ x0c
| (1,(0,(1,(1,(0,(0,(0,0))))))) ⇒ x0d
| (0,(1,(1,(1,(0,(0,(0,0))))))) ⇒ x0e
| (1,(1,(1,(1,(0,(0,(0,0))))))) ⇒ x0f
| (0,(0,(0,(0,(1,(0,(0,0))))))) ⇒ x10
| (1,(0,(0,(0,(1,(0,(0,0))))))) ⇒ x11
| (0,(1,(0,(0,(1,(0,(0,0))))))) ⇒ x12
| (1,(1,(0,(0,(1,(0,(0,0))))))) ⇒ x13
| (0,(0,(1,(0,(1,(0,(0,0))))))) ⇒ x14
| (1,(0,(1,(0,(1,(0,(0,0))))))) ⇒ x15
| (0,(1,(1,(0,(1,(0,(0,0))))))) ⇒ x16
| (1,(1,(1,(0,(1,(0,(0,0))))))) ⇒ x17
| (0,(0,(0,(1,(1,(0,(0,0))))))) ⇒ x18
| (1,(0,(0,(1,(1,(0,(0,0))))))) ⇒ x19
| (0,(1,(0,(1,(1,(0,(0,0))))))) ⇒ x1a
| (1,(1,(0,(1,(1,(0,(0,0))))))) ⇒ x1b

```
| (0,(0,(1,(1,(1,(0,(0,0))))))) ⇒ x1c
| (1,(0,(1,(1,(1,(0,(0,0))))))) ⇒ x1d
| (0,(1,(1,(1,(1,(0,(0,0))))))) ⇒ x1e
| (1,(1,(1,(1,(1,(0,(0,0))))))) ⇒ x1f
| (0,(0,(0,(0,(0,(1,(0,0))))))) ⇒ x20
| (1,(0,(0,(0,(0,(1,(0,0))))))) ⇒ x21
| (0,(1,(0,(0,(0,(1,(0,0))))))) ⇒ x22
| (1,(1,(0,(0,(0,(1,(0,0))))))) ⇒ x23
| (0,(0,(1,(0,(0,(1,(0,0))))))) ⇒ x24
| (1,(0,(1,(0,(0,(1,(0,0))))))) ⇒ x25
| (0,(1,(1,(0,(0,(1,(0,0))))))) ⇒ x26
| (1,(1,(1,(0,(0,(1,(0,0))))))) ⇒ x27
| (0,(0,(0,(1,(0,(1,(0,0))))))) ⇒ x28
| (1,(0,(0,(1,(0,(1,(0,0))))))) ⇒ x29
| (0,(1,(0,(1,(0,(1,(0,0))))))) ⇒ x2a
| (1,(1,(0,(1,(0,(1,(0,0))))))) ⇒ x2b
| (0,(0,(1,(1,(0,(1,(0,0))))))) ⇒ x2c
| (1,(0,(1,(1,(0,(1,(0,0))))))) ⇒ x2d
| (0,(1,(1,(1,(0,(1,(0,0))))))) ⇒ x2e
| (1,(1,(1,(1,(0,(1,(0,0))))))) ⇒ x2f
| (0,(0,(0,(0,(1,(1,(0,0))))))) ⇒ x30
| (1,(0,(0,(0,(1,(1,(0,0))))))) ⇒ x31
| (0,(1,(0,(0,(1,(1,(0,0))))))) ⇒ x32
| (1,(1,(0,(0,(1,(1,(0,0))))))) ⇒ x33
| (0,(0,(1,(0,(1,(1,(0,0))))))) ⇒ x34
| (1,(0,(1,(0,(1,(1,(0,0))))))) ⇒ x35
| (0,(1,(1,(0,(1,(1,(0,0))))))) ⇒ x36
| (1,(1,(1,(0,(1,(1,(0,0))))))) ⇒ x37
| (0,(0,(0,(1,(1,(1,(0,0))))))) ⇒ x38
| (1,(0,(0,(1,(1,(1,(0,0))))))) ⇒ x39
| (0,(1,(0,(1,(1,(1,(0,0))))))) ⇒ x3a
| (1,(1,(0,(1,(1,(1,(0,0))))))) ⇒ x3b
| (0,(0,(1,(1,(1,(1,(0,0))))))) ⇒ x3c
| (1,(0,(1,(1,(1,(1,(0,0))))))) ⇒ x3d
| (0,(1,(1,(1,(1,(1,(0,0))))))) ⇒ x3e
| (1,(1,(1,(1,(1,(1,(0,0))))))) ⇒ x3f
| (0,(0,(0,(0,(0,(0,(1,0))))))) ⇒ x40
| (1,(0,(0,(0,(0,(0,(1,0))))))) ⇒ x41
| (0,(1,(0,(0,(0,(0,(1,0))))))) ⇒ x42
| (1,(1,(0,(0,(0,(0,(1,0))))))) ⇒ x43
| (0,(0,(1,(0,(0,(0,(1,0))))))) ⇒ x44
| (1,(0,(1,(0,(0,(0,(1,0))))))) ⇒ x45
| (0,(1,(1,(0,(0,(0,(1,0))))))) ⇒ x46
| (1,(1,(1,(0,(0,(0,(1,0))))))) ⇒ x47
| (0,(0,(0,(1,(0,(0,(1,0))))))) ⇒ x48
```

```
| (1,(0,(0,(1,(0,(0,(1,0))))))) ⇒ x49
| (0,(1,(0,(1,(0,(0,(1,0))))))) ⇒ x4a
| (1,(1,(0,(1,(0,(0,(1,0))))))) ⇒ x4b
| (0,(0,(1,(1,(0,(0,(1,0))))))) ⇒ x4c
| (1,(0,(1,(1,(0,(0,(1,0))))))) ⇒ x4d
| (0,(1,(1,(1,(0,(0,(1,0))))))) ⇒ x4e
| (1,(1,(1,(1,(0,(0,(1,0))))))) ⇒ x4f
| (0,(0,(0,(0,(1,(0,(1,0))))))) ⇒ x50
| (1,(0,(0,(0,(1,(0,(1,0))))))) ⇒ x51
| (0,(1,(0,(0,(1,(0,(1,0))))))) ⇒ x52
| (1,(1,(0,(0,(1,(0,(1,0))))))) ⇒ x53
| (0,(0,(1,(0,(1,(0,(1,0))))))) ⇒ x54
| (1,(0,(1,(0,(1,(0,(1,0))))))) ⇒ x55
| (0,(1,(1,(0,(1,(0,(1,0))))))) ⇒ x56
| (1,(1,(1,(0,(1,(0,(1,0))))))) ⇒ x57
| (0,(0,(0,(1,(1,(0,(1,0))))))) ⇒ x58
| (1,(0,(0,(1,(1,(0,(1,0))))))) ⇒ x59
| (0,(1,(0,(1,(1,(0,(1,0))))))) ⇒ x5a
| (1,(1,(0,(1,(1,(0,(1,0))))))) ⇒ x5b
| (0,(0,(1,(1,(1,(0,(1,0))))))) ⇒ x5c
| (1,(0,(1,(1,(1,(0,(1,0))))))) ⇒ x5d
| (0,(1,(1,(1,(1,(0,(1,0))))))) ⇒ x5e
| (1,(1,(1,(1,(1,(0,(1,0))))))) ⇒ x5f
| (0,(0,(0,(0,(0,(1,(1,0))))))) ⇒ x60
| (1,(0,(0,(0,(0,(1,(1,0))))))) ⇒ x61
| (0,(1,(0,(0,(0,(1,(1,0))))))) ⇒ x62
| (1,(1,(0,(0,(0,(1,(1,0))))))) ⇒ x63
| (0,(0,(1,(0,(0,(1,(1,0))))))) ⇒ x64
| (1,(0,(1,(0,(0,(1,(1,0))))))) ⇒ x65
| (0,(1,(1,(0,(0,(1,(1,0))))))) ⇒ x66
| (1,(1,(1,(0,(0,(1,(1,0))))))) ⇒ x67
| (0,(0,(0,(1,(0,(1,(1,0))))))) ⇒ x68
| (1,(0,(0,(1,(0,(1,(1,0))))))) ⇒ x69
| (0,(1,(0,(1,(0,(1,(1,0))))))) ⇒ x6a
| (1,(1,(0,(1,(0,(1,(1,0))))))) ⇒ x6b
| (0,(0,(1,(1,(0,(1,(1,0))))))) ⇒ x6c
| (1,(0,(1,(1,(0,(1,(1,0))))))) ⇒ x6d
| (0,(1,(1,(1,(0,(1,(1,0))))))) ⇒ x6e
| (1,(1,(1,(1,(0,(1,(1,0))))))) ⇒ x6f
| (0,(0,(0,(0,(1,(1,(1,0))))))) ⇒ x70
| (1,(0,(0,(0,(1,(1,(1,0))))))) ⇒ x71
| (0,(1,(0,(0,(1,(1,(1,0))))))) ⇒ x72
| (1,(1,(0,(0,(1,(1,(1,0))))))) ⇒ x73
| (0,(0,(1,(0,(1,(1,(1,0))))))) ⇒ x74
| (1,(0,(1,(0,(1,(1,(1,0))))))) ⇒ x75
```

| (0,(1,(1,(0,(1,(1,(1,0))))))) ⇒ x76
| (1,(1,(1,(0,(1,(1,(1,0))))))) ⇒ x77
| (0,(0,(0,(1,(1,(1,(1,0))))))) ⇒ x78
| (1,(0,(0,(1,(1,(1,(1,0))))))) ⇒ x79
| (0,(1,(0,(1,(1,(1,(1,0))))))) ⇒ x7a
| (1,(1,(0,(1,(1,(1,(1,0))))))) ⇒ x7b
| (0,(0,(1,(1,(1,(1,(1,0))))))) ⇒ x7c
| (1,(0,(1,(1,(1,(1,(1,0))))))) ⇒ x7d
| (0,(1,(1,(1,(1,(1,(1,0))))))) ⇒ x7e
| (1,(1,(1,(1,(1,(1,(1,0))))))) ⇒ x7f
| (0,(0,(0,(0,(0,(0,(0,1))))))) ⇒ x80
| (1,(0,(0,(0,(0,(0,(0,1))))))) ⇒ x81
| (0,(1,(0,(0,(0,(0,(0,1))))))) ⇒ x82
| (1,(1,(0,(0,(0,(0,(0,1))))))) ⇒ x83
| (0,(0,(1,(0,(0,(0,(0,1))))))) ⇒ x84
| (1,(0,(1,(0,(0,(0,(0,1))))))) ⇒ x85
| (0,(1,(1,(0,(0,(0,(0,1))))))) ⇒ x86
| (1,(1,(1,(0,(0,(0,(0,1))))))) ⇒ x87
| (0,(0,(0,(1,(0,(0,(0,1))))))) ⇒ x88
| (1,(0,(0,(1,(0,(0,(0,1))))))) ⇒ x89
| (0,(1,(0,(1,(0,(0,(0,1))))))) ⇒ x8a
| (1,(1,(0,(1,(0,(0,(0,1))))))) ⇒ x8b
| (0,(0,(1,(1,(0,(0,(0,1))))))) ⇒ x8c
| (1,(0,(1,(1,(0,(0,(0,1))))))) ⇒ x8d
| (0,(1,(1,(1,(0,(0,(0,1))))))) ⇒ x8e
| (1,(1,(1,(1,(0,(0,(0,1))))))) ⇒ x8f
| (0,(0,(0,(0,(1,(0,(0,1))))))) ⇒ x90
| (1,(0,(0,(0,(1,(0,(0,1))))))) ⇒ x91
| (0,(1,(0,(0,(1,(0,(0,1))))))) ⇒ x92
| (1,(1,(0,(0,(1,(0,(0,1))))))) ⇒ x93
| (0,(0,(1,(0,(1,(0,(0,1))))))) ⇒ x94
| (1,(0,(1,(0,(1,(0,(0,1))))))) ⇒ x95
| (0,(1,(1,(0,(1,(0,(0,1))))))) ⇒ x96
| (1,(1,(1,(0,(1,(0,(0,1))))))) ⇒ x97
| (0,(0,(0,(1,(1,(0,(0,1))))))) ⇒ x98
| (1,(0,(0,(1,(1,(0,(0,1))))))) ⇒ x99
| (0,(1,(0,(1,(1,(0,(0,1))))))) ⇒ x9a
| (1,(1,(0,(1,(1,(0,(0,1))))))) ⇒ x9b
| (0,(0,(1,(1,(1,(0,(0,1))))))) ⇒ x9c
| (1,(0,(1,(1,(1,(0,(0,1))))))) ⇒ x9d
| (0,(1,(1,(1,(1,(0,(0,1))))))) ⇒ x9e
| (1,(1,(1,(1,(1,(0,(0,1))))))) ⇒ x9f
| (0,(0,(0,(0,(0,(1,(0,1))))))) ⇒ xa0
| (1,(0,(0,(0,(0,(1,(0,1))))))) ⇒ xa1
| (0,(1,(0,(0,(0,(1,(0,1))))))) ⇒ xa2

```
| (1,(1,(0,(0,(0,(1,(0,1))))))) ⇒ xa3
| (0,(0,(1,(0,(0,(1,(0,1))))))) ⇒ xa4
| (1,(0,(1,(0,(0,(1,(0,1))))))) ⇒ xa5
| (0,(1,(1,(0,(0,(1,(0,1))))))) ⇒ xa6
| (1,(1,(1,(0,(0,(1,(0,1))))))) ⇒ xa7
| (0,(0,(0,(1,(0,(1,(0,1))))))) ⇒ xa8
| (1,(0,(0,(1,(0,(1,(0,1))))))) ⇒ xa9
| (0,(1,(0,(1,(0,(1,(0,1))))))) ⇒ xaa
| (1,(1,(0,(1,(0,(1,(0,1))))))) ⇒ xab
| (0,(0,(1,(1,(0,(1,(0,1))))))) ⇒ xac
| (1,(0,(1,(1,(0,(1,(0,1))))))) ⇒ xad
| (0,(1,(1,(1,(0,(1,(0,1))))))) ⇒ xae
| (1,(1,(1,(1,(0,(1,(0,1))))))) ⇒ xaf
| (0,(0,(0,(0,(1,(1,(0,1))))))) ⇒ xb0
| (1,(0,(0,(0,(1,(1,(0,1))))))) ⇒ xb1
| (0,(1,(0,(0,(1,(1,(0,1))))))) ⇒ xb2
| (1,(1,(0,(0,(1,(1,(0,1))))))) ⇒ xb3
| (0,(0,(1,(0,(1,(1,(0,1))))))) ⇒ xb4
| (1,(0,(1,(0,(1,(1,(0,1))))))) ⇒ xb5
| (0,(1,(1,(0,(1,(1,(0,1))))))) ⇒ xb6
| (1,(1,(1,(0,(1,(1,(0,1))))))) ⇒ xb7
| (0,(0,(0,(1,(1,(1,(0,1))))))) ⇒ xb8
| (1,(0,(0,(1,(1,(1,(0,1))))))) ⇒ xb9
| (0,(1,(0,(1,(1,(1,(0,1))))))) ⇒ xba
| (1,(1,(0,(1,(1,(1,(0,1))))))) ⇒ xbb
| (0,(0,(1,(1,(1,(1,(0,1))))))) ⇒ xbc
| (1,(0,(1,(1,(1,(1,(0,1))))))) ⇒ xbd
| (0,(1,(1,(1,(1,(1,(0,1))))))) ⇒ xbe
| (1,(1,(1,(1,(1,(1,(0,1))))))) ⇒ xbf
| (0,(0,(0,(0,(0,(0,(1,1))))))) ⇒ xc0
| (1,(0,(0,(0,(0,(0,(1,1))))))) ⇒ xc1
| (0,(1,(0,(0,(0,(0,(1,1))))))) ⇒ xc2
| (1,(1,(0,(0,(0,(0,(1,1))))))) ⇒ xc3
| (0,(0,(1,(0,(0,(0,(1,1))))))) ⇒ xc4
| (1,(0,(1,(0,(0,(0,(1,1))))))) ⇒ xc5
| (0,(1,(1,(0,(0,(0,(1,1))))))) ⇒ xc6
| (1,(1,(1,(0,(0,(0,(1,1))))))) ⇒ xc7
| (0,(0,(0,(1,(0,(0,(1,1))))))) ⇒ xc8
| (1,(0,(0,(1,(0,(0,(1,1))))))) ⇒ xc9
| (0,(1,(0,(1,(0,(0,(1,1))))))) ⇒ xca
| (1,(1,(0,(1,(0,(0,(1,1))))))) ⇒ xcb
| (0,(0,(1,(1,(0,(0,(1,1))))))) ⇒ xcc
| (1,(0,(1,(1,(0,(0,(1,1))))))) ⇒ xcd
| (0,(1,(1,(1,(0,(0,(1,1))))))) ⇒ xce
| (1,(1,(1,(1,(0,(0,(1,1))))))) ⇒ xcf
```

```
| (0,(0,(0,(0,(1,(0,(1,1))))))) ⇒ xd0
| (1,(0,(0,(0,(1,(0,(1,1))))))) ⇒ xd1
| (0,(1,(0,(0,(1,(0,(1,1))))))) ⇒ xd2
| (1,(1,(0,(0,(1,(0,(1,1))))))) ⇒ xd3
| (0,(0,(1,(0,(1,(0,(1,1))))))) ⇒ xd4
| (1,(0,(1,(0,(1,(0,(1,1))))))) ⇒ xd5
| (0,(1,(1,(0,(1,(0,(1,1))))))) ⇒ xd6
| (1,(1,(1,(0,(1,(0,(1,1))))))) ⇒ xd7
| (0,(0,(0,(1,(1,(0,(1,1))))))) ⇒ xd8
| (1,(0,(0,(1,(1,(0,(1,1))))))) ⇒ xd9
| (0,(1,(0,(1,(1,(0,(1,1))))))) ⇒ xda
| (1,(1,(0,(1,(1,(0,(1,1))))))) ⇒ xdb
| (0,(0,(1,(1,(1,(0,(1,1))))))) ⇒ xdc
| (1,(0,(1,(1,(1,(0,(1,1))))))) ⇒ xdd
| (0,(1,(1,(1,(1,(0,(1,1))))))) ⇒ xde
| (1,(1,(1,(1,(1,(0,(1,1))))))) ⇒ xdf
| (0,(0,(0,(0,(0,(1,(1,1))))))) ⇒ xe0
| (1,(0,(0,(0,(0,(1,(1,1))))))) ⇒ xe1
| (0,(1,(0,(0,(0,(1,(1,1))))))) ⇒ xe2
| (1,(1,(0,(0,(0,(1,(1,1))))))) ⇒ xe3
| (0,(0,(1,(0,(0,(1,(1,1))))))) ⇒ xe4
| (1,(0,(1,(0,(0,(1,(1,1))))))) ⇒ xe5
| (0,(1,(1,(0,(0,(1,(1,1))))))) ⇒ xe6
| (1,(1,(1,(0,(0,(1,(1,1))))))) ⇒ xe7
| (0,(0,(0,(1,(0,(1,(1,1))))))) ⇒ xe8
| (1,(0,(0,(1,(0,(1,(1,1))))))) ⇒ xe9
| (0,(1,(0,(1,(0,(1,(1,1))))))) ⇒ xea
| (1,(1,(0,(1,(0,(1,(1,1))))))) ⇒ xeb
| (0,(0,(1,(1,(0,(1,(1,1))))))) ⇒ xec
| (1,(0,(1,(1,(0,(1,(1,1))))))) ⇒ xed
| (0,(1,(1,(1,(0,(1,(1,1))))))) ⇒ xee
| (1,(1,(1,(1,(0,(1,(1,1))))))) ⇒ xef
| (0,(0,(0,(0,(1,(1,(1,1))))))) ⇒ xf0
| (1,(0,(0,(0,(1,(1,(1,1))))))) ⇒ xf1
| (0,(1,(0,(0,(1,(1,(1,1))))))) ⇒ xf2
| (1,(1,(0,(0,(1,(1,(1,1))))))) ⇒ xf3
| (0,(0,(1,(0,(1,(1,(1,1))))))) ⇒ xf4
| (1,(0,(1,(0,(1,(1,(1,1))))))) ⇒ xf5
| (0,(1,(1,(0,(1,(1,(1,1))))))) ⇒ xf6
| (1,(1,(1,(0,(1,(1,(1,1))))))) ⇒ xf7
| (0,(0,(0,(1,(1,(1,(1,1))))))) ⇒ xf8
| (1,(0,(0,(1,(1,(1,(1,1))))))) ⇒ xf9
| (0,(1,(0,(1,(1,(1,(1,1))))))) ⇒ xfa
| (1,(1,(0,(1,(1,(1,(1,1))))))) ⇒ xfb
| (0,(0,(1,(1,(1,(1,(1,1))))))) ⇒ xfc
```

```
    | (1,(0,(1,(1,(1,(1,(1,1))))))) ⇒ xfd
    | (0,(1,(1,(1,(1,(1,(1,1))))))) ⇒ xfe
    | (1,(1,(1,(1,(1,(1,(1,1))))))) ⇒ xff
    end.
Definition to_bits (b : byte) : bool × (bool × (bool × (bool × (bool × (bool × (bool × bool))))))
  := match b with
    | x00 ⇒ (0,(0,(0,(0,(0,(0,(0,0)))))))
    | x01 ⇒ (1,(0,(0,(0,(0,(0,(0,0)))))))
    | x02 ⇒ (0,(1,(0,(0,(0,(0,(0,0)))))))
    | x03 ⇒ (1,(1,(0,(0,(0,(0,(0,0)))))))
    | x04 ⇒ (0,(0,(1,(0,(0,(0,(0,0)))))))
    | x05 ⇒ (1,(0,(1,(0,(0,(0,(0,0)))))))
    | x06 ⇒ (0,(1,(1,(0,(0,(0,(0,0)))))))
    | x07 ⇒ (1,(1,(1,(0,(0,(0,(0,0)))))))
    | x08 ⇒ (0,(0,(0,(1,(0,(0,(0,0)))))))
    | x09 ⇒ (1,(0,(0,(1,(0,(0,(0,0)))))))
    | x0a ⇒ (0,(1,(0,(1,(0,(0,(0,0)))))))
    | x0b ⇒ (1,(1,(0,(1,(0,(0,(0,0)))))))
    | x0c ⇒ (0,(0,(1,(1,(0,(0,(0,0)))))))
    | x0d ⇒ (1,(0,(1,(1,(0,(0,(0,0)))))))
    | x0e ⇒ (0,(1,(1,(1,(0,(0,(0,0)))))))
    | x0f ⇒ (1,(1,(1,(1,(0,(0,(0,0)))))))
    | x10 ⇒ (0,(0,(0,(0,(1,(0,(0,0)))))))
    | x11 ⇒ (1,(0,(0,(0,(1,(0,(0,0)))))))
    | x12 ⇒ (0,(1,(0,(0,(1,(0,(0,0)))))))
    | x13 ⇒ (1,(1,(0,(0,(1,(0,(0,0)))))))
    | x14 ⇒ (0,(0,(1,(0,(1,(0,(0,0)))))))
    | x15 ⇒ (1,(0,(1,(0,(1,(0,(0,0)))))))
    | x16 ⇒ (0,(1,(1,(0,(1,(0,(0,0)))))))
    | x17 ⇒ (1,(1,(1,(0,(1,(0,(0,0)))))))
    | x18 ⇒ (0,(0,(0,(1,(1,(0,(0,0)))))))
    | x19 ⇒ (1,(0,(0,(1,(1,(0,(0,0)))))))
    | x1a ⇒ (0,(1,(0,(1,(1,(0,(0,0)))))))
    | x1b ⇒ (1,(1,(0,(1,(1,(0,(0,0)))))))
    | x1c ⇒ (0,(0,(1,(1,(1,(0,(0,0)))))))
    | x1d ⇒ (1,(0,(1,(1,(1,(0,(0,0)))))))
    | x1e ⇒ (0,(1,(1,(1,(1,(0,(0,0)))))))
    | x1f ⇒ (1,(1,(1,(1,(1,(0,(0,0)))))))
    | x20 ⇒ (0,(0,(0,(0,(0,(1,(0,0)))))))
    | x21 ⇒ (1,(0,(0,(0,(0,(1,(0,0)))))))
    | x22 ⇒ (0,(1,(0,(0,(0,(1,(0,0)))))))
    | x23 ⇒ (1,(1,(0,(0,(0,(1,(0,0)))))))
    | x24 ⇒ (0,(0,(1,(0,(0,(1,(0,0)))))))
    | x25 ⇒ (1,(0,(1,(0,(0,(1,(0,0)))))))
```

```
| x26 ⇒ (0,(1,(1,(0,(0,(1,(0,0)))))))
| x27 ⇒ (1,(1,(1,(0,(0,(1,(0,0)))))))
| x28 ⇒ (0,(0,(0,(1,(0,(1,(0,0)))))))
| x29 ⇒ (1,(0,(0,(1,(0,(1,(0,0)))))))
| x2a ⇒ (0,(1,(0,(1,(0,(1,(0,0)))))))
| x2b ⇒ (1,(1,(0,(1,(0,(1,(0,0)))))))
| x2c ⇒ (0,(0,(1,(1,(0,(1,(0,0)))))))
| x2d ⇒ (1,(0,(1,(1,(0,(1,(0,0)))))))
| x2e ⇒ (0,(1,(1,(1,(0,(1,(0,0)))))))
| x2f ⇒ (1,(1,(1,(1,(0,(1,(0,0)))))))
| x30 ⇒ (0,(0,(0,(0,(1,(1,(0,0)))))))
| x31 ⇒ (1,(0,(0,(0,(1,(1,(0,0)))))))
| x32 ⇒ (0,(1,(0,(0,(1,(1,(0,0)))))))
| x33 ⇒ (1,(1,(0,(0,(1,(1,(0,0)))))))
| x34 ⇒ (0,(0,(1,(0,(1,(1,(0,0)))))))
| x35 ⇒ (1,(0,(1,(0,(1,(1,(0,0)))))))
| x36 ⇒ (0,(1,(1,(0,(1,(1,(0,0)))))))
| x37 ⇒ (1,(1,(1,(0,(1,(1,(0,0)))))))
| x38 ⇒ (0,(0,(0,(1,(1,(1,(0,0)))))))
| x39 ⇒ (1,(0,(0,(1,(1,(1,(0,0)))))))
| x3a ⇒ (0,(1,(0,(1,(1,(1,(0,0)))))))
| x3b ⇒ (1,(1,(0,(1,(1,(1,(0,0)))))))
| x3c ⇒ (0,(0,(1,(1,(1,(1,(0,0)))))))
| x3d ⇒ (1,(0,(1,(1,(1,(1,(0,0)))))))
| x3e ⇒ (0,(1,(1,(1,(1,(1,(0,0)))))))
| x3f ⇒ (1,(1,(1,(1,(1,(1,(0,0)))))))
| x40 ⇒ (0,(0,(0,(0,(0,(0,(1,0)))))))
| x41 ⇒ (1,(0,(0,(0,(0,(0,(1,0)))))))
| x42 ⇒ (0,(1,(0,(0,(0,(0,(1,0)))))))
| x43 ⇒ (1,(1,(0,(0,(0,(0,(1,0)))))))
| x44 ⇒ (0,(0,(1,(0,(0,(0,(1,0)))))))
| x45 ⇒ (1,(0,(1,(0,(0,(0,(1,0)))))))
| x46 ⇒ (0,(1,(1,(0,(0,(0,(1,0)))))))
| x47 ⇒ (1,(1,(1,(0,(0,(0,(1,0)))))))
| x48 ⇒ (0,(0,(0,(1,(0,(0,(1,0)))))))
| x49 ⇒ (1,(0,(0,(1,(0,(0,(1,0)))))))
| x4a ⇒ (0,(1,(0,(1,(0,(0,(1,0)))))))
| x4b ⇒ (1,(1,(0,(1,(0,(0,(1,0)))))))
| x4c ⇒ (0,(0,(1,(1,(0,(0,(1,0)))))))
| x4d ⇒ (1,(0,(1,(1,(0,(0,(1,0)))))))
| x4e ⇒ (0,(1,(1,(1,(0,(0,(1,0)))))))
| x4f ⇒ (1,(1,(1,(1,(0,(0,(1,0)))))))
| x50 ⇒ (0,(0,(0,(0,(1,(0,(1,0)))))))
| x51 ⇒ (1,(0,(0,(0,(1,(0,(1,0)))))))
| x52 ⇒ (0,(1,(0,(0,(1,(0,(1,0)))))))
```

```
| x53 ⇒ (1,(1,(0,(0,(1,(0,(1,0)))))))
| x54 ⇒ (0,(0,(1,(0,(1,(0,(1,0)))))))
| x55 ⇒ (1,(0,(1,(0,(1,(0,(1,0)))))))
| x56 ⇒ (0,(1,(1,(0,(1,(0,(1,0)))))))
| x57 ⇒ (1,(1,(1,(0,(1,(0,(1,0)))))))
| x58 ⇒ (0,(0,(0,(1,(1,(0,(1,0)))))))
| x59 ⇒ (1,(0,(0,(1,(1,(0,(1,0)))))))
| x5a ⇒ (0,(1,(0,(1,(1,(0,(1,0)))))))
| x5b ⇒ (1,(1,(0,(1,(1,(0,(1,0)))))))
| x5c ⇒ (0,(0,(1,(1,(1,(0,(1,0)))))))
| x5d ⇒ (1,(0,(1,(1,(1,(0,(1,0)))))))
| x5e ⇒ (0,(1,(1,(1,(1,(0,(1,0)))))))
| x5f ⇒ (1,(1,(1,(1,(1,(0,(1,0)))))))
| x60 ⇒ (0,(0,(0,(0,(0,(1,(1,0)))))))
| x61 ⇒ (1,(0,(0,(0,(0,(1,(1,0)))))))
| x62 ⇒ (0,(1,(0,(0,(0,(1,(1,0)))))))
| x63 ⇒ (1,(1,(0,(0,(0,(1,(1,0)))))))
| x64 ⇒ (0,(0,(1,(0,(0,(1,(1,0)))))))
| x65 ⇒ (1,(0,(1,(0,(0,(1,(1,0)))))))
| x66 ⇒ (0,(1,(1,(0,(0,(1,(1,0)))))))
| x67 ⇒ (1,(1,(1,(0,(0,(1,(1,0)))))))
| x68 ⇒ (0,(0,(0,(1,(0,(1,(1,0)))))))
| x69 ⇒ (1,(0,(0,(1,(0,(1,(1,0)))))))
| x6a ⇒ (0,(1,(0,(1,(0,(1,(1,0)))))))
| x6b ⇒ (1,(1,(0,(1,(0,(1,(1,0)))))))
| x6c ⇒ (0,(0,(1,(1,(0,(1,(1,0)))))))
| x6d ⇒ (1,(0,(1,(1,(0,(1,(1,0)))))))
| x6e ⇒ (0,(1,(1,(1,(0,(1,(1,0)))))))
| x6f ⇒ (1,(1,(1,(1,(0,(1,(1,0)))))))
| x70 ⇒ (0,(0,(0,(0,(1,(1,(1,0)))))))
| x71 ⇒ (1,(0,(0,(0,(1,(1,(1,0)))))))
| x72 ⇒ (0,(1,(0,(0,(1,(1,(1,0)))))))
| x73 ⇒ (1,(1,(0,(0,(1,(1,(1,0)))))))
| x74 ⇒ (0,(0,(1,(0,(1,(1,(1,0)))))))
| x75 ⇒ (1,(0,(1,(0,(1,(1,(1,0)))))))
| x76 ⇒ (0,(1,(1,(0,(1,(1,(1,0)))))))
| x77 ⇒ (1,(1,(1,(0,(1,(1,(1,0)))))))
| x78 ⇒ (0,(0,(0,(1,(1,(1,(1,0)))))))
| x79 ⇒ (1,(0,(0,(1,(1,(1,(1,0)))))))
| x7a ⇒ (0,(1,(0,(1,(1,(1,(1,0)))))))
| x7b ⇒ (1,(1,(0,(1,(1,(1,(1,0)))))))
| x7c ⇒ (0,(0,(1,(1,(1,(1,(1,0)))))))
| x7d ⇒ (1,(0,(1,(1,(1,(1,(1,0)))))))
| x7e ⇒ (0,(1,(1,(1,(1,(1,(1,0)))))))
| x7f ⇒ (1,(1,(1,(1,(1,(1,(1,0)))))))
```

| x80 ⇒ $(0,(0,(0,(0,(0,(0,(0,1)))))))$
| x81 ⇒ $(1,(0,(0,(0,(0,(0,(0,1)))))))$
| x82 ⇒ $(0,(1,(0,(0,(0,(0,(0,1)))))))$
| x83 ⇒ $(1,(1,(0,(0,(0,(0,(0,1)))))))$
| x84 ⇒ $(0,(0,(1,(0,(0,(0,(0,1)))))))$
| x85 ⇒ $(1,(0,(1,(0,(0,(0,(0,1)))))))$
| x86 ⇒ $(0,(1,(1,(0,(0,(0,(0,1)))))))$
| x87 ⇒ $(1,(1,(1,(0,(0,(0,(0,1)))))))$
| x88 ⇒ $(0,(0,(0,(1,(0,(0,(0,1)))))))$
| x89 ⇒ $(1,(0,(0,(1,(0,(0,(0,1)))))))$
| x8a ⇒ $(0,(1,(0,(1,(0,(0,(0,1)))))))$
| x8b ⇒ $(1,(1,(0,(1,(0,(0,(0,1)))))))$
| x8c ⇒ $(0,(0,(1,(1,(0,(0,(0,1)))))))$
| x8d ⇒ $(1,(0,(1,(1,(0,(0,(0,1)))))))$
| x8e ⇒ $(0,(1,(1,(1,(0,(0,(0,1)))))))$
| x8f ⇒ $(1,(1,(1,(1,(0,(0,(0,1)))))))$
| x90 ⇒ $(0,(0,(0,(0,(1,(0,(0,1)))))))$
| x91 ⇒ $(1,(0,(0,(0,(1,(0,(0,1)))))))$
| x92 ⇒ $(0,(1,(0,(0,(1,(0,(0,1)))))))$
| x93 ⇒ $(1,(1,(0,(0,(1,(0,(0,1)))))))$
| x94 ⇒ $(0,(0,(1,(0,(1,(0,(0,1)))))))$
| x95 ⇒ $(1,(0,(1,(0,(1,(0,(0,1)))))))$
| x96 ⇒ $(0,(1,(1,(0,(1,(0,(0,1)))))))$
| x97 ⇒ $(1,(1,(1,(0,(1,(0,(0,1)))))))$
| x98 ⇒ $(0,(0,(0,(1,(1,(0,(0,1)))))))$
| x99 ⇒ $(1,(0,(0,(1,(1,(0,(0,1)))))))$
| x9a ⇒ $(0,(1,(0,(1,(1,(0,(0,1)))))))$
| x9b ⇒ $(1,(1,(0,(1,(1,(0,(0,1)))))))$
| x9c ⇒ $(0,(0,(1,(1,(1,(0,(0,1)))))))$
| x9d ⇒ $(1,(0,(1,(1,(1,(0,(0,1)))))))$
| x9e ⇒ $(0,(1,(1,(1,(1,(0,(0,1)))))))$
| x9f ⇒ $(1,(1,(1,(1,(1,(0,(0,1)))))))$
| xa0 ⇒ $(0,(0,(0,(0,(0,(1,(0,1)))))))$
| xa1 ⇒ $(1,(0,(0,(0,(0,(1,(0,1)))))))$
| xa2 ⇒ $(0,(1,(0,(0,(0,(1,(0,1)))))))$
| xa3 ⇒ $(1,(1,(0,(0,(0,(1,(0,1)))))))$
| xa4 ⇒ $(0,(0,(1,(0,(0,(1,(0,1)))))))$
| xa5 ⇒ $(1,(0,(1,(0,(0,(1,(0,1)))))))$
| xa6 ⇒ $(0,(1,(1,(0,(0,(1,(0,1)))))))$
| xa7 ⇒ $(1,(1,(1,(0,(0,(1,(0,1)))))))$
| xa8 ⇒ $(0,(0,(0,(1,(0,(1,(0,1)))))))$
| xa9 ⇒ $(1,(0,(0,(1,(0,(1,(0,1)))))))$
| xaa ⇒ $(0,(1,(0,(1,(0,(1,(0,1)))))))$
| xab ⇒ $(1,(1,(0,(1,(0,(1,(0,1)))))))$
| xac ⇒ $(0,(0,(1,(1,(0,(1,(0,1)))))))$

```
| xad ⇒ (1,(0,(1,(1,(0,(1,(0,1)))))))
| xae ⇒ (0,(1,(1,(1,(0,(1,(0,1)))))))
| xaf ⇒ (1,(1,(1,(1,(0,(1,(0,1)))))))
| xb0 ⇒ (0,(0,(0,(0,(1,(1,(0,1)))))))
| xb1 ⇒ (1,(0,(0,(0,(1,(1,(0,1)))))))
| xb2 ⇒ (0,(1,(0,(0,(1,(1,(0,1)))))))
| xb3 ⇒ (1,(1,(0,(0,(1,(1,(0,1)))))))
| xb4 ⇒ (0,(0,(1,(0,(1,(1,(0,1)))))))
| xb5 ⇒ (1,(0,(1,(0,(1,(1,(0,1)))))))
| xb6 ⇒ (0,(1,(1,(0,(1,(1,(0,1)))))))
| xb7 ⇒ (1,(1,(1,(0,(1,(1,(0,1)))))))
| xb8 ⇒ (0,(0,(0,(1,(1,(1,(0,1)))))))
| xb9 ⇒ (1,(0,(0,(1,(1,(1,(0,1)))))))
| xba ⇒ (0,(1,(0,(1,(1,(1,(0,1)))))))
| xbb ⇒ (1,(1,(0,(1,(1,(1,(0,1)))))))
| xbc ⇒ (0,(0,(1,(1,(1,(1,(0,1)))))))
| xbd ⇒ (1,(0,(1,(1,(1,(1,(0,1)))))))
| xbe ⇒ (0,(1,(1,(1,(1,(1,(0,1)))))))
| xbf ⇒ (1,(1,(1,(1,(1,(1,(0,1)))))))
| xc0 ⇒ (0,(0,(0,(0,(0,(0,(1,1)))))))
| xc1 ⇒ (1,(0,(0,(0,(0,(0,(1,1)))))))
| xc2 ⇒ (0,(1,(0,(0,(0,(0,(1,1)))))))
| xc3 ⇒ (1,(1,(0,(0,(0,(0,(1,1)))))))
| xc4 ⇒ (0,(0,(1,(0,(0,(0,(1,1)))))))
| xc5 ⇒ (1,(0,(1,(0,(0,(0,(1,1)))))))
| xc6 ⇒ (0,(1,(1,(0,(0,(0,(1,1)))))))
| xc7 ⇒ (1,(1,(1,(0,(0,(0,(1,1)))))))
| xc8 ⇒ (0,(0,(0,(1,(0,(0,(1,1)))))))
| xc9 ⇒ (1,(0,(0,(1,(0,(0,(1,1)))))))
| xca ⇒ (0,(1,(0,(1,(0,(0,(1,1)))))))
| xcb ⇒ (1,(1,(0,(1,(0,(0,(1,1)))))))
| xcc ⇒ (0,(0,(1,(1,(0,(0,(1,1)))))))
| xcd ⇒ (1,(0,(1,(1,(0,(0,(1,1)))))))
| xce ⇒ (0,(1,(1,(1,(0,(0,(1,1)))))))
| xcf ⇒ (1,(1,(1,(1,(0,(0,(1,1)))))))
| xd0 ⇒ (0,(0,(0,(0,(1,(0,(1,1)))))))
| xd1 ⇒ (1,(0,(0,(0,(1,(0,(1,1)))))))
| xd2 ⇒ (0,(1,(0,(0,(1,(0,(1,1)))))))
| xd3 ⇒ (1,(1,(0,(0,(1,(0,(1,1)))))))
| xd4 ⇒ (0,(0,(1,(0,(1,(0,(1,1)))))))
| xd5 ⇒ (1,(0,(1,(0,(1,(0,(1,1)))))))
| xd6 ⇒ (0,(1,(1,(0,(1,(0,(1,1)))))))
| xd7 ⇒ (1,(1,(1,(0,(1,(0,(1,1)))))))
| xd8 ⇒ (0,(0,(0,(1,(1,(0,(1,1)))))))
| xd9 ⇒ (1,(0,(0,(1,(1,(0,(1,1)))))))
```

```
  | xda ⇒ (0,(1,(0,(1,(1,(0,(1,1)))))))
  | xdb ⇒ (1,(1,(0,(1,(1,(0,(1,1)))))))
  | xdc ⇒ (0,(0,(1,(1,(1,(0,(1,1)))))))
  | xdd ⇒ (1,(0,(1,(1,(1,(0,(1,1)))))))
  | xde ⇒ (0,(1,(1,(1,(1,(0,(1,1)))))))
  | xdf ⇒ (1,(1,(1,(1,(1,(0,(1,1)))))))
  | xe0 ⇒ (0,(0,(0,(0,(0,(1,(1,1)))))))
  | xe1 ⇒ (1,(0,(0,(0,(0,(1,(1,1)))))))
  | xe2 ⇒ (0,(1,(0,(0,(0,(1,(1,1)))))))
  | xe3 ⇒ (1,(1,(0,(0,(0,(1,(1,1)))))))
  | xe4 ⇒ (0,(0,(1,(0,(0,(1,(1,1)))))))
  | xe5 ⇒ (1,(0,(1,(0,(0,(1,(1,1)))))))
  | xe6 ⇒ (0,(1,(1,(0,(0,(1,(1,1)))))))
  | xe7 ⇒ (1,(1,(1,(0,(0,(1,(1,1)))))))
  | xe8 ⇒ (0,(0,(0,(1,(0,(1,(1,1)))))))
  | xe9 ⇒ (1,(0,(0,(1,(0,(1,(1,1)))))))
  | xea ⇒ (0,(1,(0,(1,(0,(1,(1,1)))))))
  | xeb ⇒ (1,(1,(0,(1,(0,(1,(1,1)))))))
  | xec ⇒ (0,(0,(1,(1,(0,(1,(1,1)))))))
  | xed ⇒ (1,(0,(1,(1,(0,(1,(1,1)))))))
  | xee ⇒ (0,(1,(1,(1,(0,(1,(1,1)))))))
  | xef ⇒ (1,(1,(1,(1,(0,(1,(1,1)))))))
  | xf0 ⇒ (0,(0,(0,(0,(1,(1,(1,1)))))))
  | xf1 ⇒ (1,(0,(0,(0,(1,(1,(1,1)))))))
  | xf2 ⇒ (0,(1,(0,(0,(1,(1,(1,1)))))))
  | xf3 ⇒ (1,(1,(0,(0,(1,(1,(1,1)))))))
  | xf4 ⇒ (0,(0,(1,(0,(1,(1,(1,1)))))))
  | xf5 ⇒ (1,(0,(1,(0,(1,(1,(1,1)))))))
  | xf6 ⇒ (0,(1,(1,(0,(1,(1,(1,1)))))))
  | xf7 ⇒ (1,(1,(1,(0,(1,(1,(1,1)))))))
  | xf8 ⇒ (0,(0,(0,(1,(1,(1,(1,1)))))))
  | xf9 ⇒ (1,(0,(0,(1,(1,(1,(1,1)))))))
  | xfa ⇒ (0,(1,(0,(1,(1,(1,(1,1)))))))
  | xfb ⇒ (1,(1,(0,(1,(1,(1,(1,1)))))))
  | xfc ⇒ (0,(0,(1,(1,(1,(1,(1,1)))))))
  | xfd ⇒ (1,(0,(1,(1,(1,(1,(1,1)))))))
  | xfe ⇒ (0,(1,(1,(1,(1,(1,(1,1)))))))
  | xff ⇒ (1,(1,(1,(1,(1,(1,(1,1)))))))
  end.

Lemma of_bits_to_bits (b : byte) : of_bits (to_bits b) = b.

Lemma to_bits_of_bits (b : _) : to_bits (of_bits b) = b.

Definition byte_of_byte (b : byte) : byte := b.

Module Export ByteSyntaxNotations.
End ByteSyntaxNotations.
```

Chapter 17

Library Coq.Init.Decimal

17.1 Decimal numbers

These numbers coded in base 10 will be used for parsing and printing other Coq numeral datatypes in an human-readable way. See the *Number* `Notation` command. We represent numbers in base 10 as lists of decimal digits, in big-endian order (most significant digit comes first).

```
Require Import Datatypes Specif.
```

Unsigned integers are just lists of digits. For instance, ten is (D1 (D0 Nil))

```
Inductive uint :=
 | Nil
 | D0 (_:uint)
 | D1 (_:uint)
 | D2 (_:uint)
 | D3 (_:uint)
 | D4 (_:uint)
 | D5 (_:uint)
 | D6 (_:uint)
 | D7 (_:uint)
 | D8 (_:uint)
 | D9 (_:uint).
```

Nil is the number terminator. Taken alone, it behaves as zero, but rather use *D0 Nil* instead, since this form will be denoted as 0, while *Nil* will be printed as *Nil*.

```
Notation zero := (D0 Nil).
```

For signed integers, we use two constructors *Pos* and *Neg*.

```
Variant int := Pos (d:uint) | Neg (d:uint).
```

For decimal numbers, we use two constructors *Decimal* and *DecimalExp*, depending on whether or not they are given with an exponent (e.g., 1.02e+01). i is the integral part while f is the fractional part (beware that leading zeroes do matter).

```
Variant decimal :=
 | Decimal (i:int) (f:uint)
```

```
  | DecimalExp (i:int) (f:uint) (e:int).

Scheme Equality for uint.
Scheme Equality for int.
Scheme Equality for decimal.

Delimit Scope dec_uint_scope with uint.

Delimit Scope dec_int_scope with int.

Fixpoint nb_digits d :=
  match d with
  | Nil ⇒ O
  | D0 d | D1 d | D2 d | D3 d | D4 d | D5 d | D6 d | D7 d | D8 d | D9 d ⇒
    S (nb_digits d)
  end.
```

This representation favors simplicity over canonicity. For normalizing numbers, we need to remove head zero digits, and choose our canonical representation of 0 (here *D0 Nil* for unsigned numbers and *Pos* (*D0 Nil*) for signed numbers).

nzhead removes all head zero digits

```
Fixpoint nzhead d :=
  match d with
  | D0 d ⇒ nzhead d
  | _ ⇒ d
  end.
```

unorm : normalization of unsigned integers

```
Definition unorm d :=
  match nzhead d with
  | Nil ⇒ zero
  | d ⇒ d
  end.
```

norm : normalization of signed integers

```
Definition norm d :=
  match d with
  | Pos d ⇒ Pos (unorm d)
  | Neg d ⇒
    match nzhead d with
    | Nil ⇒ Pos zero
    | d ⇒ Neg d
    end
  end.
```

A few easy operations. For more advanced computations, use the conversions with other Coq numeral datatypes (e.g. Z) and the operations on them.

```
Definition opp (d:int) :=
  match d with
  | Pos d ⇒ Neg d
```

```
  | Neg d ⇒ Pos d
  end.
Definition abs (d:int) : uint :=
  match d with
  | Pos d ⇒ d
  | Neg d ⇒ d
  end.
```

For conversions with binary numbers, it is easier to operate on little-endian numbers.

```
Fixpoint revapp (d d' : uint) :=
  match d with
  | Nil ⇒ d'
  | D0 d ⇒ revapp d (D0 d')
  | D1 d ⇒ revapp d (D1 d')
  | D2 d ⇒ revapp d (D2 d')
  | D3 d ⇒ revapp d (D3 d')
  | D4 d ⇒ revapp d (D4 d')
  | D5 d ⇒ revapp d (D5 d')
  | D6 d ⇒ revapp d (D6 d')
  | D7 d ⇒ revapp d (D7 d')
  | D8 d ⇒ revapp d (D8 d')
  | D9 d ⇒ revapp d (D9 d')
  end.
Definition rev d := revapp d Nil.
Definition app d d' := revapp (rev d) d'.
Definition app_int d1 d2 :=
  match d1 with Pos d1 ⇒ Pos (app d1 d2) | Neg d1 ⇒ Neg (app d1 d2) end.
```

nztail removes all trailing zero digits and return both the result and the number of removed digits.

```
Definition nztail d :=
  let fix aux d_rev :=
    match d_rev with
    | D0 d_rev ⇒ let (r, n) := aux d_rev in pair r (S n)
    | _ ⇒ pair d_rev O
    end in
  let (r, n) := aux (rev d) in pair (rev r) n.
Definition nztail_int d :=
  match d with
  | Pos d ⇒ let (r, n) := nztail d in pair (Pos r) n
  | Neg d ⇒ let (r, n) := nztail d in pair (Neg r) n
  end.
```

del_head n d removes *n* digits at beginning of *d* or returns *zero* if *d* has less than *n* digits.

```
Fixpoint del_head n d :=
```

```
  match n with
  | O ⇒ d
  | S n ⇒
    match d with
    | Nil ⇒ zero
    | D0 d | D1 d | D2 d | D3 d | D4 d | D5 d | D6 d | D7 d | D8 d | D9 d ⇒
      del_head n d
    end
  end.
```

```
Definition del_head_int n d :=
  match d with
  | Pos d ⇒ del_head n d
  | Neg d ⇒ del_head n d
  end.
```

del_tail n d removes n digits at end of d or returns *zero* if d has less than n digits.

```
Definition del_tail n d := rev (del_head n (rev d)).
```

```
Definition del_tail_int n d :=
  match d with
  | Pos d ⇒ Pos (del_tail n d)
  | Neg d ⇒ Neg (del_tail n d)
  end.
```

```
Module Little.
```

Successor of little-endian numbers

```
Fixpoint succ d :=
  match d with
  | Nil ⇒ D1 Nil
  | D0 d ⇒ D1 d
  | D1 d ⇒ D2 d
  | D2 d ⇒ D3 d
  | D3 d ⇒ D4 d
  | D4 d ⇒ D5 d
  | D5 d ⇒ D6 d
  | D6 d ⇒ D7 d
  | D7 d ⇒ D8 d
  | D8 d ⇒ D9 d
  | D9 d ⇒ D0 (succ d)
  end.
```

Doubling little-endian numbers

```
Fixpoint double d :=
  match d with
  | Nil ⇒ Nil
  | D0 d ⇒ D0 (double d)
  | D1 d ⇒ D2 (double d)
```

```
  | D2 d => D4 (double d)
  | D3 d => D6 (double d)
  | D4 d => D8 (double d)
  | D5 d => D0 (succ_double d)
  | D6 d => D2 (succ_double d)
  | D7 d => D4 (succ_double d)
  | D8 d => D6 (succ_double d)
  | D9 d => D8 (succ_double d)
  end

with succ_double d :=
  match d with
  | Nil => D1 Nil
  | D0 d => D1 (double d)
  | D1 d => D3 (double d)
  | D2 d => D5 (double d)
  | D3 d => D7 (double d)
  | D4 d => D9 (double d)
  | D5 d => D1 (succ_double d)
  | D6 d => D3 (succ_double d)
  | D7 d => D5 (succ_double d)
  | D8 d => D7 (succ_double d)
  | D9 d => D9 (succ_double d)
  end.
End Little.
```

Pseudo-conversion functions used when declaring Number Notations on *uint* and *int*.

```
Definition uint_of_uint (i:uint) := i.
Definition int_of_int (i:int) := i.
```

Chapter 18

Library Coq.Arith.Peano_dec

```
Require Import Decidable PeanoNat.
Require Eqdep_dec.
Local Open Scope nat_scope.
```

```
Implicit Types m n x y : nat.
```

```
Theorem O_or_S n : {m : nat | S m = n} + {0 = n}.
```

```
Notation eq_nat_dec := Nat.eq_dec (only parsing).
```

```
#[global]
Hint Resolve O_or_S eq_nat_dec: arith.
```

```
Theorem dec_eq_nat n m : decidable (n = m).
```

```
Definition UIP_nat:= Eqdep_dec.UIP_dec Nat.eq_dec.
```

```
Import EqNotations.
```

```
Lemma le_unique: ∀ m n (le_mn1 le_mn2 : m ≤ n), le_mn1 = le_mn2.
```

For compatibility `Require Import Le Lt.`

Chapter 19

Library Coq.Arith.EqNat

```
Require Import PeanoNat.
Local Open Scope nat_scope.
```

Equality on natural numbers

19.1 Propositional equality

```
Fixpoint eq_nat n m : Prop :=
  match n, m with
    | O, O ⇒ True
    | O, S _ ⇒ False
    | S _, O ⇒ False
    | S n1, S m1 ⇒ eq_nat n1 m1
  end.
```

```
Theorem eq_nat_refl n : eq_nat n n.
#[global]
Hint Resolve eq_nat_refl: arith.
```

eq restricted to *nat* and *eq_nat* are equivalent

```
Theorem eq_nat_is_eq n m : eq_nat n m ↔ n = m.
```

```
Lemma eq_eq_nat n m : n = m → eq_nat n m.
```

```
Lemma eq_nat_eq n m : eq_nat n m → n = m.
```

```
#[global]
Hint Immediate eq_eq_nat eq_nat_eq: arith.
```

```
Theorem eq_nat_elim :
  ∀ n (P:nat → Prop), P n → ∀ m, eq_nat n m → P m.
```

```
Theorem eq_nat_decide : ∀ n m, {eq_nat n m} + {¬ eq_nat n m}.
```

19.2 Boolean equality on *nat*.

We reuse the one already defined in module *Nat*. In scope *nat_scope*, the notation "=?" can be used.

Notation beq_nat := Nat.eqb (*only parsing*).

Notation beq_nat_true_iff := Nat.eqb_eq (*only parsing*).
Notation beq_nat_false_iff := Nat.eqb_neq (*only parsing*).

Lemma beq_nat_refl n : true = (n =? n).

Lemma beq_nat_true n m : (n =? m) = true → n=m.

Lemma beq_nat_false n m : (n =? m) = false → n≠m.

TODO: is it really useful here to have a Defined ? Otherwise we could use Nat.eqb_eq

Definition beq_nat_eq : ∀ n m, true = (n =? m) → n = m.

Chapter 20

Library Coq.Arith.Even

Nota : this file is OBSOLETE, and left only for compatibility. Please consider instead predicates *Nat.Even* and *Nat.Odd* and Boolean functions *Nat.even* and *Nat.odd*.

Here we define the predicates *even* and *odd* by mutual induction and we prove the decidability and the exclusion of those predicates. The main results about parity are proved in the module Div2.

```
Require Import PeanoNat.

Local Open Scope nat_scope.

Implicit Types m n : nat.
```

20.1 Inductive definition of *even* and *odd*

```
Inductive even : nat → Prop :=
  | even_O : even 0
  | even_S : ∀ n, odd n → even (S n)
with odd : nat → Prop :=
    odd_S : ∀ n, even n → odd (S n).
#[global]
Hint Constructors even: arith.
#[global]
Hint Constructors odd: arith.
```

20.2 Equivalence with predicates *Nat.Even* and *Nat.odd*

```
Lemma even_equiv : ∀ n, even n ↔ Nat.Even n.

Lemma odd_equiv : ∀ n, odd n ↔ Nat.Odd n.
```

Basic facts

```
Lemma even_or_odd n : even n ∨ odd n.

Lemma even_odd_dec n : {even n} + {odd n}.

Lemma not_even_and_odd n : even n → odd n → False.
```

20.3 Facts about *even* & *odd* wrt. *plus*

Ltac *parity2bool* :=
 rewrite ?even_equiv, ?odd_equiv, ← ?Nat.even_spec, ← ?Nat.odd_spec.

Ltac *parity_binop_spec* :=
 rewrite ?Nat.even_add, ?Nat.odd_add, ?Nat.even_mul, ?Nat.odd_mul.

Ltac *parity_binop* :=
 parity2bool; *parity_binop_spec*; unfold Nat.odd;
 do 2 destruct Nat.even; simpl; tauto.

Lemma even_plus_split n m :
 even $(n + m) \to$ **even** $n \land$ **even** $m \lor$ **odd** $n \land$ **odd** m.

Lemma odd_plus_split n m :
 odd $(n + m) \to$ **odd** $n \land$ **even** $m \lor$ **even** $n \land$ **odd** m.

Lemma even_even_plus n m : **even** $n \to$ **even** $m \to$ **even** $(n + m)$.

Lemma odd_plus_l n m : **odd** $n \to$ **even** $m \to$ **odd** $(n + m)$.

Lemma odd_plus_r n m : **even** $n \to$ **odd** $m \to$ **odd** $(n + m)$.

Lemma odd_even_plus n m : **odd** $n \to$ **odd** $m \to$ **even** $(n + m)$.

Lemma even_plus_aux n m :
 (**odd** $(n + m) \leftrightarrow$ **odd** $n \land$ **even** $m \lor$ **even** $n \land$ **odd** m) $\land$
 (**even** $(n + m) \leftrightarrow$ **even** $n \land$ **even** $m \lor$ **odd** $n \land$ **odd** m).

Lemma even_plus_even_inv_r n m : **even** $(n + m) \to$ **even** $n \to$ **even** m.

Lemma even_plus_even_inv_l n m : **even** $(n + m) \to$ **even** $m \to$ **even** n.

Lemma even_plus_odd_inv_r n m : **even** $(n + m) \to$ **odd** $n \to$ **odd** m.

Lemma even_plus_odd_inv_l n m : **even** $(n + m) \to$ **odd** $m \to$ **odd** n.

Lemma odd_plus_even_inv_l n m : **odd** $(n + m) \to$ **odd** $m \to$ **even** n.

Lemma odd_plus_even_inv_r n m : **odd** $(n + m) \to$ **odd** $n \to$ **even** m.

Lemma odd_plus_odd_inv_l n m : **odd** $(n + m) \to$ **even** $m \to$ **odd** n.

Lemma odd_plus_odd_inv_r n m : **odd** $(n + m) \to$ **even** $n \to$ **odd** m.

20.4 Facts about *even* and *odd* wrt. *mult*

Lemma even_mult_aux n m :
 (**odd** $(n \times m) \leftrightarrow$ **odd** $n \land$ **odd** m) $\land$ (**even** $(n \times m) \leftrightarrow$ **even** $n \lor$ **even** m).

Lemma even_mult_l n m : **even** $n \to$ **even** $(n \times m)$.

Lemma even_mult_r n m : **even** $m \to$ **even** $(n \times m)$.

Lemma even_mult_inv_r n m : **even** $(n \times m) \to$ **odd** $n \to$ **even** m.

Lemma even_mult_inv_l n m : **even** $(n \times m) \to$ **odd** $m \to$ **even** n.

Lemma odd_mult n m : **odd** $n \to$ **odd** $m \to$ **odd** $(n \times m)$.

Lemma odd_mult_inv_l $n\ m$: **odd** $(n \times m) \rightarrow$ **odd** n.

Lemma odd_mult_inv_r $n\ m$: **odd** $(n \times m) \rightarrow$ **odd** m.

#[*global*]
Hint Resolve
 even_even_plus odd_even_plus odd_plus_l odd_plus_r
 even_mult_l even_mult_r even_mult_l even_mult_r odd_mult : *arith*.

Chapter 21

Library Coq.Arith.Gt

Theorems about *gt* in *nat*.

This file is DEPRECATED now, see module *PeanoNat.Nat* instead, which favor *lt* over *gt*.

gt is defined in *Init/Peano.v* as:

```
Definition gt (n m:nat) := m < n.
```

Require Import PeanoNat Le Lt Plus.
Local Open Scope *nat_scope*.

21.1 Order and successor

Theorem gt_Sn_O n : S n > 0.

Theorem gt_Sn_n n : S n > n.

Theorem gt_n_S n m : n > m $\rightarrow$ S n > S m.

Lemma gt_S_n n m : S m > S n $\rightarrow$ m > n.

Theorem gt_S n m : S n > m $\rightarrow$ n > m $\vee$ m = n.

Lemma gt_pred n m : m > S n $\rightarrow$ pred m > n.

21.2 Irreflexivity

Lemma gt_irrefl n : $\neg$ n > n.

21.3 Asymmetry

Lemma gt_asym n m : n > m $\rightarrow$ $\neg$ m > n.

21.4 Relating strict and large orders

Lemma le_not_gt $n\ m : n \leq m \rightarrow \neg\ n > m$.

Lemma gt_not_le $n\ m : n > m \rightarrow \neg\ n \leq m$.

Theorem le_S_gt $n\ m$: $\mathsf{S}\ n \leq m \rightarrow m > n$.

Lemma gt_S_le $n\ m$: $\mathsf{S}\ m > n \rightarrow n \leq m$.

Lemma gt_le_S $n\ m$: $m > n \rightarrow \mathsf{S}\ n \leq m$.

Lemma le_gt_S $n\ m$: $n \leq m \rightarrow \mathsf{S}\ m > n$.

21.5 Transitivity

Theorem le_gt_trans $n\ m\ p : m \leq n \rightarrow m > p \rightarrow n > p$.

Theorem gt_le_trans $n\ m\ p : n > m \rightarrow p \leq m \rightarrow n > p$.

Lemma gt_trans $n\ m\ p : n > m \rightarrow m > p \rightarrow n > p$.

Theorem gt_trans_S $n\ m\ p$: $\mathsf{S}\ n > m \rightarrow m > p \rightarrow n > p$.

21.6 Comparison to 0

Theorem gt_0_eq $n : n > 0 \vee 0 = n$.

21.7 Simplification and compatibility

Lemma plus_gt_reg_l $n\ m\ p : p + n > p + m \rightarrow n > m$.

Lemma plus_gt_compat_l $n\ m\ p : n > m \rightarrow p + n > p + m$.

21.8 Hints

#[*global*]
Hint Resolve *gt_Sn_O gt_Sn_n gt_n_S* : *arith*.
#[*global*]
Hint Immediate *gt_S_n gt_pred* : *arith*.
#[*global*]
Hint Resolve *gt_irrefl gt_asym* : *arith*.
#[*global*]
Hint Resolve *le_not_gt gt_not_le* : *arith*.
#[*global*]
Hint Immediate *le_S_gt gt_S_le* : *arith*.
#[*global*]
Hint Resolve *gt_le_S le_gt_S* : *arith*.
#[*global*]

```
Hint Resolve gt_trans_S le_gt_trans gt_le_trans: arith.
#[global]
Hint Resolve plus_gt_compat_l: arith.
```

Chapter 22

Library Coq.Arith.Minus

Properties of subtraction between natural numbers.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.

minus is now an alias for *Nat.sub*, which is defined in *Init/Nat.v* as:

```
Fixpoint sub (n m:nat) : nat :=
  match n, m with
  | S k, S l => k - l
  | _, _ => n
  end
where "n - m" := (sub n m) : nat_scope.
```

Require Import PeanoNat Lt Le.

Local Open Scope *nat_scope*.

22.1 0 is right neutral

Lemma minus_n_O n : n = n - 0.

22.2 Permutation with successor

Lemma minus_Sn_m n m : $m \leq n \rightarrow$ S (n - m) = S n - m.

Theorem pred_of_minus n : pred n = n - 1.

22.3 Diagonal

Notation minus_diag := Nat.sub_diag (*only parsing*).

Lemma minus_diag_reverse n : 0 = n - n.

Notation minus_n_n := minus_diag_reverse.

22.4 Simplification

Lemma minus_plus_simpl_l_reverse n m p : n - m = p + n - (p + m).

22.5 Relation with plus

Lemma plus_minus n m p : n = m + p $\rightarrow$ p = n - m.

Lemma minus_plus n m : n + m - n = m.

Lemma le_plus_minus_r n m : $n \leq m \rightarrow n$ + (m - n) = m.

Lemma le_plus_minus n m : $n \leq m \rightarrow m$ = n + (m - n).

22.6 Relation with order

Notation minus_le_compat_r :=
Nat.sub_le_mono_r (*only parsing*).

Notation minus_le_compat_l :=
Nat.sub_le_mono_l (*only parsing*).

Notation le_minus := Nat.le_sub_l (*only parsing*). Notation lt_minus := Nat.sub_lt (*only parsing*).

Lemma lt_O_minus_lt n m : $0 < n$ - $m \rightarrow m < n$.

Theorem not_le_minus_0 n m : $\neg\, m \leq n \rightarrow n$ - m = 0.

22.7 Hints

#[*global*]
Hint Resolve *minus_n_O*: *arith*.
#[*global*]
Hint Resolve *minus_Sn_m*: *arith*.
#[*global*]
Hint Resolve *minus_diag_reverse*: *arith*.
#[*global*]
Hint Resolve *minus_plus_simpl_l_reverse*: *arith*.
#[*global*]
Hint Immediate *plus_minus*: *arith*.
#[*global*]
Hint Resolve *minus_plus*: *arith*.
#[*global*]
Hint Resolve *le_plus_minus*: *arith*.
#[*global*]
Hint Resolve *le_plus_minus_r*: *arith*.
#[*global*]
Hint Resolve *lt_minus*: *arith*.

#[global]
`Hint Immediate` *lt_O_minus_lt*: *arith*.

Chapter 23

Library Coq.Arith.Factorial

```
Require Import PeanoNat Plus Mult Lt.
Local Open Scope nat_scope.
```

Factorial

```
Fixpoint fact (n:nat) : nat :=
  match n with
    | O ⇒ 1
    | S n ⇒ S n × fact n
  end.
```

```
Lemma lt_O_fact n : 0 < fact n.
```

```
Lemma fact_neq_0 n : fact n ≠ 0.
```

```
Lemma fact_le n m : n ≤ m → fact n ≤ fact m.
```

Chapter 24

Library Coq.Arith.Max

THIS FILE IS DEPRECATED. Use *PeanoNat.Nat* instead.

```
Require Import PeanoNat.

Local Open Scope nat_scope.
Implicit Types m n p : nat.

Notation max := Nat.max (only parsing).

Definition max_0_l := Nat.max_0_l.
Definition max_0_r := Nat.max_0_r.
Definition succ_max_distr := Nat.succ_max_distr.
Definition plus_max_distr_l := Nat.add_max_distr_l.
Definition plus_max_distr_r := Nat.add_max_distr_r.
Definition max_case_strong := Nat.max_case_strong.
Definition max_spec := Nat.max_spec.
Definition max_dec := Nat.max_dec.
Definition max_case := Nat.max_case.
Definition max_idempotent := Nat.max_id.
Definition max_assoc := Nat.max_assoc.
Definition max_comm := Nat.max_comm.
Definition max_l := Nat.max_l.
Definition max_r := Nat.max_r.
Definition le_max_l := Nat.le_max_l.
Definition le_max_r := Nat.le_max_r.
Definition max_lub_l := Nat.max_lub_l.
Definition max_lub_r := Nat.max_lub_r.
Definition max_lub := Nat.max_lub.

#[global]
Hint Resolve
  Nat.max_l Nat.max_r Nat.le_max_l Nat.le_max_r : arith.

#[global]
Hint Resolve
  Nat.min_l Nat.min_r Nat.le_min_l Nat.le_min_r : arith.
```

Chapter 25

Library Coq.Arith.PeanoNat

Require Import NAxioms NProperties OrdersFacts.

Implementation of *NAxiomsSig* by *nat*

Module Nat
 <: NAxiomsSig
 <: UsualDecidableTypeFull
 <: OrderedTypeFull
 <: TotalOrder.

Operations over *nat* are defined in a separate module

Include Coq.Init.Nat.

When including property functors, inline t eq zero one two lt le succ

All operations are well-defined (trivial here since eq is Leibniz)

```
Definition eq_equiv : Equivalence (@eq nat) := eq_equivalence.
Program Instance succ_wd : Proper (eq==>eq) S.
Program Instance pred_wd : Proper (eq==>eq) pred.
Program Instance add_wd : Proper (eq==>eq==>eq) plus.
Program Instance sub_wd : Proper (eq==>eq==>eq) minus.
Program Instance mul_wd : Proper (eq==>eq==>eq) mult.
Program Instance pow_wd : Proper (eq==>eq==>eq) pow.
Program Instance div_wd : Proper (eq==>eq==>eq) div.
Program Instance mod_wd : Proper (eq==>eq==>eq) modulo.
Program Instance lt_wd : Proper (eq==>eq==>iff) lt.
Program Instance testbit_wd : Proper (eq==>eq==>eq) testbit.
```

Bi-directional induction.

```
Theorem bi_induction :
  ∀ A : nat → Prop, Proper (eq==>iff) A →
    A 0 → (∀ n : nat, A n ↔ A (S n)) → ∀ n : nat, A n.
```

Recursion function

```
Definition recursion {A} : A → (nat → A → A) → nat → A :=
```

```
  nat_rect (fun _ ⇒ A).
Instance recursion_wd {A} (Aeq : relation A) :
 Proper (Aeq ==> (eq==>Aeq==>Aeq) ==> eq ==> Aeq) recursion.
Theorem recursion_0 :
  ∀ {A} (a : A) (f : nat → A → A), recursion a f 0 = a.
Theorem recursion_succ :
  ∀ {A} (Aeq : relation A) (a : A) (f : nat → A → A),
    Aeq a a → Proper (eq==>Aeq==>Aeq) f →
      ∀ n : nat, Aeq (recursion a f (S n)) (f n (recursion a f n)).
```

25.0.1 Remaining constants not defined in Coq.Init.Nat

NB: Aliasing *le* is mandatory, since only a Definition can implement an interface Parameter...

```
Definition eq := @Logic.eq nat.
Definition le := Peano.le.
Definition lt := Peano.lt.
```

25.0.2 Basic specifications : pred add sub mul

```
Lemma pred_succ n : pred (S n) = n.
Lemma pred_0 : pred 0 = 0.
Lemma one_succ : 1 = S 0.
Lemma two_succ : 2 = S 1.
Lemma add_0_l n : 0 + n = n.
Lemma add_succ_l n m : (S n) + m = S (n + m).
Lemma sub_0_r n : n - 0 = n.
Lemma sub_succ_r n m : n - (S m) = pred (n - m).
Lemma mul_0_l n : 0 × n = 0.
Lemma mul_succ_l n m : S n × m = n × m + m.
Lemma lt_succ_r n m : n < S m ↔ n ≤ m.
```

25.0.3 Boolean comparisons

```
Lemma eqb_eq n m : eqb n m = true ↔ n = m.
Lemma leb_le n m : (n <=? m) = true ↔ n ≤ m.
Lemma ltb_lt n m : (n <? m) = true ↔ n < m.
```

25.0.4 Decidability of equality over *nat*.

```
Lemma eq_dec : ∀ n m : nat, {n = m} + {n ≠ m}.
```

25.0.5 Ternary comparison

With *nat*, it would be easier to prove first *compare_spec*, then the properties below. But then we wouldn't be able to benefit from functor *BoolOrderFacts*

Lemma compare_eq_iff n m : (n ?= m) = Eq ↔ n = m.

Lemma compare_lt_iff n m : (n ?= m) = Lt ↔ n < m.

Lemma compare_le_iff n m : (n ?= m) $\neq$ Gt ↔ $n \leq m$.

Lemma compare_antisym n m : (m ?= n) = CompOpp (n ?= m).

Lemma compare_succ n m : (S n ?= S m) = (n ?= m).

25.0.6 Minimum, maximum

Lemma max_l : ∀ n m, $m \leq n$ → max n m = n.

Lemma max_r : ∀ n m, $n \leq m$ → max n m = m.

Lemma min_l : ∀ n m, $n \leq m$ → min n m = n.

Lemma min_r : ∀ n m, $m \leq n$ → min n m = m.

Some more advanced properties of comparison and orders, including *compare_spec* and *lt_irrefl* and *lt_eq_cases*.

Include BoolOrderFacts.

We can now derive all properties of basic functions and orders, and use these properties for proving the specs of more advanced functions.

Include NBasicProp <+ UsualMinMaxLogicalProperties <+ UsualMinMaxDecProperties.

25.0.7 Power

Lemma pow_neg_r a b : b<0 → a^b = 0.

Lemma pow_0_r a : a^0 = 1.

Lemma pow_succ_r a b : 0≤b → a^(S b) = a × a^b.

25.0.8 Square

Lemma square_spec n : square n = n × n.

25.0.9 Parity

Definition Even n := ∃ m, n = 2×m.
Definition Odd n := ∃ m, n = 2×m+1.

Module Private_Parity.

Lemma Even_1 : ¬ Even 1.

Lemma Even_2 n : Even n ↔ Even (S (S n)).

Lemma Odd_0 : ¬ Odd 0.

Lemma Odd_2 n : Odd n ↔ Odd (S (S n)).

End PRIVATE_PARITY.
Import *Private_Parity*.

Lemma even_spec : ∀ n, even n = true ↔ Even n.

Lemma odd_spec : ∀ n, odd n = true ↔ Odd n.

25.0.10 Division

Lemma divmod_spec : ∀ x y q u, $u \leq y$ →
 let (q',u') := divmod x y q u in
 x + (S y)*q + (y-u) = (S y)*q' + (y-u') ∧ u' ≤ y.

Lemma div_mod x y : $y \neq 0$ → x = y*(x/y) + x mod y.

Lemma mod_bound_pos x y : $0 \leq x$ → 0<y → 0 ≤ x mod y < y.

25.0.11 Square root

Lemma sqrt_iter_spec : ∀ k p q r,
 q = p+p → $r \leq q$ →
 let s := sqrt_iter k p q r in
 $s \times s \leq k$ + $p \times p$ + (q - r) < (S s)*(S s).

Lemma sqrt_specif n : (sqrt n)*(sqrt n) ≤ n < S (sqrt n) × S (sqrt n).

Definition sqrt_spec a (Ha:$0 \leq a$) := sqrt_specif a.

Lemma sqrt_neg a : a<0 → sqrt a = 0.

25.0.12 Logarithm

Lemma log2_iter_spec : ∀ k p q r,
 2^(S p) = q + S r → r < 2^p →
 let s := log2_iter k p q r in
 2^s ≤ k + q < 2^(S s).

Lemma log2_spec n : 0<n →
 2^(log2 n) ≤ n < 2^(S (log2 n)).

Lemma log2_nonpos n : $n \leq 0$ → log2 n = 0.

25.0.13 Gcd

Definition divide x y := ∃ z, y=$z \times x$.
Notation "(x | y)" := (divide x y) (at level 0) : *nat_scope*.

Lemma gcd_divide : ∀ a b, (gcd a b | a) ∧ (gcd a b | b).

Lemma gcd_divide_l : ∀ a b, (gcd a b | a).

Lemma gcd_divide_r : ∀ a b, (gcd a b | b).

Lemma gcd_greatest : ∀ a b c, (c|a) → (c|b) → (c|gcd a b).

Lemma gcd_nonneg a b : $0 \leq$gcd a b.

25.0.14 Bitwise operations

Lemma div2_double n : div2 ($2\times n$) = n.

Lemma div2_succ_double n : div2 (S ($2\times n$)) = n.

Lemma le_div2 n : div2 (S n) $\leq n$.

Lemma lt_div2 n : $0 < n$ → div2 $n < n$.

Lemma div2_decr a n : $a \leq$ S n → div2 $a \leq n$.

Lemma double_twice : ∀ n, double n = $2\times n$.

Lemma testbit_0_l : ∀ n, testbit 0 n = false.

Lemma testbit_odd_0 a : testbit ($2\times a$+1) 0 = true.

Lemma testbit_even_0 a : testbit ($2\times a$) 0 = false.

Lemma testbit_odd_succ' a n : testbit ($2\times a$+1) (S n) = testbit a n.

Lemma testbit_even_succ' a n : testbit ($2\times a$) (S n) = testbit a n.

Lemma shiftr_specif : ∀ a n m,
 testbit (shiftr a n) m = testbit a (m+n).

Lemma shiftl_specif_high : ∀ a n m, $n \leq m$ →
 testbit (shiftl a n) m = testbit a (m-n).

Lemma shiftl_spec_low : ∀ a n m, $m < n$ →
 testbit (shiftl a n) m = false.

Lemma div2_bitwise : ∀ op n a b,
 div2 (bitwise op (S n) a b) = bitwise op n (div2 a) (div2 b).

Lemma odd_bitwise : ∀ op n a b,
 odd (bitwise op (S n) a b) = op (odd a) (odd b).

Lemma testbit_bitwise_1 : ∀ op, (∀ b, op false b = false) →
 ∀ n m a b, $a \leq n$ →
 testbit (bitwise op n a b) m = op (testbit a m) (testbit b m).

Lemma testbit_bitwise_2 : ∀ op, op false false = false →
 ∀ n m a b, $a \leq n$ → $b \leq n$ →
 testbit (bitwise op n a b) m = op (testbit a m) (testbit b m).

Lemma land_spec a b n :
 testbit (land a b) n = testbit a n && testbit b n.

Lemma ldiff_spec a b n :
 testbit (ldiff a b) n = testbit a n && negb (testbit b n).

Lemma lor_spec a b n :
 testbit (lor a b) n = testbit a n || testbit b n.

```
Lemma lxor_spec a b n :
 testbit (lxor a b) n = xorb (testbit a n) (testbit b n).
Lemma div2_spec a : div2 a = shiftr a 1.
```

Aliases with extra dummy hypothesis, to fulfil the interface

```
Definition testbit_odd_succ a n (_:0≤n) := testbit_odd_succ' a n.
Definition testbit_even_succ a n (_:0≤n) := testbit_even_succ' a n.
Lemma testbit_neg_r a n (H:n<0) : testbit a n = false.
Definition shiftl_spec_high a n m (_:0≤m) := shiftl_specif_high a n m.
Definition shiftr_spec a n m (_:0≤m) := shiftr_specif a n m.
```

Properties of advanced functions (pow, sqrt, log2, ...)

```
Include NExtraProp.
```

Properties of tail-recursive addition and multiplication

```
Lemma tail_add_spec n m : tail_add n m = n + m.
Lemma tail_addmul_spec r n m : tail_addmul r n m = r + n × m.
Lemma tail_mul_spec n m : tail_mul n m = n × m.
End Nat.
```

Re-export notations that should be available even when the *Nat* module is not imported.

```
Infix "^" := Nat.pow : nat_scope.
Infix "=?" := Nat.eqb (at level 70) : nat_scope.
Infix "<=?" := Nat.leb (at level 70) : nat_scope.
Infix "<?" := Nat.ltb (at level 70) : nat_scope.
Infix "?=" := Nat.compare (at level 70) : nat_scope.
Infix "/" := Nat.div : nat_scope.
Infix "mod" := Nat.modulo (at level 40, no associativity) : nat_scope.
#[global]
Hint Unfold Nat.le : core.
#[global]
Hint Unfold Nat.lt : core.
```

Nat contains an *order* tactic for natural numbers

Note that *Nat.order* is domain-agnostic: it will not prove 1<=2 or $x \leq x+x$, but rather things like $x \leq y \to y \leq x \to x=y$.

```
Section TestOrder.
 Let test : ∀ x y, x≤y → y≤x → x=y.
End TestOrder.
```

Chapter 26

Library Coq.Arith.Plus

Properties of addition.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.

Nat.add is defined in *Init/Nat.v* as:

```
Fixpoint add (n m:nat) : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end
where "n + m" := (add n m) : nat_scope.
```

Require Import PeanoNat.

Local Open Scope *nat_scope*.

26.1 Neutrality of 0, commutativity, associativity

Notation plus_0_l := Nat.add_0_l (*only parsing*).
Notation plus_0_r := Nat.add_0_r (*only parsing*).
Notation plus_comm := Nat.add_comm (*only parsing*).
Notation plus_assoc := Nat.add_assoc (*only parsing*).

Notation plus_permute := Nat.add_shuffle3 (*only parsing*).

Definition plus_Snm_nSm : $\forall$ n m, S n + m = n + S m :=
 Peano.plus_n_Sm.

Lemma plus_assoc_reverse n m p : n + m + p = n + (m + p).

26.2 Simplification

Lemma plus_reg_l n m p : p + n = p + m $\rightarrow$ n = m.

Lemma plus_le_reg_l n m p : p + n $\leq$ p + m $\rightarrow$ n $\leq$ m.

Lemma plus_lt_reg_l n m p : p + n < p + m $\rightarrow$ n < m.

26.3 Compatibility with order

Lemma plus_le_compat_l $n\ m\ p : n \le m \to p + n \le p + m$.

Lemma plus_le_compat_r $n\ m\ p : n \le m \to n + p \le m + p$.

Lemma plus_lt_compat_l $n\ m\ p : n < m \to p + n < p + m$.

Lemma plus_lt_compat_r $n\ m\ p : n < m \to n + p < m + p$.

Lemma plus_le_compat $n\ m\ p\ q : n \le m \to p \le q \to n + p \le m + q$.

Lemma plus_le_lt_compat $n\ m\ p\ q : n \le m \to p < q \to n + p < m + q$.

Lemma plus_lt_le_compat $n\ m\ p\ q : n < m \to p \le q \to n + p < m + q$.

Lemma plus_lt_compat $n\ m\ p\ q : n < m \to p < q \to n + p < m + q$.

Lemma le_plus_l $n\ m : n \le n + m$.

Lemma le_plus_r $n\ m : m \le n + m$.

Theorem le_plus_trans $n\ m\ p : n \le m \to n \le m + p$.

Theorem lt_plus_trans $n\ m\ p : n < m \to n < m + p$.

26.4 Inversion lemmas

Lemma plus_is_O $n\ m : n + m = 0 \to n = 0 \land m = 0$.

Definition plus_is_one $m\ n$:
  $m + n = 1 \to \{m = 0 \land n = 1\} + \{m = 1 \land n = 0\}$.

26.5 Derived properties

Notation plus_permute_2_in_4 := Nat.add_shuffle1 (*only parsing*).

26.6 Tail-recursive plus

tail_plus is an alternative definition for *plus* which is tail-recursive, whereas *plus* is not. This can be useful when extracting programs.

```
Fixpoint tail_plus n m : nat :=
  match n with
    | O ⇒ m
    | S n ⇒ tail_plus n (S m)
  end.
```

Lemma plus_tail_plus : $\forall\ n\ m,\ n + m =$ tail_plus $n\ m$.

26.7 Discrimination

Lemma succ_plus_discr n m : $n \neq$ S (m+n).

Lemma n_SSn n : $n \neq$ S (S n).

Lemma n_SSSn n : $n \neq$ S (S (S n)).

Lemma n_SSSSn n : $n \neq$ S (S (S (S n))).

26.8 Compatibility Hints

```
#[global]
Hint Immediate plus_comm : arith.
#[global]
Hint Resolve plus_assoc plus_assoc_reverse : arith.
#[global]
Hint Resolve plus_le_compat_l plus_le_compat_r : arith.
#[global]
Hint Resolve le_plus_l le_plus_r le_plus_trans : arith.
#[global]
Hint Immediate lt_plus_trans : arith.
#[global]
Hint Resolve plus_lt_compat_l plus_lt_compat_r : arith.
```

For compatibility, we "Require" the same files as before

```
Require Import Le Lt.
```

Chapter 27

Library Coq.Arith.Wf_nat

Well-founded relations and natural numbers

```
Require Import PeanoNat Lt.

Local Open Scope nat_scope.

Implicit Types m n p : nat.

Section Well_founded_Nat.

Variable A : Type.

Variable f : A → nat.
Definition ltof (a b:A) := f a < f b.
Definition gtof (a b:A) := f b > f a.

Theorem well_founded_ltof : well_founded ltof.

Theorem well_founded_gtof : well_founded gtof.
```

It is possible to directly prove the induction principle going back to primitive recursion on natural numbers (*induction_ltof1*) or to use the previous lemmas to extract a program with a fixpoint (*induction_ltof2*)

the ML-like program for *induction_ltof1* is :

```
let induction_ltof1 f F a =
  let rec indrec n k =
    match n with
    | O → error
    | S m → F k (indrec m)
  in indrec (f a + 1) a
```

the ML-like program for *induction_ltof2* is :

```
let induction_ltof2 F a = indrec a
where rec indrec a = F a indrec;;
```

```
Theorem induction_ltof1 :
  ∀ P:A → Type,
    (∀ x:A, (∀ y:A, ltof y x → P y) → P x) → ∀ a:A, P a.
```

Theorem induction_gtof1 :
  $\forall$ P:$A \rightarrow$ Type,
    ($\forall$ x:A, ($\forall$ y:A, gtof y $x \rightarrow P$ y) $\rightarrow P$ x) $\rightarrow \forall$ a:A, P a.

Theorem induction_ltof2 :
  $\forall$ P:$A \rightarrow$ Type,
    ($\forall$ x:A, ($\forall$ y:A, ltof y $x \rightarrow P$ y) $\rightarrow P$ x) $\rightarrow \forall$ a:A, P a.

Theorem induction_gtof2 :
  $\forall$ P:$A \rightarrow$ Type,
    ($\forall$ x:A, ($\forall$ y:A, gtof y $x \rightarrow P$ y) $\rightarrow P$ x) $\rightarrow \forall$ a:A, P a.

If a relation R is compatible with lt i.e. if $x\ R\ y \Rightarrow f(x) < f(y)$ then R is well-founded.

Variable R : $A \rightarrow A \rightarrow$ Prop.

Hypothesis H_compat : $\forall$ x y:A, R x y $\rightarrow f$ x < f y.

Theorem well_founded_lt_compat : well_founded R.

End Well_founded_Nat.

Lemma lt_wf : well_founded lt.

Lemma lt_wf_rect1 :
  $\forall$ n (P:**nat** $\rightarrow$ Type), ($\forall$ n, ($\forall$ m, m < $n \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma lt_wf_rect :
  $\forall$ n (P:**nat** $\rightarrow$ Type), ($\forall$ n, ($\forall$ m, m < $n \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma lt_wf_rec1 :
  $\forall$ n (P:**nat** $\rightarrow$ Set), ($\forall$ n, ($\forall$ m, m < $n \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma lt_wf_rec :
  $\forall$ n (P:**nat** $\rightarrow$ Set), ($\forall$ n, ($\forall$ m, m < $n \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma lt_wf_ind :
  $\forall$ n (P:**nat** $\rightarrow$ Prop), ($\forall$ n, ($\forall$ m, m < $n \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma gt_wf_rect :
  $\forall$ n (P:**nat** $\rightarrow$ Type), ($\forall$ n, ($\forall$ m, n > $m \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma gt_wf_rec :
  $\forall$ n (P:**nat** $\rightarrow$ Set), ($\forall$ n, ($\forall$ m, n > $m \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma gt_wf_ind :
  $\forall$ n (P:**nat** $\rightarrow$ Prop), ($\forall$ n, ($\forall$ m, n > $m \rightarrow P$ m) $\rightarrow P$ n) $\rightarrow P$ n.

Lemma lt_wf_double_rect :
 $\forall$ P:**nat** $\rightarrow$ **nat** $\rightarrow$ Type,
   ($\forall$ n m,
     ($\forall$ p q, p < $n \rightarrow P$ p q) $\rightarrow$
     ($\forall$ p, p < $m \rightarrow P$ n p) $\rightarrow P$ n m) $\rightarrow \forall$ n m, P n m.

Lemma lt_wf_double_rec :
 $\forall$ P:**nat** $\rightarrow$ **nat** $\rightarrow$ Set,
   ($\forall$ n m,
     ($\forall$ p q, p < $n \rightarrow P$ p q) $\rightarrow$

```
    (∀ p, p < m → P n p) → P n m) → ∀ n m, P n m.
Lemma lt_wf_double_ind :
  ∀ P:nat → nat → Prop,
    (∀ n m,
      (∀ p (q:nat), p < n → P p q) →
      (∀ p, p < m → P n p) → P n m) → ∀ n m, P n m.
#[global]
Hint Resolve lt_wf: arith.
#[global]
Hint Resolve well_founded_lt_compat: arith.
Section LT_WF_REL.
  Variable A : Set.
  Variable R : A → A → Prop.
  Variable F : A → nat → Prop.
  Definition inv_lt_rel x y := exists2 n, F x n & (∀ m, F y m → n < m).
  Hypothesis F_compat : ∀ x y:A, R x y → inv_lt_rel x y.
  Remark acc_lt_rel : ∀ x:A, (∃ n, F x n) → Acc R x.
  Theorem well_founded_inv_lt_rel_compat : well_founded R.
End LT_WF_REL.
Lemma well_founded_inv_rel_inv_lt_rel (A:Set) (F:A → nat → Prop) :
  well_founded (inv_lt_rel A F).
```

A constructive proof that any non empty decidable subset of natural numbers has a least element

```
Set Implicit Arguments.
Require Import Le.
Require Import Compare_dec.
Require Import Decidable.
Definition has_unique_least_element (A:Type) (R:A→A→Prop) (P:A→Prop) :=
  ∃! x, P x ∧ ∀ x', P x' → R x x'.
Lemma dec_inh_nat_subset_has_unique_least_element :
  ∀ P:nat→Prop, (∀ n, P n ∨ ¬ P n) →
    (∃ n, P n) → has_unique_least_element le P.
Unset Implicit Arguments.
Notation iter_nat n A f x := (nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) n) (only parsing).
```

Chapter 28

Library Coq.Arith.Min

THIS FILE IS DEPRECATED. Use *PeanoNat.Nat* instead.

```
Require Import PeanoNat.

Local Open Scope nat_scope.
Implicit Types m n p : nat.

Notation min := Nat.min (only parsing).

Definition min_0_l := Nat.min_0_l.
Definition min_0_r := Nat.min_0_r.
Definition succ_min_distr := Nat.succ_min_distr.
Definition plus_min_distr_l := Nat.add_min_distr_l.
Definition plus_min_distr_r := Nat.add_min_distr_r.
Definition min_case_strong := Nat.min_case_strong.
Definition min_spec := Nat.min_spec.
Definition min_dec := Nat.min_dec.
Definition min_case := Nat.min_case.
Definition min_idempotent := Nat.min_id.
Definition min_assoc := Nat.min_assoc.
Definition min_comm := Nat.min_comm.
Definition min_l := Nat.min_l.
Definition min_r := Nat.min_r.
Definition le_min_l := Nat.le_min_l.
Definition le_min_r := Nat.le_min_r.
Definition min_glb_l := Nat.min_glb_l.
Definition min_glb_r := Nat.min_glb_r.
Definition min_glb := Nat.min_glb.
```

Chapter 29

Library Coq.Arith.Lt

Strict order on natural numbers.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.

lt is defined in library *Init/Peano.v* as:

```
Definition lt (n m:nat) := S n <= m.
Infix "&lt;" := lt : nat_scope.
```

Require Import PeanoNat.

Local Open Scope *nat_scope*.

29.1 Irreflexivity

Notation lt_irrefl := Nat.lt_irrefl (*only parsing*).

#[*global*]
Hint Resolve *lt_irrefl*: *arith*.

29.2 Relationship between *le* and *lt*

Theorem lt_le_S $n\ m : n < m \to \mathsf{S}\ n \leq m$.

Theorem lt_n_Sm_le $n\ m : n < \mathsf{S}\ m \to n \leq m$.

Theorem le_lt_n_Sm $n\ m : n \leq m \to n < \mathsf{S}\ m$.

#[*global*]
Hint Immediate *lt_le_S*: *arith*.
#[*global*]
Hint Immediate *lt_n_Sm_le*: *arith*.
#[*global*]
Hint Immediate *le_lt_n_Sm*: *arith*.

Theorem le_not_lt $n\ m : n \leq m \to \neg\ m < n$.

Theorem lt_not_le $n\ m : n < m \to \neg\ m \leq n$.

#[*global*]
Hint Immediate *le_not_lt lt_not_le*: *arith*.

29.3 Asymmetry

Notation lt_asym := Nat.lt_asymm (*only parsing*).

29.4 Order and 0

Notation lt_0_Sn := Nat.lt_0_succ (*only parsing*). Notation lt_n_0 := Nat.nlt_0_r (*only parsing*).

Theorem neq_0_lt n : $0 \neq n \rightarrow 0 < n$.

Theorem lt_0_neq n : $0 < n \rightarrow 0 \neq n$.

#[*global*]
Hint Resolve *lt_0_Sn lt_n_0* : *arith*.
#[*global*]
Hint Immediate *neq_0_lt lt_0_neq*: *arith*.

29.5 Order and successor

Notation lt_n_Sn := Nat.lt_succ_diag_r (*only parsing*). Notation lt_S := Nat.lt_lt_succ_r (*only parsing*).

Theorem lt_n_S n m : $n < m \rightarrow$ S $n <$ S m.

Theorem lt_S_n n m : S $n <$ S $m \rightarrow n < m$.

#[*global*]
Hint Resolve *lt_n_Sn lt_S lt_n_S* : *arith*.
#[*global*]
Hint Immediate *lt_S_n* : *arith*.

29.6 Predecessor

Lemma S_pred n m : $m < n \rightarrow n =$ S (pred n).

Lemma S_pred_pos n: O $< n \rightarrow n =$ S (pred n).

Lemma lt_pred n m : S $n < m \rightarrow n <$ pred m.

Lemma lt_pred_n_n n : $0 < n \rightarrow$ pred $n < n$.

#[*global*]
Hint Immediate *lt_pred*: *arith*.
#[*global*]
Hint Resolve *lt_pred_n_n*: *arith*.

29.7 Transitivity properties

Notation lt_trans := Nat.lt_trans (*only parsing*).
Notation lt_le_trans := Nat.lt_le_trans (*only parsing*).
Notation le_lt_trans := Nat.le_lt_trans (*only parsing*).

#[*global*]
Hint Resolve *lt_trans lt_le_trans le_lt_trans*: *arith*.

29.8 Large = strict or equal

Notation le_lt_or_eq_iff := Nat.lt_eq_cases (*only parsing*).

Theorem le_lt_or_eq n m : $n \leq m \rightarrow n < m \lor n = m$.

Notation lt_le_weak := Nat.lt_le_incl (*only parsing*).

#[*global*]
Hint Immediate *lt_le_weak*: *arith*.

29.9 Dichotomy

Notation le_or_lt := Nat.le_gt_cases (*only parsing*).

Theorem nat_total_order n m : $n \neq m \rightarrow n < m \lor m < n$.

For compatibility, we "Require" the same files as before

Require Import Le.

Chapter 30

Library Coq.Arith.Arith

```
Require Export Arith_base.
Require Export ArithRing.
```

Chapter 31

Library Coq.Arith.Mult

31.1 Properties of multiplication.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.
Nat.mul is defined in *Init/Nat.v*.

Require Import PeanoNat.
For Compatibility: Require Export Plus Minus Le Lt.

Local Open Scope *nat_scope*.

31.2 *nat* is a semi-ring

31.2.1 Zero property

Notation mult_0_l := Nat.mul_0_l (*only parsing*). Notation mult_0_r := Nat.mul_0_r (*only parsing*).

31.2.2 1 is neutral

Notation mult_1_l := Nat.mul_1_l (*only parsing*). Notation mult_1_r := Nat.mul_1_r (*only parsing*).

#[*global*]
Hint Resolve *mult_1_l mult_1_r*: *arith*.

31.2.3 Commutativity

Notation mult_comm := Nat.mul_comm (*only parsing*).

#[*global*]
Hint Resolve *mult_comm*: *arith*.

31.2.4 Distributivity

Notation mult_plus_distr_r :=

Nat.mul_add_distr_r (*only parsing*).

Notation mult_plus_distr_l :=
Nat.mul_add_distr_l (*only parsing*).

Notation mult_minus_distr_r :=
Nat.mul_sub_distr_r (*only parsing*).

Notation mult_minus_distr_l :=
Nat.mul_sub_distr_l (*only parsing*).

#[*global*]
Hint Resolve *mult_plus_distr_r*: *arith*.
#[*global*]
Hint Resolve *mult_minus_distr_r*: *arith*.
#[*global*]
Hint Resolve *mult_minus_distr_l*: *arith*.

31.2.5 Associativity

Notation mult_assoc := Nat.mul_assoc (*only parsing*).

Lemma mult_assoc_reverse $n\ m\ p$: $n \times m \times p = n \times (m \times p)$.

#[*global*]
Hint Resolve *mult_assoc_reverse*: *arith*.
#[*global*]
Hint Resolve *mult_assoc*: *arith*.

31.2.6 Inversion lemmas

Lemma mult_is_O $n\ m$: $n \times m = 0 \rightarrow n = 0 \lor m = 0$.

Lemma mult_is_one $n\ m$: $n \times m = 1 \rightarrow n = 1 \land m = 1$.

31.2.7 Multiplication and successor

Notation mult_succ_l := Nat.mul_succ_l (*only parsing*). Notation mult_succ_r := Nat.mul_succ_r (*only parsing*).

31.3 Compatibility with orders

Lemma mult_O_le $n\ m$: $m = 0 \lor n \leq m \times n$.
#[*global*]
Hint Resolve *mult_O_le*: *arith*.

Lemma mult_le_compat_l $n\ m\ p$: $n \leq m \rightarrow p \times n \leq p \times m$.
#[*global*]
Hint Resolve *mult_le_compat_l*: *arith*.

Lemma mult_le_compat_r $n\ m\ p$: $n \leq m \rightarrow n \times p \leq m \times p$.

Lemma mult_le_compat $n\ m\ p\ q : n \leq m \to p \leq q \to n \times p \leq m \times q$.

Lemma mult_S_lt_compat_l $n\ m\ p : m < p \to S\ n \times m < S\ n \times p$.

#[*global*]
Hint Resolve *mult_S_lt_compat_l*: *arith*.

Lemma mult_lt_compat_l $n\ m\ p : n < m \to 0 < p \to p \times n < p \times m$.

Lemma mult_lt_compat_r $n\ m\ p : n < m \to 0 < p \to n \times p < m \times p$.

Lemma mult_S_le_reg_l $n\ m\ p : S\ n \times m \leq S\ n \times p \to m \leq p$.

31.4 n|->2*n and n|->2n+1 have disjoint image

Theorem odd_even_lem $p\ q : 2 \times p + 1 \neq 2 \times q$.

31.5 Tail-recursive mult

tail_mult is an alternative definition for *mult* which is tail-recursive, whereas *mult* is not. This can be useful when extracting programs.

```
Fixpoint mult_acc (s:nat) m n : nat :=
  match n with
    | O ⇒ s
    | S p ⇒ mult_acc (tail_plus m s) m p
  end.
```

Lemma mult_acc_aux : ∀ $n\ m\ p$, $m + n \times p$ = mult_acc $m\ p\ n$.

Definition tail_mult $n\ m$:= mult_acc $0\ m\ n$.

Lemma mult_tail_mult : ∀ $n\ m$, $n \times m$ = tail_mult $n\ m$.

TailSimpl transforms any *tail_plus* and *tail_mult* into *plus* and *mult* and simplify

```
Ltac tail_simpl :=
  repeat rewrite ← plus_tail_plus; repeat rewrite ← mult_tail_mult;
    simpl.
```

Chapter 32

Library Coq.Arith.Euclid

```
Require Import Mult.
Require Import Compare_dec.
Require Import Wf_nat.

Local Open Scope nat_scope.

Implicit Types a b n q r : nat.

Inductive diveucl a b : Set :=
  divex : ∀ q r, b > r → a = q × b + r → diveucl a b.

Lemma eucl_dev : ∀ n, n > 0 → ∀ m:nat, diveucl m n.

Lemma quotient :
  ∀ n,
    n > 0 →
    ∀ m:nat, {q : nat | ∃ r : nat, m = q × n + r ∧ n > r}.

Lemma modulo :
  ∀ n,
    n > 0 →
    ∀ m:nat, {r : nat | ∃ q : nat, m = q × n + r ∧ n > r}.
```

Chapter 33

Library Coq.Arith.Arith_base

```
Require Export PeanoNat.
Require Export Le.
Require Export Lt.
Require Export Plus.
Require Export Gt.
Require Export Minus.
Require Export Mult.
Require Export Between.
Require Export Peano_dec.
Require Export Compare_dec.
Require Export Factorial.
Require Export EqNat.
Require Export Wf_nat.
```

Chapter 34

Library Coq.Arith.Compare_dec

Require Import Le Lt Gt Decidable PeanoNat.

Local Open Scope *nat_scope*.

Implicit Types m n x y : **nat**.

Definition zerop n : {n = 0} + {0 < n}.

Definition lt_eq_lt_dec n m : {n < m} + {n = m} + {m < n}.

Definition gt_eq_gt_dec n m : {m > n} + {n = m} + {n > m}.

Definition le_lt_dec n m : {$n \leq m$} + {m < n}.

Definition le_le_S_dec n m : {$n \leq m$} + {S $m \leq n$}.

Definition le_ge_dec n m : {$n \leq m$} + {$n \geq m$}.

Definition le_gt_dec n m : {$n \leq m$} + {n > m}.

Definition le_lt_eq_dec n m : $n \leq m \rightarrow$ {n < m} + {n = m}.

Theorem le_dec n m : {$n \leq m$} + {$\neg$ $n \leq m$}.

Theorem lt_dec n m : {n < m} + {$\neg$ n < m}.

Theorem gt_dec n m : {n > m} + {$\neg$ n > m}.

Theorem ge_dec n m : {$n \geq m$} + {$\neg$ $n \geq m$}.

Proofs of decidability

Theorem dec_le n m : decidable ($n \leq m$).

Theorem dec_lt n m : decidable (n < m).

Theorem dec_gt n m : decidable (n > m).

Theorem dec_ge n m : decidable ($n \geq m$).

Theorem not_eq n m : $n \neq m \rightarrow n$ < $m \lor m$ < n.

Theorem not_le n m : $\neg$ $n \leq m \rightarrow n$ > m.

Theorem not_gt n m : $\neg$ n > $m \rightarrow n \leq m$.

Theorem not_ge n m : $\neg$ $n \geq m \rightarrow n$ < m.

Theorem not_lt n m : $\neg$ n < $m \rightarrow n \geq m$.

A ternary comparison function in the spirit of *Z.compare*. See now *Nat.compare* and its properties. In scope *nat_scope*, the notation for *Nat.compare* is "?="

```
Notation nat_compare_S := Nat.compare_succ (only parsing).
```

```
Lemma nat_compare_lt n m : n<m ↔ (n ?= m) = Lt.
```

```
Lemma nat_compare_gt n m : n>m ↔ (n ?= m) = Gt.
```

```
Lemma nat_compare_le n m : n≤m ↔ (n ?= m) ≠ Gt.
```

```
Lemma nat_compare_ge n m : n≥m ↔ (n ?= m) ≠ Lt.
```

Some projections of the above equivalences.

```
Lemma nat_compare_eq n m : (n ?= m) = Eq → n = m.
```

```
Lemma nat_compare_Lt_lt n m : (n ?= m) = Lt → n<m.
```

```
Lemma nat_compare_Gt_gt n m : (n ?= m) = Gt → n>m.
```

A previous definition of *nat_compare* in terms of *lt_eq_lt_dec*. The new version avoids the creation of proof parts.

```
Definition nat_compare_alt (n m:nat) :=
  match lt_eq_lt_dec n m with
    | inleft (left _) ⇒ Lt
    | inleft (right _) ⇒ Eq
    | inright _ ⇒ Gt
  end.
```

```
Lemma nat_compare_equiv n m : (n ?= m) = nat_compare_alt n m.
```

A boolean version of *le* over *nat*. See now *Nat.leb* and its properties. In scope *nat_scope*, the notation for *Nat.leb* is "<=?"

```
Notation leb := Nat.leb (only parsing).
```

```
Notation leb_iff := Nat.leb_le (only parsing).
```

```
Lemma leb_iff_conv m n : (n <=? m) = false ↔ m < n.
```

```
Lemma leb_correct m n : m ≤ n → (m <=? n) = true.
```

```
Lemma leb_complete m n : (m <=? n) = true → m ≤ n.
```

```
Lemma leb_correct_conv m n : m < n → (n <=? m) = false.
```

```
Lemma leb_complete_conv m n : (n <=? m) = false → m < n.
```

```
Lemma leb_compare n m : (n <=? m) = true ↔ (n ?= m) ≠ Gt.
```

Chapter 35

Library Coq.Arith.Le

Order on natural numbers.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.

le is defined in *Init/Peano.v* as:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : n <= n
  | le_S : forall m:nat, n <= m -> n <= S m

where "n <= m" := (le n m) : nat_scope.
```

Require Import PeanoNat.

Local Open Scope *nat_scope*.

35.1 *le* is an order on *nat*

Notation le_refl := Nat.le_refl (*only parsing*).
Notation le_trans := Nat.le_trans (*only parsing*).
Notation le_antisym := Nat.le_antisymm (*only parsing*).

#[*global*]
Hint Resolve *le_trans*: *arith*.
#[*global*]
Hint Immediate *le_antisym*: *arith*.

35.2 Properties of *le* w.r.t 0

Notation le_0_n := Nat.le_0_l (*only parsing*). Notation le_Sn_0 := Nat.nle_succ_0 (*only parsing*).

Lemma le_n_0_eq n : $n \leq 0 \rightarrow 0 = n$.

35.3 Properties of *le* w.r.t successor

See also *Nat.succ_le_mono*.

Theorem le_n_S : $\forall$ n m, $n \leq m \rightarrow$ S $n \leq$ S m.

Theorem le_S_n : $\forall$ n m, S $n \leq$ S $m \rightarrow n \leq m$.

Notation le_n_Sn := Nat.le_succ_diag_r (*only parsing*). Notation le_Sn_n := Nat.nle_succ_diag_l (*only parsing*).

Theorem le_Sn_le : $\forall$ n m, S $n \leq m \rightarrow n \leq m$.

#[*global*]
Hint Resolve *le_0_n le_Sn_0*: *arith*.
#[*global*]
Hint Resolve *le_n_S le_n_Sn le_Sn_n* : *arith*.
#[*global*]
Hint Immediate *le_n_0_eq le_Sn_le le_S_n* : *arith*.

35.4 Properties of *le* w.r.t predecessor

Notation le_pred_n := Nat.le_pred_l (*only parsing*). Notation le_pred := Nat.pred_le_mono (*only parsing*).

#[*global*]
Hint Resolve *le_pred_n*: *arith*.

35.5 A different elimination principle for the order on natural numbers

Lemma le_elim_rel :
 $\forall$ P:**nat** $\rightarrow$ **nat** $\rightarrow$ Prop,
   ($\forall$ p, P 0 p) $\rightarrow$
   ($\forall$ p (q:**nat**), $p \leq q \rightarrow P$ p $q \rightarrow P$ (S p) (S q)) $\rightarrow$
   $\forall$ n m, $n \leq m \rightarrow P$ n m.

Chapter 36

Library Coq.Arith.Between

```
Require Import Le.
Require Import Lt.
Local Open Scope nat_scope.
Implicit Types k l p q r : nat.
Section Between.
  Variables P Q : nat → Prop.
```

The *between* type expresses the concept ∀ *i*: *nat*, $k \leq i < l \to P\ i$..

```
Inductive between k : nat → Prop :=
    | bet_emp : between k k
    | bet_S : ∀ l, between k l → P l → between k (S l).
  #[local]
  Hint Constructors between: arith.
  Lemma bet_eq : ∀ k l, l = k → between k l.
  #[local]
  Hint Resolve bet_eq: arith.
  Lemma between_le : ∀ k l, between k l → k ≤ l.
  #[local]
  Hint Immediate between_le: arith.
  Lemma between_Sk_l : ∀ k l, between k l → S k ≤ l → between (S k) l.
  #[local]
  Hint Resolve between_Sk_l: arith.
  Lemma between_restr :
    ∀ k l (m:nat), k ≤ l → l ≤ m → between k m → between l m.
```

The *exists_between* type expresses the concept ∃ *i*: *nat*, $k \leq i < l \land Q\ i$.

```
Inductive exists_between k : nat → Prop :=
    | exists_S : ∀ l, exists_between k l → exists_between k (S l)
    | exists_le : ∀ l, k ≤ l → Q l → exists_between k (S l).
  #[local]
  Hint Constructors exists_between: arith.
```

Lemma exists_le_S : $\forall\ k\ l$, **exists_between** $k\ l \rightarrow$ S $k \leq l$.

Lemma exists_lt : $\forall\ k\ l$, **exists_between** $k\ l \rightarrow k < l$.
#[*local*]
`Hint Immediate` *exists_le_S exists_lt*: *arith*.

Lemma exists_S_le : $\forall\ k\ l$, **exists_between** k (S l) $\rightarrow k \leq l$.
#[*local*]
`Hint Immediate` *exists_S_le*: *arith*.

`Definition` in_int $p\ q\ r := p \leq r \wedge r < q$.

Lemma in_int_intro : $\forall\ p\ q\ r,\ p \leq r \rightarrow r < q \rightarrow$ in_int $p\ q\ r$.
#[*local*]
`Hint Resolve` *in_int_intro*: *arith*.

Lemma in_int_lt : $\forall\ p\ q\ r$, in_int $p\ q\ r \rightarrow p < q$.

Lemma in_int_p_Sq :
  $\forall\ p\ q\ r$, in_int p (S q) $r \rightarrow$ in_int $p\ q\ r \vee r = q$.

Lemma in_int_S : $\forall\ p\ q\ r$, in_int $p\ q\ r \rightarrow$ in_int p (S q) r.
#[*local*]
`Hint Resolve` *in_int_S*: *arith*.

Lemma in_int_Sp_q : $\forall\ p\ q\ r$, in_int (S p) $q\ r \rightarrow$ in_int $p\ q\ r$.
#[*local*]
`Hint Immediate` *in_int_Sp_q*: *arith*.

Lemma between_in_int :
  $\forall\ k\ l$, **between** $k\ l \rightarrow \forall\ r$, in_int $k\ l\ r \rightarrow P\ r$.

Lemma in_int_between :
  $\forall\ k\ l,\ k \leq l \rightarrow (\forall\ r$, in_int $k\ l\ r \rightarrow P\ r) \rightarrow$ **between** $k\ l$.

Lemma exists_in_int :
  $\forall\ k\ l$, **exists_between** $k\ l \rightarrow$ `exists2` m `:` **nat**`,` in_int $k\ l\ m$ & $Q\ m$.

Lemma in_int_exists : $\forall\ k\ l\ r$, in_int $k\ l\ r \rightarrow Q\ r \rightarrow$ **exists_between** $k\ l$.

Lemma between_or_exists :
  $\forall\ k\ l$,
    $k \leq l \rightarrow$
    ($\forall\ n$:**nat**, in_int $k\ l\ n \rightarrow P\ n \vee Q\ n) \rightarrow$
    between $k\ l \vee$ **exists_between** $k\ l$.

Lemma between_not_exists :
  $\forall\ k\ l$,
    between $k\ l \rightarrow$
    ($\forall\ n$:**nat**, in_int $k\ l\ n \rightarrow P\ n \rightarrow \neg\ Q\ n) \rightarrow \neg$ **exists_between** $k\ l$.

`Inductive` **P_nth** (*init*:**nat**) : **nat** $\rightarrow$ **nat** $\rightarrow$ `Prop` :=
  | nth_O : **P_nth** *init init* 0
  | nth_S :
    $\forall\ k\ l\ (n$:**nat**),
      P_nth *init* $k\ n \rightarrow$ **between** (S k) $l \rightarrow Q\ l \rightarrow$ **P_nth** *init* l (S n).

Lemma nth_le : $\forall$ ($init$:**nat**) l (n:**nat**), **P_nth** $init\ l\ n \rightarrow init \leq l$.

Definition eventually (n:**nat**) := exists2 k : **nat**, $k \leq n$ & $Q\ k$.

Lemma event_O : eventually $0 \rightarrow Q\ 0$.

End Between.

#[*global*]
Hint Resolve *nth_O bet_S bet_emp bet_eq between_Sk_l exists_S exists_le in_int_S in_int_intro*: *arith*.
#[*global*]
Hint Immediate *in_int_Sp_q exists_le_S exists_S_le*: *arith*.

Chapter 37

Library Coq.Arith.Bool_nat

```
Require Export Compare_dec.
Require Export Peano_dec.
Require Import Sumbool.
```

```
Local Open Scope nat_scope.
```

```
Implicit Types m n x y : nat.
```

The decidability of equality and order relations over type *nat* give some boolean functions with the adequate specification.

```
Definition notzerop n := sumbool_not _ _ (zerop n).
Definition lt_ge_dec : ∀ x y, {x < y} + {x ≥ y} :=
  fun n m ⇒ sumbool_not _ _ (le_lt_dec m n).
```

```
Definition nat_lt_ge_bool x y := bool_of_sumbool (lt_ge_dec x y).
Definition nat_ge_lt_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (lt_ge_dec x y)).
```

```
Definition nat_le_gt_bool x y := bool_of_sumbool (le_gt_dec x y).
Definition nat_gt_le_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (le_gt_dec x y)).
```

```
Definition nat_eq_bool x y := bool_of_sumbool (eq_nat_dec x y).
Definition nat_noteq_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (eq_nat_dec x y)).
```

```
Definition zerop_bool x := bool_of_sumbool (zerop x).
Definition notzerop_bool x := bool_of_sumbool (notzerop x).
```

Chapter 38

Library Coq.Arith.Div2

Nota : this file is OBSOLETE, and left only for compatibility. Please consider using *Nat.div2* directly, and results about it (see file PeanoNat).

```
Require Import PeanoNat Even.
```

```
Local Open Scope nat_scope.
```

```
Implicit Type n : nat.
```

Here we define $n/2$ and prove some of its properties

```
Notation div2 := Nat.div2 (only parsing).
```

Since *div2* is recursively defined on 0, 1 and $(S\ (S\ n))$, it is useful to prove the corresponding induction principle

```
Lemma ind_0_1_SS :
  ∀ P:nat → Prop,
    P 0 → P 1 → (∀ n, P n → P (S (S n))) → ∀ n, P n.
```

$0 < n \Rightarrow n/2 < n$

```
Lemma lt_div2 n : 0 < n → div2 n < n.
```

```
#[global]
Hint Resolve lt_div2: arith.
```

Properties related to the parity

```
Lemma even_div2 n : even n → div2 n = div2 (S n).
```

```
Lemma odd_div2 n : odd n → S (div2 n) = div2 (S n).
```

```
Lemma div2_even n : div2 n = div2 (S n) → even n.
```

```
Lemma div2_odd n : S (div2 n) = div2 (S n) → odd n.
```

```
#[global]
Hint Resolve even_div2 div2_even odd_div2 div2_odd: arith.
```

```
Lemma even_odd_div2 n :
 (even n ↔ div2 n = div2 (S n)) ∧
 (odd n ↔ S (div2 n) = div2 (S n)).
```

Properties related to the double $(2n)$

```
Notation double := Nat.double (only parsing).
#[global]
Hint Unfold double Nat.double: arith.
Lemma double_S n : double (S n) = S (S (double n)).
Lemma double_plus n m : double (n + m) = double n + double m.
#[global]
Hint Resolve double_S: arith.
Lemma even_odd_double n :
  (even n ↔ n = double (div2 n)) ∧ (odd n ↔ n = S (double (div2 n))).
```

Specializations

```
Lemma even_double n : even n → n = double (div2 n).
Lemma double_even n : n = double (div2 n) → even n.
Lemma odd_double n : odd n → n = S (double (div2 n)).
Lemma double_odd n : n = S (double (div2 n)) → odd n.
#[global]
Hint Resolve even_double double_even odd_double double_odd: arith.
```

Application:

- if n is even then there is a p such that $n = 2p$
- if n is odd then there is a p such that $n = 2p+1$

(Immediate: it is $n/2$)

```
Lemma even_2n : ∀ n, even n → {p : nat | n = double p}.
Lemma odd_S2n : ∀ n, odd n → {p : nat | n = S (double p)}.
```

Doubling before dividing by two brings back to the initial number.

```
Lemma div2_double n : div2 (2×n) = n.
Lemma div2_double_plus_one n : div2 (S (2×n)) = n.
```

Chapter 39

Library Coq.Arith.Compare

Equality is decidable on *nat*

Local Open Scope *nat_scope*.

Notation not_eq_sym := not_eq_sym (*only parsing*).

Implicit Types m n p q : **nat**.

Require Import Arith_base.
Require Import Peano_dec.
Require Import Compare_dec.

Definition le_or_le_S := le_le_S_dec.

Definition Pcompare := gt_eq_gt_dec.

Lemma le_dec : $\forall$ n m, {$n \leq m$} + {$m \leq n$}.

Definition lt_or_eq n m := {m > n} + {n = m}.

Lemma le_decide : $\forall$ n m, $n \leq m \rightarrow$ lt_or_eq n m.

Lemma le_le_S_eq : $\forall$ n m, $n \leq m \rightarrow$ S $n \leq m \lor n = m$.

Lemma discrete_nat :
  $\forall$ n m, $n < m \rightarrow$ S $n = m \lor (\exists\ r$: **nat**, m = S (S ($n + r$))).

Require Export Wf_nat.

Require Export Min Max.

Chapter 40

Library Coq.Unicode.Utf8

Require Export Utf8_core.

Notation "x ≤ y" := (**le** x y) (at level 70, no associativity).
Notation "x ≥ y" := (ge x y) (at level 70, no associativity).

Chapter 41

Library Coq.Unicode.Utf8_core

Notation "∀ x .. y , P" := (∀ *x*, .. (∀ *y*, *P*) ..)
(at level 200, *x binder*, *y binder*, right associativity,
format "'[' '[' ∀ x .. y ']' , '/' P ']'") : *type_scope*.
Notation "∃ x .. y , P" := (∃ *x*, .. (∃ *y*, *P*) ..)
(at level 200, *x binder*, *y binder*, right associativity,
format "'[' '[' ∃ x .. y ']' , '/' P ']'") : *type_scope*.

Notation "x ∨ y" := (*x* ∨ *y*) (at level 85, right associativity) : *type_scope*.
Notation "x ∧ y" := (*x* ∧ *y*) (at level 80, right associativity) : *type_scope*.
Notation "x → y" := (*x* → *y*)
(at level 99, *y* at level 200, right associativity): *type_scope*.

Notation "x ↔ y" := (*x* ↔ *y*) (at level 95, no associativity): *type_scope*.
Notation "¬ x" := (¬*x*) (at level 75, right associativity) : *type_scope*.
Notation "x ≠ y" := (*x* ≠ *y*) (at level 70) : *type_scope*.

Notation "'λ' x .. y , t" := (fun *x* ⇒ .. (fun *y* ⇒ *t*) ..)
(at level 200, *x binder*, *y binder*, right associativity,
format "'[' '[' 'λ' x .. y ']' , '/' t ']'").

Chapter 42

Library Coq.Logic.PropExtensionality

This module states propositional extensionality and draws consequences of it

```
Axiom propositional_extensionality :
  ∀ (P Q : Prop), (P ↔ Q) → P = Q.
```

```
Require Import ClassicalFacts.
```

```
Theorem proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.
```

Chapter 43

Library Coq.Logic.Epsilon

This file provides indefinite description under the form of Hilbert's epsilon operator; it does not assume classical logic.

Require Import ChoiceFacts.

Set Implicit Arguments.

Hilbert's epsilon: operator and specification in one statement

Axiom *epsilon_statement* :
 ∀ (A : Type) (P : A→Prop), **inhabited** A →
  { x : A | (∃ x, P x) → P x }.

Lemma constructive_indefinite_description :
 ∀ (A : Type) (P : A→Prop),
  (∃ x, P x) → { x : A | P x }.

Lemma small_drinkers'_paradox :
 ∀ (A:Type) (P:A → Prop), **inhabited** A →
  ∃ x, (∃ x, P x) → P x.

Theorem iota_statement :
 ∀ (A : Type) (P : A→Prop), **inhabited** A →
 { x : A | (∃! x : A, P x) → P x }.

Lemma constructive_definite_description :
 ∀ (A : Type) (P : A→Prop),
  (∃! x, P x) → { x : A | P x }.

Hilbert's epsilon operator and its specification

Definition epsilon (A : Type) (i:**inhabited** A) (P : A→Prop) : A
 := proj1_sig (*epsilon_statement* P i).

Definition epsilon_spec (A : Type) (i:**inhabited** A) (P : A→Prop) :
 (∃ x, P x) → P (epsilon i P)
 := proj2_sig (*epsilon_statement* P i).

Church's iota operator and its specification

Definition iota (A : Type) (i:**inhabited** A) (P : A→Prop) : A

```
  := proj1_sig (iota_statement P i).
Definition iota_spec (A : Type) (i:inhabited A) (P : A→Prop) :
  (∃! x:A, P x) → P (iota i P)
  := proj2_sig (iota_statement P i).
```

Chapter 44

Library Coq.Logic.WeakFan

A constructive proof of a non-standard version of the weak Fan Theorem in the formulation of which infinite paths are treated as predicates. The representation of paths as relations avoid the need for classical logic and unique choice. The idea of the proof comes from the proof of the weak König's lemma from separation in second-order arithmetic [*Simpson99*].

[*Simpson99*] Stephen G. Simpson. Subsystems of second order arithmetic, Cambridge University Press, 1999

```
Require Import List.
Import ListNotations.
```

inductively_barred P l means that P eventually holds above l

```
Inductive inductively_barred P : list bool → Prop :=
| now l : P l → inductively_barred P l
| propagate l :
    inductively_barred P (true::l) →
    inductively_barred P (false::l) →
    inductively_barred P l.
```

approx X l says that *l* is a boolean representation of a prefix of *X*

```
Fixpoint approx X (l:list bool) :=
  match l with
  | [] ⇒ True
  | b::l ⇒ approx X l ∧ (if b then X (length l) else ¬ X (length l))
  end.
```

barred P means that for any infinite path represented as a predicate, the property *P* holds for some prefix of the path

```
Definition barred P :=
  ∀ (X:nat → Prop), ∃ l, approx X l ∧ P l.
```

The proof proceeds by building a set *Y* of finite paths approximating either the smallest unbarred infinite path in *P*, if there is one (taking *true*>*false*), or the path *true*::*true*::... if *P* happens to be inductively_barred

```
Fixpoint Y P (l:list bool) :=
```

```
  match l with
  | [] ⇒ True
  | b::l ⇒
      Y P l ∧
      if b then inductively_barred P (false::l) else ¬ inductively_barred P (false::l)
  end.
```

Lemma Y_unique : ∀ P $l1$ $l2$, length $l1$ = length $l2$ → Y P $l1$ → Y P $l2$ → $l1$ = $l2$.

X is the translation of Y as a predicate

Definition X P n := ∃ l, length l = n ∧ Y P (true::l).

Lemma Y_approx : ∀ P l, approx (X P) l → Y P l.

Theorem WeakFanTheorem : ∀ P, barred P → **inductively_barred** P [].

Chapter 45

Library Coq.Logic.HLevels

The first three levels of homotopy type theory: homotopy propositions, homotopy sets and homotopy one types. For more information, https://github.com/HoTT/HoTT and https://homotopytypetheory.org/book

Univalence is not assumed here, and equality is Coq's usual inductive type eq in sort Prop. This is a little different from HoTT, where sort Prop does not exist and equality is directly in sort Type.

```
Require Import Coq.Logic.FunctionalExtensionality.

Definition IsHProp (P : Type) : Prop
  := ∀ p q : P, p = q.

Definition IsHSet (X : Type) : Prop
  := ∀ (x y : X) (p q : x = y), p = q.

Definition IsHOneType (X : Type) : Prop
  := ∀ (x y : X) (p q : x = y) (r s : p = q), r = s.

Lemma forall_hprop : ∀ (A : Type) (P : A → Prop),
    (∀ x:A, IsHProp (P x))
    → IsHProp (∀ x:A, P x).

Lemma and_hprop : ∀ P Q : Prop,
    IsHProp P → IsHProp Q → IsHProp (P ∧ Q).

Lemma impl_hprop : ∀ P Q : Prop,
    IsHProp Q → IsHProp (P → Q).

Lemma false_hprop : IsHProp False.

Lemma true_hprop : IsHProp True.

Lemma not_hprop : ∀ P : Type, IsHProp (P → False).

Lemma hset_hprop : ∀ X : Type,
    IsHProp X → IsHSet X.

Lemma eq_trans_cancel : ∀ {X : Type} {x y z : X} (p : x = y) (q r : y = z),
  (eq_trans p q = eq_trans p r) → q = r.

Lemma hset_hOneType : ∀ X : Type,
    IsHSet X → IsHOneType X.

Lemma hprop_hprop : ∀ X : Type,
```

IsHProp (IsHProp X).

```
Lemma hprop_hset : ∀ X : Type,
  IsHProp (IsHSet X).
```

Chapter 46

Library Coq.Logic.IndefiniteDescription

This file provides a constructive form of indefinite description that allows building choice functions; this is weaker than Hilbert's epsilon operator (which implies weakly classical properties) but stronger than the axiom of choice (which cannot be used outside the context of a theorem proof).

```
Require Import ChoiceFacts.

Set Implicit Arguments.

Axiom constructive_indefinite_description :
  ∀ (A : Type) (P : A→Prop),
    (∃ x, P x) → { x : A | P x }.

Lemma constructive_definite_description :
  ∀ (A : Type) (P : A→Prop),
    (∃! x, P x) → { x : A | P x }.

Lemma functional_choice :
  ∀ (A B : Type) (R:A→B→Prop),
    (∀ x : A, ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x : A, R x (f x)).
```

Chapter 47

Library Coq.Logic.ClassicalUniqueChoice

This file provides classical logic and unique choice; this is weaker than providing iota operator and classical logic as the definite descriptions provided by the axiom of unique choice can be used only in a propositional context (especially, they cannot be used to build functions outside the scope of a theorem proof)

Classical logic and unique choice, as shown in [*ChicliPottierSimpson02*], implies the double-negation of excluded-middle in `Set`, hence it implies a strongly classical world. Especially it conflicts with the impredicativity of `Set`.

[*ChicliPottierSimpson02*] Laurent Chicli, Loïc Pottier, Carlos Simpson, Mathematical Quotients and Quotient Types in Coq, Proceedings of TYPES 2002, Lecture Notes in Computer Science 2646, Springer Verlag.

```
Require Export Classical.

Axiom
  dependent_unique_choice :
    ∀ (A:Type) (B:A → Type) (R:∀ x:A, B x → Prop),
      (∀ x : A, ∃! y : B x, R x y) →
      (∃ f : (∀ x:A, B x), ∀ x:A, R x (f x)).
```

Unique choice reifies functional relations into functions

```
Theorem unique_choice :
 ∀ (A B:Type) (R:A → B → Prop),
   (∀ x:A, ∃! y : B, R x y) →
   (∃ f:A→B, ∀ x:A, R x (f x)).
```

The following proof comes from [*ChicliPottierSimpson02*] `Require Import Setoid.`

```
Theorem classic_set_in_prop_context :
  ∀ C:Prop, ((∀ P:Prop, {P} + {¬ P}) → C) → C.

Corollary not_not_classic_set :
  ((∀ P:Prop, {P} + {¬ P}) → False) → False.

Notation classic_set := not_not_classic_set (only parsing).
```

Chapter 48

Library Coq.Logic.ClassicalDescription

This file provides classical logic and definite description, which is equivalent to providing classical logic and Church's iota operator

Classical logic and definite descriptions implies excluded-middle in Set and leads to a classical world populated with non computable functions. It conflicts with the impredicativity of Set

```
Set Implicit Arguments.

Require Export Classical. Require Export Description. Require Import ChoiceFacts.
```

The idea for the following proof comes from *ChicliPottierSimpson02*

```
Theorem excluded_middle_informative : ∀ P:Prop, {P} + {¬ P}.

Theorem classical_definite_description :
  ∀ (A : Type) (P : A→Prop), inhabited A →
  { x : A | (∃! x : A, P x) → P x }.
```

Church's iota operator

```
Definition iota (A : Type) (i:inhabited A) (P : A→Prop) : A
  := proj1_sig (classical_definite_description P i).

Definition iota_spec (A : Type) (i:inhabited A) (P : A→Prop) :
  (∃! x:A, P x) → P (iota i P)
  := proj2_sig (classical_definite_description P i).
```

Axiom of unique "choice" (functional reification of functional relations)

```
Theorem dependent_unique_choice
:
  ∀ (A:Type) (B:A → Type) (R:∀ x:A, B x → Prop),
    (∀ x:A, ∃! y : B x, R x y) →
    (∃ f : (∀ x:A, B x), ∀ x:A, R x (f x)).

Theorem unique_choice :
  ∀ (A B:Type) (R:A → B → Prop),
    (∀ x:A, ∃! y : B, R x y) →
    (∃ f : A → B, ∀ x:A, R x (f x)).
```

Compatibility lemmas

```
Unset Implicit Arguments.
```

```
Definition dependent_description := dependent_unique_choice.
Definition description := unique_choice.
```

Chapter 49

Library Coq.Logic.Decidable

Properties of decidable propositions

```
Definition decidable (P:Prop) := P ∨ ¬ P.
Theorem dec_not_not : ∀ P:Prop, decidable P → (¬ P → False) → P.
Theorem dec_True : decidable True.
Theorem dec_False : decidable False.
Theorem dec_or :
 ∀ A B:Prop, decidable A → decidable B → decidable (A ∨ B).
Theorem dec_and :
 ∀ A B:Prop, decidable A → decidable B → decidable (A ∧ B).
Theorem dec_not : ∀ A:Prop, decidable A → decidable (¬ A).
Theorem dec_imp :
 ∀ A B:Prop, decidable A → decidable B → decidable (A → B).
Theorem dec_iff :
 ∀ A B:Prop, decidable A → decidable B → decidable (A↔B).
Theorem not_not : ∀ P:Prop, decidable P → ¬ ¬ P → P.
Theorem not_or : ∀ A B:Prop, ¬ (A ∨ B) → ¬ A ∧ ¬ B.
Theorem not_and : ∀ A B:Prop, decidable A → ¬ (A ∧ B) → ¬ A ∨ ¬ B.
Theorem not_imp : ∀ A B:Prop, decidable A → ¬ (A → B) → A ∧ ¬ B.
Theorem imp_simp : ∀ A B:Prop, decidable A → (A → B) → ¬ A ∨ B.
Theorem not_iff :
  ∀ A B:Prop, decidable A → decidable B →
    ¬ (A ↔ B) → (A ∧ ¬ B) ∨ (¬ A ∧ B).
```

Results formulated with iff, used in FSetDecide. Negation are expanded since it is unclear whether setoid rewrite will always perform conversion.

We begin with lemmas that, when read from left to right, can be understood as ways to eliminate uses of *not*.

```
Theorem not_true_iff : (True → False) ↔ False.
```

Theorem not_false_iff : (**False** $\rightarrow$ **False**) $\leftrightarrow$ **True**.

Theorem not_not_iff : $\forall$ A:Prop, decidable A $\rightarrow$
(((A $\rightarrow$ **False**) $\rightarrow$ **False**) $\leftrightarrow$ A).

Theorem contrapositive : $\forall$ A B:Prop, decidable A $\rightarrow$
(((A $\rightarrow$ **False**) $\rightarrow$ (B $\rightarrow$ **False**)) $\leftrightarrow$ (B $\rightarrow$ A)).

Lemma or_not_l_iff_1 : $\forall$ A B: Prop, decidable A $\rightarrow$
((A $\rightarrow$ **False**) $\vee$ B $\leftrightarrow$ (A $\rightarrow$ B)).

Lemma or_not_l_iff_2 : $\forall$ A B: Prop, decidable B $\rightarrow$
((A $\rightarrow$ **False**) $\vee$ B $\leftrightarrow$ (A $\rightarrow$ B)).

Lemma or_not_r_iff_1 : $\forall$ A B: Prop, decidable A $\rightarrow$
(A $\vee$ (B $\rightarrow$ **False**) $\leftrightarrow$ (B $\rightarrow$ A)).

Lemma or_not_r_iff_2 : $\forall$ A B: Prop, decidable B $\rightarrow$
(A $\vee$ (B $\rightarrow$ **False**) $\leftrightarrow$ (B $\rightarrow$ A)).

Lemma imp_not_l : $\forall$ A B: Prop, decidable A $\rightarrow$
(((A $\rightarrow$ **False**) $\rightarrow$ B) $\leftrightarrow$ (A $\vee$ B)).

Moving Negations Around: We have four lemmas that, when read from left to right, describe how to push negations toward the leaves of a proposition and, when read from right to left, describe how to pull negations toward the top of a proposition.

Theorem not_or_iff : $\forall$ A B:Prop,
(A $\vee$ B $\rightarrow$ **False**) $\leftrightarrow$ (A $\rightarrow$ **False**) $\wedge$ (B $\rightarrow$ **False**).

Lemma not_and_iff : $\forall$ A B:Prop,
(A $\wedge$ B $\rightarrow$ **False**) $\leftrightarrow$ (A $\rightarrow$ B $\rightarrow$ **False**).

Lemma not_imp_iff : $\forall$ A B:Prop, decidable A $\rightarrow$
(((A $\rightarrow$ B) $\rightarrow$ **False**) $\leftrightarrow$ A $\wedge$ (B $\rightarrow$ **False**)).

Lemma not_imp_rev_iff : $\forall$ A B : Prop, decidable A $\rightarrow$
(((A $\rightarrow$ B) $\rightarrow$ **False**) $\leftrightarrow$ (B $\rightarrow$ **False**) $\wedge$ A).

Theorem dec_functional_relation :
$\forall$ (X Y : Type) (A:$X\rightarrow Y\rightarrow$Prop), ($\forall$ y y' : Y, decidable (y=y')) $\rightarrow$
($\forall$ x, $\exists$! y, A x y) $\rightarrow$ $\forall$ x y, decidable (A x y).

With the following hint database, we can leverage `auto` to check decidability of propositions.

#[*global*]
Hint Resolve *dec_True dec_False dec_or dec_and dec_imp dec_not dec_iff*
: *decidable_prop.*

solve_decidable `using` *lib* will solve goals about the decidability of a proposition, assisted by an auxiliary database of lemmas. The database is intended to contain lemmas stating the decidability of base propositions, (e.g., the decidability of equality on a particular inductive type).

```
Tactic Notation "solve_decidable" "using" ident(db) :=
  match goal with
   | ⊢ decidable _ ⇒
     solve [ auto 100 with decidable_prop db ]
```

```
  end.
Tactic Notation "solve_decidable" :=
  solve_decidable using core.
```

Chapter 50

Library Coq.Logic.Diaconescu

Diaconescu showed that the Axiom of Choice entails Excluded-Middle in topoi [*Diaconescu75*]. Lacas and Werner adapted the proof to show that the axiom of choice in equivalence classes entails Excluded-Middle in Type Theory [*LacasWerner99*].

Three variants of Diaconescu's result in type theory are shown below.

A. A proof that the relational form of the Axiom of Choice + Extensionality for Predicates entails Excluded-Middle (by Hugo Herbelin)

B. A proof that the relational form of the Axiom of Choice + Proof Irrelevance entails Excluded-Middle for Equality Statements (by Benjamin Werner)

C. A proof that extensional Hilbert epsilon's description operator entails excluded-middle (taken from Bell [*Bell93*])

See also [*Carlström04*] for a discussion of the connection between the Extensional Axiom of Choice and Excluded-Middle

[*Diaconescu75*] Radu Diaconescu, Axiom of Choice and Complementation, in Proceedings of AMS, vol 51, pp 176-178, 1975.

[*LacasWerner99*] Samuel Lacas, Benjamin Werner, Which Choices imply the excluded middle?, preprint, 1999.

[*Bell93*] John L. Bell, Hilbert's epsilon operator and classical logic, Journal of Philosophical Logic, 22: 1-18, 1993

[*Carlström04*] Jesper Carlström, EM + Ext + AC_int is equivalent to AC_ext, Mathematical Logic Quaterly, vol 50(3), pp 236-240, 2004.


```
Require ClassicalFacts ChoiceFacts.
```

50.1 Pred. Ext. + Rel. Axiom of Choice -> Excluded-Middle

```
Section PredExt_RelChoice_imp_EM.
```

The axiom of extensionality for predicates

```
Definition PredicateExtensionality :=
  ∀ P Q:bool → Prop, (∀ b:bool, P b ↔ Q b) → P = Q.
```

From predicate extensionality we get propositional extensionality hence proof-irrelevance

```
Import ClassicalFacts.
```

```
Variable pred_extensionality : PredicateExtensionality.

Lemma prop_ext : ∀ A B:Prop, (A ↔ B) → A = B.

Lemma proof_irrel : ∀ (A:Prop) (a1 a2:A), a1 = a2.
```

From proof-irrelevance and relational choice, we get guarded relational choice

```
Import ChoiceFacts.

Variable rel_choice : RelationalChoice.

Lemma guarded_rel_choice : GuardedRelationalChoice.
```

The form of choice we need: there is a functional relation which chooses an element in any non empty subset of bool

```
Import Bool.

Lemma AC_bool_subset_to_bool :
  ∃ R : (bool → Prop) → bool → Prop,
   (∀ P:bool → Prop,
      (∃ b : bool, P b) →
       ∃ b : bool, P b ∧ R P b ∧ (∀ b':bool, R P b' → b = b')).
```

The proof of the excluded middle Remark: P could have been in Set or Type

```
Theorem pred_ext_and_rel_choice_imp_EM : ∀ P:Prop, P ∨ ¬ P.

End PredExt_RelChoice_imp_EM.
```

50.2 Proof-Irrel. + Rel. Axiom of Choice -> Excl.-Middle for Equality

This is an adaptation of Diaconescu's theorem, exploiting the form of extensionality provided by proof-irrelevance

```
Section ProofIrrel_RelChoice_imp_EqEM.

Import ChoiceFacts.

Variable rel_choice : RelationalChoice.

Variable proof_irrelevance : ∀ P:Prop , ∀ x y:P, x=y.
```

Let *a1* and *a2* be two elements in some type *A*

```
Variable A :Type.
Variables a1 a2 : A.
```

We build the subset *A'* of *A* made of *a1* and *a2*

```
Definition A' := @sigT A (fun x ⇒ x=a1 ∨ x=a2).

Definition a1':A'.
Defined.

Definition a2':A'.
Defined.
```

By proof-irrelevance, projection is a retraction

Lemma projT1_injective : $a1 = a2 \rightarrow$ a1'=a2'.

But from the actual proofs of being in *A'*, we can assert in the proof-irrelevant world the existence of relevant boolean witnesses

Lemma decide : $\forall$ x:A', $\exists$ y:**bool** ,
  (projT1 x = $a1$ $\wedge$ y = true) $\vee$ (projT1 x = $a2$ $\wedge$ y = false).

Thanks to the axiom of choice, the boolean witnesses move from the propositional world to the relevant world

Theorem proof_irrel_rel_choice_imp_eq_dec : $a1 = a2 \vee \neg a1 = a2$.

An alternative more concise proof can be done by directly using the guarded relational choice

Lemma proof_irrel_rel_choice_imp_eq_dec' : $a1 = a2 \vee \neg a1 = a2$.

End ProofIrrel_RelChoice_imp_EqEM.

50.3 Extensional Hilbert's epsilon description operator -> Excluded-Middle

Proof sketch from Bell [*Bell93*] (with thanks to P. Castéran)

Section ExtensionalEpsilon_imp_EM.

Variable *epsilon* : $\forall$ A : Type, inhabited $A \rightarrow (A \rightarrow$ Prop$) \rightarrow A$.

Hypothesis *epsilon_spec* :
  $\forall$ (A:Type) (i:inhabited A) (P:$A\rightarrow$Prop),
  ($\exists$ x, P x) $\rightarrow$ P (*epsilon* A i P).

Hypothesis *epsilon_extensionality* :
  $\forall$ (A:Type) (i:inhabited A) (P Q:$A\rightarrow$Prop),
  ($\forall$ a, P $a \leftrightarrow Q$ a) $\rightarrow$ *epsilon* A i P = *epsilon* A i Q.

Theorem extensional_epsilon_imp_EM : $\forall$ P:Prop, $P \vee \neg P$.

End ExtensionalEpsilon_imp_EM.

Chapter 51

Library Coq.Logic.Classical

Classical Logic

```
Require Export Classical_Prop.
Require Export Classical_Pred_Type.
```

Chapter 52

Library Coq.Logic.ExtensionalityFacts

Some facts and definitions about extensionality

We investigate the relations between the following extensionality principles

- Functional extensionality
- Equality of projections from diagonal
- Unicity of inverse bijections
- Bijectivity of bijective composition

Table of contents
1. Definitions
2. Functional extensionality <-> Equality of projections from diagonal
3. Functional extensionality <-> Unicity of inverse bijections
4. Functional extensionality <-> Bijectivity of bijective composition

```
Set Implicit Arguments.
```

52.1 Definitions

Being an inverse

```
Definition is_inverse A B f g := (∀ a:A, g (f a) = a) ∧ (∀ b:B, f (g b) = b).
```

The diagonal over A and the one-one correspondence with A

```
#[universes(template)]
Record Delta A := { pi1:A; pi2:A; eq:pi1=pi2 }.
```

```
Definition delta {A} (a:A) := {|pi1 := a; pi2 := a; eq := eq_refl a |}.
```

```
Lemma diagonal_projs_same_behavior : ∀ A (x:Delta A), pi1 x = pi2 x.
```

```
Lemma diagonal_inverse1 : ∀ A, is_inverse (A:=A) delta pi1.
```

```
Lemma diagonal_inverse2 : ∀ A, is_inverse (A:=A) delta pi2.
```

Functional extensionality

Equality of projections from diagonal

Unicity of bijection inverse

Bijectivity of bijective composition

Definition action $A\ B\ C\ (f{:}A{\to}B) := (\texttt{fun}\ h{:}B{\to}C \Rightarrow \texttt{fun}\ x \Rightarrow h\ (f\ x))$.

52.2 Functional extensionality <-> Equality of projections from diagonal

Theorem FunctExt_iff_EqDeltaProjs : FunctionalExtensionality ↔ EqDeltaProjs.

52.3 Functional extensionality <-> Unicity of bijection inverse

Lemma FunctExt_UniqInverse : FunctionalExtensionality → UniqueInverse.

Lemma UniqInverse_EqDeltaProjs : UniqueInverse → EqDeltaProjs.

Theorem FunctExt_iff_UniqInverse : FunctionalExtensionality ↔ UniqueInverse.

52.4 Functional extensionality <-> Bijectivity of bijective composition

Lemma FunctExt_BijComp : FunctionalExtensionality → BijectivityBijectiveComp.

Lemma BijComp_FunctExt : BijectivityBijectiveComp → FunctionalExtensionality.

Chapter 53

Library Coq.Logic.ClassicalFacts

Some facts and definitions about classical logic

Table of contents:

1. Propositional degeneracy = excluded-middle + propositional extensionality
2. Classical logic and proof-irrelevance
2.1. CC |- prop. ext. + A inhabited -> (A = A->A) -> A has fixpoint
2.2. CC |- prop. ext. + dep elim on bool -> proof-irrelevance
2.3. CIC |- prop. ext. -> proof-irrelevance
2.4. CC |- excluded-middle + dep elim on bool -> proof-irrelevance
2.5. CIC |- excluded-middle -> proof-irrelevance
3. Weak classical axioms
3.1. Weak excluded middle and classical de Morgan law
3.2. Gödel-Dummett axiom and right distributivity of implication over disjunction
3 3. Independence of general premises and drinker's paradox
4. Principles equivalent to classical logic
4.1 Classical logic = principle of unrestricted minimization
4.2 Classical logic = choice of representatives in a partition of bool

53.1 Prop degeneracy = excluded-middle + prop extensionality

i.e. $(\forall\ A,\ A{=}True \lor A{=}False) \leftrightarrow (\forall\ A,\ A\backslash/\tilde{}A) \land (\forall\ A\ B,\ (A{\leftrightarrow}B) \rightarrow A{=}B)$

prop_degeneracy (also referred to as propositional completeness) asserts (up to consistency) that there are only two distinct formulas Definition prop_degeneracy := ∀ *A*:Prop, *A* = **True** ∨ *A* = **False**.

prop_extensionality asserts that equivalent formulas are equal Definition prop_extensionality := ∀ *A* *B*:Prop, (*A* ↔ *B*) → *A* = *B*.

excluded_middle asserts that we can reason by case on the truth or falsity of any formula Definition excluded_middle := ∀ *A*:Prop, *A* ∨ ¬ *A*.

We show *prop_degeneracy* ↔ (*prop_extensionality* ∧ *excluded_middle*)

Lemma prop_degen_ext : prop_degeneracy → prop_extensionality.

Lemma prop_degen_em : prop_degeneracy → excluded_middle.

Lemma prop_ext_em_degen :
  prop_extensionality → excluded_middle → prop_degeneracy.

A weakest form of propositional extensionality: extensionality for provable propositions only

Require Import PropExtensionalityFacts.

Definition provable_prop_extensionality := ∀ A:Prop, A → A = **True**.

Lemma provable_prop_ext :
  prop_extensionality → provable_prop_extensionality.

53.2 Classical logic and proof-irrelevance

53.2.1 CC |- prop ext + A inhabited -> (A = A->A) -> A has fixpoint

We successively show that:

prop_extensionality implies equality of A and A→A for inhabited A, which implies the existence of a (trivial) retract from A→A to A (just take the identity), which implies the existence of a fixpoint operator in A (e.g. take the Y combinator of lambda-calculus)

Lemma prop_ext_A_eq_A_imp_A :
  prop_extensionality → ∀ A:Prop, inhabited A → (A → A) = A.

Record **retract** (A B:Prop) : Prop :=
  {f1 : A → B; f2 : B → A; f1_o_f2 : ∀ x:B, f1 (f2 x) = x}.

Lemma prop_ext_retract_A_A_imp_A :
  prop_extensionality → ∀ A:Prop, inhabited A → **retract** A (A → A).

Record **has_fixpoint** (A:Prop) : Prop :=
  {F : (A → A) → A; Fix : ∀ f:A → A, F f = f (F f)}.

Lemma ext_prop_fixpoint :
  prop_extensionality → ∀ A:Prop, inhabited A → **has_fixpoint** A.

Remark: *prop_extensionality* can be replaced in lemma *ext_prop_fixpoint* by the weakest property *provable_prop_extensionality*.

53.2.2 CC |- prop_ext /\ dep elim on bool -> proof-irrelevance

proof_irrelevance asserts equality of all proofs of a given formula Definition proof_irrelevance := ∀ (A:Prop) ($a1$ $a2$:A), $a1$ = $a2$.

Assume that we have booleans with the property that there is at most 2 booleans (which is equivalent to dependent case analysis). Consider the fixpoint of the negation function: it is either true or false by dependent case analysis, but also the opposite by fixpoint. Hence proof-irrelevance.

We then map equality of boolean proofs to proof irrelevance in all propositions.

Section Proof_irrelevance_gen.

  Variable *bool* : Prop.
  Variable *true* : *bool*.
  Variable *false* : *bool*.
  Hypothesis *bool_elim* : ∀ C:Prop, C → C → *bool* → C.

Hypothesis
bool_elim_redl : ∀ (C:Prop) (*c1 c2*:C), *c1* = *bool_elim C c1 c2 true*.
Hypothesis
bool_elim_redr : ∀ (C:Prop) (*c1 c2*:C), *c2* = *bool_elim C c1 c2 false*.
Let *bool_dep_induction* :=
∀ P:*bool* → Prop, P *true* → P *false* → ∀ b:*bool*, P b.

Lemma aux : prop_extensionality → *bool_dep_induction* → *true* = *false*.

Lemma ext_prop_dep_proof_irrel_gen :
prop_extensionality → *bool_dep_induction* → proof_irrelevance.

End Proof_irrelevance_gen.

In the pure Calculus of Constructions, we can define the boolean proposition bool = (C:Prop)C->C->C but we cannot prove that it has at most 2 elements.

Section Proof_irrelevance_Prop_Ext_CC.

Definition BoolP := ∀ C:Prop, C → C → C.
Definition TrueP : BoolP := fun C *c1 c2* ⇒ *c1*.
Definition FalseP : BoolP := fun C *c1 c2* ⇒ *c2*.
Definition BoolP_elim C *c1 c2* (b:BoolP) := b C *c1 c2*.
Definition BoolP_elim_redl (C:Prop) (*c1 c2*:C) :
c1 = BoolP_elim C *c1 c2* TrueP := eq_refl *c1*.
Definition BoolP_elim_redr (C:Prop) (*c1 c2*:C) :
c2 = BoolP_elim C *c1 c2* FalseP := eq_refl *c2*.

Definition BoolP_dep_induction :=
∀ P:BoolP → Prop, P TrueP → P FalseP → ∀ b:BoolP, P b.

Lemma ext_prop_dep_proof_irrel_cc :
prop_extensionality → BoolP_dep_induction → proof_irrelevance.

End Proof_irrelevance_Prop_Ext_CC.

Remark: *prop_extensionality* can be replaced in lemma *ext_prop_dep_proof_irrel_gen* by the weakest property *provable_prop_extensionality*.

53.2.3 CIC |- prop. ext. -> proof-irrelevance

In the Calculus of Inductive Constructions, inductively defined booleans enjoy dependent case analysis, hence directly proof-irrelevance from propositional extensionality.

Section Proof_irrelevance_CIC.

Inductive **boolP** : Prop :=
| trueP : **boolP**
| falseP : **boolP**.
Definition boolP_elim_redl (C:Prop) (*c1 c2*:C) :
c1 = boolP_ind C *c1 c2* trueP := eq_refl *c1*.
Definition boolP_elim_redr (C:Prop) (*c1 c2*:C) :
c2 = boolP_ind C *c1 c2* falseP := eq_refl *c2*.
Scheme boolP_indd := Induction for **boolP** Sort Prop.

```
  Lemma ext_prop_dep_proof_irrel_cic : prop_extensionality → proof_irrelevance.
End Proof_irrelevance_CIC.
```

Can we state proof irrelevance from propositional degeneracy (i.e. propositional extensionality + excluded middle) without dependent case analysis ?

Berardi [*Berardi90*] built a model of CC interpreting inhabited types by the set of all untyped lambda-terms. This model satisfies propositional degeneracy without satisfying proof-irrelevance (nor dependent case analysis). This implies that the previous results cannot be refined.

[*Berardi90*] Stefano Berardi, "Type dependence and constructive mathematics", Ph. D. thesis, Dipartimento Matematica, Università di Torino, 1990.

53.2.4 CC |- excluded-middle + dep elim on bool -> proof-irrelevance

This is a proof in the pure Calculus of Construction that classical logic in `Prop` + dependent elimination of disjunction entails proof-irrelevance.

Reference:

[*Coquand90*] T. Coquand, "Metamathematical Investigations of a Calculus of Constructions", Proceedings of Logic in Computer Science (LICS'90), 1990.

Proof skeleton: classical logic + dependent elimination of disjunction + discrimination of proofs implies the existence of a retract from `Prop` into *bool*, hence inconsistency by encoding any paradox of system U- (e.g. Hurkens' paradox).

```
Require Import Hurkens.

Section Proof_irrelevance_EM_CC.

  Variable or : Prop → Prop → Prop.
  Variable or_introl : ∀ A B:Prop, A → or A B.
  Variable or_intror : ∀ A B:Prop, B → or A B.
  Hypothesis or_elim : ∀ A B C:Prop, (A → C) → (B → C) → or A B → C.
  Hypothesis
    or_elim_redl :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (a:A),
      f a = or_elim A B C f g (or_introl A B a).
  Hypothesis
    or_elim_redr :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (b:B),
      g b = or_elim A B C f g (or_intror A B b).
  Hypothesis
    or_dep_elim :
    ∀ (A B:Prop) (P:or A B → Prop),
      (∀ a:A, P (or_introl A B a)) →
      (∀ b:B, P (or_intror A B b)) → ∀ b:or A B, P b.

  Hypothesis em : ∀ A:Prop, or A (¬ A).
  Variable B : Prop.
  Variables b1 b2 : B.
```

p2b and *b2p* form a retract if ¬*b1*=*b2*

```
  Let p2b A := or_elim A (¬ A) B (fun _ ⇒ b1) (fun _ ⇒ b2) (em A).
```

```
  Let b2p b := b1 = b.

  Lemma p2p1 : ∀ A:Prop, A → b2p (p2b A).

  Lemma p2p2 : b1 ≠ b2 → ∀ A:Prop, b2p (p2b A) → A.
```

Using excluded-middle a second time, we get proof-irrelevance

```
  Theorem proof_irrelevance_cc : b1 = b2.

End Proof_irrelevance_EM_CC.
```

Hurkens' paradox still holds with a retract from the *negative* fragment of Prop into *bool*, hence weak classical logic, i.e. ∀ A, ¬A\/~~A, is enough for deriving a weak version of proof-irrelevance. This is enough to derive a contradiction from a Set-bound weak excluded middle with an impredicative Set universe.

```
Section Proof_irrelevance_WEM_CC.

  Variable or : Prop → Prop → Prop.
  Variable or_introl : ∀ A B:Prop, A → or A B.
  Variable or_intror : ∀ A B:Prop, B → or A B.
  Hypothesis or_elim : ∀ A B C:Prop, (A → C) → (B → C) → or A B → C.
  Hypothesis
    or_elim_redl :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (a:A),
      f a = or_elim A B C f g (or_introl A B a).
  Hypothesis
    or_elim_redr :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (b:B),
      g b = or_elim A B C f g (or_intror A B b).
  Hypothesis
    or_dep_elim :
    ∀ (A B:Prop) (P:or A B → Prop),
      (∀ a:A, P (or_introl A B a)) →
      (∀ b:B, P (or_intror A B b)) → ∀ b:or A B, P b.

  Hypothesis wem : ∀ A:Prop, or (~~A) (¬ A).

  Variable B : Prop.
  Variables b1 b2 : B.
```

p2b and *b2p* form a retract if ¬*b1*=*b2*

```
  Let p2b (A:NProp) := or_elim (~~El A) (¬El A) B (fun _ ⇒ b1) (fun _ ⇒ b2) (wem (El A)).
  Let b2p b : NProp := exist (fun P⇒~~P → P) (~~(b1 = b)) (fun h x ⇒ h (fun k ⇒ k x)).

  Lemma wp2p1 : ∀ A:NProp, El A → El (b2p (p2b A)).

  Lemma wp2p2 : b1 ≠ b2 → ∀ A:NProp, El (b2p (p2b A)) → El A.
```

By Hurkens's paradox, we get a weak form of proof irrelevance.

```
  Theorem wproof_irrelevance_cc : ~~(b1 = b2).

End Proof_irrelevance_WEM_CC.
```

53.2.5 CIC |- excluded-middle -> proof-irrelevance

Since, dependent elimination is derivable in the Calculus of Inductive Constructions (CCI), we get proof-irrelevance from classical logic in the CCI.

```
Section Proof_irrelevance_CCI.
  Hypothesis em : ∀ A:Prop, A ∨ ¬ A.
  Definition or_elim_redl (A B C:Prop) (f:A → C) (g:B → C)
    (a:A) : f a = or_ind f g (or_introl B a) := eq_refl (f a).
  Definition or_elim_redr (A B C:Prop) (f:A → C) (g:B → C)
    (b:B) : g b = or_ind f g (or_intror A b) := eq_refl (g b).
  Scheme or_indd := Induction for or Sort Prop.
  Theorem proof_irrelevance_cci : ∀ (B:Prop) (b1 b2:B), b1 = b2.
End Proof_irrelevance_CCI.
```

The same holds with weak excluded middle. The proof is a little more involved, however.

```
Section Weak_proof_irrelevance_CCI.
  Hypothesis wem : ∀ A:Prop, ~~A ∨ ¬ A.
  Theorem wem_proof_irrelevance_cci : ∀ (B:Prop) (b1 b2:B), ~~b1 = b2.
End Weak_proof_irrelevance_CCI.
```

Remark: in the Set-impredicative CCI, Hurkens' paradox still holds with *bool* in Set and since ¬*true*=*false* for *true* and *false* in *bool* from Set, we get the inconsistency of *em* : ∀ *A*:Prop, {*A*}+{~*A*} in the Set-impredicative CCI.

53.3 Weak classical axioms

We show the following increasing in the strength of axioms:

- weak excluded-middle and classical De Morgan's law
- right distributivity of implication over disjunction and Gödel-Dummett axiom
- independence of general premises and drinker's paradox
- excluded-middle

53.3.1 Weak excluded-middle

The weak classical logic based on ~~*A* ∨ ¬*A* is referred to with name KC in [*ChagrovZakharyaschev97*]. See [*SorbiTerwijn11*] for a short survey.

[*ChagrovZakharyaschev97*] Alexander Chagrov and Michael Zakharyaschev, "Modal Logic", Clarendon Press, 1997.

[*SorbiTerwijn11*] Andrea Sorbi and Sebastiaan A. Terwijn, "Generalizations of the weak law of the excluded-middle", Notre Dame J. Formal Logic, vol 56(2), pp 321-331, 2015.

```
Definition weak_excluded_middle :=
  ∀ A:Prop, ~~A ∨ ¬A.
```

The interest in the equivalent variant *weak_generalized_excluded_middle* is that it holds even in logic without a primitive *False* connective (like Gödel-Dummett axiom)

Definition weak_generalized_excluded_middle :=
  ∀ A B:Prop, ((A → B) → B) ∨ (A → B).

Classical De Morgan's law

Definition classical_de_morgan_law :=
  ∀ A B:Prop, ~(A ∧ B) → ¬A ∨ ¬B.

53.3.2 Gödel-Dummett axiom

(A→B) ∨ (B→A) is studied in [*Dummett59*] and is based on [*Gödel33*].

[*Dummett59*] Michael A. E. Dummett. "A Propositional Calculus with a Denumerable Matrix", In the Journal of Symbolic Logic, vol 24(2), pp 97-103, 1959.

[*Gödel33*] Kurt Gödel. "Zum intuitionistischen Aussagenkalkül", Ergeb. Math. Koll. 4, pp. 34-38, 1933.

Definition GodelDummett := ∀ A B:Prop, (A → B) ∨ (B → A).

Lemma excluded_middle_Godel_Dummett : excluded_middle → GodelDummett.

(A→B) ∨ (B→A) is equivalent to (C → A∨B) → (C→A) ∨ (C→B) (proof from [*Dummett59*])

Definition RightDistributivityImplicationOverDisjunction :=
  ∀ A B C:Prop, (C → A∨B) → (C→A) ∨ (C→B).

Lemma Godel_Dummett_iff_right_distr_implication_over_disjunction :
  GodelDummett ↔ RightDistributivityImplicationOverDisjunction.

(A→B) ∨ (B→A) is stronger than the weak excluded middle

Lemma Godel_Dummett_weak_excluded_middle :
  GodelDummett → weak_excluded_middle.

The weak excluded middle is equivalent to the classical De Morgan's law

Lemma weak_excluded_middle_iff_classical_de_morgan_law :
  weak_excluded_middle ↔ classical_de_morgan_law.

53.3.3 Independence of general premises and drinker's paradox

Independence of general premises is the unconstrained, non constructive, version of the Independence of Premises as considered in [*Troelstra73*].

It is a generalization to predicate logic of the right distributivity of implication over disjunction (hence of Gödel-Dummett axiom) whose own constructive form (obtained by a restricting the third formula to be negative) is called Kreisel-Putnam principle [*KreiselPutnam57*].

[*KreiselPutnam57*], Georg Kreisel and Hilary Putnam. "Eine Unableitsbarkeitsbeweismethode für den intuitionistischen Aussagenkalkül". Archiv für Mathematische Logik und Graundlagenforschung, 3:74- 78, 1957.

[*Troelstra73*], Anne Troelstra, editor. Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, volume 344 of Lecture Notes in Mathematics, Springer-Verlag, 1973.

Definition IndependenceOfGeneralPremises :=

∀ (A:Type) (P:A → Prop) (Q:Prop),
inhabited $A \rightarrow (Q \rightarrow \exists\ x,\ P\ x) \rightarrow \exists\ x,\ Q \rightarrow P\ x$.

Lemma
independence_general_premises_right_distr_implication_over_disjunction :
IndependenceOfGeneralPremises → RightDistributivityImplicationOverDisjunction.

Lemma independence_general_premises_Godel_Dummett :
IndependenceOfGeneralPremises → GodelDummett.

Independence of general premises is equivalent to the drinker's paradox

Definition DrinkerParadox :=
∀ (A:Type) (P:A → Prop),
inhabited $A \rightarrow \exists\ x,\ (\exists\ x,\ P\ x) \rightarrow P\ x$.

Lemma independence_general_premises_drinker :
IndependenceOfGeneralPremises ↔ DrinkerParadox.

Independence of general premises is weaker than (generalized) excluded middle

Remark: generalized excluded middle is preferred here to avoid relying on the "ex falso quodlibet" property (i.e. *False* → ∀ A, A)

Definition generalized_excluded_middle :=
∀ A B:Prop, $A \vee (A \rightarrow B)$.

Lemma excluded_middle_independence_general_premises :
generalized_excluded_middle → DrinkerParadox.

53.4 Axioms equivalent to classical logic

53.4.1 Principle of unrestricted minimization

Require Import Coq.Arith.PeanoNat.

Definition Minimal (P:**nat** → Prop) (n:**nat**) : Prop :=
$P\ n \wedge \forall\ k,\ P\ k \rightarrow n \leq k$.

Definition Minimization_Property (P : **nat** → Prop) : Prop :=
∀ n, $P\ n \rightarrow \exists\ m$, Minimal P m.

Section Unrestricted_minimization_entails_excluded_middle.

Hypothesis *unrestricted_minimization*: ∀ P, Minimization_Property P.

Theorem unrestricted_minimization_entails_excluded_middle : ∀ A, $A \vee \neg A$.

End Unrestricted_minimization_entails_excluded_middle.

Require Import Wf_nat.

Section Excluded_middle_entails_unrestricted_minimization.

Hypothesis *em* : ∀ A, $A \vee \neg A$.

Theorem excluded_middle_entails_unrestricted_minimization :
∀ P, Minimization_Property P.

End Excluded_middle_entails_unrestricted_minimization.

However, minimization for a given predicate does not necessarily imply decidability of this predicate

```
Section Example_of_undecidable_predicate_with_the_minimization_property.
  Variable s : nat → bool.
  Let P n := ∃ k, n≤k ∧ s k = true.
  Example undecidable_predicate_with_the_minimization_property :
    Minimization_Property P.
End Example_of_undecidable_predicate_with_the_minimization_property.
```

53.4.2 Choice of representatives in a partition of bool

This is similar to Bell's "weak extensional selection principle" in [*Bell*]

[*Bell*] John L. Bell, Choice principles in intuitionistic set theory, unpublished.

```
Require Import RelationClasses.

Theorem representative_boolean_partition_imp_excluded_middle :
  representative_boolean_partition → excluded_middle.

Theorem excluded_middle_imp_representative_boolean_partition :
  excluded_middle → representative_boolean_partition.

Theorem excluded_middle_iff_representative_boolean_partition :
  excluded_middle ↔ representative_boolean_partition.
```

Chapter 54

Library Coq.Logic.SetoidChoice

This module states the functional form of the axiom of choice over setoids, commonly called extensional axiom of choice [*Carlström04*], [*Martin-Löf05*]. This is obtained by a decomposition of the axiom into the following components:

- classical logic
- relational axiom of choice
- axiom of unique choice
- a limited form of functional extensionality

Among other results, it entails:

- proof irrelevance
- choice of a representative in equivalence classes

[*Carlström04*] Jesper Carlström, EM + Ext + AC_int is equivalent to AC_ext, Mathematical Logic Quaterly, vol 50(3), pp 236-240, 2004.

[*Martin-Löf05*] *Per Martin-Löf*, 100 *years* of *Zermelo's axiom* of *choice*: *what was the problem* with *it*?, *lecture notes* for *KTH/SU colloquium*, 2005.

```
Require Export ClassicalChoice. Require Export ExtensionalFunctionRepresentative.

Require Import ChoiceFacts.
Require Import ClassicalFacts.
Require Import RelationClasses.

Theorem setoid_choice :
  ∀ A B,
  ∀ R : A → A → Prop,
  ∀ T : A → B → Prop,
  Equivalence R →
  (∀ x x' y, R x x' → T x y → T x' y) →
  (∀ x, ∃ y, T x y) →
```

```
  ∃ f : A → B, ∀ x : A, T x (f x) ∧ (∀ x' : A, R x x' → f x = f x').
Theorem representative_choice :
  ∀ A (R:A→A→Prop), (Equivalence R) →
  ∃ f : A→A, ∀ x : A, R x (f x) ∧ ∀ x', R x x' → f x = f x'.
```

Chapter 55

Library Coq.Logic.ChoiceFacts

Some facts and definitions concerning choice and description in intuitionistic logic.

55.1 References:

[*Bell*] John L. Bell, Choice principles in intuitionistic set theory, unpublished.

[*Bell93*] John L. Bell, Hilbert's Epsilon Operator in Intuitionistic Type Theories, Mathematical Logic Quarterly, volume 39, 1993.

[*Carlström04*] Jesper Carlström, EM + Ext + AC_int is equivalent to AC_ext, Mathematical Logic Quaterly, vol 50(3), pp 236-240, 2004.

[*Carlström05*] Jesper Carlström, Interpreting descriptions in intentional type theory, Journal of Symbolic Logic 70(2):488-514, 2005.

[*Werner97*] Benjamin Werner, Sets in Types, Types in Sets, TACS, 1997.

```
Require Import RelationClasses Logic.
```

```
Set Implicit Arguments.
```

55.2 Definitions

Choice, reification and description schemes

We make them all polymorphic. Most of them have existentials as conclusion so they require polymorphism otherwise their first application (e.g. to an existential in `Set`) will fix the level of A.

```
Section ChoiceSchemes.
```

```
Variables A B :Type.
```

```
Variable P:A→Prop.
```

55.2.1 Constructive choice and description

AC_rel = relational form of the (non extensional) axiom of choice (a "set-theoretic" axiom of choice)

```
Definition RelationalChoice_on :=
  ∀ R:A→B→Prop,
    (∀ x : A, ∃ y : B, R x y) →
```

($\exists$ R' : $A \to B \to$ Prop, subrelation R' R $\wedge$ $\forall$ x, $\exists!$ y, R' x y).

AC_fun = functional form of the (non extensional) axiom of choice (a "type-theoretic" axiom of choice)

Definition FunctionalChoice_on_rel (R:$A \to B \to$Prop) :=
($\forall$ x:A, $\exists$ y : B, R x y) $\to$
$\exists$ f : $A \to B$, ($\forall$ x:A, R x (f x)).

Definition FunctionalChoice_on :=
$\forall$ R:$A \to B \to$Prop,
($\forall$ x : A, $\exists$ y : B, R x y) $\to$
($\exists$ f : $A \to B$, $\forall$ x : A, R x (f x)).

AC_fun_dep = functional form of the (non extensional) axiom of choice, with dependent functions Definition DependentFunctionalChoice_on (A:Type) (B:$A \to$ Type) :=
$\forall$ R:$\forall$ x:A, B x $\to$ Prop,
($\forall$ x:A, $\exists$ y : B x, R x y) $\to$
($\exists$ f : ($\forall$ x:A, B x), $\forall$ x:A, R x (f x)).

AC_trunc = axiom of choice for propositional truncations (truncation and quantification commute) Definition InhabitedForallCommute_on (A : Type) (B : $A \to$ Type) :=
($\forall$ x, **inhabited** (B x)) $\to$ **inhabited** ($\forall$ x, B x).

DC_fun = functional form of the dependent axiom of choice

Definition FunctionalDependentChoice_on :=
$\forall$ (R:$A \to A \to$Prop),
($\forall$ x, $\exists$ y, R x y) $\to$ $\forall$ $x0$,
($\exists$ f : **nat** $\to$ A, f 0 = $x0$ $\wedge$ $\forall$ n, R (f n) (f (S n))).

ACw_fun = functional form of the countable axiom of choice

Definition FunctionalCountableChoice_on :=
$\forall$ (R:**nat**$\to A \to$Prop),
($\forall$ n, $\exists$ y, R n y) $\to$
($\exists$ f : **nat** $\to$ A, $\forall$ n, R n (f n)).

AC! = functional relation reification (known as axiom of unique choice in topos theory, sometimes called principle of definite description in the context of constructive type theory, sometimes called axiom of no choice)

Definition FunctionalRelReification_on :=
$\forall$ R:$A \to B \to$Prop,
($\forall$ x : A, $\exists!$ y : B, R x y) $\to$
($\exists$ f : $A \to B$, $\forall$ x : A, R x (f x)).

AC_dep! = functional relation reification, with dependent functions see AC! Definition DependentFunctionalRelReification_on (A:Type) (B:$A \to$ Type) :=
$\forall$ (R:$\forall$ x:A, B x $\to$ Prop),
($\forall$ x:A, $\exists!$ y : B x, R x y) $\to$
($\exists$ f : ($\forall$ x:A, B x), $\forall$ x:A, R x (f x)).

AC_fun_repr = functional choice of a representative in an equivalence class

```
Definition RepresentativeFunctionalChoice_on :=
  ∀ R:A→A→Prop,
    (Equivalence R) →
    (∃ f : A→A, ∀ x : A, (R x (f x)) ∧ ∀ x', R x x' → f x = f x').
```

AC_fun_setoid = functional form of the (so-called extensional) axiom of choice from setoids

```
Definition SetoidFunctionalChoice_on :=
  ∀ R : A → A → Prop,
  ∀ T : A → B → Prop,
  Equivalence R →
  (∀ x x' y, R x x' → T x y → T x' y) →
  (∀ x, ∃ y, T x y) →
  ∃ f : A → B, ∀ x : A, T x (f x) ∧ (∀ x' : A, R x x' → f x = f x').
```

AC_fun_setoid_gen = functional form of the general form of the (so-called extensional) axiom of choice over setoids

```
Definition GeneralizedSetoidFunctionalChoice_on :=
  ∀ R : A → A → Prop,
  ∀ S : B → B → Prop,
  ∀ T : A → B → Prop,
  Equivalence R →
  Equivalence S →
  (∀ x x' y y', R x x' → S y y' → T x y → T x' y') →
  (∀ x, ∃ y, T x y) →
  ∃ f : A → B,
    ∀ x : A, T x (f x) ∧ (∀ x' : A, R x x' → S (f x) (f x')).
```

AC_fun_setoid_simple = functional form of the (so-called extensional) axiom of choice from setoids on locally compatible relations

```
Definition SimpleSetoidFunctionalChoice_on A B :=
  ∀ R : A → A → Prop,
  ∀ T : A → B → Prop,
  Equivalence R →
  (∀ x, ∃ y, ∀ x', R x x' → T x' y) →
  ∃ f : A → B, ∀ x : A, T x (f x) ∧ (∀ x' : A, R x x' → f x = f x').
```

ID_epsilon = constructive version of indefinite description; combined with proof-irrelevance, it may be connected to Carlström's type theory with a constructive indefinite description operator

```
Definition ConstructiveIndefiniteDescription_on :=
  ∀ P:A→Prop,
    (∃ x, P x) → { x:A | P x }.
```

ID_iota = constructive version of definite description; combined with proof-irrelevance, it may be connected to Carlström's and Stenlund's type theory with a constructive definite description operator)

```
Definition ConstructiveDefiniteDescription_on :=
  ∀ P:A→Prop,
    (∃! x, P x) → { x:A | P x }.
```

55.2.2 Weakly classical choice and description

GAC_rel = guarded relational form of the (non extensional) axiom of choice

```
Definition GuardedRelationalChoice_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    (∀ x : A, P x → ∃ y : B, R x y) →
    (∃ R' : A→B→Prop,
      subrelation R' R ∧ ∀ x, P x → ∃! y, R' x y).
```

GAC_fun = guarded functional form of the (non extensional) axiom of choice

```
Definition GuardedFunctionalChoice_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    inhabited B →
    (∀ x : A, P x → ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x, P x → R x (f x)).
```

GAC! = guarded functional relation reification

```
Definition GuardedFunctionalRelReification_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    inhabited B →
    (∀ x : A, P x → ∃! y : B, R x y) →
    (∃ f : A→B, ∀ x : A, P x → R x (f x)).
```

OAC_rel = "omniscient" relational form of the (non extensional) axiom of choice

```
Definition OmniscientRelationalChoice_on :=
  ∀ R : A→B→Prop,
    ∃ R' : A→B→Prop,
      subrelation R' R ∧ ∀ x : A, (∃ y : B, R x y) → ∃! y, R' x y.
```

OAC_fun = "omniscient" functional form of the (non extensional) axiom of choice (called AC* in Bell [*Bell*])

```
Definition OmniscientFunctionalChoice_on :=
  ∀ R : A→B→Prop,
    inhabited B →
    ∃ f : A→B, ∀ x : A, (∃ y : B, R x y) → R x (f x).
```

D_epsilon = (weakly classical) indefinite description principle

```
Definition EpsilonStatement_on :=
  ∀ P:A→Prop,
    inhabited A → { x:A | (∃ x, P x) → P x }.
```

D_iota = (weakly classical) definite description principle

```
Definition IotaStatement_on :=
  ∀ P:A→Prop,
    inhabited A → { x:A | (∃! x, P x) → P x }.
End ChoiceSchemes.
```

Generalized schemes

```
Notation RelationalChoice :=
  (∀ A B : Type, RelationalChoice_on A B).
Notation FunctionalChoice :=
  (∀ A B : Type, FunctionalChoice_on A B).
Notation DependentFunctionalChoice :=
  (∀ A (B:A→Type), DependentFunctionalChoice_on B).
Notation InhabitedForallCommute :=
  (∀ A (B : A → Type), InhabitedForallCommute_on B).
Notation FunctionalDependentChoice :=
  (∀ A : Type, FunctionalDependentChoice_on A).
Notation FunctionalCountableChoice :=
  (∀ A : Type, FunctionalCountableChoice_on A).
Notation FunctionalChoiceOnInhabitedSet :=
  (∀ A B : Type, inhabited B → FunctionalChoice_on A B).
Notation FunctionalRelReification :=
  (∀ A B : Type, FunctionalRelReification_on A B).
Notation DependentFunctionalRelReification :=
  (∀ A (B:A→Type), DependentFunctionalRelReification_on B).
Notation RepresentativeFunctionalChoice :=
  (∀ A : Type, RepresentativeFunctionalChoice_on A).
Notation SetoidFunctionalChoice :=
  (∀ A B: Type, SetoidFunctionalChoice_on A B).
Notation GeneralizedSetoidFunctionalChoice :=
  (∀ A B : Type, GeneralizedSetoidFunctionalChoice_on A B).
Notation SimpleSetoidFunctionalChoice :=
  (∀ A B : Type, SimpleSetoidFunctionalChoice_on A B).

Notation GuardedRelationalChoice :=
  (∀ A B : Type, GuardedRelationalChoice_on A B).
Notation GuardedFunctionalChoice :=
  (∀ A B : Type, GuardedFunctionalChoice_on A B).
Notation GuardedFunctionalRelReification :=
  (∀ A B : Type, GuardedFunctionalRelReification_on A B).

Notation OmniscientRelationalChoice :=
  (∀ A B : Type, OmniscientRelationalChoice_on A B).
Notation OmniscientFunctionalChoice :=
  (∀ A B : Type, OmniscientFunctionalChoice_on A B).

Notation ConstructiveDefiniteDescription :=
  (∀ A : Type, ConstructiveDefiniteDescription_on A).
Notation ConstructiveIndefiniteDescription :=
  (∀ A : Type, ConstructiveIndefiniteDescription_on A).

Notation IotaStatement :=
  (∀ A : Type, IotaStatement_on A).
Notation EpsilonStatement :=
  (∀ A : Type, EpsilonStatement_on A).
```

Subclassical schemes

PI = proof irrelevance `Definition` ProofIrrelevance :=
∀ (A:`Prop`) ($a1$ $a2$:A), $a1$ `=` $a2$.

IGP = independence of general premises (an unconstrained generalisation of the constructive principle of independence of premises) `Definition` IndependenceOfGeneralPremises :=
∀ (A:`Type`) (P:$A \to$ `Prop`) (Q:`Prop`),
inhabited $A \to$
`(`$Q \to \exists\ x$`,` $P\ x$`)` $\to \exists\ x$`,` $Q \to P\ x$.

Drinker = drinker's paradox (small form) (called Ex in Bell [*Bell*]) `Definition` SmallDrinker'sParadox :=
∀ (A:`Type`) (P:$A \to$ `Prop`), **inhabited** $A \to$
$\exists\ x$`,` `(`$\exists\ x$`,` $P\ x$`)` $\to P\ x$.

EM = excluded-middle `Definition` ExcludedMiddle :=
∀ P:`Prop`, $P \lor \neg\ P$.

Extensional schemes

Ext_prop_repr = choice of a representative among extensional propositions

Ext_pred_repr = choice of a representative among extensional predicates

Ext_fun_repr = choice of a representative among extensional functions

We let also

- IPL_2 = 2nd-order impredicative minimal predicate logic (with ex. quant.)
- IPL^2 = 2nd-order functional minimal predicate logic (with ex. quant.)
- IPL_2^2 = 2nd-order impredicative, 2nd-order functional minimal pred. logic (with ex. quant.)

with no prerequisite on the non-emptiness of domains

55.3 Table of contents

1. Definitions
2. IPL_2^2 |- AC_rel + AC! = AC_fun
3.1. typed IPL_2 + Sigma-types + PI |- AC_rel = GAC_rel and IPL_2 |- AC_rel + IGP -> GAC_rel and IPL_2 |- GAC_rel = OAC_rel
3.2. IPL^2 |- AC_fun + IGP = GAC_fun = OAC_fun = AC_fun + Drinker
3.3. D_iota -> ID_iota and D_epsilon <-> ID_epsilon + Drinker
4. Derivability of choice for decidable relations with well-ordered codomain
5. AC_fun = AC_fun_dep = AC_trunc
6. Non contradiction of constructive descriptions wrt functional choices
7. Definite description transports classical logic to the computational world
8. Choice -> Dependent choice -> Countable choice
9.1. AC_fun_setoid = AC_fun + Ext_fun_repr + EM
9.2. AC_fun_setoid = AC_fun + Ext_pred_repr + PI

55.4 AC_rel + AC! = AC_fun

We show that the functional formulation of the axiom of Choice (usual formulation in type theory) is equivalent to its relational formulation (only formulation of set theory) + functional relation reification (aka axiom of unique choice, or, principle of (parametric) definite descriptions)

This shows that the axiom of choice can be assumed (under its relational formulation) without known inconsistency with classical logic, though functional relation reification conflicts with classical logic

Lemma functional_rel_reification_and_rel_choice_imp_fun_choice :
  ∀ A B : Type,
    FunctionalRelReification_on A B → RelationalChoice_on A B → FunctionalChoice_on A B.

Lemma fun_choice_imp_rel_choice :
  ∀ A B : Type, FunctionalChoice_on A B → RelationalChoice_on A B.

Lemma fun_choice_imp_functional_rel_reification :
  ∀ A B : Type, FunctionalChoice_on A B → FunctionalRelReification_on A B.

Corollary fun_choice_iff_rel_choice_and_functional_rel_reification :
  ∀ A B : Type, FunctionalChoice_on A B ↔
    RelationalChoice_on A B ∧ FunctionalRelReification_on A B.

55.5 Connection between the guarded, non guarded and omniscient choices

We show that the guarded formulations of the axiom of choice are equivalent to their "omniscient" variant and comes from the non guarded formulation in presence either of the independence of general premises or subset types (themselves derivable from subtypes thanks to proof- irrelevance)

55.5.1 AC_rel + PI -> GAC_rel and AC_rel + IGP -> GAC_rel and GAC_rel = OAC_rel

Lemma rel_choice_and_proof_irrel_imp_guarded_rel_choice :
  RelationalChoice → ProofIrrelevance → GuardedRelationalChoice.

Lemma rel_choice_indep_of_general_premises_imp_guarded_rel_choice :
  ∀ A B : Type, **inhabited** B → RelationalChoice_on A B →
    IndependenceOfGeneralPremises → GuardedRelationalChoice_on A B.

Lemma guarded_rel_choice_imp_rel_choice :
  ∀ A B : Type, GuardedRelationalChoice_on A B → RelationalChoice_on A B.

Lemma subset_types_imp_guarded_rel_choice_iff_rel_choice :
  ProofIrrelevance → (GuardedRelationalChoice ↔ RelationalChoice).

OAC_rel = GAC_rel

Corollary guarded_iff_omniscient_rel_choice :
  GuardedRelationalChoice ↔ OmniscientRelationalChoice.

55.5.2 AC_fun + IGP = GAC_fun = OAC_fun = AC_fun + Drinker

AC_fun + IGP = GAC_fun

```
Lemma guarded_fun_choice_imp_indep_of_general_premises :
  GuardedFunctionalChoice → IndependenceOfGeneralPremises.
```

```
Lemma guarded_fun_choice_imp_fun_choice :
  GuardedFunctionalChoice → FunctionalChoiceOnInhabitedSet.
```

```
Lemma fun_choice_and_indep_general_prem_imp_guarded_fun_choice :
  FunctionalChoiceOnInhabitedSet → IndependenceOfGeneralPremises
  → GuardedFunctionalChoice.
```

```
Corollary fun_choice_and_indep_general_prem_iff_guarded_fun_choice :
  FunctionalChoiceOnInhabitedSet ∧ IndependenceOfGeneralPremises
  ↔ GuardedFunctionalChoice.
```

AC_fun + Drinker = OAC_fun
This was already observed by Bell [*Bell*]

```
Lemma omniscient_fun_choice_imp_small_drinker :
  OmniscientFunctionalChoice → SmallDrinker'sParadox.
```

```
Lemma omniscient_fun_choice_imp_fun_choice :
  OmniscientFunctionalChoice → FunctionalChoiceOnInhabitedSet.
```

```
Lemma fun_choice_and_small_drinker_imp_omniscient_fun_choice :
  FunctionalChoiceOnInhabitedSet → SmallDrinker'sParadox
  → OmniscientFunctionalChoice.
```

```
Corollary fun_choice_and_small_drinker_iff_omniscient_fun_choice :
  FunctionalChoiceOnInhabitedSet ∧ SmallDrinker'sParadox
  ↔ OmniscientFunctionalChoice.
```

OAC_fun = GAC_fun
This is derivable from the intuitionistic equivalence between IGP and Drinker but we give a direct proof

```
Theorem guarded_iff_omniscient_fun_choice :
  GuardedFunctionalChoice ↔ OmniscientFunctionalChoice.
```

55.5.3 D_iota -> ID_iota and D_epsilon <-> ID_epsilon + Drinker

D_iota -> ID_iota

```
Lemma iota_imp_constructive_definite_description :
  IotaStatement → ConstructiveDefiniteDescription.
```

ID_epsilon + Drinker <-> D_epsilon

```
Lemma epsilon_imp_constructive_indefinite_description:
  EpsilonStatement → ConstructiveIndefiniteDescription.
```

```
Lemma constructive_indefinite_description_and_small_drinker_imp_epsilon :
  SmallDrinker'sParadox → ConstructiveIndefiniteDescription →
  EpsilonStatement.
```

```
Lemma epsilon_imp_small_drinker :
  EpsilonStatement → SmallDrinker'sParadox.
```

```
Theorem constructive_indefinite_description_and_small_drinker_iff_epsilon :
  (SmallDrinker'sParadox × ConstructiveIndefiniteDescription →
  EpsilonStatement) ×
  (EpsilonStatement →
   SmallDrinker'sParadox × ConstructiveIndefiniteDescription).
```

55.6 Derivability of choice for decidable relations with well-ordered codomain

Countable codomains, such as *nat*, can be equipped with a well-order, which implies the existence of a least element on inhabited decidable subsets. As a consequence, the relational form of the axiom of choice is derivable on *nat* for decidable relations.

We show instead that functional relation reification and the functional form of the axiom of choice are equivalent on decidable relation with *nat* as codomain

```
Require Import Wf_nat.
Require Import Decidable.
```

```
Lemma classical_denumerable_description_imp_fun_choice :
  ∀ A:Type,
    FunctionalRelReification_on A nat →
    ∀ R:A→nat→Prop,
      (∀ x y, decidable (R x y)) → FunctionalChoice_on_rel R.
```

55.7 AC_fun = AC_fun_dep = AC_trunc

55.7.1 Choice on dependent and non dependent function types are equivalent

The easy part

```
Theorem dep_non_dep_functional_choice :
  DependentFunctionalChoice → FunctionalChoice.
```

Deriving choice on product types requires some computation on singleton propositional types, so we need computational conjunction projections and dependent elimination of conjunction and equality

```
Scheme and_indd := Induction for and Sort Prop.
Scheme eq_indd := Induction for eq Sort Prop.
```

```
Definition proj1_inf (A B:Prop) (p : A∧B) :=
  let (a,b) := p in a.
```

```
Theorem non_dep_dep_functional_choice :
  FunctionalChoice → DependentFunctionalChoice.
```

55.7.2 Functional choice and truncation choice are equivalent

Theorem functional_choice_to_inhabited_forall_commute :
  FunctionalChoice → InhabitedForallCommute.

Theorem inhabited_forall_commute_to_functional_choice :
  InhabitedForallCommute → FunctionalChoice.

55.7.3 Reification of dependent and non dependent functional relation are equivalent

The easy part

Theorem dep_non_dep_functional_rel_reification :
  DependentFunctionalRelReification → FunctionalRelReification.

Deriving choice on product types requires some computation on singleton propositional types, so we need computational conjunction projections and dependent elimination of conjunction and equality

Theorem non_dep_dep_functional_rel_reification :
  FunctionalRelReification → DependentFunctionalRelReification.

Corollary dep_iff_non_dep_functional_rel_reification :
  FunctionalRelReification ↔ DependentFunctionalRelReification.

55.8 Non contradiction of constructive descriptions wrt functional axioms of choice

55.8.1 Non contradiction of indefinite description

Lemma relative_non_contradiction_of_indefinite_descr :
  ∀ C:Prop, (ConstructiveIndefiniteDescription → C)
  → (FunctionalChoice → C).

Lemma constructive_indefinite_descr_fun_choice :
  ConstructiveIndefiniteDescription → FunctionalChoice.

55.8.2 Non contradiction of definite description

Lemma relative_non_contradiction_of_definite_descr :
  ∀ C:Prop, (ConstructiveDefiniteDescription → C)
  → (FunctionalRelReification → C).

Lemma constructive_definite_descr_fun_reification :
  ConstructiveDefiniteDescription → FunctionalRelReification.

Remark, the following corollaries morally hold:

Definition In_propositional_context (A:Type) := forall C:Prop, (A -> C) -> C.

Corollary constructive_definite_descr_in_prop_context_iff_fun_reification : In_propositional_context ConstructiveIndefiniteDescription <-> FunctionalChoice.

Corollary constructive_definite_descr_in_prop_context_iff_fun_reification : In_propositional_context ConstructiveDefiniteDescription <-> FunctionalRelReification.

but expecting *FunctionalChoice* (resp. *FunctionalRelReification*) to be applied on the same Type universes on both sides of the first (resp. second) equivalence breaks the stratification of universes.

55.9 Excluded-middle + definite description => computational excluded-middle

The idea for the following proof comes from [*ChicliPottierSimpson02*]

Classical logic and axiom of unique choice (i.e. functional relation reification), as shown in [*ChicliPottierSimpson02*], implies the double-negation of excluded-middle in `Set` (which is incompatible with the impredicativity of `Set`).

We adapt the proof to show that constructive definite description transports excluded-middle from `Prop` to `Set`.

[*ChicliPottierSimpson02*] Laurent Chicli, Loïc Pottier, Carlos Simpson, Mathematical Quotients and Quotient Types in Coq, Proceedings of TYPES 2002, Lecture Notes in Computer Science 2646, Springer Verlag.

```
Require Import Setoid.
```

```
Theorem constructive_definite_descr_excluded_middle :
  (∀ A : Type, ConstructiveDefiniteDescription_on A) →
  (∀ P:Prop, P ∨ ¬ P) → (∀ P:Prop, {P} + {¬ P}).
```

```
Corollary fun_reification_descr_computational_excluded_middle_in_prop_context :
  FunctionalRelReification →
  (∀ P:Prop, P ∨ ¬ P) →
  ∀ C:Prop, ((∀ P:Prop, {P} + {¬ P}) → C) → C.
```

55.10 Choice => Dependent choice => Countable choice

```
Require Import Arith.
```

```
Theorem functional_choice_imp_functional_dependent_choice :
   FunctionalChoice → FunctionalDependentChoice.
```

```
Theorem functional_dependent_choice_imp_functional_countable_choice :
   FunctionalDependentChoice → FunctionalCountableChoice.
```

55.11 About the axiom of choice over setoids

```
Require Import ClassicalFacts PropExtensionalityFacts.
```

55.11.1 Consequences of the choice of a representative in an equivalence class

```
Theorem repr_fun_choice_imp_ext_prop_repr :
  RepresentativeFunctionalChoice → ExtensionalPropositionRepresentative.
```

Theorem repr_fun_choice_imp_ext_pred_repr :
RepresentativeFunctionalChoice → ExtensionalPredicateRepresentative.

Theorem repr_fun_choice_imp_ext_function_repr :
RepresentativeFunctionalChoice → ExtensionalFunctionRepresentative.

This is a variant of Diaconescu and Goodman-Myhill theorems

Theorem repr_fun_choice_imp_excluded_middle :
RepresentativeFunctionalChoice → ExcludedMiddle.

Theorem repr_fun_choice_imp_relational_choice :
RepresentativeFunctionalChoice → RelationalChoice.

55.11.2 AC_fun_setoid = AC_fun_setoid_gen = AC_fun_setoid_simple

Theorem gen_setoid_fun_choice_imp_setoid_fun_choice :
∀ A B, GeneralizedSetoidFunctionalChoice_on A B → SetoidFunctionalChoice_on A B.

Theorem setoid_fun_choice_imp_gen_setoid_fun_choice :
∀ A B, SetoidFunctionalChoice_on A B → GeneralizedSetoidFunctionalChoice_on A B.

Corollary setoid_fun_choice_iff_gen_setoid_fun_choice :
∀ A B, SetoidFunctionalChoice_on A B ↔ GeneralizedSetoidFunctionalChoice_on A B.

Theorem setoid_fun_choice_imp_simple_setoid_fun_choice :
∀ A B, SetoidFunctionalChoice_on A B → SimpleSetoidFunctionalChoice_on A B.

Theorem simple_setoid_fun_choice_imp_setoid_fun_choice :
∀ A B, SimpleSetoidFunctionalChoice_on A B → SetoidFunctionalChoice_on A B.

Corollary setoid_fun_choice_iff_simple_setoid_fun_choice :
∀ A B, SetoidFunctionalChoice_on A B ↔ SimpleSetoidFunctionalChoice_on A B.

55.11.3 AC_fun_setoid = AC! + AC_fun_repr

Theorem setoid_fun_choice_imp_fun_choice :
∀ A B, SetoidFunctionalChoice_on A B → FunctionalChoice_on A B.

Corollary setoid_fun_choice_imp_functional_rel_reification :
∀ A B, SetoidFunctionalChoice_on A B → FunctionalRelReification_on A B.

Theorem setoid_fun_choice_imp_repr_fun_choice :
SetoidFunctionalChoice → RepresentativeFunctionalChoice .

Theorem functional_rel_reification_and_repr_fun_choice_imp_setoid_fun_choice :
FunctionalRelReification → RepresentativeFunctionalChoice → SetoidFunctionalChoice.

Theorem functional_rel_reification_and_repr_fun_choice_iff_setoid_fun_choice :
FunctionalRelReification ∧ RepresentativeFunctionalChoice ↔ SetoidFunctionalChoice.

Note: What characterization to give of RepresentativeFunctionalChoice? A formulation of it as a functional relation would certainly be equivalent to the formulation of SetoidFunctionalChoice as a functional relation, but in their functional forms, SetoidFunctionalChoice seems strictly stronger

55.12 AC_fun_setoid = AC_fun + Ext_fun_repr + EM

Import *EqNotations.*

55.12.1 This is the main theorem in [*Carlström04*]

Note: all ingredients have a computational meaning when taken in separation. However, to compute with the functional choice, existential quantification has to be thought as a strong existential, which is incompatible with the computational content of excluded-middle

Theorem fun_choice_and_ext_functions_repr_and_excluded_middle_imp_setoid_fun_choice :
FunctionalChoice → ExtensionalFunctionRepresentative → ExcludedMiddle → RepresentativeFunctionalChoice.

Theorem setoid_functional_choice_first_characterization :
FunctionalChoice ∧ ExtensionalFunctionRepresentative ∧ ExcludedMiddle ↔ SetoidFunctionalChoice.

55.12.2 AC_fun_setoid = AC_fun + Ext_pred_repr + PI

Note: all ingredients have a computational meaning when taken in separation. However, to compute with the functional choice, existential quantification has to be thought as a strong existential, which is incompatible with proof-irrelevance which requires existential quantification to be truncated

Theorem fun_choice_and_ext_pred_ext_and_proof_irrel_imp_setoid_fun_choice :
FunctionalChoice → ExtensionalPredicateRepresentative → ProofIrrelevance → RepresentativeFunctionalChoice.

Theorem setoid_functional_choice_second_characterization :
FunctionalChoice ∧ ExtensionalPredicateRepresentative ∧ ProofIrrelevance ↔ SetoidFunctionalChoice.

55.13 Compatibility notations

Notation description_rel_choice_imp_funct_choice :=
functional_rel_reification_and_rel_choice_imp_fun_choice (*only parsing*).

Notation funct_choice_imp_rel_choice := fun_choice_imp_rel_choice (*only parsing*).

Notation FunChoice_Equiv_RelChoice_and_ParamDefinDescr :=
fun_choice_iff_rel_choice_and_functional_rel_reification (*only parsing*).

Notation funct_choice_imp_description := fun_choice_imp_functional_rel_reification (*only parsing*).

Chapter 56

Library Coq.Logic.SetIsType

56.1 The Set universe seen as a synonym for Type

After loading this file, Set becomes just another name for Type. This allows easily performing a Set-to-Type migration, or at least test whether a development relies or not on specific features of Set: simply insert some Require Export of this file at starting points of the development and try to recompile...

`Notation` "'Set'" := `Type` (*only parsing*).

Chapter 57

Library Coq.Logic.ProofIrrelevance

This file axiomatizes proof-irrelevance and derives some consequences

```
Require Import ProofIrrelevanceFacts.
Axiom proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.
Module PI. Definition proof_irrelevance := proof_irrelevance. End PI.
Module ProofIrrelevanceTheory := ProofIrrelevanceTheory(PI).
Export ProofIrrelevanceTheory.
```

Chapter 58

Library Coq.Logic.RelationalChoice

This file axiomatizes the relational form of the axiom of choice

```
Axiom relational_choice :
  ∀ (A B : Type) (R : A→B→Prop),
   (∀ x : A, ∃ y : B, R x y) →
     ∃ R' : A→B→Prop,
       subrelation R' R ∧ ∀ x : A, ∃! y : B, R' x y.
```

Chapter 59

Library Coq.Logic.JMeq

John Major's Equality as proposed by Conor McBride

Reference:

McBride Elimination with a Motive, Proceedings of TYPES 2000, LNCS 2277, pp 197-216, 2002.

```
Set Implicit Arguments.
```

```
Inductive JMeq (A:Type) (x:A) : ∀ B:Type, B → Prop :=
    JMeq_refl : JMeq x x.
```

```
#[global]
Hint Resolve JMeq_refl : core.
```

```
Definition JMeq_hom {A : Type} (x y : A) := JMeq x y.
```

```
Lemma JMeq_sym : ∀ (A B:Type) (x:A) (y:B), JMeq x y → JMeq y x.
#[global]
Hint Immediate JMeq_sym : core.
```

```
Lemma JMeq_trans :
 ∀ (A B C:Type) (x:A) (y:B) (z:C), JMeq x y → JMeq y z → JMeq x z.
```

```
Axiom JMeq_eq : ∀ (A:Type) (x y:A), JMeq x y → x = y.
```

```
Lemma JMeq_ind : ∀ (A:Type) (x:A) (P:A → Prop),
  P x → ∀ y, JMeq x y → P y.
```

```
Lemma JMeq_rec : ∀ (A:Type) (x:A) (P:A → Set),
  P x → ∀ y, JMeq x y → P y.
```

```
Lemma JMeq_rect : ∀ (A:Type) (x:A) (P:A→Type),
  P x → ∀ y, JMeq x y → P y.
```

```
Lemma JMeq_ind_r : ∀ (A:Type) (x:A) (P:A → Prop),
   P x → ∀ y, JMeq y x → P y.
```

```
Lemma JMeq_rec_r : ∀ (A:Type) (x:A) (P:A → Set),
   P x → ∀ y, JMeq y x → P y.
```

Lemma JMeq_rect_r : $\forall$ (A:Type) (x:A) (P:$A \rightarrow$ Type),
$P\ x \rightarrow \forall\ y$, **JMeq** $y\ x \rightarrow P\ y$.

Lemma JMeq_congr :
$\forall$ (A:Type) (x:A) (B:Type) (f:$A \rightarrow B$) (y:A), **JMeq** $x\ y \rightarrow f\ x = f\ y$.

JMeq is equivalent to *eq_dep* Type (fun $X \Rightarrow X$)

Require Import Eqdep.

Lemma JMeq_eq_dep_id :
$\forall$ (A B:Type) (x:A) (y:B), **JMeq** $x\ y \rightarrow$ **eq_dep** Type (fun $X \Rightarrow X$) $A\ x\ B\ y$.

Lemma eq_dep_id_JMeq :
$\forall$ (A B:Type) (x:A) (y:B), **eq_dep** Type (fun $X \Rightarrow X$) $A\ x\ B\ y \rightarrow$ **JMeq** $x\ y$.

eq_dep $U\ P\ p\ x\ q\ y$ is strictly finer than *JMeq* $(P\ p)\ x\ (P\ q)\ y$

Lemma eq_dep_JMeq :
$\forall\ U\ P\ p\ x\ q\ y$, **eq_dep** $U\ P\ p\ x\ q\ y \rightarrow$ **JMeq** $x\ y$.

Lemma eq_dep_strictly_stronger_JMeq :
$\exists\ U\ P\ p\ q\ x\ y$, **JMeq** $x\ y \land \neg$ **eq_dep** $U\ P\ p\ x\ q\ y$.

However, when the dependencies are equal, *JMeq* $(P\ p)\ x\ (P\ q)\ y$ is as strong as *eq_dep* $U\ P\ p\ x\ q\ y$ (this uses *JMeq_eq*)

Lemma JMeq_eq_dep :
$\forall\ U$ (P:$U \rightarrow$Type) $p\ q$ (x:$P\ p$) (y:$P\ q$),
$p = q \rightarrow$ **JMeq** $x\ y \rightarrow$ **eq_dep** $U\ P\ p\ x\ q\ y$.

Notation sym_JMeq := JMeq_sym (*only parsing*).
Notation trans_JMeq := JMeq_trans (*only parsing*).

Chapter 60

Library Coq.Logic.Berardi

This file formalizes Berardi's paradox which says that in the calculus of constructions, excluded middle (EM) and axiom of choice (AC) imply proof irrelevance (PI). Here, the axiom of choice is not necessary because of the use of inductive types.

```
@article{Barbanera-Berardi:JFP96,
   author    = {F. Barbanera and S. Berardi},
   title     = {Proof-irrelevance out of Excluded-middle and Choice
                in the Calculus of Constructions},
   journal   = {Journal of Functional Programming},
   year      = {1996},
   volume    = {6},
   number    = {3},
   pages     = {519-525}
}
```

Set Implicit Arguments.

Section Berardis_paradox.

Excluded middle Hypothesis *EM* : ∀ *P*:Prop, *P* ∨ ¬ *P*.

Conditional on any proposition. Definition IFProp (*P* *B*:Prop) (*e1* *e2*:*P*) :=
match *EM* *B* with
| or_introl _ ⇒ *e1*
| or_intror _ ⇒ *e2*
end.

Axiom of choice applied to disjunction. Provable in Coq because of dependent elimination.
Lemma AC_IF :
∀ (*P* *B*:Prop) (*e1* *e2*:*P*) (*Q*:*P* → Prop),
(*B* → *Q* *e1*) → (¬ *B* → *Q* *e2*) → *Q* (IFProp *B* *e1* *e2*).

We assume a type with two elements. They play the role of booleans. The main theorem under the current assumptions is that *T*=*F* Variable *Bool* : Prop.
Variable *T* : *Bool*.
Variable *F* : *Bool*.

The powerset operator `Definition` pow (P:Prop) := $P \to Bool$.

A piece of theory about retracts `Section` Retracts.

`Variables` A B : Prop.

`Record` **retract** : Prop :=
{i : $A \to B$; j : $B \to A$; inv : ∀ a:A, j (i a) = a}.
`Record` **retract_cond** : Prop :=
{i2 : $A \to B$; j2 : $B \to A$; inv2 : **retract** → ∀ a:A, j2 (i2 a) = a}.

The dependent elimination above implies the axiom of choice:

`Lemma` AC : ∀ r:**retract_cond**, **retract** → ∀ a:A, j2 r (i2 r a) = a.

`End` Retracts.

This lemma is basically a commutation of implication and existential quantification: (EX x | A -> P(x)) <=> (A -> EX x | P(x)) which is provable in classical logic (=> is already provable in intuitionistic logic).

`Lemma` L1 : ∀ A B:Prop, **retract_cond** (pow A) (pow B).

The paradoxical set `Definition` U := ∀ P:Prop, pow P.

Bijection between U and $(pow\ U)$ `Definition` f (u:U) : pow U := u U.

`Definition` g (h:pow U) : U :=
`fun` X ⇒ `let` lX := j2 (L1 X U) `in` `let` rU := i2 (L1 U U) `in` lX (rU h).

We deduce that the powerset of U is a retract of U. This lemma is stated in Berardi's article, but is not used afterwards. `Lemma` retract_pow_U_U : **retract** (pow U) U.

Encoding of Russel's paradox
The boolean negation. `Definition` Not_b (b:$Bool$) := IFProp (b = T) F T.

the set of elements not belonging to itself `Definition` R : U := g (`fun` u:U ⇒ Not_b (u U u)).

`Lemma` not_has_fixpoint : R R = Not_b (R R).

`Theorem` classical_proof_irrelevance : T = F.

#[*deprecated*(*since* = "8.8", *note* = "Use classical_proof_irrelevance instead.")]
`Notation` classical_proof_irrelevence := classical_proof_irrelevance.

`End` Berardis_paradox.

Chapter 61

Library Coq.Logic.FinFun

61.1 Functions on finite domains

Main result : for functions f:$A \to A$ with finite A, f injective <-> f bijective <-> f surjective.

```
Require Import List Compare_dec EqNat Decidable ListDec. Require Fin.
Set Implicit Arguments.
```

General definitions

```
Definition Injective {A B} (f : A→B) :=
 ∀ x y, f x = f y → x = y.
Definition Surjective {A B} (f : A→B) :=
 ∀ y, ∃ x, f x = y.
Definition Bijective {A B} (f : A→B) :=
 ∃ g:B→A, (∀ x, g (f x) = x) ∧ (∀ y, f (g y) = y).
```

Finiteness is defined here via exhaustive list enumeration

```
Definition Full {A:Type} (l:list A) := ∀ a:A, In a l.
Definition Finite (A:Type) := ∃ (l:list A), Full l.
```

In many following proofs, it will be convenient to have list enumerations without duplicates. As soon as we have decidability of equality (in Prop), this is equivalent to the previous notion.

```
Definition Listing {A:Type} (l:list A) := NoDup l ∧ Full l.
Definition Finite' (A:Type) := ∃ (l:list A), Listing l.
Lemma Finite_alt A (d:decidable_eq A) : Finite A ↔ Finite' A.
```

Injections characterized in term of lists

```
Lemma Injective_map_NoDup A B (f:A→B) (l:list A) :
 Injective f → NoDup l → NoDup (map f l).
Lemma Injective_list_carac A B (d:decidable_eq A)(f:A→B) :
  Injective f ↔ (∀ l, NoDup l → NoDup (map f l)).
Lemma Injective_carac A B (l:list A) : Listing l →
   ∀ (f:A→B), Injective f ↔ NoDup (map f l).
```

Surjection characterized in term of lists

Lemma Surjective_list_carac A B (f:A→B):
  Surjective f ↔ (∀ lB, ∃ lA, incl lB (map f lA)).

Lemma Surjective_carac A B : Finite B → decidable_eq B →
  ∀ f:A→B, Surjective f ↔ (∃ lA, Listing (map f lA)).

Main result :

Lemma Endo_Injective_Surjective :
 ∀ A, Finite A → decidable_eq A →
  ∀ f:A→A, Injective f ↔ Surjective f.

An injective and surjective function is bijective. We need here stronger hypothesis : decidability of equality in Type.

Definition EqDec (A:Type) := ∀ x y:A, {x=y}+{x≠y}.

First, we show that a surjective f has an inverse function g such that f.g = id.

Lemma Finite_Empty_or_not A :
  Finite A → (A→**False**) ∨ ∃ a:A, **True**.

Lemma Surjective_inverse :
  ∀ A B, Finite A → EqDec B →
   ∀ f:A→B, Surjective f →
    ∃ g:B→A, ∀ x, f (g x) = x.

Same, with more knowledge on the inverse function: g.f = f.g = id

Lemma Injective_Surjective_Bijective :
 ∀ A B, Finite A → EqDec B →
  ∀ f:A→B, Injective f → Surjective f → Bijective f.

An example of finite type : *Fin.t*

Lemma Fin_Finite n : Finite (**Fin.t** n).

Instead of working on a finite subset of nat, another solution is to use restricted *nat*→*nat* functions, and to consider them only below a certain bound n.

Definition bFun n (f:**nat**→**nat**) := ∀ x, x < n → f x < n.

Definition bInjective n (f:**nat**→**nat**) :=
 ∀ x y, x < n → y < n → f x = f y → x = y.

Definition bSurjective n (f:**nat**→**nat**) :=
 ∀ y, y < n → ∃ x, x < n ∧ f x = y.

We show that this is equivalent to the use of *Fin.t* n.

Module FIN2RESTRICT.

Notation n2f := Fin.of_nat_lt.
Definition f2n {n} (x:**Fin.t** n) := proj1_sig (Fin.to_nat x).
Definition f2n_ok n (x:**Fin.t** n) : f2n x < n := proj2_sig (Fin.to_nat x).
Definition n2f_f2n : ∀ n x, n2f (f2n_ok x) = x := @Fin.of_nat_to_nat_inv.
Definition f2n_n2f x n h : f2n (n2f h) = x := f_equal (@proj1_sig _ _) (@Fin.to_nat_of_nat x n h).
Definition n2f_ext : ∀ x n h h', n2f h = n2f h' := @Fin.of_nat_ext.

```
Definition f2n_inj : ∀ n x y, f2n x = f2n y → x = y := @Fin.to_nat_inj.

Definition extend n (f:Fin.t n → Fin.t n) : (nat→nat) :=
 fun x ⇒
   match le_lt_dec n x with
     | left _ ⇒ 0
     | right h ⇒ f2n (f (n2f h))
   end.

Definition restrict n (f:nat→nat)(hf : bFun n f) : (Fin.t n → Fin.t n) :=
 fun x ⇒ let (x',h) := Fin.to_nat x in n2f (hf _ h).

Ltac break_dec H :=
 let H' := fresh "H" in
 destruct le_lt_dec as [H'|H'];
  [elim (Lt.le_not_lt _ _ H' H)
  |try rewrite (n2f_ext H' H) in *; try clear H'].

Lemma extend_ok n f : bFun n (@extend n f).

Lemma extend_f2n n f (x:Fin.t n) : extend f (f2n x) = f2n (f x).

Lemma extend_n2f n f x (h:x<n) : n2f (extend_ok f h) = f (n2f h).

Lemma restrict_f2n n f hf (x:Fin.t n) :
 f2n (@restrict n f hf x) = f (f2n x).

Lemma restrict_n2f n f hf x (h:x<n) :
 @restrict n f hf (n2f h) = n2f (hf _ h).

Lemma extend_surjective n f :
  bSurjective n (@extend n f) ↔ Surjective f.

Lemma extend_injective n f :
  bInjective n (@extend n f) ↔ Injective f.

Lemma restrict_surjective n f h :
  Surjective (@restrict n f h) ↔ bSurjective n f.

Lemma restrict_injective n f h :
  Injective (@restrict n f h) ↔ bInjective n f.

End Fin2Restrict.
Import Fin2Restrict.
```

We can now use Proof via the equivalence ...

```
Lemma bInjective_bSurjective n (f:nat→nat) :
 bFun n f → (bInjective n f ↔ bSurjective n f).

Lemma bSurjective_bBijective n (f:nat→nat) :
 bFun n f → bSurjective n f →
 ∃ g, bFun n g ∧ ∀ x, x < n → g (f x) = x ∧ f (g x) = x.
```

Chapter 62

Library Coq.Logic.FunctionalExtensionality

This module states the axiom of (dependent) functional extensionality and (dependent) eta-expansion. It introduces a tactic `extensionality` to apply the axiom of extensionality to an equality goal.

The converse of functional extensionality.

```
Lemma equal_f : ∀ {A B : Type} {f g : A → B},
  f = g → ∀ x, f x = g x.

Lemma equal_f_dep : ∀ {A B} {f g : ∀ (x : A), B x},
  f = g → ∀ x, f x = g x.
```

Statements of functional extensionality for simple and dependent functions.

```
Axiom functional_extensionality_dep : ∀ {A} {B : A → Type},
  ∀ (f g : ∀ x : A, B x),
  (∀ x, f x = g x) → f = g.

Lemma functional_extensionality {A B} (f g : A → B) :
  (∀ x, f x = g x) → f = g.
```

Extensionality of ∀s follows from functional extensionality.

```
Lemma forall_extensionality {A} {B C : A → Type} (H : ∀ x : A, B x = C x)
: (∀ x, B x) = (∀ x, C x).

Lemma forall_extensionalityP {A} {B C : A → Prop} (H : ∀ x : A, B x = C x)
: (∀ x, B x) = (∀ x, C x).

Lemma forall_extensionalityS {A} {B C : A → Set} (H : ∀ x : A, B x = C x)
: (∀ x, B x) = (∀ x, C x).
```

A version of *functional_extensionality_dep* which is provably equal to *eq_refl* on `fun _ ⇒ eq_refl`

```
Definition functional_extensionality_dep_good
           {A} {B : A → Type}
           (f g : ∀ x : A, B x)
           (H : ∀ x, f x = g x)
  : f = g
  := eq_trans (eq_sym (functional_extensionality_dep f f (fun _ ⇒ eq_refl)))
```

```
                  (functional_extensionality_dep f g H).
Lemma functional_extensionality_dep_good_refl {A B} f
  : @functional_extensionality_dep_good A B f f (fun _ ⇒ eq_refl) = eq_refl.
Opaque functional_extensionality_dep_good.
Lemma forall_sig_eq_rect
        {A B} (f : ∀ a : A, B a)
        (P : { g : _ | (∀ a, f a = g a) } → Type)
        (k : P (exist (fun g ⇒ ∀ a, f a = g a) f (fun a ⇒ eq_refl)))
        g
: P g.
Definition forall_eq_rect
        {A B} (f : ∀ a : A, B a)
        (P : ∀ g, (∀ a, f a = g a) → Type)
        (k : P f (fun a ⇒ eq_refl))
        g H
  : P g H
  := @forall_sig_eq_rect A B f (fun g ⇒ P (proj1_sig g) (proj2_sig g)) k (exist _ g H).
Definition forall_eq_rect_comp {A B} f P k
  : @forall_eq_rect A B f P k f (fun _ ⇒ eq_refl) = k.
Definition f_equal__functional_extensionality_dep_good
             {A B f g} H a
  : f_equal (fun h ⇒ h a) (@functional_extensionality_dep_good A B f g H) = H a.
Definition f_equal__functional_extensionality_dep_good__fun
             {A B f g} H
  : (fun a ⇒ f_equal (fun h ⇒ h a) (@functional_extensionality_dep_good A B f g H)) = H.
```

Apply *functional_extensionality*, introducing variable x.

```
Tactic Notation "extensionality" ident(x) :=
  match goal with
    [ ⊢ ?X = ?Y ] ⇒
    (apply (@functional_extensionality _ _ X Y) ||
     apply (@functional_extensionality_dep _ _ X Y) ||
     apply forall_extensionalityP ||
     apply forall_extensionalityS ||
     apply forall_extensionality) ; intro x
  end.
```

Iteratively apply *functional_extensionality* on an hypothesis until finding an equality statement

```
Ltac extensionality_in_checker tac :=
  first [ tac tt | fail 1 "Anomaly: Unexpected error in extensionality tactic. Please report." ].
Tactic Notation "extensionality" "in" hyp(H) :=
  let rec check_is_extensional_equality H :=
      lazymatch type of H with
      | _ = _ ⇒ constr:(Prop)
      | ∀ a : ?A, ?T
```

```
        ⇒ let Ha := fresh in
           constr:(∀ a : A, match H a with Ha ⇒ ltac:(let v := check_is_extensional_equality
Ha in exact v) end)
      end in
  let assert_is_extensional_equality H :=
      first [ let dummy := check_is_extensional_equality H in idtac
            | fail 1 "Not an extensional equality" ] in
  let assert_not_intensional_equality H :=
      lazymatch type of H with
      | _ = _ ⇒ fail "Already an intensional equality"
      | _ ⇒ idtac
      end in
  let enforce_no_body H :=
      (tryif (let dummy := (eval unfold H in H) in idtac)
        then clearbody H
        else idtac) in
  let rec extensionality_step_make_type H :=
      lazymatch type of H with
      | ∀ a : ?A, ?f = ?g
        ⇒ constr:({ H' | (fun a ⇒ f_equal (fun h ⇒ h a) H') = H })
      | ∀ a : ?A, _
        ⇒ let H' := fresh in
           constr:(∀ a : A, match H a with H' ⇒ ltac:(let ret := extensionality_step_make_type
H' in exact ret) end)
      end in
  let rec eta_contract T :=
      lazymatch (eval cbv beta in T) with
      | context T'[fun a : ?A ⇒ ?f a]
        ⇒ let T'' := context T'[f] in
           eta_contract T''
      | ?T ⇒ T
      end in
  let rec lift_sig_extensionality H :=
      lazymatch type of H with
      | sig _ ⇒ H
      | ∀ a : ?A, _
        ⇒ let Ha := fresh in
           let ret := constr:(fun a : A ⇒ match H a with Ha ⇒ ltac:(let v := lift_sig_extensionality
Ha in exact v) end) in
           lazymatch type of ret with
           | ∀ a : ?A, sig (fun b : ?B ⇒ @?f a b = @?g a b)
             ⇒ eta_contract (exist (fun b : (∀ a : A, B) ⇒ (fun a : A ⇒ f a (b a)) = (fun a :
A ⇒ g a (b a)))
                                                (fun a : A ⇒ proj1_sig (ret a))
                                                (@functional_extensionality_dep_good _ _ _ _ (fun a : A
```

```
⇒ proj2_sig (ret a))))
           end
       end in
   let extensionality_pre_step H H_out Heq :=
       let T := extensionality_step_make_type H in
       let H' := fresh in
       assert (H' : T) by (intros; eexists; apply f_equal__functional_extensionality_dep_good__fun);
       let H''b := lift_sig_extensionality H' in
       case H''b; clear H';
       intros H_out Heq in
   let rec extensionality_rec H H_out Heq :=
       lazymatch type of H with
       | ∀ a, _ = _
         ⇒ extensionality_pre_step H H_out Heq
       | _
         ⇒ let pre_H_out' := fresh H_out in
            let H_out' := fresh pre_H_out' in
            extensionality_pre_step H H_out' Heq;
            let Heq' := fresh Heq in
            extensionality_rec H_out' H_out Heq';
            subst H_out'
       end in
   first [ assert_is_extensional_equality H | fail 1 "Not an extensional equality" ];
   first [ assert_not_intensional_equality H | fail 1 "Already an intensional equality" ];
   (tryif enforce_no_body H then idtac else clearbody H);
   let H_out := fresh in
   let Heq := fresh "Heq" in
   extensionality_in_checker ltac:(fun tt ⇒ extensionality_rec H H_out Heq);

   destruct Heq; rename H_out into H.
```

Eta expansion is built into Coq.

```
Lemma eta_expansion_dep {A} {B : A → Type} (f : ∀ x : A, B x) :
  f = fun x ⇒ f x.

Lemma eta_expansion {A B} (f : A → B) : f = fun x ⇒ f x.
```

Chapter 63

Library Coq.Logic.Classical_Prop

Classical Propositional Logic

Require Import ClassicalFacts.

#[*global*]
Hint Unfold not: *core*.

Axiom *classic* : $\forall$ P:Prop, $P \lor \neg P$.

Lemma NNPP : $\forall$ p:Prop, $\neg \neg p \to p$.

Peirce's law states $\forall$ P Q:Prop, $((P \to Q) \to P) \to P$. Thanks to $\forall$ P, $False \to P$, it is equivalent to the following form

Lemma Peirce : $\forall$ P:Prop, $((P \to$ **False**$) \to P) \to P$.

Lemma not_imply_elim : $\forall$ P Q:Prop, $\neg (P \to Q) \to P$.

Lemma not_imply_elim2 : $\forall$ P Q:Prop, $\neg (P \to Q) \to \neg Q$.

Lemma imply_to_or : $\forall$ P Q:Prop, $(P \to Q) \to \neg P \lor Q$.

Lemma imply_to_and : $\forall$ P Q:Prop, $\neg (P \to Q) \to P \land \neg Q$.

Lemma or_to_imply : $\forall$ P Q:Prop, $\neg P \lor Q \to P \to Q$.

Lemma not_and_or : $\forall$ P Q:Prop, $\neg (P \land Q) \to \neg P \lor \neg Q$.

Lemma or_not_and : $\forall$ P Q:Prop, $\neg P \lor \neg Q \to \neg (P \land Q)$.

Lemma not_or_and : $\forall$ P Q:Prop, $\neg (P \lor Q) \to \neg P \land \neg Q$.

Lemma and_not_or : $\forall$ P Q:Prop, $\neg P \land \neg Q \to \neg (P \lor Q)$.

Lemma imply_and_or : $\forall$ P Q:Prop, $(P \to Q) \to P \lor Q \to Q$.

Lemma imply_and_or2 : $\forall$ P Q R:Prop, $(P \to Q) \to P \lor R \to Q \lor R$.

Lemma proof_irrelevance : $\forall$ (P:Prop) ($p1$ $p2$:P), $p1$ = $p2$.

Ltac *classical_right* := match goal with
$\vdash$?X $\lor$ _ $\Rightarrow$ (elim (*classic* X);intro;[left;trivial|right])
end.

Ltac *classical_left* := match goal with

$\vdash$ _ $\vee$?X $\Rightarrow$ (elim (*classic* X);intro;[right;trivial|left])
end.

Require Export EqdepFacts.

Module Eq_rect_eq.

Lemma eq_rect_eq :
$\forall$ (U:Type) (p:U) (Q:U $\rightarrow$ Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

End Eq_rect_eq.

Module EqdepTheory := EqdepTheory(Eq_rect_eq).
Export *EqdepTheory*.

Chapter 64

Library Coq.Logic.ClassicalChoice

This file provides classical logic and functional choice; this especially provides both indefinite descriptions and choice functions but this is weaker than providing epsilon operator and classical logic as the indefinite descriptions provided by the axiom of choice can be used only in a propositional context (especially, they cannot be used to build choice functions outside the scope of a theorem proof)

This file extends ClassicalUniqueChoice.v with full choice. As ClassicalUniqueChoice.v, it implies the double-negation of excluded-middle in `Set` and leads to a classical world populated with non computable functions. Especially it conflicts with the impredicativity of `Set`, knowing that *true*≠*false* in `Set`.

```
Require Export ClassicalUniqueChoice.
Require Export RelationalChoice.
Require Import ChoiceFacts.

Set Implicit Arguments.

Definition subset (U:Type) (P Q:U→Prop) : Prop := ∀ x, P x → Q x.

Theorem singleton_choice :
  ∀ (A : Type) (P : A→Prop),
  (∃ x : A, P x) → ∃ P' : A→Prop, subset P' P ∧ ∃! x, P' x.

Theorem choice :
 ∀ (A B : Type) (R : A→B→Prop),
   (∀ x : A, ∃ y : B, R x y) →
    ∃ f : A→B, (∀ x : A, R x (f x)).
```

Chapter 65

Library Coq.Logic.ClassicalEpsilon

This file provides classical logic and indefinite description under the form of Hilbert's epsilon operator

Hilbert's epsilon operator and classical logic implies excluded-middle in Set and leads to a classical world populated with non computable functions. It conflicts with the impredicativity of Set

```
Require Export Classical.
Require Import ChoiceFacts.

Set Implicit Arguments.

Axiom constructive_indefinite_description :
  ∀ (A : Type) (P : A→Prop),
    (∃ x, P x) → { x : A | P x }.

Lemma constructive_definite_description :
  ∀ (A : Type) (P : A→Prop),
    (∃! x, P x) → { x : A | P x }.

Theorem excluded_middle_informative : ∀ P:Prop, {P} + {¬ P}.

Theorem classical_indefinite_description :
  ∀ (A : Type) (P : A→Prop), inhabited A →
    { x : A | (∃ x, P x) → P x }.
```

Hilbert's epsilon operator

```
Definition epsilon (A : Type) (i:inhabited A) (P : A→Prop) : A
  := proj1_sig (classical_indefinite_description P i).

Definition epsilon_spec (A : Type) (i:inhabited A) (P : A→Prop) :
  (∃ x, P x) → P (epsilon i P)
  := proj2_sig (classical_indefinite_description P i).
```

Open question: is classical_indefinite_description constructively provable from *relational_choice* and *constructive_definite_description* (at least, using the fact that *functional_choice* is provable from *relational_choice* and *unique_choice*, we know that the double negation of *classical_indefinite_description* is provable (see *relative_non_contradiction_of_indefinite_desc*).

A proof that if P is inhabited, *epsilon a P* does not depend on the actual proof that the domain of P is inhabited (proof idea kindly provided by Pierre Castéran)

```
Lemma epsilon_inh_irrelevance :
  ∀ (A:Type) (i j : inhabited A) (P:A→Prop),
  (∃ x, P x) → epsilon i P = epsilon j P.
```

```
Opaque epsilon.
```

Weaker lemmas (compatibility lemmas)

```
Theorem choice :
 ∀ (A B : Type) (R : A→B→Prop),
   (∀ x : A, ∃ y : B, R x y) →
   (∃ f : A→B, ∀ x : A, R x (f x)).
```

Chapter 66

Library Coq.Logic.WKL

A constructive proof of a version of Weak König's Lemma over a decidable predicate in the formulation of which infinite paths are treated as predicates. The representation of paths as relations avoid the need for classical logic and unique choice. The decidability condition is sufficient to ensure that some required instance of double negation for disjunction of finite paths holds.

The idea of the proof comes from the proof of the weak König's lemma from separation in second-order arithmetic.

Notice that we do not start from a tree but just from an arbitrary predicate. Original Weak Konig's Lemma is the instantiation of the lemma to a tree

```
Require Import WeakFan List.
Import ListNotations.

Require Import Arith.
```

is_path_from P n l means that there exists a path of length n from l on which P does not hold

```
Inductive is_path_from (P:list bool → Prop) : nat → list bool → Prop :=
| here l : ¬ P l → is_path_from P 0 l
| next_left l n : ¬ P l → is_path_from P n (true::l) → is_path_from P (S n) l
| next_right l n : ¬ P l → is_path_from P n (false::l) → is_path_from P (S n) l.
```

We give the characterization of is_path_from in terms of a more common arithmetical formula

```
Proposition is_path_from_characterization P n l :
  is_path_from P n l ↔ ∃ l', length l' = n ∧ ∀ n', n'≤n → ¬ P (rev (firstn n' l') ++ l).
```

infinite_from P l means that we can find arbitrary long paths along which P does not hold above l

```
Definition infinite_from (P:list bool → Prop) l := ∀ n, is_path_from P n l.
```

has_infinite_path P means that there is an infinite path (represented as a predicate) along which P does not hold at all

```
Definition has_infinite_path (P:list bool → Prop) :=
  ∃ (X:nat → Prop), ∀ l, approx X l → ¬ P l.
```

inductively_barred_at P n l means that P eventually holds above l after at most n steps upwards

```
Inductive inductively_barred_at (P:list bool → Prop) : nat → list bool → Prop :=
| now_at l n : P l → inductively_barred_at P n l
```

```
| propagate_at l n :
    inductively_barred_at P n (true::l) →
    inductively_barred_at P n (false::l) →
    inductively_barred_at P (S n) l.
```

The proof proceeds by building a set Y of finite paths approximating either the smallest unbarred infinite path in P, if there is one (taking *true*>*false*), or the path true::true::... if P happens to be inductively_barred

```
Fixpoint Y P (l:list bool) :=
  match l with
  | [] ⇒ True
  | b::l ⇒
      Y P l ∧
      if b then ∃ n, inductively_barred_at P n (false::l) else infinite_from P (false::l)
  end.

Require Import Compare_dec Le Lt.

Lemma is_path_from_restrict : ∀ P n n' l, n ≤ n' →
  is_path_from P n' l → is_path_from P n l.

Lemma inductively_barred_at_monotone : ∀ P l n n', n' ≤ n →
  inductively_barred_at P n' l → inductively_barred_at P n l.

Definition demorgan_or (P:list bool → Prop) l l' := ¬ (P l ∧ P l') → ¬ P l ∨ ¬ P l'.

Definition demorgan_inductively_barred_at P :=
  ∀ n l, demorgan_or (inductively_barred_at P n) (true::l) (false::l).

Lemma inductively_barred_at_imp_is_path_from :
  ∀ P, demorgan_inductively_barred_at P → ∀ n l,
  ¬ inductively_barred_at P n l → is_path_from P n l.

Lemma is_path_from_imp_inductively_barred_at : ∀ P n l,
   is_path_from P n l → inductively_barred_at P n l → False.

Lemma find_left_path : ∀ P l n,
  is_path_from P (S n) l → inductively_barred_at P n (false :: l) → is_path_from P n (true
:: l).

Lemma Y_unique : ∀ P, demorgan_inductively_barred_at P →
  ∀ l1 l2, length l1 = length l2 → Y P l1 → Y P l2 → l1 = l2.
```

X is the translation of Y as a predicate

```
Definition X P n := ∃ l, length l = n ∧ Y P (true::l).

Lemma Y_approx : ∀ P, demorgan_inductively_barred_at P →
  ∀ l, approx (X P) l → Y P l.
```

Main theorem

```
Theorem PreWeakKonigsLemma : ∀ P,
  demorgan_inductively_barred_at P → infinite_from P [] → has_infinite_path P.

Lemma inductively_barred_at_decidable :
```

$\forall\ P,\ (\forall\ l,\ P\ l \lor \neg\ P\ l) \to \forall\ n\ l,$ **inductively_barred_at** $P\ n\ l \lor \neg$ **inductively_barred_at** $P\ n\ l.$

Lemma inductively_barred_at_is_path_from_decidable :
$\forall\ P,\ (\forall\ l,\ P\ l \lor \neg\ P\ l) \to$ demorgan_inductively_barred_at P.

Main corollary

Corollary WeakKonigsLemma : $\forall\ P,\ (\forall\ l,\ P\ l \lor \neg\ P\ l) \to$
infinite_from P [] $\to$ has_infinite_path P.

Chapter 67

Library Coq.Logic.Eqdep_dec

We prove that there is only one proof of $x=x$, i.e *eq_refl* x. This holds if the equality upon the set of x is decidable. A corollary of this theorem is the equality of the right projections of two equal dependent pairs.

Author: Thomas Kleymann |<tms@dcs.ed.ac.uk>| in Lego adapted to Coq by B. Barras

Credit: Proofs up to *K_dec* follow an outline by Michael Hedberg

Table of contents:

1. Streicher's K and injectivity of dependent pair hold on decidable types

1.1. Definition of the functor that builds properties of dependent equalities from a proof of decidability of equality for a set in Type

1.2. Definition of the functor that builds properties of dependent equalities from a proof of decidability of equality for a set in Set

67.1 Streicher's K and injectivity of dependent pair hold on decidable types

```
Set Implicit Arguments.

Section EqdepDec.

  Variable A : Type.

  Let comp (x y y':A) (eq1:x = y) (eq2:x = y') : y = y' :=
    eq_ind _ (fun a ⇒ a = y') eq2 _ eq1.

  Remark trans_sym_eq (x y:A) (u:x = y) : comp u u = eq_refl y.

  Variable x : A.

  Variable eq_dec : ∀ y:A, x = y ∨ x ≠ y.

  Let nu (y:A) (u:x = y) : x = y :=
    match eq_dec y with
      | or_introl eqxy ⇒ eqxy
      | or_intror neqxy ⇒ False_ind _ (neqxy u)
    end.

  Let nu_constant (y:A) (u v:x = y) : nu u = nu v.
```

Qed.

Let *nu_inv* (*y*:*A*) (*v*:*x* = *y*) : *x* = *y* := *comp* (*nu* (eq_refl *x*)) *v*.

Remark nu_left_inv_on (*y*:*A*) (*u*:*x* = *y*) : *nu_inv* (*nu* *u*) = *u*.

Theorem eq_proofs_unicity_on (*y*:*A*) (*p1* *p2*:*x* = *y*) : *p1* = *p2*.

Theorem K_dec_on (*P*:*x* = *x* → Prop) (*H*:*P* (eq_refl *x*)) (*p*:*x* = *x*) : *P* *p*.

The corollary

```
Let proj (P:A → Prop) (exP:ex P) (def:P x) : P x :=
  match exP with
    | ex_intro _ x' prf ⇒
      match eq_dec x' with
        | or_introl eqprf ⇒ eq_ind x' P prf x (eq_sym eqprf)
        | _ ⇒ def
      end
  end.
```

Theorem inj_right_pair_on (*P*:*A* → Prop) (*y* *y'*:*P* *x*) :
ex_intro *P* *x* *y* = ex_intro *P* *x* *y'* → *y* = *y'*.

End EqdepDec.

Now we prove the versions that require decidable equality for the entire type rather than just on the given element. The rest of the file uses this total decidable equality. We could do everything using decidable equality at a point (because the induction rule for *eq* is really an induction rule for $\{ y : A \mid x = y \}$), but we don't currently, because changing everything would break backward compatibility and no-one has yet taken the time to define the pointed versions, and then re-define the non-pointed versions in terms of those.

Theorem eq_proofs_unicity *A* (*eq_dec* : ∀ *x* *y* : *A*, *x* = *y* ∨ *x* ≠ *y*) (*x* : *A*)
: ∀ (*y*:*A*) (*p1* *p2*:*x* = *y*), *p1* = *p2*.

Theorem K_dec *A* (*eq_dec* : ∀ *x* *y* : *A*, *x* = *y* ∨ *x* ≠ *y*) (*x* : *A*)
: ∀ *P*:*x* = *x* → Prop, *P* (eq_refl *x*) → ∀ *p*:*x* = *x*, *P* *p*.

Theorem inj_right_pair *A* (*eq_dec* : ∀ *x* *y* : *A*, *x* = *y* ∨ *x* ≠ *y*) (*x* : *A*)
: ∀ (*P*:*A* → Prop) (*y* *y'*:*P* *x*),
ex_intro *P* *x* *y* = ex_intro *P* *x* *y'* → *y* = *y'*.

Require Import EqdepFacts.

We deduce axiom *K* for (decidable) types Theorem K_dec_type (*A*:Type) (*eq_dec*:∀ *x* *y*:*A*, {*x* = *y*} + {*x* ≠ *y*}) (*x*:*A*)
(*P*:*x* = *x* → Prop) (*H*:*P* (eq_refl *x*)) (*p*:*x* = *x*) : *P* *p*.

Theorem K_dec_set :
∀ *A*:Set,
(∀ *x* *y*:*A*, {*x* = *y*} + {*x* ≠ *y*}) →
∀ (*x*:*A*) (*P*:*x* = *x* → Prop), *P* (eq_refl *x*) → ∀ *p*:*x* = *x*, *P* *p*.

We deduce the *eq_rect_eq* axiom for (decidable) types Theorem eq_rect_eq_dec :
∀ *A*:Type,
(∀ *x* *y*:*A*, {*x* = *y*} + {*x* ≠ *y*}) →

∀ $(p{:}A)$ $(Q{:}A \to$ Type) $(x{:}Q\ p)$ $(h{:}p = p)$, x = eq_rect $p\ Q\ x\ p\ h$.

We deduce the injectivity of dependent equality for decidable types Theorem eq_dep_eq_dec :

```
∀ A:Type,
   (∀ x y:A, {x = y} + {x ≠ y}) →
    ∀ (P:A→Type) (p:A) (x y:P p), eq_dep A P p x p y → x = y.
```

```
Theorem UIP_dec :
  ∀ (A:Type),
    (∀ x y:A, {x = y} + {x ≠ y}) →
    ∀ (x y:A) (p1 p2:x = y), p1 = p2.
```

```
Unset Implicit Arguments.
```

67.1.1 Definition of the functor that builds properties of dependent equalities on decidable sets in Type

The signature of decidable sets in Type

```
Module Type DecidableType.
  Axiom eq_dec : ∀ x y:U, {x = y} + {x ≠ y}.
End DecidableType.
```

The module *DecidableEqDep* collects equality properties for decidable set in Type

```
Module DecidableEqDep (M:DecidableType).
  Import M.
```

Invariance by Substitution of Reflexive Equality Proofs

```
  Lemma eq_rect_eq :
    ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.
```

Injectivity of Dependent Equality

```
  Theorem eq_dep_eq :
    ∀ (P:U→Type) (p:U) (x y:P p), eq_dep U P p x p y → x = y.
```

Uniqueness of Identity Proofs (UIP)

```
  Lemma UIP : ∀ (x y:U) (p1 p2:x = y), p1 = p2.
```

Uniqueness of Reflexive Identity Proofs

```
  Lemma UIP_refl : ∀ (x:U) (p:x = x), p = eq_refl x.
```

Streicher's axiom K

```
  Lemma Streicher_K :
    ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
```

Injectivity of equality on dependent pairs in Type

```
  Lemma inj_pairT2 :
    ∀ (P:U → Type) (p:U) (x y:P p),
      existT P p x = existT P p y → x = y.
```

Proof-irrelevance on subsets of decidable sets

Lemma inj_pairP2 :
  ∀ $(P{:}U \to$ Prop) $(x{:}U)$ $(p\ q{:}P\ x)$,
    ex_intro $P\ x\ p$ = ex_intro $P\ x\ q \to p = q$.

End DecidableEqDep.

67.1.2 Definition of the functor that builds properties of dependent equalities on decidable sets in Set

The signature of decidable sets in Set

Module Type DecidableSet.

  Parameter U:Set.
  Axiom *eq_dec* : ∀ $x\ y{:}U$, {$x = y$} + {$x \neq y$}.

End DecidableSet.

The module *DecidableEqDepSet* collects equality properties for decidable set in Set

Module DecidableEqDepSet (M:DecidableSet).

  Import M.
  Module N:=DecidableEqDep(M).

Invariance by Substitution of Reflexive Equality Proofs

  Lemma eq_rect_eq :
    ∀ $(p{:}U)$ $(Q{:}U \to$ Type) $(x{:}Q\ p)$ $(h{:}p = p)$, x = eq_rect $p\ Q\ x\ p\ h$.

Injectivity of Dependent Equality

  Theorem eq_dep_eq :
    ∀ $(P{:}U{\to}$Type) $(p{:}U)$ $(x\ y{:}P\ p)$, **eq_dep** $U\ P\ p\ x\ p\ y \to x = y$.

Uniqueness of Identity Proofs (UIP)

  Lemma UIP : ∀ $(x\ y{:}U)$ $(p1\ p2{:}x = y)$, $p1 = p2$.

Uniqueness of Reflexive Identity Proofs

  Lemma UIP_refl : ∀ $(x{:}U)$ $(p{:}x = x)$, p = eq_refl x.

Streicher's axiom K

  Lemma Streicher_K :
    ∀ $(x{:}U)$ $(P{:}x = x \to$ Prop), P (eq_refl x) → ∀ $p{:}x = x$, $P\ p$.

Proof-irrelevance on subsets of decidable sets

  Lemma inj_pairP2 :
    ∀ $(P{:}U \to$ Prop) $(x{:}U)$ $(p\ q{:}P\ x)$,
      ex_intro $P\ x\ p$ = ex_intro $P\ x\ q \to p = q$.

Injectivity of equality on dependent pairs in Type

  Lemma inj_pair2 :
    ∀ $(P{:}U \to$ Type) $(p{:}U)$ $(x\ y{:}P\ p)$,
      existT $P\ p\ x$ = existT $P\ p\ y \to x = y$.

Injectivity of equality on dependent pairs with second component in Type

Notation inj_pairT2 := inj_pair2.

End DecidableEqDepSet.

From decidability to inj_pair2 Lemma inj_pair2_eq_dec : ∀ A:Type, (∀ x y:A, {x=y}+{$x \neq y$}) → (∀ (P:A → Type) (p:A) (x y:P p), existT P p x = existT P p y → x = y).

Examples of short direct proofs of unicity of reflexivity proofs on specific domains

Lemma UIP_refl_unit (x : tt = tt) : x = eq_refl tt.

Lemma UIP_refl_bool (b:**bool**) (x : b = b) : x = eq_refl.

Lemma UIP_refl_nat (n:**nat**) (x : n = n) : x = eq_refl.

Chapter 68

Library Coq.Logic.Classical_Pred_Type

Classical Predicate Logic on Type

Require Import Classical_Prop.

Section Generic.
Variable U : Type.

de Morgan laws for quantifiers

Lemma not_all_not_ex :
$\forall$ P:U $\rightarrow$ Prop, $\neg$ ($\forall$ n:U, $\neg$ P n) $\rightarrow$ $\exists$ n : U, P n.

Lemma not_all_ex_not :
$\forall$ P:U $\rightarrow$ Prop, $\neg$ ($\forall$ n:U, P n) $\rightarrow$ $\exists$ n : U, $\neg$ P n.

Lemma not_ex_all_not :
$\forall$ P:U $\rightarrow$ Prop, $\neg$ ($\exists$ n : U, P n) $\rightarrow$ $\forall$ n:U, $\neg$ P n.

Lemma not_ex_not_all :
$\forall$ P:U $\rightarrow$ Prop, $\neg$ ($\exists$ n : U, $\neg$ P n) $\rightarrow$ $\forall$ n:U, P n.

Lemma ex_not_not_all :
$\forall$ P:U $\rightarrow$ Prop, ($\exists$ n : U, $\neg$ P n) $\rightarrow$ $\neg$ ($\forall$ n:U, P n).

Lemma all_not_not_ex :
$\forall$ P:U $\rightarrow$ Prop, ($\forall$ n:U, $\neg$ P n) $\rightarrow$ $\neg$ ($\exists$ n : U, P n).

End Generic.

Chapter 69

Library Coq.Logic.ProofIrrelevanceFacts

This defines the functor that build consequences of proof-irrelevance

```
Require Export EqdepFacts.

Module Type ProofIrrelevance.

  Axiom proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.

End ProofIrrelevance.

Module ProofIrrelevanceTheory (M:ProofIrrelevance).
```

Proof-irrelevance implies uniqueness of reflexivity proofs

```
  Module Eq_rect_eq.
    Lemma eq_rect_eq :
      ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p),
        x = eq_rect p Q x p h.
  End Eq_rect_eq.
```

Export the theory of injective dependent elimination

```
  Module EqdepTheory := EqdepTheory(Eq_rect_eq).
  Export EqdepTheory.

  Scheme eq_indd := Induction for eq Sort Prop.
```

We derive the irrelevance of the membership property for subsets

```
  Lemma subset_eq_compat :
    ∀ (U:Type) (P:U→Prop) (x y:U) (p:P x) (q:P y),
      x = y → exist P x p = exist P y q.

  Lemma subsetT_eq_compat :
    ∀ (U:Type) (P:U→Prop) (x y:U) (p:P x) (q:P y),
      x = y → existT P x p = existT P y q.

End ProofIrrelevanceTheory.
```

Chapter 70

Library Coq.Logic.ExtensionalFunctionRepresentativ

This module states a limited form axiom of functional extensionality which selects a canonical representative in each class of extensional functions

Its main interest is that it is the needed ingredient to provide axiom of choice on setoids (a.k.a. axiom of extensional choice) when combined with classical logic and axiom of (intensonal) choice

It provides extensionality of functions while still supporting (a priori) an intensional interpretation of equality

```
Axiom extensional_function_representative :
  ∀ A B, ∃ repr, ∀ (f : A → B),
  (∀ x, f x = repr f x) ∧
  (∀ g, (∀ x, f x = g x) → repr f = repr g).
```

Chapter 71

Library Coq.Logic.EqdepFacts

This file defines dependent equality and shows its equivalence with equality on dependent pairs (inhabiting sigma-types). It derives the consequence of axiomatizing the invariance by substitution of reflexive equality proofs and shows the equivalence between the 4 following statements

- Invariance by Substitution of Reflexive Equality Proofs.
- Injectivity of Dependent Equality
- Uniqueness of Identity Proofs
- Uniqueness of Reflexive Identity Proofs
- Streicher's Axiom K

These statements are independent of the calculus of constructions 2.

References:

1 T. Streicher, Semantical Investigations into Intensional Type Theory, Habilitationsschrift, LMU München, 1993. 2 M. Hofmann, T. Streicher, The groupoid interpretation of type theory, Proceedings of the meeting Twenty-five years of constructive type theory, Venice, Oxford University Press, 1998

Table of contents:

1. Definition of dependent equality and equivalence with equality of dependent pairs and with dependent pair of equalities

2. Eq_rect_eq <-> Eq_dep_eq <-> UIP <-> UIP_refl <-> K

3. Definition of the functor that builds properties of dependent equalities assuming axiom eq_rect_eq

71.1 Definition of dependent equality and equivalence with equality of dependent pairs

Import *EqNotations*.

Section Dependent_Equality.

```
Variable U : Type.
Variable P : U → Type.
```

Dependent equality

```
Inductive eq_dep (p:U) (x:P p) : ∀ q:U, P q → Prop :=
  eq_dep_intro : eq_dep p x p x.
#[local]
Hint Constructors eq_dep: core.

Lemma eq_dep_refl : ∀ (p:U) (x:P p), eq_dep p x p x.

Lemma eq_dep_sym :
  ∀ (p q:U) (x:P p) (y:P q), eq_dep p x q y → eq_dep q y p x.
#[local]
Hint Immediate eq_dep_sym: core.

Lemma eq_dep_trans :
  ∀ (p q r:U) (x:P p) (y:P q) (z:P r),
    eq_dep p x q y → eq_dep q y r z → eq_dep p x r z.

Scheme eq_indd := Induction for eq Sort Prop.
```

Equivalent definition of dependent equality as a dependent pair of equalities

```
Inductive eq_dep1 (p:U) (x:P p) (q:U) (y:P q) : Prop :=
  eq_dep1_intro : ∀ h:q = p, x = rew h in y → eq_dep1 p x q y.

Lemma eq_dep1_dep :
  ∀ (p:U) (x:P p) (q:U) (y:P q), eq_dep1 p x q y → eq_dep p x q y.

Lemma eq_dep_dep1 :
  ∀ (p q:U) (x:P p) (y:P q), eq_dep p x q y → eq_dep1 p x q y.

End Dependent_Equality.
```

Dependent equality is equivalent to equality on dependent pairs

```
Lemma eq_sigT_eq_dep :
  ∀ (U:Type) (P:U → Type) (p q:U) (x:P p) (y:P q),
    existT P p x = existT P q y → eq_dep p x q y.

Lemma eq_dep_eq_sigT :
  ∀ (U:Type) (P:U → Type) (p q:U) (x:P p) (y:P q),
    eq_dep p x q y → existT P p x = existT P q y.

Lemma eq_sigT_iff_eq_dep :
  ∀ (U:Type) (P:U → Type) (p q:U) (x:P p) (y:P q),
    existT P p x = existT P q y ↔ eq_dep p x q y.

Notation equiv_eqex_eqdep := eq_sigT_iff_eq_dep (only parsing).

Lemma eq_sig_eq_dep :
  ∀ (U:Type) (P:U → Prop) (p q:U) (x:P p) (y:P q),
    exist P p x = exist P q y → eq_dep p x q y.

Lemma eq_dep_eq_sig :
  ∀ (U:Type) (P:U → Prop) (p q:U) (x:P p) (y:P q),
```

```
    eq_dep p x q y → exist P p x = exist P q y.
Lemma eq_sig_iff_eq_dep :
  ∀ (U:Type) (P:U → Prop) (p q:U) (x:P p) (y:P q),
    exist P p x = exist P q y ↔ eq_dep p x q y.
```

Dependent equality is equivalent to a dependent pair of equalities

```
Set Implicit Arguments.
Lemma eq_sigT_sig_eq X P (x1 x2:X) H1 H2 :
  existT P x1 H1 = existT P x2 H2 ↔ {H:x1=x2 | rew H in H1 = H2}.
Lemma eq_sigT_fst X P (x1 x2:X) H1 H2 (H:existT P x1 H1 = existT P x2 H2) :
  x1 = x2.
Lemma eq_sigT_snd X P (x1 x2:X) H1 H2 (H:existT P x1 H1 = existT P x2 H2) :
  rew (eq_sigT_fst H) in H1 = H2.
Lemma eq_sig_fst X P (x1 x2:X) H1 H2 (H:exist P x1 H1 = exist P x2 H2) :
  x1 = x2.
Lemma eq_sig_snd X P (x1 x2:X) H1 H2 (H:exist P x1 H1 = exist P x2 H2) :
  rew (eq_sig_fst H) in H1 = H2.
Unset Implicit Arguments.
```

Exported hints

```
#[global]
Hint Resolve eq_dep_intro: core.
#[global]
Hint Immediate eq_dep_sym: core.
```

71.2 Eq_rect_eq <-> Eq_dep_eq <-> UIP <-> UIP_refl <-> K

```
Section Equivalences.
  Variable U:Type.
```

Invariance by Substitution of Reflexive Equality Proofs

```
  Definition Eq_rect_eq_on (p : U) (Q : U → Type) (x : Q p) :=
    ∀ (h : p = p), x = eq_rect p Q x p h.
  Definition Eq_rect_eq := ∀ p Q x, Eq_rect_eq_on p Q x.
```

Injectivity of Dependent Equality

```
  Definition Eq_dep_eq_on (P : U → Type) (p : U) (x : P p) :=
    ∀ (y : P p), eq_dep p x p y → x = y.
  Definition Eq_dep_eq := ∀ P p x, Eq_dep_eq_on P p x.
```

Uniqueness of Identity Proofs (UIP)

```
  Definition UIP_on_ (x y : U) (p1 : x = y) :=
    ∀ (p2 : x = y), p1 = p2.
  Definition UIP_ := ∀ x y p1, UIP_on_ x y p1.
```

Uniqueness of Reflexive Identity Proofs

Definition UIP_refl_on_ $(x : U)$:=
 $\forall$ $(p : x = x)$, p = eq_refl x.
Definition UIP_refl_ := $\forall$ x, UIP_refl_on_ x.

Streicher's axiom K

Definition Streicher_K_on_ $(x : U)$ $(P : x = x \to$ Prop$)$:=
 P (eq_refl x) $\to \forall$ $p : x = x$, P p.
Definition Streicher_K_ := $\forall$ x P, Streicher_K_on_ x P.

Injectivity of Dependent Equality is a consequence of Invariance by Substitution of Reflexive Equality Proof

Lemma eq_rect_eq_on__eq_dep1_eq_on $(p : U)$ $(P : U \to$ Type$)$ $(y : P\ p)$:
 Eq_rect_eq_on p P y $\to \forall$ $(x : P\ p)$, **eq_dep1** p x p y $\to x = y$.
Lemma eq_rect_eq__eq_dep1_eq :
 Eq_rect_eq $\to \forall$ $(P{:}U{\to}$Type$)$ $(p{:}U)$ $(x\ y{:}P\ p)$, **eq_dep1** p x p y $\to x = y$.

Lemma eq_rect_eq_on__eq_dep_eq_on $(p : U)$ $(P : U \to$ Type$)$ $(x : P\ p)$:
 Eq_rect_eq_on p P x $\to$ Eq_dep_eq_on P p x.
Lemma eq_rect_eq__eq_dep_eq : Eq_rect_eq $\to$ Eq_dep_eq.

Uniqueness of Identity Proofs (UIP) is a consequence of Injectivity of Dependent Equality

Lemma eq_dep_eq_on__UIP_on $(x\ y : U)$ $(p1 : x = y)$:
 Eq_dep_eq_on (fun $y \Rightarrow x = y$) x eq_refl $\to$ UIP_on_ x y $p1$.
Lemma eq_dep_eq__UIP : Eq_dep_eq $\to$ UIP_.

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma UIP_on__UIP_refl_on $(x : U)$:
 UIP_on_ x x eq_refl $\to$ UIP_refl_on_ x.
Lemma UIP__UIP_refl : UIP_ $\to$ UIP_refl_.

Streicher's axiom K is a direct consequence of Uniqueness of Reflexive Identity Proofs

Lemma UIP_refl_on__Streicher_K_on $(x : U)$ $(P : x = x \to$ Prop$)$:
 UIP_refl_on_ x $\to$ Streicher_K_on_ x P.
Lemma UIP_refl__Streicher_K : UIP_refl_ $\to$ Streicher_K_.

We finally recover from K the Invariance by Substitution of Reflexive Equality Proofs

Lemma Streicher_K_on__eq_rect_eq_on $(p : U)$ $(P : U \to$ Type$)$ $(x : P\ p)$:
 Streicher_K_on_ p (fun $h \Rightarrow x$ = rew $\to$ [P] h in x)
 $\to$ Eq_rect_eq_on p P x.
Lemma Streicher_K__eq_rect_eq : Streicher_K_ $\to$ Eq_rect_eq.

Remark: It is reasonable to think that *eq_rect_eq* is strictly stronger than *eq_rec_eq* (which is *eq_rect_eq* restricted on Set):

Definition *Eq_rec_eq* := $\forall$ $(P{:}U \to$ Set$)$ $(p{:}U)$ $(x{:}P\ p)$ $(h{:}p = p)$, x = *eq_rec* p P x p h.

Typically, *eq_rect_eq* allows proving UIP and Streicher's K what does not seem possible with *eq_rec_eq*. In particular, the proof of *UIP* requires to use *eq_rect_eq* on fun $y \to x{=}y$ which is in Type but not in Set.

End Equivalences.

UIP_refl is downward closed (a short proof of the key lemma of Voevodsky's proof of inclusion of h-level n into h-level n+1; see hlevelntosn in https://github.com/vladimirias/Foundations.git).

```
Theorem UIP_shift_on (X : Type) (x : X) :
  UIP_refl_on_ X x → ∀ y : x = x, UIP_refl_on_ (x = x) y.
Theorem UIP_shift : ∀ U, UIP_refl_ U → ∀ x:U, UIP_refl_ (x = x).
```

```
Section Corollaries.
```

```
  Variable U:Type.
```

UIP implies the injectivity of equality on dependent pairs in Type

```
 Definition Inj_dep_pair_on (P : U → Type) (p : U) (x : P p) :=
   ∀ (y : P p), existT P p x = existT P p y → x = y.
 Definition Inj_dep_pair := ∀ P p x, Inj_dep_pair_on P p x.
```

```
 Lemma eq_dep_eq_on__inj_pair2_on (P : U → Type) (p : U) (x : P p) :
   Eq_dep_eq_on U P p x → Inj_dep_pair_on P p x.
 Lemma eq_dep_eq__inj_pair2 : Eq_dep_eq U → Inj_dep_pair.
```

```
End Corollaries.
```

```
Notation Inj_dep_pairS := Inj_dep_pair.
Notation Inj_dep_pairT := Inj_dep_pair.
Notation eq_dep_eq__inj_pairT2 := eq_dep_eq__inj_pair2.
```

71.3 Definition of the functor that builds properties of dependent equalities assuming axiom eq_rect_eq

```
Module Type EqdepElimination.
```

```
  Axiom eq_rect_eq :
    ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p),
      x = eq_rect p Q x p h.
```

```
End EqdepElimination.
```

```
Module EqdepTheory (M:EqdepElimination).
```

```
  Section Axioms.
```

```
    Variable U:Type.
```

Invariance by Substitution of Reflexive Equality Proofs

```
Lemma eq_rect_eq :
  ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.
```

```
Lemma eq_rec_eq :
  ∀ (p:U) (Q:U → Set) (x:Q p) (h:p = p), x = eq_rec p Q x p h.
```

Injectivity of Dependent Equality

```
Lemma eq_dep_eq : ∀ (P:U→Type) (p:U) (x y:P p), eq_dep p x p y → x = y.
```

Uniqueness of Identity Proofs (UIP) is a consequence of Injectivity of Dependent Equality

```
Lemma UIP : ∀ (x y:U) (p1 p2:x = y), p1 = p2.
```

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma UIP_refl : $\forall$ (x:U) (p:x = x), p = eq_refl x.

Streicher's axiom K is a direct consequence of Uniqueness of Reflexive Identity Proofs

Lemma Streicher_K :
$\forall$ (x:U) (P:x = x $\rightarrow$ Prop), P (eq_refl x) $\rightarrow$ $\forall$ p:x = x, P p.

End Axioms.

UIP implies the injectivity of equality on dependent pairs in Type

Lemma inj_pair2 :
$\forall$ (U:Type) (P:U $\rightarrow$ Type) (p:U) (x y:P p),
existT P p x = existT P p y $\rightarrow$ x = y.

Notation inj_pairT2 := inj_pair2.

End EqdepTheory.

Basic facts about eq_dep

Lemma f_eq_dep :
$\forall$ U (P:$U$$\rightarrow$Type) R p q x y (f:$\forall$ p, P p $\rightarrow$ R p),
eq_dep p x q y $\rightarrow$ **eq_dep** p (f p x) q (f q y).

Lemma eq_dep_non_dep :
$\forall$ U P p q x y, @**eq_dep** U (fun _ $\Rightarrow$ P) p x q y $\rightarrow$ x = y.

Lemma f_eq_dep_non_dep :
$\forall$ U (P:$U$$\rightarrow$Type) R p q x y (f:$\forall$ p, P p $\rightarrow$ R),
eq_dep p x q y $\rightarrow$ f p x = f q y.

Chapter 72

Library Coq.Logic.Description

This file provides a constructive form of definite description; it allows building functions from the proof of their existence in any context; this is weaker than Church's iota operator

```
Require Import ChoiceFacts.

Set Implicit Arguments.

Axiom constructive_definite_description :
  ∀ (A : Type) (P : A→Prop),
    (∃! x, P x) → { x : A | P x }.
```

Chapter 73

Library Coq.Logic.StrictProp

Utilities for SProp users.

Record **Box** (A:$SProp$) : Prop := box { unbox : A }.

Inductive **Squash** (A:Type) : $SProp$:= squash : $A \rightarrow$ **Squash** A.

Inductive **sEmpty** : $SProp$:=.

Inductive **sUnit** : $SProp$:= stt.

Record **Ssig** {A:Type} (P:$A \rightarrow SProp$) := Sexists { Spr1 : A; Spr2 : P Spr1 }.

Lemma Spr1_inj {A P} {a b : @**Ssig** A P} (e : Spr1 a = Spr1 b) : a = b.

Chapter 74

Library Coq.Logic.Hurkens

Exploiting Hurkens's paradox [*Hurkens95*] for system U- so as to derive various contradictory contexts.

The file is divided into various sub-modules which all follow the same structure: a section introduces the contradictory hypotheses and a theorem named *paradox* concludes the module with a proof of *False*.

- The *Generic* module contains the actual Hurkens's paradox for a postulated shallow encoding of system U- in Coq. This is an adaptation by Arnaud Spiwack of a previous, more restricted implementation by Herman Geuvers. It is used to derive every other special cases of the paradox in this file.

- The *NoRetractToImpredicativeUniverse* module contains a simple and effective formulation by Herman Geuvers [*Geuvers01*] of a result by Thierry Coquand [*Coquand90*]. It states that no impredicative sort can contain a type of which it is a retract. This result implies that Coq with classical logic stated in impredicative Set is inconsistent and that classical logic stated in Prop implies proof-irrelevance (see *ClassicalFacts.v*)

- The *NoRetractFromSmallPropositionToProp* module is a specialisation of the *NoRetractToImpredicativeUniverse* module to the case where the impredicative sort is `Prop`.

- The *NoRetractToModalProposition* module is a strengthening of the *NoRetractFromSmallPropositionToProp* module. It shows that given a monadic modality (aka closure operator) M, the type of modal propositions (i.e. such that $M\ A \to A$) cannot be a retract of a modal proposition. It is an example of use of the paradox where the universes of system U- are not mapped to universes of Coq.

- The *NoRetractToNegativeProp* module is the specialisation of the *NoRetractFromSmallPropositionToProp* module where the modality is double-negation. This result implies that the principle of weak excluded middle (∀ *A*, ~~*A*\/~*A*) implies a weak variant of proof irrelevance.

- The *NoRetractFromTypeToProp* module proves that `Prop` cannot be a retract of a larger type.

- The *TypeNeqSmallType* module proves that `Type` is different from any smaller type.

- The *PropNeqType* module proves that `Prop` is different from any larger `Type`. It is an instance of the previous result.

References:

Coquand90 T. Coquand, "Metamathematical Investigations of a Calculus of Constructions", Proceedings of Logic in Computer Science (LICS'90), 1990.

Hurkens95 A. J. Hurkens, "A simplification of Girard's paradox", Proceedings of the 2nd international conference Typed Lambda-Calculi and Applications (TLCA'95), 1995.

Geuvers01 H. Geuvers, "Inconsistency of Classical Logic in Type Theory", 2001, revised 2007 (see external link [1]).

74.1 A modular proof of Hurkens's paradox.

It relies on an axiomatisation of a shallow embedding of system U- (i.e. types of U- are interpreted by types of Coq). The universes are encoded in a style, due to Martin-Löf, where they are given by a set of names and a family *El*:*Name*→`Type` which interprets each name into a type. This allows the encoding of universe to be decoupled from Coq's universes. Dependent products and abstractions are similarly postulated rather than encoded as Coq's dependent products and abstractions.

```
Module GENERIC.

Section Paradox.
```

74.1.1 Axiomatisation of impredicative universes in a Martin-Löf style

System U- has two impredicative universes. In the proof of the paradox they are slightly asymmetric (in particular the reduction rules of the small universe are not needed). Therefore, the axioms are duplicated allowing for a weaker requirement than the actual system U-.

Large universe

```
Variable U1 : Type.
Variable El1 : U1 → Type.
```

Closure by small product `Variable Forall1 : ∀ u:U1, (El1 u → U1) → U1.`

```
  Notation "'∀1' x : A , B" := (Forall1 A (fun x ⇒ B)).
  Notation "A '⟶1' B" := (Forall1 A (fun _ ⇒ B)).
Variable lam1 : ∀ u B, (∀ x:El1 u, El1 (B x)) → El1 (∀1 x:u, B x).
  Notation "'λ1' x , u" := (lam1 _ _ (fun x ⇒ u)).
Variable app1 : ∀ u B (f:El1 (Forall1 u B)) (x:El1 u), El1 (B x).
  Notation "f '·1' x" := (app1 _ _ f x).
Variable beta1 : ∀ u B (f:∀ x:El1 u, El1 (B x)) x,
                    (λ1 y, f y) ·1 x = f x.
```

[1] http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz

Closure by large products *U1* only needs to quantify over itself. `Variable` *ForallU1* : (*U1*→*U1*) → *U1*.

`Notation` "'$\forall_2$' A , F" := (*ForallU1* (`fun` *A* ⇒ *F*)).
`Variable` *lamU1* : ∀ *F*, (∀ *A*:*U1*, *El1* (*F A*)) → *El1* ($\forall_2$ *A*, *F A*).
`Notation` "'λ_2' x , u" := (*lamU1* _ (`fun` *x* ⇒ *u*)).
`Variable` *appU1* : ∀ *F* (*f*:*El1*($\forall_2$ *A*,*F A*)) (*A*:*U1*), *El1* (*F A*).
`Notation` "f '$\cdot_1$' [A]" := (*appU1* _ *f A*).
`Variable` *betaU1* : ∀ *F* (*f*:∀ *A*:*U1*, *El1* (*F A*)) *A*,
(λ_2 *x*, *f x*) $\cdot_1$ [*A*] = *f A*.

Small universe

The small universe is an element of the large one. `Variable` *u0* : *U1*.
`Notation` U0 := (*El1 u0*).
`Variable` *El0* : U0 → `Type`.

Closure by small product *U0* does not need reduction rules `Variable` *Forall0* : ∀ *u*:U0, (*El0 u* → U0) → U0.

`Notation` "'$\forall_0$' x : A , B" := (*Forall0 A* (`fun` *x* ⇒ *B*)).
`Notation` "A '$\longrightarrow_0$' B" := (*Forall0 A* (`fun` _ ⇒ *B*)).
`Variable` *lam0* : ∀ *u B*, (∀ *x*:*El0 u*, *El0* (*B x*)) → *El0* ($\forall_0$ *x*:*u*, *B x*).
`Notation` "'λ_0' x , u" := (*lam0* _ _ (`fun` *x* ⇒ *u*)).
`Variable` *app0* : ∀ *u B* (*f*:*El0* (*Forall0 u B*)) (*x*:*El0 u*), *El0* (*B x*).
`Notation` "f '$\cdot_0$' x" := (*app0* _ _ *f x*).

Closure by large products `Variable` *ForallU0* : ∀ *u*:*U1*, (*El1 u*→U0) → U0.

`Notation` "'$\forall_0{}^1$' A : U , F" := (*ForallU0 U* (`fun` *A* ⇒ *F*)).
`Variable` *lamU0* : ∀ *U F*, (∀ *A*:*El1 U*, *El0* (*F A*)) → *El0* ($\forall_0{}^1$ *A*:*U*, *F A*).
`Notation` "'$\lambda_0{}^1$' x , u" := (*lamU0* _ _ (`fun` *x* ⇒ *u*)).
`Variable` *appU0* : ∀ *U F* (*f*:*El0*($\forall_0{}^1$ *A*:*U*,*F A*)) (*A*:*El1 U*), *El0* (*F A*).
`Notation` "f '$\cdot_0$' [A]" := (*appU0* _ _ *f A*).

74.1.2 Automating the rewrite rules of our encoding.

```
Local Ltac simplify :=

  (repeat rewrite ?beta1, ?betaU1);
  lazy beta.
```

```
Local Ltac simplify_in h :=
  (repeat rewrite ?beta1, ?betaU1 in h);
  lazy beta in h.
```

74.1.3 Hurkens's paradox.

An inhabitant of *U0* standing for *False*. `Variable` *F*:U0.

Preliminary definitions

Definition V : *U1* := $\forall_2$ *A*, (($A \longrightarrow_1 u0$) $\longrightarrow_1 A \longrightarrow_1 u0$) $\longrightarrow_1 A \longrightarrow_1 u0$.
Definition U : *U1* := V $\longrightarrow_1 u0$.

Definition sb (z:*El1* V) : *El1* V := λ_2 *A*, λ_1 *r*, λ_1 *a*, $r \cdot_1 (z \cdot_1 [A] \cdot_1 r) \cdot_1 a$.

Definition le (i:*El1* (U$\longrightarrow_1$u0)) (x:*El1* U) : U0 :=
  $x \cdot_1 (\lambda_2 A, \lambda_1 r, \lambda_1 a, i \cdot_1 (\lambda_1 v, (\text{sb } v) \cdot_1 [A] \cdot_1 r \cdot_1 a))$.
Definition le' : *El1* ((U$\longrightarrow_1$u0) $\longrightarrow_1$ U $\longrightarrow_1 u0$) := λ_1 *i*, λ_1 *x*, le *i x*.
Definition induct (i:*El1* (U$\longrightarrow_1$u0)) : U0 :=
  $\forall_0{}^1$ x:U, le i $x \longrightarrow_0 i \cdot_1 x$.

Definition WF : *El1* U := λ_1 *z*, (induct ($z \cdot_1$[U] $\cdot_1$ le')).
Definition I (x:*El1* U) : U0 :=
  ($\forall_0{}^1$ i:U$\longrightarrow_1$u0, le i $x \longrightarrow_0 i \cdot_1 (\lambda_1 v, (\text{sb } v) \cdot_1$ [U] $\cdot_1$ le' $\cdot_1 x$)) $\longrightarrow_0 F$
.

Proof

Lemma Omega : *El0* ($\forall_0{}^1$ i:U$\longrightarrow_1$u0, induct $i \longrightarrow_0 i \cdot_1$ WF).
Proof.
  refine ($\lambda_0{}^1$ *i*, λ_0 *y*, _).
  refine ($y \cdot_0$[_]$\cdot_0$_).
  unfold le,WF,induct. *simplify.*
  refine ($\lambda_0{}^1$ *x*, λ_0 *h0*, _). *simplify.*
  refine ($y \cdot_0$[_]$\cdot_0$_).
  unfold le. *simplify.*
  unfold sb at 1. *simplify.*
  unfold le' at 1. *simplify.*
  exact *h0.*
Qed.

Lemma lemma1 : *El0* (induct (λ_1 *u*, I *u*)).
Proof.
  unfold induct.
  refine ($\lambda_0{}^1$ *x*, λ_0 *p*, _). *simplify.*
  refine (λ_0 *q*,_).
  assert (*El0* (I (λ_1 *v*, (sb *v*)$\cdot_1$[U]$\cdot_1$le'$\cdot_1$x))) as *h*.
  { generalize ($q \cdot_0$[λ_1 *u*, I *u*]$\cdot_0$p). *simplify.*
    intros *q'.*
    exact *q'.* }
  refine ($h \cdot_0$_).
  refine ($\lambda_0{}^1$ *i*,_).
  refine (λ_0 *h'*, _).
  generalize ($q \cdot_0$[λ_1 *y*, $i \cdot_1 (\lambda_1 v, (\text{sb } v) \cdot_1$[U] $\cdot_1$ le' $\cdot_1 y$)]). *simplify.*
  intros *q'.*
  refine ($q' \cdot_0$_). clear *q'.*
  unfold le at 1 in *h'*. *simplify_in h'.*

```
  unfold sb at 1 in h'. simplify_in h'.
  unfold le' at 1 in h'. simplify_in h'.
  exact h'.
Qed.
```

```
Lemma lemma2 : El0 ((∀₀¹i:U⟶₁u0, induct i ⟶₀ i ·₁WF) ⟶₀ F).
Proof.
  refine (λ₀ x, _).
  assert (El0 (I WF)) as h.
  { generalize (x ·₀[λ₁ u, I u] ·₀lemma1). simplify.
    intros q.
    exact q. }
  refine (h ·₀_). clear h.
  refine (λ₀¹ i, λ₀ h0, _).
  generalize (x ·₀[λ₁ y, i ·₁(λ₁ v, (sb v) ·₁[U] ·₁le' ·₁y)]). simplify.
  intros q.
  refine (q ·₀_). clear q.
  unfold le in h0. simplify_in h0.
  unfold WF in h0. simplify_in h0.
  exact h0.
Qed.
```

```
Theorem paradox : El0 F.
Proof.
  exact (lemma2 ·₀Omega).
Qed.
```

End Paradox.

The *paradox* tactic can be called as a shortcut to use the paradox. Ltac *paradox h* := *unshelve* (refine ((fun *h* ⇒ _) (paradox _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _))).

End Generic.

74.2 Impredicative universes are not retracts.

There can be no retract to an impredicative Coq universe from a smaller type. In this version of the proof, the impredicativity of the universe is postulated with a pair of functions from the universe to its type and back which commute with dependent product in an appropriate way.

Module NoRetractToImpredicativeUniverse.

Section Paradox.

```
Let U2 := Type.
Let U1:U2 := Type.
Variable U0:U1.
```

U1 is impredicative

```
Variable u22u1 : U2 → U1.
```

Hypothesis $u22u1\_unit$: ∀ (c:$U2$), $c \to u22u1\ c$.

$u22u1\_counit$ and $u22u1\_coherent$ only apply to dependent product so that the equations happen in the smaller $U1$ rather than $U2$. Indeed, it is not generally the case that one can project from a large universe to an impredicative universe and then get back the original type again. It would be too strong a hypothesis to require (in particular, it is not true of Prop). The formulation is reminiscent of the monadic characteristic of the projection from a large type to Prop. Hypothesis $u22u1\_counit$: ∀ (F:$U1 \to U1$), $u22u1$ (∀ A,F A) → (∀ A,F A).

Hypothesis $u22u1\_coherent$: ∀ (F:$U1 \to U1$) (f:∀ x:$U1$, $F\ x$) (x:$U1$),
$u22u1\_counit$ _ ($u22u1\_unit$ _ f) x = $f\ x$.

U0 is a retract of *U1*

Variable $u02u1$: $U0 \to U1$.
Variable $u12u0$: $U1 \to U0$.
Hypothesis $u12u0\_unit$: ∀ (b:$U1$), $b \to u02u1\ (u12u0\ b)$.
Hypothesis $u12u0\_counit$: ∀ (b:$U1$), $u02u1\ (u12u0\ b) \to b$.

74.2.1 Paradox

```
Theorem paradox : ∀ F:U1, F.
Proof.
  intros F.
  Generic.paradox h.
   Large universe    + exact U1.
  + exact (fun X ⇒ X).
  + cbn. exact (fun u F ⇒ ∀ x:u, F x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun F ⇒ u22u1 (∀ x, F x)).
  + cbn. exact (fun _ x ⇒ u22u1_unit _ x).
  + cbn. exact (fun _ x ⇒ u22u1_counit _ x).
   Small universe    + exact U0.
   The interpretation of the small universe is the image of U0 in U1.    + cbn. exact (fun X ⇒
u02u1 X).
  + cbn. exact (fun u F ⇒ u12u0 (∀ x:(u02u1 u), u02u1 (F x))).
  + cbn. exact (fun u F ⇒ u12u0 (∀ x:u, u02u1 (F x))).
  + cbn. exact (u12u0 F).
  + cbn in h.
    exact (u12u0_counit _ h).
  + cbn. easy.
  + cbn. intros **. now rewrite u22u1_coherent.
  + cbn. intros × x. exact (u12u0_unit _ x).
  + cbn. intros × x. exact (u12u0_counit _ x).
  + cbn. intros × x. exact (u12u0_unit _ x).
  + cbn. intros × x. exact (u12u0_counit _ x).
Qed.
```

End Paradox.

End NoRetractToImpredicativeUniverse.

74.3 Modal fragments of Prop are not retracts

In presence of a a monadic modality on Prop, we can define a subset of Prop of modal propositions which is also a complete Heyting algebra. These cannot be a retract of a modal proposition. This is a case where the universe in system U- are not encoded as Coq universes.

Module NoRetractToModalProposition.

74.3.1 Monadic modality

Section Paradox.

```
Variable M : Prop → Prop.
Hypothesis incr : ∀ A B:Prop, (A→B) → M A → M B.
```

```
Lemma strength: ∀ A (P:A→Prop), M(∀ x:A,P x) → ∀ x:A,M(P x).
Proof.
  intros A P h x.
  eapply incr in h; eauto.
Qed.
```

74.3.2 The universe of modal propositions

```
Definition MProp := { P:Prop | M P → P }.
Definition El : MProp → Prop := @proj1_sig _ _.
```

```
Lemma modal : ∀ P:MProp, M(El P) → El P.
Proof.
  intros [P m]. cbn.
  exact m.
Qed.
```

```
Definition Forall {A:Type} (P:A→MProp) : MProp.
Proof.
  unshelve (refine (exist _ _ _)).
  + exact (∀ x:A, El (P x)).
  + intros h x.
    eapply strength in h.
    eauto using modal.
Defined.
```

74.3.3 Retract of the modal fragment of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

```
Variable bool : MProp.
```

```
Variable p2b : MProp → El bool.
Variable b2p : El bool → MProp.
Hypothesis p2p1 : ∀ A:MProp, El (b2p (p2b A)) → El A.
Hypothesis p2p2 : ∀ A:MProp, El A → El (b2p (p2b A)).
```

74.3.4 Paradox

```
Theorem paradox : ∀ B:MProp, El B.
Proof.
  intros B.
  Generic.paradox h.
   Large universe   + exact MProp.
  + exact El.
  + exact (fun _ ⇒ Forall).
  + cbn. exact (fun _ _ f ⇒ f).
  + cbn. exact (fun _ _ f ⇒ f).
  + exact Forall.
  + cbn. exact (fun _ f ⇒ f).
  + cbn. exact (fun _ f ⇒ f).
   Small universe   + exact bool.
  + exact (fun b ⇒ El (b2p b)).
  + cbn. exact (fun _ F ⇒ p2b (Forall (fun x ⇒ b2p (F x)))).
  + exact (fun _ F ⇒ p2b (Forall (fun x ⇒ b2p (F x)))).
  + apply p2b.
    exact B.
  + cbn in h. auto.
  + cbn. easy.
  + cbn. easy.
  + cbn. auto.
  + cbn. intros × f.
    apply p2p1 in f. cbn in f.
    exact f.
  + cbn. auto.
  + cbn. intros × f.
    apply p2p1 in f. cbn in f.
    exact f.
Qed.

End Paradox.

End NoRetractToModalProposition.
```

74.4 The negative fragment of `Prop` is not a retract

The existence in the pure Calculus of Constructions of a retract from the negative fragment of `Prop` into a negative proposition is inconsistent. This is an instance of the previous result.

Module NoRetractToNegativeProp.

74.4.1 The universe of negative propositions.

```
Definition NProp := { P:Prop | ~~P → P }.
Definition El : NProp → Prop := @proj1_sig _ _.
```

Section Paradox.

74.4.2 Retract of the negative fragment of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

```
Variable bool : NProp.
Variable p2b : NProp → El bool.
Variable b2p : El bool → NProp.
Hypothesis p2p1 : ∀ A:NProp, El (b2p (p2b A)) → El A.
Hypothesis p2p2 : ∀ A:NProp, El A → El (b2p (p2b A)).
```

74.4.3 Paradox

```
Theorem paradox : ∀ B:NProp, El B.
Proof.
  intros B.
  unshelve (refine ((fun h ⇒ _) (NoRetractToModalProposition.paradox _ _ _ _ _ _ _ _))).
  + exact (fun P ⇒ ~~P).
  + exact bool.
  + exact p2b.
  + exact b2p.
  + exact B.
  + exact h.
  + cbn. auto.
  + cbn. auto.
  + cbn. auto.
Qed.
```

End Paradox.

End NoRetractToNegativeProp.

74.5 Prop is not a retract

The existence in the pure Calculus of Constructions of a retract from Prop into a small type of Prop is inconsistent. This is a special case of the previous result.

Module NoRetractFromSmallPropositionToProp.

74.5.1 The universe of propositions.

Definition NProp := { P:Prop | $P \to P$}.
Definition El : NProp → Prop := @proj1_sig _ _.

Section MParadox.

74.5.2 Retract of Prop in a small type, using the identity modality.

Variable *bool* : NProp.
Variable *p2b* : NProp → El *bool*.
Variable *b2p* : El *bool* → NProp.
Hypothesis *p2p1* : ∀ A:NProp, El (*b2p* (*p2b* A)) → El A.
Hypothesis *p2p2* : ∀ A:NProp, El A → El (*b2p* (*p2b* A)).

74.5.3 Paradox

Theorem mparadox : ∀ B:NProp, El B.
Proof.
  intros B.
  unshelve (refine ((fun h ⇒ _) (NoRetractToModalProposition.paradox _ _ _ _ _ _ _ _))).
  + exact (fun P ⇒ P).
  + exact *bool*.
  + exact *p2b*.
  + exact *b2p*.
  + exact B.
  + exact h.
  + *cbn*. auto.
  + *cbn*. auto.
  + *cbn*. auto.
Qed.

End MParadox.

Section Paradox.

74.5.4 Retract of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions. Variable *bool* : Prop.
Variable *p2b* : Prop → *bool*.
Variable *b2p* : *bool* → Prop.
Hypothesis *p2p1* : ∀ A:Prop, *b2p* (*p2b* A) → A.
Hypothesis *p2p2* : ∀ A:Prop, A → *b2p* (*p2b* A).

74.5.5 Paradox

Theorem paradox : ∀ B:Prop, B.

```
Proof.
  intros B.
  unshelve (refine (mparadox (exist _ bool (fun x ⇒ x)) _ _ _ _
                    (exist _ B (fun x ⇒ x))))).
  + intros p. red. red. exact (p2b (El p)).
  + cbn. intros b. red. ∃ (b2p b). exact (fun x ⇒ x).
  + cbn. intros [A H]. cbn. apply p2p1.
  + cbn. intros [A H]. cbn. apply p2p2.
Qed.
```

End Paradox.

End NoRetractFromSmallPropositionToProp.

74.6 Large universes are not retracts of Prop.

The existence in the Calculus of Constructions with universes of a retract from some Type universe into Prop is inconsistent.

Module NoRetractFromTypeToProp.

```
Definition Type2 := Type.
Definition Type1 := Type : Type2.
```

Section Paradox.

74.6.1 Assumption of a retract from Type into Prop

```
Variable down : Type1 → Prop.
Variable up : Prop → Type1.
Hypothesis up_down : ∀ (A:Type1), up (down A) = A :> Type1.
```

74.6.2 Paradox

```
Theorem paradox : ∀ P:Prop, P.
Proof.
  intros P.
  Generic.paradox h.
   Large universe.     + exact Type1.
  + exact (fun X ⇒ X).
  + cbn. exact (fun u F ⇒ ∀ x, F x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
  + exact (fun F ⇒ ∀ A:Prop, F(up A)).
  + cbn. exact (fun F f A ⇒ f (up A)).
  + cbn.
    intros F f A.
    specialize (f (down A)).
```

```
    rewrite up_down in f.
    exact f.
  + exact Prop.
  + cbn. exact (fun X ⇒ X).
  + cbn. exact (fun A P ⇒ ∀ x:A, P x).
  + cbn. exact (fun A P ⇒ ∀ x:A, P x).
  + cbn. exact P.
  + exact h.
  + cbn. easy.
  + cbn.
    intros F f A.
    destruct (up_down A). cbn.
    reflexivity.
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
Qed.
```

End Paradox.

End NoRetractFromTypeToProp.

74.7 $A \neq$ Type

No Coq universe can be equal to one of its elements.

Module TypeNeqSmallType.

Unset *Universe Polymorphism*.

Section Paradox.

74.7.1 Universe U is equal to one of its elements.

```
Let U := Type.
Variable A:U.
Hypothesis h : U=A.
```

74.7.2 Universe U is a retract of A

The following context is actually sufficient for the paradox to hold. The hypothesis $h{:}U{=}A$ is only used to define *down*, *up* and *up_down*.

```
Let down (X:U) : A := @eq_rect _ _ (fun X ⇒ X) X _ h.
Let up (X:A) : U := @eq_rect_r _ _ (fun X ⇒ X) X _ h.
```

```
Lemma up_down : ∀ (X:U), up (down X) = X.
Proof.
  unfold up,down.
```

```
  rewrite ← h.
  reflexivity.
Qed.

Theorem paradox : False.
Proof.
  Generic.paradox p.
    Large universe    + exact U.
  + exact (fun X⇒X).
  + cbn. exact (fun X F ⇒ ∀ x:X, F x).
  + cbn. exact (fun _ _ x ⇒ x).
  + cbn. exact (fun _ _ x ⇒ x).
  + exact (fun F ⇒ ∀ x:A, F (up x)).
  + cbn. exact (fun _ f ⇒ fun x:A ⇒ f (up x)).
  + cbn. intros × f X.
    specialize (f (down X)).
    rewrite up_down in f.
    exact f.
    Small universe    + exact A.
    The interpretation of A as a universe is U.    + cbn. exact up.
  + cbn. exact (fun _ F ⇒ down (∀ x, up (F x))).
  + cbn. exact (fun _ F ⇒ down (∀ x, up (F x))).
  + cbn. exact (down False).
  + rewrite up_down in p.
    exact p.
  + cbn. easy.
  + cbn. intros ? f X.
    destruct (up_down X). cbn.
    reflexivity.
  + cbn. intros ? ? f.
    rewrite up_down.
    exact f.
  + cbn. intros ? ? f.
    rewrite up_down in f.
    exact f.
  + cbn. intros ? ? f.
    rewrite up_down.
    exact f.
  + cbn. intros ? ? f.
    rewrite up_down in f.
    exact f.
Qed.

End Paradox.

End TypeNeqSmallType.
```

74.8 Prop≠Type.

Special case of *TypeNeqSmallType*.

Module PropNeqType.

```
Theorem paradox : Prop ≠ Type.
Proof.
  intros h.
  unshelve (refine (TypeNeqSmallType.paradox _ _)).
  + exact Prop.
  + easy.
Qed.
```

End PropNeqType.

Chapter 75

Library Coq.Logic.Eqdep

This file axiomatizes the invariance by substitution of reflexive equality proofs [*Streicher93*] and exports its consequences, such as the injectivity of the projection of the dependent pair.

[*Streicher93*] T. Streicher, Semantical Investigations into Intensional Type Theory, Habilitationsschrift, LMU München, 1993.

```
Require Export EqdepFacts.

Module Eq_rect_eq.

Axiom eq_rect_eq :
  ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

End Eq_rect_eq.

Module EqdepTheory := EqdepTheory(Eq_rect_eq).
Export EqdepTheory.
```

Exported hints

```
#[global]
Hint Resolve eq_dep_eq: eqdep.
#[global]
Hint Resolve inj_pair2 inj_pairT2: eqdep.
```

Chapter 76

Library Coq.Logic.PropExtensionalityFacts

Some facts and definitions about propositional and predicate extensionality

We investigate the relations between the following extensionality principles

- Proposition extensionality
- Predicate extensionality
- Propositional functional extensionality
- Provable-proposition extensionality
- Refutable-proposition extensionality
- Extensional proposition representatives
- Extensional predicate representatives
- Extensional propositional function representatives

Table of contents


```
Set Implicit Arguments.
```

76.1 Definitions

Propositional extensionality

Provable-proposition extensionality

Refutable-proposition extensionality

Predicate extensionality

Propositional functional extensionality

76.2 Propositional and predicate extensionality

76.2.1 Predicate extensionality <-> Propositional extensionality + Propositional functional extensionality

Lemma PredExt_imp_PropExt : PredicateExtensionality → PropositionalExtensionality.

Lemma PredExt_imp_PropFunExt : PredicateExtensionality → PropositionalFunctionalExtensionality.

Lemma PropExt_and_PropFunExt_imp_PredExt :
PropositionalExtensionality → PropositionalFunctionalExtensionality → PredicateExtensionality.

Theorem PropExt_and_PropFunExt_iff_PredExt :
PropositionalExtensionality ∧ PropositionalFunctionalExtensionality ↔ PredicateExtensionality.

76.2.2 Propositional extensionality and provable proposition extensionality

Lemma PropExt_imp_ProvPropExt : PropositionalExtensionality → ProvablePropositionExtensionality.

76.2.3 Propositional extensionality and refutable proposition extensionality

Lemma PropExt_imp_RefutPropExt : PropositionalExtensionality → RefutablePropositionExtensionality.

Chapter 77

Library Coq.Logic.PropFacts

77.1 Basic facts about Prop as a type

An intuitionistic theorem from topos theory [*LambekScott*]

References:

[*LambekScott*] Jim Lambek, Phil J. Scott, Introduction to higher order categorical logic, Cambridge Studies in Advanced Mathematics (Book 7), 1988.

```
Theorem injection_is_involution_in_Prop
  (f : Prop → Prop)
  (inj : ∀ A B, (f A ↔ f B) → (A ↔ B))
  (ext : ∀ A B, A ↔ B → f A ↔ f B)
  : ∀ A, f (f A) ↔ A.
```

Chapter 78

Library Coq.Logic.ConstructiveEpsilon

This provides with a proof of the constructive form of definite and indefinite descriptions for Sigma^0_1-formulas (hereafter called "small" formulas), which infers the sigma-existence (i.e., `Type`-existence) of a witness to a decidable predicate over a countable domain from the regular existence (i.e., `Prop`-existence).

Coq does not allow case analysis on sort `Prop` when the goal is in not in `Prop`. Therefore, one cannot eliminate ∃ *n*, *P n* in order to show {*n* : *nat* | *P n*}. However, one can perform a recursion on an inductive predicate in sort `Prop` so that the returning type of the recursion is in `Type`. This trick is described in Coq'Art book, Sect. 14.2.3 and 15.4. In particular, this trick is used in the proof of *Fix_F* in the module Coq.Init.Wf. There, recursion is done on an inductive predicate *Acc* and the resulting type is in `Type`.

To find a witness of *P* constructively, we program the well-known linear search algorithm that tries P on all natural numbers starting from 0 and going up. Such an algorithm needs a suitable termination certificate. We offer two ways for providing this termination certificate: a direct one, based on an ad-hoc predicate called *before_witness*, and another one based on the predicate *Acc*. For the first one we provide explicit and short proof terms.

Based on ideas from Benjamin Werner and Jean-François Monin

Contributed by Yevgeniy Makarov and Jean-François Monin

```
Require Import Arith.

Section ConstructiveIndefiniteGroundDescription_Direct.

Variable P : nat → Prop.

Hypothesis P_dec : ∀ n, {P n}+{~(P n)}.
```

The termination argument is *before_witness n*, which says that any number before any witness (not necessarily the *x* of ∃ *x* :*A*, *P x*) makes the search eventually stops.

```
Inductive before_witness (n:nat) : Prop :=
  | stop : P n → before_witness n
  | next : before_witness (S n) → before_witness n.

Fixpoint O_witness (n : nat) : before_witness n → before_witness 0 :=
  match n return (before_witness n → before_witness 0) with
    | 0 ⇒ fun b ⇒ b
```

```
    | S n ⇒ fun b ⇒ O_witness n (next n b)
  end.

Definition inv_before_witness :
  ∀ n, before_witness n → ~(P n) → before_witness (S n) :=
  fun n b ⇒
    match b return ¬ P n → before_witness (S n) with
      | stop _ p ⇒ fun not_p ⇒ match (not_p p) with end
      | next _ b ⇒ fun _ ⇒ b
    end.

Fixpoint linear_search m (b : before_witness m) : {n : nat | P n} :=
  match P_dec m with
    | left yes ⇒ exist (fun n ⇒ P n) m yes
    | right no ⇒ linear_search (S m) (inv_before_witness m b no)
  end.

Definition constructive_indefinite_ground_description_nat :
  (∃ n, P n) → {n:nat | P n} :=
  fun e ⇒ linear_search O (let (n, p) := e in O_witness n (stop n p)).

Fixpoint linear_search_smallest (start : nat) (pr : before_witness start) :
  ∀ k : nat, start ≤ k < proj1_sig (linear_search start pr) → ¬P k.

Definition epsilon_smallest :
  (∃ n : nat, P n)
  → { n : nat | P n ∧ ∀ k : nat, k < n → ¬P k }.

End ConstructiveIndefiniteGroundDescription_Direct.

Section ConstructiveIndefiniteGroundDescription_Acc.

Variable P : nat → Prop.

Hypothesis P_decidable : ∀ n : nat, {P n} + {¬ P n}.
```

The predicate *Acc* delineates elements that are accessible via a given relation R. An element is accessible if there are no infinite R-descending chains starting from it.

To use *Fix_F*, we define a relation R and prove that if $\exists\ n,\ P\ n$ then 0 is accessible with respect to R. Then, by induction on the definition of $Acc\ R\ 0$, we show $\{n : nat \mid P\ n\}$.

The relation R describes the connection between the two successive numbers we try. Namely, y is R-less then x if we try y after x, i.e., $y = S\ x$ and $P\ x$ is false. Then the absence of an infinite R-descending chain from 0 is equivalent to the termination of our searching algorithm.

```
Let R (x y : nat) : Prop := x = S y ∧ ¬ P y.

Lemma P_implies_acc : ∀ x : nat, P x → acc x.

Lemma P_eventually_implies_acc : ∀ (x : nat) (n : nat), P (n + x) → acc x.

Corollary P_eventually_implies_acc_ex : (∃ n : nat, P n) → acc 0.
```

In the following statement, we use the trick with recursion on *Acc*. This is also where decidability of P is used.

```
Theorem acc_implies_P_eventually : acc 0 → {n : nat | P n}.
```

Theorem constructive_indefinite_ground_description_nat_Acc :
$(\exists\ n : \textbf{nat},\ P\ n) \to \{n : \textbf{nat} \mid P\ n\}$.

End ConstructiveIndefiniteGroundDescription_Acc.

Section ConstructiveGroundEpsilon_nat.

Variable P : **nat** $\to$ Prop.

Hypothesis $P\_decidable : \forall\ x : \textbf{nat}, \{P\ x\} + \{\neg\ P\ x\}$.

Definition constructive_ground_epsilon_nat $(E : \exists\ n : \textbf{nat},\ P\ n)$: **nat**
:= proj1_sig (constructive_indefinite_ground_description_nat P $P\_decidable$ E).

Definition constructive_ground_epsilon_spec_nat $(E : (\exists\ n,\ P\ n))$: P (constructive_ground_epsilon_nat E)
:= proj2_sig (constructive_indefinite_ground_description_nat P $P\_decidable$ E).

End ConstructiveGroundEpsilon_nat.

Section ConstructiveGroundEpsilon.

For the current purpose, we say that a set A is countable if there are functions $f : A \to nat$ and $g : nat \to A$ such that g is a left inverse of f.

Variable A : Type.
Variable $f : A \to$ **nat**.
Variable g : **nat** $\to A$.

Hypothesis $gof\_eq\_id : \forall\ x : A,\ g\ (f\ x) = x$.

Variable $P : A \to$ Prop.

Hypothesis $P\_decidable : \forall\ x : A, \{P\ x\} + \{\neg\ P\ x\}$.

Definition P' $(x : \textbf{nat})$: Prop $:= P\ (g\ x)$.

Lemma P'_decidable : $\forall\ n : \textbf{nat}, \{P'\ n\} + \{\neg\ P'\ n\}$.

Lemma constructive_indefinite_ground_description : $(\exists\ x : A,\ P\ x) \to \{x : A \mid P\ x\}$.

Lemma constructive_definite_ground_description : $(\exists!\ x : A,\ P\ x) \to \{x : A \mid P\ x\}$.

Definition constructive_ground_epsilon $(E : \exists\ x : A,\ P\ x) : A$
:= proj1_sig (constructive_indefinite_ground_description E).

Definition constructive_ground_epsilon_spec $(E : (\exists\ x,\ P\ x))$: P (constructive_ground_epsilon E)
:= proj2_sig (constructive_indefinite_ground_description E).

End ConstructiveGroundEpsilon.