

# The Coq Proof Assistant

## The standard library

March 4, 2021

Version 8.13.1<sup>1</sup>

$\pi r^2$  Project (formerly LogiCal, then TypiCal)

---

<sup>1</sup>This research was partly supported by IST working group “Types”

V8.13.1, March 4, 2021

©1999-2019, Inria, CNRS and contributors

This material is distributed under the terms of the GNU Lesser General Public License Version 2.1.

# 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.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 2

# Library Coq.Unicode.Utf8\_core

Notation " $\forall x \dots y, P$ " := ( $\forall x, \dots (\forall y, P) \dots$ )  
(at level 200, *x binder*, *y binder*, right associativity,  
format "[ ' ' [ '  $\forall x \dots y$  ' ]' , '/' P ' ]'" ) : *type\_scope*.  
Notation " $\exists x \dots y, P$ " := ( $\exists x, \dots (\exists y, P) \dots$ )  
(at level 200, *x binder*, *y binder*, right associativity,  
format "[ ' ' [ '  $\exists x \dots y$  ' ]' , '/' P ' ]'" ) : *type\_scope*.  
Notation " $x \vee y$ " := ( $x \vee y$ ) (at level 85, right associativity) : *type\_scope*.  
Notation " $x \wedge y$ " := ( $x \wedge y$ ) (at level 80, right associativity) : *type\_scope*.  
Notation " $x \rightarrow y$ " := ( $x \rightarrow y$ )  
(at level 99, *y* at level 200, right associativity) : *type\_scope*.  
Notation " $x \leftrightarrow y$ " := ( $x \leftrightarrow y$ ) (at level 95, no associativity) : *type\_scope*.  
Notation " $\neg x$ " := ( $\neg x$ ) (at level 75, right associativity) : *type\_scope*.  
Notation " $x \neq y$ " := ( $x \neq y$ ) (at level 70) : *type\_scope*.  
Notation " $\lambda' x \dots y, t$ " := ( $\text{fun } x \Rightarrow \dots (\text{fun } y \Rightarrow t) \dots$ )  
(at level 200, *x binder*, *y binder*, right associativity,  
format "[ ' ' [ '  $\lambda' x \dots y$  ' ]' , '/' t ' ]'" ) : *type\_scope*.

## Chapter 3

# Library Coq.Init.Byte

### 3.1 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 ( $b : \text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times \text{bool})))))) : \text{byte}$

$:= \text{match } b \text{ with}$

```

| (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 ( $b : \text{byte}$ ) :  $\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times (\text{bool} \times \text{bool}))))))$

:= 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))))))

```

[illegible]

[illegible]





```

| 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 ( $b : \text{byte}$ ) : of\_bits (to\_bits  $b$ ) =  $b$ .

Lemma to\_bits\_of\_bits ( $b : \_$ ) : to\_bits (of\_bits  $b$ ) =  $b$ .

Definition byte\_of\_byte ( $b : \text{byte}$ ) : **byte** :=  $b$ .

Module Export BYTESYNTAXNOTATIONS.

End BYTESYNTAXNOTATIONS.



## Chapter 4

# 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 5

## Library Coq.Init.Wf

### 5.1 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* :

$\forall P:A \rightarrow \text{Set},$   
 $(\forall x:A, (\forall y:A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow \forall a:A, P\ a.$

Theorem *well\_founded\_ind* :

$\forall P:A \rightarrow \text{Prop},$   
 $(\forall x:A, (\forall y:A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow \forall a:A, P\ a.$

Well-founded fixpoints

Section *FixPoint*.

Variable  $P : A \rightarrow \text{Type}.$

Variable  $F : \forall x:A, (\forall y:A, R\ y\ x \rightarrow P\ y) \rightarrow P\ x.$

Fixpoint *Fix\_F* ( $x:A$ ) ( $a:\mathbf{Acc}\ x$ ) :  $P\ x :=$   
 $F\ (\text{fun } (y:A) (h:R\ y\ x) \Rightarrow \text{Fix\_F } (\text{Acc\_inv } a\ h)).$

Scheme *Acc\_inv\_dep* := Induction for **Acc** Sort Prop.

Lemma *Fix\_F\_eq* ( $x:A$ ) ( $r:\mathbf{Acc}\ x$ ) :  
 $F\ (\text{fun } (y:A) (p:R\ y\ x) \Rightarrow \text{Fix\_F } (x:=y) (\text{Acc\_inv } r\ p)) = \text{Fix\_F } (x:=x)\ r.$

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

$F\_ext :$   
 $\forall (x:A) (f\ g:\forall y:A, R\ y\ x \rightarrow P\ y),$   
 $(\forall (y:A) (p:R\ y\ x), f\ y\ p = g\ y\ p) \rightarrow F\ f = F\ g.$

Lemma *Fix\_F\_inv* :  $\forall (x:A) (r\ s:\mathbf{Acc}\ x), \text{Fix\_F } r = \text{Fix\_F } s.$

Lemma *Fix\_eq* :  $\forall x:A, \text{Fix } x = F\ (\text{fun } (y:A) (p:R\ y\ x) \Rightarrow \text{Fix } y).$

End *FixPoint*.

End *Well\_founded*.

Well-founded fixpoints over pairs

Section *Well\_founded\_2*.

Variables  $A\ B : \text{Type}.$

Variable  $R : A \times B \rightarrow A \times B \rightarrow \text{Prop}.$

Variable  $P : A \rightarrow B \rightarrow \text{Type}.$

Section *FixPoint\_2*.

Variable

$F :$   
 $\forall (x:A) (x':B),$   
 $(\forall (y:A) (y':B), R\ (y, y')\ (x, x') \rightarrow P\ y\ y') \rightarrow P\ x\ x'.$

Fixpoint *Fix\_F\_2* ( $x:A$ ) ( $x':B$ ) ( $a:\mathbf{Acc}\ R\ (x, x')$ ) :  $P\ x\ x' :=$   
 $F$

$(\text{fun } (y:A) (y':B) (h:R\ (y, y')\ (x, x')) \Rightarrow$

```

      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 6

## Library Coq.Init.Nat

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

### 6.1 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`.

#### 6.1.1 Constants

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

#### 6.1.2 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.

### 6.1.3 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'  $\Rightarrow$  compare n' m'
end.

```

Infix "?=" := compare (at level 70) : *nat\_scope*.

#### 6.1.4 Minimum, maximum

```

Fixpoint max n m :=
  match n, m with
  | 0, _  $\Rightarrow$  m
  | S n', 0  $\Rightarrow$  n
  | S n', S m'  $\Rightarrow$  S (max n' m')
  end.

```

```

Fixpoint min n m :=
  match n, m with
  | 0, _  $\Rightarrow$  0
  | S n', 0  $\Rightarrow$  0
  | S n', S m'  $\Rightarrow$  S (min n' m')
  end.

```

#### 6.1.5 Parity tests

```

Fixpoint even n : bool :=
  match n with
  | 0  $\Rightarrow$  true
  | 1  $\Rightarrow$  false
  | S (S n')  $\Rightarrow$  even n'
  end.

```

Definition odd n := negb (even n).

#### 6.1.6 Power

```

Fixpoint pow n m :=
  match m with
  | 0  $\Rightarrow$  1
  | S m  $\Rightarrow$  n  $\times$  (n^m)
  end

```

where "n ^ m" := (pow n m) : *nat\_scope*.

#### 6.1.7 Tail-recursive versions of *add* and *mul*

```

Fixpoint tail_add n m :=
  match n with
  | 0  $\Rightarrow$  m
  | S n  $\Rightarrow$  tail_add n (S m)
  end

```



end.

*tail\_addmul* *r n m* is  $r + n \times m$ .

Fixpoint *tail\_addmul* *r n m* :=

match *n* with

| 0  $\Rightarrow$  *r*

| *S n*  $\Rightarrow$  *tail\_addmul* (*tail\_add* *m r*) *n m*

end.

Definition *tail\_mul* *n m* := *tail\_addmul* 0 *n m*.

### 6.1.8 Conversion with a decimal representation for printing/parsing

Fixpoint of\_uint\_acc (*d*:**Decimal.uint**)(*acc*:nat) :=

match *d* with

| **Decimal.Nil**  $\Rightarrow$  *acc*

| **Decimal.D0** *d*  $\Rightarrow$  of\_uint\_acc *d* (*tail\_mul* ten *acc*)

| **Decimal.D1** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*tail\_mul* ten *acc*))

| **Decimal.D2** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*tail\_mul* ten *acc*)))

| **Decimal.D3** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))

| **Decimal.D4** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))))

| **Decimal.D5** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))))

| **Decimal.D6** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))))

| **Decimal.D7** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))))

| **Decimal.D8** *d*  $\Rightarrow$  of\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* ten *acc*))))))

| **Decimal.D9** *d*  $\Rightarrow$  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* 0.

Fixpoint of\_hex\_uint\_acc (*d*:**Hexadecimal.uint**)(*acc*:nat) :=

match *d* with

| **Hexadecimal.Nil**  $\Rightarrow$  *acc*

| **Hexadecimal.D0** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*tail\_mul* sixteen *acc*)

| **Hexadecimal.D1** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*tail\_mul* sixteen *acc*))

| **Hexadecimal.D2** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*tail\_mul* sixteen *acc*)))

| **Hexadecimal.D3** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))

| **Hexadecimal.D4** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.D5** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.D6** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.D7** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.D8** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.D9** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.Da** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.Db** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))

| **Hexadecimal.Dc** *d*  $\Rightarrow$  of\_hex\_uint\_acc *d* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*S* (*tail\_mul* sixteen *acc*))))))



```
| 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  $n$ ).

### 6.1.9 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*.

### 6.1.10 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
  | 0 ⇒ b
  | S a' ⇒ gcd (b mod (S a')) (S a')
  end.
```

### 6.1.11 Square

Definition square  $n := n \times n$ .

### 6.1.12 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  $0 \leq r \leq q$ . 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  $\text{sqrt } n := \text{sqrt\_iter } n \ 0 \ 0 \ 0$ .

### 6.1.13 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  $q + S r = 2^{(S p)}$  and  $r < 2^p$ . 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  $\Rightarrow$  log2_iter k' (S p) (S q) q
      | S r'  $\Rightarrow$  log2_iter k' p (S q) r'
    end

```

end.

Definition log2  $n := \text{log2\_iter } (\text{pred } n) \ 0 \ 1 \ 0$ .

Iterator on natural numbers

Definition iter ( $n:\text{nat}$ )  $\{A\} (f:A \rightarrow A) (x:A) : A :=$   
 $\text{nat\_rect } (\text{fun } _ \Rightarrow A) \ x \ (\text{fun } _ \Rightarrow f) \ n$ .

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  $\Rightarrow$  0
  | S 0  $\Rightarrow$  0
  | S (S n')  $\Rightarrow$  S (div2 n')
  end.

```

Fixpoint testbit  $a \ n : \text{bool} :=$

```

  match n with
  | 0  $\Rightarrow$  odd a
  | S n  $\Rightarrow$  testbit (div2 a) n
  end.

```

Definition shiftl  $a := \text{nat\_rect } _ \ a \ (\text{fun } _ \Rightarrow \text{double})$ .

Definition shiftr  $a := \text{nat\_rect } _ \ a \ (\text{fun } _ \Rightarrow \text{div2})$ .

Fixpoint bitwise ( $op:\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ )  $n \ a \ b :=$

```

  match n with
  | 0  $\Rightarrow$  0
  | S n'  $\Rightarrow$ 
    (if op (odd a) (odd b) then 1 else 0) +
    2*(bitwise op n' (div2 a) (div2 b))
  end.

```

Definition land  $a \ b := \text{bitwise andb } a \ a \ b$ .

Definition lor  $a \ b := \text{bitwise orb } (\text{max } a \ b) \ a \ b$ .

Definition ldiff  $a \ b := \text{bitwise } (\text{fun } b \ b' \Rightarrow \text{andb } b \ (\text{negb } b')) \ a \ a \ b$ .

Definition lxor  $a \ b := \text{bitwise xorb } (\text{max } a \ b) \ a \ b$ .

## Chapter 7

# Library Coq.Init.Tactics

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

### 7.1 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`,

- $H: \neg A \vdash B$  gives  $\vdash A$
- $H: \neg A \vdash \neg B$  gives  $H: B \vdash A$
- $H: A \vdash B$  gives  $\vdash \neg A$
- $H: A \vdash \neg B$  gives  $H: B \vdash \neg A$
- $H:\text{False}$  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 ( $\neg A$ ))  
  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  $\rightarrow$  t”, “apply  $\leftarrow$  t”, “apply  $\rightarrow$  t in H” and “apply  $\leftarrow$  t in H”. Here t is a term whose type consists of nested dependent and nondependent products with an equivalence  $A \leftrightarrow B$  as the conclusion. The tactics with “ $\rightarrow$ ” in their names apply  $A \rightarrow B$  while those with “ $\leftarrow$ ” in the name apply  $B \rightarrow A$ .

```
Ltac find_equiv H :=
let T := type of H in
lazymatch T with
| ?A  $\rightarrow$  ?B  $\Rightarrow$ 
  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]
|  $\forall$  x : ?t, _  $\Rightarrow$ 
  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  $\leftrightarrow$  ?B  $\Rightarrow$  idtac
| _  $\Rightarrow$  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" " $\rightarrow$ " constr(lemma) :=
bapply lemma ltac:(fun H  $\Rightarrow$  destruct H as [H _]; apply H).
```

```
Tactic Notation "apply" " $\leftarrow$ " constr(lemma) :=
bapply lemma ltac:(fun H  $\Rightarrow$  destruct H as [_ H]; apply H).
```

```
Tactic Notation "apply" " $\rightarrow$ " constr(lemma) "in" hyp(J) :=
bapply lemma ltac:(fun H  $\Rightarrow$  destruct H as [H _]; apply H in J).
```

```
Tactic Notation "apply" " $\leftarrow$ " constr(lemma) "in" hyp(J) :=
bapply lemma ltac:(fun H  $\Rightarrow$  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
  | _  $\wedge$  _  $\Rightarrow$  exact H || destruct_hyp H
  | _  $\Rightarrow$  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 8

# Library Coq.Init.Tauto

### 8.1 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:  $\forall (- : ?X1), ?X2 \vdash - \Rightarrow$ 
  flatten_contravariant_conj flags X1 X2 id

| id:  $\forall (- : \text{Coq.Init.Logic.iff } ?X1 ?X2), ?X3 \vdash - \Rightarrow$ 
  assert ( $\forall (- : \forall -:X1, X2), \forall (- : \forall -: X2, X1), X3$ )
by (do 2 intro; apply id; split; assumption);
  clear id
| id:  $\forall (- : ?X1), ?X2 \vdash - \Rightarrow$ 
  flatten_contravariant_disj flags X1 X2 id

|  $\vdash ?X1 \Rightarrow \text{is\_conj flags X1}$ ; split
|  $\vdash (\text{Coq.Init.Logic.iff } - -) \Rightarrow \text{split}$ 
|  $\vdash (\text{Coq.Init.Logic.not } -) \Rightarrow \text{red}$ 
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:  $\forall (- : \forall (- : ?X1), ?X2), ?X3 \vdash - \Rightarrow$ 
  cut X3;
  [ intro; clear id; t_tauto_intuit
  | cut ( $\forall (- : X1), X2$ );
    [ exact id
    | generalize (fun y:X2  $\Rightarrow$  id (fun x:X1  $\Rightarrow$  y)); intro; clear id;
      solve [ t_tauto_intuit ]]]
| id:  $\forall (- : \text{not } ?X1), ?X3 \vdash - \Rightarrow$ 
  cut X3;
  [ intro; clear id; t_tauto_intuit
  | cut (not X1); [ exact id | clear id; intro; solve [t_tauto_intuit ]]]
|  $\vdash ?X1 \Rightarrow$ 
  is_disj flags X1; solve [left;t_tauto_intuit | right;t_tauto_intuit]
end
||

match goal with |  $\vdash \forall (- : -), - \Rightarrow$  intro; t_tauto_intuit
|  $\vdash - \Rightarrow$  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 9

# Library Coq.Init.Datatypes

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

### 9.1 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.
```

### 9.2 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.

```

### 9.3 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*.

### 9.4 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 × 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 => 0
  | _ :: 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.

```

## 9.5 The comparison datatype

```

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

```

Lemma comparison\_eq\_stable ( $c\ c' : \text{comparison}$ ) :  $\sim\sim\ c = c' \rightarrow c = c'$ .

```

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

```

Lemma CompOpp\_involutive  $c : \text{CompOpp} (\text{CompOpp } c) = c$ .

Lemma CompOpp\_inj  $c\ c' : \text{CompOpp } c = \text{CompOpp } c' \rightarrow c = c'$ .

Lemma CompOpp\_iff :  $\forall\ c\ c', \text{CompOpp } c = c' \leftrightarrow c = \text{CompOpp } c'$ .

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 than 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*.

## 9.6 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 10

# Library Coq.Init.Notations

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

Notations for propositional connectives

Reserved Notation " $x \rightarrow y$ " (at level 99, right associativity,  $y$  at level 200).

Reserved Notation " $x \leftrightarrow y$ " (at level 95, no associativity).

Reserved Notation " $x \wedge y$ " (at level 80, right associativity).

Reserved Notation " $x \vee y$ " (at level 85, right associativity).

Reserved Notation " $\sim x$ " (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 " $x \leq y$ " (at level 70, no associativity).

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

Reserved Notation " $x \geq y$ " (at level 70, no associativity).

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

Reserved Notation " $x \leq y \leq z$ " (at level 70,  $y$  at *next* level).

Reserved Notation " $x \leq y < z$ " (at level 70,  $y$  at *next* level).

Reserved Notation " $x < y < z$ " (at level 70,  $y$  at *next* level).

Reserved Notation " $x < y \leq z$ " (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  $x \wedge y$  (at level 30, right associativity).

Notations for booleans

Reserved Notation  $x \parallel y$  (at level 50, left associativity).  
Reserved Notation  $x \&\& y$  (at level 40, left associativity).

Notations for pairs

Reserved Notation  $(x, y, \dots, z)$  (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  $\{x : A \mid P\}$

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

Reserved Notation  $\{x : A \mid P\}$  (at level 0,  $x$  at level 99).  
Reserved Notation  $\{x : A \mid P \& Q\}$  (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  $\{ ' pat \mid P \}$   
(at level 0, *pat strict pattern*, *format*  $\{ ' pat \mid P \}$ ).

Reserved Notation  $\{ ' pat \mid P \& Q \}$   
(at level 0, *pat strict pattern*, *format*  $\{ ' pat \mid P \& Q \}$ ).

Reserved Notation  $\{ ' pat : A \mid P \}$   
(at level 0, *pat strict pattern*, *format*  $\{ ' pat : A \mid P \}$ ).

Reserved Notation  $\{ ' pat : A \mid P \& Q \}$   
(at level 0, *pat strict pattern*, *format*  $\{ ' pat : A \mid P \& Q \}$ ).

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*  $\Rightarrow$  *u* | \_  $\Rightarrow$  *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 11

# 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  $\text{nat} \times \text{nat}$  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 :  $\forall n:\mathbf{nat}, n = \text{pred } (\text{S } n)$ .

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 :  $\forall n m:\mathbf{nat}, n \neq m \rightarrow \text{S } n \neq \text{S } m$ .

#[global]

Hint Resolve not\_eq\_S: core.

Definition lsSucc (n:nat) : Prop :=

match n with

| 0  $\Rightarrow$  False

| S p  $\Rightarrow$  True

end.

Zero is not the successor of a number

Theorem O\_S :  $\forall n:\mathbf{nat}, 0 \neq \text{S } n$ .

#[global]

Hint Resolve O\_S: core.

Theorem n\_Sn :  $\forall n:\mathbf{nat}, n \neq \text{S } n$ .

#[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 :  $\forall n:\mathbf{nat}, n = n + 0$ .

#[global]

Remove Hints eq\_refl: core.

#[global]

Hint Resolve plus\_n\_O eq\_refl: core.

Lemma plus\_O\_n :  $\forall n:\mathbf{nat}, 0 + n = n$ .

Lemma plus\_n\_Sm :  $\forall n m:\mathbf{nat}, \text{S } (n + m) = n + \text{S } m$ .

#[global]

Hint Resolve plus\_n\_Sm: core.

Lemma plus\_Sn\_m :  $\forall n m:\mathbf{nat}, \text{S } n + m = \text{S } (n + m)$ .

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` :  $\forall n:\mathbf{nat}, 0 = n \times 0$ .

*#[global]*

Hint Resolve `mult_n_O`: *core*.

Lemma `mult_n_Sm` :  $\forall n\ m:\mathbf{nat}, n \times m + n = n \times \mathbf{S}\ m$ .

*#[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  $n \geq m$

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` ( $n:\mathbf{nat}$ ) :  $\mathbf{nat} \rightarrow \mathbf{Prop}$  :=

| `le_n` :  $n \leq n$

| `le_S` :  $\forall m:\mathbf{nat}, n \leq m \rightarrow n \leq \mathbf{S}\ m$

where `"n <= m"` := (`le n m`) : *nat\_scope*.

*#[global]*

Hint Constructors `le`: *core*.

Definition `lt` ( $n\ m:\mathbf{nat}$ ) :=  $\mathbf{S}\ n \leq m$ .

*#[global]*

Hint Unfold `lt`: *core*.

Infix `"<"` := `lt` : *nat\_scope*.

Definition `ge` ( $n\ m:\mathbf{nat}$ ) :=  $m \leq n$ .

*#[global]*

Hint Unfold `ge`: *core*.

Infix `"≥"` := `ge` : *nat\_scope*.

Definition `gt` ( $n\ m:\mathbf{nat}$ ) :=  $m < n$ .

*#[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 12

# Library Coq.Init.Ltac

Export Set *Default Proof Mode* "Classic".

## Chapter 13

# Library Coq.Init.Number

### 13.1 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 14

# 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 15

# 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

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

$\#[\text{universes}(\text{template})]$

Inductive **sig** (A:Type) (P:A  $\rightarrow$  Prop) : Type :=  
  exist :  $\forall x:A, P \ x \rightarrow \text{sig } P$ .

$\#[\text{universes}(\text{template})]$

Inductive **sig2** (A:Type) (P Q:A  $\rightarrow$  Prop) : Type :=  
  exist2 :  $\forall x:A, P \ x \rightarrow Q \ x \rightarrow \text{sig2 } P \ Q$ .

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

$\#[\text{universes}(\text{template})]$

Inductive **sigT** (A:Type) (P:A  $\rightarrow$  Type) : Type :=  
  existT :  $\forall x:A, P \ x \rightarrow \text{sigT } P$ .

$\#[\text{universes}(\text{template})]$

Inductive **sigT2** (A:Type) (P Q:A  $\rightarrow$  Type) : Type :=  
  existT2 :  $\forall x:A, P \ x \rightarrow Q \ x \rightarrow \text{sigT2 } P \ Q$ .

Notation "{ x | P }" := (**sig** (fun x  $\Rightarrow$  P)) : type\_scope.

Notation "{ x | P & Q }" := (**sig2** (fun x  $\Rightarrow$  P) (fun x  $\Rightarrow$  Q)) : type\_scope.

Notation "{ x : A | P }" := (**sig** (A:=A) (fun x  $\Rightarrow$  P)) : type\_scope.

Notation "{ x : A | P & Q }" := (**sig2** (A:=A) (fun x  $\Rightarrow$  P) (fun x  $\Rightarrow$  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  $A : \text{Type}$ .

Variable  $P : A \rightarrow \text{Prop}$ .

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 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  $A : \text{Type}$ .

Variables  $P\ Q : A \rightarrow \text{Prop}$ .

Definition  $proj3\_sig\ (e : \mathbf{sig2}\ P\ Q) :=$

$\text{let } (a, b, c) \text{ return } Q\ (proj1\_sig\ (sig\_of\_sig2\ e)) := e \text{ in } c.$

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  $A : \text{Type}$ .

Variable  $P : A \rightarrow \text{Type}$ .

Definition  $projT1\ (x:\mathbf{sigT}\ P) : A := \text{match } x \text{ with}$

$\mid \text{existT } \_ \ a \ \_ \Rightarrow a$   
end.

Definition  $projT2\ (x:\mathbf{sigT}\ P) : P\ (projT1\ x) :=$

$\text{match } x \text{ return } P\ (projT1\ x) \text{ with}$   
 $\mid \text{existT } \_ \ h \Rightarrow h$   
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  $\text{let } (a, \_, \_) := X \text{ in } a$

Definition  $sigT\_of\_sigT2\ (A : \text{Type})\ (P\ Q : A \rightarrow \text{Type})\ (X : \mathbf{sigT2}\ P\ Q) : \mathbf{sigT}\ P$

$:= \text{existT } P$   
 $\quad (\text{let } (a, \_, \_) := X \text{ in } a)$   
 $\quad (\text{let } (x, p, \_) \text{ as } s \text{ return } (P\ (\text{let } (a, \_, \_) := s \text{ in } a)) := X \text{ in } p).$

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  $A : \text{Type}$ .

Variables  $P Q : A \rightarrow \text{Type}$ .

Definition  $\text{projT3} (e : \mathbf{sigT2} P Q) :=$

$\text{let } (a, b, c) \text{ return } Q (\text{projT1} (\text{sigT\_of\_sigT2 } e)) := e \text{ in } c.$

End Projections2.

$\text{sigT}$  of a predicate is equivalent to  $\text{sig}$

Definition  $\text{sig\_of\_sigT} (A : \text{Type}) (P : A \rightarrow \text{Prop}) (X : \mathbf{sigT} P) : \mathbf{sig} P$   
 $:= \text{exist } P (\text{projT1 } X) (\text{projT2 } X).$

Definition  $\text{sigT\_of\_sig} (A : \text{Type}) (P : A \rightarrow \text{Prop}) (X : \mathbf{sig} P) : \mathbf{sigT} P$   
 $:= \text{existT } P (\text{proj1\_sig } X) (\text{proj2\_sig } X).$

$\text{sigT2}$  of a predicate is equivalent to  $\text{sig2}$

Definition  $\text{sig2\_of\_sigT2} (A : \text{Type}) (P Q : A \rightarrow \text{Prop}) (X : \mathbf{sigT2} P Q) : \mathbf{sig2} P Q$   
 $:= \text{exist2 } P Q (\text{projT1} (\text{sigT\_of\_sigT2 } X)) (\text{projT2} (\text{sigT\_of\_sigT2 } X)) (\text{projT3 } X).$

Definition  $\text{sigT2\_of\_sig2} (A : \text{Type}) (P Q : A \rightarrow \text{Prop}) (X : \mathbf{sig2} P Q) : \mathbf{sigT2} P Q$   
 $:= \text{existT2 } P Q (\text{proj1\_sig} (\text{sig\_of\_sig2 } X)) (\text{proj2\_sig} (\text{sig\_of\_sig2 } X)) (\text{proj3\_sig } X).$

$\eta$  Principles Definition  $\text{sigT\_eta} \{A P\} (p : \{a : A \ \& \ P \ a\})$   
 $: p = \text{existT } \_ (\text{projT1 } p) (\text{projT2 } p).$

Definition  $\text{sig\_eta} \{A P\} (p : \{a : A \mid P \ a\})$   
 $: p = \text{exist } \_ (\text{proj1\_sig } p) (\text{proj2\_sig } p).$

Definition  $\text{sigT2\_eta} \{A P Q\} (p : \{a : A \ \& \ P \ a \ \& \ Q \ a\})$   
 $: p = \text{existT2 } \_ \_ (\text{projT1} (\text{sigT\_of\_sigT2 } p)) (\text{projT2} (\text{sigT\_of\_sigT2 } p)) (\text{projT3 } p).$

Definition  $\text{sig2\_eta} \{A P Q\} (p : \{a : A \mid P \ a \ \& \ Q \ a\})$   
 $: p = \text{exist2 } \_ \_ (\text{proj1\_sig} (\text{sig\_of\_sig2 } p)) (\text{proj2\_sig} (\text{sig\_of\_sig2 } p)) (\text{proj3\_sig } p).$

$\exists x : A, B$  is equivalent to  $\text{inhabited} \{x : A \mid B\}$  Lemma  $\text{exists\_to\_inhabited\_sig} \{A P\} : (\exists x : A, P \ x) \rightarrow \text{inhabited} \{x : A \mid P \ x\}.$

Lemma  $\text{inhabited\_sig\_to\_exists} \{A P\} : \text{inhabited} \{x : A \mid P \ x\} \rightarrow \exists x : A, P \ x.$

Equality of sigma types

Import *EqNotations*.

Equality for  $\text{sigT}$  Section  $\text{sigT}$ .

Local Unset Implicit Arguments.

Projecting an equality of a pair to equality of the first components Definition  $\text{projT1\_eq}$   
 $\{A\} \{P : A \rightarrow \text{Type}\} \{u \ v : \{a : A \ \& \ P \ a\}\} (p : u = v)$   
 $: u.1 = v.1$   
 $:= \text{f\_equal} (\text{fun } x \Rightarrow x.1) p.$

Projecting an equality of a pair to equality of the second components Definition  $\text{projT2\_eq}$   
 $\{A\} \{P : A \rightarrow \text{Type}\} \{u \ v : \{a : A \ \& \ P \ a\}\} (p : u = v)$   
 $: \text{rew projT1\_eq } p \text{ in } u.2 = v.2$   
 $:= \text{rew dependent } p \text{ in eq\_refl}.$

Equality of  $\text{sigT}$  is itself a  $\text{sigT}$  (forwards-reasoning version) Definition  $\text{eq\_existT\_uncurried}$   
 $\{A : \text{Type}\} \{P : A \rightarrow \text{Type}\} \{u1 \ v1 : A\} \{u2 : P \ u1\} \{v2 : P \ v1\}$   
 $(pq : \{p : u1 = v1 \ \& \ \text{rew } p \text{ in } u2 = v2\})$

: (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  $\leftrightarrow$ , but for simplicity, we don't.      **Definition** eq\_sigT\_uncurried\_iff {A P}  
 (u v : { a : A & P a })  
 : u = v  $\leftrightarrow$  { 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)

$(u \ v : \{ a : A \ \& \ P \ a \})$   
 $: u = v \leftrightarrow (u.1 = v.1)$   
 $:= \text{conj} (\text{fun } p \Rightarrow \text{f\_equal} (\text{@projT1 } \_ \_) p) (\text{eq\_sigT\_hprop } P\_hprop \ u \ v).$

Non-dependent classification of equality of *sigT*      **Definition** `eq_sigT_nondep`  $\{A \ B : \text{Type}\}$   
 $(u \ v : \{ a : A \ \& \ B \})$   
 $(p : u.1 = v.1) (q : u.2 = v.2)$   
 $: u = v$   
 $:= \text{@eq\_sigT } \_ \_ \ u \ v \ p \ (\text{eq\_trans} (\text{rew\_const } \_ \_) q).$

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

**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`  
 $\{A\} \{P : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \}\} (p : u = v)$   
 $: \text{proj1\_sig } u = \text{proj1\_sig } v$   
 $:= \text{f\_equal} (\text{@proj1\_sig } \_ \_) p.$

Projecting an equality of a pair to equality of the second components      **Definition** `proj2_sig_eq`  
 $\{A\} \{P : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \}\} (p : u = v)$   
 $: \text{rew proj1\_sig\_eq } p \text{ in proj2\_sig } u = \text{proj2\_sig } v$   
 $:= \text{rew dependent } p \text{ in eq\_refl}.$

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

Equality of *sig* is itself a *sig* (backwards-reasoning version)      **Definition** `eq_sig_uncurried`  $\{A : \text{Type}\} \{P : A \rightarrow \text{Prop}\} (u \ v : \{ a : A \mid P \ a \})$   
 $(pq : \{ p : \text{proj1\_sig } u = \text{proj1\_sig } v \mid \text{rew } p \text{ in proj2\_sig } u = \text{proj2\_sig } v \})$   
 $: u = v.$

Curried version of proving equality of sigma types      **Definition** `eq_sig`  $\{A : \text{Type}\} \{P : A \rightarrow \text{Prop}\} (u \ v : \{ a : A \mid P \ a \})$   
 $(p : \text{proj1\_sig } u = \text{proj1\_sig } v) (q : \text{rew } p \text{ in proj2\_sig } u = \text{proj2\_sig } v)$   
 $: u = v$   
 $:= \text{eq\_sig\_uncurried } u \ v \ (\text{exist } \_ \ p \ q).$

Induction principle for `@eq (sig _)`      **Definition** `eq_sig_rect`  $\{A \ P\} \{u \ v : \{ a : A \mid P \ a \}\}$   
 $(Q : u = v \rightarrow \text{Type})$   
 $(f : \forall p \ q, Q (\text{eq\_sig } u \ v \ p \ q))$   
 $: \forall p, Q \ p.$

Definition eq\_sig\_rec  $\{A\ P\ u\ v\} (Q : u = v :> \{a : A \mid P\ a\} \rightarrow \text{Set}) := \text{eq\_sig\_rect } Q.$

Definition eq\_sig\_ind  $\{A\ P\ u\ v\} (Q : u = v :> \{a : A \mid P\ a\} \rightarrow \text{Prop}) := \text{eq\_sig\_rec } Q.$

Equality of *sig* when the property is an hProp      Definition eq\_sig\_hprop  $\{A\} \{P : A \rightarrow \text{Prop}\}$   
 $(P\_hprop : \forall (x : A) (p\ q : P\ x), p = q)$   
 $(u\ v : \{a : A \mid P\ a\})$   
 $(p : \text{proj1\_sig } u = \text{proj1\_sig } v)$   
 $: u = v$   
 $:= \text{eq\_sig } u\ v\ p\ (P\_hprop\ \_ \_).$

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  $\{A\} \{P : A \rightarrow \text{Prop}\}$

$(u\ v : \{a : A \mid P\ a\})$   
 $: u = v \leftrightarrow \{p : \text{proj1\_sig } u = \text{proj1\_sig } v \mid \text{rew } p \text{ in } \text{proj2\_sig } u = \text{proj2\_sig } v\}.$

Equivalence of equality of *sig* involving hProps with equality of the first components      Definition eq\_sig\_hprop\_iff  $\{A\} \{P : A \rightarrow \text{Prop}\} (P\_hprop : \forall (x : A) (p\ q : P\ x), p = q)$

$(u\ v : \{a : A \mid P\ a\})$   
 $: u = v \leftrightarrow (\text{proj1\_sig } u = \text{proj1\_sig } v)$   
 $:= \text{conj } (\text{fun } p \Rightarrow \text{f\_equal } (@\text{proj1\_sig } \_ \_) p) (\text{eq\_sig\_hprop } P\_hprop\ u\ v).$

Lemma rew\_sig  $\{A\ x\} \{P : A \rightarrow \text{Type}\} (Q : \forall a, P\ a \rightarrow \text{Prop}) (u : \{p : P\ x \mid Q\ x\ p\}) \{y\} (H : x = y)$

$: \text{rew } [\text{fun } a \Rightarrow \{p : P\ a \mid Q\ a\ p\}] H \text{ in } u$   
 $= \text{exist}$   
 $(Q\ y)$   
 $(\text{rew } H \text{ in } \text{proj1\_sig } u)$   
 $(\text{rew dependent } H \text{ in } \text{proj2\_sig } u).$

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  $\{A\} \{P\ Q : A \rightarrow \text{Type}\} \{u\ v : \{a : A \ \&\ P\ a \ \&\ Q\ a\}\} (p : u = v)$   
 $: u = v :> \{a : A \ \&\ P\ a\}$   
 $:= \text{f\_equal } \_ p.$

Definition projT1\_of\_sigT2\_eq  $\{A\} \{P\ Q : A \rightarrow \text{Type}\} \{u\ v : \{a : A \ \&\ P\ a \ \&\ Q\ a\}\} (p : u = v)$   
 $: u.1 = v.1$   
 $:= \text{projT1\_eq } (\text{sigT\_of\_sigT2\_eq } p).$

Projecting an equality of a pair to equality of the second components      Definition projT2\_of\_sigT2\_eq  $\{A\} \{P\ Q : A \rightarrow \text{Type}\} \{u\ v : \{a : A \ \&\ P\ a \ \&\ Q\ a\}\} (p : u = v)$   
 $: \text{rew } \text{projT1\_of\_sigT2\_eq } p \text{ in } u.2 = v.2$   
 $:= \text{rew dependent } p \text{ in } \text{eq\_refl}.$

Projecting an equality of a pair to equality of the third components      Definition projT3\_eq  $\{A\} \{P\ Q : A \rightarrow \text{Type}\} \{u\ v : \{a : A \ \&\ P\ a \ \&\ Q\ a\}\} (p : u = v)$   
 $: \text{rew } \text{projT1\_of\_sigT2\_eq } p \text{ in } \text{projT3 } u = \text{projT3 } v$

`:= rew dependent p in eq_refl.`

Equality of *sigT2* is itself a *sigT2* (forwards-reasoning version)    **Definition** `eq_existT2_uncurried`  
`{A : Type} {P Q : A → 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 : Type} {P Q : A → 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 : Type} {P Q : A → 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 : ∀ (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`  
`↔ { 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 → Type)`  
`(f : ∀ p q r, R (eq_sigT2 u v p q r))`  
`: ∀ p, R p.`

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

**Definition** `eq_sigT2_ind` `{A P Q u v} (R : u = v :> { a : A & P a & Q a } → 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 : ∀ (x : A) (p q : Q x), p = q)`  
`(u v : { a : A & P a & Q a })`  
`: u = v ↔ (u = v :> { a : A & P a })`

$$:= \text{conj } (\text{fun } p \Rightarrow \text{f\_equal } (@\text{sigT\_of\_sigT2 } \_ \_ \_) p) (\text{eq\_sigT2\_hprop } Q\_hprop \ u \ v).$$

Non-dependent classification of equality of *sigT*      Definition `eq_sigT2_nondep`  $\{A \ B \ C :$   
`Type`  $\} (u \ v : \{ a : A \ \& \ B \ \& \ C \})$   

$$(p : u.1 = v.1) (q : u.2 = v.2) (r : \text{projT3 } u = \text{projT3 } v)$$
  

$$: u = v$$
  

$$:= @\text{eq\_sigT2 } \_ \_ \_ u \ v \ p \ (\text{eq\_trans } (\text{rew\_const } \_ \_) q) (\text{eq\_trans } (\text{rew\_const } \_ \_) r).$$

Classification of transporting across an equality of *sigT2s*      Lemma `rew_sigT2`  $\{A \ x\} \{P : A \rightarrow$   
`Type`  $\} (Q \ R : \forall a, P \ a \rightarrow \text{Prop})$   

$$(u : \{ p : P \ x \ \& \ Q \ x \ p \ \& \ R \ x \ p \})$$
  

$$\{y\} (H : x = y)$$
  

$$: \text{rew } [\text{fun } a \Rightarrow \{ p : P \ a \ \& \ Q \ a \ p \ \& \ R \ a \ p \}] \ H \ \text{in } u$$
  

$$= \text{existT2}$$
  

$$(Q \ y)$$
  

$$(R \ y)$$
  

$$(\text{rew } H \ \text{in } u.1)$$
  

$$(\text{rew dependent } H \ \text{in } u.2)$$
  

$$(\text{rew dependent } H \ \text{in } \text{projT3 } u).$$

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`  
 $\{A\} \{P \ Q : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \}\} (p : u = v)$   

$$: u = v :> \{ a : A \mid P \ a \}$$
  

$$:= \text{f\_equal } \_ \ p.$$

Definition `proj1_sig_of_sig2_eq`  $\{A\} \{P \ Q : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \}\} (p : u$   
 $= v)$   

$$: \text{proj1\_sig } u = \text{proj1\_sig } v$$
  

$$:= \text{proj1\_sig\_eq } (\text{sig\_of\_sig2\_eq } p).$$

Projecting an equality of a pair to equality of the second components      Definition `proj2_sig_of_sig2_eq`  
 $\{A\} \{P \ Q : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \}\} (p : u = v)$   

$$: \text{rew proj1\_sig\_of\_sig2\_eq } p \ \text{in } \text{proj2\_sig } u = \text{proj2\_sig } v$$
  

$$:= \text{rew dependent } p \ \text{in } \text{eq\_refl}.$$

Projecting an equality of a pair to equality of the third components      Definition `proj3_sig_eq`  
 $\{A\} \{P \ Q : A \rightarrow \text{Prop}\} \{u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \}\} (p : u = v)$   

$$: \text{rew proj1\_sig\_of\_sig2\_eq } p \ \text{in } \text{proj3\_sig } u = \text{proj3\_sig } v$$
  

$$:= \text{rew dependent } p \ \text{in } \text{eq\_refl}.$$

Equality of *sig2* is itself a *sig2* (fowards-reasoning version)      Definition `eq_exist2_uncurried`  
 $\{A\} \{P \ Q : A \rightarrow \text{Prop}\}$   

$$\{u1 \ v1 : A\} \{u2 : P \ u1\} \{v2 : P \ v1\} \{u3 : Q \ u1\} \{v3 : Q \ v1\}$$
  

$$(pqr : \{ p : u1 = v1$$
  

$$\mid \text{rew } p \ \text{in } u2 = v2 \ \& \ \text{rew } p \ \text{in } u3 = v3 \})$$
  

$$: \text{exist2 } \_ \_ \ u1 \ u2 \ u3 = \text{exist2 } \_ \_ \ v1 \ v2 \ v3.$$

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



$(pqr : \{ p : \text{proj1\_sig } u = \text{proj1\_sig } v$   
 $\quad | \text{rew } p \text{ in proj2\_sig } u = \text{proj2\_sig } v \ \& \ \text{rew } p \text{ in proj3\_sig } u = \text{proj3\_sig } v \})$   
 $: u = v.$

Curried version of proving equality of sigma types      **Definition** `eq_sig2`  $\{A\} \{P \ Q : A \rightarrow \text{Prop}\}$

$(u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \})$   
 $(p : \text{proj1\_sig } u = \text{proj1\_sig } v)$   
 $(q : \text{rew } p \text{ in proj2\_sig } u = \text{proj2\_sig } v)$   
 $(r : \text{rew } p \text{ in proj3\_sig } u = \text{proj3\_sig } v)$   
 $: u = v$   
 $:= \text{eq\_sig2\_uncurried } u \ v \ (\text{exist2 } \_ \_ p \ q \ r).$

Equality of *sig2* when the second property is an hProp      **Definition** `eq_sig2_hprop`  $\{A\} \{P \ Q : A \rightarrow \text{Prop}\}$

$(Q\_hprop : \forall (x : A) (p \ q : Q \ x), p = q)$   
 $(u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \})$   
 $(p : u = v :> \{ a : A \mid P \ a \})$   
 $: u = v$   
 $:= \text{eq\_sig2 } u \ v \ (\text{proj1\_sig\_eq } p) \ (\text{proj2\_sig\_eq } p) \ (Q\_hprop \_ \_).$

Equivalence of equality of *sig2* with a *sig2* of equality We could actually prove an isomorphism here, and not just  $\leftrightarrow$ , but for simplicity, we don't.      **Definition** `eq_sig2_uncurried_iff`  $\{A \ P \ Q\}$

$(u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \})$   
 $: u = v$   
 $\leftrightarrow \{ p : \text{proj1\_sig } u = \text{proj1\_sig } v$   
 $\quad | \text{rew } p \text{ in proj2\_sig } u = \text{proj2\_sig } v \ \& \ \text{rew } p \text{ in proj3\_sig } u = \text{proj3\_sig } v \}.$

Induction principle for `@eq (sig2 _ _)`      **Definition** `eq_sig2_rect`  $\{A \ P \ Q\} \{u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \}\}$

$(R : u = v \rightarrow \text{Type})$   
 $(f : \forall p \ q \ r, R \ (\text{eq\_sig2 } u \ v \ p \ q \ r))$   
 $: \forall p, R \ p.$

**Definition** `eq_sig2_rec`  $\{A \ P \ Q \ u \ v\} (R : u = v :> \{ a : A \mid P \ a \ \& \ Q \ a \} \rightarrow \text{Set}) := \text{eq\_sig2\_rect } R.$

**Definition** `eq_sig2_ind`  $\{A \ P \ Q \ u \ v\} (R : u = v :> \{ a : A \mid P \ a \ \& \ Q \ a \} \rightarrow \text{Prop}) := \text{eq\_sig2\_rec } R.$

Equivalence of equality of *sig2* involving hProps with equality of the first components      **Definition** `eq_sig2_hprop_iff`  $\{A\} \{P \ Q : A \rightarrow \text{Prop}\}$

$(Q\_hprop : \forall (x : A) (p \ q : Q \ x), p = q)$   
 $(u \ v : \{ a : A \mid P \ a \ \& \ Q \ a \})$   
 $: u = v \leftrightarrow (u = v :> \{ a : A \mid P \ a \})$   
 $:= \text{conj } (\text{fun } p \Rightarrow \text{f\_equal } (\text{@sig\_of\_sig2 } \_ \_ \_) \ p) \ (\text{eq\_sig2\_hprop } Q\_hprop \ u \ v).$

Non-dependent classification of equality of *sig*      **Definition** `eq_sig2_nondep`  $\{A\} \{B \ C : \text{Prop}\}$

$(u \ v : \text{@sig2 } A \ (\text{fun } \_ \Rightarrow B) \ (\text{fun } \_ \Rightarrow C))$   
 $(p : \text{proj1\_sig } u = \text{proj1\_sig } v) \ (q : \text{proj2\_sig } u = \text{proj2\_sig } v) \ (r : \text{proj3\_sig } u = \text{proj3\_sig } v)$   
 $: u = v$   
 $:= \text{@eq\_sig2 } \_ \_ \_ u \ v \ p \ (\text{eq\_trans } (\text{rew\_const } \_ \_) \ q) \ (\text{eq\_trans } (\text{rew\_const } \_ \_) \ r).$

Classification of transporting across an equality of *sig2*s      **Lemma** `rew_sig2`  $\{A \ x\} \{P : A \rightarrow \text{Type}\} (Q \ R : \forall a, P \ a \rightarrow \text{Prop})$

```

      (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 0 = 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 16

# Library Coq.Init.Hexadecimal

### 16.1 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 => 0
  | 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
```

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  $d_{rev} \Rightarrow \text{let } (r, n) := \text{aux } d_{rev} \text{ in pair } r \text{ (S } n)$ 
    |  $\_ \Rightarrow \text{pair } d_{rev} \text{ O}$ 
  end in
  let  $(r, n) := \text{aux } (\text{rev } d) \text{ in pair } (\text{rev } r) \text{ } n$ .

```

Definition  $\text{nztail\_int } d :=$

```

  match  $d$  with
  | Pos  $d \Rightarrow \text{let } (r, n) := \text{nztail } d \text{ in pair (Pos } r) \text{ } n$ 
  | Neg  $d \Rightarrow \text{let } (r, n) := \text{nztail } d \text{ in pair (Neg } r) \text{ } n$ 
  end.

```

$\text{del\_head } n \text{ } d$  removes  $n$  digits at beginning of  $d$  or returns *zero* if  $d$  has less than  $n$  digits.

Fixpoint  $\text{del\_head } n \text{ } d :=$

```

  match  $n$  with
  | O  $\Rightarrow d$ 
  | S  $n \Rightarrow$ 
    match  $d$  with
    | Nil  $\Rightarrow \text{zero}$ 
    | 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  $d \Rightarrow$ 
       $\text{del\_head } n \text{ } d$ 
    end
  end.

```

Definition  $\text{del\_head\_int } n \text{ } d :=$

```

  match  $d$  with
  | Pos  $d \Rightarrow \text{del\_head } n \text{ } d$ 
  | Neg  $d \Rightarrow \text{del\_head } n \text{ } d$ 
  end.

```

$\text{del\_tail } n \text{ } d$  removes  $n$  digits at end of  $d$  or returns *zero* if  $d$  has less than  $n$  digits.

Definition  $\text{del\_tail } n \text{ } d := \text{rev } (\text{del\_head } n \text{ } (\text{rev } d))$ .

Definition  $\text{del\_tail\_int } n \text{ } d :=$

```

  match  $d$  with
  | Pos  $d \Rightarrow \text{Pos } (\text{del\_tail } n \text{ } d)$ 
  | Neg  $d \Rightarrow \text{Neg } (\text{del\_tail } n \text{ } d)$ 
  end.

```

Module LITTLE.

Successor of little-endian numbers

Fixpoint  $\text{succ } d :=$

```

  match  $d$  with
  | Nil  $\Rightarrow \text{D1 Nil}$ 
  | D0  $d \Rightarrow \text{D1 } d$ 
  | D1  $d \Rightarrow \text{D2 } d$ 
  | D2  $d \Rightarrow \text{D3 } d$ 
  | D3  $d \Rightarrow \text{D4 } d$ 
  | D4  $d \Rightarrow \text{D5 } d$ 

```

```

| D5  $d \Rightarrow D6\ d$ 
| D6  $d \Rightarrow D7\ d$ 
| D7  $d \Rightarrow D8\ d$ 
| D8  $d \Rightarrow D9\ d$ 
| D9  $d \Rightarrow Da\ d$ 
| Da  $d \Rightarrow Db\ d$ 
| Db  $d \Rightarrow Dc\ d$ 
| Dc  $d \Rightarrow Dd\ d$ 
| Dd  $d \Rightarrow De\ d$ 
| De  $d \Rightarrow Df\ d$ 
| Df  $d \Rightarrow D0\ (\text{succ } d)$ 
end.

```

Doubling little-endian numbers

```

Fixpoint double  $d :=$ 
  match  $d$  with
  | Nil  $\Rightarrow$  Nil
  | D0  $d \Rightarrow D0\ (\text{double } d)$ 
  | D1  $d \Rightarrow D2\ (\text{double } d)$ 
  | D2  $d \Rightarrow D4\ (\text{double } d)$ 
  | D3  $d \Rightarrow D6\ (\text{double } d)$ 
  | D4  $d \Rightarrow D8\ (\text{double } d)$ 
  | D5  $d \Rightarrow Da\ (\text{double } d)$ 
  | D6  $d \Rightarrow Dc\ (\text{double } d)$ 
  | D7  $d \Rightarrow De\ (\text{double } d)$ 
  | D8  $d \Rightarrow D0\ (\text{succ\_double } d)$ 
  | D9  $d \Rightarrow D2\ (\text{succ\_double } d)$ 
  | Da  $d \Rightarrow D4\ (\text{succ\_double } d)$ 
  | Db  $d \Rightarrow D6\ (\text{succ\_double } d)$ 
  | Dc  $d \Rightarrow D8\ (\text{succ\_double } d)$ 
  | Dd  $d \Rightarrow Da\ (\text{succ\_double } d)$ 
  | De  $d \Rightarrow Dc\ (\text{succ\_double } d)$ 
  | Df  $d \Rightarrow De\ (\text{succ\_double } d)$ 
  end

```

```

with succ_double  $d :=$ 
  match  $d$  with
  | Nil  $\Rightarrow D1\ Nil$ 
  | D0  $d \Rightarrow D1\ (\text{double } d)$ 
  | D1  $d \Rightarrow D3\ (\text{double } d)$ 
  | D2  $d \Rightarrow D5\ (\text{double } d)$ 
  | D3  $d \Rightarrow D7\ (\text{double } d)$ 
  | D4  $d \Rightarrow D9\ (\text{double } d)$ 
  | D5  $d \Rightarrow Db\ (\text{double } d)$ 
  | D6  $d \Rightarrow Dd\ (\text{double } d)$ 
  | D7  $d \Rightarrow Df\ (\text{double } d)$ 

```



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

# 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 => 0
  | 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  $\Rightarrow$   $d$ 
| S  $n \Rightarrow$ 
  match  $d$  with
  | Nil  $\Rightarrow$  zero
  | D0  $d$  | D1  $d$  | D2  $d$  | D3  $d$  | D4  $d$  | D5  $d$  | D6  $d$  | D7  $d$  | D8  $d$  | D9  $d \Rightarrow$ 
    del_head  $n$   $d$ 
  end
end.

Definition del_head_int  $n$   $d$  :=
  match  $d$  with
  | Pos  $d \Rightarrow$  del_head  $n$   $d$ 
  | Neg  $d \Rightarrow$  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  $d \Rightarrow$  Pos (del_tail  $n$   $d$ )
  | Neg  $d \Rightarrow$  Neg (del_tail  $n$   $d$ )
  end.

Module LITTLE.

  Successor of little-endian numbers

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

  Doubling little-endian numbers

Fixpoint double  $d$  :=
  match  $d$  with
  | Nil  $\Rightarrow$  Nil
  | D0  $d \Rightarrow$  D0 (double  $d$ )
  | D1  $d \Rightarrow$  D2 (double  $d$ )

```

```

| D2  $d \Rightarrow D4$  (double  $d$ )
| D3  $d \Rightarrow D6$  (double  $d$ )
| D4  $d \Rightarrow D8$  (double  $d$ )
| D5  $d \Rightarrow D0$  (succ_double  $d$ )
| D6  $d \Rightarrow D2$  (succ_double  $d$ )
| D7  $d \Rightarrow D4$  (succ_double  $d$ )
| D8  $d \Rightarrow D6$  (succ_double  $d$ )
| D9  $d \Rightarrow D8$  (succ_double  $d$ )
end

```

```

with succ_double  $d :=$ 
  match  $d$  with
  | Nil  $\Rightarrow D1$  Nil
  | D0  $d \Rightarrow D1$  (double  $d$ )
  | D1  $d \Rightarrow D3$  (double  $d$ )
  | D2  $d \Rightarrow D5$  (double  $d$ )
  | D3  $d \Rightarrow D7$  (double  $d$ )
  | D4  $d \Rightarrow D9$  (double  $d$ )
  | D5  $d \Rightarrow D1$  (succ_double  $d$ )
  | D6  $d \Rightarrow D3$  (succ_double  $d$ )
  | D7  $d \Rightarrow D5$  (succ_double  $d$ )
  | D8  $d \Rightarrow D7$  (succ_double  $d$ )
  | D9  $d \Rightarrow D9$  (succ_double  $d$ )
  end.

```

End LITTLE.

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

Definition uint\_of\_uint ( $i:\mathbf{uint}$ ) :=  $i$ .

Definition int\_of\_int ( $i:\mathbf{int}$ ) :=  $i$ .

# Chapter 18

## Library Coq.Init.Logic

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

### 18.1 Propositional connectives

*True* is the always true proposition

```
Inductive True : Prop :=
  | : True.
```

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

*not A*, written  $\neg A$ , is the negation of *A* Definition not ( $A:\text{Prop}$ ) :=  $A \rightarrow$  **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 ( $A B:\text{Prop}$ ) : Prop :=
  conj :  $A \rightarrow B \rightarrow A \wedge B$ 
```

where "A  $\wedge$  B" := (**and** A B) : *type\_scope*.

Section Conjunction.

Variables A B : Prop.

Theorem proj1 : A  $\wedge$  B  $\rightarrow$  A.

Theorem proj2 : A  $\wedge$  B  $\rightarrow$  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  $\rightarrow$  A  $\vee$  B

| or\_intror : B  $\rightarrow$  A  $\vee$  B

where "A  $\vee$  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  $\rightarrow$  B)  $\wedge$  (B  $\rightarrow$  A).

Notation "A  $\leftrightarrow$  B" := (iff A B) : *type\_scope*.

Section Equivalence.

Theorem iff\_refl :  $\forall A:\text{Prop}, A \leftrightarrow A$ .

Theorem iff\_trans :  $\forall A B C:\text{Prop}, (A \leftrightarrow B) \rightarrow (B \leftrightarrow C) \rightarrow (A \leftrightarrow C)$ .

Theorem iff\_sym :  $\forall A B:\text{Prop}, (A \leftrightarrow B) \rightarrow (B \leftrightarrow A)$ .

End Equivalence.

#[global]

Hint Unfold iff: *extcore*.

Backward direction of the equivalences above does not need assumptions

Theorem and\_iff\_compat\_l :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  (A  $\wedge$  B  $\leftrightarrow$  A  $\wedge$  C).

Theorem and\_iff\_compat\_r :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  (B  $\wedge$  A  $\leftrightarrow$  C  $\wedge$  A).

Theorem or\_iff\_compat\_l :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  (A  $\vee$  B  $\leftrightarrow$  A  $\vee$  C).

Theorem or\_iff\_compat\_r :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  (B  $\vee$  A  $\leftrightarrow$  C  $\vee$  A).

Theorem imp\_iff\_compat\_l :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  ((A  $\rightarrow$  B)  $\leftrightarrow$  (A  $\rightarrow$  C)).

Theorem imp\_iff\_compat\_r :  $\forall A B C : \text{Prop},$

(B  $\leftrightarrow$  C)  $\rightarrow$  ((B  $\rightarrow$  A)  $\leftrightarrow$  (C  $\rightarrow$  A)).

Theorem not\_iff\_compat :  $\forall A B : \text{Prop},$

(A  $\leftrightarrow$  B)  $\rightarrow$  ( $\neg$  A  $\leftrightarrow$   $\neg$  B).



Some equivalences

Theorem `neg_false` :  $\forall A : \text{Prop}, \neg A \leftrightarrow (A \leftrightarrow \text{False})$ .

Theorem `and_cancel_l` :  $\forall A B C : \text{Prop}, (B \rightarrow A) \rightarrow (C \rightarrow A) \rightarrow ((A \wedge B \leftrightarrow A \wedge C) \leftrightarrow (B \leftrightarrow C))$ .

Theorem `and_cancel_r` :  $\forall A B C : \text{Prop}, (B \rightarrow A) \rightarrow (C \rightarrow A) \rightarrow ((B \wedge A \leftrightarrow C \wedge A) \leftrightarrow (B \leftrightarrow C))$ .

Theorem `and_comm` :  $\forall A B : \text{Prop}, A \wedge B \leftrightarrow B \wedge A$ .

Theorem `and_assoc` :  $\forall A B C : \text{Prop}, (A \wedge B) \wedge C \leftrightarrow A \wedge B \wedge C$ .

Theorem `or_cancel_l` :  $\forall A B C : \text{Prop}, (B \rightarrow \neg A) \rightarrow (C \rightarrow \neg A) \rightarrow ((A \vee B \leftrightarrow A \vee C) \leftrightarrow (B \leftrightarrow C))$ .

Theorem `or_cancel_r` :  $\forall A B C : \text{Prop}, (B \rightarrow \neg A) \rightarrow (C \rightarrow \neg A) \rightarrow ((B \vee A \leftrightarrow C \vee A) \leftrightarrow (B \leftrightarrow C))$ .

Theorem `or_comm` :  $\forall A B : \text{Prop}, (A \vee B) \leftrightarrow (B \vee A)$ .

Theorem `or_assoc` :  $\forall A B C : \text{Prop}, (A \vee B) \vee C \leftrightarrow A \vee B \vee C$ .

Lemma `iff_and` :  $\forall A B : \text{Prop}, (A \leftrightarrow B) \rightarrow (A \rightarrow B) \wedge (B \rightarrow A)$ .

Lemma `iff_to_and` :  $\forall A B : \text{Prop}, (A \leftrightarrow B) \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$ .

(*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` ( $P Q R : \text{Prop}$ ) :=  $P \wedge Q \vee \neg P \wedge R$ .

Notation "*IF*  $c1$  **then**  $c2$  **else**  $c3$ " := (`IF_then_else`  $c1 c2 c3$ )  
(at level 200, right associativity) : *type\_scope*.

## 18.2 First-order quantifiers

*ex*  $P$ , or simply  $\exists x, P x$ , or also  $\exists x:A, P x$ , 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  $\forall x:A, Q$ . By symmetry with existential quantification, the construction *all*  $P$  is provided too.

Inductive `ex` ( $A : \text{Type}$ ) ( $P : A \rightarrow \text{Prop}$ ) :  $\text{Prop}$  :=  
`ex_intro` :  $\forall x:A, P x \rightarrow \text{ex } (A:=A) P$ .

Section Projections.

Variables ( $A : \text{Prop}$ ) ( $P : A \rightarrow \text{Prop}$ ).

Definition `ex_proj1` ( $x : \text{ex } P$ ) :  $A$  :=  
`match`  $x$  **with** `ex_intro`  $_ a _ \Rightarrow a$  **end**.

Definition `ex_proj2` ( $x : \text{ex } P$ ) :  $P (\text{ex\_proj1 } x)$  :=  
`match`  $x$  **with** `ex_intro`  $_ _ b \Rightarrow b$  **end**.

End Projections.

Inductive `ex2` ( $A : \text{Type}$ ) ( $P Q : A \rightarrow \text{Prop}$ ) :  $\text{Prop}$  :=

`ex_intro2 :  $\forall x:A, P\ x \rightarrow Q\ x \rightarrow \mathbf{ex2}\ (A:=A)\ P\ Q.$`

**Definition** `all (A:Type) (P:A  $\rightarrow$  Prop) :=  $\forall x:A, P\ x.$`

**Notation** `"'exists' x .. y , p" := ( $\mathbf{ex}\ (\mathbf{fun}\ x \Rightarrow \dots (\mathbf{ex}\ (\mathbf{fun}\ y \Rightarrow p)) \dots)$ )`  
 (at level 200, *x binder*, right associativity,  
*format* `"[' 'exists' '/' 'x .. y , '/' 'p ']"`)  
*: type\_scope.*

**Notation** `"'exists2' x , p & q" := ( $\mathbf{ex2}\ (\mathbf{fun}\ x \Rightarrow p)\ (\mathbf{fun}\ x \Rightarrow q)$ )`  
 (at level 200, *x name*, *p* at level 200, right associativity) *: type\_scope.*

**Notation** `"'exists2' x : A , p & q" := ( $\mathbf{ex2}\ (A:=A)\ (\mathbf{fun}\ x \Rightarrow p)\ (\mathbf{fun}\ x \Rightarrow q)$ )`  
 (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" := ( $\mathbf{ex2}\ (\mathbf{fun}\ x \Rightarrow p)\ (\mathbf{fun}\ x \Rightarrow q)$ )`  
 (at level 200, *x strict pattern*, *p* at level 200, right associativity) *: type\_scope.*

**Notation** `"'exists2' ' x : A , p & q" := ( $\mathbf{ex2}\ (A:=A)\ (\mathbf{fun}\ x \Rightarrow p)\ (\mathbf{fun}\ x \Rightarrow q)$ )`  
 (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  $\rightarrow$  Prop.`

**Theorem** `inst :  $\forall x:A, \mathbf{all}\ (\mathbf{fun}\ x \Rightarrow P\ x) \rightarrow P\ x.$`

**Theorem** `gen :  $\forall (B:\mathbf{Prop})\ (f:\forall y:A, B \rightarrow P\ y), B \rightarrow \mathbf{all}\ P.$`

**End** `universal_quantification.`

## 18.3 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  $\rightarrow$  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" := ( $\neg x = y :>T$ ) : type_scope.`

**Notation** `"x <> y" := ( $\neg (x = y)$ ) : 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 :  $\forall A C:\text{Prop}, A \rightarrow \neg A \rightarrow C$ .

## Section equality.

Variables  $A B : \text{Type}$ .

Variable  $f : A \rightarrow B$ .

Variables  $x y z : A$ .

Theorem eq\_sym :  $x = y \rightarrow y = x$ .

Theorem eq\_trans :  $x = y \rightarrow y = z \rightarrow x = z$ .

Theorem eq\_trans\_r :  $x = y \rightarrow z = y \rightarrow x = z$ .

Theorem f\_equal :  $x = y \rightarrow f x = f y$ .

Theorem not\_eq\_sym :  $x \neq y \rightarrow y \neq x$ .

## End equality.

## Definition eq\_sind\_r :

$\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{SProp}), P x \rightarrow \forall y:A, y = x \rightarrow P y$ .

## Definition eq\_ind\_r :

$\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Prop}), P x \rightarrow \forall y:A, y = x \rightarrow P y$ .

Defined.

## Definition eq\_rec\_r :

$\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Set}), P x \rightarrow \forall y:A, y = x \rightarrow P y$ .

Defined.

## Definition eq\_rect\_r :

$\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Type}), P x \rightarrow \forall y:A, y = x \rightarrow P y$ .

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),

```

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

Theorem f\_equal\_compose  $A\ B\ C\ (a\ b:A)\ (f:A \rightarrow B)\ (g:B \rightarrow C)\ (e:a=b) :$   
 $f\_equal\ g\ (f\_equal\ f\ e) = f\_equal\ (\text{fun } a \Rightarrow g\ (f\ a))\ e.$

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  $A\ B\ (P:B \rightarrow \text{Type})\ (f:A \rightarrow B)\ x1\ x2\ (H:x1=x2)\ (y:P\ (f\ x1)) :$   
 $\text{rew } [\text{fun } x \Rightarrow P\ (f\ x)]\ H\ \text{in } y = \text{rew } f\_equal\ f\ H\ \text{in } y.$

Theorem eq\_trans\_map  $\{A\ B\}\ \{x1\ x2\ x3:A\}\ \{y1:B\ x1\}\ \{y2:B\ x2\}\ \{y3:B\ x3\}$   
 $(H1:x1=x2)\ (H2:x2=x3)\ (H1':\text{rew } H1\ \text{in } y1 = y2)\ (H2':\text{rew } H2\ \text{in } y2 = y3) :$   
 $\text{rew } eq\_trans\ H1\ H2\ \text{in } y1 = y3.$

Lemma map\_subst  $\{A\}\ \{P\ Q:A \rightarrow \text{Type}\}\ (f : \forall\ x,\ P\ x \rightarrow Q\ x)\ \{x\ y\}\ (H:x=y)\ (z:P\ x) :$   
 $\text{rew } H\ \text{in } f\ x\ z = f\ y\ (\text{rew } H\ \text{in } z).$

Lemma map\_subst\_map  $\{A\ B\}\ \{P:A \rightarrow \text{Type}\}\ \{Q:B \rightarrow \text{Type}\}\ (f:A \rightarrow B)\ (g : \forall\ x,\ P\ x \rightarrow Q\ (f\ x))$   
 $\{x\ y\}\ (H:x=y)\ (z:P\ x) :$   
 $\text{rew } f\_equal\ f\ H\ \text{in } g\ x\ z = g\ y\ (\text{rew } H\ \text{in } z).$

Lemma rew\_swap  $A\ (P:A \rightarrow \text{Type})\ x1\ x2\ (H:x1=x2)\ (y1:P\ x1)\ (y2:P\ x2) : \text{rew } H\ \text{in } y1 = y2 \rightarrow y1$   
 $= \text{rew } \leftarrow H\ \text{in } y2.$

Lemma rew\_compose  $A\ (P:A \rightarrow \text{Type})\ x1\ x2\ x3\ (H1:x1=x2)\ (H2:x2=x3)\ (y:P\ x1) :$   
 $\text{rew } H2\ \text{in } \text{rew } H1\ \text{in } y = \text{rew } (eq\_trans\ H1\ H2)\ \text{in } y.$

Extra properties of equality

Theorem eq\_id\_comm\_l  $A\ (f:A \rightarrow A)\ (Hf:\forall\ a,\ a = f\ a)\ a : f\_equal\ f\ (Hf\ a) = Hf\ (f\ a).$

Theorem eq\_id\_comm\_r  $A\ (f:A \rightarrow A)\ (Hf:\forall\ a,\ f\ a = a)\ a : f\_equal\ f\ (Hf\ a) = Hf\ (f\ a).$

Lemma eq\_refl\_map\_distr  $A\ B\ x\ (f:A \rightarrow B) : f\_equal\ f\ (eq\_refl\ x) = eq\_refl\ (f\ x).$

Lemma eq\_trans\_map\_distr  $A\ B\ x\ y\ z\ (f:A \rightarrow B)\ (e:x=y)\ (e':y=z) : f\_equal\ f\ (eq\_trans\ e\ e') = eq\_trans$   
 $(f\_equal\ f\ e)\ (f\_equal\ f\ e').$

Lemma eq\_sym\_map\_distr  $A\ B\ (x\ y:A)\ (f:A \rightarrow B)\ (e:x=y) : eq\_sym\ (f\_equal\ f\ e) = f\_equal\ f\ (eq\_sym\ e).$

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  $A\ (P:A \rightarrow \text{Type})\ (x\ y\ z:A)\ (e:x=y)\ (e':y=z)\ (k:P\ x) :$   
 $\text{rew } (eq\_trans\ e\ e')\ \text{in } k = \text{rew } e'\ \text{in } \text{rew } e\ \text{in } k.$

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`  $(A\ B : \text{Type})\ (R\ R' : A \rightarrow B \rightarrow \text{Prop}) :=$   
 $\forall x\ y, R\ x\ y \rightarrow R'\ x\ y.$

Definition `unique`  $(A : \text{Type})\ (P : A \rightarrow \text{Prop})\ (x:A) :=$   
 $P\ x \wedge \forall (x':A), P\ x' \rightarrow x=x'.$

Definition `uniqueness`  $(A:\text{Type})\ (P:A \rightarrow \text{Prop}) := \forall x\ y, P\ x \rightarrow P\ y \rightarrow x = y.$

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` :  $\forall (A:\text{Type})\ (P:A \rightarrow \text{Prop}),$   
 $((\exists x, P\ x) \wedge \text{uniqueness } P) \leftrightarrow (\exists! x, P\ x).$

Lemma `forall_exists_unique_domain_coincide` :  
 $\forall A\ (P:A \rightarrow \text{Prop}), (\exists! x, P\ x) \rightarrow$   
 $\forall Q:A \rightarrow \text{Prop}, (\forall x, P\ x \rightarrow Q\ x) \leftrightarrow (\exists x, P\ x \wedge Q\ x).$

Lemma `forall_exists_coincide_unique_domain` :  
 $\forall A\ (P:A \rightarrow \text{Prop}),$   
 $(\forall Q:A \rightarrow \text{Prop}, (\forall x, P\ x \rightarrow Q\ x) \leftrightarrow (\exists x, P\ x \wedge Q\ x))$   
 $\rightarrow (\exists! x, P\ x).$

## 18.4 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`  $(A:\text{Type}) : \text{Prop} := \text{inhabits} : A \rightarrow \text{inhabited } A.$

`#[global]`

Hint Resolve *inhabits*: core.

Lemma exists\_inhabited :  $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$   
 $(\exists x, P x) \rightarrow \text{inhabited } A.$

Lemma inhabited\_covariant  $(A B : \text{Type}) : (A \rightarrow B) \rightarrow \text{inhabited } A \rightarrow \text{inhabited } B.$

Declaration of *stepl* and *stepr* for *eq* and *iff*

Lemma eq\_stepl :  $\forall (A : \text{Type}) (x y z : A), x = y \rightarrow x = z \rightarrow z = y.$

Declare Left Step eq\_stepl.

Declare Right Step eq\_trans.

Lemma iff\_stepl :  $\forall A B C : \text{Prop}, (A \leftrightarrow B) \rightarrow (A \leftrightarrow C) \rightarrow (C \leftrightarrow B).$

Declare Left Step iff\_stepl.

Declare Right Step iff\_trans.

Equality for *ex* Section *ex*.

Local Unset Implicit Arguments.

Definition eq\_ex\_uncurried  $\{A : \text{Type}\} (P : A \rightarrow \text{Prop}) \{u1 v1 : A\} \{u2 : P u1\} \{v2 : P v1\}$   
 $(pq : \exists p : u1 = v1, \text{rew } p \text{ in } u2 = v2)$   
 $: \text{ex\_intro } P u1 u2 = \text{ex\_intro } P v1 v2.$

Definition eq\_ex  $\{A : \text{Type}\} \{P : A \rightarrow \text{Prop}\} (u1 v1 : A) (u2 : P u1) (v2 : P v1)$   
 $(p : u1 = v1) (q : \text{rew } p \text{ in } u2 = v2)$   
 $: \text{ex\_intro } P u1 u2 = \text{ex\_intro } P v1 v2$   
 $:= \text{eq\_ex\_uncurried } P (\text{ex\_intro } _ p q).$

Definition eq\_ex\_hprop  $\{A\} \{P : A \rightarrow \text{Prop}\} (P\_hprop : \forall (x : A) (p q : P x), p = q)$   
 $(u1 v1 : A) (u2 : P u1) (v2 : P v1)$   
 $(p : u1 = v1)$   
 $: \text{ex\_intro } P u1 u2 = \text{ex\_intro } P v1 v2$   
 $:= \text{eq\_ex } u1 v1 u2 v2 p (P\_hprop _ _).$

Lemma rew\_ex  $\{A x\} \{P : A \rightarrow \text{Type}\} (Q : \forall a, P a \rightarrow \text{Prop}) (u : \exists p, Q x p) \{y\} (H : x = y)$   
 $: \text{rew } [\text{fun } a \Rightarrow \exists p, Q a p] H \text{ in } u$   
 $= \text{match } u \text{ with}$   
 $\quad | \text{ex\_intro } _ u1 u2$   
 $\quad \Rightarrow \text{ex\_intro}$   
 $\quad \quad (Q y)$   
 $\quad \quad (\text{rew } H \text{ in } u1)$   
 $\quad \quad (\text{rew dependent } H \text{ in } u2)$   
 $\text{end.}$

End *ex*.

Equality for *ex2* Section *ex2*.

Local Unset Implicit Arguments.

Definition eq\_ex2\_uncurried  $\{A : \text{Type}\} (P Q : A \rightarrow \text{Prop}) \{u1 v1 : A\}$   
 $\{u2 : P u1\} \{v2 : P v1\}$   
 $\{u3 : Q u1\} \{v3 : Q v1\}$   
 $(pq : \text{exists2 } p : u1 = v1, \text{rew } p \text{ in } u2 = v2 \ \& \ \text{rew } p \text{ in } u3 = v3)$   
 $: \text{ex\_intro2 } P Q u1 u2 u3 = \text{ex\_intro2 } P Q v1 v2 v3.$



```

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 19

# Library Coq.Init.Numeral

### 19.1 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 := numeral_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 20

# Library Coq.Logic.ChoiceFacts

Some facts and definitions concerning choice and description in intuitionistic logic.

### 20.1 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<sub>int</sub> is equivalent to AC<sub>ext</sub>, Mathematical Logic Quarterly, 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.

### 20.2 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.

#### 20.2.1 Constructive choice and description

AC<sub>rel</sub> = relational form of the (non extensional) axiom of choice (a “set-theoretic” axiom of choice)

Definition RelationalChoice\_on :=

$\forall R:A \rightarrow B \rightarrow \text{Prop},$   
   $(\forall x : A, \exists y : B, R\ x\ y) \rightarrow$

$(\exists R' : A \rightarrow B \rightarrow \text{Prop}, \text{subrelation } R' R \wedge \forall x, \exists! y, R' x y).$

AC\_fun = functional form of the (non extensional) axiom of choice (a “type-theoretic” axiom of choice)

**Definition** FunctionalChoice\_on\_rel  $(R : A \rightarrow B \rightarrow \text{Prop}) :=$

$(\forall x : A, \exists y : B, R x y) \rightarrow$   
 $\exists f : A \rightarrow B, (\forall x : A, R x (f x)).$

**Definition** FunctionalChoice\_on :=

$\forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $(\forall x : A, \exists y : B, R x y) \rightarrow$   
 $(\exists f : A \rightarrow B, \forall x : A, R x (f x)).$

AC\_fun\_dep = functional form of the (non extensional) axiom of choice, with dependent functions **Definition** DependentFunctionalChoice\_on  $(A : \text{Type}) (B : A \rightarrow \text{Type}) :=$

$\forall R : \forall x : A, B x \rightarrow \text{Prop},$   
 $(\forall x : A, \exists y : B x, R x y) \rightarrow$   
 $(\exists f : (\forall x : A, B x), \forall x : A, R x (f x)).$

AC\_trunc = axiom of choice for propositional truncations (truncation and quantification commute) **Definition** InhabitedForallCommute\_on  $(A : \text{Type}) (B : A \rightarrow \text{Type}) :=$

$(\forall x, \text{inhabited } (B x)) \rightarrow \text{inhabited } (\forall x, B x).$

DC\_fun = functional form of the dependent axiom of choice

**Definition** FunctionalDependentChoice\_on :=

$\forall (R : A \rightarrow A \rightarrow \text{Prop}),$   
 $(\forall x, \exists y, R x y) \rightarrow \forall x0,$   
 $(\exists f : \mathbf{nat} \rightarrow A, f 0 = x0 \wedge \forall n, R (f n) (f (\mathbf{S} n))).$

ACw\_fun = functional form of the countable axiom of choice

**Definition** FunctionalCountableChoice\_on :=

$\forall (R : \mathbf{nat} \rightarrow A \rightarrow \text{Prop}),$   
 $(\forall n, \exists y, R n y) \rightarrow$   
 $(\exists f : \mathbf{nat} \rightarrow A, \forall n, R n (f n)).$

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 :=

$\forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $(\forall x : A, \exists! y : B, R x y) \rightarrow$   
 $(\exists f : A \rightarrow B, \forall x : A, R x (f x)).$

AC\_dep! = functional relation reification, with dependent functions see AC! **Definition** DependentFunctionalRelReification\_on  $(A : \text{Type}) (B : A \rightarrow \text{Type}) :=$

$\forall (R : \forall x : A, B x \rightarrow \text{Prop}),$   
 $(\forall x : A, \exists! y : B x, R x y) \rightarrow$   
 $(\exists f : (\forall x : A, B x), \forall x : A, R x (f x)).$

AC\_fun\_repr = functional choice of a representative in an equivalence class

**Definition** RepresentativeFunctionalChoice\_on :=

$$\begin{aligned} &\forall R : A \rightarrow A \rightarrow \text{Prop}, \\ &\quad (\text{Equivalence } R) \rightarrow \\ &\quad (\exists f : A \rightarrow A, \forall x : A, (R x (f x)) \wedge \forall x', R x x' \rightarrow f x = f x'). \end{aligned}$$

AC\_fun\_setoid = functional form of the (so-called extensional) axiom of choice from setoids

**Definition** SetoidFunctionalChoice\_on :=

$$\begin{aligned} &\forall R : A \rightarrow A \rightarrow \text{Prop}, \\ &\forall T : A \rightarrow B \rightarrow \text{Prop}, \\ &\quad \text{Equivalence } R \rightarrow \\ &\quad (\forall x x' y, R x x' \rightarrow T x y \rightarrow T x' y) \rightarrow \\ &\quad (\forall x, \exists y, T x y) \rightarrow \\ &\quad \exists f : A \rightarrow B, \forall x : A, T x (f x) \wedge (\forall x' : A, R x x' \rightarrow f x = f x'). \end{aligned}$$

AC\_fun\_setoid\_gen = functional form of the general form of the (so-called extensional) axiom of choice over setoids

**Definition** GeneralizedSetoidFunctionalChoice\_on :=

$$\begin{aligned} &\forall R : A \rightarrow A \rightarrow \text{Prop}, \\ &\forall S : B \rightarrow B \rightarrow \text{Prop}, \\ &\forall T : A \rightarrow B \rightarrow \text{Prop}, \\ &\quad \text{Equivalence } R \rightarrow \\ &\quad \text{Equivalence } S \rightarrow \\ &\quad (\forall x x' y y', R x x' \rightarrow S y y' \rightarrow T x y \rightarrow T x' y') \rightarrow \\ &\quad (\forall x, \exists y, T x y) \rightarrow \\ &\quad \exists f : A \rightarrow B, \\ &\quad \quad \forall x : A, T x (f x) \wedge (\forall x' : A, R x x' \rightarrow S (f x) (f x')). \end{aligned}$$

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 :=

$$\begin{aligned} &\forall R : A \rightarrow A \rightarrow \text{Prop}, \\ &\forall T : A \rightarrow B \rightarrow \text{Prop}, \\ &\quad \text{Equivalence } R \rightarrow \\ &\quad (\forall x, \exists y, \forall x', R x x' \rightarrow T x' y) \rightarrow \\ &\quad \exists f : A \rightarrow B, \forall x : A, T x (f x) \wedge (\forall x' : A, R x x' \rightarrow f x = f x'). \end{aligned}$$

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 :=

$$\begin{aligned} &\forall P : A \rightarrow \text{Prop}, \\ &\quad (\exists x, P x) \rightarrow \{ x : A \mid P x \}. \end{aligned}$$

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 :=

$$\begin{aligned} &\forall P : A \rightarrow \text{Prop}, \\ &\quad (\exists! x, P x) \rightarrow \{ x : A \mid P x \}. \end{aligned}$$

## 20.2.2 Weakly classical choice and description

GAC\_rel = guarded relational form of the (non extensional) axiom of choice

Definition GuardedRelationalChoice\_on :=

$$\begin{aligned} & \forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop}, \\ & (\forall x : A, P x \rightarrow \exists y : B, R x y) \rightarrow \\ & (\exists R' : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{subrelation } R' R \wedge \forall x, P x \rightarrow \exists! y, R' x y). \end{aligned}$$

GAC\_fun = guarded functional form of the (non extensional) axiom of choice

Definition GuardedFunctionalChoice\_on :=

$$\begin{aligned} & \forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{inhabited } B \rightarrow \\ & (\forall x : A, P x \rightarrow \exists y : B, R x y) \rightarrow \\ & (\exists f : A \rightarrow B, \forall x, P x \rightarrow R x (f x)). \end{aligned}$$

GAC! = guarded functional relation reification

Definition GuardedFunctionalRelReification\_on :=

$$\begin{aligned} & \forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{inhabited } B \rightarrow \\ & (\forall x : A, P x \rightarrow \exists! y : B, R x y) \rightarrow \\ & (\exists f : A \rightarrow B, \forall x : A, P x \rightarrow R x (f x)). \end{aligned}$$

OAC\_rel = “omniscient” relational form of the (non extensional) axiom of choice

Definition OmniscientRelationalChoice\_on :=

$$\begin{aligned} & \forall R : A \rightarrow B \rightarrow \text{Prop}, \\ & \exists R' : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{subrelation } R' R \wedge \forall x : A, (\exists y : B, R x y) \rightarrow \exists! y, R' x y. \end{aligned}$$

OAC\_fun = “omniscient” functional form of the (non extensional) axiom of choice (called AC\* in Bell [Bell])

Definition OmniscientFunctionalChoice\_on :=

$$\begin{aligned} & \forall R : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{inhabited } B \rightarrow \\ & \exists f : A \rightarrow B, \forall x : A, (\exists y : B, R x y) \rightarrow R x (f x). \end{aligned}$$

D\_epsilon = (weakly classical) indefinite description principle

Definition EpsilonStatement\_on :=

$$\begin{aligned} & \forall P : A \rightarrow \text{Prop}, \\ & \text{inhabited } A \rightarrow \{ x : A \mid (\exists x, P x) \rightarrow P x \}. \end{aligned}$$

D\_iota = (weakly classical) definite description principle

Definition IotaStatement\_on :=

$$\begin{aligned} & \forall P : A \rightarrow \text{Prop}, \\ & \text{inhabited } A \rightarrow \{ x : A \mid (\exists! x, P x) \rightarrow P x \}. \end{aligned}$$

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** :=

$\forall (A:\mathbf{Prop}) (a1\ a2:A), a1 = a2.$

IGP = independence of general premises (an unconstrained generalisation of the constructive principle of independence of premises) **Definition IndependenceOfGeneralPremises** :=

$\forall (A:\mathbf{Type}) (P:A \rightarrow \mathbf{Prop}) (Q:\mathbf{Prop}),$   
**inhabited**  $A \rightarrow$   
 $(Q \rightarrow \exists x, P\ x) \rightarrow \exists x, Q \rightarrow P\ x.$

Drinker = drinker's paradox (small form) (called Ex in Bell [Bell]) **Definition SmallDrinker'sParadox** :=

$\forall (A:\mathbf{Type}) (P:A \rightarrow \mathbf{Prop}),$  **inhabited**  $A \rightarrow$   
 $\exists x, (\exists x, P\ x) \rightarrow P\ x.$

EM = excluded-middle **Definition ExcludedMiddle** :=

$\forall P:\mathbf{Prop}, P \vee \neg P.$

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<sub>2</sub> = 2nd-order impredicative minimal predicate logic (with ex. quant.)
- IPL<sup>2</sup> = 2nd-order functional minimal predicate logic (with ex. quant.)
- IPL<sub>2</sub><sup>2</sup> = 2nd-order impredicative, 2nd-order functional minimal pred. logic (with ex. quant.)

with no prerequisite on the non-emptiness of domains

## 20.3 Table of contents

### 1. Definitions

2. IPL<sub>2</sub><sup>2</sup> |- AC<sub>rel</sub> + AC! = AC<sub>fun</sub>

3.1. typed IPL<sub>2</sub> + Sigma-types + PI |- AC<sub>rel</sub> = GAC<sub>rel</sub> and IPL<sub>2</sub> |- AC<sub>rel</sub> + IGP -> GAC<sub>rel</sub> and IPL<sub>2</sub> |- GAC<sub>rel</sub> = OAC<sub>rel</sub>

3.2. IPL<sup>2</sup> |- AC<sub>fun</sub> + IGP = GAC<sub>fun</sub> = OAC<sub>fun</sub> = AC<sub>fun</sub> + Drinker

3.3. D<sub>iota</sub> -> ID<sub>iota</sub> and D<sub>epsilon</sub> <-> ID<sub>epsilon</sub> + Drinker

4. Derivability of choice for decidable relations with well-ordered codomain

5. AC<sub>fun</sub> = AC<sub>fun</sub><sub>dep</sub> = AC<sub>trunc</sub>

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<sub>fun</sub><sub>setoid</sub> = AC<sub>fun</sub> + Ext<sub>fun</sub><sub>repr</sub> + EM

9.2. AC<sub>fun</sub><sub>setoid</sub> = AC<sub>fun</sub> + Ext<sub>pred</sub><sub>repr</sub> + PI

## 20.4 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 :

$\forall A B : \text{Type},$   
 $\text{FunctionalRelReification\_on } A B \rightarrow \text{RelationalChoice\_on } A B \rightarrow \text{FunctionalChoice\_on } A B.$

Lemma fun\_choice\_imp\_rel\_choice :

$\forall A B : \text{Type}, \text{FunctionalChoice\_on } A B \rightarrow \text{RelationalChoice\_on } A B.$

Lemma fun\_choice\_imp\_functional\_rel\_reification :

$\forall A B : \text{Type}, \text{FunctionalChoice\_on } A B \rightarrow \text{FunctionalRelReification\_on } A B.$

Corollary fun\_choice\_iff\_rel\_choice\_and\_functional\_rel\_reification :

$\forall A B : \text{Type}, \text{FunctionalChoice\_on } A B \leftrightarrow$   
 $\text{RelationalChoice\_on } A B \wedge \text{FunctionalRelReification\_on } A B.$

## 20.5 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)

### 20.5.1 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 :

$\text{RelationalChoice} \rightarrow \text{ProofIrrelevance} \rightarrow \text{GuardedRelationalChoice}.$

Lemma rel\_choice\_indep\_of\_general\_premises\_imp\_guarded\_rel\_choice :

$\forall A B : \text{Type}, \text{inhabited } B \rightarrow \text{RelationalChoice\_on } A B \rightarrow$   
 $\text{IndependenceOfGeneralPremises} \rightarrow \text{GuardedRelationalChoice\_on } A B.$

Lemma guarded\_rel\_choice\_imp\_rel\_choice :

$\forall A B : \text{Type}, \text{GuardedRelationalChoice\_on } A B \rightarrow \text{RelationalChoice\_on } A B.$

Lemma subset\_types\_imp\_guarded\_rel\_choice\_iff\_rel\_choice :

$\text{ProofIrrelevance} \rightarrow (\text{GuardedRelationalChoice} \leftrightarrow \text{RelationalChoice}).$

$\text{OAC\_rel} = \text{GAC\_rel}$

Corollary guarded\_iff\_omniscient\_rel\_choice :

$\text{GuardedRelationalChoice} \leftrightarrow \text{OmniscientRelationalChoice}.$

### 20.5.2 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  $\rightarrow$  IndependenceOfGeneralPremises.

Lemma guarded\_fun\_choice\_imp\_fun\_choice :

GuardedFunctionalChoice  $\rightarrow$  FunctionalChoiceOnInhabitedSet.

Lemma fun\_choice\_and\_indep\_general\_prem\_imp\_guarded\_fun\_choice :

FunctionalChoiceOnInhabitedSet  $\rightarrow$  IndependenceOfGeneralPremises  
 $\rightarrow$  GuardedFunctionalChoice.

Corollary fun\_choice\_and\_indep\_general\_prem\_iff\_guarded\_fun\_choice :

FunctionalChoiceOnInhabitedSet  $\wedge$  IndependenceOfGeneralPremises  
 $\leftrightarrow$  GuardedFunctionalChoice.

AC\_fun + Drinker = OAC\_fun

This was already observed by Bell [Bell]

Lemma omniscient\_fun\_choice\_imp\_small\_drinker :

OmniscientFunctionalChoice  $\rightarrow$  SmallDrinker'sParadox.

Lemma omniscient\_fun\_choice\_imp\_fun\_choice :

OmniscientFunctionalChoice  $\rightarrow$  FunctionalChoiceOnInhabitedSet.

Lemma fun\_choice\_and\_small\_drinker\_imp\_omniscient\_fun\_choice :

FunctionalChoiceOnInhabitedSet  $\rightarrow$  SmallDrinker'sParadox  
 $\rightarrow$  OmniscientFunctionalChoice.

Corollary fun\_choice\_and\_small\_drinker\_iff\_omniscient\_fun\_choice :

FunctionalChoiceOnInhabitedSet  $\wedge$  SmallDrinker'sParadox  
 $\leftrightarrow$  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  $\leftrightarrow$  OmniscientFunctionalChoice.

### 20.5.3 D\_iota $\rightarrow$ ID\_iota and D\_epsilon $\leftrightarrow$ ID\_epsilon + Drinker

D\_iota  $\rightarrow$  ID\_iota

Lemma iota\_imp\_constructive\_definite\_description :

IotaStatement  $\rightarrow$  ConstructiveDefiniteDescription.

ID\_epsilon + Drinker  $\leftrightarrow$  D\_epsilon

Lemma epsilon\_imp\_constructive\_indefinite\_description:

EpsilonStatement  $\rightarrow$  ConstructiveIndefiniteDescription.

Lemma constructive\_indefinite\_description\_and\_small\_drinker\_imp\_epsilon :

SmallDrinker'sParadox  $\rightarrow$  ConstructiveIndefiniteDescription  $\rightarrow$   
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).

```

## 20.6 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.

```

## 20.7 AC\_fun = AC\_fun\_dep = AC\_trunc

### 20.7.1 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.

```

## 20.7.2 Functional choice and truncation choice are equivalent

Theorem functional\_choice\_to\_inhabited\_forall\_commute :  
FunctionalChoice  $\rightarrow$  InhabitedForallCommute.

Theorem inhabited\_forall\_commute\_to\_functional\_choice :  
InhabitedForallCommute  $\rightarrow$  FunctionalChoice.

## 20.7.3 Reification of dependent and non dependent functional relation are equivalent

The easy part

Theorem dep\_non\_dep\_functional\_rel\_reification :  
DependentFunctionalRelReification  $\rightarrow$  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  $\rightarrow$  DependentFunctionalRelReification.

Corollary dep\_iff\_non\_dep\_functional\_rel\_reification :  
FunctionalRelReification  $\leftrightarrow$  DependentFunctionalRelReification.

## 20.8 Non contradiction of constructive descriptions wrt functional axioms of choice

### 20.8.1 Non contradiction of indefinite description

Lemma relative\_non\_contradiction\_of\_indefinite\_descr :  
 $\forall C:\text{Prop}, (\text{ConstructiveIndefiniteDescription} \rightarrow C)$   
 $\rightarrow (\text{FunctionalChoice} \rightarrow C).$

Lemma constructive\_indefinite\_descr\_fun\_choice :  
ConstructiveIndefiniteDescription  $\rightarrow$  FunctionalChoice.

### 20.8.2 Non contradiction of definite description

Lemma relative\_non\_contradiction\_of\_definite\_descr :  
 $\forall C:\text{Prop}, (\text{ConstructiveDefiniteDescription} \rightarrow C)$   
 $\rightarrow (\text{FunctionalRelReification} \rightarrow C).$

Lemma constructive\_definite\_descr\_fun\_reification :  
ConstructiveDefiniteDescription  $\rightarrow$  FunctionalRelReification.

Remark, the following corollaries morally hold:

Definition In\_propositional\_context (A:Type) := forall C:Prop, (A  $\rightarrow$  C)  $\rightarrow$  C.

Corollary constructive\_definite\_descr\_in\_prop\_context\_iff\_fun\_reification : In\_propositional\_context  
ConstructiveIndefiniteDescription  $\leftrightarrow$  FunctionalChoice.

Corollary `constructive_definite_descr_in_prop_context_iff_fun_reification` : `In_propositional_context`  
`ConstructiveDefiniteDescription`  $\leftrightarrow$  `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.

## 20.9 Excluded-middle + definite description $\Rightarrow$ 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` :

( $\forall A : \text{Type}, \text{ConstructiveDefiniteDescription\_on } A$ )  $\rightarrow$   
 $(\forall P : \text{Prop}, P \vee \neg P) \rightarrow (\forall P : \text{Prop}, \{P\} + \{\neg P\})$ .

Corollary `fun_reification_descr_computational_excluded_middle_in_prop_context` :

`FunctionalRelReification`  $\rightarrow$   
 $(\forall P : \text{Prop}, P \vee \neg P) \rightarrow$   
 $\forall C : \text{Prop}, ((\forall P : \text{Prop}, \{P\} + \{\neg P\}) \rightarrow C) \rightarrow C$ .

## 20.10 Choice $\Rightarrow$ Dependent choice $\Rightarrow$ Countable choice

Require Import **Arith**.

Theorem `functional_choice_imp_functional_dependent_choice` :

`FunctionalChoice`  $\rightarrow$  `FunctionalDependentChoice`.

Theorem `functional_dependent_choice_imp_functional_countable_choice` :

`FunctionalDependentChoice`  $\rightarrow$  `FunctionalCountableChoice`.

## 20.11 About the axiom of choice over setoids

Require Import **ClassicalFacts** **PropExtensionalityFacts**.

### 20.11.1 Consequences of the choice of a representative in an equivalence class

Theorem `repr_fun_choice_imp_ext_prop_repr` :

`RepresentativeFunctionalChoice`  $\rightarrow$  `ExtensionalPropositionRepresentative`.

Theorem repr\_fun\_choice\_imp\_ext\_pred\_repr :  
 RepresentativeFunctionalChoice  $\rightarrow$  ExtensionalPredicateRepresentative.

Theorem repr\_fun\_choice\_imp\_ext\_function\_repr :  
 RepresentativeFunctionalChoice  $\rightarrow$  ExtensionalFunctionRepresentative.

**This is a variant of Diaconescu and Goodman-Myhill theorems**

Theorem repr\_fun\_choice\_imp\_excluded\_middle :  
 RepresentativeFunctionalChoice  $\rightarrow$  ExcludedMiddle.

Theorem repr\_fun\_choice\_imp\_relational\_choice :  
 RepresentativeFunctionalChoice  $\rightarrow$  RelationalChoice.

### 20.11.2 AC\_fun\_setoid = AC\_fun\_setoid\_gen = AC\_fun\_setoid\_simple

Theorem gen\_setoid\_fun\_choice\_imp\_setoid\_fun\_choice :  
 $\forall A B$ , GeneralizedSetoidFunctionalChoice\_on  $A B \rightarrow$  SetoidFunctionalChoice\_on  $A B$ .

Theorem setoid\_fun\_choice\_imp\_gen\_setoid\_fun\_choice :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \rightarrow$  GeneralizedSetoidFunctionalChoice\_on  $A B$ .

Corollary setoid\_fun\_choice\_iff\_gen\_setoid\_fun\_choice :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \leftrightarrow$  GeneralizedSetoidFunctionalChoice\_on  $A B$ .

Theorem setoid\_fun\_choice\_imp\_simple\_setoid\_fun\_choice :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \rightarrow$  SimpleSetoidFunctionalChoice\_on  $A B$ .

Theorem simple\_setoid\_fun\_choice\_imp\_setoid\_fun\_choice :  
 $\forall A B$ , SimpleSetoidFunctionalChoice\_on  $A B \rightarrow$  SetoidFunctionalChoice\_on  $A B$ .

Corollary setoid\_fun\_choice\_iff\_simple\_setoid\_fun\_choice :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \leftrightarrow$  SimpleSetoidFunctionalChoice\_on  $A B$ .

### 20.11.3 AC\_fun\_setoid = AC! + AC\_fun\_repr

Theorem setoid\_fun\_choice\_imp\_fun\_choice :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \rightarrow$  FunctionalChoice\_on  $A B$ .

Corollary setoid\_fun\_choice\_imp\_functional\_rel\_reification :  
 $\forall A B$ , SetoidFunctionalChoice\_on  $A B \rightarrow$  FunctionalRelReification\_on  $A B$ .

Theorem setoid\_fun\_choice\_imp\_repr\_fun\_choice :  
 SetoidFunctionalChoice  $\rightarrow$  RepresentativeFunctionalChoice .

Theorem functional\_rel\_reification\_and\_repr\_fun\_choice\_imp\_setoid\_fun\_choice :  
 FunctionalRelReification  $\rightarrow$  RepresentativeFunctionalChoice  $\rightarrow$  SetoidFunctionalChoice.

Theorem functional\_rel\_reification\_and\_repr\_fun\_choice\_iff\_setoid\_fun\_choice :  
 FunctionalRelReification  $\wedge$  RepresentativeFunctionalChoice  $\leftrightarrow$  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

## 20.12 AC\_fun\_setoid = AC\_fun + Ext\_fun\_repr + EM

Import *EqNotations*.

### 20.12.1 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.

### 20.12.2 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.

## 20.13 Compatibility notations

Notation description\_rel\_choice\_imp\_func\_choice :=

functional\_rel\_reification\_and\_rel\_choice\_imp\_fun\_choice (*only parsing*).

Notation func\_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 func\_choice\_imp\_description := fun\_choice\_imp\_functional\_rel\_reification (*only parsing*).



## Chapter 21

# Library Coq.Logic.ProofIrrelevance

This file axiomatizes proof-irrelevance and derives some consequences

```
Require Import ProofIrrelevanceFacts.
```

```
Axiom proof_irrelevance :  $\forall (P:\text{Prop}) (p1\ p2:P), p1 = p2$ .
```

```
Module PI. Definition proof_irrelevance := proof_irrelevance. End PI.
```

```
Module PROOFIRRELEVANCETHEORY := PROOFIRRELEVANCETHEORY(PI).
```

```
Export ProofIrrelevanceTheory.
```

## Chapter 22

# 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 23

# Library Coq.Logic.Decidable

Properties of decidable propositions

Definition decidable ( $P:\text{Prop}$ ) :=  $P \vee \neg P$ .

Theorem dec\_not\_not :  $\forall P:\text{Prop}$ , decidable  $P \rightarrow (\neg P \rightarrow \mathbf{False}) \rightarrow P$ .

Theorem dec\_True : decidable **True**.

Theorem dec\_False : decidable **False**.

Theorem dec\_or :

$\forall A B:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $B \rightarrow$  decidable  $(A \vee B)$ .

Theorem dec\_and :

$\forall A B:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $B \rightarrow$  decidable  $(A \wedge B)$ .

Theorem dec\_not :  $\forall A:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $(\neg A)$ .

Theorem dec\_imp :

$\forall A B:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $B \rightarrow$  decidable  $(A \rightarrow B)$ .

Theorem dec\_iff :

$\forall A B:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $B \rightarrow$  decidable  $(A \leftrightarrow B)$ .

Theorem not\_not :  $\forall P:\text{Prop}$ , decidable  $P \rightarrow \neg \neg P \rightarrow P$ .

Theorem not\_or :  $\forall A B:\text{Prop}$ ,  $\neg (A \vee B) \rightarrow \neg A \wedge \neg B$ .

Theorem not\_and :  $\forall A B:\text{Prop}$ , decidable  $A \rightarrow \neg (A \wedge B) \rightarrow \neg A \vee \neg B$ .

Theorem not\_imp :  $\forall A B:\text{Prop}$ , decidable  $A \rightarrow \neg (A \rightarrow B) \rightarrow A \wedge \neg B$ .

Theorem imp\_simp :  $\forall A B:\text{Prop}$ , decidable  $A \rightarrow (A \rightarrow B) \rightarrow \neg A \vee B$ .

Theorem not\_iff :

$\forall A B:\text{Prop}$ , decidable  $A \rightarrow$  decidable  $B \rightarrow$   
 $\neg (A \leftrightarrow B) \rightarrow (A \wedge \neg B) \vee (\neg A \wedge B)$ .

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 :  $(\mathbf{True} \rightarrow \mathbf{False}) \leftrightarrow \mathbf{False}$ .

Theorem not\_false\_iff : (**False** → **False**) ↔ **True**.

Theorem not\_not\_iff : ∀ A:Prop, decidable A →  
(((A → **False**) → **False**) ↔ A).

Theorem contrapositive : ∀ A B:Prop, decidable A →  
(((A → **False**) → (B → **False**)) ↔ (B → A)).

Lemma or\_not\_l\_iff\_1 : ∀ A B: Prop, decidable A →  
((A → **False**) ∨ B ↔ (A → B)).

Lemma or\_not\_l\_iff\_2 : ∀ A B: Prop, decidable B →  
((A → **False**) ∨ B ↔ (A → B)).

Lemma or\_not\_r\_iff\_1 : ∀ A B: Prop, decidable A →  
(A ∨ (B → **False**) ↔ (B → A)).

Lemma or\_not\_r\_iff\_2 : ∀ A B: Prop, decidable B →  
(A ∨ (B → **False**) ↔ (B → A)).

Lemma imp\_not\_l : ∀ A B: Prop, decidable A →  
(((A → **False**) → B) ↔ (A ∨ 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 : ∀ A B:Prop,  
(A ∨ B → **False**) ↔ (A → **False**) ∧ (B → **False**).

Lemma not\_and\_iff : ∀ A B:Prop,  
(A ∧ B → **False**) ↔ (A → B → **False**).

Lemma not\_imp\_iff : ∀ A B:Prop, decidable A →  
(((A → B) → **False**) ↔ A ∧ (B → **False**)).

Lemma not\_imp\_rev\_iff : ∀ A B : Prop, decidable A →  
(((A → B) → **False**) ↔ (B → **False**) ∧ A).

Theorem dec\_functional\_relation :  
∀ (X Y : Type) (A:X→Y→Prop), (∀ y y' : Y, decidable (y=y')) →  
(∀ x, ∃! y, A x y) → ∀ 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 24

## 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

1. Definitions

2.1 Predicate extensionality  $\leftrightarrow$  Proposition extensionality + Propositional functional extensionality

2.2 Propositional extensionality  $\rightarrow$  Provable propositional extensionality

2.3 Propositional extensionality  $\rightarrow$  Refutable propositional extensionality

`Set Implicit Arguments.`

### 24.1 Definitions

Propositional extensionality

Provable-proposition extensionality

Refutable-proposition extensionality

Predicate extensionality

Propositional functional extensionality

## 24.2 Propositional and predicate extensionality

### 24.2.1 Predicate extensionality $\leftrightarrow$ Propositional extensionality + Propositional functional extensionality

Lemma `PredExt_imp_PropExt` : `PredicateExtensionality`  $\rightarrow$  `PropositionalExtensionality`.

Lemma `PredExt_imp_PropFunExt` : `PredicateExtensionality`  $\rightarrow$  `PropositionalFunctionalExtensionality`.

Lemma `PropExt_and_PropFunExt_imp_PredExt` :

`PropositionalExtensionality`  $\rightarrow$  `PropositionalFunctionalExtensionality`  $\rightarrow$  `PredicateExtensionality`.

Theorem `PropExt_and_PropFunExt_iff_PredExt` :

`PropositionalExtensionality`  $\wedge$  `PropositionalFunctionalExtensionality`  $\leftrightarrow$  `PredicateExtensionality`.

### 24.2.2 Propositional extensionality and provable proposition extensionality

Lemma `PropExt_imp_ProvPropExt` : `PropositionalExtensionality`  $\rightarrow$  `ProvablePropositionExtensionality`.

### 24.2.3 Propositional extensionality and refutable proposition extensionality

Lemma `PropExt_imp_RefutPropExt` : `PropositionalExtensionality`  $\rightarrow$  `RefutablePropositionExtensionality`.

## Chapter 25

# Library Coq.Logic.FinFun

### 25.1 Functions on finite domains

Main result : for functions  $f:A \rightarrow B$  with finite  $A$ ,  $f$  injective  $\leftrightarrow$   $f$  bijective  $\leftrightarrow$   $f$  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  $A B (f:A \rightarrow B)$ :

Surjective  $f \leftrightarrow (\forall lB, \exists lA, \text{incl } lB (\text{map } f lA))$ .

Lemma Surjective\_carac  $A B : \text{Finite } B \rightarrow \text{decidable\_eq } B \rightarrow$

$\forall f:A \rightarrow B, \text{Surjective } f \leftrightarrow (\exists lA, \text{Listing } (\text{map } f lA))$ .

Main result :

Lemma Endo\_Injective\_Surjective :

$\forall A, \text{Finite } A \rightarrow \text{decidable\_eq } A \rightarrow$

$\forall f:A \rightarrow A, \text{Injective } f \leftrightarrow \text{Surjective } f$ .

An injective and surjective function is bijective. We need here stronger hypothesis : decidability of equality in Type.

Definition EqDec  $(A:\text{Type}) := \forall x y:A, \{x=y\} + \{x \neq y\}$ .

First, we show that a surjective  $f$  has an inverse function  $g$  such that  $f.g = \text{id}$ .

Lemma Finite\_Empty\_or\_not  $A :$

$\text{Finite } A \rightarrow (A \rightarrow \text{False}) \vee \exists a:A, \text{True}$ .

Lemma Surjective\_inverse :

$\forall A B, \text{Finite } A \rightarrow \text{EqDec } B \rightarrow$

$\forall f:A \rightarrow B, \text{Surjective } f \rightarrow$

$\exists g:B \rightarrow A, \forall x, f (g x) = x$ .

Same, with more knowledge on the inverse function:  $g.f = f.g = \text{id}$

Lemma Injective\_Surjective\_Bijective :

$\forall A B, \text{Finite } A \rightarrow \text{EqDec } B \rightarrow$

$\forall f:A \rightarrow B, \text{Injective } f \rightarrow \text{Surjective } f \rightarrow \text{Bijective } f$ .

An example of finite type :  $\text{Fin.t}$

Lemma Fin\_Finite  $n : \text{Finite } (\text{Fin.t } n)$ .

Instead of working on a finite subset of  $\text{nat}$ , another solution is to use restricted  $\text{nat} \rightarrow \text{nat}$  functions, and to consider them only below a certain bound  $n$ .

Definition bFun  $n (f:\text{nat} \rightarrow \text{nat}) := \forall x, x < n \rightarrow f x < n$ .

Definition blnjective  $n (f:\text{nat} \rightarrow \text{nat}) :=$

$\forall x y, x < n \rightarrow y < n \rightarrow f x = f y \rightarrow x = y$ .

Definition bSurjective  $n (f:\text{nat} \rightarrow \text{nat}) :=$

$\forall y, y < n \rightarrow \exists x, x < n \wedge f x = y$ .

We show that this is equivalent to the use of  $\text{Fin.t } n$ .

Module FIN2RESTRICT.

Notation  $\text{n2f} := \text{Fin.of\_nat\_lt}$ .

Definition  $\text{f2n } \{n\} (x:\text{Fin.t } n) := \text{proj1\_sig } (\text{Fin.to\_nat } x)$ .

Definition  $\text{f2n\_ok } n (x:\text{Fin.t } n) : \text{f2n } x < n := \text{proj2\_sig } (\text{Fin.to\_nat } x)$ .

Definition  $\text{n2f\_f2n} : \forall n x, \text{n2f } (\text{f2n\_ok } x) = x := @\text{Fin.of\_nat\_to\_nat\_inv}$ .

Definition  $\text{f2n\_n2f } x n h : \text{f2n } (\text{n2f } h) = x := \text{f\_equal } (@\text{proj1\_sig } -) (@\text{Fin.to\_nat\_of\_nat } x n h)$ .

```

Definition n2f_ext :  $\forall x n h h', n2f\ h = n2f\ h' := @Fin.of\_nat\_ext.$ 
Definition f2n_inj :  $\forall n x y, f2n\ x = f2n\ y \rightarrow x = y := @Fin.to\_nat\_inj.$ 

Definition extend  $n\ (f : Fin.t\ n \rightarrow Fin.t\ n) : (nat \rightarrow nat) :=$ 
  fun  $x \Rightarrow$ 
    match le_lt_dec  $n\ x$  with
    | left  $_ \Rightarrow 0$ 
    | right  $h \Rightarrow f2n\ (f\ (n2f\ h))$ 
  end.

Definition restrict  $n\ (f : nat \rightarrow nat)(hf : bFun\ n\ f) : (Fin.t\ n \rightarrow Fin.t\ n) :=$ 
  fun  $x \Rightarrow$  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  $n\ f :$ 
   $bSurjective\ (@extend\ n\ f) \leftrightarrow Surjective\ f.$ 
Lemma extend_injective  $n\ f :$ 
   $bInjective\ (@extend\ n\ f) \leftrightarrow Injective\ f.$ 
Lemma restrict_surjective  $n\ f\ h :$ 
   $Surjective\ (@restrict\ n\ f\ h) \leftrightarrow bSurjective\ n\ f.$ 
Lemma restrict_injective  $n\ f\ h :$ 
   $Injective\ (@restrict\ n\ f\ h) \leftrightarrow bInjective\ n\ f.$ 
End FIN2RESTRICT.
Import Fin2Restrict.

```

We can now use Proof via the equivalence ...

```

Lemma bInjective_bSurjective  $n\ (f : nat \rightarrow nat) :$ 
   $bFun\ n\ f \rightarrow (bInjective\ n\ f \leftrightarrow bSurjective\ n\ f).$ 

Lemma bSurjective_bBijective  $n\ (f : nat \rightarrow nat) :$ 
   $bFun\ n\ f \rightarrow bSurjective\ n\ f \rightarrow$ 
   $\exists g, bFun\ n\ g \wedge \forall x, x < n \rightarrow g\ (f\ x) = x \wedge f\ (g\ x) = x.$ 

```

## Chapter 26

# 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  $\exists n, P\ n$  in order to show  $\{n : \text{nat} \mid P\ n\}$ . 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* :  $\forall n, \{P\ n\} + \{\sim(P\ n)\}$ .

The termination argument is *before\_witness* *n*, which says that any number before any witness (not necessarily the *x* of  $\exists x : A, P\ x$ ) 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 than *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  $R (x y : nat) : Prop := x = S y \wedge \neg P y$ .

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 :

$(\exists n : \mathbf{nat}, P\ n) \rightarrow \{n : \mathbf{nat} \mid P\ n\}.$

End ConstructiveIndefiniteGroundDescription\_Acc.

Section ConstructiveGroundEpsilon\_nat.

Variable  $P : \mathbf{nat} \rightarrow \text{Prop}.$

Hypothesis  $P\_decidable : \forall x : \mathbf{nat}, \{P\ x\} + \{\neg P\ x\}.$

Definition constructive\_ground\_epsilon\_nat  $(E : \exists n : \mathbf{nat}, P\ n) : \mathbf{nat}$

$:= \text{proj1\_sig } (\text{constructive\_indefinite\_ground\_description\_nat } P\ P\_decidable\ E).$

Definition constructive\_ground\_epsilon\_spec\_nat  $(E : (\exists n, P\ n)) : P\ (\text{constructive\_ground\_epsilon\_nat } E)$

$:= \text{proj2\_sig } (\text{constructive\_indefinite\_ground\_description\_nat } P\ P\_decidable\ E).$

End ConstructiveGroundEpsilon\_nat.

Section ConstructiveGroundEpsilon.

For the current purpose, we say that a set  $A$  is countable if there are functions  $f : A \rightarrow \mathbf{nat}$  and  $g : \mathbf{nat} \rightarrow A$  such that  $g$  is a left inverse of  $f$ .

Variable  $A : \text{Type}.$

Variable  $f : A \rightarrow \mathbf{nat}.$

Variable  $g : \mathbf{nat} \rightarrow A.$

Hypothesis  $gof\_eq\_id : \forall x : A, g\ (f\ x) = x.$

Variable  $P : A \rightarrow \text{Prop}.$

Hypothesis  $P\_decidable : \forall x : A, \{P\ x\} + \{\neg P\ x\}.$

Definition  $P' (x : \mathbf{nat}) : \text{Prop} := P\ (g\ x).$

Lemma  $P'\_decidable : \forall n : \mathbf{nat}, \{P'\ n\} + \{\neg P'\ n\}.$

Lemma constructive\_indefinite\_ground\_description :  $(\exists x : A, P\ x) \rightarrow \{x : A \mid P\ x\}.$

Lemma constructive\_definite\_ground\_description :  $(\exists! x : A, P\ x) \rightarrow \{x : A \mid P\ x\}.$

Definition constructive\_ground\_epsilon  $(E : \exists x : A, P\ x) : A$

$:= \text{proj1\_sig } (\text{constructive\_indefinite\_ground\_description } E).$

Definition constructive\_ground\_epsilon\_spec  $(E : (\exists x, P\ x)) : P\ (\text{constructive\_ground\_epsilon } E)$

$:= \text{proj2\_sig } (\text{constructive\_indefinite\_ground\_description } E).$

End ConstructiveGroundEpsilon.

## Chapter 27

# Library Coq.Logic.PropFacts

### 27.1 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`

$(f : \mathbf{Prop} \rightarrow \mathbf{Prop})$

$(inj : \forall A B, (f A \leftrightarrow f B) \rightarrow (A \leftrightarrow B))$

$(ext : \forall A B, A \leftrightarrow B \rightarrow f A \leftrightarrow f B)$

$: \forall A, f (f A) \leftrightarrow A.$

## Chapter 28

### Library

## Coq.Logic.ProofIrrelevanceFacts

This defines the functor that build consequences of proof-irrelevance

Require Export EqdepFacts.

Module Type PROOFIRRELEVANCE.

Axiom *proof\_irrelevance* :  $\forall (P:\text{Prop}) (p1\ p2:P), p1 = p2$ .

End PROOFIRRELEVANCE.

Module PROOFIRRELEVANCETHEORY (M:PROOFIRRELEVANCE).

Proof-irrelevance implies uniqueness of reflexivity proofs

Module EQ\_RECT\_EQ.

Lemma eq\_rect\_eq :

$\forall (U:\text{Type}) (p:U) (Q:U \rightarrow \text{Type}) (x:Q\ p) (h:p = p),$   
 $x = \text{eq\_rect}\ p\ Q\ x\ p\ h.$

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 :

$\forall (U:\text{Type}) (P:U \rightarrow \text{Prop}) (x\ y:U) (p:P\ x) (q:P\ y),$   
 $x = y \rightarrow \text{exist}\ P\ x\ p = \text{exist}\ P\ y\ q.$

Lemma subsetT\_eq\_compat :

$\forall (U:\text{Type}) (P:U \rightarrow \text{Prop}) (x\ y:U) (p:P\ x) (q:P\ y),$   
 $x = y \rightarrow \text{existT}\ P\ x\ p = \text{existT}\ P\ y\ q.$

End PROOFIRRELEVANCETHEORY.

## Chapter 29

# 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<sub>int</sub> is equivalent to AC<sub>ext</sub>, Mathematical Logic Quarterly, 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 :

$$\begin{aligned} & \forall A B, \\ & \forall R : A \rightarrow A \rightarrow \text{Prop}, \\ & \forall T : A \rightarrow B \rightarrow \text{Prop}, \\ & \text{Equivalence } R \rightarrow \\ & (\forall x x' y, R x x' \rightarrow T x y \rightarrow T x' y) \rightarrow \\ & (\forall x, \exists y, T x y) \rightarrow \end{aligned}$$



$$\exists f : A \rightarrow B, \forall x : A, T\ x\ (f\ x) \wedge (\forall x' : A, R\ x\ x' \rightarrow f\ x = f\ x').$$

Theorem representative\_choice :

$$\forall A\ (R:A \rightarrow A \rightarrow \text{Prop}),\ (\text{Equivalence}\ R) \rightarrow$$

$$\exists f : A \rightarrow A, \forall x : A, R\ x\ (f\ x) \wedge \forall x', R\ x\ x' \rightarrow f\ x = f\ x'.$$

## Chapter 30

### Library

# Coq.Logic.ExtensionalFunctionRepresentative

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 (intensional) choice

It provides extensionality of functions while still supporting (a priori) an intensional interpretation of equality

**Axiom** *extensional\_function\_representative* :

$$\begin{aligned} &\forall A B, \exists repr, \forall (f : A \rightarrow B), \\ &(\forall x, f x = repr f x) \wedge \\ &(\forall g, (\forall x, f x = g x) \rightarrow repr f = repr g). \end{aligned}$$

## Chapter 31

# 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

### 31.1 Streicher's K and injectivity of dependent pair hold on decidable types

Set Implicit Arguments.

Section EqdepDec.

Variable  $A : \text{Type}$ .

Let  $\text{comp } (x \ y \ y' : A) \ (eq1 : x = y) \ (eq2 : x = y') : y = y' :=$   
   $\text{eq\_ind } \_ \ (\text{fun } a \Rightarrow a = y') \ eq2 \ \_ \ eq1$ .

Remark  $\text{trans\_sym\_eq } (x \ y : A) \ (u : x = y) : \text{comp } u \ u = \text{eq\_refl } y$ .

Variable  $x : A$ .

Variable  $\text{eq\_dec} : \forall y : A, x = y \vee x \neq y$ .

Let  $\text{nu } (y : A) \ (u : x = y) : x = y :=$   
  match  $\text{eq\_dec } y$  with  
  |  $\text{or\_introl } eqxy \Rightarrow eqxy$   
  |  $\text{or\_intror } neqxy \Rightarrow \text{False\_ind } \_ \ (neqxy \ u)$   
  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  $K\_dec\_on (P:x = x \rightarrow Prop) (H:P\ (eq\_refl\ x)) (p:x = x) : P\ p$ .

The corollary

Let  $proj (P:A \rightarrow Prop) (exP:ex\ P) (def:P\ x) : P\ x :=$

```

  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  $inj\_right\_pair\_on (P:A \rightarrow Prop) (y\ y':P\ x) :$

$ex\_intro\ P\ x\ y = ex\_intro\ P\ x\ y' \rightarrow y = y'$ .

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  $eq\_proofs\_unicity\ A\ (eq\_dec : \forall x\ y : A, x = y \vee x \neq y) (x : A)$   
 $: \forall (y:A) (p1\ p2:x = y), p1 = p2$ .

Theorem  $K\_dec\ A\ (eq\_dec : \forall x\ y : A, x = y \vee x \neq y) (x : A)$   
 $: \forall P:x = x \rightarrow Prop, P\ (eq\_refl\ x) \rightarrow \forall p:x = x, P\ p$ .

Theorem  $inj\_right\_pair\ A\ (eq\_dec : \forall x\ y : A, x = y \vee x \neq y) (x : A)$   
 $: \forall (P:A \rightarrow Prop) (y\ y':P\ x),$   
 $ex\_intro\ P\ x\ y = ex\_intro\ P\ x\ y' \rightarrow y = y'$ .

Require Import EqdepFacts.

We deduce axiom  $K$  for (decidable) types Theorem  $K\_dec\_type (A:Type) (eq\_dec:\forall x\ y:A, \{x = y\} + \{x \neq y\}) (x:A)$   
 $(P:x = x \rightarrow Prop) (H:P\ (eq\_refl\ x)) (p:x = x) : P\ p$ .

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  $eq\_rect\_eq\_dec :$

$\forall A:\text{Type},$   
 $(\forall x y:A, \{x = y\} + \{x \neq y\}) \rightarrow$   
 $\forall (p:A) (Q:A \rightarrow \text{Type}) (x:Q p) (h:p = p), x = \text{eq\_rect } p \ Q \ x \ p \ h.$

We deduce the injectivity of dependent equality for decidable types **Theorem eq\_dep\_eq\_dec :**

$\forall A:\text{Type},$   
 $(\forall x y:A, \{x = y\} + \{x \neq y\}) \rightarrow$   
 $\forall (P:A \rightarrow \text{Type}) (p:A) (x y:P p), \text{eq\_dep } A \ P \ p \ x \ p \ y \rightarrow x = y.$

**Theorem UIP\_dec :**

$\forall (A:\text{Type}),$   
 $(\forall x y:A, \{x = y\} + \{x \neq y\}) \rightarrow$   
 $\forall (x y:A) (p1 \ p2:x = y), p1 = p2.$

Unset Implicit Arguments.

### 31.1.1 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* :  $\forall x y:U, \{x = y\} + \{x \neq y\}.$

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* :

$\forall (p:U) (Q:U \rightarrow \text{Type}) (x:Q p) (h:p = p), x = \text{eq\_rect } p \ Q \ x \ p \ h.$

Injectivity of Dependent Equality

**Theorem eq\_dep\_eq :**

$\forall (P:U \rightarrow \text{Type}) (p:U) (x y:P p), \text{eq\_dep } U \ P \ p \ x \ p \ y \rightarrow x = y.$

Uniqueness of Identity Proofs (UIP)

Lemma *UIP* :  $\forall (x y:U) (p1 \ p2:x = y), p1 = p2.$

Uniqueness of Reflexive Identity Proofs

Lemma *UIP\_refl* :  $\forall (x:U) (p:x = x), p = \text{eq\_refl } x.$

Streicher's axiom K

Lemma *Streicher\_K* :

$\forall (x:U) (P:x = x \rightarrow \text{Prop}), P (\text{eq\_refl } x) \rightarrow \forall p:x = x, P p.$

Injectivity of equality on dependent pairs in Type

Lemma *inj\_pairT2* :

$\forall (P:U \rightarrow \text{Type}) (p:U) (x y:P p),$

$\text{existT } P \ p \ x = \text{existT } P \ p \ y \rightarrow x = y.$

Proof-irrelevance on subsets of decidable sets

Lemma `inj_pairP2` :

$\forall (P:U \rightarrow \text{Prop}) (x:U) (p \ q:P \ x),$   
 $\text{ex\_intro } P \ x \ p = \text{ex\_intro } P \ x \ q \rightarrow p = q.$

End `DECIDABLEEQDEP`.

### 31.1.2 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` :  $\forall x \ y:U, \{x = y\} + \{x \neq y\}.$

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` :

$\forall (p:U) (Q:U \rightarrow \text{Type}) (x:Q \ p) (h:p = p), x = \text{eq\_rect } p \ Q \ x \ p \ h.$

Injectivity of Dependent Equality

Theorem `eq_dep_eq` :

$\forall (P:U \rightarrow \text{Type}) (p:U) (x \ y:P \ p), \text{eq\_dep } U \ P \ p \ x \ p \ y \rightarrow x = y.$

Uniqueness of Identity Proofs (UIP)

Lemma `UIP` :  $\forall (x \ y:U) (p1 \ p2:x = y), p1 = p2.$

Uniqueness of Reflexive Identity Proofs

Lemma `UIP_refl` :  $\forall (x:U) (p:x = x), p = \text{eq\_refl } x.$

Streicher's axiom K

Lemma `Streicher_K` :

$\forall (x:U) (P:x = x \rightarrow \text{Prop}), P (\text{eq\_refl } x) \rightarrow \forall p:x = x, P \ p.$

Proof-irrelevance on subsets of decidable sets

Lemma `inj_pairP2` :

$\forall (P:U \rightarrow \text{Prop}) (x:U) (p \ q:P \ x),$   
 $\text{ex\_intro } P \ x \ p = \text{ex\_intro } P \ x \ q \rightarrow p = q.$

Injectivity of equality on dependent pairs in `Type`

Lemma `inj_pair2` :

$\forall (P:U \rightarrow \text{Type}) (p:U) (x \ y:P \ p),$

$\text{existT } P \ p \ x = \text{existT } P \ p \ y \rightarrow x = y.$

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` :  $\forall A:\text{Type}, (\forall x \ y:A, \{x=y\}+\{x\neq y\}) \rightarrow$   
 $(\forall (P:A \rightarrow \text{Type}) (p:A) (x \ y:P \ p), \text{existT } P \ p \ x = \text{existT } P \ p \ y \rightarrow x = y).$

Examples of short direct proofs of unicity of reflexivity proofs on specific domains

Lemma `UIP_refl_unit`  $(x : \text{tt} = \text{tt}) : x = \text{eq\_refl } \text{tt}.$

Lemma `UIP_refl_bool`  $(b:\text{bool}) (x : b = b) : x = \text{eq\_refl}.$

Lemma `UIP_refl_nat`  $(n:\text{nat}) (x : n = n) : x = \text{eq\_refl}.$

## Chapter 32

# 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** :  $\forall P:\text{Prop}, \{P\} + \{\neg P\}$ .

**Theorem classical\_definite\_description** :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow$   
 $\{x : A \mid (\exists! x : A, P x) \rightarrow P x\}$ .

Church's iota operator

**Definition iota** ( $A : \text{Type}$ ) ( $i:\text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  $A$   
:= `proj1_sig (classical_definite_description P i)`.

**Definition iota\_spec** ( $A : \text{Type}$ ) ( $i:\text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  
 $(\exists! x:A, P x) \rightarrow P (\text{iota } i P)$   
:= `proj2_sig (classical_definite_description P i)`.

Axiom of unique “choice” (functional reification of functional relations) **Theorem dependent\_unique\_choice** :

$\forall (A:\text{Type}) (B:A \rightarrow \text{Type}) (R:\forall x:A, B x \rightarrow \text{Prop}),$   
 $(\forall x:A, \exists! y : B x, R x y) \rightarrow$   
 $(\exists f : (\forall x:A, B x), \forall x:A, R x (f x)).$

**Theorem unique\_choice** :

$\forall (A B:\text{Type}) (R:A \rightarrow B \rightarrow \text{Prop}),$   
 $(\forall x:A, \exists! y : B, R x y) \rightarrow$   
 $(\exists f : A \rightarrow B, \forall x:A, R x (f x)).$

Compatibility lemmas

**Unset Implicit Arguments.**



Definition dependent\_description := dependent\_unique\_choice.  
Definition description := unique\_choice.

## Chapter 33

# 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 :
```

```
   $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$   
     $(\exists! x, P x) \rightarrow \{ x : A \mid P x \}.$ 
```

## Chapter 34

# 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* :

$$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow \{ x : A \mid (\exists x, P x) \rightarrow P x \}.$$

Lemma *constructive\_indefinite\_description* :

$$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), (\exists x, P x) \rightarrow \{ x : A \mid P x \}.$$

Lemma *small\_drinkers'\_paradox* :

$$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow \exists x, (\exists x, P x) \rightarrow P x.$$

Theorem *iota\_statement* :

$$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow \{ x : A \mid (\exists! x : A, P x) \rightarrow P x \}.$$

Lemma *constructive\_definite\_description* :

$$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), (\exists! x, P x) \rightarrow \{ x : A \mid P x \}.$$

Hilbert's epsilon operator and its specification

Definition *epsilon* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  $A$   
:= proj1\_sig (*epsilon\_statement*  $P i$ ).

Definition *epsilon\_spec* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  
( $\exists x, P x$ )  $\rightarrow P$  (*epsilon*  $i P$ )  
:= proj2\_sig (*epsilon\_statement*  $P i$ ).

Church's iota operator and its specification

Definition *iota* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  $A$

```

:= 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 35

### 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 36

# 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 König'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**  $\rightarrow$  Prop) : **nat**  $\rightarrow$  **list bool**  $\rightarrow$  Prop :=

| here  $l$  :  $\neg P$   $l \rightarrow$  **is\_path\_from**  $P$  0  $l$

| next\_left  $l$   $n$  :  $\neg P$   $l \rightarrow$  **is\_path\_from**  $P$   $n$  (true:: $l$ )  $\rightarrow$  **is\_path\_from**  $P$  (S  $n$ )  $l$

| next\_right  $l$   $n$  :  $\neg P$   $l \rightarrow$  **is\_path\_from**  $P$   $n$  (false:: $l$ )  $\rightarrow$  **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**  $P$   $n$   $l \leftrightarrow \exists l', \text{length } l' = n \wedge \forall n', n' \leq n \rightarrow \neg P$  (**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**  $\rightarrow$  Prop)  $l$  :=  $\forall n, \text{is\_path\_from } P$   $n$   $l$ .

*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**  $\rightarrow$  Prop) :=

$\exists (X$ :**nat**  $\rightarrow$  Prop),  $\forall l, \text{approx } X$   $l \rightarrow \neg P$   $l$ .

*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**  $\rightarrow$  Prop) : **nat**  $\rightarrow$  **list bool**  $\rightarrow$  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  $true::true::\dots$  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 :

$\forall P, (\forall l, P\ l \vee \neg P\ l) \rightarrow \forall n\ l, \text{inductively\_barred\_at } P\ n\ l \vee \neg \text{inductively\_barred\_at } P\ n\ l.$

Lemma inductively\_barred\_at\_is\_path\_from\_decidable :

$\forall P, (\forall l, P\ l \vee \neg P\ l) \rightarrow \text{demorgan\_inductively\_barred\_at } P.$

Main corollary

Corollary WeakKonigsLemma :  $\forall P, (\forall l, P\ l \vee \neg P\ l) \rightarrow$

$\text{infinite\_from } P\ \square \rightarrow \text{has\_infinite\_path } P.$



## Chapter 37

# Library Coq.Logic.StrictProp

Utilities for *SProp* users.

Record **Box** (*A:SProp*) : Prop := box { unbox : *A* }.

Inductive **Squash** (*A*:Type) : *SProp* := squash : *A* → **Squash** *A*.

Inductive **sEmpty** : *SProp* :=.

Inductive **sUnit** : *SProp* := stt.

Record **Ssig** {*A*:Type} (*P*:*A*→*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 38

# 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

### 38.1 Prop degeneracy = excluded-middle + prop extensionality

i.e.  $(\forall A, A = \text{True} \vee A = \text{False}) \leftrightarrow (\forall A, A \vee \neg A) \wedge (\forall A B, (A \leftrightarrow B) \rightarrow A = B)$

*prop\_degeneracy* (also referred to as propositional completeness) asserts (up to consistency) that there are only two distinct formulas **Definition** *prop\_degeneracy* :=  $\forall A:\text{Prop}, A = \text{True} \vee A = \text{False}$ .

*prop\_extensionality* asserts that equivalent formulas are equal **Definition** *prop\_extensionality* :=  $\forall A B:\text{Prop}, (A \leftrightarrow B) \rightarrow A = B$ .

*excluded\_middle* asserts that we can reason by case on the truth or falsity of any formula **Definition** *excluded\_middle* :=  $\forall A:\text{Prop}, A \vee \neg A$ .

We show *prop\_degeneracy*  $\leftrightarrow$  (*prop\_extensionality*  $\wedge$  *excluded\_middle*)

**Lemma** *prop\_degen\_ext* : *prop\_degeneracy*  $\rightarrow$  *prop\_extensionality*.

**Lemma** *prop\_degen\_em* : *prop\_degeneracy*  $\rightarrow$  *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 :=  $\forall A:\text{Prop}, A \rightarrow A = \text{True}$ .

Lemma provable\_prop\_ext :

prop\_extensionality → provable\_prop\_extensionality.

## 38.2 Classical logic and proof-irrelevance

### 38.2.1 CC |- prop ext + A inhabited -> (A = A->A) -> A has fixpoint

We successively show that:

*prop\_extensionality* implies equality of  $A$  and  $A \rightarrow A$  for inhabited  $A$ , which implies the existence of a (trivial) retract from  $A \rightarrow A$  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 →  $\forall A:\text{Prop}, \text{inhabited } A \rightarrow (A \rightarrow A) = A$ .

Record **retract** ( $A B:\text{Prop}$ ) : Prop :=

{f1 :  $A \rightarrow B$ ; f2 :  $B \rightarrow A$ ; f1\_o\_f2 :  $\forall x:B, f1 (f2 x) = x$ }.

Lemma prop\_ext\_retract\_A\_A\_imp\_A :

prop\_extensionality →  $\forall A:\text{Prop}, \text{inhabited } A \rightarrow \text{retract } A (A \rightarrow A)$ .

Record **has\_fixpoint** ( $A:\text{Prop}$ ) : Prop :=

{F :  $(A \rightarrow A) \rightarrow A$ ; Fix :  $\forall f:A \rightarrow A, F f = f (F f)$ }.

Lemma ext\_prop\_fixpoint :

prop\_extensionality →  $\forall A:\text{Prop}, \text{inhabited } A \rightarrow \text{has\_fixpoint } A$ .

Remark: *prop\_extensionality* can be replaced in lemma *ext\_prop\_fixpoint* by the weakest property *provable\_prop\_extensionality*.

### 38.2.2 CC |- prop\_ext /\ dep elim on bool -> proof-irrelevance

*proof\_irrelevance* asserts equality of all proofs of a given formula Definition proof\_irrelevance :=  $\forall (A:\text{Prop}) (a1 a2:A), a1 = a2$ .

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* :  $\forall C:\text{Prop}, C \rightarrow C \rightarrow \text{bool} \rightarrow C$ .

```

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  $\text{bool} = (C:\text{Prop})C \rightarrow C \rightarrow C$  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*.

### 38.2.3 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.

### 38.2.4 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  $p2b\ A := or\_elim\ A\ (\neg A)\ B\ (\text{fun } \_ \Rightarrow b1)\ (\text{fun } \_ \Rightarrow b2)\ (em\ A)$ .

Let  $b2p\ b := b1 = b$ .

Lemma  $p2p1 : \forall A:\text{Prop}, A \rightarrow b2p\ (p2b\ A)$ .

Lemma  $p2p2 : b1 \neq b2 \rightarrow \forall A:\text{Prop}, b2p\ (p2b\ A) \rightarrow A$ .

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.  $\forall A, \neg A \setminus \sim \sim A$ , 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  $or : \text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop}$ .

Variable  $or\_introl : \forall A\ B:\text{Prop}, A \rightarrow or\ A\ B$ .

Variable  $or\_intror : \forall A\ B:\text{Prop}, B \rightarrow or\ A\ B$ .

Hypothesis  $or\_elim : \forall A\ B\ C:\text{Prop}, (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow or\ A\ B \rightarrow C$ .

Hypothesis

$or\_elim\_redl :$

$\forall (A\ B\ C:\text{Prop})\ (f:A \rightarrow C)\ (g:B \rightarrow C)\ (a:A),$   
 $f\ a = or\_elim\ A\ B\ C\ f\ g\ (or\_introl\ A\ B\ a).$

Hypothesis

$or\_elim\_redr :$

$\forall (A\ B\ C:\text{Prop})\ (f:A \rightarrow C)\ (g:B \rightarrow C)\ (b:B),$   
 $g\ b = or\_elim\ A\ B\ C\ f\ g\ (or\_intror\ A\ B\ b).$

Hypothesis

$or\_dep\_elim :$

$\forall (A\ B:\text{Prop})\ (P:or\ A\ B \rightarrow \text{Prop}),$   
 $(\forall a:A, P\ (or\_introl\ A\ B\ a)) \rightarrow$   
 $(\forall b:B, P\ (or\_intror\ A\ B\ b)) \rightarrow \forall b:or\ A\ B, P\ b.$

Hypothesis  $wem : \forall A:\text{Prop}, or\ (\sim \sim A)\ (\neg A)$ .

Variable  $B : \text{Prop}$ .

Variables  $b1\ b2 : B$ .

$p2b$  and  $b2p$  form a retract if  $\neg b1 = b2$

Let  $p2b\ (A:\text{NProp}) := or\_elim\ (\sim \sim \text{El}\ A)\ (\neg \text{El}\ A)\ B\ (\text{fun } \_ \Rightarrow b1)\ (\text{fun } \_ \Rightarrow b2)\ (wem\ (\text{El}\ A))$ .

Let  $b2p\ b : \text{NProp} := \text{exist}\ (\text{fun } P \Rightarrow \sim \sim P \rightarrow P)\ (\sim \sim (b1 = b))\ (\text{fun } h\ x \Rightarrow h\ (\text{fun } k \Rightarrow k\ x))$ .

Lemma  $wp2p1 : \forall A:\text{NProp}, \text{El}\ A \rightarrow \text{El}\ (b2p\ (p2b\ A))$ .

Lemma  $wp2p2 : b1 \neq b2 \rightarrow \forall A:\text{NProp}, \text{El}\ (b2p\ (p2b\ A)) \rightarrow \text{El}\ A$ .

By Hurkens's paradox, we get a weak form of proof irrelevance.

Theorem  $wproof\_irrelevance\_cc : \sim \sim (b1 = b2)$ .

End Proof\_irrelevance\_WEM\_CC.

### 38.2.5 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* :  $\forall A:\text{Prop}, A \vee \neg A$ .

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* :  $\forall (B:\text{Prop}) (b1\ b2:B), b1 = b2$ .

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* :  $\forall A:\text{Prop}, \neg\neg A \vee \neg A$ .

Theorem *wem\_proof\_irrelevance\_cci* :  $\forall (B:\text{Prop}) (b1\ b2:B), \neg\neg b1 = b2$ .

End Weak\_proof\_irrelevance\_CCI.

Remark: in the Set-impredicative CCI, Hurkens' paradox still holds with *bool* in **Set** and since  $\neg\text{true}=\text{false}$  for *true* and *false* in *bool* from **Set**, we get the inconsistency of *em* :  $\forall A:\text{Prop}, \{A\}+\{\neg A\}$  in the Set-impredicative CCI.

## 38.3 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

### 38.3.1 Weak excluded-middle

The weak classical logic based on  $\neg\neg A \vee \neg A$  is referred to with name KC in [ChagrovZakharyashev97]. See [SorbiTerwijn11] for a short survey.

[ChagrovZakharyashev97] Alexander Chagrov and Michael Zakharyashev, "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* :=

$\forall A:\text{Prop}, \sim\sim A \vee \neg A.$

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* :=

$\forall A B:\text{Prop}, ((A \rightarrow B) \rightarrow B) \vee (A \rightarrow B).$

Classical De Morgan's law

**Definition** *classical\_de\_morgan\_law* :=

$\forall A B:\text{Prop}, \sim(A \wedge B) \rightarrow \neg A \vee \neg B.$

### 38.3.2 Gödel-Dummett axiom

$(A \rightarrow B) \vee (B \rightarrow A)$  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* :=  $\forall A B:\text{Prop}, (A \rightarrow B) \vee (B \rightarrow A).$

**Lemma** *excluded\_middle\_Godel\_Dummett* : *excluded\_middle*  $\rightarrow$  *GodelDummett*.

$(A \rightarrow B) \vee (B \rightarrow A)$  is equivalent to  $(C \rightarrow A \vee B) \rightarrow (C \rightarrow A) \vee (C \rightarrow B)$  (proof from [Dummett59])

**Definition** *RightDistributivityImplicationOverDisjunction* :=

$\forall A B C:\text{Prop}, (C \rightarrow A \vee B) \rightarrow (C \rightarrow A) \vee (C \rightarrow B).$

**Lemma** *Godel\_Dummett\_iff\_right\_distr\_implication\_over\_disjunction* :

*GodelDummett*  $\leftrightarrow$  *RightDistributivityImplicationOverDisjunction*.

$(A \rightarrow B) \vee (B \rightarrow A)$  is stronger than the weak excluded middle

**Lemma** *Godel\_Dummett\_weak\_excluded\_middle* :

*GodelDummett*  $\rightarrow$  *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*  $\leftrightarrow$  *classical\_de\_morgan\_law*.

### 38.3.3 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 Grundlagenforschung*, 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 :=

$\forall (A:\text{Type}) (P:A \rightarrow \text{Prop}) (Q:\text{Prop}),$   
 $\text{inhabited } A \rightarrow (Q \rightarrow \exists x, P x) \rightarrow \exists x, Q \rightarrow P x.$

Lemma

`independence_general_premises_right_distr_implication_over_disjunction :`  
`IndependenceOfGeneralPremises  $\rightarrow$  RightDistributivityImplicationOverDisjunction.`

Lemma `independence_general_premises_Godel_Dummett :`

`IndependenceOfGeneralPremises  $\rightarrow$  GodelDummett.`

Independence of general premises is equivalent to the drinker's paradox

Definition DrinkerParadox :=

$\forall (A:\text{Type}) (P:A \rightarrow \text{Prop}),$   
 $\text{inhabited } A \rightarrow \exists x, (\exists x, P x) \rightarrow P x.$

Lemma `independence_general_premises_drinker :`

`IndependenceOfGeneralPremises  $\leftrightarrow$  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.  $\text{False} \rightarrow \forall A, A$ )

Definition `generalized_excluded_middle :=`

$\forall A B:\text{Prop}, A \vee (A \rightarrow B).$

Lemma `excluded_middle_independence_general_premises :`

`generalized_excluded_middle  $\rightarrow$  DrinkerParadox.`

## 38.4 Axioms equivalent to classical logic

### 38.4.1 Principle of unrestricted minimization

Require Import Coq.Arith.PeanoNat.

Definition `Minimal (P:nat  $\rightarrow$  Prop) (n:nat) : Prop :=`

$P n \wedge \forall k, P k \rightarrow n \leq k.$

Definition `Minimization_Property (P : nat  $\rightarrow$  Prop) : Prop :=`

$\forall n, P n \rightarrow \exists m, \text{Minimal } P m.$

Section `Unrestricted_minimization_entails_excluded_middle.`

Hypothesis `unrestricted_minimization`:  $\forall P, \text{Minimization\_Property } P.$

Theorem `unrestricted_minimization_entails_excluded_middle` :  $\forall A, A \vee \neg A.$

End `Unrestricted_minimization_entails_excluded_middle.`

Require Import Wf\_nat.

Section `Excluded_middle_entails_unrestricted_minimization.`

Hypothesis `em` :  $\forall A, A \vee \neg A.$

Theorem `excluded_middle_entails_unrestricted_minimization` :

$\forall P, \text{Minimization\_Property } P.$

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  $s : \mathbf{nat} \rightarrow \mathbf{bool}$ .

Let  $P\ n := \exists\ k, n \leq k \wedge s\ k = \mathbf{true}$ .

Example undecidable\_predicate\_with\_the\_minimization\_property :  
Minimization\_Property  $P$ .

End Example\_of\_undecidable\_predicate\_with\_the\_minimization\_property.

### 38.4.2 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  $\rightarrow$  excluded\_middle.

Theorem excluded\_middle\_imp\_representative\_boolean\_partition :  
excluded\_middle  $\rightarrow$  representative\_boolean\_partition.

Theorem excluded\_middle\_iff\_representative\_boolean\_partition :  
excluded\_middle  $\leftrightarrow$  representative\_boolean\_partition.

## Chapter 39

# 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  $EM : \forall P:\text{Prop}, P \vee \neg P$ .

Conditional on any proposition. Definition  $\text{IFProp} (P B:\text{Prop}) (e1 e2:P) :=$

```
match EM B with  
| or_introl _  $\Rightarrow e1$   
| or_intror _  $\Rightarrow e2$   
end.
```

Axiom of choice applied to disjunction. Provable in Coq because of dependent elimination.

Lemma AC\_IF :

```
 $\forall (P B:\text{Prop}) (e1 e2:P) (Q:P \rightarrow \text{Prop}),$   
 $(B \rightarrow Q e1) \rightarrow (\neg B \rightarrow Q e2) \rightarrow Q (\text{IFProp } B e1 e2).$ 
```

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  $Bool : \text{Prop}$ .

Variable  $T : Bool$ .

Variable  $F : Bool$ .

The powerset operator **Definition**  $\text{pow} (P:\text{Prop}) := P \rightarrow \text{Bool}$ .

A piece of theory about retracts **Section** Retracts.

**Variables**  $A B : \text{Prop}$ .

**Record** **retract** :  $\text{Prop} :=$

$\{i : A \rightarrow B; j : B \rightarrow A; \text{inv} : \forall a:A, j (i a) = a\}$ .

**Record** **retract\_cond** :  $\text{Prop} :=$

$\{i2 : A \rightarrow B; j2 : B \rightarrow A; \text{inv2} : \text{retract} \rightarrow \forall a:A, j2 (i2 a) = a\}$ .

The dependent elimination above implies the axiom of choice:

**Lemma** **AC** :  $\forall r:\text{retract\_cond}, \text{retract} \rightarrow \forall a:A, j2 r (i2 r a) = a$ .

**End** Retracts.

This lemma is basically a commutation of implication and existential quantification:  $(\exists x \mid A \rightarrow P(x)) \Leftrightarrow (A \rightarrow \exists x \mid P(x))$  which is provable in classical logic ( $\Rightarrow$  is already provable in intuitionistic logic).

**Lemma** **L1** :  $\forall A B:\text{Prop}, \text{retract\_cond} (\text{pow } A) (\text{pow } B)$ .

The paradoxical set **Definition**  $U := \forall P:\text{Prop}, \text{pow } P$ .

Bijection between  $U$  and  $(\text{pow } U)$  **Definition**  $f (u:U) : \text{pow } U := u U$ .

**Definition**  $g (h:\text{pow } U) : U :=$

$\text{fun } X \Rightarrow \text{let } lX := j2 (L1 X U) \text{ in let } rU := i2 (L1 U U) \text{ in } lX (rU h)$ .

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** :  $\text{retract} (\text{pow } U) U$ .

Encoding of Russel's paradox

The boolean negation. **Definition** **Not\_b**  $(b:\text{Bool}) := \text{IFProp} (b = T) F T$ .

the set of elements not belonging to itself **Definition** **R** :  $U := g (\text{fun } u:U \Rightarrow \text{Not\_b } (u U u))$ .

**Lemma** **not\_has\_fixpoint** :  $R R = \text{Not\_b } (R R)$ .

**Theorem** **classical\_proof\_irrelevance** :  $T = F$ .

$\#[\text{deprecated}(\text{since} = "8.8", \text{note} = "Use \text{classical\_proof\_irrelevance} \text{ instead.}")]$

**Notation** **classical\_proof\_irrelevance** := **classical\_proof\_irrelevance**.

**End** Berardis\_paradox.

## Chapter 40

### 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` :  $\forall \{A\} \{B : \text{Type}\} \{f\} \{g : A \rightarrow B\},$   
 $f = g \rightarrow \forall x, f\ x = g\ x.$

**Lemma** `equal_f_dep` :  $\forall \{A\} \{B\} \{f\} \{g : \forall (x : A), B\ x\},$   
 $f = g \rightarrow \forall x, f\ x = g\ x.$

Statements of functional extensionality for simple and dependent functions.

**Axiom** `functional_extensionality_dep` :  $\forall \{A\} \{B : A \rightarrow \text{Type}\},$   
 $\forall (f\ g : \forall x : A, B\ x),$   
 $(\forall x, f\ x = g\ x) \rightarrow f = g.$

**Lemma** `functional_extensionality`  $\{A\} \{B\} (f\ g : A \rightarrow B) :$   
 $(\forall x, f\ x = g\ x) \rightarrow f = g.$

Extensionality of  $\forall$ s follows from functional extensionality. **Lemma** `forall_extensionality`  $\{A\} \{B$   
 $C : A \rightarrow \text{Type}\} (H : \forall x : A, B\ x = C\ x)$   
 $: (\forall x, B\ x) = (\forall x, C\ x).$

**Lemma** `forall_extensionalityP`  $\{A\} \{B\} \{C : A \rightarrow \text{Prop}\} (H : \forall x : A, B\ x = C\ x)$   
 $: (\forall x, B\ x) = (\forall x, C\ x).$

**Lemma** `forall_extensionalityS`  $\{A\} \{B\} \{C : A \rightarrow \text{Set}\} (H : \forall x : A, B\ x = C\ x)$   
 $: (\forall x, B\ x) = (\forall x, C\ x).$

A version of `functional_extensionality_dep` which is provably equal to `eq_refl` on `fun _ => eq_refl`  
**Definition** `functional_extensionality_dep_good`

$\{A\} \{B : A \rightarrow \text{Type}\}$   
 $(f\ g : \forall x : A, B\ x)$   
 $(H : \forall x, f\ x = g\ x)$   
 $: f = g$   
 $:= \text{eq\_trans } (\text{eq\_sym } (\text{functional\_extensionality\_dep } f\ f\ (\text{fun } _ \Rightarrow \text{eq\_refl})))$

```

      (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 :=
    lazy match 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 :=
  lazy match 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 :=
  lazy match 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 :=
  lazy match (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 :=
  lazy match 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
      lazy match 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 :=
  lazy match 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 41

# 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 lsHProp ( $P : \text{Type}$ ) : `Prop`  
:=  $\forall p\ q : P, p = q$ .

Definition lsHSet ( $X : \text{Type}$ ) : `Prop`  
:=  $\forall (x\ y : X) (p\ q : x = y), p = q$ .

Definition lsHOneType ( $X : \text{Type}$ ) : `Prop`  
:=  $\forall (x\ y : X) (p\ q : x = y) (r\ s : p = q), r = s$ .

Lemma forall\_hprop :  $\forall (A : \text{Type}) (P : A \rightarrow \text{Prop})$ ,  
( $\forall x:A, \text{lsHProp } (P\ x)$ )  
 $\rightarrow \text{lsHProp } (\forall x:A, P\ x)$ .

Lemma and\_hprop :  $\forall P\ Q : \text{Prop}$ ,  
 $\text{lsHProp } P \rightarrow \text{lsHProp } Q \rightarrow \text{lsHProp } (P \wedge Q)$ .

Lemma impl\_hprop :  $\forall P\ Q : \text{Prop}$ ,  
 $\text{lsHProp } Q \rightarrow \text{lsHProp } (P \rightarrow Q)$ .

Lemma false\_hprop : `lsHProp False`.

Lemma true\_hprop : `lsHProp True`.

Lemma not\_hprop :  $\forall P : \text{Type}, \text{lsHProp } (P \rightarrow \text{False})$ .

Lemma hset\_hprop :  $\forall X : \text{Type}$ ,  
 $\text{lsHProp } X \rightarrow \text{lsHSet } X$ .

Lemma eq\_trans\_cancel :  $\forall \{X : \text{Type}\} \{x\ y\ z : X\} (p : x = y) (q\ r : y = z)$ ,  
 $(\text{eq\_trans } p\ q = \text{eq\_trans } p\ r) \rightarrow q = r$ .

Lemma hset\_hOneType :  $\forall X : \text{Type}$ ,  
 $\text{lsHSet } X \rightarrow \text{lsHOneType } X$ .

Lemma hprop\_hprop :  $\forall X : \text{Type}$ ,

```
IsHProp (IsHProp X).  
Lemma hprop_hset : ∀ X : Type,  
  IsHProp (IsHSet X).
```

## Chapter 42

# 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* :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$   
 $(\exists x, P x) \rightarrow \{ x : A \mid P x \}.$

Lemma *constructive\_definite\_description* :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}),$   
 $(\exists! x, P x) \rightarrow \{ x : A \mid P x \}.$

Theorem *excluded\_middle\_informative* :  $\forall P : \text{Prop}, \{P\} + \{\neg P\}.$

Theorem *classical\_indefinite\_description* :

$\forall (A : \text{Type}) (P : A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow$   
 $\{ x : A \mid (\exists x, P x) \rightarrow P x \}.$

Hilbert's epsilon operator

Definition *epsilon* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  $A$   
:= proj1\_sig (classical\_indefinite\_description  $P$   $i$ ).

Definition *epsilon\_spec* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  
 $(\exists x, P x) \rightarrow P$  (*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 Cast  ran)

Lemma epsilon\_inh\_irrelevance :

$\forall (A:\text{Type}) (i\ j : \mathbf{inhabited}\ A) (P:A\rightarrow\text{Prop}),$   
 $(\exists x, P\ x) \rightarrow \text{epsilon}\ i\ P = \text{epsilon}\ j\ P.$

Opaque epsilon.

### Weaker lemmas (compatibility lemmas)

Theorem choice :

$\forall (A\ B : \text{Type}) (R : A\rightarrow B\rightarrow\text{Prop}),$   
 $(\forall x : A, \exists y : B, R\ x\ y) \rightarrow$   
 $(\exists f : A\rightarrow B, \forall x : A, R\ x\ (f\ x)).$

## Chapter 43

# 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 44

# 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  $\leftrightarrow$  Equality of projections from diagonal
3. Functional extensionality  $\leftrightarrow$  Unicity of inverse bijections
4. Functional extensionality  $\leftrightarrow$  Bijectivity of bijective composition

Set Implicit Arguments.

### 44.1 Definitions

Being an inverse

**Definition** `is_inverse`  $A\ B\ f\ g := (\forall a:A, g\ (f\ a) = a) \wedge (\forall b:B, f\ (g\ b) = b)$ .

The diagonal over A and the one-one correspondence with A

`#[universes(template)]`

**Record** `Delta`  $A := \{ \text{pi1}:A; \text{pi2}:A; \text{eq:pi1=pi2} \}$ .

**Definition** `delta`  $\{A\}\ (a:A) := \{ | \text{pi1} := a; \text{pi2} := a; \text{eq} := \text{eq_refl } a \}$ .

**Lemma** `diagonal_projs_same_behavior` :  $\forall A\ (x:\text{Delta } A), \text{pi1 } x = \text{pi2 } x$ .

**Lemma** `diagonal_inverse1` :  $\forall A, \text{is\_inverse } (A:=A) \text{ delta pi1}$ .

**Lemma** `diagonal_inverse2` :  $\forall A, \text{is\_inverse } (A:=A) \text{ delta pi2}$ .

Functional extensionality

Equality of projections from diagonal

Unicity of bijection inverse

Bijectivity of bijective composition

Definition action  $A \ B \ C \ (f:A \rightarrow B) := (\text{fun } h:B \rightarrow C \Rightarrow \text{fun } x \Rightarrow h \ (f \ x))$ .

## 44.2 Functional extensionality $\leftrightarrow$ Equality of projections from diagonal

Theorem `FuncExt_iff_EqDeltaProjs` : `FunctionalExtensionality`  $\leftrightarrow$  `EqDeltaProjs`.

## 44.3 Functional extensionality $\leftrightarrow$ Unicity of bijection inverse

Lemma `FuncExt_UniqInverse` : `FunctionalExtensionality`  $\rightarrow$  `UniquelInverse`.

Lemma `UniqInverse_EqDeltaProjs` : `UniquelInverse`  $\rightarrow$  `EqDeltaProjs`.

Theorem `FuncExt_iff_UniqInverse` : `FunctionalExtensionality`  $\leftrightarrow$  `UniquelInverse`.

## 44.4 Functional extensionality $\leftrightarrow$ Bijectivity of bijective composition

Lemma `FuncExt_BijComp` : `FunctionalExtensionality`  $\rightarrow$  `BijectivityBijectiveComp`.

Lemma `BijComp_FuncExt` : `BijectivityBijectiveComp`  $\rightarrow$  `FunctionalExtensionality`.

## Chapter 45

### 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 :

$\forall P:U \rightarrow \text{Prop}, \neg (\forall n:U, \neg P\ n) \rightarrow \exists n : U, P\ n.$

Lemma not\_all\_ex\_not :

$\forall P:U \rightarrow \text{Prop}, \neg (\forall n:U, P\ n) \rightarrow \exists n : U, \neg P\ n.$

Lemma not\_ex\_all\_not :

$\forall P:U \rightarrow \text{Prop}, \neg (\exists n : U, P\ n) \rightarrow \forall n:U, \neg P\ n.$

Lemma not\_ex\_not\_all :

$\forall P:U \rightarrow \text{Prop}, \neg (\exists n : U, \neg P\ n) \rightarrow \forall n:U, P\ n.$

Lemma ex\_not\_not\_all :

$\forall P:U \rightarrow \text{Prop}, (\exists n : U, \neg P\ n) \rightarrow \neg (\forall n:U, P\ n).$

Lemma all\_not\_not\_ex :

$\forall P:U \rightarrow \text{Prop}, (\forall n:U, \neg P\ n) \rightarrow \neg (\exists n : U, P\ n).$

End Generic.



## Chapter 46

# 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.  $\text{Eq\_rect\_eq} \leftrightarrow \text{Eq\_dep\_eq} \leftrightarrow \text{UIP} \leftrightarrow \text{UIP\_refl} \leftrightarrow \text{K}$
3. Definition of the functor that builds properties of dependent equalities assuming axiom `eq_rect_eq`

### 46.1 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**  $p\ x\ q\ y \rightarrow \text{exist } P\ p\ x = \text{exist } P\ q\ y$ .

Lemma **eq\_sig\_iff\_eq\_dep** :

$\forall (U:\text{Type}) (P:U \rightarrow \text{Prop}) (p\ q:U) (x:P\ p) (y:P\ q),$   
 $\text{exist } P\ p\ x = \text{exist } P\ q\ y \leftrightarrow \mathbf{eq\_dep}\ p\ x\ q\ y.$

Dependent equality is equivalent to a dependent pair of equalities

Set Implicit Arguments.

Lemma **eq\_sigT\_sig\_eq**  $X\ P\ (x1\ x2:X)\ H1\ H2 :$

$\text{existT } P\ x1\ H1 = \text{existT } P\ x2\ H2 \leftrightarrow \{H:x1=x2 \mid \text{rew } H \text{ in } H1 = H2\}.$

Lemma **eq\_sigT\_fst**  $X\ P\ (x1\ x2:X)\ H1\ H2\ (H:\text{existT } P\ x1\ H1 = \text{existT } P\ x2\ H2) :$   
 $x1 = x2.$

Lemma **eq\_sigT\_snd**  $X\ P\ (x1\ x2:X)\ H1\ H2\ (H:\text{existT } P\ x1\ H1 = \text{existT } P\ x2\ H2) :$   
 $\text{rew } (\text{eq\_sigT\_fst } H) \text{ in } H1 = H2.$

Lemma **eq\_sig\_fst**  $X\ P\ (x1\ x2:X)\ H1\ H2\ (H:\text{exist } P\ x1\ H1 = \text{exist } P\ x2\ H2) :$   
 $x1 = x2.$

Lemma **eq\_sig\_snd**  $X\ P\ (x1\ x2:X)\ H1\ H2\ (H:\text{exist } P\ x1\ H1 = \text{exist } P\ x2\ H2) :$   
 $\text{rew } (\text{eq\_sig\_fst } H) \text{ in } H1 = H2.$

Unset Implicit Arguments.

Exported hints

*#[global]*

Hint Resolve *eq\_dep\_intro*: core.

*#[global]*

Hint Immediate *eq\_dep\_sym*: core.

## 46.2 Eq\_rect\_eq <-> Eq\_dep\_eq <-> UIP <-> UIP\_refl <-> K

Section Equivalences.

Variable  $U:\text{Type}.$

Invariance by Substitution of Reflexive Equality Proofs

Definition **Eq\_rect\_eq\_on**  $(p : U) (Q : U \rightarrow \text{Type}) (x : Q\ p) :=$

$\forall (h : p = p), x = \text{eq\_rect } p\ Q\ x\ p\ h.$

Definition **Eq\_rect\_eq**  $:= \forall p\ Q\ x, \text{Eq\_rect\_eq\_on } p\ Q\ x.$

Injectivity of Dependent Equality

Definition **Eq\_dep\_eq\_on**  $(P : U \rightarrow \text{Type}) (p : U) (x : P\ p) :=$

$\forall (y : P\ p), \mathbf{eq\_dep}\ p\ x\ p\ y \rightarrow x = y.$

Definition **Eq\_dep\_eq**  $:= \forall P\ p\ x, \text{Eq\_dep\_eq\_on } P\ p\ x.$

Uniqueness of Identity Proofs (UIP)

Definition **UIP\_on\_**  $(x\ y : U) (p1 : x = y) :=$

$\forall (p2 : x = y), p1 = p2.$

Definition **UIP\_**  $:= \forall x\ y\ p1, \text{UIP\_on\_ } x\ y\ p1.$

Uniqueness of Reflexive Identity Proofs

**Definition** `UIP_refl_on_` ( $x : U$ ) :=

$\forall (p : x = x), p = \text{eq\_refl } x.$

**Definition** `UIP_refl_` :=  $\forall x, \text{UIP\_refl\_on\_ } x.$

Streicher's axiom K

**Definition** `Streicher_K_on_` ( $x : U$ ) ( $P : x = x \rightarrow \text{Prop}$ ) :=

$P (\text{eq\_refl } x) \rightarrow \forall p : x = x, P p.$

**Definition** `Streicher_K_` :=  $\forall x P, \text{Streicher\_K\_on\_ } x P.$

Injectivity of Dependent Equality is a consequence of Invariance by Substitution of Reflexive Equality Proof

**Lemma** `eq_rect_eq_on__eq_dep1_eq_on` ( $p : U$ ) ( $P : U \rightarrow \text{Type}$ ) ( $y : P p$ ) :

$\text{Eq\_rect\_eq\_on } p P y \rightarrow \forall (x : P p), \text{eq\_dep1 } p x p y \rightarrow x = y.$

**Lemma** `eq_rect_eq__eq_dep1_eq` :

$\text{Eq\_rect\_eq} \rightarrow \forall (P : U \rightarrow \text{Type}) (p : U) (x y : P p), \text{eq\_dep1 } p x p y \rightarrow x = y.$

**Lemma** `eq_rect_eq_on__eq_dep_eq_on` ( $p : U$ ) ( $P : U \rightarrow \text{Type}$ ) ( $x : P p$ ) :

$\text{Eq\_rect\_eq\_on } p P x \rightarrow \text{Eq\_dep\_eq\_on } P p x.$

**Lemma** `eq_rect_eq__eq_dep_eq` :  $\text{Eq\_rect\_eq} \rightarrow \text{Eq\_dep\_eq}.$

Uniqueness of Identity Proofs (UIP) is a consequence of Injectivity of Dependent Equality

**Lemma** `eq_dep_eq_on__UIP_on` ( $x y : U$ ) ( $p1 : x = y$ ) :

$\text{Eq\_dep\_eq\_on } (\text{fun } y \Rightarrow x = y) x \text{eq\_refl} \rightarrow \text{UIP\_on\_ } x y p1.$

**Lemma** `eq_dep_eq__UIP` :  $\text{Eq\_dep\_eq} \rightarrow \text{UIP\_}.$

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

**Lemma** `UIP_on__UIP_refl_on` ( $x : U$ ) :

$\text{UIP\_on\_ } x x \text{eq\_refl} \rightarrow \text{UIP\_refl\_on\_ } x.$

**Lemma** `UIP__UIP_refl` :  $\text{UIP\_} \rightarrow \text{UIP\_refl\_}.$

Streicher's axiom K is a direct consequence of Uniqueness of Reflexive Identity Proofs

**Lemma** `UIP_refl_on__Streicher_K_on` ( $x : U$ ) ( $P : x = x \rightarrow \text{Prop}$ ) :

$\text{UIP\_refl\_on\_ } x \rightarrow \text{Streicher\_K\_on\_ } x P.$

**Lemma** `UIP_refl__Streicher_K` :  $\text{UIP\_refl\_} \rightarrow \text{Streicher\_K\_}.$

We finally recover from K the Invariance by Substitution of Reflexive Equality Proofs

**Lemma** `Streicher_K_on__eq_rect_eq_on` ( $p : U$ ) ( $P : U \rightarrow \text{Type}$ ) ( $x : P p$ ) :

$\text{Streicher\_K\_on\_ } p (\text{fun } h \Rightarrow x = \text{rew} \rightarrow [P] h \text{ in } x) \rightarrow \text{Eq\_rect\_eq\_on } p P x.$

**Lemma** `Streicher_K__eq_rect_eq` :  $\text{Streicher\_K\_} \rightarrow \text{Eq\_rect\_eq}.$

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` :=  $\forall (P : U \rightarrow \text{Set}) (p : U) (x : P p) (h : p = p), x = \text{eq\_rec } p P x p h.$

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 → 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.

## 46.3 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 :  $\forall (x\ y:U) (p1\ p2:x = y), p1 = p2$ .

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma UIP\_refl :  $\forall (x:U) (p:x = x), p = \text{eq\_refl } x$ .

Streicher's axiom K is a direct consequence of Uniqueness of Reflexive Identity Proofs

Lemma Streicher\_K :

$\forall (x:U) (P:x = x \rightarrow \text{Prop}), P (\text{eq\_refl } x) \rightarrow \forall p:x = x, P\ p$ .

End Axioms.

UIP implies the injectivity of equality on dependent pairs in Type

Lemma inj\_pair2 :

$\forall (U:\text{Type}) (P:U \rightarrow \text{Type}) (p:U) (x\ y:P\ p),$   
 $\text{existT } P\ p\ x = \text{existT } P\ p\ y \rightarrow x = y$ .

Notation inj\_pairT2 := inj\_pair2.

End EQDEPTHEORY.

Basic facts about eq\_dep

Lemma f\_eq\_dep :

$\forall U (P:U \rightarrow \text{Type}) R\ p\ q\ x\ y (f:\forall p, P\ p \rightarrow R\ p),$   
 $\text{eq\_dep } p\ x\ q\ y \rightarrow \text{eq\_dep } p\ (f\ p\ x)\ q\ (f\ q\ y)$ .

Lemma eq\_dep\_non\_dep :

$\forall U\ P\ p\ q\ x\ y, @ \text{eq\_dep } U\ (\text{fun } _ \Rightarrow P) p\ x\ q\ y \rightarrow x = y$ .

Lemma f\_eq\_dep\_non\_dep :

$\forall U (P:U \rightarrow \text{Type}) R\ p\ q\ x\ y (f:\forall p, P\ p \rightarrow R),$   
 $\text{eq\_dep } p\ x\ q\ y \rightarrow f\ p\ x = f\ q\ y$ .

## Chapter 47

# 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) :  $\forall B:\text{Type}, B \rightarrow \text{Prop} :=$   
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 :  $\forall (A B:\text{Type}) (x:A) (y:B), \text{JMeq } x \ y \rightarrow \text{JMeq } y \ x$ .

#[global]

Hint Immediate JMeq\_sym : core.

Lemma JMeq\_trans :

$\forall (A \ B \ C:\text{Type}) (x:A) (y:B) (z:C), \text{JMeq } x \ y \rightarrow \text{JMeq } y \ z \rightarrow \text{JMeq } x \ z$ .

Axiom JMeq\_eq :  $\forall (A:\text{Type}) (x \ y:A), \text{JMeq } x \ y \rightarrow x = y$ .

Lemma JMeq\_ind :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Prop}),$   
 $P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y$ .

Lemma JMeq\_rec :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Set}),$   
 $P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y$ .

Lemma JMeq\_rect :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Type}),$   
 $P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y$ .

Lemma JMeq\_ind\_r :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Prop}),$   
 $P \ x \rightarrow \forall y, \text{JMeq } y \ x \rightarrow P \ y$ .

Lemma JMeq\_rec\_r :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Set}),$   
 $P \ x \rightarrow \forall y, \text{JMeq } y \ x \rightarrow P \ y$ .

Lemma JMeq\_rect\_r :  $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Type}),$   
 $P\ x \rightarrow \forall y, \mathbf{JMeq}\ y\ x \rightarrow P\ y.$

Lemma JMeq\_congr :

$\forall (A:\text{Type}) (x:A) (B:\text{Type}) (f:A \rightarrow B) (y:A), \mathbf{JMeq}\ x\ y \rightarrow f\ x = f\ y.$

$JMeq$  is equivalent to  $eq\_dep\ \text{Type}\ (\text{fun } X \Rightarrow X)$

Require Import Eqdep.

Lemma JMeq\_eq\_dep\_id :

$\forall (A\ B:\text{Type}) (x:A) (y:B), \mathbf{JMeq}\ x\ y \rightarrow \mathbf{eq\_dep}\ \text{Type}\ (\text{fun } X \Rightarrow X)\ A\ x\ B\ y.$

Lemma eq\_dep\_id\_JMeq :

$\forall (A\ B:\text{Type}) (x:A) (y:B), \mathbf{eq\_dep}\ \text{Type}\ (\text{fun } X \Rightarrow X)\ A\ x\ B\ y \rightarrow \mathbf{JMeq}\ x\ y.$

$eq\_dep\ U\ P\ p\ x\ q\ y$  is strictly finer than  $JMeq\ (P\ p)\ x\ (P\ q)\ y$

Lemma eq\_dep\_JMeq :

$\forall U\ P\ p\ x\ q\ y, \mathbf{eq\_dep}\ U\ P\ p\ x\ q\ y \rightarrow \mathbf{JMeq}\ x\ y.$

Lemma eq\_dep\_strictly\_stronger\_JMeq :

$\exists U\ P\ p\ q\ x\ y, \mathbf{JMeq}\ x\ y \wedge \neg \mathbf{eq\_dep}\ U\ P\ p\ x\ q\ y.$

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 :

$\forall U\ (P:U \rightarrow \text{Type})\ p\ q\ (x:P\ p)\ (y:P\ q),$   
 $p = q \rightarrow \mathbf{JMeq}\ x\ y \rightarrow \mathbf{eq\_dep}\ U\ P\ p\ x\ q\ y.$

Notation sym\_JMeq := JMeq\_sym (*only parsing*).

Notation trans\_JMeq := JMeq\_trans (*only parsing*).



## Chapter 48

# Library Coq.Logic.SetIsType

### 48.1 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 49

# Library Coq.Logic.Classical

Classical Logic

Require Export Classical\_Prop.

Require Export Classical\_Pred\_Type.

## Chapter 50

# 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  $M A \rightarrow A$ ) 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 ( $\forall A, \sim\sim A \vee \sim A$ ) 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 <sup>1</sup>).

## 50.1 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.

### 50.1.1 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* :  $\forall u:U1, (El1\ u \rightarrow U1) \rightarrow U1$ .

Notation " $\forall_1$ ’ *x* : *A*, *B*" := (*Forall1* *A* (**fun** *x*  $\Rightarrow$  *B*)).

Notation "*A*  $\rightarrow_1$ ’ *B*" := (*Forall1* *A* (**fun** \_  $\Rightarrow$  *B*)).

Variable *lam1* :  $\forall u\ B, (\forall x:El1\ u, El1\ (B\ x)) \rightarrow El1\ (\forall_1\ x:u, B\ x)$ .

Notation " $\lambda_1$ ’ *x*, *u*" := (*lam1* \_ \_ (**fun** *x*  $\Rightarrow$  *u*)).

Variable *app1* :  $\forall u\ B\ (f:El1\ (Forall1\ u\ B))\ (x:El1\ u), El1\ (B\ x)$ .

Notation "*f*  $\cdot_1$ ’ *x*" := (*app1* \_ \_ *f* *x*).

Variable *beta1* :  $\forall u\ B\ (f:\forall x:El1\ u, El1\ (B\ x))\ x,$   
 $(\lambda_1\ y, f\ y) \cdot_1\ x = f\ x$ .

---

<sup>1</sup><http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz>

**Closure by large products**  $U1$  only needs to quantify over itself. **Variable**  $ForallU1$  :  $(U1 \rightarrow U1) \rightarrow U1$ .

Notation " $\forall_2$ "  $A, F$ " :=  $(ForallU1 \text{ (fun } A \Rightarrow F))$ .

**Variable**  $lamU1$  :  $\forall F, (\forall A:U1, El1 (F A)) \rightarrow El1 (\forall_2 A, F A)$ .

Notation " $\lambda_2$ "  $x, u$ " :=  $(lamU1 \text{ - (fun } x \Rightarrow u))$ .

**Variable**  $appU1$  :  $\forall F (f:El1(\forall_2 A, F A)) (A:U1), El1 (F A)$ .

Notation " $f \cdot_1$ "  $[ A ]$ " :=  $(appU1 \text{ - } f A)$ .

**Variable**  $betaU1$  :  $\forall F (f:\forall A:U1, El1 (F A)) A, (\lambda_2 x, f x) \cdot_1 [ A ] = f A$ .

## Small universe

The small universe is an element of the large one. **Variable**  $u0$  :  $U1$ .

Notation  $U0$  :=  $(El1 u0)$ .

**Variable**  $El0$  :  $U0 \rightarrow \text{Type}$ .

**Closure by small product**  $U0$  does not need reduction rules **Variable**  $Forall0$  :  $\forall u:U0, (El0 u \rightarrow U0) \rightarrow U0$ .

Notation " $\forall_0$ "  $x : A, B$ " :=  $(Forall0 A \text{ (fun } x \Rightarrow B))$ .

Notation " $A \rightarrow_0 B$ " :=  $(Forall0 A \text{ (fun - } \Rightarrow B))$ .

**Variable**  $lam0$  :  $\forall u B, (\forall x:El0 u, El0 (B x)) \rightarrow El0 (\forall_0 x:u, B x)$ .

Notation " $\lambda_0$ "  $x, u$ " :=  $(lam0 \text{ - - (fun } x \Rightarrow u))$ .

**Variable**  $app0$  :  $\forall u B (f:El0 (Forall0 u B)) (x:El0 u), El0 (B x)$ .

Notation " $f \cdot_0$ "  $x$ " :=  $(app0 \text{ - - } f x)$ .

**Closure by large products** **Variable**  $ForallU0$  :  $\forall u:U1, (El1 u \rightarrow U0) \rightarrow U0$ .

Notation " $\forall_0^1$ "  $A : U, F$ " :=  $(ForallU0 U \text{ (fun } A \Rightarrow F))$ .

**Variable**  $lamU0$  :  $\forall U F, (\forall A:El1 U, El0 (F A)) \rightarrow El0 (\forall_0^1 A:U, F A)$ .

Notation " $\lambda_0^1$ "  $x, u$ " :=  $(lamU0 \text{ - - (fun } x \Rightarrow u))$ .

**Variable**  $appU0$  :  $\forall U F (f:El0(\forall_0^1 A:U, F A)) (A:El1 U), El0 (F A)$ .

Notation " $f \cdot_0$ "  $[ A ]$ " :=  $(appU0 \text{ - - } f A)$ .

### 50.1.2 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.
```

### 50.1.3 Hurkens's paradox.

An inhabitant of  $U0$  standing for *False*. **Variable**  $F:U0$ .

## Preliminary definitions

Definition  $V : U1 := \forall_2 A, ((A \rightarrow_1 u0) \rightarrow_1 A \rightarrow_1 u0) \rightarrow_1 A \rightarrow_1 u0$ .

Definition  $U : U1 := V \rightarrow_1 u0$ .

Definition  $sb (z:El1\ V) : El1\ V := \lambda_2 A, \lambda_1 r, \lambda_1 a, r \cdot_1 (z \cdot_1 [A] \cdot_1 r) \cdot_1 a$ .

Definition  $le (i:El1\ (U \rightarrow_1 u0)) (x:El1\ U) : U0 :=$

$x \cdot_1 (\lambda_2 A, \lambda_1 r, \lambda_1 a, i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [A] \cdot_1 r \cdot_1 a))$ .

Definition  $le' : El1\ ((U \rightarrow_1 u0) \rightarrow_1 U \rightarrow_1 u0) := \lambda_1 i, \lambda_1 x, le\ i\ x$ .

Definition  $induct (i:El1\ (U \rightarrow_1 u0)) : U0 :=$

$\forall_0^1 x:U, le\ i\ x \rightarrow_0 i \cdot_1 x$ .

Definition  $WF : El1\ U := \lambda_1 z, (induct\ (z \cdot_1 [U] \cdot_1 le'))$ .

Definition  $l (x:El1\ U) : U0 :=$

$(\forall_0^1 i:U \rightarrow_1 u0, le\ i\ x \rightarrow_0 i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 x)) \rightarrow_0 F$

## Proof

Lemma  $\Omega : El0\ (\forall_0^1 i:U \rightarrow_1 u0, induct\ i \rightarrow_0 i \cdot_1 WF)$ .

Proof.

```

refine (λ₀¹ i, λ₀ y, -).
refine (y ·₀ [-] ·₀ -).
unfold le, WF, induct. simplify.
refine (λ₀¹ x, λ₀ h0, -). simplify.
refine (y ·₀ [-] ·₀ -).
unfold le. simplify.
unfold sb at 1. simplify.
unfold le' at 1. simplify.
exact h0.

```

Qed.

Lemma  $lemma1 : El0\ (induct\ (\lambda_1 u, l\ u))$ .

Proof.

```

unfold induct.
refine (λ₀¹ x, λ₀ p, -). simplify.
refine (λ₀ q, -).
assert (El0 (l (λ₁ v, (sb v) ·₁ [U] ·₁ le' ·₁ x))) as h.
{ generalize (q ·₀ [λ₁ u, l u] ·₀ p). simplify.
  intros q'.
  exact q'. }
refine (h ·₀ -).
refine (λ₀¹ i, -).
refine (λ₀ h', -).
generalize (q ·₀ [λ₁ y, i ·₁ (λ₁ v, (sb v) ·₁ [U] ·₁ le' ·₁ y)]). simplify.
intros q'.
refine (q' ·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 :  $Elo ((\forall_0^1 i:U \longrightarrow_1 u0, \text{induct } i \longrightarrow_0 i \cdot_1 WF) \longrightarrow_0 F)$ .
Proof.
  refine ( $\lambda_0 x, -$ ).
  assert ( $Elo (I WF)$ ) as h.
  { generalize ( $x \cdot_0 [\lambda_1 u, I u] \cdot_0 \text{lemma1}$ ). simplify.
    intros q.
    exact q. }
  refine ( $h \cdot_0 -$ ). clear h.
  refine ( $\lambda_0^1 i, \lambda_0 h0, -$ ).
  generalize ( $x \cdot_0 [\lambda_1 y, i \cdot_1 (\lambda_1 v, (sb v) \cdot_1 [U] \cdot_1 le' \cdot_1 y)]$ ). simplify.
  intros q.
  refine ( $q \cdot_0 -$ ). clear q.
  unfold le in h0. simplify_in h0.
  unfold WF in h0. simplify_in h0.
  exact h0.
Qed.

Theorem paradox :  $Elo F$ .
Proof.
  exact (lemma2 ·  $0\Omega$ ).
Qed.

End Paradox.

```

The *paradox* tactic can be called as a shortcut to use the paradox. `Ltac paradox h := unshelve (refine ((fun h  $\Rightarrow$  -) (paradox - - - - - ))).`

End GENERIC.

## 50.2 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  $U2 := \text{Type}$ .

Let  $U1:U2 := \text{Type}$ .

Variable  $U0:U1$ .

**$U1$  is impredicative**

Variable  $u2u1 : U2 \rightarrow U1$ .

**Hypothesis**  $u22u1\_unit : \forall (c:U2), c \rightarrow u22u1\ c$ .

$u22u1\_counit$  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**  $u22u1\_counit : \forall (F:U1 \rightarrow U1), u22u1\ (\forall A, F\ A) \rightarrow (\forall A, F\ A)$ .

**Hypothesis**  $u22u1\_coherent : \forall (F:U1 \rightarrow U1) (f:\forall x:U1, F\ x) (x:U1),$   
 $u22u1\_counit\ _\text{ (}u22u1\_unit\ _\text{ f)}\ x = f\ x$ .

## **$U0$ is a retract of $U1$**

**Variable**  $u02u1 : U0 \rightarrow U1$ .

**Variable**  $u12u0 : U1 \rightarrow U0$ .

**Hypothesis**  $u12u0\_unit : \forall (b:U1), b \rightarrow u02u1\ (u12u0\ b)$ .

**Hypothesis**  $u12u0\_counit : \forall (b:U1), u02u1\ (u12u0\ b) \rightarrow b$ .

### **50.2.1 Paradox**

**Theorem**  $paradox : \forall F:U1, F$ .

**Proof.**

**intros**  $F$ .

*Generic.paradox*  $h$ .

Large universe      + **exact**  $U1$ .

+ **exact**  $(\text{fun } X \Rightarrow X)$ .

+ *cbn*. **exact**  $(\text{fun } u\ F \Rightarrow \forall x:u, F\ x)$ .

+ *cbn*. **exact**  $(\text{fun } \_ \_ x \Rightarrow x)$ .

+ *cbn*. **exact**  $(\text{fun } \_ \_ x \Rightarrow x)$ .

+ *cbn*. **exact**  $(\text{fun } F \Rightarrow u22u1\ (\forall x, F\ x))$ .

+ *cbn*. **exact**  $(\text{fun } \_ x \Rightarrow u22u1\_unit\ \_ x)$ .

+ *cbn*. **exact**  $(\text{fun } \_ x \Rightarrow u22u1\_counit\ \_ x)$ .

Small universe      + **exact**  $U0$ .

The interpretation of the small universe is the image of  $U0$  in  $U1$ .      + *cbn*. **exact**  $(\text{fun } X \Rightarrow u02u1\ X)$ .

+ *cbn*. **exact**  $(\text{fun } u\ F \Rightarrow u12u0\ (\forall x:(u02u1\ u), u02u1\ (F\ x)))$ .

+ *cbn*. **exact**  $(\text{fun } u\ F \Rightarrow u12u0\ (\forall x:u, u02u1\ (F\ x)))$ .

+ *cbn*. **exact**  $(u12u0\ F)$ .

+ *cbn* **in**  $h$ .

**exact**  $(u12u0\_counit\ \_ h)$ .

+ *cbn*. *easy*.

+ *cbn*. **intros**  $**$ . *now* **rewrite**  $u22u1\_coherent$ .

+ *cbn*. **intros**  $\times x$ . **exact**  $(u12u0\_unit\ \_ x)$ .

+ *cbn*. **intros**  $\times x$ . **exact**  $(u12u0\_counit\ \_ x)$ .

+ *cbn*. **intros**  $\times x$ . **exact**  $(u12u0\_unit\ \_ x)$ .

+ *cbn*. **intros**  $\times x$ . **exact**  $(u12u0\_counit\ \_ x)$ .

**Qed.**



End Paradox.

End NORETRACTTOIMPREDICATIVEUNIVERSE.

## 50.3 Modal fragments of Prop are not retracts

In presence of 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.

### 50.3.1 Monadic modality

Section Paradox.

Variable  $M : \text{Prop} \rightarrow \text{Prop}$ .

Hypothesis  $incr : \forall A B : \text{Prop}, (A \rightarrow B) \rightarrow M A \rightarrow M B$ .

Lemma strength:  $\forall A (P : A \rightarrow \text{Prop}), M(\forall x : A, P x) \rightarrow \forall x : A, M(P x)$ .

Proof.

intros  $A P h x$ .

eapply  $incr$  in  $h$ ; eauto.

Qed.

### 50.3.2 The universe of modal propositions

Definition  $M\text{Prop} := \{ P : \text{Prop} \mid M P \rightarrow P \}$ .

Definition  $\text{El} : M\text{Prop} \rightarrow \text{Prop} := @proj1\_sig \_ \_$ .

Lemma modal :  $\forall P : M\text{Prop}, M(\text{El } P) \rightarrow \text{El } P$ .

Proof.

intros  $[P m]$ . *cbn*.

exact  $m$ .

Qed.

Definition  $\text{Forall} \{A : \text{Type}\} (P : A \rightarrow M\text{Prop}) : M\text{Prop}$ .

Proof.

*unshelve* (*refine* (*exist*  $\_ \_$ )).

+ *exact* ( $\forall x : A, \text{El } (P x)$ ).

+ *intros*  $h x$ .

eapply *strength* in  $h$ .

eauto using *modal*.

Defined.

### 50.3.3 Retract of the modal fragment of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

Variable  $bool : M\text{Prop}$ .

Variable  $p2b : \text{MProp} \rightarrow \text{El } \text{bool}$ .  
 Variable  $b2p : \text{El } \text{bool} \rightarrow \text{MProp}$ .  
 Hypothesis  $p2p1 : \forall A:\text{MProp}, \text{El } (b2p (p2b A)) \rightarrow \text{El } A$ .  
 Hypothesis  $p2p2 : \forall A:\text{MProp}, \text{El } A \rightarrow \text{El } (b2p (p2b A))$ .

### 50.3.4 Paradox

Theorem paradox :  $\forall B:\text{MProp}, \text{El } B$ .

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.

## 50.4 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.

### 50.4.1 The universe of negative propositions.

Definition NProp := {  $P:\text{Prop} \mid \sim\sim P \rightarrow P$  }.

Definition El : NProp → Prop := @proj1\_sig \_ \_.

Section Paradox.

### 50.4.2 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* :  $\forall A:\text{NProp}, \text{El } (b2p (p2b A)) \rightarrow \text{El } A$ .

Hypothesis *p2p2* :  $\forall A:\text{NProp}, \text{El } A \rightarrow \text{El } (b2p (p2b A))$ .

### 50.4.3 Paradox

Theorem paradox :  $\forall B:\text{NProp}, \text{El } B$ .

Proof.

  intros *B*.

  unshelve (refine ((fun *h* ⇒ \_) (NoRetractToModalProposition.paradox \_ \_ \_ \_ \_))).

  + exact (fun *P* ⇒  $\sim\sim P$ ).

  + exact *bool*.

  + exact *p2b*.

  + exact *b2p*.

  + exact *B*.

  + exact *h*.

  + *cbn*. auto.

  + *cbn*. auto.

  + *cbn*. auto.

Qed.

End Paradox.

End NORETRACTTONEGATIVEPROP.

## 50.5 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.

### 50.5.1 The universe of propositions.

Definition NProp := { P:Prop | P → P }.

Definition El : NProp → Prop := @proj1\_sig \_ \_.

Section MParadox.

### 50.5.2 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)).

### 50.5.3 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.

### 50.5.4 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).

### 50.5.5 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.  $\exists$  (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.

## 50.6 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.

### 50.6.1 Assumption of a retract from Type into Prop

Variable down : Type1 → Prop.

Variable up : Prop → Type1.

Hypothesis up\_down :  $\forall (A:\text{Type1}), \text{up} (\text{down } A) = A :> \text{Type1}.$

### 50.6.2 Paradox

Theorem paradox :  $\forall P:\text{Prop}, P.$

Proof.

```

intros P.
Generic.paradox h.
  Large universe.    + exact Type1.
+ exact (fun X => X).
+ cbn. exact (fun u F =>  $\forall x, F x$ ).
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
+ exact (fun F =>  $\forall A:\text{Prop}, F(\text{up } A)$ ).
+ 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.

```

## 50.7 $A \neq \text{Type}$

No Coq universe can be equal to one of its elements.

Module TYPENEQSMALLTYPE.

Unset *Universe Polymorphism*.

Section Paradox.

### 50.7.1 Universe $U$ is equal to one of its elements.

Let  $U := \text{Type}$ .

Variable  $A:U$ .

Hypothesis  $h : U=A$ .

### 50.7.2 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  $\text{down} (X:U) : A := @eq\_rect \_ \_ (\text{fun } X \Rightarrow X) X \_ h$ .

Let  $\text{up} (X:A) : U := @eq\_rect\_r \_ \_ (\text{fun } X \Rightarrow X) X \_ h$ .

Lemma  $\text{up\_down} : \forall (X:U), \text{up} (\text{down } X) = X$ .

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.

```

## 50.8 Prop $\neq$ Type.

Special case of *TypeNeqSmallType*.

Module PROPNEQTYPE.

Theorem paradox : Prop  $\neq$  Type.

Proof.

```
  intros h.  
  unshelve (refine (TypeNeqSmallType.paradox _ _)).  
  + exact Prop.  
  + easy.
```

Qed.

End PROPNEQTYPE.



## Chapter 51

# 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 :
```

```
   $\forall (U:\text{Type}) (p:U) (Q:U \rightarrow \text{Type}) (x:Q\ p) (h:p = p), x = \text{eq\_rect } p\ Q\ x\ p\ h.$ 
```

```
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 52

# 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
|  $\square$   $\Rightarrow$  True
|  $b :: l \Rightarrow$ 
     $\forall P\ l \wedge$ 
    if  $b$  then inductively_barred  $P$  ( $\text{false} :: l$ ) else  $\neg$  inductively_barred  $P$  ( $\text{false} :: l$ )
end.

Lemma Y_unique :  $\forall P\ l1\ l2, \text{length } l1 = \text{length } l2 \rightarrow \forall P\ l1 \rightarrow \forall P\ l2 \rightarrow l1 = l2$ .

 $X$  is the translation of  $Y$  as a predicate
Definition  $X\ P\ n := \exists l, \text{length } l = n \wedge \forall P\ (\text{true} :: l)$ .

Lemma Y_approx :  $\forall P\ l, \text{approx } (X\ P)\ l \rightarrow \forall P\ l$ .

Theorem WeakFanTheorem :  $\forall P, \text{barred } P \rightarrow \text{inductively_barred } P\ \square$ .

```

## Chapter 53

# Library Coq.Logic.PropExtensionality

This module states propositional extensionality and draws consequences of it

Axiom *propositional\_extensionality* :

$\forall (P\ Q : \text{Prop}), (P \leftrightarrow Q) \rightarrow P = Q.$

Require Import ClassicalFacts.

Theorem proof\_irrelevance :  $\forall (P:\text{Prop}) (p1\ p2:P), p1 = p2.$

## Chapter 54

# Library Coq.Logic.Classical\_Prop

Classical Propositional Logic

Require Import ClassicalFacts.

#[global]

Hint Unfold not: core.

Axiom *classic* :  $\forall P:\text{Prop}, P \vee \neg P$ .

Lemma NNPP :  $\forall p:\text{Prop}, \neg \neg p \rightarrow p$ .

Peirce's law states  $\forall P Q:\text{Prop}, ((P \rightarrow Q) \rightarrow P) \rightarrow P$ . Thanks to  $\forall P, \text{False} \rightarrow P$ , it is equivalent to the following form

Lemma Peirce :  $\forall P:\text{Prop}, ((P \rightarrow \text{False}) \rightarrow P) \rightarrow P$ .

Lemma not\_imply\_elim :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow P$ .

Lemma not\_imply\_elim2 :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow \neg Q$ .

Lemma imply\_to\_or :  $\forall P Q:\text{Prop}, (P \rightarrow Q) \rightarrow \neg P \vee Q$ .

Lemma imply\_to\_and :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow P \wedge \neg Q$ .

Lemma or\_to\_imply :  $\forall P Q:\text{Prop}, \neg P \vee Q \rightarrow P \rightarrow Q$ .

Lemma not\_and\_or :  $\forall P Q:\text{Prop}, \neg (P \wedge Q) \rightarrow \neg P \vee \neg Q$ .

Lemma or\_not\_and :  $\forall P Q:\text{Prop}, \neg P \vee \neg Q \rightarrow \neg (P \wedge Q)$ .

Lemma not\_or\_and :  $\forall P Q:\text{Prop}, \neg (P \vee Q) \rightarrow \neg P \wedge \neg Q$ .

Lemma and\_not\_or :  $\forall P Q:\text{Prop}, \neg P \wedge \neg Q \rightarrow \neg (P \vee Q)$ .

Lemma imply\_and\_or :  $\forall P Q:\text{Prop}, (P \rightarrow Q) \rightarrow P \vee Q \rightarrow Q$ .

Lemma imply\_and\_or2 :  $\forall P Q R:\text{Prop}, (P \rightarrow Q) \rightarrow P \vee R \rightarrow Q \vee R$ .

Lemma proof\_irrelevance :  $\forall (P:\text{Prop}) (p1 p2:P), p1 = p2$ .

Ltac *classical\_right* := match goal with

$\vdash ?X \vee \_ \Rightarrow (\text{elim } (\text{classic } X); \text{intro}; [\text{left}; \text{trivial} | \text{right}])$

end.

Ltac *classical\_left* := match goal with

```

⊢ _ ∨ ?X ⇒ (elim (classic X);intro;[right;trivial|left])
end.

Require Export EqdepFacts.

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.

Module EQDEPTHEORY := EQDEPTHEORY(EQ_RECT_EQ).
Export EqdepTheory.

```

## Chapter 55

### 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* :

$\forall (A:\text{Type}) (B:A \rightarrow \text{Type}) (R:\forall x:A, B\ x \rightarrow \text{Prop}),$   
 $(\forall x:A, \exists! y:B\ x, R\ x\ y) \rightarrow$   
 $(\exists f:(\forall x:A, B\ x), \forall x:A, R\ x\ (f\ x)).$

Unique choice reifies functional relations into functions

Theorem unique\_choice :

$\forall (A\ B:\text{Type}) (R:A \rightarrow B \rightarrow \text{Prop}),$   
 $(\forall x:A, \exists! y:B, R\ x\ y) \rightarrow$   
 $(\exists f:A \rightarrow B, \forall x:A, R\ x\ (f\ x)).$

The following proof comes from [ChicliPottierSimpson02] Require Import **Setoid**.

Theorem classic\_set\_in\_prop\_context :

$\forall C:\text{Prop}, ((\forall P:\text{Prop}, \{P\} + \{\neg P\}) \rightarrow C) \rightarrow C.$

Corollary not\_not\_classic\_set :

$((\forall P:\text{Prop}, \{P\} + \{\neg P\}) \rightarrow \text{False}) \rightarrow \text{False}.$

Notation classic\_set := not\_not\_classic\_set (*only parsing*).

## Chapter 56

# 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 Quarterly, vol 50(3), pp 236-240, 2004.

Require ClassicalFacts ChoiceFacts.

### 56.1 Pred. Ext. + Rel. Axiom of Choice -> Excluded-Middle

Section PredExt\_RelChoice\_imp\_EM.

The axiom of extensionality for predicates

Definition PredicateExtensionality :=

$\forall P Q:\mathbf{bool} \rightarrow \mathbf{Prop}, (\forall b:\mathbf{bool}, P\ b \leftrightarrow Q\ b) \rightarrow P = Q.$

From predicate extensionality we get propositional extensionality hence proof-irrelevance

Import ClassicalFacts.



Variable *pred\_extensionality* : PredicateExtensionality.

Lemma prop\_ext :  $\forall A B:\text{Prop}, (A \leftrightarrow B) \rightarrow A = B$ .

Lemma proof\_irrel :  $\forall (A:\text{Prop}) (a1\ a2:A), a1 = a2$ .

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 :

$\exists R : (\text{bool} \rightarrow \text{Prop}) \rightarrow \text{bool} \rightarrow \text{Prop},$   
 $(\forall P:\text{bool} \rightarrow \text{Prop},$   
 $(\exists b : \text{bool}, P\ b) \rightarrow$   
 $\exists b : \text{bool}, P\ b \wedge R\ P\ b \wedge (\forall b':\text{bool}, R\ P\ b' \rightarrow b = b')).$

The proof of the excluded middle Remark: P could have been in Set or Type

Theorem pred\_ext\_and\_rel\_choice\_imp\_EM :  $\forall P:\text{Prop}, P \vee \neg P$ .

End PredExt\_RelChoice\_imp\_EM.

## 56.2 Proof-Irrel. + Rel. Axiom of Choice $\rightarrow$ 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* :  $\forall P:\text{Prop}, \forall x\ y:P, x=y$ .

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*  $\Rightarrow x=a1 \vee x=a2$ ).

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 :  $\forall x:A', \exists y:\mathbf{bool}$  ,  
 $(\text{projT1 } x = a1 \wedge y = \mathbf{true}) \vee (\text{projT1 } x = a2 \wedge y = \mathbf{false})$ .

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.

### 56.3 Extensional Hilbert's epsilon description operator $\rightarrow$ Excluded-Middle

Proof sketch from Bell [Bell93] (with thanks to P. Castéran)

Section ExtensionalEpsilon\_imp\_EM.

Variable *epsilon* :  $\forall A : \mathbf{Type}, \text{inhabited } A \rightarrow (A \rightarrow \mathbf{Prop}) \rightarrow A$ .

Hypothesis *epsilon\_spec* :

$\forall (A:\mathbf{Type}) (i:\text{inhabited } A) (P:A\rightarrow\mathbf{Prop}),$   
 $(\exists x, P x) \rightarrow P (\text{epsilon } A i P)$ .

Hypothesis *epsilon\_extensionality* :

$\forall (A:\mathbf{Type}) (i:\text{inhabited } A) (P Q:A\rightarrow\mathbf{Prop}),$   
 $(\forall a, P a \leftrightarrow Q a) \rightarrow \text{epsilon } A i P = \text{epsilon } A i Q$ .

Theorem extensional\_epsilon\_imp\_EM :  $\forall P:\mathbf{Prop}, P \vee \neg P$ .

End ExtensionalEpsilon\_imp\_EM.

## Chapter 57

# 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.

### 57.1 0 is right neutral

Lemma minus\_n\_O  $n$  :  $n = n - 0$ .

### 57.2 Permutation with successor

Lemma minus\_Sn\_m  $n m$  :  $m \leq n \rightarrow S (n - m) = S n - m$ .

Theorem pred\_of\_minus  $n$  :  $\text{pred } n = n - 1$ .

### 57.3 Diagonal

Notation minus\_diag := Nat.sub\_diag (*only parsing*).

Lemma minus\_diag\_reverse  $n$  :  $0 = n - n$ .

Notation minus\_n\_n := minus\_diag\_reverse.

## 57.4 Simplification

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

## 57.5 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`  $n\ m : n \leq m \rightarrow m = n + (m - n)$ .

## 57.6 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`  $n\ m : \neg m \leq n \rightarrow n - m = 0$ .

## 57.7 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 58

# 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:  $\forall m n (le\_mn1\ le\_mn2 : m \leq n), le\_mn1 = le\_mn2$ .
  For compatibility Require Import Le Lt.
```

## Chapter 59

# 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 :  $\forall x y, \{x < y\} + \{x \geq y\}$  :=  
fun *n m*  $\Rightarrow$  **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 60

# 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 ≤ m} + {m < n}.
Definition le_le_S_dec n m : {n ≤ m} + {S m ≤ n}.
Definition le_ge_dec n m : {n ≤ m} + {n ≥ m}.
Definition le_gt_dec n m : {n ≤ m} + {n > m}.
Definition le_lt_eq_dec n m : n ≤ m → {n < m} + {n = m}.
Theorem le_dec n m : {n ≤ m} + {¬ n ≤ m}.
Theorem lt_dec n m : {n < m} + {¬ n < m}.
Theorem gt_dec n m : {n > m} + {¬ n > m}.
Theorem ge_dec n m : {n ≥ m} + {¬ n ≥ m}.
```

Proofs of decidability

```
Theorem dec_le n m : decidable (n ≤ m).
Theorem dec_lt n m : decidable (n < m).
Theorem dec_gt n m : decidable (n > m).
Theorem dec_ge n m : decidable (n ≥ m).
Theorem not_eq n m : n ≠ m → n < m ∨ m < n.
Theorem not_le n m : ¬ n ≤ m → n > m.
Theorem not_gt n m : ¬ n > m → n ≤ m.
Theorem not_ge n m : ¬ n ≥ m → 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`  $:= \text{Nat.compare\_succ}$  (*only parsing*).

Lemma `nat_compare_lt`  $n\ m : n < m \leftrightarrow (n\ ?=\ m) = \text{Lt}$ .

Lemma `nat_compare_gt`  $n\ m : n > m \leftrightarrow (n\ ?=\ m) = \text{Gt}$ .

Lemma `nat_compare_le`  $n\ m : n \leq m \leftrightarrow (n\ ?=\ m) \neq \text{Gt}$ .

Lemma `nat_compare_ge`  $n\ m : n \geq m \leftrightarrow (n\ ?=\ m) \neq \text{Lt}$ .

Some projections of the above equivalences.

Lemma `nat_compare_eq`  $n\ m : (n\ ?=\ m) = \text{Eq} \rightarrow n = m$ .

Lemma `nat_compare_lt_lt`  $n\ m : (n\ ?=\ m) = \text{Lt} \rightarrow n < m$ .

Lemma `nat_compare_gt_gt`  $n\ m : (n\ ?=\ m) = \text{Gt} \rightarrow n > m$ .

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`  $(n\ m:\text{nat}) :=$

`match lt_eq_lt_dec n m with`

`| inleft (left _) => Lt`

`| inleft (right _) => Eq`

`| inright _ => Gt`

`end.`

Lemma `nat_compare_equiv`  $n\ m : (n\ ?=\ m) = \text{nat\_compare\_alt } n\ m$ .

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`  $:= \text{Nat.leb}$  (*only parsing*).

Notation `leb_iff`  $:= \text{Nat.leb\_le}$  (*only parsing*).

Lemma `leb_iff_conv`  $m\ n : (n\ <=?\ m) = \text{false} \leftrightarrow m < n$ .

Lemma `leb_correct`  $m\ n : m \leq n \rightarrow (m\ <=?\ n) = \text{true}$ .

Lemma `leb_complete`  $m\ n : (m\ <=?\ n) = \text{true} \rightarrow m \leq n$ .

Lemma `leb_correct_conv`  $m\ n : m < n \rightarrow (n\ <=?\ m) = \text{false}$ .

Lemma `leb_complete_conv`  $m\ n : (n\ <=?\ m) = \text{false} \rightarrow m < n$ .

Lemma `leb_compare`  $n\ m : (n\ <=?\ m) = \text{true} \leftrightarrow (n\ ?=\ m) \neq \text{Gt}$ .

## Chapter 61

# 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* :  $\forall n\ m, n \leq m \rightarrow \text{lt\_or\_eq } n\ m$ .

Lemma *le\_le\_S\_eq* :  $\forall n\ m, n \leq m \rightarrow S\ n \leq m \vee n = m$ .

Lemma *discrete\_nat* :

$\forall n\ m, n < m \rightarrow S\ n = m \vee (\exists r : \mathbf{nat}, m = S\ (S\ (n + r)))$ .

Require Export Wf\_nat.

Require Export Min Max.

## Chapter 62

# 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  $\forall i: \text{nat}, k \leq i < l \rightarrow P i$ .
  Inductive between k :
nat → Prop :=
  | bet_emp : between k k
  | bet_S :  $\forall l, \text{between } k l \rightarrow P l \rightarrow \text{between } k (S l)$ .

  #[local]
  Hint Constructors between: arith.

  Lemma bet_eq :  $\forall k l, l = k \rightarrow \text{between } k l$ .

  #[local]
  Hint Resolve bet_eq: arith.

  Lemma between_le :  $\forall k l, \text{between } k l \rightarrow k \leq l$ .

  #[local]
  Hint Immediate between_le: arith.

  Lemma between_Sk_l :  $\forall k l, \text{between } k l \rightarrow S k \leq l \rightarrow \text{between } (S k) l$ .

  #[local]
  Hint Resolve between_Sk_l: arith.

  Lemma between_restr :
     $\forall k l (m:\text{nat}), k \leq l \rightarrow l \leq m \rightarrow \text{between } k m \rightarrow \text{between } l m$ .

  The exists_between type expresses the concept  $\exists i: \text{nat}, k \leq i < l \wedge Q i$ .
  Inductive exists_between k : nat → Prop :=
  | exists_S :  $\forall l, \text{exists\_between } k l \rightarrow \text{exists\_between } k (S l)$ 
  | exists_le :  $\forall l, k \leq l \rightarrow Q l \rightarrow \text{exists\_between } k (S l)$ .

  #[local]
```

Hint Constructors **exists\_between**: *arith*.

Lemma exists\_le\_S :  $\forall k l, \text{exists\_between } k l \rightarrow S k \leq l$ .

Lemma exists\_lt :  $\forall k l, \text{exists\_between } k l \rightarrow k < l$ .

#[local]

Hint Immediate exists\_le\_S exists\_lt: *arith*.

Lemma exists\_S\_le :  $\forall k l, \text{exists\_between } k (S l) \rightarrow k \leq l$ .

#[local]

Hint Immediate exists\_S\_le: *arith*.

Definition in\_int  $p q r := p \leq r \wedge r < q$ .

Lemma in\_int\_intro :  $\forall p q r, p \leq r \rightarrow r < q \rightarrow \text{in\_int } p q r$ .

#[local]

Hint Resolve in\_int\_intro: *arith*.

Lemma in\_int\_lt :  $\forall p q r, \text{in\_int } p q r \rightarrow p < q$ .

Lemma in\_int\_p\_Sq :

$\forall p q r, \text{in\_int } p (S q) r \rightarrow \text{in\_int } p q r \vee r = q$ .

Lemma in\_int\_S :  $\forall p q r, \text{in\_int } p q r \rightarrow \text{in\_int } p (S q) r$ .

#[local]

Hint Resolve in\_int\_S: *arith*.

Lemma in\_int\_Sp\_q :  $\forall p q r, \text{in\_int } (S p) q r \rightarrow \text{in\_int } p q r$ .

#[local]

Hint Immediate in\_int\_Sp\_q: *arith*.

Lemma between\_in\_int :

$\forall k l, \text{between } k l \rightarrow \forall r, \text{in\_int } k l r \rightarrow P r$ .

Lemma in\_int\_between :

$\forall k l, k \leq l \rightarrow (\forall r, \text{in\_int } k l r \rightarrow P r) \rightarrow \text{between } k l$ .

Lemma exists\_in\_int :

$\forall k l, \text{exists\_between } k l \rightarrow \text{exists2 } m : \text{nat}, \text{in\_int } k l m \ \& \ Q m$ .

Lemma in\_int\_exists :  $\forall k l r, \text{in\_int } k l r \rightarrow Q r \rightarrow \text{exists\_between } k l$ .

Lemma between\_or\_exists :

$\forall k l,$   
 $k \leq l \rightarrow$   
 $(\forall n:\text{nat}, \text{in\_int } k l n \rightarrow P n \vee Q n) \rightarrow$   
 $\text{between } k l \vee \text{exists\_between } k l$ .

Lemma between\_not\_exists :

$\forall k l,$   
 $\text{between } k l \rightarrow$   
 $(\forall n:\text{nat}, \text{in\_int } k l n \rightarrow P n \rightarrow \neg Q n) \rightarrow \neg \text{exists\_between } k l$ .

Inductive **P\_nth** (*init*:nat) : nat → nat → Prop :=

| nth\_O : **P\_nth** *init* *init* 0

| nth\_S :

$\forall k l (n:\text{nat}),$

```

      P_nth init  $k\ n \rightarrow \text{between } (S\ k)\ l \rightarrow Q\ l \rightarrow \text{P\_nth } \textit{init}\ l\ (S\ n)$ .
Lemma nth_le :  $\forall\ (init:\mathbf{nat})\ l\ (n:\mathbf{nat}),\ \text{P\_nth } \textit{init}\ l\ n \rightarrow init \leq l$ .
Definition eventually ( $n:\mathbf{nat}$ ) := exists2  $k : \mathbf{nat},\ k \leq n \ \&\ Q\ k$ .
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 63

# Library Coq.Arith.Mult

### 63.1 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*.

### 63.2 *nat* is a semi-ring

#### 63.2.1 Zero property

Notation *mult\_0\_l* := *Nat.mul\_0\_l* (*only parsing*). Notation *mult\_0\_r* := *Nat.mul\_0\_r* (*only parsing*).

#### 63.2.2 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*.

#### 63.2.3 Commutativity

Notation *mult\_comm* := *Nat.mul\_comm* (*only parsing*).

*#[global]*

Hint Resolve *mult\_comm*: *arith*.

### 63.2.4 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*.

### 63.2.5 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*.

### 63.2.6 Inversion lemmas

Lemma `mult_is_O`  $n\ m : n \times m = 0 \rightarrow n = 0 \vee m = 0$ .

Lemma `mult_is_one`  $n\ m : n \times m = 1 \rightarrow n = 1 \wedge m = 1$ .

### 63.2.7 Multiplication and successor

Notation `mult_succ_l` := `Nat.mul_succ_l` (*only parsing*). Notation `mult_succ_r` := `Nat.mul_succ_r` (*only parsing*).

## 63.3 Compatibility with orders

Lemma `mult_O_le`  $n\ m : m = 0 \vee n \leq m \times n$ .

`#[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*  $n\ m\ p\ q : n \leq m \rightarrow p \leq q \rightarrow n \times p \leq m \times q$ .

Lemma *mult\_S\_lt\_compat\_l*  $n\ m\ p : m < p \rightarrow S\ n \times m < S\ n \times p$ .

#[*global*]

Hint Resolve *mult\_S\_lt\_compat\_l*: *arith*.

Lemma *mult\_lt\_compat\_l*  $n\ m\ p : n < m \rightarrow 0 < p \rightarrow p \times n < p \times m$ .

Lemma *mult\_lt\_compat\_r*  $n\ m\ p : n < m \rightarrow 0 < p \rightarrow n \times p < m \times p$ .

Lemma *mult\_S\_le\_reg\_l*  $n\ m\ p : S\ n \times m \leq S\ n \times p \rightarrow m \leq p$ .

## 63.4 $n|->2*n$ and $n|->2n+1$ have disjoint image

Theorem *odd\_even\_lem*  $p\ q : 2 \times p + 1 \neq 2 \times q$ .

## 63.5 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
  | 0 => s
  | S p => mult_acc (tail_plus m s) m p
  end.
```

Lemma *mult\_acc\_aux* :  $\forall\ n\ m\ p, m + n \times p = \text{mult\_acc}\ m\ p\ n$ .

Definition *tail\_mult*  $n\ m := \text{mult\_acc}\ 0\ m\ n$ .

Lemma *mult\_tail\_mult* :  $\forall\ n\ m, n \times m = \text{tail\_mult}\ n\ m$ .

*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 64

# 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 :  $\forall q\ r, b > r \rightarrow a = q \times b + r \rightarrow \text{diveucl } a\ b.$ 

Lemma eucl_dev :  $\forall n, n > 0 \rightarrow \forall m:\text{nat}, \text{diveucl } m\ n.$ 

Lemma quotient :
   $\forall n,$ 
   $n > 0 \rightarrow$ 
   $\forall m:\text{nat}, \{q : \text{nat} \mid \exists r : \text{nat}, m = q \times n + r \wedge n > r\}.$ 

Lemma modulo :
   $\forall n,$ 
   $n > 0 \rightarrow$ 
   $\forall m:\text{nat}, \{r : \text{nat} \mid \exists q : \text{nat}, m = q \times n + r \wedge n > r\}.$ 
```

## Chapter 65

# 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 :

$\forall P:\text{nat} \rightarrow \text{Prop},$

$P\ 0 \rightarrow P\ 1 \rightarrow (\forall n, P\ n \rightarrow P\ (S\ (S\ n))) \rightarrow \forall n, P\ n.$

$0 < n \Rightarrow n/2 < n$

Lemma lt\_div2 n :  $0 < n \rightarrow \text{div2}\ n < n.$

#[global]

Hint Resolve lt\_div2: arith.

Properties related to the parity

Lemma even\_div2 n : **even** n  $\rightarrow \text{div2}\ n = \text{div2}\ (S\ n).$

Lemma odd\_div2 n : **odd** n  $\rightarrow S\ (\text{div2}\ n) = \text{div2}\ (S\ n).$

Lemma div2\_even n :  $\text{div2}\ n = \text{div2}\ (S\ n) \rightarrow \text{even}\ n.$

Lemma div2\_odd n :  $S\ (\text{div2}\ n) = \text{div2}\ (S\ n) \rightarrow \text{odd}\ n.$

#[global]

Hint Resolve even\_div2 div2\_even odd\_div2 div2\_odd: arith.

Lemma even\_odd\_div2 n :

$(\text{even}\ n \leftrightarrow \text{div2}\ n = \text{div2}\ (S\ n)) \wedge$

$(\text{odd}\ n \leftrightarrow S\ (\text{div2}\ n) = \text{div2}\ (S\ n)).$

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**  $n \leftrightarrow n = \text{double } (\text{div2 } n)$ )  $\wedge$  (**odd**  $n \leftrightarrow n = S (\text{double } (\text{div2 } n))$ ).

Specializations

Lemma `even_double n` : **even**  $n \rightarrow n = \text{double } (\text{div2 } n)$ .

Lemma `double_even n` :  $n = \text{double } (\text{div2 } n) \rightarrow$  **even**  $n$ .

Lemma `odd_double n` : **odd**  $n \rightarrow n = S (\text{double } (\text{div2 } n))$ .

Lemma `double_odd n` :  $n = S (\text{double } (\text{div2 } n)) \rightarrow$  **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` :  $\forall n, \text{even } n \rightarrow \{p : \text{nat} \mid n = \text{double } p\}$ .

Lemma `odd_S2n` :  $\forall n, \text{odd } n \rightarrow \{p : \text{nat} \mid n = S (\text{double } p)\}$ .

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 66

# 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* : nat\_scope.

Require Import PeanoNat.

Local Open Scope *nat\_scope*.

### 66.1 Irreflexivity

Notation *lt\_irrefl* := Nat.lt\_irrefl (*only parsing*).

*#[global]*

Hint Resolve *lt\_irrefl*: *arith*.

### 66.2 Relationship between *le* and *lt*

Theorem *lt\_le\_S* *n m* : *n* < *m* → S *n* ≤ *m*.

Theorem *lt\_n\_Sm\_le* *n m* : *n* < S *m* → *n* ≤ *m*.

Theorem *le\_lt\_n\_Sm* *n m* : *n* ≤ *m* → *n* < 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* ≤ *m* → ¬ *m* < *n*.

Theorem *lt\_not\_le* *n m* : *n* < *m* → ¬ *m* ≤ *n*.

#[global]  
 Hint Immediate *le\_not\_lt lt\_not\_le*: *arith*.

## 66.3 Asymmetry

Notation *lt\_asym* := *Nat.lt\_asymm* (*only parsing*).

## 66.4 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*.

## 66.5 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*.

## 66.6 Predecessor

Lemma *S\_pred*  $n\ m : m < n \rightarrow n = S\ (\text{pred } n)$ .

Lemma *S\_pred\_pos*  $n : 0 < n \rightarrow n = S\ (\text{pred } n)$ .

Lemma *lt\_pred*  $n\ m : S\ n < m \rightarrow n < \text{pred } m$ .

Lemma *lt\_pred\_n\_n*  $n : 0 < n \rightarrow \text{pred } n < n$ .

#[global]  
 Hint Immediate *lt\_pred*: *arith*.  
 #[global]  
 Hint Resolve *lt\_pred\_n\_n*: *arith*.

## 66.7 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*.

## 66.8 Large = strict or equal

Notation `le_lt_or_eq_iff` := `Nat.lt_eq_cases` (*only parsing*).

Theorem `le_lt_or_eq n m` :  $n \leq m \rightarrow n < m \vee n = m$ .

Notation `lt_le_weak` := `Nat.lt_le_incl` (*only parsing*).

*#[global]*

Hint Immediate `lt_le_weak`: *arith*.

## 66.9 Dichotomy

Notation `le_or_lt` := `Nat.le_gt_cases` (*only parsing*).

Theorem `nat_total_order n m` :  $n \neq m \rightarrow n < m \vee m < n$ .

For compatibility, we “Require” the same files as before  
Require Import Le.

## Chapter 67

# 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
  | 0 => m
  | S p => S (p + m)
  end
where "n + m" := (add n m) : nat_scope.
```

Require Import PeanoNat.

Local Open Scope *nat\_scope*.

### 67.1 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* :  $n + m + p = n + (m + p)$ .

### 67.2 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* :  $p + n < p + m \rightarrow n < m$ .

## 67.3 Compatibility with order

Lemma `plus_le_compat_l`  $n\ m\ p : n \leq m \rightarrow p + n \leq p + m$ .

Lemma `plus_le_compat_r`  $n\ m\ p : n \leq m \rightarrow n + p \leq m + p$ .

Lemma `plus_lt_compat_l`  $n\ m\ p : n < m \rightarrow p + n < p + m$ .

Lemma `plus_lt_compat_r`  $n\ m\ p : n < m \rightarrow n + p < m + p$ .

Lemma `plus_le_compat`  $n\ m\ p\ q : n \leq m \rightarrow p \leq q \rightarrow n + p \leq m + q$ .

Lemma `plus_le_lt_compat`  $n\ m\ p\ q : n \leq m \rightarrow p < q \rightarrow n + p < m + q$ .

Lemma `plus_lt_le_compat`  $n\ m\ p\ q : n < m \rightarrow p \leq q \rightarrow n + p < m + q$ .

Lemma `plus_lt_compat`  $n\ m\ p\ q : n < m \rightarrow p < q \rightarrow n + p < m + q$ .

Lemma `le_plus_l`  $n\ m : n \leq n + m$ .

Lemma `le_plus_r`  $n\ m : m \leq n + m$ .

Theorem `le_plus_trans`  $n\ m\ p : n \leq m \rightarrow n \leq m + p$ .

Theorem `lt_plus_trans`  $n\ m\ p : n < m \rightarrow n < m + p$ .

## 67.4 Inversion lemmas

Lemma `plus_is_O`  $n\ m : n + m = 0 \rightarrow n = 0 \wedge m = 0$ .

Definition `plus_is_one`  $m\ n :$

$m + n = 1 \rightarrow \{m = 0 \wedge n = 1\} + \{m = 1 \wedge n = 0\}$ .

## 67.5 Derived properties

Notation `plus_permute_2_in_4`  $:= \text{Nat.add\_shuffle1}$  (*only parsing*).

## 67.6 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`  $n\ m : \mathbf{nat} :=$

```

  match  $n$  with
  |  $O \Rightarrow m$ 
  |  $S\ n \Rightarrow \text{tail\_plus } n\ (S\ m)$ 
  end.
```

Lemma `plus_tail_plus`  $: \forall\ n\ m, n + m = \text{tail\_plus } n\ m$ .



## 67.7 Discrimination

Lemma `succ_plus_discr`  $n\ m : n \neq S\ (m+n)$ .

Lemma `n_SS`  $n : n \neq S\ (S\ n)$ .

Lemma `n_SSS`  $n : n \neq S\ (S\ (S\ n))$ .

Lemma `n_SSSS`  $n : n \neq S\ (S\ (S\ (S\ n)))$ .

## 67.8 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 68

# 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  $f$  :  $A \rightarrow$  **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

|  $O \rightarrow$  *error*

|  $S\ m \rightarrow F\ k\ (indrec\ m)$

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 :

$\forall P:A \rightarrow$  Type,

$(\forall x:A, (\forall y:A, ltof\ y\ x \rightarrow P\ y) \rightarrow P\ x) \rightarrow \forall a:A, P\ a.$

Theorem induction\_gtof1 :

$\forall P:A \rightarrow \text{Type},$   
 $(\forall x:A, (\forall y:A, \text{gtof } y \ x \rightarrow P \ y) \rightarrow P \ x) \rightarrow \forall a:A, P \ a.$

Theorem induction\_ltof2 :

$\forall P:A \rightarrow \text{Type},$   
 $(\forall x:A, (\forall y:A, \text{ltof } y \ x \rightarrow P \ y) \rightarrow P \ x) \rightarrow \forall a:A, P \ a.$

Theorem induction\_gtof2 :

$\forall P:A \rightarrow \text{Type},$   
 $(\forall x:A, (\forall y:A, \text{gtof } y \ x \rightarrow P \ y) \rightarrow P \ x) \rightarrow \forall a:A, P \ a.$

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  $R : A \rightarrow A \rightarrow \text{Prop}.$

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 :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Type}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma lt\_wf\_rect :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Type}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma lt\_wf\_rec1 :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Set}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma lt\_wf\_rec :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Set}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma lt\_wf\_ind :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Prop}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma gt\_wf\_rect :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Type}), (\forall n, (\forall m, n > m \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma gt\_wf\_rec :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Set}), (\forall n, (\forall m, n > m \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma gt\_wf\_ind :

$\forall n \ (P:\mathbf{nat} \rightarrow \text{Prop}), (\forall n, (\forall m, n > m \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

Lemma lt\_wf\_double\_rect :

$\forall P:\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \text{Type},$   
 $(\forall n \ m,$   
 $(\forall p \ q, p < n \rightarrow P \ p \ q) \rightarrow$   
 $(\forall p, p < m \rightarrow P \ n \ p) \rightarrow P \ n \ m) \rightarrow \forall n \ m, P \ n \ m.$

Lemma lt\_wf\_double\_rec :

$\forall P:\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \text{Set},$   
 $(\forall n \ m,$   
 $(\forall p \ q, p < n \rightarrow P \ p \ q) \rightarrow$

$(\forall p, p < m \rightarrow P \ n \ p) \rightarrow P \ n \ m) \rightarrow \forall n \ m, P \ n \ m.$

Lemma lt\_wf\_double\_ind :  
 $\forall P:\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \text{Prop},$   
 $(\forall n \ m,$   
 $(\forall p \ (q:\mathbf{nat}), p < n \rightarrow P \ p \ q) \rightarrow$   
 $(\forall p, p < m \rightarrow P \ n \ p) \rightarrow P \ n \ m) \rightarrow \forall n \ m, P \ n \ m.$

#[global]  
 Hint Resolve lt\_wf: arith.  
 #[global]  
 Hint Resolve well\_founded\_lt\_compat: arith.

Section LT\_WF\_REL.  
 Variable  $A : \text{Set}.$   
 Variable  $R : A \rightarrow A \rightarrow \text{Prop}.$   
 Variable  $F : A \rightarrow \mathbf{nat} \rightarrow \text{Prop}.$   
 Definition inv\_lt\_rel  $x \ y := \text{exists2 } n, F \ x \ n \ \& \ (\forall m, F \ y \ m \rightarrow n < m).$   
 Hypothesis  $F\_compat : \forall x \ y:A, R \ x \ y \rightarrow \text{inv\_lt\_rel } x \ y.$   
 Remark acc\_lt\_rel :  $\forall x:A, (\exists n, F \ x \ n) \rightarrow \mathbf{Acc} \ R \ x.$   
 Theorem well\_founded\_inv\_lt\_rel\_compat : well\_founded  $R.$

End LT\_WF\_REL.

Lemma well\_founded\_inv\_rel\_inv\_lt\_rel  $(A:\text{Set}) \ (F:A \rightarrow \mathbf{nat} \rightarrow \text{Prop}) :$   
 well\_founded  $(\text{inv\_lt\_rel } A \ F).$

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  $(A:\text{Type}) \ (R:A \rightarrow A \rightarrow \text{Prop}) \ (P:A \rightarrow \text{Prop}) :=$   
 $\exists! x, P \ x \wedge \forall x', P \ x' \rightarrow R \ x \ x'.$

Lemma dec\_inh\_nat\_subset\_has\_unique\_least\_element :  
 $\forall P:\mathbf{nat} \rightarrow \text{Prop}, (\forall n, P \ n \vee \neg P \ n) \rightarrow$   
 $(\exists n, P \ n) \rightarrow \text{has\_unique\_least\_element } \mathbf{le} \ P.$

Unset Implicit Arguments.

Notation iter\_nat  $n \ A \ f \ x := (\text{nat\_rect } (\text{fun } _ \Rightarrow A) \ x \ (\text{fun } _ \Rightarrow f) \ n) \ (\text{only parsing}).$

## Chapter 69

# 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 70

# 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*.

### 70.1 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* → *S n* > *S m*.

Lemma *gt\_S\_n* *n m* : *S m* > *S n* → *m* > *n*.

Theorem *gt\_S* *n m* : *S n* > *m* → *n* > *m* ∨ *m* = *n*.

Lemma *gt\_pred* *n m* : *m* > *S n* → *pred m* > *n*.

### 70.2 Irreflexivity

Lemma *gt\_irrefl* *n* : ¬ *n* > *n*.

### 70.3 Asymmetry

Lemma *gt\_asym* *n m* : *n* > *m* → ¬ *m* > *n*.

### 70.4 Relating strict and large orders

Lemma *le\_not\_gt* *n m* : *n* ≤ *m* → ¬ *n* > *m*.

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

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

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

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

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

## 70.5 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`  $n\ m\ p : \mathbb{S}\ n > m \rightarrow m > p \rightarrow n > p$ .

## 70.6 Comparison to 0

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

## 70.7 Simplification and compatibility

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

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

## 70.8 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 71

# Library Coq.Arith.Factorial

```
Require Import PeanoNat Plus Mult Lt.
```

```
Local Open Scope nat_scope.
```

```
Factorial
```

```
Fixpoint fact (n:nat) : nat :=
```

```
  match n with
```

```
    | 0 => 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 72

# Library Coq.Arith.EqNat

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

Equality on natural numbers

### 72.1 Propositional equality

```
Fixpoint eq_nat n m : Prop :=  
  match n, m with  
  | O, O  $\Rightarrow$  True  
  | O, S _  $\Rightarrow$  False  
  | S _, O  $\Rightarrow$  False  
  | S n1, S m1  $\Rightarrow$  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  $n$   $m$   $\leftrightarrow$   $n = m$ .

Lemma eq\_eq\_nat  $n$   $m$  :  $n = m \rightarrow$  eq\_nat  $n$   $m$ .

Lemma eq\_nat\_eq  $n$   $m$  : eq\_nat  $n$   $m \rightarrow n = m$ .

*#[global]*

Hint Immediate eq\_eq\_nat eq\_nat\_eq: arith.

Theorem eq\_nat\_elim :

$\forall n (P:\mathbf{nat} \rightarrow \mathbf{Prop}), P\ n \rightarrow \forall m, \text{eq\_nat } n\ m \rightarrow P\ m$ .

Theorem eq\_nat\_decide :  $\forall n\ m, \{\text{eq\_nat } n\ m\} + \{\neg \text{eq\_nat } n\ m\}$ .

## 72.2 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`  $\rightarrow n=m$ .

Lemma `beq_nat_false`  $n$   $m$  :  $(n =? m)$  = `false`  $\rightarrow n \neq m$ .

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

Definition `beq_nat_eq` :  $\forall n\ m, \text{true} = (n =? m) \rightarrow n = m$ .

## Chapter 73

# 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.
```

### 73.1 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.
```

### 73.2 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.
```

### 73.3 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) → even n ∧ even m ∨ odd n ∧ odd m.

Lemma odd_plus_split n m :
  odd (n + m) → odd n ∧ even m ∨ even n ∧ odd m.

Lemma even_even_plus n m : even n → even m → even (n + m).

Lemma odd_plus_l n m : odd n → even m → odd (n + m).

Lemma odd_plus_r n m : even n → odd m → odd (n + m).

Lemma odd_even_plus n m : odd n → odd m → even (n + m).

Lemma even_plus_aux n m :
  (odd (n + m) ↔ odd n ∧ even m ∨ even n ∧ odd m) ∧
  (even (n + m) ↔ even n ∧ even m ∨ odd n ∧ odd m).

Lemma even_plus_even_inv_r n m : even (n + m) → even n → even m.

Lemma even_plus_even_inv_l n m : even (n + m) → even m → even n.

Lemma even_plus_odd_inv_r n m : even (n + m) → odd n → odd m.

Lemma even_plus_odd_inv_l n m : even (n + m) → odd m → odd n.

Lemma odd_plus_even_inv_l n m : odd (n + m) → odd m → even n.

Lemma odd_plus_even_inv_r n m : odd (n + m) → odd n → even m.

Lemma odd_plus_odd_inv_l n m : odd (n + m) → even m → odd n.

Lemma odd_plus_odd_inv_r n m : odd (n + m) → even n → odd m.

```

### 73.4 Facts about *even* and *odd* wrt. *mult*

```

Lemma even_mult_aux n m :
  (odd (n × m) ↔ odd n ∧ odd m) ∧ (even (n × m) ↔ even n ∨ even m).

Lemma even_mult_l n m : even n → even (n × m).

Lemma even_mult_r n m : even m → even (n × m).

Lemma even_mult_inv_r n m : even (n × m) → odd n → even m.

Lemma even_mult_inv_l n m : even (n × m) → odd m → even n.

Lemma odd_mult n m : odd n → odd m → odd (n × 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 74

# Library Coq.Arith.Arith

```
Require Export Arith_base.  
Require Export ArithRing.
```

## Chapter 75

# 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.
```

### 75.1 *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.
```

### 75.2 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 ≤ 0 → 0 = n.
```

### 75.3 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*.

## 75.4 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*.

## 75.5 A different elimination principle for the order on natural numbers

Lemma `le_elim_rel` :

$\forall P:\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{Prop},$

$(\forall p, P\ 0\ p) \rightarrow$

$(\forall p\ (q:\mathbf{nat}), p \leq q \rightarrow P\ p\ q \rightarrow P\ (S\ p)\ (S\ q)) \rightarrow$

$\forall n\ m, n \leq m \rightarrow P\ n\ m.$



## Chapter 76

# 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 77

# Library Coq.Arith.PeanoNat

Require Import **N**Axioms **N**Properties OrdersFacts.

Implementation of *NAxiomsSig* by *nat*

Module NAT

<: **N**AXIOMSIG

<: **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 :

$\forall A : \mathbf{nat} \rightarrow \mathbf{Prop}, \mathbf{Proper} \text{ (eq==>iff) } A \rightarrow$

$A \ 0 \rightarrow (\forall n : \mathbf{nat}, A \ n \leftrightarrow A \ (S \ n)) \rightarrow \forall n : \mathbf{nat}, A \ n.$

Recursion function

Definition recursion {A} : A  $\rightarrow$  (nat  $\rightarrow$  A  $\rightarrow$  A)  $\rightarrow$  nat  $\rightarrow$  A :=

nat\_rect (fun \_  $\Rightarrow$  A).

Instance recursion\_wd {A} (Aeq : **relation** A) :  
**Proper** (Aeq ==> (**eq**==>Aeq==>Aeq) ==> **eq** ==> Aeq) recursion.

Theorem recursion\_0 :  
 $\forall \{A\} (a : A) (f : \mathbf{nat} \rightarrow A \rightarrow A), \text{recursion } a \ f \ 0 = a.$

Theorem recursion\_succ :  
 $\forall \{A\} (Aeq : \mathbf{relation} \ A) (a : A) (f : \mathbf{nat} \rightarrow A \rightarrow A),$   
 $Aeq \ a \ a \rightarrow \mathbf{Proper} \ (\mathbf{eq}==>Aeq==>Aeq) \ f \rightarrow$   
 $\forall n : \mathbf{nat}, Aeq \ (\text{recursion } a \ f \ (\mathbf{S} \ n)) \ (f \ n \ (\text{recursion } a \ f \ n)).$

### 77.0.1 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.

### 77.0.2 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  $\times$  n = 0.

Lemma mul\_succ\_l n m : S n  $\times$  m = n  $\times$  m + m.

Lemma lt\_succ\_r n m : n < S m  $\leftrightarrow$  n  $\leq$  m.

### 77.0.3 Boolean comparisons

Lemma eqb\_eq n m : eqb n m = true  $\leftrightarrow$  n = m.

Lemma leb\_le n m : (n <=? m) = true  $\leftrightarrow$  n  $\leq$  m.

Lemma ltb\_lt n m : (n <? m) = true  $\leftrightarrow$  n < m.

### 77.0.4 Decidability of equality over nat.

Lemma eq\_dec :  $\forall n \ m : \mathbf{nat}, \{n = m\} + \{n \neq m\}.$

### 77.0.5 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*  $n\ m : (n\ ?=\ m) = \text{Eq} \leftrightarrow n = m$ .

Lemma *compare\_lt\_iff*  $n\ m : (n\ ?=\ m) = \text{Lt} \leftrightarrow n < m$ .

Lemma *compare\_le\_iff*  $n\ m : (n\ ?=\ m) \neq \text{Gt} \leftrightarrow n \leq m$ .

Lemma *compare\_antisym*  $n\ m : (m\ ?=\ n) = \text{CompOpp}\ (n\ ?=\ m)$ .

Lemma *compare\_succ*  $n\ m : (\text{S}\ n\ ?=\ \text{S}\ m) = (n\ ?=\ m)$ .

### 77.0.6 Minimum, maximum

Lemma *max\_l*  $\forall\ n\ m, m \leq n \rightarrow \text{max}\ n\ m = n$ .

Lemma *max\_r*  $\forall\ n\ m, n \leq m \rightarrow \text{max}\ n\ m = m$ .

Lemma *min\_l*  $\forall\ n\ m, n \leq m \rightarrow \text{min}\ n\ m = n$ .

Lemma *min\_r*  $\forall\ n\ m, m \leq n \rightarrow \text{min}\ n\ m = m$ .

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**.

### 77.0.7 Power

Lemma *pow\_neg\_r*  $a\ b : b < 0 \rightarrow a^b = 0$ .

Lemma *pow\_0\_r*  $a : a^0 = 1$ .

Lemma *pow\_succ\_r*  $a\ b : 0 \leq b \rightarrow a^{(\text{S}\ b)} = a \times a^b$ .

### 77.0.8 Square

Lemma *square\_spec*  $n : \text{square}\ n = n \times n$ .

### 77.0.9 Parity

Definition *Even*  $n := \exists\ m, n = 2 \times m$ .

Definition *Odd*  $n := \exists\ m, n = 2 \times m + 1$ .

Module **PRIVATE\_PARITY**.

Lemma *Even\_1*  $\neg \text{Even}\ 1$ .

Lemma *Even\_2*  $n : \text{Even}\ n \leftrightarrow \text{Even}\ (\text{S}\ (\text{S}\ n))$ .

Lemma Odd\_0 :  $\neg \text{Odd } 0$ .  
 Lemma Odd\_2 n :  $\text{Odd } n \leftrightarrow \text{Odd } (\text{S } (\text{S } n))$ .  
 End PRIVATE\_PARITY.  
 Import Private\_Parity.  
 Lemma even\_spec :  $\forall n, \text{even } n = \text{true} \leftrightarrow \text{Even } n$ .  
 Lemma odd\_spec :  $\forall n, \text{odd } n = \text{true} \leftrightarrow \text{Odd } n$ .

### 77.0.10 Division

Lemma divmod\_spec :  $\forall x \ y \ q \ u, u \leq y \rightarrow$   
   let  $(q', u') := \text{divmod } x \ y \ q \ u$  in  
    $x + (\text{S } y) * q + (y - u) = (\text{S } y) * q' + (y - u') \wedge u' \leq y$ .  
 Lemma div\_mod x y :  $y \neq 0 \rightarrow x = y * (x / y) + x \bmod y$ .  
 Lemma mod\_bound\_pos x y :  $0 \leq x \rightarrow 0 < y \rightarrow 0 \leq x \bmod y < y$ .

### 77.0.11 Square root

Lemma sqrt\_iter\_spec :  $\forall k \ p \ q \ r,$   
    $q = p + p \rightarrow r \leq q \rightarrow$   
   let  $s := \text{sqrt\_iter } k \ p \ q \ r$  in  
    $s * s \leq k + p * p + (q - r) < (\text{S } s) * (\text{S } s)$ .  
 Lemma sqrt\_specif n :  $(\text{sqrt } n) * (\text{sqrt } n) \leq n < \text{S } (\text{sqrt } n) * \text{S } (\text{sqrt } n)$ .  
 Definition sqrt\_spec a (Ha :  $0 \leq a$ ) := sqrt\_specif a.  
 Lemma sqrt\_neg a :  $a < 0 \rightarrow \text{sqrt } a = 0$ .

### 77.0.12 Logarithm

Lemma log2\_iter\_spec :  $\forall k \ p \ q \ r,$   
    $2^{\text{S } p} = q + \text{S } r \rightarrow r < 2^p \rightarrow$   
   let  $s := \text{log2\_iter } k \ p \ q \ r$  in  
    $2^s \leq k + q < 2^{\text{S } s}$ .  
 Lemma log2\_spec n :  $0 < n \rightarrow$   
    $2^{\text{log2 } n} \leq n < 2^{\text{S } (\text{log2 } n)}$ .  
 Lemma log2\_nonpos n :  $n \leq 0 \rightarrow \text{log2 } n = 0$ .

### 77.0.13 Gcd

Definition divide x y :=  $\exists z, y = z * x$ .  
 Notation " $x \mid y$ " := (divide x y) (at level 0) : nat\_scope.  
 Lemma gcd\_divide :  $\forall a \ b, (\text{gcd } a \ b \mid a) \wedge (\text{gcd } a \ b \mid b)$ .  
 Lemma gcd\_divide\_l :  $\forall a \ b, (\text{gcd } a \ b \mid a)$ .

Lemma gcd\_divide\_r :  $\forall a b, (\text{gcd } a b \mid b)$ .

Lemma gcd\_greatest :  $\forall a b c, (c \mid a) \rightarrow (c \mid b) \rightarrow (c \mid \text{gcd } a b)$ .

Lemma gcd\_nonneg a b :  $0 \leq \text{gcd } a b$ .

### 77.0.14 Bitwise operations

Lemma div2\_double n :  $\text{div2 } (2 \times n) = n$ .

Lemma div2\_succ\_double n :  $\text{div2 } (S (2 \times n)) = n$ .

Lemma le\_div2 n :  $\text{div2 } (S n) \leq n$ .

Lemma lt\_div2 n :  $0 < n \rightarrow \text{div2 } n < n$ .

Lemma div2\_decr a n :  $a \leq S n \rightarrow \text{div2 } a \leq n$ .

Lemma double\_twice :  $\forall n, \text{double } n = 2 \times n$ .

Lemma testbit\_0\_l :  $\forall n, \text{testbit } 0 n = \text{false}$ .

Lemma testbit\_odd\_0 a :  $\text{testbit } (2 \times a + 1) 0 = \text{true}$ .

Lemma testbit\_even\_0 a :  $\text{testbit } (2 \times a) 0 = \text{false}$ .

Lemma testbit\_odd\_succ' a n :  $\text{testbit } (2 \times a + 1) (S n) = \text{testbit } a n$ .

Lemma testbit\_even\_succ' a n :  $\text{testbit } (2 \times a) (S n) = \text{testbit } a n$ .

Lemma shiftr\_specif :  $\forall a n m,$   
 $\text{testbit } (\text{shiftr } a n) m = \text{testbit } a (m + n)$ .

Lemma shiftl\_specif\_high :  $\forall a n m, n \leq m \rightarrow$   
 $\text{testbit } (\text{shiftl } a n) m = \text{testbit } a (m - n)$ .

Lemma shiftl\_spec\_low :  $\forall a n m, m < n \rightarrow$   
 $\text{testbit } (\text{shiftl } a n) m = \text{false}$ .

Lemma div2\_bitwise :  $\forall op n a b,$   
 $\text{div2 } (\text{bitwise } op (S n) a b) = \text{bitwise } op n (\text{div2 } a) (\text{div2 } b)$ .

Lemma odd\_bitwise :  $\forall op n a b,$   
 $\text{odd } (\text{bitwise } op (S n) a b) = op (\text{odd } a) (\text{odd } b)$ .

Lemma testbit\_bitwise\_1 :  $\forall op, (\forall b, op \text{ false } b = \text{false}) \rightarrow$   
 $\forall n m a b, a \leq n \rightarrow$   
 $\text{testbit } (\text{bitwise } op n a b) m = op (\text{testbit } a m) (\text{testbit } b m)$ .

Lemma testbit\_bitwise\_2 :  $\forall op, op \text{ false } \text{false} = \text{false} \rightarrow$   
 $\forall n m a b, a \leq n \rightarrow b \leq n \rightarrow$   
 $\text{testbit } (\text{bitwise } op n a b) m = op (\text{testbit } a m) (\text{testbit } b m)$ .

Lemma land\_spec a b n :  
 $\text{testbit } (\text{land } a b) n = \text{testbit } a n \ \&\& \ \text{testbit } b n$ .

Lemma ldiff\_spec a b n :  
 $\text{testbit } (\text{ldiff } a b) n = \text{testbit } a n \ \&\& \ \text{negb } (\text{testbit } b n)$ .

Lemma lor\_spec a b n :  
 $\text{testbit } (\text{lor } a b) n = \text{testbit } a n \ || \ \text{testbit } b n$ .

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  $a\ n\ (\_ : 0 \leq n)$  := testbit\_odd\_succ'  $a\ n$ .

Definition testbit\_even\_succ  $a\ n\ (\_ : 0 \leq n)$  := testbit\_even\_succ'  $a\ n$ .

Lemma testbit\_neg\_r  $a\ n\ (H : n < 0)$  : testbit  $a\ n$  = false.

Definition shiftl\_spec\_high  $a\ n\ m\ (\_ : 0 \leq m)$  := shiftl\_specif\_high  $a\ n\ m$ .

Definition shiftr\_spec  $a\ n\ m\ (\_ : 0 \leq m)$  := shiftr\_specif  $a\ n\ m$ .

Properties of advanced functions (pow, sqrt, log2, ...)

Include **NEXTRA**PROP.

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  $r\ n\ m$  =  $r + n \times m$ .

Lemma tail\_mul\_spec  $n\ m$  : tail\_mul  $n\ m$  =  $n \times m$ .

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 " $\leq?$ " := 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  $1 \leq 2$  or  $x \leq x + x$ , but rather things like  $x \leq y \rightarrow y \leq x \rightarrow x = y$ .

Section TestOrder.

Let *test* :  $\forall\ x\ y,\ x \leq y \rightarrow y \leq x \rightarrow x = y$ .

End TestOrder.

## Chapter 78

# 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.*