

# The Coq Proof Assistant

## The standard library

July 15, 2017

Version 8.6<sup>1</sup>

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

---

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

V8.6, July 15, 2017

©INRIA 1999-2004 (CoQ versions 7.x)

©INRIA 2004-2016 (CoQ versions 8.x)

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.Arith.PeanoNat**

**Require Import** NAXioms NProperties OrdersFacts.

Implementation of *NAXiomsSig* by *nat*

**Module** NAT

<: NAXIOMSIG

<: 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 :

$\forall A : \text{nat} \rightarrow \text{Prop}, \text{Proper } (eq ==> iff) A \rightarrow$   
 $A\ 0 \rightarrow (\forall n : \text{nat}, A\ n \leftrightarrow A\ (S\ n)) \rightarrow \forall n : \text{nat}, A\ n.$

Recursion fonction

**Definition** recursion {A} : A  $\rightarrow$  (nat  $\rightarrow$  A  $\rightarrow$  A)  $\rightarrow$  nat  $\rightarrow$  A :=

```

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

```

### 1.0.1 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.

```

### 1.0.2 Basic specifications : pred add sub mul

```

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

```

### 1.0.3 Boolean comparisons

```

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

```

### 1.0.4 Decidability of equality over *nat*.

```

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

```

### 1.0.5 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*  $n\ m : (\text{S } n\ ?=\ \text{S } m) = (n\ ?=\ m)$ .

### 1.0.6 Minimum, maximum

Lemma *max\_l*  $\forall n\ m, m \leq n \rightarrow \max\ n\ m = n$ .

Lemma *max\_r*  $\forall n\ m, n \leq m \rightarrow \max\ n\ m = m$ .

Lemma *min\_l*  $\forall n\ m, n \leq m \rightarrow \min\ n\ m = n$ .

Lemma *min\_r*  $\forall n\ m, m \leq n \rightarrow \min\ n\ m = m$ .

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*.

### 1.0.7 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*  $a\ b : 0 \leq b \rightarrow a^{(\text{S } b)} = a \times a^b$ .

### 1.0.8 Square

Lemma *square\_spec*  $n : \text{square } n = n \times n$ .

### 1.0.9 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 :  $\neg$  Odd 0.  
 Lemma Odd\_2 n : Odd n  $\leftrightarrow$  Odd (S (S n)).  
 End PRIVATE\_PARITY.  
 Import Private\_Parity.  
 Lemma even\_spec :  $\forall$  n, even n = true  $\leftrightarrow$  Even n.  
 Lemma odd\_spec :  $\forall$  n, odd n = true  $\leftrightarrow$  Odd n.

### 1.0.10 Division

Lemma divmod\_spec :  $\forall$  x y q u,  $u \leq y \rightarrow$   
 let (q', u') := divmod x y q u in  
 $x + (S y) * q + (y - u) = (S y) * q' + (y - u') \wedge u' \leq y$ .  
 Lemma div\_mod x y :  $y \neq 0 \rightarrow x = y * (x / y) + x \bmod y$ .  
 Lemma mod\_bound\_pos x y :  $0 \leq x \rightarrow 0 < y \rightarrow 0 \leq x \bmod y < y$ .

### 1.0.11 Square root

Lemma sqrt\_iter\_spec :  $\forall$  k p q r,  
 $q = p + p \rightarrow r \leq q \rightarrow$   
 let s := sqrt\_iter k p q r in  
 $s * s \leq k + p * p + (q - r) < (S s) * (S s)$ .  
 Lemma sqrt\_specif n :  $(\text{sqrt } n) * (\text{sqrt } n) \leq n < S (\text{sqrt } n) * S (\text{sqrt } n)$ .  
 Definition sqrt\_spec a (Ha :  $0 \leq a$ ) := sqrt\_specif a.  
 Lemma sqrt\_neg a :  $a < 0 \rightarrow \text{sqrt } a = 0$ .

### 1.0.12 Logarithm

Lemma log2\_iter\_spec :  $\forall$  k p q r,  
 $2^{(S p)} = q + S r \rightarrow r < 2^p \rightarrow$   
 let s := log2\_iter k p q r in  
 $2^s \leq k + q < 2^{(S s)}$ .  
 Lemma log2\_spec n :  $0 < n \rightarrow$   
 $2^{(\text{log2 } n)} \leq n < 2^{(S (\text{log2 } n))}$ .  
 Lemma log2\_nonpos n :  $n \leq 0 \rightarrow \text{log2 } n = 0$ .

### 1.0.13 Gcd

Definition divide x y :=  $\exists z, y = z * x$ .  
 Notation "( x | 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 :  $0 \leq \text{gcd } a b$ .

### 1.0.14 Bitwise operations

Lemma div2\_double n :  $\text{div2 } (2 \times n) = n$ .

Lemma div2\_succ\_double n :  $\text{div2 } (\text{S } (2 \times n)) = n$ .

Lemma le\_div2 n :  $\text{div2 } (\text{S } n) \leq n$ .

Lemma lt\_div2 n :  $0 < n \rightarrow \text{div2 } n < n$ .

Lemma div2\_decr a n :  $a \leq \text{S } n \rightarrow \text{div2 } a \leq n$ .

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 :  $\text{testbit } (2 \times a + 1) (\text{S } n) = \text{testbit } a n$ .

Lemma testbit\_even\_succ' a n :  $\text{testbit } (2 \times a) (\text{S } n) = \text{testbit } a n$ .

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 :  $\forall op n a b,$   
 $\text{div2 } (\text{bitwise } op (\text{S } n) a b) = \text{bitwise } op n (\text{div2 } a) (\text{div2 } b)$ .

Lemma odd\_bitwise :  $\forall op n a b,$   
 $\text{odd } (\text{bitwise } op (\text{S } n) a b) = op (\text{odd } a) (\text{odd } b)$ .

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 ldif\_spec a b n :  
 $\text{testbit } (\text{ldif } a b) n = \text{testbit } a n \ \&\& \ \text{negb } (\text{testbit } b n)$ .

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 ≤ n ) := testbit_odd_succ' a n.`

**Definition** `testbit_even_succ a n ( _:0 ≤ n ) := testbit_even_succ' a n.`

**Lemma** `testbit_neg_r a n ( H : n < 0 ) : testbit a n = false.`

**Definition** `shiftr_spec_high a n m ( _:0 ≤ m ) := shiftr_specif_high a n m.`

**Definition** `shiftr_spec a n m ( _:0 ≤ m ) := shiftr_specif a n m.`

Properties of advanced functions (pow, sqrt, log2, ...)

**Include** `NEXTRAPROP.`

**End** `NAT.`

Re-export notations that should be available even when the *Nat* module is not imported.

**Infix** `"^" := Nat.pow : nat_scope.`

**Infix** `"=" := Nat.eqb (at level 70) : nat_scope.`

**Infix** `"<=" := Nat.leb (at level 70) : nat_scope.`

**Infix** `"<" := Nat.ltb (at level 70) : nat_scope.`

**Infix** `"?=" := Nat.compare (at level 70) : nat_scope.`

**Infix** `"/" := Nat.div : nat_scope.`

**Infix** `"mod" := Nat.modulo (at level 40, no associativity) : nat_scope.`

**Hint** `Unfold Nat.le : core.`

**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 2

# 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 3

# Library **Coq.Arith.Arith**

```
Require Export Arith_base.  
Require Export ArithRing.
```

## Chapter 4

# 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.

  Inductive between k : nat → Prop :=
    | bet_emp : between k k
    | bet_S : ∀ l, between k l → P l → between k (S l).

  Hint Constructors between: arith.

  Lemma bet_eq : ∀ k l, l = k → between k l.

  Hint Resolve bet_eq: arith.

  Lemma between_le : ∀ k l, between k l → k ≤ l.
  Hint Immediate between_le: arith.

  Lemma between_Sk_l : ∀ k l, between k l → S k ≤ l → between (S k) l.
  Hint Resolve between_Sk_l: arith.

  Lemma between_restr :
    ∀ k l (m:nat), k ≤ l → l ≤ m → between k m → between l m.

  Inductive exists_between k : nat → Prop :=
    | exists_S : ∀ l, exists_between k l → exists_between k (S l)
    | exists_le : ∀ l, k ≤ l → Q l → exists_between k (S l).

  Hint Constructors exists_between: arith.

  Lemma exists_le_S : ∀ k l, exists_between k l → S k ≤ l.

  Lemma exists_lt : ∀ k l, exists_between k l → k < l.
  Hint Immediate exists_le_S exists_lt: arith.

  Lemma exists_S_le : ∀ k l, exists_between k (S l) → k ≤ l.
  Hint Immediate exists_S_le: arith.
```

`Definition in_int p q r := p ≤ r ∧ r < q.`  
`Lemma in_int_intro : ∀ p q r, p ≤ r → r < q → in_int p q r.`  
`Hint Resolve in_int_intro: arith.`  
`Lemma in_int_lt : ∀ p q r, in_int p q r → p < q.`  
`Lemma in_int_p_Sq :`  
 `∀ p q r, in_int p (S q) r → in_int p q r ∨ r = q :>nat.`  
`Lemma in_int_S : ∀ p q r, in_int p q r → in_int p (S q) r.`  
`Hint Resolve in_int_S: arith.`  
`Lemma in_int_Sp_q : ∀ p q r, in_int (S p) q r → in_int p q r.`  
`Hint Immediate in_int_Sp_q: arith.`  
`Lemma between_in_int :`  
 `∀ k l, between k l → ∀ r, in_int k l r → P r.`  
`Lemma in_int_between :`  
 `∀ k l, k ≤ l → (∀ r, in_int k l r → P r) → between k l.`  
`Lemma exists_in_int :`  
 `∀ k l, exists_between k l → exists2 m : nat, in_int k l m & Q m.`  
`Lemma in_int_exists : ∀ k l r, in_int k l r → Q r → exists_between k l.`  
`Lemma between_or_exists :`  
 `∀ k l,`  
 `k ≤ l →`  
 `(∀ n:nat, in_int k l n → P n ∨ Q n) →`  
 `between k l ∨ exists_between k l.`  
`Lemma between_not_exists :`  
 `∀ k l,`  
 `between k l →`  
 `(∀ n:nat, in_int k l n → P n → ¬ Q n) → ¬ exists_between k l.`  
`Inductive P_nth (init:nat) : nat → nat → Prop :=`  
`| nth_O : P_nth init init 0`  
`| nth_S :`  
 `∀ k l (n:nat),`  
 `P_nth init k n → between (S k) l → Q l → P_nth init l (S n).`  
`Lemma nth_le : ∀ (init:nat) l (n:nat), P_nth init l n → init ≤ l.`  
`Definition eventually (n:nat) := exists2 k : nat, k ≤ n & Q k.`  
`Lemma event_O : eventually 0 → Q 0.`  
`End Between.`  
`Hint Resolve nth_O bet_S bet_emp bet_eq between_Sk_l exists_S exists_le`  
 `in_int_S in_int_intro: arith.`  
`Hint Immediate in_int_Sp_q exists_le_S exists_S_le: arith.`

## Chapter 5

# 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 => 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 6

# 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_leS_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 : ¬ n < m → n ≥ 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 := Nat.compare (compat "8.4").
Notation nat_compare_spec := Nat.compare_spec (compat "8.4").
Notation nat_compare_eq_iff := Nat.compare_eq_iff (compat "8.4").
Notation nat_compare_S := Nat.compare_succ (compat "8.4").

Lemma nat_compare_lt n m : n < m ↔ (n ?= m) = Lt.
Lemma nat_compare_gt n m : n > m ↔ (n ?= m) = Gt.
Lemma nat_compare_le n m : n ≤ m ↔ (n ?= m) ≠ Gt.
Lemma nat_compare_ge n m : n ≥ m ↔ (n ?= m) ≠ Lt.

```

Some projections of the above equivalences.

```

Lemma nat_compare_eq n m : (n ?= m) = Eq → n = m.
Lemma nat_compare_Lt_Lt n m : (n ?= m) = Lt → n < m.
Lemma nat_compare_Gt_gt n m : (n ?= m) = Gt → n > m.

```

A previous definition of *nat\_compare* in terms of *lt\_eq\_lt\_dec*. The new version avoids the creation of proof parts.

```

Definition nat_compare_alt (n m : nat) :=
  match lt_eq_lt_dec n m with
  | inleft (left _) ⇒ Lt
  | inleft (right _) ⇒ Eq
  | inright _ ⇒ Gt
  end.

Lemma nat_compare_equiv n m : (n ?= m) = nat_compare_alt n m.

```

A boolean version of *le* over *nat*. See now *Nat.leb* and its properties. In scope *nat\_scope*, the notation for *Nat.leb* is “*<=?*”

```

Notation leb := Nat.leb (compat "8.4").
Notation leb_iff := Nat.leb_le (compat "8.4").

Lemma leb_iff_conv m n : (n <=? m) = false ↔ m < n.
Lemma leb_correct m n : m ≤ n → (m <=? n) = true.
Lemma leb_complete m n : (m <=? n) = true → m ≤ n.
Lemma leb_correct_conv m n : m < n → (n <=? m) = false.
Lemma leb_complete_conv m n : (n <=? m) = false → m < n.
Lemma leb_compare n m : (n <=? m) = true ↔ (n ?= m) ≠ Gt.

```

## Chapter 7

# 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* :

$\forall n\ m, n < m \rightarrow S\ n = m \vee (\exists r : \text{nat}, m = S\ (S\ (n + r)))$ .

**Require Export** *Wf\_nat*.

**Require Export** *Min Max*.

## Chapter 8

# 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**  $\text{div2} := \text{Nat.div2}$  (*compat* "8.4").

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  $n$  :  $0 < n \rightarrow \text{div2}\ n < n.$

**Hint Resolve** lt\_div2: arith.

Properties related to the parity

**Lemma** even\_div2  $n$  :  $\text{even}\ n \rightarrow \text{div2}\ n = \text{div2}\ (S\ n).$

**Lemma** odd\_div2  $n$  :  $\text{odd}\ n \rightarrow S\ (\text{div2}\ n) = \text{div2}\ (S\ n).$

**Lemma** div2\_even  $n$  :  $\text{div2}\ n = \text{div2}\ (S\ n) \rightarrow \text{even}\ n.$

**Lemma** div2\_odd  $n$  :  $S\ (\text{div2}\ n) = \text{div2}\ (S\ n) \rightarrow \text{odd}\ n.$

**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 (*compat* "8.4").

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  $m$ .

Hint Resolve double\_S: *arith*.

Lemma even\_odd\_double  $n$  :  
 (even  $n \leftrightarrow n = \text{double } (\text{div2 } n)$ )  $\wedge$  (odd  $n \leftrightarrow n = \text{S } (\text{double } (\text{div2 } n))$ ).

Specializations

Lemma even\_double  $n$  : even  $n \rightarrow n = \text{double } (\text{div2 } n)$ .

Lemma double\_even  $n$  :  $n = \text{double } (\text{div2 } n) \rightarrow \text{even } n$ .

Lemma odd\_double  $n$  : odd  $n \rightarrow n = \text{S } (\text{double } (\text{div2 } n))$ .

Lemma double\_odd  $n$  :  $n = \text{S } (\text{double } (\text{div2 } n)) \rightarrow \text{odd } n$ .

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 :  $\forall n, \text{odd } n \rightarrow \{p : \text{nat} \mid n = \text{S } (\text{double } p)\}$ .

Doubling before dividing by two brings back to the initial number.

Lemma div2\_double  $n$  : div2 ( $2 \times n$ ) =  $n$ .

Lemma div2\_double\_plus\_one  $n$  : div2 (S ( $2 \times n$ )) =  $n$ .

## Chapter 9

# Library `Coq.Arith.EqNat`

```
Require Import PeanoNat.  
Local Open Scope nat_scope.  
    Equality on natural numbers
```

### 9.1 Propositional equality

```
Fixpoint eq_nat n m : Prop :=  
  match n, m with  
  | O, O => True  
  | O, S _ => False  
  | S _, O => False  
  | S n1, S m1 => eq_nat n1 m1  
  end.  
  
Theorem eq_nat_refl n : eq_nat n n.  
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.  
Hint Immediate eq_eq_nat eq_nat_eq: arith.  
  
Theorem eq_nat_elim :  
   $\forall n (P : \text{nat} \rightarrow \text{Prop}), P\ n \rightarrow \forall m, \text{eq\_nat}\ n\ m \rightarrow P\ m$ .  
Theorem eq_nat_decide :  $\forall n\ m, \{\text{eq\_nat}\ n\ m\} + \{\neg \text{eq\_nat}\ n\ m\}$ .
```

### 9.2 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 (compat "8.4").
Notation beq_nat_true_iff := Nat.eqb_eq (compat "8.4").
Notation beq_nat_false_iff := Nat.eqb_neq (compat "8.4").
Lemma beq_nat_refl n : true = (n =? n).
Lemma beq_nat_true n m : (n =? m) = true → n=m.
Lemma beq_nat_false n m : (n =? m) = false → n≠m.

```

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

```

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

```

## Chapter 10

# 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 11

## 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.
```

### 11.1 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).  
Hint Constructors even: arith.  
Hint Constructors odd: arith.
```

### 11.2 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.
```



### 11.3 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.

```

### 11.4 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  $m$ .

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 12

# 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 13

# 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.

### 13.1 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.

### 13.2 Irreflexivity

Lemma gt\_irrefl n : ¬ n > n.

### 13.3 Asymmetry

Lemma gt\_asym n m : n > m → ¬ m > n.

## 13.4 Relating strict and large orders

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

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

Theorem `le_S_gt`  $n\ m : S\ n \leq m \rightarrow m > n$ .

Lemma `gt_S_le`  $n\ m : S\ m > n \rightarrow n \leq m$ .

Lemma `gt_le_S`  $n\ m : m > n \rightarrow S\ n \leq m$ .

Lemma `le_gt_S`  $n\ m : n \leq m \rightarrow S\ m > n$ .

## 13.5 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`  $n\ m\ p : S\ n > m \rightarrow m > p \rightarrow n > p$ .

## 13.6 Comparison to 0

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

## 13.7 Simplification and compatibility

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

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

## 13.8 Hints

Hint `Resolve` `gt_Sn_O` `gt_Sn_n` `gt_n_S` : *arith*.

Hint `Immediate` `gt_S_n` `gt_pred` : *arith*.

Hint `Resolve` `gt_irrefl` `gt_asym` : *arith*.

Hint `Resolve` `le_not_gt` `gt_not_le` : *arith*.

Hint `Immediate` `le_S_gt` `gt_S_le` : *arith*.

Hint `Resolve` `gt_le_S` `le_gt_S` : *arith*.

Hint `Resolve` `gt_trans_S` `le_gt_trans` `gt_le_trans` : *arith*.

Hint `Resolve` `plus_gt_compat_l` : *arith*.

## Chapter 14

# 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 " $n \leq m$ " := (le n m) : nat\_scope.

Require Import PeanoNat.

Local Open Scope nat\_scope.

### 14.1 *le* is an order on *nat*

Notation le\_refl := Nat.le\_refl (compat "8.4").

Notation le\_trans := Nat.le\_trans (compat "8.4").

Notation le\_antisym := Nat.le\_antisymm (compat "8.4").

Hint Resolve le\_trans: arith.

Hint Immediate le\_antisym: arith.

### 14.2 Properties of *le* w.r.t 0

Notation le\_0\_n := Nat.le\_0\_l (compat "8.4"). Notation le\_Sn\_0 := Nat.nle\_succ\_0 (compat "8.4").

Lemma le\_n\_0\_eq  $n$  :  $n \leq 0 \rightarrow 0 = n$ .

### 14.3 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` (*compat* "8.4"). **Notation** `le_Sn_n` := `Nat.nle_succ_diag_l` (*compat* "8.4").

**Theorem** `le_Sn_le` :  $\forall n\ m, S\ n \leq m \rightarrow n \leq m$ .

**Hint** `Resolve` `le_0_n` `le_Sn_0` : *arith*.

**Hint** `Resolve` `le_n_S` `le_n_Sn` `le_Sn_n` : *arith*.

**Hint** `Immediate` `le_n_0_eq` `le_Sn_le` `le_S_n` : *arith*.

## 14.4 Properties of *le* w.r.t predecessor

**Notation** `le_pred_n` := `Nat.le_pred_l` (*compat* "8.4"). **Notation** `le_pred` := `Nat.pred_le_mono` (*compat* "8.4").

**Hint** `Resolve` `le_pred_n` : *arith*.

## 14.5 A different elimination principle for the order on natural numbers

**Lemma** `le_elim_rel` :

$\forall P:\text{nat} \rightarrow \text{nat} \rightarrow \text{Prop},$   
     $(\forall p, P\ 0\ p) \rightarrow$   
     $(\forall p\ (q:\text{nat}), p \leq q \rightarrow P\ p\ q \rightarrow P\ (S\ p)\ (S\ q)) \rightarrow$   
     $\forall n\ m, n \leq m \rightarrow P\ n\ m.$

## Chapter 15

# 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.
```

### 15.1 Irreflexivity

```
Notation lt_irrefl := Nat.lt_irrefl (compat "8.4").
```

```
Hint Resolve lt_irrefl: arith.
```

### 15.2 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.
```

```
Hint Immediate lt_le_S: arith.
```

```
Hint Immediate lt_n_Sm_le: arith.
```

```
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.
```

```
Hint Immediate le_not_lt lt_not_le: arith.
```



### 15.3 Asymmetry

**Notation** `lt_asym` := `Nat.lt_asymm` (*compat* "8.4").

### 15.4 Order and 0

**Notation** `lt_0_Sn` := `Nat.lt_0_succ` (*compat* "8.4"). **Notation** `lt_n_0` := `Nat.nlt_0_r` (*compat* "8.4").

**Theorem** `neq_0_lt`  $n : 0 \neq n \rightarrow 0 < n$ .

**Theorem** `lt_0_neq`  $n : 0 < n \rightarrow 0 \neq n$ .

**Hint** `Resolve` `lt_0_Sn` `lt_n_0` : *arith*.

**Hint** `Immediate` `neq_0_lt` `lt_0_neq` : *arith*.

### 15.5 Order and successor

**Notation** `lt_n_Sn` := `Nat.lt_succ_diag_r` (*compat* "8.4"). **Notation** `lt_S` := `Nat.lt_lt_succ_r` (*compat* "8.4").

**Theorem** `lt_n_S`  $n\ m : n < m \rightarrow S\ n < S\ m$ .

**Theorem** `lt_S_n`  $n\ m : S\ n < S\ m \rightarrow n < m$ .

**Hint** `Resolve` `lt_n_Sn` `lt_S` `lt_n_S` : *arith*.

**Hint** `Immediate` `lt_S_n` : *arith*.

### 15.6 Predecessor

**Lemma** `S_pred`  $n\ m : m < n \rightarrow n = S\ (\text{pred } n)$ .

**Lemma** `lt_pred`  $n\ m : S\ n < m \rightarrow n < \text{pred } m$ .

**Lemma** `lt_pred_n_n`  $n : 0 < n \rightarrow \text{pred } n < n$ .

**Hint** `Immediate` `lt_pred` : *arith*.

**Hint** `Resolve` `lt_pred_n_n` : *arith*.

### 15.7 Transitivity properties

**Notation** `lt_trans` := `Nat.lt_trans` (*compat* "8.4").

**Notation** `lt_le_trans` := `Nat.lt_le_trans` (*compat* "8.4").

**Notation** `le_lt_trans` := `Nat.le_lt_trans` (*compat* "8.4").

**Hint** `Resolve` `lt_trans` `lt_le_trans` `le_lt_trans` : *arith*.

## 15.8 Large = strict or equal

**Notation** `le_lt_or_eq_iff` := `Nat.lt_eq_cases` (*compat* "8.4").

**Theorem** `le_lt_or_eq`  $n\ m : n \leq m \rightarrow n < m \vee n = m$ .

**Notation** `lt_le_weak` := `Nat.lt_le_incl` (*compat* "8.4").

**Hint Immediate** `lt_le_weak`: *arith*.

## 15.9 Dichotomy

**Notation** `le_or_lt` := `Nat.le_gt_cases` (*compat* "8.4").

**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 16

# 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.

Hint Resolve
  Nat.max_l Nat.max_r Nat.le_max_l Nat.le_max_r : arith.

Hint Resolve
  Nat.min_l Nat.min_r Nat.le_min_l Nat.le_min_r : arith.
```

## Chapter 17

# 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*.

### 17.1 0 is right neutral

**Lemma** *minus\_n\_0*  $n : n = n - 0$ .

### 17.2 Permutation with successor

**Lemma** *minus\_Sn\_m*  $n m : m \leq n \rightarrow S (n - m) = S n - m$ .

**Theorem** *pred\_of\_minus*  $n : \text{pred } n = n - 1$ .

### 17.3 Diagonal

**Notation** *minus\_diag* := *Nat.sub\_diag* (*compat* "8.4").

**Lemma** *minus\_diag\_reverse*  $n : 0 = n - n$ .

**Notation** *minus\_n\_n* := *minus\_diag\_reverse*.

## 17.4 Simplification

**Lemma** `minus_plus_simpl_l_reverse`  $n\ m\ p : n - m = p + n - (p + m)$ .

## 17.5 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`  $n\ m : n \leq m \rightarrow m = n + (m - n)$ .

## 17.6 Relation with order

**Notation** `minus_le_compat_r` :=  
    `Nat.sub_le_mono_r` (*compat* "8.4").

**Notation** `minus_le_compat_l` :=  
    `Nat.sub_le_mono_l` (*compat* "8.4").

**Notation** `le_minus` := `Nat.le_sub_l` (*compat* "8.4").   **Notation** `lt_minus` := `Nat.sub_lt` (*compat* "8.4").

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

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

## 17.7 Hints

**Hint** `Resolve` *minus\_n\_O*: *arith*.

**Hint** `Resolve` *minus\_Sn\_m*: *arith*.

**Hint** `Resolve` *minus\_diag\_reverse*: *arith*.

**Hint** `Resolve` *minus\_plus\_simpl\_l\_reverse*: *arith*.

**Hint** `Immediate` *plus\_minus*: *arith*.

**Hint** `Resolve` *minus\_plus*: *arith*.

**Hint** `Resolve` *le\_plus\_minus*: *arith*.

**Hint** `Resolve` *le\_plus\_minus\_r*: *arith*.

**Hint** `Resolve` *lt\_minus*: *arith*.

**Hint** `Immediate` *lt\_O\_minus\_lt*: *arith*.

## Chapter 18

# 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 19

## Library **Coq.Arith.Mult**

### 19.1 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*.

### 19.2 *nat* is a semi-ring

#### 19.2.1 Zero property

**Notation** *mult\_0\_l* := *Nat.mul\_0\_l* (*compat* "8.4"). **Notation** *mult\_0\_r* := *Nat.mul\_0\_r* (*compat* "8.4").

#### 19.2.2 1 is neutral

**Notation** *mult\_1\_l* := *Nat.mul\_1\_l* (*compat* "8.4"). **Notation** *mult\_1\_r* := *Nat.mul\_1\_r* (*compat* "8.4").

**Hint Resolve** *mult\_1\_l mult\_1\_r*: *arith*.

#### 19.2.3 Commutativity

**Notation** *mult\_comm* := *Nat.mul\_comm* (*compat* "8.4").

**Hint Resolve** *mult\_comm*: *arith*.

#### 19.2.4 Distributivity

**Notation** *mult\_plus\_distr\_r* :=  
*Nat.mul\_add\_distr\_r* (*compat* "8.4").

```

Notation mult_plus_distr_l :=
  Nat.mul_add_distr_l (compat "8.4").
Notation mult_minus_distr_r :=
  Nat.mul_sub_distr_r (compat "8.4").
Notation mult_minus_distr_l :=
  Nat.mul_sub_distr_l (compat "8.4").
Hint Resolve mult_plus_distr_r: arith.
Hint Resolve mult_minus_distr_r: arith.
Hint Resolve mult_minus_distr_l: arith.

```

### 19.2.5 Associativity

```

Notation mult_assoc := Nat.mul_assoc (compat "8.4").
Lemma mult_assoc_reverse n m p :  $n \times m \times p = n \times (m \times p)$ .
Hint Resolve mult_assoc_reverse: arith.
Hint Resolve mult_assoc: arith.

```

### 19.2.6 Inversion lemmas

```

Lemma mult_is_O n m :  $n \times m = 0 \rightarrow n = 0 \vee m = 0$ .
Lemma mult_is_one n m :  $n \times m = 1 \rightarrow n = 1 \wedge m = 1$ .

```

### 19.2.7 Multiplication and successor

```

Notation mult_succ_l := Nat.mul_succ_l (compat "8.4"). Notation mult_succ_r := Nat.mul_succ_r
  (compat "8.4").

```

## 19.3 Compatibility with orders

```

Lemma mult_O_le n m :  $m = 0 \vee n \leq m \times n$ .
Hint Resolve mult_O_le: arith.
Lemma mult_le_compat_l n m p :  $n \leq m \rightarrow p \times n \leq p \times m$ .
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 :  $m < p \rightarrow S\ n \times m < S\ n \times p$ .
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 :  $S\ n \times m \leq S\ n \times p \rightarrow m \leq p$ .

```



## 19.4 $n|->2*n$ and $n|->2n+1$ have disjoint image

**Theorem** `odd_even_lem`  $p\ q : 2 \times p + 1 \neq 2 \times q$ .

## 19.5 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 20

# Library `Coq.Arith.Peano_dec`

```
Require Import Decidable PeanoNat.
Require Export Eqty_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 (compat "8.4").
Hint Resolve O_or_S eq_nat_dec: arith.
Theorem dec_eq_nat n m : decidable (n = m).
Definition UIP_nat := Eqty_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 21

## 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*.

### 21.1 Neutrality of 0, commutativity, associativity

**Notation** *plus\_0\_l* := *Nat.add\_0\_l* (*compat* "8.4").

**Notation** *plus\_0\_r* := *Nat.add\_0\_r* (*compat* "8.4").

**Notation** *plus\_comm* := *Nat.add\_comm* (*compat* "8.4").

**Notation** *plus\_assoc* := *Nat.add\_assoc* (*compat* "8.4").

**Notation** *plus\_permute* := *Nat.add\_shuffle3* (*compat* "8.4").

**Definition** *plus\_Snm\_nSm* :  $\forall n\ m, S\ n + m = n + S\ m :=$   
*Peano.plus\_n\_Sm*.

**Lemma** *plus\_assoc\_reverse* *n m p* :  $n + m + p = n + (m + p)$ .

### 21.2 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* :  $p + n < p + m \rightarrow n < m$ .

## 21.3 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`  $n\ m\ p : n < m \rightarrow n < m + p$ .

## 21.4 Inversion lemmas

Lemma `plus_is_O`  $n\ m : n + m = 0 \rightarrow n = 0 \wedge m = 0$ .  
Definition `plus_is_one`  $m\ n :$   
 $m + n = 1 \rightarrow \{m = 0 \wedge n = 1\} + \{m = 1 \wedge n = 0\}$ .

## 21.5 Derived properties

Notation `plus_permute_2_in_4`  $:= \text{Nat.add\_shuffle1 } (\text{compat } "8.4")$ .

## 21.6 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`  $n\ m : \text{nat} :=$   
  match  $n$  with  
  |  $O \Rightarrow m$   
  |  $S\ n \Rightarrow \text{tail\_plus } n\ (S\ m)$   
  end.

Lemma `plus_tail_plus`  $: \forall\ n\ m, n + m = \text{tail\_plus } n\ m$ .

## 21.7 Discrimination

`Lemma succ_plus_discr n m : n ≠ S (m+n).`

`Lemma n_SSn n : n ≠ S (S n).`

`Lemma n_SSSn n : n ≠ S (S (S n)).`

`Lemma n_SSSSn n : n ≠ S (S (S (S n))).`

## 21.8 Compatibility Hints

`Hint Immediate plus_comm : arith.`

`Hint Resolve plus_assoc plus_assoc_reverse : arith.`

`Hint Resolve plus_le_compat_l plus_le_compat_r : arith.`

`Hint Resolve le_plus_l le_plus_r le_plus_trans : arith.`

`Hint Immediate lt_plus_trans : arith.`

`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 22

# Library **Coq.Arith.Wf\_nat**

Well-founded relations and natural numbers

```
Require Import PeanoNat Lt.
```

```
Local Open Scope nat_scope.
```

```
Implicit Types m n p : nat.
```

```
Section Well_founded_Nat.
```

```
Variable A : Type.
```

```
Variable f : A → nat.
```

```
Definition ltof (a b:A) := f a < f b.
```

```
Definition gtof (a b:A) := f b > f a.
```

```
Theorem well_founded_ltof : well_founded ltof.
```

```
Theorem well_founded_gtof : well_founded gtof.
```

It is possible to directly prove the induction principle going back to primitive recursion on natural numbers (*induction\_ltof1*) or to use the previous lemmas to extract a program with a fixpoint (*induction\_ltof2*)

the ML-like program for *induction\_ltof1* is :

```
let induction_ltof1 f F a =  
  let rec indrec n k =  
    match n with  
    | O → error  
    | S m → F k (indrec m)  
  in indrec (f a + 1) a
```

the ML-like program for *induction\_ltof2* is :

```
let induction_ltof2 F a = indrec a  
where rec indrec a = F a indrec;;
```

```
Theorem induction_ltof1 :
```

```
  ∀ P:A → Set,
```

```
    (∀ x:A, (∀ y:A, ltof y x → P y) → P x) → ∀ a:A, P a.
```

```
Theorem induction_gtof1 :
```

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

**Theorem** `induction_ltof2` :

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

**Theorem** `induction_gtof2` :

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

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_rec1` :

$\forall n \ (P:\text{nat} \rightarrow \text{Set}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

**Lemma** `lt_wf_rec` :

$\forall n \ (P:\text{nat} \rightarrow \text{Set}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

**Lemma** `lt_wf_ind` :

$\forall n \ (P:\text{nat} \rightarrow \text{Prop}), (\forall n, (\forall m, m < n \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

**Lemma** `gt_wf_rec` :

$\forall n \ (P:\text{nat} \rightarrow \text{Set}), (\forall n, (\forall m, n > m \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

**Lemma** `gt_wf_ind` :

$\forall n \ (P:\text{nat} \rightarrow \text{Prop}), (\forall n, (\forall m, n > m \rightarrow P \ m) \rightarrow P \ n) \rightarrow P \ n.$

**Lemma** `lt_wf_double_rec` :

$\forall P:\text{nat} \rightarrow \text{nat} \rightarrow \text{Set},$   
 $(\forall n \ m,$   
 $(\forall p \ q, p < n \rightarrow P \ p \ q) \rightarrow$   
 $(\forall p, p < m \rightarrow P \ n \ p) \rightarrow P \ n \ m) \rightarrow \forall n \ m, P \ n \ m.$

**Lemma** `lt_wf_double_ind` :

$\forall P:\text{nat} \rightarrow \text{nat} \rightarrow \text{Prop},$   
 $(\forall n \ m,$   
 $(\forall p \ (q:\text{nat}), p < n \rightarrow P \ p \ q) \rightarrow$   
 $(\forall p, p < m \rightarrow P \ n \ p) \rightarrow P \ n \ m) \rightarrow \forall n \ m, P \ n \ m.$

**Hint** `Resolve`  $lt\_wf$ : *arith*.

**Hint** `Resolve`  $well\_founded\_lt\_compat$ : *arith*.

**Section** `LT_WF_REL.`

**Variable**  $A : \text{Set}.$

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

**Variable**  $F : A \rightarrow \text{nat} \rightarrow \text{Prop}.$

```

Definition inv_lt_rel x y := exists2 n, F x n & (∀ m, F y m → n < m).
Hypothesis F_compat : ∀ x y:A, R x y → inv_lt_rel x y.
Remark acc_lt_rel : ∀ x:A, (∃ n, F x n) → Acc R x.

Theorem well_founded_inv_lt_rel_compat : well_founded R.
End LT_WF_REL.

Lemma well_founded_inv_rel_inv_lt_rel :
  ∀ (A:Set) (F:A → nat → Prop), well_founded (inv_lt_rel A F).

  A constructive proof that any non empty decidable subset of natural numbers has a least element

Set Implicit Arguments.
Require Import Le.
Require Import Compare_dec.
Require Import Decidable.

Definition has_unique_least_element (A:Type) (R:A→A→Prop) (P:A→Prop) :=
  ∃! x, P x ∧ ∀ x', P x' → R x x'.

Lemma dec_inh_nat_subset_has_unique_least_element :
  ∀ P:nat→Prop, (∀ n, P n ∨ ¬ P n) →
    (∃ n, P n) → has_unique_least_element le P.

Unset Implicit Arguments.

Notation iter_nat n A f x := (nat_rect (fun _ ⇒ A) x (fun _ ⇒ f) n) (only parsing).

```



## Chapter 23

# 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**  $\text{AC\_IF} :$

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

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

**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` :  $T = F$ .

**End** `Berardis_paradox`.

## Chapter 24

# Library **Coq.Logic.ChoiceFacts**

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

We investigate the relations between the following choice and description principles

- AC\_rel = relational form of the (non extensional) axiom of choice (a “set-theoretic” axiom of choice)
- AC\_fun = functional form of the (non extensional) axiom of choice (a “type-theoretic” axiom of choice)
- DC\_fun = functional form of the dependent axiom of choice
- ACw\_fun = functional form of the countable axiom of choice
- 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)
- GAC\_rel = guarded relational form of the (non extensional) axiom of choice
- GAC\_fun = guarded functional form of the (non extensional) axiom of choice
- GAC! = guarded functional relation reification
- OAC\_rel = “omniscient” relational form of the (non extensional) axiom of choice
- OAC\_fun = “omniscient” functional form of the (non extensional) axiom of choice (called AC\* in Bell [*Bell*])
- OAC!
- ID\_iota = intuitionistic definite description
- ID\_epsilon = intuitionistic indefinite description
- D\_iota = (weakly classical) definite description principle
- D\_epsilon = (weakly classical) indefinite description principle
- PI = proof irrelevance

- IGP = independence of general premises (an unconstrained generalisation of the constructive principle of independence of premises)
- Drinker = drinker's paradox (small form) (called Ex in Bell [*Bell*])

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

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. Equivalence of choices on dependent or non dependent functional types

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

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.

[*Carlström05*] Jesper Carlström, Interpreting descriptions in intentional type theory, Journal of Symbolic Logic 70(2):488-514, 2005.

**Set Implicit Arguments.**

## 24.1 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.

**Variable** *R*:*A*→*B*→Prop.

### 24.1.1 Constructive choice and description

AC\_rel

**Definition** RelationalChoice\_on :=

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

AC\_fun

**Definition** FunctionalChoice\_on :=

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

DC\_fun

**Definition** FunctionalDependentChoice\_on :=

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

ACw\_fun

**Definition** FunctionalCountableChoice\_on :=

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

AC! or Functional Relation Reification (known as Axiom of Unique Choice in topos theory; also called principle of definite description

**Definition** FunctionalRelReification\_on :=

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

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 :=

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

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 :=

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

### 24.1.2 Weakly classical choice and description

GAC\_rel

**Definition** GuardedRelationalChoice\_on :=  
 $\forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $(\forall x : A, P x \rightarrow \exists y : B, R x y) \rightarrow$   
 $(\exists R' : A \rightarrow B \rightarrow \text{Prop},$   
 $\text{subrelation } R' R \wedge \forall x, P x \rightarrow \exists! y, R' x y).$   
 GAC\_fun

**Definition** GuardedFunctionalChoice\_on :=  
 $\forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $\text{inhabited } B \rightarrow$   
 $(\forall x : A, P x \rightarrow \exists y : B, R x y) \rightarrow$   
 $(\exists f : A \rightarrow B, \forall x, P x \rightarrow R x (f x)).$   
 GFR\_fun

**Definition** GuardedFunctionalRelReification\_on :=  
 $\forall P : A \rightarrow \text{Prop}, \forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $\text{inhabited } B \rightarrow$   
 $(\forall x : A, P x \rightarrow \exists! y : B, R x y) \rightarrow$   
 $(\exists f : A \rightarrow B, \forall x : A, P x \rightarrow R x (f x)).$   
 OAC\_rel

**Definition** OmniscientRelationalChoice\_on :=  
 $\forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $\exists R' : A \rightarrow B \rightarrow \text{Prop},$   
 $\text{subrelation } R' R \wedge \forall x : A, (\exists y : B, R x y) \rightarrow \exists! y, R' x y.$   
 OAC\_fun

**Definition** OmniscientFunctionalChoice\_on :=  
 $\forall R : A \rightarrow B \rightarrow \text{Prop},$   
 $\text{inhabited } B \rightarrow$   
 $\exists f : A \rightarrow B, \forall x : A, (\exists y : B, R x y) \rightarrow R x (f x).$   
 D\_epsilon

**Definition** EpsilonStatement\_on :=  
 $\forall P : A \rightarrow \text{Prop},$   
 $\text{inhabited } A \rightarrow \{ x : A \mid (\exists x, P x) \rightarrow P x \}.$   
 D\_iota

**Definition** IotaStatement\_on :=  
 $\forall P : A \rightarrow \text{Prop},$   
 $\text{inhabited } A \rightarrow \{ x : A \mid (\exists! x, P x) \rightarrow P x \}.$

**End** ChoiceSchemes.

Generalized schemes

**Notation** RelationalChoice :=  
 $(\forall A B : \text{Type}, \text{RelationalChoice\_on } A B).$

**Notation** FunctionalChoice :=  
 $(\forall A B : \text{Type}, \text{FunctionalChoice\_on } A B).$

**Definition** FunctionalDependentChoice :=  
 $(\forall A : \text{Type}, \text{FunctionalDependentChoice\_on } A).$   
**Definition** FunctionalCountableChoice :=  
 $(\forall A : \text{Type}, \text{FunctionalCountableChoice\_on } A).$   
**Notation** FunctionalChoiceOnInhabitedSet :=  
 $(\forall A B : \text{Type}, \text{inhabited } B \rightarrow \text{FunctionalChoice\_on } A B).$   
**Notation** FunctionalRelReification :=  
 $(\forall A B : \text{Type}, \text{FunctionalRelReification\_on } A B).$   
**Notation** GuardedRelationalChoice :=  
 $(\forall A B : \text{Type}, \text{GuardedRelationalChoice\_on } A B).$   
**Notation** GuardedFunctionalChoice :=  
 $(\forall A B : \text{Type}, \text{GuardedFunctionalChoice\_on } A B).$   
**Notation** GuardedFunctionalRelReification :=  
 $(\forall A B : \text{Type}, \text{GuardedFunctionalRelReification\_on } A B).$   
**Notation** OmniscientRelationalChoice :=  
 $(\forall A B : \text{Type}, \text{OmniscientRelationalChoice\_on } A B).$   
**Notation** OmniscientFunctionalChoice :=  
 $(\forall A B : \text{Type}, \text{OmniscientFunctionalChoice\_on } A B).$   
**Notation** ConstructiveDefiniteDescription :=  
 $(\forall A : \text{Type}, \text{ConstructiveDefiniteDescription\_on } A).$   
**Notation** ConstructiveIndefiniteDescription :=  
 $(\forall A : \text{Type}, \text{ConstructiveIndefiniteDescription\_on } A).$   
**Notation** IotaStatement :=  
 $(\forall A : \text{Type}, \text{IotaStatement\_on } A).$   
**Notation** EpsilonStatement :=  
 $(\forall A : \text{Type}, \text{EpsilonStatement\_on } A).$   
 Subclassical schemes  
**Definition** ProofIrrelevance :=  
 $\forall (A:\text{Prop}) (a1\ a2:A), a1 = a2.$   
**Definition** IndependenceOfGeneralPremises :=  
 $\forall (A:\text{Type}) (P:A \rightarrow \text{Prop}) (Q:\text{Prop}),$   
 $\text{inhabited } A \rightarrow$   
 $(Q \rightarrow \exists x, P\ x) \rightarrow \exists x, Q \rightarrow P\ x.$   
**Definition** SmallDrinker'sParadox :=  
 $\forall (A:\text{Type}) (P:A \rightarrow \text{Prop}), \text{inhabited } A \rightarrow$   
 $\exists x, (\exists x, P\ x) \rightarrow P\ x.$

## 24.2 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** `description_rel_choice_imp_func_choice` :  
 $\forall A B : \text{Type},$   
 $\text{FunctionalRelReification\_on } A B \rightarrow \text{RelationalChoice\_on } A B \rightarrow \text{FunctionalChoice\_on } A B.$

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

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

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

## 24.3 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)

### 24.3.1 $\text{AC\_rel} + \text{PI} \rightarrow \text{GAC\_rel}$ and $\text{AC\_rel} + \text{IGP} \rightarrow \text{GAC\_rel}$ and $\text{GAC\_rel} = \text{OAC\_rel}$

**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` :  
 $\text{GuardedRelationalChoice} \leftrightarrow \text{OmniscientRelationalChoice}.$

### 24.3.2 $\text{AC\_fun} + \text{IGP} = \text{GAC\_fun} = \text{OAC\_fun} = \text{AC\_fun} + \text{Drinker}$

$\text{AC\_fun} + \text{IGP} = \text{GAC\_fun}$

**Lemma** `guarded_fun_choice_imp_indep_of_general_premises` :  
 $\text{GuardedFunctionalChoice} \rightarrow \text{IndependenceOfGeneralPremises}.$



**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.

### 24.3.3 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  $\rightarrow$  SmallDrinker'sParadox.

**Theorem** constructive\_indefinite\_description\_and\_small\_drinker\_iff\_epsilon :

(SmallDrinker'sParadox  $\times$  ConstructiveIndefiniteDescription  $\rightarrow$

EpsilonStatement) ×  
 (EpsilonStatement →  
 SmallDrinker'sParadox × ConstructiveIndefiniteDescription).

## 24.4 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.

**Definition** FunctionalChoice\_on\_rel (A B:Type) (R:A→B→Prop) :=  
 (∀ x:A, ∃ y : B, R x y) →  
 ∃ f : A → B, (∀ x:A, R x (f x)).

**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.

## 24.5 Choice on dependent and non dependent function types are equivalent

### 24.5.1 Choice on dependent and non dependent function types are equivalent

**Definition** DependentFunctionalChoice\_on (A:Type) (B:A → Type) :=  
 ∀ R:∀ x:A, B x → Prop,  
 (∀ x:A, ∃ y : B x, R x y) →  
 (∃ f : (∀ x:A, B x), ∀ x:A, R x (f x)).

**Notation** DependentFunctionalChoice :=

(∀ A (B:A→Type), DependentFunctionalChoice\_on B).

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.`

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

**Definition** `DependentFunctionalRelReification_on (A:Type) (B:A → Type) :=  
 ∀ (R:∀ x:A, B x → Prop),  
 (∀ x:A, ∃! y : B x, R x y) →  
 (∃ f : (∀ x:A, B x), ∀ x:A, R x (f x)).`

**Notation** `DependentFunctionalRelReification :=  
 (∀ A (B:A→Type), DependentFunctionalRelReification_on B).`

The easy part

**Theorem** `dep_non_dep_functional_rel_reification :`  
`DependentFunctionalRelReification → FunctionalRelReification.`

Deriving choice on product types requires some computation on singleton propositional types, so we need computational conjunction projections and dependent elimination of conjunction and equality

**Theorem** `non_dep_dep_functional_rel_reification :`  
`FunctionalRelReification → DependentFunctionalRelReification.`

**Corollary** `dep_iff_non_dep_functional_rel_reification :`  
`FunctionalRelReification ↔ DependentFunctionalRelReification.`

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

### 24.6.1 Non contradiction of indefinite description

**Lemma** `relative_non_contradiction_of_indefinite_descr :`  
`∀ C:Prop, (ConstructiveIndefiniteDescription → C)  
 → (FunctionalChoice → C).`

**Lemma** `constructive_indefinite_descr_fun_choice :`  
`ConstructiveIndefiniteDescription → FunctionalChoice.`

### 24.6.2 Non contradiction of definite description

**Lemma** `relative_non_contradiction_of_definite_descr :`  
`∀ C:Prop, (ConstructiveDefiniteDescription → C)  
 → (FunctionalRelReification → C).`

**Lemma** `constructive_definite_descr_fun_reification :`

`ConstructiveDefiniteDescription`  $\rightarrow$  `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.

## 24.7 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` :

( $\forall A : Type$ , `ConstructiveDefiniteDescription_on`  $A$ )  $\rightarrow$   
( $\forall P:Prop$ ,  $P \vee \neg P$ )  $\rightarrow$  ( $\forall P:Prop$ ,  $\{P\} + \{\neg P\}$ ).

**Corollary** `fun_reification_descr_computational_excluded_middle_in_prop_context` :

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

## 24.8 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`.

## Chapter 25

# 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 26

# 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 : Type) (i:inhabited A) (P : A  $\rightarrow$  Prop) : A  
:= proj1_sig (classical_definite_description P i).`

`Definition iota_spec (A : Type) (i:inhabited A) (P : A  $\rightarrow$  Prop) :  
 $(\exists! x:A, P x) \rightarrow P$  (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 27

# 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* ( $A : \text{Type}$ ) ( $i : \text{inhabited } A$ ) ( $P : A \rightarrow \text{Prop}$ ) :  
 $(\exists x, P x) \rightarrow P (\text{epsilon } i P)$   
:= *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 :
   $\forall (A : \text{Type}) (i\ j : \text{inhabited } A) (P : A \rightarrow \text{Prop}),$ 
   $(\exists x, P\ x) \rightarrow \text{epsilon } i\ P = \text{epsilon } j\ P.$ 
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 28

# 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
  - 3.2. Gödel-Dummett axiom and right distributivity of implication over disjunction
  - 3.3. Independence of general premises and drinker's paradox
4. Classical logic and principle of unrestricted minimization

### 28.1 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*  $\rightarrow$  *excluded\_middle*  $\rightarrow$  *prop\_degeneracy*.

A weakest form of propositional extensionality: extensionality for provable propositions only

**Definition** `provable_prop_extensionality` :=  $\forall A:\text{Prop}, A \rightarrow A = \text{True}$ .

**Lemma** `provable_prop_ext` :  
`prop_extensionality`  $\rightarrow$  `provable_prop_extensionality`.

## 28.2 Classical logic and proof-irrelevance

### 28.2.1 CC |- `prop_ext` + `A inhabited` $\rightarrow$ (`A = A $\rightarrow$ A`) $\rightarrow$ `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`  $\rightarrow \forall A:\text{Prop}, \text{inhabited } A \rightarrow (A \rightarrow A) = A$ .

**Record** `retract` ( $A B:\text{Prop}$ ) : `Prop` :=  
 $\{\text{f1} : A \rightarrow B; \text{f2} : B \rightarrow A; \text{f1\_o\_f2} : \forall x:B, \text{f1} (\text{f2 } x) = x\}$ .

**Lemma** `prop_ext_retract_A_A_imp_A` :  
`prop_extensionality`  $\rightarrow \forall A:\text{Prop}, \text{inhabited } A \rightarrow \text{retract } A (A \rightarrow A)$ .

**Record** `has_fixpoint` ( $A:\text{Prop}$ ) : `Prop` :=  
 $\{\text{F} : (A \rightarrow A) \rightarrow A; \text{Fix} : \forall f:A \rightarrow A, \text{F } f = f (\text{F } f)\}$ .

**Lemma** `ext_prop_fixpoint` :  
`prop_extensionality`  $\rightarrow \forall A:\text{Prop}, \text{inhabited } A \rightarrow \text{has\_fixpoint } A$ .

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

### 28.2.2 CC |- `prop_ext` /\ `dep elim on bool` $\rightarrow$ `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**  
 $\text{bool\_elim\_redl} : \forall (C:\text{Prop}) (c1 c2:C), c1 = \text{bool\_elim } C c1 c2 \text{ true}.$   
**Hypothesis**

```

    bool_elim_redr :  $\forall (C:\mathbf{Prop}) (c1\ c2:C), c2 = \text{bool\_elim } C\ c1\ c2\ \text{false}.$ 
  Let bool_dep_induction :=
 $\forall P:\text{bool} \rightarrow \mathbf{Prop}, P\ \text{true} \rightarrow P\ \text{false} \rightarrow \forall b:\text{bool}, P\ b.$ 
  Lemma aux : prop_extensionality  $\rightarrow$  bool_dep_induction  $\rightarrow$  true = false.
  Lemma ext_prop_dep_proof_irrel_gen :
    prop_extensionality  $\rightarrow$  bool_dep_induction  $\rightarrow$  proof_irrelevance.
End Proof_irrelevance_gen.

```

In the pure Calculus of Constructions, we can define the boolean proposition  $\text{bool} = (C:\mathbf{Prop})C \rightarrow C \rightarrow C$  but we cannot prove that it has at most 2 elements.

#### Section Proof\_irrelevance\_Prop\_Ext\_CC.

```

  Definition BoolP :=  $\forall C:\mathbf{Prop}, C \rightarrow C \rightarrow C.$ 
  Definition TrueP : BoolP := fun C c1 c2  $\Rightarrow$  c1.
  Definition FalseP : BoolP := fun C c1 c2  $\Rightarrow$  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 :=
     $\forall P:\text{BoolP} \rightarrow \mathbf{Prop}, P\ \text{TrueP} \rightarrow P\ \text{FalseP} \rightarrow \forall b:\text{BoolP}, P\ b.$ 
  Lemma ext_prop_dep_proof_irrel_cc :
    prop_extensionality  $\rightarrow$  BoolP_dep_induction  $\rightarrow$  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*.

### 28.2.3 CIC |- prop. ext. $\rightarrow$ 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  $\rightarrow$  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.

#### 28.2.4 CC |- excluded-middle + dep elim on bool -> proof-irrelevance

This is a proof in the pure Calculus of Construction that classical logic in **Prop** + dependent elimination of disjunction entails proof-irrelevance.

Reference:

[Coquand90] T. Coquand, “Metamathematical Investigations of a Calculus of Constructions”, Proceedings of Logic in Computer Science (LICS’90), 1990.

Proof skeleton: classical logic + dependent elimination of disjunction + discrimination of proofs implies the existence of a retract from **Prop** into **bool**, hence inconsistency by encoding any paradox of system U- (e.g. Hurkens’ paradox).

**Require Import** Hurkens.

**Section** Proof\_irrelevance\_EM\_CC.

```

Variable or : Prop → Prop → Prop.
Variable or_introl : ∀ A B:Prop, A → or A B.
Variable or_intror : ∀ A B:Prop, B → or A B.
Hypothesis or_elim : ∀ A B C:Prop, (A → C) → (B → C) → or A B → C.
Hypothesis
  or_elim_redl :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (a:A),
      f a = or_elim A B C f g (or_introl A B a).
Hypothesis
  or_elim_redr :
    ∀ (A B C:Prop) (f:A → C) (g:B → C) (b:B),
      g b = or_elim A B C f g (or_intror A B b).
Hypothesis
  or_dep_elim :
    ∀ (A B:Prop) (P:or A B → Prop),
      (∀ a:A, P (or_introl A B a)) →
      (∀ b:B, P (or_intror A B b)) → ∀ b:or A B, P b.
Hypothesis em : ∀ A:Prop, or A (¬ A).
Variable B : Prop.
Variables b1 b2 : B.

p2b and b2p form a retract if ¬b1=b2
Let p2b A := or_elim A (¬ A) B (fun _ => b1) (fun _ => b2) (em A).
Let b2p b := b1 = b.
Lemma p2p1 : ∀ A:Prop, A → b2p (p2b A).
```

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

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 : Prop → Prop → Prop.

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

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

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

Hypothesis

or\_elim\_redl :

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

Hypothesis

or\_elim\_redr :

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

Hypothesis

or\_dep\_elim :

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

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

Variable B : Prop.

Variables b1 b2 : B.

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

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

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.

### 28.2.5 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.

## 28.3 Weak classical axioms

We show the following increasing in the strength of axioms:

- weak excluded-middle
- right distributivity of implication over disjunction and Gödel-Dummett axiom
- independence of general premises and drinker's paradox
- excluded-middle

### 28.3.1 Weak excluded-middle

The weak classical logic based on  $\neg\neg A \vee \neg A$  is referred to with name KC in {*ChagrovZakharyashev97* [*ChagrovZakharyashev97*] Alexander Chagrov and Michael Zakharyashev, "Modal Logic", Clarendon Press, 1997.

Definition *weak\_excluded\_middle* :=  
 $\forall A:\text{Prop}, \neg\neg A \vee \neg A$ .

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

### 28.3.2 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 No. 2(1959), pp 97-103.

[Gödel33] Kurt Gödel. “Zum intuitionistischen Aussagenkalkül”, Ergeb. Math. Koll. 4 (1933), pp. 34-38.

**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`.

### 28.3.3 Independence of general premises and drinker’s paradox

Independence of general premises is the unconstrained, non constructive, version of the Independence of Premises as considered in [Troelstra73].

It is a generalization to predicate logic of the right distributivity of implication over disjunction (hence of Gödel-Dummett axiom) whose own constructive form (obtained by a restricting the third formula to be negative) is called Kreisel-Putnam principle [KreiselPutnam57].

[KreiselPutnam57], Georg Kreisel and Hilary Putnam. “Eine Unableitsbarkeitsbeweismethode für den intuitionistischen Aussagenkalkül”. Archiv für Mathematische Logik und Graundlagenforschung, 3:74- 78, 1957.

[Troelstra73], Anne Troelstra, editor. Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, volume 344 of Lecture Notes in Mathematics, Springer-Verlag, 1973.

**Definition** `IndependenceOfGeneralPremises` :=

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

**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.  $False \rightarrow \forall A, A$ )

**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.

#### 28.3.4 Principle of unrestricted minimization

**Require Import** Coq.Arith.PeanoNat.

**Definition** Minimal ( $P:\text{nat} \rightarrow \text{Prop}$ ) ( $n:\text{nat}$ ) :  $\text{Prop}$  :=  
 $P\ n \wedge \forall k, P\ k \rightarrow n \leq k$ .

**Definition** Minimization\_Property ( $P : \text{nat} \rightarrow \text{Prop}$ ) :  $\text{Prop}$  :=  
 $\forall n, P\ n \rightarrow \exists m, \text{Minimal } P\ m$ .

**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 :  $\text{nat} \rightarrow \text{bool}$ .

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

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

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

## Chapter 29

# 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 30

# Library `Coq.Logic.Classical_Prop`

Classical Propositional Logic

```
Require Import ClassicalFacts.
```

```
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
| _ : _ |- ?X1  $\vee$  _  $\Rightarrow$  (elim (classic X1); intro; [left; trivial | right])
end.
```

```
Ltac classical_left := match goal with
| _ : _  $\vdash$  _  $\vee$  ?X1  $\Rightarrow$  (elim (classic X1); intro; [right; trivial | left])
```

```

end.

Require Export EqdepFacts.

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.

Module EQDEPTHEORY := EQDEPTHEORY(EQ_RECT_EQ).
Export EqdepTheory.

```

# Chapter 31

## 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 32

# Library **Coq.Logic.Classical**

Classical Logic

**Require Export** Classical\_Prop.

**Require Export** Classical\_Pred\_Type.

## Chapter 33

# 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

**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)).

End ConstructiveIndefiniteGroundDescription_Direct.

```

Require Import Arith.

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 :

(∃ *n* : nat, *P* *n*) → {*n* : nat | *P* *n*}.

End ConstructiveIndefiniteGroundDescription\_Acc.

Section ConstructiveGroundEpsilon\_nat.



Variable  $P : \text{nat} \rightarrow \text{Prop}$ .  
 Hypothesis  $P\_decidable : \forall x : \text{nat}, \{P\ x\} + \{\neg P\ x\}$ .  
 Definition  $\text{constructive\_ground\_epsilon\_nat} (E : \exists n : \text{nat}, P\ n) : \text{nat}$   
 $\quad := \text{proj1\_sig} (\text{constructive\_indefinite\_ground\_description\_nat } P\ P\_decidable\ E)$ .  
 Definition  $\text{constructive\_ground\_epsilon\_spec\_nat} (E : (\exists n, P\ n)) : P (\text{constructive\_ground\_epsilon\_nat } E)$   
 $\quad := \text{proj2\_sig} (\text{constructive\_indefinite\_ground\_description\_nat } P\ P\_decidable\ E)$ .  
 End ConstructiveGroundEpsilon\_nat.  
  
 Section ConstructiveGroundEpsilon.  
 For the current purpose, we say that a set  $A$  is countable if there are functions  $f : A \rightarrow \text{nat}$  and  $g : \text{nat} \rightarrow A$  such that  $g$  is a left inverse of  $f$ .  
 Variable  $A : \text{Type}$ .  
 Variable  $f : A \rightarrow \text{nat}$ .  
 Variable  $g : \text{nat} \rightarrow A$ .  
 Hypothesis  $\text{gof\_eq\_id} : \forall x : A, g\ (f\ x) = x$ .  
 Variable  $P : A \rightarrow \text{Prop}$ .  
 Hypothesis  $P\_decidable : \forall x : A, \{P\ x\} + \{\neg P\ x\}$ .  
 Definition  $P' (x : \text{nat}) : \text{Prop} := P\ (g\ x)$ .  
 Lemma  $P'\_decidable : \forall n : \text{nat}, \{P'\ n\} + \{\neg P'\ n\}$ .  
 Lemma  $\text{constructive\_indefinite\_ground\_description} : (\exists x : A, P\ x) \rightarrow \{x : A \mid P\ x\}$ .  
 Lemma  $\text{constructive\_definite\_ground\_description} : (\exists! x : A, P\ x) \rightarrow \{x : A \mid P\ x\}$ .  
 Definition  $\text{constructive\_ground\_epsilon} (E : \exists x : A, P\ x) : A$   
 $\quad := \text{proj1\_sig} (\text{constructive\_indefinite\_ground\_description } E)$ .  
 Definition  $\text{constructive\_ground\_epsilon\_spec} (E : (\exists x, P\ x)) : P (\text{constructive\_ground\_epsilon } E)$   
 $\quad := \text{proj2\_sig} (\text{constructive\_indefinite\_ground\_description } E)$ .  
 End ConstructiveGroundEpsilon.

## Chapter 34

# Library **Coq.Logic.Decidable**

Properties of decidable propositions

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

**Theorem** `dec_not_not` :  $\forall P:\text{Prop}, \text{decidable } P \rightarrow (\neg P \rightarrow \text{False}) \rightarrow P$ .

**Theorem** `dec_True` : `decidable True`.

**Theorem** `dec_False` : `decidable False`.

**Theorem** `dec_or` :

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

**Theorem** `dec_and` :

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

**Theorem** `dec_not` :  $\forall A:\text{Prop}, \text{decidable } A \rightarrow \text{decidable } (\neg A)$ .

**Theorem** `dec_imp` :

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

**Theorem** `dec_iff` :

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

**Theorem** `not_not` :  $\forall P:\text{Prop}, \text{decidable } P \rightarrow \neg \neg P \rightarrow P$ .

**Theorem** `not_or` :  $\forall A B:\text{Prop}, \neg (A \vee B) \rightarrow \neg A \wedge \neg B$ .

**Theorem** `not_and` :  $\forall A B:\text{Prop}, \text{decidable } A \rightarrow \neg (A \wedge B) \rightarrow \neg A \vee \neg B$ .

**Theorem** `not_imp` :  $\forall A B:\text{Prop}, \text{decidable } A \rightarrow \neg (A \rightarrow B) \rightarrow A \wedge \neg B$ .

**Theorem** `imp_simp` :  $\forall A B:\text{Prop}, \text{decidable } A \rightarrow (A \rightarrow B) \rightarrow \neg A \vee B$ .

**Theorem** `not_iff` :

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

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

```

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.

```

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 35

# 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 36

# 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 [Carlström](#) 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 <-> AC\_ext, Mathematical Logic Quarterly, vol 50(3), pp 236-240, 2004.

### 36.1 Pred. Ext. + Rel. Axiom of Choice -> Excluded-Middle

**Section** `PredExt_RelChoice_imp_EM`.

The axiom of extensionality for predicates

**Definition** `PredicateExtensionality` :=

$\forall P Q : \text{bool} \rightarrow \text{Prop}, (\forall b : \text{bool}, P\ b \leftrightarrow Q\ b) \rightarrow P = Q.$

From predicate extensionality we get propositional extensionality hence proof-irrelevance

**Require 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

Require 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

Require 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.

## 36.2 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.

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  $A:\text{Type}$ .

Variables  $a1\ a2 : A$ .

We build the subset  $A'$  of  $A$  made of  $a1$  and  $a2$

Definition  $A' := @sigT\ A\ (\text{fun } x \Rightarrow x=a1 \vee x=a2)$ .

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` :  $\forall x:A', \exists y:\text{bool},$   
 $(\text{projT1 } x = a1 \wedge y = \text{true}) \vee (\text{projT1 } x = a2 \wedge y = \text{false}).$

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`.

### 36.3 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` :  $\forall A : \text{Type}, \text{inhabited } A \rightarrow (A \rightarrow \text{Prop}) \rightarrow A.$

**Hypothesis** `epsilon_spec` :

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

**Hypothesis** `epsilon_extensionality` :

$\forall (A:\text{Type}) (i:\text{inhabited } A) (P Q:A\rightarrow\text{Prop}),$   
 $(\forall a, P a \leftrightarrow Q a) \rightarrow \text{epsilon } A i P = \text{epsilon } A i Q.$

**Theorem** `extensional_epsilon_imp_EM` :  $\forall P:\text{Prop}, P \vee \neg P.$

**End** `ExtensionalEpsilon_imp_EM`.



## Chapter 37

# Library Coq.Logic.Epsilon

This file provides indefinite description under the form of Hilbert's epsilon operator; it does not assume classical logic.

```
Require Import ChoiceFacts.
```

```
Set Implicit Arguments.
```

Hilbert's epsilon: operator and specification in one statement

```
Axiom epsilon_statement :
```

```
  ∀ (A : Type) (P : A → Prop), inhabited A →  
    { x : A | (∃ x, P x) → P x }.
```

```
Lemma constructive_indefinite_description :
```

```
  ∀ (A : Type) (P : A → Prop),  
    (∃ x, P x) → { x : A | P x }.
```

```
Lemma small_drinkers'_paradox :
```

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

```
Theorem iota_statement :
```

```
  ∀ (A : Type) (P : A → Prop), inhabited A →  
    { x : A | (∃! x : A, P x) → P x }.
```

```
Lemma constructive_definite_description :
```

```
  ∀ (A : Type) (P : A → Prop),  
    (∃! x, P x) → { x : A | P x }.
```

Hilbert's epsilon operator and its specification

```
Definition epsilon (A : Type) (i : inhabited A) (P : A → Prop) : A  
:= proj1_sig (epsilon_statement P i).
```

```
Definition epsilon_spec (A : Type) (i : inhabited A) (P : A → Prop) :  
  (∃ x, P x) → P (epsilon i P)  
:= proj2_sig (epsilon_statement P i).
```

Church's iota operator and its specification

```
Definition iota (A : Type) (i : inhabited A) (P : A → Prop) : A
```

```

:= proj1_sig (iota_statement  $P$   $i$ ).
Definition iota_spec ( $A$  : Type) ( $i$ :inhabited  $A$ ) ( $P$  :  $A \rightarrow$  Prop) :
  ( $\exists!$   $x:A$ ,  $P$   $x$ )  $\rightarrow P$  (iota  $i$   $P$ )
:= proj2_sig (iota_statement  $P$   $i$ ).

```

## Chapter 38

# 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

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

`Set Implicit Arguments.`

`Section EqdepDec.`

`Variable A : Type.`

`Let comp (x y y':A) (eq1:x = y) (eq2:x = y') : y = y' :=  
 eq_ind _ (fun a => a = y') eq2 _ eq1.`

`Remark trans_sym_eq : ∀ (x y:A) (u:x = y), comp u u = eq_refl y.`

`Variable x : A.`

`Variable eq_dec : ∀ y:A, x = y ∨ x ≠ y.`

`Let nu (y:A) (u:x = y) : x = y :=  
 match eq_dec y with  
 | or_introl eqxy => eqxy  
 | or_intror neqxy => False_ind _ (neqxy u)  
 end.`

`Let nu_constant : ∀ (y:A) (u v:x = y), nu u = nu v.`

Qed.

Let  $nu\_inv (y:A) (v:x = y) : x = y := comp (nu (eq\_refl x)) v$ .

Remark  $nu\_left\_inv\_on : \forall (y:A) (u:x = y), nu\_inv (nu u) = u$ .

Theorem  $eq\_proofs\_unicity\_on : \forall (y:A) (p1 p2:x = y), p1 = p2$ .

Theorem  $K\_dec\_on :$

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

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  $inj\_right\_pair\_on :$

$\forall (P:A \rightarrow Prop) (y y':P x),$   
 $ex\_intro P x y = ex\_intro P x y' \rightarrow y = y'$ .

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  $K\_dec\_type :$

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

Theorem  $K\_dec\_set :$

$\forall A:Set,$   
 $(\forall x y:A, \{x = y\} + \{x \neq y\}) \rightarrow$   
 $\forall (x:A) (P:x = x \rightarrow Prop), P (eq\_refl x) \rightarrow \forall p:x = x, P p$ .

We deduce the *eq\_rect\_eq* axiom for (decidable) types **Theorem** *eq\_rect\_eq\_dec* :

```

∀ A:Type,
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (p:A) (Q:A → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

```

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

```

∀ A:Type,
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (P:A→Type) (p:A) (x y:P p), eq_dep A P p x p y → x = y.

```

**Theorem** *UIP\_dec* :

```

∀ (A:Type),
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (x y:A) (p1 p2:x = y), p1 = p2.

```

**Unset Implicit Arguments.**

### 38.1.1 Definition of the functor that builds properties of dependent equalities on decidable sets in **Type**

The signature of decidable sets in **Type**

**Module** **Type** *DECIDABLETYPE*.

**Axiom** *eq\_dec* : ∀ x y:U, {x = y} + {x ≠ y}.

**End** *DECIDABLETYPE*.

The module *DecidableEqDep* collects equality properties for decidable set in **Type**

**Module** *DECIDABLEEQDEP* (M:*DECIDABLETYPE*).

**Import** M.

Invariance by Substitution of Reflexive Equality Proofs

**Lemma** *eq\_rect\_eq* :

∀ (p:U) (Q:U → **Type**) (x:Q p) (h:p = p), x = eq\_rect p Q x p h.

Injectivity of Dependent Equality

**Theorem** *eq\_dep\_eq* :

∀ (P:U→**Type**) (p:U) (x y:P p), eq\_dep U P p x p y → x = y.

Uniqueness of Identity Proofs (UIP)

**Lemma** *UIP* : ∀ (x y:U) (p1 p2:x = y), p1 = p2.

Uniqueness of Reflexive Identity Proofs

**Lemma** *UIP\_refl* : ∀ (x:U) (p:x = x), p = eq\_refl x.

Streicher's axiom K

**Lemma** *Streicher\_K* :

∀ (x:U) (P:x = x → **Prop**), P (eq\_refl x) → ∀ p:x = x, P p.

Injectivity of equality on dependent pairs in **Type**

**Lemma** *inj\_pairT2* :

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

Proof-irrelevance on subsets of decidable sets

**Lemma** `inj_pairP2` :  
 $\forall (P:U \rightarrow \mathbf{Prop}) (x:U) (p\ q:P\ x),$   
 $\text{ex\_intro } P\ x\ p = \text{ex\_intro } P\ x\ q \rightarrow p = q.$

**End** `DECIDABLEEQDEP`.

### 38.1.2 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` :  
 $\forall (p:U) (Q:U \rightarrow \mathbf{Type}) (x:Q\ p) (h:p = p), x = \text{eq\_rect } p\ Q\ x\ p\ h.$

Injectivity of Dependent Equality

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

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

Proof-irrelevance on subsets of decidable sets

**Lemma** `inj_pairP2` :  
 $\forall (P:U \rightarrow \mathbf{Prop}) (x:U) (p\ q:P\ x),$   
 $\text{ex\_intro } P\ x\ p = \text{ex\_intro } P\ x\ q \rightarrow p = q.$

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 39

# 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`

### 39.1 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.
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.
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.

Notation eq_sigS_eq_dep := eq_sigT_eq_dep (compat "8.2").

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),

```

$\text{eq\_dep } p \ x \ q \ y \rightarrow \text{exist } P \ p \ x = \text{exist } P \ q \ y.$

Lemma `eq_sig_iff_eq_dep` :

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

Dependent equality is equivalent to a dependent pair of equalities

Set Implicit Arguments.

Lemma `eq_sigT_sig_eq` :  $\forall X \ P \ (x1 \ x2:X) \ H1 \ H2, \text{existT } P \ x1 \ H1 = \text{existT } P \ x2 \ H2 \leftrightarrow$   
 $\{H:x1=x2 \mid \text{rew } H \text{ in } H1 = H2\}.$

Lemma `eq_sigT_fst` :

$\forall X \ P \ (x1 \ x2:X) \ H1 \ H2 \ (H:\text{existT } P \ x1 \ H1 = \text{existT } P \ x2 \ H2), x1 = x2.$

Lemma `eq_sigT_snd` :

$\forall X \ P \ (x1 \ x2:X) \ H1 \ H2 \ (H:\text{existT } P \ x1 \ H1 = \text{existT } P \ x2 \ H2), \text{rew } (\text{eq\_sigT\_fst } H) \text{ in } H1 =$   
 $H2.$

Lemma `eq_sig_fst` :

$\forall X \ P \ (x1 \ x2:X) \ H1 \ H2 \ (H:\text{exist } P \ x1 \ H1 = \text{exist } P \ x2 \ H2), x1 = x2.$

Lemma `eq_sig_snd` :

$\forall X \ P \ (x1 \ x2:X) \ H1 \ H2 \ (H:\text{exist } P \ x1 \ H1 = \text{exist } P \ x2 \ H2), \text{rew } (\text{eq\_sig\_fst } H) \text{ in } H1 = H2.$

Unset Implicit Arguments.

Exported hints

Hint Resolve `eq_dep_intro`: core.

Hint Immediate `eq_dep_sym`: core.

## 39.2 Eq\_rect\_eq <-> Eq\_dep\_eq <-> UIP <-> UIP\_refl <-> K

Section Equivalences.

Variable `U:Type`.

Invariance by Substitution of Reflexive Equality Proofs

Definition `Eq_rect_eq_on` ( $p : U$ ) ( $Q : U \rightarrow \text{Type}$ ) ( $x : Q \ p$ ) :=

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

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` ( $P : U \rightarrow \text{Type}$ ) ( $p : U$ ) ( $x : P \ p$ ) :=

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

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 hlevelIntosn in <https://github.com/vladimirias/Foundations.git>).

**Theorem** UIP\_shift\_on  $(X : \text{Type}) (x : X) :$

UIP\_refl\_on\_  $X \ x \rightarrow \forall y : x = x, \text{UIP\_refl\_on\_} (x = x) \ y.$

**Theorem** UIP\_shift :  $\forall U, \text{UIP\_refl\_} U \rightarrow \forall x:U, \text{UIP\_refl\_} (x = x).$

**Section** Corollaries.

**Variable**  $U:\text{Type}.$

UIP implies the injectivity of equality on dependent pairs in Type

**Definition** Inj\_dep\_pair\_on  $(P : U \rightarrow \text{Type}) (p : U) (x : P \ p) :=$

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

**Definition** Inj\_dep\_pair :=  $\forall P \ p \ x, \text{Inj\_dep\_pair\_on } P \ p \ x.$

**Lemma** eq\_dep\_eq\_on\_\_inj\_pair2\_on  $(P : U \rightarrow \text{Type}) (p : U) (x : P \ p) :$

Eq\_dep\_eq\_on  $U \ P \ p \ x \rightarrow \text{Inj\_dep\_pair\_on } P \ p \ x.$

**Lemma** eq\_dep\_eq\_\_inj\_pair2 :  $\text{Eq\_dep\_eq } U \rightarrow \text{Inj\_dep\_pair}.$

**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.

### 39.3 Definition of the functor that builds properties of dependent equalities assuming axiom eq\_rect\_eq

**Module** Type EQDEPELIMINATION.

**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** EQDEPELIMINATION.

**Module** EQDEPTHEORY ( $M:\text{EQDEPELIMINATION}$ ).

**Section** Axioms.

**Variable**  $U:\text{Type}.$

Invariance by Substitution of Reflexive Equality Proofs

**Lemma** eq\_rect\_eq :

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

**Lemma** eq\_rec\_eq :

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

Injectivity of Dependent Equality

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

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 40

# 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

```
Hint Resolve eq_dep_eq: eqdep.
```

```
Hint Resolve inj_pair2 inj_pairT2: eqdep.
```

## Chapter 41

# 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
| []  $\Rightarrow$  True
|  $b :: l \Rightarrow$ 
     $Y\ P\ l \wedge$ 
    if  $b$  then inductively_barred  $P$  (false:: $l$ ) else  $\neg$  inductively_barred  $P$  (false:: $l$ )
end.

Lemma Y-unique :  $\forall\ P\ l1\ l2, \text{length } l1 = \text{length } l2 \rightarrow Y\ P\ l1 \rightarrow Y\ P\ l2 \rightarrow l1 = l2$ .

 $X$  is the translation of  $Y$  as a predicate
Definition X  $P\ n := \exists\ l, \text{length } l = n \wedge Y\ P\ (\text{true} :: l)$ .

Lemma Y-approx :  $\forall\ P\ l, \text{approx } (X\ P)\ l \rightarrow Y\ P\ l$ .

Theorem WeakFanTheorem :  $\forall\ P, \text{barred } P \rightarrow \text{inductively\_barred } P\ []$ .

```



## Chapter 42

# 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 Omega.
```

*is\_path\_from*  $P\ n\ l$  means that there exists a path of length  $n$  from  $l$  on which  $P$  does not hold

```
Inductive is_path_from (P:list bool → Prop) : nat → list bool → Prop :=  
| here l : ¬ P l → is_path_from P 0 l  
| next_left l n : ¬ P l → is_path_from P n (true::l) → is_path_from P (S n) l  
| next_right l n : ¬ P l → is_path_from P n (false::l) → is_path_from P (S n) l.
```

We give the characterization of *is\_path\_from* in terms of a more common arithmetical formula

**Proposition** *is\_path\_from\_characterization*  $P\ n\ l$  :

$\text{is\_path\_from } P\ n\ l \leftrightarrow \exists l', \text{ length } l' = n \wedge \forall n', n' \leq n \rightarrow \neg P (\text{rev } (\text{firstn } n' l') ++ l).$

*infinite\_from*  $P\ l$  means that we can find arbitrary long paths along which  $P$  does not hold above  $l$

**Definition** *infinite\_from*  $(P:\text{list bool} \rightarrow \text{Prop})\ l := \forall n, \text{is\_path\_from } P\ n\ l.$

*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*  $(P:\text{list bool} \rightarrow \text{Prop}) :=$

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

*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 → Prop) : nat → list bool → Prop :=  
| now_at l n : P l → inductively_barred_at P n l
```

```

| propagate_at l n :
  inductively_barred_at P n (true::l) →
  inductively_barred_at P n (false::l) →
  inductively_barred_at P (S n) l.

```

The proof proceeds by building a set  $Y$  of finite paths approximating either the smallest unbarred infinite path in  $P$ , if there is one (taking  $true > false$ ), or the path  $true::true::\dots$  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 :
  ∀ P, (∀ l, P l ∨ ¬ P l) → ∀ n l, inductively_barred_at P n l ∨ ¬ 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` :  $\forall P, (\forall l, P\ l \vee \neg P\ l) \rightarrow$   
`infinite_from`  $P\ [] \rightarrow \text{has\_infinite\_path } P.$

## Chapter 43

### 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  $\rightarrow$  Type} (H :  $\forall x : A, B\ x = C\ x$ ) :  $(\forall x, B\ x) = (\forall x, C\ x).$

```
Lemma forall_extensionalityP {A} {B C : A  $\rightarrow$  Prop} (H :  $\forall x : A, B\ x = C\ x$ )  
:  $(\forall x, B\ x) = (\forall x, C\ x).$ 
```

```
Lemma forall_extensionalityS {A} {B C : A  $\rightarrow$  Set} (H :  $\forall x : A, B\ x = C\ x$ )  
:  $(\forall x, B\ x) = (\forall x, C\ x).$ 
```

Apply *functional\_extensionality*, introducing variable x.

```
Tactic Notation "extensionality" ident(x) :=  
  match goal with  
  |  $\vdash ?X = ?Y$  |  $\Rightarrow$   
    (apply (@functional_extensionality _ _ X Y) ||  
     apply (@functional_extensionality_dep _ _ X Y) ||  
     apply forall_extensionalityP ||
```

```

    apply forall_extensionalityS ||
    apply forall_extensionality) ; intro  $x$ 
end.

```

Eta expansion follows from extensionality.

**Lemma** `eta_expansion_dep`  $\{A\} \{B : A \rightarrow \text{Type}\} (f : \forall x : A, B\ x) :$   
 $f = \text{fun } x \Rightarrow f\ x.$

**Lemma** `eta_expansion`  $\{A\ B\} (f : A \rightarrow B) : f = \text{fun } x \Rightarrow f\ x.$

## 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

**Record** `Delta`  $A := \{ \text{pi1}:A; \text{pi2}:A; \text{eq}:\text{pi1}=\text{pi2} \}$ .

**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` :  $\forall A, \text{is\_inverse}\ (A:=A)\ \text{delta}\ \text{pi1}$ .

**Lemma** `diagonal_inverse2` :  $\forall A, \text{is\_inverse}\ (A:=A)\ \text{delta}\ \text{pi2}$ .

Functional extensionality

Equality of projections from diagonal

Unicity of bijection inverse

Bijectivity of bijective composition

**Definition** `action A B C (f:A→B) := (fun h:B→C => fun x => h (f x)).`

## 44.2 Functional extensionality <-> Equality of projections from diagonal

**Theorem** `FuncExt_iff_EqDeltaProjs : FunctionalExtensionality <=> EqDeltaProjs.`

## 44.3 Functional extensionality <-> Unicity of bijection inverse

**Lemma** `FuncExt_UniqInverse : FunctionalExtensionality -> UniqueInverse.`

**Lemma** `UniqInverse_EqDeltaProjs : UniqueInverse -> EqDeltaProjs.`

**Theorem** `FuncExt_iff_UniqInverse : FunctionalExtensionality <=> UniqueInverse.`

## 44.4 Functional extensionality <-> Bijectivity of bijective composition

**Lemma** `FuncExt_BijComp : FunctionalExtensionality -> BijectivityBijectiveComp.`

**Lemma** `BijComp_FuncExt : BijectivityBijectiveComp -> FunctionalExtensionality.`

## Chapter 45

# 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 ( $\forall A, \sim\sim A \setminus \sim A$ ) 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 *PropNegType* 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 <sup>1</sup>).

## 45.1 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*.

### 45.1.1 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** " $f \cdot_0$ "  $[A]$  :=  $(appU0 \text{ _ _ } f A)$ .

### 45.1.2 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.
```

### 45.1.3 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**  $I (x : El1\ U) : U0 :=$

$(\forall_0^1 i : U \rightarrow_1 u0, le\ i\ x \rightarrow_0 i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 x)) \rightarrow_0 F$

## Proof

**Lemma**  $\Omega : El0\ (\forall_0^1 i : U \rightarrow_1 u0, induct\ i \rightarrow_0 i \cdot_1 WF)$ .

**Proof.**

```

refine ( $\lambda_0^1 i, \lambda_0 y, -$ ).
refine ( $y \cdot_0 [-] \cdot_0 -$ ).
unfold le, WF, induct. simplify.
refine ( $\lambda_0^1 x, \lambda_0 h0, -$ ). simplify.
refine ( $y \cdot_0 [-] \cdot_0 -$ ).
unfold le. simplify.
unfold sb at 1. simplify.
unfold le' at 1. simplify.
exact h0.

```

**Qed.**

**Lemma**  $lemma1 : El0\ (induct\ (\lambda_1 u, I\ u))$ .

**Proof.**

```

unfold induct.
refine ( $\lambda_0^1 x, \lambda_0 p, -$ ). simplify.
refine ( $\lambda_0 q, -$ ).
assert ( $El0\ (I\ (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 x)))$  as h.
{ generalize ( $q \cdot_0 [\lambda_1 u, I\ u] \cdot_0 p$ ). simplify.
  intros q'.
  exact q'. }
refine ( $h \cdot_0 -$ ).
refine ( $\lambda_0^1 i, -$ ).
refine ( $\lambda_0 h', -$ ).
generalize ( $q \cdot_0 [\lambda_1 y, i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 y)]$ ). simplify.
intros q'.
refine ( $q' \cdot_0 -$ ). clear q'.
unfold le at 1 in h'. simplify_in h'.

```



**Hypothesis**  $u2u1\_unit : \forall (c:U2), c \rightarrow u2u1\ c$ .

$u2u1\_counit$  and  $u2u1\_coherent$  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**  $u2u1\_counit : \forall (F:U1 \rightarrow U1), u2u1\ (\forall A, F\ A) \rightarrow (\forall A, F\ A)$ .

**Hypothesis**  $u2u1\_coherent : \forall (F:U1 \rightarrow U1) (f:\forall x:U1, F\ x) (x:U1),$   
 $u2u1\_counit\ _\_ (u2u1\_unit\ _\_ f)\ x = f\ x$ .

**$U0$  is a retract of  $U1$**

**Variable**  $u0u1 : U0 \rightarrow U1$ .

**Variable**  $u1u0 : U1 \rightarrow U0$ .

**Hypothesis**  $u1u0\_unit : \forall (b:U1), b \rightarrow u0u1\ (u1u0\ b)$ .

**Hypothesis**  $u1u0\_counit : \forall (b:U1), u0u1\ (u1u0\ b) \rightarrow b$ .

### 45.2.1 Paradox

**Theorem**  $paradox : \forall F:U1, F$ .

**Proof.**

intros  $F$ .

*Generic.paradox*  $h$ .

Large universe + exact  $U1$ .

+ exact (fun  $X \Rightarrow X$ ).

+ *cbn*. exact (fun  $u\ F \Rightarrow \forall x:u, F\ x$ ).

+ *cbn*. exact (fun  $\_ \_ x \Rightarrow x$ ).

+ *cbn*. exact (fun  $\_ \_ x \Rightarrow x$ ).

+ *cbn*. exact (fun  $F \Rightarrow u2u1\ (\forall x, F\ x)$ ).

+ *cbn*. exact (fun  $\_ x \Rightarrow u2u1\_unit\ \_ x$ ).

+ *cbn*. exact (fun  $\_ x \Rightarrow u2u1\_counit\ \_ x$ ).

Small universe + exact  $U0$ .

The interpretation of the small universe is the image of  $U0$  in  $U1$ . + *cbn*. exact (fun  $X \Rightarrow u0u1\ X$ ).

+ *cbn*. exact (fun  $u\ F \Rightarrow u1u0\ (\forall x:(u0u1\ u), u0u1\ (F\ x))$ ).

+ *cbn*. exact (fun  $u\ F \Rightarrow u1u0\ (\forall x:u, u0u1\ (F\ x))$ ).

+ *cbn*. exact (u1u0  $F$ ).

+ *cbn* in  $h$ .

exact (u1u0\_counit  $\_ h$ ).

+ *cbn*. *easy*.

+ *cbn*. intros \*\*. now rewrite  $u2u1\_coherent$ .

+ *cbn*. intros  $\times x$ . exact (u1u0\_unit  $\_ x$ ).

+ *cbn*. intros  $\times x$ . exact (u1u0\_counit  $\_ x$ ).

+ *cbn*. intros  $\times x$ . exact (u1u0\_unit  $\_ x$ ).

+ *cbn*. intros  $\times x$ . exact (u1u0\_counit  $\_ x$ ).

**Qed.**

End Paradox.

End NORETRACTTOIMPREDICATIVEUNIVERSE.

### 45.3 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.

#### 45.3.1 Monadic modality

Section Paradox.

Variable  $M : \text{Prop} \rightarrow \text{Prop}$ .

Hypothesis  $unit : \forall A:\text{Prop}, A \rightarrow M A$ .

Hypothesis  $join : \forall A:\text{Prop}, M (M A) \rightarrow M A$ .

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.

eauto.

Qed.

#### 45.3.2 The universe of modal propositions

Definition  $M\text{Prop} := \{ P:\text{Prop} \mid M P \rightarrow P \}$ .

Definition  $El : M\text{Prop} \rightarrow \text{Prop} := @proj1\_sig \_ \_$ .

Lemma  $modal : \forall P:M\text{Prop}, M(El P) \rightarrow El P$ .

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  $(\forall x:A, El (P x))$ .

+ intros  $h x$ .

  eapply *strength* in  $h$ .

  eauto using *modal*.

Defined.

#### 45.3.3 Retract of the modal fragment of **Prop** in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

```

Variable bool : MProp.
Variable p2b : MProp → El bool.
Variable b2p : El bool → MProp.
Hypothesis p2p1 : ∀ A:MProp, El (b2p (p2b A)) → El A.
Hypothesis p2p2 : ∀ A:MProp, El A → El (b2p (p2b A)).

```

#### 45.3.4 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.

### 45.4 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.

#### 45.4.1 The universe of negative propositions.

Definition  $\mathbf{NProp} := \{ P:\mathbf{Prop} \mid \sim\sim P \rightarrow P \}.$

```
Definition El : NProp → Prop := @proj1_sig _.
```

Section Paradox.

#### 45.4.2 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  $p2b$  :  $\text{NProp} \rightarrow \text{El } \text{bool}$ .

Variable  $b2p$  :  $\text{El } \text{bool} \rightarrow \text{NProp}$ .

$$\text{Hypothesis } p2p1 : \forall A:\mathbf{NProp}, \text{El } (b2p \ (p2b \ A)) \rightarrow \text{El } A.$$

**Hypothesis**  $p2p2$  :  $\forall A:\mathbf{NProp}, \text{El } A \rightarrow \text{El } (b2p \ (p2b \ A))$ .

### 45.4.3 Paradox

**Theorem paradox** :  $\forall B:\text{NProp}, \text{El } B.$

Proof.

intros  $B$ .

```
unshelve (refine ((fun h => _) (NoRetractToModalProposition.paradox _ _ _ _ _ _ _ _ _ _))).
```

$$+ \text{exact } (\text{fun } P \Rightarrow \sim\sim P).$$

- + exact *bool*.

- + exact *p2b*.

$$+ \text{exact } b2p.$$

+ exact  $B$ .

- + exact  $h$ .

+ *cbn.* auto. $+ cbn.$  auto. $+ cbn.$  auto.

+ auto.

+ auto.

Qed.

End Paradox.

End NoRETRACTToNEGATIVEPROP.

### 45.5 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.



```

Definition NProp := { P:Prop | P → P}.
Definition El : NProp → Prop := @proj1_sig _ _ .
Section MParadox.

```

```

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)).

```

**Theorem** mparadox :  $\forall B:\mathbf{NProp}, \text{El } B$ .  
**Proof.**

#### 45.5.4 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**  $\rightarrow$  *bool*.  
**Variable** *b2p* : *bool*  $\rightarrow$  **Prop**.  
**Hypothesis** *p2p1* :  $\forall A:\mathbf{Prop}, b2p (p2b A) \rightarrow A$ .  
**Hypothesis** *p2p2* :  $\forall A:\mathbf{Prop}, A \rightarrow b2p (p2b A)$ .

### 45.5.5 Paradox

**Theorem** `paradox` :  $\forall B:\text{Prop}, B$ .

**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`.

## 45.6 Large universes are no 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`.

### 45.6.1 Assumption of a retract from `Type` into `Prop`

**Variable** `down` : `Type1`  $\rightarrow$  `Prop`.

**Variable** `up` : `Prop`  $\rightarrow$  `Type1`.

**Hypothesis** `up_down` :  $\forall (A:\text{Type1}), \text{up } (\text{down } A) = A \text{ :> Type1}$ .

### 45.6.2 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  $\Rightarrow$  X).
+ cbn. exact (fun A P  $\Rightarrow \forall x:A, P\ x$ ).
+ cbn. exact (fun A P  $\Rightarrow \forall x:A, P\ x$ ).
+ cbn. exact P.
+ exact h.
+ cbn. easy.
+ cbn.
  intros F f A.
  destruct (up_down A). cbn.
  reflexivity.
+ cbn. exact (fun _ _ x  $\Rightarrow x$ ).
+ cbn. exact (fun _ _ x  $\Rightarrow x$ ).
+ cbn. exact (fun _ _ x  $\Rightarrow x$ ).
+ cbn. exact (fun _ _ x  $\Rightarrow x$ ).
Qed.
End Paradox.
End NORETRACTFROMTYPETOPROP.

```

## 45.7 $A \neq \text{Type}$

No Coq universe can be equal to one of its elements.

```

Module TYPENEQSMALLTYPE.
Unset Universe Polymorphism.
Section Paradox.

```

### 45.7.1 Universe $U$ is equal to one of its elements.

```

Let U := Type.
Variable A:U.
Hypothesis h : U=A.

```

### 45.7.2 Universe $U$ is a retract of $A$

The following context is actually sufficient for the paradox to hold. The hypothesis  $h:U=A$  is only used to define *down*, *up* and *up\_down*.

```

Let down (X:U) : A := @eq_rect _ _ (fun X  $\Rightarrow$  X) X _ h.
Let up (X:A) : U := @eq_rect_r _ _ (fun X  $\Rightarrow$  X) X _ h.
Lemma up_down :  $\forall (X:U), up\ (down\ X) = X$ .

```

Proof.

```
unfold up,down.  
rewrite ← h.  
reflexivity.
```

Qed.

Theorem paradox : False.

Proof.

```
Generic.paradox p.  
  Large universe      + exact U.  
+ exact (fun X ⇒ X).  
+ cbn. exact (fun X F ⇒ ∀ x:X, F x).  
+ cbn. exact (fun _ _ x ⇒ x).  
+ cbn. exact (fun _ _ x ⇒ x).  
+ exact (fun F ⇒ ∀ x:A, F (up x)).  
+ cbn. exact (fun _ f ⇒ fun x:A ⇒ f (up x)).  
+ cbn. intros × f X.  
  specialize (f (down X)).  
  rewrite up_down in f.  
  exact f.  
  Small universe      + exact A.  
  The interpretation of A as a universe is U.      + cbn. exact up.  
+ cbn. exact (fun _ F ⇒ down (∀ x, up (F x))).  
+ cbn. exact (fun _ F ⇒ down (∀ x, up (F x))).  
+ cbn. exact (down False).  
+ rewrite up_down in p.  
  exact p.  
+ cbn. easy.  
+ cbn. intros ? f X.  
  destruct (up_down X). cbn.  
  reflexivity.  
+ cbn. intros ? ? f.  
  rewrite up_down.  
  exact f.  
+ cbn. intros ? ? f.  
  rewrite up_down in f.  
  exact f.  
+ cbn. intros ? ? f.  
  rewrite up_down.  
  exact f.  
+ cbn. intros ? ? f.  
  rewrite up_down in f.  
  exact f.
```

Qed.

End Paradox.

End TYPENEQSMALLTYPE.

## 45.8 $\text{Prop} \neq \text{Type}$ .

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 46

# Library **Coq.Logic.IndefiniteDescription**

This file provides a constructive form of indefinite description that allows building choice functions; this is weaker than Hilbert's epsilon operator (which implies weakly classical properties) but stronger than the axiom of choice (which cannot be used outside the context of a theorem proof).

```
Require Import ChoiceFacts.
```

```
Set Implicit Arguments.
```

```
Axiom constructive_indefinite_description :
```

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

```
Lemma constructive_definite_description :
```

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

```
Lemma functional_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 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:Type, B  $\rightarrow$  Prop :=  
JMeq\_refl : JMeq x x.

**Hint Resolve** JMeq\_refl.

**Definition** JMeq\_hom {A : Type} (x y : A) := JMeq x y.

**Lemma** JMeq\_sym :  $\forall$  (A B:Type) (x:A) (y:B), JMeq x y  $\rightarrow$  JMeq y x.

**Hint Immediate** JMeq\_sym.

**Lemma** JMeq\_trans :

$\forall$  (A B C:Type) (x:A) (y:B) (z:C), JMeq x y  $\rightarrow$  JMeq y z  $\rightarrow$  JMeq x z.

**Axiom** JMeq\_eq :  $\forall$  (A:Type) (x y:A), JMeq x y  $\rightarrow$  x = y.

**Lemma** JMeq\_ind :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Prop),  
P x  $\rightarrow \forall$  y, JMeq x y  $\rightarrow$  P y.

**Lemma** JMeq\_rec :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Set),  
P x  $\rightarrow \forall$  y, JMeq x y  $\rightarrow$  P y.

**Lemma** JMeq\_rect :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Type),  
P x  $\rightarrow \forall$  y, JMeq x y  $\rightarrow$  P y.

**Lemma** JMeq\_ind\_r :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Prop),  
P x  $\rightarrow \forall$  y, JMeq y x  $\rightarrow$  P y.

**Lemma** JMeq\_rec\_r :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Set),  
P x  $\rightarrow \forall$  y, JMeq y x  $\rightarrow$  P y.

**Lemma** JMeq\_rect\_r :  $\forall$  (A:Type) (x:A) (P:A  $\rightarrow$  Type),  
P x  $\rightarrow \forall$  y, JMeq y x  $\rightarrow$  P y.

**Lemma** JMeq\_congr :

$\forall (A:\text{Type}) (x:A) (B:\text{Type}) (f:A \rightarrow B) (y:A), \text{JMeq } x \ y \rightarrow f \ x = f \ y.$   
 $\text{JMeq}$  is equivalent to  $\text{eq\_dep Type (fun } X \Rightarrow X)$

**Require Import** Eqdep.

**Lemma**  $\text{JMeq\_eq\_dep\_id} :$   
 $\forall (A \ B:\text{Type}) (x:A) (y:B), \text{JMeq } x \ y \rightarrow \text{eq\_dep Type (fun } X \Rightarrow X) \ A \ x \ B \ y.$

**Lemma**  $\text{eq\_dep\_id\_JMeq} :$   
 $\forall (A \ B:\text{Type}) (x:A) (y:B), \text{eq\_dep Type (fun } X \Rightarrow X) \ A \ x \ B \ y \rightarrow \text{JMeq } x \ y.$   
 $\text{eq\_dep } U \ P \ p \ x \ q \ y$  is strictly finer than  $\text{JMeq } (P \ p) \ x \ (P \ q) \ y$

**Lemma**  $\text{eq\_dep\_JMeq} :$   
 $\forall U \ P \ p \ x \ q \ y, \text{eq\_dep } U \ P \ p \ x \ q \ y \rightarrow \text{JMeq } x \ y.$

**Lemma**  $\text{eq\_dep\_strictly\_stronger\_JMeq} :$   
 $\exists U \ P \ p \ q \ x \ y, \text{JMeq } x \ y \wedge \neg \text{eq\_dep } U \ P \ p \ x \ q \ y.$

However, when the dependencies are equal,  $\text{JMeq } (P \ p) \ x \ (P \ q) \ y$  is as strong as  $\text{eq\_dep } U \ P \ p \ x \ q \ y$  (this uses  $\text{JMeq\_eq}$ )

**Lemma**  $\text{JMeq\_eq\_dep} :$   
 $\forall U \ (P:U \rightarrow \text{Prop}) \ p \ q \ (x:P \ p) \ (y:P \ q),$   
 $p = q \rightarrow \text{JMeq } x \ y \rightarrow \text{eq\_dep } U \ P \ p \ x \ q \ y.$

**Notation**  $\text{sym\_JMeq} := \text{JMeq\_sym (only parsing)}.$   
**Notation**  $\text{trans\_JMeq} := \text{JMeq\_trans (only parsing)}.$



## Chapter 48

# 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 ( $M:\text{PROOFIRRELEVANCE}$ ).
```

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 49

# Library **Coq.Logic.ProofIrrelevance**

This file axiomatizes proof-irrelevance and derives some consequences

```
Require Import ProofIrrelevanceFacts.
```

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

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

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

```
Export ProofIrrelevanceTheory.
```

## Chapter 50

# 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 51

# Library **Coq.Logic.SetIsType**

### 51.1 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 52

# Library **Coq.Logic.FinFun**

### 52.1 Functions on finite domains

Main result : for functions  $f:A \rightarrow A$  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  $\{A\ B\}$   $(f : A \rightarrow B) :=$   
 $\forall x\ y, f\ x = f\ y \rightarrow x = y.$

**Definition** Surjective  $\{A\ B\}$   $(f : A \rightarrow B) :=$   
 $\forall y, \exists x, f\ x = y.$

**Definition** Bijective  $\{A\ B\}$   $(f : A \rightarrow B) :=$   
 $\exists g:B \rightarrow A, (\forall x, g\ (f\ x) = x) \wedge (\forall y, f\ (g\ y) = y).$

Finiteness is defined here via exhaustive list enumeration

**Definition** Full  $\{A:\text{Type}\}$   $(l:\text{list } A) := \forall a:A, \text{In } a\ l.$

**Definition** Finite  $(A:\text{Type}) := \exists (l:\text{list } A), \text{Full } l.$

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  $\{A:\text{Type}\}$   $(l:\text{list } A) := \text{NoDup } l \wedge \text{Full } l.$

**Definition** Finite'  $(A:\text{Type}) := \exists (l:\text{list } A), \text{Listing } l.$

**Lemma** Finite\_alt  $A\ (d:\text{decidable\_eq } A) : \text{Finite } A \leftrightarrow \text{Finite' } A.$

Injections characterized in term of lists

**Lemma** Injective\_map\_NoDup  $A\ B\ (f:A \rightarrow B)\ (l:\text{list } A) :$   
 $\text{Injective } f \rightarrow \text{NoDup } l \rightarrow \text{NoDup } (\text{map } f\ l).$

**Lemma** Injective\_list\_carac  $A\ B\ (d:\text{decidable\_eq } A)\ (f:A \rightarrow B) :$   
 $\text{Injective } f \leftrightarrow (\forall l, \text{NoDup } l \rightarrow \text{NoDup } (\text{map } f\ l)).$

**Lemma** Injective\_carac  $A\ B\ (l:\text{list } A) : \text{Listing } l \rightarrow$   
 $\forall (f:A \rightarrow B), \text{Injective } f \leftrightarrow \text{NoDup } (\text{map } f\ l).$

Surjection characterized in term of lists

**Lemma** `Surjective_list_carac`  $A\ B\ (f:A \rightarrow B)$ :  
 $\text{Surjective } f \leftrightarrow (\forall lB, \exists lA, \text{incl } lB\ (\text{map } f\ lA)).$

**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 : *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** `bInjective`  $n\ (f:\text{nat} \rightarrow \text{nat}) :=$   
 $\forall x\ y, x < n \rightarrow y < n \rightarrow f\ x = f\ y \rightarrow x = y.$

**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 *Fin.t n*.

**Module** `FIN2RESTRICT`.

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

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

**Definition** `f2n_ok`  $n\ (x:\text{Fin.t } n) : f2n\ x < n := \text{proj2\_sig } (\text{Fin.to\_nat } x).$

**Definition** `n2f_f2n`  $: \forall n\ x, n2f\ (f2n\_ok\ x) = x := @\text{Fin.of\_nat\_to\_nat\_inv}.$

**Definition** `f2n_n2f`  $x\ n\ h : f2n\ (n2f\ h) = x := \text{f\_equal } (@\text{proj1\_sig } \_)\ (@\text{Fin.to\_nat\_of\_nat } x\ n\ h).$

```

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 n (@extend n f)  $\leftrightarrow$  Surjective f.
Lemma extend_injective n f :
  bInjective n (@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 53

# 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 54

# Library **Coq.Unicode.Utf8\_core**

```
Notation "∀ x .. y , P" := (∀ x, .. (∀ y, P) ..)
  (at level 200, x binder, y binder, right associativity) : type_scope.
Notation "∃ x .. y , P" := (∃ x, .. (∃ y, P) ..)
  (at level 200, x binder, y binder, right associativity) : type_scope.
Notation "x ∨ y" := (x ∨ y) (at level 85, right associativity) : type_scope.
Notation "x ∧ y" := (x ∧ y) (at level 80, right associativity) : type_scope.
Notation "x → y" := (x → y)
  (at level 99, y at level 200, right associativity) : type_scope.
Notation "x ↔ y" := (x ↔ y) (at level 95, no associativity) : type_scope.
Notation "¬ x" := (¬ x) (at level 75, right associativity) : type_scope.
Notation "x ≠ y" := (x ≠ y) (at level 70) : type_scope.
Notation "λ' x .. y , t" := (fun x ⇒ .. (fun y ⇒ t) ..)
  (at level 200, x binder, y binder, right associativity).
```

## Chapter 55

# Library **Coq.Init.Datatypes**

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

### 55.1 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.
```

### 55.2 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.
Hint Resolve andb_prop: bool.
Lemma andb_true_intro :
  ∀ b1 b2:bool, b1 = true ∧ b2 = true → andb b1 b2 = true.
Hint Resolve andb_true_intro: bool.

Interpretation of booleans as propositions
Inductive eq_true : bool → Prop := is_eq_true : eq_true true.
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 = ¬P.
Inductive BoolSpec (P Q : Prop) : bool → Prop :=
| BoolSpecT : P → BoolSpec P Q true
| BoolSpecF : Q → BoolSpec P Q false.
Hint Constructors BoolSpec.

```

### 55.3 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 *nat\_scope* with *nat*.

## 55.4 Container datatypes

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

```

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*

```

Inductive sum (A B:Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.

```

Notation " $x + y$ " := (*sum x y*) : *type\_scope*.

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

```

Inductive prod (A B:Type) : Type :=
  pair : A → B → prod A B.

```

Add Printing Let *prod*.

Notation " $x * y$ " := (*prod x y*) : *type\_scope*.

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.

Hint Resolve *pair inl inr*: *core*.

Lemma surjective\_pairing :

```

  ∀ (A B:Type) (p:A × B), p = pair (fst p) (snd p).

```

```

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

Definition prod_uncurry (A B C:Type) (f:prod A B → C)
  (x:A) (y:B) : C := f (pair x y).

Definition prod_curry (A B C:Type) (f:A → B → C)
  (p:prod A B) : C := match p with
    | pair x y ⇒ f x y
  end.

```

Polymorphic lists and some operations

```

Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.

Infix "::" := cons (at level 60, right associativity) : list_scope.
Delimit Scope list_scope with list.

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.

```

## 55.5 The comparison datatype

```

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

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

```

```

| Gt ⇒ Lt
end.

```

**Lemma** `CompOpp_involutive` :  $\forall c, \text{CompOpp } (\text{CompOpp } c) = c$ .

**Lemma** `CompOpp_inj` :  $\forall c \ c', \text{CompOpp } c = \text{CompOpp } c' \rightarrow c = c'$ .

**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.
Hint Constructors CompareSpec.

```

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.
Hint Constructors CompareSpecT.

```

**Lemma** `CompareSpec2Type` :  $\forall \text{Peq Plt Pgt } c,$   
 $\text{CompareSpec Peq Plt Pgt } c \rightarrow \text{CompareSpecT Peq Plt Pgt } c.$

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).`

**Hint Unfold** `CompSpec CompSpecT`.

**Lemma** `CompSpec2Type` :  $\forall A \ (eq \ lt : A \rightarrow A \rightarrow \text{Prop}) \ x \ y \ c,$   
 $\text{CompSpec } eq \ lt \ x \ y \ c \rightarrow \text{CompSpecT } eq \ lt \ x \ y \ c.$

## 55.6 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* whose sole inhabitant is denoted *refl\_identity* *A* *a*

**Inductive** `identity` (A : Type) (a : A) : A → Type :=  
`identity_refl : identity a a.`

**Hint Resolve** *identity\_refl*: *core*.

Identity type

**Definition** ID :=  $\forall A:\mathbf{Type}, A \rightarrow A$ .

**Definition** id : ID := fun A x  $\Rightarrow$  x.

**Definition** IDProp :=  $\forall A:\mathbf{Prop}, A \rightarrow A$ .

**Definition** idProp : IDProp := fun A x  $\Rightarrow$  x.

## Chapter 56

# 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** *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* :

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

**Defined**.

**Definition** *identity\_rec\_r* :

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

**Defined**.

**Definition** *identity\_rect\_r* :

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

**Defined**.

**Hint Immediate** *identity\_sym not\_identity\_sym*: *core*.



```
Notation refl_id := identity_refl (compat "8.3").
Notation sym_id := identity_sym (compat "8.3").
Notation trans_id := identity_trans (compat "8.3").
Notation sym_not_id := not_identity_sym (compat "8.3").
```

## Chapter 57

# Library Coq.Init.Logic

Set Implicit Arguments.

Require Export Notations.

Notation "A -> B" := ( $\forall$  ( $_$  : A), B) : type\_scope.

### 57.1 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:Prop) := A  $\rightarrow$  False.

Notation "~ x" := (not x) : type\_scope.

Hint Unfold not: core.

*and* A B, written  $A \wedge B$ , is the conjunction of A and B

*conj* p q is a proof of  $A \wedge B$  as soon as p is a proof of A and q a proof of B

*proj1* and *proj2* are first and second projections of a conjunction

Inductive and (A B:Prop) : Prop :=  
conj : A  $\rightarrow$  B  $\rightarrow$  A  $\wedge$  B

where "A /\ 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.

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

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  $Q$

**Definition**  $IF\_then\_else\ (P\ Q\ R:Prop) := P \wedge Q \vee \neg P \wedge R$ .

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

## 57.2 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**  $ex\ (A:Type)\ (P:A \rightarrow Prop) : Prop :=$   
 $ex\_intro : \forall\ x:A,\ P\ x \rightarrow ex\ (A:=A)\ P$ .

**Inductive**  $ex2\ (A:Type)\ (P\ Q:A \rightarrow Prop) : Prop :=$   
 $ex\_intro2 : \forall\ x:A,\ P\ x \rightarrow Q\ x \rightarrow ex2\ (A:=A)\ P\ Q$ .

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

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

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

**Notation** "'exists2' x : t , p & q" :=  $(ex2\ (fun\ x:t \Rightarrow p)\ (fun\ x:t \Rightarrow q))$   
(at level 200, *x ident*, *t* at level 200, *p* at level 200, right associativity,  
*format* "'[ 'exists2' ' / ' x : t , ' / ' ' p & ' / ' q ]' ' '")  
: *type\_scope*.

Derived rules for universal quantification

**Section** *universal\_quantification*.

**Variable**  $A : Type$ .

**Variable**  $P : A \rightarrow Prop$ .

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

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

**End** *universal\_quantification*.

## 57.3 Equality

$eq\ x\ y$ , or simply  $x=y$  expresses the equality of  $x$  and  $y$ . Both  $x$  and  $y$  must belong to the same type  $A$ . The definition is inductive and states the reflexivity of the equality. The others properties (symmetry, transitivity, replacement of equals by equals) are proved below. The type of  $x$  and  $y$

can be made explicit using the notation  $x = y :> A$ . This is Leibniz equality as it expresses that  $x$  and  $y$  are equal iff every property on  $A$  which is true of  $x$  is also true of  $y$

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

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

```
Notation "x = y" := (x = y :>_) : type_scope.
```

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

```
Notation "x <> y" := (x ≠ y :>_) : type_scope.
```

```
Hint Resolve I conj or_introl or_intror : core.
```

```
Hint Resolve eq_refl: core.
```

```
Hint Resolve ex_intro ex_intro2: core.
```

```
Section Logic_lemmas.
```

```
Theorem absurd : ∀ A C:Prop, A → ¬ A → C.
```

```
Section equality.
```

```
Variables A B : Type.
```

```
Variable f : A → B.
```

```
Variables x y z : A.
```

```
Theorem eq_sym : x = y → y = x.
```

```
Theorem eq_trans : x = y → y = z → x = z.
```

```
Theorem f_equal : x = y → f x = f y.
```

```
Theorem not_eq_sym : x ≠ y → y ≠ x.
```

```
End equality.
```

```
Definition eq_ind_r :
```

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

```
Defined.
```

```
Definition eq_rec_r :
```

```
  ∀ (A:Type) (x:A) (P:A → Set), P x → ∀ y:A, y = x → P y.
```

```
Defined.
```

```
Definition eq_rect_r :
```

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

```
Defined.
```

```
End Logic_lemmas.
```

```
Module EQNOTATIONS.
```

```
Notation "'rew' H 'in' H'" := (eq_rect _ _ H' _ H)
```

```
  (at level 10, H' at level 10,
   format "'[' 'rew' H in '/' H' ]'").
```

```
Notation "'rew' [ P ] H 'in' H'" := (eq_rect _ P H' _ H)
```

```
  (at level 10, H' at level 10,
   format "'[' 'rew' [ P ] '/' H in '/' H' ]'").
```

**Notation** "'rew' <- H 'in' H'" := (eq\_rect\_r \_ H' H)  
 (at level 10, H' at level 10,  
 format "'[' 'rew' <- H in '/' H' ']'").  
**Notation** "'rew' <- [ P ] H 'in' H'" := (eq\_rect\_r P H' H)  
 (at level 10, H' at level 10,  
 format "'[' 'rew' <- [ P ] '/' H in '/' H' ']'").  
**Notation** "'rew' -> H 'in' H'" := (eq\_rect \_ \_ H' \_ H)  
 (at level 10, H' at level 10, only parsing).  
**Notation** "'rew' -> [ P ] H 'in' H'" := (eq\_rect \_ P H' \_ H)  
 (at level 10, H' at level 10, only parsing).

End EQNOTATIONS.

Import EqNotations.

Lemma rew\_opp\_r :  $\forall A (P:A \rightarrow \text{Type}) (x\ y:A) (H:x=y) (a:P\ y), \text{rew } H \text{ in } \text{rew } \leftarrow H \text{ in } a = a.$

Lemma rew\_opp\_l :  $\forall A (P:A \rightarrow \text{Type}) (x\ y:A) (H:x=y) (a:P\ x), \text{rew } \leftarrow H \text{ in } \text{rew } H \text{ in } a = a.$

Theorem f\_equal2 :  
 $\forall (A1\ A2\ B:\text{Type}) (f:A1 \rightarrow A2 \rightarrow B) (x1\ y1:A1)$   
 $(x2\ y2:A2), x1 = y1 \rightarrow x2 = y2 \rightarrow f\ x1\ x2 = f\ y1\ y2.$

Theorem f\_equal3 :  
 $\forall (A1\ A2\ A3\ B:\text{Type}) (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow B) (x1\ y1:A1)$   
 $(x2\ y2:A2) (x3\ y3:A3),$   
 $x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow f\ x1\ x2\ x3 = f\ y1\ y2\ y3.$

Theorem f\_equal4 :  
 $\forall (A1\ A2\ A3\ A4\ B:\text{Type}) (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B)$   
 $(x1\ y1:A1) (x2\ y2:A2) (x3\ y3:A3) (x4\ y4:A4),$   
 $x1 = y1 \rightarrow x2 = y2 \rightarrow x3 = y3 \rightarrow x4 = y4 \rightarrow f\ x1\ x2\ x3\ x4 = f\ y1\ y2\ y3\ y4.$

Theorem f\_equal5 :  
 $\forall (A1\ A2\ A3\ A4\ A5\ B:\text{Type}) (f:A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow A5 \rightarrow B)$   
 $(x1\ y1:A1) (x2\ y2:A2) (x3\ y3:A3) (x4\ y4:A4) (x5\ y5:A5),$   
 $x1 = y1 \rightarrow$   
 $x2 = y2 \rightarrow$   
 $x3 = y3 \rightarrow x4 = y4 \rightarrow x5 = y5 \rightarrow f\ x1\ x2\ x3\ x4\ x5 = f\ y1\ y2\ y3\ y4\ y5.$

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

The groupoid structure of equality

Theorem eq\_trans\_refl\_l :  $\forall A (x\ y:A) (e:x=y), \text{eq\_trans } \text{eq\_refl } e = e.$

Theorem eq\_trans\_refl\_r :  $\forall A (x\ y:A) (e:x=y), \text{eq\_trans } e\ \text{eq\_refl} = e.$

Theorem eq\_sym\_involutive :  $\forall A (x\ y:A) (e:x=y), \text{eq\_sym } (\text{eq\_sym } e) = e.$

Theorem eq\_trans\_sym\_inv\_l :  $\forall A (x\ y:A) (e:x=y), \text{eq\_trans } (\text{eq\_sym } e)\ e = \text{eq\_refl}.$

Theorem eq\_trans\_sym\_inv\_r :  $\forall A (x\ y:A) (e:x=y), \text{eq\_trans } e\ (\text{eq\_sym } e) = \text{eq\_refl}.$

Theorem eq\_trans\_assoc :  $\forall A (x\ y\ z\ t:A) (e:x=y) (e':y=z) (e'':z=t),$   
 $\text{eq\_trans } e\ (\text{eq\_trans } e'\ e'') = \text{eq\_trans } (\text{eq\_trans } e\ e')\ e''.$

Extra properties of equality

**Theorem** `eq_id_comm_l` :  $\forall A (f:A \rightarrow A) (Hf:\forall a, a = f a), \forall a, f\_equal\ f\ (Hf\ a) = Hf\ (f\ a)$ .

**Theorem** `eq_id_comm_r` :  $\forall A (f:A \rightarrow A) (Hf:\forall a, f\ a = a), \forall a, f\_equal\ f\ (Hf\ a) = Hf\ (f\ a)$ .

**Lemma** `eq_trans_map_distr` :  $\forall A\ B\ x\ y\ z\ (f:A \rightarrow B)\ (e:x=y)\ (e':y=z), f\_equal\ f\ (eq\_trans\ e\ e') = eq\_trans\ (f\_equal\ f\ e)\ (f\_equal\ f\ e')$ .

**Lemma** `eq_sym_map_distr` :  $\forall A\ B\ (x\ y:A)\ (f:A \rightarrow B)\ (e:x=y), eq\_sym\ (f\_equal\ f\ e) = f\_equal\ f\ (eq\_sym\ e)$ .

**Lemma** `eq_trans_sym_distr` :  $\forall A\ (x\ y\ z:A)\ (e:x=y)\ (e':y=z), eq\_sym\ (eq\_trans\ e\ e') = eq\_trans\ (eq\_sym\ e')\ (eq\_sym\ e)$ .

**Notation** `sym_eq` := `eq_sym` (*compat* "8.3").

**Notation** `trans_eq` := `eq_trans` (*compat* "8.3").

**Notation** `sym_not_eq` := `not_eq_sym` (*compat* "8.3").

**Notation** `refl_equal` := `eq_refl` (*compat* "8.3").

**Notation** `sym_equal` := `eq_sym` (*compat* "8.3").

**Notation** `trans_equal` := `eq_trans` (*compat* "8.3").

**Notation** `sym_not_equal` := `not_eq_sym` (*compat* "8.3").

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

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

## 57.4 Being inhabited

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

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

```
Hint Resolve inhabits: core.
```

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

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

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

```
Declare Left Step eq_stepl.
```

```
Declare Right Step eq_trans.
```

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

```
Declare Left Step iff_stepl.
```

```
Declare Right Step iff_trans.
```



## Chapter 58

# Library **Coq.Init.Notations**

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

Notations for propositional connectives

Reserved Notation "x -> y" (at level 99, right associativity, *y* at level 200).

Reserved Notation "x <-> y" (at level 95, no associativity).

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

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

Reserved Notation "~ x" (at level 75, right associativity).

Notations for equality and inequalities

Reserved Notation "x = y :> T"

(at level 70, *y* at *next level*, no associativity).

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

Reserved Notation "x = y = z"

(at level 70, no associativity, *y* at *next level*).

Reserved Notation "x <> y :> T"

(at level 70, *y* at *next level*, no associativity).

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

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

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

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

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

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

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

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

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

Arithmetical notations (also used for type constructors)

Reserved Notation "x + y" (at level 50, left associativity).

Reserved Notation "x - y" (at level 50, left associativity).

Reserved Notation "x \* y" (at level 40, left associativity).

Reserved Notation "x / y" (at level 40, left associativity).

Reserved Notation "- x" (at level 35, right associativity).

Reserved Notation  $/ x$  (at level 35, right associativity).

Reserved Notation  $x \wedge y$  (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  $\{x \mid P\}$  (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  $\{x : A \& P\}$  (at level 0,  $x$  at level 99).

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

Delimit Scope *type\_scope* with *type*.

Delimit Scope *function\_scope* with *function*.

Delimit Scope *core\_scope* with *core*.

Open Scope *core\_scope*.

Open Scope *function\_scope*.

Open Scope *type\_scope*.

ML Tactic Notations

## Chapter 59

# 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 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).
Hint Resolve f_equal_nat: core.
```

The predecessor function

```
Notation pred := Nat.pred (compat "8.4").
Definition f_equal_pred := f_equal pred.
Theorem pred_Sn :  $\forall n:\text{nat}, n = \text{pred } (S\ n)$ .
```

Injectivity of successor

**Definition** `eq_add_S`  $n\ m\ (H : S\ n = S\ m) : n = m := f\_equal\ pred\ H$ .

**Hint** `Immediate` `eq_add_S`: *core*.

**Theorem** `not_eq_S` :  $\forall\ n\ m : nat, n \neq m \rightarrow S\ n \neq S\ m$ .

**Hint** `Resolve` `not_eq_S`: *core*.

**Definition** `IsSucc`  $(n : nat) : Prop :=$

```
  match n with
  | 0 => False
  | S p => True
end.
```

Zero is not the successor of a number

**Theorem** `O_S` :  $\forall\ n : nat, 0 \neq S\ n$ .

**Hint** `Resolve` `O_S`: *core*.

**Theorem** `n_Sn` :  $\forall\ n : nat, n \neq S\ n$ .

**Hint** `Resolve` `n_Sn`: *core*.

Addition

**Notation** `plus` := `Nat.add` (*compat* "8.4").

**Infix** "+" := `Nat.add` : *nat\_scope*.

**Definition** `f_equal2_plus` := `f_equal2` `plus`.

**Definition** `f_equal2_nat` := `f_equal2` ( $A1 := nat$ ) ( $A2 := nat$ ).

**Hint** `Resolve` `f_equal2_nat`: *core*.

**Lemma** `plus_n_O` :  $\forall\ n : nat, n = n + 0$ .

**Hint** `Resolve` `plus_n_O`: *core*.

**Lemma** `plus_O_n` :  $\forall\ n : nat, 0 + n = n$ .

**Lemma** `plus_n_Sm` :  $\forall\ n\ m : nat, S\ (n + m) = n + S\ m$ .

**Hint** `Resolve` `plus_n_Sm`: *core*.

**Lemma** `plus_Sn_m` :  $\forall\ n\ m : nat, S\ n + m = S\ (n + m)$ .

Standard associated names

**Notation** `plus_0_r_reverse` := `plus_n_O` (*compat* "8.2").

**Notation** `plus_succ_r_reverse` := `plus_n_Sm` (*compat* "8.2").

Multiplication

**Notation** `mult` := `Nat.mul` (*compat* "8.4").

**Infix** "×" := `Nat.mul` : *nat\_scope*.

**Definition** `f_equal2_mult` := `f_equal2` `mult`.

**Hint** `Resolve` `f_equal2_mult`: *core*.

**Lemma** `mult_n_O` :  $\forall\ n : nat, 0 = n \times 0$ .

**Hint** `Resolve` `mult_n_O`: *core*.

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

**Hint** `Resolve` `mult_n_Sm`: *core*.

Standard associated names

**Notation** `mult_0_r_reverse` := `mult_n_O` (*compat* "8.2").

**Notation** `mult_succ_r_reverse` := `mult_n_Sm` (*compat* "8.2").

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

**Notation** `minus` := `Nat.sub` (*compat* "8.4").

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

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

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

**Hint Constructors** `le`: *core*.

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

**Hint Unfold** `lt`: *core*.

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

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

**Hint Unfold** `ge`: *core*.

**Infix** `"\geq"` := `ge` : *nat\_scope*.

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

**Hint Unfold** `gt`: *core*.

**Infix** `">"` := `gt` : *nat\_scope*.

**Notation** `"x <= y <= z"` := ( $x \leq y \wedge y \leq z$ ) : *nat\_scope*.

**Notation** `"x <= y < z"` := ( $x \leq y \wedge y < z$ ) : *nat\_scope*.

**Notation** `"x < y < z"` := ( $x < y \wedge y < z$ ) : *nat\_scope*.

**Notation** `"x < y <= z"` := ( $x < y \wedge y \leq z$ ) : *nat\_scope*.

**Theorem** `le_pred` :  $\forall n\ m, n \leq m \rightarrow \text{pred } n \leq \text{pred } m$ .

**Theorem** `le_S_n` :  $\forall n\ m, S\ n \leq S\ m \rightarrow n \leq m$ .

**Theorem** `le_0_n` :  $\forall n, 0 \leq n$ .

**Theorem** `le_n_S` :  $\forall n\ m, n \leq m \rightarrow S\ n \leq S\ m$ .

Case analysis

**Theorem** `nat_case` :

$\forall (n:\text{nat}) (P:\text{nat} \rightarrow \text{Prop}), P\ 0 \rightarrow (\forall m:\text{nat}, P\ (S\ m)) \rightarrow P\ n$ .

Principle of double induction

**Theorem** `nat_double_ind` :

$\forall R:\text{nat} \rightarrow \text{nat} \rightarrow \text{Prop},$   
 $(\forall n:\text{nat}, R\ 0\ n) \rightarrow$   
 $(\forall n:\text{nat}, R\ (S\ n)\ 0) \rightarrow$   
 $(\forall n\ m:\text{nat}, R\ n\ m \rightarrow R\ (S\ n)\ (S\ m)) \rightarrow \forall n\ m:\text{nat}, R\ n\ m$ .

Maximum and minimum : definitions and specifications

**Notation** `max` := `Nat.max` (*compat* "8.4").

**Notation** `min` := `Nat.min` (*compat* "8.4").

**Lemma** `max_l`  $n\ m : m \leq n \rightarrow \text{Nat.max } n\ m = n$ .

**Lemma** `max_r`  $n\ m : n \leq m \rightarrow \text{Nat.max } n\ m = m$ .

**Lemma** `min_l`  $n\ m : n \leq m \rightarrow \text{Nat.min } n\ m = n$ .

**Lemma** `min_r`  $n\ m : m \leq n \rightarrow \text{Nat.min } n\ m = m$ .

**Lemma** `nat_rect_succ_r`  $\{A\} (f: A \rightarrow A) (x:A) n :$   
`nat_rect` (`fun _  $\Rightarrow$  A`)  $x$  (`fun _  $\Rightarrow$  f`) (`S n`) = `nat_rect` (`fun _  $\Rightarrow$  A`) ( $f\ x$ ) (`fun _  $\Rightarrow$  f`)  $n$ .

**Theorem** `nat_rect_plus` :

$\forall (n\ m:\text{nat}) \{A\} (f:A \rightarrow A) (x:A),$   
`nat_rect` (`fun _  $\Rightarrow$  A`)  $x$  (`fun _  $\Rightarrow$  f`) ( $n + m$ ) =  
`nat_rect` (`fun _  $\Rightarrow$  A`) (`nat_rect` (`fun _  $\Rightarrow$  A`)  $x$  (`fun _  $\Rightarrow$  f`)  $m$ ) (`fun _  $\Rightarrow$  f`)  $n$ .

## Chapter 60

# Library **Coq.Init.Prelude**

```
Require Export Notations.  
Require Export Logic.  
Require Export Logic_Type.  
Require Export Datatypes.  
Require Export Specif.  
Require Coq.Init.Nat.  
Require Export Peano.  
Require Export Coq.Init.Wf.  
Require Export Coq.Init.Tactics.  
Require Export Coq.Init.Tauto.  
Add Search Blacklist "_subproof" "Private_".
```

## Chapter 61

# Library **Coq.Init.Specif**

Basic specifications : sets that may contain logical information

**Set Implicit Arguments.**

**Require Import** Notations.

**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  $Q$ .

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

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

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

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

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

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

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

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

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

**Notation** " $\{ x : A \ \& \ P \}$ " :=  $(\text{sigT} \ (A:=A) \ (\text{fun } x \Rightarrow P)) : \text{type\_scope}.$

**Notation** " $\{ x : A \ \& \ P \ \& \ Q \}$ " :=  $(\text{sigT2} \ (A:=A) \ (\text{fun } x \Rightarrow P) \ (\text{fun } x \Rightarrow Q)) :$   
   $\text{type\_scope}.$

**Add Printing Let**  $\text{sig}.$

**Add Printing Let**  $\text{sig2}.$



Add Printing Let *sigT*.

Add Printing Let *sigT2*.

Projections of *sig*

An element *y* of a subset  $\{x:A \mid (P\ x)\}$  is the pair of an *a* of type *A* and of a proof *h* that *a* satisfies *P*. Then (*proj1\_sig y*) is the witness *a* and (*proj2\_sig y*) is the proof of (*P a*)

Section Subset\_projections.

Variable *A* : Type.

Variable *P* : *A* → Prop.

Definition proj1\_sig (*e*:sig *P*) := match *e* with  
| exist \_ *a b* ⇒ *a*  
end.

Definition proj2\_sig (*e*:sig *P*) :=  
match *e* return *P* (proj1\_sig *e*) with  
| exist \_ *a b* ⇒ *b*  
end.

End Subset\_projections.

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

This allows *proj1\_sig* and *proj2\_sig* to be usable with *sig2*.

The **let** statements occur in the body of the *exist* so that *proj1\_sig* of a coerced *X* : *sig2 P Q* will unify with **let** (*a*, -, -) := *X* in *a*

Definition sig\_of\_sig2 (*A* : Type) (*P Q* : *A* → Prop) (*X* : sig2 *P Q*) : sig *P*  
:= exist *P*  
  (let (*a*, -, -) := *X* in *a*)  
  (let (*x*, *p*, -) as *s* return (*P* (let (*a*, -, -) := *s* in *a*)) := *X* in *p*).

Projections of *sig2*

An element *y* of a subset  $\{x:A \mid (P\ x) \ \& \ (Q\ x)\}$  is the triple of an *a* of type *A*, a of a proof *h* that *a* satisfies *P*, and a proof *h'* that *a* satisfies *Q*. Then (*proj1\_sig (sig\_of\_sig2 y)*) is the witness *a*, (*proj2\_sig (sig\_of\_sig2 y)*) is the proof of (*P a*), and (*proj3\_sig y*) is the proof of (*Q a*).

Section Subset\_projections2.

Variable *A* : Type.

Variables *P Q* : *A* → Prop.

Definition proj3\_sig (*e* : sig2 *P Q*) :=  
  let (*a*, *b*, *c*) return *Q* (proj1\_sig (sig\_of\_sig2 *e*)) := *e* in *c*.

End Subset\_projections2.

Projections of *sigT*

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

Section Projections.

Variable *A* : Type.

Variable *P* : *A* → Type.

```

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

```

```

Definition projT2 ( $x$ :sigT  $P$ ) :  $P$  (projT1  $x$ ) :=
  match  $x$  return  $P$  (projT1  $x$ ) with
  | existT _ _  $h$   $\Rightarrow$   $h$ 
  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 **let** (*a*,  $-$ ,  $-$ ) := *X* in *a*

```

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

```

## Projections of $sigT2$

An element  $x$  of a sigma-type  $\{y:A \ \& \ P \ y \ \& \ Q \ y\}$  is a dependent pair made of an  $a$  of type  $A$ , an  $h$  of type  $P \ a$ , and an  $h'$  of type  $Q \ a$ . Then,  $(projT1 \ (sigT\_of\_sigT2 \ x))$  is the first projection,  $(projT2 \ (sigT\_of\_sigT2 \ x))$  is the second projection, and  $(projT3 \ x)$  is the third projection, the types of which depends on the  $projT1$ .

## Section Projections2.

Variable  $A$  : Type.

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

```

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

```

## End Projections2.

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

```

Definition sig_of_sigT (A : Type) (P : A → Prop) (X : sigT P) : sig P
  := exist P (projT1 X) (projT2 X).

```

```

Definition sigT_of_sig (A : Type) (P : A → Prop) (X : sig P) : sigT P
:= existT P (proj1_sig X) (proj2_sig X).

```

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

```

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

```

```

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

```

*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

```

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.

```

Variables 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 :  $\forall (A:\text{Prop}) (C:\text{Set}), A \rightarrow \neg A \rightarrow C$ .

Hint Resolve left right inleft inright: core.

Hint Resolve exist exist2 existT existT2: core.

Notation sigS := sigT (compat "8.2").

Notation existS := existT (compat "8.2").

Notation sigS\_rect := sigT\_rect (compat "8.2").

Notation sigS\_rec := sigT\_rec (compat "8.2").

Notation sigS\_ind := sigT\_ind (compat "8.2").

Notation projS1 := projT1 (compat "8.2").

Notation projS2 := projT2 (compat "8.2").

Notation sigS2 := sigT2 (compat "8.2").

Notation existS2 := existT2 (compat "8.2").

Notation sigS2\_rect := sigT2\_rect (compat "8.2").

Notation sigS2\_rec := sigT2\_rec (compat "8.2").

Notation sigS2\_ind := sigT2\_ind (compat "8.2").

## Chapter 62

# Library **Coq.Init.Tactics**

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

### 62.1 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,

- $H: \sim A \mid\text{-} B$  gives  $\mid\text{-} A$
- $H: \sim A \mid\text{-} \sim B$  gives  $H: B \mid\text{-} A$
- $H: A \mid\text{-} B$  gives  $\mid\text{-} \sim A$
- $H: A \mid\text{-} \sim B$  gives  $H: B \mid\text{-} \sim A$
- $H:\text{False}$  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
|  $\vdash (\neg \_)$   $\Rightarrow$  negneg H
|  $\vdash (\_ \rightarrow \text{False})$   $\Rightarrow$  negneg H
|  $\vdash \_ \Rightarrow$  negpos H
end in
let pos H := match goal with
|  $\vdash (\neg \_)$   $\Rightarrow$  posneg H
|  $\vdash (\_ \rightarrow \text{False})$   $\Rightarrow$  posneg H
|  $\vdash \_ \Rightarrow$  pospos H
end in
match type of H with
|  $(\neg \_)$   $\Rightarrow$  neg H
|  $(\_ \rightarrow \text{False})$   $\Rightarrow$  neg H
|  $\_ \Rightarrow$  (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
  |  $x : t \vdash \_ \Rightarrow$  destruct x; destruct_all t
  |  $\_ \Rightarrow$  idtac
  end.

Tactic Notation "rewrite_all" constr(eq) := repeat rewrite eq in *.
Tactic Notation "rewrite_all" "<-" constr(eq) := repeat rewrite  $\leftarrow$  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" with
    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
    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*.

# Chapter 63

## Library **Coq.Init.Wf**

### 63.1 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** **Logic.**

**Require Import** **Datatypes.**

Well-founded induction principle on **Prop**

**Section** **Well\_founded.**

**Variable**  $A : \text{Type}.$

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

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** :  $(\forall y:A, R\ y\ x \rightarrow \text{Acc}\ y) \rightarrow \text{Acc}\ x.$

**Lemma** **Acc\_inv** :  $\forall x:A, \text{Acc}\ x \rightarrow \forall y:A, R\ y\ x \rightarrow \text{Acc}\ y.$

A relation is well-founded if every element is accessible

**Definition** **well\_founded** :=  $\forall a:A, \text{Acc}\ a.$

Well-founded induction on **Set** and **Prop**

**Hypothesis**  $Rwf : \text{well\_founded}.$

**Theorem** **well\_founded\_induction\_type** :

$\forall P:A \rightarrow \text{Type},$

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

**Theorem** **well\_founded\_induction** :

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

**Theorem** `well_founded_ind` :

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

Well-founded fixpoints

**Section** `FixPoint`.

**Variable**  $P : A \rightarrow \mathbf{Type}.$

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

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

**Scheme** `Acc_inv_dep` := **Induction** for `Acc` Sort `Prop`.

**Lemma** `Fix_F_eq` :

$\forall (x:A) (r:\mathbf{Acc}\ x),$   
 $F\ (\mathbf{fun}\ (y:A) (p:R\ y\ x) \Rightarrow \mathbf{Fix\_F}\ (x:=y) (\mathbf{Acc\_inv}\ r\ p)) = \mathbf{Fix\_F}\ (x:=x)\ r.$

**Definition** `Fix`  $(x:A) := \mathbf{Fix\_F}\ (Rwf\ x).$

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` :  $\forall (x:A) (r\ s:\mathbf{Acc}\ x), \mathbf{Fix\_F}\ r = \mathbf{Fix\_F}\ s.$

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

**End** `FixPoint`.

**End** `Well_founded`.

Well-founded fixpoints over pairs

**Section** `Well_founded_2`.

**Variables**  $A\ B : \mathbf{Type}.$

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

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

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

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

```

    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 64

## Library **Coq.Init.Nat**

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

```
Local Open Scope nat_scope.
```

### 64.1 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.
```

#### 64.1.1 Constants

```
Definition zero := 0.
```

```
Definition one := 1.
```

```
Definition two := 2.
```

#### 64.1.2 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  $\Rightarrow$  0
  | S  $p \Rightarrow m + p \times m$ 
  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  $k, S l \Rightarrow k - l$ 
  | -, -  $\Rightarrow n$ 
  end

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

```

### 64.1.3 Comparisons

```

Fixpoint eqb  $n$   $m$  : bool :=
  match  $n, m$  with
  | 0, 0  $\Rightarrow$  true
  | 0, S _  $\Rightarrow$  false
  | S _, 0  $\Rightarrow$  false
  | S  $n', S m' \Rightarrow$  eqb  $n' m'$ 
  end.

Fixpoint leb  $n$   $m$  : bool :=
  match  $n, m$  with
  | 0, _  $\Rightarrow$  true
  | _, 0  $\Rightarrow$  false
  | S  $n', S m' \Rightarrow$  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  $\Rightarrow$  Eq
  | 0, S _  $\Rightarrow$  Lt
  | S _, 0  $\Rightarrow$  Gt
  | S  $n', S m' \Rightarrow$  compare  $n' m'$ 
  end.

Infix "?=" := compare (at level 70) : nat_scope.

```

#### 64.1.4 Minimum, maximum

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

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

#### 64.1.5 Parity tests

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

Definition odd n := negb (even n).

#### 64.1.6 Power

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

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

#### 64.1.7 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.

```

### 64.1.8 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.

```

### 64.1.9 Square

```

Definition square n := n × n.

```

### 64.1.10 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
  | 0 => p
  | S k' => match r with
    | 0 => sqrt_iter k' (S p) (S (S q)) (S (S q))
    | S r' => sqrt_iter k' p q r'
  end
end

```



end.

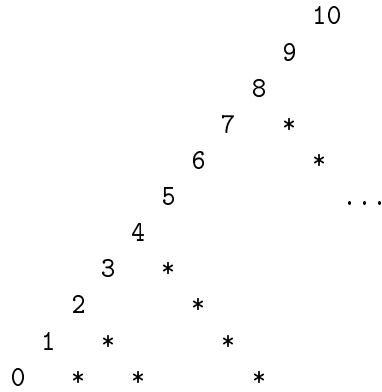
**Definition** `sqrt`  $n := \text{sqrt\_iter } n \ 0 \ 0 \ 0$ .

### 64.1.11 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  $r$ ) :



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
  | 0  $\Rightarrow p$ 
  | S  $k' \Rightarrow$  match  $r$  with
    | 0  $\Rightarrow$  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 ( $n:\text{nat}$ ) { $A$ } ( $f:A \rightarrow A$ ) ( $x:A$ ) :  $A :=$ 
  nat_rect (fun _  $\Rightarrow A$ )  $x$  (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 ⇒ 0
    | S 0 ⇒ 0
    | S (S n') ⇒ S (div2 n')
    end.

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

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

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

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

```

## Chapter 65

# Library **Coq.Init.Tauto**

```
Require Import Notations.
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

|  $\vdash ?X1 \Rightarrow$  is_conj flags X1; split
|  $\vdash (\text{Coq.Init.Logic.iff } \_ \_) \Rightarrow$  split
|  $\vdash (\text{Coq.Init.Logic.not } \_) \Rightarrow$  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.

```