

# The Coq Proof Assistant

## The standard library

August 24, 2012

Version 8.4<sup>1</sup>

TypiCal Project (formerly LogiCal)

---

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

V8.4, August 24, 2012

©INRIA 1999-2004 (CoQ versions 7.x)

©INRIA 2004-2012 (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.Init.Datatypes**

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

### 1.1 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.
```

### 1.2 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 >-> Prop.
Additional rewriting lemmas about eq_true

```

```

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

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

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

```

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

```

Inductive BoolSpec (P Q : Prop) : bool → Prop :=
| BoolSpecT : P → BoolSpec P Q true
| BoolSpecF : Q → BoolSpec P Q false.
Hint Constructors BoolSpec.

```

### 1.3 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*.

## 1.4 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 :=
  match o with
  | Some a ⇒ Some (f a)
  | None ⇒ None
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*.

Variables  $A$   $B$  : 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* :

$\forall (A B:Type) (p:A \times B), p = \text{pair } (\text{fst } p) (\text{snd } p).$

Lemma *injective\_projections* :

```

 $\forall (A\ B:\text{Type}) (p1\ p2:A \times B),$ 
   $\text{fst } p1 = \text{fst } p2 \rightarrow \text{snd } p1 = \text{snd } p2 \rightarrow p1 = p2.$ 
Definition prod_uncurry (A B C:Type) (f:prod A B  $\rightarrow$  C)
  (x:A) (y:B) : C := f (pair x y).
Definition prod_curry (A B C:Type) (f:A  $\rightarrow$  B  $\rightarrow$  C)
  (p:prod A B) : C := match p with
    | pair x y  $\Rightarrow$  f x y
  end.

```

Polymorphic lists and some operations

```

Inductive list (A : Type) : Type :=
| nil : list A
| cons : A  $\rightarrow$  list A  $\rightarrow$  list A.
Infix ":: $\rightarrow$ " := 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  $\rightarrow$  nat :=
  fix length l :=
  match l with
  | nil  $\Rightarrow$  0
  | _ :: l'  $\Rightarrow$  S (length l')
  end.

```

Concatenation of two lists

```

Definition app (A : Type) : list A  $\rightarrow$  list A  $\rightarrow$  list A :=
  fix app l m :=
  match l with
  | nil  $\Rightarrow$  m
  | a :: l1  $\Rightarrow$  a :: app l1 m
  end.
Infix "++" := app (right associativity, at level 60) : list_scope.

```

## 1.5 The comparison datatype

```

Inductive comparison : Set :=
| Eq : comparison
| Lt : comparison
| Gt : comparison.
Definition CompOpp (r:comparison) :=
  match r with
  | Eq  $\Rightarrow$  Eq
  | Lt  $\Rightarrow$  Gt
  | Gt  $\Rightarrow$  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`  $\rightarrow$  `Prop` :=

| `CompEq` : *Peq*  $\rightarrow$  `CompareSpec` *Peq Plt Pgt Eq*  
| `CompLt` : *Plt*  $\rightarrow$  `CompareSpec` *Peq Plt Pgt Lt*  
| `CompGt` : *Pgt*  $\rightarrow$  `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`  $\rightarrow$  `Type` :=

| `CompEqT` : *Peq*  $\rightarrow$  `CompareSpecT` *Peq Plt Pgt Eq*  
| `CompLtT` : *Plt*  $\rightarrow$  `CompareSpecT` *Peq Plt Pgt Lt*  
| `CompGtT` : *Pgt*  $\rightarrow$  `CompareSpecT` *Peq Plt Pgt Gt*.

**Hint Constructors** `CompareSpecT`.

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

`CompareSpec` *Peq Plt Pgt* *c*  $\rightarrow$  `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*  $\rightarrow$  *A*  $\rightarrow$  `Prop`) (*x y* : *A*) : `comparison`  $\rightarrow$  `Prop` :=

`CompareSpec` (*eq x y*) (*lt x y*) (*lt y x*).

**Definition** `CompSpecT` {*A*} (*eq lt* : *A*  $\rightarrow$  *A*  $\rightarrow$  `Prop`) (*x y* : *A*) : `comparison`  $\rightarrow$  `Type` :=

`CompareSpecT` (*eq x y*) (*lt x y*) (*lt y x*).

**Hint Unfold** `CompSpec` `CompSpecT`.

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

`CompSpec` *eq lt x y c*  $\rightarrow$  `CompSpecT` *eq lt x y c*.

## 1.6 Misc Other Datatypes

*identity A a* is the family of datatypes on *A* whose sole non-empty member is the singleton datatype

*identity A a a* whose sole inhabitant is denoted *refl\_identity A a*

**Inductive** `identity` (*A* : `Type`) (*a* : *A*) : *A*  $\rightarrow$  `Type` :=

`identity_refl` : `identity` *a a*.

**Hint Resolve** `identity_refl`: *core*.

Identity type

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

Definition  $\text{id} : \text{ID} := \text{fun } A \ x \Rightarrow x$ .

## Chapter 2

# 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 :
```

```
  ∀ (A:Type) (a:A) (P:A → Prop), P a → ∀ y:A, identity y a → P y.
```

```
Defined.
```

```
Definition identity_rec_r :
```

```
  ∀ (A:Type) (a:A) (P:A → Set), P a → ∀ y:A, identity y a → P y.
```

```
Defined.
```

```
Definition identity_rect_r :
```

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

```
Defined.
```

```
Hint Immediate identity_sym not_identity_sym: core v62.
```

```
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 3

# Library Coq.Init.Logic

Set Implicit Arguments.

Require Import Notations.

### 3.1 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 → 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 → B → A ∧ B`

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

`Section Conjunction.`

`Variables A B : Prop.`

`Theorem proj1 : A ∧ B → A.`

`Theorem proj2 : A ∧ B → B.`

`End Conjunction.`

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

`Inductive or (A B:Prop) : Prop :=  
| or_introl : A → A ∨ B  
| or_intror : B → A ∨ B`

where "A  $\vee$  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.

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

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

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

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

## 3.2 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.

Remark:  $\exists x, Q$  denotes  $ex\ (\text{fun } x \Rightarrow Q)$  so that  $\exists x, P\ x$  is in fact equivalent to  $ex\ (\text{fun } x \Rightarrow P\ x)$  which may be not convertible to  $ex\ P$  if  $P$  is not itself an abstraction

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

**Inductive** `ex2` ( $A:\text{Type}$ ) ( $P\ Q:A \rightarrow \text{Prop}$ ) : **Prop** :=  
`ex_intro2` :  $\forall x:A, P\ x \rightarrow Q\ x \rightarrow ex2\ (A:=A)\ P\ Q$ .

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

**Notation** `"'exists' x .. y , p"` := (`ex` (`fun`  $x \Rightarrow \dots$  (`ex` (`fun`  $y \Rightarrow p$ ))  $\dots$ ))  
(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**  $A : \text{Type}$ .

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

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

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

**End** `universal_quantification`.

## 3.3 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 | conj or_introl or_intror 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.
```

```
Opaque eq_sym.
```

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

```
Opaque eq_trans.
```

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

```
Opaque f_equal.
```

```
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).
```

```
Notation "'rew' <- H 'in' H'" := (eq_rect_r _ H' H)
```



```

    (at level 10, H' at level 10).
  Notation "'rew' -> H 'in' H'" := (eq_rect _ _ H' _ H)
    (at level 10, H' at level 10, only parsing).
End EQNOTATIONS.

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

Theorem f_equal3 :
  ∀ (A1 A2 A3 B:Type) (f:A1 → A2 → A3 → B) (x1 y1:A1)
    (x2 y2:A2) (x3 y3:A3),
    x1 = y1 → x2 = y2 → x3 = y3 → f x1 x2 x3 = f y1 y2 y3.

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

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

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

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

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

Unique existence

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

```

```

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

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

```

### 3.4 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  $\rightarrow$  inhabited A.

```

```

Hint Resolve inhabits: core.

```

```

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

```

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

```

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

Declare Left Step eq_stepl.
Declare Right Step eq_trans.

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

Declare Left Step iff_stepl.
Declare Right Step iff_trans.

```

## Chapter 4

# Library **Coq.Init.Notations**

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

Notations for propositional connectives

Reserved Notation " $x \leftrightarrow y$ " (at level 95, no associativity).  
Reserved Notation " $x \wedge y$ " (at level 80, right associativity).  
Reserved Notation " $x \vee y$ " (at level 85, right associativity).  
Reserved Notation " $\sim x$ " (at level 75, right associativity).

Notations for equality and inequalities

Reserved Notation " $x = y \Rightarrow T$ "  
(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 " $x \leftrightarrow y \Rightarrow T$ "  
(at level 70,  $y$  at *next level*, no associativity).  
Reserved Notation " $x \leftrightarrow y$ " (at level 70, no associativity).  
Reserved Notation " $x \leq y$ " (at level 70, no associativity).  
Reserved Notation " $x < y$ " (at level 70, no associativity).  
Reserved Notation " $x \geq y$ " (at level 70, no associativity).  
Reserved Notation " $x > y$ " (at level 70, no associativity).  
Reserved Notation " $x \leq y \leq z$ " (at level 70,  $y$  at *next level*).  
Reserved Notation " $x \leq y < z$ " (at level 70,  $y$  at *next level*).  
Reserved Notation " $x < y < z$ " (at level 70,  $y$  at *next level*).  
Reserved Notation " $x < y \leq z$ " (at level 70,  $y$  at *next level*).

Arithmetical notations (also used for type constructors)

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

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

Notations for booleans

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

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

Notations for pairs

Reserved Notation " $(x, y, \dots, z)$ " (at level 0).

Notation " $\{x\}$ " is reserved and has a special status as component of other notations such as " $\{A\} + \{B\}$ " and " $A + \{B\}$ " (which are at the same level than " $x + y$ "); " $\{x\}$ " 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 *core\_scope* with *core*.

Open Scope *core\_scope*.

Open Scope *type\_scope*.

## Chapter 5

# 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.
```

```
Open Scope nat_scope.
```

```
Definition eq_S := f_equal S.
```

```
Hint Resolve (f_equal S): v62.
```

```
Hint Resolve (f_equal (A:=nat)): core.
```

The predecessor function

```
Definition pred (n:nat) : nat := match n with
| O => n
| S u => u
end.
```

```
Hint Resolve (f_equal pred): v62.
```

Theorem `pred_Sn` :  $\forall n:\text{nat}, n = \text{pred } (\text{S } n)$ .

Injectivity of successor

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

Hint `Immediate eq_add_S`: *core*.

Theorem `not_eq_S` :  $\forall n\ m:\text{nat}, n \neq m \rightarrow \text{S } n \neq \text{S } m$ .

Hint `Resolve not_eq_S`: *core*.

Definition `IsSucc`  $(n:\text{nat}) : \text{Prop} :=$

```
match n with
| O  $\Rightarrow$  False
| S p  $\Rightarrow$  True
end.
```

Zero is not the successor of a number

Theorem `O_S` :  $\forall n:\text{nat}, 0 \neq \text{S } n$ .

Hint `Resolve O_S`: *core*.

Theorem `n_Sn` :  $\forall n:\text{nat}, n \neq \text{S } n$ .

Hint `Resolve n_Sn`: *core*.

Addition

Fixpoint `plus`  $(n\ m:\text{nat}) : \text{nat} :=$

```
match n with
| O  $\Rightarrow$  m
| S p  $\Rightarrow$  S (p + m)
end
```

where `"n + m"` :=  $(\text{plus } n\ m) : \text{nat\_scope}$ .

Hint `Resolve (f_equal2 plus)`: *v62*.

Hint `Resolve (f_equal2 (A1:=nat) (A2:=nat))`: *core*.

Lemma `plus_n_O` :  $\forall n:\text{nat}, n = n + 0$ .

Hint `Resolve plus_n_O`: *core*.

Lemma `plus_O_n` :  $\forall n:\text{nat}, 0 + n = n$ .

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

Hint `Resolve plus_n_Sm`: *core*.

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

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

Fixpoint `mult`  $(n\ m:\text{nat}) : \text{nat} :=$

```
match n with
| O  $\Rightarrow$  0
| S p  $\Rightarrow$  m + p  $\times$  m
```

```

end

where "n * m" := (mult n m) : nat_scope.
Hint Resolve (f_equal2 mult): core.
Lemma mult_n_O :  $\forall n:\text{nat}, 0 = n \times 0$ .
Hint Resolve mult_n_O: core.
Lemma mult_n_Sm :  $\forall n m:\text{nat}, n \times m + n = n \times S m$ .
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  $n \geq m$ 
Fixpoint minus (n m: nat) : nat :=
  match n, m with
  | O, _  $\Rightarrow$  n
  | S k, O  $\Rightarrow$  n
  | S k, S l  $\Rightarrow$  k - l
  end

where "n - m" := (minus n m) : nat_scope.

Definition of the usual orders, the basic properties of le and lt can be found in files Le and Lt
Inductive le (n: nat) : nat  $\rightarrow$  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: nat) := S n  $\leq$  m.
Hint Unfold lt: core.
Infix "<" := lt : nat_scope.
Definition ge (n m: nat) := m  $\leq$  n.
Hint Unfold ge: core.
Infix " $\geq$ " := ge : nat_scope.
Definition gt (n m: 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 :  $\forall n\ m, S\ n \leq S\ m \rightarrow n \leq m$ .

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

**Fixpoint** max  $n\ m : \text{nat} :=$

match  $n, m$  with  
 |  $O, - \Rightarrow m$   
 |  $S\ n', O \Rightarrow n$   
 |  $S\ n', S\ m' \Rightarrow S\ (\text{max } n'\ m')$   
 end.

**Fixpoint** min  $n\ m : \text{nat} :=$

match  $n, m$  with  
 |  $O, - \Rightarrow 0$   
 |  $S\ n', O \Rightarrow 0$   
 |  $S\ n', S\ m' \Rightarrow S\ (\text{min } n'\ m')$   
 end.

**Theorem** max\_l :  $\forall n\ m : \text{nat}, m \leq n \rightarrow \text{max } n\ m = n$ .

**Theorem** max\_r :  $\forall n\ m : \text{nat}, n \leq m \rightarrow \text{max } n\ m = m$ .

**Theorem** min\_l :  $\forall n\ m : \text{nat}, n \leq m \rightarrow \text{min } n\ m = n$ .

**Theorem** min\_r :  $\forall n\ m : \text{nat}, m \leq n \rightarrow \text{min } n\ m = m$ .

$n$ th iteration of the function  $f$

**Fixpoint** nat\_iter  $(n:\text{nat}) \{A\} (f:A \rightarrow A) (x:A) : A :=$

match  $n$  with  
 |  $O \Rightarrow x$   
 |  $S\ n' \Rightarrow f\ (\text{nat\_iter } n'\ f\ x)$   
 end.

**Lemma** nat\_iter\_succ\_r  $n \{A\} (f:A \rightarrow A) (x:A) :$

$\text{nat\_iter } (S\ n)\ f\ x = \text{nat\_iter } n\ f\ (f\ x)$ .

**Theorem** nat\_iter\_plus :

$\forall (n\ m:\text{nat}) \{A\} (f:A \rightarrow A) (x:A),$   
 $\text{nat\_iter } (n + m)\ f\ x = \text{nat\_iter } n\ f\ (\text{nat\_iter } m\ f\ x)$ .

Preservation of invariants : if  $f : A \rightarrow A$  preserves the invariant  $Inv$ , then the iterates of  $f$  also preserve it.



**Theorem** `nat_iter_invariant` :

$$\forall (n:\text{nat}) \{A\} (f:A \rightarrow A) (P : A \rightarrow \text{Prop}),$$

$$(\forall x, P\ x \rightarrow P\ (f\ x)) \rightarrow$$

$$\forall x, P\ x \rightarrow P\ (\text{nat\_iter}\ n\ f\ x).$$

## Chapter 6

# Library `Coq.Init.Prelude`

```
Require Export Notations.  
Require Export Logic.  
Require Export Datatypes.  
Require Export Specif.  
Require Export Peano.  
Require Export Coq.Init.Wf.  
Require Export Coq.Init.Tactics.  
Add Search Blacklist "_admitted" "_subproof" "Private_".
```

## Chapter 7

# 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** " $\{ x : A \mid P \}$ " :=  $(\text{sig} \ (\text{fun } x:A \Rightarrow P)) : \text{type\_scope}.$

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

**Notation** " $\{ x : A \ \& \ P \}$ " :=  $(\text{sigT} \ (\text{fun } x:A \Rightarrow P)) : \text{type\_scope}.$

**Notation** " $\{ x : A \ \& \ P \ \& \ Q \}$ " :=  $(\text{sigT2} \ (\text{fun } x:A \Rightarrow P) \ (\text{fun } x:A \Rightarrow Q)) :$   
   $\text{type\_scope}.$

**Add Printing Let**  $\text{sig}.$

Add Printing Let  $\text{sig}2$ .  
 Add Printing Let  $\text{sig}T$ .  
 Add Printing Let  $\text{sig}T2$ .

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

```

Definition projl_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.

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

```

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.

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

$$\text{Lemma sig\_of\_sigT} : \forall (A:\text{Type}) (P:A \rightarrow \text{Prop}), \text{sigT } P \rightarrow \text{sig } P.$$
$$\text{Lemma sigT\_of\_sig : } \forall (A:\text{Type}) (P:A \rightarrow \text{Prop}), \text{sig } P \rightarrow \text{sigT } P.$$

```
Coercion sigT_of_sig : sig >-> sig T.
```

Coercion `sig_of_sigT` :  $\text{sig } T \rightarrow \text{sig}$ .

*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.

```

Definition Exc := option.

```

```

Definition value := Some.

```

```

Definition error := @None.

```

```

Definition except := False_rec.

```

```

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

```

Hint Resolve left right inleft inright: *core v62*.  
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 8

# Library `Coq.Init.Tactics`

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

### 8.1 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 swap H :=
  idtac "swap is OBSOLETE: use contradict instead.";
  intro; apply H; clear H.

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; clear H; intro 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" "<-" 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" "<-" 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 rec do_atom :=
  solve [reflexivity | symmetry; trivial] ||
  contradiction ||
  (split; do_atom)
with do_ccl := trivial with eq_true; repeat do_intro; do_atom in
(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 9

# Library `Coq.Init.Wf`

### 9.1 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 : Type.
```

```
Variable R : A → A → Prop.
```

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

```
Inductive Acc (x: A) : Prop :=
```

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

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

```
Global Implicit Arguments Acc_inv [x y] [x].
```

A relation is well-founded if every element is accessible

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

Well-founded induction on `Set` and `Prop`

```
Hypothesis Rwf : well_founded.
```

```
Theorem well_founded_induction_type :
```

```
  ∀ P:A → Type,
```

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

**Theorem** `well_founded_induction` :

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

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

(**fun** ( $y:A$ ) ( $y':B$ ) ( $h:R$  ( $y, y'$ ) ( $x, x'$ ))  $\Rightarrow$   
 $\text{Fix\_F\_2 } (x:=y) (x':=y') (\text{Acc\_inv } a (y, y') h)).$

**End** FixPoint\_2.

**Hypothesis**  $Rwf$  :  $\text{well\_founded } R$ .

**Theorem**  $\text{well\_founded\_induction\_type\_2}$  :

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

**End** Well\_founded\_2.

**Notation**  $\text{Acc\_iter} := \text{Fix\_F } (\text{only parsing})$ . **Notation**  $\text{Acc\_iter\_2} := \text{Fix\_F\_2 } (\text{only parsing})$ .

## Chapter 10

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

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

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

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

The dependent elimination above implies the axiom of choice: **Lemma**  $\text{AC} : \forall r:\text{retract\_cond}, \text{retract} \rightarrow \forall a:A, j2 r (i2 r a) = a$ .

**End** **Retracts**.

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

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

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

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

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

Encoding of Russel's paradox

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

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

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

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

**End** **Berardis\\_paradox**.



# Chapter 11

## 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.**

## 11.1 Definitions

Choice, reification and description schemes

**Section** ChoiceSchemes.

**Variables** A B :Type.

**Variable** P:A→Prop.

**Variable** R:A→B→Prop.

### 11.1.1.1 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 :=

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

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 :=

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

### 11.1.2 Weakly classical choice and description

GAC\_rel

**Definition** GuardedRelationalChoice\_on :=

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

GAC\_fun

**Definition** GuardedFunctionalChoice\_on :=

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

GFR\_fun

**Definition** GuardedFunctionalRelReification\_on :=

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

OAC\_rel

**Definition** OmniscientRelationalChoice\_on :=

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

OAC\_fun

**Definition** OmniscientFunctionalChoice\_on :=

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

D\_epsilon

**Definition** EpsilonStatement\_on :=

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

D\_iota

**Definition** IotaStatement\_on :=

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

**End** ChoiceSchemes.

Generalized schemes

**Notation** RelationalChoice :=

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

## 11.2 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` :

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

**Lemma** `func_choice_imp_description` :

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

**Corollary** `FunChoice_Equiv_RelChoice_and_ParamDefinDescr` :

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

## 11.3 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 independance of general premises or subset types (themselves derivable from subtypes thanks to proof- irrelevance)

### 11.3.1 AC\_rel + PI -> GAC\_rel and AC\_rel + IGP -> GAC\_rel and GAC\_rel = OAC\_rel

**Lemma** `rel_choice_and_proof_irrel_imp_guarded_rel_choice` :

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

**Lemma** `rel_choice_indep_of_general_premises_imp_guarded_rel_choice` :

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

**Lemma** `guarded_rel_choice_imp_rel_choice` :

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

**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` :

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

### 11.3.2 AC\_fun + IGP = GAC\_fun = OAC\_fun = AC\_fun + Drinker

AC\_fun + IGP = GAC\_fun

**Lemma** guarded\_fun\_choice\_imp\_indep\_of\_general\_premises :

GuardedFunctionalChoice  $\rightarrow$  IndependenceOfGeneralPremises.

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

GuardedFunctionalChoice  $\rightarrow$  FunctionalChoiceOnInhabitedSet.

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

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

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

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

AC\_fun + Drinker = OAC\_fun

This was already observed by Bell [*Bell*]

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

OmniscientFunctionalChoice  $\rightarrow$  SmallDrinker'sParadox.

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

OmniscientFunctionalChoice  $\rightarrow$  FunctionalChoiceOnInhabitedSet.

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

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

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

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

OAC\_fun = GAC\_fun

This is derivable from the intuitionistic equivalence between IGP and Drinker but we give a direct proof

**Theorem** guarded\_iff\_omniscient\_fun\_choice :

GuardedFunctionalChoice  $\leftrightarrow$  OmniscientFunctionalChoice.

### 11.3.3 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` :  
 $(\text{SmallDrinker'sParadox} \times \text{ConstructiveIndefiniteDescription} \rightarrow$   
 $\text{EpsilonStatement}) \times$   
 $(\text{EpsilonStatement} \rightarrow$   
 $\text{SmallDrinker'sParadox} \times \text{ConstructiveIndefiniteDescription})$ .

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

**Lemma** `classical_denumerable_description_imp_fun_choice` :  
 $\forall A:\text{Type},$   
 $\text{FunctionalRelReification\_on } A\ \text{nat} \rightarrow$   
 $\forall R:A \rightarrow \text{nat} \rightarrow \text{Prop},$   
 $(\forall x\ y, \text{decidable } (R\ x\ y)) \rightarrow \text{FunctionalChoice\_on\_rel } R$ .

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

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

**Definition** `DependentFunctionalChoice_on` ( $A:\text{Type}$ ) ( $B:A \rightarrow \text{Type}$ ) :=  
 $\forall R:\forall x:A, B\ x \rightarrow \text{Prop},$   
 $(\forall x:A, \exists y : B\ x, R\ x\ y) \rightarrow$   
 $(\exists f : (\forall x:A, B\ x), \forall x:A, R\ x\ (f\ x))$ .

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

The easy part

**Theorem** `dep_non_dep_functional_choice` :  
`DependentFunctionalChoice`  $\rightarrow$  `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.

### 11.5.2 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.

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

### 11.6.1 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.

### 11.6.2 Non contradiction of definite description

**Lemma** `relative_non_contradiction_of_definite_descr` :

$\forall C:\text{Prop}, (\text{ConstructiveDefiniteDescription} \rightarrow C)$   
 $\rightarrow (\text{FunctionalRelReification} \rightarrow C).$

**Lemma** `constructive_definite_descr_fun_reification` :

$\text{ConstructiveDefiniteDescription} \rightarrow \text{FunctionalRelReification}.$

Remark, the following corollaries morally hold:

Definition `In_propositional_context`  $(A:\text{Type}) := \text{forall } C:\text{Prop}, (A \rightarrow C) \rightarrow C.$

Corollary `constructive_definite_descr_in_prop_context_iff_fun_reification` : `In_propositional_context`  
 $\text{ConstructiveIndefiniteDescription} \leftrightarrow \text{FunctionalChoice}.$

Corollary `constructive_definite_descr_in_prop_context_iff_fun_reification` : `In_propositional_context`  
 $\text{ConstructiveDefiniteDescription} \leftrightarrow \text{FunctionalRelReification}.$

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.

## 11.7 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` :

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

**Corollary** `fun_reification_descr_computational_excluded_middle_in_prop_context` :

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

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

**Require Import** Arith.

**Theorem** `functional_choice_imp_functional_dependent_choice` :

$\text{FunctionalChoice} \rightarrow \text{FunctionalDependentChoice}.$

**Theorem** `functional_dependent_choice_imp_functional_countable_choice` :  
    `FunctionalDependentChoice`  $\rightarrow$  `FunctionalCountableChoice`.

## Chapter 12

# 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 13

# 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.

**Local Notation** inhabited  $A := A$  (*only parsing*).

The idea for the following proof comes from *ChicliPottierSimpson02*

**Theorem** excluded\_middle\_informative :  $\forall P:\text{Prop}, \{P\} + \{\neg P\}$ .

**Theorem** classical\_definite\_description :

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

Church's iota operator

**Definition** iota  $(A : \text{Type}) (i:\text{inhabited } A) (P : A \rightarrow \text{Prop}) : A$   
 $:= \text{proj1\_sig } (\text{classical\_definite\_description } P i)$ .

**Definition** iota\_spec  $(A : \text{Type}) (i:\text{inhabited } A) (P : A \rightarrow \text{Prop}) :$   
 $(\exists! x:A, P x) \rightarrow P (\text{iota } i P)$   
 $:= \text{proj2\_sig } (\text{classical\_definite\_description } P i)$ .

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 14

# 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 Castéran)

**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 15

## 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

### 15.1 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`.

## 15.2 Classical logic and proof-irrelevance

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

We successively show that:

*prop\_extensionality* implies equality of  $A$  and  $A \rightarrow A$  for inhabited  $A$ , which implies the existence of a (trivial) retract from  $A \rightarrow A$  to  $A$  (just take the identity), which implies the existence of a fixpoint operator in  $A$  (e.g. take the Y combinator of lambda-calculus)

**Local Notation** `inhabited A` :=  $A$  (*only parsing*).

**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: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: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*.

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

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

Assume that we have booleans with the property that there is at most 2 booleans (which is equivalent to dependent case analysis). Consider the fixpoint of the negation function: it is either true or false by dependent case analysis, but also the opposite by fixpoint. Hence proof-irrelevance.

We then map equality of boolean proofs to proof irrelevance in all propositions.

**Section** `Proof_irrelevance_gen`.

**Variable** `bool` : `Prop`.

**Variable** `true` : `bool`.

**Variable** `false` : `bool`.

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

**Hypothesis**

*bool\_elim\_redl* :  $\forall (C:\text{Prop}) (c1 \ c2:C), c1 = \text{bool\_elim } C \ c1 \ c2 \ \text{true}$ .

**Hypothesis**

*bool\_elim\_redr* :  $\forall (C:\text{Prop}) (c1 \ c2:C), c2 = \text{bool\_elim } C \ c1 \ c2 \ \text{false}$ .

```

Let bool_dep_induction :=
  ∀ P:bool → Prop, P true → P false → ∀ b:bool, P b.

Lemma aux : prop_extensionality → bool_dep_induction → true = false.

Lemma ext_prop_dep_proof_irrel_gen :
  prop_extensionality → bool_dep_induction → proof_irrelevance.

End Proof_irrelevance_gen.

```

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

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

```

Definition BoolP := ∀ C:Prop, C → C → C.
Definition TrueP : BoolP := fun C c1 c2 ⇒ c1.
Definition FalseP : BoolP := fun C c1 c2 ⇒ c2.
Definition BoolP_elim C c1 c2 (b:BoolP) := b C c1 c2.
Definition BoolP_elim_redl (C:Prop) (c1 c2:C) :
  c1 = BoolP_elim C c1 c2 TrueP := eq_refl c1.
Definition BoolP_elim_redr (C:Prop) (c1 c2:C) :
  c2 = BoolP_elim C c1 c2 FalseP := eq_refl c2.
Definition BoolP_dep_induction :=
  ∀ P:BoolP → Prop, P TrueP → P FalseP → ∀ b:BoolP, P b.

Lemma ext_prop_dep_proof_irrel_cc :
  prop_extensionality → BoolP_dep_induction → proof_irrelevance.

End Proof_irrelevance_Prop_Ext_CC.

```

Remark: *prop\_extensionality* can be replaced in lemma *ext\_prop\_dep\_proof\_irrel\_gen* by the weakest property *provable\_prop\_extensionality*.

### 15.2.3 CIC |- prop. ext. -> proof-irrelevance

In the Calculus of Inductive Constructions, inductively defined booleans enjoy dependent case analysis, hence directly proof-irrelevance from propositional extensionality.

#### Section *Proof\_irrelevance\_CIC*.

```

Inductive boolP : Prop :=
  | trueP : boolP
  | falseP : boolP.
Definition boolP_elim_redl (C:Prop) (c1 c2:C) :
  c1 = boolP_ind C c1 c2 trueP := eq_refl c1.
Definition boolP_elim_redr (C:Prop) (c1 c2:C) :
  c2 = boolP_ind C c1 c2 falseP := eq_refl c2.
Scheme boolP_indd := Induction for boolP Sort Prop.
Lemma ext_prop_dep_proof_irrel_cic : prop_extensionality → proof_irrelevance.

End Proof_irrelevance_CIC.

```

Can we state proof irrelevance from propositional degeneracy (i.e. propositional extensionality + excluded middle) without dependent case analysis ?

Berardi [Berardi90] built a model of CC interpreting inhabited types by the set of all untyped lambda-terms. This model satisfies propositional degeneracy without satisfying proof-irrelevance (nor dependent case analysis). This implies that the previous results cannot be refined.

[Berardi90] Stefano Berardi, “Type dependence and constructive mathematics”, Ph. D. thesis, Dipartimento Matematica, Università di Torino, 1990.

#### 15.2.4 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
Definition p2b A := or_elim A (¬ A) B (fun _ => b1) (fun _ => b2) (em A).
Definition b2p b := b1 = b.
Lemma p2p1 : ∀ A:Prop, A → b2p (p2b A).
Lemma p2p2 : b1 ≠ b2 → ∀ A:Prop, b2p (p2b A) → A.

```

Using excluded-middle a second time, we get proof-irrelevance

**Theorem** `proof_irrelevance_cc` :  $b1 = b2$ .

**End** `Proof_irrelevance_EM_CC`.

Remark: 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 proof-irrelevance.

### 15.2.5 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` ( $A\ B\ C:\text{Prop}$ ) ( $f:A \rightarrow C$ ) ( $g:B \rightarrow C$ )

( $a:A$ ) :  $f\ a = \text{or\_ind}\ f\ g\ (\text{or\_introl}\ B\ a) := \text{eq\_refl}\ (f\ a)$ .

**Definition** `or_elim_redr` ( $A\ B\ C:\text{Prop}$ ) ( $f:A \rightarrow C$ ) ( $g:B \rightarrow C$ )

( $b:B$ ) :  $g\ b = \text{or\_ind}\ f\ g\ (\text{or\_intror}\ A\ b) := \text{eq\_refl}\ (g\ b)$ .

**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`.

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` :  $\forall A:\text{Prop}, \{A\} + \{\sim A\}$  in the Set-impredicative CCI.

## 15.3 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

### 15.3.1 Weak excluded-middle

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

**Definition** `weak_excluded_middle` :=

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

The interest in the equivalent variant `weak_generalized_excluded_middle` is that it holds even in logic without a primitive *False* connective (like Gödel-Dummett axiom)

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

### 15.3.2 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`.

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

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

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

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

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

**Definition** `IndependenceOfGeneralPremises` :=

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

**Lemma**

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

**Lemma** `independence_general_premises_Godel_Dummett` :

`IndependenceOfGeneralPremises`  $\rightarrow$  `GodelDummett`.

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

**Definition** `DrinkerParadox` :=

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

**Lemma** `independence_general_premises_drinker` :

`IndependenceOfGeneralPremises`  $\leftrightarrow$  `DrinkerParadox`.

Independence of general premises is weaker than (generalized) excluded middle

Remark: generalized excluded middle is preferred here to avoid relying on the “ex falso quodlibet” property (i.e.  $\text{False} \rightarrow \forall A, A$ )

**Definition** `generalized_excluded_middle` :=

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

**Lemma** `excluded_middle_independence_general_premises` :

`generalized_excluded_middle`  $\rightarrow$  `DrinkerParadox`.

## Chapter 16

# Library **Coq.Logic.Classical\_Pred\_Set**

This file is obsolete, use Classical\_Pred\_Type.v via Classical.v instead  
Classical Predicate Logic on Set

**Require Import** Classical\_Pred\_Type.

**Section** Generic.

**Variable**  $U$  : Set.

de Morgan laws for quantifiers

**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\_all\_not\_ex :

$\forall P:U \rightarrow \text{Prop}, \neg (\forall n:U, \neg P\ n) \rightarrow \exists n : U, P\ n.$

**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 17

# 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 :`

`∀ P:U → Prop, ¬ (∀ n:U, ¬ P n) → ∃ n : U, P n.`

`Lemma not_all_ex_not :`

`∀ P:U → Prop, ¬ (∀ n:U, P n) → ∃ n : U, ¬ P n.`

`Lemma not_ex_all_not :`

`∀ P:U → Prop, ¬ (∃ n : U, P n) → ∀ n:U, ¬ P n.`

`Lemma not_ex_not_all :`

`∀ P:U → Prop, ¬ (∃ n : U, ¬ P n) → ∀ n:U, P n.`

`Lemma ex_not_not_all :`

`∀ P:U → Prop, (∃ n : U, ¬ P n) → ¬ (∀ n:U, P n).`

`Lemma all_not_not_ex :`

`∀ P:U → Prop, (∀ n:U, ¬ P n) → ¬ (∃ n : U, P n).`

`End Generic.`

## Chapter 18

# 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
| _ : _  $\vdash$  _  $\wedge$  ?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 19

# Library **Coq.Logic.Classical\_Type**

This file is obsolete, use `Classical.v` instead  
Classical Logic for Type

**Require Export** `Classical_Prop`.

**Require Export** `Classical_Pred_Type`.

## Chapter 20

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

Unique choice reifies functional relations into functions

**Theorem** *unique\_choice* :

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

The following proof comes from [*ChicliPottierSimpson02*]

**Require Import** Setoid.

**Theorem** *classic\_set\_in\_prop\_context* :

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

**Corollary** *not\_not\_classic\_set* :

$((\forall P:\mathbf{Prop}, \{P\} + \{\neg P\}) \rightarrow \mathbf{False}) \rightarrow \mathbf{False}.$

**Notation** `classic_set` := `not_not_classic_set` (*only parsing*).

## Chapter 21

# Library **Coq.Logic.Classical**

Classical Logic

**Require Export** Classical\_Prop.

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

## Chapter 22

# 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* : nat → Prop :=

| **stop** :  $\forall n, P\ n \rightarrow \text{before\_witness } n$   
| **next** :  $\forall n, \text{before\_witness } (S\ n) \rightarrow \text{before\_witness } n$ .

**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 in before_witness n return ¬ P n → before_witness (S n) with
    | stop n p => fun not_p => match (not_p p) with end
    | next n 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 :=  $x = S y \wedge \neg P y$ .

Local Notation acc *x* := (Acc *R* *x*).

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

Definition constructive\_ground\_epsilon\_spec\_nat ( $E : (\exists n, P\ n)$ ) :  $P$  (constructive\_ground\_epsilon\_nat  $E$ )  
:= proj2\_sig (constructive\_indefinite\_ground\_description\_nat  $P\ P\_decidable\ E$ ).

End ConstructiveGroundEpsilon\_nat.

Section ConstructiveGroundEpsilon.

For the current purpose, we say that a set  $A$  is countable if there are functions  $f : A \rightarrow \text{nat}$  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  $gof\_eq\_id : \forall x : A, g\ (f\ x) = x$ .

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

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

Definition constructive\_ground\_epsilon ( $E : \exists x : A, P\ x$ ) :  $A$   
:= proj1\_sig (constructive\_indefinite\_ground\_description  $E$ ).

Definition constructive\_ground\_epsilon\_spec ( $E : (\exists x, P\ x)$ ) :  $P$  (constructive\_ground\_epsilon  $E$ )  
:= proj2\_sig (constructive\_indefinite\_ground\_description  $E$ ).

End ConstructiveGroundEpsilon.

## Chapter 23

# Library Coq.Logic.Decidable

Properties of decidable propositions

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

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

**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).

```

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 24

# Library **Coq.Logic.Description**

This file provides a constructive form of definite description; it allows to build 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 :
```

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

## Chapter 25

# 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.

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

first we exhibit the choice functional relation R the actual “decision”: is (R class\_of\_true) = true or false? the actual “decision”: is (R class\_of\_false) = true or false? case where P is false: (R class\_of\_true)=true /\ (R class\_of\_false)=false cases where P is true

End PredExt\_RelChoice\_imp\_EM.

## 25.2 B. Proof-Irrel. + Rel. Axiom of Choice -> Excl.-Middle for Equality

This is an adaptation of Diaconescu’s theorem, exploiting the form of extensionality provided by proof-irrelevance

Section ProofIrrel\_RelChoice\_imp\_EqEM.

Variable rel\_choice : RelationalChoice.

Variable proof\_irrelevance :  $\forall P:\text{Prop}, \forall x\ y:P, x=y$ .

Let  $a1$  and  $a2$  be two elements in some type  $A$

Variable A :Type.

Variables a1 a2 : A.

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

Definition A' := sigT (fun x => 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`.

## 25.3 Extensional Hilbert's epsilon description operator $\rightarrow$ Excluded-Middle

Proof sketch from Bell *Bell93* (with thanks to P. Castéran)

**Local Notation** `inhabited`  $A := A$  (*only parsing*).

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

**Local Notation** `eps` := (`epsilon bool true`) (*only parsing*).

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

**End** `ExtensionalEpsilon_imp_EM`.



## Chapter 26

# 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 27

# 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

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

**Set Implicit Arguments.**

**Section EqdepDec.**

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

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

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

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

**Variable**  $x : A.$

**Let**  $\text{nu} (y:A) (u:x = y) : x = y :=$   
     $\text{match eq\_dec } x\ y \text{ with}$   
         $| \text{or\_introl } eqxy \Rightarrow eqxy$   
         $| \text{or\_intror } neqxy \Rightarrow \text{False\_ind\_} \_ (\text{neqxy } u)$   
     $\text{end}.$

Let  $nu\_constant : \forall (y:A) (u\ v:x = y), nu\ u = nu\ v$ .

Qed.

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

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

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

Theorem  $K\_dec :$

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

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' x with
  | or_introl eqprf => eq_ind x' P prf x eqprf
  | _ => def
end
end.
```

Theorem  $inj\_right\_pair :$

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

End  $EqdepDec$ .

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  $Theorem\ eq\_rect\_eq\_dec :$

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

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

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

Theorem  $UIP\_dec :$

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

Unset Implicit Arguments.

### 27.1.1 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.
```

```
Parameter U:Type.
```

```
Axiom eq_dec :  $\forall x\ y:U, \{x = y\} + \{x \neq y\}$ .
```

```
End DECIDABLETYPE.
```

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

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

```
Import M.
```

Invariance by Substitution of Reflexive Equality Proofs

```
Lemma eq_rect_eq :
```

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

Injectivity of Dependent Equality

```
Theorem eq_dep_eq :
```

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

Uniqueness of Identity Proofs (UIP)

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

Uniqueness of Reflexive Identity Proofs

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

Streicher's axiom K

```
Lemma Streicher_K :
```

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

Injectivity of equality on dependent pairs in `Type`

```
Lemma inj_pairT2 :
```

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

Proof-irrelevance on subsets of decidable sets

```
Lemma inj_pairP2 :
```

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

```
End DECIDABLEEQDEP.
```

### 27.1.2 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:Type.`

`Axiom eq_dec :  $\forall x y:U, \{x = y\} + \{x \neq y\}$ .`

`End DECIDABLESET.`

The module *DecidableEqDepSet* collects equality properties for decidable set in `Set`

`Module DECIDABLEEQDEPSET (M:DECIDABLESET).`

`Import M.`

`Module N:=DECIDABLEEQDEP(M).`

Invariance by Substitution of Reflexive Equality Proofs

`Lemma eq_rect_eq :`

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

Injectivity of Dependent Equality

`Theorem eq_dep_eq :`

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

Uniqueness of Identity Proofs (UIP)

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

Uniqueness of Reflexive Identity Proofs

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

Streicher's axiom K

`Lemma Streicher_K :`

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

Proof-irrelevance on subsets of decidable sets

`Lemma inj_pairP2 :`

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

Injectivity of equality on dependent pairs in `Type`

`Lemma inj_pair2 :`

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

Injectivity of equality on dependent pairs with second component in `Type`

`Notation inj_pairT2 := inj_pair2.`

`End DECIDABLEEQDEPSET.`

From decidability to `inj_pair2` `Lemma inj_pair2_eq_dec :  $\forall A:\text{Type}, (\forall x y:A, \{x=y\} + \{x \neq y\}) \rightarrow$   
 $(\forall (P:A \rightarrow \text{Type}) (p:A) (x y:P p), \text{existT } P \ p \ x = \text{existT } P \ p \ y \rightarrow x = y)$ .`

# Chapter 28

## 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`

### 28.1 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_eq_dep := eq_sigT_iff_eq_dep (only parsing).

Lemma eq_sig_eq_dep :
  ∀ (U:Prop) (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:Prop) (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**  $\text{eq\_sig\_iff\_eq\_dep}$  :

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

Dependent equality is equivalent to a dependent pair of equalities

**Set Implicit Arguments.**

**Lemma**  $\text{eq\_sigT\_sig\_eq}$  :  $\forall X \ P \ (x1 \ x2:X) \ H1 \ H2, \text{existT } P \ x1 \ H1 = \text{existT } P \ x2 \ H2 \leftrightarrow \{H:x1=x2 \mid \text{rew } H \text{ in } H1 = H2\}.$

**Lemma**  $\text{eq\_sigT\_fst}$  :

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

**Lemma**  $\text{eq\_sigT\_snd}$  :

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

**Lemma**  $\text{eq\_sig\_fst}$  :

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

**Lemma**  $\text{eq\_sig\_snd}$  :

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

**Unset Implicit Arguments.**

Exported hints

**Hint**  $\text{Resolve eq\_dep\_intro}$ : *core*.

**Hint**  $\text{Immediate eq\_dep\_sym}$ : *core*.

## 28.2 $\text{Eq\_rect\_eq} \leftrightarrow \text{Eq\_dep\_eq} \leftrightarrow \text{UIP} \leftrightarrow \text{UIP\_refl} \leftrightarrow \text{K}$

**Section**  $\text{Equivalences}$ .

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

Invariance by Substitution of Reflexive Equality Proofs

**Definition**  $\text{Eq\_rect\_eq}$  :=

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

Injectivity of Dependent Equality

**Definition**  $\text{Eq\_dep\_eq}$  :=

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

Uniqueness of Identity Proofs (UIP)

**Definition**  $\text{UIP\_}$  :=

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

Uniqueness of Reflexive Identity Proofs

**Definition**  $\text{UIP\_refl\_}$  :=

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

Streicher's axiom K

**Definition** `Streicher_K_` :=

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

Injectivity of Dependent Equality is a consequence of Invariance by Substitution of Reflexive Equality Proof

**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__eq_dep_eq` : `Eq_rect_eq`  $\rightarrow$  `Eq_dep_eq`.

Uniqueness of Identity Proofs (UIP) is a consequence of Injectivity of Dependent Equality

**Lemma** `eq_dep_eq__UIP` : `Eq_dep_eq`  $\rightarrow$  `UIP_`.

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

**Lemma** `UIP__UIP_refl` : `UIP_`  $\rightarrow$  `UIP_refl_`.

Streicher's axiom K is a direct consequence of Uniqueness of Reflexive Identity Proofs

**Lemma** `UIP_refl__Streicher_K` : `UIP_refl_`  $\rightarrow$  `Streicher_K_`.

We finally recover from K the Invariance by Substitution of Reflexive Equality Proofs

**Lemma** `Streicher_K__eq_rect_eq` : `Streicher_K_`  $\rightarrow$  `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 to prove 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  $\rightarrow$  x=y` which is in `Type` but not in `Set`.

**End** Equivalences.

**Section** Corollaries.

**Variable** `U`:`Type`.

UIP implies the injectivity of equality on dependent pairs in `Type`

**Definition** `Inj_dep_pair` :=

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

**Lemma** `eq_dep_eq__inj_pair2` : `Eq_dep_eq` `U`  $\rightarrow$  `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`.

## 28.3 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:EQDEPELIMINATION).

  Section Axioms.

    Variable U: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.

```

## Chapter 29

# 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 v62.
```

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

## Chapter 30

### 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 :  $\forall \{A\ B : \text{Type}\} \{f\ g : A \rightarrow B\},$   
   $f = g \rightarrow \forall x, f\ x = g\ x.$ 
```

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) :  
   $(\forall x, f\ x = g\ x) \rightarrow f = g.$ 
```

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)) ; intro x  
  end.
```

Eta expansion follows from extensionality.

```
Lemma eta_expansion_dep {A} {B : A  $\rightarrow$  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 31

# Library **Coq.Logic.Hurkens**

This is Hurkens paradox *Hurkens* in system U-, adapted by Herman Geuvers *Geuvers* to show the inconsistency in the pure calculus of constructions of a retract from Prop into a small type.

References:

- *Hurkens* A. J. Hurkens, “A simplification of Girard’s paradox”, Proceedings of the 2nd international conference Typed Lambda-Calculi and Applications (TLCA’95), 1995.
- *Geuvers* “Inconsistency of Classical Logic in Type Theory”, 2001 (see <http://www.cs.kun.nl/~herman/note.ps>).

Section Paradox.

Variable *bool* : Prop.

Variable *p2b* : Prop → bool.

Variable *b2p* : bool → Prop.

Hypothesis *p2p1* : ∀ *A*:Prop, *b2p* (*p2b* *A*) → *A*.

Hypothesis *p2p2* : ∀ *A*:Prop, *A* → *b2p* (*p2b* *A*).

Variable *B* : Prop.

Definition *V* := ∀ *A*:Prop, ((*A* → bool) → *A* → bool) → *A* → bool.

Definition *U* := *V* → bool.

Definition *sb* (*z*:*V*) : *V* := fun *A r a* => *r* (*z* *A r*) *a*.

Definition *le* (*i*:*U* → bool) (*x*:*U*) : bool :=

*x* (fun *A r a* => *i* (fun *v* => *sb* *v* *A r a*)).

Definition *induct* (*i*:*U* → bool) : Prop :=

∀ *x*:*U*, *b2p* (*le* *i* *x*) → *b2p* (*i* *x*).

Definition *WF* : *U* := fun *z* => *p2b* (*induct* (*z* *U* *le*)).

Definition *I* (*x*:*U*) : Prop :=

(∀ *i*:*U* → bool, *b2p* (*le* *i* *x*) → *b2p* (*i* (fun *v* => *sb* *v* *U* *le* *x*))) → *B*.

Lemma *Omega* : ∀ *i*:*U* → bool, *induct* *i* → *b2p* (*i* *WF*).

Lemma *lemma1* : *induct* (fun *u* => *p2b* (*I* *u*)).

Lemma *lemma2* : (∀ *i*:*U* → bool, *induct* *i* → *b2p* (*i* *WF*)) → *B*.

Theorem *paradox* : *B*.

End Paradox.

## Chapter 32

# Library **Coq.Logic.IndefiniteDescription**

This file provides a constructive form of indefinite description that allows to build 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 33

# 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.**

**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:Type) (x:A) (B:Type) (f:A  $\rightarrow$  B) (y:A), JMeq x y  $\rightarrow$  f x = f y.



$JMeq$  is equivalent to  $eq\_dep \text{ Type } (\text{fun } X \Rightarrow X)$

**Require Import** Eqdep.

**Lemma** JMeq\_eq\_dep\_id :

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

**Lemma** eq\_dep\_id\_JMeq :

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

$eq\_dep\ U\ P\ p\ x\ q\ y$  is strictly finer than  $JMeq\ (P\ p)\ x\ (P\ q)\ y$

**Lemma** eq\_dep\_JMeq :

$\forall U\ P\ p\ x\ q\ y, eq\_dep\ U\ P\ p\ x\ q\ y \rightarrow JMeq\ x\ y.$

**Lemma** eq\_dep\_strictly\_stronger\_JMeq :

$\exists U\ P\ p\ q\ x\ y, JMeq\ x\ y \wedge \neg eq\_dep\ U\ P\ p\ x\ q\ y.$

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

**Lemma** JMeq\_eq\_dep :

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

**Notation** sym\_JMeq := JMeq\_sym (*only parsing*).

**Notation** trans\_JMeq := JMeq\_trans (*only parsing*).

## Chapter 34

# Library **Coq.Logic.ProofIrrelevanceFacts**

This defines the functor that build consequences of proof-irrelevance

```
Require Export EqdepFacts.
```

```
Module Type PROOFIRRELEVANCE.
```

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

```
End PROOFIRRELEVANCE.
```

```
Module PROOFIRRELEVANCETHEORY ( $M:\mathbf{PROOFIRRELEVANCE}$ ).
```

Proof-irrelevance implies uniqueness of reflexivity proofs

```
Module EQ_RECT_EQ.
```

```
  Lemma eq_rect_eq :
```

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

```
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 :
```

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

```
Lemma subsetT_eq_compat :
```

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

```
End PROOFIRRELEVANCETHEORY.
```

## Chapter 35

# 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 36

# 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 37

# Library `Coq.Logic.SetIsType`

### 37.1 The Set universe seen as a synonym for Type

After loading this file, Set becomes just another name for Type. This allows to easily perform 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 38

# Library `Coq.Arith.Arith_base`

```
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 39

# Library **Coq.Arith.Arith**

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

## Chapter 40

# 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 v62.

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

  Hint Resolve bet_eq: arith v62.

  Lemma between_le : ∀ k l, between k l → k ≤ l.
  Hint Immediate between_le: arith v62.

  Lemma between_Sk_l : ∀ k l, between k l → S k ≤ l → between (S k) l.
  Hint Resolve between_Sk_l: arith v62.

  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 v62.

  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 v62.

  Lemma exists_S_le : ∀ k l, exists_between k (S l) → k ≤ l.
  Hint Immediate exists_S_le: arith v62.
```



**Definition** `in_int`  $p\ q\ r := p \leq r \wedge r < q$ .  
**Lemma** `in_int_intro` :  $\forall p\ q\ r, p \leq r \rightarrow r < q \rightarrow \text{in\_int } p\ q\ r$ .  
**Hint** `Resolve in_int_intro`: *arith v62*.  
**Lemma** `in_int_lt` :  $\forall p\ q\ r, \text{in\_int } p\ q\ r \rightarrow p < q$ .  
**Lemma** `in_int_p_Sq` :  
 $\forall p\ q\ r, \text{in\_int } p\ (\text{S } q)\ r \rightarrow \text{in\_int } p\ q\ r \vee r = q$  :>**nat**.  
**Lemma** `in_int_S` :  $\forall p\ q\ r, \text{in\_int } p\ q\ r \rightarrow \text{in\_int } p\ (\text{S } q)\ r$ .  
**Hint** `Resolve in_int_S`: *arith v62*.  
**Lemma** `in_int_Sp_q` :  $\forall p\ q\ r, \text{in\_int } (\text{S } p)\ q\ r \rightarrow \text{in\_int } p\ q\ r$ .  
**Hint** `Immediate in_int_Sp_q`: *arith v62*.  
**Lemma** `between_in_int` :  
 $\forall k\ l, \text{between } k\ l \rightarrow \forall r, \text{in\_int } k\ l\ r \rightarrow P\ r$ .  
**Lemma** `in_int_between` :  
 $\forall k\ l, k \leq l \rightarrow (\forall r, \text{in\_int } k\ l\ r \rightarrow P\ r) \rightarrow \text{between } k\ l$ .  
**Lemma** `exists_in_int` :  
 $\forall k\ l, \text{exists\_between } k\ l \rightarrow \text{exists2 } m : \text{nat}, \text{in\_int } k\ l\ m \ \& \ Q\ m$ .  
**Lemma** `in_int_exists` :  $\forall k\ l\ r, \text{in\_int } k\ l\ r \rightarrow Q\ r \rightarrow \text{exists\_between } k\ l$ .  
**Lemma** `between_or_exists` :  
 $\forall k\ l,$   
 $k \leq l \rightarrow$   
 $(\forall n:\text{nat}, \text{in\_int } k\ l\ n \rightarrow P\ n \vee Q\ n) \rightarrow$   
 $\text{between } k\ l \vee \text{exists\_between } k\ l$ .  
**Lemma** `between_not_exists` :  
 $\forall k\ l,$   
 $\text{between } k\ l \rightarrow$   
 $(\forall n:\text{nat}, \text{in\_int } k\ l\ n \rightarrow P\ n \rightarrow \neg Q\ n) \rightarrow \neg \text{exists\_between } k\ l$ .  
**Inductive** `P_nth` (*init*:**nat**) : **nat**  $\rightarrow$  **nat**  $\rightarrow$  **Prop** :=  
| `nth_O` : `P_nth` *init* *init* 0  
| `nth_S` :  
 $\forall k\ l\ (n:\text{nat}),$   
 $\text{P\_nth } \text{init } k\ n \rightarrow \text{between } (\text{S } k)\ l \rightarrow Q\ l \rightarrow \text{P\_nth } \text{init } l\ (\text{S } n)$ .  
**Lemma** `nth_le` :  $\forall (init:\text{nat})\ l\ (n:\text{nat}), \text{P\_nth } \text{init } l\ n \rightarrow \text{init} \leq l$ .  
**Definition** `eventually` (*n*:**nat**) := `exists2` *k* : **nat**,  $k \leq n \ \& \ Q\ k$ .  
**Lemma** `event_O` : `eventually` 0  $\rightarrow Q\ 0$ .  
**End** `Between`.  
**Hint** `Resolve nth_O bet_S bet_emp bet_eq between_Sk_l exists_S exists_le`  
`in_int_S in_int_intro`: *arith v62*.  
**Hint** `Immediate in_int_Sp_q exists_le_S exists_S_le`: *arith v62*.

## Chapter 41

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

```
Require Export Compare_dec.  
Require Export Peano_dec.  
Require Import Sumbool.  
Local Open Scope nat_scope.  
Implicit Types m n x y : nat.
```

The decidability of equality and order relations over type *nat* give some boolean functions with the adequate specification.

```
Definition notzerop n := sumbool_not _ _ (zerop n).  
Definition lt_ge_dec :  $\forall x y, \{x < y\} + \{x \geq y\}$  :=  
  fun n m  $\Rightarrow$  sumbool_not _ _ (le_lt_dec m n).  
Definition nat_lt_ge_bool x y := bool_of_sumbool (lt_ge_dec x y).  
Definition nat_ge_lt_bool x y :=  
  bool_of_sumbool (sumbool_not _ _ (lt_ge_dec x y)).  
Definition nat_le_gt_bool x y := bool_of_sumbool (le_gt_dec x y).  
Definition nat_gt_le_bool x y :=  
  bool_of_sumbool (sumbool_not _ _ (le_gt_dec x y)).  
Definition nat_eq_bool x y := bool_of_sumbool (eq_nat_dec x y).  
Definition nat_noteq_bool x y :=  
  bool_of_sumbool (sumbool_not _ _ (eq_nat_dec x y)).  
Definition zerop_bool x := bool_of_sumbool (zerop x).  
Definition notzerop_bool x := bool_of_sumbool (notzerop x).
```

## Chapter 42

# Library `Coq.Arith.Compare_dec`

```
Require Import Le.
Require Import Lt.
Require Import Gt.
Require Import Decidable.

Local Open Scope nat_scope.

Implicit Types m n x y : nat.

Definition zerop n : {n = 0} + {0 < n}.

Definition lt_eq_lt_dec n m : {n < m} + {n = m} + {m < n}.

Definition gt_eq_gt_dec n m : {m > n} + {n = m} + {n > m}.

Definition le_lt_dec n m : {n ≤ m} + {m < n}.

Definition le_le_S_dec n m : {n ≤ m} + {S m ≤ n}.

Definition le_ge_dec n m : {n ≤ m} + {n ≥ m}.

Definition le_gt_dec n m : {n ≤ m} + {n > m}.

Definition le_lt_eq_dec n m : n ≤ m → {n < m} + {n = m}.

Theorem le_dec : ∀ n m, {n ≤ m} + {¬ n ≤ m}.

Theorem lt_dec : ∀ n m, {n < m} + {¬ n < m}.

Theorem gt_dec : ∀ n m, {n > m} + {¬ n > m}.

Theorem ge_dec : ∀ n m, {n ≥ m} + {¬ n ≥ m}.
```

Proofs of decidability

```
Theorem dec_le : ∀ n m, decidable (n ≤ m).
Theorem dec_lt : ∀ n m, decidable (n < m).
Theorem dec_gt : ∀ n m, decidable (n > m).
Theorem dec_ge : ∀ n m, decidable (n ≥ m).
Theorem not_eq : ∀ n m, n ≠ m → n < m ∨ m < n.
Theorem not_le : ∀ n m, ¬ n ≤ m → n > m.
```

Theorem not\_gt :  $\forall n\ m, \neg n > m \rightarrow n \leq m$ .

Theorem not\_ge :  $\forall n\ m, \neg n \geq m \rightarrow n < m$ .

Theorem not\_lt :  $\forall n\ m, \neg n < m \rightarrow n \geq m$ .

A ternary comparison function in the spirit of *Z.compare*.

```
Fixpoint nat_compare n m :=
  match n, m with
  | O, O => Eq
  | O, S _ => Lt
  | S _, O => Gt
  | S n', S m' => nat_compare n' m'
  end.
```

Lemma nat\_compare\_S :  $\forall n\ m, \text{nat\_compare } (S\ n) (S\ m) = \text{nat\_compare } n\ m$ .

Lemma nat\_compare\_eq\_iff :  $\forall n\ m, \text{nat\_compare } n\ m = \text{Eq} \leftrightarrow n = m$ .

Lemma nat\_compare\_eq :  $\forall n\ m, \text{nat\_compare } n\ m = \text{Eq} \rightarrow n = m$ .

Lemma nat\_compare\_lt :  $\forall n\ m, n < m \leftrightarrow \text{nat\_compare } n\ m = \text{Lt}$ .

Lemma nat\_compare\_gt :  $\forall n\ m, n > m \leftrightarrow \text{nat\_compare } n\ m = \text{Gt}$ .

Lemma nat\_compare\_le :  $\forall n\ m, n \leq m \leftrightarrow \text{nat\_compare } n\ m \neq \text{Gt}$ .

Lemma nat\_compare\_ge :  $\forall n\ m, n \geq m \leftrightarrow \text{nat\_compare } n\ m \neq \text{Lt}$ .

Lemma nat\_compare\_spec :

$\forall x\ y, \text{CompareSpec } (x=y) (x<y) (y<x) (\text{nat\_compare } x\ y)$ .

Some projections of the above equivalences.

Lemma nat\_compare\_Lt\_Lt :  $\forall n\ m, \text{nat\_compare } n\ m = \text{Lt} \rightarrow n < m$ .

Lemma nat\_compare\_Gt\_gt :  $\forall n\ m, \text{nat\_compare } n\ m = \text{Gt} \rightarrow n > m$ .

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

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

Lemma nat\_compare\_equiv :  $\forall n\ m,$   
 $\text{nat\_compare } n\ m = \text{nat\_compare\_alt } n\ m$ .

A boolean version of *le* over *nat*.

```
Fixpoint leb (m:nat) : nat → bool :=
  match m with
  | O => fun _:nat => true
  | S m' =>
    fun n:nat => match n with
```

```

      | O  $\Rightarrow$  false
      | S  $n'$   $\Rightarrow$  leb  $m'$   $n'$ 
    end

  end.

Lemma leb_correct :  $\forall m\ n, m \leq n \rightarrow \text{leb } m\ n = \text{true}$ .
Lemma leb_complete :  $\forall m\ n, \text{leb } m\ n = \text{true} \rightarrow m \leq n$ .
Lemma leb_iff :  $\forall m\ n, \text{leb } m\ n = \text{true} \leftrightarrow m \leq n$ .
Lemma leb_correct_conv :  $\forall m\ n, m < n \rightarrow \text{leb } n\ m = \text{false}$ .
Lemma leb_complete_conv :  $\forall m\ n, \text{leb } n\ m = \text{false} \rightarrow m < n$ .
Lemma leb_iff_conv :  $\forall m\ n, \text{leb } n\ m = \text{false} \leftrightarrow m < n$ .
Lemma leb_compare :  $\forall n\ m, \text{leb } n\ m = \text{true} \leftrightarrow \text{nat\_compare } n\ m \neq \text{Gt}$ .

```

## Chapter 43

# 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 44

# Library `Coq.Arith.Div2`

```
Require Import Lt.
Require Import Plus.
Require Import Compare_dec.
Require Import Even.

Local Open Scope nat_scope.

Implicit Type n : nat.
```

Here we define  $n/2$  and prove some of its properties

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

Since *div2* is recursively defined on 0, 1 and  $(S (S n))$ , it is useful to prove the corresponding induction principle

```
Lemma ind_0_1_SS :
  ∀ P:nat → Prop,
    P 0 → P 1 → (∀ n, P n → P (S (S n))) → ∀ n, P n.

0 < n ⇒ n/2 < n
```

```
Lemma lt_div2 : ∀ n, 0 < n → div2 n < n.
```

```
Hint Resolve lt_div2: arith.
```

Properties related to the parity

```
Lemma even_div2 : ∀ n, even n → div2 n = div2 (S n)
with odd_div2 : ∀ n, odd n → S (div2 n) = div2 (S n).
```

```
Lemma div2_even n : div2 n = div2 (S n) → even n
with div2_odd n : S (div2 n) = div2 (S n) → odd n.
```

```
Hint Resolve even_div2 div2_even odd_div2 div2_odd: arith.
```

```
Lemma even_odd_div2 n :
```

(*even*  $n \leftrightarrow \text{div2 } n = \text{div2 } (\text{S } n)$ )  $\wedge$   
(*odd*  $n \leftrightarrow \text{S } (\text{div2 } n) = \text{div2 } (\text{S } n)$ ).

Properties related to the double ( $2n$ )

**Definition** *double*  $n := n + n$ .

**Hint** *Unfold* *double*: *arith*.

**Lemma** *double\_S* :  $\forall n, \text{double } (\text{S } n) = \text{S } (\text{double } n)$ .

**Lemma** *double\_plus* :  $\forall n (m:\text{nat}), \text{double } (n + m) = \text{double } n + \text{double } m$ .

**Hint** *Resolve* *double\_S*: *arith*.

**Lemma** *even\_odd\_double* :

$\forall n,$   
(*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* :  $\forall n, \text{even } n \rightarrow n = \text{double } (\text{div2 } n)$ .

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

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

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

**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* :  $\forall n:\text{nat}, \text{div2 } (2 \times n) = n$ .

**Lemma** *div2\_double\_plus\_one* :  $\forall n:\text{nat}, \text{div2 } (\text{S } (2 \times n)) = n$ .



## Chapter 45

# Library **Coq.Arith.EqNat**

Equality on natural numbers

**Local Open Scope** *nat\_scope*.

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

### 45.1 Propositional equality

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

**Theorem** eq\_nat\_refl :  $\forall n$ , eq\_nat *n n*.

**Hint Resolve** eq\_nat\_refl: *arith v62*.

*eq* restricted to *nat* and *eq\_nat* are equivalent

**Lemma** eq\_eq\_nat :  $\forall n m$ ,  $n = m \rightarrow$  eq\_nat *n m*.

**Hint Immediate** eq\_eq\_nat: *arith v62*.

**Lemma** eq\_nat\_eq :  $\forall n m$ , eq\_nat *n m*  $\rightarrow n = m$ .

**Hint Immediate** eq\_nat\_eq: *arith v62*.

**Theorem** eq\_nat\_is\_eq :  $\forall n m$ , eq\_nat *n m*  $\leftrightarrow n = m$ .

**Theorem** eq\_nat\_elim :

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

**Theorem** eq\_nat\_decide :  $\forall n m$ , {eq\_nat *n m*} + { $\neg$  eq\_nat *n m*}.

### 45.2 Boolean equality on *nat*

```
Fixpoint beq_nat n m : bool :=
```

```

match n, m with
| O, O  $\Rightarrow$  true
| O, S _  $\Rightarrow$  false
| S _, O  $\Rightarrow$  false
| S n1, S m1  $\Rightarrow$  beq_nat n1 m1
end.

Lemma beq_nat_refl :  $\forall$  n, true = beq_nat n n.

Definition beq_nat_eq :  $\forall$  x y, true = beq_nat x y  $\rightarrow$  x = y.

Lemma beq_nat_true :  $\forall$  x y, beq_nat x y = true  $\rightarrow$  x=y.

Lemma beq_nat_false :  $\forall$  x y, beq_nat x y = false  $\rightarrow$  x $\neq$ y.

Lemma beq_nat_true_iff :  $\forall$  x y, beq_nat x y = true  $\leftrightarrow$  x=y.

Lemma beq_nat_false_iff :  $\forall$  x y, beq_nat x y = false  $\leftrightarrow$  x $\neq$ y.

```

## Chapter 46

# 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 47

# Library **Coq.Arith.Even**

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.

```
Local Open Scope nat_scope.
```

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

### 47.1 Definition of *even* and *odd*, and basic facts

```
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.

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.
```

### 47.2 Facts about *even* & *odd* wrt. *plus*

```
Lemma even_plus_split : ∀ n m,
  (even (n + m) → even n ∧ even m ∨ odd n ∧ odd m)
with 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)
with 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)
with odd_even_plus : ∀ n m, odd n → odd m → even (n + m).
```

Lemma even\_plus\_aux :  $\forall n m,$   
 $(\text{odd } (n + m) \leftrightarrow \text{odd } n \wedge \text{even } m \vee \text{even } n \wedge \text{odd } m) \wedge$   
 $(\text{even } (n + m) \leftrightarrow \text{even } n \wedge \text{even } m \vee \text{odd } n \wedge \text{odd } m).$

Lemma even\_plus\_even\_inv\_r :  $\forall n m, \text{even } (n + m) \rightarrow \text{even } n \rightarrow \text{even } m.$

Lemma even\_plus\_even\_inv\_l :  $\forall n m, \text{even } (n + m) \rightarrow \text{even } m \rightarrow \text{even } n.$

Lemma even\_plus\_odd\_inv\_r :  $\forall n m, \text{even } (n + m) \rightarrow \text{odd } n \rightarrow \text{odd } m.$

Lemma even\_plus\_odd\_inv\_l :  $\forall n m, \text{even } (n + m) \rightarrow \text{odd } m \rightarrow \text{odd } n.$

Hint Resolve even\_even\_plus odd\_even\_plus: *arith*.

Lemma odd\_plus\_even\_inv\_l :  $\forall n m, \text{odd } (n + m) \rightarrow \text{odd } m \rightarrow \text{even } n.$

Lemma odd\_plus\_even\_inv\_r :  $\forall n m, \text{odd } (n + m) \rightarrow \text{odd } n \rightarrow \text{even } m.$

Lemma odd\_plus\_odd\_inv\_l :  $\forall n m, \text{odd } (n + m) \rightarrow \text{even } m \rightarrow \text{odd } n.$

Lemma odd\_plus\_odd\_inv\_r :  $\forall n m, \text{odd } (n + m) \rightarrow \text{even } n \rightarrow \text{odd } m.$

Hint Resolve odd\_plus\_l odd\_plus\_r: *arith*.

### 47.3 Facts about *even* and *odd* wrt. *mult*

Lemma even\_mult\_aux :  
 $\forall n m,$   
 $(\text{odd } (n \times m) \leftrightarrow \text{odd } n \wedge \text{odd } m) \wedge (\text{even } (n \times m) \leftrightarrow \text{even } n \vee \text{even } m).$

Lemma even\_mult\_l :  $\forall n m, \text{even } n \rightarrow \text{even } (n \times m).$

Lemma even\_mult\_r :  $\forall n m, \text{even } m \rightarrow \text{even } (n \times m).$

Hint Resolve even\_mult\_l even\_mult\_r: *arith*.

Lemma even\_mult\_inv\_r :  $\forall n m, \text{even } (n \times m) \rightarrow \text{odd } n \rightarrow \text{even } m.$

Lemma even\_mult\_inv\_l :  $\forall n m, \text{even } (n \times m) \rightarrow \text{odd } m \rightarrow \text{even } n.$

Lemma odd\_mult :  $\forall n m, \text{odd } n \rightarrow \text{odd } m \rightarrow \text{odd } (n \times m).$

Hint Resolve even\_mult\_l even\_mult\_r odd\_mult: *arith*.

Lemma odd\_mult\_inv\_l :  $\forall n m, \text{odd } (n \times m) \rightarrow \text{odd } n.$

Lemma odd\_mult\_inv\_r :  $\forall n m, \text{odd } (n \times m) \rightarrow \text{odd } m.$

## Chapter 48

# Library **Coq.Arith.Factorial**

```
Require Import Plus.  
Require Import Mult.  
Require Import 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:nat, 0 < fact n.
```

```
Lemma fact_neq_0 : ∀ n:nat, fact n ≠ 0.
```

```
Lemma fact_le : ∀ n m:nat, n ≤ m → fact n ≤ fact m.
```

## Chapter 49

# Library **Coq.Arith.Gt**

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

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

```
Require Import Le.  
Require Import Lt.  
Require Import Plus.  
Local Open Scope nat_scope.  
Implicit Types m n p : nat.
```

### 49.1 Order and successor

```
Theorem gt_Sn_O :  $\forall n, S\ n > 0$ .  
Hint Resolve gt_Sn_O: arith v62.  
  
Theorem gt_Sn_n :  $\forall n, S\ n > n$ .  
Hint Resolve gt_Sn_n: arith v62.  
  
Theorem gt_n_S :  $\forall n\ m, n > m \rightarrow S\ n > S\ m$ .  
Hint Resolve gt_n_S: arith v62.  
  
Lemma gt_S_n :  $\forall n\ m, S\ m > S\ n \rightarrow m > n$ .  
Hint Immediate gt_S_n: arith v62.  
  
Theorem gt_S :  $\forall n\ m, S\ n > m \rightarrow n > m \vee m = n$ .  
  
Lemma gt_pred :  $\forall n\ m, m > S\ n \rightarrow pred\ m > n$ .  
Hint Immediate gt_pred: arith v62.
```

### 49.2 Irreflexivity

```
Lemma gt_irrefl :  $\forall n, \neg n > n$ .  
Hint Resolve gt_irrefl: arith v62.
```

### 49.3 Asymmetry

Lemma `gt_asym` :  $\forall n\ m, n > m \rightarrow \neg m > n$ .

Hint `Resolve` `gt_asym`: *arith v62*.

### 49.4 Relating strict and large orders

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

Hint `Resolve` `le_not_gt`: *arith v62*.

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

Hint `Resolve` `gt_not_le`: *arith v62*.

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

Hint `Immediate` `le_S_gt`: *arith v62*.

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

Hint `Immediate` `gt_S_le`: *arith v62*.

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

Hint `Resolve` `gt_le_S`: *arith v62*.

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

Hint `Resolve` `le_gt_S`: *arith v62*.

### 49.5 Transitivity

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

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

Lemma `gt_trans` :  $\forall n\ m\ p, n > m \rightarrow m > p \rightarrow n > p$ .

Theorem `gt_trans_S` :  $\forall n\ m\ p, S\ n > m \rightarrow m > p \rightarrow n > p$ .

Hint `Resolve` `gt_trans_S` `le_gt_trans` `gt_le_trans`: *arith v62*.

### 49.6 Comparison to 0

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

### 49.7 Simplification and compatibility

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

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

Hint `Resolve` `plus_gt_compat_l`: *arith v62*.



## Chapter 50

# Library **Coq.Arith.Le**

Order on natural numbers. *le* is defined in *Init/Peano.v* as:

```
Inductive le (n:nat) : nat -> Prop :=  
  | le_n : n <= n  
  | le_S : forall m:nat, n <= m -> n <= S m
```

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

```
Local Open Scope nat_scope.
```

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

### 50.1 *le* is a pre-order

Reflexivity **Theorem** *le\_refl* :  $\forall n, n \leq n$ .

Transitivity **Theorem** *le\_trans* :  $\forall n\ m\ p, n \leq m \rightarrow m \leq p \rightarrow n \leq p$ .

**Hint Resolve** *le\_trans*: *arith v62*.

### 50.2 Properties of *le* w.r.t. successor, predecessor and 0

Comparison to 0

**Theorem** *le\_0\_n* :  $\forall n, 0 \leq n$ .

**Theorem** *le\_Sn\_0* :  $\forall n, \neg S\ n \leq 0$ .

**Hint Resolve** *le\_0\_n le\_Sn\_0*: *arith v62*.

**Theorem** *le\_n\_0\_eq* :  $\forall n, n \leq 0 \rightarrow 0 = n$ .

**Hint Immediate** *le\_n\_0\_eq*: *arith v62*.

*le* and successor

**Theorem** *le\_n\_S* :  $\forall n\ m, n \leq m \rightarrow S\ n \leq S\ m$ .

**Theorem** *le\_n\_Sn* :  $\forall n, n \leq S\ n$ .

**Hint Resolve** *le\_n\_S le\_n\_Sn*: *arith v62*.

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

**Hint Immediate** `le_Sn_le`: *arith v62*.

**Theorem** `le_S_n` :  $\forall n\ m, \text{S } n \leq \text{S } m \rightarrow n \leq m$ .

**Hint Immediate** `le_S_n`: *arith v62*.

**Theorem** `le_Sn_n` :  $\forall n, \neg \text{S } n \leq n$ .

**Hint Resolve** `le_Sn_n`: *arith v62*.

*le* and predecessor

**Theorem** `le_pred_n` :  $\forall n, \text{pred } n \leq n$ .

**Hint Resolve** `le_pred_n`: *arith v62*.

**Theorem** `le_pred` :  $\forall n\ m, n \leq m \rightarrow \text{pred } n \leq \text{pred } m$ .

### 50.3 *le* is a order on *nat*

Antisymmetry

**Theorem** `le_antisym` :  $\forall n\ m, n \leq m \rightarrow m \leq n \rightarrow n = m$ .

**Hint Immediate** `le_antisym`: *arith v62*.

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

**Lemma** `le_elim_rel` :

$\forall P:\text{nat} \rightarrow \text{nat} \rightarrow \text{Prop},$

$(\forall p, P\ 0\ p) \rightarrow$

$(\forall p\ (q:\text{nat}), p \leq q \rightarrow P\ p\ q \rightarrow P\ (\text{S } p)\ (\text{S } q)) \rightarrow$

$\forall n\ m, n \leq m \rightarrow P\ n\ m.$

## Chapter 51

# Library **Coq.Arith.Lt**

Theorems about *lt* in nat. *lt* is defined in library *Init/Peano.v* as:

```
Definition lt (n m:nat) := S n <= m.  
Infix "<" := lt : nat_scope.
```

```
Require Import Le.  
Local Open Scope nat_scope.  
Implicit Types m n p : nat.
```

### 51.1 Irreflexivity

```
Theorem lt_irrefl :  $\forall n, \neg n < n$ .  
Hint Resolve lt_irrefl: arith v62.
```

### 51.2 Relationship between *le* and *lt*

```
Theorem lt_le_S :  $\forall n\ m, n < m \rightarrow S\ n \leq m$ .  
Hint Immediate lt_le_S: arith v62.  
  
Theorem lt_n_Sm_le :  $\forall n\ m, n < S\ m \rightarrow n \leq m$ .  
Hint Immediate lt_n_Sm_le: arith v62.  
  
Theorem le_lt_n_Sm :  $\forall n\ m, n \leq m \rightarrow n < S\ m$ .  
Hint Immediate le_lt_n_Sm: arith v62.  
  
Theorem le_not_lt :  $\forall n\ m, n \leq m \rightarrow \neg m < n$ .  
Theorem lt_not_le :  $\forall n\ m, n < m \rightarrow \neg m \leq n$ .  
Hint Immediate le_not_lt lt_not_le: arith v62.
```

### 51.3 Asymmetry

```
Theorem lt_asym :  $\forall n\ m, n < m \rightarrow \neg m < n$ .
```

## 51.4 Order and successor

Theorem `lt_n_Sn` :  $\forall n, n < S\ n$ .

Hint `Resolve lt_n_Sn`: *arith v62*.

Theorem `lt_S` :  $\forall n\ m, n < m \rightarrow n < S\ m$ .

Hint `Resolve lt_S`: *arith v62*.

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

Hint `Resolve lt_n_S`: *arith v62*.

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

Hint `Immediate lt_S_n`: *arith v62*.

Theorem `lt_0_Sn` :  $\forall n, 0 < S\ n$ .

Hint `Resolve lt_0_Sn`: *arith v62*.

Theorem `lt_n_0` :  $\forall n, \neg n < 0$ .

Hint `Resolve lt_n_0`: *arith v62*.

## 51.5 Predecessor

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

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

Hint `Immediate lt_pred`: *arith v62*.

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

Hint `Resolve lt_pred_n_n`: *arith v62*.

## 51.6 Transitivity properties

Theorem `lt_trans` :  $\forall n\ m\ p, n < m \rightarrow m < p \rightarrow n < p$ .

Theorem `lt_le_trans` :  $\forall n\ m\ p, n < m \rightarrow m \leq p \rightarrow n < p$ .

Theorem `le_lt_trans` :  $\forall n\ m\ p, n \leq m \rightarrow m < p \rightarrow n < p$ .

Hint `Resolve lt_trans lt_le_trans le_lt_trans`: *arith v62*.

## 51.7 Large = strict or equal

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

Theorem `le_lt_or_eq_iff` :  $\forall n\ m, n \leq m \leftrightarrow n < m \vee n = m$ .

Theorem `lt_le_weak` :  $\forall n\ m, n < m \rightarrow n \leq m$ .

Hint `Immediate lt_le_weak`: *arith v62*.

## 51.8 Dichotomy

**Theorem** `le_or_lt` :  $\forall n\ m, n \leq m \vee m < n$ .

**Theorem** `nat_total_order` :  $\forall n\ m, n \neq m \rightarrow n < m \vee m < n$ .

## 51.9 Comparison to 0

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

**Hint Immediate** `neq_0_lt`: *arith v62*.

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

**Hint Immediate** `lt_0_neq`: *arith v62*.

## Chapter 52

# Library **Coq.Arith.Max**

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

```
Require Import NPeano.
```

```
Local Open Scope nat_scope.
```

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

```
Notation max := Peano.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 v62.
```

```
Hint Resolve
```

```
  Nat.min_l Nat.min_r Nat.le_min_l Nat.le_min_r : arith v62.
```

## Chapter 53

# Library **Coq.Arith.Minus**

*minus* (difference between two natural numbers) is defined in *Init/Peano.v* as:

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | 0, _ => n
  | S k, 0 => S k
  | S k, S l => k - l
  end
where "n - m" := (minus n m) : nat_scope.
```

```
Require Import Lt.
Require Import Le.
Local Open Scope nat_scope.
Implicit Types m n p : nat.
```

### 53.1 0 is right neutral

**Lemma** `minus_n_0` :  $\forall n, n = n - 0$ .  
**Hint** `Resolve minus_n_0`: *arith v62*.

### 53.2 Permutation with successor

**Lemma** `minus_Sn_m` :  $\forall n\ m, m \leq n \rightarrow S\ (n - m) = S\ n - m$ .  
**Hint** `Resolve minus_Sn_m`: *arith v62*.

**Theorem** `pred_of_minus` :  $\forall n, \text{pred } n = n - 1$ .

### 53.3 Diagonal

**Lemma** `minus_diag` :  $\forall n, n - n = 0$ .

Lemma minus\_diag\_reverse :  $\forall n, 0 = n - n$ .  
 Hint Resolve minus\_diag\_reverse: arith v62.  
 Notation minus\_n\_n := minus\_diag\_reverse.

## 53.4 Simplification

Lemma minus\_plus\_simpl\_l\_reverse :  $\forall n m p, n - m = p + n - (p + m)$ .  
 Hint Resolve minus\_plus\_simpl\_l\_reverse: arith v62.

## 53.5 Relation with plus

Lemma plus\_minus :  $\forall n m p, n = m + p \rightarrow p = n - m$ .  
 Hint Immediate plus\_minus: arith v62.  
 Lemma minus\_plus :  $\forall n m, n + m - n = m$ .  
 Hint Resolve minus\_plus: arith v62.  
 Lemma le\_plus\_minus :  $\forall n m, n \leq m \rightarrow m = n + (m - n)$ .  
 Hint Resolve le\_plus\_minus: arith v62.  
 Lemma le\_plus\_minus\_r :  $\forall n m, n \leq m \rightarrow n + (m - n) = m$ .  
 Hint Resolve le\_plus\_minus\_r: arith v62.

## 53.6 Relation with order

Theorem minus\_le\_compat\_r :  $\forall n m p : \text{nat}, n \leq m \rightarrow n - p \leq m - p$ .  
 Theorem minus\_le\_compat\_l :  $\forall n m p : \text{nat}, n \leq m \rightarrow p - m \leq p - n$ .  
 Corollary le\_minus :  $\forall n m, n - m \leq n$ .  
 Lemma lt\_minus :  $\forall n m, m \leq n \rightarrow 0 < m \rightarrow n - m < n$ .  
 Hint Resolve lt\_minus: arith v62.  
 Lemma lt\_O\_minus\_lt :  $\forall n m, 0 < n - m \rightarrow m < n$ .  
 Hint Immediate lt\_O\_minus\_lt: arith v62.  
 Theorem not\_le\_minus\_0 :  $\forall n m, \neg m \leq n \rightarrow n - m = 0$ .



## Chapter 54

# Library **Coq.Arith.Min**

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

```
Require Import NPeano.

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

Notation min := Peano.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 55

# Library **Coq.Arith.Mult**

```
Require Export Plus.  
Require Export Minus.  
Require Export Lt.  
Require Export Le.  
Local Open Scope nat_scope.  
Implicit Types m n p : nat.
```

Theorems about multiplication in *nat*. *mult* is defined in module *Init/Peano.v*.

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

#### 55.1.1 Zero property

Lemma `mult_0_r` :  $\forall n, n \times 0 = 0$ .

Lemma `mult_0_l` :  $\forall n, 0 \times n = 0$ .

#### 55.1.2 1 is neutral

Lemma `mult_1_l` :  $\forall n, 1 \times n = n$ .

Hint Resolve `mult_1_l`: *arith v62*.

Lemma `mult_1_r` :  $\forall n, n \times 1 = n$ .

Hint Resolve `mult_1_r`: *arith v62*.

#### 55.1.3 Commutativity

Lemma `mult_comm` :  $\forall n m, n \times m = m \times n$ .

Hint Resolve `mult_comm`: *arith v62*.

#### 55.1.4 Distributivity

Lemma `mult_plus_distr_r` :  $\forall n m p, (n + m) \times p = n \times p + m \times p$ .

Hint Resolve mult\_plus\_distr\_r: *arith v62*.

Lemma mult\_plus\_distr\_l :  $\forall n m p, n \times (m + p) = n \times m + n \times p$ .

Lemma mult\_minus\_distr\_r :  $\forall n m p, (n - m) \times p = n \times p - m \times p$ .

Hint Resolve mult\_minus\_distr\_r: *arith v62*.

Lemma mult\_minus\_distr\_l :  $\forall n m p, n \times (m - p) = n \times m - n \times p$ .

Hint Resolve mult\_minus\_distr\_l: *arith v62*.

### 55.1.5 Associativity

Lemma mult\_assoc\_reverse :  $\forall n m p, n \times m \times p = n \times (m \times p)$ .

Hint Resolve mult\_assoc\_reverse: *arith v62*.

Lemma mult\_assoc :  $\forall n m p, n \times (m \times p) = n \times m \times p$ .

Hint Resolve mult\_assoc: *arith v62*.

### 55.1.6 Inversion lemmas

Lemma mult\_is\_O :  $\forall n m, n \times m = 0 \rightarrow n = 0 \vee m = 0$ .

Lemma mult\_is\_one :  $\forall n m, n \times m = 1 \rightarrow n = 1 \wedge m = 1$ .

### 55.1.7 Multiplication and successor

Lemma mult\_succ\_l :  $\forall n m : \text{nat}, S n \times m = n \times m + m$ .

Lemma mult\_succ\_r :  $\forall n m : \text{nat}, n \times S m = n \times m + n$ .

## 55.2 Compatibility with orders

Lemma mult\_O\_le :  $\forall n m, m = 0 \vee n \leq m \times n$ .

Hint Resolve mult\_O\_le: *arith v62*.

Lemma mult\_le\_compat\_l :  $\forall n m p, n \leq m \rightarrow p \times n \leq p \times m$ .

Hint Resolve mult\_le\_compat\_l: *arith*.

Lemma mult\_le\_compat\_r :  $\forall n m p, n \leq m \rightarrow n \times p \leq m \times p$ .

Lemma mult\_le\_compat :

$\forall n m p (q : \text{nat}), n \leq m \rightarrow p \leq q \rightarrow n \times p \leq m \times q$ .

Lemma mult\_S\_lt\_compat\_l :  $\forall n m p, m < p \rightarrow S n \times m < S n \times p$ .

Hint Resolve mult\_S\_lt\_compat\_l: *arith*.

Lemma mult\_lt\_compat\_l :  $\forall n m p, n < m \rightarrow 0 < p \rightarrow p \times n < p \times m$ .

Lemma mult\_lt\_compat\_r :  $\forall n m p, n < m \rightarrow 0 < p \rightarrow n \times p < m \times p$ .

Lemma mult\_S\_le\_reg\_l :  $\forall n m p, S n \times m \leq S n \times p \rightarrow m \leq p$ .

### 55.3 $n|->2*n$ and $n|->2n+1$ have disjoint image

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

### 55.4 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 56

# Library `Coq.Arith.Peano_dec`

```
Require Import Decidable.
Require Import Eqty_dec.
Require Import Le Lt.
Local Open Scope nat_scope.

Implicit Types m n x y : nat.

Theorem O_or_S :  $\forall n, \{m : \text{nat} \mid S\ m = n\} + \{0 = n\}$ .
Theorem eq_nat_dec :  $\forall n\ m, \{n = m\} + \{n \neq m\}$ .
Hint Resolve O_or_S eq_nat_dec: arith.
Theorem dec_eq_nat :  $\forall n\ m, \text{decidable } (n = m)$ .
Definition UIP_nat := Eqty_dec.UIP_dec eq_nat_dec.
Lemma le_unique:  $\forall m\ n\ (h1\ h2: m \leq n), h1 = h2$ .
```

## Chapter 57

# Library **Coq.Arith.Plus**

Properties of addition. *add* is defined in *Init/Peano.v* as:

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | 0 => m
  | S p => S (p + m)
  end
where "n + m" := (plus n m) : nat_scope.
```

```
Require Import Le.
Require Import Lt.
Local Open Scope nat_scope.
Implicit Types m n p q : nat.
```

### 57.1 Zero is neutral

Deprecated : Already in *Init/Peano.v* **Notation** `plus_0_l` := `plus_O_n` (*only parsing*).

**Definition** `plus_0_r` `n` := `eq_sym` (`plus_n_O n`).

### 57.2 Commutativity

**Lemma** `plus_comm` :  $\forall n\ m, n + m = m + n$ .

**Hint Immediate** `plus_comm`: *arith v62*.

### 57.3 Associativity

**Definition** `plus_Snm_nSm` :  $\forall n\ m, S\ n + m = n + S\ m$  := `plus_n_Sm`.

**Lemma** `plus_assoc` :  $\forall n\ m\ p, n + (m + p) = n + m + p$ .

**Hint Resolve** `plus_assoc`: *arith v62*.

Lemma plus\_permute :  $\forall n m p, n + (m + p) = m + (n + p)$ .  
 Lemma plus\_assoc\_reverse :  $\forall n m p, n + m + p = n + (m + p)$ .  
 Hint Resolve plus\_assoc\_reverse: arith v62.

## 57.4 Simplification

Lemma plus\_reg\_l :  $\forall n m p, p + n = p + m \rightarrow n = m$ .  
 Lemma plus\_le\_reg\_l :  $\forall n m p, p + n \leq p + m \rightarrow n \leq m$ .  
 Lemma plus\_lt\_reg\_l :  $\forall n m p, p + n < p + m \rightarrow n < m$ .

## 57.5 Compatibility with order

Lemma plus\_le\_compat\_l :  $\forall n m p, n \leq m \rightarrow p + n \leq p + m$ .  
 Hint Resolve plus\_le\_compat\_l: arith v62.  
 Lemma plus\_le\_compat\_r :  $\forall n m p, n \leq m \rightarrow n + p \leq m + p$ .  
 Hint Resolve plus\_le\_compat\_r: arith v62.  
 Lemma le\_plus\_l :  $\forall n m, n \leq n + m$ .  
 Hint Resolve le\_plus\_l: arith v62.  
 Lemma le\_plus\_r :  $\forall n m, m \leq n + m$ .  
 Hint Resolve le\_plus\_r: arith v62.  
 Theorem le\_plus\_trans :  $\forall n m p, n \leq m \rightarrow n \leq m + p$ .  
 Hint Resolve le\_plus\_trans: arith v62.  
 Theorem lt\_plus\_trans :  $\forall n m p, n < m \rightarrow n < m + p$ .  
 Hint Immediate lt\_plus\_trans: arith v62.  
 Lemma plus\_lt\_compat\_l :  $\forall n m p, n < m \rightarrow p + n < p + m$ .  
 Hint Resolve plus\_lt\_compat\_l: arith v62.  
 Lemma plus\_lt\_compat\_r :  $\forall n m p, n < m \rightarrow n + p < m + p$ .  
 Hint Resolve plus\_lt\_compat\_r: arith v62.  
 Lemma plus\_le\_compat :  $\forall n m p q, n \leq m \rightarrow p \leq q \rightarrow n + p \leq m + q$ .  
 Lemma plus\_le\_lt\_compat :  $\forall n m p q, n \leq m \rightarrow p < q \rightarrow n + p < m + q$ .  
 Lemma plus\_lt\_le\_compat :  $\forall n m p q, n < m \rightarrow p \leq q \rightarrow n + p < m + q$ .  
 Lemma plus\_lt\_compat :  $\forall n m p q, n < m \rightarrow p < q \rightarrow n + p < m + q$ .

## 57.6 Inversion lemmas

Lemma plus\_is\_O :  $\forall n m, n + m = 0 \rightarrow n = 0 \wedge m = 0$ .  
 Definition plus\_is\_one :  
 $\forall m n, m + n = 1 \rightarrow \{m = 0 \wedge n = 1\} + \{m = 1 \wedge n = 0\}$ .

## 57.7 Derived properties

Lemma `plus_permute_2_in_4` :  $\forall n\ m\ p\ q, n + m + (p + q) = n + p + (m + q)$ .

## 57.8 Tail-recursive plus

`tail_plus` is an alternative definition for `plus` which is tail-recursive, whereas `plus` is not. This can be useful when extracting programs.

```
Fixpoint tail_plus n m : nat :=  
  match n with  
  | O  $\Rightarrow$  m  
  | S n  $\Rightarrow$  tail_plus n (S m)  
  end.
```

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

## 57.9 Discrimination

Lemma `succ_plus_discr` :  $\forall n\ m, n \neq S\ (\text{plus } m\ n)$ .

Lemma `n_SS` :  $\forall n, n \neq S\ (S\ n)$ .

Lemma `n_SSS` :  $\forall n, n \neq S\ (S\ (S\ n))$ .

Lemma `n_SSSS` :  $\forall n, n \neq S\ (S\ (S\ (S\ n)))$ .



## Chapter 58

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

Well-founded relations and natural numbers

```
Require Import 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 :
```

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

**Theorem** `induction_ltof2` :

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

**Theorem** `induction_gtof2` :

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

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**  $R : A \rightarrow A \rightarrow \mathbf{Prop}.$

**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` :

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

**Lemma** `lt_wf_rec` :

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

**Lemma** `lt_wf_ind` :

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

**Lemma** `gt_wf_rec` :

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

**Lemma** `gt_wf_ind` :

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

**Lemma** `lt_wf_double_rec` :

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

**Lemma** `lt_wf_double_ind` :

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

**Hint** `Resolve` `lt_wf`: *arith*.

**Hint** `Resolve` `well_founded_lt_compat`: *arith*.

**Section** `LT_WF_REL.`

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

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

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

```

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 := @nat_iter (only parsing).
Notation iter_nat_plus := @nat_iter_plus (only parsing).
Notation iter_nat_invariant := @nat_iter_invariant (only parsing).

```