The Coq Proof Assistant
The standard library

April 12, 2015

Version 8.4pl6[1]

TypiCal Project (formerly LogiCal)

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

V8.4pl6, April 12, 2015

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 >-> Sortclass.

Additional rewriting lemmas about *eq_true*

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

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

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

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

```
Inductive BoolSpec (P Q : Prop) : bool → Prop :=
  | BoolSpecT : P → BoolSpec P Q true
  | BoolSpecF : Q → BoolSpec P Q false.
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 :
  ∀ (A B:Type) (p:A × B), p = pair (fst p) (snd p).
Lemma injective_projections :
```

```
  ∀ (A B:Type) (p1 p2:A × B),
    fst p1 = fst p2 → snd p1 = snd p2 → p1 = p2.
Definition prod_uncurry (A B C:Type) (f:prod A B → C)
  (x:A) (y:B) : C := f (pair x y).
Definition prod_curry (A B C:Type) (f:A → B → C)
  (p:prod A B) : C := match p with
                        | pair x y ⇒ f x y
                        end.
```

Polymorphic lists and some operations

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
Infix "::" := cons (at level 60, right associativity) : list_scope.
Delimit Scope list_scope with list.
Local Open Scope list_scope.
Definition length (A : Type) : list A → nat :=
  fix length l :=
  match l with
    | nil ⇒ O
    | _ :: l' ⇒ S (length l')
  end.
```

Concatenation of two lists

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

1.5 The comparison datatype

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

```
Lemma CompOpp_involutive : ∀ c, CompOpp (CompOpp c) = c.

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

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

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

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

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

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

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

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

```
Definition CompSpec {A} (eq lt : A→A→Prop)(x y:A) : comparison → Prop :=
 CompareSpec (eq x y) (lt x y) (lt y x).
Definition CompSpecT {A} (eq lt : A→A→Prop)(x y:A) : comparison → Type :=
 CompareSpecT (eq x y) (lt x y) (lt y x).
Hint Unfold CompSpec CompSpecT.

Lemma CompSpec2Type : ∀ A (eq lt:A→A→Prop) x y c,
 CompSpec eq lt x y c → 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 → Type :=
  identity_refl : identity a a.
Hint Resolve identity_refl: core.
```

Identity type

```
Definition ID := ∀ A:Type, A → A.
```

```
Definition id : ID := fun A x => 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 :=
  I : True.
```

False is the always false proposition

```
Inductive False : Prop :=.
```

not A, written $\neg A$, is the negation of A

```
Definition not (A:Prop) := A → False.

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

Hint Unfold not: core.
```

and A B, written $A \land B$, is the conjunction of A and B
conj p q is a proof of $A \land B$ 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 $A \lor B$, is the disjunction of A and B

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

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

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

```
Definition iff (A B:Prop) := (A → B) ∧ (B → A).
Notation "A <-> B" := (iff A B) : type_scope.
Section Equivalence.
Theorem iff_refl : ∀ A:Prop, A ↔ A.
Theorem iff_trans : ∀ A B C:Prop, (A ↔ B) → (B ↔ C) → (A ↔ C).
Theorem iff_sym : ∀ A B:Prop, (A ↔ B) → (B ↔ A).
End Equivalence.
Hint Unfold iff: extcore.
```

Some equivalences

```
Theorem neg_false : ∀ A : Prop, ¬ A ↔ (A ↔ False).
Theorem and_cancel_l : ∀ A B C : Prop,
  (B → A) → (C → A) → ((A ∧ B ↔ A ∧ C) ↔ (B ↔ C)).
Theorem and_cancel_r : ∀ A B C : Prop,
  (B → A) → (C → A) → ((B ∧ A ↔ C ∧ A) ↔ (B ↔ C)).
Theorem and_comm : ∀ A B : Prop, A ∧ B ↔ B ∧ A.
Theorem and_assoc : ∀ A B C : Prop, (A ∧ B) ∧ C ↔ A ∧ B ∧ C.
Theorem or_cancel_l : ∀ A B C : Prop,
  (B → ¬ A) → (C → ¬ A) → ((A ∨ B ↔ A ∨ C) ↔ (B ↔ C)).
Theorem or_cancel_r : ∀ A B C : Prop,
  (B → ¬ A) → (C → ¬ A) → ((B ∨ A ↔ C ∨ A) ↔ (B ↔ C)).
Theorem or_comm : ∀ A B : Prop, (A ∨ B) ↔ (B ∨ A).
Theorem or_assoc : ∀ A B C : Prop, (A ∨ B) ∨ C ↔ A ∨ B ∨ C.
```

Backward direction of the equivalences above does not need assumptions

```
Theorem and_iff_compat_l : ∀ A B C : Prop,
  (B ↔ C) → (A ∧ B ↔ A ∧ C).
Theorem and_iff_compat_r : ∀ A B C : Prop,
  (B ↔ C) → (B ∧ A ↔ C ∧ A).
Theorem or_iff_compat_l : ∀ A B C : Prop,
  (B ↔ C) → (A ∨ B ↔ A ∨ C).
Theorem or_iff_compat_r : ∀ A B C : Prop,
  (B ↔ C) → (B ∨ A ↔ C ∨ A).
Lemma iff_and : ∀ A B : Prop, (A ↔ B) → (A → B) ∧ (B → A).
Lemma iff_to_and : ∀ A B : Prop, (A ↔ B) ↔ (A → B) ∧ (B → A).
```

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

```
Definition IF_then_else (P Q R:Prop) := P ∧ Q ∨ ¬ P ∧ 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.

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

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

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

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

Notation "'exists2' x , p & q" := (ex2 (fun x ⇒ p) (fun x ⇒ q))
  (at level 200, x ident, p at level 200, right associativity) : type_scope.
Notation "'exists2' x : t , p & q" := (ex2 (fun x:t ⇒ p) (fun x:t ⇒ q))
  (at level 200, x ident, t at level 200, p at level 200, right associativity,
    format "'[' 'exists2' '/ ' x : t , '/ ' '[' p & '/' q ']' ']'")
  : type_scope.
```

Derived rules for universal quantification

```
Section universal_quantification.

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

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

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

End universal_quantification.
```

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 I 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 : ∀ (A:Type) (P:A→Prop),
```

((∃ x, P x) ∧ uniqueness P) ↔ (∃! x, P x).

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

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

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 → inhabited A.

Hint Resolve inhabits: *core*.

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

Declaration of stepl and stepr for eq and iff

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

Declare Left Step eq_stepl.
Declare Right Step eq_trans.

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

Declare Left Step iff_stepl.
Declare Right Step iff_trans.

Chapter 4

Library Coq.Init.Notations

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

Notations for propositional connectives

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

Notations for equality and inequalities

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

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

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

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

Arithmetical notations (also used for type constructors)

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

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

Notations for booleans

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

Notations for pairs

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

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

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

Notations for sigma-types or subsets

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

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

Reserved Notation "{ x : 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 *nat* × *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 : ∀ n:nat, n = pred (S n).
```

Injectivity of successor

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

Theorem not_eq_S : ∀ n m:nat, n ≠ m → S n ≠ S m.
Hint Resolve not_eq_S: core.

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

Zero is not the successor of a number

```
Theorem O_S : ∀ n:nat, 0 ≠ S n.
Hint Resolve O_S: core.

Theorem n_Sn : ∀ n:nat, n ≠ S n.
Hint Resolve n_Sn: core.
```

Addition

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m)
  end

where "n + m" := (plus n m) : nat_scope.

Hint Resolve (f_equal2 plus): v62.
Hint Resolve (f_equal2 (A1:=nat) (A2:=nat)): core.

Lemma plus_n_O : ∀ n:nat, n = n + 0.
Hint Resolve plus_n_O: core.

Lemma plus_O_n : ∀ n:nat, 0 + n = n.

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

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

Standard associated names

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

Multiplication

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O ⇒ 0
  | S p ⇒ m + p × m
```

```
  end

where "n * m" := (mult n m) : nat_scope.

Hint Resolve (f_equal2 mult): core.

Lemma mult_n_O : ∀ n:nat, 0 = n × 0.
Hint Resolve mult_n_O: core.

Lemma mult_n_Sm : ∀ n m:nat, n × m + n = n × 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*≥*m*

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O, _ ⇒ n
  | S k, O ⇒ n
  | S k, S l ⇒ 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 → Prop :=
  | le_n : n ≤ n
  | le_S : ∀ m:nat, n ≤ m → n ≤ S m

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

Hint Constructors le: core.

Definition lt (n m:nat) := S n ≤ m.
Hint Unfold lt: core.

Infix "<" := lt : nat_scope.

Definition ge (n m:nat) := m ≤ n.
Hint Unfold ge: core.

Infix "≥" := ge : nat_scope.

Definition gt (n m:nat) := m < n.
Hint Unfold gt: core.

Infix ">" := gt : nat_scope.

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

```
Theorem le_pred : ∀ n m, n ≤ m → pred n ≤ pred m.
Theorem le_S_n : ∀ n m, S n ≤ S m → n ≤ m.
```

Case analysis

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

Principle of double induction

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

Maximum and minimum : definitions and specifications

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

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

Theorem max_l : ∀ n m : nat, m ≤ n → max n m = n.
Theorem max_r : ∀ n m : nat, n ≤ m → max n m = m.
Theorem min_l : ∀ n m : nat, n ≤ m → min n m = n.
Theorem min_r : ∀ n m : nat, m ≤ n → min n m = m.
```

nth iteration of the function f

```
Fixpoint nat_iter (n:nat) {A} (f:A→A) (x:A) : A :=
  match n with
    | O ⇒ x
    | S n' ⇒ f (nat_iter n' f x)
  end.

Lemma nat_iter_succ_r n {A} (f:A→A) (x:A) :
  nat_iter (S n) f x = nat_iter n f (f x).

Theorem nat_iter_plus :
  ∀ (n m:nat) {A} (f:A → A) (x:A),
    nat_iter (n + m) f x = nat_iter n f (nat_iter m f x).
```

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

```
Theorem nat_iter_invariant :
  ∀ (n:nat) {A} (f:A → A) (P : A → Prop),
    (∀ x, P x → P (f x)) →
    ∀ x, P x → P (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

(*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 (*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 sig (A:Type) (P:A → Prop) : Type :=
    exist : ∀ x:A, P x → sig P.

Inductive sig2 (A:Type) (P Q:A → Prop) : Type :=
    exist2 : ∀ x:A, P x → Q x → sig2 P Q.
```

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

```
Inductive sigT (A:Type) (P:A → Type) : Type :=
    existT : ∀ x:A, P x → sigT P.

Inductive sigT2 (A:Type) (P Q:A → Type) : Type :=
    existT2 : ∀ x:A, P x → Q x → sigT2 P Q.

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

Add Printing Let sig.
```

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

Projections of *sig*

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

```
Section Subset_projections.

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

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

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

End Subset_projections.
```

Projections of *sigT*

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

```
Section Projections.

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

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

End Projections.
```

sigT of a predicate is equivalent to *sig*

```
Lemma sig_of_sigT : ∀ (A:Type) (P:A→Prop), sigT P → sig P.

Lemma sigT_of_sig : ∀ (A:Type) (P:A→Prop), sig P → sigT P.

Coercion sigT_of_sig : sig >-> sigT.
Coercion sig_of_sigT : sigT >-> 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 O = x0 ∧ ∀ n, R (f n) (f (S n))}.
End Dependent_choice_lemmas.
```

A result of type (*Exc A*) is either a normal value of type *A* or an *error* :
`Inductive Exc [A:Type] : Type := value : A->(Exc A) | error : (Exc A).`
It is implemented using the option type.

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

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

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

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

Ltac 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 ⊢ _ ⇒ destruct x; destruct_all t
  | _ ⇒ idtac
 end.
```

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

Tactics for applying equivalences.

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

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

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

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

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

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

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

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

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

```
    end
  with do_intro := let H := fresh in intro H; use_hyp H
  with destruct_hyp H := case H; clear H; do_intro; do_intro in
  let rec use_hyps :=
    match goal with
    | H : _ ∧ _ ⊢ _ ⇒ exact H || (destruct_hyp H; use_hyps)
    | H : _ ⊢ _ ⇒ solve [inversion H]
    | _ ⇒ idtac
    end in
  let 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 :
  ∀ P:A → Set,
    (∀ x:A, (∀ y:A, R y x → P y) → P x) → ∀ a:A, P a.

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

Well-founded fixpoints

```
Section FixPoint.

  Variable P : A → Type.
  Variable F : ∀ x:A, (∀ y:A, R y x → P y) → P x.

  Fixpoint Fix_F (x:A) (a:Acc x) : P x :=
    F (fun (y:A) (h:R y x) ⇒ Fix_F (Acc_inv a h)).

  Scheme Acc_inv_dep := Induction for Acc Sort Prop.

  Lemma Fix_F_eq :
   ∀ (x:A) (r:Acc x),
     F (fun (y:A) (p:R y x) ⇒ Fix_F (x:=y) (Acc_inv r p)) = Fix_F (x:=x) r.

  Definition Fix (x:A) := Fix_F (Rwf x).
```

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

```
  Hypothesis
    F_ext :
      ∀ (x:A) (f g:∀ y:A, R y x → P y),
        (∀ (y:A) (p:R y x), f y p = g y p) → F f = F g.

  Lemma Fix_F_inv : ∀ (x:A) (r s:Acc x), Fix_F r = Fix_F s.

  Lemma Fix_eq : ∀ x:A, Fix x = F (fun (y:A) (p:R y x) ⇒ Fix y).

 End FixPoint.

End Well_founded.
```

Well-founded fixpoints over pairs

```
Section Well_founded_2.

  Variables A B : Type.
  Variable R : A × B → A × B → Prop.

  Variable P : A → B → Type.

  Section FixPoint_2.

  Variable
    F :
      ∀ (x:A) (x':B),
        (∀ (y:A) (y':B), R (y, y') (x, x') → P y y') → P x x'.

  Fixpoint Fix_F_2 (x:A) (x':B) (a:Acc R (x, x')) : P x x' :=
    F
```

```
      (fun (y:A) (y':B) (h:R (y, y') (x, x')) ⇒
          Fix_F_2 (x:=y) (x':=y') (Acc_inv a (y,y') h)).

  End FixPoint_2.

  Hypothesis Rwf : well_founded R.

  Theorem well_founded_induction_type_2 :
   (∀ (x:A) (x':B),
      (∀ (y:A) (y':B), R (y, y') (x, x') → P y y') → P x x') →
   ∀ (a:A) (b:B), P a b.

End Well_founded_2.

Notation Acc_iter := Fix_F (only parsing). Notation Acc_iter_2 := Fix_F_2 (only parsing).
```

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 : ∀ P:Prop, P ∨ ¬ P.`

Conditional on any proposition. `Definition IFProp (P B:Prop) (e1 e2:P) :=`

```
  match EM B with
  | or_introl _ ⇒ e1
  | or_intror _ ⇒ e2
  end.
```

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

```
Lemma AC_IF :
 ∀ (P B:Prop) (e1 e2:P) (Q:P → Prop),
   (B → Q e1) → (¬ B → Q e2) → Q (IFProp B e1 e2).
```

We assume a type with two elements. They play the role of booleans. The main theorem under the current assumptions is that $T=F$ `Variable Bool : Prop.`

```
Variable T : Bool.
Variable F : Bool.
```

The powerset operator `Definition pow (P:Prop) := P → Bool.`

A piece of theory about retracts `Section Retracts.`

```
Variables A B : Prop.

Record retract : Prop :=
  {i : A → B; j : B → A; inv : ∀ a:A, j (i a) = a}.

Record retract_cond : Prop :=
  {i2 : A → B; j2 : B → A; inv2 : retract → ∀ a:A, j2 (i2 a) = a}.
```

The dependent elimination above implies the axiom of choice: `Lemma AC : ∀ r:retract_cond, retract → ∀ a:A, j2 r (i2 r a) = a.`

```
End Retracts.
```

This lemma is basically a commutation of implication and existential quantification: (EX x | A -> P(x)) <=> (A -> EX x | P(x)) which is provable in classical logic (=> is already provable in intuitionnistic logic).

```
Lemma L1 : ∀ A B:Prop, retract_cond (pow A) (pow B).
```

The paradoxical set `Definition U := ∀ P:Prop, pow P.`

Bijection between U and $(pow\ U)$ `Definition f (u:U) : pow U := u U.`

```
Definition g (h:pow U) : U :=
  fun X ⇒ let lX := j2 (L1 X U) in let rU := i2 (L1 U U) in lX (rU h).
```

We deduce that the powerset of U is a retract of U. This lemma is stated in Berardi's article, but is not used afterwards. `Lemma retract_pow_U_U : retract (pow U) U.`

Encoding of Russel's paradox

The boolean negation. `Definition Not_b (b:Bool) := IFProp (b = T) F T.`

the set of elements not belonging to itself `Definition R : U := g (fun u:U ⇒ Not_b (u U u)).`

```
Lemma not_has_fixpoint : R R = Not_b (R R).

Theorem classical_proof_irrelevence : 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_2 = 2nd-order impredicative minimal predicate logic (with ex. quant.)
- IPL^2 = 2nd-order functional minimal predicate logic (with ex. quant.)
- IPL_2^2 = 2nd-order impredicative, 2nd-order functional minimal pred. logic (with ex. quant.)

with no prerequisite on the non-emptyness of domains

Table of contents

1. Definitions

2. IPL_2^2 |- AC_rel + AC! = AC_fun

3.1. typed IPL_2 + Sigma-types + PI |- AC_rel = GAC_rel and IPL_2 |- AC_rel + IGP -> GAC_rel and IPL_2 |- GAC_rel = OAC_rel

3.2. IPL^2 |- AC_fun + IGP = GAC_fun = OAC_fun = AC_fun + Drinker

3.3. D_iota -> ID_iota and D_epsilon <-> ID_epsilon + Drinker

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

5. 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 Constructive choice and description

AC_rel

```
Definition RelationalChoice_on :=
  ∀ R:A→B→Prop,
    (∀ x : A, ∃ y : B, R x y) →
    (∃ R' : A→B→Prop, subrelation R' R ∧ ∀ x, ∃! y, R' x y).
```

AC_fun

```
Definition FunctionalChoice_on :=
  ∀ R:A→B→Prop,
    (∀ x : A, ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x : A, R x (f x)).
```

DC_fun

```
Definition FunctionalDependentChoice_on :=
  ∀ (R:A→A→Prop),
    (∀ x, ∃ y, R x y) → ∀ x0,
    (∃ f : nat → A, f 0 = x0 ∧ ∀ n, R (f n) (f (S n))).
```

ACw_fun

```
Definition FunctionalCountableChoice_on :=
  ∀ (R:nat→A→Prop),
    (∀ n, ∃ y, R n y) →
    (∃ f : nat → A, ∀ n, R n (f n)).
```

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

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

ID_epsilon (constructive version of indefinite description; combined with proof-irrelevance, it may be connected to Carlström's type theory with a constructive indefinite description operator)

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

ID_iota (constructive version of definite description; combined with proof-irrelevance, it may be connected to Carlström's and Stenlund's type theory with a constructive definite description operator)

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

11.1.2 Weakly classical choice and description

GAC_rel

```
Definition GuardedRelationalChoice_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    (∀ x : A, P x → ∃ y : B, R x y) →
    (∃ R' : A→B→Prop,
      subrelation R' R ∧ ∀ x, P x → ∃! y, R' x y).
```

GAC_fun

```
Definition GuardedFunctionalChoice_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    inhabited B →
    (∀ x : A, P x → ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x, P x → R x (f x)).
```

GFR_fun

```
Definition GuardedFunctionalRelReification_on :=
  ∀ P : A→Prop, ∀ R : A→B→Prop,
    inhabited B →
    (∀ x : A, P x → ∃! y : B, R x y) →
    (∃ f : A→B, ∀ x : A, P x → R x (f x)).
```

OAC_rel

```
Definition OmniscientRelationalChoice_on :=
  ∀ R : A→B→Prop,
    ∃ R' : A→B→Prop,
      subrelation R' R ∧ ∀ x : A, (∃ y : B, R x y) → ∃! y, R' x y.
```

OAC_fun

```
Definition OmniscientFunctionalChoice_on :=
  ∀ R : A→B→Prop,
    inhabited B →
    ∃ f : A→B, ∀ x : A, (∃ y : B, R x y) → R x (f x).
```

D_epsilon

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

D_iota

```
Definition IotaStatement_on :=
  ∀ P:A→Prop,
    inhabited A → { x:A | (∃! x, P x) → P x }.
End ChoiceSchemes.
```

Generalized schemes

```
Notation RelationalChoice :=
```

```
  (∀ A B, RelationalChoice_on A B).
Notation FunctionalChoice :=
  (∀ A B, FunctionalChoice_on A B).
Definition FunctionalDependentChoice :=
  (∀ A, FunctionalDependentChoice_on A).
Definition FunctionalCountableChoice :=
  (∀ A, FunctionalCountableChoice_on A).
Notation FunctionalChoiceOnInhabitedSet :=
  (∀ A B, inhabited B → FunctionalChoice_on A B).
Notation FunctionalRelReification :=
  (∀ A B, FunctionalRelReification_on A B).

Notation GuardedRelationalChoice :=
  (∀ A B, GuardedRelationalChoice_on A B).
Notation GuardedFunctionalChoice :=
  (∀ A B, GuardedFunctionalChoice_on A B).
Notation GuardedFunctionalRelReification :=
  (∀ A B, GuardedFunctionalRelReification_on A B).

Notation OmniscientRelationalChoice :=
  (∀ A B, OmniscientRelationalChoice_on A B).
Notation OmniscientFunctionalChoice :=
  (∀ A B, OmniscientFunctionalChoice_on A B).

Notation ConstructiveDefiniteDescription :=
  (∀ A, ConstructiveDefiniteDescription_on A).
Notation ConstructiveIndefiniteDescription :=
  (∀ A, ConstructiveIndefiniteDescription_on A).

Notation IotaStatement :=
  (∀ A, IotaStatement_on A).
Notation EpsilonStatement :=
  (∀ A, EpsilonStatement_on A).
```

Subclassical schemes

```
Definition ProofIrrelevance :=
  ∀ (A:Prop) (a1 a2:A), a1 = a2.

Definition IndependenceOfGeneralPremises :=
  ∀ (A:Type) (P:A → Prop) (Q:Prop),
    inhabited A →
    (Q → ∃ x, P x) → ∃ x, Q → P x.

Definition SmallDrinker'sParadox :=
  ∀ (A:Type) (P:A → Prop), inhabited A →
    ∃ x, (∃ x, P x) → 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_funct_choice :
  ∀ A B : Type,
    FunctionalRelReification_on A B → RelationalChoice_on A B → FunctionalChoice_on A B.
Lemma funct_choice_imp_rel_choice :
  ∀ A B, FunctionalChoice_on A B → RelationalChoice_on A B.
Lemma funct_choice_imp_description :
  ∀ A B, FunctionalChoice_on A B → FunctionalRelReification_on A B.
Corollary FunChoice_Equiv_RelChoice_and_ParamDefinDescr :
  ∀ A B, FunctionalChoice_on A B ↔
    RelationalChoice_on A B ∧ 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 :
  RelationalChoice → ProofIrrelevance → GuardedRelationalChoice.
Lemma rel_choice_indep_of_general_premises_imp_guarded_rel_choice :
  ∀ A B, inhabited B → RelationalChoice_on A B →
    IndependenceOfGeneralPremises → GuardedRelationalChoice_on A B.
Lemma guarded_rel_choice_imp_rel_choice :
  ∀ A B, GuardedRelationalChoice_on A B → RelationalChoice_on A B.
Lemma subset_types_imp_guarded_rel_choice_iff_rel_choice :
  ProofIrrelevance → (GuardedRelationalChoice ↔ RelationalChoice).
```

OAC_rel = GAC_rel

```
Corollary guarded_iff_omniscient_rel_choice :
  GuardedRelationalChoice ↔ OmniscientRelationalChoice.
```

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 → IndependenceOfGeneralPremises.

Lemma guarded_fun_choice_imp_fun_choice :
GuardedFunctionalChoice → FunctionalChoiceOnInhabitedSet.

Lemma fun_choice_and_indep_general_prem_imp_guarded_fun_choice :
FunctionalChoiceOnInhabitedSet → IndependenceOfGeneralPremises
→ GuardedFunctionalChoice.

Corollary fun_choice_and_indep_general_prem_iff_guarded_fun_choice :
FunctionalChoiceOnInhabitedSet ∧ IndependenceOfGeneralPremises
↔ GuardedFunctionalChoice.

AC_fun + Drinker = OAC_fun
This was already observed by Bell [*Bell*]

Lemma omniscient_fun_choice_imp_small_drinker :
OmniscientFunctionalChoice → SmallDrinker'sParadox.

Lemma omniscient_fun_choice_imp_fun_choice :
OmniscientFunctionalChoice → FunctionalChoiceOnInhabitedSet.

Lemma fun_choice_and_small_drinker_imp_omniscient_fun_choice :
FunctionalChoiceOnInhabitedSet → SmallDrinker'sParadox
→ OmniscientFunctionalChoice.

Corollary fun_choice_and_small_drinker_iff_omniscient_fun_choice :
FunctionalChoiceOnInhabitedSet ∧ SmallDrinker'sParadox
↔ OmniscientFunctionalChoice.

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

Theorem guarded_iff_omniscient_fun_choice :
GuardedFunctionalChoice ↔ OmniscientFunctionalChoice.

11.3.3 D_iota -> ID_iota and D_epsilon <-> ID_epsilon + Drinker

D_iota -> ID_iota

Lemma iota_imp_constructive_definite_description :
IotaStatement → ConstructiveDefiniteDescription.

ID_epsilon + Drinker <-> D_epsilon

Lemma epsilon_imp_constructive_indefinite_description:
EpsilonStatement → ConstructiveIndefiniteDescription.

Lemma constructive_indefinite_description_and_small_drinker_imp_epsilon :
SmallDrinker'sParadox → ConstructiveIndefiniteDescription →
EpsilonStatement.

```
Lemma epsilon_imp_small_drinker :
  EpsilonStatement → SmallDrinker'sParadox.
```

```
Theorem constructive_indefinite_description_and_small_drinker_iff_epsilon :
  (SmallDrinker'sParadox × ConstructiveIndefiniteDescription →
  EpsilonStatement) ×
  (EpsilonStatement →
   SmallDrinker'sParadox × ConstructiveIndefiniteDescription).
```

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

```
Lemma classical_denumerable_description_imp_fun_choice :
  ∀ A:Type,
    FunctionalRelReification_on A nat →
    ∀ R:A→nat→Prop,
      (∀ x y, decidable (R x y)) → FunctionalChoice_on_rel R.
```

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

```
Notation DependentFunctionalChoice :=
  (∀ A (B:A→Type), DependentFunctionalChoice_on B).
```

The easy part

```
Theorem dep_non_dep_functional_choice :
  DependentFunctionalChoice → FunctionalChoice.
```

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

```
Scheme and_indd := Induction for and Sort Prop.
Scheme eq_indd := Induction for eq Sort Prop.
```

```
Definition proj1_inf (A B:Prop) (p : A∧B) :=
  let (a,b) := p in a.
```

```
Theorem non_dep_dep_functional_choice :
  FunctionalChoice → DependentFunctionalChoice.
```

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

```
Lemma constructive_definite_descr_fun_reification :
  ConstructiveDefiniteDescription → FunctionalRelReification.
```

Remark, the following corollaries morally hold:
Definition In_propositional_context (A:Type) := forall C:Prop, (A -> C) -> C.
Corollary constructive_definite_descr_in_prop_context_iff_fun_reification : In_propositional_context ConstructiveIndefiniteDescription <-> FunctionalChoice.
Corollary constructive_definite_descr_in_prop_context_iff_fun_reification : In_propositional_context ConstructiveDefiniteDescription <-> FunctionalRelReification.
but expecting *FunctionalChoice* (resp. *FunctionalRelReification*) to be applied on the same Type universes on both sides of the first (resp. second) equivalence breaks the stratification of universes.

11.7 Excluded-middle + definite description => computational excluded-middle

The idea for the following proof comes from [*ChicliPottierSimpson02*]

Classical logic and axiom of unique choice (i.e. functional relation reification), as shown in [*ChicliPottierSimpson02*], implies the double-negation of excluded-middle in Set (which is incompatible with the impredicativity of Set).

We adapt the proof to show that constructive definite description transports excluded-middle from Prop to Set.

[*ChicliPottierSimpson02*] Laurent Chicli, Loïc Pottier, Carlos Simpson, Mathematical Quotients and Quotient Types in Coq, Proceedings of TYPES 2002, Lecture Notes in Computer Science 2646, Springer Verlag.

```
Require Import Setoid.
```

```
Theorem constructive_definite_descr_excluded_middle :
  ConstructiveDefiniteDescription →
  (∀ P:Prop, P ∨ ¬ P) → (∀ P:Prop, {P} + {¬ P}).
```

```
Corollary fun_reification_descr_computational_excluded_middle_in_prop_context :
  FunctionalRelReification →
  (∀ P:Prop, P ∨ ¬ P) →
  ∀ C:Prop, ((∀ P:Prop, {P} + {¬ P}) → C) → C.
```

11.8 Choice => Dependent choice => Countable choice

```
Require Import Arith.
```

```
Theorem functional_choice_imp_functional_dependent_choice :
   FunctionalChoice → FunctionalDependentChoice.
```

```
Theorem functional_dependent_choice_imp_functional_countable_choice :
  FunctionalDependentChoice → FunctionalCountableChoice.
```

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 : ∀ P:Prop, {P} + {¬ P}.

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

Church's iota operator

```
Definition iota (A : Type) (i:inhabited A) (P : A→Prop) : A
  := proj1_sig (classical_definite_description P i).

Definition iota_spec (A : Type) (i:inhabited A) (P : A→Prop) :
  (∃! x:A, P x) → P (iota i P)
  := proj2_sig (classical_definite_description P i).
```

Axiom of unique "choice" (functional reification of functional relations) Theorem dependent_unique_choice :

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

Theorem unique_choice :
  ∀ (A B:Type) (R:A → B → Prop),
    (∀ x:A, ∃! y : B, R x y) →
    (∃ f : A → B, ∀ x:A, R x (f x)).
```

Compatibility lemmas

```
Unset Implicit Arguments.

Definition dependent_description := dependent_unique_choice.
Definition description := unique_choice.
```

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

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

Theorem excluded_middle_informative : ∀ P:Prop, {P} + {¬ P}.

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

Hilbert's epsilon operator

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

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

Open question: is classical_indefinite_description constructively provable from *relational_choice* and *constructive_definite_description* (at least, using the fact that *functional_choice* is provable from *relational_choice* and *unique_choice*, we know that the double negation of *classical_indefinite_description* is provable (see *relative_non_contradiction_of_indefinite_desc*).

A proof that if P is inhabited, *epsilon a P* does not depend on the actual proof that the domain of P is inhabited (proof idea kindly provided by Pierre Castéran)

```
Lemma epsilon_inh_irrelevance :
    ∀ (A:Type) (i j : inhabited A) (P:A→Prop),
    (∃ x, P x) → epsilon i P = epsilon j P.

Opaque epsilon.
```

Weaker lemmas (compatibility lemmas)

```
Theorem choice :
  ∀ (A B : Type) (R : A→B→Prop),
    (∀ x : A, ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x : A, R x (f x)).
```

Chapter 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*=*True* $\lor$ *A*=*False*) $\leftrightarrow$ ($\forall$ *A*, *A*\/~*A*) $\land$ ($\forall$ *A* *B*, (*A*$\leftrightarrow$*B*) $\rightarrow$ *A*=*B*)

prop_degeneracy (also referred to as propositional completeness) asserts (up to consistency) that there are only two distinct formulas `Definition` prop_degeneracy := ∀ *A*:`Prop`, *A* = True ∨ *A* = False.

prop_extensionality asserts that equivalent formulas are equal `Definition` prop_extensionality := ∀ *A* *B*:`Prop`, (*A* ↔ *B*) → *A* = *B*.

excluded_middle asserts that we can reason by case on the truth or falsity of any formula `Definition` excluded_middle := ∀ *A*:`Prop`, *A* ∨ ¬ *A*.

We show *prop_degeneracy* ↔ (*prop_extensionality* ∧ *excluded_middle*)

`Lemma` prop_degen_ext : prop_degeneracy → prop_extensionality.

`Lemma` prop_degen_em : prop_degeneracy → excluded_middle.

`Lemma` prop_ext_em_degen :
  prop_extensionality → excluded_middle → prop_degeneracy.

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

```
Definition provable_prop_extensionality := ∀ A:Prop, A → A = True.

Lemma provable_prop_ext :
  prop_extensionality → 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 $A \to A$ for inhabited A, which implies the existence of a (trivial) retract from $A \to A$ 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 → ∀ A:Prop, inhabited A → (A → A) = A.

Record retract (A B:Prop) : Prop :=
  {f1 : A → B; f2 : B → A; f1_o_f2 : ∀ x:B, f1 (f2 x) = x}.

Lemma prop_ext_retract_A_A_imp_A :
  prop_extensionality → ∀ A:Prop, inhabited A → retract A (A → A).

Record has_fixpoint (A:Prop) : Prop :=
  {F : (A → A) → A; Fix : ∀ f:A → A, F f = f (F f)}.

Lemma ext_prop_fixpoint :
  prop_extensionality → ∀ A:Prop, inhabited A → has_fixpoint A.
```

Remark: *prop_extensionality* can be replaced in lemma *ext_prop_fixpoint* by the weakest property *provable_prop_extensionality*.

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 := ∀ (A:Prop) (a1 a2:A), a1 = a2.`

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

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

```
Section Proof_irrelevance_gen.

  Variable bool : Prop.
  Variable true : bool.
  Variable false : bool.
  Hypothesis bool_elim : ∀ C:Prop, C → C → bool → C.
  Hypothesis
    bool_elim_redl : ∀ (C:Prop) (c1 c2:C), c1 = bool_elim C c1 c2 true.
  Hypothesis
    bool_elim_redr : ∀ (C:Prop) (c1 c2:C), c2 = bool_elim C c1 c2 false.
```

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

  Lemma aux : prop_extensionality → bool_dep_induction → true = false.

  Lemma ext_prop_dep_proof_irrel_gen :
    prop_extensionality → bool_dep_induction → proof_irrelevance.

End Proof_irrelevance_gen.
```

In the pure Calculus of Constructions, we can define the boolean proposition bool = (C:Prop)C->C->C but we cannot prove that it has at most 2 elements.

```
Section Proof_irrelevance_Prop_Ext_CC.

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

  Definition BoolP_dep_induction :=
    ∀ P:BoolP → Prop, P TrueP → P FalseP → ∀ b:BoolP, P b.

  Lemma ext_prop_dep_proof_irrel_cc :
    prop_extensionality → BoolP_dep_induction → proof_irrelevance.

End Proof_irrelevance_Prop_Ext_CC.
```

Remark: *prop_extensionality* can be replaced in lemma *ext_prop_dep_proof_irrel_gen* by the weakest property *provable_prop_extensionality*.

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 $\neg b1 = b2$

```
  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. ∀ A, ¬A\/~~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 : ∀ A:Prop, A ∨ ¬ A.

  Definition or_elim_redl (A B C:Prop) (f:A → C) (g:B → C)
    (a:A) : f a = or_ind f g (or_introl B a) := eq_refl (f a).
  Definition or_elim_redr (A B C:Prop) (f:A → C) (g:B → C)
    (b:B) : g b = or_ind f g (or_intror A b) := eq_refl (g b).
  Scheme or_indd := Induction for or Sort Prop.

  Theorem proof_irrelevance_cci : ∀ (B:Prop) (b1 b2:B), b1 = b2.

End Proof_irrelevance_CCI.
```

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

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 ~~A ∨ ¬A is referred to with name KC in {*ChagrovZakharyaschev97*]

[*ChagrovZakharyaschev97*] Alexander Chagrov and Michael Zakharyaschev, "Modal Logic", Clarendon Press, 1997.

```
Definition weak_excluded_middle :=
  ∀ A:Prop, ~~A ∨ ¬A.
```

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

```
Definition weak_generalized_excluded_middle :=
  ∀ A B:Prop, ((A → B) → B) ∨ (A → B).
```

15.3.2 Gödel-Dummett axiom

$(A \to B) \lor (B \to A)$ 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 := ∀ A B:Prop, (A → B) ∨ (B → A).

Lemma excluded_middle_Godel_Dummett : excluded_middle → GodelDummett.
```

$(A \to B) \lor (B \to A)$ is equivalent to $(C \to A \lor B) \to (C \to A) \lor (C \to B)$ (proof from [*Dummett59*])

```
Definition RightDistributivityImplicationOverDisjunction :=
  ∀ A B C:Prop, (C → A∨B) → (C→A) ∨ (C→B).

Lemma Godel_Dummett_iff_right_distr_implication_over_disjunction :
  GodelDummett ↔ RightDistributivityImplicationOverDisjunction.
```

$(A \to B) \lor (B \to A)$ is stronger than the weak excluded middle

```
Lemma Godel_Dummett_weak_excluded_middle :
  GodelDummett → 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 Graundlagenforschung, 3:74- 78, 1957.

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

```
Definition IndependenceOfGeneralPremises :=
  ∀ (A:Type) (P:A → Prop) (Q:Prop),
    inhabited A → (Q → ∃ x, P x) → ∃ x, Q → P x.

Lemma
  independence_general_premises_right_distr_implication_over_disjunction :
  IndependenceOfGeneralPremises → RightDistributivityImplicationOverDisjunction.

Lemma independence_general_premises_Godel_Dummett :
  IndependenceOfGeneralPremises → GodelDummett.
```

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

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

Lemma independence_general_premises_drinker :
  IndependenceOfGeneralPremises ↔ DrinkerParadox.
```

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

Remark: generalized excluded middle is preferred here to avoid relying on the "ex falso quodlibet" property (i.e. *False* → ∀ *A*, *A*)

```
Definition generalized_excluded_middle :=
  ∀ A B:Prop, A ∨ (A → B).

Lemma excluded_middle_independence_general_premises :
  generalized_excluded_middle → 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 :
 ∀ P:U → Prop, ¬ (∀ n:U, P n) → ∃ n : U, ¬ P n.

Lemma not_all_not_ex :
 ∀ 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 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 : ∀ P:Prop, P ∨ ¬ P.

Lemma NNPP : ∀ p:Prop, ¬ ¬ p → p.
```

Peirce's law states ∀ P Q:Prop, ((P → Q) → P) → P. Thanks to ∀ P, *False* → P, it is equivalent to the following form

```
Lemma Peirce : ∀ P:Prop, ((P → False) → P) → P.

Lemma not_imply_elim : ∀ P Q:Prop, ¬ (P → Q) → P.

Lemma not_imply_elim2 : ∀ P Q:Prop, ¬ (P → Q) → ¬ Q.

Lemma imply_to_or : ∀ P Q:Prop, (P → Q) → ¬ P ∨ Q.

Lemma imply_to_and : ∀ P Q:Prop, ¬ (P → Q) → P ∧ ¬ Q.

Lemma or_to_imply : ∀ P Q:Prop, ¬ P ∨ Q → P → Q.

Lemma not_and_or : ∀ P Q:Prop, ¬ (P ∧ Q) → ¬ P ∨ ¬ Q.

Lemma or_not_and : ∀ P Q:Prop, ¬ P ∨ ¬ Q → ¬ (P ∧ Q).

Lemma not_or_and : ∀ P Q:Prop, ¬ (P ∨ Q) → ¬ P ∧ ¬ Q.

Lemma and_not_or : ∀ P Q:Prop, ¬ P ∧ ¬ Q → ¬ (P ∨ Q).

Lemma imply_and_or : ∀ P Q:Prop, (P → Q) → P ∨ Q → Q.

Lemma imply_and_or2 : ∀ P Q R:Prop, (P → Q) → P ∨ R → Q ∨ R.

Lemma proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.

Ltac classical_right := match goal with
 | _:_ |-?X1 ∨ _ ⇒ (elim (classic X1);intro;[left;trivial|right])
end.

Ltac classical_left := match goal with
| _:_ ⊢ _ \/?X1 ⇒ (elim (classic X1);intro;[right;trivial|left])
```

```
end.

Require Export EqdepFacts.

Module Eq_rect_eq.

Lemma eq_rect_eq :
  ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

End Eq_rect_eq.

Module EqdepTheory := EqdepTheory(Eq_rect_eq).
Export EqdepTheory.
```

Chapter 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 :
    ∀ (A:Type) (B:A → Type) (R:∀ x:A, B x → Prop),
      (∀ x : A, ∃! y : B x, R x y) →
      (∃ f : (∀ x:A, B x), ∀ x:A, R x (f x)).
```

Unique choice reifies functional relations into functions

```
Theorem unique_choice :
 ∀ (A B:Type) (R:A → B → Prop),
   (∀ x:A, ∃! y : B, R x y) →
   (∃ f:A→B, ∀ x:A, R x (f x)).
```

The following proof comes from [*ChicliPottierSimpson02*]

```
Require Import Setoid.
```

```
Theorem classic_set_in_prop_context :
  ∀ C:Prop, ((∀ P:Prop, {P} + {¬ P}) → C) → C.
```

```
Corollary not_not_classic_set :
  ((∀ P:Prop, {P} + {¬ P}) → False) → False.
```

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

Chapter 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 $\{n : nat \mid P\ n\}$. 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 : ∀ n, {P n}+{~(P n)}.
```

The termination argument is *before_witness n*, which says that any number before any witness (not necessarily the *x* of $\exists\ x : A,\ P\ x$) makes the search eventually stops.

```
Inductive before_witness : nat → Prop :=
  | stop : ∀ n, P n → before_witness n
  | next : ∀ n, before_witness (S n) → before_witness n.

Fixpoint O_witness (n : nat) : before_witness n → before_witness 0 :=
  match n return (before_witness n → before_witness 0) with
    | 0 ⇒ fun b ⇒ b
    | S n ⇒ fun b ⇒ O_witness n (next n b)
```

```
  end.
Definition inv_before_witness :
  ∀ n, before_witness n → ~(P n) → before_witness (S n) :=
  fun n b ⇒
    match b 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 then x if we try y after x, i.e., $y = S\ x$ and $P\ x$ is false. Then the absence of an infinite R-descending chain from 0 is equivalent to the termination of our searching algorithm.

```
Let R (x y : nat) : Prop := x = S y ∧ ¬ P y.
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 : nat → Prop.

Hypothesis P_decidable : ∀ x : nat, {P x} + {¬ P x}.

Definition constructive_ground_epsilon_nat (E : ∃ n : nat, P n) : nat
  := proj1_sig (constructive_indefinite_ground_description_nat P P_decidable E).

Definition constructive_ground_epsilon_spec_nat (E : (∃ n, P n)) : P (constructive_ground_epsilon_nat
E)
  := proj2_sig (constructive_indefinite_ground_description_nat P P_decidable E).

End ConstructiveGroundEpsilon_nat.

Section ConstructiveGroundEpsilon.
```

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

```
Variable A : Type.
Variable f : A → nat.
Variable g : nat → A.

Hypothesis gof_eq_id : ∀ x : A, g (f x) = x.

Variable P : A → Prop.

Hypothesis P_decidable : ∀ x : A, {P x} + {¬ P x}.

Definition P' (x : nat) : Prop := P (g x).

Lemma P'_decidable : ∀ n : nat, {P' n} + {¬ P' n}.

Lemma constructive_indefinite_ground_description : (∃ x : A, P x) → {x : A | P x}.

Lemma constructive_definite_ground_description : (∃! x : A, P x) → {x : A | P x}.

Definition constructive_ground_epsilon (E : ∃ x : A, P x) : A
  := proj1_sig (constructive_indefinite_ground_description E).

Definition constructive_ground_epsilon_spec (E : (∃ 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:Prop) := P ∨ ¬ P.

Theorem dec_not_not : ∀ P:Prop, decidable P → (¬ P → False) → P.

Theorem dec_True : decidable True.

Theorem dec_False : decidable False.

Theorem dec_or :
 ∀ A B:Prop, decidable A → decidable B → decidable (A ∨ B).

Theorem dec_and :
 ∀ A B:Prop, decidable A → decidable B → decidable (A ∧ B).

Theorem dec_not : ∀ A:Prop, decidable A → decidable (¬ A).

Theorem dec_imp :
 ∀ A B:Prop, decidable A → decidable B → decidable (A → B).

Theorem dec_iff :
 ∀ A B:Prop, decidable A → decidable B → decidable (A↔B).

Theorem not_not : ∀ P:Prop, decidable P → ¬ ¬ P → P.

Theorem not_or : ∀ A B:Prop, ¬ (A ∨ B) → ¬ A ∧ ¬ B.

Theorem not_and : ∀ A B:Prop, decidable A → ¬ (A ∧ B) → ¬ A ∨ ¬ B.

Theorem not_imp : ∀ A B:Prop, decidable A → ¬ (A → B) → A ∧ ¬ B.

Theorem imp_simp : ∀ A B:Prop, decidable A → (A → B) → ¬ A ∨ B.

Theorem not_iff :
  ∀ A B:Prop, decidable A → decidable B →
    ¬ (A ↔ B) → (A ∧ ¬ B) ∨ (¬ A ∧ B).
```

Results formulated with iff, used in FSetDecide. Negation are expanded since it is unclear whether setoid rewrite will always perform conversion.

We begin with lemmas that, when read from left to right, can be understood as ways to eliminate uses of *not*.

```
Theorem not_true_iff : (True → False) ↔ False.
```

```
Theorem not_false_iff : (False → 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 :
  ∀ (A : Type) (P : A→Prop),
    (∃! x, P x) → { x : A | 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 Quaterly, 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 :=
  ∀ P Q:bool → Prop, (∀ b:bool, P b ↔ Q b) → P = Q.
```

From predicate extensionality we get propositional extensionality hence proof-irrelevance

```
Require Import ClassicalFacts.
```

```
Variable pred_extensionality : PredicateExtensionality.
```

```
Lemma prop_ext : ∀ A B:Prop, (A ↔ B) → A = B.
```

```
Lemma proof_irrel : ∀ (A:Prop) (a1 a2:A), a1 = a2.
```

From proof-irrelevance and relational choice, we get guarded relational choice

```
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 :
  ∃ R : (bool → Prop) → bool → Prop,
   (∀ P:bool → Prop,
     (∃ b : bool, P b) →
      ∃ b : bool, P b ∧ R P b ∧ (∀ b':bool, R P b' → b = b')).
```

The proof of the excluded middle Remark: P could have been in Set or Type

```
Theorem pred_ext_and_rel_choice_imp_EM : ∀ P:Prop, P ∨ ¬ P.
```

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 : ∀ P:Prop , ∀ x y:P, x=y.
```

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

```
Variable A :Type.
Variables a1 a2 : A.
```

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

```
Definition A' := sigT (fun x ⇒ x=a1 ∨ x=a2).
```

```
Definition a1':A'.
Defined.
```

```
Definition a2':A'.
Defined.
```

By proof-irrelevance, projection is a retraction

Lemma projT1_injective : $a1$=$a2$ → 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 : ∀ x:A', ∃ y:bool ,
  (projT1 x = $a1$ ∧ y = true) ∨ (projT1 x = $a2$ ∧ y = false).

Thanks to the axiom of choice, the boolean witnesses move from the propositional world to the relevant world

Theorem proof_irrel_rel_choice_imp_eq_dec : $a1$=$a2$ ∨ ¬$a1$=$a2$.

An alternative more concise proof can be done by directly using the guarded relational choice

Lemma proof_irrel_rel_choice_imp_eq_dec' : $a1$=$a2$ ∨ ¬$a1$=$a2$.

End ProofIrrel_RelChoice_imp_EqEM.

25.3 Extensional Hilbert's epsilon description operator -> 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* : ∀ A : Type, inhabited A → (A → Prop) → A.

Hypothesis *epsilon_spec* :
  ∀ (A:Type) (i:inhabited A) (P:A→Prop),
  (∃ x, P x) → P (*epsilon* A i P).

Hypothesis *epsilon_extensionality* :
  ∀ (A:Type) (i:inhabited A) (P Q:A→Prop),
  (∀ a, P a ↔ Q a) → *epsilon* A i P = *epsilon* A i Q.

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

Theorem extensional_epsilon_imp_EM : ∀ P:Prop, P ∨ ¬ 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→Prop) :
  (∃! x:A, P x) → 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 : Type.

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

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

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

  Variable x : A.

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

```
  Let nu_constant : ∀ (y:A) (u v:x = y), nu u = nu v.
  Qed.
  Let nu_inv (y:A) (v:x = y) : x = y := comp (nu (eq_refl x)) v.
  Remark nu_left_inv : ∀ (y:A) (u:x = y), nu_inv (nu u) = u.
  Theorem eq_proofs_unicity : ∀ (y:A) (p1 p2:x = y), p1 = p2.
  Theorem K_dec :
    ∀ P:x = x → Prop, P (eq_refl x) → ∀ p:x = x, P p.
```

The corollary

```
  Let proj (P:A → Prop) (exP:ex P) (def:P x) : P x :=
    match exP with
      | ex_intro x' prf ⇒
        match eq_dec x' x with
          | or_introl eqprf ⇒ eq_ind x' P prf x eqprf
          | _ ⇒ def
        end
    end.
  Theorem inj_right_pair :
    ∀ (P:A → Prop) (y y':P x),
      ex_intro P x y = ex_intro P x y' → y = y'.
End EqdepDec.
Require Import EqdepFacts.
```

We deduce axiom K for (decidable) types `Theorem K_dec_type :`

```
  ∀ A:Type,
    (∀ x y:A, {x = y} + {x ≠ y}) →
    ∀ (x:A) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
Theorem K_dec_set :
  ∀ A:Set,
    (∀ x y:A, {x = y} + {x ≠ y}) →
    ∀ (x:A) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
```

We deduce the *eq_rect_eq* axiom for (decidable) types `Theorem eq_rect_eq_dec :`

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

We deduce the injectivity of dependent equality for decidable types `Theorem eq_dep_eq_dec :`

```
  ∀ A:Type,
    (∀ x y:A, {x = y} + {x ≠ y}) →
     ∀ (P:A→Type) (p:A) (x y:P p), eq_dep A P p x p y → x = y.
Theorem UIP_dec :
  ∀ (A:Type),
    (∀ x y:A, {x = y} + {x ≠ y}) →
    ∀ (x y:A) (p1 p2:x = y), p1 = p2.
```

```
Unset Implicit Arguments.
```

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 : ∀ x y:U, {x = y} + {x ≠ y}.
End DecidableType.
```

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

```
Module DecidableEqDep (M:DecidableType).
  Import M.
```

Invariance by Substitution of Reflexive Equality Proofs

```
  Lemma eq_rect_eq :
    ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.
```

Injectivity of Dependent Equality

```
  Theorem eq_dep_eq :
    ∀ (P:U→Type) (p:U) (x y:P p), eq_dep U P p x p y → x = y.
```

Uniqueness of Identity Proofs (UIP)

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

Uniqueness of Reflexive Identity Proofs

```
  Lemma UIP_refl : ∀ (x:U) (p:x = x), p = eq_refl x.
```

Streicher's axiom K

```
  Lemma Streicher_K :
    ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
```

Injectivity of equality on dependent pairs in `Type`

```
  Lemma inj_pairT2 :
    ∀ (P:U → Type) (p:U) (x y:P p),
      existT P p x = existT P p y → x = y.
```

Proof-irrelevance on subsets of decidable sets

```
  Lemma inj_pairP2 :
    ∀ (P:U → Prop) (x:U) (p q:P x),
      ex_intro P x p = ex_intro P x q → 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 : ∀ x y:U, {x = y} + {x ≠ y}.
End DecidableSet.
```

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

```
Module DecidableEqDepSet (M:DecidableSet).
  Import M.
  Module N:=DecidableEqDep(M).
```

Invariance by Substitution of Reflexive Equality Proofs

```
  Lemma eq_rect_eq :
    ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.
```

Injectivity of Dependent Equality

```
  Theorem eq_dep_eq :
    ∀ (P:U→Type) (p:U) (x y:P p), eq_dep U P p x p y → x = y.
```

Uniqueness of Identity Proofs (UIP)

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

Uniqueness of Reflexive Identity Proofs

```
  Lemma UIP_refl : ∀ (x:U) (p:x = x), p = eq_refl x.
```

Streicher's axiom K

```
  Lemma Streicher_K :
    ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
```

Proof-irrelevance on subsets of decidable sets

```
  Lemma inj_pairP2 :
    ∀ (P:U → Prop) (x:U) (p q:P x),
      ex_intro P x p = ex_intro P x q → p = q.
```

Injectivity of equality on dependent pairs in Type

```
  Lemma inj_pair2 :
    ∀ (P:U → Type) (p:U) (x y:P p),
      existT P p x = existT P p y → x = y.
```

Injectivity of equality on dependent pairs with second component in Type

```
  Notation inj_pairT2 := inj_pair2.
End DecidableEqDepSet.
```

From decidability to inj_pair2 `Lemma inj_pair2_eq_dec : ∀ A:Type, (∀ x y:A, {x=y}+{x≠y}) → ( ∀ (P:A → Type) (p:A) (x y:P p), existT P p x = existT P p y → 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. Eq_rect_eq <-> Eq_dep_eq <-> UIP <-> UIP_refl <-> K

3. Definition of the functor that builds properties of dependent equalities assuming axiom eq_rect_eq

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_eqdep := 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),
```

eq_dep p x q y → exist P p x = exist P q y.

Lemma eq_sig_iff_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.

Dependent equality is equivalent to a dependent pair of equalities

Set Implicit Arguments.

Lemma eq_sigT_sig_eq : ∀ X P (x1 x2:X) H1 H2, existT P x1 H1 = existT P x2 H2 ↔ {H:x1=x2 | rew H in H1 = H2}.

Lemma eq_sigT_fst :
∀ X P (x1 x2:X) H1 H2 (H:existT P x1 H1 = existT P x2 H2), x1 = x2.

Lemma eq_sigT_snd :
∀ X P (x1 x2:X) H1 H2 (H:existT P x1 H1 = existT P x2 H2), rew (eq_sigT_fst H) in H1 = H2.

Lemma eq_sig_fst :
∀ X P (x1 x2:X) H1 H2 (H:exist P x1 H1 = exist P x2 H2), x1 = x2.

Lemma eq_sig_snd :
∀ X P (x1 x2:X) H1 H2 (H:exist P x1 H1 = exist P x2 H2), rew (eq_sig_fst H) in H1 = H2.

Unset Implicit Arguments.

Exported hints

Hint Resolve eq_dep_intro: core.
Hint Immediate eq_dep_sym: core.

28.2 Eq_rect_eq <-> Eq_dep_eq <-> UIP <-> UIP_refl <-> K

Section Equivalences.

Variable U:Type.

Invariance by Substitution of Reflexive Equality Proofs

Definition Eq_rect_eq :=
∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

Injectivity of Dependent Equality

Definition Eq_dep_eq :=
∀ (P:U→Type) (p:U) (x y:P p), eq_dep p x p y → x = y.

Uniqueness of Identity Proofs (UIP)

Definition UIP_ :=
∀ (x y:U) (p1 p2:x = y), p1 = p2.

Uniqueness of Reflexive Identity Proofs

Definition UIP_refl_ :=
∀ (x:U) (p:x = x), p = eq_refl x.

Streicher's axiom K

```
Definition Streicher_K_ :=
  ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ 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 :
  Eq_rect_eq → ∀ (P:U→Type) (p:U) (x y:P p), eq_dep1 p x p y → x = y.
```

```
Lemma eq_rect_eq__eq_dep_eq : Eq_rect_eq → Eq_dep_eq.
```

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

```
Lemma eq_dep_eq__UIP : Eq_dep_eq → UIP_.
```

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

```
Lemma UIP__UIP_refl : UIP_ → UIP_refl_.
```

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

```
Lemma UIP_refl__Streicher_K : UIP_refl_ → Streicher_K_.
```

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

```
Lemma Streicher_K__eq_rect_eq : Streicher_K_ → 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* := ∀ (P:U → Set) (p:U) (x:P p) (h:p = p), x = *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 → 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 :=
  ∀ (P:U → Type) (p:U) (x y:P p), existT P p x = existT P p y → x = y.
```

```
Lemma eq_dep_eq__inj_pair2 : Eq_dep_eq U → Inj_dep_pair.
```

```
End Corollaries.
```

```
Notation Inj_dep_pairS := Inj_dep_pair.
Notation Inj_dep_pairT := Inj_dep_pair.
Notation eq_dep_eq__inj_pairT2 := eq_dep_eq__inj_pair2.
```

28.3 Definition of the functor that builds properties of dependent equalities assuming axiom eq_rect_eq

```
Module Type EqdepElimination.
```

```
  Axiom eq_rect_eq :
    ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p),
      x = eq_rect p Q x p h.
End EqdepElimination.
Module EqdepTheory (M:EqdepElimination).
  Section Axioms.
    Variable U:Type.
```

Invariance by Substitution of Reflexive Equality Proofs

```
Lemma eq_rect_eq :
  ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.
Lemma eq_rec_eq :
  ∀ (p:U) (Q:U → Set) (x:Q p) (h:p = p), x = eq_rec p Q x p h.
```

Injectivity of Dependent Equality

```
Lemma eq_dep_eq : ∀ (P:U→Type) (p:U) (x y:P p), eq_dep p x p y → x = y.
```

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

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

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

```
Lemma UIP_refl : ∀ (x:U) (p:x = x), p = eq_refl x.
```

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

```
Lemma Streicher_K :
  ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.
End Axioms.
```

UIP implies the injectivity of equality on dependent pairs in Type

```
Lemma inj_pair2 :
 ∀ (U:Type) (P:U → Type) (p:U) (x y:P p),
   existT P p x = existT P p y → 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 :
  ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

End Eq_rect_eq.

Module EqdepTheory := EqdepTheory(Eq_rect_eq).
Export EqdepTheory.
```

Exported hints

```
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 : ∀ {A B : Type} {f g : A → B},
  f = g → ∀ x, f x = g x.
```

Statements of functional extensionality for simple and dependent functions.

```
Axiom functional_extensionality_dep : ∀ {A} {B : A → Type},
  ∀ (f g : ∀ x : A, B x),
  (∀ x, f x = g x) → f = g.

Lemma functional_extensionality {A B} (f g : A → B) :
  (∀ x, f x = g x) → f = g.
```

Apply *functional_extensionality*, introducing variable x.

```
Tactic Notation "extensionality" ident(x) :=
  match goal with
    [ ⊢ ?X = ?Y ] ⇒
    (apply (@functional_extensionality _ _ X Y) ||
     apply (@functional_extensionality_dep _ _ X Y)) ; intro x
  end.
```

Eta expansion follows from extensionality.

```
Lemma eta_expansion_dep {A} {B : A → Type} (f : ∀ x : A, B x) :
  f = fun x ⇒ f x.

Lemma eta_expansion {A B} (f : A → B) : f = fun x ⇒ f x.
```

Chapter 31

Library Coq.Logic.ExtensionalityFacts

Some facts and definitions about extensionality

We investigate the relations between the following extensionality principles

- Functional extensionality
- Equality of projections from diagonal
- Unicity of inverse bijections
- Bijectivity of bijective composition

Table of contents

1. Definitions
2. Functional extensionality <-> Equality of projections from diagonal
3. Functional extensionality <-> Unicity of inverse bijections
4. Functional extensionality <-> Bijectivity of bijective composition

```
Set Implicit Arguments.
```

31.1 Definitions

Being an inverse

```
Definition is_inverse A B f g := (∀ a:A, g (f a) = a) ∧ (∀ b:B, f (g b) = b).
```

The diagonal over A and the one-one correspondence with A

```
Record Delta A := { pi1:A; pi2:A; eq:pi1=pi2 }.

Definition delta {A} (a:A) := {|pi1 := a; pi2 := a; eq := eq_refl a |}.

Lemma diagonal_projs_same_behavior : ∀ A (x:Delta A), pi1 x = pi2 x.

Lemma diagonal_inverse1 : ∀ A, is_inverse (A:=A) delta pi1.

Lemma diagonal_inverse2 : ∀ A, is_inverse (A:=A) delta pi2.
```

Functional extensionality

```
Local Notation FunctionalExtensionality :=
```

```
(∀ A B (f g : A → B), (∀ x, f x = g x) → f = g).
```

Equality of projections from diagonal

```
Local Notation EqDeltaProjs := (∀ A, pi1 = pi2 :> (Delta A → A)).
```

Unicity of bijection inverse

```
Local Notation UniqueInverse := (∀ A B (f:A→B) g1 g2, is_inverse f g1 → is_inverse f g2 →
g1 = g2).
```

Bijectivity of bijective composition

```
Definition action A B C (f:A→B) := (fun h:B→C ⇒ fun x ⇒ h (f x)).
Local Notation BijectivityBijectiveComp := (∀ A B C (f:A→B) g,
  is_inverse f g → is_inverse (A:=B→C) (action f) (action g)).
```

31.2 Functional extensionality <-> Equality of projections from diagonal

```
Theorem FunctExt_iff_EqDeltaProjs : FunctionalExtensionality ↔ EqDeltaProjs.
```

31.3 Functional extensionality <-> Unicity of bijection inverse

```
Lemma FunctExt_UniqInverse : FunctionalExtensionality → UniqueInverse.
Lemma UniqInverse_EqDeltaProjs : UniqueInverse → EqDeltaProjs.
Theorem FunctExt_iff_UniqInverse : FunctionalExtensionality ↔ UniqueInverse.
```

31.4 Functional extensionality <-> Bijectivity of bijective composition

```
Lemma FunctExt_BijComp : FunctionalExtensionality → BijectivityBijectiveComp.
Lemma BijComp_FunctExt : BijectivityBijectiveComp → FunctionalExtensionality.
```

Chapter 32

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.g

```
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 33

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

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

Lemma functional_choice :
  ∀ (A B : Type) (R:A→B→Prop),
    (∀ x : A, ∃ y : B, R x y) →
    (∃ f : A→B, ∀ x : A, R x (f x)).
```

Chapter 34

Library Coq.Logic.JMeq

John Major's Equality as proposed by Conor McBride

Reference:

McBride Elimination with a Motive, Proceedings of TYPES 2000, LNCS 2277, pp 197-216, 2002.

```
Set Implicit Arguments.

Inductive JMeq (A:Type) (x:A) : ∀ B:Type, B → Prop :=
    JMeq_refl : JMeq x x.

Hint Resolve JMeq_refl.

Lemma JMeq_sym : ∀ (A B:Type) (x:A) (y:B), JMeq x y → JMeq y x.

Hint Immediate JMeq_sym.

Lemma JMeq_trans :
 ∀ (A B C:Type) (x:A) (y:B) (z:C), JMeq x y → JMeq y z → JMeq x z.

Axiom JMeq_eq : ∀ (A:Type) (x y:A), JMeq x y → x = y.

Lemma JMeq_ind : ∀ (A:Type) (x:A) (P:A → Prop),
  P x → ∀ y, JMeq x y → P y.

Lemma JMeq_rec : ∀ (A:Type) (x:A) (P:A → Set),
  P x → ∀ y, JMeq x y → P y.

Lemma JMeq_rect : ∀ (A:Type) (x:A) (P:A→Type),
  P x → ∀ y, JMeq x y → P y.

Lemma JMeq_ind_r : ∀ (A:Type) (x:A) (P:A → Prop),
   P x → ∀ y, JMeq y x → P y.

Lemma JMeq_rec_r : ∀ (A:Type) (x:A) (P:A → Set),
   P x → ∀ y, JMeq y x → P y.

Lemma JMeq_rect_r : ∀ (A:Type) (x:A) (P:A → Type),
   P x → ∀ y, JMeq y x → P y.

Lemma JMeq_congr :
 ∀ (A:Type) (x:A) (B:Type) (f:A→B) (y:A), JMeq x y → f x = f y.
```

JMeq is equivalent to *eq_dep* Type (fun $X \Rightarrow X$)

```
Require Import Eqdep.

Lemma JMeq_eq_dep_id :
 ∀ (A B:Type) (x:A) (y:B), JMeq x y → eq_dep Type (fun X ⇒ X) A x B y.

Lemma eq_dep_id_JMeq :
 ∀ (A B:Type) (x:A) (y:B), eq_dep Type (fun X ⇒ X) A x B y → 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 :
 ∀ U P p x q y, eq_dep U P p x q y → JMeq x y.

Lemma eq_dep_strictly_stronger_JMeq :
 ∃ U P p q x y, JMeq x y ∧ ¬ 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 :
  ∀ U (P:U→Prop) p q (x:P p) (y:P q),
  p = q → JMeq x y → eq_dep U P p x q y.

Notation sym_JMeq := JMeq_sym (only parsing).
Notation trans_JMeq := JMeq_trans (only parsing).
```

Chapter 35

Library Coq.Logic.ProofIrrelevanceFacts

This defines the functor that build consequences of proof-irrelevance

```
Require Export EqdepFacts.

Module Type ProofIrrelevance.

  Axiom proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.

End ProofIrrelevance.

Module ProofIrrelevanceTheory (M:ProofIrrelevance).
```

Proof-irrelevance implies uniqueness of reflexivity proofs

```
  Module Eq_rect_eq.
   Lemma eq_rect_eq :
     ∀ (U:Type) (p:U) (Q:U → Type) (x:Q p) (h:p = p),
       x = eq_rect p Q x p h.
  End Eq_rect_eq.
```

Export the theory of injective dependent elimination

```
  Module EqdepTheory := EqdepTheory(Eq_rect_eq).
  Export EqdepTheory.

  Scheme eq_indd := Induction for eq Sort Prop.
```

We derive the irrelevance of the membership property for subsets

```
  Lemma subset_eq_compat :
    ∀ (U:Set) (P:U→Prop) (x y:U) (p:P x) (q:P y),
      x = y → exist P x p = exist P y q.

  Lemma subsetT_eq_compat :
    ∀ (U:Type) (P:U→Prop) (x y:U) (p:P x) (q:P y),
      x = y → existT P x p = existT P y q.

End ProofIrrelevanceTheory.
```

Chapter 36

Library Coq.Logic.ProofIrrelevance

This file axiomatizes proof-irrelevance and derives some consequences

```
Require Import ProofIrrelevanceFacts.

Axiom proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.

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

Module ProofIrrelevanceTheory := ProofIrrelevanceTheory(PI).
Export ProofIrrelevanceTheory.
```

Chapter 37

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 38

Library Coq.Logic.SetIsType

38.1 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 39

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 40

Library Coq.Arith.Arith

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

Chapter 41

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 ≤ r ∧ r < q.

  Lemma in_int_intro : ∀ p q r, p ≤ r → r < q → in_int p q r.
  Hint Resolve in_int_intro: arith v62.

  Lemma in_int_lt : ∀ p q r, in_int p q r → p < q.

  Lemma in_int_p_Sq :
    ∀ p q r, in_int p (S q) r → in_int p q r ∨ r = q :>nat.

  Lemma in_int_S : ∀ p q r, in_int p q r → in_int p (S q) r.
  Hint Resolve in_int_S: arith v62.

  Lemma in_int_Sp_q : ∀ p q r, in_int (S p) q r → in_int p q r.
  Hint Immediate in_int_Sp_q: arith v62.

  Lemma between_in_int :
    ∀ k l, between k l → ∀ r, in_int k l r → P r.

  Lemma in_int_between :
    ∀ k l, k ≤ l → (∀ r, in_int k l r → P r) → between k l.

  Lemma exists_in_int :
    ∀ k l, exists_between k l → exists2 m : nat, in_int k l m & Q m.

  Lemma in_int_exists : ∀ k l r, in_int k l r → Q r → exists_between k l.

  Lemma between_or_exists :
    ∀ k l,
      k ≤ l →
      (∀ n:nat, in_int k l n → P n ∨ Q n) →
      between k l ∨ exists_between k l.

  Lemma between_not_exists :
    ∀ k l,
      between k l →
      (∀ n:nat, in_int k l n → P n → ¬ Q n) → ¬ exists_between k l.

  Inductive P_nth (init:nat) : nat → nat → Prop :=
    | nth_O : P_nth init init 0
    | nth_S :
      ∀ k l (n:nat),
        P_nth init k n → between (S k) l → Q l → P_nth init l (S n).

  Lemma nth_le : ∀ (init:nat) l (n:nat), P_nth init l n → init ≤ l.

  Definition eventually (n:nat) := exists2 k : nat, k ≤ n & Q k.

  Lemma event_O : eventually 0 → Q 0.

End Between.

Hint Resolve nth_O bet_S bet_emp bet_eq between_Sk_l exists_S exists_le
  in_int_S in_int_intro: arith v62.
Hint Immediate in_int_Sp_q exists_le_S exists_S_le: arith v62.
```

Chapter 42

Library Coq.Arith.Bool_nat

```
Require Export Compare_dec.
Require Export Peano_dec.
Require Import Sumbool.

Local Open Scope nat_scope.

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

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

```
Definition notzerop n := sumbool_not _ _ (zerop n).
Definition lt_ge_dec : ∀ x y, {x < y} + {x ≥ y} :=
  fun n m ⇒ sumbool_not _ _ (le_lt_dec m n).

Definition nat_lt_ge_bool x y := bool_of_sumbool (lt_ge_dec x y).
Definition nat_ge_lt_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (lt_ge_dec x y)).

Definition nat_le_gt_bool x y := bool_of_sumbool (le_gt_dec x y).
Definition nat_gt_le_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (le_gt_dec x y)).

Definition nat_eq_bool x y := bool_of_sumbool (eq_nat_dec x y).
Definition nat_noteq_bool x y :=
  bool_of_sumbool (sumbool_not _ _ (eq_nat_dec x y)).

Definition zerop_bool x := bool_of_sumbool (zerop x).
Definition notzerop_bool x := bool_of_sumbool (notzerop x).
```

Chapter 43

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 : $\{n \leq m\} + \{m < n\}$.

Definition le_le_S_dec n m : $\{n \leq m\} + \{\mathsf{S}\ m \leq n\}$.

Definition le_ge_dec n m : $\{n \leq m\} + \{n \geq m\}$.

Definition le_gt_dec n m : $\{n \leq m\} + \{n > m\}$.

Definition le_lt_eq_dec n m : $n \leq m \rightarrow \{n < m\} + \{n = m\}$.

Theorem le_dec : $\forall\ n\ m, \{n \leq m\} + \{\neg\ n \leq m\}$.

Theorem lt_dec : $\forall\ n\ m, \{n < m\} + \{\neg\ n < m\}$.

Theorem gt_dec : $\forall\ n\ m, \{n > m\} + \{\neg\ n > m\}$.

Theorem ge_dec : $\forall\ n\ m, \{n \geq m\} + \{\neg\ n \geq m\}$.

Proofs of decidability

Theorem dec_le : $\forall\ n\ m$, decidable $(n \leq m)$.

Theorem dec_lt : $\forall\ n\ m$, decidable $(n < m)$.

Theorem dec_gt : $\forall\ n\ m$, decidable $(n > m)$.

Theorem dec_ge : $\forall\ n\ m$, decidable $(n \geq m)$.

Theorem not_eq : $\forall\ n\ m, n \neq m \rightarrow n < m \lor m < n$.

Theorem not_le : $\forall\ n\ m, \neg\ n \leq m \rightarrow n > m$.

```
Theorem not_gt : ∀ n m, ¬ n > m → n ≤ m.
```

```
Theorem not_ge : ∀ n m, ¬ n ≥ m → n < m.
```

```
Theorem not_lt : ∀ n m, ¬ n < m → n ≥ m.
```

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

```
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 : ∀ n m, nat_compare (S n) (S m) = nat_compare n m.
```

```
Lemma nat_compare_eq_iff : ∀ n m, nat_compare n m = Eq ↔ n = m.
```

```
Lemma nat_compare_eq : ∀ n m, nat_compare n m = Eq → n = m.
```

```
Lemma nat_compare_lt : ∀ n m, n<m ↔ nat_compare n m = Lt.
```

```
Lemma nat_compare_gt : ∀ n m, n>m ↔ nat_compare n m = Gt.
```

```
Lemma nat_compare_le : ∀ n m, n≤m ↔ nat_compare n m ≠ Gt.
```

```
Lemma nat_compare_ge : ∀ n m, n≥m ↔ nat_compare n m ≠ Lt.
```

```
Lemma nat_compare_spec :
  ∀ x y, CompareSpec (x=y) (x<y) (y<x) (nat_compare x y).
```

Some projections of the above equivalences.

```
Lemma nat_compare_Lt_lt : ∀ n m, nat_compare n m = Lt → n<m.
```

```
Lemma nat_compare_Gt_gt : ∀ n m, nat_compare n m = Gt → n>m.
```

A previous definition of *nat_compare* in terms of *lt_eq_lt_dec*. The new version avoids the creation of proof parts.

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

```
Lemma nat_compare_equiv: ∀ n m,
 nat_compare n m = 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 ⇒ false
                  | S n' ⇒ leb m' n'
                end
  end.
Lemma leb_correct : ∀ m n, m ≤ n → leb m n = true.
Lemma leb_complete : ∀ m n, leb m n = true → m ≤ n.
Lemma leb_iff : ∀ m n, leb m n = true ↔ m ≤ n.
Lemma leb_correct_conv : ∀ m n, m < n → leb n m = false.
Lemma leb_complete_conv : ∀ m n, leb n m = false → m < n.
Lemma leb_iff_conv : ∀ m n, leb n m = false ↔ m < n.
Lemma leb_compare : ∀ n m, leb n m = true ↔ nat_compare n m ≠ Gt.
```

Chapter 44

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 : ∀ n m, {n ≤ m} + {m ≤ n}.

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

Lemma le_decide : ∀ n m, n ≤ m → lt_or_eq n m.

Lemma le_le_S_eq : ∀ n m, n ≤ m → S n ≤ m ∨ n = m.

Lemma discrete_nat :
  ∀ n m, n < m → S n = m ∨ (∃ r : nat, m = S (S (n + r))).

Require Export Wf_nat.

Require Export Min Max.
```

Chapter 45

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
  | O ⇒ 0
  | S O ⇒ 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.
```

$0 < n \Rightarrow n/2 < n$

```
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 ↔ div2 n = div2 (S n)) ∧
  (odd n ↔ S (div2 n) = div2 (S n)).
```

Properties related to the double ($2n$)

```
Definition double n := n + n.

Hint Unfold double: arith.

Lemma double_S : ∀ n, double (S n) = S (S (double n)).

Lemma double_plus : ∀ n (m:nat), double (n + m) = double n + double m.

Hint Resolve double_S: arith.

Lemma even_odd_double :
  ∀ n,
    (even n ↔ n = double (div2 n)) ∧ (odd n ↔ n = S (double (div2 n))).
```

Specializations

```
Lemma even_double : ∀ n, even n → n = double (div2 n).

Lemma double_even : ∀ n, n = double (div2 n) → even n.

Lemma odd_double : ∀ n, odd n → n = S (double (div2 n)).

Lemma double_odd : ∀ n, n = S (double (div2 n)) → odd n.

Hint Resolve even_double double_even odd_double double_odd: arith.
```

Application:

- if n is even then there is a p such that $n = 2p$
- if n is odd then there is a p such that $n = 2p{+}1$

(Immediate: it is $n/2$)

```
Lemma even_2n : ∀ n, even n → {p : nat | n = double p}.

Lemma odd_S2n : ∀ n, odd n → {p : nat | n = S (double p)}.
```

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

```
Lemma div2_double : ∀ n:nat, div2 (2×n) = n.

Lemma div2_double_plus_one : ∀ n:nat, div2 (S (2×n)) = n.
```

Chapter 46

Library Coq.Arith.EqNat

Equality on natural numbers

```
Local Open Scope nat_scope.

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

46.1 Propositional equality

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

Theorem eq_nat_refl : ∀ n, eq_nat n n.
Hint Resolve eq_nat_refl: arith v62.
```

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

```
Lemma eq_eq_nat : ∀ n m, n = m → eq_nat n m.
Hint Immediate eq_eq_nat: arith v62.

Lemma eq_nat_eq : ∀ n m, eq_nat n m → n = m.
Hint Immediate eq_nat_eq: arith v62.

Theorem eq_nat_is_eq : ∀ n m, eq_nat n m ↔ n = m.

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

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

46.2 Boolean equality on *nat*

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

```
  match n, m with
    | O, O ⇒ true
    | O, S _ ⇒ false
    | S _, O ⇒ false
    | S n1, S m1 ⇒ beq_nat n1 m1
  end.
```

```
Lemma beq_nat_refl : ∀ n, true = beq_nat n n.
```

```
Definition beq_nat_eq : ∀ x y, true = beq_nat x y → x = y.
```

```
Lemma beq_nat_true : ∀ x y, beq_nat x y = true → x=y.
```

```
Lemma beq_nat_false : ∀ x y, beq_nat x y = false → x≠y.
```

```
Lemma beq_nat_true_iff : ∀ x y, beq_nat x y = true ↔ x=y.
```

```
Lemma beq_nat_false_iff : ∀ x y, beq_nat x y = false ↔ x≠y.
```

Chapter 47

Library Coq.Arith.Euclid

```
Require Import Mult.
Require Import Compare_dec.
Require Import Wf_nat.

Local Open Scope nat_scope.

Implicit Types a b n q r : nat.

Inductive diveucl a b : Set :=
  divex : ∀ q r, b > r → a = q × b + r → diveucl a b.

Lemma eucl_dev : ∀ n, n > 0 → ∀ m:nat, diveucl m n.

Lemma quotient :
  ∀ n,
    n > 0 →
    ∀ m:nat, {q : nat | ∃ r : nat, m = q × n + r ∧ n > r}.

Lemma modulo :
  ∀ n,
    n > 0 →
    ∀ m:nat, {r : nat | ∃ q : nat, m = q × n + r ∧ n > r}.
```

Chapter 48

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.
```

48.1 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.
```

48.2 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,
    (odd (n + m) $\leftrightarrow$ odd n $\wedge$ even m $\vee$ even n $\wedge$ odd m) $\wedge$
    (even (n + m) $\leftrightarrow$ even n $\wedge$ even m $\vee$ odd n $\wedge$ odd m).

Lemma even_plus_even_inv_r : $\forall$ n m, even (n + m) $\rightarrow$ even n $\rightarrow$ even m.

Lemma even_plus_even_inv_l : $\forall$ n m, even (n + m) $\rightarrow$ even m $\rightarrow$ even n.

Lemma even_plus_odd_inv_r : $\forall$ n m, even (n + m) $\rightarrow$ odd n $\rightarrow$ odd m.

Lemma even_plus_odd_inv_l : $\forall$ n m, even (n + m) $\rightarrow$ odd m $\rightarrow$ odd n.
Hint Resolve even_even_plus odd_even_plus: *arith*.

Lemma odd_plus_even_inv_l : $\forall$ n m, odd (n + m) $\rightarrow$ odd m $\rightarrow$ even n.

Lemma odd_plus_even_inv_r : $\forall$ n m, odd (n + m) $\rightarrow$ odd n $\rightarrow$ even m.

Lemma odd_plus_odd_inv_l : $\forall$ n m, odd (n + m) $\rightarrow$ even m $\rightarrow$ odd n.

Lemma odd_plus_odd_inv_r : $\forall$ n m, odd (n + m) $\rightarrow$ even n $\rightarrow$ odd m.
Hint Resolve odd_plus_l odd_plus_r: *arith*.

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

Lemma even_mult_aux :
  $\forall$ n m,
    (odd ($n \times m$) $\leftrightarrow$ odd n $\wedge$ odd m) $\wedge$ (even ($n \times m$) $\leftrightarrow$ even n $\vee$ even m).

Lemma even_mult_l : $\forall$ n m, even n $\rightarrow$ even ($n \times m$).

Lemma even_mult_r : $\forall$ n m, even m $\rightarrow$ even ($n \times m$).
Hint Resolve even_mult_l even_mult_r: *arith*.

Lemma even_mult_inv_r : $\forall$ n m, even ($n \times m$) $\rightarrow$ odd n $\rightarrow$ even m.

Lemma even_mult_inv_l : $\forall$ n m, even ($n \times m$) $\rightarrow$ odd m $\rightarrow$ even n.

Lemma odd_mult : $\forall$ n m, odd n $\rightarrow$ odd m $\rightarrow$ odd ($n \times m$).
Hint Resolve even_mult_l even_mult_r odd_mult: *arith*.

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

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

Chapter 49

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
    | O ⇒ 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 50

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.
```

50.1 Order and successor

```
Theorem gt_Sn_O : ∀ n, S n > 0.
Hint Resolve gt_Sn_O: arith v62.
Theorem gt_Sn_n : ∀ n, S n > n.
Hint Resolve gt_Sn_n: arith v62.
Theorem gt_n_S : ∀ n m, n > m → S n > S m.
Hint Resolve gt_n_S: arith v62.
Lemma gt_S_n : ∀ n m, S m > S n → m > n.
Hint Immediate gt_S_n: arith v62.
Theorem gt_S : ∀ n m, S n > m → n > m ∨ m = n.
Lemma gt_pred : ∀ n m, m > S n → pred m > n.
Hint Immediate gt_pred: arith v62.
```

50.2 Irreflexivity

```
Lemma gt_irrefl : ∀ n, ¬ n > n.
Hint Resolve gt_irrefl: arith v62.
```

50.3 Asymmetry

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

Hint Resolve gt_asym: *arith v62*.

50.4 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 : $\forall\ n\ m,\ \mathsf{S}\ n \leq m \rightarrow m > n$.
Hint Immediate le_S_gt: *arith v62*.

Lemma gt_S_le : $\forall\ n\ m,\ \mathsf{S}\ m > n \rightarrow n \leq m$.
Hint Immediate gt_S_le: *arith v62*.

Lemma gt_le_S : $\forall\ n\ m,\ m > n \rightarrow \mathsf{S}\ n \leq m$.
Hint Resolve gt_le_S: *arith v62*.

Lemma le_gt_S : $\forall\ n\ m,\ n \leq m \rightarrow \mathsf{S}\ m > n$.
Hint Resolve le_gt_S: *arith v62*.

50.5 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 : $\forall\ n\ m\ p,\ \mathsf{S}\ n > m \rightarrow m > p \rightarrow n > p$.

Hint Resolve gt_trans_S le_gt_trans gt_le_trans: *arith v62*.

50.6 Comparison to 0

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

50.7 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 51

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.

51.1 *le* is a pre-order

Reflexivity Theorem le_refl : $\forall$ n, $n \leq n$.

Transitivity Theorem le_trans : $\forall$ n m p, $n \leq m \to m \leq p \to n \leq p$.
Hint Resolve le_trans: *arith v62*.

51.2 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, $n \leq 0 \to 0 = n$.
Hint Immediate le_n_0_eq: *arith v62*.

le and successor

Theorem le_n_S : $\forall$ n m, $n \leq m \to$ 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 : $\forall\ n\ m,\ \mathsf{S}\ n \le m \to n \le m$.
Hint Immediate le_Sn_le: *arith v62*.

Theorem le_S_n : $\forall\ n\ m,\ \mathsf{S}\ n \le \mathsf{S}\ m \to n \le m$.
Hint Immediate le_S_n: *arith v62*.

Theorem le_Sn_n : $\forall\ n,\ \neg\ \mathsf{S}\ n \le n$.
Hint Resolve le_Sn_n: *arith v62*.

le and predecessor

Theorem le_pred_n : $\forall\ n$, pred $n \le n$.
Hint Resolve le_pred_n: *arith v62*.

Theorem le_pred : $\forall\ n\ m,\ n \le m \to$ pred $n \le$ pred m.

51.3 *le* is a order on *nat*

Antisymmetry

Theorem le_antisym : $\forall\ n\ m,\ n \le m \to m \le n \to n = m$.
Hint Immediate le_antisym: *arith v62*.

51.4 A different elimination principle for the order on natural numbers

Lemma le_elim_rel :
 $\forall\ P$:nat $\to$ nat $\to$ Prop,
  $(\forall\ p,\ P\ 0\ p) \to$
  $(\forall\ p\ (q{:}\mathsf{nat}),\ p \le q \to P\ p\ q \to P\ (\mathsf{S}\ p)\ (\mathsf{S}\ q)) \to$
  $\forall\ n\ m,\ n \le m \to P\ n\ m$.

Chapter 52

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.

52.1 Irreflexivity

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

52.2 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*.

52.3 Asymmetry

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

52.4 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*.

52.5 Predecessor

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

Lemma lt_pred : $\forall\ n\ m,\ S\ n < m \rightarrow n < \mathrm{pred}\ m$.
Hint Immediate lt_pred: *arith v62*.

Lemma lt_pred_n_n : $\forall\ n,\ 0 < n \rightarrow \mathrm{pred}\ n < n$.
Hint Resolve lt_pred_n_n: *arith v62*.

52.6 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*.

52.7 Large = strict or equal

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

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

Theorem lt_le_weak : $\forall\ n\ m,\ n < m \rightarrow n \leq m$.
Hint Immediate lt_le_weak: *arith v62*.

52.8 Dichotomy

Theorem le_or_lt : $\forall\ n\ m,\ n \le m \lor m < n$.

Theorem nat_total_order : $\forall\ n\ m,\ n \neq m \to n < m \lor m < n$.

52.9 Comparison to 0

Theorem neq_0_lt : $\forall\ n,\ 0 \neq n \to 0 < n$.
Hint Immediate neq_0_lt: *arith v62*.

Theorem lt_0_neq : $\forall\ n,\ 0 < n \to 0 \neq n$.
Hint Immediate lt_0_neq: *arith v62*.

Chapter 53

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 54

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
  | O, _ => n
  | S k, O => 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.

54.1 0 is right neutral

Lemma minus_n_O : ∀ n, n = n - 0.
Hint Resolve minus_n_O: *arith v62*.

54.2 Permutation with successor

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

Theorem pred_of_minus : ∀ n, pred n = n - 1.

54.3 Diagonal

Lemma minus_diag : ∀ n, n - n = 0.

```
Lemma minus_diag_reverse : ∀ n, 0 = n - n.
Hint Resolve minus_diag_reverse: arith v62.
Notation minus_n_n := minus_diag_reverse.
```

54.4 Simplification

```
Lemma minus_plus_simpl_l_reverse : ∀ n m p, n - m = p + n - (p + m).
Hint Resolve minus_plus_simpl_l_reverse: arith v62.
```

54.5 Relation with plus

```
Lemma plus_minus : ∀ n m p, n = m + p → p = n - m.
Hint Immediate plus_minus: arith v62.
Lemma minus_plus : ∀ n m, n + m - n = m.
Hint Resolve minus_plus: arith v62.
Lemma le_plus_minus : ∀ n m, n ≤ m → m = n + (m - n).
Hint Resolve le_plus_minus: arith v62.
Lemma le_plus_minus_r : ∀ n m, n ≤ m → n + (m - n) = m.
Hint Resolve le_plus_minus_r: arith v62.
```

54.6 Relation with order

```
Theorem minus_le_compat_r : ∀ n m p : nat, n ≤ m → n - p ≤ m - p.
Theorem minus_le_compat_l : ∀ n m p : nat, n ≤ m → p - m ≤ p - n.
Corollary le_minus : ∀ n m, n - m ≤ n.
Lemma lt_minus : ∀ n m, m ≤ n → 0 < m → n - m < n.
Hint Resolve lt_minus: arith v62.
Lemma lt_O_minus_lt : ∀ n m, 0 < n - m → m < n.
Hint Immediate lt_O_minus_lt: arith v62.
Theorem not_le_minus_0 : ∀ n m, ¬ m ≤ n → n - m = 0.
```

Chapter 55

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 56

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*.

56.1 *nat* is a semi-ring

56.1.1 Zero property

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

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

56.1.2 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*.

56.1.3 Commutativity

Lemma mult_comm : $\forall$ n m, $n \times m = m \times n$.
Hint Resolve mult_comm: *arith v62*.

56.1.4 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*.

56.1.5 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*.

56.1.6 Inversion lemmas

Lemma mult_is_O : $\forall$ n m, $n \times m = 0 \rightarrow n = 0 \lor m = 0$.

Lemma mult_is_one : $\forall$ n m, $n \times m = 1 \rightarrow n = 1 \land m = 1$.

56.1.7 Multiplication and successor

Lemma mult_succ_l : $\forall$ n m:nat, S $n \times m = n \times m + m$.

Lemma mult_succ_r : $\forall$ n m:nat, $n \times$ S $m = n \times m + n$.

56.2 Compatibility with orders

Lemma mult_O_le : $\forall$ n m, $m = 0 \lor n \leq m \times n$.
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: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 $n \times p \rightarrow m \leq p$.

56.3 n|->2*n and n|->2n+1 have disjoint image

```
Theorem odd_even_lem : ∀ p q, 2 × p + 1 ≠ 2 × q.
```

56.4 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
    | O ⇒ s
    | S p ⇒ mult_acc (tail_plus m s) m p
  end.
```

```
Lemma mult_acc_aux : ∀ n m p, m + n × p = mult_acc m p n.
```

```
Definition tail_mult n m := mult_acc 0 m n.
```

```
Lemma mult_tail_mult : ∀ n m, n × m = tail_mult n m.
```

TailSimpl transforms any *tail_plus* and *tail_mult* into *plus* and *mult* and simplify

```
Ltac tail_simpl :=
  repeat rewrite ← plus_tail_plus; repeat rewrite ← mult_tail_mult;
    simpl.
```

Chapter 57

Library Coq.Arith.Peano_dec

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

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

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

```
Theorem eq_nat_dec : ∀ n m, {n = m} + {n ≠ m}.
```

```
Hint Resolve O_or_S eq_nat_dec: arith.
```

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

```
Definition UIP_nat:= Eqdep_dec.UIP_dec eq_nat_dec.
```

```
Lemma le_unique: ∀ m n (h1 h2: m ≤ n), h1 = h2.
```

Chapter 58

Library Coq.Arith.Plus

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

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O => 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.

58.1 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*).

58.2 Commutativity

Lemma plus_comm : ∀ *n m*, *n* + *m* = *m* + *n*.
Hint Immediate plus_comm: *arith v62*.

58.3 Associativity

Definition plus_Snm_nSm : ∀ *n m*, S *n* + *m* = *n* + S *m*:=
plus_n_Sm.

Lemma plus_assoc : ∀ *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*.

58.4 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.

58.5 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.

58.6 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}.

58.7 Derived properties

```
Lemma plus_permute_2_in_4 : ∀ n m p q, n + m + (p + q) = n + p + (m + q).
```

58.8 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 ⇒ m
    | S n ⇒ tail_plus n (S m)
  end.
```

```
Lemma plus_tail_plus : ∀ n m, n + m = tail_plus n m.
```

58.9 Discrimination

```
Lemma succ_plus_discr : ∀ n m, n ≠ S (plus m n).
```

```
Lemma n_SSn : ∀ n, n ≠ S (S n).
```

```
Lemma n_SSSn : ∀ n, n ≠ S (S (S n)).
```

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

Chapter 59

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 :
```

```
  ∀ P:A → Set,
    (∀ x:A, (∀ y:A, gtof y x → P y) → P x) → ∀ a:A, P a.
Theorem induction_ltof2 :
  ∀ P:A → Set,
    (∀ x:A, (∀ y:A, ltof y x → P y) → P x) → ∀ a:A, P a.
Theorem induction_gtof2 :
  ∀ P:A → Set,
    (∀ x:A, (∀ y:A, gtof y x → P y) → P x) → ∀ 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 → A → Prop.
Hypothesis H_compat : ∀ x y:A, R x y → 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 :
  ∀ n (P:nat → Set), (∀ n, (∀ m, m < n → P m) → P n) → P n.
Lemma lt_wf_rec :
  ∀ n (P:nat → Set), (∀ n, (∀ m, m < n → P m) → P n) → P n.
Lemma lt_wf_ind :
  ∀ n (P:nat → Prop), (∀ n, (∀ m, m < n → P m) → P n) → P n.
Lemma gt_wf_rec :
  ∀ n (P:nat → Set), (∀ n, (∀ m, n > m → P m) → P n) → P n.
Lemma gt_wf_ind :
  ∀ n (P:nat → Prop), (∀ n, (∀ m, n > m → P m) → P n) → P n.
Lemma lt_wf_double_rec :
 ∀ P:nat → nat → Set,
   (∀ n m,
     (∀ p q, p < n → P p q) →
     (∀ p, p < m → P n p) → P n m) → ∀ n m, P n m.
Lemma lt_wf_double_ind :
  ∀ P:nat → nat → Prop,
    (∀ n m,
      (∀ p (q:nat), p < n → P p q) →
      (∀ p, p < m → P n p) → P n m) → ∀ n m, P n m.
Hint Resolve lt_wf: arith.
Hint Resolve well_founded_lt_compat: arith.
Section LT_WF_REL.
  Variable A : Set.
  Variable R : A → A → Prop.
  Variable F : A → nat → Prop.
```

```
  Definition inv_lt_rel x y := exists2 n, F x n & (∀ m, F y m → n < m).

  Hypothesis F_compat : ∀ x y:A, R x y → inv_lt_rel x y.
  Remark acc_lt_rel : ∀ x:A, (∃ n, F x n) → Acc R x.

  Theorem well_founded_inv_lt_rel_compat : well_founded R.

End LT_WF_REL.

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

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

```
Set Implicit Arguments.

Require Import Le.
Require Import Compare_dec.
Require Import Decidable.

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

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

Unset Implicit Arguments.

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