

The Coq Proof Assistant

The standard library

May 31, 2019

Version 8.9.1¹

πr^2 Project (formerly LogiCal, then TypiCal)

¹This research was partly supported by IST working group “Types”

V8.9.1, May 31, 2019

©INRIA 1999-2004 (CoQ versions 7.x)

©INRIA 2004-2017 (CoQ versions 8.x)

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

Contents

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

The standard library is composed of the following subdirectories:

Logic Classical logic and dependent equality

Bool Booleans (basic functions and results)

Arith Basic Peano arithmetic

ZArith Basic integer arithmetic

Reals Classical Real Numbers and Analysis

Lists Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)

Sets Sets (classical, constructive, finite, infinite, power set, etc.)

Relations Relations (definitions and basic results).

Sorting Sorted list (basic definitions and heapsort correctness).

Wellfounded Well-founded relations (basic results).

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

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

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

Chapter 1

Library `Coq.Arith.Arith`

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

Chapter 2

Library `Coq.Arith.Arith_base`

```
Require Export PeanoNat.  
Require Export Le.  
Require Export Lt.  
Require Export Plus.  
Require Export Gt.  
Require Export Minus.  
Require Export Mult.  
Require Export Between.  
Require Export Peano_dec.  
Require Export Compare_dec.  
Require Export Factorial.  
Require Export EqNat.  
Require Export Wf_nat.
```

Chapter 3

Library `Coq.Arith.Between`

`Require Import Le.`

`Require Import Lt.`

`Local Open Scope nat_scope.`

`Implicit Types k l p q r : nat.`

`Section Between.`

`Variables P Q : nat → Prop.`

The `between` type expresses the concept $\forall i: \text{nat}, k \leq i < l \rightarrow P i$. `Inductive between k :`
`nat → Prop :=`

| `bet_emp` : `between k k`
| `bet_S` : $\forall l, \text{between } k l \rightarrow P l \rightarrow \text{between } k (S l)$.

`Hint Constructors between: arith.`

`Lemma bet_eq` : $\forall k l, l = k \rightarrow \text{between } k l$.

`Hint Resolve bet_eq: arith.`

`Lemma between_le` : $\forall k l, \text{between } k l \rightarrow k \leq l$.

`Hint Immediate between_le: arith.`

`Lemma between_Sk_l` : $\forall k l, \text{between } k l \rightarrow S k \leq l \rightarrow \text{between } (S k) l$.

`Hint Resolve between_Sk_l: arith.`

`Lemma between_restr` :

$\forall k l (m:\text{nat}), k \leq l \rightarrow l \leq m \rightarrow \text{between } k m \rightarrow \text{between } l m$.

The `exists_between` type expresses the concept $\exists i: \text{nat}, k \leq i < l \wedge Q i$. `Inductive`
`exists_between k : nat → Prop :=`

| `exists_S` : $\forall l, \text{exists_between } k l \rightarrow \text{exists_between } k (S l)$
| `exists_le` : $\forall l, k \leq l \rightarrow Q l \rightarrow \text{exists_between } k (S l)$.

`Hint Constructors exists_between: arith.`

`Lemma exists_le_S` : $\forall k l, \text{exists_between } k l \rightarrow S k \leq l$.

`Lemma exists_lt` : $\forall k l, \text{exists_between } k l \rightarrow k < l$.

`Hint Immediate exists_le_S exists_lt: arith.`

`Lemma exists_S_le` : $\forall k l, \text{exists_between } k (S l) \rightarrow k \leq l$.

Hint Immediate *exists_S_le*: *arith*.

Definition *in_int* $p\ q\ r := p \leq r \wedge r < q$.

Lemma *in_int_intro* : $\forall p\ q\ r, p \leq r \rightarrow r < q \rightarrow \text{in_int } p\ q\ r$.

Hint Resolve *in_int_intro*: *arith*.

Lemma *in_int_lt* : $\forall p\ q\ r, \text{in_int } p\ q\ r \rightarrow p < q$.

Lemma *in_int_p_Sq* :

$\forall p\ q\ r, \text{in_int } p\ (\text{S } q)\ r \rightarrow \text{in_int } p\ q\ r \vee r = q$.

Lemma *in_int_S* : $\forall p\ q\ r, \text{in_int } p\ q\ r \rightarrow \text{in_int } p\ (\text{S } q)\ r$.

Hint Resolve *in_int_S*: *arith*.

Lemma *in_int_Sp_q* : $\forall p\ q\ r, \text{in_int } (\text{S } p)\ q\ r \rightarrow \text{in_int } p\ q\ r$.

Hint Immediate *in_int_Sp_q*: *arith*.

Lemma *between_in_int* :

$\forall k\ l, \text{between } k\ l \rightarrow \forall r, \text{in_int } k\ l\ r \rightarrow P\ r$.

Lemma *in_int_between* :

$\forall k\ l, k \leq l \rightarrow (\forall r, \text{in_int } k\ l\ r \rightarrow P\ r) \rightarrow \text{between } k\ l$.

Lemma *exists_in_int* :

$\forall k\ l, \text{exists_between } k\ l \rightarrow \text{exists2 } m : \text{nat}, \text{in_int } k\ l\ m \ \& \ Q\ m$.

Lemma *in_int_exists* : $\forall k\ l\ r, \text{in_int } k\ l\ r \rightarrow Q\ r \rightarrow \text{exists_between } k\ l$.

Lemma *between_or_exists* :

$\forall k\ l,$
 $k \leq l \rightarrow$
 $(\forall n:\text{nat}, \text{in_int } k\ l\ n \rightarrow P\ n \vee Q\ n) \rightarrow$
 $\text{between } k\ l \vee \text{exists_between } k\ l$.

Lemma *between_not_exists* :

$\forall k\ l,$
 $\text{between } k\ l \rightarrow$
 $(\forall n:\text{nat}, \text{in_int } k\ l\ n \rightarrow P\ n \rightarrow \neg Q\ n) \rightarrow \neg \text{exists_between } k\ l$.

Inductive *P_nth* (*init*:nat) : nat \rightarrow nat \rightarrow Prop :=

| *nth_O* : *P_nth* *init* *init* 0

| *nth_S* :

$\forall k\ l\ (n:\text{nat}),$

P_nth *init* *k* *n* \rightarrow *between* (*S* *k*) *l* \rightarrow *Q* *l* \rightarrow *P_nth* *init* *l* (*S* *n*).

Lemma *nth_le* : $\forall (init:\text{nat})\ l\ (n:\text{nat}), \text{P_nth } init\ l\ n \rightarrow init \leq l$.

Definition *eventually* (*n*:nat) := *exists2* *k* : nat, $k \leq n \ \& \ Q\ k$.

Lemma *event_O* : *eventually* 0 \rightarrow *Q* 0.

End *Between*.

Hint Resolve *nth_O* *bet_S* *bet_emp* *bet_eq* *between_Sk_l* *exists_S* *exists_le*
in_int_S *in_int_intro*: *arith*.

Hint Immediate *in_int_Sp_q* *exists_le_S* *exists_S_le*: *arith*.

Chapter 4

Library `Coq.Arith.Bool_nat`

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

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

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

Chapter 5

Library `Coq.Arith.Compare`

Equality is decidable on *nat*

`Local Open Scope nat_scope.`

`Notation not_eq_sym := not_eq_sym (only parsing).`

`Implicit Types m n p q : nat.`

`Require Import Arith_base.`

`Require Import Peano_dec.`

`Require Import Compare_dec.`

`Definition le_or_le_S := le_le_S_dec.`

`Definition Pcompare := gt_eq_gt_dec.`

`Lemma le_dec : $\forall n m, \{n \leq m\} + \{m \leq n\}$.`

`Definition lt_or_eq n m := $\{m > n\} + \{n = m\}$.`

`Lemma le_decide : $\forall n m, n \leq m \rightarrow \text{lt_or_eq } n m$.`

`Lemma le_le_S_eq : $\forall n m, n \leq m \rightarrow \text{S } n \leq m \vee n = m$.`

`Lemma discrete_nat :`

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

`Require Export Wf_nat.`

`Require Export Min Max.`

Chapter 6

Library `Coq.Arith.Compare_dec`

`Require Import Le Lt Gt Decidable PeanoNat.`

`Local Open Scope nat_scope.`

`Implicit Types m n x y : nat.`

`Definition zeroP n : {n = 0} + {0 < n}.`

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

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

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

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

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

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

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

`Theorem le_dec n m : {n ≤ m} + {¬ n ≤ m}.`

`Theorem lt_dec n m : {n < m} + {¬ n < m}.`

`Theorem gt_dec n m : {n > m} + {¬ n > m}.`

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

Proofs of decidability

`Theorem dec_le n m : decidable (n ≤ m).`

`Theorem dec_lt n m : decidable (n < m).`

`Theorem dec_gt n m : decidable (n > m).`

`Theorem dec_ge n m : decidable (n ≥ m).`

`Theorem not_eq n m : n ≠ m → n < m ∨ m < n.`

`Theorem not_le n m : ¬ n ≤ m → n > m.`

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

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

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

A ternary comparison function in the spirit of *Z.compare*. See now *Nat.compare* and its properties. In scope *nat_scope*, the notation for *Nat.compare* is “*?=*”

Notation `nat_compare` := `Nat.compare` (*compat* "8.7").

Notation `nat_compare_spec` := `Nat.compare_spec` (*compat* "8.7").

Notation `nat_compare_eq_iff` := `Nat.compare_eq_iff` (*compat* "8.7").

Notation `nat_compare_S` := `Nat.compare_succ` (*only parsing*).

Lemma `nat_compare_lt` $n\ m : n < m \leftrightarrow (n\ ? =\ m) = \text{Lt}$.

Lemma `nat_compare_gt` $n\ m : n > m \leftrightarrow (n\ ? =\ m) = \text{Gt}$.

Lemma `nat_compare_le` $n\ m : n \leq m \leftrightarrow (n\ ? =\ m) \neq \text{Gt}$.

Lemma `nat_compare_ge` $n\ m : n \geq m \leftrightarrow (n\ ? =\ m) \neq \text{Lt}$.

Some projections of the above equivalences.

Lemma `nat_compare_eq` $n\ m : (n\ ? =\ m) = \text{Eq} \rightarrow n = m$.

Lemma `nat_compare_Lt_lt` $n\ m : (n\ ? =\ m) = \text{Lt} \rightarrow n < m$.

Lemma `nat_compare_Gt_gt` $n\ m : (n\ ? =\ m) = \text{Gt} \rightarrow n > m$.

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

Definition `nat_compare_alt` ($n\ m : \text{nat}$) :=

```

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

```

Lemma `nat_compare_equiv` $n\ m : (n\ ? =\ m) = \text{nat_compare_alt}\ n\ m$.

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

Notation `leb` := `Nat.leb` (*only parsing*).

Notation `leb_iff` := `Nat.leb_le` (*only parsing*).

Lemma `leb_iff_conv` $m\ n : (n\ <=?\ m) = \text{false} \leftrightarrow m < n$.

Lemma `leb_correct` $m\ n : m \leq n \rightarrow (m\ <=?\ n) = \text{true}$.

Lemma `leb_complete` $m\ n : (m\ <=?\ n) = \text{true} \rightarrow m \leq n$.

Lemma `leb_correct_conv` $m\ n : m < n \rightarrow (n\ <=?\ m) = \text{false}$.

Lemma `leb_complete_conv` $m\ n : (n\ <=?\ m) = \text{false} \rightarrow m < n$.

Lemma `leb_compare` $n\ m : (n\ <=?\ m) = \text{true} \leftrightarrow (n\ ? =\ m) \neq \text{Gt}$.

Chapter 7

Library `Coq.Arith.Div2`

Nota : this file is OBSOLETE, and left only for compatibility. Please consider using `Nat.div2` directly, and results about it (see file `PeanoNat`).

```
Require Import PeanoNat Even.
```

```
Local Open Scope nat_scope.
```

```
Implicit Type n : nat.
```

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

```
Notation div2 := Nat.div2 (only parsing).
```

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

```
Lemma ind_0_1_SS :
```

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

```
  0 < n ⇒ 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).
```

```
Lemma odd_div2 n : odd n → S (div2 n) = div2 (S n).
```

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

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

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

```
Lemma even_odd_div2 n :
```

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

Properties related to the double $(2n)$

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

Hint `Unfold double Nat.double`: *arith*.

Lemma `double_S n` : `double (S n) = S (S (double n))`.

Lemma `double_plus n m` : `double (n + m) = double n + double m`.

Hint `Resolve double_S`: *arith*.

Lemma `even_odd_double n` :

`(even n ↔ 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` : `div2 (2×n) = n`.

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

Chapter 8

Library `Coq.Arith.EqNat`

```
Require Import PeanoNat.
```

```
Local Open Scope nat_scope.
```

Equality on natural numbers

8.1 Propositional equality

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

```
Theorem eq_nat_refl n : eq_nat n n.
```

```
Hint Resolve eq_nat_refl: arith.
```

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

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

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

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

```
Hint Immediate eq_eq_nat eq_nat_eq: arith.
```

```
Theorem eq_nat_elim :
```

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

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

8.2 Boolean equality on *nat*.

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

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

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

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

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

Lemma `beq_nat_true` n m : $(n =? m)$ = `true` $\rightarrow n=m$.

Lemma `beq_nat_false` n m : $(n =? m)$ = `false` $\rightarrow n \neq m$.

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

Definition `beq_nat_eq` : $\forall n m, \text{true} = (n =? m) \rightarrow n = m$.

Chapter 9

Library `Coq.Arith.Euclid`

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

Local Open Scope nat_scope.

Implicit Types a b n q r : nat.

Inductive diveucl a b : Set :=
  divex :  $\forall q r, b > r \rightarrow a = q \times b + r \rightarrow \text{diveucl } a b$ .

Lemma eucl_dev :  $\forall n, n > 0 \rightarrow \forall m:\text{nat}, \text{diveucl } m n$ .

Lemma quotient :
   $\forall n,$ 
   $n > 0 \rightarrow$ 
   $\forall m:\text{nat}, \{q : \text{nat} \mid \exists r : \text{nat}, m = q \times n + r \wedge n > r\}$ .

Lemma modulo :
   $\forall n,$ 
   $n > 0 \rightarrow$ 
   $\forall m:\text{nat}, \{r : \text{nat} \mid \exists q : \text{nat}, m = q \times n + r \wedge n > r\}$ .
```

Chapter 10

Library `Coq.Arith.Even`

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

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

```
Require Import PeanoNat.
```

```
Local Open Scope nat_scope.
```

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

10.1 Inductive definition of `even` and `odd`

```
Inductive even : nat → Prop :=  
  | even_O : even 0  
  | even_S : ∀ n, odd n → even (S n)  
with odd : nat → Prop :=  
  odd_S : ∀ n, even n → odd (S n).
```

```
Hint Constructors even: arith.
```

```
Hint Constructors odd: arith.
```

10.2 Equivalence with predicates `Nat.Even` and `Nat.odd`

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

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

Basic facts

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

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

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

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

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

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

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

```
Lemma even_plus_split n m :  
  even (n + m) → even n ∧ even m ∨ odd n ∧ odd m.
```

```
Lemma odd_plus_split n m :  
  odd (n + m) → odd n ∧ even m ∨ even n ∧ odd m.
```

```
Lemma even_even_plus n m : even n → even m → even (n + m).
```

```
Lemma odd_plus_l n m : odd n → even m → odd (n + m).
```

```
Lemma odd_plus_r n m : even n → odd m → odd (n + m).
```

```
Lemma odd_even_plus n m : odd n → odd m → even (n + m).
```

```
Lemma even_plus_aux n m :  
  (odd (n + m) ↔ odd n ∧ even m ∨ even n ∧ odd m) ∧  
  (even (n + m) ↔ even n ∧ even m ∨ odd n ∧ odd m).
```

```
Lemma even_plus_even_inv_r n m : even (n + m) → even n → even m.
```

```
Lemma even_plus_even_inv_l n m : even (n + m) → even m → even n.
```

```
Lemma even_plus_odd_inv_r n m : even (n + m) → odd n → odd m.
```

```
Lemma even_plus_odd_inv_l n m : even (n + m) → odd m → odd n.
```

```
Lemma odd_plus_even_inv_l n m : odd (n + m) → odd m → even n.
```

```
Lemma odd_plus_even_inv_r n m : odd (n + m) → odd n → even m.
```

```
Lemma odd_plus_odd_inv_l n m : odd (n + m) → even m → odd n.
```

```
Lemma odd_plus_odd_inv_r n m : odd (n + m) → even n → odd m.
```

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

```
Lemma even_mult_aux n m :  
  (odd (n × m) ↔ odd n ∧ odd m) ∧ (even (n × m) ↔ even n ∨ even m).
```

```
Lemma even_mult_l n m : even n → even (n × m).
```

```
Lemma even_mult_r n m : even m → even (n × m).
```

```
Lemma even_mult_inv_r n m : even (n × m) → odd n → even m.
```

```
Lemma even_mult_inv_l n m : even (n × m) → odd m → even n.
```

```
Lemma odd_mult n m : odd n → odd m → odd (n × m).
```

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

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

Hint `Resolve`

*even_even_plus odd_even_plus odd_plus_l odd_plus_r
even_mult_l even_mult_r even_mult_l even_mult_r odd_mult : arith.*

Chapter 11

Library `Coq.Arith.Factorial`

```
Require Import PeanoNat Plus Mult Lt.
```

```
Local Open Scope nat_scope.
```

```
Factorial
```

```
Fixpoint fact (n:nat) : nat :=
```

```
  match n with
```

```
    | 0 => 1
```

```
    | S n => S n × fact n
```

```
  end.
```

```
Lemma lt_0_fact n : 0 < fact n.
```

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

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

Chapter 12

Library `Coq.Arith.Gt`

Theorems about *gt* in *nat*.

This file is DEPRECATED now, see module *PeanoNat.Nat* instead, which favor *lt* over *gt*.
gt is defined in *Init/Peano.v* as:

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

Require Import `PeanoNat Le Lt Plus.`

Local Open Scope `nat_scope.`

12.1 Order and successor

Theorem `gt_Sn_O n : S n > 0.`

Theorem `gt_Sn_n n : S n > n.`

Theorem `gt_n_S n m : n > m → S n > S m.`

Lemma `gt_S_n n m : S m > S n → m > n.`

Theorem `gt_S n m : S n > m → n > m ∨ m = n.`

Lemma `gt_pred n m : m > S n → pred m > n.`

12.2 Irreflexivity

Lemma `gt_irrefl n : ¬ n > n.`

12.3 Asymmetry

Lemma `gt_asym n m : n > m → ¬ m > n.`

12.4 Relating strict and large orders

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

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

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

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

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

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

12.5 Transitivity

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

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

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

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

12.6 Comparison to 0

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

12.7 Simplification and compatibility

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

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

12.8 Hints

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

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

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

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

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

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

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

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

Chapter 13

Library `Coq.Arith.Le`

Order on natural numbers.

This file is mostly OBSOLETE now, see module *PeanoNat.Nat* instead.
le is defined in *Init/Peano.v* as:

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

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

`Require Import PeanoNat.`

`Local Open Scope nat_scope.`

13.1 *le* is an order on *nat*

`Notation le_refl` := `Nat.le_refl` (*only parsing*).

`Notation le_trans` := `Nat.le_trans` (*only parsing*).

`Notation le_antisym` := `Nat.le_antisymm` (*only parsing*).

`Hint Resolve le_trans`: *arith*.

`Hint Immediate le_antisym`: *arith*.

13.2 Properties of *le* w.r.t 0

`Notation le_0_n` := `Nat.le_0_1` (*only parsing*). `Notation le_Sn_0` := `Nat.nle_succ_0` (*only parsing*).

`Lemma le_n_0_eq` *n* : $n \leq 0 \rightarrow 0 = n$.

13.3 Properties of *le* w.r.t successor

See also *Nat.succ_le_mono*.

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

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

Notation `le_n_Sn` := `Nat.le_succ_diag_r` (*only parsing*). **Notation** `le_Sn_n` := `Nat.nle_succ_diag_l` (*only parsing*).

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

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

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

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

13.4 Properties of *le* w.r.t predecessor

Notation `le_pred_n` := `Nat.le_pred_l` (*only parsing*). **Notation** `le_pred` := `Nat.pred_le_mono` (*only parsing*).

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

13.5 A different elimination principle for the order on natural numbers

Lemma `le_elim_rel` :

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

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

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

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

Chapter 14

Library `Coq.Arith.Lt`

Strict order on natural numbers.

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

`lt` is defined in library `Init/Peano.v` as:

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

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

```
Require Import PeanoNat.
```

```
Local Open Scope nat_scope.
```

14.1 Irreflexivity

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

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

14.2 Relationship between `le` and `lt`

```
Theorem lt_le_S n m : n < m → S n ≤ m.
```

```
Theorem lt_n_Sm_le n m : n < S m → n ≤ m.
```

```
Theorem le_lt_n_Sm n m : n ≤ m → n < S m.
```

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

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

```
Hint Immediate le_lt_n_Sm: arith.
```

```
Theorem le_not_lt n m : n ≤ m → ¬ m < n.
```

```
Theorem lt_not_le n m : n < m → ¬ m ≤ n.
```

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

14.3 Asymmetry

Notation `lt_asym` := `Nat.lt_asymm` (*only parsing*).

14.4 Order and 0

Notation `lt_0_Sn` := `Nat.lt_0_succ` (*only parsing*). **Notation** `lt_n_0` := `Nat.nlt_0_r` (*only parsing*).

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

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

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

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

14.5 Order and successor

Notation `lt_n_Sn` := `Nat.lt_succ_diag_r` (*only parsing*). **Notation** `lt_S` := `Nat.lt_lt_succ_r` (*only parsing*).

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

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

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

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

14.6 Predecessor

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

Lemma `S_pred_pos` $n : 0 < n \rightarrow n = S\ (\text{pred } n)$.

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

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

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

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

14.7 Transitivity properties

Notation `lt_trans` := `Nat.lt_trans` (*only parsing*).

Notation `lt_le_trans` := `Nat.lt_le_trans` (*only parsing*).

Notation `le_lt_trans` := `Nat.le_lt_trans` (*only parsing*).

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

14.8 Large = strict or equal

Notation `le_lt_or_eq_iff` := `Nat.lt_eq_cases` (*only parsing*).

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

Notation `lt_le_weak` := `Nat.lt_le_incl` (*only parsing*).

Hint Immediate `lt_le_weak`: *arith*.

14.9 Dichotomy

Notation `le_or_lt` := `Nat.le_gt_cases` (*only parsing*).

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

For compatibility, we “Require” the same files as before

Require Import `Le`.

Chapter 15

Library `Coq.Arith.Max`

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

```
Require Import PeanoNat.
```

```
Local Open Scope nat_scope.
```

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

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

```
Definition max_0_l := Nat.max_0_l.
```

```
Definition max_0_r := Nat.max_0_r.
```

```
Definition succ_max_distr := Nat.succ_max_distr.
```

```
Definition plus_max_distr_l := Nat.add_max_distr_l.
```

```
Definition plus_max_distr_r := Nat.add_max_distr_r.
```

```
Definition max_case_strong := Nat.max_case_strong.
```

```
Definition max_spec := Nat.max_spec.
```

```
Definition max_dec := Nat.max_dec.
```

```
Definition max_case := Nat.max_case.
```

```
Definition max_idempotent := Nat.max_id.
```

```
Definition max_assoc := Nat.max_assoc.
```

```
Definition max_comm := Nat.max_comm.
```

```
Definition max_l := Nat.max_l.
```

```
Definition max_r := Nat.max_r.
```

```
Definition le_max_l := Nat.le_max_l.
```

```
Definition le_max_r := Nat.le_max_r.
```

```
Definition max_lub_l := Nat.max_lub_l.
```

```
Definition max_lub_r := Nat.max_lub_r.
```

```
Definition max_lub := Nat.max_lub.
```

```
Hint Resolve
```

```
  Nat.max_l Nat.max_r Nat.le_max_l Nat.le_max_r : arith.
```

```
Hint Resolve
```

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

Chapter 16

Library `Coq.Arith.Min`

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

```
Require Import PeanoNat.
```

```
Local Open Scope nat_scope.
```

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

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

```
Definition min_0_l := Nat.min_0_l.
```

```
Definition min_0_r := Nat.min_0_r.
```

```
Definition succ_min_distr := Nat.succ_min_distr.
```

```
Definition plus_min_distr_l := Nat.add_min_distr_l.
```

```
Definition plus_min_distr_r := Nat.add_min_distr_r.
```

```
Definition min_case_strong := Nat.min_case_strong.
```

```
Definition min_spec := Nat.min_spec.
```

```
Definition min_dec := Nat.min_dec.
```

```
Definition min_case := Nat.min_case.
```

```
Definition min_idempotent := Nat.min_id.
```

```
Definition min_assoc := Nat.min_assoc.
```

```
Definition min_comm := Nat.min_comm.
```

```
Definition min_l := Nat.min_l.
```

```
Definition min_r := Nat.min_r.
```

```
Definition le_min_l := Nat.le_min_l.
```

```
Definition le_min_r := Nat.le_min_r.
```

```
Definition min_glb_l := Nat.min_glb_l.
```

```
Definition min_glb_r := Nat.min_glb_r.
```

```
Definition min_glb := Nat.min_glb.
```

Chapter 17

Library `Coq.Arith.Minus`

Properties of subtraction between natural numbers.

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

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

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

`Require Import PeanoNat Lt Le.`

`Local Open Scope nat_scope.`

17.1 0 is right neutral

`Lemma minus_n_0 n : n = n - 0.`

17.2 Permutation with successor

`Lemma minus_Sn_m n m : m ≤ n → S (n - m) = S n - m.`

`Theorem pred_of_minus n : pred n = n - 1.`

17.3 Diagonal

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

`Lemma minus_diag_reverse n : 0 = n - n.`

`Notation minus_n_n := minus_diag_reverse.`

17.4 Simplification

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

17.5 Relation with plus

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

Lemma `minus_plus` $n\ m : n + m - n = m$.

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

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

17.6 Relation with order

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

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

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

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

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

17.7 Hints

Hint `Resolve` `minus_n_0`: *arith*.

Hint `Resolve` `minus_Sn_m`: *arith*.

Hint `Resolve` `minus_diag_reverse`: *arith*.

Hint `Resolve` `minus_plus_simpl_l_reverse`: *arith*.

Hint `Immediate` `plus_minus`: *arith*.

Hint `Resolve` `minus_plus`: *arith*.

Hint `Resolve` `le_plus_minus`: *arith*.

Hint `Resolve` `le_plus_minus_r`: *arith*.

Hint `Resolve` `lt_minus`: *arith*.

Hint `Immediate` `lt_O_minus_lt`: *arith*.

Chapter 18

Library `Coq.Arith.Mult`

18.1 Properties of multiplication.

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

`Nat.mul` is defined in `Init/Nat.v`.

`Require Import` `PeanoNat`.

For Compatibility: `Require Export` `Plus Minus Le Lt`.

`Local Open Scope` `nat_scope`.

18.2 `nat` is a semi-ring

18.2.1 Zero property

`Notation` `mult_0_l` := `Nat.mul_0_l` (*only parsing*). `Notation` `mult_0_r` := `Nat.mul_0_r` (*only parsing*).

18.2.2 1 is neutral

`Notation` `mult_1_l` := `Nat.mul_1_l` (*only parsing*). `Notation` `mult_1_r` := `Nat.mul_1_r` (*only parsing*).

`Hint Resolve` `mult_1_l` `mult_1_r`: *arith*.

18.2.3 Commutativity

`Notation` `mult_comm` := `Nat.mul_comm` (*only parsing*).

`Hint Resolve` `mult_comm`: *arith*.

18.2.4 Distributivity

`Notation` `mult_plus_distr_r` :=
`Nat.mul_add_distr_r` (*only parsing*).

Notation `mult_plus_distr_l` :=
 `Nat.mul_add_distr_l` (*only parsing*).

Notation `mult_minus_distr_r` :=
 `Nat.mul_sub_distr_r` (*only parsing*).

Notation `mult_minus_distr_l` :=
 `Nat.mul_sub_distr_l` (*only parsing*).

Hint Resolve `mult_plus_distr_r`: *arith*.

Hint Resolve `mult_minus_distr_r`: *arith*.

Hint Resolve `mult_minus_distr_l`: *arith*.

18.2.5 Associativity

Notation `mult_assoc` := `Nat.mul_assoc` (*only parsing*).

Lemma `mult_assoc_reverse` $n\ m\ p$: $n \times m \times p = n \times (m \times p)$.

Hint Resolve `mult_assoc_reverse`: *arith*.

Hint Resolve `mult_assoc`: *arith*.

18.2.6 Inversion lemmas

Lemma `mult_is_O` $n\ m$: $n \times m = 0 \rightarrow n = 0 \vee m = 0$.

Lemma `mult_is_one` $n\ m$: $n \times m = 1 \rightarrow n = 1 \wedge m = 1$.

18.2.7 Multiplication and successor

Notation `mult_succ_l` := `Nat.mul_succ_l` (*only parsing*). **Notation** `mult_succ_r` := `Nat.mul_succ_r` (*only parsing*).

18.3 Compatibility with orders

Lemma `mult_O_le` $n\ m$: $m = 0 \vee n \leq m \times n$.

Hint Resolve `mult_O_le`: *arith*.

Lemma `mult_le_compat_l` $n\ m\ p$: $n \leq m \rightarrow p \times n \leq p \times m$.

Hint Resolve `mult_le_compat_l`: *arith*.

Lemma `mult_le_compat_r` $n\ m\ p$: $n \leq m \rightarrow n \times p \leq m \times p$.

Lemma `mult_le_compat` $n\ m\ p\ q$: $n \leq m \rightarrow p \leq q \rightarrow n \times p \leq m \times q$.

Lemma `mult_S_lt_compat_l` $n\ m\ p$: $m < p \rightarrow \mathbf{S}\ n \times m < \mathbf{S}\ n \times p$.

Hint Resolve `mult_S_lt_compat_l`: *arith*.

Lemma `mult_lt_compat_l` $n\ m\ p$: $n < m \rightarrow 0 < p \rightarrow p \times n < p \times m$.

Lemma `mult_lt_compat_r` $n\ m\ p$: $n < m \rightarrow 0 < p \rightarrow n \times p < m \times p$.

Lemma `mult_S_le_reg_l` $n\ m\ p$: $\mathbf{S}\ n \times m \leq \mathbf{S}\ n \times p \rightarrow m \leq p$.

18.4 $n \mapsto 2 \cdot n$ and $n \mapsto 2n + 1$ have disjoint image

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

18.5 Tail-recursive mult

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

```
Fixpoint mult_acc (s:nat) m n : nat :=  
  match n with  
  | 0 => s  
  | S p => mult_acc (tail_plus m s) m p  
  end.
```

Lemma `mult_acc_aux` : $\forall n\ m\ p, m + n \times p = \text{mult_acc}\ m\ p\ n$.

Definition `tail_mult` $n\ m := \text{mult_acc}\ 0\ m\ n$.

Lemma `mult_tail_mult` : $\forall n\ m, n \times m = \text{tail_mult}\ n\ m$.

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

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

Chapter 19

Library `Coq.Arith.PeanoNat`

`Require Import` `NAxioms` `NProperties` `OrdersFacts`.

Implementation of *NAxiomsSig* by *nat*

`Module` `NAT`

<: `NAXIOMSIG`

<: `USUALDECIDABLETYPEFULL`

<: `ORDEREDTYPEFULL`

<: `TOTALORDER`.

Operations over *nat* are defined in a separate module

`Include` `COQ.INIT.NAT`.

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

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

`Definition` `eq_equiv` : `Equivalence` (`@eq nat`) := `eq_equivalence`.

`Program Instance` `succ_wd` : `Proper` (`eq==>eq`) `S`.

`Program Instance` `pred_wd` : `Proper` (`eq==>eq`) `pred`.

`Program Instance` `add_wd` : `Proper` (`eq==>eq==>eq`) `plus`.

`Program Instance` `sub_wd` : `Proper` (`eq==>eq==>eq`) `minus`.

`Program Instance` `mul_wd` : `Proper` (`eq==>eq==>eq`) `mult`.

`Program Instance` `pow_wd` : `Proper` (`eq==>eq==>eq`) `pow`.

`Program Instance` `div_wd` : `Proper` (`eq==>eq==>eq`) `div`.

`Program Instance` `mod_wd` : `Proper` (`eq==>eq==>eq`) `modulo`.

`Program Instance` `lt_wd` : `Proper` (`eq==>eq==>iff`) `lt`.

`Program Instance` `testbit_wd` : `Proper` (`eq==>eq==>eq`) `testbit`.

Bi-directional induction.

`Theorem` `bi_induction` :

$\forall A : \text{nat} \rightarrow \text{Prop}, \text{Proper } (\text{eq}==>\text{iff}) A \rightarrow$

$A\ 0 \rightarrow (\forall n : \text{nat}, A\ n \leftrightarrow A\ (\text{S } n)) \rightarrow \forall n : \text{nat}, A\ n.$

Recursion fonction

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

`nat_rect (fun _ => A).`
Instance `recursion_wd {A} (Aeq : relation A) :`
`Proper (Aeq ==> (eq==>Aeq==>Aeq) ==> eq ==> Aeq) recursion.`
Theorem `recursion_0 :`
 $\forall \{A\} (a : A) (f : \text{nat} \rightarrow A \rightarrow A), \text{recursion } a \ f \ 0 = a.$
Theorem `recursion_succ :`
 $\forall \{A\} (Aeq : \text{relation } A) (a : A) (f : \text{nat} \rightarrow A \rightarrow A),$
 $Aeq \ a \ a \rightarrow \text{Proper } (eq==>Aeq==>Aeq) \ f \rightarrow$
 $\forall n : \text{nat}, Aeq (\text{recursion } a \ f \ (\text{S } n)) (f \ n (\text{recursion } a \ f \ n)).$

19.0.1 Remaining constants not defined in Coq.Init.Nat

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

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

19.0.2 Basic specifications : pred add sub mul

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

19.0.3 Boolean comparisons

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

19.0.4 Decidability of equality over *nat*.

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

19.0.5 Ternary comparison

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

Lemma *compare_eq_iff* $n\ m : (n\ ?=\ m) = \text{Eq} \leftrightarrow n = m$.

Lemma *compare_lt_iff* $n\ m : (n\ ?=\ m) = \text{Lt} \leftrightarrow n < m$.

Lemma *compare_le_iff* $n\ m : (n\ ?=\ m) \neq \text{Gt} \leftrightarrow n \leq m$.

Lemma *compare_antisym* $n\ m : (m\ ?=\ n) = \text{CompOpp}\ (n\ ?=\ m)$.

Lemma *compare_succ* $n\ m : (\text{S}\ n\ ?=\ \text{S}\ m) = (n\ ?=\ m)$.

19.0.6 Minimum, maximum

Lemma *max_l* $\forall n\ m, m \leq n \rightarrow \text{max}\ n\ m = n$.

Lemma *max_r* $\forall n\ m, n \leq m \rightarrow \text{max}\ n\ m = m$.

Lemma *min_l* $\forall n\ m, n \leq m \rightarrow \text{min}\ n\ m = n$.

Lemma *min_r* $\forall n\ m, m \leq n \rightarrow \text{min}\ n\ m = m$.

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

Include `BOOLORDERFACTS`.

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

Include `NBASICPROP <+ USUALMINMAXLOGICALPROPERTIES <+ USUALMINMAXDECPROPERTIES`.

19.0.7 Power

Lemma *pow_neg_r* $a\ b : b < 0 \rightarrow a^b = 0$.

Lemma *pow_0_r* $a : a^0 = 1$.

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

19.0.8 Square

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

19.0.9 Parity

Definition *Even* $n := \exists m, n = 2 \times m$.

Definition *Odd* $n := \exists m, n = 2 \times m + 1$.

Module `PRIVATE_PARITY`.

Lemma *Even_1* $:\neg \text{Even}\ 1$.

Lemma *Even_2* $n : \text{Even}\ n \leftrightarrow \text{Even}\ (\text{S}\ (\text{S}\ n))$.

Lemma Odd_0 : \neg Odd 0.

Lemma Odd_2 n : Odd n \leftrightarrow Odd (S (S n)).

End PRIVATE_PARITY.

Import Private_Parity.

Lemma even_spec : \forall n, even n = true \leftrightarrow Even n.

Lemma odd_spec : \forall n, odd n = true \leftrightarrow Odd n.

19.0.10 Division

Lemma divmod_spec : \forall x y q u, $u \leq y \rightarrow$
let (q', u') := divmod x y q u in
 $x + (S y)*q + (y-u) = (S y)*q' + (y-u') \wedge u' \leq y$.

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

Lemma mod_bound_pos x y : $0 \leq x \rightarrow 0 < y \rightarrow 0 \leq x \bmod y < y$.

19.0.11 Square root

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

Lemma sqrt_specif n : $(\text{sqrt } n) * (\text{sqrt } n) \leq n < S (\text{sqrt } n) \times S (\text{sqrt } n)$.

Definition sqrt_spec a (Ha: $0 \leq a$) := sqrt_specif a.

Lemma sqrt_neg a : $a < 0 \rightarrow \text{sqrt } a = 0$.

19.0.12 Logarithm

Lemma log2_iter_spec : \forall k p q r,
 $2^{(S p)} = q + S r \rightarrow r < 2^p \rightarrow$
let s := log2_iter k p q r in
 $2^s \leq k + q < 2^{(S s)}$.

Lemma log2_spec n : $0 < n \rightarrow$
 $2^{(\text{log2 } n)} \leq n < 2^{(S (\text{log2 } n))}$.

Lemma log2_nonpos n : $n \leq 0 \rightarrow \text{log2 } n = 0$.

19.0.13 Gcd

Definition divide x y := $\exists z, y = z \times x$.

Notation "(x | y)" := (divide x y) (at level 0) : nat_scope.

Lemma gcd_divide : $\forall a b, (\text{gcd } a b \mid a) \wedge (\text{gcd } a b \mid b)$.

Lemma gcd_divide_l : $\forall a b, (\text{gcd } a b \mid a)$.

Lemma gcd_divide_r : $\forall a b, (\text{gcd } a b \mid b)$.

Lemma gcd_greatest : $\forall a b c, (c \mid a) \rightarrow (c \mid b) \rightarrow (c \mid \text{gcd } a b)$.

Lemma gcd_nonneg a b : $0 \leq \text{gcd } a b$.

19.0.14 Bitwise operations

Lemma div2_double n : $\text{div2 } (2 \times n) = n$.

Lemma div2_succ_double n : $\text{div2 } (\text{S } (2 \times n)) = n$.

Lemma le_div2 n : $\text{div2 } (\text{S } n) \leq n$.

Lemma lt_div2 n : $0 < n \rightarrow \text{div2 } n < n$.

Lemma div2_decr a n : $a \leq \text{S } n \rightarrow \text{div2 } a \leq n$.

Lemma double_twice : $\forall n, \text{double } n = 2 \times n$.

Lemma testbit_0_1 : $\forall n, \text{testbit } 0 n = \text{false}$.

Lemma testbit_odd_0 a : $\text{testbit } (2 \times a + 1) 0 = \text{true}$.

Lemma testbit_even_0 a : $\text{testbit } (2 \times a) 0 = \text{false}$.

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

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

Lemma shiftr_specif : $\forall a n m,$
 $\text{testbit } (\text{shiftr } a n) m = \text{testbit } a (m+n)$.

Lemma shiftl_specif_high : $\forall a n m, n \leq m \rightarrow$
 $\text{testbit } (\text{shiftl } a n) m = \text{testbit } a (m-n)$.

Lemma shiftl_spec_low : $\forall a n m, m < n \rightarrow$
 $\text{testbit } (\text{shiftl } a n) m = \text{false}$.

Lemma div2_bitwise : $\forall op n a b,$
 $\text{div2 } (\text{bitwise } op (\text{S } n) a b) = \text{bitwise } op n (\text{div2 } a) (\text{div2 } b)$.

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

Lemma testbit_bitwise_1 : $\forall op, (\forall b, op \text{ false } b = \text{false}) \rightarrow$
 $\forall n m a b, a \leq n \rightarrow$
 $\text{testbit } (\text{bitwise } op n a b) m = op (\text{testbit } a m) (\text{testbit } b m)$.

Lemma testbit_bitwise_2 : $\forall op, op \text{ false } \text{false} = \text{false} \rightarrow$
 $\forall n m a b, a \leq n \rightarrow b \leq n \rightarrow$
 $\text{testbit } (\text{bitwise } op n a b) m = op (\text{testbit } a m) (\text{testbit } b m)$.

Lemma land_spec a b n :
 $\text{testbit } (\text{land } a b) n = \text{testbit } a n \ \&\& \ \text{testbit } b n$.

Lemma ldiff_spec a b n :
 $\text{testbit } (\text{ldiff } a b) n = \text{testbit } a n \ \&\& \ \text{negb } (\text{testbit } b n)$.

Lemma lor_spec a b n :
 $\text{testbit } (\text{lor } a b) n = \text{testbit } a n \ \|\| \ \text{testbit } b n$.

Lemma `lxor_spec a b n` :
testbit (lxor a b) n = xorb (testbit a n) (testbit b n).

Lemma `div2_spec a` : div2 a = shiftr a 1.

Aliases with extra dummy hypothesis, to fulfil the interface

Definition `testbit_odd_succ a n` ($_:0 \leq n$) := testbit_odd_succ' a n.

Definition `testbit_even_succ a n` ($_:0 \leq n$) := testbit_even_succ' a n.

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

Definition `shiftr_spec_high a n m` ($_:0 \leq m$) := shiftr_specif_high a n m.

Definition `shiftr_spec a n m` ($_:0 \leq m$) := shiftr_specif a n m.

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

Include `NEXTRAPROP`.

Properties of tail-recursive addition and multiplication

Lemma `tail_add_spec n m` : tail_add n m = n + m.

Lemma `tail_addmul_spec r n m` : tail_addmul r n m = r + n × m.

Lemma `tail_mul_spec n m` : tail_mul n m = n × m.

End `NAT`.

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

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

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

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

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

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

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

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

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

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

`Nat` contains an *order* tactic for natural numbers

Note that `Nat.order` is domain-agnostic: it will not prove $1 \leq 2$ or $x \leq x+x$, but rather things like $x \leq y \rightarrow y \leq x \rightarrow x=y$.

Section `TestOrder`.

Let *test* : $\forall x y, x \leq y \rightarrow y \leq x \rightarrow x=y$.

End `TestOrder`.

Chapter 20

Library `Coq.Arith.Peano_dec`

```
Require Import Decidable PeanoNat.
Require Eqdep_dec.
Local Open Scope nat_scope.
Implicit Types m n x y : nat.
Theorem O_or_S n : {m : nat | S m = n} + {0 = n}.
Notation eq_nat_dec := Nat.eq_dec (only parsing).
Hint Resolve O_or_S eq_nat_dec: arith.
Theorem dec_eq_nat n m : decidable (n = m).
Definition UIP_nat := Eqdep_dec.UIP_dec Nat.eq_dec.
Import EqNotations.
Lemma le_unique:  $\forall m n (le\_mn1 le\_mn2 : m \leq n), le\_mn1 = le\_mn2$ .
  For compatibility Require Import Le Lt.
```

Chapter 21

Library **Coq.Arith.Plus**

Properties of addition.

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

Nat.add is defined in *Init/Nat.v* as:

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

Require Import PeanoNat.

Local Open Scope *nat_scope*.

21.1 Neutrality of 0, commutativity, associativity

Notation *plus_0_l* := *Nat.add_0_l* (*only parsing*).

Notation *plus_0_r* := *Nat.add_0_r* (*only parsing*).

Notation *plus_comm* := *Nat.add_comm* (*only parsing*).

Notation *plus_assoc* := *Nat.add_assoc* (*only parsing*).

Notation *plus_permute* := *Nat.add_shuffle3* (*only parsing*).

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

Lemma *plus_assoc_reverse* *n m p* : $n + m + p = n + (m + p)$.

21.2 Simplification

Lemma *plus_reg_l* *n m p* : $p + n = p + m \rightarrow n = m$.

Lemma *plus_le_reg_l* *n m p* : $p + n \leq p + m \rightarrow n \leq m$.

Lemma *plus_lt_reg_l* *n m p* : $p + n < p + m \rightarrow n < m$.

21.3 Compatibility with order

Lemma `plus_le_compat_l` $n\ m\ p : n \leq m \rightarrow p + n \leq p + m$.

Lemma `plus_le_compat_r` $n\ m\ p : n \leq m \rightarrow n + p \leq m + p$.

Lemma `plus_lt_compat_l` $n\ m\ p : n < m \rightarrow p + n < p + m$.

Lemma `plus_lt_compat_r` $n\ m\ p : n < m \rightarrow n + p < m + p$.

Lemma `plus_le_compat` $n\ m\ p\ q : n \leq m \rightarrow p \leq q \rightarrow n + p \leq m + q$.

Lemma `plus_le_lt_compat` $n\ m\ p\ q : n \leq m \rightarrow p < q \rightarrow n + p < m + q$.

Lemma `plus_lt_le_compat` $n\ m\ p\ q : n < m \rightarrow p \leq q \rightarrow n + p < m + q$.

Lemma `plus_lt_compat` $n\ m\ p\ q : n < m \rightarrow p < q \rightarrow n + p < m + q$.

Lemma `le_plus_l` $n\ m : n \leq n + m$.

Lemma `le_plus_r` $n\ m : m \leq n + m$.

Theorem `le_plus_trans` $n\ m\ p : n \leq m \rightarrow n \leq m + p$.

Theorem `lt_plus_trans` $n\ m\ p : n < m \rightarrow n < m + p$.

21.4 Inversion lemmas

Lemma `plus_is_0` $n\ m : n + m = 0 \rightarrow n = 0 \wedge m = 0$.

Definition `plus_is_one` $m\ n :$

$m + n = 1 \rightarrow \{m = 0 \wedge n = 1\} + \{m = 1 \wedge n = 0\}$.

21.5 Derived properties

Notation `plus_permute_2_in_4` := `Nat.add_shuffle1` (*only parsing*).

21.6 Tail-recursive plus

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

Fixpoint `tail_plus` $n\ m : \text{nat} :=$

```
match n with
| 0 => m
| S n => tail_plus n (S m)
end.
```

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

21.7 Discrimination

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

`Lemma n_SS n : n ≠ S (S n).`

`Lemma n_SSS n : n ≠ S (S (S n)).`

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

21.8 Compatibility Hints

`Hint Immediate plus_comm : arith.`

`Hint Resolve plus_assoc plus_assoc_reverse : arith.`

`Hint Resolve plus_le_compat_l plus_le_compat_r : arith.`

`Hint Resolve le_plus_l le_plus_r le_plus_trans : arith.`

`Hint Immediate lt_plus_trans : arith.`

`Hint Resolve plus_lt_compat_l plus_lt_compat_r : arith.`

For compatibility, we “Require” the same files as before

`Require Import Le Lt.`

Chapter 22

Library `Coq.Arith.Wf_nat`

Well-founded relations and natural numbers

```
Require Import PeanoNat Lt.
```

```
Local Open Scope nat_scope.
```

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

```
Section Well_founded_Nat.
```

```
Variable A : Type.
```

```
Variable f : A → nat.
```

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

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

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

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

It is possible to directly prove the induction principle going back to primitive recursion on natural numbers (*induction_ltof1*) or to use the previous lemmas to extract a program with a fixpoint (*induction_ltof2*)

the ML-like program for *induction_ltof1* is :

```
let induction_ltof1 f F a =  
  let rec indrec n k =  
    match n with  
    | 0 → error  
    | S m → F k (indrec m)  
  in indrec (f a + 1) a
```

the ML-like program for *induction_ltof2* is :

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

```
Theorem induction_ltof1 :
```

```
  ∀ P:A → Set,
```

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

```
Theorem induction_gtof1 :
```

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

Theorem `induction_ltof2` :

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

Theorem `induction_gtof2` :

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

If a relation R is compatible with lt i.e. if $x \ R \ y \Rightarrow f(x) < f(y)$ then R is well-founded.

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

Hypothesis $H_compat : \forall x \ y:A, R \ x \ y \rightarrow f \ x < f \ y.$

Theorem `well_founded_lt_compat` : `well_founded` $R.$

End `Well_founded_Nat.`

Lemma `lt_wf` : `well_founded` $lt.$

Lemma `lt_wf_rec1` :

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

Lemma `lt_wf_rec` :

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

Lemma `lt_wf_ind` :

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

Lemma `gt_wf_rec` :

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

Lemma `gt_wf_ind` :

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

Lemma `lt_wf_double_rec` :

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

Lemma `lt_wf_double_ind` :

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

Hint `Resolve` lt_wf : *arith*.

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

Section `LT_WF_REL.`

Variable $A : \text{Set}.$

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

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

Definition `inv_lt_rel` $x\ y := \text{exists2 } n, F\ x\ n \ \& \ (\forall\ m, F\ y\ m \rightarrow n < m)$.
Hypothesis `F_compat` : $\forall\ x\ y:A, R\ x\ y \rightarrow \text{inv_lt_rel } x\ y$.
Remark `acc_lt_rel` : $\forall\ x:A, (\exists\ n, F\ x\ n) \rightarrow \text{Acc } R\ x$.
Theorem `well_founded_inv_lt_rel_compat` : `well_founded` R .
End `LT_WF_REL`.
Lemma `well_founded_inv_rel_inv_lt_rel` :
 $\forall\ (A:\text{Set})\ (F:A \rightarrow \text{nat} \rightarrow \text{Prop}), \text{well_founded } (\text{inv_lt_rel } A\ F)$.
A constructive proof that any non empty decidable subset of natural numbers has a least element
Set Implicit Arguments.
Require Import `Le`.
Require Import `Compare_dec`.
Require Import `Decidable`.
Definition `has_unique_least_element` $(A:\text{Type})\ (R:A \rightarrow A \rightarrow \text{Prop})\ (P:A \rightarrow \text{Prop}) :=$
 $\exists! x, P\ x \wedge \forall\ x', P\ x' \rightarrow R\ x\ x'$.
Lemma `dec_inh_nat_subset_has_unique_least_element` :
 $\forall\ P:\text{nat} \rightarrow \text{Prop}, (\forall\ n, P\ n \vee \neg P\ n) \rightarrow$
 $(\exists\ n, P\ n) \rightarrow \text{has_unique_least_element } \text{le } P$.
Unset Implicit Arguments.
Notation `iter_nat` $n\ A\ f\ x := (\text{nat_rect } (\text{fun } _ \Rightarrow A)\ x\ (\text{fun } _ \Rightarrow f)\ n)$ (*only parsing*).

Chapter 23

Library `Coq.Init.Datatypes`

`Set Implicit Arguments.`

`Require Import Notations.`

`Require Import Logic.`

23.1 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.`

23.2 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.`

23.3 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 O)`

Inductive `nat : Set :=`

```

| O : nat
| S : nat → nat.

```

Delimit Scope *nat_scope* with *nat*.

23.4 Container datatypes

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

```

Inductive option (A:Type) : Type :=
| Some : A → option A
| None : option A.

```

```

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

```

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

```

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

```

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

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

```

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

```

Add Printing Let *prod*.

Notation " $x * y$ " := (*prod* $x y$) : *type_scope*.

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

Section projections.

```

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

```

```

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

```

```

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

```

End projections.

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

Lemma *surjective_pairing* :

```

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

```

Lemma `injective_projections` :

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

Definition `prod_uncurry` (A B C:Type) (f:prod A B → C)

```
(x:A) (y:B) : C := f (pair x y).
```

Definition `prod_curry` (A B C:Type) (f:A → B → C)

```
(p:prod A B) : C := match p with
  | pair x y ⇒ f x y
end.
```

Polymorphic lists and some operations

Inductive `list` (A : Type) : Type :=

```
| nil : list A
| cons : A → list A → list A.
```

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

Delimit Scope `list_scope` with `list`.

Local Open Scope `list_scope`.

Definition `length` (A : Type) : list A → nat :=

```
  fix length l :=
  match l with
  | nil ⇒ 0
  | _ :: l' ⇒ S (length l')
end.
```

Concatenation of two lists

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

```
  fix app l m :=
  match l with
  | nil ⇒ m
  | a :: l1 ⇒ a :: app l1 m
end.
```

Infix `++` := `app` (right associativity, at level 60) : `list_scope`.

23.5 The comparison datatype

Inductive `comparison` : Set :=

```
| Eq : comparison
| Lt : comparison
| Gt : comparison.
```

Lemma `comparison_eq_stable` : ∀ c c' : comparison, $\sim\sim c = c' \rightarrow c = c'$.

Definition `CompOpp` (r:comparison) :=

```
  match r with
  | Eq ⇒ Eq
```

```

| Lt ⇒ Gt
| Gt ⇒ Lt
end.

```

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

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

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

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

```

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

```

Hint Constructors `CompareSpec`.

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

```

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

```

Hint Constructors `CompareSpecT`.

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

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

```

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

```

```

Definition CompSpecT {A} (eq lt : A → A → Prop)(x y : A) : comparison → Type :=
  CompareSpecT (eq x y) (lt x y) (lt y x).

```

Hint Unfold `CompSpec` `CompSpecT`.

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

23.6 Misc Other Datatypes

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

```

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

```

Hint `Resolve identity_refl`: *core*.

Identity type

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

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

Definition `IDProp` := $\forall A:\text{Prop}, A \rightarrow A$.

Definition `idProp` : `IDProp` := `fun A x => x`.

Chapter 24

Library `Coq.Init.Decimal`

24.1 Decimal numbers

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

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

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

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

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

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

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

Delimit Scope `dec_uint_scope` with `uint`.

Delimit Scope `dec_int_scope` with `int`.

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

nzhead removes all head zero digits

```
Fixpoint nzhead d :=
```

```

match d with
| D0 d ⇒ nzhead d
| _ ⇒ d
end.

```

unorm : normalization of unsigned integers

```

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

```

norm : normalization of signed integers

```

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

```

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

```

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

```

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

```

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

```

```

Definition rev d := revapp d Nil.

```

Module LITTLE.

Successor of little-endian numbers

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

Doubling little-endian numbers

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

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

```
| D9 d ⇒ D9 (succ_double d)
end.
```

End LITTLE.

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

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

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

Chapter 25

Library `Coq.Init.Logic`

`Set Implicit Arguments.`

`Require Export Notations.`

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

25.1 Propositional connectives

`True` is the always true proposition

`Inductive True : Prop :=`

`! : True.`

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

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

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

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

`Hint Unfold not: core.`

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

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

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

`Inductive and (A B:Prop) : Prop :=`

`conj : A \rightarrow B \rightarrow A \wedge B`

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

`Section Conjunction.`

`Variables A B : Prop.`

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

`Theorem proj2 : A \wedge B \rightarrow B.`

`End Conjunction.`

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

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

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

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

```
Definition iff (A B:Prop) := (A → 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*.

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).
```

```
Theorem imp_iff_compat_l : ∀ A B C : Prop,
(B ↔ C) → ((A → B) ↔ (A → C)).
```

```
Theorem imp_iff_compat_r : ∀ A B C : Prop,
(B ↔ C) → ((B → A) ↔ (C → A)).
```

```
Theorem not_iff_compat : ∀ A B : Prop,
(A ↔ B) → (¬ A ↔ ¬ B).
```

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

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

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

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

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

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

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

Definition `IF_then_else (P Q R:Prop) := P \wedge Q \vee \neg P \wedge R.`

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

25.2 First-order quantifiers

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

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

Universal quantification is primitively written $\forall x:A, Q$. By symmetry with existential quantification, the construction `all P` is provided too.

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

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

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

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

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

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

Notation "'exists2' ' x , p & q" := (`ex2 (fun x \Rightarrow p) (fun x \Rightarrow q)`)
(at level 200, *x strict pattern*, *p* at level 200, right associativity) : `type_scope`.

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

Derived rules for universal quantification

Section universal_quantification.

Variable $A : \text{Type}$.

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

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

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

End universal_quantification.

25.3 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 ($A:\text{Type}$) ($x:A$) : $A \rightarrow \text{Prop} :=$

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" := ($\neg x = y :> T$) : type_scope.

Notation "x <> y" := (x \neq y :>_) : type_scope.

Hint Resolve I conj or_introl or_intror : core.

Hint Resolve eq_refl: core.

Hint Resolve ex_intro ex_intro2: core.

Section Logic_lemmas.

Theorem absurd : $\forall A C:\text{Prop}, A \rightarrow \neg A \rightarrow C$.

Section equality.

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

Variable $f : A \rightarrow B$.

Variables $x y z : A$.

Theorem eq_sym : $x = y \rightarrow y = x$.

Theorem eq_trans : $x = y \rightarrow y = z \rightarrow x = z$.

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

Theorem not_eq_sym : $x \neq y \rightarrow y \neq x$.

End equality.

Definition eq_ind_r :

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

Defined.

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

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

End `Logic_lemmas`.

Module `EQNOTATIONS`.

Notation `"'rew' H 'in' H'"` := `(eq_rect _ _ H' _ H)`
(at level 10, *H'* at level 10,
format `"[' 'rew' H in '/' H' ']"`).

Notation `"'rew' [P] H 'in' H'"` := `(eq_rect _ P H' _ H)`
(at level 10, *H'* at level 10,
format `"[' 'rew' [P] '/' H in '/' H' ']"`).

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

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

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

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

End `EQNOTATIONS`.

Import `EqNotations`.

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

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

Theorem `f_equal2` :

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

Theorem `f_equal3` :

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

Theorem `f_equal4` :

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

Theorem `f_equal5` :

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

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

Theorem `f_equal_compose` : $\forall A B C (a b:A) (f:A \rightarrow B) (g:B \rightarrow C) (e:a=b),$
`f_equal g (f_equal f e) = f_equal (fun a => g (f a)) e.`

The groupoid structure of equality

Theorem `eq_trans_refl_l` : $\forall A (x y:A) (e:x=y),$ `eq_trans eq_refl e = e.`

Theorem `eq_trans_refl_r` : $\forall A (x y:A) (e:x=y),$ `eq_trans e eq_refl = e.`

Theorem `eq_sym_involutive` : $\forall A (x y:A) (e:x=y),$ `eq_sym (eq_sym e) = e.`

Theorem `eq_trans_sym_inv_l` : $\forall A (x y:A) (e:x=y),$ `eq_trans (eq_sym e) e = eq_refl.`

Theorem `eq_trans_sym_inv_r` : $\forall A (x y:A) (e:x=y),$ `eq_trans e (eq_sym e) = eq_refl.`

Theorem `eq_trans_assoc` : $\forall A (x y z t:A) (e:x=y) (e':y=z) (e'':z=t),$
`eq_trans e (eq_trans e' e'') = eq_trans (eq_trans e e') e''.`

Extra properties of equality

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

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

Lemma `eq_refl_map_distr` : $\forall A B x (f:A \rightarrow B),$ `f_equal f (eq_refl x) = eq_refl (f x).`

Lemma `eq_trans_map_distr` : $\forall A B x y z (f:A \rightarrow B) (e:x=y) (e':y=z),$ `f_equal f (eq_trans e e') = eq_trans (f_equal f e) (f_equal f e').`

Lemma `eq_sym_map_distr` : $\forall A B (x y:A) (f:A \rightarrow B) (e:x=y),$ `eq_sym (f_equal f e) = f_equal f (eq_sym e).`

Lemma `eq_trans_sym_distr` : $\forall A (x y z:A) (e:x=y) (e':y=z),$ `eq_sym (eq_trans e e') = eq_trans (eq_sym e') (eq_sym e).`

Lemma `eq_trans_rew_distr` : $\forall A (P:A \rightarrow \text{Type}) (x y z:A) (e:x=y) (e':y=z) (k:P x),$
`rew (eq_trans e e') in k = rew e' in rew e in k.`

Lemma `rew_const` : $\forall A P (x y:A) (e:x=y) (k:P),$
`rew [fun _ => P] e in k = k.`

Notation `sym_eq` := `eq_sym` (*only parsing*).

Notation `trans_eq` := `eq_trans` (*only parsing*).

Notation `sym_not_eq` := `not_eq_sym` (*only parsing*).

Notation `refl_equal` := `eq_refl` (*only parsing*).

Notation `sym_equal` := `eq_sym` (*only parsing*).

Notation `trans_equal` := `eq_trans` (*only parsing*).

Notation `sym_not_equal` := `not_eq_sym` (*only parsing*).

Hint Immediate `eq_sym not_eq_sym`: *core.*

Basic definitions about relations and properties

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

Definition `unique` ($A : \text{Type}$) ($P : A \rightarrow \text{Prop}$) ($x:A$) :=

$P x \wedge \forall (x':A), P x' \rightarrow x=x'$.

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

Unique existence

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

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

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

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

25.4 Being inhabited

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

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

Hint Resolve *inhabits*: *core*.

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

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

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

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

Declare Left Step *eq_stepl*.

Declare Right Step *eq_trans*.

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

Declare Left Step *iff_stepl*.

Declare Right Step *iff_trans*.

Equality for *ex* **Section** *ex*.

Local Unset Implicit Arguments.

Definition eq_ex_uncurried $\{A : \text{Type}\} (P : A \rightarrow \text{Prop}) \{u1 v1 : A\} \{u2 : P u1\} \{v2 : P v1\}$
 $(pq : \exists p : u1 = v1, \text{rew } p \text{ in } u2 = v2)$

: `ex_intro P u1 u2 = ex_intro P v1 v2`.

Definition `eq_ex` {*A* : Type} {*P* : *A* → Prop} (*u1 v1* : *A*) (*u2* : *P u1*) (*v2* : *P v1*)
 (*p* : *u1* = *v1*) (*q* : `rew p in u2 = v2`)

: `ex_intro P u1 u2 = ex_intro P v1 v2`
 := `eq_ex_uncurried P (ex_intro _ p q)`.

Definition `eq_ex_hprop` {*A*} {*P* : *A* → Prop} (*P_hprop* : ∀ (*x* : *A*) (*p q* : *P x*), *p* = *q*)
 (*u1 v1* : *A*) (*u2* : *P u1*) (*v2* : *P v1*)
 (*p* : *u1* = *v1*)

: `ex_intro P u1 u2 = ex_intro P v1 v2`
 := `eq_ex u1 v1 u2 v2 p (P_hprop _ _)`.

Lemma `rew_ex` {*A x*} {*P* : *A* → Type} (*Q* : ∀ *a*, *P a* → Prop) (*u* : ∃ *p*, *Q x p*) {*y*} (*H* : *x* = *y*)
 : `rew [fun a => ∃ p, Q a p] H in u`

= `match u with`
 | `ex_intro _ u1 u2`
 => `ex_intro`
 (*Q y*)
 (`rew H in u1`)
 (`rew dependent H in u2`)

`end`.

End `ex`.

Equality for `ex2` Section `ex2`.

Local Unset Implicit Arguments.

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

: `ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3`.

Definition `eq_ex2` {*A* : Type} {*P Q* : *A* → Prop}
 (*u1 v1* : *A*)
 (*u2* : *P u1*) (*v2* : *P v1*)
 (*u3* : *Q u1*) (*v3* : *Q v1*)
 (*p* : *u1* = *v1*) (*q* : `rew p in u2 = v2`) (*r* : `rew p in u3 = v3`)

: `ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3`
 := `eq_ex2_uncurried P Q (ex_intro2 _ _ p q r)`.

Definition `eq_ex2_hprop` {*A*} {*P Q* : *A* → Prop}
 (*P_hprop* : ∀ (*x* : *A*) (*p q* : *P x*), *p* = *q*)
 (*Q_hprop* : ∀ (*x* : *A*) (*p q* : *Q x*), *p* = *q*)
 (*u1 v1* : *A*) (*u2* : *P u1*) (*v2* : *P v1*) (*u3* : *Q u1*) (*v3* : *Q v1*)
 (*p* : *u1* = *v1*)

: `ex_intro2 P Q u1 u2 u3 = ex_intro2 P Q v1 v2 v3`
 := `eq_ex2 u1 v1 u2 v2 u3 v3 p (P_hprop _ _ _) (Q_hprop _ _)`.

Lemma `rew_ex2` {*A x*} {*P* : *A* → Type}
 (*Q* : ∀ *a*, *P a* → Prop)

```

      (R : ∀ a, P a → Prop)
      (u : exists2 p, Q x p & R x p) {y} (H : x = y)
: rew [fun a => exists2 p, Q a p & R a p] H in u
= match u with
| ex_intro2 _ _ u1 u2 u3
  => ex_intro2
      (Q y)
      (R y)
      (rew H in u1)
      (rew dependent H in u2)
      (rew dependent H in u3)
end.
End ex2.

```

Chapter 26

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.`

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

Chapter 27

Library `Coq.Init.Nat`

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

27.1 Peano natural numbers, definitions of operations

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

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

27.1.1 Constants

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

27.1.2 Basic operations

```
Definition succ := S.
```

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

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

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

Definition `double n := n + n.`

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

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

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

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

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

27.1.3 Comparisons

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

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

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

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

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

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

```
Fixpoint compare n m : comparison :=  
  match n, m with  
  | 0, 0 => Eq  
  | 0, S _ => Lt  
  | S _, 0 => Gt  
  | S n', S m' => compare n' m'  
  end.
```

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

27.1.4 Minimum, maximum

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

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

27.1.5 Parity tests

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

Definition odd n := negb (even n).

27.1.6 Power

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

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

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

```
Fixpoint tail_add n m :=
  match n with
  | 0 => m
  | S n => tail_add n (S m)
  end.
```

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

```
Fixpoint tail_addmul r n m :=
  match n with
```

```

| O ⇒ r
| S n ⇒ tail_addmul (tail_add m r) n m
end.

```

Definition `tail_mul` $n\ m := \text{tail_addmul } 0\ n\ m$.

27.1.8 Conversion with a decimal representation for printing/parsing

```

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

```

Definition `of_uint` $(d:\text{Decimal.uint}) := \text{of_uint_acc } d\ 0$.

```

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

```

Definition `to_uint` $n := \text{Decimal.rev (to_little_uint } n\ \text{Decimal.zero)}$.

```

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

```

Definition `to_int` $n := \text{Decimal.Pos (to_uint } n)$.

27.1.9 Euclidean division

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

```

Fixpoint divmod x y q u :=
  match x with
  | 0 ⇒ (q, u)

```

```

    | S x' => match u with
      | 0 => divmod x' y (S q) y
      | S u' => divmod x' y q u'
    end
  end.

Definition div x y :=
  match y with
  | 0 => y
  | S y' => fst (divmod x y' 0 y')
  end.

Definition modulo x y :=
  match y with
  | 0 => y
  | S y' => y' - snd (divmod x y' 0 y')
  end.

Infix "/" := div : nat_scope.
Infix "mod" := modulo (at level 40, no associativity) : nat_scope.

```

27.1.10 Greatest common divisor

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

```

Fixpoint gcd a b :=
  match a with
  | 0 => b
  | S a' => gcd (b mod (S a')) (S a')
  end.

```

27.1.11 Square

```

Definition square n := n × n.

```

27.1.12 Square root

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

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

```

Fixpoint sqrt_iter k p q r :=
  match k with
  | 0 => p
  | S k' => match r with

```

```

    | 0 => sqrt_iter k' (S p) (S (S q)) (S (S q))
    | S r' => sqrt_iter k' p q r'
end

```

end.

Definition `sqrt n := sqrt_iter n 0 0 0`.

27.1.13 Log2

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

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

Graphically (numbers are q , stars are r) :

```

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

```

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

```

Fixpoint log2_iter k p q r :=
  match k with
  | 0 => p
  | S k' => match r with
            | 0 => log2_iter k' (S p) (S q) q
            | S r' => log2_iter k' p (S q) r'
          end
end

```

end.

Definition `log2 n := log2_iter (pred n) 0 1 0`.

Iterator on natural numbers

```

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

```

Bitwise operations

We provide here some bitwise operations for unary numbers. Some might be really naive, they are just there for fulfilling the same interface as other for natural representations. As soon as binary

representations such as NArith are available, it is clearly better to convert to/from them and use their ops.

Fixpoint div2 $n :=$

```

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

```

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

```

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

```

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

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

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

```

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

```

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

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

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

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

Chapter 28

Library `Coq.Init.Notations`

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

Notations for propositional connectives

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

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

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

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

Reserved Notation " $\sim x$ " (at level 75, right associativity).

Notations for equality and inequalities

Reserved Notation " $x = y \text{ :> } T$ "

(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 " $x \langle \rangle y \text{ :> } T$ "

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

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

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

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

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

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

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

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

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

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

Arithmetical notations (also used for type constructors)

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

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

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

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

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

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

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

Notations for booleans

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

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

Notations for pairs

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

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

Reserved Notation $\{x\}$ (at level 0, x at level 99).

Notations for sigma-types or subsets

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

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

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

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

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

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

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

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

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

Reserved Notation $\{ \text{pat} \mid P \}$

(at level 0, *pat strict* pattern, *format* $\{ \text{pat} \mid P \}$).

Reserved Notation $\{ \text{pat} \mid P \& Q \}$

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

Reserved Notation $\{ \text{pat} : A \mid P \}$

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

Reserved Notation $\{ \text{pat} : A \mid P \& Q \}$

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

Reserved Notation $\{ \text{pat} : A \& P \}$

(at level 0, *pat strict* pattern, *format* $\{ \text{pat} : A \& P \}$).

Reserved Notation $\{ \text{pat} : A \& P \& Q \}$

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

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

Module IFNOTATIONS.

Notation "if c is p then u else v" :=

(*match* c with $p \Rightarrow u \mid _ \Rightarrow v$ *end*)

(at level 200, p pattern at level 100).

End IFNOTATIONS.

Scopes

Delimit Scope *type_scope* with *type*.
Delimit Scope *function_scope* with *function*.
Delimit Scope *core_scope* with *core*.

Open Scope *core_scope*.
Open Scope *function_scope*.
Open Scope *type_scope*.

ML Tactic Notations

Chapter 29

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.
Require Coq.Init.Nat.

Open Scope nat_scope.

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

The predecessor function

```
Notation pred := Nat.pred (only parsing).
Definition f_equal_pred := f_equal pred.
Theorem pred_Sn : ∀ n:nat, n = pred (S n).
```

Injectivity of successor

Definition `eq_add_S` $n\ m$ ($H: S\ n = S\ m$): $n = m := f_equal\ pred\ H$.

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

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

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

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

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

Zero is not the successor of a number

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

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

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

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

Addition

Notation `plus` := `Nat.add` (*only parsing*).

Infix "+" := `Nat.add` : *nat_scope*.

Definition `f_equal2_plus` := `f_equal2 plus`.

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

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

Lemma `plus_n_0` : $\forall\ n:\mathit{nat},\ n = n + 0$.

Hint `Resolve` `plus_n_0` `eq_refl`: *core*.

Lemma `plus_0_n` : $\forall\ n:\mathit{nat},\ 0 + n = n$.

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

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

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

Standard associated names

Notation `plus_0_r_reverse` := `plus_n_0` (*only parsing*).

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

Multiplication

Notation `mult` := `Nat.mul` (*only parsing*).

Infix "×" := `Nat.mul` : *nat_scope*.

Definition `f_equal2_mult` := `f_equal2 mult`.

Hint `Resolve` `f_equal2_mult`: *core*.

Lemma `mult_n_0` : $\forall\ n:\mathit{nat},\ 0 = n \times 0$.

Hint `Resolve` `mult_n_0`: *core*.

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

Hint `Resolve` `mult_n_Sm`: *core*.

Standard associated names

Notation `mult_0_r_reverse` := `mult_n_O` (*only parsing*).

Notation `mult_succ_r_reverse` := `mult_n_Sm` (*only parsing*).

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

Notation `minus` := `Nat.sub` (*only parsing*).

Infix `"-"` := `Nat.sub` : *nat_scope*.

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

Inductive `le` ($n:\text{nat}$) : `nat` → `Prop` :=

| `le_n` : $n \leq n$

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

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

Hint Constructors `le`: *core*.

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

Hint Unfold `lt`: *core*.

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

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

Hint Unfold `ge`: *core*.

Infix `"≥"` := `ge` : *nat_scope*.

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

Hint Unfold `gt`: *core*.

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

Notation `"x <= y <= z"` := $(x \leq y \wedge y \leq z)$: *nat_scope*.

Notation `"x <= y < z"` := $(x \leq y \wedge y < z)$: *nat_scope*.

Notation `"x < y < z"` := $(x < y \wedge y < z)$: *nat_scope*.

Notation `"x < y <= z"` := $(x < y \wedge y \leq z)$: *nat_scope*.

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

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

Theorem `le_0_n` : $\forall n, 0 \leq n$.

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

Case analysis

Theorem `nat_case` :

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

Principle of double induction

Theorem `nat_double_ind` :

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

$(\forall n:\text{nat}, R 0 n) \rightarrow$

$(\forall n:\text{nat}, R (S n) 0) \rightarrow$

$(\forall n m:\text{nat}, R n m \rightarrow R (S n) (S m)) \rightarrow \forall n m:\text{nat}, R n m$.

Maximum and minimum : definitions and specifications

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

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

Lemma `max_l` $n\ m : m \leq n \rightarrow \text{Nat.max } n\ m = n$.

Lemma `max_r` $n\ m : n \leq m \rightarrow \text{Nat.max } n\ m = m$.

Lemma `min_l` $n\ m : n \leq m \rightarrow \text{Nat.min } n\ m = n$.

Lemma `min_r` $n\ m : m \leq n \rightarrow \text{Nat.min } n\ m = m$.

Lemma `nat_rect_succ_r` $\{A\} (f: A \rightarrow A) (x:A) n :$

`nat_rect (fun _ => A) x (fun _ => f) (S n) = nat_rect (fun _ => A) (f x) (fun _ => f) n`.

Theorem `nat_rect_plus` :

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

`nat_rect (fun _ => A) x (fun _ => f) (n + m) =`

`nat_rect (fun _ => A) (nat_rect (fun _ => A) x (fun _ => f) m) (fun _ => f) n`.

Chapter 30

Library `Coq.Init.Prelude`

```
Require Export Notations.  
Require Export Logic.  
Require Export Logic_Type.  
Require Export Datatypes.  
Require Export Specif.  
Require Coq.Init.Decimal.  
Require Coq.Init.Nat.  
Require Export Peano.  
Require Export Coq.Init.Wf.  
Require Export Coq.Init.Tactics.  
Require Export Coq.Init.Tauto.  
  
Add Search Blacklist "_subproof" "_subterm" "Private_".
```

Chapter 31

Library `Coq.Init.Specif`

Basic specifications : sets that may contain logical information

`Set Implicit Arguments.`

`Require Import Notations.`

`Require Import Datatypes.`

`Require Import Logic.`

Subsets and Sigma-types

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

`Inductive 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.`

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

`Inductive 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 (A:=A) (fun x => P)) : type_scope.`

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

`Notation "{ x & P }" := (sigT (fun x => P)) : type_scope.`

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

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

`Notation "{ ' pat | P }" := (sig (fun pat => P)) : type_scope.`

Notation "{ ' pat | P & Q }" := (sig2 (fun pat => P) (fun pat => Q)) : type_scope.

Notation "{ ' pat : A | P }" := (sig (A:=A) (fun pat => P)) : type_scope.

Notation "{ ' pat : A | P & Q }" := (sig2 (A:=A) (fun pat => P) (fun pat => Q)) : type_scope.

Notation "{ ' pat : A & P }" := (sigT (A:=A) (fun pat => P)) : type_scope.

Notation "{ ' pat : A & P & Q }" := (sigT2 (A:=A) (fun pat => P) (fun pat => Q)) : type_scope.

Add Printing Let sig.

Add Printing Let sig2.

Add Printing Let sigT.

Add Printing Let sigT2.

Projections of sig

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

Section Subset_projections.

Variable A : Type.

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

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

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

End Subset_projections.

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

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

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

Definition sig_of_sig2 (A : Type) ($P Q$: $A \rightarrow \text{Prop}$) (X : sig2 $P Q$) : sig P
:= exist P
(let ($a, -, -$) := X in a)
(let ($x, p, -$) as s return (P (let ($a, -, -$) := s in a)) := X in p).

Projections of $sig2$

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

Section Subset_projections2.

Variable A : Type.

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

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

End Subset_projections2.

Projections of $\text{sig}T$

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, $(\text{proj}T1 \ x)$ is the first projection and $(\text{proj}T2 \ x)$ is the second projection, the type of which depends on the $\text{proj}T1$.

Section Projections.

Variable $A : \text{Type}$.

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

Definition $\text{proj}T1 \ (x:\text{sig}T \ P) : A := \text{match } x \text{ with}$
| $\text{exist}T \ _ \ a \ _ \Rightarrow a$
end.

Definition $\text{proj}T2 \ (x:\text{sig}T \ P) : P \ (\text{proj}T1 \ x) :=$
 $\text{match } x \ \text{return } P \ (\text{proj}T1 \ x) \ \text{with}$
| $\text{exist}T \ _ \ _ \ h \Rightarrow h$
end.

End Projections.

$\text{sig}T2$ of a predicate can be projected to a $\text{sig}T$.

This allows $\text{proj}T1$ and $\text{proj}T2$ to be usable with $\text{sig}T2$.

The **let** statements occur in the body of the $\text{exist}T$ so that $\text{proj}T1$ of a coerced $X : \text{sig}T2 \ P \ Q$ will unify with **let** $(a, -, -) := X$ in a

Definition $\text{sig}T_of_sigT2 \ (A : \text{Type}) \ (P \ Q : A \rightarrow \text{Type}) \ (X : \text{sig}T2 \ P \ Q) : \text{sig}T \ P$
 $:= \text{exist}T \ P$
 $(\text{let } (a, -, -) := X \ \text{in } a)$
 $(\text{let } (x, p, -) \ \text{as } s \ \text{return } (P \ (\text{let } (a, -, -) := s \ \text{in } a))) := X \ \text{in } p).$

Projections of $\text{sig}T2$

An element x of a sigma-type $\{y:A \ \& \ P \ y \ \& \ Q \ y\}$ is a dependent pair made of an a of type A , an h of type $P \ a$, and an h' of type $Q \ a$. Then, $(\text{proj}T1 \ (\text{sig}T_of_sigT2 \ x))$ is the first projection, $(\text{proj}T2 \ (\text{sig}T_of_sigT2 \ x))$ is the second projection, and $(\text{proj}T3 \ x)$ is the third projection, the types of which depends on the $\text{proj}T1$.

Section Projections2.

Variable $A : \text{Type}$.

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

Definition $\text{proj}T3 \ (e : \text{sig}T2 \ P \ Q) :=$
 $\text{let } (a, b, c) \ \text{return } Q \ (\text{proj}T1 \ (\text{sig}T_of_sigT2 \ e)) := e \ \text{in } c.$

End Projections2.

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

Definition $\text{sig_of_sig}T \ (A : \text{Type}) \ (P : A \rightarrow \text{Prop}) \ (X : \text{sig}T \ P) : \text{sig} \ P$
 $:= \text{exist} \ P \ (\text{proj}T1 \ X) \ (\text{proj}T2 \ X).$

Definition $\text{sig}T_of_sig \ (A : \text{Type}) \ (P : A \rightarrow \text{Prop}) \ (X : \text{sig} \ P) : \text{sig}T \ P$
 $:= \text{exist}T \ P \ (\text{proj}1_sig \ X) \ (\text{proj}2_sig \ X).$

$sigT2$ of a predicate is equivalent to $sig2$

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

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

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

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

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

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

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

Lemma $inhabited_sig_to_exists$ $\{A P\} : inhabited \{x : A \mid P x\} \rightarrow \exists x : A, P x$.

Equality of sigma types **Import** *EqNotations*.

Equality for $sigT$ **Section** $sigT$.

Local Unset Implicit Arguments.

Projecting an equality of a pair to equality of the first components **Definition** $projT1_eq$
 $\{A\} \{P : A \rightarrow Type\} \{u v : \{a : A \& P a\}\} (p : u = v)$
 $: projT1 u = projT1 v$
 $:= f_equal$ ($@projT1 _ _$) p .

Projecting an equality of a pair to equality of the second components **Definition** $projT2_eq$
 $\{A\} \{P : A \rightarrow Type\} \{u v : \{a : A \& P a\}\} (p : u = v)$
 $: rew projT1_eq p$ **in** $projT2 u = projT2 v$
 $:= rew dependent p$ **in** eq_refl .

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

Equality of $sigT$ is itself a $sigT$ (backwards-reasoning version) **Definition** $eq_sigT_uncurried$
 $\{A : Type\} \{P : A \rightarrow Type\} (u v : \{a : A \& P a\})$
 $(pq : \{p : projT1 u = projT1 v \& rew p$ **in** $projT2 u = projT2 v\})$
 $: u = v$.

Curried version of proving equality of sigma types **Definition** eq_sigT $\{A : Type\} \{P : A \rightarrow Type\} (u v : \{a : A \& P a\})$
 $(p : projT1 u = projT1 v) (q : rew p$ **in** $projT2 u = projT2 v)$
 $: u = v$
 $:= eq_sigT_uncurried u v$ ($existT _ p q$).

Equality of $sigT$ when the property is an $hProp$ **Definition** eq_sigT_hprop $\{A P\} (P_hprop : \forall (x : A) (p q : P x), p = q)$

```

      (u v : { a : A & P a })
      (p : projT1 u = projT1 v)
    : u = v
    := eq_sigT u v p (P_hprop - -).

```

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

```

      (u v : { a : A & P a })
    : u = v  $\leftrightarrow$  { p : projT1 u = projT1 v & rew p in projT2 u = projT2 v }.

```

Induction principle for @eq (sigT -) **Definition** eq_sigT_rect {A P} {u v : { a : A & P a }} (Q : u = v \rightarrow Type)

```

      (f :  $\forall$  p q, Q (eq_sigT u v p q))
    :  $\forall$  p, Q p.

```

Definition eq_sigT_rec {A P u v} (Q : u = v :> { a : A & P a } \rightarrow Set) := eq_sigT_rect Q.

Definition eq_sigT_ind {A P u v} (Q : u = v :> { a : A & P a } \rightarrow Prop) := eq_sigT_rec Q.

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

```

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

```

Non-dependent classification of equality of *sigT* **Definition** eq_sigT_nondep {A B : Type} (u v : { a : A & B })

```

      (p : projT1 u = projT1 v) (q : projT2 u = projT2 v)
    : u = v
    := @eq_sigT - - u v p (eq_trans (rew_const - -) q).

```

Classification of transporting across an equality of *sigTs* **Lemma** rew_sigT {A x} {P : A \rightarrow Type} (Q : \forall a, P a \rightarrow Prop) (u : { p : P x & Q x p }) {y} (H : x = y)

```

    : rew [fun a  $\Rightarrow$  { p : P a & Q a p }] H in u
      = existT
        (Q y)
        (rew H in projT1 u)
        (rew dependent H in (projT2 u)).

```

End sigT.

Equality for *sig* **Section** sig.

Local Unset Implicit Arguments.

Projecting an equality of a pair to equality of the first components **Definition** proj1_sig_eq {A} {P : A \rightarrow Prop} {u v : { a : A | P a }} (p : u = v)

```

    : proj1_sig u = proj1_sig v
    := f_equal (@proj1_sig - -) p.

```

Projecting an equality of a pair to equality of the second components **Definition** proj2_sig_eq {A} {P : A \rightarrow Prop} {u v : { a : A | P a }} (p : u = v)

```

    : rew proj1_sig_eq p in proj2_sig u = proj2_sig v
    := rew dependent p in eq_refl.

```

Equality of *sig* is itself a *sig* (forwards-reasoning version) **Definition** eq_exist_uncurried {A

`: Type` } {`P : A → Prop`} {`u1 v1 : A`} {`u2 : P u1`} {`v2 : P v1`}
`(pq : { p : u1 = v1 | rew p in u2 = v2 })`
`: exist _ u1 u2 = exist _ v1 v2.`

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

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

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

Definition `eq_sig_rec` {`A P u v`} (`Q : u = v :> { a : A | P a } → Set`) := `eq_sig_rect Q`.

Definition `eq_sig_ind` {`A P u v`} (`Q : u = v :> { a : A | P a } → Prop`) := `eq_sig_rec Q`.

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

Equivalence of equality of *sig* with a *sig* of equality We could actually prove an isomorphism
 here, and not just \leftrightarrow , but for simplicity, we don't. **Definition** `eq_sig_uncurried_iff` {`A`} {`P : A`
`→ Prop`}

`(u v : { a : A | P a })`
`: u = v ↔ { p : proj1_sig u = proj1_sig v | rew p in proj2_sig u = proj2_sig v }.`

Equivalence of equality of *sig* involving hProps with equality of the first components **Definition**
`eq_sig_hprop_iff` {`A`} {`P : A → Prop`} (`P_hprop : ∀ (x : A) (p q : P x), p = q`)
`(u v : { a : A | P a })`
`: u = v ↔ (proj1_sig u = proj1_sig v)`
`:= conj (fun p ⇒ f_equal (@proj1_sig _ _) p) (eq_sig_hprop P_hprop u v).`

Lemma `rew_sig` {`A x`} {`P : A → Type`} (`Q : ∀ a, P a → Prop`) (`u : { p : P x | Q x p }`) {`y`} (`H`
`: x = y`)
`: rew [fun a ⇒ { p : P a | Q a p }] H in u`
`= exist`
`(Q y)`
`(rew H in proj1_sig u)`
`(rew dependent H in proj2_sig u).`

End sig.

Equality for *sigT* **Section** `sigT2`.

$: u = v$
 $\leftrightarrow \{ p : \text{projT1 } u = \text{projT1 } v$
 $\quad \& \text{rew } p \text{ in projT2 } u = \text{projT2 } v \& \text{rew } p \text{ in projT3 } u = \text{projT3 } v \}$.

Induction principle for $\text{@eq } (sigT2 _ _)$ **Definition** $\text{eq_sigT2_rect } \{A P Q\} \{u v : \{ a : A$
 $\& P a \& Q a \}\} (R : u = v \rightarrow \text{Type})$
 $(f : \forall p q r, R (\text{eq_sigT2 } u v p q r))$
 $: \forall p, R p$.

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

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

Equivalence of equality of $sigT2$ involving hProps with equality of the first components **Definition**
 $\text{eq_sigT2_hprop_iff } \{A P Q\} (Q_hprop : \forall (x : A) (p q : Q x), p = q)$
 $(u v : \{ a : A \& P a \& Q a \})$

$: u = v \leftrightarrow (u = v :> \{ a : A \& P a \})$
 $:= \text{conj } (\text{fun } p \Rightarrow \text{f_equal } (\text{@sigT_of_sigT2 } _ _ _) p) (\text{eq_sigT2_hprop } Q_hprop u v)$.

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

$(p : \text{projT1 } u = \text{projT1 } v) (q : \text{projT2 } u = \text{projT2 } v) (r : \text{projT3 } u = \text{projT3 } v)$
 $: u = v$
 $:= \text{@eq_sigT2 } _ _ _ u v p (\text{eq_trans } (\text{rew_const } _ _) q) (\text{eq_trans } (\text{rew_const } _ _) r)$.

Classification of transporting across an equality of $sigT2$ s **Lemma** $\text{rew_sigT2 } \{A x\} \{P : A \rightarrow$
 $\text{Type}\} (Q R : \forall a, P a \rightarrow \text{Prop})$

$(u : \{ p : P x \& Q x p \& R x p \})$
 $\{y\} (H : x = y)$
 $: \text{rew } [\text{fun } a \Rightarrow \{ p : P a \& Q a p \& R a p \}] H \text{ in } u$
 $= \text{existT2}$
 $(Q y)$
 $(R y)$
 $(\text{rew } H \text{ in projT1 } u)$
 $(\text{rew dependent } H \text{ in projT2 } u)$
 $(\text{rew dependent } H \text{ in projT3 } u)$.

End sigT2.

Equality for $sig2$ **Section** sig2.

Local Unset Implicit Arguments.

Projecting an equality of a pair to equality of the first components **Definition** sig_of_sig2_eq
 $\{A\} \{P Q : A \rightarrow \text{Prop}\} \{u v : \{ a : A \mid P a \& Q a \}\} (p : u = v)$

$: u = v :> \{ a : A \mid P a \}$
 $:= \text{f_equal } _ p$.

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

$: \text{proj1_sig } u = \text{proj1_sig } v$
 $:= \text{proj1_sig_eq } (\text{sig_of_sig2_eq } p)$.

Projecting an equality of a pair to equality of the second components **Definition** $\text{proj2_sig_of_sig2_eq}$

$\{A\} \{P Q : A \rightarrow \text{Prop}\} \{u v : \{ a : A \mid P a \ \& \ Q a \}\} (p : u = v)$
 $\quad : \text{rew proj1_sig_of_sig2_eq } p \text{ in proj2_sig } u = \text{proj2_sig } v$
 $\quad := \text{rew dependent } p \text{ in eq_refl.}$

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

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

Equality of *sig2* is itself a *sig2* (backwards-reasoning version) Definition eq_sig2_uncurried
 $\{A\} \{P Q : A \rightarrow \text{Prop}\} (u v : \{ a : A \mid P a \ \& \ Q a \})$
 $\quad (pqr : \{ p : \text{proj1_sig } u = \text{proj1_sig } v$
 $\quad \quad \mid \text{rew } p \text{ in } \text{proj2_sig } u = \text{proj2_sig } v \ \& \ \text{rew } p \text{ in } \text{proj3_sig } u = \text{proj3_sig } v \})$
 $\quad : u = v.$

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

Equality of *sig2* when the second property is an hProp Definition eq_sig2_hprop $\{A\} \{P Q : A \rightarrow \text{Prop}\} (Q_hprop : \forall (x : A) (p q : Q x), p = q)$
 $\quad (u v : \{ a : A \mid P a \ \& \ Q a \})$
 $\quad (p : u = v \rightarrow \{ a : A \mid P a \})$
 $\quad : u = v$
 $\quad := \text{eq_sig2 } u v (\text{proj1_sig_eq } p) (\text{proj2_sig_eq } p) (Q_hprop _ _).$

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

Induction principle for $\text{@eq } (sig2 _ _)$ Definition eq_sig2_rect $\{A P Q\} \{u v : \{ a : A \mid P a \ \& \ Q a \}\} (R : u = v \rightarrow \text{Type})$
 $\quad (f : \forall p q r, R (\text{eq_sig2 } u v p q r))$
 $\quad : \forall p, R p.$

Definition eq_sig2_rec $\{A P Q u v\} (R : u = v \rightarrow \{ a : A \mid P a \ \& \ Q a \} \rightarrow \text{Set}) := \text{eq_sig2_rect } R.$

Definition `eq_sig2_ind` $\{A P Q u v\} (R : u = v \rightarrow \{a : A \mid P a \ \& \ Q a\} \rightarrow \text{Prop}) := \text{eq_sig2_rec } R$.

Equivalence of equality of *sig2* involving hProps with equality of the first components **Definition** `eq_sig2_hprop_iff` $\{A\} \{P Q : A \rightarrow \text{Prop}\} (Q_hprop : \forall (x : A) (p q : Q x), p = q)$
 $(u v : \{a : A \mid P a \ \& \ Q a\})$
 $: u = v \leftrightarrow (u = v \rightarrow \{a : A \mid P a\})$
 $:= \text{conj } (\text{fun } p \Rightarrow \text{f_equal } (@\text{sig_of_sig2 } _ _ _) p) (\text{eq_sig2_hprop } Q_hprop \ u \ v)$.

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

Classification of transporting across an equality of *sig2*s **Lemma** `rew_sig2` $\{A x\} \{P : A \rightarrow \text{Type}\} (Q R : \forall a, P a \rightarrow \text{Prop})$
 $(u : \{p : P x \mid Q x \ p \ \& \ R x \ p\})$
 $\{y\} (H : x = y)$
 $: \text{rew } [\text{fun } a \Rightarrow \{p : P a \mid Q a \ p \ \& \ R a \ p\}] \ H \ \text{in } u$
 $= \text{exist2}$
 $(Q \ y)$
 $(R \ y)$
 $(\text{rew } H \ \text{in } \text{proj1_sig } u)$
 $(\text{rew } \text{dependent } H \ \text{in } \text{proj2_sig } u)$
 $(\text{rew } \text{dependent } H \ \text{in } \text{proj3_sig } u)$.

End sig2.

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

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

Add Printing *If sumbool.*

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

Inductive `sumor` $(A : \text{Type}) (B : \text{Prop}) : \text{Type} :=$
 $| \text{inleft} : A \rightarrow A + \{B\}$
 $| \text{inright} : B \rightarrow A + \{B\}$
where $"A + \{B\}" := (\text{sumor } A \ B) : \text{type_scope}$.

Add Printing *If sumor.*

Various forms of the axiom of choice for specifications

Section `Choice_lemmas`.

Variables $S \ S' : \text{Set}$.
Variable $R : S \rightarrow S' \rightarrow \text{Prop}$.

```

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

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

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

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

End Choice_lemmas.

Section Dependent_choice_lemmas.

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

Lemma dependent_choice :
  (∀ x:X, {y | R x y}) →
  ∀ x0, {f : nat → X | f 0 = x0 ∧ ∀ n, R (f n) (f (S n))}.

End Dependent_choice_lemmas.

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

Variable A : Type.

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

End Exc.

Definition except := False_rec.

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

Hint Resolve left right inleft inright: core.
Hint Resolve exist exist2 existT existT2: core.

Notation sigS := sigT (compat "8.7").
Notation existS := existT (compat "8.7").
Notation sigS_rect := sigT_rect (compat "8.7").
Notation sigS_rec := sigT_rec (compat "8.7").
Notation sigS_ind := sigT_ind (compat "8.7").
Notation projS1 := projT1 (compat "8.7").
Notation projS2 := projT2 (compat "8.7").

Notation sigS2 := sigT2 (compat "8.7").
Notation existS2 := existT2 (compat "8.7").
Notation sigS2_rect := sigT2_rect (compat "8.7").
Notation sigS2_rec := sigT2_rec (compat "8.7").
Notation sigS2_ind := sigT2_ind (compat "8.7").

```

Chapter 32

Library `Coq.Init.Tactics`

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

32.1 Useful tactics

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

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

A tactic for proof by contradiction. With `contradict H`,

- $H: \sim A \mid\text{-} B$ gives $\mid\text{-} A$
- $H: \sim A \mid\text{-} \sim B$ gives $H: B \mid\text{-} A$
- $H: A \mid\text{-} B$ gives $\mid\text{-} \sim A$
- $H: A \mid\text{-} \sim B$ gives $H: B \mid\text{-} \sim A$
- $H:\text{False}$ leads to a resolved subgoal.

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

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

```

in
let neg H := match goal with
|  $\vdash (\neg\_)$   $\Rightarrow$  negneg H
|  $\vdash (\_ \rightarrow \text{False})$   $\Rightarrow$  negneg H
|  $\vdash \_ \Rightarrow$  negpos H
end in
let pos H := match goal with
|  $\vdash (\neg\_)$   $\Rightarrow$  posneg H
|  $\vdash (\_ \rightarrow \text{False})$   $\Rightarrow$  posneg H
|  $\vdash \_ \Rightarrow$  pospos H
end in
match type of H with
|  $(\neg\_)$   $\Rightarrow$  neg H
|  $(\_ \rightarrow \text{False})$   $\Rightarrow$  neg H
|  $\_ \Rightarrow$  (elim H;fail) || pos H
end.

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

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

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

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

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

  Break every hypothesis of a certain type

Ltac destruct_all t :=
  match goal with
  |  $x : t \vdash \_ \Rightarrow$  destruct x; destruct_all t
  |  $\_ \Rightarrow$  idtac
  end.

Tactic Notation "rewrite_all" constr(eq) := repeat rewrite eq in *.
Tactic Notation "rewrite_all" "<-" constr(eq) := repeat rewrite  $\leftarrow$  eq in *.

  Tactics for applying equivalences.

```

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

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

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

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

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

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

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

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

```
Ltac easy :=
let rec use_hyp H :=
  match type of H with
  | -  $\wedge$  -  $\Rightarrow$  exact H || destruct_hyp H
  | -  $\Rightarrow$  try solve [inversion H]
  end
with do_intro := let H := fresh in intro H; use_hyp H
with destruct_hyp H := case H; clear H; do_intro; do_intro in
```

```

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

```

Tactic Notation "now" *tactic*(*t*) := *t*; *easy*.

Slightly more than *easy*

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

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

Ltac *now_show* *c* := change *c*.

Support for rewriting decidability statements

Set Implicit Arguments.

Lemma *decide_left* : $\forall (C:\text{Prop}) (decide:\{C\}+\{\neg C\}),$
 $C \rightarrow \forall P:\{C\}+\{\neg C\}\rightarrow\text{Prop}, (\forall H:C, P (\text{left } H)) \rightarrow P \text{ decide}.$

Lemma *decide_right* : $\forall (C:\text{Prop}) (decide:\{C\}+\{\neg C\}),$
 $\neg C \rightarrow \forall P:\{C\}+\{\neg C\}\rightarrow\text{Prop}, (\forall H:\neg C, P (\text{right } H)) \rightarrow P \text{ decide}.$

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*.

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

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

inversion_sigma

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

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

Ltac *induction_sigma_in_using H rect* :=
let *H0* := fresh *H* in
let *H1* := fresh *H* in
induction *H* as [*H0 H1*] using (*rect* - - - -);
simpl_proj_exist_in H0;
simpl_proj_exist_in H1.

Ltac *induction_sigma2_in_using H rect* :=

```

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

```

A version of *time* that works for constrs

```

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

```

Useful combinators

```

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

```

Chapter 33

Library `Coq.Init.Tauto`

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

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

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

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

    assert X2; [exact (id0 id1) | clear id0]
    | id: ∀ (- : ?X1), ?X2 ⊢ _ ⇒
      is_unit_or_eq flags X1; cut X2;
    [ intro; clear id
    |
    cut X1; [exact id| constructor 1; fail]
    ]
    )
```

```

| id:  $\forall$  ( $\_ : ?X1$ ),  $?X2 \vdash \_ \Rightarrow$ 
  flatten_contravariant_conj flags X1 X2 id

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

|  $\vdash ?X1 \Rightarrow$  is_conj flags X1; split
|  $\vdash$  (Coq.Init.Logic.iff  $\_ \_$ )  $\Rightarrow$  split
|  $\vdash$  (Coq.Init.Logic.not  $\_$ )  $\Rightarrow$  red
end;
not_dep_intros).

Local Ltac tauto_intuit flags t_reduce t_solver :=
  let rec t_tauto_intuit :=
    (simplif flags; axioms flags
  || match reverse goal with
    | id: $\forall$ ( $\_ : \forall$  ( $\_ : ?X1$ ),  $?X2$ ),  $?X3 \vdash \_ \Rightarrow$ 
      cut X3;
      [ intro; clear id; t_tauto_intuit
      | cut ( $\forall$  ( $\_ : X1$ ), X2);
        [ exact id
        | generalize (fun y:X2  $\Rightarrow$  id (fun x:X1  $\Rightarrow$  y)); intro; clear id;
          solve [ t_tauto_intuit ]]]
    | id: $\forall$  ( $\_ : \text{not } ?X1$ ),  $?X3 \vdash \_ \Rightarrow$ 
      cut X3;
      [ intro; clear id; t_tauto_intuit
      | cut (not X1); [ exact id | clear id; intro; solve [ t_tauto_intuit ]]]
    |  $\vdash ?X1 \Rightarrow$ 
      is_disj flags X1; solve [left;t_tauto_intuit | right;t_tauto_intuit]
      end
    ||
  match goal with |  $\vdash \forall$  ( $\_ : \_$ ),  $\_ \Rightarrow$  intro; t_tauto_intuit
  |  $\vdash \_ \Rightarrow$  t_reduce; t_solver
  end
  ||
  t_solver
) in t_tauto_intuit.

Local Ltac intuition_gen flags solver := tauto_intuit flags reduction_not_iff solver.
Local Ltac tauto_intuitionistic flags := intuition_gen flags fail || fail "tauto failed".
Local Ltac tauto_classical flags :=
  (apply_nnpp || fail "tauto failed"); (tauto_intuitionistic flags || fail "Classical tauto failed").

```

```

Local Ltac tauto_gen flags := tauto_intuitionistic flags || tauto_classical flags.
Ltac tauto := with_uniform_flags ltac:(fun flags => tauto_gen flags).
Ltac dtauto := with_power_flags ltac:(fun flags => tauto_gen flags).
Ltac intuition := with_uniform_flags ltac:(fun flags => intuition_gen flags ltac:(auto with *)).
Local Ltac intuition_then tac := with_uniform_flags ltac:(fun flags => intuition_gen flags tac).
Ltac dintuition := with_power_flags ltac:(fun flags => intuition_gen flags ltac:(auto with *)).
Local Ltac dintuition_then tac := with_power_flags ltac:(fun flags => intuition_gen flags tac).
Tactic Notation "intuition" := intuition.
Tactic Notation "intuition" tactic(t) := intuition_then t.
Tactic Notation "dintuition" := dintuition.
Tactic Notation "dintuition" tactic(t) := dintuition_then t.

```

Chapter 34

Library `Coq.Init.Wf`

34.1 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.`

A relation is well-founded if every element is accessible

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

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

`Hypothesis Rwf : well_founded.`

`Theorem well_founded_induction_type :`

`∀ P:A → Type,`

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

`Theorem well_founded_induction :`

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

Theorem `well_founded_ind` :

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

Well-founded fixpoints

Section `FixPoint`.

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

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

Fixpoint `Fix_F` $(x:A) (a:\mathbf{Acc}\ x) : P\ x :=$
 $F\ (\mathbf{fun}\ (y:A)\ (h:R\ y\ x) \Rightarrow \mathbf{Fix_F}\ (\mathbf{Acc_inv}\ a\ h)).$

Scheme `Acc_inv_dep` := **Induction** for `Acc` Sort `Prop`.

Lemma `Fix_F_eq` :

$\forall (x:A) (r:\mathbf{Acc}\ x),$
 $F\ (\mathbf{fun}\ (y:A)\ (p:R\ y\ x) \Rightarrow \mathbf{Fix_F}\ (x:=y)\ (\mathbf{Acc_inv}\ r\ p)) = \mathbf{Fix_F}\ (x:=x)\ r.$

Definition `Fix` $(x:A) := \mathbf{Fix_F}\ (Rwf\ x).$

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

Hypothesis

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

Lemma `Fix_F_inv` : $\forall (x:A) (r\ s:\mathbf{Acc}\ x), \mathbf{Fix_F}\ r = \mathbf{Fix_F}\ s.$

Lemma `Fix_eq` : $\forall x:A, \mathbf{Fix}\ x = F\ (\mathbf{fun}\ (y:A)\ (p:R\ y\ x) \Rightarrow \mathbf{Fix}\ y).$

End `FixPoint`.

End `Well_founded`.

Well-founded fixpoints over pairs

Section `Well_founded_2`.

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

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

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

Section `FixPoint_2`.

Variable

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

Fixpoint `Fix_F_2` $(x:A) (x':B) (a:\mathbf{Acc}\ R\ (x, x')) : P\ x\ x' :=$
 F
 $(\mathbf{fun}\ (y:A) (y':B) (h:R\ (y, y')\ (x, x')) \Rightarrow$

`Fix_F_2 (x:=y) (x':=y') (Acc_inv a (y, y') h)).`

`End FixPoint_2.`

`Hypothesis Rwf : well_founded R.`

`Theorem well_founded_induction_type_2 :`

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

`End Well_founded_2.`

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

`Section Acc_generator.`

`Variable A : Type.`

`Variable R : A → A → Prop.`

`Fixpoint Acc_intro_generator n (wf : well_founded R) :=`

`match n with`
`| O ⇒ wf`
`| S n ⇒ fun x ⇒ Acc_intro x (fun y _ ⇒ Acc_intro_generator n (Acc_intro_generator n`
`wf) y)`
`end.`

`End Acc_generator.`

Chapter 35

Library `Coq.Logic.Berardi`

This file formalizes Berardi's paradox which says that in the calculus of constructions, excluded middle (EM) and axiom of choice (AC) imply proof irrelevance (PI). Here, the axiom of choice is not necessary because of the use of inductive types.

```
@article{Barbanera-Berardi:JFP96,  
  author    = {F. Barbanera and S. Berardi},  
  title     = {Proof-irrelevance out of Excluded-middle and Choice  
              in the Calculus of Constructions},  
  journal   = {Journal of Functional Programming},  
  year      = {1996},  
  volume    = {6},  
  number    = {3},  
  pages     = {519-525}  
}
```

Set Implicit Arguments.

Section `Berardis_paradox.`

Excluded middle **Hypothesis** `EM : $\forall P:\text{Prop}, P \vee \neg P$.`

Conditional on any proposition. **Definition** `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 :`

`$\forall (P B:\text{Prop}) (e1 e2:P) (Q:P \rightarrow \text{Prop}),$`
 `$(B \rightarrow Q e1) \rightarrow (\neg B \rightarrow Q e2) \rightarrow Q$ (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, r.(j2) (r.(i2) a) = a`.

End `Retracts`.

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

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

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

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

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

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

Encoding of Russel's paradox

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

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

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

Theorem `classical_proof_irrelevance : T = F`.

Notation `classical_proof_irrelevance := classical_proof_irrelevance (compat "8.8")`.

End `Berardis_paradox`.

Chapter 36

Library `Coq.Logic.ChoiceFacts`

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

36.1 References:

[*Bell*] John L. Bell, Choice principles in intuitionistic set theory, unpublished.

[*Bell93*] John L. Bell, Hilbert's Epsilon Operator in Intuitionistic Type Theories, *Mathematical Logic Quarterly*, volume 39, 1993.

[*Carlström04*] Jesper Carlström, EM + Ext + AC_{int} is equivalent to AC_{ext}, *Mathematical Logic Quarterly*, vol 50(3), pp 236-240, 2004.

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

[*Werner97*] Benjamin Werner, Sets in Types, Types in Sets, TACS, 1997.

`Require Import RelationClasses Logic.`

`Set Implicit Arguments.`

36.2 Definitions

Choice, reification and description schemes

We make them all polymorphic. Most of them have existentials as conclusion so they require polymorphism otherwise their first application (e.g. to an existential in `Set`) will fix the level of `A`.

`Section ChoiceSchemes.`

`Variables A B :Type.`

`Variable P:A→Prop.`

36.2.1 Constructive choice and description

AC_{rel} = relational form of the (non extensional) axiom of choice (a “set-theoretic” axiom of choice)

`Definition RelationalChoice_on :=`

`∀ R:A→B→Prop,`
`(∀ x : A, ∃ y : B, R x y) →`

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

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

Definition FunctionalChoice_on_rel $(R : A \rightarrow B \rightarrow \mathbf{Prop}) :=$

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

Definition FunctionalChoice_on :=

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

AC_fun_dep = functional form of the (non extensional) axiom of choice, with dependent functions **Definition** DependentFunctionalChoice_on $(A : \mathbf{Type}) (B : A \rightarrow \mathbf{Type}) :=$

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

AC_trunc = axiom of choice for propositional truncations (truncation and quantification commute) **Definition** InhabitedForallCommute_on $(A : \mathbf{Type}) (B : A \rightarrow \mathbf{Type}) :=$

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

DC_fun = functional form of the dependent axiom of choice

Definition FunctionalDependentChoice_on :=

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

ACw_fun = functional form of the countable axiom of choice

Definition FunctionalCountableChoice_on :=

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

AC! = functional relation reification (known as axiom of unique choice in topos theory, sometimes called principle of definite description in the context of constructive type theory, sometimes called axiom of no choice)

Definition FunctionalRelReification_on :=

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

AC_dep! = functional relation reification, with dependent functions see AC! **Definition** DependentFunctionalRelReification_on $(A : \mathbf{Type}) (B : A \rightarrow \mathbf{Type}) :=$

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

AC_fun_repr = functional choice of a representative in an equivalence class

Definition RepresentativeFunctionalChoice_on :=

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

AC_fun_setoid = functional form of the (so-called extensional) axiom of choice from setoids

Definition SetoidFunctionalChoice_on :=

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

AC_fun_setoid_gen = functional form of the general form of the (so-called extensional) axiom of choice over setoids

Definition GeneralizedSetoidFunctionalChoice_on :=

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

AC_fun_setoid_simple = functional form of the (so-called extensional) axiom of choice from setoids on locally compatible relations

Definition SimpleSetoidFunctionalChoice_on A B :=

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

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

Definition ConstructiveIndefiniteDescription_on :=

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

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

Definition ConstructiveDefiniteDescription_on :=

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

36.2.2 Weakly classical choice and description

GAC_rel = guarded relational form of the (non extensional) axiom of choice

Definition GuardedRelationalChoice_on :=

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

GAC_fun = guarded functional form of the (non extensional) axiom of choice

Definition GuardedFunctionalChoice_on :=

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

GAC! = guarded functional relation reification

Definition GuardedFunctionalRelReification_on :=

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

OAC_rel = “omniscient” relational form of the (non extensional) axiom of choice

Definition OmniscientRelationalChoice_on :=

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

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

Definition OmniscientFunctionalChoice_on :=

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

D_epsilon = (weakly classical) indefinite description principle

Definition EpsilonStatement_on :=

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

D_iota = (weakly classical) definite description principle

Definition IotaStatement_on :=

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

End ChoiceSchemes.

Generalized schemes

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

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

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

Notation InhabitedForallCommute :=
 $(\forall A (B : A \rightarrow \text{Type}), \text{InhabitedForallCommute_on } B).$

Notation FunctionalDependentChoice :=
 $(\forall A : \text{Type}, \text{FunctionalDependentChoice_on } A).$

Notation FunctionalCountableChoice :=
 $(\forall A : \text{Type}, \text{FunctionalCountableChoice_on } A).$

Notation FunctionalChoiceOnInhabitedSet :=
 $(\forall A B : \text{Type}, \text{inhabited } B \rightarrow \text{FunctionalChoice_on } A B).$

Notation FunctionalRelReification :=
 $(\forall A B : \text{Type}, \text{FunctionalRelReification_on } A B).$

Notation DependentFunctionalRelReification :=
 $(\forall A (B : A \rightarrow \text{Type}), \text{DependentFunctionalRelReification_on } B).$

Notation RepresentativeFunctionalChoice :=
 $(\forall A : \text{Type}, \text{RepresentativeFunctionalChoice_on } A).$

Notation SetoidFunctionalChoice :=
 $(\forall A B : \text{Type}, \text{SetoidFunctionalChoice_on } A B).$

Notation GeneralizedSetoidFunctionalChoice :=
 $(\forall A B : \text{Type}, \text{GeneralizedSetoidFunctionalChoice_on } A B).$

Notation SimpleSetoidFunctionalChoice :=
 $(\forall A B : \text{Type}, \text{SimpleSetoidFunctionalChoice_on } A B).$

Notation GuardedRelationalChoice :=
 $(\forall A B : \text{Type}, \text{GuardedRelationalChoice_on } A B).$

Notation GuardedFunctionalChoice :=
 $(\forall A B : \text{Type}, \text{GuardedFunctionalChoice_on } A B).$

Notation GuardedFunctionalRelReification :=
 $(\forall A B : \text{Type}, \text{GuardedFunctionalRelReification_on } A B).$

Notation OmniscientRelationalChoice :=
 $(\forall A B : \text{Type}, \text{OmniscientRelationalChoice_on } A B).$

Notation OmniscientFunctionalChoice :=
 $(\forall A B : \text{Type}, \text{OmniscientFunctionalChoice_on } A B).$

Notation ConstructiveDefiniteDescription :=
 $(\forall A : \text{Type}, \text{ConstructiveDefiniteDescription_on } A).$

Notation ConstructiveIndefiniteDescription :=
 $(\forall A : \text{Type}, \text{ConstructiveIndefiniteDescription_on } A).$

Notation IotaStatement :=
 $(\forall A : \text{Type}, \text{IotaStatement_on } A).$

Notation EpsilonStatement :=
 $(\forall A : \text{Type}, \text{EpsilonStatement_on } A).$

Subclassical schemes

PI = proof irrelevance **Definition ProofIrrelevance** :=

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

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

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

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

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

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

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

Extensional schemes

Ext_prop_repr = choice of a representative among extensional propositions

Ext_pred_repr = choice of a representative among extensional predicates

Ext_fun_repr = choice of a representative among extensional functions

We let also

- IPL₂ = 2nd-order impredicative minimal predicate logic (with ex. quant.)
- IPL² = 2nd-order functional minimal predicate logic (with ex. quant.)
- IPL₂² = 2nd-order impredicative, 2nd-order functional minimal pred. logic (with ex. quant.)

with no prerequisite on the non-emptiness of domains

36.3 Table of contents

1. Definitions

2. IPL₂² |- AC_{rel} + AC! = AC_{fun}

3.1. typed IPL₂ + Sigma-types + PI |- AC_{rel} = GAC_{rel} and IPL₂ |- AC_{rel} + IGP -> GAC_{rel} and IPL₂ |- GAC_{rel} = OAC_{rel}

3.2. IPL² |- AC_{fun} + IGP = GAC_{fun} = OAC_{fun} = AC_{fun} + Drinker

3.3. D_{iota} -> ID_{iota} and D_{epsilon} <-> ID_{epsilon} + Drinker

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

5. AC_{fun} = AC_{fun}_dep = AC_{trunc}

6. Non contradiction of constructive descriptions wrt functional choices

7. Definite description transports classical logic to the computational world

8. Choice -> Dependent choice -> Countable choice

9.1. AC_{fun}_setoid = AC_{fun} + Ext_{fun}_repr + EM

9.2. AC_{fun}_setoid = AC_{fun} + Ext_{pred}_repr + PI

36.4 AC_rel + AC! = AC_fun

We show that the functional formulation of the axiom of Choice (usual formulation in type theory) is equivalent to its relational formulation (only formulation of set theory) + functional relation reification (aka axiom of unique choice, or, principle of (parametric) definite descriptions)

This shows that the axiom of choice can be assumed (under its relational formulation) without known inconsistency with classical logic, though functional relation reification conflicts with classical logic

Lemma `functional_rel_reification_and_rel_choice_imp_fun_choice` :

$\forall A B : \text{Type},$
`FunctionalRelReification_on A B` \rightarrow `RelationalChoice_on A B` \rightarrow `FunctionalChoice_on A B`.

Lemma `fun_choice_imp_rel_choice` :

$\forall A B : \text{Type},$ `FunctionalChoice_on A B` \rightarrow `RelationalChoice_on A B`.

Lemma `fun_choice_imp_functional_rel_reification` :

$\forall A B : \text{Type},$ `FunctionalChoice_on A B` \rightarrow `FunctionalRelReification_on A B`.

Corollary `fun_choice_iff_rel_choice_and_functional_rel_reification` :

$\forall A B : \text{Type},$ `FunctionalChoice_on A B` \leftrightarrow
`RelationalChoice_on A B` \wedge `FunctionalRelReification_on A B`.

36.5 Connection between the guarded, non guarded and omniscient choices

We show that the guarded formulations of the axiom of choice are equivalent to their “omniscient” variant and comes from the non guarded formulation in presence either of the independence of general premises or subset types (themselves derivable from subtypes thanks to proof-irrelevance)

36.5.1 AC_rel + PI \rightarrow GAC_rel and AC_rel + IGP \rightarrow GAC_rel and GAC_rel = OAC_rel

Lemma `rel_choice_and_proof_irrel_imp_guarded_rel_choice` :

`RelationalChoice` \rightarrow `ProofIrrelevance` \rightarrow `GuardedRelationalChoice`.

Lemma `rel_choice_indep_of_general_premises_imp_guarded_rel_choice` :

$\forall A B : \text{Type},$ `inhabited B` \rightarrow `RelationalChoice_on A B` \rightarrow
`IndependenceOfGeneralPremises` \rightarrow `GuardedRelationalChoice_on A B`.

Lemma `guarded_rel_choice_imp_rel_choice` :

$\forall A B : \text{Type},$ `GuardedRelationalChoice_on A B` \rightarrow `RelationalChoice_on A B`.

Lemma `subset_types_imp_guarded_rel_choice_iff_rel_choice` :

`ProofIrrelevance` \rightarrow (`GuardedRelationalChoice` \leftrightarrow `RelationalChoice`).

`OAC_rel` = `GAC_rel`

Corollary `guarded_iff_omniscient_rel_choice` :

`GuardedRelationalChoice` \leftrightarrow `OmniscientRelationalChoice`.

36.5.2 AC_fun + IGP = GAC_fun = OAC_fun = AC_fun + Drinker

AC_fun + IGP = GAC_fun

Lemma guarded_fun_choice_imp_indep_of_general_premises :

GuardedFunctionalChoice \rightarrow IndependenceOfGeneralPremises.

Lemma guarded_fun_choice_imp_fun_choice :

GuardedFunctionalChoice \rightarrow FunctionalChoiceOnInhabitedSet.

Lemma fun_choice_and_indep_general_prem_imp_guarded_fun_choice :

FunctionalChoiceOnInhabitedSet \rightarrow IndependenceOfGeneralPremises
 \rightarrow GuardedFunctionalChoice.

Corollary fun_choice_and_indep_general_prem_iff_guarded_fun_choice :

FunctionalChoiceOnInhabitedSet \wedge IndependenceOfGeneralPremises
 \leftrightarrow GuardedFunctionalChoice.

AC_fun + Drinker = OAC_fun

This was already observed by Bell [*Bell*]

Lemma omniscient_fun_choice_imp_small_drinker :

OmniscientFunctionalChoice \rightarrow SmallDrinker'sParadox.

Lemma omniscient_fun_choice_imp_fun_choice :

OmniscientFunctionalChoice \rightarrow FunctionalChoiceOnInhabitedSet.

Lemma fun_choice_and_small_drinker_imp_omniscient_fun_choice :

FunctionalChoiceOnInhabitedSet \rightarrow SmallDrinker'sParadox
 \rightarrow OmniscientFunctionalChoice.

Corollary fun_choice_and_small_drinker_iff_omniscient_fun_choice :

FunctionalChoiceOnInhabitedSet \wedge SmallDrinker'sParadox
 \leftrightarrow OmniscientFunctionalChoice.

OAC_fun = GAC_fun

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

Theorem guarded_iff_omniscient_fun_choice :

GuardedFunctionalChoice \leftrightarrow OmniscientFunctionalChoice.

36.5.3 D_iota \rightarrow ID_iota and D_epsilon \leftrightarrow ID_epsilon + Drinker

D_iota \rightarrow ID_iota

Lemma iota_imp_constructive_definite_description :

IotaStatement \rightarrow ConstructiveDefiniteDescription.

ID_epsilon + Drinker \leftrightarrow D_epsilon

Lemma epsilon_imp_constructive_indefinite_description:

EpsilonStatement \rightarrow ConstructiveIndefiniteDescription.

Lemma constructive_indefinite_description_and_small_drinker_imp_epsilon :

SmallDrinker'sParadox \rightarrow ConstructiveIndefiniteDescription \rightarrow
EpsilonStatement.

Lemma `epsilon_imp_small_drinker` :
EpsilonStatement \rightarrow SmallDrinker'sParadox.

Theorem `constructive_indefinite_description_and_small_drinker_iff_epsilon` :
(SmallDrinker'sParadox \times ConstructiveIndefiniteDescription \rightarrow
EpsilonStatement) \times
(EpsilonStatement \rightarrow
SmallDrinker'sParadox \times ConstructiveIndefiniteDescription).

36.6 Derivability of choice for decidable relations with well-ordered codomain

Countable codomains, such as `nat`, can be equipped with a well-order, which implies the existence of a least element on inhabited decidable subsets. As a consequence, the relational form of the axiom of choice is derivable on `nat` for decidable relations.

We show instead that functional relation reification and the functional form of the axiom of choice are equivalent on decidable relation with `nat` as codomain

Require Import Wf_nat.

Require Import Decidable.

Lemma `classical_denumerable_description_imp_fun_choice` :

$\forall A:\text{Type}$,
FunctionalRelReification_on A nat \rightarrow
 $\forall R:A \rightarrow \text{nat} \rightarrow \text{Prop}$,
($\forall x y$, decidable (R x y)) \rightarrow FunctionalChoice_on_rel R.

36.7 AC_fun = AC_fun_dep = AC_trunc

36.7.1 Choice on dependent and non dependent function types are equivalent

The easy part

Theorem `dep_non_dep_functional_choice` :
DependentFunctionalChoice \rightarrow FunctionalChoice.

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

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

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

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

Theorem `non_dep_dep_functional_choice` :
FunctionalChoice \rightarrow DependentFunctionalChoice.

36.7.2 Functional choice and truncation choice are equivalent

Theorem `functional_choice_to_inhabited_forall_commute` :
FunctionalChoice \rightarrow InhabitedForallCommute.

Theorem `inhabited_forall_commute_to_functional_choice` :
InhabitedForallCommute \rightarrow FunctionalChoice.

36.7.3 Reification of dependent and non dependent functional relation are equivalent

The easy part

Theorem `dep_non_dep_functional_rel_reification` :
DependentFunctionalRelReification \rightarrow FunctionalRelReification.

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

Theorem `non_dep_dep_functional_rel_reification` :
FunctionalRelReification \rightarrow DependentFunctionalRelReification.

Corollary `dep_iff_non_dep_functional_rel_reification` :
FunctionalRelReification \leftrightarrow DependentFunctionalRelReification.

36.8 Non contradiction of constructive descriptions wrt functional axioms of choice

36.8.1 Non contradiction of indefinite description

Lemma `relative_non_contradiction_of_indefinite_descr` :
 $\forall C:\text{Prop}, (\text{ConstructiveIndefiniteDescription} \rightarrow C)$
 $\rightarrow (\text{FunctionalChoice} \rightarrow C)$.

Lemma `constructive_indefinite_descr_fun_choice` :
ConstructiveIndefiniteDescription \rightarrow FunctionalChoice.

36.8.2 Non contradiction of definite description

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

Lemma `constructive_definite_descr_fun_reification` :
ConstructiveDefiniteDescription \rightarrow FunctionalRelReification.

Remark, the following corollaries morally hold:

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

Corollary `constructive_definite_descr_in_prop_context_iff_fun_reification` : In_propositional_context
ConstructiveIndefiniteDescription \leftrightarrow FunctionalChoice.

Corollary `constructive_definite_descr_in_prop_context_iff_fun_reification` : `In_propositional_context ConstructiveDefiniteDescription <-> 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.

36.9 Excluded-middle + definite description \Rightarrow computational excluded-middle

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

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

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

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

Require Import Setoid.

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

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

36.10 Choice \Rightarrow Dependent choice \Rightarrow Countable choice

Require Import Arith.

Theorem `functional_choice_imp_functional_dependent_choice` :
`FunctionalChoice` \rightarrow `FunctionalDependentChoice`.

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

36.11 About the axiom of choice over setoids

Require Import ClassicalFacts PropExtensionalityFacts.

36.11.1 Consequences of the choice of a representative in an equivalence class

Theorem `repr_fun_choice_imp_ext_prop_repr` :
`RepresentativeFunctionalChoice` \rightarrow `ExtensionalPropositionRepresentative`.

Theorem repr_fun_choice_imp_ext_pred_repr :
RepresentativeFunctionalChoice \rightarrow ExtensionalPredicateRepresentative.

Theorem repr_fun_choice_imp_ext_function_repr :
RepresentativeFunctionalChoice \rightarrow ExtensionalFunctionRepresentative.

This is a variant of Diaconescu and Goodman-Myhill theorems

Theorem repr_fun_choice_imp_excluded_middle :
RepresentativeFunctionalChoice \rightarrow ExcludedMiddle.

Theorem repr_fun_choice_imp_relational_choice :
RepresentativeFunctionalChoice \rightarrow RelationalChoice.

36.11.2 AC_fun_setoid = AC_fun_setoid_gen = AC_fun_setoid_simple

Theorem gen_setoid_fun_choice_imp_setoid_fun_choice :
 $\forall A B$, GeneralizedSetoidFunctionalChoice_on $A B \rightarrow$ SetoidFunctionalChoice_on $A B$.

Theorem setoid_fun_choice_imp_gen_setoid_fun_choice :
 $\forall A B$, SetoidFunctionalChoice_on $A B \rightarrow$ GeneralizedSetoidFunctionalChoice_on $A B$.

Corollary setoid_fun_choice_iff_gen_setoid_fun_choice :
 $\forall A B$, SetoidFunctionalChoice_on $A B \leftrightarrow$ GeneralizedSetoidFunctionalChoice_on $A B$.

Theorem setoid_fun_choice_imp_simple_setoid_fun_choice :
 $\forall A B$, SetoidFunctionalChoice_on $A B \rightarrow$ SimpleSetoidFunctionalChoice_on $A B$.

Theorem simple_setoid_fun_choice_imp_setoid_fun_choice :
 $\forall A B$, SimpleSetoidFunctionalChoice_on $A B \rightarrow$ SetoidFunctionalChoice_on $A B$.

Corollary setoid_fun_choice_iff_simple_setoid_fun_choice :
 $\forall A B$, SetoidFunctionalChoice_on $A B \leftrightarrow$ SimpleSetoidFunctionalChoice_on $A B$.

36.11.3 AC_fun_setoid = AC! + AC_fun_repr

Theorem setoid_fun_choice_imp_fun_choice :
 $\forall A B$, SetoidFunctionalChoice_on $A B \rightarrow$ FunctionalChoice_on $A B$.

Corollary setoid_fun_choice_imp_functional_rel_reification :
 $\forall A B$, SetoidFunctionalChoice_on $A B \rightarrow$ FunctionalRelReification_on $A B$.

Theorem setoid_fun_choice_imp_repr_fun_choice :
SetoidFunctionalChoice \rightarrow RepresentativeFunctionalChoice .

Theorem functional_rel_reification_and_repr_fun_choice_imp_setoid_fun_choice :
FunctionalRelReification \rightarrow RepresentativeFunctionalChoice \rightarrow SetoidFunctionalChoice.

Theorem functional_rel_reification_and_repr_fun_choice_iff_setoid_fun_choice :
FunctionalRelReification \wedge RepresentativeFunctionalChoice \leftrightarrow SetoidFunctionalChoice.

Note: What characterization to give of RepresentativeFunctionalChoice? A formulation of it as a functional relation would certainly be equivalent to the formulation of SetoidFunctionalChoice as a functional relation, but in their functional forms, SetoidFunctionalChoice seems strictly stronger

36.12 AC_fun_setoid = AC_fun + Ext_fun_repr + EM

Import *EqNotations*.

36.12.1 This is the main theorem in [*Carlström04*]

Note: all ingredients have a computational meaning when taken in separation. However, to compute with the functional choice, existential quantification has to be thought as a strong existential, which is incompatible with the computational content of excluded-middle

Theorem `fun_choice_and_ext_functions_repr_and_excluded_middle_imp_setoid_fun_choice` :

`FunctionalChoice` \rightarrow `ExtensionalFunctionRepresentative` \rightarrow `ExcludedMiddle` \rightarrow `RepresentativeFunctionalChoice`.

Theorem `setoid_functional_choice_first_characterization` :

`FunctionalChoice` \wedge `ExtensionalFunctionRepresentative` \wedge `ExcludedMiddle` \leftrightarrow `SetoidFunctionalChoice`.

36.12.2 AC_fun_setoid = AC_fun + Ext_pred_repr + PI

Note: all ingredients have a computational meaning when taken in separation. However, to compute with the functional choice, existential quantification has to be thought as a strong existential, which is incompatible with proof-irrelevance which requires existential quantification to be truncated

Theorem `fun_choice_and_ext_pred_ext_and_proof_irrel_imp_setoid_fun_choice` :

`FunctionalChoice` \rightarrow `ExtensionalPredicateRepresentative` \rightarrow `ProofIrrelevance` \rightarrow `RepresentativeFunctionalChoice`.

Theorem `setoid_functional_choice_second_characterization` :

`FunctionalChoice` \wedge `ExtensionalPredicateRepresentative` \wedge `ProofIrrelevance` \leftrightarrow `SetoidFunctionalChoice`.

36.13 Compatibility notations

Notation `description_rel_choice_imp_func_choice` :=

`functional_rel_reification_and_rel_choice_imp_func_choice` (*only parsing*).

Notation `func_choice_imp_rel_choice` := `fun_choice_imp_rel_choice` (*only parsing*).

Notation `FunChoice_Equiv_RelChoice_and_ParamDefinDescr` :=

`fun_choice_iff_rel_choice_and_functional_rel_reification` (*only parsing*).

Notation `func_choice_imp_description` := `fun_choice_imp_functional_rel_reification` (*only parsing*).

Chapter 37

Library `Coq.Logic.Classical`

Classical Logic

`Require Export` Classical_Prop.

`Require Export` Classical_Pred_Type.

Chapter 38

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 $true \neq false$ in `Set`.

```
Require Export ClassicalUniqueChoice.
```

```
Require Export RelationalChoice.
```

```
Require Import ChoiceFacts.
```

```
Set Implicit Arguments.
```

```
Definition subset (U:Type) (P Q:U→Prop) : Prop :=  $\forall x, P x \rightarrow Q x$ .
```

```
Theorem singleton_choice :
```

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

```
Theorem choice :
```

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

Chapter 39

Library `Coq.Logic.ClassicalDescription`

This file provides classical logic and definite description, which is equivalent to providing classical logic and Church's iota operator

Classical logic and definite descriptions implies excluded-middle in `Set` and leads to a classical world populated with non computable functions. It conflicts with the impredicativity of `Set`

`Set Implicit Arguments.`

`Require Export Classical. Require Export Description. Require Import ChoiceFacts.`

The idea for the following proof comes from *ChicliPottierSimpson02*

Theorem `excluded_middle_informative` : $\forall P:\text{Prop}, \{P\} + \{\neg P\}$.

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

Church's iota operator

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

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

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

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

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

Compatibility lemmas

`Unset Implicit Arguments.`

Definition `dependent_description` := `dependent_unique_choice`.

Definition `description` := `unique_choice`.

Chapter 40

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` :

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

Opaque `epsilon`.

Weaker lemmas (compatibility lemmas)

Theorem `choice` :

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

Chapter 41

Library `Coq.Logic.ClassicalFacts`

Some facts and definitions about classical logic

Table of contents:

1. Propositional degeneracy = excluded-middle + propositional extensionality
2. Classical logic and proof-irrelevance
 - 2.1. CC |- prop. ext. + A inhabited -> (A = A->A) -> A has fixpoint
 - 2.2. CC |- prop. ext. + dep elim on bool -> proof-irrelevance
 - 2.3. CIC |- prop. ext. -> proof-irrelevance
 - 2.4. CC |- excluded-middle + dep elim on bool -> proof-irrelevance
 - 2.5. CIC |- excluded-middle -> proof-irrelevance
3. Weak classical axioms
 - 3.1. Weak excluded middle
 - 3.2. Gödel-Dummett axiom and right distributivity of implication over disjunction
 - 3 3. Independence of general premises and drinker's paradox
4. Principles equivalent to classical logic
 - 4.1 Classical logic = principle of unrestricted minimization
 - 4.2 Classical logic = choice of representatives in a partition of bool

41.1 Prop degeneracy = excluded-middle + prop extensionality

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

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

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

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

We show $\text{prop_degeneracy} \leftrightarrow (\text{prop_extensionality} \wedge \text{excluded_middle})$

Lemma `prop_degen_ext` : `prop_degeneracy` \rightarrow `prop_extensionality`.

Lemma `prop_degen_em` : `prop_degeneracy` \rightarrow `excluded_middle`.

Lemma `prop_ext_em_degen` :

`prop_extensionality → excluded_middle → prop_degeneracy`.

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

Require Import PropExtensionalityFacts.

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

Lemma `provable_prop_ext` :

`prop_extensionality → provable_prop_extensionality`.

41.2 Classical logic and proof-irrelevance

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

We successively show that:

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

Lemma `prop_ext_A_eq_A_imp_A` :

`prop_extensionality → $\forall A:\text{Prop}, \text{inhabited } A \rightarrow (A \rightarrow A) = A$` .

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

{`f1` : $A \rightarrow B$; `f2` : $B \rightarrow A$; `f1_o_f2` : $\forall x:B, f1 (f2 x) = x$ }.

Lemma `prop_ext_retract_A_A_imp_A` :

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

Record `has_fixpoint` ($A:\text{Prop}$) : **Prop** :=

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

Lemma `ext_prop_fixpoint` :

`prop_extensionality → $\forall A:\text{Prop}, \text{inhabited } A \rightarrow \text{has_fixpoint } A$` .

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

41.2.2 CC |- prop_ext /\ dep elim on bool -> proof-irrelevance

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

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

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

Section `Proof_irrelevance_gen`.

Variable `bool` : **Prop**.

Variable `true` : `bool`.

Variable `false` : `bool`.

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

Hypothesis

$bool_elim_redl : \forall (C:\mathbf{Prop}) (c1\ c2:C), c1 = bool_elim\ C\ c1\ c2\ true.$

Hypothesis

$bool_elim_redr : \forall (C:\mathbf{Prop}) (c1\ c2:C), c2 = bool_elim\ C\ c1\ c2\ false.$

Let $bool_dep_induction :=$

$\forall P:bool \rightarrow \mathbf{Prop}, P\ true \rightarrow P\ false \rightarrow \forall b:bool, P\ b.$

Lemma $aux : prop_extensionality \rightarrow bool_dep_induction \rightarrow true = false.$

Lemma $ext_prop_dep_proof_irrel_gen :$

$prop_extensionality \rightarrow bool_dep_induction \rightarrow proof_irrelevance.$

End $Proof_irrelevance_gen.$

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

Section $Proof_irrelevance_Prop_Ext_CC.$

Definition $BoolP := \forall C:\mathbf{Prop}, C \rightarrow C \rightarrow C.$

Definition $TrueP : BoolP := \mathbf{fun}\ C\ c1\ c2 \Rightarrow c1.$

Definition $FalseP : BoolP := \mathbf{fun}\ C\ c1\ c2 \Rightarrow c2.$

Definition $BoolP_elim\ C\ c1\ c2\ (b:BoolP) := b\ C\ c1\ c2.$

Definition $BoolP_elim_redl\ (C:\mathbf{Prop})\ (c1\ c2:C) :$

$c1 = BoolP_elim\ C\ c1\ c2\ TrueP := eq_refl\ c1.$

Definition $BoolP_elim_redr\ (C:\mathbf{Prop})\ (c1\ c2:C) :$

$c2 = BoolP_elim\ C\ c1\ c2\ FalseP := eq_refl\ c2.$

Definition $BoolP_dep_induction :=$

$\forall P:BoolP \rightarrow \mathbf{Prop}, P\ TrueP \rightarrow P\ FalseP \rightarrow \forall b:BoolP, P\ b.$

Lemma $ext_prop_dep_proof_irrel_cc :$

$prop_extensionality \rightarrow BoolP_dep_induction \rightarrow proof_irrelevance.$

End $Proof_irrelevance_Prop_Ext_CC.$

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

41.2.3 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 $boolP : \mathbf{Prop} :=$

| $trueP : boolP$

| $falseP : boolP.$

Definition $boolP_elim_redl\ (C:\mathbf{Prop})\ (c1\ c2:C) :$

$c1 = boolP_ind\ C\ c1\ c2\ trueP := eq_refl\ c1.$

Definition $boolP_elim_redr\ (C:\mathbf{Prop})\ (c1\ c2:C) :$

$c2 = boolP_ind\ C\ c1\ c2\ falseP := eq_refl\ c2.$

Scheme $boolP_indd := \mathbf{Induction\ for}\ boolP\ \mathbf{Sort}\ \mathbf{Prop}.$

Lemma `ext_prop_dep_proof_irrel_cic` : `prop_extensionality` \rightarrow `proof_irrelevance`.

End `Proof_irrelevance_CIC`.

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

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

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

41.2.4 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` \rightarrow `Prop` \rightarrow `Prop`.

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

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

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

Hypothesis

`or_elim_redl` :

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

Hypothesis

`or_elim_redr` :

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

Hypothesis

`or_dep_elim` :

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

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

Variable `B` : `Prop`.

Variables `b1 b2` : `B`.

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

Let `p2b` `A` := `or_elim` `A` `($\neg A$)` `B` `(fun _ \Rightarrow b1)` `(fun _ \Rightarrow b2)` `(em A)`.

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

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

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

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

Theorem $\text{proof_irrelevance_cc} : b1 = b2$.

End $\text{Proof_irrelevance_EM_CC}$.

Hurkens' paradox still holds with a retract from the *negative* fragment of Prop into bool , hence weak classical logic, i.e. $\forall A, \neg A \setminus \sim \sim A$, is enough for deriving a weak version of proof-irrelevance. This is enough to derive a contradiction from a Set -bound weak excluded middle with an impredicative Set universe.

Section $\text{Proof_irrelevance_WEM_CC}$.

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

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

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

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

Hypothesis

$or_elim_redl :$

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

Hypothesis

$or_elim_redr :$

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

Hypothesis

$or_dep_elim :$

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

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

Variable $B : \text{Prop}$.

Variables $b1\ b2 : B$.

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

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

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

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

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

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

Theorem $\text{wproof_irrelevance_cc} : \sim \sim (b1 = b2)$.

End $\text{Proof_irrelevance_WEM_CC}$.

41.2.5 CIC |- excluded-middle -> proof-irrelevance

Since, dependent elimination is derivable in the Calculus of Inductive Constructions (CCI), we get proof-irrelevance from classical logic in the CCI.

Section Proof_irrelevance_CCI.

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

Definition *or_elim_redl* (*A B C*:Prop) (*f*:*A* → *C*) (*g*:*B* → *C*)
 (*a*:*A*) : *f a* = *or_ind f g (or_introl B a)* := *eq_refl (f a)*.

Definition *or_elim_redr* (*A B C*:Prop) (*f*:*A* → *C*) (*g*:*B* → *C*)
 (*b*:*B*) : *g b* = *or_ind f g (or_intror A b)* := *eq_refl (g b)*.

Scheme *or_indd* := **Induction for or** Sort Prop.

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

End Proof_irrelevance_CCI.

The same holds with weak excluded middle. The proof is a little more involved, however.

Section Weak_proof_irrelevance_CCI.

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

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

End Weak_proof_irrelevance_CCI.

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

41.3 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

41.3.1 Weak excluded-middle

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

Definition *weak_excluded_middle* :=

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

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

Definition *weak_generalized_excluded_middle* :=

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

41.3.2 Gödel-Dummett axiom

$(A \rightarrow B) \vee (B \rightarrow A)$ is studied in [Dummett59] and is based on [Gödel33].

[Dummett59] Michael A. E. Dummett. “A Propositional Calculus with a Denumerable Matrix”, In the Journal of Symbolic Logic, Vol 24 No. 2(1959), pp 97-103.

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

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

Lemma excluded_middle_Godel_Dummett : excluded_middle \rightarrow GodelDummett.

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

Definition RightDistributivityImplicationOverDisjunction :=

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

Lemma Godel_Dummett_iff_right_distr_implication_over_disjunction :

GodelDummett \leftrightarrow RightDistributivityImplicationOverDisjunction.

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

Lemma Godel_Dummett_weak_excluded_middle :

GodelDummett \rightarrow weak_excluded_middle.

41.3.3 Independence of general premises and drinker’s paradox

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

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

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

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

Definition IndependenceOfGeneralPremises :=

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

Lemma

independence_general_premises_right_distr_implication_over_disjunction :
IndependenceOfGeneralPremises \rightarrow RightDistributivityImplicationOverDisjunction.

Lemma independence_general_premises_Godel_Dummett :

IndependenceOfGeneralPremises \rightarrow GodelDummett.

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

Definition DrinkerParadox :=

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

Lemma independence_general_premises_drinker :

IndependenceOfGeneralPremises \leftrightarrow DrinkerParadox.

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

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

Definition generalized_excluded_middle :=

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

Lemma excluded_middle_independence_general_premises :
generalized_excluded_middle \rightarrow DrinkerParadox.

41.4 Axioms equivalent to classical logic

41.4.1 Principle of unrestricted minimization

Require Import Coq.Arith.PeanoNat.

Definition Minimal ($P:\text{nat} \rightarrow \text{Prop}$) ($n:\text{nat}$) : **Prop** :=

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

Definition Minimization_Property ($P : \text{nat} \rightarrow \text{Prop}$) : **Prop** :=

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

Section Unrestricted_minimization_entails_excluded_middle.

Hypothesis unrestricted_minimization: $\forall P, \text{Minimization_Property}\ P$.

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

End Unrestricted_minimization_entails_excluded_middle.

Require Import Wf_nat.

Section Excluded_middle_entails_unrestricted_minimization.

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

Theorem excluded_middle_entails_unrestricted_minimization :

$\forall P, \text{Minimization_Property}\ P$.

End Excluded_middle_entails_unrestricted_minimization.

However, minimization for a given predicate does not necessarily imply decidability of this predicate

Section Example_of_undecidable_predicate_with_the_minimization_property.

Variable $s : \text{nat} \rightarrow \text{bool}$.

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

Example undecidable_predicate_with_the_minimization_property :

$\text{Minimization_Property}\ P$.

End Example_of_undecidable_predicate_with_the_minimization_property.

41.4.2 Choice of representatives in a partition of bool

This is similar to Bell’s “weak extensional selection principle” in [Bell]

[*Bell*] John L. Bell, Choice principles in intuitionistic set theory, unpublished.

Require Import RelationClasses.

Theorem representative_boolean_partition_imp_excluded_middle :
representative_boolean_partition \rightarrow excluded_middle.

Theorem excluded_middle_imp_representative_boolean_partition :
excluded_middle \rightarrow representative_boolean_partition.

Theorem excluded_middle_iff_representative_boolean_partition :
excluded_middle \leftrightarrow representative_boolean_partition.

Chapter 42

Library

Coq.Logic.ClassicalUniqueChoice

This file provides classical logic and unique choice; this is weaker than providing iota operator and classical logic as the definite descriptions provided by the axiom of unique choice can be used only in a propositional context (especially, they cannot be used to build functions outside the scope of a theorem proof)

Classical logic and unique choice, as shown in [*ChicliPottierSimpson02*], implies the double-negation of excluded-middle in **Set**, hence it implies a strongly classical world. Especially it conflicts with the impredicativity of **Set**.

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

Require Export Classical.

Axiom

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

Unique choice reifies functional relations into functions

Theorem *unique_choice* :

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

The following proof comes from [*ChicliPottierSimpson02*] **Require Import** Setoid.

Theorem *classic_set_in_prop_context* :

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

Corollary *not_not_classic_set* :

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

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

Chapter 43

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 44

Library `Coq.Logic.Classical_Prop`

Classical Propositional Logic

```
Require Import ClassicalFacts.
```

```
Hint Unfold not: core.
```

```
Axiom classic :  $\forall P:\text{Prop}, P \vee \neg P$ .
```

```
Lemma NNPP :  $\forall p:\text{Prop}, \neg \neg p \rightarrow p$ .
```

Peirce's law states $\forall P Q:\text{Prop}, ((P \rightarrow Q) \rightarrow P) \rightarrow P$. Thanks to $\forall P, \text{False} \rightarrow P$, it is equivalent to the following form

```
Lemma Peirce :  $\forall P:\text{Prop}, ((P \rightarrow \text{False}) \rightarrow P) \rightarrow P$ .
```

```
Lemma not_imply_elim :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow P$ .
```

```
Lemma not_imply_elim2 :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow \neg Q$ .
```

```
Lemma imply_to_or :  $\forall P Q:\text{Prop}, (P \rightarrow Q) \rightarrow \neg P \vee Q$ .
```

```
Lemma imply_to_and :  $\forall P Q:\text{Prop}, \neg (P \rightarrow Q) \rightarrow P \wedge \neg Q$ .
```

```
Lemma or_to_imply :  $\forall P Q:\text{Prop}, \neg P \vee Q \rightarrow P \rightarrow Q$ .
```

```
Lemma not_and_or :  $\forall P Q:\text{Prop}, \neg (P \wedge Q) \rightarrow \neg P \vee \neg Q$ .
```

```
Lemma or_not_and :  $\forall P Q:\text{Prop}, \neg P \vee \neg Q \rightarrow \neg (P \wedge Q)$ .
```

```
Lemma not_or_and :  $\forall P Q:\text{Prop}, \neg (P \vee Q) \rightarrow \neg P \wedge \neg Q$ .
```

```
Lemma and_not_or :  $\forall P Q:\text{Prop}, \neg P \wedge \neg Q \rightarrow \neg (P \vee Q)$ .
```

```
Lemma imply_and_or :  $\forall P Q:\text{Prop}, (P \rightarrow Q) \rightarrow P \vee Q \rightarrow Q$ .
```

```
Lemma imply_and_or2 :  $\forall P Q R:\text{Prop}, (P \rightarrow Q) \rightarrow P \vee R \rightarrow Q \vee R$ .
```

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

```
Ltac classical_right := match goal with  
| ?X  $\vee$  _  $\Rightarrow$  (elim (classic X);intro;[left;trivial|right])  
end.
```

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

```
end.  
Require Export EqdepFacts.  
Module EQ_RECT_EQ.  
Lemma eq_rect_eq :  
   $\forall (U:\text{Type}) (p:U) (Q:U \rightarrow \text{Type}) (x:Q p) (h:p = p), x = \text{eq\_rect } p \ Q \ x \ p \ h.$   
End EQ_RECT_EQ.  
Module EQDEPTHEORY := EQDEPTHEORY(EQ_RECT_EQ).  
Export EqdepTheory.
```

Chapter 45

Library `Coq.Logic.ConstructiveEpsilon`

This provides with a proof of the constructive form of definite and indefinite descriptions for Σ^0_1 -formulas (hereafter called “small” formulas), which infers the sigma-existence (i.e., `Type`-existence) of a witness to a decidable predicate over a countable domain from the regular existence (i.e., `Prop`-existence).

Coq does not allow case analysis on sort `Prop` when the goal is in not in `Prop`. Therefore, one cannot eliminate $\exists n, P n$ in order to show $\{n : \text{nat} \mid P n\}$. However, one can perform a recursion on an inductive predicate in sort `Prop` so that the returning type of the recursion is in `Type`. This trick is described in Coq’Art book, Sect. 14.2.3 and 15.4. In particular, this trick is used in the proof of `Fix_F` in the module `Coq.Init.Wf`. There, recursion is done on an inductive predicate `Acc` and the resulting type is in `Type`.

To find a witness of P constructively, we program the well-known linear search algorithm that tries P on all natural numbers starting from 0 and going up. Such an algorithm needs a suitable termination certificate. We offer two ways for providing this termination certificate: a direct one, based on an ad-hoc predicate called `before_witness`, and another one based on the predicate `Acc`. For the first one we provide explicit and short proof terms.

Based on ideas from Benjamin Werner and Jean-François Monin

Contributed by Yevgeniy Makarov and Jean-François Monin

`Section ConstructiveIndefiniteGroundDescription_Direct.`

`Variable P : nat → Prop.`

`Hypothesis P_dec : ∀ 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 (n:nat) : Prop :=`

`| stop : P n → before_witness n`

`| next : before_witness (S n) → before_witness n.`

`Fixpoint O_witness (n : nat) : before_witness n → before_witness 0 :=`

`match n return (before_witness n → before_witness 0) with`

`| 0 ⇒ fun b ⇒ b`

`| S n ⇒ fun b ⇒ O_witness n (next n b)`

end.

Definition `inv_before_witness` :

```
∀ n, before_witness n → ¬(P n) → before_witness (S n) :=  
fun n b ⇒  
  match b return ¬ P n → before_witness (S n) with  
  | stop _ p ⇒ fun not_p ⇒ match (not_p p) with end  
  | next _ b ⇒ fun _ ⇒ b  
end.
```

Fixpoint `linear_search m (b : before_witness m) : {n : nat | P n} :=`

```
match P_dec m with  
| left yes ⇒ exist (fun n ⇒ P n) m yes  
| right no ⇒ linear_search (S m) (inv_before_witness m b no)  
end.
```

Definition `constructive_indefinite_ground_description_nat` :

```
(∃ n, P n) → {n : nat | P n} :=  
fun e ⇒ linear_search O (let (n, p) := e in O_witness n (stop n p)).
```

End `ConstructiveIndefiniteGroundDescription_Direct`.

Require Import `Arith`.

Section `ConstructiveIndefiniteGroundDescription_Acc`.

Variable `P : nat → Prop`.

Hypothesis `P_decidable` : $\forall n : \text{nat}, \{P n\} + \{\neg P n\}$.

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 $\{n : \text{nat} \mid P n\}$.

The relation *R* describes the connection between the two successive numbers we try. Namely, *y* is *R*-less than *x* if we try *y* after *x*, i.e., $y = S x$ and $P x$ is false. Then the absence of an infinite *R*-descending chain from 0 is equivalent to the termination of our searching algorithm.

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

Lemma `P_implies_acc` : $\forall x : \text{nat}, P x \rightarrow \text{acc } x$.

Lemma `P_eventually_implies_acc` : $\forall (x : \text{nat}) (n : \text{nat}), P (n + x) \rightarrow \text{acc } x$.

Corollary `P_eventually_implies_acc_ex` : $(\exists n : \text{nat}, P n) \rightarrow \text{acc } 0$.

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` : $\text{acc } 0 \rightarrow \{n : \text{nat} \mid P n\}$.

Theorem `constructive_indefinite_ground_description_nat_Acc` :

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

End `ConstructiveIndefiniteGroundDescription_Acc`.

Section `ConstructiveGroundEpsilon_nat`.

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

Hypothesis $P_decidable : \forall x : \text{nat}, \{P\ x\} + \{\neg P\ x\}$.

Definition $\text{constructive_ground_epsilon_nat} (E : \exists n : \text{nat}, P\ n) : \text{nat}$
:= $\text{proj1_sig} (\text{constructive_indefinite_ground_description_nat } P\ P_decidable\ E)$.

Definition $\text{constructive_ground_epsilon_spec_nat} (E : (\exists n, P\ n)) : P (\text{constructive_ground_epsilon_nat } E)$
:= $\text{proj2_sig} (\text{constructive_indefinite_ground_description_nat } P\ P_decidable\ E)$.

End ConstructiveGroundEpsilon_nat.

Section ConstructiveGroundEpsilon.

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

Variable $A : \text{Type}$.

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

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

Hypothesis $\text{gof_eq_id} : \forall x : A, g (f\ x) = x$.

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

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

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

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

Lemma $\text{constructive_indefinite_ground_description} : (\exists x : A, P\ x) \rightarrow \{x : A \mid P\ x\}$.

Lemma $\text{constructive_definite_ground_description} : (\exists! x : A, P\ x) \rightarrow \{x : A \mid P\ x\}$.

Definition $\text{constructive_ground_epsilon} (E : \exists x : A, P\ x) : A$
:= $\text{proj1_sig} (\text{constructive_indefinite_ground_description } E)$.

Definition $\text{constructive_ground_epsilon_spec} (E : (\exists x, P\ x)) : P (\text{constructive_ground_epsilon } E)$
:= $\text{proj2_sig} (\text{constructive_indefinite_ground_description } E)$.

End ConstructiveGroundEpsilon.

Chapter 46

Library `Coq.Logic.Decidable`

Properties of decidable propositions

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

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

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

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

Theorem `dec_or` :

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

Theorem `dec_and` :

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

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

Theorem `dec_imp` :

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

Theorem `dec_iff` :

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

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

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

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

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

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

Theorem `not_iff` :

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

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

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

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

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

Theorem `not_not_iff` : $\forall A:\text{Prop}, \text{decidable } A \rightarrow ((A \rightarrow \text{False}) \rightarrow \text{False}) \leftrightarrow A$.

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

Lemma `or_not_l_iff_1` : $\forall A B:\text{Prop}, \text{decidable } A \rightarrow ((A \rightarrow \text{False}) \vee B \leftrightarrow (A \rightarrow B))$.

Lemma `or_not_l_iff_2` : $\forall A B:\text{Prop}, \text{decidable } B \rightarrow ((A \rightarrow \text{False}) \vee B \leftrightarrow (A \rightarrow B))$.

Lemma `or_not_r_iff_1` : $\forall A B:\text{Prop}, \text{decidable } A \rightarrow (A \vee (B \rightarrow \text{False}) \leftrightarrow (B \rightarrow A))$.

Lemma `or_not_r_iff_2` : $\forall A B:\text{Prop}, \text{decidable } B \rightarrow (A \vee (B \rightarrow \text{False}) \leftrightarrow (B \rightarrow A))$.

Lemma `imp_not_l` : $\forall A B:\text{Prop}, \text{decidable } A \rightarrow (((A \rightarrow \text{False}) \rightarrow B) \leftrightarrow (A \vee B))$.

Moving Negations Around: We have four lemmas that, when read from left to right, describe how to push negations toward the leaves of a proposition and, when read from right to left, describe how to pull negations toward the top of a proposition.

Theorem `not_or_iff` : $\forall A B:\text{Prop}, (A \vee B \rightarrow \text{False}) \leftrightarrow (A \rightarrow \text{False}) \wedge (B \rightarrow \text{False})$.

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

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

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

Theorem `dec_functional_relation` :
 $\forall (X Y : \text{Type}) (A:X \rightarrow Y \rightarrow \text{Prop}), (\forall y y' : Y, \text{decidable } (y=y')) \rightarrow (\forall x, \exists! y, A x y) \rightarrow \forall x y, \text{decidable } (A x y)$.

With the following hint database, we can leverage `auto` to check decidability of propositions.

Hint `Resolve` `dec_True` `dec_False` `dec_or` `dec_and` `dec_imp` `dec_not` `dec_iff`
: `decidable_prop`.

`solve_decidable` `using` `lib` will solve goals about the decidability of a proposition, assisted by an auxiliary database of lemmas. The database is intended to contain lemmas stating the decidability of base propositions, (e.g., the decidability of equality on a particular inductive type).

Tactic Notation "solve_decidable" "using" `ident`(`db`) :=
`match goal with`
| `⊢ decidable _ =>`
`solve [auto 100 with decidable_prop db]`
`end.`

```
Tactic Notation "solve_decidable" :=  
  solve_decidable using core.
```

Chapter 47

Library `Coq.Logic.Description`

This file provides a constructive form of definite description; it allows building functions from the proof of their existence in any context; this is weaker than Church's iota operator

```
Require Import ChoiceFacts.
```

```
Set Implicit Arguments.
```

```
Axiom constructive_definite_description :
```

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

Chapter 48

Library `Coq.Logic.Diaconescu`

Diaconescu showed that the Axiom of Choice entails Excluded-Middle in topoi [*Diaconescu75*]. Lacas and Werner adapted the proof to show that the axiom of choice in equivalence classes entails Excluded-Middle in Type Theory [*LacasWerner99*].

Three variants of Diaconescu's result in type theory are shown below.

A. A proof that the relational form of the Axiom of Choice + Extensionality for Predicates entails Excluded-Middle (by Hugo Herbelin)

B. A proof that the relational form of the Axiom of Choice + Proof Irrelevance entails Excluded-Middle for Equality Statements (by Benjamin Werner)

C. A proof that extensional Hilbert epsilon's description operator entails excluded-middle (taken from Bell [*Bell93*])

See also [*Carlström04*] for a discussion of the connection between the Extensional Axiom of Choice and Excluded-Middle

[*Diaconescu75*] Radu Diaconescu, Axiom of Choice and Complementation, in Proceedings of AMS, vol 51, pp 176-178, 1975.

[*LacasWerner99*] Samuel Lacas, Benjamin Werner, Which Choices imply the excluded middle?, preprint, 1999.

[*Bell93*] John L. Bell, Hilbert's epsilon operator and classical logic, Journal of Philosophical Logic, 22: 1-18, 1993

[*Carlström04*] Jesper Carlström, EM + Ext + AC_int is equivalent to AC_ext, Mathematical Logic Quarterly, vol 50(3), pp 236-240, 2004. `Require ClassicalFacts ChoiceFacts`.

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

`Section PredExt_RelChoice_imp_EM.`

The axiom of extensionality for predicates

`Definition PredicateExtensionality :=`

`∀ P Q:bool → Prop, (∀ b:bool, P b ↔ Q b) → P = Q.`

From predicate extensionality we get propositional extensionality hence proof-irrelevance

`Import ClassicalFacts.`

`Variable pred_extensionality : PredicateExtensionality.`

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

Lemma proof_irrel : $\forall (A:\text{Prop}) (a1 a2:A), a1 = a2$.

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

Import *ChoiceFacts*.

Variable *rel_choice* : RelationalChoice.

Lemma guarded_rel_choice : GuardedRelationalChoice.

The form of choice we need: there is a functional relation which chooses an element in any non empty subset of bool

Import *Bool*.

Lemma AC_bool_subset_to_bool :

$\exists R : (\text{bool} \rightarrow \text{Prop}) \rightarrow \text{bool} \rightarrow \text{Prop},$

$(\forall P:\text{bool} \rightarrow \text{Prop},$

$(\exists b : \text{bool}, P b) \rightarrow$

$\exists b : \text{bool}, P b \wedge R P b \wedge (\forall b':\text{bool}, R P b' \rightarrow b = b'))$.

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

Theorem pred_ext_and_rel_choice_imp_EM : $\forall P:\text{Prop}, P \vee \neg P$.

End PredExt_RelChoice_imp_EM.

48.2 Proof-Irrel. + Rel. Axiom of Choice \rightarrow Excl.-Middle for Equality

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

Section ProofIrrel_RelChoice_imp_EqEM.

Import *ChoiceFacts*.

Variable *rel_choice* : RelationalChoice.

Variable *proof_irrelevance* : $\forall P:\text{Prop}, \forall x y:P, x=y$.

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

Variable *A* :Type.

Variables *a1 a2* : *A*.

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

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

Definition *a1'*:*A'*.

Defined.

Definition *a2'*:*A'*.

Defined.

By proof-irrelevance, projection is a retraction

Lemma projT1_injective : *a1*=*a2* \rightarrow *a1'*=*a2'*.

But from the actual proofs of being in A' , we can assert in the proof-irrelevant world the existence of relevant boolean witnesses

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

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

Theorem `proof_irrel_rel_choice_imp_eq_dec` : $a1=a2 \vee \neg a1=a2 .$

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

Lemma `proof_irrel_rel_choice_imp_eq_dec'` : $a1=a2 \vee \neg a1=a2 .$

End `ProofIrrel_RelChoice_imp_EqEM`.

48.3 Extensional Hilbert's epsilon description operator \rightarrow Excluded-Middle

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

Section `ExtensionalEpsilon_imp_EM`.

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

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

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

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

End `ExtensionalEpsilon_imp_EM`.

Chapter 49

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

Chapter 50

Library `Coq.Logic.Eqdep`

This file axiomatizes the invariance by substitution of reflexive equality proofs [Streicher93] and exports its consequences, such as the injectivity of the projection of the dependent pair.

[Streicher93] T. Streicher, Semantical Investigations into Intensional Type Theory, Habilitationsschrift, LMU München, 1993.

```
Require Export EqdepFacts.
```

```
Module EQ_RECT_EQ.
```

```
Axiom eq_rect_eq :
```

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

```
End EQ_RECT_EQ.
```

```
Module EQDEPTHEORY := EQDEPTHEORY(EQ_RECT_EQ).
```

```
Export EqdepTheory.
```

Exported hints

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

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

Chapter 51

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`

51.1 Definition of dependent equality and equivalence with equality of dependent pairs

`Import EqNotations.`

`Section Dependent_Equality.`

Variable $U : \text{Type}$.

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

Dependent equality

Inductive $\text{eq_dep} (p:U) (x:P p) : \forall q:U, P q \rightarrow \text{Prop} :=$

$\text{eq_dep_intro} : \text{eq_dep} p x p x.$

Hint Constructors eq_dep : core.

Lemma $\text{eq_dep_refl} : \forall (p:U) (x:P p), \text{eq_dep} p x p x.$

Lemma $\text{eq_dep_sym} :$

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

Hint Immediate eq_dep_sym : core.

Lemma $\text{eq_dep_trans} :$

$\forall (p q r:U) (x:P p) (y:P q) (z:P r),$
 $\text{eq_dep} p x q y \rightarrow \text{eq_dep} q y r z \rightarrow \text{eq_dep} p x r z.$

Scheme $\text{eq_indd} := \text{Induction for eq Sort Prop}.$

Equivalent definition of dependent equality as a dependent pair of equalities

Inductive $\text{eq_dep1} (p:U) (x:P p) (q:U) (y:P q) : \text{Prop} :=$

$\text{eq_dep1_intro} : \forall h:q = p, x = \text{rew } h \text{ in } y \rightarrow \text{eq_dep1} p x q y.$

Lemma $\text{eq_dep1_dep} :$

$\forall (p:U) (x:P p) (q:U) (y:P q), \text{eq_dep1} p x q y \rightarrow \text{eq_dep} p x q y.$

Lemma $\text{eq_dep_dep1} :$

$\forall (p q:U) (x:P p) (y:P q), \text{eq_dep} p x q y \rightarrow \text{eq_dep1} p x q y.$

End $\text{Dependent_Equality}.$

Dependent equality is equivalent to equality on dependent pairs

Lemma $\text{eq_sigT_eq_dep} :$

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

Notation $\text{eq_sigS_eq_dep} := \text{eq_sigT_eq_dep} (\text{compat "8.7"}).$

Lemma $\text{eq_dep_eq_sigT} :$

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

Lemma $\text{eq_sigT_iff_eq_dep} :$

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

Notation $\text{equiv_eqex_eqdep} := \text{eq_sigT_iff_eq_dep} (\text{only parsing}).$

Lemma $\text{eq_sig_eq_dep} :$

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

Lemma $\text{eq_dep_eq_sig} :$

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

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

Lemma `eq_sig_iff_eq_dep` :

`∀ (U:Type) (P:U → Prop) (p q:U) (x:P p) (y:P q),
 exist P p x = exist P q y ↔ eq_dep p x q y.`

Dependent equality is equivalent to a dependent pair of equalities

Set Implicit Arguments.

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

Lemma `eq_sigT_fst` :

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

Lemma `eq_sigT_snd` :

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

Lemma `eq_sig_fst` :

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

Lemma `eq_sig_snd` :

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

Unset Implicit Arguments.

Exported hints

Hint Resolve `eq_dep_intro`: *core*.

Hint Immediate `eq_dep_sym`: *core*.

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

Section `Equivalences`.

Variable `U:Type`.

Invariance by Substitution of Reflexive Equality Proofs

Definition `Eq_rect_eq_on` (`p : U`) (`Q : U → Type`) (`x : Q p`) :=

`∀ (h : p = p), x = eq_rect p Q x p h.`

Definition `Eq_rect_eq` := `∀ p Q x, Eq_rect_eq_on p Q x.`

Injectivity of Dependent Equality

Definition `Eq_dep_eq_on` (`P : U → Type`) (`p : U`) (`x : P p`) :=

`∀ (y : P p), eq_dep p x p y → x = y.`

Definition `Eq_dep_eq` := `∀ P p x, Eq_dep_eq_on P p x.`

Uniqueness of Identity Proofs (UIP)

Definition `UIP_on` (`x y : U`) (`p1 : x = y`) :=

`∀ (p2 : x = y), p1 = p2.`

Definition `UIP` := `∀ x y p1, UIP_on x y p1.`

Uniqueness of Reflexive Identity Proofs

Definition `UIP_refl_on_` ($x : U$) :=
 $\forall (p : x = x), p = \text{eq_refl } x$.

Definition `UIP_refl_` := $\forall x, \text{UIP_refl_on_ } x$.

Streicher's axiom K

Definition `Streicher_K_on_` ($x : U$) ($P : x = x \rightarrow \text{Prop}$) :=
 $P (\text{eq_refl } x) \rightarrow \forall p : x = x, P p$.

Definition `Streicher_K_` := $\forall x P, \text{Streicher_K_on_ } x P$.

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

Lemma `eq_rect_eq_on__eq_dep1_eq_on` ($p : U$) ($P : U \rightarrow \text{Type}$) ($y : P p$) :
 $\text{Eq_rect_eq_on } p P y \rightarrow \forall (x : P p), \text{eq_dep1 } p x p y \rightarrow x = y$.

Lemma `eq_rect_eq__eq_dep1_eq` :
 $\text{Eq_rect_eq} \rightarrow \forall (P : U \rightarrow \text{Type}) (p : U) (x y : P p), \text{eq_dep1 } p x p y \rightarrow x = y$.

Lemma `eq_rect_eq_on__eq_dep_eq_on` ($p : U$) ($P : U \rightarrow \text{Type}$) ($x : P p$) :
 $\text{Eq_rect_eq_on } p P x \rightarrow \text{Eq_dep_eq_on } P p x$.

Lemma `eq_rect_eq__eq_dep_eq` : $\text{Eq_rect_eq} \rightarrow \text{Eq_dep_eq}$.

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

Lemma `eq_dep_eq_on__UIP_on` ($x y : U$) ($p1 : x = y$) :
 $\text{Eq_dep_eq_on } (\text{fun } y \Rightarrow x = y) x \text{eq_refl} \rightarrow \text{UIP_on_ } x y p1$.

Lemma `eq_dep_eq__UIP` : $\text{Eq_dep_eq} \rightarrow \text{UIP_}$.

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma `UIP_on__UIP_refl_on` ($x : U$) :
 $\text{UIP_on_ } x x \text{eq_refl} \rightarrow \text{UIP_refl_on_ } x$.

Lemma `UIP__UIP_refl` : $\text{UIP_} \rightarrow \text{UIP_refl_}$.

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

Lemma `UIP_refl_on__Streicher_K_on` ($x : U$) ($P : x = x \rightarrow \text{Prop}$) :
 $\text{UIP_refl_on_ } x \rightarrow \text{Streicher_K_on_ } x P$.

Lemma `UIP_refl__Streicher_K` : $\text{UIP_refl_} \rightarrow \text{Streicher_K_}$.

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

Lemma `Streicher_K_on__eq_rect_eq_on` ($p : U$) ($P : U \rightarrow \text{Type}$) ($x : P p$) :
 $\text{Streicher_K_on_ } p (\text{fun } h \Rightarrow x = \text{rew} \rightarrow [P] h \text{ in } x) \rightarrow \text{Eq_rect_eq_on } p P x$.

Lemma `Streicher_K__eq_rect_eq` : $\text{Streicher_K_} \rightarrow \text{Eq_rect_eq}$.

Remark: It is reasonable to think that `eq_rect_eq` is strictly stronger than `eq_rec_eq` (which is `eq_rec_eq` restricted on `Set`):

Definition `Eq_rec_eq` := $\forall (P : U \rightarrow \text{Set}) (p : U) (x : P p) (h : p = p), x = \text{eq_rec } p P x p h$.

Typically, `eq_rect_eq` allows proving UIP and Streicher's K what does not seem possible with `eq_rec_eq`. In particular, the proof of `UIP` requires to use `eq_rect_eq` on `fun y → x=y` which is in `Type` but not in `Set`.

End Equivalences.

UIP_refl is downward closed (a short proof of the key lemma of Voevodsky’s proof of inclusion of h-level n into h-level n+1; see hlevelntosn in <https://github.com/vladimirias/Foundations.git>).

Theorem UIP_shift_on $(X : \text{Type}) (x : X) :$

UIP_refl_on_ $X x \rightarrow \forall y : x = x, \text{UIP_refl_on_} (x = x) y.$

Theorem UIP_shift : $\forall U, \text{UIP_refl_} U \rightarrow \forall x:U, \text{UIP_refl_} (x = x).$

Section Corollaries.

Variable $U:\text{Type}.$

UIP implies the injectivity of equality on dependent pairs in Type

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

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

Definition Inj_dep_pair := $\forall P p x, \text{Inj_dep_pair_on } P p x.$

Lemma eq_dep_eq_on_inj_pair2_on $(P : U \rightarrow \text{Type}) (p : U) (x : P p) :$

Eq_dep_eq_on $U P p x \rightarrow \text{Inj_dep_pair_on } P p x.$

Lemma eq_dep_eq_inj_pair2 : $\text{Eq_dep_eq } U \rightarrow \text{Inj_dep_pair}.$

End Corollaries.

Notation Inj_dep_pairS := Inj_dep_pair.

Notation Inj_dep_pairT := Inj_dep_pair.

Notation eq_dep_eq_inj_pairT2 := eq_dep_eq_inj_pair2.

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

Module Type EQDEPELIMINATION.

Axiom eq_rect_eq :

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

End EQDEPELIMINATION.

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

Section Axioms.

Variable $U:\text{Type}.$

Invariance by Substitution of Reflexive Equality Proofs

Lemma eq_rect_eq :

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

Lemma eq_rec_eq :

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

Injectivity of Dependent Equality

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

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

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

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

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

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

Lemma `Streicher_K` :

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

End Axioms.

UIP implies the injectivity of equality on dependent pairs in Type

Lemma `inj_pair2` :

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

Notation `inj_pairT2` := `inj_pair2`.

End `EQDEPTHEORY`.

Basic facts about `eq_dep`

Lemma `f_eq_dep` :

$\forall U (P:U \rightarrow \text{Type}) R p q x y (f:\forall p, P p \rightarrow R p),$
 $\text{eq_dep } p x q y \rightarrow \text{eq_dep } p (f p x) q (f q y)$.

Lemma `eq_dep_non_dep` :

$\forall U P p q x y, @\text{eq_dep } U (\text{fun } _ \Rightarrow P) p x q y \rightarrow x = y$.

Lemma `f_eq_dep_non_dep` :

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

Chapter 52

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`

52.1 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 : $\forall (x y:A) (u:x = y)$, comp u u = eq_refl y.`

`Variable x : A.`

`Variable eq_dec : $\forall y:A$, $x = y \vee x \neq y$.`

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

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

Qed.

Let $nu_inv (y:A) (v:x = y) : x = y := comp (nu (eq_refl x)) v$.

Remark $nu_left_inv_on : \forall (y:A) (u:x = y), nu_inv (nu u) = u$.

Theorem $eq_proofs_unicity_on : \forall (y:A) (p1 p2:x = y), p1 = p2$.

Theorem $K_dec_on :$

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

The corollary

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

```
match exP with
| ex_intro _ x' prf =>
  match eq_dec x' with
  | or_introl eqprf => eq_ind x' P prf x (eq_sym eqprf)
  | _ => def
end
end.
```

Theorem $inj_right_pair_on :$

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

End EqdepDec.

Now we prove the versions that require decidable equality for the entire type rather than just on the given element. The rest of the file uses this total decidable equality. We could do everything using decidable equality at a point (because the induction rule for eq is really an induction rule for $\{ y : A \mid x = y \}$), but we don't currently, because changing everything would break backward compatibility and no-one has yet taken the time to define the pointed versions, and then re-define the non-pointed versions in terms of those.

Theorem $eq_proofs_unicity A (eq_dec : \forall x y : A, x = y \vee x \neq y) (x : A)$
 $: \forall (y:A) (p1 p2:x = y), p1 = p2$.

Theorem $K_dec A (eq_dec : \forall x y : A, x = y \vee x \neq y) (x : A)$
 $: \forall P:x = x \rightarrow Prop, P (eq_refl x) \rightarrow \forall p:x = x, P p$.

Theorem $inj_right_pair A (eq_dec : \forall x y : A, x = y \vee x \neq y) (x : A)$
 $: \forall (P:A \rightarrow Prop) (y y':P x),$
 $ex_intro P x y = ex_intro P x y' \rightarrow y = y'$.

Require Import EqdepFacts.

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

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

Theorem $K_dec_set :$

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

We deduce the *eq_rect_eq* axiom for (decidable) types **Theorem** `eq_rect_eq_dec` :

```

∀ A:Type,
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (p:A) (Q:A → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

```

We deduce the injectivity of dependent equality for decidable types **Theorem** `eq_dep_eq_dec` :

```

∀ A:Type,
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (P:A→Type) (p:A) (x y:P p), eq_dep A P p x p y → x = y.

```

Theorem `UIP_dec` :

```

∀ (A:Type),
  (∀ x y:A, {x = y} + {x ≠ y}) →
  ∀ (x y:A) (p1 p2:x = y), p1 = p2.

```

Unset Implicit Arguments.

52.1.1 Definition of the functor that builds properties of dependent equalities on decidable sets in `Type`

The signature of decidable sets in `Type`

```

Module Type DECIDABLETYPE.

```

```

  Axiom eq_dec : ∀ x y:U, {x = y} + {x ≠ y}.

```

```

End DECIDABLETYPE.

```

The module *DecidableEqDep* collects equality properties for decidable set in `Type`

```

Module DECIDABLEEQDEP (M:DECIDABLETYPE).

```

```

  Import M.

```

Invariance by Substitution of Reflexive Equality Proofs

```

Lemma eq_rect_eq :

```

```

  ∀ (p:U) (Q:U → Type) (x:Q p) (h:p = p), x = eq_rect p Q x p h.

```

Injectivity of Dependent Equality

```

Theorem eq_dep_eq :

```

```

  ∀ (P:U→Type) (p:U) (x y:P p), eq_dep U P p x p y → x = y.

```

Uniqueness of Identity Proofs (UIP)

```

Lemma UIP : ∀ (x y:U) (p1 p2:x = y), p1 = p2.

```

Uniqueness of Reflexive Identity Proofs

```

Lemma UIP_refl : ∀ (x:U) (p:x = x), p = eq_refl x.

```

Streicher's axiom K

```

Lemma Streicher_K :

```

```

  ∀ (x:U) (P:x = x → Prop), P (eq_refl x) → ∀ p:x = x, P p.

```

Injectivity of equality on dependent pairs in `Type`

```

Lemma inj_pairT2 :

```

$\forall (P:U \rightarrow \text{Type}) (p:U) (x y:P p),$
 $\text{existT } P p x = \text{existT } P p y \rightarrow x = y.$

Proof-irrelevance on subsets of decidable sets

Lemma `inj_pairP2` :

$\forall (P:U \rightarrow \text{Prop}) (x:U) (p q:P x),$
 $\text{ex_intro } P x p = \text{ex_intro } P x q \rightarrow p = q.$

End `DECIDABLEEQDEP`.

52.1.2 Definition of the functor that builds properties of dependent equalities on decidable sets in `Set`

The signature of decidable sets in `Set`

Module `Type` `DECIDABLESET`.

Parameter `U:Set`.

Axiom `eq_dec` : $\forall x y:U, \{x = y\} + \{x \neq y\}.$

End `DECIDABLESET`.

The module `DecidableEqDepSet` collects equality properties for decidable set in `Set`

Module `DECIDABLEEQDEPSET` (`M:DECIDABLESET`).

Import `M`.

Module `N:=DECIDABLEEQDEP(M)`.

Invariance by Substitution of Reflexive Equality Proofs

Lemma `eq_rect_eq` :

$\forall (p:U) (Q:U \rightarrow \text{Type}) (x:Q p) (h:p = p), x = \text{eq_rect } p Q x p h.$

Injectivity of Dependent Equality

Theorem `eq_dep_eq` :

$\forall (P:U \rightarrow \text{Type}) (p:U) (x y:P p), \text{eq_dep } U P p x p y \rightarrow x = y.$

Uniqueness of Identity Proofs (UIP)

Lemma `UIP` : $\forall (x y:U) (p1 p2:x = y), p1 = p2.$

Uniqueness of Reflexive Identity Proofs

Lemma `UIP_refl` : $\forall (x:U) (p:x = x), p = \text{eq_refl } x.$

Streicher's axiom K

Lemma `Streicher_K` :

$\forall (x:U) (P:x = x \rightarrow \text{Prop}), P (\text{eq_refl } x) \rightarrow \forall p:x = x, P p.$

Proof-irrelevance on subsets of decidable sets

Lemma `inj_pairP2` :

$\forall (P:U \rightarrow \text{Prop}) (x:U) (p q:P x),$
 $\text{ex_intro } P x p = \text{ex_intro } P x q \rightarrow p = q.$

Injectivity of equality on dependent pairs in `Type`

Lemma `inj_pair2` :

$\forall (P:U \rightarrow \text{Type}) (p:U) (x y:P p),$
 $\text{existT } P p x = \text{existT } P p y \rightarrow x = y.$

Injectivity of equality on dependent pairs with second component in **Type**

Notation `inj_pairT2` := `inj_pair2`.

End `DECIDABLEEQDEPSET`.

From decidability to `inj_pair2` **Lemma** `inj_pair2_eq_dec` : $\forall A:\text{Type}, (\forall x y:A, \{x=y\}+\{x \neq y\})$
 \rightarrow
 $(\forall (P:A \rightarrow \text{Type}) (p:A) (x y:P p), \text{existT } P p x = \text{existT } P p y \rightarrow x = y).$

Examples of short direct proofs of unicity of reflexivity proofs on specific domains

Lemma `UIP_refl_unit` $(x : \text{tt} = \text{tt}) : x = \text{eq_refl } \text{tt}.$

Lemma `UIP_refl_bool` $(b:\text{bool}) (x : b = b) : x = \text{eq_refl}.$

Lemma `UIP_refl_nat` $(n:\text{nat}) (x : n = n) : x = \text{eq_refl}.$

Chapter 53

Library

Coq.Logic.ExtensionalFunctionRepresentative

This module states a limited form axiom of functional extensionality which selects a canonical representative in each class of extensional functions

Its main interest is that it is the needed ingredient to provide axiom of choice on setoids (a.k.a. axiom of extensional choice) when combined with classical logic and axiom of (intensional) choice

It provides extensionality of functions while still supporting (a priori) an intensional interpretation of equality

Axiom *extensional_function_representative* :

$$\forall A B, \exists repr, \forall (f : A \rightarrow B),$$
$$(\forall x, f x = repr f x) \wedge$$
$$(\forall g, (\forall x, f x = g x) \rightarrow repr f = repr g).$$

Chapter 54

Library `Coq.Logic.ExtensionalityFacts`

Some facts and definitions about extensionality

We investigate the relations between the following extensionality principles

- Functional extensionality
- Equality of projections from diagonal
- Unicity of inverse bijections
- Bijectivity of bijective composition

Table of contents

1. Definitions
2. Functional extensionality \leftrightarrow Equality of projections from diagonal
3. Functional extensionality \leftrightarrow Unicity of inverse bijections
4. Functional extensionality \leftrightarrow Bijectivity of bijective composition

`Set Implicit Arguments.`

54.1 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

Equality of projections from diagonal

Unicity of bijection inverse

Bijectivity of bijective composition

Definition `action A B C (f:A→B) := (fun h:B→C => fun x => h (f x)).`

54.2 Functional extensionality \leftrightarrow Equality of projections from diagonal

Theorem `FuncExt_iff_EqDeltaProjs : FunctionalExtensionality ↔ EqDeltaProjs.`

54.3 Functional extensionality \leftrightarrow Unicity of bijection inverse

Lemma `FuncExt_UniqInverse : FunctionalExtensionality → UniqueInverse.`

Lemma `UniqInverse_EqDeltaProjs : UniqueInverse → EqDeltaProjs.`

Theorem `FuncExt_iff_UniqInverse : FunctionalExtensionality ↔ UniqueInverse.`

54.4 Functional extensionality \leftrightarrow Bijectivity of bijective composition

Lemma `FuncExt_BijComp : FunctionalExtensionality → BijectivityBijectiveComp.`

Lemma `BijComp_FuncExt : BijectivityBijectiveComp → FunctionalExtensionality.`

Chapter 55

Library `Coq.Logic.FinFun`

55.1 Functions on finite domains

Main result : for functions $f:A \rightarrow A$ with finite A , f injective \leftrightarrow f bijective \leftrightarrow f surjective.

`Require Import List Compare_dec EqNat Decidable ListDec. Require Fin.`
`Set Implicit Arguments.`

General definitions

`Definition Injective {A B} (f : A → B) :=`
 `∀ x y, f x = f y → x = y.`

`Definition Surjective {A B} (f : A → B) :=`
 `∀ y, ∃ x, f x = y.`

`Definition Bijective {A B} (f : A → B) :=`
 `∃ g : B → A, (∀ x, g (f x) = x) ∧ (∀ y, f (g y) = y).`

Finiteness is defined here via exhaustive list enumeration

`Definition Full {A:Type} (l:list A) := ∀ a:A, In a l.`

`Definition Finite (A:Type) := ∃ (l:list A), Full l.`

In many following proofs, it will be convenient to have list enumerations without duplicates. As soon as we have decidability of equality (in Prop), this is equivalent to the previous notion.

`Definition Listing {A:Type} (l:list A) := NoDup l ∧ Full l.`

`Definition Finite' (A:Type) := ∃ (l:list A), Listing l.`

`Lemma Finite_alt A (d:decidable_eq A) : Finite A ↔ Finite' A.`

Injections characterized in term of lists

`Lemma Injective_map_NoDup A B (f:A → B) (l:list A) :`
 `Injective f → NoDup l → NoDup (map f l).`

`Lemma Injective_list_carac A B (d:decidable_eq A)(f:A → B) :`
 `Injective f ↔ (∀ l, NoDup l → NoDup (map f l)).`

`Lemma Injective_carac A B (l:list A) : Listing l →`
 `∀ (f:A → B), Injective f ↔ NoDup (map f l).`

Surjection characterized in term of lists

Lemma `Surjective_list_carac` $A B (f:A \rightarrow B)$:
`Surjective f` $\leftrightarrow (\forall lB, \exists lA, \text{incl } lB (\text{map } f lA))$.

Lemma `Surjective_carac` $A B : \text{Finite } B \rightarrow \text{decidable_eq } B \rightarrow$
 $\forall f:A \rightarrow B, \text{Surjective } f \leftrightarrow (\exists lA, \text{Listing } (\text{map } f lA))$.

Main result :

Lemma `Endo_Injective_Surjective` :
 $\forall A, \text{Finite } A \rightarrow \text{decidable_eq } A \rightarrow$
 $\forall f:A \rightarrow A, \text{Injective } f \leftrightarrow \text{Surjective } f$.

An injective and surjective function is bijective. We need here stronger hypothesis : decidability of equality in Type.

Definition `EqDec` $(A:\text{Type}) := \forall x y:A, \{x=y\} + \{x \neq y\}$.

First, we show that a surjective f has an inverse function g such that $f.g = \text{id}$.

Lemma `Finite_Empty_or_not` A :
 $\text{Finite } A \rightarrow (A \rightarrow \text{False}) \vee \exists a:A, \text{True}$.

Lemma `Surjective_inverse` :
 $\forall A B, \text{Finite } A \rightarrow \text{EqDec } B \rightarrow$
 $\forall f:A \rightarrow B, \text{Surjective } f \rightarrow$
 $\exists g:B \rightarrow A, \forall x, f (g x) = x$.

Same, with more knowledge on the inverse function: $g.f = f.g = \text{id}$

Lemma `Injective_Surjective_Bijective` :
 $\forall A B, \text{Finite } A \rightarrow \text{EqDec } B \rightarrow$
 $\forall f:A \rightarrow B, \text{Injective } f \rightarrow \text{Surjective } f \rightarrow \text{Bijective } f$.

An example of finite type : *Fin.t*

Lemma `Fin_Finite` $n : \text{Finite } (\text{Fin.t } n)$.

Instead of working on a finite subset of nat , another solution is to use restricted $\text{nat} \rightarrow \text{nat}$ functions, and to consider them only below a certain bound n .

Definition `bFun` $n (f:\text{nat} \rightarrow \text{nat}) := \forall x, x < n \rightarrow f x < n$.

Definition `bInjective` $n (f:\text{nat} \rightarrow \text{nat}) :=$
 $\forall x y, x < n \rightarrow y < n \rightarrow f x = f y \rightarrow x = y$.

Definition `bSurjective` $n (f:\text{nat} \rightarrow \text{nat}) :=$
 $\forall y, y < n \rightarrow \exists x, x < n \wedge f x = y$.

We show that this is equivalent to the use of *Fin.t n*.

Module `FIN2RESTRICT`.

Notation `n2f` $:= \text{Fin.of_nat_lt}$.

Definition `f2n` $\{n\} (x:\text{Fin.t } n) := \text{proj1_sig } (\text{Fin.to_nat } x)$.

Definition `f2n_ok` $n (x:\text{Fin.t } n) : \text{f2n } x < n := \text{proj2_sig } (\text{Fin.to_nat } x)$.

Definition `n2f_f2n` $: \forall n x, \text{n2f } (\text{f2n_ok } x) = x := @\text{Fin.of_nat_to_nat_inv}$.

Definition `f2n_n2f` $x n h : \text{f2n } (\text{n2f } h) = x := \text{f_equal } (@\text{proj1_sig } _ _) (@\text{Fin.to_nat_of_nat } x n h)$.

Definition `n2f_ext` : $\forall x n h h', n2f\ h = n2f\ h' := @Fin.of_nat_ext$.

Definition `f2n_inj` : $\forall n x y, f2n\ x = f2n\ y \rightarrow x = y := @Fin.to_nat_inj$.

Definition `extend n` ($f:Fin.t\ n \rightarrow Fin.t\ n$) : $(nat \rightarrow nat) :=$

```
fun x =>
  match le_lt_dec n x with
  | left _ => 0
  | right h => f2n (f (n2f h))
end.
```

Definition `restrict n` ($f:nat \rightarrow nat$) ($hf : bFun\ n\ f$) : $(Fin.t\ n \rightarrow Fin.t\ n) :=$

```
fun x => let (x',h) := Fin.to_nat x in n2f (hf _ h).
```

Ltac `break_dec H` :=

```
let H' := fresh "H" in
destruct le_lt_dec as [H'|H'];
|elim (Lt.le_not_lt _ _ H' H)
|try rewrite (n2f_ext H' H) in *; try clear H'.
```

Lemma `extend_ok n f` : $bFun\ n\ (@extend\ n\ f)$.

Lemma `extend_f2n n f` ($x:Fin.t\ n$) : $extend\ f\ (f2n\ x) = f2n\ (f\ x)$.

Lemma `extend_n2f n f x` ($h:x < n$) : $n2f\ (extend_ok\ f\ h) = f\ (n2f\ h)$.

Lemma `restrict_f2n n f hf` ($x:Fin.t\ n$) :

$f2n\ (@restrict\ n\ f\ hf\ x) = f\ (f2n\ x)$.

Lemma `restrict_n2f n f hf x` ($h:x < n$) :

$@restrict\ n\ f\ hf\ (n2f\ h) = n2f\ (hf\ _\ h)$.

Lemma `extend_surjective n f` :

$bSurjective\ n\ (@extend\ n\ f) \leftrightarrow Surjective\ f$.

Lemma `extend_injective n f` :

$bInjective\ n\ (@extend\ n\ f) \leftrightarrow Injective\ f$.

Lemma `restrict_surjective n f h` :

$Surjective\ (@restrict\ n\ f\ h) \leftrightarrow bSurjective\ n\ f$.

Lemma `restrict_injective n f h` :

$Injective\ (@restrict\ n\ f\ h) \leftrightarrow bInjective\ n\ f$.

End `FIN2RESTRICT`.

Import `Fin2Restrict`.

We can now use Proof via the equivalence ...

Lemma `bInjective_bSurjective n` ($f:nat \rightarrow nat$) :

$bFun\ n\ f \rightarrow (bInjective\ n\ f \leftrightarrow bSurjective\ n\ f)$.

Lemma `bSurjective_bBijective n` ($f:nat \rightarrow nat$) :

$bFun\ n\ f \rightarrow bSurjective\ n\ f \rightarrow$

$\exists g, bFun\ n\ g \wedge \forall x, x < n \rightarrow g\ (f\ x) = x \wedge f\ (g\ x) = x$.

Chapter 56

Library

Coq.Logic.FunctionalExtensionality

This module states the axiom of (dependent) functional extensionality and (dependent) eta-expansion. It introduces a tactic `extensionality` to apply the axiom of extensionality to an equality goal.

The converse of functional extensionality.

Lemma `equal_f` : $\forall \{A B : \text{Type}\} \{f g : A \rightarrow B\}$,
 $f = g \rightarrow \forall x, f x = g x$.

Lemma `equal_f_dep` : $\forall \{A B\} \{f g : \forall (x : A), B x\}$,
 $f = g \rightarrow \forall x, f x = g x$.

Statements of functional extensionality for simple and dependent functions.

Axiom `functional_extensionality_dep` : $\forall \{A\} \{B : A \rightarrow \text{Type}\}$,
 $\forall (f g : \forall x : A, B x)$,
 $(\forall x, f x = g x) \rightarrow f = g$.

Lemma `functional_extensionality` $\{A B\} (f g : A \rightarrow B)$:
 $(\forall x, f x = g x) \rightarrow f = g$.

Extensionality of \forall s follows from functional extensionality. **Lemma** `forall_extensionality` $\{A\} \{B C : A \rightarrow \text{Type}\} (H : \forall x : A, B x = C x)$
: $(\forall x, B x) = (\forall x, C x)$.

Lemma `forall_extensionalityP` $\{A\} \{B C : A \rightarrow \text{Prop}\} (H : \forall x : A, B x = C x)$
: $(\forall x, B x) = (\forall x, C x)$.

Lemma `forall_extensionalityS` $\{A\} \{B C : A \rightarrow \text{Set}\} (H : \forall x : A, B x = C x)$
: $(\forall x, B x) = (\forall x, C x)$.

A version of `functional_extensionality_dep` which is provably equal to `eq_refl` on `fun _ => eq_refl`

Definition `functional_extensionality_dep_good`

$\{A\} \{B : A \rightarrow \text{Type}\}$
 $(f g : \forall x : A, B x)$
 $(H : \forall x, f x = g x)$
: $f = g$
:= `eq_trans (eq_sym (functional_extensionality_dep f f (fun _ => eq_refl)))`

(*functional_extensionality_dep* *f g H*).

Lemma *functional_extensionality_dep_good_refl* {*A B*} *f*
 : @*functional_extensionality_dep_good* *A B f f* (*fun* _ => *eq_refl*) = *eq_refl*.

Opaque *functional_extensionality_dep_good*.

Lemma *forall_sig_eq_rect*
 {*A B*} (*f* : $\forall a : A, B a$)
 (*P* : { *g* : _ | ($\forall a, f a = g a$) } → **Type**)
 (*k* : *P* (*exist* (*fun g* => $\forall a, f a = g a$) *f* (*fun a* => *eq_refl*)))
g
 : *P g*.

Definition *forall_eq_rect*
 {*A B*} (*f* : $\forall a : A, B a$)
 (*P* : $\forall g, (\forall a, f a = g a) \rightarrow$ **Type**)
 (*k* : *P f* (*fun a* => *eq_refl*))
g H
 : *P g H*
 := @*forall_sig_eq_rect* *A B f* (*fun g* => *P* (*proj1_sig* *g*) (*proj2_sig* *g*)) *k* (*exist* _ *g H*).

Definition *forall_eq_rect_comp* {*A B*} *f P k*
 : @*forall_eq_rect* *A B f P k f* (*fun* _ => *eq_refl*) = *k*.

Definition *f_equal_functional_extensionality_dep_good*
 {*A B f g*} *H a*
 : *f_equal* (*fun h* => *h a*) (@*functional_extensionality_dep_good* *A B f g H*) = *H a*.

Definition *f_equal_functional_extensionality_dep_good_fun*
 {*A B f g*} *H*
 : (*fun a* => *f_equal* (*fun h* => *h a*) (@*functional_extensionality_dep_good* *A B f g H*)) = *H*.

Apply *functional_extensionality*, introducing variable *x*.

Tactic Notation "extensionality" *ident(x)* :=
 match goal with
 [$\vdash ?X = ?Y$] =>
 (apply (@*functional_extensionality* _ _ *X Y*) ||
 apply (@*functional_extensionality_dep* _ _ *X Y*) ||
 apply *forall_extensionalityP* ||
 apply *forall_extensionalityS* ||
 apply *forall_extensionality*) ; **intro** *x*
 end.

Iteratively apply *functional_extensionality* on an hypothesis until finding an equality statement

Ltac *extensionality_in_checker* *tac* :=
 first [*tac tt* | fail 1 "Anomaly: Unexpected error in extensionality tactic. Please report."].

Tactic Notation "extensionality" "in" *hyp(H)* :=
 let rec *check_is_extensional_equality* *H* :=
 lazy match type of *H* with
 | _ = _ => **constr**:(**Prop**)
 | $\forall a : ?A, ?T$

```

    ⇒ let Ha := fresh in
      constr:(∀ a : A, match H a with Ha ⇒ ltac:(let v := check_is_extensional_equality
Ha in exact v) end)
    end in
let assert_is_extensional_equality H :=
  first [ let dummy := check_is_extensional_equality H in idtac
        | fail 1 "Not an extensional equality" ] in
let assert_not_intensional_equality H :=
  lazymatch type of H with
  | _ = _ ⇒ fail "Already an intensional equality"
  | _ ⇒ idtac
  end in
let enforce_no_body H :=
  (tryif (let dummy := (eval unfold H in H) in idtac)
  then clearbody H
  else idtac) in
let rec extensionality_step_make_type H :=
  lazymatch type of H with
  | ∀ a : ?A, ?f = ?g
    ⇒ constr:({ H' | (fun a ⇒ f_equal (fun h ⇒ h a) H') = H })
  | ∀ a : ?A, _
    ⇒ let H' := fresh in
      constr:(∀ a : A, match H a with H' ⇒ ltac:(let ret := extensionality_step_make_type
H' in exact ret) end)
    end in
let rec eta_contract T :=
  lazymatch (eval cbv beta in T) with
  | context T'[fun a : ?A ⇒ ?f a]
    ⇒ let T'' := context T'[f] in
      eta_contract T''
  | ?T ⇒ T
  end in
let rec lift_sig_extensionality H :=
  lazymatch type of H with
  | sig _ ⇒ H
  | ∀ a : ?A, _
    ⇒ let Ha := fresh in
      let ret := constr:(fun a : A ⇒ match H a with Ha ⇒ ltac:(let v := lift_sig_extensionality
Ha in exact v) end) in
      lazymatch type of ret with
      | ∀ a : ?A, sig (fun b : ?B ⇒ @?f a b = @?g a b)
        ⇒ eta_contract (exist (fun b : (∀ a : A, B) ⇒ (fun a : A ⇒ f a (b a)) = (fun a :
A ⇒ g a (b a)))
          (fun a : A ⇒ proj1_sig (ret a))
          (@functional_extensionality_dep_good _ _ _ _ (fun a : A

```

```

⇒ proj2_sig (ret a)))
      end
    end in
  let extensionality_pre_step H H_out Heq :=
    let T := extensionality_step_make_type H in
    let H' := fresh in
    assert (H' : T) by (intros; eexists; apply f_equal__functional_extensionality_dep_good__fun);
    let H''b := lift_sig_extensionality H' in
    case H''b; clear H';
    intros H_out Heq in
  let rec extensionality_rec H H_out Heq :=
    lazymatch type of H with
    | ∀ a, _ = _
      ⇒ extensionality_pre_step H H_out Heq
    | -
      ⇒ let pre_H_out' := fresh H_out in
         let H_out' := fresh pre_H_out' in
           extensionality_pre_step H H_out' Heq;
         let Heq' := fresh Heq in
           extensionality_rec H_out' H_out Heq';
         subst H_out'
    end in
  first [ assert_is_extensional_equality H | fail 1 "Not an extensional equality" ];
  first [ assert_not_intensional_equality H | fail 1 "Already an intensional equality" ];
  (tryif enforce_no_body H then idtac else clearbody H);
  let H_out := fresh in
  let Heq := fresh "Heq" in
  extensionality_in_checker ltac:(fun tt ⇒ extensionality_rec H H_out Heq);

  destruct Heq; rename H_out into H.

```

Eta expansion is built into Coq.

Lemma eta_expansion_dep {A} {B : A → Type} (f : ∀ x : A, B x) :
 f = fun x ⇒ f x.

Lemma eta_expansion {A B} (f : A → B) : f = fun x ⇒ f x.

Chapter 57

Library `Coq.Logic.Hurkens`

Exploiting Hurkens’s paradox [Hurkens95] for system U- so as to derive various contradictory contexts.

The file is divided into various sub-modules which all follow the same structure: a section introduces the contradictory hypotheses and a theorem named *paradox* concludes the module with a proof of *False*.

- The *Generic* module contains the actual Hurkens’s paradox for a postulated shallow encoding of system U- in Coq. This is an adaptation by Arnaud Spiwack of a previous, more restricted implementation by Herman Geuvers. It is used to derive every other special cases of the paradox in this file.
- The *NoRetractToImpredicativeUniverse* module contains a simple and effective formulation by Herman Geuvers [Geuvers01] of a result by Thierry Coquand [Coquand90]. It states that no impredicative sort can contain a type of which it is a retract. This result implies that Coq with classical logic stated in impredicative Set is inconsistent and that classical logic stated in Prop implies proof-irrelevance (see *ClassicalFacts.v*)
- The *NoRetractFromSmallPropositionToProp* module is a specialisation of the *NoRetractToImpredicativeUniverse* module to the case where the impredicative sort is `Prop`.
- The *NoRetractToModalProposition* module is a strengthening of the *NoRetractFromSmallPropositionToProp* module. It shows that given a monadic modality (aka closure operator) M , the type of modal propositions (i.e. such that $M A \rightarrow A$) cannot be a retract of a modal proposition. It is an example of use of the paradox where the universes of system U- are not mapped to universes of Coq.
- The *NoRetractToNegativeProp* module is the specialisation of the *NoRetractFromSmallPropositionToProp* module where the modality is double-negation. This result implies that the principle of weak excluded middle ($\forall A, \sim\sim A \vee \sim A$) implies a weak variant of proof irrelevance.
- The *NoRetractFromTypeToProp* module proves that `Prop` cannot be a retract of a larger type.
- The *TypeNeqSmallType* module proves that `Type` is different from any smaller type.

- The *PropNeqType* module proves that **Prop** is different from any larger **Type**. It is an instance of the previous result.

References:

- Coquand90* T. Coquand, “Metamathematical Investigations of a Calculus of Constructions”, Proceedings of Logic in Computer Science (LICS’90), 1990.
- Hurkens95* A. J. Hurkens, “A simplification of Girard’s paradox”, Proceedings of the 2nd international conference Typed Lambda-Calculi and Applications (TLCA’95), 1995.
- Geuvers01* H. Geuvers, “Inconsistency of Classical Logic in Type Theory”, 2001, revised 2007 (see ¹).

57.1 A modular proof of Hurkens’s paradox.

It relies on an axiomatisation of a shallow embedding of system U- (i.e. types of U- are interpreted by types of Coq). The universes are encoded in a style, due to Martin-Löf, where they are given by a set of names and a family $El:Name \rightarrow Type$ which interprets each name into a type. This allows the encoding of universe to be decoupled from Coq’s universes. Dependent products and abstractions are similarly postulated rather than encoded as Coq’s dependent products and abstractions.

Module **GENERIC**.

Section **Paradox**.

57.1.1 Axiomatisation of impredicative universes in a Martin-Löf style

System U- has two impredicative universes. In the proof of the paradox they are slightly asymmetric (in particular the reduction rules of the small universe are not needed). Therefore, the axioms are duplicated allowing for a weaker requirement than the actual system U-.

Large universe

Variable $U1 : Type$.

Variable $El1 : U1 \rightarrow Type$.

Closure by small product Variable $Forall1 : \forall u:U1, (El1 u \rightarrow U1) \rightarrow U1$.

Notation " \forall_1 x : A , B" := (Forall1 A (fun x \Rightarrow B)).

Notation "A \rightarrow_1 B" := (Forall1 A (fun _ \Rightarrow B)).

Variable $lam1 : \forall u B, (\forall x:El1 u, El1 (B x)) \rightarrow El1 (\forall_1 x:u, B x)$.

Notation " λ_1 x , u" := (lam1 _ _ (fun x \Rightarrow u)).

Variable $app1 : \forall u B (f:El1 (Forall1 u B)) (x:El1 u), El1 (B x)$.

Notation "f \cdot_1 x" := (app1 _ _ f x).

Variable $beta1 : \forall u B (f:\forall x:El1 u, El1 (B x)) x,$
 $(\lambda_1 y, f y) \cdot_1 x = f x$.

¹<http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz>

Closure by large products $U1$ only needs to quantify over itself. **Variable** $ForallU1 : (U1 \rightarrow U1) \rightarrow U1$.

Notation " \forall_2 " A, F := ($ForallU1$ (\mathbf{fun} $A \Rightarrow F$)).

Variable $lamU1 : \forall F, (\forall A:U1, El1 (F A)) \rightarrow El1 (\forall_2 A, F A)$.

Notation " λ_2 " x, u := ($lamU1$ $_$ (\mathbf{fun} $x \Rightarrow u$)).

Variable $appU1 : \forall F (f:El1(\forall_2 A, F A)) (A:U1), El1 (F A)$.

Notation " $f \cdot_1$ " $[A]$:= ($appU1$ $_$ $f A$).

Variable $betaU1 : \forall F (f:\forall A:U1, El1 (F A)) A, (\lambda_2 x, f x) \cdot_1 [A] = f A$.

Small universe

The small universe is an element of the large one. **Variable** $u0 : U1$.

Notation $U0$:= ($El1$ $u0$).

Variable $Elo : U0 \rightarrow \mathbf{Type}$.

Closure by small product $U0$ does not need reduction rules **Variable** $Forall0 : \forall u:U0, (Elo u \rightarrow U0) \rightarrow U0$.

Notation " \forall_0 " $x : A, B$:= ($Forall0$ A (\mathbf{fun} $x \Rightarrow B$)).

Notation " $A \rightarrow_0 B$ " := ($Forall0$ A (\mathbf{fun} $_ \Rightarrow B$)).

Variable $lam0 : \forall u B, (\forall x:Elo u, Elo (B x)) \rightarrow Elo (\forall_0 x:u, B x)$.

Notation " λ_0 " x, u := ($lam0$ $_ _$ (\mathbf{fun} $x \Rightarrow u$)).

Variable $app0 : \forall u B (f:Elo (Forall0 u B)) (x:Elo u), Elo (B x)$.

Notation " $f \cdot_0$ " x := ($app0$ $_ _$ $f x$).

Closure by large products **Variable** $ForallU0 : \forall u:U1, (El1 u \rightarrow U0) \rightarrow U0$.

Notation " \forall_0^1 " $A : U, F$:= ($ForallU0$ U (\mathbf{fun} $A \Rightarrow F$)).

Variable $lamU0 : \forall U F, (\forall A:El1 U, Elo (F A)) \rightarrow Elo (\forall_0^1 A:U, F A)$.

Notation " λ_0^1 " x, u := ($lamU0$ $_ _$ (\mathbf{fun} $x \Rightarrow u$)).

Variable $appU0 : \forall U F (f:Elo(\forall_0^1 A:U, F A)) (A:El1 U), Elo (F A)$.

Notation " $f \cdot_0$ " $[A]$:= ($appU0$ $_ _$ $f A$).

57.1.2 Automating the rewrite rules of our encoding.

Local Ltac $simplify$:=

```
(repeat rewrite ?beta1, ?betaU1);
lazy beta.
```

Local Ltac $simplify_in\ h$:=

```
(repeat rewrite ?beta1, ?betaU1 in h);
lazy beta in h.
```

57.1.3 Hurkens's paradox.

An inhabitant of $U0$ standing for $False$. **Variable** $F:U0$.

Preliminary definitions

Definition V : $U1 := \forall_2 A, ((A \rightarrow_1 u0) \rightarrow_1 A \rightarrow_1 u0) \rightarrow_1 A \rightarrow_1 u0$.

Definition U : $U1 := V \rightarrow_1 u0$.

Definition sb ($z:El1\ V$) : $El1\ V := \lambda_2 A, \lambda_1 r, \lambda_1 a, r \cdot_1 (z \cdot_1 [A] \cdot_1 r) \cdot_1 a$.

Definition le ($i:El1\ (U \rightarrow_1 u0)$) ($x:El1\ U$) : $U0 :=$
 $x \cdot_1 (\lambda_2 A, \lambda_1 r, \lambda_1 a, i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [A] \cdot_1 r \cdot_1 a))$.

Definition le' : $El1\ ((U \rightarrow_1 u0) \rightarrow_1 U \rightarrow_1 u0) := \lambda_1 i, \lambda_1 x, le\ i\ x$.

Definition induct ($i:El1\ (U \rightarrow_1 u0)$) : $U0 :=$

$\forall_0^1 x:U, le\ i\ x \rightarrow_0 i \cdot_1 x$.

Definition WF : $El1\ U := \lambda_1 z, (induct\ (z \cdot_1 [U] \cdot_1 le'))$.

Definition I ($x:El1\ U$) : $U0 :=$

$(\forall_0^1 i:U \rightarrow_1 u0, le\ i\ x \rightarrow_0 i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 x)) \rightarrow_0 F$

Proof

Lemma Omega : $ELO\ (\forall_0^1 i:U \rightarrow_1 u0, induct\ i \rightarrow_0 i \cdot_1 WF)$.

Proof.

```

refine ( $\lambda_0^1 i, \lambda_0 y, -$ ).
refine ( $y \cdot_0 [-] \cdot_0 -$ ).
unfold le,WF,induct. simplify.
refine ( $\lambda_0^1 x, \lambda_0 h0, -$ ). simplify.
refine ( $y \cdot_0 [-] \cdot_0 -$ ).
unfold le. simplify.
unfold sb at 1. simplify.
unfold le' at 1. simplify.
exact h0.

```

Qed.

Lemma lemma1 : $ELO\ (induct\ (\lambda_1 u, I\ u))$.

Proof.

```

unfold induct.
refine ( $\lambda_0^1 x, \lambda_0 p, -$ ). simplify.
refine ( $\lambda_0 q, -$ ).
assert ( $ELO\ (I\ (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 x)))$ ) as h.
{ generalize ( $q \cdot_0 [\lambda_1 u, I\ u] \cdot_0 p$ ). simplify.
  intros q'.
  exact q'. }
refine ( $h \cdot_0 -$ ).
refine ( $\lambda_0^1 i, -$ ).
refine ( $\lambda_0 h', -$ ).
generalize ( $q \cdot_0 [\lambda_1 y, i \cdot_1 (\lambda_1 v, (sb\ v) \cdot_1 [U] \cdot_1 le' \cdot_1 y)]$ ). simplify.
intros q'.
refine ( $q' \cdot_0 -$ ). clear q'.
unfold le at 1 in h'. simplify_in h'.

```


Hypothesis $u22u1_unit : \forall (c:U2), c \rightarrow u22u1\ c$.

$u22u1_counit$ and $u22u1_coherent$ only apply to dependent product so that the equations happen in the smaller $U1$ rather than $U2$. Indeed, it is not generally the case that one can project from a large universe to an impredicative universe and then get back the original type again. It would be too strong a hypothesis to require (in particular, it is not true of **Prop**). The formulation is reminiscent of the monadic characteristic of the projection from a large type to **Prop**. **Hypothesis** $u22u1_counit : \forall (F:U1 \rightarrow U1), u22u1\ (\forall A, F\ A) \rightarrow (\forall A, F\ A)$.

Hypothesis $u22u1_coherent : \forall (F:U1 \rightarrow U1) (f:\forall x:U1, F\ x) (x:U1),$
 $u22u1_counit\ _\ (u22u1_unit\ _\ f)\ x = f\ x$.

$U0$ is a retract of $U1$

Variable $u02u1 : U0 \rightarrow U1$.

Variable $u12u0 : U1 \rightarrow U0$.

Hypothesis $u12u0_unit : \forall (b:U1), b \rightarrow u02u1\ (u12u0\ b)$.

Hypothesis $u12u0_counit : \forall (b:U1), u02u1\ (u12u0\ b) \rightarrow b$.

57.2.1 Paradox

Theorem $paradox : \forall F:U1, F$.

Proof.

`intros F.`

`Generic.paradox h.`

`Large universe + exact U1.`

`+ exact (fun X => X).`

`+ cbn. exact (fun u F => \forall x:u, F x).`

`+ cbn. exact (fun _ x => x).`

`+ cbn. exact (fun _ x => x).`

`+ cbn. exact (fun F => u22u1 (\forall x, F x)).`

`+ cbn. exact (fun _ x => u22u1_unit _ x).`

`+ cbn. exact (fun _ x => u22u1_counit _ x).`

`Small universe + exact U0.`

The interpretation of the small universe is the image of $U0$ in $U1$. `+ cbn. exact (fun X => u02u1 X).`

`+ cbn. exact (fun u F => u12u0 (\forall x:(u02u1 u), u02u1 (F x))).`

`+ cbn. exact (fun u F => u12u0 (\forall x:u, u02u1 (F x))).`

`+ cbn. exact (u12u0 F).`

`+ cbn in h.`

`exact (u12u0_counit _ h).`

`+ cbn. easy.`

`+ cbn. intros **. now rewrite u22u1_coherent.`

`+ cbn. intros \times x. exact (u12u0_unit _ x).`

`+ cbn. intros \times x. exact (u12u0_counit _ x).`

`+ cbn. intros \times x. exact (u12u0_unit _ x).`

`+ cbn. intros \times x. exact (u12u0_counit _ x).`

Qed.

End Paradox.

End NORETRACTTOIMPREDICATIVEUNIVERSE.

57.3 Modal fragments of `Prop` are not retracts

In presence of a monadic modality on `Prop`, we can define a subset of `Prop` of modal propositions which is also a complete Heyting algebra. These cannot be a retract of a modal proposition. This is a case where the universe in system U- are not encoded as Coq universes.

Module NORETRACTTOMODALPROPOSITION.

57.3.1 Monadic modality

Section Paradox.

Variable $M : \text{Prop} \rightarrow \text{Prop}$.

Hypothesis $incr : \forall A B:\text{Prop}, (A \rightarrow B) \rightarrow M A \rightarrow M B$.

Lemma strength: $\forall A (P:A \rightarrow \text{Prop}), M(\forall x:A, P x) \rightarrow \forall x:A, M(P x)$.

Proof.

intros $A P h x$.

eapply $incr$ in h ; eauto.

Qed.

57.3.2 The universe of modal propositions

Definition $\text{MProp} := \{ P:\text{Prop} \mid M P \rightarrow P \}$.

Definition $\text{El} : \text{MProp} \rightarrow \text{Prop} := @\text{proj1_sig} _ _$.

Lemma modal : $\forall P:\text{MProp}, M(\text{El } P) \rightarrow \text{El } P$.

Proof.

intros $[P m]$. *cbn*.

exact m .

Qed.

Definition $\text{Forall} \{A:\text{Type}\} (P:A \rightarrow \text{MProp}) : \text{MProp}$.

Proof.

unshelve (refine (exist _ _ _)).

+ exact $(\forall x:A, \text{El } (P x))$.

+ intros $h x$.

eapply strength in h .

eauto using modal.

Defined.

57.3.3 Retract of the modal fragment of `Prop` in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

Variable $bool : \text{MProp}$.

Variable $p2b : \text{MProp} \rightarrow \text{El } \text{bool}$.
 Variable $b2p : \text{El } \text{bool} \rightarrow \text{MProp}$.
 Hypothesis $p2p1 : \forall A:\text{MProp}, \text{El } (b2p (p2b A)) \rightarrow \text{El } A$.
 Hypothesis $p2p2 : \forall A:\text{MProp}, \text{El } A \rightarrow \text{El } (b2p (p2b A))$.

57.3.4 Paradox

Theorem `paradox` : $\forall B:\text{MProp}, \text{El } B$.

Proof.

```

intros B.
Generic.paradox h.
  Large universe    + exact MProp.
+ exact El.
+ exact (fun _ => Forall).
+ cbn. exact (fun _ _ f => f).
+ cbn. exact (fun _ _ f => f).
+ exact Forall.
+ cbn. exact (fun _ f => f).
+ cbn. exact (fun _ f => f).
  Small universe   + exact bool.
+ exact (fun b => El (b2p b)).
+ cbn. exact (fun _ F => p2b (Forall (fun x => b2p (F x)))).
+ exact (fun _ F => p2b (Forall (fun x => b2p (F x)))).
+ apply p2b.
  exact B.
+ cbn in h. auto.
+ cbn. easy.
+ cbn. easy.
+ cbn. auto.
+ cbn. intros × f.
  apply p2p1 in f. cbn in f.
  exact f.
+ cbn. auto.
+ cbn. intros × f.
  apply p2p1 in f. cbn in f.
  exact f.
  
```

Qed.

End `Paradox`.

End `NORETRACTTOMODALPROPOSITION`.

57.4 The negative fragment of `Prop` is not a retract

The existence in the pure Calculus of Constructions of a retract from the negative fragment of `Prop` into a negative proposition is inconsistent. This is an instance of the previous result.

Module NORETRACTTONEGATIVEPROP.

57.4.1 The universe of negative propositions.

Definition NProp := { P:Prop | $\sim\sim P \rightarrow P$ }.

Definition El : NProp → Prop := @proj1_sig _ _.

Section Paradox.

57.4.2 Retract of the negative fragment of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions.

Variable bool : NProp.

Variable p2b : NProp → El bool.

Variable b2p : El bool → NProp.

Hypothesis p2p1 : $\forall A:NProp, El (b2p (p2b A)) \rightarrow El A$.

Hypothesis p2p2 : $\forall A:NProp, El A \rightarrow El (b2p (p2b A))$.

57.4.3 Paradox

Theorem paradox : $\forall B:NProp, El B$.

Proof.

intros B.

unshelve (refine ((fun h => _) (NoRetractToModalProposition.paradox _ _ _ _ _))).

+ exact (fun P => $\sim\sim P$).

+ exact bool.

+ exact p2b.

+ exact b2p.

+ exact B.

+ exact h.

+ cbn. auto.

+ cbn. auto.

+ cbn. auto.

Qed.

End Paradox.

End NORETRACTTONEGATIVEPROP.

57.5 Prop is not a retract

The existence in the pure Calculus of Constructions of a retract from Prop into a small type of Prop is inconsistent. This is a special case of the previous result.

Module NORETRACTFROMSMALLPROPOSITIONTOPROP.

57.5.1 The universe of propositions.

Definition NProp := { P:Prop | P → P }.

Definition El : NProp → Prop := @proj1_sig _ _.

Section MParadox.

57.5.2 Retract of Prop in a small type, using the identity modality.

Variable bool : NProp.

Variable p2b : NProp → El bool.

Variable b2p : El bool → NProp.

Hypothesis p2p1 : ∀ A:NProp, El (b2p (p2b A)) → El A.

Hypothesis p2p2 : ∀ A:NProp, El A → El (b2p (p2b A)).

57.5.3 Paradox

Theorem mparadox : ∀ B:NProp, El B.

Proof.

intros B.

unshelve (refine ((fun h ⇒ _) (NoRetractToModalProposition.paradox _ _ _ _ _))).

+ exact (fun P ⇒ P).

+ exact bool.

+ exact p2b.

+ exact b2p.

+ exact B.

+ exact h.

+ cbn. auto.

+ cbn. auto.

+ cbn. auto.

Qed.

End MParadox.

Section Paradox.

57.5.4 Retract of Prop in a small type

The retract is axiomatized using logical equivalence as the equality on propositions. Variable bool : Prop.

Variable p2b : Prop → bool.

Variable b2p : bool → Prop.

Hypothesis p2p1 : ∀ A:Prop, b2p (p2b A) → A.

Hypothesis p2p2 : ∀ A:Prop, A → b2p (p2b A).

57.5.5 Paradox

Theorem paradox : ∀ B:Prop, B.

Proof.

```
intros B.
unshelve (refine (mparadox (exist _ bool (fun x => x)) _ _ _ _
  (exist _ B (fun x => x)))).
+ intros p. red. red. exact (p2b (El p)).
+ cbn. intros b. red.  $\exists$  (b2p b). exact (fun x => x).
+ cbn. intros [A H]. cbn. apply p2p1.
+ cbn. intros [A H]. cbn. apply p2p2.
```

Qed.

End Paradox.

End NORETRACTFROMSMALLPROPOSITIONTOPROP.

57.6 Large universes are not retracts of Prop.

The existence in the Calculus of Constructions with universes of a retract from some `Type` universe into `Prop` is inconsistent.

Module NORETRACTFROMTYPETOPROP.

Definition Type2 := Type.

Definition Type1 := Type : Type2.

Section Paradox.

57.6.1 Assumption of a retract from Type into Prop

Variable down : Type1 \rightarrow Prop.

Variable up : Prop \rightarrow Type1.

Hypothesis up_down : \forall (A:Type1), up (down A) = A :> Type1.

57.6.2 Paradox

Theorem paradox : \forall P:Prop, P.

Proof.

```
intros P.
Generic.paradox h.
Large universe. + exact Type1.
+ exact (fun X => X).
+ cbn. exact (fun u F =>  $\forall$  x, F x).
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
+ exact (fun F =>  $\forall$  A:Prop, F (up A)).
+ cbn. exact (fun F f A => f (up A)).
+ cbn.
  intros F f A.
  specialize (f (down A)).
```

```

    rewrite up_down in f.
    exact f.
+ exact Prop.
+ cbn. exact (fun X => X).
+ cbn. exact (fun A P => ∀ x:A, P x).
+ cbn. exact (fun A P => ∀ x:A, P x).
+ cbn. exact P.
+ exact h.
+ cbn. easy.
+ cbn.
  intros F f A.
  destruct (up_down A). cbn.
  reflexivity.
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
Qed.
End Paradox.
End NORETRACTFROMTYPETOPROP.

```

57.7 $A \neq \text{Type}$

No Coq universe can be equal to one of its elements.

```

Module TYPENEQSMALLTYPE.
Unset Universe Polymorphism.
Section Paradox.

```

57.7.1 Universe U is equal to one of its elements.

```

Let U := Type.
Variable A:U.
Hypothesis h : U=A.

```

57.7.2 Universe U is a retract of A

The following context is actually sufficient for the paradox to hold. The hypothesis $h:U=A$ is only used to define *down*, *up* and *up_down*.

```

Let down (X:U) : A := @eq_rect _ _ (fun X => X) X _ h.
Let up (X:A) : U := @eq_rect_r _ _ (fun X => X) X _ h.
Lemma up_down : ∀ (X:U), up (down X) = X.
Proof.
  unfold up,down.

```

```

rewrite ← h.
reflexivity.
Qed.
Theorem paradox : False.
Proof.
  Generic.paradox p.
  Large universe   + exact U.
+ exact (fun X => X).
+ cbn. exact (fun X F => ∀ x:X, F x).
+ cbn. exact (fun _ _ x => x).
+ cbn. exact (fun _ _ x => x).
+ exact (fun F => ∀ x:A, F (up x)).
+ cbn. exact (fun _ f => fun x:A => f (up x)).
+ cbn. intros × f X.
  specialize (f (down X)).
  rewrite up_down in f.
  exact f.
  Small universe   + exact A.
  The interpretation of A as a universe is U.   + cbn. exact up.
+ cbn. exact (fun _ F => down (∀ x, up (F x))).
+ cbn. exact (fun _ F => down (∀ x, up (F x))).
+ cbn. exact (down False).
+ rewrite up_down in p.
  exact p.
+ cbn. easy.
+ cbn. intros ? f X.
  destruct (up_down X). cbn.
  reflexivity.
+ cbn. intros ? ? f.
  rewrite up_down.
  exact f.
+ cbn. intros ? ? f.
  rewrite up_down in f.
  exact f.
+ cbn. intros ? ? f.
  rewrite up_down.
  exact f.
+ cbn. intros ? ? f.
  rewrite up_down in f.
  exact f.
Qed.
End Paradox.
End TYPENEQSMALLTYPE.

```

57.8 Prop ≠ Type.

Special case of *TypeNeqSmallType*.

Module PROPNEQTYPE.

Theorem paradox : Prop ≠ Type.

Proof.

```
  intros h.  
  unshelve (refine (TypeNeqSmallType.paradox - -)).  
  + exact Prop.  
  + easy.
```

Qed.

End PROPNEQTYPE.

Chapter 58

Library `Coq.Logic.IndefiniteDescription`

This file provides a constructive form of indefinite description that allows building choice functions; this is weaker than Hilbert's epsilon operator (which implies weakly classical properties) but stronger than the axiom of choice (which cannot be used outside the context of a theorem proof).

`Require Import ChoiceFacts.`

`Set Implicit Arguments.`

`Axiom constructive_indefinite_description :`

`∀ (A : Type) (P : A → Prop),
 (∃ x, P x) → { x : A | P x }.`

`Lemma constructive_definite_description :`

`∀ (A : Type) (P : A → Prop),
 (∃! x, P x) → { x : A | P x }.`

`Lemma functional_choice :`

`∀ (A B : Type) (R : A → B → Prop),
 (∀ x : A, ∃ y : B, R x y) →
 (∃ f : A → B, ∀ x : A, R x (f x)).`

Chapter 59

Library `Coq.Logic.JMeq`

John Major's Equality as proposed by Conor McBride

Reference:

McBride Elimination with a Motive, Proceedings of TYPES 2000, LNCS 2277, pp 197-216, 2002.

`Set Implicit Arguments.`

`Inductive JMeq (A:Type) (x:A) : $\forall B:\text{Type}, B \rightarrow \text{Prop}$:=
 JMeq_refl : JMeq x x.`

`Hint Resolve JMeq_refl.`

`Definition JMeq_hom {A : Type} (x y : A) := JMeq x y.`

`Lemma JMeq_sym : $\forall (A B:\text{Type}) (x:A) (y:B), \text{JMeq } x \ y \rightarrow \text{JMeq } y \ x.$`

`Hint Immediate JMeq_sym.`

`Lemma JMeq_trans :`

`$\forall (A \ B \ C:\text{Type}) (x:A) (y:B) (z:C), \text{JMeq } x \ y \rightarrow \text{JMeq } y \ z \rightarrow \text{JMeq } x \ z.$`

`Axiom JMeq_eq : $\forall (A:\text{Type}) (x \ y:A), \text{JMeq } x \ y \rightarrow x = y.$`

`Lemma JMeq_ind : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Prop}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y.$`

`Lemma JMeq_rec : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Set}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y.$`

`Lemma JMeq_rect : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Type}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } x \ y \rightarrow P \ y.$`

`Lemma JMeq_ind_r : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Prop}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } y \ x \rightarrow P \ y.$`

`Lemma JMeq_rec_r : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Set}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } y \ x \rightarrow P \ y.$`

`Lemma JMeq_rect_r : $\forall (A:\text{Type}) (x:A) (P:A \rightarrow \text{Type}),$`

`$P \ x \rightarrow \forall y, \text{JMeq } y \ x \rightarrow P \ y.$`

`Lemma JMeq_congr :`

$\forall (A:\text{Type}) (x:A) (B:\text{Type}) (f:A \rightarrow B) (y:A), \text{JMeq } x \ y \rightarrow f \ x = f \ y.$

JMeq is equivalent to $\text{eq_dep Type (fun } X \Rightarrow X)$

Require Import Eqdep.

Lemma JMeq_eq_dep_id :

$\forall (A \ B:\text{Type}) (x:A) (y:B), \text{JMeq } x \ y \rightarrow \text{eq_dep Type (fun } X \Rightarrow X) \ A \ x \ B \ y.$

Lemma eq_dep_id_JMeq :

$\forall (A \ B:\text{Type}) (x:A) (y:B), \text{eq_dep Type (fun } X \Rightarrow X) \ A \ x \ B \ y \rightarrow \text{JMeq } x \ y.$

$\text{eq_dep } U \ P \ p \ x \ q \ y$ is strictly finer than $\text{JMeq } (P \ p) \ x \ (P \ q) \ y$

Lemma eq_dep_JMeq :

$\forall U \ P \ p \ x \ q \ y, \text{eq_dep } U \ P \ p \ x \ q \ y \rightarrow \text{JMeq } x \ y.$

Lemma eq_dep_strictly_stronger_JMeq :

$\exists U \ P \ p \ q \ x \ y, \text{JMeq } x \ y \wedge \neg \text{eq_dep } U \ P \ p \ x \ q \ y.$

However, when the dependencies are equal, $\text{JMeq } (P \ p) \ x \ (P \ q) \ y$ is as strong as $\text{eq_dep } U \ P \ p \ x \ q \ y$ (this uses JMeq_eq)

Lemma JMeq_eq_dep :

$\forall U (P:U \rightarrow \text{Type}) \ p \ q (x:P \ p) (y:P \ q),$
 $p = q \rightarrow \text{JMeq } x \ y \rightarrow \text{eq_dep } U \ P \ p \ x \ q \ y.$

Notation sym_JMeq := JMeq_sym (*only parsing*).

Notation trans_JMeq := JMeq_trans (*only parsing*).

Chapter 60

Library `Coq.Logic.ProofIrrelevance`

This file axiomatizes proof-irrelevance and derives some consequences

```
Require Import ProofIrrelevanceFacts.
```

```
Axiom proof_irrelevance :  $\forall (P:\text{Prop}) (p1\ p2:P), p1 = p2$ .
```

```
Module PI. Definition proof_irrelevance := proof_irrelevance. End PI.
```

```
Module PROOFIRRELEVANCETHEORY := PROOFIRRELEVANCETHEORY(PI).
```

```
Export ProofIrrelevanceTheory.
```

Chapter 61

Library `Coq.Logic.ProofIrrelevanceFacts`

This defines the functor that build consequences of proof-irrelevance

```
Require Export EqdepFacts.
```

```
Module Type PROOFIRRELEVANCE.
```

```
  Axiom proof_irrelevance :  $\forall (P:\text{Prop}) (p1 p2:P), p1 = p2$ .
```

```
End PROOFIRRELEVANCE.
```

```
Module PROOFIRRELEVANCETHEORY (M:PROOFIRRELEVANCE).
```

Proof-irrelevance implies uniqueness of reflexivity proofs

```
Module EQ_RECT_EQ.
```

```
  Lemma eq_rect_eq :
```

```
     $\forall (U:\text{Type}) (p:U) (Q:U \rightarrow \text{Type}) (x:Q p) (h:p = p),$   
       $x = \text{eq\_rect } p \ Q \ x \ p \ h$ .
```

```
End EQ_RECT_EQ.
```

Export the theory of injective dependent elimination

```
Module EQDEPTHEORY := EQDEPTHEORY(EQ_RECT_EQ).
```

```
Export EqdepTheory.
```

```
Scheme eq_indd := Induction for eq Sort Prop.
```

We derive the irrelevance of the membership property for subsets

```
Lemma subset_eq_compat :
```

```
   $\forall (U:\text{Type}) (P:U \rightarrow \text{Prop}) (x y:U) (p:P x) (q:P y),$   
     $x = y \rightarrow \text{exist } P \ x \ p = \text{exist } P \ y \ q$ .
```

```
Lemma subsetT_eq_compat :
```

```
   $\forall (U:\text{Type}) (P:U \rightarrow \text{Prop}) (x y:U) (p:P x) (q:P y),$   
     $x = y \rightarrow \text{existT } P \ x \ p = \text{existT } P \ y \ q$ .
```

```
End PROOFIRRELEVANCETHEORY.
```

Chapter 62

Library `Coq.Logic.PropExtensionality`

This module states propositional extensionality and draws consequences of it

Axiom `propositional_extensionality` :

$\forall (P Q : \text{Prop}), (P \leftrightarrow Q) \rightarrow P = Q.$

Require Import `ClassicalFacts`.

Theorem `proof_irrelevance` : $\forall (P:\text{Prop}) (p1 p2:P), p1 = p2.$

Chapter 63

Library

Coq.Logic.PropExtensionalityFacts

Some facts and definitions about propositional and predicate extensionality

We investigate the relations between the following extensionality principles

- Proposition extensionality
- Predicate extensionality
- Propositional functional extensionality
- Provable-proposition extensionality
- Refutable-proposition extensionality
- Extensional proposition representatives
- Extensional predicate representatives
- Extensional propositional function representatives

Table of contents

1. Definitions

2.1 Predicate extensionality \leftrightarrow Proposition extensionality + Propositional functional extensionality

2.2 Propositional extensionality \rightarrow Provable propositional extensionality

2.3 Propositional extensionality \rightarrow Refutable propositional extensionality

Set Implicit Arguments.

63.1 Definitions

Propositional extensionality

Provable-proposition extensionality

Refutable-proposition extensionality

Predicate extensionality

Propositional functional extensionality

63.2 Propositional and predicate extensionality

63.2.1 Predicate extensionality \leftrightarrow Propositional extensionality + Propositional functional extensionality

Lemma `PredExt_imp_PropExt` : `PredicateExtensionality` \rightarrow `PropositionalExtensionality`.

Lemma `PredExt_imp_PropFunExt` : `PredicateExtensionality` \rightarrow `PropositionalFunctionalExtensionality`.

Lemma `PropExt_and_PropFunExt_imp_PredExt` :
`PropositionalExtensionality` \rightarrow `PropositionalFunctionalExtensionality` \rightarrow `PredicateExtensionality`.

Theorem `PropExt_and_PropFunExt_iff_PredExt` :
`PropositionalExtensionality` \wedge `PropositionalFunctionalExtensionality` \leftrightarrow `PredicateExtensionality`.

63.2.2 Propositional extensionality and provable proposition extensionality

Lemma `PropExt_imp_ProvPropExt` : `PropositionalExtensionality` \rightarrow `ProvablePropositionExtensionality`.

63.2.3 Propositional extensionality and refutable proposition extensionality

Lemma `PropExt_imp_RefutPropExt` : `PropositionalExtensionality` \rightarrow `RefutablePropositionExtensionality`.

Chapter 64

Library `Coq.Logic.PropFacts`

64.1 Basic facts about `Prop` as a type

An intuitionistic theorem from topos theory [*LambekScott*]

References:

[*LambekScott*] Jim Lambek, Phil J. Scott, Introduction to higher order categorical logic, Cambridge Studies in Advanced Mathematics (Book 7), 1988.

Theorem `injection_is_involution_in_Prop`

$(f : \mathbf{Prop} \rightarrow \mathbf{Prop})$

$(inj : \forall A B, (f A \leftrightarrow f B) \rightarrow (A \leftrightarrow B))$

$(ext : \forall A B, A \leftrightarrow B \rightarrow f A \leftrightarrow f B)$

$: \forall A, f (f A) \leftrightarrow A.$

Chapter 65

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 66

Library `Coq.Logic.SetIsType`

66.1 The Set universe seen as a synonym for Type

After loading this file, Set becomes just another name for Type. This allows easily performing a Set-to-Type migration, or at least test whether a development relies or not on specific features of Set: simply insert some `Require Export` of this file at starting points of the development and try to recompile...

Notation `"Set"` := `Type` (*only parsing*).

Chapter 67

Library `Coq.Logic.SetoidChoice`

This module states the functional form of the axiom of choice over setoids, commonly called extensional axiom of choice [*Carlström04*], [*Martin-Löf05*]. This is obtained by a decomposition of the axiom into the following components:

- classical logic
- relational axiom of choice
- axiom of unique choice
- a limited form of functional extensionality

Among other results, it entails:

- proof irrelevance
- choice of a representative in equivalence classes

[*Carlström04*] Jesper Carlström, EM + Ext + AC_{int} is equivalent to AC_{ext}, *Mathematical Logic Quarterly*, vol 50(3), pp 236-240, 2004.

[*Martin-Löf05*] *Per Martin-Löf, 100 years of Zermelo's axiom of choice: what was the problem with it?, lecture notes for KTH/SU colloquium*, 2005.

`Require Export ClassicalChoice.` `Require Export ExtensionalFunctionRepresentative.`

`Require Import ChoiceFacts.`

`Require Import ClassicalFacts.`

`Require Import RelationClasses.`

`Theorem setoid_choice :`

`$\forall A B,$`

`$\forall R : A \rightarrow A \rightarrow \text{Prop},$`

`$\forall T : A \rightarrow B \rightarrow \text{Prop},$`

`Equivalence R \rightarrow`

`$(\forall x x' y, R x x' \rightarrow T x y \rightarrow T x' y) \rightarrow$`

`$(\forall x, \exists y, T x y) \rightarrow$`

$\exists f : A \rightarrow B, \forall x : A, T x (f x) \wedge (\forall x' : A, R x x' \rightarrow f x = f x')$.

Theorem `representative_choice` :

$\forall A (R:A \rightarrow A \rightarrow \text{Prop}), (\text{Equivalence } R) \rightarrow$

$\exists f : A \rightarrow A, \forall x : A, R x (f x) \wedge \forall x', R x x' \rightarrow f x = f x'$.

Chapter 68

Library `Coq.Logic.WKL`

A constructive proof of a version of Weak König's Lemma over a decidable predicate in the formulation of which infinite paths are treated as predicates. The representation of paths as relations avoid the need for classical logic and unique choice. The decidability condition is sufficient to ensure that some required instance of double negation for disjunction of finite paths holds.

The idea of the proof comes from the proof of the weak König's lemma from separation in second-order arithmetic.

Notice that we do not start from a tree but just from an arbitrary predicate. Original Weak König's Lemma is the instantiation of the lemma to a tree

```
Require Import WeakFan List.
```

```
Import ListNotations.
```

```
Require Import Omega.
```

is_path_from $P\ n\ l$ means that there exists a path of length n from l on which P does not hold

```
Inductive is_path_from (P:list bool → Prop) : nat → list bool → Prop :=  
| here l : ¬ P l → is_path_from P 0 l  
| next_left l n : ¬ P l → is_path_from P n (true::l) → is_path_from P (S n) l  
| next_right l n : ¬ P l → is_path_from P n (false::l) → is_path_from P (S n) l.
```

We give the characterization of `is_path_from` in terms of a more common arithmetical formula

Proposition `is_path_from_characterization` $P\ n\ l$:

```
is_path_from P n l ↔ ∃ l', length l' = n ∧ ∀ n', n' ≤ n → ¬ P (rev (firstn n' l') ++ l).
```

infinite_from $P\ l$ means that we can find arbitrary long paths along which P does not hold above l

Definition `infinite_from` $(P:list\ bool\ →\ Prop)\ l := \forall n, is_path_from\ P\ n\ l.$

has_infinite_path P means that there is an infinite path (represented as a predicate) along which P does not hold at all

Definition `has_infinite_path` $(P:list\ bool\ →\ Prop) :=$

```
∃ (X:nat → Prop), ∀ l, approx X l → ¬ P l.
```

inductively_barred_at $P\ n\ l$ means that P eventually holds above l after at most n steps upwards

Inductive `inductively_barred_at` $(P:list\ bool\ →\ Prop) : nat → list\ bool → Prop :=$

```
| now_at l n : P l → inductively_barred_at P n l
```

```

| propagate_at l n :
  inductively_barred_at P n (true::l) →
  inductively_barred_at P n (false::l) →
  inductively_barred_at P (S n) l.

```

The proof proceeds by building a set Y of finite paths approximating either the smallest unbarred infinite path in P , if there is one (taking $true > false$), or the path $true::true::\dots$ if P happens to be inductively_barred

```

Fixpoint Y P (l:list bool) :=
  match l with
  | [] ⇒ True
  | b::l ⇒
    Y P l ∧
    if b then ∃ n, inductively_barred_at P n (false::l) else infinite_from P (false::l)
  end.

```

Require Import Compare_dec Le Lt.

Lemma is_path_from_restrict : $\forall P n n' l, n \leq n' \rightarrow$
 $is_path_from P n' l \rightarrow is_path_from P n l.$

Lemma inductively_barred_at_monotone : $\forall P l n n', n' \leq n \rightarrow$
 $inductively_barred_at P n' l \rightarrow inductively_barred_at P n l.$

Definition demorgan_or (P:list bool \rightarrow Prop) l l' := $\neg (P l \wedge P l') \rightarrow \neg P l \vee \neg P l'.$

Definition demorgan_inductively_barred_at P :=
 $\forall n l, demorgan_or (inductively_barred_at P n) (true::l) (false::l).$

Lemma inductively_barred_at_imp_is_path_from :
 $\forall P, demorgan_inductively_barred_at P \rightarrow \forall n l,$
 $\neg inductively_barred_at P n l \rightarrow is_path_from P n l.$

Lemma is_path_from_imp_inductively_barred_at : $\forall P n l,$
 $is_path_from P n l \rightarrow inductively_barred_at P n l \rightarrow False.$

Lemma find_left_path : $\forall P l n,$
 $is_path_from P (S n) l \rightarrow inductively_barred_at P n (false::l) \rightarrow is_path_from P n (true::l).$

Lemma Y_unique : $\forall P, demorgan_inductively_barred_at P \rightarrow$
 $\forall l1 l2, length l1 = length l2 \rightarrow Y P l1 \rightarrow Y P l2 \rightarrow l1 = l2.$

X is the translation of Y as a predicate

Definition X P n := $\exists l, length l = n \wedge Y P (true::l).$

Lemma Y_approx : $\forall P, demorgan_inductively_barred_at P \rightarrow$
 $\forall l, approx (X P) l \rightarrow Y P l.$

Main theorem

Theorem PreWeakKonigsLemma : $\forall P,$
 $demorgan_inductively_barred_at P \rightarrow infinite_from P [] \rightarrow has_infinite_path P.$

Lemma inductively_barred_at_decidable :
 $\forall P, (\forall l, P l \vee \neg P l) \rightarrow \forall n l, inductively_barred_at P n l \vee \neg inductively_barred_at P n l.$

Lemma `inductively_barred_at_is_path_from_decidable` :
 $\forall P, (\forall l, P l \vee \neg P l) \rightarrow \text{demorgan_inductively_barred_at } P.$

Main corollary

Corollary `WeakKonigsLemma` : $\forall P, (\forall l, P l \vee \neg P l) \rightarrow$
`infinite_from` P `[]` \rightarrow `has_infinite_path` $P.$

Chapter 69

Library `Coq.Logic.WeakFan`

A constructive proof of a non-standard version of the weak Fan Theorem in the formulation of which infinite paths are treated as predicates. The representation of paths as relations avoid the need for classical logic and unique choice. The idea of the proof comes from the proof of the weak König's lemma from separation in second-order arithmetic [*Simpson99*].

[*Simpson99*] Stephen G. Simpson. Subsystems of second order arithmetic, Cambridge University Press, 1999

```
Require Import List.
```

```
Import ListNotations.
```

inductively_barred P l means that P eventually holds above l

```
Inductive inductively_barred  $P$  : list bool  $\rightarrow$  Prop :=
```

```
| now  $l$  :  $P$   $l$   $\rightarrow$  inductively_barred  $P$   $l$ 
```

```
| propagate  $l$  :
```

```
  inductively_barred  $P$  (true:: $l$ )  $\rightarrow$ 
```

```
  inductively_barred  $P$  (false:: $l$ )  $\rightarrow$ 
```

```
  inductively_barred  $P$   $l$ .
```

approx X l says that l is a boolean representation of a prefix of X

```
Fixpoint approx  $X$  ( $l$ :list bool) :=
```

```
  match  $l$  with
```

```
  | []  $\Rightarrow$  True
```

```
  |  $b$ :: $l$   $\Rightarrow$  approx  $X$   $l$   $\wedge$  (if  $b$  then  $X$  (length  $l$ ) else  $\neg$   $X$  (length  $l$ ))
```

```
  end.
```

barred P means that for any infinite path represented as a predicate, the property P holds for some prefix of the path

```
Definition barred  $P$  :=
```

```
   $\forall$  ( $X$ :nat  $\rightarrow$  Prop),  $\exists$   $l$ , approx  $X$   $l$   $\wedge$   $P$   $l$ .
```

The proof proceeds by building a set Y of finite paths approximating either the smallest unbarred infinite path in P , if there is one (taking *true*>*false*), or the path *true::true::...* if P happens to be inductively_barred

```
Fixpoint  $Y$   $P$  ( $l$ :list bool) :=
```

```

match l with
| [] => True
| b::l =>
  Y P l ∧
  if b then inductively_barred P (false::l) else ¬ inductively_barred P (false::l)
end.

```

Lemma `Y_unique` : $\forall P\ l1\ l2, \text{length } l1 = \text{length } l2 \rightarrow Y\ P\ l1 \rightarrow Y\ P\ l2 \rightarrow l1 = l2$.

X is the translation of Y as a predicate

Definition `X P n` := $\exists l, \text{length } l = n \wedge Y\ P\ (\text{true}::l)$.

Lemma `Y_approx` : $\forall P\ l, \text{approx } (X\ P)\ l \rightarrow Y\ P\ l$.

Theorem `WeakFanTheorem` : $\forall P, \text{barred } P \rightarrow \text{inductively_barred } P\ []$.

Chapter 70

Library `Coq.Unicode.Utf8`

`Require Export Utf8_core.`

`Notation "x ≤ y" := (le x y) (at level 70, no associativity).`

`Notation "x ≥ y" := (ge x y) (at level 70, no associativity).`

Chapter 71

Library `Coq.Unicode.Utf8_core`

Notation " $\forall x .. y, P$ " := $(\forall x, .. (\forall y, P) ..)$
(at level 200, *x binder*, *y binder*, right associativity,
format "[' ' [' \forall x .. y ']' , '/' P ']'") : *type_scope*.

Notation " $\exists x .. y, P$ " := $(\exists x, .. (\exists y, P) ..)$
(at level 200, *x binder*, *y binder*, right associativity,
format "[' ' [' \exists x .. y ']' , '/' P ']'") : *type_scope*.

Notation " $x \vee y$ " := $(x \vee y)$ (at level 85, right associativity) : *type_scope*.

Notation " $x \wedge y$ " := $(x \wedge y)$ (at level 80, right associativity) : *type_scope*.

Notation " $x \rightarrow y$ " := $(x \rightarrow y)$
(at level 99, *y* at level 200, right associativity) : *type_scope*.

Notation " $x \leftrightarrow y$ " := $(x \leftrightarrow y)$ (at level 95, no associativity) : *type_scope*.

Notation " $\neg x$ " := $(\neg x)$ (at level 75, right associativity) : *type_scope*.

Notation " $x \neq y$ " := $(x \neq y)$ (at level 70) : *type_scope*.

Notation " $\lambda' x .. y, t$ " := $(\text{fun } x \Rightarrow .. (\text{fun } y \Rightarrow t) ..)$
(at level 200, *x binder*, *y binder*, right associativity,
format "[' ' [' λ' x .. y ']' , '/' t ']'").