The Coq Proof Assistant

The standard library

July 21, 2007

Version v8.1[1]

LogiCal Project

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

Vv8.1, July 21, 2007

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).

IntMap Representation of finite sets by an efficient structure of map (trees indexed by binary integers).

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

Module 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}
}
```

Section *Berardis_paradox*.

Excluded middle
Hypothesis $EM : \forall\ P$:Prop, $P \vee \neg\ P$.

Conditional on any proposition.
Definition *IFProp* $(P\ B$:Prop$)\ (e1\ e2$:$P) :=$
  match $EM\ B$ with
  | $or\_introl\ \_ \Rightarrow e1$
  | $or\_intror\ \_ \Rightarrow e2$
  end.

Axiom of choice applied to disjunction. Provable in Coq because of dependent elimination.
Lemma AC_IF :
 $\forall\ (P\ B$:Prop$)\ (e1\ e2$:$P)\ (Q$:$P \rightarrow$ Prop),
   $(B \rightarrow Q\ e1) \rightarrow (\tilde{}\ B \rightarrow Q\ e2) \rightarrow Q\ (IFProp\ B\ e1\ e2)$.

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) := $P \to Bool$.

A piece of theory about retracts
Section $Retracts$.

Variables A B : Prop.

Record $retract$: Prop :=
  $\{i : A \to B; j : B \to A; inv : \forall\ a{:}A, j\ (i\ a) = a\}$.

Record $retract\_cond$: Prop :=
  $\{i2 : A \to B; j2 : B \to A; inv2 : retract \to \forall\ a{:}A, j2\ (i2\ a) = a\}$.

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

End $Retracts$.

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

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

The paradoxical set
Definition $U := \forall\ P$:Prop, $pow\ P$.

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

Definition $g\ (h{:}pow\ U) : U :=$
  fun $X \Rightarrow$ 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 $u{:}U \Rightarrow Not\_b\ (u\ U\ u)$).

Lemma $not\_has\_fixpoint : R\ R = Not\_b\ (R\ R)$.

Theorem $classical\_proof\_irrelevence : T = F$.

End $Berardis\_paradox$.

Chapter 2

Module Coq.Logic.ChoiceFacts

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

We investigate the relations between the following choice and description principles

- AC_rel = relational form of the (non extensional) axiom of choice (a "set-theoretic" axiom of choice)
- AC_fun = functional form of the (non extensional) axiom of choice (a "type-theoretic" axiom of choice)
- AC! = functional relation reification (known as axiom of unique choice in topos theory, sometimes called principle of definite description in the context of constructive type theory)

- GAC_rel = guarded relational form of the (non extensional) axiom of choice
- GAC_fun = guarded functional form of the (non extensional) axiom of choice
- GAC! = guarded functional relation reification

- OAC_rel = "omniscient" relational form of the (non extensional) axiom of choice
- OAC_fun = "omniscient" functional form of the (non extensional) axiom of choice (called AC* in Bell *Bell*)
- OAC!

- ID_iota = intuitionistic definite description
- ID_epsilon = intuitionistic indefinite description

- D_iota = (weakly classical) definite description principle

- D_epsilon = (weakly classical) indefinite description principle

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

We let also

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

Table of contents

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.

Carlstrøm05 Jesper Carlstrøm, Interpreting descriptions in intentional type theory, Journal of Symbolic Logic 70(2):488-514, 2005.

Notation Local "'*inhabited' A*" := *A* (*at level* 10, *only parsing*).

2.1 Definitions

Choice, reification and description schemes

Section *ChoiceSchemes*.

Variables *A B* :Type.

Variables *P*:*A*→Prop.

Variables *R*:*A*→*B*→Prop.

2.1.1 Constructive choice and description

AC_rel

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

AC_fun

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

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

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

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

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

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

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

2.1.2 Weakly classical choice and description

GAC_rel

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

GAC_fun

Definition *GuardedFunctionalChoice_on* :=

$\forall\ P : A \to \mathsf{Prop}, \forall\ R : A \to B \to \mathsf{Prop},$
$\quad inhabited\ B \to$
$\quad (\forall\ x : A, P\ x \to \exists\ y : B, R\ x\ y) \to$
$\quad (\exists\ f : A \to B, \forall\ x, P\ x \to R\ x\ (f\ x)).$

GFR_fun

Definition *GuardedFunctionalRelReification_on* :=
$\forall\ P : A \to \mathsf{Prop}, \forall\ R : A \to B \to \mathsf{Prop},$
$\quad inhabited\ B \to$
$\quad (\forall\ x : A, P\ x \to \exists!\ y : B, R\ x\ y) \to$
$\quad (\exists\ f : A \to B, \forall\ x : A, P\ x \to R\ x\ (f\ x)).$

OAC_rel

Definition *OmniscientRelationalChoice_on* :=
$\forall\ R : A \to B \to \mathsf{Prop},$
$\quad \exists\ R' : A \to B \to \mathsf{Prop},$
$\quad\quad subrelation\ R'\ R \wedge \forall\ x : A, (\exists\ y : B, R\ x\ y) \to \exists!\ y, R'\ x\ y.$

OAC_fun

Definition *OmniscientFunctionalChoice_on* :=
$\forall\ R : A \to B \to \mathsf{Prop},$
$\quad inhabited\ B \to$
$\quad \exists\ f : A \to B, \forall\ x : A, (\exists\ y : B, R\ x\ y) \to R\ x\ (f\ x).$

D_epsilon

Definition *ClassicalIndefiniteDescription* :=
$\forall\ P{:}A \to \mathsf{Prop},$
$\quad A \to \{\ x{:}A \mid (\exists\ x, P\ x) \to P\ x\ \}.$

D_iota

Definition *ClassicalDefiniteDescription* :=
$\forall\ P{:}A \to \mathsf{Prop},$
$\quad A \to \{\ x{:}A \mid (\exists!\ x, P\ x) \to P\ x\ \}.$

End *ChoiceSchemes*.

Generalized schemes

Notation *RelationalChoice* :=
$(\forall\ A\ B,\ RelationalChoice\_on\ A\ B).$
Notation *FunctionalChoice* :=
$(\forall\ A\ B,\ FunctionalChoice\_on\ A\ B).$
Notation *FunctionalChoiceOnInhabitedSet* :=
$(\forall\ A\ B,\ inhabited\ B \to FunctionalChoice\_on\ A\ B).$
Notation *FunctionalRelReification* :=
$(\forall\ A\ B,\ FunctionalRelReification\_on\ A\ B).$

Notation *GuardedRelationalChoice* :=
$(\forall\ A\ B,\ GuardedRelationalChoice\_on\ A\ B).$

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

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

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

Subclassical schemes

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

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

Definition *SmallDrinker'sParadox* :=
∀ (*A*:Type) (*P*:*A* → Prop), *inhabited A* →
∃ *x*, (∃ *x*, *P x*) → *P x*.

2.2 AC_rel + PDP = 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) + the axiom of (parametric) definite description (aka axiom of unique choice)

This shows that the axiom of choice can be assumed (under its relational formulation) without known inconsistency with classical logic, though definite description conflicts with classical logic

Lemma *description_rel_choice_imp_funct_choice* :
∀ *A B* : Type,
FunctionalRelReification_on A B → *RelationalChoice_on A B* → *FunctionalChoice_on A B*.

Lemma *funct_choice_imp_rel_choice* :
∀ *A B*, *FunctionalChoice_on A B* → *RelationalChoice_on A B*.

Lemma *funct_choice_imp_description* :
∀ *A B*, *FunctionalChoice_on A B* → *FunctionalRelReification_on A B*.

Theorem *FunChoice_Equiv_RelChoice_and_ParamDefinDescr* :
∀ *A B*, *FunctionalChoice_on A B* ↔
RelationalChoice_on A B ∧ *FunctionalRelReification_on A B*.

2.3 Connection between the guarded, non guarded and descriptive choices and

We show that the guarded relational formulation of the axiom of Choice comes from the non guarded formulation in presence either of the independance of premises or proof-irrelevance

2.3.1 AC_rel + PI -> GAC_rel and AC_rel + IGP -> GAC_rel and GAC_rel = OAC_rel

Lemma *rel_choice_and_proof_irrel_imp_guarded_rel_choice* :
RelationalChoice → *ProofIrrelevance* → *GuardedRelationalChoice*.

Lemma *rel_choice_indep_of_general_premises_imp_guarded_rel_choice* :
∀ *A B*, *inhabited B* → *RelationalChoice_on A B* →
IndependenceOfGeneralPremises → *GuardedRelationalChoice_on A B*.

Lemma *guarded_rel_choice_imp_rel_choice* :
∀ *A B*, *GuardedRelationalChoice_on A B* → *RelationalChoice_on A B*.

OAC_rel = GAC_rel

Lemma *guarded_iff_omniscient_rel_choice* :
GuardedRelationalChoice ↔ *OmniscientRelationalChoice*.

2.3.2 AC_fun + IGP = GAC_fun = OAC_fun = AC_fun + Drinker

AC_fun + IGP = GAC_fun

Lemma *guarded_fun_choice_imp_indep_of_general_premises* :
GuardedFunctionalChoice → *IndependenceOfGeneralPremises*.

Lemma *guarded_fun_choice_imp_fun_choice* :
GuardedFunctionalChoice → *FunctionalChoiceOnInhabitedSet*.

Lemma *fun_choice_and_indep_general_prem_imp_guarded_fun_choice* :
FunctionalChoiceOnInhabitedSet → *IndependenceOfGeneralPremises*
→ *GuardedFunctionalChoice*.

AC_fun + Drinker = OAC_fun

This was already observed by Bell *Bell*

Lemma *omniscient_fun_choice_imp_small_drinker* :
OmniscientFunctionalChoice → *SmallDrinker'sParadox*.

Lemma *omniscient_fun_choice_imp_fun_choice* :
OmniscientFunctionalChoice → *FunctionalChoiceOnInhabitedSet*.

Lemma *fun_choice_and_small_drinker_imp_omniscient_fun_choice* :
FunctionalChoiceOnInhabitedSet → *SmallDrinker'sParadox*
→ *OmniscientFunctionalChoice*.

OAC_fun = GAC_fun

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

Lemma *guarded_iff_omniscient_fun_choice* :
GuardedFunctionalChoice ↔ *OmniscientFunctionalChoice*.

2.4 Derivability of choice for decidable relations with well-ordered codomain

Countable codomains, such as *nat*, can be equipped with a well-order, which implies the existence of a least element on inhabited decidable subsets. As a consequence, the relational form of the axiom of choice is derivable on *nat* for decidable relations.

We show instead that functional relation reification and the functional form of the axiom of choice are equivalent on decidable relation with *nat* as codomain

Require Import *Wf_nat*.
Require Import *Compare_dec*.
Require Import *Decidable*.
Require Import *Arith*.

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

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

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

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

2.5 Choice on dependent and non dependent function types are equivalent

2.5.1 Choice on dependent and non dependent function types are equivalent

Definition *DependentFunctionalChoice_on* (*A*:Type) (*B*:*A* → Type) :=

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

Notation *DependentFunctionalChoice* :=
$(\forall\ A\ (B{:}A{\to}\mathsf{Type}),$ *DependentFunctionalChoice_on* $B).$

The easy part

Theorem *dep_non_dep_functional_choice* :
DependentFunctionalChoice → *FunctionalChoice*.

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

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

Definition *proj1_inf* $(A\ B{:}\mathsf{Prop})\ (p : A{\wedge}B) :=$
let $(a,b) := p$ in a.

Theorem *non_dep_dep_functional_choice* :
FunctionalChoice → *DependentFunctionalChoice*.

2.5.2 Reification of dependent and non dependent functional relation are equivalent

Definition *DependentFunctionalRelReification_on* $(A{:}\mathsf{Type})\ (B{:}A \to \mathsf{Type}) :=$
$\forall\ (R{:}\forall\ x{:}A,\ B\ x \to \mathsf{Prop}),$
$(\forall\ x{:}A, \exists!\ y : B\ x, R\ x\ y) \to$
$(\exists\ f : (\forall\ x{:}A,\ B\ x), \forall\ x{:}A,\ R\ x\ (f\ x)).$

Notation *DependentFunctionalRelReification* :=
$(\forall\ A\ (B{:}A{\to}\mathsf{Type}),$ *DependentFunctionalRelReification_on* $B).$

The easy part

Theorem *dep_non_dep_functional_rel_reification* :
DependentFunctionalRelReification → *FunctionalRelReification*.

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

Theorem *non_dep_dep_functional_rel_reification* :
FunctionalRelReification → *DependentFunctionalRelReification*.

2.6 Non contradiction of constructive descriptions wrt functional axioms of choice

2.6.1 Non contradiction of indefinite description

Lemma *relative_non_contradiction_of_indefinite_desc* :
(*ConstructiveIndefiniteDescription* → *False*)
→ (*FunctionalChoice* → *False*).

Lemma *constructive_indefinite_descr_fun_choice* :
ConstructiveIndefiniteDescription → *FunctionalChoice*.

2.6.2 Non contradiction of definite description

Lemma *relative_non_contradiction_of_definite_descr* :
(*ConstructiveDefiniteDescription* → *False*)
→ (*FunctionalRelReification* → *False*).

Lemma *constructive_definite_descr_fun_reification* :
ConstructiveDefiniteDescription → *FunctionalRelReification*.

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

The idea for the following proof comes from *ChicliPottierSimpson02*

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

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

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

Require Import *Setoid*.

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

Chapter 3

Module Coq.Logic.ClassicalChoice

This file provides classical logic, and functional choice

This file extends ClassicalUniqueChoice.v with the axiom of choice. As ClassicalUniqueChoice.v, it implies the double-negation of excluded-middle in Set and leads to a classical world populated with non computable functions. Especially it conflicts with the impredicativity of Set, knowing that *true≠false* in Set.

Require Export *ClassicalUniqueChoice*.
Require Export *RelationalChoice*.
Require Import *ChoiceFacts*.

Definition *subset* (*U*:Type) (*P Q*:*U*→Prop) : Prop := ∀ *x*, *P x* → *Q x*.

Theorem *singleton_choice* :
∀ (*A* : Type) (*P* : *A*→Prop),
(∃ *x* : *A*, *P x*) → ∃ *P'* : *A*→Prop, *subset P' P* ∧ ∃! *x*, *P' x*.

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

Chapter 4

Module Coq.Logic.ClassicalDescription

This file provides classical logic and definite description

Classical definite description operator (i.e. iota) 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*.

Notation Local "'*inhabited*' *A*" := *A* (*at level* 200, *only parsing*).

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

The idea for the following proof comes from *ChicliPottierSimpson02*

Theorem *excluded_middle_informative* : ∀ *P*:Prop, {*P*} + {~ *P*}.

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

Church's iota operator

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

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

Axiom of unique "choice" (functional reification of functional relations)
Theorem *dependent_unique_choice* :
  ∀ (*A*:Type) (*B*:*A* → Type) (*R*:∀ *x*:*A*, *B x* → Prop),
    (∀ *x*:*A*, ∃! *y* : *B x*, *R x y*) →
    (∃ *f* : (∀ *x*:*A*, *B x*), ∀ *x*:*A*, *R x* (*f x*)).

Theorem *unique_choice* :
  ∀ (*A B*:Type) (*R*:*A* → *B* → Prop),

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

Compatibility lemmas

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

Chapter 5

Module Coq.Logic.ClassicalEpsilon

This file provides classical logic and indefinite description (Hilbert's epsilon operator)

Classical epsilon's operator (i.e. indefinite description) 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*.

Axiom *constructive_indefinite_description* :
  $\forall$ (A : Type) (P : A→Prop),
    ($\exists$ x, P x) → { x : A | P x }.

Lemma *constructive_definite_description* :
  $\forall$ (A : Type) (P : A→Prop),
    ($\exists!$ x, P x) → { x : A | P x }.

Theorem *excluded_middle_informative* : $\forall$ P:Prop, {P} + {~ P}.

Theorem *classical_indefinite_description* :
  $\forall$ (A : Type) (P : A→Prop), *inhabited* A →
    { x : A | ($\exists$ 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) :
  ($\exists$ x, P x) → P (*epsilon* i P)
  := *proj2_sig* (*classical_indefinite_description* P i).

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

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

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

Opaque epsilon.

Weaker lemmas (compatibility lemmas)

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

Chapter 6

Module Coq.Logic.ClassicalFacts

Some facts and definitions about classical logic

Table of contents:

6.1 Prop degeneracy = excluded-middle + prop extensionality

i.e. $(\forall\ A,\ A{=}\mathit{True} \vee A{=}\mathit{False}) \leftrightarrow (\forall\ A,\ A \setminus/ \tilde{}\ A) \wedge (\forall\ A\ B,\ (A \leftrightarrow B) \rightarrow A{=}B)$

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

prop_extensionality asserts that equivalent formulas are equal
Definition *prop_extensionality* := $\forall\ A\ B$: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* := $\forall\ A$:Prop, $A \vee \neg\ A$.

We show *prop_degeneracy* $\leftrightarrow$ (*prop_extensionality* $\wedge$ *excluded_middle*)

Lemma *prop_degen_ext* : *prop_degeneracy* → *prop_extensionality*.

Lemma *prop_degen_em* : *prop_degeneracy* → *excluded_middle*.

Lemma *prop_ext_em_degen* :
prop_extensionality → *excluded_middle* → *prop_degeneracy*.

6.2 Classical logic and proof-irrelevance

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

We successively show that:

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

Definition *inhabited* (A:Prop) := A.

Lemma *prop_ext_A_eq_A_imp_A* :
prop_extensionality → ∀ A:Prop, *inhabited* $A \to (A \to A) = A$.

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

Lemma *prop_ext_retract_A_A_imp_A* :
prop_extensionality → ∀ A:Prop, *inhabited* $A \to$ *retract* $A\ (A \to A)$.

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

Lemma *ext_prop_fixpoint* :
prop_extensionality → ∀ A:Prop, *inhabited* $A \to$ *has_fixpoint* A.

6.2.2 CC |- prop_ext /\ dep elim on bool -> proof-irrelevance

proof_irrelevance asserts equality of all proofs of a given formula
Definition *proof_irrelevance* := ∀ (A:Prop) ($a1$ $a2$:A), $a1 = a2$.

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

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

Section *Proof_irrelevance_gen*.

Variable *bool* : Prop.
Variable *true* : *bool*.
Variable *false* : *bool*.

Hypothesis *bool_elim* : ∀ *C*:Prop, *C* → *C* → *bool* → *C*.
Hypothesis
bool_elim_redl : ∀ (*C*:Prop) (*c1 c2*:*C*), *c1* = *bool_elim C c1 c2 true*.
Hypothesis
bool_elim_redr : ∀ (*C*:Prop) (*c1 c2*:*C*), *c2* = *bool_elim C c1 c2 false*.
Let *bool_dep_induction* :=
∀ *P*:*bool* → Prop, *P true* → *P false* → ∀ *b*:*bool*, *P b*.

Lemma *aux* : *prop_extensionality* → *bool_dep_induction* → *true* = *false*.

Lemma *ext_prop_dep_proof_irrel_gen* :
prop_extensionality → *bool_dep_induction* → *proof_irrelevance*.

End *Proof_irrelevance_gen*.

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

Section *Proof_irrelevance_Prop_Ext_CC*.

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

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

Lemma *ext_prop_dep_proof_irrel_cc* :
prop_extensionality → *BoolP_dep_induction* → *proof_irrelevance*.

End *Proof_irrelevance_Prop_Ext_CC*.

6.2.3 CIC |- prop. ext. -> proof-irrelevance

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

Section *Proof_irrelevance_CIC*.

Inductive *boolP* : Prop :=
| *trueP* : *boolP*
| *falseP* : *boolP*.
Definition *boolP_elim_redl* (*C*:Prop) (*c1 c2*:*C*) :
c1 = *boolP_ind C c1 c2 trueP* := *refl_equal c1*.
Definition *boolP_elim_redr* (*C*:Prop) (*c1 c2*:*C*) :
c2 = *boolP_ind C c1 c2 falseP* := *refl_equal c2*.

Scheme *boolP_indd* := *Induction for boolP Sort* Prop.

Lemma *ext_prop_dep_proof_irrel_cic* : *prop_extensionality* → *proof_irrelevance*.

End *Proof_irrelevance_CIC*.

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

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

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

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

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

Reference:

[*Coquand90*] T. Coquand, "Metamathematical Investigations of a Calculus of Constructions", Proceedings of Logic in Computer Science (LICS'90), 1990.

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

Require Import *Hurkens*.

Section *Proof_irrelevance_EM_CC*.

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

Hypothesis em : $\forall$ A:Prop, or A ($\tilde{}$ A).
Variable B : Prop.
Variables $b1$ $b2$: B.

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

Definition $p2b$ A := or_elim A ($\tilde{}$ A) B (fun _ $\Rightarrow$ $b1$) (fun _ $\Rightarrow$ $b2$) (em A).
Definition $b2p$ b := $b1$ = b.

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

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

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

Theorem $proof\_irrelevance\_cc$: $b1 = b2$.

End $Proof\_irrelevance\_EM\_CC$.

Remark: Hurkens' paradox still holds with a retract from the _negative_fragment of Prop into $bool$, hence weak classical logic, i.e. $\forall$ A, $\neg A \backslash/ \tilde{}\tilde{} A$, is enough for deriving proof-irrelevance.

6.2.5 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:Prop, $A \vee \neg A$.

Definition or_elim_redl (A B C:Prop) (f:$A \rightarrow C$) (g:$B \rightarrow C$)
(a:A) : f a = or_ind f g (or_introl B a) := $refl\_equal$ (f a).
Definition or_elim_redr (A B C:Prop) (f:$A \rightarrow C$) (g:$B \rightarrow C$)
(b:B) : g b = or_ind f g (or_intror A b) := $refl\_equal$ (g b).
Scheme or_indd := $Induction\ for\ or\ Sort$ Prop.

Theorem $proof\_irrelevance\_cci$: $\forall$ (B:Prop) ($b1$ $b2$:B), $b1 = b2$.

End $Proof\_irrelevance\_CCI$.

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

6.3 Weak classical axioms

We show the following increasing in the strength of axioms:

- weak excluded-middle
- right distributivity of implication over disjunction and Gödel-Dummet axiom

- independence of general premises and drinker's paradox
- excluded-middle

6.3.1 Weak excluded-middle

The weak classical logic based on $\tilde{}\tilde{}A \vee \neg A$ is referred to with name KC in {*ChagrovZakharyaschev97*]

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

Definition *weak_excluded_middle* :=
$\forall$ *A*:Prop, $\tilde{}\tilde{}A \vee \neg A$.

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*:Prop, $((A \to B) \to B) \vee (A \to B)$.

6.3.2 Gödel-Dummett axiom

$(A \to B) \vee (B \to A)$ is studied in [*Dummett59*] and is based on [*Gödel33*].

[*Dummett59*] Michael A. E. Dummett. "A Propositional Calculus with a Denumerable Matrix", In the Journal of Symbolic Logic, Vol 24 No. 2(1959), pp 97-103.

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

Definition *GodelDummett* := $\forall$ *A B*:Prop, $(A \to B) \vee (B \to A)$.

Lemma *excluded_middle_Godel_Dummett* : *excluded_middle* $\to$ *GodelDummett*.

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

Definition *RightDistributivityImplicationOverDisjunction* :=
$\forall$ *A B C*:Prop, $(C \to A \vee B) \to (C \to A) \vee (C \to B)$.

Lemma *Godel_Dummett_iff_right_distr_implication_over_disjunction* :
GodelDummett $\leftrightarrow$ *RightDistributivityImplicationOverDisjunction*.

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

Lemma *Godel_Dummett_weak_excluded_middle* :
GodelDummett $\to$ *weak_excluded_middle*.

6.3.3 Independence of general premises and drinker's paradox

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

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

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

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

Notation Local "'*inhabited' A*" := *A* (*at level* 10, *only parsing*).

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

Lemma
  independence_general_premises_right_distr_implication_over_disjunction :
  IndependenceOfGeneralPremises → *RightDistributivityImplicationOverDisjunction*.

Lemma *independence_general_premises_Godel_Dummett* :
  IndependenceOfGeneralPremises → *GodelDummett*.

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

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

Lemma *independence_general_premises_drinker* :
  IndependenceOfGeneralPremises ↔ *DrinkerParadox*.

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

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

Lemma *excluded_middle_independence_general_premises* :
  generalized_excluded_middle → *DrinkerParadox*.

Chapter 7

Module Coq.Logic.Classical_Pred_Set

This file is obsolete, use Classical_Pred_Type.v via Classical.v instead

Classical Predicate Logic on Set

Require Import *Classical_Pred_Type*.

Section *Generic*.
Variable U : Set.

de Morgan laws for quantifiers

Lemma *not_all_ex_not* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\forall\ n{:}U, P\ n) \to \exists\ n : U, \neg\ P\ n.$

Lemma *not_all_not_ex* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\forall\ n{:}U, \neg\ P\ n) \to \exists\ n : U, P\ n.$

Lemma *not_ex_all_not* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\exists\ n : U, P\ n) \to \forall\ n{:}U, \neg\ P\ n.$

Lemma *not_ex_not_all* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\exists\ n : U, \neg\ P\ n) \to \forall\ n{:}U, P\ n.$

Lemma *ex_not_not_all* :
$\forall\ P{:}U \to \mathsf{Prop}, (\exists\ n : U, \neg\ P\ n) \to \neg\ (\forall\ n{:}U, P\ n).$

Lemma *all_not_not_ex* :
$\forall\ P{:}U \to \mathsf{Prop}, (\forall\ n{:}U, \neg\ P\ n) \to \neg\ (\exists\ n : U, P\ n).$

End *Generic*.

Chapter 8

Module Coq.Logic.Classical_Pred_Type

Classical Predicate Logic on Type

Require Import *Classical_Prop*.

Section *Generic*.
Variable U : Type.

de Morgan laws for quantifiers

Lemma *not_all_not_ex* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\forall\ n{:}U, \neg\ P\ n) \to \exists\ n : U, P\ n.$

Lemma *not_all_ex_not* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\forall\ n{:}U, P\ n) \to \exists\ n : U, \neg\ P\ n.$

Lemma *not_ex_all_not* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\exists\ n : U, P\ n) \to \forall\ n{:}U, \neg\ P\ n.$

Lemma *not_ex_not_all* :
$\forall\ P{:}U \to \mathsf{Prop}, \neg\ (\exists\ n : U, \neg\ P\ n) \to \forall\ n{:}U, P\ n.$

Lemma *ex_not_not_all* :
$\forall\ P{:}U \to \mathsf{Prop}, (\exists\ n : U, \neg\ P\ n) \to \neg\ (\forall\ n{:}U, P\ n).$

Lemma *all_not_not_ex* :
$\forall\ P{:}U \to \mathsf{Prop}, (\forall\ n{:}U, \neg\ P\ n) \to \neg\ (\exists\ n : U, P\ n).$

End *Generic*.

Chapter 9

Module Coq.Logic.Classical_Prop

Classical Propositional Logic

Require Import *ClassicalFacts*.

Hint *Unfold not*: *core*.

Axiom *classic* : $\forall$ *P*:Prop, *P* $\vee$ $\neg$ *P*.

Lemma *NNPP* : $\forall$ *p*:Prop, $\neg$ $\neg$ *p* $\rightarrow$ *p*.

Peirce's law states $\forall$ *P Q*:Prop, ((*P* $\rightarrow$ *Q*) $\rightarrow$ *P*) $\rightarrow$ *P*. Thanks to $\forall$ *P*, *False* $\rightarrow$ *P*, it is equivalent to the following form

Lemma *Peirce* : $\forall$ *P*:Prop, ((*P* $\rightarrow$ *False*) $\rightarrow$ *P*) $\rightarrow$ *P*.

Lemma *not_imply_elim* : $\forall$ *P Q*:Prop, $\neg$ (*P* $\rightarrow$ *Q*) $\rightarrow$ *P*.

Lemma *not_imply_elim2* : $\forall$ *P Q*:Prop, $\neg$ (*P* $\rightarrow$ *Q*) $\rightarrow$ $\neg$ *Q*.

Lemma *imply_to_or* : $\forall$ *P Q*:Prop, (*P* $\rightarrow$ *Q*) $\rightarrow$ $\neg$ *P* $\vee$ *Q*.

Lemma *imply_to_and* : $\forall$ *P Q*:Prop, $\neg$ (*P* $\rightarrow$ *Q*) $\rightarrow$ *P* $\wedge$ $\neg$ *Q*.

Lemma *or_to_imply* : $\forall$ *P Q*:Prop, $\neg$ *P* $\vee$ *Q* $\rightarrow$ *P* $\rightarrow$ *Q*.

Lemma *not_and_or* : $\forall$ *P Q*:Prop, $\neg$ (*P* $\wedge$ *Q*) $\rightarrow$ $\neg$ *P* $\vee$ $\neg$ *Q*.

Lemma *or_not_and* : $\forall$ *P Q*:Prop, $\neg$ *P* $\vee$ $\neg$ *Q* $\rightarrow$ $\neg$ (*P* $\wedge$ *Q*).

Lemma *not_or_and* : $\forall$ *P Q*:Prop, $\neg$ (*P* $\vee$ *Q*) $\rightarrow$ $\neg$ *P* $\wedge$ $\neg$ *Q*.

Lemma *and_not_or* : $\forall$ *P Q*:Prop, $\neg$ *P* $\wedge$ $\neg$ *Q* $\rightarrow$ $\neg$ (*P* $\vee$ *Q*).

Lemma *imply_and_or* : $\forall$ *P Q*:Prop, (*P* $\rightarrow$ *Q*) $\rightarrow$ *P* $\vee$ *Q* $\rightarrow$ *Q*.

Lemma *imply_and_or2* : $\forall$ *P Q R*:Prop, (*P* $\rightarrow$ *Q*) $\rightarrow$ *P* $\vee$ *R* $\rightarrow$ *Q* $\vee$ *R*.

Lemma *proof_irrelevance* : $\forall$ (*P*:Prop) (*p1 p2*:*P*), *p1* = *p2*.

Ltac *classical_right* := match *goal* with
| _:_ |-?*X1* $\vee$ _ $\Rightarrow$ (*elim* (*classic X1*);*intro*;[left;trivial|right])
end.

Ltac *classical_left* := match *goal* with

```
  | _:_ ⊢ _ \/ ?X1 ⇒ (elim (classic X1);intro;[right;trivial|left])
  end.

Require Export EqdepFacts.

Module Eq_rect_eq.

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

End Eq_rect_eq.

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

Chapter 10

Module Coq.Logic.Classical_Type

This file is obsolete, use Classical.v instead

Classical Logic for Type

Require Export *Classical_Prop*.
Require Export *Classical_Pred_Type*.

Chapter 11

Module Coq.Logic.ClassicalUniqueChoice

This file provides classical logic and unique choice

Classical logic and unique choice, as shown in *ChicliPottierSimpson02*, implies the double-negation of excluded-middle in Set, hence it implies a strongly classical world. Especially it conflicts with the impredicativity of Set.

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

```
Require Export Classical.

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

Unique choice reifies functional relations into functions

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

The following proof comes from *ChicliPottierSimpson02*

```
Require Import Setoid.

Theorem classic_set : ((∀ P:Prop, {P} + {~ P}) → False) → False.
```

Chapter 12

Module Coq.Logic.Classical

Classical Logic

Require Export *Classical_Prop*.
Require Export *Classical_Pred_Type*.

Chapter 13

Module Coq.Logic.ConstructiveEpsilon

This module proves the constructive description schema, which infers the sigma-existence (i.e., Set-existence) of a witness to a predicate from the regular existence (i.e., Prop-existence). One requires that the underlying set is countable and that the predicate is decidable.

Coq does not allow case analysis on sort Set when the goal is in Prop. Therefore, one cannot eliminate $\exists\ n,\ P\ n$ in order to show $\{n : nat \mid P\ n\}$. However, one can perform a recursion on an inductive predicate in sort Prop so that the returning type of the recursion is in Set. 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 *Acc_iter* in the module Coq.Init.Wf. There, recursion is done on an inductive predicate *Acc* and the resulting type is in Type.

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 *Acc_iter*, we define a relation R and prove that if $\exists\ n,\ P\ n$ then 0 is accessible with respect to R. Then, by induction on the definition of $Acc\ R\ 0$, we show $\{n : nat \mid P\ n\}$.

Contributed by Yevgeniy Makarov

Require Import *Arith*.

Section *ConstructiveIndefiniteDescription*.

Variable $P : nat \to$ Prop.

Hypothesis *P_decidable* : $\forall\ x : nat, \{P\ x\} + \{\tilde{}\ P\ x\}$.

To find a witness of P constructively, we define an algorithm that tries P on all natural numbers starting from 0 and going up. The relation R describes the connection between the two successive numbers we try. Namely, y is R-less then x if we try y after x, i.e., $y = S\ x$ and $P\ x$ is false. Then the absence of an infinite R-descending chain from 0 is equivalent to the termination of our searching algorithm.

Let $R\ (x\ y : nat) := (x = S\ y \wedge \neg\ P\ y)$.
Notation Local "'*acc' x*" := $(Acc\ R\ x)$ $(at\ level\ 10)$.

Lemma *P_implies_acc* : $\forall\ x : nat, P\ x \to acc\ x$.

Lemma *P_eventually_implies_acc* : $\forall\ (x : nat)\ (n : nat), P\ (n + x) \to acc\ x$.

Corollary $P\_eventually\_implies\_acc\_ex : (\exists\ n : nat,\ P\ n) \rightarrow acc\ 0$.

In the following statement, we use the trick with recursion on *Acc*. This is also where decidability of P is used.

Theorem $acc\_implies\_P\_eventually : acc\ 0 \rightarrow \{n : nat \mid P\ n\}$.

Theorem $constructive\_indefinite\_description\_nat : (\exists\ n : nat,\ P\ n) \rightarrow \{n : nat \mid P\ n\}$.

End *ConstructiveIndefiniteDescription*.

Section *ConstructiveEpsilon*.

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

Variable A : Type.
Variable $f : A \rightarrow nat$.
Variable $g : nat \rightarrow A$.

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

Variable $P : A \rightarrow$ Prop.

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

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

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

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

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

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

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

End *ConstructiveEpsilon*.

Theorem *choice* :
$\forall\ (A\ B :$ Type$)\ (f : B \rightarrow nat)\ (g : nat \rightarrow B)$,
$(\forall\ x : B,\ g\ (f\ x) = x) \rightarrow$
$\forall\ (R : A \rightarrow B \rightarrow$ Prop$)$,
$(\forall\ (x : A)\ (y : B),\ \{R\ x\ y\} + \{\tilde{}\ R\ x\ y\}) \rightarrow$
$(\forall\ x : A, \exists\ y : B,\ R\ x\ y) \rightarrow$
$(\exists\ f : A \rightarrow B, \forall\ x : A,\ R\ x\ (f\ x))$.

Chapter 14

Module Coq.Logic.DecidableTypeEx

```
Require Import DecidableType OrderedType OrderedTypeEx.
```

14.1 Examples of Decidable Type structures.

A particular case of *DecidableType* where the equality is the usual one of Coq.

```
Module Type UsualDecidableType.
 Parameter t : Set.
 Definition eq := @eq t.
 Definition eq_refl := @refl_equal t.
 Definition eq_sym := @sym_eq t.
 Definition eq_trans := @trans_eq t.
 Parameter eq_dec : ∀ x y, { eq x y }+{~eq x y }.
End UsualDecidableType.
```

a *UsualDecidableType* is in particular an *DecidableType*.

```
Module UDT_to_DT (U:UsualDecidableType) <: DecidableType := U.
```

An OrderedType can be seen as a DecidableType

```
Module OT_as_DT (O:OrderedType) <: DecidableType.
 Module OF := OrderedTypeFacts O.
 Definition t := O.t.
 Definition eq := O.eq.
 Definition eq_refl := O.eq_refl.
 Definition eq_sym := O.eq_sym.
 Definition eq_trans := O.eq_trans.
 Definition eq_dec := OF.eq_dec.
End OT_as_DT.
```

(Usual) Decidable Type for *nat*, *positive*, *N*, *Z*

```
Module Nat_as_DT <: UsualDecidableType := OT_as_DT (Nat_as_OT).
Module Positive_as_DT <: UsualDecidableType := OT_as_DT (Positive_as_OT).
```

Module $N\_as\_DT <: UsualDecidableType := OT\_as\_DT\ (N\_as\_OT)$.
Module $Z\_as\_DT <: UsualDecidableType := OT\_as\_DT\ (Z\_as\_OT)$.

Chapter 15

Module Coq.Logic.DecidableType

Require Export *SetoidList*.

15.1 Types with decidable Equalities (but no ordering)

Module Type *DecidableType*.

Parameter t : Set.

Parameter eq : $t \to t \to$ Prop.

Axiom *eq_refl* : $\forall\ x : t, eq\ x\ x$.
Axiom *eq_sym* : $\forall\ x\ y : t, eq\ x\ y \to eq\ y\ x$.
Axiom *eq_trans* : $\forall\ x\ y\ z : t, eq\ x\ y \to eq\ y\ z \to eq\ x\ z$.

Parameter *eq_dec* : $\forall\ x\ y : t, \{\ eq\ x\ y\ \} + \{\ \neg\ eq\ x\ y\ \}$.

Hint Immediate *eq_sym*.
Hint *Resolve eq_refl eq_trans*.

End *DecidableType*.

15.2 Additional notions about keys and datas used in FMap

Module *KeyDecidableType*(*D*:*DecidableType*).
Import *D*.

Section *Elt*.
Variable *elt* : Set.
Notation *key*:=*t*.

Definition *eqk* $(p\ p'{:}key \times elt) := eq\ (fst\ p)\ (fst\ p')$.
Definition *eqke* $(p\ p'{:}key \times elt) :=$
$eq\ (fst\ p)\ (fst\ p') \wedge (snd\ p) = (snd\ p')$.

Hint *Unfold eqk eqke*.

Hint *Extern* 2 (*eqke ?a ?b*) ⇒ *split*.

Lemma *eqke_eqk* : ∀ *x x'*, *eqke x x'* → *eqk x x'*.

Lemma *eqk_refl* : ∀ *e*, *eqk e e*.

Lemma *eqke_refl* : ∀ *e*, *eqke e e*.

Lemma *eqk_sym* : ∀ *e e'*, *eqk e e'* → *eqk e' e*.

Lemma *eqke_sym* : ∀ *e e'*, *eqke e e'* → *eqke e' e*.

Lemma *eqk_trans* : ∀ *e e' e"*, *eqk e e'* → *eqk e' e"* → *eqk e e"*.

Lemma *eqke_trans* : ∀ *e e' e"*, *eqke e e'* → *eqke e' e"* → *eqke e e"*.

Hint *Resolve eqk_trans eqke_trans eqk_refl eqke_refl*.
Hint Immediate *eqk_sym eqke_sym*.

Lemma *InA_eqke_eqk* :
∀ *x m*, *InA eqke x m* → *InA eqk x m*.
Hint *Resolve InA_eqke_eqk*.

Lemma *InA_eqk* : ∀ *p q m*, *eqk p q* → *InA eqk p m* → *InA eqk q m*.

Definition *MapsTo* (*k*:*key*)(*e*:*elt*):= *InA eqke* (*k*,*e*).
Definition *In k m* := ∃ *e*:*elt*, *MapsTo k e m*.

Hint *Unfold MapsTo In*.

Lemma *In_alt* : ∀ *k l*, *In k l* ↔ ∃ *e*, *InA eqk* (*k*,*e*) *l*.

Lemma *MapsTo_eq* : ∀ *l x y e*, *eq x y* → *MapsTo x e l* → *MapsTo y e l*.

Lemma *In_eq* : ∀ *l x y*, *eq x y* → *In x l* → *In y l*.

Lemma *In_inv* : ∀ *k k' e l*, *In k* ((*k'*,*e*) :: *l*) → *eq k k'* ∨ *In k l*.

Lemma *In_inv_2* : ∀ *k k' e e' l*,
InA eqk (*k*, *e*) ((*k'*, *e'*) :: *l*) → ¬ *eq k k'* → *InA eqk* (*k*, *e*) *l*.

Lemma *In_inv_3* : ∀ *x x' l*,
InA eqke x (*x'* :: *l*) → ¬ *eqk x x'* → *InA eqke x l*.

End *Elt*.

Hint *Unfold eqk eqke*.
Hint *Extern* 2 (*eqke ?a ?b*) ⇒ *split*.
Hint *Resolve eqk_trans eqke_trans eqk_refl eqke_refl*.
Hint Immediate *eqk_sym eqke_sym*.
Hint *Resolve InA_eqke_eqk*.
Hint *Unfold MapsTo In*.
Hint *Resolve In_inv_2 In_inv_3*.

End *KeyDecidableType*.

Chapter 16

Module Coq.Logic.Decidable

Properties of decidable propositions

Definition *decidable* (P:Prop) := $P \vee \neg P$.

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

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

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

Theorem *dec_or* :
$\forall$ A B:Prop, *decidable* $A \rightarrow$ *decidable* $B \rightarrow$ *decidable* $(A \vee B)$.

Theorem *dec_and* :
$\forall$ A B:Prop, *decidable* $A \rightarrow$ *decidable* $B \rightarrow$ *decidable* $(A \wedge B)$.

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

Theorem *dec_imp* :
$\forall$ A B:Prop, *decidable* $A \rightarrow$ *decidable* $B \rightarrow$ *decidable* $(A \rightarrow B)$.

Theorem *not_not* : $\forall$ P:Prop, *decidable* $P \rightarrow \neg \neg P \rightarrow P$.

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

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

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

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

Chapter 17

Module 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 *CarlstrÃ¶m* 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

CarlstrÃ¶m04 Jesper CarlstrÃ¶m, EM + Ext_+ AC_int <-> AC_ext, Mathematical Logic Quaterly, vol 50(3), pp 236-240, 2004.

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

Section *PredExt_RelChoice_imp_EM*.

The axiom of extensionality for predicates

Definition *PredicateExtensionality* :=
  $\forall\ P\ Q$:*bool* $\rightarrow$ Prop, $(\forall\ b$:*bool*, $P\ b \leftrightarrow Q\ b) \rightarrow P = Q$.

From predicate extensionality we get propositional extensionality hence proof-irrelevance

Require Import *ClassicalFacts*.

Variable *pred_extensionality* : *PredicateExtensionality*.

Lemma *prop_ext* : $\forall\ A\ B$:Prop, $(A \leftrightarrow B) \rightarrow A = B$.

Lemma *proof_irrel* : $\forall\ (A$:Prop$)\ (a1\ \ a2$:$A)$, $a1 = a2$.

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

Require Import *ChoiceFacts*.

Variable *rel_choice* : *RelationalChoice*.

Lemma *guarded_rel_choice* : *GuardedRelationalChoice*.

The form of choice we need: there is a functional relation which chooses an element in any non empty subset of bool

Require Import *Bool*.

Lemma *AC_bool_subset_to_bool* :
$\exists\ R : (bool \rightarrow$ Prop$) \rightarrow bool \rightarrow$ Prop,
$(\forall\ P$:$bool \rightarrow$ Prop,
$(\exists\ b : bool, P\ b) \rightarrow$
$\exists\ b : bool, P\ b \wedge R\ P\ b \wedge (\forall\ b'$:$bool, R\ P\ b' \rightarrow b = b'))$.

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

Theorem *pred_ext_and_rel_choice_imp_EM* : $\forall\ P$:Prop, $P \vee \neg\ P$.

first we exhibit the choice functional relation R

the actual "decision": is (R class_of_true) = true or false?

the actual "decision": is (R class_of_false) = true or false?

case where P is false: (R class_of_true)=true /\ (R class_of_false)=false

cases where P is true

End *PredExt_RelChoice_imp_EM*.

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

This is an adaptation of Diaconescu's paradox exploiting that proof-irrelevance is some form of extensionality

Section *ProofIrrel_RelChoice_imp_EqEM*.

Variable *rel_choice* : *RelationalChoice*.

Variable *proof_irrelevance* : ∀ *P*:Prop , ∀ *x y*:*P*, *x*=*y*.

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

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

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

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

Definition *a1'*:*A'*.

Definition *a2'*:*A'*.

By proof-irrelevance, projection is a retraction

Lemma *projT1_injective* : *a1*=*a2* → *a1'*=*a2'*.

But from the actual proofs of being in *A'*, we can assert in the proof-irrelevant world the existence of relevant boolean witnesses

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

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

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

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

Declare Implicit Tactic *auto*.

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

End *ProofIrrel_RelChoice_imp_EqEM*.

17.3 Extensional Hilbert's epsilon description operator -> Excluded-Middle

Proof sketch from Bell *Bell93* (with thanks to P. CastÃ©ran)

Notation Local "'*inhabited' A*" := *A* (*at level* 10, *only parsing*).

Section *ExtensionalEpsilon_imp_EM*.

Variable *epsilon* : ∀ *A* : Type, *inhabited A* → (*A* → Prop) → *A*.

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

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

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

Theorem *extensional_epsilon_imp_EM* : ∀ *P*:Prop, *P* ∨ ¬ *P*.

End *ExtensionalEpsilon_imp_EM*.

Chapter 18

Module Coq.Logic.Eqdep_dec

We prove that there is only one proof of $x=x$, i.e *refl_equal 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:

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

Section *EqdepDec*.

Variable A : Type.

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

Remark *trans_sym_eq* : $\forall\ (x\ y{:}A)\ (u{:}x = y)$, *comp* $u\ u =$ *refl_equal* y.

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

Variable x : A.

Let *nu* $(y{:}A)\ (u{:}x = y) : x = y :=$
  match *eq_dec* $x\ y$ with
    | *or_introl eqxy* $\Rightarrow$ *eqxy*

```
    | or_intror neqxy ⇒ False_ind _ (neqxy u)
  end.

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

Let nu_inv (y:A) (v:x = y) : x = y := comp (nu (refl_equal x)) v.

Remark nu_left_inv : ∀ (y:A) (u:x = y), nu_inv (nu u) = u.

Theorem eq_proofs_unicity : ∀ (y:A) (p1 p2:x = y), p1 = p2.

Theorem K_dec :
  ∀ P:x = x → Prop, P (refl_equal x) → ∀ p:x = x, P p.
```

The corollary

```
Let proj (P:A → Prop) (exP:ex P) (def:P x) : P x :=
  match exP with
    | ex_intro x' prf ⇒
      match eq_dec x' x with
        | or_introl eqprf ⇒ eq_ind x' P prf x eqprf
        | _ ⇒ def
      end
  end.

Theorem inj_right_pair :
  ∀ (P:A → Prop) (y y':P x),
    ex_intro P x y = ex_intro P x y' → y = y'.

End EqdepDec.

Require Import EqdepFacts.
```

We deduce axiom K for (decidable) types

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

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

We deduce the *eq_rect_eq* axiom for (decidable) types

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

18.1.1 Definition of the functor that builds properties of dependent equalities on decidable sets in Type

The signature of decidable sets in Type

Module Type *DecidableType*.

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

End *DecidableType*.

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

Module *DecidableEqDep* (M:*DecidableType*).

Import M.

Invariance by Substitution of Reflexive Equality Proofs

Lemma *eq_rect_eq* :
$\forall\ (p{:}U)\ (Q{:}U \to \mathsf{Type})\ (x{:}Q\ p)\ (h{:}p = p), x = eq\_rect\ p\ Q\ x\ p\ h$.

Injectivity of Dependent Equality

Theorem *eq_dep_eq* :
$\forall\ (P{:}U{\to}\mathsf{Type})\ (p{:}U)\ (x\ y{:}P\ p), eq\_dep\ U\ P\ p\ x\ p\ y \to x = y$.

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* : $\forall\ (x{:}U)\ (p{:}x = x), p = refl\_equal\ x$.

Streicher's axiom K

Lemma *Streicher_K* :
$\forall\ (x{:}U)\ (P{:}x = x \to \mathsf{Prop}), P\ (refl\_equal\ x) \to \forall\ p{:}x = x, P\ p$.

Injectivity of equality on dependent pairs in Type

Lemma *inj_pairT2* :
$\forall\ (P{:}U \to \mathsf{Type})\ (p{:}U)\ (x\ y{:}P\ p)$,
$existT\ P\ p\ x = existT\ P\ p\ y \to x = y$.

Proof-irrelevance on subsets of decidable sets

Lemma *inj_pairP2* :
$\forall\ (P{:}U \to \mathsf{Prop})\ (x{:}U)\ (p\ q{:}P\ x)$,
$ex\_intro\ P\ x\ p = ex\_intro\ P\ x\ q \to p = q$.

End *DecidableEqDep*.

18.1.2 B Definition of the functor that builds properties of dependent equalities on decidable sets in Set

The signature of decidable sets in Set

Module Type *DecidableSet*.

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

End *DecidableSet*.

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

Module *DecidableEqDepSet* (M:*DecidableSet*).

Import M.
Module N:=*DecidableEqDep*(M).

Invariance by Substitution of Reflexive Equality Proofs

Lemma *eq_rect_eq* :
$\forall\ (p{:}U)\ (Q{:}U \to \mathsf{Type})\ (x{:}Q\ p)\ (h{:}p = p), x = \mathit{eq\_rect}\ p\ Q\ x\ p\ h$.

Injectivity of Dependent Equality

Theorem *eq_dep_eq* :
$\forall\ (P{:}U \to \mathsf{Type})\ (p{:}U)\ (x\ y{:}P\ p), \mathit{eq\_dep}\ U\ P\ p\ x\ p\ y \to x = y$.

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* : $\forall\ (x{:}U)\ (p{:}x = x), p = \mathit{refl\_equal}\ x$.

Streicher's axiom K

Lemma *Streicher_K* :
$\forall\ (x{:}U)\ (P{:}x = x \to \mathsf{Prop}), P\ (\mathit{refl\_equal}\ x) \to \forall\ p{:}x = x, P\ p$.

Proof-irrelevance on subsets of decidable sets

Lemma *inj_pairP2* :
$\forall\ (P{:}U \to \mathsf{Prop})\ (x{:}U)\ (p\ q{:}P\ x)$,
$\mathit{ex\_intro}\ P\ x\ p = \mathit{ex\_intro}\ P\ x\ q \to p = q$.

Injectivity of equality on dependent pairs in Type

Lemma *inj_pair2* :
$\forall\ (P{:}U \to \mathsf{Type})\ (p{:}U)\ (x\ y{:}P\ p)$,
$\mathit{existS}\ P\ p\ x = \mathit{existS}\ P\ p\ y \to x = y$.

Injectivity of equality on dependent pairs with second component in Type

Notation *inj_pairT2* := *inj_pair2*.

End *DecidableEqDepSet*.

Chapter 19

Module 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:

19.1 Definition of dependent equality and equivalence with equality of dependent pairs

Section *Dependent_Equality*.

Variable U : Type.
Variable P : $U \rightarrow$ Type.

Dependent equality

Inductive *eq_dep* $(p{:}U)$ $(x{:}P\ p)$: $\forall\ q{:}U,\ P\ q \rightarrow$ Prop :=
eq_dep_intro : *eq_dep p x p x*.
Hint *Constructors eq_dep*: *core v62*.

Lemma *eq_dep_refl* : $\forall\ (p{:}U)\ (x{:}P\ p)$, *eq_dep p x p x*.

Lemma *eq_dep_sym* :
$\forall\ (p\ q{:}U)\ (x{:}P\ p)\ (y{:}P\ q)$, *eq_dep p x q y* $\rightarrow$ *eq_dep q y p x*.
Hint Immediate *eq_dep_sym*: *core v62*.

Lemma *eq_dep_trans* :
$\forall\ (p\ q\ r{:}U)\ (x{:}P\ p)\ (y{:}P\ q)\ (z{:}P\ r)$,
eq_dep p x q y $\rightarrow$ *eq_dep q y r z* $\rightarrow$ *eq_dep p x r z*.

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

Equivalent definition of dependent equality expressed as a non dependent inductive type

Inductive *eq_dep1* $(p{:}U)$ $(x{:}P\ p)$ $(q{:}U)$ $(y{:}P\ q)$: Prop :=
eq_dep1_intro : $\forall\ h{:}q = p,\ x = eq\_rect\ q\ P\ y\ p\ h \rightarrow$ *eq_dep1 p x q y*.

Lemma *eq_dep1_dep* :
$\forall\ (p{:}U)\ (x{:}P\ p)\ (q{:}U)\ (y{:}P\ q)$, *eq_dep1 p x q y* $\rightarrow$ *eq_dep p x q y*.

Lemma *eq_dep_dep1* :
$\forall\ (p\ q{:}U)\ (x{:}P\ p)\ (y{:}P\ q)$, *eq_dep p x q y* $\rightarrow$ *eq_dep1 p x q y*.

End *Dependent_Equality*.

Implicit *Arguments eq_dep* [U P].
Implicit *Arguments eq_dep1* [U P].

Dependent equality is equivalent to equality on dependent pairs

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

Lemma *equiv_eqex_eqdep* :
$\forall\ (U{:}\text{Type})\ (P{:}U \rightarrow \text{Type})\ (p\ q{:}U)\ (x{:}P\ p)\ (y{:}P\ q)$,
existS P p x = *existS P q y* $\leftrightarrow$ *eq_dep p x q y*.

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

Lemma *eq_dep_eq_sigT* :
∀ (*U*:Type) (*P*:*U* → Type) (*p q*:*U*) (*x*:*P p*) (*y*:*P q*),
eq_dep p x q y → *existT P p x* = *existT P q y*.

Exported hints

Hint *Resolve eq_dep_intro*: *core v62*.
Hint Immediate *eq_dep_sym*: *core v62*.

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

Section *Equivalences*.

Variable *U*:Type.

Invariance by Substitution of Reflexive Equality Proofs

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

Injectivity of Dependent Equality

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

Uniqueness of Identity Proofs (UIP)

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

Uniqueness of Reflexive Identity Proofs

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

Streicher's axiom K

Definition *Streicher_K_* :=
∀ (*x*:*U*) (*P*:*x* = *x* → Prop), *P* (*refl_equal x*) → ∀ *p*:*x* = *x*, *P p*.

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

Lemma *eq_rect_eq__eq_dep1_eq* :
Eq_rect_eq → ∀ (*P*:*U*→Type) (*p*:*U*) (*x y*:*P p*), *eq_dep1 p x p y* → *x* = *y*.

Lemma *eq_rect_eq__eq_dep_eq* : *Eq_rect_eq* → *Eq_dep_eq*.

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

Lemma *eq_dep_eq__UIP* : *Eq_dep_eq* → *UIP_*.

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma *UIP__UIP_refl* : *UIP_* → *UIP_refl_*.

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

Lemma *UIP_refl__Streicher_K* : *UIP_refl_* → *Streicher_K_*.

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

Lemma *Streicher_K__eq_rect_eq* : *Streicher_K_* → *Eq_rect_eq*.

Remark: It is reasonable to think that *eq_rect_eq* is strictly stronger than *eq_rec_eq* (which is *eq_rect_eq* restricted on Set):

Definition *Eq_rec_eq* := ∀ (*P*:*U* → Set) (*p*:*U*) (*x*:*P p*) (*h*:*p* = *p*), *x* = *eq_rec p P x p h*.

Typically, *eq_rect_eq* allows to prove UIP and Streicher's K what does not seem possible with *eq_rec_eq*. In particular, the proof of *UIP* requires to use *eq_rect_eq* on fun *y* → *x*=*y* which is in Type but not in Set.

End *Equivalences*.

Section *Corollaries*.

Variable *U*:Type.

UIP implies the injectivity of equality on dependent pairs in Type

Definition *Inj_dep_pair* :=
∀ (*P*:*U* → Type) (*p*:*U*) (*x y*:*P p*), *existT P p x* = *existT P p y* → *x* = *y*.

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

End *Corollaries*.

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

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

Module Type *EqdepElimination*.

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

End *EqdepElimination*.

Module *EqdepTheory* (*M*:*EqdepElimination*).

Section *Axioms*.

Variable *U*:Type.

Invariance by Substitution of Reflexive Equality Proofs

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

Lemma *eq_rec_eq* :
∀ (*p*:*U*) (*Q*:*U* → Set) (*x*:*Q p*) (*h*:*p* = *p*), *x* = *eq_rec p Q x p h*.

Injectivity of Dependent Equality

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

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

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

Uniqueness of Reflexive Identity Proofs is a direct instance of UIP

Lemma *UIP_refl* : ∀ (*x*:*U*) (*p*:*x* = *x*), *p* = *refl_equal x*.

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

Lemma *Streicher_K* :
∀ (*x*:*U*) (*P*:*x* = *x* → Prop), *P* (*refl_equal x*) → ∀ *p*:*x* = *x*, *P p*.

End *Axioms*.

UIP implies the injectivity of equality on dependent pairs in Type

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

Notation *inj_pairT2* := *inj_pair2*.

End *EqdepTheory*.

Implicit *Arguments eq_dep* [].
Implicit *Arguments eq_dep1* [].

Chapter 20

Module 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 MÃ¼nchen, 1993.

Require Export *EqdepFacts*.

Module *Eq_rect_eq*.

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

End *Eq_rect_eq*.

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

Exported hints

Hint *Resolve eq_dep_eq*: *core v62*.
Hint *Resolve inj_pair2 inj_pairT2*: *core*.

Chapter 21

Module Coq.Logic.Hurkens

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

References:

- *Hurkens* A. J. Hurkens, "A simplification of Girard's paradox", Proceedings of the 2nd international conference Typed Lambda-Calculi and Applications (TLCA'95), 1995.

- *Geuvers* "Inconsistency of Classical Logic in Type Theory", 2001 (see www.cs.kun.nl/~herman/note.ps.gz)


```
Section Paradox.

Variable bool : Prop.
Variable p2b : Prop → bool.
Variable b2p : bool → Prop.
Hypothesis p2p1 : ∀ A:Prop, b2p (p2b A) → A.
Hypothesis p2p2 : ∀ A:Prop, A → b2p (p2b A).
Variable B : Prop.

Definition V := ∀ A:Prop, ((A → bool) → A → bool) → A → bool.
Definition U := V → bool.
Definition sb (z:V) : V := fun A r a ⇒ r (z A r) a.
Definition le (i:U → bool) (x:U) : bool :=
  x (fun A r a ⇒ i (fun v ⇒ sb v A r a)).
Definition induct (i:U → bool) : Prop :=
  ∀ x:U, b2p (le i x) → b2p (i x).
Definition WF : U := fun z ⇒ p2b (induct (z U le)).
Definition I (x:U) : Prop :=
  (∀ i:U → bool, b2p (le i x) → b2p (i (fun v ⇒ sb v U le x))) → B.

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

Lemma lemma1 : induct (fun u ⇒ p2b (I u)).

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

Theorem *paradox* : *B*.

End *Paradox*.

Chapter 22

Module 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.

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

Hint *Resolve JMeq_refl*.

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

Hint Immediate *sym_JMeq*.

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

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

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

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

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

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

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

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

JMeq is equivalent to (*eq_dep* Type [*X*]*X*)

Require Import *Eqdep*.

Lemma *JMeq_eq_dep* :

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

Lemma eq_dep_JMeq :
$\forall\ (A\ B$:Type$)\ (x{:}A)\ (y{:}B),\ eq_dep$ Type (fun $X \Rightarrow X$) $A\ x\ B\ y \rightarrow JMeq\ x\ y$.

Chapter 23

Module Coq.Logic.ProofIrrelevanceFacts

This defines the functor that build consequences of proof-irrelevance

Require Export *EqdepFacts*.

Module Type *ProofIrrelevance*.

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

End *ProofIrrelevance*.

Module *ProofIrrelevanceTheory* (*M*:*ProofIrrelevance*).

Proof-irrelevance implies uniqueness of reflexivity proofs

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

Export the theory of injective dependent elimination

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

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

We derive the irrelevance of the membership property for subsets

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

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

End *ProofIrrelevanceTheory*.

Chapter 24

Module Coq.Logic.ProofIrrelevance

This file axiomatizes proof-irrelevance and derives some consequences

Require Import *ProofIrrelevanceFacts*.

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

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

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

Chapter 25

Module 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 26

Module Coq.Bool.BoolEq

Properties of a boolean equality

Require Export *Bool*.

Section *Bool_eq_dec*.

Variable A : Set.

Variable $beq : A \to A \to bool$.

Variable $beq\_refl : \forall\ x{:}A,\ true = beq\ x\ x$.

Variable $beq\_eq : \forall\ x\ y{:}A,\ true = beq\ x\ y \to x = y$.

Definition $beq\_eq\_true : \forall\ x\ y{:}A,\ x = y \to true = beq\ x\ y$.

Definition $beq\_eq\_not\_false : \forall\ x\ y{:}A,\ x = y \to false \neq beq\ x\ y$.

Definition $beq\_false\_not\_eq : \forall\ x\ y{:}A,\ false = beq\ x\ y \to x \neq y$.

Definition $exists\_beq\_eq : \forall\ x\ y{:}A,\ \{b : bool \mid b = beq\ x\ y\}$.

Definition $not\_eq\_false\_beq : \forall\ x\ y{:}A,\ x \neq y \to false = beq\ x\ y$.

Definition $eq\_dec : \forall\ x\ y{:}A,\ \{x = y\} + \{x \neq y\}$.

End *Bool_eq_dec*.

Chapter 27

Module Coq.Bool.Bool

The type *bool* is defined in the prelude as Inductive *bool* : Set := *true* : *bool* | *false* : *bool*

Interpretation of booleans as propositions
Definition *Is_true* (*b*:*bool*) :=
  match *b* with
    | *true* ⇒ *True*
    | *false* ⇒ *False*
  end.

27.1 Decidability

Lemma *bool_dec* : ∀ *b1 b2* : *bool*, {*b1* = *b2*} + {*b1* ≠ *b2*}.

27.2 Discrimination

Lemma *diff_true_false* : *true* ≠ *false*.
Hint *Resolve diff_true_false* : *bool v62*.

Lemma *diff_false_true* : *false* ≠ *true*.
Hint *Resolve diff_false_true* : *bool v62*.
Hint *Extern* 1 (*false* ≠ *true*) ⇒ *exact diff_false_true*.

Lemma *eq_true_false_abs* : ∀ *b*:*bool*, *b* = *true* → *b* = *false* → *False*.

Lemma *not_true_is_false* : ∀ *b*:*bool*, *b* ≠ *true* → *b* = *false*.

Lemma *not_false_is_true* : ∀ *b*:*bool*, *b* ≠ *false* → *b* = *true*.

27.3 Order on booleans

Definition *leb* (*b1 b2*:*bool*) :=
  match *b1* with
    | *true* ⇒ *b2* = *true*

```
    | false ⇒ True
  end.
Hint Unfold leb: bool v62.
```

27.4 Equality

```
Definition eqb (b1 b2:bool) : bool :=
  match b1, b2 with
    | true, true ⇒ true
    | true, false ⇒ false
    | false, true ⇒ false
    | false, false ⇒ true
  end.
Lemma eqb_subst :
  ∀ (P:bool → Prop) (b1 b2:bool), eqb b1 b2 = true → P b1 → P b2.
Lemma eqb_reflx : ∀ b:bool, eqb b b = true.
Lemma eqb_prop : ∀ a b:bool, eqb a b = true → a = b.
```

27.5 Logical combinators

```
Definition ifb (b1 b2 b3:bool) : bool :=
  match b1 with
    | true ⇒ b2
    | false ⇒ b3
  end.
Definition andb (b1 b2:bool) : bool := ifb b1 b2 false.
Definition orb (b1 b2:bool) : bool := ifb b1 true b2.
Definition implb (b1 b2:bool) : bool := ifb b1 b2 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 (at level 50, left associativity) : bool_scope.
Infix "&&" := andb (at level 40, left associativity) : bool_scope.
Open Scope bool_scope.
Delimit Scope bool_scope with bool.
```

27.6 De Morgan laws

Lemma *negb_orb* : ∀ *b1 b2*:*bool*, *negb* (*b1* || *b2*) = *negb b1* && *negb b2*.

Lemma *negb_andb* : ∀ *b1 b2*:*bool*, *negb* (*b1* && *b2*) = *negb b1* || *negb b2*.

27.7 Properties of *negb*

Lemma *negb_involutive* : ∀ *b*:*bool*, *negb* (*negb b*) = *b*.

Lemma *negb_involutive_reverse* : ∀ *b*:*bool*, *b* = *negb* (*negb b*).

Notation *negb_elim* := *negb_involutive* (*only parsing*).
Notation *negb_intro* := *negb_involutive_reverse* (*only parsing*).

Lemma *negb_sym* : ∀ *b b'*:*bool*, *b'* = *negb b* → *b* = *negb b'*.

Lemma *no_fixpoint_negb* : ∀ *b*:*bool*, *negb b* ≠ *b*.

Lemma *eqb_negb1* : ∀ *b*:*bool*, *eqb* (*negb b*) *b* = *false*.

Lemma *eqb_negb2* : ∀ *b*:*bool*, *eqb b* (*negb b*) = *false*.

Lemma *if_negb* :
∀ (*A*:Set) (*b*:*bool*) (*x y*:*A*),
(if *negb b* then *x* else *y*) = (if *b* then *y* else *x*).

27.8 Properties of *orb*

Lemma *orb_true_elim* :
∀ *b1 b2*:*bool*, *b1* || *b2* = *true* → {*b1* = *true*} + {*b2* = *true*}.

Lemma *orb_prop* : ∀ *a b*:*bool*, *a* || *b* = *true* → *a* = *true* ∨ *b* = *true*.

Lemma *orb_true_intro* :
∀ *b1 b2*:*bool*, *b1* = *true* ∨ *b2* = *true* → *b1* || *b2* = *true*.
Hint *Resolve orb_true_intro*: *bool v62*.

Lemma *orb_false_intro* :
∀ *b1 b2*:*bool*, *b1* = *false* → *b2* = *false* → *b1* || *b2* = *false*.
Hint *Resolve orb_false_intro*: *bool v62*.

true is a zero for *orb*

Lemma *orb_true_r* : ∀ *b*:*bool*, *b* || *true* = *true*.
Hint *Resolve orb_true_r*: *bool v62*.

Lemma *orb_true_l* : ∀ *b*:*bool*, *true* || *b* = *true*.

Notation *orb_b_true* := *orb_true_r* (*only parsing*).
Notation *orb_true_b* := *orb_true_l* (*only parsing*).

false is neutral for *orb*

Lemma *orb_false_r* : ∀ *b*:*bool*, *b* || *false* = *b*.
Hint *Resolve orb_false_r*: *bool v62*.

Lemma *orb_false_l* : ∀ *b*:*bool*, *false* || *b* = *b*.
Hint *Resolve orb_false_l*: *bool v62*.

Notation *orb_b_false* := *orb_false_r* (*only parsing*).
Notation *orb_false_b* := *orb_false_l* (*only parsing*).

Lemma *orb_false_elim* :
∀ *b1 b2*:*bool*, *b1* || *b2* = *false* → *b1* = *false* ∧ *b2* = *false*.

Complementation

Lemma *orb_negb_r* : ∀ *b*:*bool*, *b* || *negb b* = *true*.
Hint *Resolve orb_negb_r*: *bool v62*.

Notation *orb_neg_b* := *orb_negb_r* (*only parsing*).

Commutativity

Lemma *orb_comm* : ∀ *b1 b2*:*bool*, *b1* || *b2* = *b2* || *b1*.

Associativity

Lemma *orb_assoc* : ∀ *b1 b2 b3*:*bool*, *b1* || (*b2* || *b3*) = *b1* || *b2* || *b3*.
Hint *Resolve orb_comm orb_assoc*: *bool v62*.

27.9 Properties of *andb*

Lemma *andb_prop* : ∀ *a b*:*bool*, *a* && *b* = *true* → *a* = *true* ∧ *b* = *true*.
Hint *Resolve andb_prop*: *bool v62*.

Lemma *andb_true_eq* :
∀ *a b*:*bool*, *true* = *a* && *b* → *true* = *a* ∧ *true* = *b*.

Lemma *andb_true_intro* :
∀ *b1 b2*:*bool*, *b1* = *true* ∧ *b2* = *true* → *b1* && *b2* = *true*.
Hint *Resolve andb_true_intro*: *bool v62*.

Lemma *andb_false_intro1* : ∀ *b1 b2*:*bool*, *b1* = *false* → *b1* && *b2* = *false*.

Lemma *andb_false_intro2* : ∀ *b1 b2*:*bool*, *b2* = *false* → *b1* && *b2* = *false*.

false is a zero for *andb*

Lemma *andb_false_r* : ∀ *b*:*bool*, *b* && *false* = *false*.

Lemma *andb_false_l* : ∀ *b*:*bool*, *false* && *b* = *false*.

Notation *andb_b_false* := *andb_false_r* (*only parsing*).
Notation *andb_false_b* := *andb_false_l* (*only parsing*).

true is neutral for *andb*

Lemma *andb_true_r* : ∀ *b*:*bool*, *b* && *true* = *b*.

Lemma *andb_true_l* : ∀ *b*:*bool*, *true* && *b* = *b*.

Notation *andb_b_true* := *andb_true_r* (*only parsing*).
Notation *andb_true_b* := *andb_true_l* (*only parsing*).

Lemma *andb_false_elim* :
∀ *b1 b2*:*bool*, *b1* && *b2* = *false* → {*b1* = *false*} + {*b2* = *false*}.
Hint *Resolve andb_false_elim*: *bool v62*.

Complementation

Lemma *andb_negb_r* : ∀ *b*:*bool*, *b* && *negb b* = *false*.
Hint *Resolve andb_negb_r*: *bool v62*.

Notation *andb_neg_b* := *andb_negb_r* (*only parsing*).

Commutativity

Lemma *andb_comm* : ∀ *b1 b2*:*bool*, *b1* && *b2* = *b2* && *b1*.

Associativity

Lemma *andb_assoc* : ∀ *b1 b2 b3*:*bool*, *b1* && (*b2* && *b3*) = *b1* && *b2* && *b3*.

Hint *Resolve andb_comm andb_assoc*: *bool v62*.

27.10 Properties mixing *andb* and *orb*

Distributivity

Lemma *andb_orb_distrib_r* :
∀ *b1 b2 b3*:*bool*, *b1* && (*b2* || *b3*) = *b1* && *b2* || *b1* && *b3*.

Lemma *andb_orb_distrib_l* :
∀ *b1 b2 b3*:*bool*, (*b1* || *b2*) && *b3* = *b1* && *b3* || *b2* && *b3*.

Lemma *orb_andb_distrib_r* :
∀ *b1 b2 b3*:*bool*, *b1* || *b2* && *b3* = (*b1* || *b2*) && (*b1* || *b3*).

Lemma *orb_andb_distrib_l* :
∀ *b1 b2 b3*:*bool*, *b1* && *b2* || *b3* = (*b1* || *b3*) && (*b2* || *b3*).

Notation *demorgan1* := *andb_orb_distrib_r* (*only parsing*).
Notation *demorgan2* := *andb_orb_distrib_l* (*only parsing*).
Notation *demorgan3* := *orb_andb_distrib_r* (*only parsing*).
Notation *demorgan4* := *orb_andb_distrib_l* (*only parsing*).

Absorption

Lemma *absoption_andb* : ∀ *b1 b2*:*bool*, *b1* && (*b1* || *b2*) = *b1*.

Lemma *absoption_orb* : ∀ *b1 b2*:*bool*, *b1* || *b1* && *b2* = *b1*.

27.11 Properties of *xorb*

false is neutral for *xorb*

Lemma *xorb_false_r* : ∀ *b*:*bool*, *xorb b false* = *b*.

Lemma *xorb_false_l* : ∀ *b*:*bool*, *xorb false b* = *b*.

Notation *xorb_false* := *xorb_false_r* (*only parsing*).
Notation *false_xorb* := *xorb_false_l* (*only parsing*).

true is "complementing" for *xorb*

Lemma *xorb_true_r* : ∀ *b*:*bool*, *xorb b true* = *negb b*.

Lemma *xorb_true_l* : ∀ *b*:*bool*, *xorb true b* = *negb b*.

Notation *xorb_true* := *xorb_true_r* (*only parsing*).
Notation *true_xorb* := *xorb_true_l* (*only parsing*).

Nilpotency (alternatively: identity is a inverse for *xorb*)

Lemma *xorb_nilpotent* : ∀ *b*:*bool*, *xorb b b* = *false*.

Commutativity

Lemma *xorb_comm* : ∀ *b b'*:*bool*, *xorb b b'* = *xorb b' b*.

Associativity

Lemma *xorb_assoc_reverse* :
∀ *b b' b"*:*bool*, *xorb* (*xorb b b'*) *b"* = *xorb b* (*xorb b' b"*).

Notation *xorb_assoc* := *xorb_assoc_reverse* (*only parsing*).
Lemma *xorb_eq* : ∀ *b b'*:*bool*, *xorb b b'* = *false* → *b* = *b'*.

Lemma *xorb_move_l_r_1* :
∀ *b b' b"*:*bool*, *xorb b b'* = *b"* → *b'* = *xorb b b"*.

Lemma *xorb_move_l_r_2* :
∀ *b b' b"*:*bool*, *xorb b b'* = *b"* → *b* = *xorb b" b'*.

Lemma *xorb_move_r_l_1* :
∀ *b b' b"*:*bool*, *b* = *xorb b' b"* → *xorb b' b* = *b"*.

Lemma *xorb_move_r_l_2* :
∀ *b b' b"*:*bool*, *b* = *xorb b' b"* → *xorb b b"* = *b'*.

Lemmas about the *b* = *true* embedding of *bool* to Prop

Lemma *eq_true_iff_eq* : ∀ *b1 b2*, (*b1* = *true* ↔ *b2* = *true*) → *b1* = *b2*.

Notation *bool_1* := *eq_true_iff_eq* (*only parsing*).
Lemma *eq_true_negb_classical* : ∀ *b*:*bool*, *negb b* ≠ *true* → *b* = *true*.

Notation *bool_3* := *eq_true_negb_classical* (*only parsing*).
Lemma *eq_true_not_negb* : ∀ *b*:*bool*, *b* ≠ *true* → *negb b* = *true*.

Notation *bool_6* := *eq_true_not_negb* (*only parsing*).

Hint *Resolve eq_true_not_negb : bool.*

Lemma *absurd_eq_bool* : ∀ *b b'*:*bool*, *False* → *b* = *b'*.

Lemma *absurd_eq_true* : ∀ *b*, *False* → *b* = *true*.
Hint *Resolve absurd_eq_true.*

Lemma *trans_eq_bool* : ∀ *x y z*:*bool*, *x* = *y* → *y* = *z* → *x* = *z*.
Hint *Resolve trans_eq_bool.*

27.12 Reflection of *bool* into Prop

Is_true and equality

Hint *Unfold Is_true*: *bool.*

Lemma *Is_true_eq_true* : ∀ *x*:*bool*, *Is_true x* → *x* = *true*.

Lemma *Is_true_eq_left* : ∀ *x*:*bool*, *x* = *true* → *Is_true x*.

Lemma *Is_true_eq_right* : ∀ *x*:*bool*, *true* = *x* → *Is_true x*.

Notation *Is_true_eq_true2* := *Is_true_eq_right* (*only parsing*).

Hint Immediate *Is_true_eq_right Is_true_eq_left*: *bool.*

Lemma *eqb_refl* : ∀ *x*:*bool*, *Is_true* (*eqb x x*).

Lemma *eqb_eq* : ∀ *x y*:*bool*, *Is_true* (*eqb x y*) → *x* = *y*.

Is_true and connectives

Lemma *orb_prop_elim* :
∀ *a b*:*bool*, *Is_true* (*a* || *b*) → *Is_true a* ∨ *Is_true b*.

Notation *orb_prop2* := *orb_prop_elim* (*only parsing*).

Lemma *orb_prop_intro* :
∀ *a b*:*bool*, *Is_true a* ∨ *Is_true b* → *Is_true* (*a* || *b*).

Lemma *andb_prop_intro* :
∀ *b1 b2*:*bool*, *Is_true b1* ∧ *Is_true b2* → *Is_true* (*b1* && *b2*).
Hint *Resolve andb_prop_intro*: *bool v62.*

Notation *andb_true_intro2* :=
(fun *b1 b2 H1 H2* ⇒ *andb_prop_intro b1 b2* (*conj H1 H2*))
(*only parsing*).

Lemma *andb_prop_elim* :
∀ *a b*:*bool*, *Is_true* (*a* && *b*) → *Is_true a* ∧ *Is_true b*.
Hint *Resolve andb_prop_elim*: *bool v62.*

Notation *andb_prop2* := *andb_prop_elim* (*only parsing*).

Lemma *eq_bool_prop_intro* :
∀ *b1 b2*, (*Is_true b1* ↔ *Is_true b2*) → *b1* = *b2*.

Lemma $eq\_bool\_prop\_elim : \forall\ b1\ b2,\ b1 = b2 \rightarrow (Is\_true\ b1 \leftrightarrow Is\_true\ b2)$.

Lemma $negb\_prop\_elim : \forall\ b,\ Is\_true\ (negb\ b) \rightarrow \neg\ Is\_true\ b$.

Lemma $negb\_prop\_intro : \forall\ b,\ \neg\ Is\_true\ b \rightarrow Is\_true\ (negb\ b)$.

Lemma $negb\_prop\_classical : \forall\ b,\ \neg\ Is\_true\ (negb\ b) \rightarrow Is\_true\ b$.

Lemma $negb\_prop\_involutive : \forall\ b,\ Is\_true\ b \rightarrow \neg\ Is\_true\ (negb\ b)$.

Chapter 28

Module Coq.Bool.Bvector

Bit vectors. Contribution by Jean Duprat (ENS Lyon).

Require Export *Bool*.
Require Export *Sumbool*.
Require Import *Arith*.

Open Local *Scope nat_scope*.

On s'inspire de List.v pour fabriquer les vecteurs de bits. La dimension du vecteur est un paramètre trop important pour se contenter de la fonction "length". La première idée est de faire un record avec la liste et la longueur. Malheureusement, cette verification a posteriori amene a faire de nombreux lemmes pour gerer les longueurs. La seconde idée est de faire un type dépendant dans lequel la longueur est un paramètre de construction. Cela complique un peu les inductions structurelles, la solution qui a ma préférence est alors d'utiliser un terme de preuve comme définition, car le mécanisme d'inférence du type du filtrage n'est pas aussi puissant que celui implanté par les tactiques d'élimination.

Section *VECTORS*.

Un vecteur est une liste de taille n d'éléments d'un ensemble A. Si la taille est non nulle, on peut extraire la première composante et le reste du vecteur, la dernière composante ou rajouter ou enlever une composante (carry) ou repeter la dernière composante en fin de vecteur. On peut aussi tronquer le vecteur de ses p dernières composantes ou au contraire l'étendre (concaténer) d'un vecteur de longueur p. Une fonction unaire sur A génère une fonction des vecteurs de taille n dans les vecteurs de taille n en appliquant f terme à terme. Une fonction binaire sur A génère une fonction des couples de vecteurs de taille n dans les vecteurs de taille n en appliquant f terme à terme.

Variable A : Type.

Inductive $vector : nat \rightarrow$ Type :=
  | $Vnil : vector\ 0$
  | $Vcons : \forall\ (a{:}A)\ (n{:}nat),\ vector\ n \rightarrow vector\ (S\ n)$.

Definition $Vhead : \forall\ n{:}nat,\ vector\ (S\ n) \rightarrow A$.

Definition $Vtail : \forall\ n{:}nat,\ vector\ (S\ n) \rightarrow vector\ n$.

Definition $Vlast : \forall\ n{:}nat,\ vector\ (S\ n) \rightarrow A$.

Definition $Vconst : \forall\ (a{:}A)\ (n{:}nat),\ vector\ n.$

Lemma $Vshiftout : \forall\ n{:}nat,\ vector\ (S\ n) \rightarrow vector\ n.$

Lemma $Vshiftin : \forall\ n{:}nat,\ A \rightarrow vector\ n \rightarrow vector\ (S\ n).$

Lemma $Vshiftrepeat : \forall\ n{:}nat,\ vector\ (S\ n) \rightarrow vector\ (S\ (S\ n)).$

Lemma $Vtrunc : \forall\ n\ p{:}nat,\ n > p \rightarrow vector\ n \rightarrow vector\ (n\ -\ p).$

Lemma $Vextend : \forall\ n\ p{:}nat,\ vector\ n \rightarrow vector\ p \rightarrow vector\ (n\ +\ p).$

Variable $f : A \rightarrow A.$

Lemma $Vunary : \forall\ n{:}nat,\ vector\ n \rightarrow vector\ n.$

Variable $g : A \rightarrow A \rightarrow A.$

Lemma $Vbinary : \forall\ n{:}nat,\ vector\ n \rightarrow vector\ n \rightarrow vector\ n.$

Definition $Vid : \forall\ n{:}nat,\ vector\ n \rightarrow vector\ n.$

Lemma $Vid\_eq : \forall\ (n{:}nat)\ (v{:}vector\ n),\ v{=}(Vid\ n\ v).$

Lemma $VSn\_eq :$
$\forall\ (n : nat)\ (v : vector\ (S\ n)),\ v = Vcons\ (Vhead\ \_\ v)\ \_\ (Vtail\ \_\ v).$

Lemma $V0\_eq : \forall\ (v : vector\ 0),\ v = Vnil.$

End $VECTORS.$

Section $BOOLEAN\_VECTORS.$

Un vecteur de bits est un vecteur sur l'ensemble des booléens de longueur fixe. ATTENTION : le stockage s'effectue poids FAIBLE en tête. On en extrait le bit de poids faible (head) et la fin du vecteur (tail). On calcule la négation d'un vecteur, le et, le ou et le xor bit à bit de 2 vecteurs. On calcule les décalages d'une position vers la gauche (vers les poids forts, on utilise donc Vshiftout, vers la droite (vers les poids faibles, on utilise Vshiftin) en insérant un bit 'carry' (logique) ou en répétant le bit de poids fort (arithmétique). ATTENTION : Tous les décalages prennent la taille moins un comme paramètre (ils ne travaillent que sur des vecteurs au moins de longueur un).

Definition $Bvector := vector\ bool.$

Definition $Bnil := @Vnil\ bool.$

Definition $Bcons := @Vcons\ bool.$

Definition $Bvect\_true := Vconst\ bool\ true.$

Definition $Bvect\_false := Vconst\ bool\ false.$

Definition $Blow := Vhead\ bool.$

Definition $Bhigh := Vtail\ bool.$

Definition $Bsign := Vlast\ bool.$

```
Definition Bneg := Vunary bool negb.

Definition BVand := Vbinary bool andb.

Definition BVor := Vbinary bool orb.

Definition BVxor := Vbinary bool xorb.

Definition BshiftL (n:nat) (bv:Bvector (S n)) (carry:bool) :=
  Bcons carry n (Vshiftout bool n bv).

Definition BshiftRl (n:nat) (bv:Bvector (S n)) (carry:bool) :=
  Bhigh (S n) (Vshiftin bool (S n) carry bv).

Definition BshiftRa (n:nat) (bv:Bvector (S n)) :=
  Bhigh (S n) (Vshiftrepeat bool n bv).

Fixpoint BshiftL_iter (n:nat) (bv:Bvector (S n)) (p:nat) {struct p} :
  Bvector (S n) :=
  match p with
    | O ⇒ bv
    | S p' ⇒ BshiftL n (BshiftL_iter n bv p') false
  end.

Fixpoint BshiftRl_iter (n:nat) (bv:Bvector (S n)) (p:nat) {struct p} :
  Bvector (S n) :=
  match p with
    | O ⇒ bv
    | S p' ⇒ BshiftRl n (BshiftRl_iter n bv p') false
  end.

Fixpoint BshiftRa_iter (n:nat) (bv:Bvector (S n)) (p:nat) {struct p} :
  Bvector (S n) :=
  match p with
    | O ⇒ bv
    | S p' ⇒ BshiftRa n (BshiftRa_iter n bv p')
  end.

End BOOLEAN_VECTORS.
```

Chapter 29

Module Coq.Bool.DecBool

Definition $ifdec$ $(A\ B$:Prop$)$ $(C$:Type$)$ $(H$:$\{A\} + \{B\})$ $(x\ y$:$C)$: C :=
  if H then x else y.

Theorem $ifdec\_left$:
  $\forall\ (A\ B$:Prop$)$ $(C$:Set$)$ $(H$:$\{A\} + \{B\})$,
    $\neg\ B \rightarrow \forall\ x\ y$:$C$, $ifdec\ H\ x\ y = x$.

Theorem $ifdec\_right$:
  $\forall\ (A\ B$:Prop$)$ $(C$:Set$)$ $(H$:$\{A\} + \{B\})$,
    $\neg\ A \rightarrow \forall\ x\ y$:$C$, $ifdec\ H\ x\ y = y$.

Chapter 30

Module Coq.Bool.IfProp

Require Import *Bool*.

Inductive *IfProp* (*A B*:Prop) : *bool* → Prop :=
  | *Iftrue* : *A* → *IfProp A B true*
  | *Iffalse* : *B* → *IfProp A B false*.

Hint *Resolve Iftrue Iffalse*: *bool v62*.

Lemma *Iftrue_inv* : ∀ (*A B*:Prop) (*b*:*bool*), *IfProp A B b* → *b* = *true* → *A*.

Lemma *Iffalse_inv* :
∀ (*A B*:Prop) (*b*:*bool*), *IfProp A B b* → *b* = *false* → *B*.

Lemma *IfProp_true* : ∀ *A B*:Prop, *IfProp A B true* → *A*.

Lemma *IfProp_false* : ∀ *A B*:Prop, *IfProp A B false* → *B*.

Lemma *IfProp_or* : ∀ (*A B*:Prop) (*b*:*bool*), *IfProp A B b* → *A* ∨ *B*.

Lemma *IfProp_sum* : ∀ (*A B*:Prop) (*b*:*bool*), *IfProp A B b* → {*A*} + {*B*}.

Chapter 31

Module Coq.Bool.Sumbool

Here are collected some results about the type sumbool (see INIT/Specif.v) *sumbool A B*, which is written $\{A\}+\{B\}$, is the informative disjunction "A or B", where A and B are logical propositions. Its extraction is isomorphic to the type of booleans.

A boolean is either *true* or *false*, and this is decidable

Definition *sumbool_of_bool* : $\forall$ *b*:*bool*, $\{b = true\} + \{b = false\}$.

Hint *Resolve sumbool_of_bool*: *bool*.

Definition *bool_eq_rec* :
$\forall$ (*b*:*bool*) (*P*:*bool* $\rightarrow$ Set),
$(b = true \rightarrow P\ true) \rightarrow (b = false \rightarrow P\ false) \rightarrow P\ b$.

Definition *bool_eq_ind* :
$\forall$ (*b*:*bool*) (*P*:*bool* $\rightarrow$ Prop),
$(b = true \rightarrow P\ true) \rightarrow (b = false \rightarrow P\ false) \rightarrow P\ b$.

Logic connectives on type *sumbool*

Section *connectives*.

Variables $A\ B\ C\ D$: Prop.

Hypothesis *H1* : $\{A\} + \{B\}$.
Hypothesis *H2* : $\{C\} + \{D\}$.

Definition *sumbool_and* : $\{A \wedge C\} + \{B \vee D\}$.

Definition *sumbool_or* : $\{A \vee C\} + \{B \wedge D\}$.

Definition *sumbool_not* : $\{B\} + \{A\}$.

End *connectives*.

Hint *Resolve sumbool_and sumbool_or*: *core*.
Hint Immediate *sumbool_not* : *core*.

Any decidability function in type *sumbool* can be turned into a function returning a boolean with the corresponding specification:

Definition *bool_of_sumbool* :

$\forall\ A\ B$:Prop, $\{A\} + \{B\} \rightarrow \{b : bool \mid \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ B\}$.
Implicit *Arguments bool_of_sumbool*.

Chapter 32

Module Coq.Bool.Zerob

Require Import *Arith*.
Require Import *Bool*.

Open Local *Scope nat_scope*.

Definition *zerob* (*n*:*nat*) : *bool* :=
  match *n* with
    | *O* ⇒ *true*
    | *S* _ ⇒ *false*
  end.

Lemma *zerob_true_intro* : ∀ *n*:*nat*, *n* = 0 → *zerob n* = *true*.
Hint *Resolve zerob_true_intro*: *bool*.

Lemma *zerob_true_elim* : ∀ *n*:*nat*, *zerob n* = *true* → *n* = 0.

Lemma *zerob_false_intro* : ∀ *n*:*nat*, *n* ≠ 0 → *zerob n* = *false*.
Hint *Resolve zerob_false_intro*: *bool*.

Lemma *zerob_false_elim* : ∀ *n*:*nat*, *zerob n* = *false* → *n* ≠ 0.

Chapter 33

Module Coq.Arith.Arith_base

Require Export *Le*.
Require Export *Lt*.
Require Export *Plus*.
Require Export *Gt*.
Require Export *Minus*.
Require Export *Mult*.
Require Export *Between*.
Require Export *Peano_dec*.
Require Export *Compare_dec*.
Require Export *Factorial*.

Chapter 34

Module Coq.Arith.Arith

Require Export *Arith_base*.
Require Export *ArithRing*.

Chapter 35

Module Coq.Arith.Between

Require Import *Le*.
Require Import *Lt*.

Open Local *Scope nat_scope*.

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

Section *Between*.
Variables *P Q* : *nat* → Prop.

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

Hint *Constructors between*: *arith v62*.

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

Hint *Resolve bet_eq*: *arith v62*.

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

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

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

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

Hint *Constructors exists_between*: *arith v62*.

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

Lemma *exists_lt* : ∀ *k l*, *exists_between k l* → *k* < *l*.
Hint Immediate *exists_le_S exists_lt*: *arith v62*.

Lemma *exists_S_le* : ∀ *k l*, *exists_between k* (*S l*) → *k* ≤ *l*.

Hint Immediate *exists_S_le*: *arith v62*.

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

Lemma *in_int_intro* : $\forall\ p\ q\ r, p \leq r \rightarrow r < q \rightarrow$ *in_int p q r*.
Hint *Resolve in_int_intro*: *arith v62*.

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

Lemma *in_int_p_Sq* :
$\forall\ p\ q\ r$, *in_int p* (*S q*) *r* $\rightarrow$ *in_int p q r* $\vee\ r = q$:>*nat*.

Lemma *in_int_S* : $\forall\ p\ q\ r$, *in_int p q r* $\rightarrow$ *in_int p* (*S q*) *r*.
Hint *Resolve in_int_S*: *arith v62*.

Lemma *in_int_Sp_q* : $\forall\ p\ q\ r$, *in_int* (*S p*) *q r* $\rightarrow$ *in_int p q r*.
Hint Immediate *in_int_Sp_q*: *arith v62*.

Lemma *between_in_int* :
$\forall\ k\ l$, *between k l* $\rightarrow \forall\ r$, *in_int k l r* $\rightarrow P\ r$.

Lemma *in_int_between* :
$\forall\ k\ l, k \leq l \rightarrow (\forall\ r,$ *in_int k l r* $\rightarrow P\ r) \rightarrow$ *between k l*.

Lemma *exists_in_int* :
$\forall\ k\ l$, *exists_between k l* $\rightarrow$ *exists2 m* : *nat*, *in_int k l m* & *Q m*.

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

Lemma *between_or_exists* :
$\forall\ k\ l$,
$k \leq l \rightarrow$
$(\forall\ n$:*nat*, *in_int k l n* $\rightarrow P\ n \vee Q\ n) \rightarrow$
between k l $\vee$ *exists_between k l*.

Lemma *between_not_exists* :
$\forall\ k\ l$,
between k l $\rightarrow$
$(\forall\ n$:*nat*, *in_int k l n* $\rightarrow P\ n \rightarrow \neg\ Q\ n) \rightarrow \neg$ *exists_between k l*.

Inductive *P_nth* (*init*:*nat*) : *nat* $\rightarrow$ *nat* $\rightarrow$ Prop :=
| *nth_O* : *P_nth init init* 0
| *nth_S* :
$\forall\ k\ l\ (n$:*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*:*nat*) *l* (*n*:*nat*), *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 v62*.
Hint Immediate *in_int_Sp_q exists_le_S exists_S_le*: *arith v62*.

Chapter 36

Module Coq.Arith.Bool_nat

Require Export *Compare_dec*.
Require Export *Peano_dec*.
Require Import *Sumbool*.

Open Local *Scope nat_scope*.

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

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

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

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

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

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

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

Chapter 37

Module Coq.Arith.Compare_dec

Require Import *Le*.
Require Import *Lt*.
Require Import *Gt*.
Require Import *Decidable*.

Open Local *Scope nat_scope*.

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

Definition *zerop* n : $\{n = 0\} + \{0 < n\}$.

Definition *lt_eq_lt_dec* n m : $\{n < m\} + \{n = m\} + \{m < n\}$.

Definition *gt_eq_gt_dec* n m : $\{m > n\} + \{n = m\} + \{n > m\}$.

Definition *le_lt_dec* n m : $\{n \leq m\} + \{m < n\}$.

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

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

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

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

Proofs of decidability

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

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

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

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

Theorem *not_eq* : $\forall\ n\ m, n \neq m \to n < m \vee m < n$.

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

Theorem *not_gt* : $\forall\ n\ m, \neg\ n > m \to n \leq m$.

Theorem *not_ge* : $\forall\ n\ m, \neg\ n \geq m \to n < m$.

Theorem *not_lt* : $\forall\ n\ m, \neg\ n < m \to n \geq m$.

A ternary comparison function in the spirit of *Zcompare*.

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

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

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

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

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

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

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

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

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

Lemma $leb\_correct : \forall\ m\ n{:}nat,\ m \leq n \rightarrow leb\ m\ n = true.$

Lemma $leb\_complete : \forall\ m\ n{:}nat,\ leb\ m\ n = true \rightarrow m \leq n.$

Lemma $leb\_correct\_conv : \forall\ m\ n{:}nat,\ m < n \rightarrow leb\ n\ m = false.$

Lemma $leb\_complete\_conv : \forall\ m\ n{:}nat,\ leb\ n\ m = false \rightarrow m < n.$

Lemma $leb\_compare : \forall\ n\ m,\ leb\ n\ m = true \leftrightarrow nat\_compare\ n\ m \neq Gt.$

Chapter 38

Module Coq.Arith.Compare

Equality is decidable on *nat*

Open Local *Scope nat_scope.*

Notation *not_eq_sym* := *sym_not_eq.*

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* : $\forall\ n\ m, n \leq m \rightarrow lt\_or\_eq\ n\ m$.

Lemma *le_le_S_eq* : $\forall\ n\ m, n \leq m \rightarrow S\ n \leq m \vee n = m$.

Lemma *discrete_nat* :
$\forall\ n\ m, n < m \rightarrow S\ n = m \vee (\exists\ r : nat, m = S\ (S\ (n + r)))$.

Require Export *Wf_nat.*

Require Export *Min.*

Chapter 39

Module Coq.Arith.Div2

Require Import *Lt*.
Require Import *Plus*.
Require Import *Compare_dec*.
Require Import *Even*.

Open Local *Scope nat_scope*.

Implicit Type n : *nat*.

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

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

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

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

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

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

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

Properties related to the parity

```
Lemma even_odd_div2 :
  ∀ n,
    (even n ↔ div2 n = div2 (S n)) ∧ (odd n ↔ S (div2 n) = div2 (S n)).
```

Specializations

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

Lemma $div2\_even : \forall\ n,\ div2\ n = div2\ (S\ n) \rightarrow even\ n.$

Lemma $odd\_div2 : \forall\ n,\ odd\ n \rightarrow S\ (div2\ n) = div2\ (S\ n).$

Lemma $div2\_odd : \forall\ n,\ S\ (div2\ n) = div2\ (S\ n) \rightarrow odd\ n.$

Hint *Resolve even_div2 div2_even odd_div2 div2_odd: arith.*

Properties related to the double ($2n$)

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

Hint *Unfold double: arith.*

Lemma $double\_S : \forall\ n,\ double\ (S\ n) = S\ (S\ (double\ n)).$

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

Hint *Resolve double_S: arith.*

Lemma *even_odd_double* :
$\forall\ n,$
$(even\ n \leftrightarrow n = double\ (div2\ n)) \wedge (odd\ n \leftrightarrow n = S\ (double\ (div2\ n))).$

Specializations

Lemma $even\_double : \forall\ n,\ even\ n \rightarrow n = double\ (div2\ n).$

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

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

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

Hint *Resolve even_double double_even odd_double double_odd: arith.*

Application:

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

(Immediate: it is $n/2$)

Lemma $even\_2n : \forall\ n,\ even\ n \rightarrow \{p : nat \mid n = double\ p\}.$

Lemma $odd\_S2n : \forall\ n,\ odd\ n \rightarrow \{p : nat \mid n = S\ (double\ p)\}.$

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

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

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

Chapter 40

Module Coq.Arith.Div

Euclidean division

Open Local *Scope nat_scope.*

Require *Le.*
Require *Euclid_def.*
Require *Compare_dec.*

Implicit Variables Type *n,a,b,q,r:nat.*

```
Fixpoint inf_dec [n:nat] : nat→bool :=
  [m:nat] Cases n m of
              O _ ⇒ true
              | (S n') O ⇒ false
              | (S n') (S m') ⇒ (inf_dec n' m')
            end.
```

Theorem *div1* : *(b:nat)(gt b O)->(a:nat)(diveucl a b).*

Theorem *div2* : *(b:nat)(gt b O)->(a:nat)(diveucl a b).*

Chapter 41

Module Coq.Arith.EqNat

Equality on natural numbers

Open Local *Scope nat_scope.*

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

41.1 Propositional equality

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

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

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

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

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

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

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

Theorem *eq_nat_decide* : ∀ *n m*, {*eq_nat n m*} + {~ *eq_nat n m*}.

41.2 Boolean equality on *nat*

```
Fixpoint beq_nat n m {struct n} : bool :=
  match n, m with
    | O, O ⇒ true
    | O, S _ ⇒ false
    | S _, O ⇒ false
    | S n1, S m1 ⇒ beq_nat n1 m1
  end.

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

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

Chapter 42

Module Coq.Arith.Euclid

Require Import *Mult*.
Require Import *Compare_dec*.
Require Import *Wf_nat*.

Open Local *Scope nat_scope*.

Implicit *Types* $a\ b\ n\ q\ r : nat$.

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

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

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

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

Chapter 43

Module Coq.Arith.Even

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

Open Local *Scope nat_scope.*

Implicit *Types m n* : *nat.*

43.1 Definition of *even* and *odd*, and basic facts

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

Hint *Constructors even*: *arith.*
Hint *Constructors odd*: *arith.*

Lemma *even_or_odd* : ∀ *n*, *even n* ∨ *odd n.*

Lemma *even_odd_dec* : ∀ *n*, {*even n*} + {*odd n*}.

Lemma *not_even_and_odd* : ∀ *n*, *even n* → *odd n* → *False.*

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

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_even_plus* : ∀ *n m*, *even n* → *even m* → *even* (*n* + *m*).

Lemma *odd_even_plus* : ∀ *n m*, *odd n* → *odd m* → *even* (*n* + *m*).

Lemma $even\_plus\_even\_inv\_r : \forall\ n\ m,\ even\ (n + m) \to even\ n \to even\ m.$

Lemma $even\_plus\_even\_inv\_l : \forall\ n\ m,\ even\ (n + m) \to even\ m \to even\ n.$

Lemma $even\_plus\_odd\_inv\_r : \forall\ n\ m,\ even\ (n + m) \to odd\ n \to odd\ m.$

Lemma $even\_plus\_odd\_inv\_l : \forall\ n\ m,\ even\ (n + m) \to odd\ m \to odd\ n.$
Hint *Resolve even_even_plus odd_even_plus*: *arith*.

Lemma $odd\_plus\_l : \forall\ n\ m,\ odd\ n \to even\ m \to odd\ (n + m).$

Lemma $odd\_plus\_r : \forall\ n\ m,\ even\ n \to odd\ m \to odd\ (n + m).$

Lemma $odd\_plus\_even\_inv\_l : \forall\ n\ m,\ odd\ (n + m) \to odd\ m \to even\ n.$

Lemma $odd\_plus\_even\_inv\_r : \forall\ n\ m,\ odd\ (n + m) \to odd\ n \to even\ m.$

Lemma $odd\_plus\_odd\_inv\_l : \forall\ n\ m,\ odd\ (n + m) \to even\ m \to odd\ n.$

Lemma $odd\_plus\_odd\_inv\_r : \forall\ n\ m,\ odd\ (n + m) \to even\ n \to odd\ m.$
Hint *Resolve odd_plus_l odd_plus_r*: *arith*.

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

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

Lemma $even\_mult\_l : \forall\ n\ m,\ even\ n \to even\ (n \times m).$

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

Lemma $even\_mult\_inv\_r : \forall\ n\ m,\ even\ (n \times m) \to odd\ n \to even\ m.$

Lemma $even\_mult\_inv\_l : \forall\ n\ m,\ even\ (n \times m) \to odd\ m \to even\ n.$

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

Lemma $odd\_mult\_inv\_l : \forall\ n\ m,\ odd\ (n \times m) \to odd\ n.$

Lemma $odd\_mult\_inv\_r : \forall\ n\ m,\ odd\ (n \times m) \to odd\ m.$

Chapter 44

Module Coq.Arith.Factorial

Require Import *Plus*.
Require Import *Mult*.
Require Import *Lt*.
Open Local *Scope nat_scope*.

Factorial

Lemma *lt_O_fact* : $\forall$ *n*:*nat*, $0 < fact\ n$.

Lemma *fact_neq_0* : $\forall$ *n*:*nat*, $fact\ n \neq 0$.

Lemma *fact_le* : $\forall\ n\ m$:*nat*, $n \leq m \rightarrow fact\ n \leq fact\ m$.

Chapter 45

Module Coq.Arith.Gt

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

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

Require Import *Le*.
Require Import *Lt*.
Require Import *Plus*.
Open Local *Scope nat_scope*.

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

45.1 Order and successor

Theorem *gt_Sn_O* : $\forall$ n, S $n > 0$.
Hint *Resolve gt_Sn_O*: *arith v62*.

Theorem *gt_Sn_n* : $\forall$ n, S $n > n$.
Hint *Resolve gt_Sn_n*: *arith v62*.

Theorem *gt_n_S* : $\forall$ n m, $n > m \rightarrow S$ $n > S$ m.
Hint *Resolve gt_n_S*: *arith v62*.

Lemma *gt_S_n* : $\forall$ n m, S $m > S$ $n \rightarrow m > n$.
Hint Immediate *gt_S_n*: *arith v62*.

Theorem *gt_S* : $\forall$ n m, S $n > m \rightarrow n > m \vee m = n$.

Lemma *gt_pred* : $\forall$ n m, $m > S$ $n \rightarrow$ *pred* $m > n$.
Hint Immediate *gt_pred*: *arith v62*.

45.2 Irreflexivity

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

45.3 Asymmetry

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

Hint *Resolve gt_asym: arith v62.*

45.4 Relating strict and large orders

Lemma $le\_not\_gt : \forall\ n\ m,\ n \leq m \rightarrow \neg\ n > m.$
Hint *Resolve le_not_gt: arith v62.*

Lemma $gt\_not\_le : \forall\ n\ m,\ n > m \rightarrow \neg\ n \leq m.$

Hint *Resolve gt_not_le: arith v62.*

Theorem $le\_S\_gt : \forall\ n\ m,\ S\ n \leq m \rightarrow m > n.$
Hint Immediate *le_S_gt: arith v62.*

Lemma $gt\_S\_le : \forall\ n\ m,\ S\ m > n \rightarrow n \leq m.$
Hint Immediate *gt_S_le: arith v62.*

Lemma $gt\_le\_S : \forall\ n\ m,\ m > n \rightarrow S\ n \leq m.$
Hint *Resolve gt_le_S: arith v62.*

Lemma $le\_gt\_S : \forall\ n\ m,\ n \leq m \rightarrow S\ m > n.$
Hint *Resolve le_gt_S: arith v62.*

45.5 Transitivity

Theorem $le\_gt\_trans : \forall\ n\ m\ p,\ m \leq n \rightarrow m > p \rightarrow n > p.$

Theorem $gt\_le\_trans : \forall\ n\ m\ p,\ n > m \rightarrow p \leq m \rightarrow n > p.$

Lemma $gt\_trans : \forall\ n\ m\ p,\ n > m \rightarrow m > p \rightarrow n > p.$

Theorem $gt\_trans\_S : \forall\ n\ m\ p,\ S\ n > m \rightarrow m > p \rightarrow n > p.$

Hint *Resolve gt_trans_S le_gt_trans gt_le_trans: arith v62.*

45.6 Comparison to 0

Theorem $gt\_O\_eq : \forall\ n,\ n > 0 \vee 0 = n.$

45.7 Simplification and compatibility

Lemma $plus\_gt\_reg\_l : \forall\ n\ m\ p,\ p + n > p + m \rightarrow n > m.$

Lemma $plus\_gt\_compat\_l : \forall\ n\ m\ p,\ n > m \rightarrow p + n > p + m.$
Hint *Resolve plus_gt_compat_l: arith v62.*

Chapter 46

Module Coq.Arith.Le

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

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

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

Open Local *Scope nat_scope.*

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

46.1 *le* is a pre-order

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

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

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

Comparison to 0

Theorem *le_O_n* : $\forall\ n,\ 0 \leq n$.

Theorem *le_Sn_O* : $\forall\ n,\ \neg\ S\ n \leq 0$.

Hint *Resolve le_O_n le_Sn_O*: *arith v62.*

Theorem *le_n_O_eq* : $\forall\ n,\ n \leq 0 \rightarrow 0 = n$.
Hint Immediate *le_n_O_eq*: *arith v62.*

le and successor

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

Theorem *le_n_Sn* : $\forall\ n,\ n \leq S\ n$.

Hint *Resolve le_n_S le_n_Sn* : *arith v62*.

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

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

Theorem *le_Sn_n* : $\forall\ n,\ \neg\ S\ n \leq n$.
Hint *Resolve le_Sn_n*: *arith v62*.

le and predecessor

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

Theorem *le_pred* : $\forall\ n\ m,\ n \leq m \rightarrow pred\ n \leq pred\ m$.

46.3 *le* is a order on *nat*

Antisymmetry

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

46.4 A different elimination principle for the order on natural numbers

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

Chapter 47

Module Coq.Arith.Lt

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

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

Require Import *Le*.
Open Local *Scope nat_scope*.

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

47.1 Irreflexivity

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

47.2 Relationship between *le* and *lt*

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

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

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

Theorem *le_not_lt* : $\forall\ n\ m, n \leq m \rightarrow \neg\ m < n$.

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

47.3 Asymmetry

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

47.4 Order and successor

Theorem lt_n_Sn : $\forall\ n,\ n < S\ n$.
Hint *Resolve lt_n_Sn: arith v62.*

Theorem lt_S : $\forall\ n\ m,\ n < m \rightarrow n < S\ m$.
Hint *Resolve lt_S: arith v62.*

Theorem lt_n_S : $\forall\ n\ m,\ n < m \rightarrow S\ n < S\ m$.
Hint *Resolve lt_n_S: arith v62.*

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

Theorem lt_O_Sn : $\forall\ n,\ 0 < S\ n$.
Hint *Resolve lt_O_Sn: arith v62.*

Theorem lt_n_O : $\forall\ n,\ \neg\ n < 0$.
Hint *Resolve lt_n_O: arith v62.*

47.5 Predecessor

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

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

Lemma $lt\_pred\_n\_n$: $\forall\ n,\ 0 < n \rightarrow pred\ n < n$.
Hint *Resolve lt_pred_n_n: arith v62.*

47.6 Transitivity properties

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

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

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

Hint *Resolve lt_trans lt_le_trans le_lt_trans: arith v62.*

47.7 Large = strict or equal

Theorem $le\_lt\_or\_eq : \forall\ n\ m,\ n \leq m \rightarrow n < m \vee n = m.$

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

47.8 Dichotomy

Theorem $le\_or\_lt : \forall\ n\ m,\ n \leq m \vee m < n.$

Theorem $nat\_total\_order : \forall\ n\ m,\ n \neq m \rightarrow n < m \vee m < n.$

47.9 Comparison to 0

Theorem $neq\_O\_lt : \forall\ n,\ 0 \neq n \rightarrow 0 < n.$
Hint Immediate neq_O_lt: $arith\ v62.$

Theorem $lt\_O\_neq : \forall\ n,\ 0 < n \rightarrow 0 \neq n.$
Hint Immediate lt_O_neq: $arith\ v62.$

Chapter 48

Module Coq.Arith.Max

Require Import *Arith*.

Open Local *Scope nat_scope*.

Implicit *Types m n* : *nat*.

48.1 maximum of two natural numbers

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

48.2 Simplifications of *max*

Lemma *max_SS* : ∀ *n m*, *S* (*max n m*) = *max* (*S n*) (*S m*).

Lemma *max_comm* : ∀ *n m*, *max n m* = *max m n*.

48.3 *max* and *le*

Lemma *max_l* : ∀ *n m*, *m* ≤ *n* → *max n m* = *n*.

Lemma *max_r* : ∀ *n m*, *n* ≤ *m* → *max n m* = *m*.

Lemma *le_max_l* : ∀ *n m*, *n* ≤ *max n m*.

Lemma *le_max_r* : ∀ *n m*, *m* ≤ *max n m*.
Hint *Resolve max_r max_l le_max_l le_max_r*: *arith v62*.

48.4 *max n m* is equal to *n* or *m*

Lemma max_dec : $\forall\ n\ m, \{max\ n\ m = n\} + \{max\ n\ m = m\}$.

Lemma max_case : $\forall\ n\ m\ (P{:}nat \to \mathsf{Type}), P\ n \to P\ m \to P\ (max\ n\ m)$.

Notation $max\_case2 := max\_case\ (only\ parsing)$.

Chapter 49

Module Coq.Arith.Minus

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

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

Require Import *Lt*.
Require Import *Le*.

Open Local *Scope nat_scope*.

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

49.1 0 is right neutral

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

49.2 Permutation with successor

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

Theorem *pred_of_minus* : $\forall$ n, *pred* $n = n$ - 1.

49.3 Diagonal

Lemma *minus_n_n* : $\forall\ n,\ 0 = n\ \text{-}\ n$.
Hint *Resolve minus_n_n*: *arith v62*.

49.4 Simplification

Lemma *minus_plus_simpl_l_reverse* : $\forall\ n\ m\ p,\ n\ \text{-}\ m = p + n\ \text{-}\ (p + m)$.
Hint *Resolve minus_plus_simpl_l_reverse*: *arith v62*.

49.5 Relation with plus

Lemma *plus_minus* : $\forall\ n\ m\ p,\ n = m + p \rightarrow p = n\ \text{-}\ m$.
Hint Immediate *plus_minus*: *arith v62*.

Lemma *minus_plus* : $\forall\ n\ m,\ n + m\ \text{-}\ n = m$.
Hint *Resolve minus_plus*: *arith v62*.

Lemma *le_plus_minus* : $\forall\ n\ m,\ n \leq m \rightarrow m = n + (m\ \text{-}\ n)$.
Hint *Resolve le_plus_minus*: *arith v62*.

Lemma *le_plus_minus_r* : $\forall\ n\ m,\ n \leq m \rightarrow n + (m\ \text{-}\ n) = m$.
Hint *Resolve le_plus_minus_r*: *arith v62*.

49.6 Relation with order

Theorem *le_minus* : $\forall\ n\ m,\ n\ \text{-}\ m \leq n$.

Lemma *lt_minus* : $\forall\ n\ m,\ m \leq n \rightarrow 0 < m \rightarrow n\ \text{-}\ m < n$.
Hint *Resolve lt_minus*: *arith v62*.

Lemma *lt_O_minus_lt* : $\forall\ n\ m,\ 0 < n\ \text{-}\ m \rightarrow m < n$.
Hint Immediate *lt_O_minus_lt*: *arith v62*.

Theorem *not_le_minus_0* : $\forall\ n\ m,\ \neg\ m \leq n \rightarrow n\ \text{-}\ m = 0$.

Chapter 50

Module Coq.Arith.Min

Require Import *Le*.

Open Local *Scope nat_scope*.

Implicit *Types m n* : *nat*.

50.1 minimum of two natural numbers

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

50.2 Simplifications of *min*

Lemma *min_SS* : ∀ *n m*, *S* (*min n m*) = *min* (*S n*) (*S m*).

Lemma *min_comm* : ∀ *n m*, *min n m* = *min m n*.

50.3 *min* and *le*

Lemma *min_l* : ∀ *n m*, *n* ≤ *m* → *min n m* = *n*.

Lemma *min_r* : ∀ *n m*, *m* ≤ *n* → *min n m* = *m*.

Lemma *le_min_l* : ∀ *n m*, *min n m* ≤ *n*.

Lemma *le_min_r* : ∀ *n m*, *min n m* ≤ *m*.
Hint *Resolve min_l min_r le_min_l le_min_r*: *arith v62*.

50.4 *min n m* is equal to *n* or *m*

Lemma $min\_dec : \forall\ n\ m, \{min\ n\ m = n\} + \{min\ n\ m = m\}$.

Lemma $min\_case : \forall\ n\ m\ (P{:}nat \to \mathsf{Type}),\ P\ n \to P\ m \to P\ (min\ n\ m)$.

Notation $min\_case2 := min\_case\ (only\ parsing)$.

Chapter 51

Module Coq.Arith.Mult

Require Export *Plus*.
Require Export *Minus*.
Require Export *Lt*.
Require Export *Le*.

Open Local *Scope nat_scope*.

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

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

51.1 *nat* is a semi-ring

51.1.1 Zero property

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

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

51.1.2 1 is neutral

Lemma *mult_1_l* : $\forall\ n,\ 1 \times n = n$.
Hint *Resolve mult_1_l*: *arith v62*.

Lemma *mult_1_r* : $\forall\ n,\ n \times 1 = n$.
Hint *Resolve mult_1_r*: *arith v62*.

51.1.3 Commutativity

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

51.1.4 Distributivity

Lemma *mult_plus_distr_r* : $\forall\ n\ m\ p, (n + m) \times p = n \times p + m \times p$.
Hint *Resolve mult_plus_distr_r*: *arith v62*.

Lemma *mult_plus_distr_l* : $\forall\ n\ m\ p, n \times (m + p) = n \times m + n \times p$.

Lemma *mult_minus_distr_r* : $\forall\ n\ m\ p, (n - m) \times p = n \times p - m \times p$.
Hint *Resolve mult_minus_distr_r*: *arith v62*.

Lemma *mult_minus_distr_l* : $\forall\ n\ m\ p, n \times (m - p) = n \times m - n \times p$.
Hint *Resolve mult_minus_distr_l*: *arith v62*.

51.1.5 Associativity

Lemma *mult_assoc_reverse* : $\forall\ n\ m\ p, n \times m \times p = n \times (m \times p)$.
Hint *Resolve mult_assoc_reverse*: *arith v62*.

Lemma *mult_assoc* : $\forall\ n\ m\ p, n \times (m \times p) = n \times m \times p$.
Hint *Resolve mult_assoc*: *arith v62*.

51.2 Compatibility with orders

Lemma *mult_O_le* : $\forall\ n\ m, m = 0 \vee n \leq m \times n$.
Hint *Resolve mult_O_le*: *arith v62*.

Lemma *mult_le_compat_l* : $\forall\ n\ m\ p, n \leq m \rightarrow p \times n \leq p \times m$.
Hint *Resolve mult_le_compat_l*: *arith*.

Lemma *mult_le_compat_r* : $\forall\ n\ m\ p, n \leq m \rightarrow n \times p \leq m \times p$.

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

Lemma *mult_S_lt_compat_l* : $\forall\ n\ m\ p, m < p \rightarrow S\ n \times m < S\ n \times p$.

Hint *Resolve mult_S_lt_compat_l*: *arith*.

Lemma *mult_lt_compat_r* : $\forall\ n\ m\ p, n < m \rightarrow 0 < p \rightarrow n \times p < m \times p$.

Lemma *mult_S_le_reg_l* : $\forall\ n\ m\ p, S\ n \times m \leq S\ n \times p \rightarrow m \leq p$.

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

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

51.4 Tail-recursive mult

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

```
Fixpoint mult_acc (s:nat) m n {struct n} : nat :=
  match n with
    | O ⇒ s
    | S p ⇒ mult_acc (tail_plus m s) m p
  end.
```

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

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

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

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

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

Chapter 52

Module Coq.Arith.Peano_dec

Require Import *Decidable*.

Open Local *Scope nat_scope*.

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

Theorem $O\_or\_S : \forall\ n, \{m : nat \mid S\ m = n\} + \{0 = n\}$.

Theorem $eq\_nat\_dec : \forall\ n\ m, \{n = m\} + \{n \neq m\}$.

Hint *Resolve* $O\_or\_S\ eq\_nat\_dec$: *arith*.

Theorem $dec\_eq\_nat : \forall\ n\ m,\ decidable\ (n = m)$.

Chapter 53

Module Coq.Arith.Plus

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

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

Require Import *Le*.
Require Import *Lt*.

Open Local *Scope nat_scope*.

Implicit *Types* m n p q : *nat*.

53.1 Zero is neutral

Lemma *plus_0_l* : $\forall\ n, 0 + n = n$.

Lemma *plus_0_r* : $\forall\ n, n + 0 = n$.

53.2 Commutativity

Lemma *plus_comm* : $\forall\ n\ m, n + m = m + n$.
Hint Immediate *plus_comm*: *arith v62*.

53.3 Associativity

Lemma *plus_Snm_nSm* : $\forall\ n\ m, S\ n + m = n + S\ m$.

Lemma *plus_assoc* : $\forall\ n\ m\ p, n + (m + p) = n + m + p$.
Hint *Resolve plus_assoc*: *arith v62*.

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

Lemma *plus_assoc_reverse* : $\forall\ n\ m\ p, n + m + p = n + (m + p)$.
Hint *Resolve plus_assoc_reverse*: *arith v62*.

53.4 Simplification

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

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

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

53.5 Compatibility with order

Lemma *plus_le_compat_l* : $\forall\ n\ m\ p, n \leq m \rightarrow p + n \leq p + m$.
Hint *Resolve plus_le_compat_l*: *arith v62*.

Lemma *plus_le_compat_r* : $\forall\ n\ m\ p, n \leq m \rightarrow n + p \leq m + p$.
Hint *Resolve plus_le_compat_r*: *arith v62*.

Lemma *le_plus_l* : $\forall\ n\ m, n \leq n + m$.
Hint *Resolve le_plus_l*: *arith v62*.

Lemma *le_plus_r* : $\forall\ n\ m, m \leq n + m$.
Hint *Resolve le_plus_r*: *arith v62*.

Theorem *le_plus_trans* : $\forall\ n\ m\ p, n \leq m \rightarrow n \leq m + p$.
Hint *Resolve le_plus_trans*: *arith v62*.

Theorem *lt_plus_trans* : $\forall\ n\ m\ p, n < m \rightarrow n < m + p$.
Hint Immediate *lt_plus_trans*: *arith v62*.

Lemma *plus_lt_compat_l* : $\forall\ n\ m\ p, n < m \rightarrow p + n < p + m$.
Hint *Resolve plus_lt_compat_l*: *arith v62*.

Lemma *plus_lt_compat_r* : $\forall\ n\ m\ p, n < m \rightarrow n + p < m + p$.
Hint *Resolve plus_lt_compat_r*: *arith v62*.

Lemma *plus_le_compat* : $\forall\ n\ m\ p\ q, n \leq m \rightarrow p \leq q \rightarrow n + p \leq m + q$.

Lemma *plus_le_lt_compat* : $\forall\ n\ m\ p\ q, n \leq m \rightarrow p < q \rightarrow n + p < m + q$.

Lemma *plus_lt_le_compat* : $\forall\ n\ m\ p\ q, n < m \rightarrow p \leq q \rightarrow n + p < m + q$.

Lemma *plus_lt_compat* : $\forall\ n\ m\ p\ q, n < m \rightarrow p < q \rightarrow n + p < m + q$.

53.6 Inversion lemmas

Lemma $plus\_is\_O : \forall\ n\ m,\ n + m = 0 \rightarrow n = 0 \wedge m = 0$.

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

53.7 Derived properties

Lemma $plus\_permute\_2\_in\_4 : \forall\ n\ m\ p\ q,\ n + m + (p + q) = n + p + (m + q)$.

53.8 Tail-recursive plus

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

Fixpoint $plus\_acc\ q\ n\ \{struct\ n\} : nat :=$
  match n with
    $| O \Rightarrow q$
    $| S\ p \Rightarrow plus\_acc\ (S\ q)\ p$
  end.

Definition $tail\_plus\ n\ m := plus\_acc\ m\ n$.

Lemma $plus\_tail\_plus : \forall\ n\ m,\ n + m = tail\_plus\ n\ m$.

53.9 Discrimination

Lemma $succ\_plus\_discr : \forall\ n\ m,\ n \neq S\ (plus\ m\ n)$.

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

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

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

Chapter 54

Module Coq.Arith.Wf_nat

Well-founded relations and natural numbers

Require Import *Lt*.

Open Local *Scope nat_scope*.

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

Section *Well_founded_Nat*.

Variable A : Type.

Variable f : $A \to nat$.
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 a = indrec ((f a)+1) a
  where rec indrec =
        function 0 → (function a → error)
                |(S m) → (function a → (F a (function y → indrec y m)));;
```

the ML-like program for *induction_ltof2* is :

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

Theorem *induction_ltof1* :
$\forall\ P{:}A \to$ Set,
$(\forall\ x{:}A, (\forall\ y{:}A, ltof\ y\ x \to P\ y) \to P\ x) \to \forall\ a{:}A, P\ a$.

Theorem *induction_gtof1* :
$\forall\ P{:}A \to$ Set,

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

Theorem *induction_ltof2* :
$\forall\ P{:}A \to \mathsf{Set},$
$(\forall\ x{:}A,\ (\forall\ y{:}A,\ ltof\ y\ x \to P\ y) \to P\ x) \to \forall\ a{:}A,\ P\ a.$

Theorem *induction_gtof2* :
$\forall\ P{:}A \to \mathsf{Set},$
$(\forall\ x{:}A,\ (\forall\ y{:}A,\ gtof\ y\ x \to P\ y) \to P\ x) \to \forall\ a{:}A,\ P\ a.$

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 $R : A \to A \to \mathsf{Prop}.$

Hypothesis *H_compat* : $\forall\ x\ y{:}A,\ R\ x\ y \to f\ x < f\ y.$

Theorem *well_founded_lt_compat* : *well_founded R*.

End *Well_founded_Nat*.

Lemma *lt_wf* : *well_founded lt*.

Lemma *lt_wf_rec1* :
$\forall\ n\ (P{:}nat \to \mathsf{Set}),\ (\forall\ n,\ (\forall\ m,\ m < n \to P\ m) \to P\ n) \to P\ n.$

Lemma *lt_wf_rec* :
$\forall\ n\ (P{:}nat \to \mathsf{Set}),\ (\forall\ n,\ (\forall\ m,\ m < n \to P\ m) \to P\ n) \to P\ n.$

Lemma *lt_wf_ind* :
$\forall\ n\ (P{:}nat \to \mathsf{Prop}),\ (\forall\ n,\ (\forall\ m,\ m < n \to P\ m) \to P\ n) \to P\ n.$

Lemma *gt_wf_rec* :
$\forall\ n\ (P{:}nat \to \mathsf{Set}),\ (\forall\ n,\ (\forall\ m,\ n > m \to P\ m) \to P\ n) \to P\ n.$

Lemma *gt_wf_ind* :
$\forall\ n\ (P{:}nat \to \mathsf{Prop}),\ (\forall\ n,\ (\forall\ m,\ n > m \to P\ m) \to P\ n) \to P\ n.$

Lemma *lt_wf_double_rec* :
$\forall\ P{:}nat \to nat \to \mathsf{Set},$
$(\forall\ n\ m,$
$(\forall\ p\ q,\ p < n \to P\ p\ q) \to$
$(\forall\ p,\ p < m \to P\ n\ p) \to P\ n\ m) \to \forall\ n\ m,\ P\ n\ m.$

Lemma *lt_wf_double_ind* :
$\forall\ P{:}nat \to nat \to \mathsf{Prop},$
$(\forall\ n\ m,$
$(\forall\ p\ (q{:}nat),\ p < n \to P\ p\ q) \to$
$(\forall\ p,\ p < m \to P\ n\ p) \to P\ n\ m) \to \forall\ n\ m,\ P\ n\ m.$

Hint *Resolve lt_wf*: *arith*.
Hint *Resolve well_founded_lt_compat*: *arith*.

Section *LT_WF_REL*.
Variable $A : \mathsf{Set}.$
Variable $R : A \to A \to \mathsf{Prop}.$

Variable $F : A \to nat \to$ Prop.
Definition $inv\_lt\_rel\ x\ y := exists2\ n, F\ x\ n\ \&\ (\forall\ m, F\ y\ m \to n < m)$.

Hypothesis $F\_compat : \forall\ x\ y{:}A, R\ x\ y \to inv\_lt\_rel\ x\ y$.
Remark $acc\_lt\_rel : \forall\ x{:}A, (\exists\ n, F\ x\ n) \to Acc\ R\ x$.

Theorem $well\_founded\_inv\_lt\_rel\_compat : well\_founded\ R$.

End LT_WF_REL.

Lemma $well\_founded\_inv\_rel\_inv\_lt\_rel$:
$\forall\ (A{:}$Set$)\ (F{:}A \to nat \to$ Prop$), well_founded\ (inv_lt_rel\ A\ F)$.

Chapter 55

Module Coq.ZArith.auxiliary

Binary Integers (Pierre Crégut, CNET, Lannion, France)

Require Export *Arith_base*.
Require Import *BinInt*.
Require Import *Zorder*.
Require Import *Decidable*.
Require Import *Peano_dec*.
Require Export *Compare_dec*.

Open Local *Scope Z_scope*.

55.1 Moving terms from one side to the other of an inequality

Theorem *Zne_left* : $\forall\ n\ m{:}Z,\ Zne\ n\ m \rightarrow Zne\ (n + \text{-}\ m)\ 0$.

Theorem *Zegal_left* : $\forall\ n\ m{:}Z,\ n = m \rightarrow n + \text{-}\ m = 0$.

Theorem *Zle_left* : $\forall\ n\ m{:}Z,\ n \leq m \rightarrow 0 \leq m + \text{-}\ n$.

Theorem *Zle_left_rev* : $\forall\ n\ m{:}Z,\ 0 \leq m + \text{-}\ n \rightarrow n \leq m$.

Theorem *Zlt_left_rev* : $\forall\ n\ m{:}Z,\ 0 < m + \text{-}\ n \rightarrow n < m$.

Theorem *Zlt_left* : $\forall\ n\ m{:}Z,\ n < m \rightarrow 0 \leq m + \text{-}1 + \text{-}\ n$.

Theorem *Zlt_left_lt* : $\forall\ n\ m{:}Z,\ n < m \rightarrow 0 < m + \text{-}\ n$.

Theorem *Zge_left* : $\forall\ n\ m{:}Z,\ n \geq m \rightarrow 0 \leq n + \text{-}\ m$.

Theorem *Zgt_left* : $\forall\ n\ m{:}Z,\ n > m \rightarrow 0 \leq n + \text{-}1 + \text{-}\ m$.

Theorem *Zgt_left_gt* : $\forall\ n\ m{:}Z,\ n > m \rightarrow n + \text{-}\ m > 0$.

Theorem *Zgt_left_rev* : $\forall\ n\ m{:}Z,\ n + \text{-}\ m > 0 \rightarrow n > m$.

55.2 Factorization lemmas

Theorem *Zred_factor0* : $\forall\ n{:}Z,\ n = n \times 1$.

Theorem *Zred_factor1* : $\forall\ n{:}Z,\ n + n = n \times 2$.

Theorem *Zred_factor2* : $\forall\ n\ m{:}Z,\ n + n \times m = n \times (1 + m)$.

Theorem *Zred_factor3* : $\forall\ n\ m{:}Z,\ n \times m + n = n \times (1 + m)$.

Theorem *Zred_factor4* : $\forall\ n\ m\ p{:}Z,\ n \times m + n \times p = n \times (m + p)$.

Theorem *Zred_factor5* : $\forall\ n\ m{:}Z,\ n \times 0 + m = m$.

Theorem *Zred_factor6* : $\forall\ n{:}Z,\ n = n + 0$.

Theorem *Zle_mult_approx* :
$\forall\ n\ m\ p{:}Z,\ n > 0 \rightarrow p > 0 \rightarrow 0 \leq m \rightarrow 0 \leq m \times n + p$.

Theorem *Zmult_le_approx* :
$\forall\ n\ m\ p{:}Z,\ n > 0 \rightarrow n > p \rightarrow 0 \leq m \times n + p \rightarrow 0 \leq m$.

Chapter 56

Module Coq.ZArith.BinInt

Binary Integers (Pierre Crégut, CNET, Lannion, France)
Require Export *BinPos*.
Require Export *Pnat*.
Require Import *BinNat*.
Require Import *Plus*.
Require Import *Mult*.

56.1 Binary integer numbers

Inductive Z : Set :=
  | $Z0 : Z$
  | $Zpos : positive \to Z$
  | $Zneg : positive \to Z$.

Automatically open scope positive_scope for the constructors of Z
Delimit Scope Z_scope with *Z*.

56.1.1 Subtraction of positive into Z

Definition $Zdouble\_plus\_one\ (x{:}Z) :=$
  match x with
    | $Z0 \Rightarrow Zpos\ 1$
    | $Zpos\ p \Rightarrow Zpos\ (xI\ p)$
    | $Zneg\ p \Rightarrow Zneg\ (Pdouble\_minus\_one\ p)$
  end.

Definition $Zdouble\_minus\_one\ (x{:}Z) :=$
  match x with
    | $Z0 \Rightarrow Zneg\ 1$
    | $Zneg\ p \Rightarrow Zneg\ (xI\ p)$
    | $Zpos\ p \Rightarrow Zpos\ (Pdouble\_minus\_one\ p)$

```
  end.

Definition Zdouble (x:Z) :=
  match x with
    | Z0 ⇒ Z0
    | Zpos p ⇒ Zpos (xO p)
    | Zneg p ⇒ Zneg (xO p)
  end.

Fixpoint ZPminus (x y:positive) {struct y} : Z :=
  match x, y with
    | xI x', xI y' ⇒ Zdouble (ZPminus x' y')
    | xI x', xO y' ⇒ Zdouble_plus_one (ZPminus x' y')
    | xI x', xH ⇒ Zpos (xO x')
    | xO x', xI y' ⇒ Zdouble_minus_one (ZPminus x' y')
    | xO x', xO y' ⇒ Zdouble (ZPminus x' y')
    | xO x', xH ⇒ Zpos (Pdouble_minus_one x')
    | xH, xI y' ⇒ Zneg (xO y')
    | xH, xO y' ⇒ Zneg (Pdouble_minus_one y')
    | xH, xH ⇒ Z0
  end.
```

56.1.2 Addition on integers

```
Definition Zplus (x y:Z) :=
  match x, y with
    | Z0, y ⇒ y
    | x, Z0 ⇒ x
    | Zpos x', Zpos y' ⇒ Zpos (x' + y')
    | Zpos x', Zneg y' ⇒
      match (x' ?= y')%positive Eq with
        | Eq ⇒ Z0
        | Lt ⇒ Zneg (y' - x')
        | Gt ⇒ Zpos (x' - y')
      end
    | Zneg x', Zpos y' ⇒
      match (x' ?= y')%positive Eq with
        | Eq ⇒ Z0
        | Lt ⇒ Zpos (y' - x')
        | Gt ⇒ Zneg (x' - y')
      end
    | Zneg x', Zneg y' ⇒ Zneg (x' + y')
  end.

Infix "+" := Zplus : Z_scope.
```

56.1.3 Opposite

```
Definition Zopp (x:Z) :=
  match x with
    | Z0 ⇒ Z0
    | Zpos x ⇒ Zneg x
    | Zneg x ⇒ Zpos x
  end.

Notation "- x" := (Zopp x) : Z_scope.
```

56.1.4 Successor on integers

```
Definition Zsucc (x:Z) := (x + Zpos 1)%Z.
```

56.1.5 Predecessor on integers

```
Definition Zpred (x:Z) := (x + Zneg 1)%Z.
```

56.1.6 Subtraction on integers

```
Definition Zminus (m n:Z) := (m + - n)%Z.

Infix "-" := Zminus : Z_scope.
```

56.1.7 Multiplication on integers

```
Definition Zmult (x y:Z) :=
  match x, y with
    | Z0, _ ⇒ Z0
    | _, Z0 ⇒ Z0
    | Zpos x', Zpos y' ⇒ Zpos (x' × y')
    | Zpos x', Zneg y' ⇒ Zneg (x' × y')
    | Zneg x', Zpos y' ⇒ Zneg (x' × y')
    | Zneg x', Zneg y' ⇒ Zpos (x' × y')
  end.

Infix "×" := Zmult : Z_scope.
```

56.1.8 Comparison of integers

```
Definition Zcompare (x y:Z) :=
  match x, y with
    | Z0, Z0 ⇒ Eq
    | Z0, Zpos y' ⇒ Lt
```

```
    | Z0, Zneg y' ⇒ Gt
    | Zpos x', Z0 ⇒ Gt
    | Zpos x', Zpos y' ⇒ (x' ?= y')%positive Eq
    | Zpos x', Zneg y' ⇒ Gt
    | Zneg x', Z0 ⇒ Lt
    | Zneg x', Zpos y' ⇒ Lt
    | Zneg x', Zneg y' ⇒ CompOpp ((x' ?= y')%positive Eq)
  end.

Infix "?=" := Zcompare (at level 70, no associativity) : Z_scope.

Ltac elim_compare com1 com2 :=
  case (Dcompare (com1 ?= com2)%Z);
    [ idtac | let x := fresh "H" in
      (intro x; case x; clear x) ].
```

56.1.9 Sign function

```
Definition Zsgn (z:Z) : Z :=
  match z with
    | Z0 ⇒ Z0
    | Zpos p ⇒ Zpos 1
    | Zneg p ⇒ Zneg 1
  end.
```

56.1.10 Direct, easier to handle variants of successor and addition

```
Definition Zsucc' (x:Z) :=
  match x with
    | Z0 ⇒ Zpos 1
    | Zpos x' ⇒ Zpos (Psucc x')
    | Zneg x' ⇒ ZPminus 1 x'
  end.

Definition Zpred' (x:Z) :=
  match x with
    | Z0 ⇒ Zneg 1
    | Zpos x' ⇒ ZPminus x' 1
    | Zneg x' ⇒ Zneg (Psucc x')
  end.

Definition Zplus' (x y:Z) :=
  match x, y with
    | Z0, y ⇒ y
    | x, Z0 ⇒ x
    | Zpos x', Zpos y' ⇒ Zpos (x' + y')
    | Zpos x', Zneg y' ⇒ ZPminus x' y'
```

```
    | Zneg x', Zpos y' ⇒ ZPminus y' x'
    | Zneg x', Zneg y' ⇒ Zneg (x' + y')
  end.
```

Open Local *Scope Z_scope.*

56.1.11 Inductive specification of Z

Theorem *Zind* :
∀ *P*:*Z* → Prop,
P Z0 →
(∀ *x*:*Z*, *P x* → *P* (*Zsucc' x*)) →
(∀ *x*:*Z*, *P x* → *P* (*Zpred' x*)) → ∀ *n*:*Z*, *P n*.

56.2 Misc properties about binary integer operations

56.2.1 Properties of opposite on binary integer numbers

Theorem *Zopp_neg* : ∀ *p*:*positive*, - *Zneg p* = *Zpos p*.

opp is involutive

Theorem *Zopp_involutive* : ∀ *n*:*Z*, - - *n* = *n*.

Injectivity of the opposite

Theorem *Zopp_inj* : ∀ *n m*:*Z*, - *n* = - *m* → *n* = *m*.

56.2.2 Properties of the direct definition of successor and predecessor

Lemma *Zpred'_succ'* : ∀ *n*:*Z*, *Zpred'* (*Zsucc' n*) = *n*.

Lemma *Zsucc'_discr* : ∀ *n*:*Z*, *n* ≠ *Zsucc' n*.

56.2.3 Other properties of binary integer numbers

Lemma *ZL0* : 2%*nat* = (1 + 1)%*nat*.

56.3 Properties of the addition on integers

56.3.1 zero is left neutral for addition

Theorem *Zplus_0_l* : ∀ *n*:*Z*, *Z0* + *n* = *n*.

zero is right neutral for addition

Theorem *Zplus_0_r* : $\forall\ n{:}Z,\ n + Z0 = n$.

56.3.2 addition is commutative

Theorem *Zplus_comm* : $\forall\ n\ m{:}Z,\ n + m = m + n$.

56.3.3 opposite distributes over addition

Theorem *Zopp_plus_distr* : $\forall\ n\ m{:}Z,\ -\ (n + m) = -\ n + -\ m$.

56.3.4 opposite is inverse for addition

Theorem *Zplus_opp_r* : $\forall\ n{:}Z,\ n + -\ n = Z0$.

Theorem *Zplus_opp_l* : $\forall\ n{:}Z,\ -\ n + n = Z0$.

Hint Local *Resolve Zplus_0_l Zplus_0_r*.

56.3.5 addition is associative

Lemma *weak_assoc* :
$\forall\ (p\ q{:}positive)\ (n{:}Z),\ Zpos\ p + (Zpos\ q + n) = Zpos\ p + Zpos\ q + n$.

Hint Local *Resolve weak_assoc*.

Theorem *Zplus_assoc* : $\forall\ n\ m\ p{:}Z,\ n + (m + p) = n + m + p$.

Lemma *Zplus_assoc_reverse* : $\forall\ n\ m\ p{:}Z,\ n + m + p = n + (m + p)$.

56.3.6 Associativity mixed with commutativity

Theorem *Zplus_permute* : $\forall\ n\ m\ p{:}Z,\ n + (m + p) = m + (n + p)$.

56.3.7 addition simplifies

Theorem *Zplus_reg_l* : $\forall\ n\ m\ p{:}Z,\ n + m = n + p \rightarrow m = p$.

56.3.8 addition and successor permutes

Lemma *Zplus_succ_l* : $\forall\ n\ m{:}Z,\ Zsucc\ n + m = Zsucc\ (n + m)$.

Lemma *Zplus_succ_r* : $\forall\ n\ m{:}Z,\ Zsucc\ (n + m) = n + Zsucc\ m$.

Lemma *Zplus_succ_comm* : $\forall\ n\ m{:}Z,\ Zsucc\ n + m = n + Zsucc\ m$.

56.3.9 Misc properties, usually redundant or non natural

Lemma *Zplus_0_r_reverse* : $\forall\ n{:}Z,\ n = n + Z0$.

Lemma *Zplus_0_simpl_l* : $\forall\ n\ m{:}Z,\ n + Z0 = m \rightarrow n = m$.

Lemma *Zplus_0_simpl_l_reverse* : $\forall\ n\ m{:}Z,\ n = m + Z0 \rightarrow n = m$.

Lemma *Zplus_eq_compat* : $\forall\ n\ m\ p\ q{:}Z,\ n = m \rightarrow p = q \rightarrow n + p = m + q$.

Lemma *Zplus_opp_expand* : $\forall\ n\ m\ p{:}Z,\ n + - m = n + - p + (p + - m)$.

56.4 Properties of successor and predecessor on binary integer numbers

Theorem *Zsucc_discr* : $\forall\ n{:}Z,\ n \neq Zsucc\ n$.

Theorem *Zpos_succ_morphism* :
$\forall\ p{:}positive,\ Zpos\ (Psucc\ p) = Zsucc\ (Zpos\ p)$.

successor and predecessor are inverse functions

Theorem *Zsucc_pred* : $\forall\ n{:}Z,\ n = Zsucc\ (Zpred\ n)$.

Hint Immediate *Zsucc_pred*: *zarith*.

Theorem *Zpred_succ* : $\forall\ n{:}Z,\ n = Zpred\ (Zsucc\ n)$.

Theorem *Zsucc_inj* : $\forall\ n\ m{:}Z,\ Zsucc\ n = Zsucc\ m \rightarrow n = m$.

Misc properties, usually redundant or non natural

Lemma *Zsucc_eq_compat* : $\forall\ n\ m{:}Z,\ n = m \rightarrow Zsucc\ n = Zsucc\ m$.

Lemma *Zsucc_inj_contrapositive* : $\forall\ n\ m{:}Z,\ n \neq m \rightarrow Zsucc\ n \neq Zsucc\ m$.

56.5 Properties of subtraction on binary integer numbers

56.5.1 *minus* and *Z0*

Lemma *Zminus_0_r* : $\forall\ n{:}Z,\ n - Z0 = n$.

Lemma *Zminus_0_l_reverse* : $\forall\ n{:}Z,\ n = n - Z0$.

Lemma *Zminus_diag* : $\forall\ n{:}Z,\ n - n = Z0$.

Lemma *Zminus_diag_reverse* : $\forall\ n{:}Z,\ Z0 = n - n$.

56.5.2 Relating *minus* with *plus* and *Zsucc*

Lemma *Zplus_minus_eq* : $\forall\ n\ m\ p{:}Z,\ n = m + p \rightarrow p = n - m.$

Lemma *Zminus_plus* : $\forall\ n\ m{:}Z,\ n + m - n = m.$

Lemma *Zplus_minus* : $\forall\ n\ m{:}Z,\ n + (m - n) = m.$

Lemma *Zminus_succ_l* : $\forall\ n\ m{:}Z,\ Zsucc\ (n - m) = Zsucc\ n - m.$

Lemma *Zminus_plus_simpl_l* : $\forall\ n\ m\ p{:}Z,\ p + n - (p + m) = n - m.$

Lemma *Zminus_plus_simpl_l_reverse* : $\forall\ n\ m\ p{:}Z,\ n - m = p + n - (p + m).$

Lemma *Zminus_plus_simpl_r* : $\forall\ n\ m\ p{:}Z,\ n + p - (m + p) = n - m.$

56.5.3 Misc redundant properties

Lemma *Zeq_minus* : $\forall\ n\ m{:}Z,\ n = m \rightarrow n - m = Z0.$

Lemma *Zminus_eq* : $\forall\ n\ m{:}Z,\ n - m = Z0 \rightarrow n = m.$

56.6 Properties of multiplication on binary integer numbers

Theorem *Zpos_mult_morphism* :
$\forall\ p\ q{:}positive,\ Zpos\ (p \times q) = Zpos\ p \times Zpos\ q.$

56.6.1 One is neutral for multiplication

Theorem *Zmult_1_l* : $\forall\ n{:}Z,\ Zpos\ 1 \times n = n.$

Theorem *Zmult_1_r* : $\forall\ n{:}Z,\ n \times Zpos\ 1 = n.$

56.6.2 Zero property of multiplication

Theorem *Zmult_0_l* : $\forall\ n{:}Z,\ Z0 \times n = Z0.$

Theorem *Zmult_0_r* : $\forall\ n{:}Z,\ n \times Z0 = Z0.$

Hint Local *Resolve Zmult_0_l Zmult_0_r.*

Lemma *Zmult_0_r_reverse* : $\forall\ n{:}Z,\ Z0 = n \times Z0.$

56.6.3 Commutativity of multiplication

Theorem *Zmult_comm* : $\forall\ n\ m{:}Z,\ n \times m = m \times n.$

56.6.4 Associativity of multiplication

Theorem $Zmult\_assoc : \forall\ n\ m\ p{:}Z,\ n \times (m \times p) = n \times m \times p.$

Lemma $Zmult\_assoc\_reverse : \forall\ n\ m\ p{:}Z,\ n \times m \times p = n \times (m \times p).$

56.6.5 Associativity mixed with commutativity

Theorem $Zmult\_permute : \forall\ n\ m\ p{:}Z,\ n \times (m \times p) = m \times (n \times p).$

56.6.6 Z is integral

Theorem $Zmult\_integral\_l : \forall\ n\ m{:}Z,\ n \neq Z0 \rightarrow m \times n = Z0 \rightarrow m = Z0.$

Theorem $Zmult\_integral : \forall\ n\ m{:}Z,\ n \times m = Z0 \rightarrow n = Z0 \vee m = Z0.$

Lemma $Zmult\_1\_inversion\_l :$
$\forall\ n\ m{:}Z,\ n \times m = Zpos\ 1 \rightarrow n = Zpos\ 1 \vee n = Zneg\ 1.$

56.6.7 Multiplication and Opposite

Theorem $Zopp\_mult\_distr\_l : \forall\ n\ m{:}Z,\ -\ (n \times m) = -\ n \times m.$

Theorem $Zopp\_mult\_distr\_r : \forall\ n\ m{:}Z,\ -\ (n \times m) = n \times -\ m.$

Lemma $Zopp\_mult\_distr\_l\_reverse : \forall\ n\ m{:}Z,\ -\ n \times m = -\ (n \times m).$

Theorem $Zmult\_opp\_comm : \forall\ n\ m{:}Z,\ -\ n \times m = n \times -\ m.$

Theorem $Zmult\_opp\_opp : \forall\ n\ m{:}Z,\ -\ n \times -\ m = n \times m.$

Theorem $Zopp\_eq\_mult\_neg\_1 : \forall\ n{:}Z,\ -\ n = n \times Zneg\ 1.$

56.6.8 Distributivity of multiplication over addition

Lemma $weak\_Zmult\_plus\_distr\_r :$
$\forall\ (p{:}positive)\ (n\ m{:}Z),\ Zpos\ p \times (n + m) = Zpos\ p \times n + Zpos\ p \times m.$

Theorem $Zmult\_plus\_distr\_r : \forall\ n\ m\ p{:}Z,\ n \times (m + p) = n \times m + n \times p.$

Theorem $Zmult\_plus\_distr\_l : \forall\ n\ m\ p{:}Z,\ (n + m) \times p = n \times p + m \times p.$

56.6.9 Distributivity of multiplication over subtraction

Lemma $Zmult\_minus\_distr\_r : \forall\ n\ m\ p{:}Z,\ (n\ -\ m) \times p = n \times p\ -\ m \times p.$

Lemma $Zmult\_minus\_distr\_l : \forall\ n\ m\ p{:}Z,\ p \times (n\ -\ m) = p \times n\ -\ p \times m.$

56.6.10 Simplification of multiplication for non-zero integers

Lemma $Zmult\_reg\_l : \forall\ n\ m\ p{:}Z,\ p \neq Z0 \rightarrow p \times n = p \times m \rightarrow n = m.$

Lemma $Zmult\_reg\_r : \forall\ n\ m\ p{:}Z,\ p \neq Z0 \rightarrow n \times p = m \times p \rightarrow n = m.$

56.6.11 Addition and multiplication by 2

Lemma $Zplus\_diag\_eq\_mult\_2 : \forall\ n{:}Z,\ n + n = n \times Zpos\ 2.$

56.6.12 Multiplication and successor

Lemma $Zmult\_succ\_r : \forall\ n\ m{:}Z,\ n \times Zsucc\ m = n \times m + n.$

Lemma $Zmult\_succ\_r\_reverse : \forall\ n\ m{:}Z,\ n \times m + n = n \times Zsucc\ m.$

Lemma $Zmult\_succ\_l : \forall\ n\ m{:}Z,\ Zsucc\ n \times m = n \times m + m.$

Lemma $Zmult\_succ\_l\_reverse : \forall\ n\ m{:}Z,\ n \times m + m = Zsucc\ n \times m.$

56.6.13 Misc redundant properties

Lemma $Z\_eq\_mult : \forall\ n\ m{:}Z,\ m = Z0 \rightarrow m \times n = Z0.$

56.7 Relating binary positive numbers and binary integers

Lemma $Zpos\_xI : \forall\ p{:}positive,\ Zpos\ (xI\ p) = Zpos\ 2 \times Zpos\ p + Zpos\ 1.$

Lemma $Zpos\_xO : \forall\ p{:}positive,\ Zpos\ (xO\ p) = Zpos\ 2 \times Zpos\ p.$

Lemma $Zneg\_xI : \forall\ p{:}positive,\ Zneg\ (xI\ p) = Zpos\ 2 \times Zneg\ p$ - $Zpos\ 1.$

Lemma $Zneg\_xO : \forall\ p{:}positive,\ Zneg\ (xO\ p) = Zpos\ 2 \times Zneg\ p.$

Lemma $Zpos\_plus\_distr : \forall\ p\ q{:}positive,\ Zpos\ (p + q) = Zpos\ p + Zpos\ q.$

Lemma $Zneg\_plus\_distr : \forall\ p\ q{:}positive,\ Zneg\ (p + q) = Zneg\ p + Zneg\ q.$

56.8 Order relations

Definition $Zlt\ (x\ y{:}Z) := (x\ ?{=}\ y) = Lt.$
Definition $Zgt\ (x\ y{:}Z) := (x\ ?{=}\ y) = Gt.$
Definition $Zle\ (x\ y{:}Z) := (x\ ?{=}\ y) \neq Gt.$
Definition $Zge\ (x\ y{:}Z) := (x\ ?{=}\ y) \neq Lt.$
Definition $Zne\ (x\ y{:}Z) := x \neq y.$

Infix "$\leq$" $:= Zle : Z\_scope.$

```
Infix "<" := Zlt : Z_scope.
Infix "≥" := Zge : Z_scope.
Infix ">" := Zgt : Z_scope.

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

56.9 Absolute value on integers

```
Definition Zabs_nat (x:Z) : nat :=
  match x with
    | Z0 ⇒ 0%nat
    | Zpos p ⇒ nat_of_P p
    | Zneg p ⇒ nat_of_P p
  end.

Definition Zabs (z:Z) : Z :=
  match z with
    | Z0 ⇒ Z0
    | Zpos p ⇒ Zpos p
    | Zneg p ⇒ Zpos p
  end.
```

56.10 From *nat* to *Z*

```
Definition Z_of_nat (x:nat) :=
  match x with
    | O ⇒ Z0
    | S y ⇒ Zpos (P_of_succ_nat y)
  end.

Require Import BinNat.

Definition Zabs_N (z:Z) :=
  match z with
    | Z0 ⇒ 0%N
    | Zpos p ⇒ Npos p
    | Zneg p ⇒ Npos p
  end.

Definition Z_of_N (x:N) := match x with
                             | N0 ⇒ Z0
                             | Npos p ⇒ Zpos p
                           end.
```

Chapter 57

Module Coq.ZArith.Int

An axiomatization of integers.

We define a signature for an integer datatype based on *Z*. The goal is to allow a switch after extraction to ocaml's *big_int* or even *int* when finiteness isn't a problem (typically : when mesuring the height of an AVL tree).

Require Import *ZArith*.
Require Import *ROmega*.
Delimit Scope Int_scope with *I*.

57.1 a specification of integers

Module Type *Int*.

Open Scope Int_scope.

Parameter *int* : Set.

Parameter *i2z* : *int* → *Z*.

Parameter *_0* : *int*.
Parameter *_1* : *int*.
Parameter *_2* : *int*.
Parameter *_3* : *int*.
Parameter *plus* : *int* → *int* → *int*.
Parameter *opp* : *int* → *int*.
Parameter *minus* : *int* → *int* → *int*.
Parameter *mult* : *int* → *int* → *int*.
Parameter *max* : *int* → *int* → *int*.

Notation "0" := *_0* : *Int_scope*.
Notation "1" := *_1* : *Int_scope*.
Notation "2" := *_2* : *Int_scope*.
Notation "3" := *_3* : *Int_scope*.
Infix "+" := *plus* : *Int_scope*.

Infix "-" := *minus* : *Int_scope*.
Infix "×" := *mult* : *Int_scope*.
Notation "- x" := (*opp* x) : *Int_scope*.

For logical relations, we can rely on their counterparts in Z, since they don't appear after extraction. Moreover, using tactics like omega is easier this way.

Notation "$x == y$" := (*i2z* x = *i2z* y)
(*at level* 70, y *at next level*, *no associativity*) : *Int_scope*.
Notation "$x \leq y$" := (*Zle* (*i2z* x) (*i2z* y)): *Int_scope*.
Notation "$x < y$" := (*Zlt* (*i2z* x) (*i2z* y)) : *Int_scope*.
Notation "$x \geq y$" := (*Zge* (*i2z* x) (*i2z* y)) : *Int_scope*.
Notation "$x > y$" := (*Zgt* (*i2z* x) (*i2z* y)): *Int_scope*.
Notation "$x \leq y \leq z$" := ($x \leq y \wedge y \leq z$) : *Int_scope*.
Notation "$x \leq y < z$" := ($x \leq y \wedge y < z$) : *Int_scope*.
Notation "$x < y < z$" := ($x < y \wedge y < z$) : *Int_scope*.
Notation "$x < y \leq z$" := ($x < y \wedge y \leq z$) : *Int_scope*.

Some decidability fonctions (informative).

Axiom *gt_le_dec* : $\forall$ x y: *int*, $\{x > y\} + \{x \leq y\}$.
Axiom *ge_lt_dec* : $\forall$ x y : *int*, $\{x \geq y\} + \{x < y\}$.
Axiom *eq_dec* : $\forall$ x y : *int*, { $x == y$ } + {~ x==y }.

Specifications

First, we ask *i2z* to be injective. Said otherwise, our ad-hoc equality == and the generic = are in fact equivalent. We define == nonetheless since the translation to *Z* for using automatic tactic is easier.

Axiom *i2z_eq* : $\forall$ n p : *int*, $n == p \rightarrow n = p$.

Then, we express the specifications of the above parameters using their Z counterparts.

Open Scope Z_scope.
Axiom *i2z_0* : *i2z* *_0* = 0.
Axiom *i2z_1* : *i2z* *_1* = 1.
Axiom *i2z_2* : *i2z* *_2* = 2.
Axiom *i2z_3* : *i2z* *_3* = 3.
Axiom *i2z_plus* : $\forall$ n p, *i2z* (n + p) = *i2z* n + *i2z* p.
Axiom *i2z_opp* : $\forall$ n, *i2z* (-n) = -*i2z* n.
Axiom *i2z_minus* : $\forall$ n p, *i2z* (n - p) = *i2z* n - *i2z* p.
Axiom *i2z_mult* : $\forall$ n p, *i2z* ($n \times p$) = *i2z* $n \times$ *i2z* p.
Axiom *i2z_max* : $\forall$ n p, *i2z* (*max* n p) = *Zmax* (*i2z* n) (*i2z* p).

End *Int*.

57.2 Facts and tactics using *Int*

Module *MoreInt* (*I*:*Int*).

```
Import I.

Open Scope Int_scope.
```

A magic (but costly) tactic that goes from *int* back to the *Z* friendly world ...

```
Hint Rewrite →
  i2z_0 i2z_1 i2z_2 i2z_3 i2z_plus i2z_opp i2z_minus i2z_mult i2z_max : i2z.

Ltac i2z := match goal with
                | H : (eq (A:=int) ?a ?b) ⊢ _ ⇒
                  generalize (f_equal i2z H);
                    try autorewrite with i2z; clear H; intro H; i2z
                | ⊢ (eq (A:=int) ?a ?b) ⇒ apply (i2z_eq a b); try autorewrite with i2z; i2z
                | H : _ ⊢ _ ⇒ progress autorewrite with i2z in H; i2z
                | _ ⇒ try autorewrite with i2z
              end.
```

A reflexive version of the *i2z* tactic

this *i2z_refl* is actually weaker than *i2z*. For instance, if a *i2z* is buried deep inside a subterm, *i2z_refl* may miss it. See also the limitation about Set or Type part below. Anyhow, *i2z_refl* is enough for applying *romega*.

```
Ltac i2z_gen := match goal with
  | ⊢ (eq (A:=int) ?a ?b) ⇒ apply (i2z_eq a b); i2z_gen
  | H : (eq (A:=int) ?a ?b) ⊢ _ ⇒
     generalize (f_equal i2z H); clear H; i2z_gen
  | H : (eq (A:=Z) ?a ?b) ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : (Zlt ?a ?b) ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : (Zle ?a ?b) ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : (Zgt ?a ?b) ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : (Zge ?a ?b) ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : _ → ?X ⊢ _ ⇒
          match type of X with
    | Type ⇒ clear H; i2z_gen
    | Prop ⇒ generalize H; clear H; i2z_gen
    end
  | H : _ ↔ _ ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : _ ∧ _ ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : _ ∨ _ ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | H : ¬ _ ⊢ _ ⇒ generalize H; clear H; i2z_gen
  | _ ⇒ idtac
 end.

Inductive ExprI : Set :=
  | EI0 : ExprI
  | EI1 : ExprI
  | EI2 : ExprI
  | EI3 : ExprI
```

```
  | EIplus : ExprI → ExprI → ExprI
  | EIopp : ExprI → ExprI
  | EIminus : ExprI → ExprI → ExprI
  | EImult : ExprI → ExprI → ExprI
  | EImax : ExprI → ExprI → ExprI
  | EIraw : int → ExprI.

Inductive ExprZ : Set :=
  | EZplus : ExprZ → ExprZ → ExprZ
  | EZopp : ExprZ → ExprZ
  | EZminus : ExprZ → ExprZ → ExprZ
  | EZmult : ExprZ → ExprZ → ExprZ
  | EZmax : ExprZ → ExprZ → ExprZ
  | EZofI : ExprI → ExprZ
  | EZraw : Z → ExprZ.

Inductive ExprP : Type :=
  | EPeq : ExprZ → ExprZ → ExprP
  | EPlt : ExprZ → ExprZ → ExprP
  | EPle : ExprZ → ExprZ → ExprP
  | EPgt : ExprZ → ExprZ → ExprP
  | EPge : ExprZ → ExprZ → ExprP
  | EPimpl : ExprP → ExprP → ExprP
  | EPequiv : ExprP → ExprP → ExprP
  | EPand : ExprP → ExprP → ExprP
  | EPor : ExprP → ExprP → ExprP
  | EPneg : ExprP → ExprP
  | EPraw : Prop → ExprP.
```

int to *ExprI*

```
Ltac i2ei trm :=
  match constr:trm with
    | 0 ⇒ constr:EI0
    | 1 ⇒ constr:EI1
    | 2 ⇒ constr:EI2
    | 3 ⇒ constr:EI3
    | ?x + ?y ⇒ let ex := i2ei x with ey := i2ei y in constr:(EIplus ex ey)
    | ?x - ?y ⇒ let ex := i2ei x with ey := i2ei y in constr:(EIminus ex ey)
    | ?x × ?y ⇒ let ex := i2ei x with ey := i2ei y in constr:(EImult ex ey)
    | max ?x ?y ⇒ let ex := i2ei x with ey := i2ei y in constr:(EImax ex ey)
    | - ?x ⇒ let ex := i2ei x in constr:(EIopp ex)
    | ?x ⇒ constr:(EIraw x)
  end
(** [Z] to [ExprZ] *)
  with z2ez trm :=
  match constr:trm with
    | (?x+?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EZplus ex ey)
```

```
    | (?x-?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EZminus ex ey)
    | (?x*?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EZmult ex ey)
    | (Zmax ?x ?y) ⇒ let ex := z2ez x with ey := z2ez y in constr:(EZmax ex ey)
    | (-?x)%Z ⇒ let ex := z2ez x in constr:(EZopp ex)
    | i2z ?x ⇒ let ex := i2ei x in constr:(EZofI ex)
    | ?x ⇒ constr:(EZraw x)
  end.
```

Prop to *ExprP*

```
Ltac p2ep trm :=
  match constr:trm with
    | (?x ↔ ?y) ⇒ let ex := p2ep x with ey := p2ep y in constr:(EPequiv ex ey)
    | (?x → ?y) ⇒ let ex := p2ep x with ey := p2ep y in constr:(EPimpl ex ey)
    | (?x ∧ ?y) ⇒ let ex := p2ep x with ey := p2ep y in constr:(EPand ex ey)
    | (?x ∨ ?y) ⇒ let ex := p2ep x with ey := p2ep y in constr:(EPor ex ey)
    | (~ ?x) ⇒ let ex := p2ep x in constr:(EPneg ex)
    | (eq (A:=Z) ?x ?y) ⇒ let ex := z2ez x with ey := z2ez y in constr:(EPeq ex ey)
    | (?x<?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EPlt ex ey)
    | (?x<=?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EPle ex ey)
    | (?x>?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EPgt ex ey)
    | (?x>=?y)%Z ⇒ let ex := z2ez x with ey := z2ez y in constr:(EPge ex ey)
    | ?x ⇒ constr:(EPraw x)
  end.
```

ExprI to *int*

```
Fixpoint ei2i (e:ExprI) : int :=
  match e with
    | EI0 ⇒ 0
    | EI1 ⇒ 1
    | EI2 ⇒ 2
    | EI3 ⇒ 3
    | EIplus e1 e2 ⇒ (ei2i e1)+(ei2i e2)
    | EIminus e1 e2 ⇒ (ei2i e1)-(ei2i e2)
    | EImult e1 e2 ⇒ (ei2i e1)*(ei2i e2)
    | EImax e1 e2 ⇒ max (ei2i e1) (ei2i e2)
    | EIopp e ⇒ -(ei2i e)
    | EIraw i ⇒ i
  end.
```

ExprZ to *Z*

```
Fixpoint ez2z (e:ExprZ) : Z :=
  match e with
    | EZplus e1 e2 ⇒ ((ez2z e1)+(ez2z e2))%Z
    | EZminus e1 e2 ⇒ ((ez2z e1)-(ez2z e2))%Z
    | EZmult e1 e2 ⇒ ((ez2z e1)*(ez2z e2))%Z
    | EZmax e1 e2 ⇒ Zmax (ez2z e1) (ez2z e2)
```

```
    | EZopp e ⇒ (-(ez2z e))%Z
    | EZofI e ⇒ i2z (ei2i e)
    | EZraw z ⇒ z
  end.
```

ExprP to Prop

```
Fixpoint ep2p (e:ExprP) : Prop :=
  match e with
    | EPeq e1 e2 ⇒ (ez2z e1) = (ez2z e2)
    | EPlt e1 e2 ⇒ ((ez2z e1)<(ez2z e2))%Z
    | EPle e1 e2 ⇒ ((ez2z e1)<=(ez2z e2))%Z
    | EPgt e1 e2 ⇒ ((ez2z e1)>(ez2z e2))%Z
    | EPge e1 e2 ⇒ ((ez2z e1)>=(ez2z e2))%Z
    | EPimpl e1 e2 ⇒ (ep2p e1) → (ep2p e2)
    | EPequiv e1 e2 ⇒ (ep2p e1) ↔ (ep2p e2)
    | EPand e1 e2 ⇒ (ep2p e1) ∧ (ep2p e2)
    | EPor e1 e2 ⇒ (ep2p e1) ∨ (ep2p e2)
    | EPneg e ⇒ ¬ (ep2p e)
    | EPraw p ⇒ p
  end.
```

ExprI (supposed under a *i2z*) to a simplified *ExprZ*

```
Fixpoint norm_ei (e:ExprI) : ExprZ :=
  match e with
    | EI0 ⇒ EZraw (0%Z)
    | EI1 ⇒ EZraw (1%Z)
    | EI2 ⇒ EZraw (2%Z)
    | EI3 ⇒ EZraw (3%Z)
    | EIplus e1 e2 ⇒ EZplus (norm_ei e1) (norm_ei e2)
    | EIminus e1 e2 ⇒ EZminus (norm_ei e1) (norm_ei e2)
    | EImult e1 e2 ⇒ EZmult (norm_ei e1) (norm_ei e2)
    | EImax e1 e2 ⇒ EZmax (norm_ei e1) (norm_ei e2)
    | EIopp e ⇒ EZopp (norm_ei e)
    | EIraw i ⇒ EZofI (EIraw i)
  end.
```

ExprZ to a simplified *ExprZ*

```
Fixpoint norm_ez (e:ExprZ) : ExprZ :=
  match e with
    | EZplus e1 e2 ⇒ EZplus (norm_ez e1) (norm_ez e2)
    | EZminus e1 e2 ⇒ EZminus (norm_ez e1) (norm_ez e2)
    | EZmult e1 e2 ⇒ EZmult (norm_ez e1) (norm_ez e2)
    | EZmax e1 e2 ⇒ EZmax (norm_ez e1) (norm_ez e2)
    | EZopp e ⇒ EZopp (norm_ez e)
    | EZofI e ⇒ norm_ei e
    | EZraw z ⇒ EZraw z
```

```
  end.
```

ExprP to a simplified *ExprP*

```
  Fixpoint norm_ep (e:ExprP) : ExprP :=
    match e with
      | EPeq e1 e2 ⇒ EPeq (norm_ez e1) (norm_ez e2)
      | EPlt e1 e2 ⇒ EPlt (norm_ez e1) (norm_ez e2)
      | EPle e1 e2 ⇒ EPle (norm_ez e1) (norm_ez e2)
      | EPgt e1 e2 ⇒ EPgt (norm_ez e1) (norm_ez e2)
      | EPge e1 e2 ⇒ EPge (norm_ez e1) (norm_ez e2)
      | EPimpl e1 e2 ⇒ EPimpl (norm_ep e1) (norm_ep e2)
      | EPequiv e1 e2 ⇒ EPequiv (norm_ep e1) (norm_ep e2)
      | EPand e1 e2 ⇒ EPand (norm_ep e1) (norm_ep e2)
      | EPor e1 e2 ⇒ EPor (norm_ep e1) (norm_ep e2)
      | EPneg e ⇒ EPneg (norm_ep e)
      | EPraw p ⇒ EPraw p
    end.

  Lemma norm_ei_correct : ∀ e:ExprI, ez2z (norm_ei e) = i2z (ei2i e).

  Lemma norm_ez_correct : ∀ e:ExprZ, ez2z (norm_ez e) = ez2z e.

  Lemma norm_ep_correct :
    ∀ e:ExprP, ep2p (norm_ep e) ↔ ep2p e.

  Lemma norm_ep_correct2 :
    ∀ e:ExprP, ep2p (norm_ep e) → ep2p e.

  Ltac i2z_refl :=
    i2z_gen;
    match goal with ⊢ ?t ⇒
      let e := p2ep t
        in
        (change (ep2p e);
          apply norm_ep_correct2;
            simpl)
    end.

  Ltac iauto := i2z_refl; auto.
  Ltac iomega := i2z_refl; intros; romega.

  Open Scope Z_scope.

  Lemma max_spec : ∀ (x y:Z),
    x ≥ y ∧ Zmax x y = x ∨
    x < y ∧ Zmax x y = y.

  Ltac omega_max_genspec x y :=
    generalize (max_spec x y);
    (let z := fresh "z" in let Hz := fresh "Hz" in
     set (z:=Zmax x y); clearbody z).
```

```
  Ltac omega_max_loop :=
    match goal with
              | ⊢ context [ i2z (?f ?x) ] ⇒
            let i := fresh "i2z" in (set (i:=i2z (f x)); clearbody i); omega_max_loop
      | ⊢ context [ Zmax ?x ?y ] ⇒ omega_max_genspec x y; omega_max_loop
      | _ ⇒ intros
    end.

  Ltac omega_max := i2z_refl; omega_max_loop; try romega.

  Ltac false_omega := i2z_refl; intros; romega.
  Ltac false_omega_max := elimtype False; omega_max.

  Open Scope Int_scope.
End MoreInt.
```

57.3 An implementation of *Int*

It's always nice to know that our *Int* interface is realizable :-)

```
Module Z_as_Int <: Int.
  Open Scope Z_scope.
  Definition int := Z.
  Definition _0 := 0.
  Definition _1 := 1.
  Definition _2 := 2.
  Definition _3 := 3.
  Definition plus := Zplus.
  Definition opp := Zopp.
  Definition minus := Zminus.
  Definition mult := Zmult.
  Definition max := Zmax.
  Definition gt_le_dec := Z_gt_le_dec.
  Definition ge_lt_dec := Z_ge_lt_dec.
  Definition eq_dec := Z_eq_dec.
  Definition i2z : int → Z := fun n ⇒ n.
  Lemma i2z_eq : ∀ n p, i2z n=i2z p → n = p.  Lemma i2z_0 : i2z _0 = 0.  Lemma i2z_1 : i2z
_1 = 1.  Lemma i2z_2 : i2z _2 = 2.  Lemma i2z_3 : i2z _3 = 3.  Lemma i2z_plus : ∀ n p, i2z
(n + p) = i2z n + i2z p.  Lemma i2z_opp : ∀ n, i2z (- n) = - i2z n.  Lemma i2z_minus : ∀ n
p, i2z (n - p) = i2z n - i2z p.  Lemma i2z_mult : ∀ n p, i2z (n × p) = i2z n × i2z p.  Lemma
i2z_max : ∀ n p, i2z (max n p) = Zmax (i2z n) (i2z p). End Z_as_Int.
```

Chapter 58

Module Coq.ZArith.Wf_Z

Require Import *BinInt*.
Require Import *Zcompare*.
Require Import *Zorder*.
Require Import *Znat*.
Require Import *Zmisc*.
Require Import *Wf_nat*.
Open Local *Scope Z_scope*.

Our purpose is to write an induction shema for {0,1,2,...} similar to the *nat* schema (Theorem *Natlike_rec*). For that the following implications will be used :

```
(n:nat)(Q n)==(n:nat)(P (inject_nat n)) ===> (x:Z)'x > 0) -> (P x)

            /\
            ||
            ||

 (Q 0) (n:nat)(Q n)->(Q (S n)) <=== (P 0) (x:Z) (P x) -> (P (Zs x))

                                <=== (inject_nat (S n))=(Zs (inject_nat n))

                                <=== inject_nat_complete
```

Then the diagram will be closed and the theorem proved.

Lemma *Z_of_nat_complete* :
$\forall\ x{:}Z,\ 0 \leq x \rightarrow \exists\ n : nat,\ x = Z\_of\_nat\ n.$

Lemma *ZL4_inf* : $\forall\ y{:}positive,\ \{h : nat \mid nat\_of\_P\ y = S\ h\}.$

Lemma *Z_of_nat_complete_inf* :
$\forall\ x{:}Z,\ 0 \leq x \rightarrow \{n : nat \mid x = Z\_of\_nat\ n\}.$

Lemma *Z_of_nat_prop* :
$\forall\ P{:}Z \rightarrow$ Prop,
$(\forall\ n{:}nat,\ P\ (Z\_of\_nat\ n)) \rightarrow \forall\ x{:}Z,\ 0 \leq x \rightarrow P\ x.$

Lemma *Z_of_nat_set* :
$\forall\ P{:}Z \to \mathsf{Set},$
$(\forall\ n{:}nat,\ P\ (Z\_of\_nat\ n)) \to \forall\ x{:}Z,\ 0 \le x \to P\ x.$

Lemma *natlike_ind* :
$\forall\ P{:}Z \to \mathsf{Prop},$
$P\ 0 \to$
$(\forall\ x{:}Z,\ 0 \le x \to P\ x \to P\ (Zsucc\ x)) \to \forall\ x{:}Z,\ 0 \le x \to P\ x.$

Lemma *natlike_rec* :
$\forall\ P{:}Z \to \mathsf{Set},$
$P\ 0 \to$
$(\forall\ x{:}Z,\ 0 \le x \to P\ x \to P\ (Zsucc\ x)) \to \forall\ x{:}Z,\ 0 \le x \to P\ x.$

Section *Efficient_Rec.*

natlike_rec2 is the same as *natlike_rec*, but with a different proof, designed to give a better extracted term.

Let $R\ (a\ b{:}Z) := 0 \le a \wedge a < b.$

Let R_wf : $well\_founded\ R.$

Lemma *natlike_rec2* :
$\forall\ P{:}Z \to \mathsf{Type},$
$P\ 0 \to$
$(\forall\ z{:}Z,\ 0 \le z \to P\ z \to P\ (Zsucc\ z)) \to \forall\ z{:}Z,\ 0 \le z \to P\ z.$

A variant of the previous using *Zpred* instead of *Zs*.

Lemma *natlike_rec3* :
$\forall\ P{:}Z \to \mathsf{Type},$
$P\ 0 \to$
$(\forall\ z{:}Z,\ 0 < z \to P\ (Zpred\ z) \to P\ z) \to \forall\ z{:}Z,\ 0 \le z \to P\ z.$

A more general induction principle on non-negative numbers using *Zlt*.

Lemma *Zlt_0_rec* :
$\forall\ P{:}Z \to \mathsf{Type},$
$(\forall\ x{:}Z,\ (\forall\ y{:}Z,\ 0 \le y < x \to P\ y) \to 0 \le x \to P\ x) \to$
$\forall\ x{:}Z,\ 0 \le x \to P\ x.$

Lemma *Zlt_0_ind* :
$\forall\ P{:}Z \to \mathsf{Prop},$
$(\forall\ x{:}Z,\ (\forall\ y{:}Z,\ 0 \le y < x \to P\ y) \to 0 \le x \to P\ x) \to$
$\forall\ x{:}Z,\ 0 \le x \to P\ x.$

Obsolete version of *Zlt* induction principle on non-negative numbers

Lemma *Z_lt_rec* :
$\forall\ P{:}Z \to \mathsf{Type},$
$(\forall\ x{:}Z,\ (\forall\ y{:}Z,\ 0 \le y < x \to P\ y) \to P\ x) \to$
$\forall\ x{:}Z,\ 0 \le x \to P\ x.$

Lemma *Z_lt_induction* :
∀ P:$Z \to$ Prop,
(∀ x:Z, (∀ y:Z, $0 \leq y < x \to P\ y$) $\to P\ x$) $\to$
∀ x:Z, $0 \leq x \to P\ x$.

An even more general induction principle using *Zlt*.

Lemma *Zlt_lower_bound_rec* :
∀ P:$Z \to$ Type, ∀ z:Z,
(∀ x:Z, (∀ y:Z, $z \leq y < x \to P\ y$) $\to z \leq x \to P\ x$) $\to$
∀ x:Z, $z \leq x \to P\ x$.

Lemma *Zlt_lower_bound_ind* :
∀ P:$Z \to$ Prop, ∀ z:Z,
(∀ x:Z, (∀ y:Z, $z \leq y < x \to P\ y$) $\to z \leq x \to P\ x$) $\to$
∀ x:Z, $z \leq x \to P\ x$.

End *Efficient_Rec*.

Chapter 59

Module Coq.ZArith.Zabs

Binary Integers (Pierre Crégut (CNET, Lannion, France)

Require Import *Arith_base*.
Require Import *BinPos*.
Require Import *BinInt*.
Require Import *Zorder*.
Require Import *ZArith_dec*.

Open Local *Scope Z_scope*.

59.1 Properties of absolute value

Lemma *Zabs_eq* : $\forall\ n{:}Z,\ 0 \leq n \rightarrow Zabs\ n = n$.

Lemma *Zabs_non_eq* : $\forall\ n{:}Z,\ n \leq 0 \rightarrow Zabs\ n = -\ n$.

Theorem *Zabs_Zopp* : $\forall\ n{:}Z,\ Zabs\ (-\ n) = Zabs\ n$.

59.2 Proving a property of the absolute value by cases

Lemma *Zabs_ind* :
$\forall\ (P{:}Z \rightarrow \mathsf{Prop})\ (n{:}Z),$
$(n \geq 0 \rightarrow P\ n) \rightarrow (n \leq 0 \rightarrow P\ (-\ n)) \rightarrow P\ (Zabs\ n)$.

Theorem *Zabs_intro* : $\forall\ P\ (n{:}Z),\ P\ (-\ n) \rightarrow P\ n \rightarrow P\ (Zabs\ n)$.

Definition *Zabs_dec* : $\forall\ x{:}Z,\ \{x = Zabs\ x\} + \{x = -\ Zabs\ x\}$.

Lemma *Zabs_pos* : $\forall\ n{:}Z,\ 0 \leq Zabs\ n$.

Theorem *Zabs_eq_case* : $\forall\ n\ m{:}Z,\ Zabs\ n = Zabs\ m \rightarrow n = m \vee n = -\ m$.

59.3 Triangular inequality

Hint Local *Resolve Zle_neg_pos*: *zarith*.

Theorem *Zabs_triangle* : $\forall\ n\ m{:}Z,\ Zabs\ (n + m) \leq Zabs\ n + Zabs\ m$.

59.4 Absolute value and multiplication

Lemma *Zsgn_Zabs* : $\forall\ n{:}Z,\ n \times Zsgn\ n = Zabs\ n$.

Lemma *Zabs_Zsgn* : $\forall\ n{:}Z,\ Zabs\ n \times Zsgn\ n = n$.

Theorem *Zabs_Zmult* : $\forall\ n\ m{:}Z,\ Zabs\ (n \times m) = Zabs\ n \times Zabs\ m$.

59.5 Absolute value in nat is compatible with order

Lemma *Zabs_nat_lt* :
$\forall\ n\ m{:}Z,\ 0 \leq n \wedge n < m \rightarrow (Zabs\_nat\ n < Zabs\_nat\ m)\%nat$.

Chapter 60

Module Coq.ZArith.ZArith_base

Library for manipulating integers based on binary encoding. These are the basic modules, required by *Omega* and *Ring* for instance. The full library is *ZArith*.

Require Export *BinPos*.
Require Export *BinNat*.
Require Export *BinInt*.
Require Export *Zcompare*.
Require Export *Zorder*.
Require Export *Zeven*.
Require Export *Zmin*.
Require Export *Zmax*.
Require Export *Zminmax*.
Require Export *Zabs*.
Require Export *Znat*.
Require Export *auxiliary*.
Require Export *ZArith_dec*.
Require Export *Zbool*.
Require Export *Zmisc*.
Require Export *Wf_Z*.

Hint *Resolve Zle_refl Zplus_comm Zplus_assoc Zmult_comm Zmult_assoc Zplus_0_l Zplus_0_r Zmult_1_l Zplus_opp_l Zplus_opp_r Zmult_plus_distr_l Zmult_plus_distr_r*: *zarith*.

Require Export *Zhints*.

Chapter 61

Module Coq.ZArith.ZArith_dec

Require Import *Sumbool*.

Require Import *BinInt*.
Require Import *Zorder*.
Require Import *Zcompare*.
Open Local *Scope Z_scope*.

Lemma *Dcompare_inf* : $\forall\ r$:*comparison*, $\{r = Eq\} + \{r = Lt\} + \{r = Gt\}$.

Lemma *Zcompare_rec* :
$\forall\ (P$:Set$)\ (n\ m$:$Z)$,
$((n\ ?=\ m) = Eq \to P) \to ((n\ ?=\ m) = Lt \to P) \to ((n\ ?=\ m) = Gt \to P) \to P$.

Section *decidability*.

Variables $x\ y : Z$.

61.1 Decidability of equality on binary integers

Definition $Z\_eq\_dec : \{x = y\} + \{x \neq y\}$.

61.2 Decidability of order on binary integers

Definition $Z\_lt\_dec : \{x < y\} + \{\tilde{}\ x < y\}$.

Definition $Z\_le\_dec : \{x \leq y\} + \{\tilde{}\ x \leq y\}$.

Definition $Z\_gt\_dec : \{x > y\} + \{\tilde{}\ x > y\}$.

Definition $Z\_ge\_dec : \{x \geq y\} + \{\tilde{}\ x \geq y\}$.

Definition $Z\_lt\_ge\_dec : \{x < y\} + \{x \geq y\}$.

Lemma $Z\_lt\_le\_dec : \{x < y\} + \{y \leq x\}$.

Definition $Z\_le\_gt\_dec : \{x \leq y\} + \{x > y\}$.

Definition $Z\_gt\_le\_dec : \{x > y\} + \{x \leq y\}$.

Definition $Z\_ge\_lt\_dec : \{x \geq y\} + \{x < y\}$.

Definition $Z\_le\_lt\_eq\_dec : x \leq y \rightarrow \{x < y\} + \{x = y\}$.

End *decidability*.

61.3 Cotransitivity of order on binary integers

Lemma $Zlt\_cotrans : \forall\ n\ m{:}Z, n < m \rightarrow \forall\ p{:}Z, \{n < p\} + \{p < m\}$.

Lemma $Zlt\_cotrans\_pos : \forall\ n\ m{:}Z, 0 < n + m \rightarrow \{0 < n\} + \{0 < m\}$.

Lemma $Zlt\_cotrans\_neg : \forall\ n\ m{:}Z, n + m < 0 \rightarrow \{n < 0\} + \{m < 0\}$.

Lemma $not\_Zeq\_inf : \forall\ n\ m{:}Z, n \neq m \rightarrow \{n < m\} + \{m < n\}$.

Lemma $Z\_dec : \forall\ n\ m{:}Z, \{n < m\} + \{n > m\} + \{n = m\}$.

Lemma $Z\_dec' : \forall\ n\ m{:}Z, \{n < m\} + \{m < n\} + \{n = m\}$.

Definition $Z\_zerop : \forall\ x{:}Z, \{x = 0\} + \{x \neq 0\}$.

Definition $Z\_notzerop\ (x{:}Z) := sumbool\_not\ \_\ \_\ (Z\_zerop\ x)$.

Definition $Z\_noteq\_dec\ (x\ y{:}Z) := sumbool\_not\ \_\ \_\ (Z\_eq\_dec\ x\ y)$.

Chapter 62

Module Coq.ZArith.ZArith

Library for manipulating integers based on binary encoding

Require Export *ZArith_base*.

Extra modules using *Omega* or *Ring*.

Require Export *Zcomplements*.
Require Export *Zsqrt*.
Require Export *Zpower*.
Require Export *Zdiv*.
Require Export *Zlogarithm*.

Export *ZArithRing*.

Chapter 63

Module Coq.ZArith.Zbinary

Bit vectors interpreted as integers. Contribution by Jean Duprat (ENS Lyon).

Require Import *Bvector*.
Require Import *ZArith*.
Require Export *Zpower*.
Require Import *Omega*.

L'évaluation des vecteurs de booléens se font à la fois en binaire et en complément à deux. Le nombre appartient à Z. On utilise donc Omega pour faire les calculs dans Z. De plus, on utilise les fonctions 2^n où n est un naturel, ici la longueur. two_power_nat = *n:nat*(POS (shift_nat n xH)) : nat->Z two_power_nat_S : (n:nat)‘(two_power_nat (S n)) = 2*(two_power_nat n)‘ Z_lt_ge_dec : (x,y:Z){‘x < y‘}+{‘x >= y‘}

Section *VALUE_OF_BOOLEAN_VECTORS*.

Les calculs sont effectués dans la convention positive usuelle. Les valeurs correspondent soit à l'écriture binaire (nat), soit au complément à deux (int). On effectue le calcul suivant le schéma de Horner. Le complément à deux n'a de sens que sur les vecteurs de taille supérieure ou égale à un, le bit de signe étant évalué négativement.

```
Definition bit_value (b:bool) : Z :=
  match b with
    | true ⇒ 1%Z
    | false ⇒ 0%Z
  end.

Lemma binary_value : ∀ n:nat, Bvector n → Z.

Lemma two_compl_value : ∀ n:nat, Bvector (S n) → Z.
```

End *VALUE_OF_BOOLEAN_VECTORS*.

Section *ENCODING_VALUE*.

On calcule la valeur binaire selon un schema de Horner. Le calcul s'arrete à la longueur du vecteur sans vérification. On definit une fonction Zmod2 calquee sur Zdiv2 mais donnant le quotient de la division z=2q+r avec 0<=r<=1. La valeur en complément à deux est calculée selon un schema de Horner avec Zmod2, le paramètre est la taille moins un.

```
Definition Zmod2 (z:Z) :=
  match z with
    | Z0 ⇒ 0%Z
    | Zpos p ⇒ match p with
                 | xI q ⇒ Zpos q
                 | xO q ⇒ Zpos q
                 | xH ⇒ 0%Z
               end
    | Zneg p ⇒
      match p with
        | xI q ⇒ (Zneg q - 1)%Z
        | xO q ⇒ Zneg q
        | xH ⇒ (-1)%Z
      end
  end.

Lemma Zmod2_twice :
  ∀ z:Z, z = (2 × Zmod2 z + bit_value (Zeven.Zodd_bool z))%Z.



Lemma Z_to_binary : ∀ n:nat, Z → Bvector n.

Lemma Z_to_two_compl : ∀ n:nat, Z → Bvector (S n).

End ENCODING_VALUE.

Section Z_BRIC_A_BRAC.
```

Bibliotheque de lemmes utiles dans la section suivante. Utilise largement ZArith. Mériterait d'être récrite.

```
Lemma binary_value_Sn :
  ∀ (n:nat) (b:bool) (bv:Bvector n),
    binary_value (S n) (Vcons bool b n bv) =
    (bit_value b + 2 × binary_value n bv)%Z.

Lemma Z_to_binary_Sn :
  ∀ (n:nat) (b:bool) (z:Z),
    (z ≥ 0)%Z →
    Z_to_binary (S n) (bit_value b + 2 × z) = Bcons b n (Z_to_binary n z).

Lemma binary_value_pos :
  ∀ (n:nat) (bv:Bvector n), (binary_value n bv ≥ 0)%Z.

Lemma two_compl_value_Sn :
  ∀ (n:nat) (bv:Bvector (S n)) (b:bool),
    two_compl_value (S n) (Bcons b (S n) bv) =
    (bit_value b + 2 × two_compl_value n bv)%Z.
```

Lemma *Z_to_two_compl_Sn* :
∀ (*n*:*nat*) (*b*:*bool*) (*z*:*Z*),
Z_to_two_compl (*S n*) (*bit_value b* + 2 × *z*) =
Bcons b (*S n*) (*Z_to_two_compl n z*).

Lemma *Z_to_binary_Sn_z* :
∀ (*n*:*nat*) (*z*:*Z*),
Z_to_binary (*S n*) *z* =
Bcons (*Zeven.Zodd_bool z*) *n* (*Z_to_binary n* (*Zeven.Zdiv2 z*)).

Lemma *Z_div2_value* :
∀ *z*:*Z*,
(*z* ≥ 0)%*Z* → (*bit_value* (*Zeven.Zodd_bool z*) + 2 × *Zeven.Zdiv2 z*)%*Z* = *z*.

Lemma *Pdiv2* : ∀ *z*:*Z*, (*z* ≥ 0)%*Z* → (*Zeven.Zdiv2 z* ≥ 0)%*Z*.

Lemma *Zdiv2_two_power_nat* :
∀ (*z*:*Z*) (*n*:*nat*),
(*z* ≥ 0)%*Z* →
(*z* < *two_power_nat* (*S n*))%*Z* → (*Zeven.Zdiv2 z* < *two_power_nat n*)%*Z*.

Lemma *Z_to_two_compl_Sn_z* :
∀ (*n*:*nat*) (*z*:*Z*),
Z_to_two_compl (*S n*) *z* =
Bcons (*Zeven.Zodd_bool z*) (*S n*) (*Z_to_two_compl n* (*Zmod2 z*)).

Lemma *Zeven_bit_value* :
∀ *z*:*Z*, *Zeven.Zeven z* → *bit_value* (*Zeven.Zodd_bool z*) = 0%*Z*.

Lemma *Zodd_bit_value* :
∀ *z*:*Z*, *Zeven.Zodd z* → *bit_value* (*Zeven.Zodd_bool z*) = 1%*Z*.

Lemma *Zge_minus_two_power_nat_S* :
∀ (*n*:*nat*) (*z*:*Z*),
(*z* ≥ - *two_power_nat* (*S n*))%*Z* → (*Zmod2 z* ≥ - *two_power_nat n*)%*Z*.

Lemma *Zlt_two_power_nat_S* :
∀ (*n*:*nat*) (*z*:*Z*),
(*z* < *two_power_nat* (*S n*))%*Z* → (*Zmod2 z* < *two_power_nat n*)%*Z*.

End *Z_BRIC_A_BRAC*.

Section *COHERENT_VALUE*.

On vérifie que dans l'intervalle de définition les fonctions sont réciproques l'une de l'autre. Elles utilisent les lemmes du bric-a-brac.

Lemma *binary_to_Z_to_binary* :
∀ (*n*:*nat*) (*bv*:*Bvector n*), *Z_to_binary n* (*binary_value n bv*) = *bv*.

Lemma *two_compl_to_Z_to_two_compl* :

$\forall\ (n{:}nat)\ (bv{:}Bvector\ n)\ (b{:}bool),$
$Z\_to\_two\_compl\ n\ (two\_compl\_value\ n\ (Bcons\ b\ n\ bv)) = Bcons\ b\ n\ bv.$

Lemma $Z\_to\_binary\_to\_Z$:
$\forall\ (n{:}nat)\ (z{:}Z),$
$(z \geq 0)\%Z \rightarrow$
$(z < two\_power\_nat\ n)\%Z \rightarrow binary\_value\ n\ (Z\_to\_binary\ n\ z) = z.$

Lemma $Z\_to\_two\_compl\_to\_Z$:
$\forall\ (n{:}nat)\ (z{:}Z),$
$(z \geq -\ two\_power\_nat\ n)\%Z \rightarrow$
$(z < two\_power\_nat\ n)\%Z \rightarrow two\_compl\_value\ n\ (Z\_to\_two\_compl\ n\ z) = z.$

End $COHERENT\_VALUE.$

Chapter 64

Module Coq.ZArith.Zbool

```
Require Import BinInt.
Require Import Zeven.
Require Import Zorder.
Require Import Zcompare.
Require Import ZArith_dec.
Require Import Sumbool.
```

64.1 Boolean operations from decidabilty of order

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

```
Definition Z_lt_ge_bool (x y:Z) := bool_of_sumbool (Z_lt_ge_dec x y).
Definition Z_ge_lt_bool (x y:Z) := bool_of_sumbool (Z_ge_lt_dec x y).

Definition Z_le_gt_bool (x y:Z) := bool_of_sumbool (Z_le_gt_dec x y).
Definition Z_gt_le_bool (x y:Z) := bool_of_sumbool (Z_gt_le_dec x y).

Definition Z_eq_bool (x y:Z) := bool_of_sumbool (Z_eq_dec x y).
Definition Z_noteq_bool (x y:Z) := bool_of_sumbool (Z_noteq_dec x y).

Definition Zeven_odd_bool (x:Z) := bool_of_sumbool (Zeven_odd_dec x).
```

64.2 Boolean comparisons of binary integers

```
Definition Zle_bool (x y:Z) :=
  match (x ?= y)%Z with
    | Gt ⇒ false
    | _ ⇒ true
  end.

Definition Zge_bool (x y:Z) :=
  match (x ?= y)%Z with
```

```
    | Lt ⇒ false
    | _ ⇒ true
  end.

Definition Zlt_bool (x y:Z) :=
  match (x ?= y)%Z with
    | Lt ⇒ true
    | _ ⇒ false
  end.

Definition Zgt_bool (x y:Z) :=
  match (x ?= y)%Z with
    | Gt ⇒ true
    | _ ⇒ false
  end.

Definition Zeq_bool (x y:Z) :=
  match (x ?= y)%Z with
    | Eq ⇒ true
    | _ ⇒ false
  end.

Definition Zneq_bool (x y:Z) :=
  match (x ?= y)%Z with
    | Eq ⇒ false
    | _ ⇒ true
  end.

Lemma Zle_cases :
  ∀ n m:Z, if Zle_bool n m then (n ≤ m)%Z else (n > m)%Z.

Lemma Zlt_cases :
  ∀ n m:Z, if Zlt_bool n m then (n < m)%Z else (n ≥ m)%Z.

Lemma Zge_cases :
  ∀ n m:Z, if Zge_bool n m then (n ≥ m)%Z else (n < m)%Z.

Lemma Zgt_cases :
  ∀ n m:Z, if Zgt_bool n m then (n > m)%Z else (n ≤ m)%Z.
```

Lemmas on *Zle_bool* used in contrib/graphs

```
Lemma Zle_bool_imp_le : ∀ n m:Z, Zle_bool n m = true → (n ≤ m)%Z.

Lemma Zle_imp_le_bool : ∀ n m:Z, (n ≤ m)%Z → Zle_bool n m = true.

Lemma Zle_bool_refl : ∀ n:Z, Zle_bool n n = true.

Lemma Zle_bool_antisym :
  ∀ n m:Z, Zle_bool n m = true → Zle_bool m n = true → n = m.

Lemma Zle_bool_trans :
  ∀ n m p:Z,
    Zle_bool n m = true → Zle_bool m p = true → Zle_bool n p = true.
```

Definition Zle_bool_total :
$\forall\ x\ y{:}Z, \{Zle\_bool\ x\ y = true\} + \{Zle\_bool\ y\ x = true\}$.

Lemma $Zle\_bool\_plus\_mono$:
$\forall\ n\ m\ p\ q{:}Z,$
$Zle\_bool\ n\ m = true \rightarrow$
$Zle\_bool\ p\ q = true \rightarrow Zle\_bool\ (n + p)\ (m + q) = true$.

Lemma $Zone\_pos$: $Zle\_bool\ 1\ 0 = false$.

Lemma $Zone\_min\_pos$: $\forall\ n{:}Z, Zle\_bool\ n\ 0 = false \rightarrow Zle\_bool\ 1\ n = true$.

Lemma $Zle\_is\_le\_bool$: $\forall\ n\ m{:}Z, (n \leq m)\%Z \leftrightarrow Zle\_bool\ n\ m = true$.

Lemma $Zge\_is\_le\_bool$: $\forall\ n\ m{:}Z, (n \geq m)\%Z \leftrightarrow Zle\_bool\ m\ n = true$.

Lemma $Zlt\_is\_le\_bool$:
$\forall\ n\ m{:}Z, (n < m)\%Z \leftrightarrow Zle\_bool\ n\ (m - 1) = true$.

Lemma $Zgt\_is\_le\_bool$:
$\forall\ n\ m{:}Z, (n > m)\%Z \leftrightarrow Zle\_bool\ m\ (n - 1) = true$.

Chapter 65

Module Coq.ZArith.Zcompare

Binary Integers (Pierre Crégut, CNET, Lannion, France)
Require Export *BinPos*.
Require Export *BinInt*.
Require Import *Lt*.
Require Import *Gt*.
Require Import *Plus*.
Require Import *Mult*.

Open Local *Scope Z_scope*.

65.1 Comparison on integers

Lemma *Zcompare_refl* : $\forall\ n{:}Z,\ (n\ ?{=}\ n) = Eq$.

Lemma *Zcompare_Eq_eq* : $\forall\ n\ m{:}Z,\ (n\ ?{=}\ m) = Eq \rightarrow n = m$.

Lemma *Zcompare_Eq_iff_eq* : $\forall\ n\ m{:}Z,\ (n\ ?{=}\ m) = Eq \leftrightarrow n = m$.

Lemma *Zcompare_antisym* : $\forall\ n\ m{:}Z,\ CompOpp\ (n\ ?{=}\ m) = (m\ ?{=}\ n)$.

Lemma *Zcompare_Gt_Lt_antisym* : $\forall\ n\ m{:}Z,\ (n\ ?{=}\ m) = Gt \leftrightarrow (m\ ?{=}\ n) = Lt$.

65.2 Transitivity of comparison

Lemma *Zcompare_Gt_trans* :
$\forall\ n\ m\ p{:}Z,\ (n\ ?{=}\ m) = Gt \rightarrow (m\ ?{=}\ p) = Gt \rightarrow (n\ ?{=}\ p) = Gt$.

65.3 Comparison and opposite

Lemma *Zcompare_opp* : $\forall\ n\ m{:}Z,\ (n\ ?{=}\ m) = (-\ m\ ?{=}\ -\ n)$.

Hint Local *Resolve Pcompare_refl*.

65.4 Comparison first-order specification

Lemma *Zcompare_Gt_spec* :
$\forall\ n\ m{:}Z, (n\ ?{=}\ m) = Gt \rightarrow \exists\ h : positive, n + \text{-}\ m = Zpos\ h.$

65.5 Comparison and addition

Lemma *weaken_Zcompare_Zplus_compatible* :
$(\forall\ (n\ m{:}Z)\ (p{:}positive), (Zpos\ p + n\ ?{=}\ Zpos\ p + m) = (n\ ?{=}\ m)) \rightarrow$
$\forall\ n\ m\ p{:}Z, (p + n\ ?{=}\ p + m) = (n\ ?{=}\ m).$

Hint Local *Resolve ZC4*.

Lemma *weak_Zcompare_Zplus_compatible* :
$\forall\ (n\ m{:}Z)\ (p{:}positive), (Zpos\ p + n\ ?{=}\ Zpos\ p + m) = (n\ ?{=}\ m).$

Lemma *Zcompare_plus_compat* : $\forall\ n\ m\ p{:}Z, (p + n\ ?{=}\ p + m) = (n\ ?{=}\ m).$

Lemma *Zplus_compare_compat* :
$\forall\ (r{:}comparison)\ (n\ m\ p\ q{:}Z),$
$(n\ ?{=}\ m) = r \rightarrow (p\ ?{=}\ q) = r \rightarrow (n + p\ ?{=}\ m + q) = r.$

Lemma *Zcompare_succ_Gt* : $\forall\ n{:}Z, (Zsucc\ n\ ?{=}\ n) = Gt.$

Lemma *Zcompare_Gt_not_Lt* : $\forall\ n\ m{:}Z, (n\ ?{=}\ m) = Gt \leftrightarrow (n\ ?{=}\ m + 1) \neq Lt.$

65.6 Successor and comparison

Lemma *Zcompare_succ_compat* : $\forall\ n\ m{:}Z, (Zsucc\ n\ ?{=}\ Zsucc\ m) = (n\ ?{=}\ m).$

65.7 Multiplication and comparison

Lemma *Zcompare_mult_compat* :
$\forall\ (p{:}positive)\ (n\ m{:}Z), (Zpos\ p \times n\ ?{=}\ Zpos\ p \times m) = (n\ ?{=}\ m).$

65.8 Reverting x ?= y to trichotomy

Lemma *rename* :
$\forall\ (A{:}\mathsf{Type})\ (P{:}A \rightarrow \mathsf{Prop})\ (x{:}A), (\forall\ y{:}A, x = y \rightarrow P\ y) \rightarrow P\ x.$

Lemma *Zcompare_elim* :
$\forall\ (c1\ c2\ c3{:}\mathsf{Prop})\ (n\ m{:}Z),$
$(n = m \rightarrow c1) \rightarrow$
$(n < m \rightarrow c2) \rightarrow$
$(n > m \rightarrow c3) \rightarrow \mathsf{match}\ n\ ?{=}\ m\ \mathsf{with}$

```
                    | Eq ⇒ c1
                    | Lt ⇒ c2
                    | Gt ⇒ c3
                  end.

Lemma Zcompare_eq_case :
  ∀ (c1 c2 c3:Prop) (n m:Z),
    c1 → n = m → match n ?= m with
                    | Eq ⇒ c1
                    | Lt ⇒ c2
                    | Gt ⇒ c3
                  end.
```

65.9 Decompose an egality between two ?= relations into 3 implications

```
Lemma Zcompare_egal_dec :
  ∀ n m p q:Z,
    (n < m → p < q) →
    ((n ?= m) = Eq → (p ?= q) = Eq) →
    (n > m → p > q) → (n ?= m) = (p ?= q).
```

65.10 Relating *x* ?= *y* to *Zle*, *Zlt*, *Zge* or *Zgt*

```
Lemma Zle_compare :
  ∀ n m:Z,
    n ≤ m → match n ?= m with
              | Eq ⇒ True
              | Lt ⇒ True
              | Gt ⇒ False
            end.

Lemma Zlt_compare :
  ∀ n m:Z,
   n < m → match n ?= m with
             | Eq ⇒ False
             | Lt ⇒ True
             | Gt ⇒ False
           end.

Lemma Zge_compare :
  ∀ n m:Z,
    n ≥ m → match n ?= m with
              | Eq ⇒ True
              | Lt ⇒ False
```

```
                | Gt ⇒ True
              end.

Lemma Zgt_compare :
  ∀ n m:Z,
    n > m → match n ?= m with
              | Eq ⇒ False
              | Lt ⇒ False
              | Gt ⇒ True
            end.
```

65.11 Other properties

Lemma *Zmult_compare_compat_l* :
$\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow (n\ ?{=}\ m) = (p \times n\ ?{=}\ p \times m)$.

Lemma *Zmult_compare_compat_r* :
$\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow (n\ ?{=}\ m) = (n \times p\ ?{=}\ m \times p)$.

Chapter 66

Module Coq.ZArith.Zcomplements

```
Require Import ZArithRing.
Require Import ZArith_base.
Require Import Omega.
Require Import Wf_nat.
Open Local Scope Z_scope.
```

About parity

```
Lemma two_or_two_plus_one :
  ∀ n:Z, {y : Z | n = 2 × y} + {y : Z | n = 2 × y + 1}.
```

The biggest power of 2 that is stricly less than a

Easy to compute: replace all "1" of the binary representation by "0", except the first "1" (or the first one :-)

```
Fixpoint floor_pos (a:positive) : positive :=
  match a with
    | xH ⇒ 1%positive
    | xO a' ⇒ xO (floor_pos a')
    | xI b' ⇒ xO (floor_pos b')
  end.

Definition floor (a:positive) := Zpos (floor_pos a).

Lemma floor_gt0 : ∀ p:positive, floor p > 0.

Lemma floor_ok : ∀ p:positive, floor p ≤ Zpos p < 2 × floor p.
```

Two more induction principles over Z.

```
Theorem Z_lt_abs_rec :
  ∀ P:Z → Set,
    (∀ n:Z, (∀ m:Z, Zabs m < Zabs n → P m) → P n) →
```

$\forall\ n{:}Z,\ P\ n.$

Theorem *Z_lt_abs_induction* :
$\forall\ P{:}Z \to \mathsf{Prop},$
$(\forall\ n{:}Z, (\forall\ m{:}Z, Zabs\ m < Zabs\ n \to P\ m) \to P\ n) \to$
$\forall\ n{:}Z,\ P\ n.$

To do case analysis over the sign of z

Lemma *Zcase_sign* :
$\forall\ (n{:}Z)\ (P{:}\mathsf{Prop}), (n = 0 \to P) \to (n > 0 \to P) \to (n < 0 \to P) \to P.$

Lemma *sqr_pos* : $\forall\ n{:}Z,\ n \times n \geq 0.$

A list length in Z, tail recursive.

Require Import *List*.

Fixpoint *Zlength_aux* $(acc{:}Z)\ (A{:}\mathsf{Set})\ (l{:}list\ A)\ \{struct\ l\}$: $Z :=$
match l with
| $nil \Rightarrow acc$
| $\_ :: l \Rightarrow Zlength\_aux\ (Zsucc\ acc)\ A\ l$
end.

Definition *Zlength* := *Zlength_aux* 0.
Implicit *Arguments Zlength* [A].

Section *Zlength_properties*.

Variable A : Set.

Implicit Type l : $list\ A$.

Lemma *Zlength_correct* : $\forall\ l,\ Zlength\ l = Z\_of\_nat\ (length\ l).$

Lemma *Zlength_nil* : $Zlength\ (A{:=}A)\ nil = 0.$

Lemma *Zlength_cons* : $\forall\ (x{:}A)\ l,\ Zlength\ (x :: l) = Zsucc\ (Zlength\ l).$

Lemma *Zlength_nil_inv* : $\forall\ l,\ Zlength\ l = 0 \to l = nil.$

End *Zlength_properties*.

Implicit *Arguments Zlength_correct* [A].
Implicit *Arguments Zlength_cons* [A].
Implicit *Arguments Zlength_nil_inv* [A].

Chapter 67

Module Coq.ZArith.Zdiv

Euclidean Division

Defines first of function that allows Coq to normalize. Then only after proves the main required property.

Require Export *ZArith_base*.
Require Import *Zbool*.
Require Import *Omega*.
Require Import *ZArithRing*.
Require Import *Zcomplements*.
Open Local *Scope Z_scope*.

67.1 Definitions of Euclidian operations

Euclidean division of a positive by a integer (that is supposed to be positive).

Total function than returns an arbitrary value when divisor is not positive

Euclidean division of integers.

Total function than returns (0,0) when dividing by 0.

The pseudo-code is:

if b = 0 : (0,0)

if b <> 0 and a = 0 : (0,0)

if b > 0 and a < 0 : let (q,r) = div_eucl_pos (-a) b in if r = 0 then (-q,0) else (-(q+1),b-r)

if b < 0 and a < 0 : let (q,r) = div_eucl (-a) (-b) in (q,-r)

if b < 0 and a > 0 : let (q,r) = div_eucl a (-b) in if r = 0 then (-q,0) else (-(q+1),b+r)

In other word, when b is non-zero, q is chosen to be the greatest integer smaller or equal to a/b. And sgn(r)=sgn(b) and |r| < |b|.

```
Definition Zdiv_eucl (a b:Z) : Z × Z :=
  match a, b with
    | Z0, _ ⇒ (0, 0)
    | _, Z0 ⇒ (0, 0)
    | Zpos a', Zpos _ ⇒ Zdiv_eucl_POS a' b
    | Zneg a', Zpos _ ⇒
      let (q, r) := Zdiv_eucl_POS a' b in
        match r with
          | Z0 ⇒ (- q, 0)
          | _ ⇒ (- (q + 1), b - r)
        end
    | Zneg a', Zneg b' ⇒ let (q, r) := Zdiv_eucl_POS a' (Zpos b') in (q, - r)
    | Zpos a', Zneg b' ⇒
      let (q, r) := Zdiv_eucl_POS a' (Zpos b') in
        match r with
          | Z0 ⇒ (- q, 0)
          | _ ⇒ (- (q + 1), b + r)
        end
  end.
```

Division and modulo are projections of *Zdiv_eucl*

```
Definition Zdiv (a b:Z) : Z := let (q, _) := Zdiv_eucl a b in q.
```

```
Definition Zmod (a b:Z) : Z := let (_, r) := Zdiv_eucl a b in r.
```

Syntax

```
Infix "/" := Zdiv : Z_scope.
Infix "mod" := Zmod (at level 40, no associativity) : Z_scope.
```

67.2 Main division theorem

First a lemma for positive

```
Lemma Z_div_mod_POS :
  ∀ b:Z,
    b > 0 →
    ∀ a:positive,
      let (q, r) := Zdiv_eucl_POS a b in Zpos a = b × q + r ∧ 0 ≤ r < b.
```

```
Theorem Z_div_mod :
  ∀ a b:Z,
    b > 0 → let (q, r) := Zdiv_eucl a b in a = b × q + r ∧ 0 ≤ r < b.
```

Existence theorems

Theorem *Zdiv_eucl_exist* :
  ∀ *b*:*Z*,
    b > 0 →
    ∀ *a*:*Z*, {*qr* : *Z* × *Z* | let (*q*, *r*) := *qr* in *a* = *b* × *q* + *r* ∧ 0 ≤ *r* < *b*}.

Implicit *Arguments Zdiv_eucl_exist.*

Theorem *Zdiv_eucl_extended* :
  ∀ *b*:*Z*,
    b ≠ 0 →
    ∀ *a*:*Z*,
      {*qr* : *Z* × *Z* | let (*q*, *r*) := *qr* in *a* = *b* × *q* + *r* ∧ 0 ≤ *r* < *Zabs b*}.

Implicit *Arguments Zdiv_eucl_extended.*

67.3 Auxiliary lemmas about *Zdiv* and *Zmod*

Lemma *Z_div_mod_eq* : ∀ *a b*:*Z*, *b* > 0 → *a* = *b* × *Zdiv a b* + *Zmod a b*.

Lemma *Z_mod_lt* : ∀ *a b*:*Z*, *b* > 0 → 0 ≤ *Zmod a b* < *b*.

Lemma *Z_div_POS_ge0* :
  ∀ (*b*:*Z*) (*a*:*positive*), let (*q*, _) := *Zdiv_eucl_POS a b* in *q* ≥ 0.

Lemma *Z_div_ge0* : ∀ *a b*:*Z*, *b* > 0 → *a* ≥ 0 → *Zdiv a b* ≥ 0.

Lemma *Z_div_lt* : ∀ *a b*:*Z*, *b* ≥ 2 → *a* > 0 → *Zdiv a b* < *a*.

67.4 Other lemmas (now using the syntax for *Zdiv* and *Zmod*).

Lemma *Z_div_ge* : ∀ *a b c*:*Z*, *c* > 0 → *a* ≥ *b* → *a* / *c* ≥ *b* / *c*.

Lemma *Z_mod_plus* : ∀ *a b c*:*Z*, *c* > 0 → (*a* + *b* × *c*) *mod c* = *a mod c*.

Lemma *Z_div_plus* : ∀ *a b c*:*Z*, *c* > 0 → (*a* + *b* × *c*) / *c* = *a* / *c* + *b*.

Lemma *Z_div_mult* : ∀ *a b*:*Z*, *b* > 0 → *a* × *b* / *b* = *a*.

Lemma *Z_mult_div_ge* : ∀ *a b*:*Z*, *b* > 0 → *b* × (*a* / *b*) ≤ *a*.

Lemma *Z_mod_same* : ∀ *a*:*Z*, *a* > 0 → *a mod a* = 0.

Lemma *Z_div_same* : ∀ *a*:*Z*, *a* > 0 → *a* / *a* = 1.

Lemma *Z_div_exact_1* : ∀ *a b*:*Z*, *b* > 0 → *a* = *b* × (*a* / *b*) → *a mod b* = 0.

Lemma *Z_div_exact_2* : ∀ *a b*:*Z*, *b* > 0 → *a mod b* = 0 → *a* = *b* × (*a* / *b*).

Lemma *Z_mod_zero_opp* : ∀ *a b*:*Z*, *b* > 0 → *a mod b* = 0 → - *a mod b* = 0.

Chapter 68

Module Coq.ZArith.Zeven

Require Import *BinInt*.

About parity: even and odd predicates on Z, division by 2 on Z

68.1 *Zeven*, *Zodd* and their related properties

```
Definition Zeven (z:Z) :=
  match z with
    | Z0 ⇒ True
    | Zpos (xO _) ⇒ True
    | Zneg (xO _) ⇒ True
    | _ ⇒ False
  end.
Definition Zodd (z:Z) :=
  match z with
    | Zpos xH ⇒ True
    | Zneg xH ⇒ True
    | Zpos (xI _) ⇒ True
    | Zneg (xI _) ⇒ True
    | _ ⇒ False
  end.
Definition Zeven_bool (z:Z) :=
  match z with
    | Z0 ⇒ true
    | Zpos (xO _) ⇒ true
    | Zneg (xO _) ⇒ true
    | _ ⇒ false
  end.
Definition Zodd_bool (z:Z) :=
  match z with
    | Z0 ⇒ false
```

```
    | Zpos (xO _) ⇒ false
    | Zneg (xO _) ⇒ false
    | _ ⇒ true
  end.
```

Definition *Zeven_odd_dec* : ∀ *z*:*Z*, {*Zeven z*} + {*Zodd z*}.

Definition *Zeven_dec* : ∀ *z*:*Z*, {*Zeven z*} + {~ *Zeven z*}.

Definition *Zodd_dec* : ∀ *z*:*Z*, {*Zodd z*} + {~ *Zodd z*}.

Lemma *Zeven_not_Zodd* : ∀ *n*:*Z*, *Zeven n* → ¬ *Zodd n*.

Lemma *Zodd_not_Zeven* : ∀ *n*:*Z*, *Zodd n* → ¬ *Zeven n*.

Lemma *Zeven_Sn* : ∀ *n*:*Z*, *Zodd n* → *Zeven* (*Zsucc n*).

Lemma *Zodd_Sn* : ∀ *n*:*Z*, *Zeven n* → *Zodd* (*Zsucc n*).

Lemma *Zeven_pred* : ∀ *n*:*Z*, *Zodd n* → *Zeven* (*Zpred n*).

Lemma *Zodd_pred* : ∀ *n*:*Z*, *Zeven n* → *Zodd* (*Zpred n*).

Hint *Unfold Zeven Zodd*: *zarith*.

68.2 Definition of *Zdiv2* and properties wrt *Zeven* and *Zodd*

Zdiv2 is defined on all *Z*, but notice that for odd negative integers it is not the euclidean quotient: in that case we have *n* = 2*(*n*/2)-1

```
Definition Zdiv2 (z:Z) :=
  match z with
    | Z0 ⇒ 0%Z
    | Zpos xH ⇒ 0%Z
    | Zpos p ⇒ Zpos (Pdiv2 p)
    | Zneg xH ⇒ 0%Z
    | Zneg p ⇒ Zneg (Pdiv2 p)
  end.
```

Lemma *Zeven_div2* : ∀ *n*:*Z*, *Zeven n* → *n* = (2 × *Zdiv2 n*)%*Z*.

Lemma *Zodd_div2* : ∀ *n*:*Z*, (*n* ≥ 0)%*Z* → *Zodd n* → *n* = (2 × *Zdiv2 n* + 1)%*Z*.

Lemma *Zodd_div2_neg* :
∀ *n*:*Z*, (*n* ≤ 0)%*Z* → *Zodd n* → *n* = (2 × *Zdiv2 n* - 1)%*Z*.

Lemma *Z_modulo_2* :
∀ *n*:*Z*, {*y* : *Z* | *n* = (2 × *y*)%*Z*} + {*y* : *Z* | *n* = (2 × *y* + 1)%*Z*}.

Lemma *Zsplit2* :
∀ *n*:*Z*,
{*p* : *Z* × *Z* |
let (*x1*, *x2*) := *p* in *n* = (*x1* + *x2*)%*Z* ∧ (*x1* = *x2* ∨ *x2* = (*x1* + 1)%*Z*)}.

Chapter 69

Module Coq.ZArith.Zhints

This file centralizes the lemmas about Z, classifying them according to the way they can be used in automatic search

Lemmas which clearly leads to simplification during proof search are
declared as Hints. A definite status (Hint or not) for the other lemmas
remains to be given

Structure of the file

- simplification lemmas (only those are declared as Hints)
- reversible lemmas relating operators
- useful Bottom-up lemmas
- irreversible lemmas with meta-variables
- unclear or too specific lemmas
- lemmas to be used as rewrite rules

Lemmas involving positive and compare are not taken into account

Require Import *BinInt*.
Require Import *Zorder*.
Require Import *Zmin*.
Require Import *Zabs*.
Require Import *Zcompare*.
Require Import *Znat*.
Require Import *auxiliary*.
Require Import *Zmisc*.
Require Import *Wf_Z*.

69.1 Simplification lemmas

No subgoal or smaller subgoals

Hint *Resolve*
(** ** *Reversible simplification lemmas (no loss of information)* *)
(** *Should clearly be declared* as *hints* *)

(** *Lemmas ending by eq* *)
Zsucc_eq_compat
(** *Lemmas ending by Zgt* *)
Zsucc_gt_compat *Zgt_succ* *Zorder.Zgt_pos_0* *Zplus_gt_compat_l* *Zplus_gt_compat_r*
(** *Lemmas ending by Zlt* *)
Zlt_succ *Zsucc_lt_compat* *Zlt_pred* *Zplus_lt_compat_l* *Zplus_lt_compat_r*
(** *Lemmas ending by Zle* *)
Zle_0_nat *Zorder.Zle_0_pos* *Zle_refl* *Zle_succ* *Zsucc_le_compat* *Zle_pred* *Zle_min_l* *Zle_min_r* *Zplus_le_compat_l* *Zplus_le_compat_r* *Zabs_pos*
(** ** *Irreversible simplification lemmas* *)
(** *Probably to be declared* as *hints, when no other simplification is possible* *)

(** *Lemmas ending by eq* *)
BinInt.Z_eq_mult *Zplus_eq_compat*
(** *Lemmas ending by Zge* *)
Zorder.Zmult_ge_compat_r *Zorder.Zmult_ge_compat_l* *Zorder.Zmult_ge_compat*
(** *Lemmas ending by Zlt* *)
Zorder.Zmult_gt_0_compat *Zlt_lt_succ*
(** *Lemmas ending by Zle* *)
Zorder.Zmult_le_0_compat *Zorder.Zmult_le_compat_r* *Zorder.Zmult_le_compat_l* *Zplus_le_0_compat* *Zle_le_succ* *Zplus_le_compat*
: *zarith.*

69.2 Reversible lemmas relating operators

Probably to be declared as hints but need to define precedences

69.2.1 Conversion between comparisons/predicates and arithmetic operators

Lemmas ending by eq

```
Zegal_left: (x,y:Z)`x = y`->`x+(-y) = 0`
Zabs_eq: (x:Z)`0 <= x`->`|x| = x`
Zeven_div2: (x:Z)(Zeven x)->`x = 2*(Zdiv2 x)`
Zodd_div2: (x:Z)`x >= 0`->(Zodd x)->`x = 2*(Zdiv2 x)+1`
```

Lemmas ending by Zgt

```
Zgt_left_rev: (x,y:Z)`x+(-y) > 0`->`x > y`
Zgt_left_gt: (x,y:Z)`x > y`->`x+(-y) > 0`
```

Lemmas ending by Zlt

```
Zlt_left_rev: (x,y:Z)`0 < y+(-x)`->`x < y`
Zlt_left_lt: (x,y:Z)`x < y`->`0 < y+(-x)`
Zlt_0_minus_lt: (n,m:Z)`0 < n-m`->`m < n`
```

Lemmas ending by Zle

```
Zle_left: (x,y:Z)`x <= y`->`0 <= y+(-x)`
Zle_left_rev: (x,y:Z)`0 <= y+(-x)`->`x <= y`
Zlt_left: (x,y:Z)`x < y`->`0 <= y+(-1)+(-x)`
Zge_left: (x,y:Z)`x >= y`->`0 <= x+(-y)`
Zgt_left: (x,y:Z)`x > y`->`0 <= x+(-1)+(-y)`
```

69.2.2 Conversion between nat comparisons and Z comparisons

Lemmas ending by eq

```
inj_eq: (x,y:nat)x=y->`(inject_nat x) = (inject_nat y)`
```

Lemmas ending by Zge

```
inj_ge: (x,y:nat)(ge x y)->`(inject_nat x) >= (inject_nat y)`
```

Lemmas ending by Zgt

```
inj_gt: (x,y:nat)(gt x y)->`(inject_nat x) > (inject_nat y)`
```

Lemmas ending by Zlt

```
inj_lt: (x,y:nat)(lt x y)->`(inject_nat x) < (inject_nat y)`
```

Lemmas ending by Zle

```
inj_le: (x,y:nat)(le x y)->`(inject_nat x) <= (inject_nat y)`
```

69.2.3 Conversion between comparisons

Lemmas ending by Zge

```
not_Zlt: (x,y:Z)~`x < y`->`x >= y`
Zle_ge: (m,n:Z)`m <= n`->`n >= m`
```

Lemmas ending by Zgt

```
Zle_gt_S: (n,p:Z)`n <= p`->`(Zs p) > n`
not_Zle: (x,y:Z)~`x <= y`->`x > y`
Zlt_gt: (m,n:Z)`m < n`->`n > m`
Zle_S_gt: (n,m:Z)`(Zs n) <= m`->`m > n`
```

Lemmas ending by Zlt

```
not_Zge: (x,y:Z)~`x >= y`->`x < y`
Zgt_lt: (m,n:Z)`m > n`->`n < m`
Zle_lt_n_Sm: (n,m:Z)`n <= m`->`n < (Zs m)`
```

Lemmas ending by Zle

```
Zlt_ZERO_pred_le_ZERO: (x:Z)`0 < x`->`0 <= (Zpred x)`
not_Zgt: (x,y:Z)~`x > y`->`x <= y`
Zgt_le_S: (n,p:Z)`p > n`->`(Zs n) <= p`
Zgt_S_le: (n,p:Z)`(Zs p) > n`->`n <= p`
Zge_le: (m,n:Z)`m >= n`->`n <= m`
Zlt_le_S: (n,p:Z)`n < p`->`(Zs n) <= p`
Zlt_n_Sm_le: (n,m:Z)`n < (Zs m)`->`n <= m`
Zlt_le_weak: (n,m:Z)`n < m`->`n <= m`
Zle_refl: (n,m:Z)`n = m`->`n <= m`
```

69.2.4 Irreversible simplification involving several comparaisons

useful with clear precedences

Lemmas ending by Zlt

```
Zlt_le_reg :(a,b,c,d:Z)`a < b`->`c <= d`->`a+c < b+d`
Zle_lt_reg : (a,b,c,d:Z)`a <= b`->`c < d`->`a+c < b+d`
```

69.2.5 What is decreasing here ?

Lemmas ending by eq

```
Zplus_minus: (n,m,p:Z)‘n = m+p‘->‘p = n-m‘
```

Lemmas ending by Zgt

```
Zgt_pred: (n,p:Z)‘p > (Zs n)‘->‘(Zpred p) > n‘
```

Lemmas ending by Zlt

```
Zlt_pred: (n,p:Z)‘(Zs n) < p‘->‘n < (Zpred p)‘
```

69.3 Useful Bottom-up lemmas

69.3.1 Bottom-up simplification: should be used

Lemmas ending by eq

```
Zeq_add_S: (n,m:Z)‘(Zs n) = (Zs m)‘->‘n = m‘
Zsimpl_plus_l: (n,m,p:Z)‘n+m = n+p‘->‘m = p‘
Zplus_unit_left: (n,m:Z)‘n+0 = m‘->‘n = m‘
Zplus_unit_right: (n,m:Z)‘n = m+0‘->‘n = m‘
```

Lemmas ending by Zgt

```
Zsimpl_gt_plus_l: (n,m,p:Z)‘p+n > p+m‘->‘n > m‘
Zsimpl_gt_plus_r: (n,m,p:Z)‘n+p > m+p‘->‘n > m‘
Zgt_S_n: (n,p:Z)‘(Zs p) > (Zs n)‘->‘p > n‘
```

Lemmas ending by Zlt

```
Zsimpl_lt_plus_l: (n,m,p:Z)‘p+n < p+m‘->‘n < m‘
Zsimpl_lt_plus_r: (n,m,p:Z)‘n+p < m+p‘->‘n < m‘
Zlt_S_n: (n,m:Z)‘(Zs n) < (Zs m)‘->‘n < m‘
```

Lemmas ending by Zle

```
Zsimpl_le_plus_l: (p,n,m:Z)`p+n <= p+m`->`n <= m`
Zsimpl_le_plus_r: (p,n,m:Z)`n+p <= m+p`->`n <= m`
Zle_S_n: (n,m:Z)`(Zs m) <= (Zs n)`->`m <= n` >> *)

(** ** Bottom-up irreversible (syntactic) simplification *)

(** Lemmas ending by Zle *)
(**
<<
Zle_trans_S: (n,m:Z)`(Zs n) <= m`->`n <= m`
```

69.3.2 Other unclearly simplifying lemmas

Lemmas ending by Zeq

```
Zmult_eq: (x,y:Z)`x <> 0`->`y*x = 0`->`y = 0`
```

```
Zmult_gt: (x,y:Z)`x > 0`->`x*y > 0`->`y > 0`
```

```
pZmult_lt: (x,y:Z)`x > 0`->`0 < y*x`->`0 < y`
```

```
Zmult_le: (x,y:Z)`x > 0`->`0 <= y*x`->`0 <= y`
OMEGA1: (x,y:Z)`x = y`->`0 <= x`->`0 <= y`
```

69.4 Irreversible lemmas with meta-variables

To be used by EAuto

Lemmas ending by eq

```
Zle_antisym: (n,m:Z)`n <= m`->`m <= n`->`n = m`
```

Lemmas ending by Zge

```
Zge_trans: (n,m,p:Z)`n >= m`->`m >= p`->`n >= p`
```

Lemmas ending by Zgt

```
Zgt_trans: (n,m,p:Z)`n > m`->`m > p`->`n > p`
Zgt_trans_S: (n,m,p:Z)`(Zs n) > m`->`m > p`->`n > p`
Zle_gt_trans: (n,m,p:Z)`m <= n`->`m > p`->`n > p`
Zgt_le_trans: (n,m,p:Z)`n > m`->`p <= m`->`n > p`
```

Lemmas ending by Zlt

```
Zlt_trans: (n,m,p:Z)`n < m`->`m < p`->`n < p`
Zlt_le_trans: (n,m,p:Z)`n < m`->`m <= p`->`n < p`
Zle_lt_trans: (n,m,p:Z)`n <= m`->`m < p`->`n < p`
```

Lemmas ending by Zle

```
Zle_trans: (n,m,p:Z)`n <= m`->`m <= p`->`n <= p`
```

69.5 Unclear or too specific lemmas

Not to be used ?

69.5.1 Irreversible and too specific (not enough regular)

Lemmas ending by Zle

```
Zle_mult: (x,y:Z)`x > 0`->`0 <= y`->`0 <= y*x`
Zle_mult_approx: (x,y,z:Z)`x > 0`->`z > 0`->`0 <= y`->`0 <= y*x+z`
OMEGA6: (x,y,z:Z)`0 <= x`->`y = 0`->`0 <= x+y*z`
OMEGA7: (x,y,z,t:Z)`z > 0`->`t > 0`->`0 <= x`->`0 <= y`->`0 <= x*z+y*t`
```

69.5.2 Expansion and too specific ?

Lemmas ending by Zge

```
Zge_mult_simpl: (a,b,c:Z)`c > 0`->`a*c >= b*c`->`a >= b`
```

Lemmas ending by Zgt

```
Zgt_mult_simpl: (a,b,c:Z)`c > 0`->`a*c > b*c`->`a > b`
Zgt_square_simpl: (x,y:Z)`x >= 0`->`y >= 0`->`x*x > y*y`->`x > y`
```

Lemmas ending by Zle

```
Zle_mult_simpl: (a,b,c:Z)`c > 0`->`a*c <= b*c`->`a <= b`
Zmult_le_approx: (x,y,z:Z)`x > 0`->`x > z`->`0 <= y*x+z`->`0 <= y`
```

69.5.3 Reversible but too specific ?

Lemmas ending by Zlt

```
Zlt_minus: (n,m:Z)`0 < m`->`n-m < n`
```

69.6 Lemmas to be used as rewrite rules

but can also be used as hints

Left-to-right simplification lemmas (a symbol disappears)

```
Zcompare_n_S: (n,m:Z)(Zcompare (Zs n) (Zs m))=(Zcompare n m)
Zmin_n_n: (n:Z)`(Zmin n n) = n`
Zmult_1_n: (n:Z)`1*n = n`
Zmult_n_1: (n:Z)`n*1 = n`
Zminus_plus: (n,m:Z)`n+m-n = m`
Zle_plus_minus: (n,m:Z)`n+(m-n) = m`
Zopp_Zopp: (x:Z)`(-(-x)) = x`
Zero_left: (x:Z)`0+x = x`
Zero_right: (x:Z)`x+0 = x`
Zplus_inverse_r: (x:Z)`x+(-x) = 0`
Zplus_inverse_l: (x:Z)`(-x)+x = 0`
Zopp_intro: (x,y:Z)`(-x) = (-y)`->`x = y`
Zmult_one: (x:Z)`1*x = x`
Zero_mult_left: (x:Z)`0*x = 0`
Zero_mult_right: (x:Z)`x*0 = 0`
Zmult_Zopp_Zopp: (x,y:Z)`(-x)*(-y) = x*y`
```

Right-to-left simplification lemmas (a symbol disappears)

```
Zpred_Sn: (m:Z)`m = (Zpred (Zs m))`
Zs_pred: (n:Z)`n = (Zs (Zpred n))`
Zplus_n_O: (n:Z)`n = n+0`
Zmult_n_O: (n:Z)`0 = n*0`
Zminus_n_O: (n:Z)`n = n-0`
Zminus_n_n: (n:Z)`0 = n-n`
Zred_factor6: (x:Z)`x = x+0`
Zred_factor0: (x:Z)`x = x*1`
```

Unclear orientation (no symbol disappears)

```
Zplus_n_Sm: (n,m:Z)`(Zs (n+m)) = n+(Zs m)`
Zmult_n_Sm: (n,m:Z)`n*m+n = n*(Zs m)`
Zmin_SS: (n,m:Z)`(Zs (Zmin n m)) = (Zmin (Zs n) (Zs m))`
Zplus_assoc_l: (n,m,p:Z)`n+(m+p) = n+m+p`
Zplus_assoc_r: (n,m,p:Z)`n+m+p = n+(m+p)`
Zplus_permute: (n,m,p:Z)`n+(m+p) = m+(n+p)`
Zplus_Snm_nSm: (n,m:Z)`(Zs n)+m = n+(Zs m)`
Zminus_plus_simpl: (n,m,p:Z)`n-m = p+n-(p+m)`
Zminus_Sn_m: (n,m:Z)`(Zs (n-m)) = (Zs n)-m`
Zmult_plus_distr_l: (n,m,p:Z)`(n+m)*p = n*p+m*p`
Zmult_minus_distr: (n,m,p:Z)`(n-m)*p = n*p-m*p`
Zmult_assoc_r: (n,m,p:Z)`n*m*p = n*(m*p)`
Zmult_assoc_l: (n,m,p:Z)`n*(m*p) = n*m*p`
Zmult_permute: (n,m,p:Z)`n*(m*p) = m*(n*p)`
Zmult_Sm_n: (n,m:Z)`n*m+m = (Zs n)*m`
Zmult_Zplus_distr: (x,y,z:Z)`x*(y+z) = x*y+x*z`
Zmult_plus_distr: (n,m,p:Z)`(n+m)*p = n*p+m*p`
Zopp_Zplus: (x,y:Z)`(-(x+y)) = (-x)+(-y)`
Zplus_sym: (x,y:Z)`x+y = y+x`
Zplus_assoc: (x,y,z:Z)`x+(y+z) = x+y+z`
Zmult_sym: (x,y:Z)`x*y = y*x`
Zmult_assoc: (x,y,z:Z)`x*(y*z) = x*y*z`
Zopp_Zmult: (x,y:Z)`(-x)*y = (-(x*y))`
Zplus_S_n: (x,y:Z)`(Zs x)+y = (Zs (x+y))`
Zopp_one: (x:Z)`(-x) = x*(-1)`
Zopp_Zmult_r: (x,y:Z)`(-(x*y)) = x*(-y)`
Zmult_Zopp_left: (x,y:Z)`(-x)*y = x*(-y)`
Zopp_Zmult_l: (x,y:Z)`(-(x*y)) = (-x)*y`
Zred_factor1: (x:Z)`x+x = x*2`
Zred_factor2: (x,y:Z)`x+x*y = x*(1+y)`
Zred_factor3: (x,y:Z)`x*y+x = x*(1+y)`
Zred_factor4: (x,y,z:Z)`x*y+x*z = x*(y+z)`
Zminus_Zplus_compatible: (x,y,n:Z)`x+n-(y+n) = x-y`
Zmin_plus: (x,y,n:Z)`(Zmin (x+n) (y+n)) = (Zmin x y)+n`
```

nat <-> Z

```
inj_S: (y:nat)`(inject_nat (S y)) = (Zs (inject_nat y))`
inj_plus: (x,y:nat)`(inject_nat (plus x y)) = (inject_nat x)+(inject_nat y)`
inj_mult: (x,y:nat)`(inject_nat (mult x y)) = (inject_nat x)*(inject_nat y)`
inj_minus1:
 (x,y:nat)(le y x)->`(inject_nat (minus x y)) = (inject_nat x)-(inject_nat y)`
inj_minus2: (x,y:nat)(gt y x)->`(inject_nat (minus x y)) = 0`
```

Too specific ?

```
Zred_factor5: (x,y:Z)`x*0+y = y`
```

Chapter 70

Module Coq.ZArith.Zlogarithm

The integer logarithms with base 2.

There are three logarithms, depending on the rounding of the real 2-based logarithm:

- *Log_inf*: $y = (Log\_inf\ x)$ *iff* $2\hat{}y \leq x < 2\hat{}(y+1)$ i.e. *Log_inf* x is the biggest integer that is smaller than *Log* x
- *Log_sup*: $y = (Log\_sup\ x)$ *iff* $2\hat{}(y\text{-}1) < x \leq 2\hat{}y$ i.e. *Log_inf* x is the smallest integer that is bigger than *Log* x
- *Log_nearest*: $y = (Log\_nearest\ x)$ *iff* $2\hat{}(y\text{-}1/2) < x \leq 2\hat{}(y+1/2)$ i.e. *Log_nearest* x is the integer nearest from *Log* x

```
Require Import ZArith_base.
Require Import Omega.
Require Import Zcomplements.
Require Import Zpower.
Open Local Scope Z_scope.
```

Section *Log_pos*.

First we build *log_inf* and *log_sup*

```
  Fixpoint log_inf (p:positive) : Z :=
    match p with
      | xH ⇒ 0        | xO q ⇒ Zsucc (log_inf q)        | xI q ⇒ Zsucc (log_inf q)      end.
  Fixpoint log_sup (p:positive) : Z :=
    match p with
      | xH ⇒ 0        | xO n ⇒ Zsucc (log_sup n)        | xI n ⇒ Zsucc (Zsucc (log_inf n))
end.
```

Hint *Unfold log_inf log_sup*.

Then we give the specifications of *log_inf* and *log_sup* and prove their validity

Hint *Resolve Zle_trans*: *zarith*.

Theorem *log_inf_correct* :
∀ *x*:*positive*,
$0 \leq log\_inf\ x \wedge two\_p\ (log\_inf\ x) \leq Zpos\ x < two\_p\ (Zsucc\ (log\_inf\ x))$.

Definition *log_inf_correct1* (*p*:*positive*) := *proj1* (*log_inf_correct p*).
Definition *log_inf_correct2* (*p*:*positive*) := *proj2* (*log_inf_correct p*).

Opaque log_inf_correct1 log_inf_correct2.

Hint *Resolve log_inf_correct1 log_inf_correct2*: *zarith.*

Lemma *log_sup_correct1* : ∀ *p*:*positive*, $0 \leq log\_sup\ p$.

For every p, either p is a power of two and $(log\_inf\ p)=(log\_sup\ p)$ either $(log\_sup\ p)=(log\_inf\ p)+1$

Theorem *log_sup_log_inf* :
∀ *p*:*positive*,
IF Zpos p = two_p (*log_inf p*) then *Zpos p = two_p* (*log_sup p*)
else $log\_sup\ p = Zsucc\ (log\_inf\ p)$.

Theorem *log_sup_correct2* :
∀ *x*:*positive*, $two\_p\ (Zpred\ (log\_sup\ x)) < Zpos\ x \leq two\_p\ (log\_sup\ x)$.

Lemma *log_inf_le_log_sup* : ∀ *p*:*positive*, $log\_inf\ p \leq log\_sup\ p$.

Lemma *log_sup_le_Slog_inf* : ∀ *p*:*positive*, $log\_sup\ p \leq Zsucc\ (log\_inf\ p)$.

Now it's possible to specify and build the *Log* rounded to the nearest

```
Fixpoint log_near (x:positive) : Z :=
  match x with
    | xH ⇒ 0
    | xO xH ⇒ 1
    | xI xH ⇒ 2
    | xO y ⇒ Zsucc (log_near y)
    | xI y ⇒ Zsucc (log_near y)
  end.
```

Theorem *log_near_correct1* : ∀ *p*:*positive*, $0 \leq log\_near\ p$.

Theorem *log_near_correct2* :
∀ *p*:*positive*, $log\_near\ p = log\_inf\ p \vee log\_near\ p = log\_sup\ p$.

End *Log_pos.*

Section *divers.*

Number of significative digits.

```
Definition N_digits (x:Z) :=
  match x with
    | Zpos p ⇒ log_inf p
    | Zneg p ⇒ log_inf p
    | Z0 ⇒ 0
```

end.

Lemma $ZERO\_le\_N\_digits$: $\forall\ x{:}Z,\ 0 \leq N\_digits\ x$.

Lemma $log\_inf\_shift\_nat$: $\forall\ n{:}nat,\ log\_inf\ (shift\_nat\ n\ 1) = Z\_of\_nat\ n$.

Lemma $log\_sup\_shift\_nat$: $\forall\ n{:}nat,\ log\_sup\ (shift\_nat\ n\ 1) = Z\_of\_nat\ n$.

$Is\_power\ p$ means that p is a power of two

Fixpoint $Is\_power\ (p{:}positive)$: Prop :=
match p with
| $xH \Rightarrow True$
| $xO\ q \Rightarrow Is\_power\ q$
| $xI\ q \Rightarrow False$
end.

Lemma $Is\_power\_correct$:
$\forall\ p{:}positive,\ Is\_power\ p \leftrightarrow (\exists\ y : nat,\ p = shift\_nat\ y\ 1)$.

Lemma Is_power_or : $\forall\ p{:}positive,\ Is\_power\ p \vee \neg\ Is\_power\ p$.

End *divers*.

Chapter 71

Module Coq.ZArith.Zmax

```
Require Import Arith_base.
Require Import BinInt.
Require Import Zcompare.
Require Import Zorder.

Open Local Scope Z_scope.
```

Maximum of two binary integer numbers

```
Definition Zmax m n :=
  match m ?= n with
    | Eq | Gt ⇒ m
    | Lt ⇒ n
  end.
```

71.1 Characterization of maximum on binary integer numbers

```
Lemma Zmax_case : ∀ (n m:Z) (P:Z → Type), P n → P m → P (Zmax n m).

Lemma Zmax_case_strong : ∀ (n m:Z) (P:Z → Type),
  (m≤n → P n) → (n≤m → P m) → P (Zmax n m).
```

71.2 Least upper bound properties of max

```
Lemma Zle_max_l : ∀ n m:Z, n ≤ Zmax n m.

Notation Zmax1 := Zle_max_l (only parsing).

Lemma Zle_max_r : ∀ n m:Z, m ≤ Zmax n m.

Notation Zmax2 := Zle_max_r (only parsing).

Lemma Zmax_lub : ∀ n m p:Z, n ≤ p → m ≤ p → Zmax n m ≤ p.
```

71.3 Semi-lattice properties of max

Lemma $Zmax\_idempotent : \forall\ n{:}Z,\ Zmax\ n\ n = n.$

Lemma $Zmax\_comm : \forall\ n\ m{:}Z,\ Zmax\ n\ m = Zmax\ m\ n.$

Lemma $Zmax\_assoc : \forall\ n\ m\ p{:}Z,\ Zmax\ n\ (Zmax\ m\ p) = Zmax\ (Zmax\ n\ m)\ p.$

71.4 Additional properties of max

Lemma $Zmax\_irreducible\_inf : \forall\ n\ m{:}Z,\ Zmax\ n\ m = n \vee Zmax\ n\ m = m.$

Lemma $Zmax\_le\_prime\_inf : \forall\ n\ m\ p{:}Z,\ p \leq Zmax\ n\ m \rightarrow p \leq n \vee p \leq m.$

71.5 Operations preserving max

Lemma $Zsucc\_max\_distr :$
$\forall\ n\ m{:}Z,\ Zsucc\ (Zmax\ n\ m) = Zmax\ (Zsucc\ n)\ (Zsucc\ m).$

Lemma $Zplus\_max\_distr\_r : \forall\ n\ m\ p{:}Z,\ Zmax\ (n + p)\ (m + p) = Zmax\ n\ m + p.$

Chapter 72

Module Coq.ZArith.Zminmax

Require Import *Zmin Zmax*.
Require Import *BinInt Zorder*.

Open Local *Scope Z_scope*.

Lattice properties of min and max on Z

Absorption

Lemma $Zmin\_max\_absorption\_r\_r : \forall\ n\ m,\ Zmax\ n\ (Zmin\ n\ m) = n.$

Lemma $Zmax\_min\_absorption\_r\_r : \forall\ n\ m,\ Zmin\ n\ (Zmax\ n\ m) = n.$

Distributivity

Lemma $Zmax\_min\_distr\_r$:
$\forall\ n\ m\ p,\ Zmax\ n\ (Zmin\ m\ p) = Zmin\ (Zmax\ n\ m)\ (Zmax\ n\ p).$

Lemma $Zmin\_max\_distr\_r$:
$\forall\ n\ m\ p,\ Zmin\ n\ (Zmax\ m\ p) = Zmax\ (Zmin\ n\ m)\ (Zmin\ n\ p).$

Modularity

Lemma $Zmax\_min\_modular\_r$:
$\forall\ n\ m\ p,\ Zmax\ n\ (Zmin\ m\ (Zmax\ n\ p)) = Zmin\ (Zmax\ n\ m)\ (Zmax\ n\ p).$

Lemma $Zmin\_max\_modular\_r$:
$\forall\ n\ m\ p,\ Zmin\ n\ (Zmax\ m\ (Zmin\ n\ p)) = Zmax\ (Zmin\ n\ m)\ (Zmin\ n\ p).$

Disassociativity

Lemma $max\_min\_disassoc : \forall\ n\ m\ p,\ Zmin\ n\ (Zmax\ m\ p) \leq Zmax\ (Zmin\ n\ m)\ p.$

Chapter 73

Module Coq.ZArith.Zmin

Initial version from Pierre Crégut (CNET, Lannion, France), 1996. Further extensions by the Coq development team, with suggestions from Russell O'Connor (Radbout U., Nijmegen, The Netherlands).

Require Import *Arith_base*.
Require Import *BinInt*.
Require Import *Zcompare*.
Require Import *Zorder*.

Open Local *Scope Z_scope*.

Minimum on binary integer numbers

73.1 Characterization of the minimum on binary integer numbers

Lemma *Zmin_case_strong* : $\forall$ (n m:Z) (P:$Z \to$ Type),
($n \leq m \to P\ n$) $\to$ ($m \leq n \to P\ m$) $\to P\ (Zmin\ n\ m)$.

Lemma *Zmin_case* : $\forall$ (n m:Z) (P:$Z \to$ Type), $P\ n \to P\ m \to P\ (Zmin\ n\ m)$.

73.2 Greatest lower bound properties of min

Lemma *Zle_min_l* : $\forall\ n\ m$:Z, $Zmin\ n\ m \leq n$.

Lemma *Zle_min_r* : $\forall\ n\ m$:Z, $Zmin\ n\ m \leq m$.

Lemma *Zmin_glb* : $\forall\ n\ m\ p$:Z, $p \leq n \to p \leq m \to p \leq Zmin\ n\ m$.

73.3 Semi-lattice properties of min

Lemma *Zmin_idempotent* : $\forall\ n$:Z, $Zmin\ n\ n = n$.

Notation *Zmin_n_n* := *Zmin_idempotent* (*only parsing*).

Lemma $Zmin\_comm : \forall\ n\ m{:}Z,\ Zmin\ n\ m = Zmin\ m\ n.$

Lemma $Zmin\_assoc : \forall\ n\ m\ p{:}Z,\ Zmin\ n\ (Zmin\ m\ p) = Zmin\ (Zmin\ n\ m)\ p.$

73.4 Additional properties of min

Lemma $Zmin\_irreducible\_inf : \forall\ n\ m{:}Z,\ \{Zmin\ n\ m = n\} + \{Zmin\ n\ m = m\}.$

Lemma $Zmin\_irreducible : \forall\ n\ m{:}Z,\ Zmin\ n\ m = n \vee Zmin\ n\ m = m.$

Notation $Zmin\_or := Zmin\_irreducible\ (only\ parsing).$

Lemma $Zmin\_le\_prime\_inf : \forall\ n\ m\ p{:}Z,\ Zmin\ n\ m \leq p \rightarrow \{n \leq p\} + \{m \leq p\}.$

73.5 Operations preserving min

Lemma $Zsucc\_min\_distr :$
$\forall\ n\ m{:}Z,\ Zsucc\ (Zmin\ n\ m) = Zmin\ (Zsucc\ n)\ (Zsucc\ m).$

Notation $Zmin\_SS := Zsucc\_min\_distr\ (only\ parsing).$

Lemma $Zplus\_min\_distr\_r : \forall\ n\ m\ p{:}Z,\ Zmin\ (n + p)\ (m + p) = Zmin\ n\ m + p.$

Notation $Zmin\_plus := Zplus\_min\_distr\_r\ (only\ parsing).$

Chapter 74

Module Coq.ZArith.Zmisc

```
Require Import BinInt.
Require Import Zcompare.
Require Import Zorder.
Require Import Bool.
Open Local Scope Z_scope.
```

Iterators

nth iteration of the function f

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

Fixpoint iter_pos (n:positive) (A:Set) (f:A → A) (x:A) {struct n} : A :=
  match n with
    | xH ⇒ f x
    | xO n' ⇒ iter_pos n' A f (iter_pos n' A f x)
    | xI n' ⇒ f (iter_pos n' A f (iter_pos n' A f x))
  end.

Definition iter (n:Z) (A:Set) (f:A → A) (x:A) :=
  match n with
    | Z0 ⇒ x
    | Zpos p ⇒ iter_pos p A f x
    | Zneg p ⇒ x
  end.

Theorem iter_nat_plus :
  ∀ (n m:nat) (A:Set) (f:A → A) (x:A),
    iter_nat (n + m) A f x = iter_nat n A f (iter_nat m A f x).

Theorem iter_nat_of_P :
  ∀ (p:positive) (A:Set) (f:A → A) (x:A),
    iter_pos p A f x = iter_nat (nat_of_P p) A f x.
```

Theorem *iter_pos_plus* :
∀ (*p q*:*positive*) (*A*:Set) (*f*:*A* → *A*) (*x*:*A*),
iter_pos (*p* + *q*) *A f x* = *iter_pos p A f* (*iter_pos q A f x*).

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

Theorem *iter_nat_invariant* :
∀ (*n*:*nat*) (*A*:Set) (*f*:*A* → *A*) (*Inv*:*A* → Prop),
(∀ *x*:*A*, *Inv x* → *Inv* (*f x*)) →
∀ *x*:*A*, *Inv x* → *Inv* (*iter_nat n A f x*).

Theorem *iter_pos_invariant* :
∀ (*p*:*positive*) (*A*:Set) (*f*:*A* → *A*) (*Inv*:*A* → Prop),
(∀ *x*:*A*, *Inv x* → *Inv* (*f x*)) →
∀ *x*:*A*, *Inv x* → *Inv* (*iter_pos p A f x*).

Chapter 75

Module Coq.ZArith.Znat

Binary Integers (Pierre Crégut, CNET, Lannion, France)

Require Export *Arith_base*.
Require Import *BinPos*.
Require Import *BinInt*.
Require Import *Zcompare*.
Require Import *Zorder*.
Require Import *Decidable*.
Require Import *Peano_dec*.
Require Export *Compare_dec*.

Open Local *Scope Z_scope*.

Definition *neq* $(x\ y{:}nat) := x \neq y$.

Properties of the injection from nat into Z

Theorem *inj_S* : $\forall\ n{:}nat,\ Z\_of\_nat\ (S\ n) = Zsucc\ (Z\_of\_nat\ n)$.

Theorem *inj_plus* : $\forall\ n\ m{:}nat,\ Z\_of\_nat\ (n + m) = Z\_of\_nat\ n + Z\_of\_nat\ m$.

Theorem *inj_mult* : $\forall\ n\ m{:}nat,\ Z\_of\_nat\ (n \times m) = Z\_of\_nat\ n \times Z\_of\_nat\ m$.

Theorem *inj_neq* : $\forall\ n\ m{:}nat,\ neq\ n\ m \rightarrow Zne\ (Z\_of\_nat\ n)\ (Z\_of\_nat\ m)$.

Theorem *inj_le* : $\forall\ n\ m{:}nat,\ (n \leq m)\%nat \rightarrow Z\_of\_nat\ n \leq Z\_of\_nat\ m$.

Theorem *inj_lt* : $\forall\ n\ m{:}nat,\ (n < m)\%nat \rightarrow Z\_of\_nat\ n < Z\_of\_nat\ m$.

Theorem *inj_gt* : $\forall\ n\ m{:}nat,\ (n > m)\%nat \rightarrow Z\_of\_nat\ n > Z\_of\_nat\ m$.

Theorem *inj_ge* : $\forall\ n\ m{:}nat,\ (n \geq m)\%nat \rightarrow Z\_of\_nat\ n \geq Z\_of\_nat\ m$.

Theorem *inj_eq* : $\forall\ n\ m{:}nat,\ n = m \rightarrow Z\_of\_nat\ n = Z\_of\_nat\ m$.

Theorem *intro_Z* :
$\forall\ n{:}nat, \exists\ y : Z,\ Z\_of\_nat\ n = y \wedge 0 \leq y \times 1 + 0$.

Theorem *inj_minus1* :
$\forall\ n\ m{:}nat,\ (m \leq n)\%nat \rightarrow Z\_of\_nat\ (n - m) = Z\_of\_nat\ n - Z\_of\_nat\ m$.

Theorem *inj_minus2* : $\forall\ n\ m{:}nat,\ (m > n)\%nat \rightarrow Z\_of\_nat\ (n - m) = 0$.

Theorem *Zpos_eq_Z_of_nat_o_nat_of_P* :
∀ *p*:*positive*, *Zpos p* = *Z_of_nat* (*nat_of_P p*).

Chapter 76

Module Coq.ZArith.Znumtheory

Require Import *ZArith_base*.
Require Import *ZArithRing*.
Require Import *Zcomplements*.
Require Import *Zdiv*.
Require Import *Ndigits*.
Require Import *Wf_nat*.
Open Local *Scope Z_scope*.

This file contains some notions of number theory upon Z numbers:

- a divisibility predicate *Zdivide*
- a gcd predicate *gcd*
- Euclid algorithm *euclid*
- a relatively prime predicate *rel_prime*
- a prime predicate *prime*
- an efficient *Zgcd* function

76.1 Divisibility

Inductive *Zdivide* (*a b*:*Z*) : Prop :=
Zdivide_intro : ∀ *q*:*Z*, *b* = *q* × *a* → *Zdivide a b*.

Syntax for divisibility

Notation "(*a* | *b*)" := (*Zdivide a b*) (*at level* 0) : *Z_scope*.

Results concerning divisibility

Lemma *Zdivide_refl* : ∀ *a*:*Z*, (*a* | *a*).

Lemma *Zone_divide* : ∀ *a*:*Z*, (1 | *a*).

Lemma *Zdivide_0* : $\forall\ a{:}Z,\ (a \mid 0)$.

Hint *Resolve Zdivide_refl Zone_divide Zdivide_0*: *zarith*.

Lemma *Zmult_divide_compat_l* : $\forall\ a\ b\ c{:}Z,\ (a \mid b) \rightarrow (c \times a \mid c \times b)$.

Lemma *Zmult_divide_compat_r* : $\forall\ a\ b\ c{:}Z,\ (a \mid b) \rightarrow (a \times c \mid b \times c)$.

Hint *Resolve Zmult_divide_compat_l Zmult_divide_compat_r*: *zarith*.

Lemma *Zdivide_plus_r* : $\forall\ a\ b\ c{:}Z,\ (a \mid b) \rightarrow (a \mid c) \rightarrow (a \mid b + c)$.

Lemma *Zdivide_opp_r* : $\forall\ a\ b{:}Z,\ (a \mid b) \rightarrow (a \mid -\ b)$.

Lemma *Zdivide_opp_r_rev* : $\forall\ a\ b{:}Z,\ (a \mid -\ b) \rightarrow (a \mid b)$.

Lemma *Zdivide_opp_l* : $\forall\ a\ b{:}Z,\ (a \mid b) \rightarrow (-\ a \mid b)$.

Lemma *Zdivide_opp_l_rev* : $\forall\ a\ b{:}Z,\ (-\ a \mid b) \rightarrow (a \mid b)$.

Lemma *Zdivide_minus_l* : $\forall\ a\ b\ c{:}Z,\ (a \mid b) \rightarrow (a \mid c) \rightarrow (a \mid b - c)$.

Lemma *Zdivide_mult_l* : $\forall\ a\ b\ c{:}Z,\ (a \mid b) \rightarrow (a \mid b \times c)$.

Lemma *Zdivide_mult_r* : $\forall\ a\ b\ c{:}Z,\ (a \mid c) \rightarrow (a \mid b \times c)$.

Lemma *Zdivide_factor_r* : $\forall\ a\ b{:}Z,\ (a \mid a \times b)$.

Lemma *Zdivide_factor_l* : $\forall\ a\ b{:}Z,\ (a \mid b \times a)$.

Hint *Resolve Zdivide_plus_r Zdivide_opp_r Zdivide_opp_r_rev Zdivide_opp_l Zdivide_opp_l_rev Zdivide_minus_l Zdivide_mult_l Zdivide_mult_r Zdivide_factor_r Zdivide_factor_l*: *zarith*.

Auxiliary result.

Lemma *Zmult_one* : $\forall\ x\ y{:}Z,\ x \geq 0 \rightarrow x \times y = 1 \rightarrow x = 1$.

Only 1 and -1 divide 1.

Lemma *Zdivide_1* : $\forall\ x{:}Z,\ (x \mid 1) \rightarrow x = 1 \vee x = -1$.

If a divides b and b divides a then a is b or $-b$.

Lemma *Zdivide_antisym* : $\forall\ a\ b{:}Z,\ (a \mid b) \rightarrow (b \mid a) \rightarrow a = b \vee a = -\ b$.

If a divides b and $b \neq 0$ then $|a| \leq |b|$.

Lemma *Zdivide_bounds* : $\forall\ a\ b{:}Z,\ (a \mid b) \rightarrow b \neq 0 \rightarrow Zabs\ a \leq Zabs\ b$.

76.2 Greatest common divisor (gcd).

There is no unicity of the gcd; hence we define the predicate $gcd\ a\ b\ d$ expressing that d is a gcd of a and b. (We show later that the *gcd* is actually unique if we discard its sign.)

Inductive *Zis_gcd* $(a\ b\ d{:}Z)$: Prop :=
Zis_gcd_intro :
$(d \mid a) \rightarrow$

$(d \mid b) \rightarrow (\forall\ x{:}Z, (x \mid a) \rightarrow (x \mid b) \rightarrow (x \mid d)) \rightarrow Zis\_gcd\ a\ b\ d.$

Trivial properties of *gcd*

Lemma $Zis\_gcd\_sym : \forall\ a\ b\ d{:}Z, Zis\_gcd\ a\ b\ d \rightarrow Zis\_gcd\ b\ a\ d.$

Lemma $Zis\_gcd\_0 : \forall\ a{:}Z, Zis\_gcd\ a\ 0\ a.$

Lemma $Zis\_gcd\_1 : \forall\ a, Zis\_gcd\ a\ 1\ 1.$

Lemma $Zis\_gcd\_refl : \forall\ a, Zis\_gcd\ a\ a\ a.$

Lemma $Zis\_gcd\_minus : \forall\ a\ b\ d{:}Z, Zis\_gcd\ a\ (-\ b)\ d \rightarrow Zis\_gcd\ b\ a\ d.$

Lemma $Zis\_gcd\_opp : \forall\ a\ b\ d{:}Z, Zis\_gcd\ a\ b\ d \rightarrow Zis\_gcd\ b\ a\ (-\ d).$

Lemma $Zis\_gcd\_0\_abs : \forall\ a{:}Z, Zis\_gcd\ 0\ a\ (Zabs\ a).$

Hint *Resolve Zis_gcd_sym Zis_gcd_0 Zis_gcd_minus Zis_gcd_opp*: *zarith*.

76.3 Extended Euclid algorithm.

Euclid's algorithm to compute the *gcd* mainly relies on the following property.

Lemma *Zis_gcd_for_euclid* :
$\forall\ a\ b\ d\ q{:}Z, Zis\_gcd\ b\ (a\ -\ q \times b)\ d \rightarrow Zis\_gcd\ a\ b\ d.$

Lemma *Zis_gcd_for_euclid2* :
$\forall\ b\ d\ q\ r{:}Z, Zis\_gcd\ r\ b\ d \rightarrow Zis\_gcd\ b\ (b \times q\ +\ r)\ d.$

We implement the extended version of Euclid's algorithm, i.e. the one computing Bezout's coefficients as it computes the *gcd*. We follow the algorithm given in Knuth's "Art of Computer Programming", vol 2, page 325.

Section *extended_euclid_algorithm*.

Variables $a\ b : Z.$

The specification of Euclid's algorithm is the existence of u, v and d such that $ua+vb=d$ and $(gcd\ a\ b\ d)$.

Inductive *Euclid* : Set :=
Euclid_intro :
$\forall\ u\ v\ d{:}Z, u \times a\ +\ v \times b\ =\ d \rightarrow Zis\_gcd\ a\ b\ d \rightarrow Euclid.$

The recursive part of Euclid's algorithm uses well-founded recursion of non-negative integers. It maintains 6 integers $u1$,$u2$,$u3$,$v1$,$v2$,$v3$ such that the following invariant holds: $u1 \times a+u2 \times b=u3$ and $v1 \times a+v2 \times b=v3$ and $gcd(u2,v3)=gcd(a,b)$.

Lemma *euclid_rec* :
$\forall\ v3{:}Z,$
$0 \leq v3 \rightarrow$
$\forall\ u1\ u2\ u3\ v1\ v2{:}Z,$
$u1 \times a\ +\ u2 \times b\ =\ u3 \rightarrow$
$v1 \times a\ +\ v2 \times b\ =\ v3 \rightarrow$

$(\forall\ d{:}Z,\ Zis\_gcd\ u3\ v3\ d \to Zis\_gcd\ a\ b\ d) \to Euclid.$

We get Euclid's algorithm by applying *euclid_rec* on 1,0,a,0,1,b when $b \geq 0$ and 1,0,a,0,-1,-b when $b < 0$.

Lemma *euclid* : *Euclid*.

End *extended_euclid_algorithm*.

Theorem *Zis_gcd_uniqueness_apart_sign* :
$\forall\ a\ b\ d\ d'{:}Z,\ Zis\_gcd\ a\ b\ d \to Zis\_gcd\ a\ b\ d' \to d = d' \vee d = -\ d'$.

76.4 Bezout's coefficients

Inductive *Bezout* $(a\ b\ d{:}Z)$: Prop :=
Bezout_intro : $\forall\ u\ v{:}Z,\ u \times a + v \times b = d \to Bezout\ a\ b\ d$.

Existence of Bezout's coefficients for the *gcd* of a and b

Lemma *Zis_gcd_bezout* : $\forall\ a\ b\ d{:}Z,\ Zis\_gcd\ a\ b\ d \to Bezout\ a\ b\ d$.

gcd of ca and cb is c $gcd(a,b)$.

Lemma *Zis_gcd_mult* :
$\forall\ a\ b\ c\ d{:}Z,\ Zis\_gcd\ a\ b\ d \to Zis\_gcd\ (c \times a)\ (c \times b)\ (c \times d)$.

76.5 Relative primality

Definition *rel_prime* $(a\ b{:}Z)$: Prop := $Zis\_gcd\ a\ b\ 1$.

Bezout's theorem: a and b are relatively prime if and only if there exist u and v such that $ua+vb = 1$.

Lemma *rel_prime_bezout* : $\forall\ a\ b{:}Z,\ rel\_prime\ a\ b \to Bezout\ a\ b\ 1$.

Lemma *bezout_rel_prime* : $\forall\ a\ b{:}Z,\ Bezout\ a\ b\ 1 \to rel\_prime\ a\ b$.

Gauss's theorem: if a divides bc and if a and b are relatively prime, then a divides c.

Theorem *Gauss* : $\forall\ a\ b\ c{:}Z,\ (a \mid b \times c) \to rel\_prime\ a\ b \to (a \mid c)$.

If a is relatively prime to b and c, then it is to bc

Lemma *rel_prime_mult* :
$\forall\ a\ b\ c{:}Z,\ rel\_prime\ a\ b \to rel\_prime\ a\ c \to rel\_prime\ a\ (b \times c)$.

Lemma *rel_prime_cross_prod* :
$\forall\ a\ b\ c\ d{:}Z,$
$rel\_prime\ a\ b \to$
$rel\_prime\ c\ d \to b > 0 \to d > 0 \to a \times d = b \times c \to a = c \wedge b = d$.

After factorization by a gcd, the original numbers are relatively prime.

Lemma *Zis_gcd_rel_prime* :
$\forall\ a\ b\ g{:}Z,$
$b > 0 \to g \geq 0 \to Zis\_gcd\ a\ b\ g \to rel\_prime\ (a\ /\ g)\ (b\ /\ g)$.

76.6 Primality

Inductive *prime* (*p*:*Z*) : Prop :=
  prime_intro :
    $1 < p \rightarrow (\forall\ n{:}Z, 1 \leq n < p \rightarrow rel\_prime\ n\ p) \rightarrow prime\ p$.

The sole divisors of a prime number p are -1, 1, p and -p.

Lemma *prime_divisors* :
  $\forall\ p{:}Z,$
    $prime\ p \rightarrow \forall\ a{:}Z, (a \mid p) \rightarrow a = -1 \vee a = 1 \vee a = p \vee a = -p$.

A prime number is relatively prime with any number it does not divide

Lemma *prime_rel_prime* :
  $\forall\ p{:}Z, prime\ p \rightarrow \forall\ a{:}Z, \neg\ (p \mid a) \rightarrow rel\_prime\ p\ a$.

Hint *Resolve prime_rel_prime*: *zarith*.

Zdivide can be expressed using *Zmod*.

Lemma *Zmod_divide* : $\forall\ a\ b{:}Z, b > 0 \rightarrow a\ mod\ b = 0 \rightarrow (b \mid a)$.

Lemma *Zdivide_mod* : $\forall\ a\ b{:}Z, b > 0 \rightarrow (b \mid a) \rightarrow a\ mod\ b = 0$.

Zdivide is hence decidable

Lemma *Zdivide_dec* : $\forall\ a\ b{:}Z, \{(a \mid b)\} + \{\tilde{}\ (a \mid b)\}$.

If a prime p divides ab then it divides either a or b

Lemma *prime_mult* :
  $\forall\ p{:}Z, prime\ p \rightarrow \forall\ a\ b{:}Z, (p \mid a \times b) \rightarrow (p \mid a) \vee (p \mid b)$.

We could obtain a *Zgcd* function via Euclid algorithm. But we propose here a binary version of *Zgcd*, faster and executable within Coq.

Algorithm:

gcd 0 b = b gcd a 0 = a gcd (2a) (2b) = 2(gcd a b) gcd (2a+1) (2b) = gcd (2a+1) b gcd (2a) (2b+1) = gcd a (2b+1) gcd (2a+1) (2b+1) = gcd (b-a) (2*a+1) or gcd (a-b) (2*b+1), depending on whether a<b

Open Scope positive_scope.

```
Fixpoint Pgcdn (n: nat) (a b : positive) { struct n } : positive :=
  match n with
    | O ⇒ 1
    | S n ⇒
      match a,b with
        | xH, _ ⇒ 1
        | _, xH ⇒ 1
        | xO a, xO b ⇒ xO (Pgcdn n a b)
        | a, xO b ⇒ Pgcdn n a b
        | xO a, b ⇒ Pgcdn n a b
```

```
        | xI a', xI b' ⇒ match Pcompare a' b' Eq with
                                | Eq ⇒ a
                                | Lt ⇒ Pgcdn n (b'-a') a
                                | Gt ⇒ Pgcdn n (a'-b') b
                              end
      end
  end.

Fixpoint Pggcdn (n: nat) (a b : positive) { struct n } : (positive*(positive×positive)) :=
  match n with
    | O ⇒ (1,(a,b))
    | S n ⇒
      match a,b with
        | xH, b ⇒ (1,(1,b))
        | a, xH ⇒ (1,(a,1))
        | xO a, xO b ⇒
          let (g,p) := Pggcdn n a b in
            (xO g,p)
        | a, xO b ⇒
          let (g,p) := Pggcdn n a b in
            let (aa,bb) := p in
              (g,(aa, xO bb))
        | xO a, b ⇒
          let (g,p) := Pggcdn n a b in
            let (aa,bb) := p in
              (g,(xO aa, bb))
        | xI a', xI b' ⇒ match Pcompare a' b' Eq with
                                | Eq ⇒ (a,(1,1))
                                | Lt ⇒
                                  let (g,p) := Pggcdn n (b'-a') a in
                                    let (ba,aa) := p in
                                      (g,(aa, aa + xO ba))
                                | Gt ⇒
                                  let (g,p) := Pggcdn n (a'-b') b in
                                    let (ab,bb) := p in
                                      (g,(bb+xO ab, bb))
                              end
      end
  end.

Definition Pgcd (a b: positive) := Pgcdn (Psize a + Psize b)%nat a b.
Definition Pggcd (a b: positive) := Pggcdn (Psize a + Psize b)%nat a b.

Open Scope Z_scope.

Definition Zgcd (a b : Z) : Z := match a,b with
                                   | Z0, _ ⇒ Zabs b
                                   | _, Z0 ⇒ Zabs a
```

```
                                  | Zpos a, Zpos b ⇒ Zpos (Pgcd a b)
                                  | Zpos a, Zneg b ⇒ Zpos (Pgcd a b)
                                  | Zneg a, Zpos b ⇒ Zpos (Pgcd a b)
                                  | Zneg a, Zneg b ⇒ Zpos (Pgcd a b)
                                end.

Definition Zggcd (a b : Z) : Z*(Z×Z) := match a,b with
                                           | Z0, _ ⇒ (Zabs b,(0, Zsgn b))
                                           | _, Z0 ⇒ (Zabs a,(Zsgn a, 0))
                                           | Zpos a, Zpos b ⇒
                                             let (g,p) := Pggcd a b in
                                               let (aa,bb) := p in
                                                 (Zpos g, (Zpos aa, Zpos bb))
                                           | Zpos a, Zneg b ⇒
                                             let (g,p) := Pggcd a b in
                                               let (aa,bb) := p in
                                                 (Zpos g, (Zpos aa, Zneg bb))
                                           | Zneg a, Zpos b ⇒
                                             let (g,p) := Pggcd a b in
                                               let (aa,bb) := p in
                                                 (Zpos g, (Zneg aa, Zpos bb))
                                           | Zneg a, Zneg b ⇒
                                             let (g,p) := Pggcd a b in
                                               let (aa,bb) := p in
                                                 (Zpos g, (Zneg aa, Zneg bb))
                                         end.

Lemma Zgcd_is_pos : ∀ a b, 0 ≤ Zgcd a b.

Lemma Psize_monotone : ∀ p q, Pcompare p q Eq = Lt → (Psize p ≤ Psize q)%nat.

Lemma Pminus_Zminus : ∀ a b, Pcompare a b Eq = Lt →
  Zpos (b-a) = Zpos b - Zpos a.

Lemma Zis_gcd_even_odd : ∀ a b g, Zis_gcd (Zpos a) (Zpos (xI b)) g →
  Zis_gcd (Zpos (xO a)) (Zpos (xI b)) g.

Lemma Pgcdn_correct : ∀ n a b, (Psize a + Psize b≤n)%nat →
  Zis_gcd (Zpos a) (Zpos b) (Zpos (Pgcdn n a b)).

Lemma Pgcd_correct : ∀ a b, Zis_gcd (Zpos a) (Zpos b) (Zpos (Pgcd a b)).

Lemma Zgcd_is_gcd : ∀ a b, Zis_gcd a b (Zgcd a b).

Lemma Pggcdn_gcdn : ∀ n a b,
  fst (Pggcdn n a b) = Pgcdn n a b.

Lemma Pggcd_gcd : ∀ a b, fst (Pggcd a b) = Pgcd a b.

Lemma Zggcd_gcd : ∀ a b, fst (Zggcd a b) = Zgcd a b.

Open Scope positive_scope.
```

Lemma $Pggcdn\_correct\_divisors : \forall\ n\ a\ b,$
  let $(g,p) := Pggcdn\ n\ a\ b$ in
    let $(aa,bb) := p$ in
      $(a = g \times aa) \wedge (b = g \times bb)$.

Lemma $Pggcd\_correct\_divisors : \forall\ a\ b,$
  let $(g,p) := Pggcd\ a\ b$ in
    let $(aa,bb) := p$ in
      $(a = g \times aa) \wedge (b = g \times bb)$.

Open Scope Z_scope.

Lemma $Zggcd\_correct\_divisors : \forall\ a\ b,$
  let $(g,p) := Zggcd\ a\ b$ in
    let $(aa,bb) := p$ in
      $(a = g \times aa) \wedge (b = g \times bb)$.

Theorem $Zgcd\_spec : \forall\ x\ y : Z, \{z : Z \mid Zis\_gcd\ x\ y\ z \wedge 0 \leq z\}$.

Chapter 77

Module Coq.ZArith.Zorder

Binary Integers (Pierre Crégut (CNET, Lannion, France)

Require Import *BinPos*.
Require Import *BinInt*.
Require Import *Arith_base*.
Require Import *Decidable*.
Require Import *Zcompare*.

Open Local *Scope Z_scope*.

Implicit *Types* $x\ y\ z : Z$.

Properties of the order relations on binary integers

77.1 Trichotomy

Theorem *Ztrichotomy_inf* : $\forall\ n\ m{:}Z, \{n < m\} + \{n = m\} + \{n > m\}$.

Theorem *Ztrichotomy* : $\forall\ n\ m{:}Z, n < m \vee n = m \vee n > m$.

77.2 Decidability of equality and order on Z

Theorem *dec_eq* : $\forall\ n\ m{:}Z,\ decidable\ (n = m)$.

Theorem *dec_Zne* : $\forall\ n\ m{:}Z,\ decidable\ (Zne\ n\ m)$.

Theorem *dec_Zle* : $\forall\ n\ m{:}Z,\ decidable\ (n \leq m)$.

Theorem *dec_Zgt* : $\forall\ n\ m{:}Z,\ decidable\ (n > m)$.

Theorem *dec_Zge* : $\forall\ n\ m{:}Z,\ decidable\ (n \geq m)$.

Theorem *dec_Zlt* : $\forall\ n\ m{:}Z,\ decidable\ (n < m)$.

Theorem *not_Zeq* : $\forall\ n\ m{:}Z, n \neq m \rightarrow n < m \vee m < n$.

77.3 Relating strict and large orders

Lemma $Zgt\_lt : \forall\ n\ m{:}Z,\ n > m \rightarrow m < n.$

Lemma $Zlt\_gt : \forall\ n\ m{:}Z,\ n < m \rightarrow m > n.$

Lemma $Zge\_le : \forall\ n\ m{:}Z,\ n \geq m \rightarrow m \leq n.$

Lemma $Zle\_ge : \forall\ n\ m{:}Z,\ n \leq m \rightarrow m \geq n.$

Lemma $Zle\_not\_gt : \forall\ n\ m{:}Z,\ n \leq m \rightarrow \neg\ n > m.$

Lemma $Zgt\_not\_le : \forall\ n\ m{:}Z,\ n > m \rightarrow \neg\ n \leq m.$

Lemma $Zle\_not\_lt : \forall\ n\ m{:}Z,\ n \leq m \rightarrow \neg\ m < n.$

Lemma $Zlt\_not\_le : \forall\ n\ m{:}Z,\ n < m \rightarrow \neg\ m \leq n.$

Lemma $Znot\_ge\_lt : \forall\ n\ m{:}Z,\ \neg\ n \geq m \rightarrow n < m.$

Lemma $Znot\_lt\_ge : \forall\ n\ m{:}Z,\ \neg\ n < m \rightarrow n \geq m.$

Lemma $Znot\_gt\_le : \forall\ n\ m{:}Z,\ \neg\ n > m \rightarrow n \leq m.$

Lemma $Znot\_le\_gt : \forall\ n\ m{:}Z,\ \neg\ n \leq m \rightarrow n > m.$

Lemma $Zge\_iff\_le : \forall\ n\ m{:}Z,\ n \geq m \leftrightarrow m \leq n.$

Lemma $Zgt\_iff\_lt : \forall\ n\ m{:}Z,\ n > m \leftrightarrow m < n.$

77.4 Equivalence and order properties

Reflexivity

Lemma $Zle\_refl : \forall\ n{:}Z,\ n \leq n.$

Lemma $Zeq\_le : \forall\ n\ m{:}Z,\ n = m \rightarrow n \leq m.$

Hint *Resolve Zle_refl: zarith.*

Antisymmetry

Lemma $Zle\_antisym : \forall\ n\ m{:}Z,\ n \leq m \rightarrow m \leq n \rightarrow n = m.$

Asymmetry

Lemma $Zgt\_asym : \forall\ n\ m{:}Z,\ n > m \rightarrow \neg\ m > n.$

Lemma $Zlt\_asym : \forall\ n\ m{:}Z,\ n < m \rightarrow \neg\ m < n.$

Irreflexivity

Lemma $Zgt\_irrefl : \forall\ n{:}Z,\ \neg\ n > n.$

Lemma $Zlt\_irrefl : \forall\ n{:}Z,\ \neg\ n < n.$

Lemma $Zlt\_not\_eq : \forall\ n\ m{:}Z,\ n < m \rightarrow n \neq m.$

Large = strict or equal

Lemma $Zlt\_le\_weak : \forall\ n\ m{:}Z,\ n < m \to n \leq m.$

Lemma $Zle\_lt\_or\_eq : \forall\ n\ m{:}Z,\ n \leq m \to n < m \vee n = m.$

Dichotomy

Lemma $Zle\_or\_lt : \forall\ n\ m{:}Z,\ n \leq m \vee m < n.$

Transitivity of strict orders

Lemma $Zgt\_trans : \forall\ n\ m\ p{:}Z,\ n > m \to m > p \to n > p.$

Lemma $Zlt\_trans : \forall\ n\ m\ p{:}Z,\ n < m \to m < p \to n < p.$

Mixed transitivity

Lemma $Zle\_gt\_trans : \forall\ n\ m\ p{:}Z,\ m \leq n \to m > p \to n > p.$

Lemma $Zgt\_le\_trans : \forall\ n\ m\ p{:}Z,\ n > m \to p \leq m \to n > p.$

Lemma $Zlt\_le\_trans : \forall\ n\ m\ p{:}Z,\ n < m \to m \leq p \to n < p.$

Lemma $Zle\_lt\_trans : \forall\ n\ m\ p{:}Z,\ n \leq m \to m < p \to n < p.$

Transitivity of large orders

Lemma $Zle\_trans : \forall\ n\ m\ p{:}Z,\ n \leq m \to m \leq p \to n \leq p.$

Lemma $Zge\_trans : \forall\ n\ m\ p{:}Z,\ n \geq m \to m \geq p \to n \geq p.$

Hint *Resolve Zle_trans*: *zarith*.

77.5 Compatibility of order and operations on Z

77.5.1 Successor

Compatibility of successor wrt to order

Lemma $Zsucc\_le\_compat : \forall\ n\ m{:}Z,\ m \leq n \to Zsucc\ m \leq Zsucc\ n.$

Lemma $Zsucc\_gt\_compat : \forall\ n\ m{:}Z,\ m > n \to Zsucc\ m > Zsucc\ n.$

Lemma $Zsucc\_lt\_compat : \forall\ n\ m{:}Z,\ n < m \to Zsucc\ n < Zsucc\ m.$

Hint *Resolve Zsucc_le_compat*: *zarith*.

Simplification of successor wrt to order

Lemma $Zsucc\_gt\_reg : \forall\ n\ m{:}Z,\ Zsucc\ m > Zsucc\ n \to m > n.$

Lemma $Zsucc\_le\_reg : \forall\ n\ m{:}Z,\ Zsucc\ m \leq Zsucc\ n \to m \leq n.$

Lemma $Zsucc\_lt\_reg : \forall\ n\ m{:}Z,\ Zsucc\ n < Zsucc\ m \to n < m.$

Special base instances of order

Lemma $Zgt\_succ : \forall\ n{:}Z,\ Zsucc\ n > n.$

Lemma *Znot_le_succ* : $\forall\ n{:}Z, \neg\ Zsucc\ n \leq n$.

Lemma *Zlt_succ* : $\forall\ n{:}Z, n < Zsucc\ n$.

Lemma *Zlt_pred* : $\forall\ n{:}Z, Zpred\ n < n$.

Relating strict and large order using successor or predecessor

Lemma *Zgt_le_succ* : $\forall\ n\ m{:}Z, m > n \rightarrow Zsucc\ n \leq m$.

Lemma *Zlt_gt_succ* : $\forall\ n\ m{:}Z, n \leq m \rightarrow Zsucc\ m > n$.

Lemma *Zle_lt_succ* : $\forall\ n\ m{:}Z, n \leq m \rightarrow n < Zsucc\ m$.

Lemma *Zlt_le_succ* : $\forall\ n\ m{:}Z, n < m \rightarrow Zsucc\ n \leq m$.

Lemma *Zgt_succ_le* : $\forall\ n\ m{:}Z, Zsucc\ m > n \rightarrow n \leq m$.

Lemma *Zlt_succ_le* : $\forall\ n\ m{:}Z, n < Zsucc\ m \rightarrow n \leq m$.

Lemma *Zlt_succ_gt* : $\forall\ n\ m{:}Z, Zsucc\ n \leq m \rightarrow m > n$.

Weakening order

Lemma *Zle_succ* : $\forall\ n{:}Z, n \leq Zsucc\ n$.

Hint *Resolve Zle_succ*: *zarith*.

Lemma *Zle_pred* : $\forall\ n{:}Z, Zpred\ n \leq n$.

Lemma *Zlt_lt_succ* : $\forall\ n\ m{:}Z, n < m \rightarrow n < Zsucc\ m$.

Lemma *Zle_le_succ* : $\forall\ n\ m{:}Z, n \leq m \rightarrow n \leq Zsucc\ m$.

Lemma *Zle_succ_le* : $\forall\ n\ m{:}Z, Zsucc\ n \leq m \rightarrow n \leq m$.

Hint *Resolve Zle_le_succ*: *zarith*.

Relating order wrt successor and order wrt predecessor

Lemma *Zgt_succ_pred* : $\forall\ n\ m{:}Z, m > Zsucc\ n \rightarrow Zpred\ m > n$.

Lemma *Zlt_succ_pred* : $\forall\ n\ m{:}Z, Zsucc\ n < m \rightarrow n < Zpred\ m$.

Relating strict order and large order on positive

Lemma *Zlt_0_le_0_pred* : $\forall\ n{:}Z, 0 < n \rightarrow 0 \leq Zpred\ n$.

Lemma *Zgt_0_le_0_pred* : $\forall\ n{:}Z, n > 0 \rightarrow 0 \leq Zpred\ n$.

Special cases of ordered integers

Lemma *Zlt_0_1* : $0 < 1$.

Lemma *Zle_0_1* : $0 \leq 1$.

Lemma *Zle_neg_pos* : $\forall\ p\ q{:}positive, Zneg\ p \leq Zpos\ q$.

Lemma *Zgt_pos_0* : $\forall\ p{:}positive, Zpos\ p > 0$.

Lemma *Zle_0_pos* : $\forall\ p{:}positive, 0 \leq Zpos\ p$.

Lemma *Zlt_neg_0* : $\forall\ p{:}positive, Zneg\ p < 0$.

Lemma $Zle\_0\_nat : \forall\ n{:}nat, 0 \leq Z\_of\_nat\ n.$

Hint Immediate $Zeq\_le{:}\ zarith.$

Transitivity using successor

Lemma $Zge\_trans\_succ : \forall\ n\ m\ p{:}Z, Zsucc\ n > m \rightarrow m > p \rightarrow n > p.$

Derived lemma

Lemma $Zgt\_succ\_gt\_or\_eq : \forall\ n\ m{:}Z, Zsucc\ n > m \rightarrow n > m \vee m = n.$

77.5.2 Addition

Compatibility of addition wrt to order

Lemma $Zplus\_gt\_compat\_l : \forall\ n\ m\ p{:}Z, n > m \rightarrow p + n > p + m.$

Lemma $Zplus\_gt\_compat\_r : \forall\ n\ m\ p{:}Z, n > m \rightarrow n + p > m + p.$

Lemma $Zplus\_le\_compat\_l : \forall\ n\ m\ p{:}Z, n \leq m \rightarrow p + n \leq p + m.$

Lemma $Zplus\_le\_compat\_r : \forall\ n\ m\ p{:}Z, n \leq m \rightarrow n + p \leq m + p.$

Lemma $Zplus\_lt\_compat\_l : \forall\ n\ m\ p{:}Z, n < m \rightarrow p + n < p + m.$

Lemma $Zplus\_lt\_compat\_r : \forall\ n\ m\ p{:}Z, n < m \rightarrow n + p < m + p.$

Lemma $Zplus\_lt\_le\_compat : \forall\ n\ m\ p\ q{:}Z, n < m \rightarrow p \leq q \rightarrow n + p < m + q.$

Lemma $Zplus\_le\_lt\_compat : \forall\ n\ m\ p\ q{:}Z, n \leq m \rightarrow p < q \rightarrow n + p < m + q.$

Lemma $Zplus\_le\_compat : \forall\ n\ m\ p\ q{:}Z, n \leq m \rightarrow p \leq q \rightarrow n + p \leq m + q.$

Lemma $Zplus\_lt\_compat : \forall\ n\ m\ p\ q{:}Z, n < m \rightarrow p < q \rightarrow n + p < m + q.$

Compatibility of addition wrt to being positive

Lemma $Zplus\_le\_0\_compat : \forall\ n\ m{:}Z, 0 \leq n \rightarrow 0 \leq m \rightarrow 0 \leq n + m.$

Simplification of addition wrt to order

Lemma $Zplus\_gt\_reg\_l : \forall\ n\ m\ p{:}Z, p + n > p + m \rightarrow n > m.$

Lemma $Zplus\_gt\_reg\_r : \forall\ n\ m\ p{:}Z, n + p > m + p \rightarrow n > m.$

Lemma $Zplus\_le\_reg\_l : \forall\ n\ m\ p{:}Z, p + n \leq p + m \rightarrow n \leq m.$

Lemma $Zplus\_le\_reg\_r : \forall\ n\ m\ p{:}Z, n + p \leq m + p \rightarrow n \leq m.$

Lemma $Zplus\_lt\_reg\_l : \forall\ n\ m\ p{:}Z, p + n < p + m \rightarrow n < m.$

Lemma $Zplus\_lt\_reg\_r : \forall\ n\ m\ p{:}Z, n + p < m + p \rightarrow n < m.$

77.5.3 Multiplication

Compatibility of multiplication by a positive wrt to order

Lemma *Zmult_le_compat_r* : $\forall\ n\ m\ p{:}Z,\ n \leq m \rightarrow 0 \leq p \rightarrow n \times p \leq m \times p.$

Lemma *Zmult_le_compat_l* : $\forall\ n\ m\ p{:}Z,\ n \leq m \rightarrow 0 \leq p \rightarrow p \times n \leq p \times m.$

Lemma *Zmult_lt_compat_r* : $\forall\ n\ m\ p{:}Z,\ 0 < p \rightarrow n < m \rightarrow n \times p < m \times p.$

Lemma *Zmult_gt_compat_r* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n > m \rightarrow n \times p > m \times p.$

Lemma *Zmult_gt_0_lt_compat_r* :
$\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n < m \rightarrow n \times p < m \times p.$

Lemma *Zmult_gt_0_le_compat_r* :
$\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n \leq m \rightarrow n \times p \leq m \times p.$

Lemma *Zmult_lt_0_le_compat_r* :
$\forall\ n\ m\ p{:}Z,\ 0 < p \rightarrow n \leq m \rightarrow n \times p \leq m \times p.$

Lemma *Zmult_gt_0_lt_compat_l* :
$\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n < m \rightarrow p \times n < p \times m.$

Lemma *Zmult_lt_compat_l* : $\forall\ n\ m\ p{:}Z,\ 0 < p \rightarrow n < m \rightarrow p \times n < p \times m.$

Lemma *Zmult_gt_compat_l* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n > m \rightarrow p \times n > p \times m.$

Lemma *Zmult_ge_compat_r* : $\forall\ n\ m\ p{:}Z,\ n \geq m \rightarrow p \geq 0 \rightarrow n \times p \geq m \times p.$

Lemma *Zmult_ge_compat_l* : $\forall\ n\ m\ p{:}Z,\ n \geq m \rightarrow p \geq 0 \rightarrow p \times n \geq p \times m.$

Lemma *Zmult_ge_compat* :
$\forall\ n\ m\ p\ q{:}Z,\ n \geq p \rightarrow m \geq q \rightarrow p \geq 0 \rightarrow q \geq 0 \rightarrow n \times m \geq p \times q.$

Lemma *Zmult_le_compat* :
$\forall\ n\ m\ p\ q{:}Z,\ n \leq p \rightarrow m \leq q \rightarrow 0 \leq n \rightarrow 0 \leq m \rightarrow n \times m \leq p \times q.$

Simplification of multiplication by a positive wrt to being positive

Lemma *Zmult_gt_0_lt_reg_r* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n \times p < m \times p \rightarrow n < m.$

Lemma *Zmult_lt_reg_r* : $\forall\ n\ m\ p{:}Z,\ 0 < p \rightarrow n \times p < m \times p \rightarrow n < m.$

Lemma *Zmult_le_reg_r* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n \times p \leq m \times p \rightarrow n \leq m.$

Lemma *Zmult_lt_0_le_reg_r* : $\forall\ n\ m\ p{:}Z,\ 0 < p \rightarrow n \times p \leq m \times p \rightarrow n \leq m.$

Lemma *Zmult_ge_reg_r* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n \times p \geq m \times p \rightarrow n \geq m.$

Lemma *Zmult_gt_reg_r* : $\forall\ n\ m\ p{:}Z,\ p > 0 \rightarrow n \times p > m \times p \rightarrow n > m.$

Compatibility of multiplication by a positive wrt to being positive

Lemma *Zmult_le_0_compat* : $\forall\ n\ m{:}Z,\ 0 \leq n \rightarrow 0 \leq m \rightarrow 0 \leq n \times m.$

Lemma *Zmult_gt_0_compat* : $\forall\ n\ m{:}Z,\ n > 0 \rightarrow m > 0 \rightarrow n \times m > 0.$

Lemma *Zmult_lt_0_compat* : $\forall\ n\ m{:}Z,\ 0 < n \rightarrow 0 < m \rightarrow 0 < n \times m.$

For compatibility

Notation *Zmult_lt_O_compat* := *Zmult_lt_0_compat* (*only parsing*).

Lemma *Zmult_gt_0_le_0_compat* : $\forall\ n\ m{:}Z,\ n > 0 \to 0 \leq m \to 0 \leq m \times n$.

Simplification of multiplication by a positive wrt to being positive

Lemma *Zmult_le_0_reg_r* : $\forall\ n\ m{:}Z,\ n > 0 \to 0 \leq m \times n \to 0 \leq m$.

Lemma *Zmult_gt_0_lt_0_reg_r* : $\forall\ n\ m{:}Z,\ n > 0 \to 0 < m \times n \to 0 < m$.

Lemma *Zmult_lt_0_reg_r* : $\forall\ n\ m{:}Z,\ 0 < n \to 0 < m \times n \to 0 < m$.

Lemma *Zmult_gt_0_reg_l* : $\forall\ n\ m{:}Z,\ n > 0 \to n \times m > 0 \to m > 0$.

77.5.4 Square

Simplification of square wrt order

Lemma *Zgt_square_simpl* :
$\forall\ n\ m{:}Z,\ n \geq 0 \to n \times n > m \times m \to n > m$.

Lemma *Zlt_square_simpl* :
$\forall\ n\ m{:}Z,\ 0 \leq n \to m \times m < n \times n \to m < n$.

77.6 Equivalence between inequalities

Lemma *Zle_plus_swap* : $\forall\ n\ m\ p{:}Z,\ n + p \leq m \leftrightarrow n \leq m$ - p.

Lemma *Zlt_plus_swap* : $\forall\ n\ m\ p{:}Z,\ n + p < m \leftrightarrow n < m$ - p.

Lemma *Zeq_plus_swap* : $\forall\ n\ m\ p{:}Z,\ n + p = m \leftrightarrow n = m$ - p.

Lemma *Zlt_minus_simpl_swap* : $\forall\ n\ m{:}Z,\ 0 < m \to n$ - $m < n$.

Lemma *Zlt_0_minus_lt* : $\forall\ n\ m{:}Z,\ 0 < n$ - $m \to m < n$.

Lemma *Zle_0_minus_le* : $\forall\ n\ m{:}Z,\ 0 \leq n$ - $m \to m \leq n$.

Lemma *Zle_minus_le_0* : $\forall\ n\ m{:}Z,\ m \leq n \to 0 \leq n$ - m.

For compatibility
Notation *Zlt_O_minus_lt* := *Zlt_0_minus_lt* (*only parsing*).

Chapter 78

Module Coq.ZArith.Zpow_def

```
Require Import ZArith_base.
Require Import Ring_theory.

Open Local Scope Z_scope.

Zpower_pos z n is the n-th power of z when n is an binary integer (type positive) and z a signed
integer (type Z)
Definition Zpower_pos (z:Z) (n:positive) := iter_pos n Z (fun x:Z ⇒ z × x) 1.

Definition Zpower (x y:Z) :=
  match y with
    | Zpos p ⇒ Zpower_pos x p
    | Z0 ⇒ 1
    | Zneg p ⇒ 0
  end.

Lemma Zpower_theory : power_theory 1 Zmult (eq (A:=Z)) Z_of_N Zpower.
```

Chapter 79

Module Coq.ZArith.Zpower

Require Import *ZArith_base*.
Require Export *Zpow_def*.
Require Import *Omega*.
Require Import *Zcomplements*.
Open Local *Scope Z_scope*.

Section *section1*.

79.1 Definition of powers over *Z*

Zpower_nat z n is the n-th power of *z* when *n* is an unary integer (type *nat*) and *z* a signed integer (type *Z*)

Definition *Zpower_nat* (*z*:*Z*) (*n*:*nat*) := *iter_nat n Z* (fun *x*:*Z* ⇒ *z* × *x*) 1.

Zpower_nat_is_exp says *Zpower_nat* is a morphism for *plus* : *nat*→*nat* and *Zmult* : *Z*→*Z*

Lemma *Zpower_nat_is_exp* :
∀ (*n m*:*nat*) (*z*:*Z*),
Zpower_nat z (*n* + *m*) = *Zpower_nat z n* × *Zpower_nat z m*.

This theorem shows that powers of unary and binary integers are the same thing, modulo the function convert : *positive* → *nat*

Theorem *Zpower_pos_nat* :
∀ (*z*:*Z*) (*p*:*positive*), *Zpower_pos z p* = *Zpower_nat z* (*nat_of_P p*).

Using the theorem *Zpower_pos_nat* and the lemma *Zpower_nat_is_exp* we deduce that the function [*n*:*positive*](*Zpower_pos z n*) is a morphism for *add* : *positive*→*positive* and *Zmult* : *Z*→*Z*

Theorem *Zpower_pos_is_exp* :
∀ (*n m*:*positive*) (*z*:*Z*),
Zpower_pos z (*n* + *m*) = *Zpower_pos z n* × *Zpower_pos z m*.

Infix "^" := *Zpower* : *Z_scope*.

Hint Immediate *Zpower_nat_is_exp*: *zarith*.

Hint Immediate *Zpower_pos_is_exp*: *zarith*.
Hint *Unfold Zpower_pos*: *zarith*.
Hint *Unfold Zpower_nat*: *zarith*.

Lemma *Zpower_exp* :
$\forall\ x\ n\ m{:}Z,\ n \geq 0 \rightarrow m \geq 0 \rightarrow x\ \hat{}\ (n + m) = x\ \hat{}\ n \times x\ \hat{}\ m$.

End *section1*.

Exporting notation "^"

Infix "^" := *Zpower* : *Z_scope*.

Hint Immediate *Zpower_nat_is_exp*: *zarith*.
Hint Immediate *Zpower_pos_is_exp*: *zarith*.
Hint *Unfold Zpower_pos*: *zarith*.
Hint *Unfold Zpower_nat*: *zarith*.

Section *Powers_of_2*.

79.2 Powers of 2

For the powers of two, that will be widely used, a more direct calculus is possible. We will also prove some properties such as $(x{:}positive)\ x < 2\hat{}x$ that are true for all integers bigger than 2 but more difficult to prove and useless.

shift $n\ m$ computes $2\hat{}n \times m$, or m shifted by n positions

Definition *shift_nat* $(n{:}nat)\ (z{:}positive) :=$ *iter_nat* n *positive* $xO\ z$.
Definition *shift_pos* $(n\ z{:}positive) :=$ *iter_pos* n *positive* $xO\ z$.
Definition *shift* $(n{:}Z)\ (z{:}positive) :=$
  match n with
    | $Z0 \Rightarrow z$
    | $Zpos\ p \Rightarrow$ *iter_pos* p *positive* $xO\ z$
    | $Zneg\ p \Rightarrow z$
  end.

Definition *two_power_nat* $(n{:}nat) :=$ *Zpos* (*shift_nat* $n\ 1$).
Definition *two_power_pos* $(x{:}positive) :=$ *Zpos* (*shift_pos* $x\ 1$).

Lemma *two_power_nat_S* :
$\forall\ n{:}nat,$ *two_power_nat* $(S\ n) = 2 \times$ *two_power_nat* n.

Lemma *shift_nat_plus* :
$\forall\ (n\ m{:}nat)\ (x{:}positive),$
  shift_nat $(n + m)\ x =$ *shift_nat* n (*shift_nat* $m\ x$).

Theorem *shift_nat_correct* :
$\forall\ (n{:}nat)\ (x{:}positive),$ *Zpos* (*shift_nat* $n\ x$) = *Zpower_nat* $2\ n \times$ *Zpos* x.

Theorem *two_power_nat_correct* :
$\forall\ n{:}nat,$ *two_power_nat* $n =$ *Zpower_nat* $2\ n$.

Second we show that *two_power_pos* and *two_power_nat* are the same

Lemma *shift_pos_nat* :
$\forall\ p\ x{:}positive,\ shift\_pos\ p\ x = shift\_nat\ (nat\_of\_P\ p)\ x.$

Lemma *two_power_pos_nat* :
$\forall\ p{:}positive,\ two\_power\_pos\ p = two\_power\_nat\ (nat\_of\_P\ p).$

Then we deduce that *two_power_pos* is also correct

Theorem *shift_pos_correct* :
$\forall\ p\ x{:}positive,\ Zpos\ (shift\_pos\ p\ x) = Zpower\_pos\ 2\ p \times Zpos\ x.$

Theorem *two_power_pos_correct* :
$\forall\ x{:}positive,\ two\_power\_pos\ x = Zpower\_pos\ 2\ x.$

Some consequences

Theorem *two_power_pos_is_exp* :
$\forall\ x\ y{:}positive,$
$two\_power\_pos\ (x + y) = two\_power\_pos\ x \times two\_power\_pos\ y.$

The exponentiation $z \rightarrow 2\hat{}z$ for z a signed integer. For convenience, we assume that $2\hat{}z = 0$ for all $z < 0$ We could also define a inductive type *Log_result* with 3 contructors *Zero* | *Pos positive* → | *minus_infty* but it's more complexe and not so useful.

```
Definition two_p (x:Z) :=
  match x with
    | Z0 ⇒ 1
    | Zpos y ⇒ two_power_pos y
    | Zneg y ⇒ 0
  end.
```

Theorem *two_p_is_exp* :
$\forall\ x\ y{:}Z,\ 0 \leq x \rightarrow 0 \leq y \rightarrow two\_p\ (x + y) = two\_p\ x \times two\_p\ y.$

Lemma *two_p_gt_ZERO* : $\forall\ x{:}Z,\ 0 \leq x \rightarrow two\_p\ x > 0.$

Lemma *two_p_S* : $\forall\ x{:}Z,\ 0 \leq x \rightarrow two\_p\ (Zsucc\ x) = 2 \times two\_p\ x.$

Lemma *two_p_pred* : $\forall\ x{:}Z,\ 0 \leq x \rightarrow two\_p\ (Zpred\ x) < two\_p\ x.$

Lemma *Zlt_lt_double* : $\forall\ x\ y{:}Z,\ 0 \leq x < y \rightarrow x < 2 \times y.$

End *Powers_of_2*.

Hint *Resolve two_p_gt_ZERO*: *zarith*.
Hint Immediate *two_p_pred two_p_S*: *zarith*.

Section *power_div_with_rest*.

79.3 Division by a power of two.

To $n{:}Z$ and $p{:}positive$, q,r are associated such that $n = 2\hat{}p.q + r$ and $0 \leq r < 2\hat{}p$

```
Invariant: d×q + r = d'×q + r ∧ d' = 2×d ∧ 0≤ r < d ∧ 0 ≤ r' < d'
  Definition Zdiv_rest_aux (qrd:Z × Z × Z) :=
    let (qr, d) := qrd in
      let (q, r) := qr in
        (match q with
           | Z0 ⇒ (0, r)
           | Zpos xH ⇒ (0, d + r)
           | Zpos (xI n) ⇒ (Zpos n, d + r)
           | Zpos (xO n) ⇒ (Zpos n, r)
           | Zneg xH ⇒ (-1, d + r)
           | Zneg (xI n) ⇒ (Zneg n - 1, d + r)
           | Zneg (xO n) ⇒ (Zneg n, r)
         end, 2 × d).

  Definition Zdiv_rest (x:Z) (p:positive) :=
    let (qr, d) := iter_pos p _ Zdiv_rest_aux (x, 0, 1) in qr.

  Lemma Zdiv_rest_correct1 :
    ∀ (x:Z) (p:positive),
      let (qr, d) := iter_pos p _ Zdiv_rest_aux (x, 0, 1) in d = two_power_pos p.

  Lemma Zdiv_rest_correct2 :
    ∀ (x:Z) (p:positive),
      let (qr, d) := iter_pos p _ Zdiv_rest_aux (x, 0, 1) in
        let (q, r) := qr in x = q × d + r ∧ 0 ≤ r < d.

  Inductive Zdiv_rest_proofs (x:Z) (p:positive) : Set :=
    Zdiv_rest_proof :
    ∀ q r:Z,
      x = q × two_power_pos p + r →
      0 ≤ r → r < two_power_pos p → Zdiv_rest_proofs x p.

  Lemma Zdiv_rest_correct : ∀ (x:Z) (p:positive), Zdiv_rest_proofs x p.

End power_div_with_rest.
```

Chapter 80

Module Coq.ZArith.Zsqrt

```
Require Import ZArithRing.
Require Import Omega.
Require Export ZArith_base.
Open Local Scope Z_scope.
```

Definition and properties of square root on Z

The following tactic replaces all instances of (POS (xI ...)) by '2*(POS ...)+1', but only when ... is not made only with xO, XI, or xH.

```
Ltac compute_POS :=
  match goal with
    | ⊢ context [(Zpos (xI ?X1))] ⇒
      match constr:X1 with
        | context [1%positive] ⇒ fail 1
        | _ ⇒ rewrite (BinInt.Zpos_xI X1)
      end
    | ⊢ context [(Zpos (xO ?X1))] ⇒
      match constr:X1 with
        | context [1%positive] ⇒ fail 1
        | _ ⇒ rewrite (BinInt.Zpos_xO X1)
      end
  end.
```

```
Inductive sqrt_data (n:Z) : Set :=
  c_sqrt : ∀ s r:Z, n = s × s + r → 0 ≤ r ≤ 2 × s → sqrt_data n.
```

```
Definition sqrtrempos : ∀ p:positive, sqrt_data (Zpos p).
```

Define with integer input, but with a strong (readable) specification.

```
Definition Zsqrt :
  ∀ x:Z,
    0 ≤ x →
    {s : Z & {r : Z | x = s × s + r ∧ s × s ≤ x < (s + 1) × (s + 1)}}.
```

Define a function of type Z->Z that computes the integer square root, but only for positive numbers, and 0 for others.

```
Definition Zsqrt_plain (x:Z) : Z :=
  match x with
    | Zpos p ⇒
      match Zsqrt (Zpos p) (Zorder.Zle_0_pos p) with
        | existS s _ ⇒ s
      end
    | Zneg p ⇒ 0
    | Z0 ⇒ 0
  end.
```

A basic theorem about Zsqrt_plain

```
Theorem Zsqrt_interval :
  ∀ n:Z,
    0 ≤ n →
    Zsqrt_plain n × Zsqrt_plain n ≤ n <
    (Zsqrt_plain n + 1) × (Zsqrt_plain n + 1).
```

Chapter 81

Module Coq.ZArith.Zwf

Require Import *ZArith_base*.
Require Export *Wf_nat*.
Require Import *Omega*.
Open Local *Scope Z_scope*.

Well-founded relations on Z.

We define the following family of relations on $Z\ x\ Z$:

$x\ (Zwf\ c)\ y$ iff $x < y\ \&\ c \leq y$

Definition $Zwf\ (c\ x\ y{:}Z) := c \leq y \wedge x < y$.

and we prove that $(Zwf\ c)$ is well founded

Section *wf_proof*.

Variable $c : Z$.

The proof of well-foundness is classic: we do the proof by induction on a measure in nat, which is here $|x\text{-}c|$

Let $f\ (z{:}Z) := Zabs\_nat\ (z - c)$.

Lemma *Zwf_well_founded* : *well_founded* $(Zwf\ c)$.
n= 0
inductive case

End *wf_proof*.

Hint *Resolve Zwf_well_founded*: *datatypes v62*.

We also define the other family of relations:

$x\ (Zwf\_up\ c)\ y$ iff $y < x \leq c$

Definition $Zwf\_up\ (c\ x\ y{:}Z) := y < x \leq c$.

and we prove that $(Zwf\_up\ c)$ is well founded

Section *wf_proof_up*.

Variable $c : Z$.

The proof of well-foundness is classic: we do the proof by induction on a measure in nat, which is here $|c\text{-}x|$

Let f $(z{:}Z)$:= $Zabs\_nat$ $(c$ - $z)$.

Lemma $Zwf\_up\_well\_founded$: $well\_founded$ $(Zwf\_up$ $c)$.

End wf_proof_up.

Hint $Resolve$ $Zwf\_up\_well\_founded$: $datatypes$ $v62$.

Chapter 82

Module Coq.QArith.QArith_base

Require Export *ZArith*.
Require Export *ZArithRing*.
Require Export *Setoid*.

82.1 Definition of Q and basic properties

Rationals are pairs of Z and *positive* numbers.

Record Q : Set := $Qmake$ {$Qnum$: Z; $Qden$: $positive$}.

Delimit Scope Q_scope with Q.
Open Scope Q_scope.
Ltac *simpl_mult* := *repeat rewrite Zpos_mult_morphism*.

$a\#b$ denotes the fraction a over b.

Notation "a # b" := ($Qmake$ a b) (*at level* 55, *no associativity*) : Q_scope.

Definition *inject_Z* (x : Z) := $Qmake$ x 1.

Notation " '*QDen*' p " := ($Zpos$ ($Qden$ p)) (*at level* 20, *no associativity*) : Q_scope.
Notation " 0 " := (0#1) : Q_scope.
Notation " 1 " := (1#1) : Q_scope.

Definition Qeq (p q : Q) := ($Qnum$ p × $QDen$ q)%Z = ($Qnum$ q × $QDen$ p)%Z.
Definition Qle (x y : Q) := ($Qnum$ x × $QDen$ y ≤ $Qnum$ y × $QDen$ x)%Z.
Definition Qlt (x y : Q) := ($Qnum$ x × $QDen$ y < $Qnum$ y × $QDen$ x)%Z.
Notation Qgt := (fun a b : Q ⇒ Qlt b a).
Notation Qge := (fun a b : Q ⇒ Qle b a).

Infix "==" := Qeq (*at level* 70, *no associativity*) : Q_scope.
Infix "<" := Qlt : Q_scope.
Infix ">" := Qgt : Q_scope.
Infix "≤" := Qle : Q_scope.
Infix "≥" := Qge : Q_scope.
Notation "x ≤ y ≤ z" := (x≤y∧y≤z) : Q_scope.

Another approach : using Qcompare for defining order relations.

Definition $Qcompare\ (p\ q : Q) := (Qnum\ p \times QDen\ q ?= Qnum\ q \times QDen\ p)\%Z$.
Notation "$p ?= q$" := $(Qcompare\ p\ q) : Q\_scope$.

Lemma $Qeq\_alt : \forall\ p\ q, (p == q) \leftrightarrow (p ?= q) = Eq$.

Lemma $Qlt\_alt : \forall\ p\ q, (p<q) \leftrightarrow (p?=q = Lt)$.

Lemma $Qgt\_alt : \forall\ p\ q, (p>q) \leftrightarrow (p?=q = Gt)$.

Lemma $Qle\_alt : \forall\ p\ q, (p\leq q) \leftrightarrow (p?=q \neq Gt)$.

Lemma $Qge\_alt : \forall\ p\ q, (p\geq q) \leftrightarrow (p?=q \neq Lt)$.

Hint $Unfold\ Qeq\ Qlt\ Qle$: $qarith$.
Hint $Extern\ 5\ (?X1 \neq ?X2) \Rightarrow intro; discriminate$: $qarith$.

82.2 Properties of equality.

Theorem $Qeq\_refl : \forall\ x, x == x$.

Theorem $Qeq\_sym : \forall\ x\ y, x == y \rightarrow y == x$.

Theorem $Qeq\_trans : \forall\ x\ y\ z, x == y \rightarrow y == z \rightarrow x == z$.

Furthermore, this equality is decidable:

Theorem $Qeq\_dec : \forall\ x\ y, \{x==y\} + \{\tilde{}\ x==y\}$.

We now consider Q seen as a setoid.

Definition $Q\_Setoid : Setoid\_Theory\ Q\ Qeq$.

$Add\ Setoid\ Q\ Qeq\ Q\_Setoid$ as $Qsetoid$.

Hint $Resolve\ (Seq\_refl\ Q\ Qeq\ Q\_Setoid)$: $qarith$.
Hint $Resolve\ (Seq\_sym\ Q\ Qeq\ Q\_Setoid)$: $qarith$.
Hint $Resolve\ (Seq\_trans\ Q\ Qeq\ Q\_Setoid)$: $qarith$.

82.3 Addition, multiplication and opposite

The addition, multiplication and opposite are defined in the straightforward way:

Definition $Qplus\ (x\ y : Q) :=$
$(Qnum\ x \times QDen\ y + Qnum\ y \times QDen\ x) \#\ (Qden\ x \times Qden\ y)$.

Definition $Qmult\ (x\ y : Q) := (Qnum\ x \times Qnum\ y) \#\ (Qden\ x \times Qden\ y)$.

Definition $Qopp\ (x : Q) := (-\ Qnum\ x) \#\ (Qden\ x)$.

Definition $Qminus\ (x\ y : Q) := Qplus\ x\ (Qopp\ y)$.

Definition $Qinv\ (x : Q) :=$
match $Qnum\ x$ with

```
  | Z0 ⇒ 0
  | Zpos p ⇒ (QDen x)#p
  | Zneg p ⇒ (Zneg (Qden x))#p
  end.

Definition Qdiv (x y : Q) := Qmult x (Qinv y).

Infix "+" := Qplus : Q_scope.
Notation "- x" := (Qopp x) : Q_scope.
Infix "-" := Qminus : Q_scope.
Infix "×" := Qmult : Q_scope.
Notation "/ x" := (Qinv x) : Q_scope.
Infix "/" := Qdiv : Q_scope.
```

A light notation for *Zpos*

```
Notation " ' x " := (Zpos x) (at level 20, no associativity) : Z_scope.
```

82.4 Setoid compatibility results

```
Add Morphism Qplus : Qplus_comp.
  Open Scope Z_scope.
  Close Scope Z_scope.

Add Morphism Qopp : Qopp_comp.
  Open Scope Z_scope.
  Close Scope Z_scope.

Add Morphism Qminus : Qminus_comp.

Add Morphism Qmult : Qmult_comp.
  Open Scope Z_scope.
  Close Scope Z_scope.

Add Morphism Qinv : Qinv_comp.
  Open Scope Z_scope.
  Close Scope Z_scope.

Add Morphism Qdiv : Qdiv_comp.

Add Morphism Qle with signature Qeq ==> Qeq ==> iff as Qle_comp.

  Open Scope Z_scope.
  Close Scope Z_scope.

Add Morphism Qlt with signature Qeq ==> Qeq ==> iff as Qlt_comp.

  Open Scope Z_scope.
  Close Scope Z_scope.

Lemma Qcompare_egal_dec: ∀ n m p q : Q,
  (n<m → p<q) → (n==m → p==q) → (n>m → p>q) → ((n?=m) = (p?=q)).
```

Add Morphism Qcompare : Qcompare_comp.

0 and 1 are apart

Lemma *Q_apart_0_1* : $\neg\ 1 == 0$.

82.5 Properties of *Qadd*

Addition is associative:

Theorem *Qplus_assoc* : $\forall\ x\ y\ z,\ x+(y+z)==(x+y)+z$.

0 is a neutral element for addition:

Lemma *Qplus_0_l* : $\forall\ x,\ 0+x == x$.

Lemma *Qplus_0_r* : $\forall\ x,\ x+0 == x$.

Commutativity of addition:

Theorem *Qplus_comm* : $\forall\ x\ y,\ x+y == y+x$.

82.6 Properties of *Qopp*

Lemma *Qopp_involutive* : $\forall\ q,\ - -q == q$.

Theorem *Qplus_opp_r* : $\forall\ q,\ q+(-q) == 0$.

82.7 Properties of *Qmult*

Multiplication is associative:

Theorem *Qmult_assoc* : $\forall\ n\ m\ p,\ n*(m\times p)==(n\times m)*p$.

1 is a neutral element for multiplication:

Lemma *Qmult_1_l* : $\forall\ n,\ 1\times n == n$.

Theorem *Qmult_1_r* : $\forall\ n,\ n\times 1==n$.

Commutativity of multiplication

Theorem *Qmult_comm* : $\forall\ x\ y,\ x\times y==y\times x$.

Distributivity over *Qadd*

Theorem *Qmult_plus_distr_r* : $\forall\ x\ y\ z,\ x*(y+z)==(x\times y)+(x\times z)$.

Theorem *Qmult_plus_distr_l* : $\forall\ x\ y\ z,\ (x+y)*z==(x\times z)+(y\times z)$.

Integrality

Theorem *Qmult_integral* : $\forall\ x\ y,\ x\times y==0 \rightarrow x==0 \vee y==0$.

Theorem *Qmult_integral_l* : $\forall\ x\ y,\ \neg\ x == 0 \rightarrow x\times y == 0 \rightarrow y == 0$.

82.8 Inverse and division.

Theorem $Qmult\_inv\_r : \forall\ x, \neg\ x == 0 \rightarrow x*(/x) == 1.$

Lemma $Qinv\_mult\_distr : \forall\ p\ q, /\ (p \times q) == /p \times /q.$

Theorem $Qdiv\_mult\_l : \forall\ x\ y, \neg\ y == 0 \rightarrow (x \times y)/y == x.$

Theorem $Qmult\_div\_r : \forall\ x\ y, \neg\ y == 0 \rightarrow y*(x/y) == x.$

82.9 Properties of order upon Q.

Lemma $Qle\_refl : \forall\ x, x \leq x.$

Lemma $Qle\_antisym : \forall\ x\ y, x \leq y \rightarrow y \leq x \rightarrow x == y.$

Lemma $Qle\_trans : \forall\ x\ y\ z, x \leq y \rightarrow y \leq z \rightarrow x \leq z.$
Open Scope Z_scope.
Close Scope Z_scope.

Lemma $Qlt\_not\_eq : \forall\ x\ y, x < y \rightarrow \neg\ x == y.$

Large = strict or equal

Lemma $Qlt\_le\_weak : \forall\ x\ y, x < y \rightarrow x \leq y.$

Lemma $Qle\_lt\_trans : \forall\ x\ y\ z, x \leq y \rightarrow y < z \rightarrow x < z.$
Open Scope Z_scope.
Close Scope Z_scope.

Lemma $Qlt\_le\_trans : \forall\ x\ y\ z, x < y \rightarrow y \leq z \rightarrow x < z.$
Open Scope Z_scope.
Close Scope Z_scope.

Lemma $Qlt\_trans : \forall\ x\ y\ z, x < y \rightarrow y < z \rightarrow x < z.$

$x < y$ iff $\tilde{}\ (y \leq x)$

Lemma $Qnot\_lt\_le : \forall\ x\ y, \neg\ x < y \rightarrow y \leq x.$

Lemma $Qnot\_le\_lt : \forall\ x\ y, \neg\ x \leq y \rightarrow y < x.$

Lemma $Qlt\_not\_le : \forall\ x\ y, x < y \rightarrow \neg\ y \leq x.$

Lemma $Qle\_not\_lt : \forall\ x\ y, x \leq y \rightarrow \neg\ y < x.$

Lemma $Qle\_lt\_or\_eq : \forall\ x\ y, x \leq y \rightarrow x < y \vee x == y.$

Some decidability results about orders.

Lemma $Q\_dec : \forall\ x\ y, \{x < y\} + \{y < x\} + \{x == y\}.$

Lemma $Qlt\_le\_dec : \forall\ x\ y, \{x < y\} + \{y \leq x\}.$

Compatibility of operations with respect to order.

Lemma $Qopp\_le\_compat : \forall\ p\ q, p \leq q \rightarrow -q \leq -p.$

Lemma *Qle_minus_iff* : $\forall\ p\ q,\ p \leq q \leftrightarrow 0 \leq q+\text{-}p$.

Lemma *Qlt_minus_iff* : $\forall\ p\ q,\ p < q \leftrightarrow 0 < q+\text{-}p$.

Lemma *Qplus_le_compat* :
$\forall\ x\ y\ z\ t,\ x \leq y \rightarrow z \leq t \rightarrow x+z \leq y+t$.
Open Scope Z_scope.
Close Scope Z_scope.

Lemma *Qmult_le_compat_r* : $\forall\ x\ y\ z,\ x \leq y \rightarrow 0 \leq z \rightarrow x \times z \leq y \times z$.
Open Scope Z_scope.
Close Scope Z_scope.

Lemma *Qmult_lt_0_le_reg_r* : $\forall\ x\ y\ z,\ 0 < z \rightarrow x \times z \leq y \times z \rightarrow x \leq y$.
Open Scope Z_scope.
Close Scope Z_scope.

Lemma *Qmult_lt_compat_r* : $\forall\ x\ y\ z,\ 0 < z \rightarrow x < y \rightarrow x \times z < y \times z$.
Open Scope Z_scope.
Close Scope Z_scope.

82.10 Rational to the n-th power

Fixpoint *Qpower* $(q{:}Q)(n{:}nat)\ \{\ struct\ n\ \} : Q :=$
match n with
$\mid O \Rightarrow 1$
$\mid S\ n \Rightarrow q \times (Qpower\ q\ n)$
end.

Notation " $q\ \hat{}\ n$ " := $(Qpower\ q\ n) : Q\_scope$.

Lemma *Qpower_1* : $\forall\ n,\ 1\hat{}n == 1$.

Lemma *Qpower_0* : $\forall\ n,\ n \neq O \rightarrow 0\hat{}n == 0$.

Lemma *Qpower_pos* : $\forall\ p\ n,\ 0 \leq p \rightarrow 0 \leq p\hat{}n$.

Lemma *Qinv_power_n* : $\forall\ n\ p,\ (1\#p)\hat{}n == /(inject\_Z\ ('p))\hat{}n$.

Chapter 83

Module Coq.QArith.QArith

Require Export *QArith_base*.
Require Export *Qring*.
Require Export *Qreduction*.

Chapter 84

Module Coq.QArith.Qcanon

Require Import *Field*.
Require Import *QArith*.
Require Import *Znumtheory*.
Require Import *Eqdep_dec*.

Qc : A canonical representation of rational numbers. based on the setoid representation *Q*.

Record *Qc* : Set := *Qcmake* { *this* :> *Q* ; *canon* : *Qred this* = *this* }.

Delimit Scope Qc_scope with *Qc*.
Open Scope Qc_scope.

Lemma *Qred_identity* :
  ∀ *q*:*Q*, *Zgcd* (*Qnum q*) (*QDen q*) = 1%*Z* → *Qred q* = *q*.

Lemma *Qred_identity2* :
  ∀ *q*:*Q*, *Qred q* = *q* → *Zgcd* (*Qnum q*) (*QDen q*) = 1%*Z*.

Lemma *Qred_iff* : ∀ *q*:*Q*, *Qred q* = *q* ↔ *Zgcd* (*Qnum q*) (*QDen q*) = 1%*Z*.

Lemma *Qred_involutive* : ∀ *q*:*Q*, *Qred* (*Qred q*) = *Qred q*.

Definition *Q2Qc* (*q*:*Q*) : *Qc* := *Qcmake* (*Qred q*) (*Qred_involutive q*).
Notation " !! " := *Q2Qc* : *Qc_scope*.

Lemma *Qc_is_canon* : ∀ *q q'* : *Qc*, *q* == *q'* → *q* = *q'*.
Hint *Resolve Qc_is_canon*.

Notation " 0 " := (!!0) : *Qc_scope*.
Notation " 1 " := (!!1) : *Qc_scope*.

Definition *Qcle* (*x y* : *Qc*) := (*x* ≤ *y*)%*Q*.
Definition *Qclt* (*x y* : *Qc*) := (*x* < *y*)%*Q*.
Notation *Qcgt* := (fun *x y* : *Qc* ⇒ *Qlt y x*).
Notation *Qcge* := (fun *x y* : *Qc* ⇒ *Qle y x*).
Infix "<" := *Qclt* : *Qc_scope*.
Infix "≤" := *Qcle* : *Qc_scope*.
Infix ">" := *Qcgt* : *Qc_scope*.
Infix "≥" := *Qcge* : *Qc_scope*.

Notation "$x \leq y \leq z$" := $(x{\leq}y{\wedge}y{\leq}z)$: Qc_scope.

Definition $Qccompare$ $(p\ q : Qc) := (Qcompare\ p\ q)$.
Notation "p ?= q" := $(Qccompare\ p\ q)$: Qc_scope.

Lemma $Qceq\_alt$: $\forall\ p\ q, (p = q) \leftrightarrow (p\ ?=\ q) = Eq$.

Lemma $Qclt\_alt$: $\forall\ p\ q, (p{<}q) \leftrightarrow (p?{=}q = Lt)$.

Lemma $Qcgt\_alt$: $\forall\ p\ q, (p{>}q) \leftrightarrow (p?{=}q = Gt)$.

Lemma Qle_alt : $\forall\ p\ q, (p{\leq}q) \leftrightarrow (p?{=}q \neq Gt)$.

Lemma Qge_alt : $\forall\ p\ q, (p{\geq}q) \leftrightarrow (p?{=}q \neq Lt)$.

equality on Qc is decidable:

Theorem Qc_eq_dec : $\forall\ x\ y{:}Qc, \{x{=}y\} + \{x{\neq}y\}$.

The addition, multiplication and opposite are defined in the straightforward way:

Definition $Qcplus$ $(x\ y : Qc) := !!(x{+}y)$.
Infix "+" := $Qcplus$: Qc_scope.
Definition $Qcmult$ $(x\ y : Qc) := !!(x{\times}y)$.
Infix "×" := $Qcmult$: Qc_scope.
Definition $Qcopp$ $(x : Qc) := !!(\text{-}x)$.
Notation "- x" := $(Qcopp\ x)$: Qc_scope.
Definition $Qcminus$ $(x\ y : Qc) := x{+}\text{-}y$.
Infix "-" := $Qcminus$: Qc_scope.
Definition $Qcinv$ $(x : Qc) := !!(/x)$.
Notation "/ x" := $(Qcinv\ x)$: Qc_scope.
Definition $Qcdiv$ $(x\ y : Qc) := x{*}/y$.
Infix "/" := $Qcdiv$: Qc_scope.

0 and 1 are apart

Lemma $Q\_apart\_0\_1$: $1 \neq 0$.

Ltac qc := match $goal$ with
| $q{:}Qc \vdash \_ \Rightarrow destruct\ q$; qc
| $\_ \Rightarrow apply\ Qc\_is\_canon$; $simpl$; $repeat\ rewrite\ Qred\_correct$
end.

$Opaque\ Qred$.

Addition is associative:

Theorem $Qcplus\_assoc$: $\forall\ x\ y\ z, x{+}(y{+}z){=}(x{+}y){+}z$.

0 is a neutral element for addition:

Lemma $Qcplus\_0\_l$: $\forall\ x, 0{+}x = x$.

Lemma $Qcplus\_0\_r$: $\forall\ x, x{+}0 = x$.

Commutativity of addition:

Theorem $Qcplus\_comm$: $\forall\ x\ y, x{+}y = y{+}x$.

Properties of *Qopp*

Lemma *Qcopp_involutive* : $\forall\ q, - -q = q$.

Theorem *Qcplus_opp_r* : $\forall\ q, q+(-q) = 0$.

Multiplication is associative:

Theorem *Qcmult_assoc* : $\forall\ n\ m\ p, n*(m\times p)=(n\times m)*p$.

1 is a neutral element for multiplication:

Lemma *Qcmult_1_l* : $\forall\ n, 1\times n = n$.

Theorem *Qcmult_1_r* : $\forall\ n, n\times 1=n$.

Commutativity of multiplication

Theorem *Qcmult_comm* : $\forall\ x\ y, x\times y=y\times x$.

Distributivity

Theorem *Qcmult_plus_distr_r* : $\forall\ x\ y\ z, x*(y+z)=(x\times y)+(x\times z)$.

Theorem *Qcmult_plus_distr_l* : $\forall\ x\ y\ z, (x+y)*z=(x\times z)+(y\times z)$.

Integrality

Theorem *Qcmult_integral* : $\forall\ x\ y, x\times y=0 \rightarrow x=0 \vee y=0$.

Theorem *Qcmult_integral_l* : $\forall\ x\ y, \neg\ x = 0 \rightarrow x\times y = 0 \rightarrow y = 0$.

Inverse and division.

Theorem *Qcmult_inv_r* : $\forall\ x, x\neq 0 \rightarrow x*(/x) = 1$.

Theorem *Qcmult_inv_l* : $\forall\ x, x\neq 0 \rightarrow (/x)*x = 1$.

Lemma *Qcinv_mult_distr* : $\forall\ p\ q, /\ (p\ \times\ q) = /p\ \times\ /q$.

Theorem *Qcdiv_mult_l* : $\forall\ x\ y, y\neq 0 \rightarrow (x\times y)/y = x$.

Theorem *Qcmult_div_r* : $\forall\ x\ y, \neg\ y = 0 \rightarrow y*(x/y) = x$.

Properties of order upon Q.

Lemma *Qcle_refl* : $\forall\ x, x\leq x$.

Lemma *Qcle_antisym* : $\forall\ x\ y, x\leq y \rightarrow y\leq x \rightarrow x=y$.

Lemma *Qcle_trans* : $\forall\ x\ y\ z, x\leq y \rightarrow y\leq z \rightarrow x\leq z$.

Lemma *Qclt_not_eq* : $\forall\ x\ y, x<y \rightarrow x\neq y$.

Large = strict or equal

Lemma *Qclt_le_weak* : $\forall\ x\ y, x<y \rightarrow x\leq y$.

Lemma *Qcle_lt_trans* : $\forall\ x\ y\ z, x\leq y \rightarrow y<z \rightarrow x<z$.

Lemma *Qclt_le_trans* : $\forall\ x\ y\ z, x<y \rightarrow y\leq z \rightarrow x<z$.

Lemma *Qlt_trans* : $\forall\ x\ y\ z, x<y \rightarrow y<z \rightarrow x<z$.

$x<y$ iff $\tilde{}(y \leq x)$

Lemma $Qcnot\_lt\_le : \forall\ x\ y, \neg\ x<y \rightarrow y \leq x.$

Lemma $Qcnot\_le\_lt : \forall\ x\ y, \neg\ x \leq y \rightarrow y<x.$

Lemma $Qclt\_not\_le : \forall\ x\ y, x<y \rightarrow \neg\ y \leq x.$

Lemma $Qcle\_not\_lt : \forall\ x\ y, x \leq y \rightarrow \neg\ y<x.$

Lemma $Qcle\_lt\_or\_eq : \forall\ x\ y, x \leq y \rightarrow x<y \vee x==y.$

Some decidability results about orders.

Lemma $Qc\_dec : \forall\ x\ y, \{x<y\} + \{y<x\} + \{x=y\}.$

Lemma $Qclt\_le\_dec : \forall\ x\ y, \{x<y\} + \{y \leq x\}.$

Compatibility of operations with respect to order.

Lemma $Qcopp\_le\_compat : \forall\ p\ q, p \leq q \rightarrow -q \leq -p.$

Lemma $Qcle\_minus\_iff : \forall\ p\ q, p \leq q \leftrightarrow 0 \leq q+-p.$

Lemma $Qclt\_minus\_iff : \forall\ p\ q, p < q \leftrightarrow 0 < q+-p.$

Lemma $Qcplus\_le\_compat :$
$\forall\ x\ y\ z\ t, x \leq y \rightarrow z \leq t \rightarrow x+z \leq y+t.$

Lemma $Qcmult\_le\_compat\_r : \forall\ x\ y\ z, x \leq y \rightarrow 0 \leq z \rightarrow x \times z \leq y \times z.$

Lemma $Qcmult\_lt\_0\_le\_reg\_r : \forall\ x\ y\ z, 0 < z \rightarrow x \times z \leq y \times z \rightarrow x \leq y.$

Lemma $Qcmult\_lt\_compat\_r : \forall\ x\ y\ z, 0 < z \rightarrow x < y \rightarrow x \times z < y \times z.$

Rational to the n-th power

Fixpoint $Qcpower\ (q{:}Qc)(n{:}nat)\ \{\ struct\ n\ \} : Qc :=$
  match n with
    | $O \Rightarrow 1$
    | $S\ n \Rightarrow q \times (Qcpower\ q\ n)$
  end.

Notation " $q\ \hat{}\ n$ " := $(Qcpower\ q\ n) : Qc\_scope.$

Lemma $Qcpower\_1 : \forall\ n, 1\hat{}n = 1.$

Lemma $Qcpower\_0 : \forall\ n, n \neq O \rightarrow 0\hat{}n = 0.$

Lemma $Qpower\_pos : \forall\ p\ n, 0 \leq p \rightarrow 0 \leq p\hat{}n.$

And now everything is easier concerning tactics:

A ring tactic for rational numbers

Definition $Qc\_eq\_bool\ (x\ y : Qc) :=$
  if $Qc\_eq\_dec\ x\ y$ then $true$ else $false.$

Lemma $Qc\_eq\_bool\_correct : \forall\ x\ y : Qc, Qc\_eq\_bool\ x\ y = true \rightarrow x=y.$

Definition $Qcrt : ring\_theory\ 0\ 1\ Qcplus\ Qcmult\ Qcminus\ Qcopp\ (eq(A{:=}Qc)).$

Definition $Qcft$:
$field\_theory\ 0\%Qc\ 1\%Qc\ Qcplus\ Qcmult\ Qcminus\ Qcopp\ Qcdiv\ Qcinv\ (eq(A{:=}Qc))$.

$Add\ Field\ Qcfield\ :\ Qcft$.

A field tactic for rational numbers

$Example\ test\_field\ :\ (\forall\ x\ \ y\ :\ Qc,\ y{\neq}0 \rightarrow (x/y)^{*}y = x)\%Qc$.

Chapter 85

Module Coq.QArith.Qreals

Require Export *Rbase*.
Require Export *QArith_base*.

A field tactic for rational numbers.

Since field cannot operate on setoid datatypes (yet?), we translate Q goals into reals before applying field.

Lemma *IZR_nz* : $\forall\ p : positive, IZR\ (Zpos\ p) \neq 0\%R$.

Hint Immediate *IZR_nz*.
Hint *Resolve Rmult_integral_contrapositive*.

Definition *Q2R* $(x : Q) : R := (IZR\ (Qnum\ x) \times /\ IZR\ (QDen\ x))\%R$.

Lemma *eqR_Qeq* : $\forall\ x\ y : Q, Q2R\ x = Q2R\ y \rightarrow x{==}y$.

Lemma *Qeq_eqR* : $\forall\ x\ y : Q, x{==}y \rightarrow Q2R\ x = Q2R\ y$.

Lemma *Rle_Qle* : $\forall\ x\ y : Q, (Q2R\ x \leq Q2R\ y)\%R \rightarrow x{\leq}y$.

Lemma *Qle_Rle* : $\forall\ x\ y : Q, x{\leq}y \rightarrow (Q2R\ x \leq Q2R\ y)\%R$.

Lemma *Rlt_Qlt* : $\forall\ x\ y : Q, (Q2R\ x < Q2R\ y)\%R \rightarrow x{<}y$.

Lemma *Qlt_Rlt* : $\forall\ x\ y : Q, x{<}y \rightarrow (Q2R\ x < Q2R\ y)\%R$.

Lemma *Q2R_plus* : $\forall\ x\ y : Q, Q2R\ (x{+}y) = (Q2R\ x + Q2R\ y)\%R$.

Lemma *Q2R_mult* : $\forall\ x\ y : Q, Q2R\ (x{\times}y) = (Q2R\ x \times Q2R\ y)\%R$.

Lemma *Q2R_opp* : $\forall\ x : Q, Q2R\ (-\ x) = (-\ Q2R\ x)\%R$.

Lemma *Q2R_minus* : $\forall\ x\ y : Q, Q2R\ (x{-}y) = (Q2R\ x - Q2R\ y)\%R$.

Lemma *Q2R_inv* : $\forall\ x : Q, \neg\ x{==}0\#1 \rightarrow Q2R\ (/x) = (/\ Q2R\ x)\%R$.

Lemma *Q2R_div* :
$\forall\ x\ y : Q, \neg\ y{==}0\#1 \rightarrow Q2R\ (x/y) = (Q2R\ x\ /\ Q2R\ y)\%R$.

Hint *Rewrite Q2R_plus Q2R_mult Q2R_opp Q2R_minus Q2R_inv Q2R_div : q2r_simpl*.

Ltac *QField* := *apply eqR_Qeq*; *autorewrite* with *q2r_simpl*; *try field*; *auto*.

Examples of use:

Goal $\forall\ x\ y\ z : Q, (x+y)*z == (x\times z)+(y\times z)$.

Goal $\forall\ x\ y : Q, \neg\ y==0\#1 \rightarrow (x/y)*y == x$.

Chapter 86

Module Coq.QArith.Qreduction

Normalisation functions for rational numbers.

```
Require Export QArith_base.
Require Import Znumtheory.
```

First, a function that (tries to) build a positive back from a Z.

```
Definition Z2P (z : Z) :=
  match z with
  | Z0 ⇒ 1%positive
  | Zpos p ⇒ p
  | Zneg p ⇒ p
  end.
```

```
Lemma Z2P_correct : ∀ z : Z, (0 < z)%Z → Zpos (Z2P z) = z.
```

```
Lemma Z2P_correct2 : ∀ z : Z, 0%Z ≠ z → Zpos (Z2P z) = Zabs z.
```

Simplification of fractions using *Zgcd*. This version can compute within Coq.

```
Definition Qred (q:Q) :=
  let (q1,q2) := q in
  let (r1,r2) := snd (Zggcd q1 ('q2))
  in r1#(Z2P r2).
```

```
Lemma Qred_correct : ∀ q, (Qred q) == q.
  Open Scope Z_scope.
  Close Scope Z_scope.
```

```
Lemma Qred_complete : ∀ p q, p==q → Qred p = Qred q.
  Open Scope Z_scope.
  Close Scope Z_scope.
```

```
Add Morphism Qred : Qred_comp.
```

```
Definition Qplus' (p q : Q) := Qred (Qplus p q).
Definition Qmult' (p q : Q) := Qred (Qmult p q).
```

```
Lemma Qplus'_correct : ∀ p q : Q, (Qplus' p q)==(Qplus p q).
```

Lemma *Qmult'_correct* : $\forall$ *p q* : *Q*, (*Qmult' p q*)==(*Qmult p q*).

Add Morphism Qplus' : *Qplus'_comp*.

Add Morphism Qmult' : *Qmult'_comp*.

Chapter 87

Module Coq.QArith.Qring

```
Require Export Ring.
Require Export QArith_base.
```

87.1 A ring tactic for rational numbers

```
Definition Qeq_bool (x y : Q) :=
  if Qeq_dec x y then true else false.

Lemma Qeq_bool_correct : ∀ x y : Q, Qeq_bool x y = true → x==y.

Definition Qsrt : ring_theory 0 1 Qplus Qmult Qminus Qopp Qeq.

Ltac isQcst t :=
  match t with
  | inject_Z ?z ⇒ isZcst z
  | Qmake ?n ?d ⇒
    match isZcst n with
      true ⇒ isPcst d
    | _ ⇒ false
    end
  | _ ⇒ false
  end.

Ltac Qcst t :=
  match isQcst t with
    true ⇒ t
    | _ ⇒ NotConstant
  end.

Add Ring Qring : Qsrt (decidable Qeq_bool_correct, constants [Qcst]).
```

Exemple of use:

```
Section Examples.

Let ex1 : ∀ x y z : Q, (x+y)*z == (x×z)+(y×z).
```

Let $ex2 : \forall\ x\ y : Q, x+y == y+x$.

Let $ex3 : \forall\ x\ y\ z : Q, (x+y)+z == x+(y+z)$.

Let $ex4 : (inject\_Z\ 1)+(inject\_Z\ 1)==(inject\_Z\ 2)$.

Let $ex5 : 1+1 == 2\#1$.

Let $ex6 : (1\#1)+(1\#1) == 2\#1$.

Let $ex7 : \forall\ x : Q, x\text{-}x== 0\#1$.

End *Examples*.

Lemma $Qopp\_plus : \forall\ a\ b, \text{-}(a+b) == \text{-}a + \text{-}b$.

Lemma $Qopp\_opp : \forall\ q, \text{- -}q==q$.

Chapter 88

Module Coq.Reals.Alembert

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *SeqProp*.
Require Import *PartSum*.
Require Import *Max*.

Open Local *Scope R_scope*.

Lemma *Alembert_C1* :
  $\forall$ *An*:*nat* $\to$ *R*,
    ($\forall$ *n*:*nat*, 0 $<$ *An n*) $\to$
    Un_cv (fun *n*:*nat* $\Rightarrow$ *Rabs* (*An* (*S n*) / *An n*)) 0 $\to$
    sigT (fun *l*:*R* $\Rightarrow$ *Un_cv* (fun *N*:*nat* $\Rightarrow$ *sum_f_R0 An N*) *l*).

Lemma *Alembert_C2* :
  $\forall$ *An*:*nat* $\to$ *R*,
    ($\forall$ *n*:*nat*, *An n* $\neq$ 0) $\to$
    Un_cv (fun *n*:*nat* $\Rightarrow$ *Rabs* (*An* (*S n*) / *An n*)) 0 $\to$
    sigT (fun *l*:*R* $\Rightarrow$ *Un_cv* (fun *N*:*nat* $\Rightarrow$ *sum_f_R0 An N*) *l*).

Lemma *AlembertC3_step1* :
  $\forall$ (*An*:*nat* $\to$ *R*) (*x*:*R*),
    x $\neq$ 0 $\to$
    ($\forall$ *n*:*nat*, *An n* $\neq$ 0) $\to$
    Un_cv (fun *n*:*nat* $\Rightarrow$ *Rabs* (*An* (*S n*) / *An n*)) 0 $\to$
    sigT (fun *l*:*R* $\Rightarrow$ *Pser An x l*).

Lemma *AlembertC3_step2* :
  $\forall$ (*An*:*nat* $\to$ *R*) (*x*:*R*), *x* = 0 $\to$ *sigT* (fun *l*:*R* $\Rightarrow$ *Pser An x l*).

An useful criterion of convergence for power series
Theorem *Alembert_C3* :
  $\forall$ (*An*:*nat* $\to$ *R*) (*x*:*R*),
    ($\forall$ *n*:*nat*, *An n* $\neq$ 0) $\to$
    Un_cv (fun *n*:*nat* $\Rightarrow$ *Rabs* (*An* (*S n*) / *An n*)) 0 $\to$

sigT (fun *l*:*R* ⇒ *Pser An x l*).

Lemma *Alembert_C4* :
∀ (*An*:*nat* → *R*) (*k*:*R*),
0 ≤ *k* < 1 →
(∀ *n*:*nat*, 0 < *An n*) →
Un_cv (fun *n*:*nat* ⇒ *Rabs* (*An* (*S n*) / *An n*)) *k* →
sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *sum_f_R0 An N*) *l*).

Lemma *Alembert_C5* :
∀ (*An*:*nat* → *R*) (*k*:*R*),
0 ≤ *k* < 1 →
(∀ *n*:*nat*, *An n* ≠ 0) →
Un_cv (fun *n*:*nat* ⇒ *Rabs* (*An* (*S n*) / *An n*)) *k* →
sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *sum_f_R0 An N*) *l*).

Convergence of power series in D(O,1/k) k=0 is described in Alembert_C3
Lemma *Alembert_C6* :
∀ (*An*:*nat* → *R*) (*x k*:*R*),
0 < *k* →
(∀ *n*:*nat*, *An n* ≠ 0) →
Un_cv (fun *n*:*nat* ⇒ *Rabs* (*An* (*S n*) / *An n*)) *k* →
Rabs x < / *k* → *sigT* (fun *l*:*R* ⇒ *Pser An x l*).

Chapter 89

Module Coq.Reals.AltSeries

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *SeqProp*.
Require Import *PartSum*.
Require Import *Max*.
Open Local *Scope R_scope*.

89.1 Formalization of alternated series

Definition *tg_alt* (*Un*:*nat* → *R*) (*i*:*nat*) : *R* := (-1) ^ *i* × *Un i*.
Definition *positivity_seq* (*Un*:*nat* → *R*) : Prop := ∀ *n*:*nat*, 0 ≤ *Un n*.

Lemma *CV_ALT_step0* :
∀ *Un*:*nat* → *R*,
Un_decreasing Un →
Un_growing (fun *N*:*nat* ⇒ *sum_f_R0* (*tg_alt Un*) (*S* (2 × *N*))).

Lemma *CV_ALT_step1* :
∀ *Un*:*nat* → *R*,
Un_decreasing Un →
Un_decreasing (fun *N*:*nat* ⇒ *sum_f_R0* (*tg_alt Un*) (2 × *N*)).

Lemma *CV_ALT_step2* :
∀ (*Un*:*nat* → *R*) (*N*:*nat*),
Un_decreasing Un →
positivity_seq Un →
sum_f_R0 (fun *i*:*nat* ⇒ *tg_alt Un* (*S i*)) (*S* (2 × *N*)) ≤ 0.

A more general inequality
Lemma *CV_ALT_step3* :
∀ (*Un*:*nat* → *R*) (*N*:*nat*),
Un_decreasing Un →
positivity_seq Un → *sum_f_R0* (fun *i*:*nat* ⇒ *tg_alt Un* (*S i*)) *N* ≤ 0.

Lemma CV_ALT_step4 :
$\forall\ Un{:}nat \to R,$
$Un\_decreasing\ Un \to$
$positivity\_seq\ Un \to$
$has\_ub\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ (tg\_alt\ Un)\ (S\ (2 \times N)))$.

This lemma gives an interesting result about alternated series
Lemma CV_ALT :
$\forall\ Un{:}nat \to R,$
$Un\_decreasing\ Un \to$
$positivity\_seq\ Un \to$
$Un\_cv\ Un\ 0 \to$
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ (tg\_alt\ Un)\ N)\ l)$.

89.2 Convergence of alternated series

Theorem $alternated\_series$:
$\forall\ Un{:}nat \to R,$
$Un\_decreasing\ Un \to$
$Un\_cv\ Un\ 0 \to$
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ (tg\_alt\ Un)\ N)\ l)$.

Theorem $alternated\_series\_ineq$:
$\forall\ (Un{:}nat \to R)\ (l{:}R)\ (N{:}nat),$
$Un\_decreasing\ Un \to$
$Un\_cv\ Un\ 0 \to$
$Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ (tg\_alt\ Un)\ N)\ l \to$
$sum\_f\_R0\ (tg\_alt\ Un)\ (S\ (2 \times N)) \leq l \leq sum\_f\_R0\ (tg\_alt\ Un)\ (2 \times N)$.

89.3 Application : construction of PI

Definition $PI\_tg\ (n{:}nat) := /\ INR\ (2 \times n + 1)$.

Lemma $PI\_tg\_pos : \forall\ n{:}nat, 0 \leq PI\_tg\ n$.

Lemma $PI\_tg\_decreasing : Un\_decreasing\ PI\_tg$.

Lemma $PI\_tg\_cv : Un\_cv\ PI\_tg\ 0$.

Lemma $exist\_PI$:
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ (tg\_alt\ PI\_tg)\ N)\ l)$.

Now, PI is defined
Definition $PI : R :=$
$4 \times \mathsf{match}\ exist\_PI\ \mathsf{with}$
$\mid existT\ a\ b \Rightarrow a$
end.

We can get an approximation of PI with the following inequality
Lemma PI_ineq :

$\forall\ N{:}nat,$
$sum\_f\_R0\ (tg\_alt\ PI\_tg)\ (S\ (2 \times N)) \leq PI\ /\ 4 \leq$
$sum\_f\_R0\ (tg\_alt\ PI\_tg)\ (2 \times N).$

Lemma $PI\_RGT\_0 : 0 < PI.$

Chapter 90

Module Coq.Reals.ArithProp

Require Import *Rbase*.
Require Import *Rbasic_fun*.
Require Import *Even*.
Require Import *Div2*.
Require Import *ArithRing*.

Open Local *Scope Z_scope*.
Open Local *Scope R_scope*.

Lemma *minus_neq_O* : $\forall\ n\ i{:}nat,\ (i < n)\%nat \rightarrow (n\ -\ i)\%nat \neq 0\%nat$.

Lemma *le_minusni_n* : $\forall\ n\ i{:}nat,\ (i \leq n)\%nat \rightarrow (n\ -\ i \leq n)\%nat$.

Lemma *lt_minus_O_lt* : $\forall\ m\ n{:}nat,\ (m < n)\%nat \rightarrow (0 < n\ -\ m)\%nat$.

Lemma *even_odd_cor* :
$\forall\ n{:}nat, \exists\ p\ :\ nat,\ n = (2 \times p)\%nat \vee n = S\ (2 \times p)$.

Lemma *le_double* : $\forall\ m\ n{:}nat,\ (2 \times m \leq 2 \times n)\%nat \rightarrow (m \leq n)\%nat$.

Here, we have the euclidian division
This lemma is used in the proof of sin_eq_0 : (sin x)=0<->x=kPI
Lemma *euclidian_division* :
$\forall\ x\ y{:}R,$
$y \neq 0 \rightarrow$
$\exists\ k\ :\ Z,\ (\exists\ r\ :\ R,\ x = IZR\ k \times y\ +\ r \wedge 0 \leq r < Rabs\ y)$.

Lemma *tech8* : $\forall\ n\ i{:}nat,\ (n \leq S\ n\ +\ i)\%nat$.

Chapter 91

Module Coq.Reals.Binomial

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *PartSum*.
Open Local *Scope R_scope*.

Definition $C\ (n\ p{:}nat) : R :=$
$INR\ (fact\ n)\ /\ (INR\ (fact\ p) \times INR\ (fact\ (n\ -\ p)))$.

Lemma *pascal_step1* : $\forall\ n\ i{:}nat, (i \leq n)\%nat \rightarrow C\ n\ i = C\ n\ (n\ -\ i)$.

Lemma *pascal_step2* :
$\forall\ n\ i{:}nat,$
$(i \leq n)\%nat \rightarrow C\ (S\ n)\ i = INR\ (S\ n)\ /\ INR\ (S\ n\ -\ i) \times C\ n\ i$.

Lemma *pascal_step3* :
$\forall\ n\ i{:}nat, (i < n)\%nat \rightarrow C\ n\ (S\ i) = INR\ (n\ -\ i)\ /\ INR\ (S\ i) \times C\ n\ i$.

Lemma *pascal* :
$\forall\ n\ i{:}nat, (i < n)\%nat \rightarrow C\ n\ i + C\ n\ (S\ i) = C\ (S\ n)\ (S\ i)$.

Lemma *binomial* :
$\forall\ (x\ y{:}R)\ (n{:}nat),$
$(x\ +\ y)\ \hat{}\ n = sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow C\ n\ i \times x\ \hat{}\ i \times y\ \hat{}\ (n\ -\ i))\ n$.

Chapter 92

Module Coq.Reals.Cauchy_prod

```
Require Import Rbase.
Require Import Rfunctions.
Require Import Rseries.
Require Import PartSum.
Open Local Scope R_scope.

Lemma sum_N_predN :
  ∀ (An:nat → R) (N:nat),
    (0 < N)%nat → sum_f_R0 An N = sum_f_R0 An (pred N) + An N.

Lemma sum_plus :
  ∀ (An Bn:nat → R) (N:nat),
    sum_f_R0 (fun l:nat ⇒ An l + Bn l) N = sum_f_R0 An N + sum_f_R0 Bn N.

Theorem cauchy_finite :
  ∀ (An Bn:nat → R) (N:nat),
    (0 < N)%nat →
    sum_f_R0 An N × sum_f_R0 Bn N =
    sum_f_R0 (fun k:nat ⇒ sum_f_R0 (fun p:nat ⇒ An p × Bn (k - p)%nat) k) N +
    sum_f_R0
    (fun k:nat ⇒
      sum_f_R0 (fun l:nat ⇒ An (S (l + k)) × Bn (N - l)%nat)
      (pred (N - k))) (pred N).
```

Chapter 93

Module Coq.Reals.Cos_plus

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo_def*.
Require Import *Cos_rel*.
Require Import *Max*. *Open* Local *Scope nat_scope*. *Open* Local *Scope R_scope*.

Definition *Majxy* (*x y*:*R*) (*n*:*nat*) : *R* :=
  Rmax 1 (*Rmax* (*Rabs x*) (*Rabs y*)) ^ (4 × *S n*) / *INR* (*fact n*).

Lemma *Majxy_cv_R0* : ∀ *x y*:*R*, *Un_cv* (*Majxy x y*) 0.

Lemma *reste1_maj* :
  ∀ (*x y*:*R*) (*N*:*nat*),
    (0 < *N*)%*nat* → *Rabs* (*Reste1 x y N*) ≤ *Majxy x y* (*pred N*).

Lemma *reste2_maj* :
  ∀ (*x y*:*R*) (*N*:*nat*), (0 < *N*)%*nat* → *Rabs* (*Reste2 x y N*) ≤ *Majxy x y N*.

Lemma *reste1_cv_R0* : ∀ *x y*:*R*, *Un_cv* (*Reste1 x y*) 0.

Lemma *reste2_cv_R0* : ∀ *x y*:*R*, *Un_cv* (*Reste2 x y*) 0.

Lemma *reste_cv_R0* : ∀ *x y*:*R*, *Un_cv* (*Reste x y*) 0.

Theorem *cos_plus* : ∀ *x y*:*R*, *cos* (*x* + *y*) = *cos x* × *cos y* - *sin x* × *sin y*.

Chapter 94

Module Coq.Reals.Cos_rel

```
Require Import Rbase.
Require Import Rfunctions.
Require Import SeqSeries.
Require Import Rtrigo_def.
Open Local Scope R_scope.

Definition A1 (x:R) (N:nat) : R :=
  sum_f_R0 (fun k:nat ⇒ (-1) ^ k / INR (fact (2 × k)) × x ^ (2 × k)) N.

Definition B1 (x:R) (N:nat) : R :=
  sum_f_R0 (fun k:nat ⇒ (-1) ^ k / INR (fact (2 × k + 1)) × x ^ (2 × k + 1))
    N.

Definition C1 (x y:R) (N:nat) : R :=
  sum_f_R0 (fun k:nat ⇒ (-1) ^ k / INR (fact (2 × k)) × (x + y) ^ (2 × k)) N.

Definition Reste1 (x y:R) (N:nat) : R :=
  sum_f_R0
    (fun k:nat ⇒
       sum_f_R0
         (fun l:nat ⇒
            (-1) ^ S (l + k) / INR (fact (2 × S (l + k))) ×
            x ^ (2 × S (l + k)) × ((-1) ^ (N - l) / INR (fact (2 × (N - l)))) ×
            y ^ (2 × (N - l))) (pred (N - k))) (pred N).

Definition Reste2 (x y:R) (N:nat) : R :=
  sum_f_R0
    (fun k:nat ⇒
       sum_f_R0
         (fun l:nat ⇒
            (-1) ^ S (l + k) / INR (fact (2 × S (l + k) + 1)) ×
            x ^ (2 × S (l + k) + 1) ×
            ((-1) ^ (N - l) / INR (fact (2 × (N - l) + 1))) ×
            y ^ (2 × (N - l) + 1)) (pred (N - k))) (
    pred N).
```

Definition $Reste\ (x\ y{:}R)\ (N{:}nat) : R := Reste2\ x\ y\ N$ - $Reste1\ x\ y\ (S\ N)$.

Theorem cos_plus_form :
$\forall\ (x\ y{:}R)\ (n{:}nat)$,
$(0 < n)\%nat \rightarrow$
$A1\ x\ (S\ n) \times A1\ y\ (S\ n)$ - $B1\ x\ n \times B1\ y\ n + Reste\ x\ y\ n = C1\ x\ y\ (S\ n)$.

Lemma $pow\_sqr : \forall\ (x{:}R)\ (i{:}nat), x\ \hat{}\ (2 \times i) = (x \times x)\ \hat{}\ i$.

Lemma $A1\_cvg : \forall\ x{:}R, Un\_cv\ (A1\ x)\ (cos\ x)$.

Lemma $C1\_cvg : \forall\ x\ y{:}R, Un\_cv\ (C1\ x\ y)\ (cos\ (x + y))$.

Lemma $B1\_cvg : \forall\ x{:}R, Un\_cv\ (B1\ x)\ (sin\ x)$.

Chapter 95

Module Coq.Reals.DiscrR

```
Require Import RIneq.
Require Import Omega. Open Local Scope R_scope.

Lemma Rlt_R0_R2 : 0 < 2.

Lemma Rplus_lt_pos : ∀ x y:R, 0 < x → 0 < y → 0 < x + y.

Lemma IZR_eq : ∀ z1 z2:Z, z1 = z2 → IZR z1 = IZR z2.

Lemma IZR_neq : ∀ z1 z2:Z, z1 ≠ z2 → IZR z1 ≠ IZR z2.

Ltac discrR :=
  try
   match goal with
   | ⊢ (?X1 ≠ ?X2) ⇒
         change 2 with (IZR 2);
         change 1 with (IZR 1);
         change 0 with (IZR 0);
         repeat
           rewrite ← plus_IZR ||
             rewrite ← mult_IZR ||
             rewrite ← Ropp_Ropp_IZR || rewrite Z_R_minus;
         apply IZR_neq; try discriminate
   end.

Ltac prove_sup0 :=
  match goal with
  | ⊢ (0 < 1) ⇒ apply Rlt_0_1
  | ⊢ (0 < ?X1) ⇒
       repeat
        (apply Rmult_lt_0_compat || apply Rplus_lt_pos;
          try apply Rlt_0_1 || apply Rlt_R0_R2)
  | ⊢ (?X1 > 0) ⇒ change (0 < X1) in ⊢ ×; prove_sup0
  end.

Ltac omega_sup :=
  change 2 with (IZR 2);
```

```
  change 1 with (IZR 1);
  change 0 with (IZR 0);
  repeat
    rewrite ← plus_IZR ||
      rewrite ← mult_IZR || rewrite ← Ropp_Ropp_IZR || rewrite Z_R_minus;
  apply IZR_lt; omega.

Ltac prove_sup :=
  match goal with
  | ⊢ (?X1 > ?X2) ⇒ change (X2 < X1) in ⊢ ×; prove_sup
  | ⊢ (0 < ?X1) ⇒ prove_sup0
  | ⊢ (- ?X1 < 0) ⇒ rewrite ← Ropp_0; prove_sup
  | ⊢ (- ?X1 < - ?X2) ⇒ apply Ropp_lt_gt_contravar; prove_sup
  | ⊢ (- ?X1 < ?X2) ⇒ apply Rlt_trans with 0; prove_sup
  | ⊢ (?X1 < ?X2) ⇒ omega_sup
  | _ ⇒ idtac
  end.

Ltac Rcompute :=
  change 2 with (IZR 2);
  change 1 with (IZR 1);
  change 0 with (IZR 0);
  repeat
    rewrite ← plus_IZR ||
      rewrite ← mult_IZR || rewrite ← Ropp_Ropp_IZR || rewrite Z_R_minus;
  apply IZR_eq; try reflexivity.
```

Chapter 96

Module Coq.Reals.Exp_prop

```
Require Import Rbase.
Require Import Rfunctions.
Require Import SeqSeries.
Require Import Rtrigo.
Require Import Ranalysis1.
Require Import PSeries_reg.
Require Import Div2.
Require Import Even.
Require Import Max.
Open Local Scope nat_scope.
Open Local Scope R_scope.

Definition E1 (x:R) (N:nat) : R :=
  sum_f_R0 (fun k:nat ⇒ / INR (fact k) × x ^ k) N.

Lemma E1_cvg : ∀ x:R, Un_cv (E1 x) (exp x).

Definition Reste_E (x y:R) (N:nat) : R :=
  sum_f_R0
  (fun k:nat ⇒
    sum_f_R0
    (fun l:nat ⇒
      / INR (fact (S (l + k))) × x ^ S (l + k) ×
      (/ INR (fact (N - l)) × y ^ (N - l))) (
        pred (N - k))) (pred N).

Lemma exp_form :
  ∀ (x y:R) (n:nat),
    (0 < n)%nat → E1 x n × E1 y n - Reste_E x y n = E1 (x + y) n.

Definition maj_Reste_E (x y:R) (N:nat) : R :=
  4 ×
  (Rmax 1 (Rmax (Rabs x) (Rabs y)) ^ (2 × N) /
    Rsqr (INR (fact (div2 (pred N))))).

Lemma Rle_Rinv : ∀ x y:R, 0 < x → 0 < y → x ≤ y → / y ≤ / x.
```

Lemma $div2\_double : \forall\ N{:}nat,\ div2\ (2 \times N) = N.$

Lemma $div2\_S\_double : \forall\ N{:}nat,\ div2\ (S\ (2 \times N)) = N.$

Lemma $div2\_not\_R0 : \forall\ N{:}nat,\ (1 < N)\%nat \to (0 < div2\ N)\%nat.$

Lemma $Reste\_E\_maj :$
$\forall\ (x\ y{:}R)\ (N{:}nat),$
$(0 < N)\%nat \to Rabs\ (Reste\_E\ x\ y\ N) \leq maj\_Reste\_E\ x\ y\ N.$

Lemma $maj\_Reste\_cv\_R0 : \forall\ x\ y{:}R,\ Un\_cv\ (maj\_Reste\_E\ x\ y)\ 0.$

Lemma $Reste\_E\_cv : \forall\ x\ y{:}R,\ Un\_cv\ (Reste\_E\ x\ y)\ 0.$

Lemma $exp\_plus : \forall\ x\ y{:}R,\ exp\ (x + y) = exp\ x \times exp\ y.$

Lemma $exp\_pos\_pos : \forall\ x{:}R,\ 0 < x \to 0 < exp\ x.$

Lemma $exp\_pos : \forall\ x{:}R,\ 0 < exp\ x.$

Lemma $derivable\_pt\_lim\_exp\_0 : derivable\_pt\_lim\ exp\ 0\ 1.$

Lemma $derivable\_pt\_lim\_exp : \forall\ x{:}R,\ derivable\_pt\_lim\ exp\ x\ (exp\ x).$

Chapter 97

Module Coq.Reals.Integration

Require Export *NewtonInt*.
Require Export *RiemannInt_SF*.
Require Export *RiemannInt*.

Chapter 98

Module Coq.Reals.LegacyRfield

Require Export *Raxioms*.
Require Export *LegacyField*.
Import *LegacyRing_theory*.

Section *LegacyRfield*.

Open Scope R_scope.

Lemma *RLegacyTheory* : *Ring_Theory Rplus Rmult* 1 0 *Ropp* (fun *x y*:*R* ⇒ *false*).

End *LegacyRfield*.

Add Legacy Field
R Rplus Rmult 1%*R* 0%*R Ropp* (fun *x y*:*R* ⇒ *false*) *Rinv RLegacyTheory Rinv_l*
with *minus* := *Rminus div* := *Rdiv*.

Chapter 99

Module Coq.Reals.MVT

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Ranalysis1*.
Require Import *Rtopology*. *Open* Local *Scope R_scope*.

Theorem *MVT* :
$\forall$ (f g:$R \to R$) (a b:R) ($pr1$:$\forall$ c:R, $a < c < b \to$ *derivable_pt* f c)
($pr2$:$\forall$ c:R, $a < c < b \to$ *derivable_pt* g c),
$a < b \to$
($\forall$ c:R, $a \leq c \leq b \to$ *continuity_pt* f c) $\to$
($\forall$ c:R, $a \leq c \leq b \to$ *continuity_pt* g c) $\to$
$\exists$ c : R,
($\exists$ P : $a < c < b$,
(g b - g a) $\times$ *derive_pt* f c ($pr1$ c P) =
(f b - f a) $\times$ *derive_pt* g c ($pr2$ c P)).

Lemma *MVT_cor1* :
$\forall$ (f:$R \to R$) (a b:R) (pr:*derivable* f),
$a < b \to$
$\exists$ c : R, f b - f a = *derive_pt* f c (pr c) $\times$ (b - a) $\wedge$ $a < c < b$.

Theorem *MVT_cor2* :
$\forall$ (f f':$R \to R$) (a b:R),
$a < b \to$
($\forall$ c:R, $a \leq c \leq b \to$ *derivable_pt_lim* f c (f' c)) $\to$
$\exists$ c : R, f b - f a = f' c $\times$ (b - a) $\wedge$ $a < c < b$.

Lemma *MVT_cor3* :
$\forall$ (f f':$R \to R$) (a b:R),
$a < b \to$
($\forall$ x:R, $a \leq x \to x \leq b \to$ *derivable_pt_lim* f x (f' x)) $\to$
$\exists$ c : R, $a \leq c \wedge c \leq b \wedge f$ b = f a + f' c $\times$ (b - a).

Lemma *Rolle* :
$\forall$ (f:$R \to R$) (a b:R) (pr:$\forall$ x:R, $a < x < b \to$ *derivable_pt* f x),

$(\forall\ x{:}R,\ a \leq x \leq b \rightarrow \mathit{continuity\_pt}\ f\ x) \rightarrow$
$a < b \rightarrow$
$f\ a = f\ b \rightarrow$
$\exists\ c : R, (\exists\ P : a < c < b, \mathit{derive\_pt}\ f\ c\ (\mathit{pr}\ c\ P) = 0).$

Lemma *nonneg_derivative_1* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$(\forall\ x{:}R, 0 \leq \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x)) \rightarrow \mathit{increasing}\ f.$

Lemma *nonpos_derivative_0* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$\mathit{decreasing}\ f \rightarrow \forall\ x{:}R, \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x) \leq 0.$

Lemma *increasing_decreasing_opp* :
$\forall\ f{:}R \rightarrow R, \mathit{increasing}\ f \rightarrow \mathit{decreasing}\ (\text{-}\ f)\%F.$

Lemma *nonpos_derivative_1* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$(\forall\ x{:}R, \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x) \leq 0) \rightarrow \mathit{decreasing}\ f.$

Lemma *positive_derivative* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$(\forall\ x{:}R, 0 < \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x)) \rightarrow \mathit{strict\_increasing}\ f.$

Lemma *strictincreasing_strictdecreasing_opp* :
$\forall\ f{:}R \rightarrow R, \mathit{strict\_increasing}\ f \rightarrow \mathit{strict\_decreasing}\ (\text{-}\ f)\%F.$

Lemma *negative_derivative* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$(\forall\ x{:}R, \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x) < 0) \rightarrow \mathit{strict\_decreasing}\ f.$

Lemma *null_derivative_0* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$\mathit{constant}\ f \rightarrow \forall\ x{:}R, \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x) = 0.$

Lemma *increasing_decreasing* :
$\forall\ f{:}R \rightarrow R, \mathit{increasing}\ f \rightarrow \mathit{decreasing}\ f \rightarrow \mathit{constant}\ f.$

Lemma *null_derivative_1* :
$\forall\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$(\forall\ x{:}R, \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x) = 0) \rightarrow \mathit{constant}\ f.$

Lemma *derive_increasing_interv_ax* :
$\forall\ (a\ b{:}R)\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$a < b \rightarrow$
$((\forall\ t{:}R, a < t < b \rightarrow 0 < \mathit{derive\_pt}\ f\ t\ (\mathit{pr}\ t)) \rightarrow$
$\forall\ x\ y{:}R, a \leq x \leq b \rightarrow a \leq y \leq b \rightarrow x < y \rightarrow f\ x < f\ y) \wedge$
$((\forall\ t{:}R, a < t < b \rightarrow 0 \leq \mathit{derive\_pt}\ f\ t\ (\mathit{pr}\ t)) \rightarrow$
$\forall\ x\ y{:}R, a \leq x \leq b \rightarrow a \leq y \leq b \rightarrow x < y \rightarrow f\ x \leq f\ y).$

Lemma *derive_increasing_interv* :
$\forall\ (a\ b{:}R)\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$a < b \rightarrow$

$(\forall\ t{:}R,\ a < t < b \rightarrow 0 < \mathit{derive\_pt}\ f\ t\ (\mathit{pr}\ t)) \rightarrow$
$\forall\ x\ y{:}R,\ a \leq x \leq b \rightarrow a \leq y \leq b \rightarrow x < y \rightarrow f\ x < f\ y.$

Lemma *derive_increasing_interv_var* :
$\forall\ (a\ b{:}R)\ (f{:}R \rightarrow R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$a < b \rightarrow$
$(\forall\ t{:}R,\ a < t < b \rightarrow 0 \leq \mathit{derive\_pt}\ f\ t\ (\mathit{pr}\ t)) \rightarrow$
$\forall\ x\ y{:}R,\ a \leq x \leq b \rightarrow a \leq y \leq b \rightarrow x < y \rightarrow f\ x \leq f\ y.$

Theorem *IAF* :
$\forall\ (f{:}R \rightarrow R)\ (a\ b\ k{:}R)\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$a \leq b \rightarrow$
$(\forall\ c{:}R,\ a \leq c \leq b \rightarrow \mathit{derive\_pt}\ f\ c\ (\mathit{pr}\ c) \leq k) \rightarrow$
$f\ b - f\ a \leq k \times (b - a).$

Lemma *IAF_var* :
$\forall\ (f\ g{:}R \rightarrow R)\ (a\ b{:}R)\ (\mathit{pr1}{:}\mathit{derivable}\ f)\ (\mathit{pr2}{:}\mathit{derivable}\ g),$
$a \leq b \rightarrow$
$(\forall\ c{:}R,\ a \leq c \leq b \rightarrow \mathit{derive\_pt}\ g\ c\ (\mathit{pr2}\ c) \leq \mathit{derive\_pt}\ f\ c\ (\mathit{pr1}\ c)) \rightarrow$
$g\ b - g\ a \leq f\ b - f\ a.$

Lemma *null_derivative_loc* :
$\forall\ (f{:}R \rightarrow R)\ (a\ b{:}R)\ (\mathit{pr}{:}\forall\ x{:}R,\ a < x < b \rightarrow \mathit{derivable\_pt}\ f\ x),$
$(\forall\ x{:}R,\ a \leq x \leq b \rightarrow \mathit{continuity\_pt}\ f\ x) \rightarrow$
$(\forall\ (x{:}R)\ (P{:}a < x < b),\ \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x\ P) = 0) \rightarrow$
$\mathit{constant\_D\_eq}\ f\ (\mathsf{fun}\ x{:}R \Rightarrow a \leq x \leq b)\ (f\ a).$

Lemma *antiderivative_Ucte* :
$\forall\ (f\ g1\ g2{:}R \rightarrow R)\ (a\ b{:}R),$
$\mathit{antiderivative}\ f\ g1\ a\ b \rightarrow$
$\mathit{antiderivative}\ f\ g2\ a\ b \rightarrow$
$\exists\ c : R,\ (\forall\ x{:}R,\ a \leq x \leq b \rightarrow g1\ x = g2\ x + c).$

Chapter 100

Module Coq.Reals.NewtonInt

```
Require Import Rbase.
Require Import Rfunctions.
Require Import SeqSeries.
Require Import Rtrigo.
Require Import Ranalysis. Open Local Scope R_scope.

Definition Newton_integrable (f:R → R) (a b:R) : Type :=
  sigT (fun g:R → R ⇒ antiderivative f g a b ∨ antiderivative f g b a).

Definition NewtonInt (f:R → R) (a b:R) (pr:Newton_integrable f a b) : R :=
  let g := match pr with
             | existT a b ⇒ a
           end in g b - g a.

Lemma FTCN_step1 :
  ∀ (f:Differential) (a b:R),
    Newton_integrable (fun x:R ⇒ derive_pt f x (cond_diff f x)) a b.

Lemma FTC_Newton :
  ∀ (f:Differential) (a b:R),
    NewtonInt (fun x:R ⇒ derive_pt f x (cond_diff f x)) a b
    (FTCN_step1 f a b) = f b - f a.

Lemma NewtonInt_P1 : ∀ (f:R → R) (a:R), Newton_integrable f a a.

Lemma NewtonInt_P2 :
  ∀ (f:R → R) (a:R), NewtonInt f a a (NewtonInt_P1 f a) = 0.

Lemma NewtonInt_P3 :
  ∀ (f:R → R) (a b:R) (X:Newton_integrable f a b),
    Newton_integrable f b a.

Lemma NewtonInt_P4 :
  ∀ (f:R → R) (a b:R) (pr:Newton_integrable f a b),
    NewtonInt f a b pr = - NewtonInt f b a (NewtonInt_P3 f a b pr).

Lemma NewtonInt_P5 :
  ∀ (f g:R → R) (l a b:R),
```

Newton_integrable f a b →
Newton_integrable g a b →
Newton_integrable (fun *x*:*R* ⇒ *l* × *f x* + *g x*) *a b*.

Lemma *antiderivative_P1* :
∀ (*f g F G*:*R* → *R*) (*l a b*:*R*),
antiderivative f F a b →
antiderivative g G a b →
antiderivative (fun *x*:*R* ⇒ *l* × *f x* + *g x*) (fun *x*:*R* ⇒ *l* × *F x* + *G x*) *a b*.

Lemma *NewtonInt_P6* :
∀ (*f g*:*R* → *R*) (*l a b*:*R*) (*pr1*:*Newton_integrable f a b*)
(*pr2*:*Newton_integrable g a b*),
NewtonInt (fun *x*:*R* ⇒ *l* × *f x* + *g x*) *a b* (*NewtonInt_P5 f g l a b pr1 pr2*) =
l × *NewtonInt f a b pr1* + *NewtonInt g a b pr2*.

Lemma *antiderivative_P2* :
∀ (*f F0 F1*:*R* → *R*) (*a b c*:*R*),
antiderivative f F0 a b →
antiderivative f F1 b c →
antiderivative f
(fun *x*:*R* ⇒
match *Rle_dec x b* with
| *left* _ ⇒ *F0 x*
| *right* _ ⇒ *F1 x* + (*F0 b* - *F1 b*)
end) *a c*.

Lemma *antiderivative_P3* :
∀ (*f F0 F1*:*R* → *R*) (*a b c*:*R*),
antiderivative f F0 a b →
antiderivative f F1 c b →
antiderivative f F1 c a ∨ *antiderivative f F0 a c*.

Lemma *antiderivative_P4* :
∀ (*f F0 F1*:*R* → *R*) (*a b c*:*R*),
antiderivative f F0 a b →
antiderivative f F1 a c →
antiderivative f F1 b c ∨ *antiderivative f F0 c b*.

Lemma *NewtonInt_P7* :
∀ (*f*:*R* → *R*) (*a b c*:*R*),
a < *b* →
b < *c* →
Newton_integrable f a b →
Newton_integrable f b c → *Newton_integrable f a c*.

Lemma *NewtonInt_P8* :
∀ (*f*:*R* → *R*) (*a b c*:*R*),
Newton_integrable f a b →
Newton_integrable f b c → *Newton_integrable f a c*.

Lemma *NewtonInt_P9* :
∀ (*f*:*R* → *R*) (*a b c*:*R*) (*pr1*:*Newton_integrable f a b*)
(*pr2*:*Newton_integrable f b c*),
NewtonInt f a c (*NewtonInt_P8 f a b c pr1 pr2*) =
NewtonInt f a b pr1 + *NewtonInt f b c pr2*.

Chapter 101

Module Coq.Reals.PartSum

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *Rcomplete*.
Require Import *Max*.
Open Local *Scope R_scope*.

Lemma *tech1* :
$\forall\ (An{:}nat \to R)\ (N{:}nat),$
$(\forall\ n{:}nat, (n \leq N)\%nat \to 0 < An\ n) \to 0 < sum\_f\_R0\ An\ N.$

Lemma *tech2* :
$\forall\ (An{:}nat \to R)\ (m\ n{:}nat),$
$(m < n)\%nat \to$
$sum\_f\_R0\ An\ n =$
$sum\_f\_R0\ An\ m + sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow An\ (S\ m + i)\%nat)\ (n - S\ m).$

Lemma *tech3* :
$\forall\ (k{:}R)\ (N{:}nat),$
$k \neq 1 \to sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow k\ \hat{}\ i)\ N = (1 - k\ \hat{}\ S\ N) / (1 - k).$

Lemma *tech4* :
$\forall\ (An{:}nat \to R)\ (k{:}R)\ (N{:}nat),$
$0 \leq k \to (\forall\ i{:}nat, An\ (S\ i) < k \times An\ i) \to An\ N \leq An\ 0\%nat \times k\ \hat{}\ N.$

Lemma *tech5* :
$\forall\ (An{:}nat \to R)\ (N{:}nat), sum\_f\_R0\ An\ (S\ N) = sum\_f\_R0\ An\ N + An\ (S\ N).$

Lemma *tech6* :
$\forall\ (An{:}nat \to R)\ (k{:}R)\ (N{:}nat),$
$0 \leq k \to$
$(\forall\ i{:}nat, An\ (S\ i) < k \times An\ i) \to$
$sum\_f\_R0\ An\ N \leq An\ 0\%nat \times sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow k\ \hat{}\ i)\ N.$

Lemma *tech7* : $\forall\ r1\ r2{:}R, r1 \neq 0 \to r2 \neq 0 \to r1 \neq r2 \to / r1 \neq / r2.$

Lemma *tech11* :

$\forall$ (An Bn Cn:$nat \to R$) (N:nat),
($\forall$ i:nat, $An\ i = Bn\ i$ - $Cn\ i$) $\to$
$sum\_f\_R0\ An\ N = sum\_f\_R0\ Bn\ N$ - $sum\_f\_R0\ Cn\ N$.

Lemma *tech12* :
$\forall$ (An:$nat \to R$) (x l:R),
Un_cv (fun N:$nat \Rightarrow sum\_f\_R0$ (fun i:$nat \Rightarrow An\ i \times x\ \hat{}\ i$) N) $l \to$
$Pser\ An\ x\ l$.

Lemma *scal_sum* :
$\forall$ (An:$nat \to R$) (N:nat) (x:R),
$x \times sum\_f\_R0\ An\ N = sum\_f\_R0$ (fun i:$nat \Rightarrow An\ i \times x$) N.

Lemma *decomp_sum* :
$\forall$ (An:$nat \to R$) (N:nat),
$(0 < N)\%nat \to$
$sum\_f\_R0\ An\ N = An\ 0\%nat + sum\_f\_R0$ (fun i:$nat \Rightarrow An\ (S\ i)$) ($pred\ N$).

Lemma *plus_sum* :
$\forall$ (An Bn:$nat \to R$) (N:nat),
sum_f_R0 (fun i:$nat \Rightarrow An\ i + Bn\ i$) $N = sum\_f\_R0\ An\ N + sum\_f\_R0\ Bn\ N$.

Lemma *sum_eq* :
$\forall$ (An Bn:$nat \to R$) (N:nat),
($\forall$ i:nat, $(i \leq N)\%nat \to An\ i = Bn\ i$) $\to$
$sum\_f\_R0\ An\ N = sum\_f\_R0\ Bn\ N$.

Lemma *uniqueness_sum* :
$\forall$ (An:$nat \to R$) ($l1$ $l2$:R),
$infinit\_sum\ An\ l1 \to infinit\_sum\ An\ l2 \to l1 = l2$.

Lemma *minus_sum* :
$\forall$ (An Bn:$nat \to R$) (N:nat),
sum_f_R0 (fun i:$nat \Rightarrow An\ i$ - $Bn\ i$) $N = sum\_f\_R0\ An\ N$ - $sum\_f\_R0\ Bn\ N$.

Lemma *sum_decomposition* :
$\forall$ (An:$nat \to R$) (N:nat),
sum_f_R0 (fun l:$nat \Rightarrow An\ (2 \times l)\%nat$) ($S\ N$) +
sum_f_R0 (fun l:$nat \Rightarrow An\ (S\ (2 \times l))$) $N = sum\_f\_R0\ An\ (2 \times S\ N)$.

Lemma *sum_Rle* :
$\forall$ (An Bn:$nat \to R$) (N:nat),
($\forall$ n:nat, $(n \leq N)\%nat \to An\ n \leq Bn\ n$) $\to$
$sum\_f\_R0\ An\ N \leq sum\_f\_R0\ Bn\ N$.

Lemma *Rsum_abs* :
$\forall$ (An:$nat \to R$) (N:nat),
$Rabs\ (sum\_f\_R0\ An\ N) \leq sum\_f\_R0$ (fun l:$nat \Rightarrow Rabs\ (An\ l)$) N.

Lemma *sum_cte* :
$\forall$ (x:R) (N:nat), sum_f_R0 (fun _:$nat \Rightarrow x$) $N = x \times INR\ (S\ N)$.

Lemma *sum_growing* :

$\forall\ (An\ Bn{:}nat \to R)\ (N{:}nat),$
$(\forall\ n{:}nat,\ An\ n \leq Bn\ n) \to sum\_f\_R0\ An\ N \leq sum\_f\_R0\ Bn\ N.$

Lemma *Rabs_triang_gen* :
$\forall\ (An{:}nat \to R)\ (N{:}nat),$
$Rabs\ (sum\_f\_R0\ An\ N) \leq sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow Rabs\ (An\ i))\ N.$

Lemma *cond_pos_sum* :
$\forall\ (An{:}nat \to R)\ (N{:}nat),$
$(\forall\ n{:}nat,\ 0 \leq An\ n) \to 0 \leq sum\_f\_R0\ An\ N.$

Definition *Cauchy_crit_series* $(An{:}nat \to R)$: Prop :=
$Cauchy\_crit\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ An\ N).$

Lemma *cauchy_abs* :
$\forall\ An{:}nat \to R,$
$Cauchy\_crit\_series\ (\mathsf{fun}\ i{:}nat \Rightarrow Rabs\ (An\ i)) \to Cauchy\_crit\_series\ An.$

Lemma *cv_cauchy_1* :
$\forall\ An{:}nat \to R,$
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ An\ N)\ l) \to$
$Cauchy\_crit\_series\ An.$

Lemma *cv_cauchy_2* :
$\forall\ An{:}nat \to R,$
$Cauchy\_crit\_series\ An \to$
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow sum\_f\_R0\ An\ N)\ l).$

Lemma *sum_eq_R0* :
$\forall\ (An{:}nat \to R)\ (N{:}nat),$
$(\forall\ n{:}nat,\ (n \leq N)\%nat \to An\ n = 0) \to sum\_f\_R0\ An\ N = 0.$

Definition *SP* $(fn{:}nat \to R \to R)\ (N{:}nat)\ (x{:}R) : R :=$
$sum\_f\_R0\ (\mathsf{fun}\ k{:}nat \Rightarrow fn\ k\ x)\ N.$

Lemma *sum_incr* :
$\forall\ (An{:}nat \to R)\ (N{:}nat)\ (l{:}R),$
$Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow sum\_f\_R0\ An\ n)\ l \to$
$(\forall\ n{:}nat,\ 0 \leq An\ n) \to sum\_f\_R0\ An\ N \leq l.$

Lemma *sum_cv_maj* :
$\forall\ (An{:}nat \to R)\ (fn{:}nat \to R \to R)\ (x\ l1\ l2{:}R),$
$Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow SP\ fn\ n\ x)\ l1 \to$
$Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow sum\_f\_R0\ An\ n)\ l2 \to$
$(\forall\ n{:}nat,\ Rabs\ (fn\ n\ x) \leq An\ n) \to Rabs\ l1 \leq l2.$

Chapter 102

Module Coq.Reals.PSeries_reg

```
Require Import Rbase.
Require Import Rfunctions.
Require Import SeqSeries.
Require Import Ranalysis1.
Require Import Max.
Require Import Even. Open Local Scope R_scope.

Definition Boule (x:R) (r:posreal) (y:R) : Prop := Rabs (y - x) < r.

Uniform convergence
Definition CVU (fn:nat → R → R) (f:R → R) (x:R)
  (r:posreal) : Prop :=
  ∀ eps:R,
    0 < eps →
    ∃ N : nat,
      (∀ (n:nat) (y:R),
        (N ≤ n)%nat → Boule x r y → Rabs (f y - fn n y) < eps).

Normal convergence
Definition CVN_r (fn:nat → R → R) (r:posreal) : Type :=
  sigT
  (fun An:nat → R ⇒
    sigT
    (fun l:R ⇒
      Un_cv (fun n:nat ⇒ sum_f_R0 (fun k:nat ⇒ Rabs (An k)) n) l ∧
      (∀ (n:nat) (y:R), Boule 0 r y → Rabs (fn n y) ≤ An n))).

Definition CVN_R (fn:nat → R → R) : Type := ∀ r:posreal, CVN_r fn r.

Definition SFL (fn:nat → R → R)
  (cv:∀ x:R, sigT (fun l:R ⇒ Un_cv (fun N:nat ⇒ SP fn N x) l))
  (y:R) : R := match cv y with
                 | existT a b ⇒ a
               end.
```

In a complete space, normal convergence implies uniform convergence

Lemma CVN_CVU :
$\forall\ (fn{:}nat \to R \to R)$
$(cv{:}\forall\ x{:}R,\ sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow SP\ fn\ N\ x)\ l))$
$(r{:}posreal),\ CVN\_r\ fn\ r \to CVU\ (\mathsf{fun}\ n{:}nat \Rightarrow SP\ fn\ n)\ (SFL\ fn\ cv)\ 0\ r.$

Each limit of a sequence of functions which converges uniformly is continue
Lemma $CVU\_continuity$:
$\forall\ (fn{:}nat \to R \to R)\ (f{:}R \to R)\ (x{:}R)\ (r{:}posreal),$
$CVU\ fn\ f\ x\ r \to$
$(\forall\ (n{:}nat)\ (y{:}R),\ Boule\ x\ r\ y \to continuity\_pt\ (fn\ n)\ y) \to$
$\forall\ y{:}R,\ Boule\ x\ r\ y \to continuity\_pt\ f\ y.$

Lemma $continuity\_pt\_finite\_SF$:
$\forall\ (fn{:}nat \to R \to R)\ (N{:}nat)\ (x{:}R),$
$(\forall\ n{:}nat,\ (n \leq N)\%nat \to continuity\_pt\ (fn\ n)\ x) \to$
$continuity\_pt\ (\mathsf{fun}\ y{:}R \Rightarrow sum\_f\_R0\ (\mathsf{fun}\ k{:}nat \Rightarrow fn\ k\ y)\ N)\ x.$

Continuity and normal convergence
Lemma $SFL\_continuity\_pt$:
$\forall\ (fn{:}nat \to R \to R)$
$(cv{:}\forall\ x{:}R,\ sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow SP\ fn\ N\ x)\ l))$
$(r{:}posreal),$
$CVN\_r\ fn\ r \to$
$(\forall\ (n{:}nat)\ (y{:}R),\ Boule\ 0\ r\ y \to continuity\_pt\ (fn\ n)\ y) \to$
$\forall\ y{:}R,\ Boule\ 0\ r\ y \to continuity\_pt\ (SFL\ fn\ cv)\ y.$

Lemma $SFL\_continuity$:
$\forall\ (fn{:}nat \to R \to R)$
$(cv{:}\forall\ x{:}R,\ sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow SP\ fn\ N\ x)\ l)),$
$CVN\_R\ fn \to (\forall\ n{:}nat,\ continuity\ (fn\ n)) \to continuity\ (SFL\ fn\ cv).$

As R is complete, normal convergence implies that (fn) is simply-uniformly convergent
Lemma CVN_R_CVS :
$\forall\ fn{:}nat \to R \to R,$
$CVN\_R\ fn \to \forall\ x{:}R,\ sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow SP\ fn\ N\ x)\ l).$

Chapter 103

Module Coq.Reals.Ranalysis1

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Export *Rlimit*.
Require Export *Rderiv*. *Open* Local *Scope R_scope*.
Implicit Type $f : R \to R$.

103.1 Basic operations on functions

Definition *plus_fct f1 f2* (*x*:*R*) : *R* := *f1 x* + *f2 x*.
Definition *opp_fct f* (*x*:*R*) : *R* := - *f x*.
Definition *mult_fct f1 f2* (*x*:*R*) : *R* := *f1 x* × *f2 x*.
Definition *mult_real_fct* (*a*:*R*) *f* (*x*:*R*) : *R* := *a* × *f x*.
Definition *minus_fct f1 f2* (*x*:*R*) : *R* := *f1 x* - *f2 x*.
Definition *div_fct f1 f2* (*x*:*R*) : *R* := *f1 x* / *f2 x*.
Definition *div_real_fct* (*a*:*R*) *f* (*x*:*R*) : *R* := *a* / *f x*.
Definition *comp f1 f2* (*x*:*R*) : *R* := *f1* (*f2 x*).
Definition *inv_fct f* (*x*:*R*) : *R* := / *f x*.

Delimit Scope Rfun_scope with *F*.

Infix "+" := *plus_fct* : *Rfun_scope*.
Notation "- *x*" := (*opp_fct x*) : *Rfun_scope*.
Infix "×" := *mult_fct* : *Rfun_scope*.
Infix "-" := *minus_fct* : *Rfun_scope*.
Infix "/" := *div_fct* : *Rfun_scope*.
Notation Local "*f1 'o' f2*" := (*comp f1 f2*)
(*at level* 20, *right associativity*) : *Rfun_scope*.
Notation "/ *x*" := (*inv_fct x*) : *Rfun_scope*.

Definition *fct_cte* (*a x*:*R*) : *R* := *a*.
Definition *id* (*x*:*R*) := *x*.

103.2 Variations of functions

Definition *increasing* f : Prop := $\forall\ x\ y{:}R,\ x \leq y \to f\ x \leq f\ y$.
Definition *decreasing* f : Prop := $\forall\ x\ y{:}R,\ x \leq y \to f\ y \leq f\ x$.
Definition *strict_increasing* f : Prop := $\forall\ x\ y{:}R,\ x < y \to f\ x < f\ y$.
Definition *strict_decreasing* f : Prop := $\forall\ x\ y{:}R,\ x < y \to f\ y < f\ x$.
Definition *constant* f : Prop := $\forall\ x\ y{:}R,\ f\ x = f\ y$.

Definition *no_cond* $(x{:}R)$: Prop := *True*.

Definition *constant_D_eq* $f\ (D{:}R \to \text{Prop})\ (c{:}R)$: Prop :=
$\forall\ x{:}R,\ D\ x \to f\ x = c$.

103.3 Definition of continuity as a limit

Definition *continuity_pt* $f\ (x0{:}R)$: Prop := *continue_in* f *no_cond* $x0$.
Definition *continuity* f : Prop := $\forall\ x{:}R$, *continuity_pt* $f\ x$.

Lemma *continuity_pt_plus* :
$\forall\ f1\ f2\ (x0{:}R)$,
continuity_pt $f1\ x0 \to$ *continuity_pt* $f2\ x0 \to$ *continuity_pt* $(f1 + f2)\ x0$.

Lemma *continuity_pt_opp* :
$\forall\ f\ (x0{:}R)$, *continuity_pt* $f\ x0 \to$ *continuity_pt* $(-f)\ x0$.

Lemma *continuity_pt_minus* :
$\forall\ f1\ f2\ (x0{:}R)$,
continuity_pt $f1\ x0 \to$ *continuity_pt* $f2\ x0 \to$ *continuity_pt* $(f1 - f2)\ x0$.

Lemma *continuity_pt_mult* :
$\forall\ f1\ f2\ (x0{:}R)$,
continuity_pt $f1\ x0 \to$ *continuity_pt* $f2\ x0 \to$ *continuity_pt* $(f1 \times f2)\ x0$.

Lemma *continuity_pt_const* : $\forall\ f\ (x0{:}R)$, *constant* $f \to$ *continuity_pt* $f\ x0$.

Lemma *continuity_pt_scal* :
$\forall\ f\ (a\ x0{:}R)$,
continuity_pt $f\ x0 \to$ *continuity_pt* $(mult\_real\_fct\ a\ f)\ x0$.

Lemma *continuity_pt_inv* :
$\forall\ f\ (x0{:}R)$, *continuity_pt* $f\ x0 \to f\ x0 \neq 0 \to$ *continuity_pt* $(/\ f)\ x0$.

Lemma *div_eq_inv* : $\forall\ f1\ f2, (f1\ /\ f2)\%F = (f1 \times /\ f2)\%F$.

Lemma *continuity_pt_div* :
$\forall\ f1\ f2\ (x0{:}R)$,
continuity_pt $f1\ x0 \to$
continuity_pt $f2\ x0 \to f2\ x0 \neq 0 \to$ *continuity_pt* $(f1\ /\ f2)\ x0$.

Lemma *continuity_pt_comp* :
$\forall\ f1\ f2\ (x{:}R)$,
continuity_pt $f1\ x \to$ *continuity_pt* $f2\ (f1\ x) \to$ *continuity_pt* $(f2\ o\ f1)\ x$.

Lemma *continuity_plus* :
  ∀ *f1 f2*, *continuity f1* → *continuity f2* → *continuity* (*f1* + *f2*).

Lemma *continuity_opp* : ∀ *f*, *continuity f* → *continuity* (- *f*).

Lemma *continuity_minus* :
  ∀ *f1 f2*, *continuity f1* → *continuity f2* → *continuity* (*f1* - *f2*).

Lemma *continuity_mult* :
  ∀ *f1 f2*, *continuity f1* → *continuity f2* → *continuity* (*f1* × *f2*).

Lemma *continuity_const* : ∀ *f*, *constant f* → *continuity f*.

Lemma *continuity_scal* :
  ∀ *f* (*a*:*R*), *continuity f* → *continuity* (*mult_real_fct a f*).

Lemma *continuity_inv* :
  ∀ *f*, *continuity f* → (∀ *x*:*R*, *f x* ≠ 0) → *continuity* (/ *f*).

Lemma *continuity_div* :
  ∀ *f1 f2*,
    continuity f1 →
    continuity f2 → (∀ *x*:*R*, *f2 x* ≠ 0) → *continuity* (*f1* / *f2*).

Lemma *continuity_comp* :
  ∀ *f1 f2*, *continuity f1* → *continuity f2* → *continuity* (*f2* o *f1*).

103.4 Derivative's definition using Landau's kernel

Definition *derivable_pt_lim f* (*x l*:*R*) : Prop :=
  ∀ *eps*:*R*,
    0 < *eps* →
    ∃ *delta* : *posreal*,
      (∀ *h*:*R*,
        h ≠ 0 → *Rabs h* < *delta* → *Rabs* ((*f* (*x* + *h*) - *f x*) / *h* - *l*) < *eps*).

Definition *derivable_pt_abs f* (*x l*:*R*) : Prop := *derivable_pt_lim f x l*.

Definition *derivable_pt f* (*x*:*R*) := *sigT* (*derivable_pt_abs f x*).
Definition *derivable f* := ∀ *x*:*R*, *derivable_pt f x*.

Definition *derive_pt f* (*x*:*R*) (*pr*:*derivable_pt f x*) := *projT1 pr*.
Definition *derive f* (*pr*:*derivable f*) (*x*:*R*) := *derive_pt f x* (*pr x*).

Definition *antiderivative f* (*g*:*R* → *R*) (*a b*:*R*) : Prop :=
  (∀ *x*:*R*,
    a ≤ *x* ≤ *b* → ∃ *pr* : *derivable_pt g x*, *f x* = *derive_pt g x pr*) ∧
  a ≤ *b*.

103.5 Class of differential functions

Record *Differential* : Type := *mkDifferential*
{*d1* :> $R \to R$; *cond_diff* : *derivable d1*}.

Record *Differential_D2* : Type := *mkDifferential_D2*
{*d2* :> $R \to R$;
cond_D1 : *derivable d2*;
cond_D2 : *derivable* (*derive d2 cond_D1*)}.

Lemma *uniqueness_step1* :
$\forall$ *f* (*x l1 l2*:*R*),
limit1_in (fun *h*:*R* $\Rightarrow$ (*f* (*x* + *h*) - *f x*) / *h*) (fun *h*:*R* $\Rightarrow$ *h* $\neq$ 0) *l1* 0 $\to$
limit1_in (fun *h*:*R* $\Rightarrow$ (*f* (*x* + *h*) - *f x*) / *h*) (fun *h*:*R* $\Rightarrow$ *h* $\neq$ 0) *l2* 0 $\to$
l1 = *l2*.

Lemma *uniqueness_step2* :
$\forall$ *f* (*x l*:*R*),
derivable_pt_lim f x l $\to$
limit1_in (fun *h*:*R* $\Rightarrow$ (*f* (*x* + *h*) - *f x*) / *h*) (fun *h*:*R* $\Rightarrow$ *h* $\neq$ 0) *l* 0.

Lemma *uniqueness_step3* :
$\forall$ *f* (*x l*:*R*),
limit1_in (fun *h*:*R* $\Rightarrow$ (*f* (*x* + *h*) - *f x*) / *h*) (fun *h*:*R* $\Rightarrow$ *h* $\neq$ 0) *l* 0 $\to$
derivable_pt_lim f x l.

Lemma *uniqueness_limite* :
$\forall$ *f* (*x l1 l2*:*R*),
derivable_pt_lim f x l1 $\to$ *derivable_pt_lim f x l2* $\to$ *l1* = *l2*.

Lemma *derive_pt_eq* :
$\forall$ *f* (*x l*:*R*) (*pr*:*derivable_pt f x*),
derive_pt f x pr = *l* $\leftrightarrow$ *derivable_pt_lim f x l*.

Lemma *derive_pt_eq_0* :
$\forall$ *f* (*x l*:*R*) (*pr*:*derivable_pt f x*),
derivable_pt_lim f x l $\to$ *derive_pt f x pr* = *l*.

Lemma *derive_pt_eq_1* :
$\forall$ *f* (*x l*:*R*) (*pr*:*derivable_pt f x*),
derive_pt f x pr = *l* $\to$ *derivable_pt_lim f x l*.

103.6 Equivalence of this definition with the one using limit concept

Lemma *derive_pt_D_in* :
$\forall$ *f* (*df*:$R \to R$) (*x*:*R*) (*pr*:*derivable_pt f x*),
D_in f df no_cond x $\leftrightarrow$ *derive_pt f x pr* = *df x*.

Lemma *derivable_pt_lim_D_in* :
$\forall$ *f* (*df*:$R \to R$) (*x*:*R*),
D_in f df no_cond x $\leftrightarrow$ *derivable_pt_lim f x* (*df x*).

103.7 derivability -> continuity

Lemma *derivable_derive* :
$\forall\ f\ (x{:}R)\ (pr{:}derivable\_pt\ f\ x), \exists\ l : R,\ derive\_pt\ f\ x\ pr = l.$

Theorem *derivable_continuous_pt* :
$\forall\ f\ (x{:}R),\ derivable\_pt\ f\ x \to continuity\_pt\ f\ x.$

Theorem *derivable_continuous* : $\forall\ f,\ derivable\ f \to continuity\ f.$

103.8 Main rules

Lemma *derivable_pt_lim_plus* :
$\forall\ f1\ f2\ (x\ l1\ l2{:}R),$
$derivable\_pt\_lim\ f1\ x\ l1 \to$
$derivable\_pt\_lim\ f2\ x\ l2 \to derivable\_pt\_lim\ (f1 + f2)\ x\ (l1 + l2).$

Lemma *derivable_pt_lim_opp* :
$\forall\ f\ (x\ l{:}R),\ derivable\_pt\_lim\ f\ x\ l \to derivable\_pt\_lim\ (-\ f)\ x\ (-\ l).$

Lemma *derivable_pt_lim_minus* :
$\forall\ f1\ f2\ (x\ l1\ l2{:}R),$
$derivable\_pt\_lim\ f1\ x\ l1 \to$
$derivable\_pt\_lim\ f2\ x\ l2 \to derivable\_pt\_lim\ (f1 - f2)\ x\ (l1 - l2).$

Lemma *derivable_pt_lim_mult* :
$\forall\ f1\ f2\ (x\ l1\ l2{:}R),$
$derivable\_pt\_lim\ f1\ x\ l1 \to$
$derivable\_pt\_lim\ f2\ x\ l2 \to$
$derivable\_pt\_lim\ (f1 \times f2)\ x\ (l1 \times f2\ x + f1\ x \times l2).$

Lemma *derivable_pt_lim_const* : $\forall\ a\ x{:}R,\ derivable\_pt\_lim\ (fct\_cte\ a)\ x\ 0.$

Lemma *derivable_pt_lim_scal* :
$\forall\ f\ (a\ x\ l{:}R),$
$derivable\_pt\_lim\ f\ x\ l \to derivable\_pt\_lim\ (mult\_real\_fct\ a\ f)\ x\ (a \times l).$

Lemma *derivable_pt_lim_id* : $\forall\ x{:}R,\ derivable\_pt\_lim\ id\ x\ 1.$

Lemma *derivable_pt_lim_Rsqr* : $\forall\ x{:}R,\ derivable\_pt\_lim\ Rsqr\ x\ (2 \times x).$

Lemma *derivable_pt_lim_comp* :
$\forall\ f1\ f2\ (x\ l1\ l2{:}R),$
$derivable\_pt\_lim\ f1\ x\ l1 \to$
$derivable\_pt\_lim\ f2\ (f1\ x)\ l2 \to derivable\_pt\_lim\ (f2\ o\ f1)\ x\ (l2 \times l1).$

Lemma *derivable_pt_plus* :
$\forall\ f1\ f2\ (x{:}R),$
$derivable\_pt\ f1\ x \to derivable\_pt\ f2\ x \to derivable\_pt\ (f1 + f2)\ x.$

Lemma *derivable_pt_opp* :
$\forall\ f\ (x{:}R),\ derivable\_pt\ f\ x \to derivable\_pt\ (-\ f)\ x.$

Lemma *derivable_pt_minus* :
∀ *f1 f2* (*x*:*R*),
derivable_pt f1 x → *derivable_pt f2 x* → *derivable_pt* (*f1 - f2*) *x*.

Lemma *derivable_pt_mult* :
∀ *f1 f2* (*x*:*R*),
derivable_pt f1 x → *derivable_pt f2 x* → *derivable_pt* (*f1* × *f2*) *x*.

Lemma *derivable_pt_const* : ∀ *a x*:*R*, *derivable_pt* (*fct_cte a*) *x*.

Lemma *derivable_pt_scal* :
∀ *f* (*a x*:*R*), *derivable_pt f x* → *derivable_pt* (*mult_real_fct a f*) *x*.

Lemma *derivable_pt_id* : ∀ *x*:*R*, *derivable_pt id x*.

Lemma *derivable_pt_Rsqr* : ∀ *x*:*R*, *derivable_pt Rsqr x*.

Lemma *derivable_pt_comp* :
∀ *f1 f2* (*x*:*R*),
derivable_pt f1 x → *derivable_pt f2* (*f1 x*) → *derivable_pt* (*f2 o f1*) *x*.

Lemma *derivable_plus* :
∀ *f1 f2*, *derivable f1* → *derivable f2* → *derivable* (*f1* + *f2*).

Lemma *derivable_opp* : ∀ *f*, *derivable f* → *derivable* (- *f*).

Lemma *derivable_minus* :
∀ *f1 f2*, *derivable f1* → *derivable f2* → *derivable* (*f1 - f2*).

Lemma *derivable_mult* :
∀ *f1 f2*, *derivable f1* → *derivable f2* → *derivable* (*f1* × *f2*).

Lemma *derivable_const* : ∀ *a*:*R*, *derivable* (*fct_cte a*).

Lemma *derivable_scal* :
∀ *f* (*a*:*R*), *derivable f* → *derivable* (*mult_real_fct a f*).

Lemma *derivable_id* : *derivable id*.

Lemma *derivable_Rsqr* : *derivable Rsqr*.

Lemma *derivable_comp* :
∀ *f1 f2*, *derivable f1* → *derivable f2* → *derivable* (*f2 o f1*).

Lemma *derive_pt_plus* :
∀ *f1 f2* (*x*:*R*) (*pr1*:*derivable_pt f1 x*) (*pr2*:*derivable_pt f2 x*),
derive_pt (*f1* + *f2*) *x* (*derivable_pt_plus _ _ _ pr1 pr2*) =
derive_pt f1 x pr1 + *derive_pt f2 x pr2*.

Lemma *derive_pt_opp* :
∀ *f* (*x*:*R*) (*pr1*:*derivable_pt f x*),
derive_pt (- *f*) *x* (*derivable_pt_opp _ _ pr1*) = - *derive_pt f x pr1*.

Lemma *derive_pt_minus* :
∀ *f1 f2* (*x*:*R*) (*pr1*:*derivable_pt f1 x*) (*pr2*:*derivable_pt f2 x*),
derive_pt (*f1 - f2*) *x* (*derivable_pt_minus _ _ _ pr1 pr2*) =

derive_pt f1 x pr1 - derive_pt f2 x pr2.

Lemma *derive_pt_mult* :
∀ *f1 f2* (*x*:*R*) (*pr1*:*derivable_pt f1 x*) (*pr2*:*derivable_pt f2 x*),
derive_pt (*f1* × *f2*) *x* (*derivable_pt_mult _ _ _ pr1 pr2*) =
derive_pt f1 x pr1 × *f2 x* + *f1 x* × *derive_pt f2 x pr2.*

Lemma *derive_pt_const* :
∀ *a x*:*R*, *derive_pt* (*fct_cte a*) *x* (*derivable_pt_const a x*) = 0.

Lemma *derive_pt_scal* :
∀ *f* (*a x*:*R*) (*pr*:*derivable_pt f x*),
derive_pt (*mult_real_fct a f*) *x* (*derivable_pt_scal _ _ _ pr*) =
a × *derive_pt f x pr.*

Lemma *derive_pt_id* : ∀ *x*:*R*, *derive_pt id x* (*derivable_pt_id _*) = 1.

Lemma *derive_pt_Rsqr* :
∀ *x*:*R*, *derive_pt Rsqr x* (*derivable_pt_Rsqr _*) = 2 × *x.*

Lemma *derive_pt_comp* :
∀ *f1 f2* (*x*:*R*) (*pr1*:*derivable_pt f1 x*) (*pr2*:*derivable_pt f2* (*f1 x*)),
derive_pt (*f2 o f1*) *x* (*derivable_pt_comp _ _ _ pr1 pr2*) =
derive_pt f2 (*f1 x*) *pr2* × *derive_pt f1 x pr1.*

Definition *pow_fct* (*n*:*nat*) (*y*:*R*) : *R* := *y* ^ *n.*

Lemma *derivable_pt_lim_pow_pos* :
∀ (*x*:*R*) (*n*:*nat*),
(0 < *n*)%*nat* → *derivable_pt_lim* (fun *y*:*R* ⇒ *y* ^ *n*) *x* (*INR n* × *x* ^ *pred n*).

Lemma *derivable_pt_lim_pow* :
∀ (*x*:*R*) (*n*:*nat*),
derivable_pt_lim (fun *y*:*R* ⇒ *y* ^ *n*) *x* (*INR n* × *x* ^ *pred n*).

Lemma *derivable_pt_pow* :
∀ (*n*:*nat*) (*x*:*R*), *derivable_pt* (fun *y*:*R* ⇒ *y* ^ *n*) *x.*

Lemma *derivable_pow* : ∀ *n*:*nat*, *derivable* (fun *y*:*R* ⇒ *y* ^ *n*).

Lemma *derive_pt_pow* :
∀ (*n*:*nat*) (*x*:*R*),
derive_pt (fun *y*:*R* ⇒ *y* ^ *n*) *x* (*derivable_pt_pow n x*) = *INR n* × *x* ^ *pred n.*

Lemma *pr_nu* :
∀ *f* (*x*:*R*) (*pr1 pr2*:*derivable_pt f x*),
derive_pt f x pr1 = *derive_pt f x pr2.*

103.9 Local extremum's condition

Theorem *deriv_maximum* :
∀ *f* (*a b c*:*R*) (*pr*:*derivable_pt f c*),

$a < c \rightarrow$
$c < b \rightarrow$
$(\forall\ x{:}R,\ a < x \rightarrow x < b \rightarrow f\ x \leq f\ c) \rightarrow \mathit{derive\_pt}\ f\ c\ \mathit{pr} = 0.$

Theorem *deriv_minimum* :
$\forall\ f\ (a\ b\ c{:}R)\ (\mathit{pr}{:}\mathit{derivable\_pt}\ f\ c),$
$a < c \rightarrow$
$c < b \rightarrow$
$(\forall\ x{:}R,\ a < x \rightarrow x < b \rightarrow f\ c \leq f\ x) \rightarrow \mathit{derive\_pt}\ f\ c\ \mathit{pr} = 0.$

Theorem *deriv_constant2* :
$\forall\ f\ (a\ b\ c{:}R)\ (\mathit{pr}{:}\mathit{derivable\_pt}\ f\ c),$
$a < c \rightarrow$
$c < b \rightarrow (\forall\ x{:}R,\ a < x \rightarrow x < b \rightarrow f\ x = f\ c) \rightarrow \mathit{derive\_pt}\ f\ c\ \mathit{pr} = 0.$

Lemma *nonneg_derivative_0* :
$\forall\ f\ (\mathit{pr}{:}\mathit{derivable}\ f),$
$\mathit{increasing}\ f \rightarrow \forall\ x{:}R,\ 0 \leq \mathit{derive\_pt}\ f\ x\ (\mathit{pr}\ x).$

Chapter 104

Module Coq.Reals.Ranalysis2

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Ranalysis1*. *Open* Local *Scope R_scope*.

Lemma *formule* :
$\forall\ (x\ h\ l1\ l2{:}R)\ (f1\ f2{:}R \to R)$,
$h \neq 0 \to$
$f2\ x \neq 0 \to$
$f2\ (x + h) \neq 0 \to$
$(f1\ (x + h) / f2\ (x + h) - f1\ x / f2\ x) / h -$
$(l1 \times f2\ x - l2 \times f1\ x) / Rsqr\ (f2\ x) =$
$/ f2\ (x + h) \times ((f1\ (x + h) - f1\ x) / h - l1) +$
$l1 / (f2\ x \times f2\ (x + h)) \times (f2\ x - f2\ (x + h)) -$
$f1\ x / (f2\ x \times f2\ (x + h)) \times ((f2\ (x + h) - f2\ x) / h - l2) +$
$l2 \times f1\ x / (Rsqr\ (f2\ x) \times f2\ (x + h)) \times (f2\ (x + h) - f2\ x)$.

Lemma *Rmin_pos* : $\forall\ x\ y{:}R, 0 < x \to 0 < y \to 0 < Rmin\ x\ y$.

Lemma *maj_term1* :
$\forall\ (x\ h\ eps\ l1\ alp\_f2{:}R)\ (eps\_f2\ alp\_f1d{:}posreal)$
$(f1\ f2{:}R \to R)$,
$0 < eps \to$
$f2\ x \neq 0 \to$
$f2\ (x + h) \neq 0 \to$
$(\forall\ h{:}R,$
$h \neq 0 \to$
$Rabs\ h < alp\_f1d \to$
$Rabs\ ((f1\ (x + h) - f1\ x) / h - l1) < Rabs\ (eps \times f2\ x / 8)) \to$
$(\forall\ a{:}R,$
$Rabs\ a < Rmin\ eps\_f2\ alp\_f2 \to / Rabs\ (f2\ (x + a)) < 2 / Rabs\ (f2\ x)) \to$
$h \neq 0 \to$
$Rabs\ h < alp\_f1d \to$
$Rabs\ h < Rmin\ eps\_f2\ alp\_f2 \to$
$Rabs\ (/ f2\ (x + h) \times ((f1\ (x + h) - f1\ x) / h - l1)) < eps / 4$.

Lemma *maj_term2* :
∀ (*x h eps l1 alp_f2 alp_f2t2*:*R*) (*eps_f2*:*posreal*)
(*f2*:*R* → *R*),
0 < *eps* →
f2 x ≠ 0 →
f2 (*x* + *h*) ≠ 0 →
(∀ *a*:*R*,
Rabs a < *alp_f2t2* →
Rabs (*f2* (*x* + *a*) - *f2 x*) < *Rabs* (*eps* × *Rsqr* (*f2 x*) / (8 × *l1*))) →
(∀ *a*:*R*,
Rabs a < *Rmin eps_f2 alp_f2* → / *Rabs* (*f2* (*x* + *a*)) < 2 / *Rabs* (*f2 x*)) →
h ≠ 0 →
Rabs h < *alp_f2t2* →
Rabs h < *Rmin eps_f2 alp_f2* →
l1 ≠ 0 → *Rabs* (*l1* / (*f2 x* × *f2* (*x* + *h*)) × (*f2 x* - *f2* (*x* + *h*))) < *eps* / 4.

Lemma *maj_term3* :
∀ (*x h eps l2 alp_f2*:*R*) (*eps_f2 alp_f2d*:*posreal*)
(*f1 f2*:*R* → *R*),
0 < *eps* →
f2 x ≠ 0 →
f2 (*x* + *h*) ≠ 0 →
(∀ *h*:*R*,
h ≠ 0 →
Rabs h < *alp_f2d* →
Rabs ((*f2* (*x* + *h*) - *f2 x*) / *h* - *l2*) <
Rabs (*Rsqr* (*f2 x*) × *eps* / (8 × *f1 x*))) →
(∀ *a*:*R*,
Rabs a < *Rmin eps_f2 alp_f2* → / *Rabs* (*f2* (*x* + *a*)) < 2 / *Rabs* (*f2 x*)) →
h ≠ 0 →
Rabs h < *alp_f2d* →
Rabs h < *Rmin eps_f2 alp_f2* →
f1 x ≠ 0 →
Rabs (*f1 x* / (*f2 x* × *f2* (*x* + *h*)) × ((*f2* (*x* + *h*) - *f2 x*) / *h* - *l2*)) <
eps / 4.

Lemma *maj_term4* :
∀ (*x h eps l2 alp_f2 alp_f2c*:*R*) (*eps_f2*:*posreal*)
(*f1 f2*:*R* → *R*),
0 < *eps* →
f2 x ≠ 0 →
f2 (*x* + *h*) ≠ 0 →
(∀ *a*:*R*,
Rabs a < *alp_f2c* →
Rabs (*f2* (*x* + *a*) - *f2 x*) <
Rabs (*Rsqr* (*f2 x*) × *f2 x* × *eps* / (8 × *f1 x* × *l2*))) →
(∀ *a*:*R*,

$Rabs\ a < Rmin\ eps\_f2\ alp\_f2 \rightarrow / Rabs\ (f2\ (x + a)) < 2 / Rabs\ (f2\ x)) \rightarrow$
$h \neq 0 \rightarrow$
$Rabs\ h < alp\_f2c \rightarrow$
$Rabs\ h < Rmin\ eps\_f2\ alp\_f2 \rightarrow$
$f1\ x \neq 0 \rightarrow$
$l2 \neq 0 \rightarrow$
$Rabs\ (l2 \times f1\ x / (Rsqr\ (f2\ x) \times f2\ (x + h)) \times (f2\ (x + h)$ - $f2\ x)) <$
$eps / 4.$

Lemma $D\_x\_no\_cond : \forall\ x\ a{:}R,\ a \neq 0 \rightarrow D\_x\ no\_cond\ x\ (x + a).$

Lemma $Rabs\_4$:
$\forall\ a\ b\ c\ d{:}R,\ Rabs\ (a + b + c + d) \leq Rabs\ a + Rabs\ b + Rabs\ c + Rabs\ d.$

Lemma Rlt_4 :
$\forall\ a\ b\ c\ d\ e\ f\ g\ h{:}R,$
$a < b \rightarrow c < d \rightarrow e < f \rightarrow g < h \rightarrow a + c + e + g < b + d + f + h.$

Lemma $Rmin\_2 : \forall\ a\ b\ c{:}R,\ a < b \rightarrow a < c \rightarrow a < Rmin\ b\ c.$

Lemma $quadruple : \forall\ x{:}R,\ 4 \times x = x + x + x + x.$

Lemma $quadruple\_var : \forall\ x{:}R,\ x = x / 4 + x / 4 + x / 4 + x / 4.$

Lemma $continuous\_neg\_0$:
$\forall\ (f{:}R \rightarrow R)\ (x0{:}R),$
$continuity\_pt\ f\ x0 \rightarrow$
$f\ x0 \neq 0 \rightarrow$
$\exists\ eps : posreal, (\forall\ h{:}R,\ Rabs\ h < eps \rightarrow f\ (x0 + h) \neq 0).$

Chapter 105

Module Coq.Reals.Ranalysis3

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Ranalysis1*.
Require Import *Ranalysis2*. *Open* Local *Scope R_scope*.

Division

Theorem *derivable_pt_lim_div* :
∀ (*f1 f2*:$R \to R$) (*x l1 l2*:*R*),
derivable_pt_lim f1 x l1 →
derivable_pt_lim f2 x l2 →
f2 x ≠ 0 →
derivable_pt_lim (*f1 / f2*) *x* ((*l1* × *f2 x* - *l2* × *f1 x*) / *Rsqr* (*f2 x*)).

Lemma *derivable_pt_div* :
∀ (*f1 f2*:$R \to R$) (*x*:*R*),
derivable_pt f1 x →
derivable_pt f2 x → *f2 x* ≠ 0 → *derivable_pt* (*f1 / f2*) *x*.

Lemma *derivable_div* :
∀ *f1 f2*:$R \to R$,
derivable f1 →
derivable f2 → (∀ *x*:*R*, *f2 x* ≠ 0) → *derivable* (*f1 / f2*).

Lemma *derive_pt_div* :
∀ (*f1 f2*:$R \to R$) (*x*:*R*) (*pr1*:*derivable_pt f1 x*)
(*pr2*:*derivable_pt f2 x*) (*na*:*f2 x* ≠ 0),
derive_pt (*f1 / f2*) *x* (*derivable_pt_div _ _ _ pr1 pr2 na*) =
(*derive_pt f1 x pr1* × *f2 x* - *derive_pt f2 x pr2* × *f1 x*) / *Rsqr* (*f2 x*).

Chapter 106

Module Coq.Reals.Ranalysis4

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo*.
Require Import *Ranalysis1*.
Require Import *Ranalysis3*.
Require Import *Exp_prop*. *Open* Local *Scope R_scope*.

Lemma *derivable_pt_inv* :
  $\forall$ (f:$R \to R$) (x:R),
    $f\ x \neq 0 \to$ *derivable_pt* $f\ x \to$ *derivable_pt* ($/ f$) x.

Lemma *pr_nu_var* :
  $\forall$ ($f\ g$:$R \to R$) (x:R) (*pr1*:*derivable_pt* $f\ x$) (*pr2*:*derivable_pt* $g\ x$),
    $f = g \to$ *derive_pt* $f\ x$ *pr1* = *derive_pt* $g\ x$ *pr2*.

Lemma *pr_nu_var2* :
  $\forall$ ($f\ g$:$R \to R$) (x:R) (*pr1*:*derivable_pt* $f\ x$) (*pr2*:*derivable_pt* $g\ x$),
    ($\forall$ h:R, $f\ h = g\ h$) $\to$ *derive_pt* $f\ x$ *pr1* = *derive_pt* $g\ x$ *pr2*.

Lemma *derivable_inv* :
  $\forall$ f:$R \to R$, ($\forall$ x:R, $f\ x \neq 0$) $\to$ *derivable* $f \to$ *derivable* ($/ f$).

Lemma *derive_pt_inv* :
  $\forall$ (f:$R \to R$) (x:R) (*pr*:*derivable_pt* $f\ x$) (*na*:$f\ x \neq 0$),
    derive_pt ($/ f$) x (*derivable_pt_inv* $f\ x$ *na pr*) =
    - *derive_pt* $f\ x$ *pr* / *Rsqr* ($f\ x$).

Rabsolu
Lemma *Rabs_derive_1* : $\forall$ x:R, $0 < x \to$ *derivable_pt_lim Rabs* x 1.

Lemma *Rabs_derive_2* : $\forall$ x:R, $x < 0 \to$ *derivable_pt_lim Rabs* x (-1).

Rabsolu is derivable for all x <> 0
Lemma *Rderivable_pt_abs* : $\forall$ x:R, $x \neq 0 \to$ *derivable_pt Rabs* x.

Rabsolu is continuous for all x
Lemma *Rcontinuity_abs* : *continuity Rabs*.

Finite sums : Sum a_k x^k

Lemma *continuity_finite_sum* :
∀ (*An*:*nat* → *R*) (*N*:*nat*),
continuity (fun *y*:*R* ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *An k* × *y* ^ *k*) *N*).

Lemma *derivable_pt_lim_fs* :
∀ (*An*:*nat* → *R*) (*x*:*R*) (*N*:*nat*),
(0 < *N*)%*nat* →
derivable_pt_lim (fun *y*:*R* ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *An k* × *y* ^ *k*) *N*) *x*
(*sum_f_R0* (fun *k*:*nat* ⇒ *INR* (*S k*) × *An* (*S k*) × *x* ^ *k*) (*pred N*)).

Lemma *derivable_pt_lim_finite_sum* :
∀ (*An*:*nat* → *R*) (*x*:*R*) (*N*:*nat*),
derivable_pt_lim (fun *y*:*R* ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *An k* × *y* ^ *k*) *N*) *x*
match *N* with
| *O* ⇒ 0
| _ ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *INR* (*S k*) × *An* (*S k*) × *x* ^ *k*) (*pred N*)
end.

Lemma *derivable_pt_finite_sum* :
∀ (*An*:*nat* → *R*) (*N*:*nat*) (*x*:*R*),
derivable_pt (fun *y*:*R* ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *An k* × *y* ^ *k*) *N*) *x*.

Lemma *derivable_finite_sum* :
∀ (*An*:*nat* → *R*) (*N*:*nat*),
derivable (fun *y*:*R* ⇒ *sum_f_R0* (fun *k*:*nat* ⇒ *An k* × *y* ^ *k*) *N*).

Regularity of hyperbolic functions

Lemma *derivable_pt_lim_cosh* : ∀ *x*:*R*, *derivable_pt_lim cosh x* (*sinh x*).

Lemma *derivable_pt_lim_sinh* : ∀ *x*:*R*, *derivable_pt_lim sinh x* (*cosh x*).

Lemma *derivable_pt_exp* : ∀ *x*:*R*, *derivable_pt exp x*.

Lemma *derivable_pt_cosh* : ∀ *x*:*R*, *derivable_pt cosh x*.

Lemma *derivable_pt_sinh* : ∀ *x*:*R*, *derivable_pt sinh x*.

Lemma *derivable_exp* : *derivable exp*.

Lemma *derivable_cosh* : *derivable cosh*.

Lemma *derivable_sinh* : *derivable sinh*.

Lemma *derive_pt_exp* :
∀ *x*:*R*, *derive_pt exp x* (*derivable_pt_exp x*) = *exp x*.

Lemma *derive_pt_cosh* :
∀ *x*:*R*, *derive_pt cosh x* (*derivable_pt_cosh x*) = *sinh x*.

Lemma *derive_pt_sinh* :
∀ *x*:*R*, *derive_pt sinh x* (*derivable_pt_sinh x*) = *cosh x*.

Chapter 107

Module Coq.Reals.Ranalysis

```
Require Import Rbase.
Require Import Rfunctions.
Require Import Rtrigo.
Require Import SeqSeries.
Require Export Ranalysis1.
Require Export Ranalysis2.
Require Export Ranalysis3.
Require Export Rtopology.
Require Export MVT.
Require Export PSeries_reg.
Require Export Exp_prop.
Require Export Rtrigo_reg.
Require Export Rsqrt_def.
Require Export R_sqrt.
Require Export Rtrigo_calc.
Require Export Rgeom.
Require Export RList.
Require Export Sqrt_reg.
Require Export Ranalysis4.
Require Export Rpower. Open Local Scope R_scope.

Axiom AppVar : R.

Ltac intro_hyp_glob trm :=
  match constr:trm with
    | (?X1 + ?X2)%F ⇒
      match goal with
        | ⊢ (derivable _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
        | ⊢ (continuity _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
        | _ ⇒ idtac
      end
    | (?X1 - ?X2)%F ⇒
      match goal with
        | ⊢ (derivable _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
```

```
      | ⊢ (continuity _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
      | _ ⇒ idtac
    end
| (?X1 × ?X2)%F ⇒
    match goal with
      | ⊢ (derivable _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
      | ⊢ (continuity _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
      | _ ⇒ idtac
    end
| (?X1 / ?X2)%F ⇒
    let aux := constr:X2 in
      match goal with
        | _:(∀ x0:R, aux x0 ≠ 0) ⊢ (derivable _) ⇒
          intro_hyp_glob X1; intro_hyp_glob X2
        | _:(∀ x0:R, aux x0 ≠ 0) ⊢ (continuity _) ⇒
          intro_hyp_glob X1; intro_hyp_glob X2
        | ⊢ (derivable _) ⇒
          cut (∀ x0:R, aux x0 ≠ 0);
            [ intro; intro_hyp_glob X1; intro_hyp_glob X2 | try assumption ]
        | ⊢ (continuity _) ⇒
          cut (∀ x0:R, aux x0 ≠ 0);
            [ intro; intro_hyp_glob X1; intro_hyp_glob X2 | try assumption ]
        | _ ⇒ idtac
      end
| (comp ?X1 ?X2) ⇒
    match goal with
      | ⊢ (derivable _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
      | ⊢ (continuity _) ⇒ intro_hyp_glob X1; intro_hyp_glob X2
      | _ ⇒ idtac
    end
| (- ?X1)%F ⇒
    match goal with
      | ⊢ (derivable _) ⇒ intro_hyp_glob X1
      | ⊢ (continuity _) ⇒ intro_hyp_glob X1
      | _ ⇒ idtac
    end
| (/ ?X1)%F ⇒
    let aux := constr:X1 in
      match goal with
        | _:(∀ x0:R, aux x0 ≠ 0) ⊢ (derivable _) ⇒
          intro_hyp_glob X1
        | _:(∀ x0:R, aux x0 ≠ 0) ⊢ (continuity _) ⇒
          intro_hyp_glob X1
        | ⊢ (derivable _) ⇒
          cut (∀ x0:R, aux x0 ≠ 0);
```

```
                [ intro; intro_hyp_glob X1 | try assumption ]
            | ⊢ (continuity _) ⇒
              cut (∀ x0:R, aux x0 ≠ 0);
                [ intro; intro_hyp_glob X1 | try assumption ]
            | _ ⇒ idtac
          end
      | cos ⇒ idtac
      | sin ⇒ idtac
      | cosh ⇒ idtac
      | sinh ⇒ idtac
      | exp ⇒ idtac
      | Rsqr ⇒ idtac
      | sqrt ⇒ idtac
      | id ⇒ idtac
      | (fct_cte _) ⇒ idtac
      | (pow_fct _) ⇒ idtac
      | Rabs ⇒ idtac
      | ?X1 ⇒
        let p := constr:X1 in
          match goal with
            | _:(derivable p) ⊢ _ ⇒ idtac
            | ⊢ (derivable p) ⇒ idtac
            | ⊢ (derivable _) ⇒
              cut (True → derivable p);
                [ intro HYPPD; cut (derivable p);
                  [ intro; clear HYPPD | apply HYPPD; clear HYPPD; trivial ]
                  | idtac ]
            | _:(continuity p) ⊢ _ ⇒ idtac
            | ⊢ (continuity p) ⇒ idtac
            | ⊢ (continuity _) ⇒
              cut (True → continuity p);
                [ intro HYPPD; cut (continuity p);
                  [ intro; clear HYPPD | apply HYPPD; clear HYPPD; trivial ]
                  | idtac ]
            | _ ⇒ idtac
          end
  end.

Ltac intro_hyp_pt trm pt :=
  match constr:trm with
    | (?X1 + ?X2)%F ⇒
      match goal with
        | ⊢ (derivable_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
        | ⊢ (continuity_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
        | ⊢ (derive_pt _ _ _ = _) ⇒
          intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
```

```
    | _ ⇒ idtac
  end
| (?X1 - ?X2)%F ⇒
  match goal with
    | ⊢ (derivable_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | ⊢ (continuity_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | ⊢ (derive_pt _ _ _ = _) ⇒
      intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | _ ⇒ idtac
  end
| (?X1 × ?X2)%F ⇒
  match goal with
    | ⊢ (derivable_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | ⊢ (continuity_pt _ _) ⇒ intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | ⊢ (derive_pt _ _ _ = _) ⇒
      intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
    | _ ⇒ idtac
  end
| (?X1 / ?X2)%F ⇒
  let aux := constr:X2 in
    match goal with
      | _:(aux pt ≠ 0) ⊢ (derivable_pt _ _) ⇒
        intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | _:(aux pt ≠ 0) ⊢ (continuity_pt _ _) ⇒
        intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | _:(aux pt ≠ 0) ⊢ (derive_pt _ _ _ = _) ⇒
        intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (derivable_pt _ _) ⇒
        generalize (id pt); intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (continuity_pt _ _) ⇒
        generalize (id pt); intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (derive_pt _ _ _ = _) ⇒
        generalize (id pt); intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt
      | ⊢ (derivable_pt _ _) ⇒
        cut (aux pt ≠ 0);
          [ intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt | try assumption ]
      | ⊢ (continuity_pt _ _) ⇒
        cut (aux pt ≠ 0);
          [ intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt | try assumption ]
      | ⊢ (derive_pt _ _ _ = _) ⇒
        cut (aux pt ≠ 0);
          [ intro; intro_hyp_pt X1 pt; intro_hyp_pt X2 pt | try assumption ]
      | _ ⇒ idtac
    end
| (comp ?X1 ?X2) ⇒
```

```
    match goal with
      | ⊢ (derivable_pt _ _) ⇒
        let pt_f1 := eval cbv beta in (X2 pt) in
          (intro_hyp_pt X1 pt_f1; intro_hyp_pt X2 pt)
      | ⊢ (continuity_pt _ _) ⇒
        let pt_f1 := eval cbv beta in (X2 pt) in
          (intro_hyp_pt X1 pt_f1; intro_hyp_pt X2 pt)
      | ⊢ (derive_pt _ _ _ = _) ⇒
        let pt_f1 := eval cbv beta in (X2 pt) in
          (intro_hyp_pt X1 pt_f1; intro_hyp_pt X2 pt)
      | _ ⇒ idtac
    end
  | (- ?X1)%F ⇒
    match goal with
      | ⊢ (derivable_pt _ _) ⇒ intro_hyp_pt X1 pt
      | ⊢ (continuity_pt _ _) ⇒ intro_hyp_pt X1 pt
      | ⊢ (derive_pt _ _ _ = _) ⇒ intro_hyp_pt X1 pt
      | _ ⇒ idtac
    end
  | (/ ?X1)%F ⇒
    let aux := constr:X1 in
      match goal with
        | _:(aux pt ≠ 0) ⊢ (derivable_pt _ _) ⇒
          intro_hyp_pt X1 pt
        | _:(aux pt ≠ 0) ⊢ (continuity_pt _ _) ⇒
          intro_hyp_pt X1 pt
        | _:(aux pt ≠ 0) ⊢ (derive_pt _ _ _ = _) ⇒
          intro_hyp_pt X1 pt
        | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (derivable_pt _ _) ⇒
          generalize (id pt); intro; intro_hyp_pt X1 pt
        | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (continuity_pt _ _) ⇒
          generalize (id pt); intro; intro_hyp_pt X1 pt
        | id:(∀ x0:R, aux x0 ≠ 0) ⊢ (derive_pt _ _ _ = _) ⇒
          generalize (id pt); intro; intro_hyp_pt X1 pt
        | ⊢ (derivable_pt _ _) ⇒
          cut (aux pt ≠ 0); [ intro; intro_hyp_pt X1 pt | try assumption ]
        | ⊢ (continuity_pt _ _) ⇒
          cut (aux pt ≠ 0); [ intro; intro_hyp_pt X1 pt | try assumption ]
        | ⊢ (derive_pt _ _ _ = _) ⇒
          cut (aux pt ≠ 0); [ intro; intro_hyp_pt X1 pt | try assumption ]
        | _ ⇒ idtac
      end
  | cos ⇒ idtac
  | sin ⇒ idtac
  | cosh ⇒ idtac
```

```
| sinh ⇒ idtac
| exp ⇒ idtac
| Rsqr ⇒ idtac
| id ⇒ idtac
| (fct_cte _) ⇒ idtac
| (pow_fct _) ⇒ idtac
| sqrt ⇒
  match goal with
    | ⊢ (derivable_pt _ _) ⇒ cut (0 < pt); [ intro | try assumption ]
    | ⊢ (continuity_pt _ _) ⇒
      cut (0 ≤ pt); [ intro | try assumption ]
    | ⊢ (derive_pt _ _ _ = _) ⇒
      cut (0 < pt); [ intro | try assumption ]
    | _ ⇒ idtac
  end
| Rabs ⇒
  match goal with
    | ⊢ (derivable_pt _ _) ⇒
      cut (pt ≠ 0); [ intro | try assumption ]
    | _ ⇒ idtac
  end
| ?X1 ⇒
  let p := constr:X1 in
    match goal with
      | _:(derivable_pt p pt) ⊢ _ ⇒ idtac
      | ⊢ (derivable_pt p pt) ⇒ idtac
      | ⊢ (derivable_pt _ _) ⇒
        cut (True → derivable_pt p pt);
          [ intro HYPPD; cut (derivable_pt p pt);
            [ intro; clear HYPPD | apply HYPPD; clear HYPPD; trivial ]
            | idtac ]
      | _:(continuity_pt p pt) ⊢ _ ⇒ idtac
      | ⊢ (continuity_pt p pt) ⇒ idtac
      | ⊢ (continuity_pt _ _) ⇒
        cut (True → continuity_pt p pt);
          [ intro HYPPD; cut (continuity_pt p pt);
            [ intro; clear HYPPD | apply HYPPD; clear HYPPD; trivial ]
            | idtac ]
      | ⊢ (derive_pt _ _ _ = _) ⇒
        cut (True → derivable_pt p pt);
          [ intro HYPPD; cut (derivable_pt p pt);
            [ intro; clear HYPPD | apply HYPPD; clear HYPPD; trivial ]
            | idtac ]
      | _ ⇒ idtac
    end
```

```
  end.

Ltac is_diff_pt :=
  match goal with
    | ⊢ (derivable_pt Rsqr _) ⇒

          apply derivable_pt_Rsqr
    | ⊢ (derivable_pt id ?X1) ⇒ apply (derivable_pt_id X1)
    | ⊢ (derivable_pt (fct_cte _) _) ⇒ apply derivable_pt_const
    | ⊢ (derivable_pt sin _) ⇒ apply derivable_pt_sin
    | ⊢ (derivable_pt cos _) ⇒ apply derivable_pt_cos
    | ⊢ (derivable_pt sinh _) ⇒ apply derivable_pt_sinh
    | ⊢ (derivable_pt cosh _) ⇒ apply derivable_pt_cosh
    | ⊢ (derivable_pt exp _) ⇒ apply derivable_pt_exp
    | ⊢ (derivable_pt (pow_fct _) _) ⇒
      unfold pow_fct in ⊢ ×; apply derivable_pt_pow
    | ⊢ (derivable_pt sqrt ?X1) ⇒
      apply (derivable_pt_sqrt X1);
        assumption ||
          unfold plus_fct, minus_fct, opp_fct, mult_fct, div_fct, inv_fct,
            comp, id, fct_cte, pow_fct in ⊢ ×
    | ⊢ (derivable_pt Rabs ?X1) ⇒
      apply (Rderivable_pt_abs X1);
        assumption ||
          unfold plus_fct, minus_fct, opp_fct, mult_fct, div_fct, inv_fct,
            comp, id, fct_cte, pow_fct in ⊢ ×
                    | ⊢ (derivable_pt (?X1 + ?X2) ?X3) ⇒
      apply (derivable_pt_plus X1 X2 X3); is_diff_pt
          | ⊢ (derivable_pt (?X1 - ?X2) ?X3) ⇒
      apply (derivable_pt_minus X1 X2 X3); is_diff_pt
          | ⊢ (derivable_pt (- ?X1) ?X2) ⇒
      apply (derivable_pt_opp X1 X2);
        is_diff_pt
          | ⊢ (derivable_pt (mult_real_fct ?X1 ?X2) ?X3) ⇒
      apply (derivable_pt_scal X2 X1 X3); is_diff_pt
          | ⊢ (derivable_pt (?X1 × ?X2) ?X3) ⇒
      apply (derivable_pt_mult X1 X2 X3); is_diff_pt
          | ⊢ (derivable_pt (?X1 / ?X2) ?X3) ⇒
      apply (derivable_pt_div X1 X2 X3);
        [ is_diff_pt | is_diff_pt | try assumption || unfold plus_fct, mult_fct, div_fct, minus_fct,
opp_fct, inv_fct, comp, pow_fct, id, fct_cte in |- * ]
    | ⊢ (derivable_pt (/ ?X1) ?X2) ⇒

            apply (derivable_pt_inv X1 X2);
        [ assumption || unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, comp,
pow_fct, id, fct_cte in |- * | is_diff_pt ]
```

```
    | ⊢ (derivable_pt (comp ?X1 ?X2) ?X3) ⇒

                apply (derivable_pt_comp X2 X1 X3); is_diff_pt
    | _:(derivable_pt ?X1 ?X2) ⊢ (derivable_pt ?X1 ?X2) ⇒
      assumption
    | _:(derivable ?X1) ⊢ (derivable_pt ?X1 ?X2) ⇒
      cut (derivable X1); [ intro HypDDPT; apply HypDDPT | assumption ]
    | ⊢ (True → derivable_pt _ _) ⇒
      intro HypTruE; clear HypTruE; is_diff_pt
    | _ ⇒
      try
        unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id,
          fct_cte, comp, pow_fct in ⊢ ×
  end.

Ltac is_diff_glob :=
  match goal with
    | ⊢ (derivable Rsqr) ⇒
        apply derivable_Rsqr
    | ⊢ (derivable id) ⇒ apply derivable_id
    | ⊢ (derivable (fct_cte _)) ⇒ apply derivable_const
    | ⊢ (derivable sin) ⇒ apply derivable_sin
    | ⊢ (derivable cos) ⇒ apply derivable_cos
    | ⊢ (derivable cosh) ⇒ apply derivable_cosh
    | ⊢ (derivable sinh) ⇒ apply derivable_sinh
    | ⊢ (derivable exp) ⇒ apply derivable_exp
    | ⊢ (derivable (pow_fct _)) ⇒
      unfold pow_fct in ⊢ ×;
        apply derivable_pow
                    | ⊢ (derivable (?X1 + ?X2)) ⇒
      apply (derivable_plus X1 X2); is_diff_glob
            | ⊢ (derivable (?X1 - ?X2)) ⇒
      apply (derivable_minus X1 X2); is_diff_glob
            | ⊢ (derivable (- ?X1)) ⇒
      apply (derivable_opp X1);
        is_diff_glob
            | ⊢ (derivable (mult_real_fct ?X1 ?X2)) ⇒
      apply (derivable_scal X2 X1); is_diff_glob
            | ⊢ (derivable (?X1 × ?X2)) ⇒
      apply (derivable_mult X1 X2); is_diff_glob
            | ⊢ (derivable (?X1 / ?X2)) ⇒
      apply (derivable_div X1 X2);
        [ is_diff_glob | is_diff_glob | try assumption || unfold plus_fct, mult_fct, div_fct, minus_fct,
opp_fct, inv_fct, id, fct_cte, comp, pow_fct in |- * ]
    | ⊢ (derivable (/ ?X1)) ⇒
```

```
                apply (derivable_inv X1);
          [ try assumption || unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id,
fct_cte, comp, pow_fct in |- * | is_diff_glob ]
    | ⊢ (derivable (comp sqrt _)) ⇒

                unfold derivable in ⊢ ×; intro; try is_diff_pt
    | ⊢ (derivable (comp Rabs _)) ⇒
      unfold derivable in ⊢ ×; intro; try is_diff_pt
    | ⊢ (derivable (comp ?X1 ?X2)) ⇒
      apply (derivable_comp X2 X1); is_diff_glob
    | _:(derivable ?X1) ⊢ (derivable ?X1) ⇒ assumption
    | ⊢ (True → derivable _) ⇒
      intro HypTruE; clear HypTruE; is_diff_glob
    | _ ⇒
      try
        unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id,
          fct_cte, comp, pow_fct in ⊢ ×
  end.

Ltac is_cont_pt :=
  match goal with
    | ⊢ (continuity_pt Rsqr _) ⇒

                apply derivable_continuous_pt; apply derivable_pt_Rsqr
    | ⊢ (continuity_pt id ?X1) ⇒
      apply derivable_continuous_pt; apply (derivable_pt_id X1)
    | ⊢ (continuity_pt (fct_cte _) _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_const
    | ⊢ (continuity_pt sin _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_sin
    | ⊢ (continuity_pt cos _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_cos
    | ⊢ (continuity_pt sinh _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_sinh
    | ⊢ (continuity_pt cosh _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_cosh
    | ⊢ (continuity_pt exp _) ⇒
      apply derivable_continuous_pt; apply derivable_pt_exp
    | ⊢ (continuity_pt (pow_fct _) _) ⇒
      unfold pow_fct in ⊢ ×; apply derivable_continuous_pt;
        apply derivable_pt_pow
    | ⊢ (continuity_pt sqrt ?X1) ⇒
      apply continuity_pt_sqrt;
        assumption ||
          unfold plus_fct, minus_fct, opp_fct, mult_fct, div_fct, inv_fct,
            comp, id, fct_cte, pow_fct in ⊢ ×
```

```
    | ⊢ (continuity_pt Rabs ?X1) ⇒
      apply (Rcontinuity_abs X1)
                    | ⊢ (continuity_pt (?X1 + ?X2) ?X3) ⇒
      apply (continuity_pt_plus X1 X2 X3); is_cont_pt
          | ⊢ (continuity_pt (?X1 - ?X2) ?X3) ⇒
      apply (continuity_pt_minus X1 X2 X3); is_cont_pt
          | ⊢ (continuity_pt (- ?X1) ?X2) ⇒
      apply (continuity_pt_opp X1 X2);
        is_cont_pt
          | ⊢ (continuity_pt (mult_real_fct ?X1 ?X2) ?X3) ⇒
      apply (continuity_pt_scal X2 X1 X3); is_cont_pt
          | ⊢ (continuity_pt (?X1 × ?X2) ?X3) ⇒
      apply (continuity_pt_mult X1 X2 X3); is_cont_pt
          | ⊢ (continuity_pt (?X1 / ?X2) ?X3) ⇒
      apply (continuity_pt_div X1 X2 X3);
        [ is_cont_pt | is_cont_pt | try assumption || unfold plus_fct, mult_fct, div_fct, minus_fct,
opp_fct, inv_fct, comp, id, fct_cte, pow_fct in |- * ]
    | ⊢ (continuity_pt (/ ?X1) ?X2) ⇒

            apply (continuity_pt_inv X1 X2);
        [ is_cont_pt | assumption || unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct,
comp, id, fct_cte, pow_fct in |- * ]
    | ⊢ (continuity_pt (comp ?X1 ?X2) ?X3) ⇒

            apply (continuity_pt_comp X2 X1 X3); is_cont_pt
    | _:(continuity_pt ?X1 ?X2) ⊢ (continuity_pt ?X1 ?X2) ⇒
      assumption
    | _:(continuity ?X1) ⊢ (continuity_pt ?X1 ?X2) ⇒
      cut (continuity X1); [ intro HypDDPT; apply HypDDPT | assumption ]
    | _:(derivable_pt ?X1 ?X2) ⊢ (continuity_pt ?X1 ?X2) ⇒
      apply derivable_continuous_pt; assumption
    | _:(derivable ?X1) ⊢ (continuity_pt ?X1 ?X2) ⇒
      cut (continuity X1);
        [ intro HypDDPT; apply HypDDPT | apply derivable_continuous; assumption ]
    | ⊢ (True → continuity_pt _ _) ⇒
      intro HypTruE; clear HypTruE; is_cont_pt
    | _ ⇒
      try
        unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id,
          fct_cte, comp, pow_fct in ⊢ ×
  end.

Ltac is_cont_glob :=
  match goal with
    | ⊢ (continuity Rsqr) ⇒
```

```
                apply derivable_continuous; apply derivable_Rsqr
    | ⊢ (continuity id) ⇒ apply derivable_continuous; apply derivable_id
    | ⊢ (continuity (fct_cte _)) ⇒
      apply derivable_continuous; apply derivable_const
    | ⊢ (continuity sin) ⇒ apply derivable_continuous; apply derivable_sin
    | ⊢ (continuity cos) ⇒ apply derivable_continuous; apply derivable_cos
    | ⊢ (continuity exp) ⇒ apply derivable_continuous; apply derivable_exp
    | ⊢ (continuity (pow_fct _)) ⇒
      unfold pow_fct in ⊢ ×; apply derivable_continuous; apply derivable_pow
    | ⊢ (continuity sinh) ⇒
      apply derivable_continuous; apply derivable_sinh
    | ⊢ (continuity cosh) ⇒
      apply derivable_continuous; apply derivable_cosh
    | ⊢ (continuity Rabs) ⇒
      apply Rcontinuity_abs
                    | ⊢ (continuity (?X1 + ?X2)) ⇒
      apply (continuity_plus X1 X2);
        try is_cont_glob || assumption
                | ⊢ (continuity (?X1 - ?X2)) ⇒
      apply (continuity_minus X1 X2);
        try is_cont_glob || assumption
                | ⊢ (continuity (- ?X1)) ⇒
      apply (continuity_opp X1); try is_cont_glob || assumption
                                                | ⊢ (continuity (/ ?X1)) ⇒
      apply (continuity_inv X1);
        try is_cont_glob || assumption
                | ⊢ (continuity (mult_real_fct ?X1 ?X2)) ⇒
      apply (continuity_scal X2 X1);
        try is_cont_glob || assumption
                | ⊢ (continuity (?X1 × ?X2)) ⇒
      apply (continuity_mult X1 X2);
        try is_cont_glob || assumption
                | ⊢ (continuity (?X1 / ?X2)) ⇒
      apply (continuity_div X1 X2);
        [ try is_cont_glob || assumption | try is_cont_glob || assumption | try assumption || unfold
plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id, fct_cte, pow_fct in |- * ]
    | ⊢ (continuity (comp sqrt _)) ⇒

                unfold continuity_pt in ⊢ ×; intro; try is_cont_pt
    | ⊢ (continuity (comp ?X1 ?X2)) ⇒
      apply (continuity_comp X2 X1); try is_cont_glob || assumption
    | _:(continuity ?X1) ⊢ (continuity ?X1) ⇒ assumption
    | ⊢ (True → continuity _) ⇒
      intro HypTruE; clear HypTruE; is_cont_glob
    | _:(derivable ?X1) ⊢ (continuity ?X1) ⇒
```

```
      apply derivable_continuous; assumption
    | _ ⇒
      try
        unfold plus_fct, mult_fct, div_fct, minus_fct, opp_fct, inv_fct, id,
          fct_cte, comp, pow_fct in ⊢ ×
  end.

Ltac rew_term trm :=
  match constr:trm with
    | (?X1 + ?X2) ⇒
      let p1 := rew_term X1 with p2 := rew_term X2 in
        match constr:p1 with
          | (fct_cte ?X3) ⇒
            match constr:p2 with
              | (fct_cte ?X4) ⇒ constr:(fct_cte (X3 + X4))
              | _ ⇒ constr:(p1 + p2)%F
            end
          | _ ⇒ constr:(p1 + p2)%F
        end
    | (?X1 - ?X2) ⇒
      let p1 := rew_term X1 with p2 := rew_term X2 in
        match constr:p1 with
          | (fct_cte ?X3) ⇒
            match constr:p2 with
              | (fct_cte ?X4) ⇒ constr:(fct_cte (X3 - X4))
              | _ ⇒ constr:(p1 - p2)%F
            end
          | _ ⇒ constr:(p1 - p2)%F
        end
    | (?X1 / ?X2) ⇒
      let p1 := rew_term X1 with p2 := rew_term X2 in
        match constr:p1 with
          | (fct_cte ?X3) ⇒
            match constr:p2 with
              | (fct_cte ?X4) ⇒ constr:(fct_cte (X3 / X4))
              | _ ⇒ constr:(p1 / p2)%F
            end
          | _ ⇒
            match constr:p2 with
              | (fct_cte ?X4) ⇒ constr:(p1 × fct_cte (/ X4))%F
              | _ ⇒ constr:(p1 / p2)%F
            end
        end
    | (?X1 × / ?X2) ⇒
      let p1 := rew_term X1 with p2 := rew_term X2 in
        match constr:p1 with
```

```
        | (fct_cte ?X3) ⇒
          match constr:p2 with
            | (fct_cte ?X4) ⇒ constr:(fct_cte (X3 / X4))
            | _ ⇒ constr:(p1 / p2)%F
          end
        | _ ⇒
          match constr:p2 with
            | (fct_cte ?X4) ⇒ constr:(p1 × fct_cte (/ X4))%F
            | _ ⇒ constr:(p1 / p2)%F
          end
      end
  | (?X1 × ?X2) ⇒
    let p1 := rew_term X1 with p2 := rew_term X2 in
      match constr:p1 with
        | (fct_cte ?X3) ⇒
          match constr:p2 with
            | (fct_cte ?X4) ⇒ constr:(fct_cte (X3 × X4))
            | _ ⇒ constr:(p1 × p2)%F
          end
        | _ ⇒ constr:(p1 × p2)%F
      end
  | (- ?X1) ⇒
    let p := rew_term X1 in
      match constr:p with
        | (fct_cte ?X2) ⇒ constr:(fct_cte (- X2))
        | _ ⇒ constr:(- p)%F
      end
  | (/ ?X1) ⇒
    let p := rew_term X1 in
      match constr:p with
        | (fct_cte ?X2) ⇒ constr:(fct_cte (/ X2))
        | _ ⇒ constr:(/ p)%F
      end
  | (?X1 AppVar) ⇒ constr:X1
  | (?X1 ?X2) ⇒
    let p := rew_term X2 in
      match constr:p with
        | (fct_cte ?X3) ⇒ constr:(fct_cte (X1 X3))
        | _ ⇒ constr:(comp X1 p)
      end
  | AppVar ⇒ constr:id
  | (AppVar ^ ?X1) ⇒ constr:(pow_fct X1)
  | (?X1 ^ ?X2) ⇒
    let p := rew_term X1 in
      match constr:p with
```

```
          | (fct_cte ?X3) ⇒ constr:(fct_cte (pow_fct X2 X3))
          | _ ⇒ constr:(comp (pow_fct X2) p)
        end
    | ?X1 ⇒ constr:(fct_cte X1)
  end.

Ltac deriv_proof trm pt :=
  match constr:trm with
    | (?X1 + ?X2)%F ⇒
      let p1 := deriv_proof X1 pt with p2 := deriv_proof X2 pt in
        constr:(derivable_pt_plus X1 X2 pt p1 p2)
    | (?X1 - ?X2)%F ⇒
      let p1 := deriv_proof X1 pt with p2 := deriv_proof X2 pt in
        constr:(derivable_pt_minus X1 X2 pt p1 p2)
    | (?X1 × ?X2)%F ⇒
      let p1 := deriv_proof X1 pt with p2 := deriv_proof X2 pt in
        constr:(derivable_pt_mult X1 X2 pt p1 p2)
    | (?X1 / ?X2)%F ⇒
      match goal with
        | id:(?X2 pt ≠ 0) ⊢ _ ⇒
          let p1 := deriv_proof X1 pt with p2 := deriv_proof X2 pt in
            constr:(derivable_pt_div X1 X2 pt p1 p2 id)
        | _ ⇒ constr:False
      end
    | (/ ?X1)%F ⇒
      match goal with
        | id:(?X1 pt ≠ 0) ⊢ _ ⇒
          let p1 := deriv_proof X1 pt in
            constr:(derivable_pt_inv X1 pt p1 id)
        | _ ⇒ constr:False
      end
    | (comp ?X1 ?X2) ⇒
      let pt_f1 := eval cbv beta in (X2 pt) in
        let p1 := deriv_proof X1 pt_f1 with p2 := deriv_proof X2 pt in
          constr:(derivable_pt_comp X2 X1 pt p2 p1)
    | (- ?X1)%F ⇒
      let p1 := deriv_proof X1 pt in
        constr:(derivable_pt_opp X1 pt p1)
    | sin ⇒ constr:(derivable_pt_sin pt)
    | cos ⇒ constr:(derivable_pt_cos pt)
    | sinh ⇒ constr:(derivable_pt_sinh pt)
    | cosh ⇒ constr:(derivable_pt_cosh pt)
    | exp ⇒ constr:(derivable_pt_exp pt)
    | id ⇒ constr:(derivable_pt_id pt)
    | Rsqr ⇒ constr:(derivable_pt_Rsqr pt)
    | sqrt ⇒
```

```
      match goal with
        | id:(0 < pt) ⊢ _ ⇒ constr:(derivable_pt_sqrt pt id)
        | _ ⇒ constr:False
      end
    | (fct_cte ?X1) ⇒ constr:(derivable_pt_const X1 pt)
    | ?X1 ⇒
      let aux := constr:X1 in
        match goal with
          | id:(derivable_pt aux pt) ⊢ _ ⇒ constr:id
          | id:(derivable aux) ⊢ _ ⇒ constr:(id pt)
          | _ ⇒ constr:False
        end
  end.

Ltac simplify_derive trm pt :=
  match constr:trm with
    | (?X1 + ?X2)%F ⇒
      try rewrite derive_pt_plus; simplify_derive X1 pt;
        simplify_derive X2 pt
    | (?X1 - ?X2)%F ⇒
      try rewrite derive_pt_minus; simplify_derive X1 pt;
        simplify_derive X2 pt
    | (?X1 × ?X2)%F ⇒
      try rewrite derive_pt_mult; simplify_derive X1 pt;
        simplify_derive X2 pt
    | (?X1 / ?X2)%F ⇒
      try rewrite derive_pt_div; simplify_derive X1 pt; simplify_derive X2 pt
    | (comp ?X1 ?X2) ⇒
      let pt_f1 := eval cbv beta in (X2 pt) in
        (try rewrite derive_pt_comp; simplify_derive X1 pt_f1;
          simplify_derive X2 pt)
    | (- ?X1)%F ⇒ try rewrite derive_pt_opp; simplify_derive X1 pt
    | (/ ?X1)%F ⇒
      try rewrite derive_pt_inv; simplify_derive X1 pt
    | (fct_cte ?X1) ⇒ try rewrite derive_pt_const
    | id ⇒ try rewrite derive_pt_id
    | sin ⇒ try rewrite derive_pt_sin
    | cos ⇒ try rewrite derive_pt_cos
    | sinh ⇒ try rewrite derive_pt_sinh
    | cosh ⇒ try rewrite derive_pt_cosh
    | exp ⇒ try rewrite derive_pt_exp
    | Rsqr ⇒ try rewrite derive_pt_Rsqr
    | sqrt ⇒ try rewrite derive_pt_sqrt
    | ?X1 ⇒
      let aux := constr:X1 in
        match goal with
```

```
          | id:(derive_pt aux pt ?X2 = _),H:(derivable aux) ⊢ _ ⇒
            try replace (derive_pt aux pt (H pt)) with (derive_pt aux pt X2);
              [ rewrite id | apply pr_nu ]
          | id:(derive_pt aux pt ?X2 = _),H:(derivable_pt aux pt) ⊢ _ ⇒
            try replace (derive_pt aux pt H) with (derive_pt aux pt X2);
              [ rewrite id | apply pr_nu ]
          | _ ⇒ idtac
        end
    | _ ⇒ idtac
  end.

Ltac reg :=
  match goal with
    | ⊢ (derivable_pt ?X1 ?X2) ⇒
      let trm := eval cbv beta in (X1 AppVar) in
        let aux := rew_term trm in
          (intro_hyp_pt aux X2;
            try (change (derivable_pt aux X2) in ⊢ ×; is_diff_pt) || is_diff_pt)
    | ⊢ (derivable ?X1) ⇒
      let trm := eval cbv beta in (X1 AppVar) in
        let aux := rew_term trm in
          (intro_hyp_glob aux;
            try (change (derivable aux) in ⊢ ×; is_diff_glob) || is_diff_glob)
    | ⊢ (continuity ?X1) ⇒
      let trm := eval cbv beta in (X1 AppVar) in
        let aux := rew_term trm in
          (intro_hyp_glob aux;
            try (change (continuity aux) in ⊢ ×; is_cont_glob) || is_cont_glob)
    | ⊢ (continuity_pt ?X1 ?X2) ⇒
      let trm := eval cbv beta in (X1 AppVar) in
        let aux := rew_term trm in
          (intro_hyp_pt aux X2;
            try (change (continuity_pt aux X2) in ⊢ ×; is_cont_pt) || is_cont_pt)
    | ⊢ (derive_pt ?X1 ?X2 ?X3 = ?X4) ⇒
      let trm := eval cbv beta in (X1 AppVar) in
      let aux := rew_term trm in
      intro_hyp_pt aux X2;
      (let aux2 := deriv_proof aux X2 in
       try
         (replace (derive_pt X1 X2 X3) with (derive_pt aux X2 aux2);
           [ simplify_derive aux X2; try unfold plus_fct, minus_fct, mult_fct, div_fct, id, fct_cte,
inv_fct, opp_fct in |- *; ring || ring_simplify | try apply pr_nu ]) || is_diff_pt)
  end.
```

Chapter 108

Module Coq.Reals.Raxioms

Axiomatisation of the classical reals
Require Export *ZArith_base*.
Require Export *Rdefinitions*.
Open Local *Scope R_scope*.

108.1 Field axioms

108.1.1 Addition

Axiom *Rplus_comm* : $\forall$ *r1 r2*:*R*, *r1* + *r2* = *r2* + *r1*.
Hint *Resolve Rplus_comm*: *real*.

Axiom *Rplus_assoc* : $\forall$ *r1 r2 r3*:*R*, *r1* + *r2* + *r3* = *r1* + (*r2* + *r3*).
Hint *Resolve Rplus_assoc*: *real*.

Axiom *Rplus_opp_r* : $\forall$ *r*:*R*, *r* + - *r* = 0.
Hint *Resolve Rplus_opp_r*: *real v62*.

Axiom *Rplus_0_l* : $\forall$ *r*:*R*, 0 + *r* = *r*.
Hint *Resolve Rplus_0_l*: *real*.

108.1.2 Multiplication

Axiom *Rmult_comm* : $\forall$ *r1 r2*:*R*, *r1* × *r2* = *r2* × *r1*.
Hint *Resolve Rmult_comm*: *real v62*.

Axiom *Rmult_assoc* : $\forall$ *r1 r2 r3*:*R*, *r1* × *r2* × *r3* = *r1* × (*r2* × *r3*).
Hint *Resolve Rmult_assoc*: *real v62*.

Axiom *Rinv_l* : $\forall$ *r*:*R*, *r* ≠ 0 → / *r* × *r* = 1.
Hint *Resolve Rinv_l*: *real*.

Axiom *Rmult_1_l* : $\forall$ *r*:*R*, 1 × *r* = *r*.
Hint *Resolve Rmult_1_l*: *real*.

Axiom $R1\_neq\_R0 : 1 \neq 0$.
Hint *Resolve R1_neq_R0: real.*

108.1.3 Distributivity

Axiom
$Rmult\_plus\_distr\_l : \forall\ r1\ r2\ r3{:}R,\ r1 \times (r2 + r3) = r1 \times r2 + r1 \times r3$.
Hint *Resolve Rmult_plus_distr_l: real v62.*

108.2 Order axioms

108.2.1 Total Order

Axiom $total\_order\_T : \forall\ r1\ r2{:}R,\ \{r1 < r2\} + \{r1 = r2\} + \{r1 > r2\}$.

108.2.2 Lower

Axiom $Rlt\_asym : \forall\ r1\ r2{:}R,\ r1 < r2 \rightarrow \neg\ r2 < r1$.

Axiom $Rlt\_trans : \forall\ r1\ r2\ r3{:}R,\ r1 < r2 \rightarrow r2 < r3 \rightarrow r1 < r3$.

Axiom $Rplus\_lt\_compat\_l : \forall\ r\ r1\ r2{:}R,\ r1 < r2 \rightarrow r + r1 < r + r2$.

Axiom
$Rmult\_lt\_compat\_l : \forall\ r\ r1\ r2{:}R,\ 0 < r \rightarrow r1 < r2 \rightarrow r \times r1 < r \times r2$.

Hint *Resolve Rlt_asym Rplus_lt_compat_l Rmult_lt_compat_l: real.*

108.3 Injection from N to R

108.4 Injection from Z to R

Definition $IZR\ (z{:}Z) : R :=$
  match z with
  | $Z0 \Rightarrow 0$
  | $Zpos\ n \Rightarrow INR\ (nat\_of\_P\ n)$
  | $Zneg\ n \Rightarrow$ - $INR\ (nat\_of\_P\ n)$
  end.

108.5 R Archimedian

Axiom $archimed : \forall\ r{:}R,\ IZR\ (up\ r) > r \wedge IZR\ (up\ r)$ - $r \leq 1$.

108.6 *R* Complete

Definition $is\_upper\_bound\ (E{:}R \to \mathsf{Prop})\ (m{:}R) := \forall\ x{:}R,\ E\ x \to x \leq m.$

Definition $bound\ (E{:}R \to \mathsf{Prop}) := \exists\ m : R,\ is\_upper\_bound\ E\ m.$

Definition $is\_lub\ (E{:}R \to \mathsf{Prop})\ (m{:}R) :=$
$is\_upper\_bound\ E\ m \wedge (\forall\ b{:}R,\ is\_upper\_bound\ E\ b \to m \leq b).$

Axiom
completeness :
$\forall\ E{:}R \to \mathsf{Prop},$
$bound\ E \to (\exists\ x : R,\ E\ x) \to sigT\ (\mathsf{fun}\ m{:}R \Rightarrow is\_lub\ E\ m).$

Chapter 109

Module Coq.Reals.Rbase

Require Export *Rdefinitions*.
Require Export *Raxioms*.
Require Export *RIneq*.
Require Export *DiscrR*.

Chapter 110

Module Coq.Reals.Rbasic_fun

Complements for the real numbers
Require Import *Rbase*.
Require Import *R_Ifp*.
Require Import *Fourier*. *Open* Local *Scope R_scope*.

Implicit Type $r : R$.

110.1 Rmin

Definition $Rmin\ (x\ y{:}R) : R :=$
  match $Rle\_dec\ x\ y$ with
    | $left\ \_ \Rightarrow x$
    | $right\ \_ \Rightarrow y$
  end.

Lemma $Rmin\_Rgt\_l : \forall\ r1\ r2\ r, Rmin\ r1\ r2 > r \rightarrow r1 > r \wedge r2 > r.$

Lemma $Rmin\_Rgt\_r : \forall\ r1\ r2\ r, r1 > r \wedge r2 > r \rightarrow Rmin\ r1\ r2 > r.$

Lemma $Rmin\_Rgt : \forall\ r1\ r2\ r, Rmin\ r1\ r2 > r \leftrightarrow r1 > r \wedge r2 > r.$

Lemma $Rmin\_l : \forall\ x\ y{:}R, Rmin\ x\ y \leq x.$

Lemma $Rmin\_r : \forall\ x\ y{:}R, Rmin\ x\ y \leq y.$

Lemma $Rmin\_comm : \forall\ a\ b{:}R, Rmin\ a\ b = Rmin\ b\ a.$

Lemma $Rmin\_stable\_in\_posreal : \forall\ x\ y{:}posreal, 0 < Rmin\ x\ y.$

110.2 Rmax

Definition $Rmax\ (x\ y{:}R) : R :=$
  match $Rle\_dec\ x\ y$ with
    | $left\ \_ \Rightarrow y$
    | $right\ \_ \Rightarrow x$
  end.

Lemma $Rmax\_Rle : \forall\ r1\ r2\ r, r \leq Rmax\ r1\ r2 \leftrightarrow r \leq r1 \vee r \leq r2.$

Lemma $RmaxLess1 : \forall\ r1\ r2, r1 \leq Rmax\ r1\ r2.$

Lemma $RmaxLess2 : \forall\ r1\ r2, r2 \leq Rmax\ r1\ r2.$

Lemma $Rmax\_comm : \forall\ p\ q{:}R, Rmax\ p\ q = Rmax\ q\ p.$

Notation $RmaxSym := Rmax\_comm\ (only\ parsing).$

Lemma $RmaxRmult :$
$\forall\ (p\ q{:}R)\ r, 0 \leq r \rightarrow Rmax\ (r \times p)\ (r \times q) = r \times Rmax\ p\ q.$

Lemma $Rmax\_stable\_in\_negreal : \forall\ x\ y{:}negreal, Rmax\ x\ y < 0.$

110.3 Rabsolu

Lemma $Rcase\_abs : \forall\ r, \{r < 0\} + \{r \geq 0\}.$

Definition $Rabs\ r : R :=$
match $Rcase\_abs\ r$ with
| $left\ \_ \Rightarrow$ - r
| $right\ \_ \Rightarrow r$
end.

Lemma $Rabs\_R0 : Rabs\ 0 = 0.$

Lemma $Rabs\_R1 : Rabs\ 1 = 1.$

Lemma $Rabs\_no\_R0 : \forall\ r, r \neq 0 \rightarrow Rabs\ r \neq 0.$

Lemma $Rabs\_left : \forall\ r, r < 0 \rightarrow Rabs\ r =$ - $r.$

Lemma $Rabs\_right : \forall\ r, r \geq 0 \rightarrow Rabs\ r = r.$

Lemma $Rabs\_left1 : \forall\ a{:}R, a \leq 0 \rightarrow Rabs\ a =$ - $a.$

Lemma $Rabs\_pos : \forall\ x{:}R, 0 \leq Rabs\ x.$

Lemma $RRle\_abs : \forall\ x{:}R, x \leq Rabs\ x.$

Lemma $Rabs\_pos\_eq : \forall\ x{:}R, 0 \leq x \rightarrow Rabs\ x = x.$

Lemma $Rabs\_Rabsolu : \forall\ x{:}R, Rabs\ (Rabs\ x) = Rabs\ x.$

Lemma $Rabs\_pos\_lt : \forall\ x{:}R, x \neq 0 \rightarrow 0 < Rabs\ x.$

Lemma $Rabs\_minus\_sym : \forall\ x\ y{:}R, Rabs\ (x$ - $y) = Rabs\ (y$ - $x).$

Lemma $Rabs\_mult : \forall\ x\ y{:}R, Rabs\ (x \times y) = Rabs\ x \times Rabs\ y.$

Lemma $Rabs\_Rinv : \forall\ r, r \neq 0 \rightarrow Rabs\ (/\ r) = /\ Rabs\ r.$

Lemma $Rabs\_Ropp : \forall\ x{:}R, Rabs\ (-\ x) = Rabs\ x.$

Lemma $Rabs\_triang : \forall\ a\ b{:}R, Rabs\ (a + b) \leq Rabs\ a + Rabs\ b.$

Lemma $Rabs\_triang\_inv : \forall\ a\ b{:}R, Rabs\ a$ - $Rabs\ b \leq Rabs\ (a$ - $b).$

Lemma $Rabs\_triang\_inv2 : \forall\ a\ b{:}R,\ Rabs\ (Rabs\ a\ -\ Rabs\ b) \leq Rabs\ (a\ -\ b)$.

Lemma $Rabs\_def1 : \forall\ x\ a{:}R,\ x < a \rightarrow -\ a < x \rightarrow Rabs\ x < a$.

Lemma $Rabs\_def2 : \forall\ x\ a{:}R,\ Rabs\ x < a \rightarrow x < a \wedge -\ a < x$.

Lemma $RmaxAbs :$
$\forall\ (p\ q{:}R)\ r,\ p \leq q \rightarrow q \leq r \rightarrow Rabs\ q \leq Rmax\ (Rabs\ p)\ (Rabs\ r)$.

Lemma $Rabs\_Zabs : \forall\ z{:}Z,\ Rabs\ (IZR\ z) = IZR\ (Zabs\ z)$.

Chapter 111

Module Coq.Reals.Rcomplete

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *SeqProp*.
Require Import *Max*.
Open Local *Scope R_scope*.

Theorem *R_complete* :
∀ *Un*:*nat* → *R*, *Cauchy_crit Un* → *sigT* (fun *l*:*R* ⇒ *Un_cv Un l*).

Chapter 112

Module Coq.Reals.Rdefinitions

Definitions for the axiomatization
Require Export *ZArith_base*.

Parameter R : Set.

Delimit Scope R_scope with R.

Parameter $R0$: R.
Parameter $R1$: R.
Parameter $Rplus$: $R \to R \to R$.
Parameter $Rmult$: $R \to R \to R$.
Parameter $Ropp$: $R \to R$.
Parameter $Rinv$: $R \to R$.
Parameter Rlt : $R \to R \to$ Prop.
Parameter up : $R \to Z$.

Infix "$+$" := $Rplus$: R_scope.
Infix "$\times$" := $Rmult$: R_scope.
Notation "- x" := $(Ropp\ x)$: R_scope.
Notation "/ x" := $(Rinv\ x)$: R_scope.

Infix "$<$" := Rlt : R_scope.

Definition $Rgt\ (r1\ r2{:}R)$: Prop := $(r2 < r1)\%R$.

Definition $Rle\ (r1\ r2{:}R)$: Prop := $(r1 < r2)\%R \vee r1 = r2$.

Definition $Rge\ (r1\ r2{:}R)$: Prop := $Rgt\ r1\ r2 \vee r1 = r2$.

Definition $Rminus\ (r1\ r2{:}R)$: R := $(r1 + - r2)\%R$.

Definition $Rdiv\ (r1\ r2{:}R)$: R := $(r1 \times / r2)\%R$.

Infix "-" := $Rminus$: R_scope.
Infix "/" := $Rdiv$: R_scope.

Infix "$\leq$" := Rle : R_scope.
Infix "$\geq$" := Rge : R_scope.
Infix "$>$" := Rgt : R_scope.

Notation "$x \leq y \leq z$" := $((x \leq y)\%R \wedge (y \leq z)\%R) : R\_scope$.
Notation "$x \leq y < z$" := $((x \leq y)\%R \wedge (y < z)\%R) : R\_scope$.
Notation "$x < y < z$" := $((x < y)\%R \wedge (y < z)\%R) : R\_scope$.
Notation "$x < y \leq z$" := $((x < y)\%R \wedge (y \leq z)\%R) : R\_scope$.

Chapter 113

Module Coq.Reals.Rderiv

Definition of the derivative,continuity
Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rlimit*.
Require Import *Fourier*.
Require Import *Classical_Prop*.
Require Import *Classical_Pred_Type*.
Require Import *Omega*. *Open* Local *Scope R_scope*.

Definition *D_x* (*D*:*R* → Prop) (*y x*:*R*) : Prop := *D x* ∧ *y* ≠ *x*.

Definition *continue_in* (*f*:*R* → *R*) (*D*:*R* → Prop) (*x0*:*R*) : Prop :=
limit1_in f (*D_x D x0*) (*f x0*) *x0*.

Definition *D_in* (*f d*:*R* → *R*) (*D*:*R* → Prop) (*x0*:*R*) : Prop :=
limit1_in (fun *x*:*R* ⇒ (*f x* - *f x0*) / (*x* - *x0*)) (*D_x D x0*) (*d x0*) *x0*.

Lemma *cont_deriv* :
∀ (*f d*:*R* → *R*) (*D*:*R* → Prop) (*x0*:*R*),
D_in f d D x0 → *continue_in f D x0*.

Lemma *Dconst* :
∀ (*D*:*R* → Prop) (*y x0*:*R*), *D_in* (fun *x*:*R* ⇒ *y*) (fun *x*:*R* ⇒ 0) *D x0*.

Lemma *Dx* :
∀ (*D*:*R* → Prop) (*x0*:*R*), *D_in* (fun *x*:*R* ⇒ *x*) (fun *x*:*R* ⇒ 1) *D x0*.

Lemma *Dadd* :
∀ (*D*:*R* → Prop) (*df dg f g*:*R* → *R*) (*x0*:*R*),
D_in f df D x0 →
D_in g dg D x0 →
D_in (fun *x*:*R* ⇒ *f x* + *g x*) (fun *x*:*R* ⇒ *df x* + *dg x*) *D x0*.

Lemma *Dmult* :
∀ (*D*:*R* → Prop) (*df dg f g*:*R* → *R*) (*x0*:*R*),
D_in f df D x0 →
D_in g dg D x0 →

$D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow f\ x \times g\ x)\ (\mathsf{fun}\ x{:}R \Rightarrow df\ x \times g\ x + f\ x \times dg\ x)\ D\ x0.$

Lemma $Dmult\_const$:
$\forall\ (D{:}R \to \mathsf{Prop})\ (f\ df{:}R \to R)\ (x0\ a{:}R),$
$D\_in\ f\ df\ D\ x0 \to D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow a \times f\ x)\ (\mathsf{fun}\ x{:}R \Rightarrow a \times df\ x)\ D\ x0.$

Lemma $Dopp$:
$\forall\ (D{:}R \to \mathsf{Prop})\ (f\ df{:}R \to R)\ (x0{:}R),$
$D\_in\ f\ df\ D\ x0 \to D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow \text{-}\ f\ x)\ (\mathsf{fun}\ x{:}R \Rightarrow \text{-}\ df\ x)\ D\ x0.$

Lemma $Dminus$:
$\forall\ (D{:}R \to \mathsf{Prop})\ (df\ dg\ f\ g{:}R \to R)\ (x0{:}R),$
$D\_in\ f\ df\ D\ x0 \to$
$D\_in\ g\ dg\ D\ x0 \to$
$D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow f\ x \text{-}\ g\ x)\ (\mathsf{fun}\ x{:}R \Rightarrow df\ x \text{-}\ dg\ x)\ D\ x0.$

Lemma Dx_pow_n :
$\forall\ (n{:}nat)\ (D{:}R \to \mathsf{Prop})\ (x0{:}R),$
$D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow x\ \hat{}\ n)\ (\mathsf{fun}\ x{:}R \Rightarrow INR\ n \times x\ \hat{}\ (n \text{-}\ 1))\ D\ x0.$

Lemma $Dcomp$:
$\forall\ (Df\ Dg{:}R \to \mathsf{Prop})\ (df\ dg\ f\ g{:}R \to R)\ (x0{:}R),$
$D\_in\ f\ df\ Df\ x0 \to$
$D\_in\ g\ dg\ Dg\ (f\ x0) \to$
$D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow g\ (f\ x))\ (\mathsf{fun}\ x{:}R \Rightarrow df\ x \times dg\ (f\ x))\ (Dgf\ Df\ Dg\ f)\ x0.$

Lemma D_pow_n :
$\forall\ (n{:}nat)\ (D{:}R \to \mathsf{Prop})\ (x0{:}R)\ (expr\ dexpr{:}R \to R),$
$D\_in\ expr\ dexpr\ D\ x0 \to$
$D\_in\ (\mathsf{fun}\ x{:}R \Rightarrow expr\ x\ \hat{}\ n)$
$(\mathsf{fun}\ x{:}R \Rightarrow INR\ n \times expr\ x\ \hat{}\ (n \text{-}\ 1) \times dexpr\ x)\ ($
$Dgf\ D\ D\ expr)\ x0.$

Chapter 114

Module Coq.Reals.Reals

The library REALS is divided in 6 parts :

- Rbase: basic lemmas on R equalities and inequalities Ring and Field are instantiated on R
- Rfunctions: some useful functions (Rabsolu, Rmin, Rmax, fact...)
- SeqSeries: theory of sequences and series
- Rtrigo: theory of trigonometric functions
- Ranalysis: some topology and general results of real analysis (mean value theorem, intermediate value theorem,...)
- Integration: Newton and Riemann' integrals

Tactics are:

- DiscrR: for goals like "?1<>0"
- Sup: for goals like "?1<?2"
- RCompute: for equalities with constants like "10*10==100"
- Reg: for goals like (continuity_pt ?1 ?2) or (derivable_pt ?1 ?2)

Require Export *Rbase*.
Require Export *Rfunctions*.
Require Export *SeqSeries*.
Require Export *Rtrigo*.
Require Export *Ranalysis*.
Require Export *Integration*.

Chapter 115

Module Coq.Reals.Rfunctions

Definition of the sum functions
Require Export *ArithRing*.

Require Import *Rbase*.
Require Export *Rpow_def*.
Require Export *R_Ifp*.
Require Export *Rbasic_fun*.
Require Export *R_sqr*.
Require Export *SplitAbsolu*.
Require Export *SplitRmult*.
Require Export *ArithProp*.
Require Import *Omega*.
Require Import *Zpower*.
Open Local *Scope nat_scope*.
Open Local *Scope R_scope*.

115.1 Lemmas about factorial

Lemma *INR_fact_neq_0* : $\forall$ *n*:*nat*, *INR* (*fact n*) $\neq 0$.

Lemma *fact_simpl* : $\forall$ *n*:*nat*, *fact* (*S n*) = (*S n* $\times$ *fact n*)%*nat*.

Lemma *simpl_fact* :
$\forall$ *n*:*nat*, / *INR* (*fact* (*S n*)) $\times$ / / *INR* (*fact n*) = / *INR* (*S n*).

115.2 Power

Infix "^" := *pow* : *R_scope*.

Lemma *pow_O* : $\forall$ *x*:*R*, *x* ^ 0 = 1.

Lemma *pow_1* : $\forall$ *x*:*R*, *x* ^ 1 = *x*.

Lemma *pow_add* : $\forall$ (*x*:*R*) (*n m*:*nat*), *x* ^ (*n* + *m*) = *x* ^ *n* $\times$ *x* ^ *m*.

Lemma *pow_nonzero* : $\forall\ (x{:}R)\ (n{:}nat),\ x \neq 0 \to x\ \hat{}\ n \neq 0$.

Hint *Resolve pow_O pow_1 pow_add pow_nonzero*: *real*.

Lemma *pow_RN_plus* :
$\forall\ (x{:}R)\ (n\ m{:}nat),\ x \neq 0 \to x\ \hat{}\ n = x\ \hat{}\ (n + m) \times / x\ \hat{}\ m$.

Lemma *pow_lt* : $\forall\ (x{:}R)\ (n{:}nat),\ 0 < x \to 0 < x\ \hat{}\ n$.
Hint *Resolve pow_lt*: *real*.

Lemma *Rlt_pow_R1* : $\forall\ (x{:}R)\ (n{:}nat),\ 1 < x \to (0 < n)\%nat \to 1 < x\ \hat{}\ n$.
Hint *Resolve Rlt_pow_R1*: *real*.

Lemma *Rlt_pow* : $\forall\ (x{:}R)\ (n\ m{:}nat),\ 1 < x \to (n < m)\%nat \to x\ \hat{}\ n < x\ \hat{}\ m$.
Hint *Resolve Rlt_pow*: *real*.

Lemma *tech_pow_Rmult* : $\forall\ (x{:}R)\ (n{:}nat),\ x \times x\ \hat{}\ n = x\ \hat{}\ S\ n$.

Lemma *tech_pow_Rplus* :
$\forall\ (x{:}R)\ (a\ n{:}nat),\ x\ \hat{}\ a + INR\ n \times x\ \hat{}\ a = INR\ (S\ n) \times x\ \hat{}\ a$.

Lemma *poly* : $\forall\ (n{:}nat)\ (x{:}R),\ 0 < x \to 1 + INR\ n \times x \leq (1 + x)\ \hat{}\ n$.

Lemma *Power_monotonic* :
$\forall\ (x{:}R)\ (m\ n{:}nat),$
$Rabs\ x > 1 \to (m \leq n)\%nat \to Rabs\ (x\ \hat{}\ m) \leq Rabs\ (x\ \hat{}\ n)$.

Lemma *RPow_abs* : $\forall\ (x{:}R)\ (n{:}nat),\ Rabs\ x\ \hat{}\ n = Rabs\ (x\ \hat{}\ n)$.

Lemma *Pow_x_infinity* :
$\forall\ x{:}R,$
$Rabs\ x > 1 \to$
$\forall\ b{:}R,$
$\exists\ N : nat,\ (\forall\ n{:}nat,\ (n \geq N)\%nat \to Rabs\ (x\ \hat{}\ n) \geq b)$.

Lemma *pow_ne_zero* : $\forall\ n{:}nat,\ n \neq 0\%nat \to 0\ \hat{}\ n = 0$.

Lemma *Rinv_pow* : $\forall\ (x{:}R)\ (n{:}nat),\ x \neq 0 \to / x\ \hat{}\ n = (/ x)\ \hat{}\ n$.

Lemma *pow_lt_1_zero* :
$\forall\ x{:}R,$
$Rabs\ x < 1 \to$
$\forall\ y{:}R,$
$0 < y \to$
$\exists\ N : nat,\ (\forall\ n{:}nat,\ (n \geq N)\%nat \to Rabs\ (x\ \hat{}\ n) < y)$.

Lemma *pow_R1* : $\forall\ (r{:}R)\ (n{:}nat),\ r\ \hat{}\ n = 1 \to Rabs\ r = 1 \vee n = 0\%nat$.

Lemma *pow_Rsqr* : $\forall\ (x{:}R)\ (n{:}nat),\ x\ \hat{}\ (2 \times n) = Rsqr\ x\ \hat{}\ n$.

Lemma *pow_le* : $\forall\ (a{:}R)\ (n{:}nat),\ 0 \leq a \to 0 \leq a\ \hat{}\ n$.

Lemma *pow_1_even* : $\forall\ n{:}nat,\ (\text{-}1)\ \hat{}\ (2 \times n) = 1$.

Lemma *pow_1_odd* : $\forall\ n{:}nat,\ (\text{-}1)\ \hat{}\ S\ (2 \times n) = \text{-}1$.

Lemma *pow_1_abs* : $\forall\ n{:}nat,\ Rabs\ ((\text{-}1)\ \hat{}\ n) = 1$.

Lemma *pow_mult* : ∀ (*x*:*R*) (*n1 n2*:*nat*), *x* ^ (*n1* × *n2*) = (*x* ^ *n1*) ^ *n2*.

Lemma *pow_incr* : ∀ (*x y*:*R*) (*n*:*nat*), 0 ≤ *x* ≤ *y* → *x* ^ *n* ≤ *y* ^ *n*.

Lemma *pow_R1_Rle* : ∀ (*x*:*R*) (*k*:*nat*), 1 ≤ *x* → 1 ≤ *x* ^ *k*.

Lemma *Rle_pow* :
∀ (*x*:*R*) (*m n*:*nat*), 1 ≤ *x* → (*m* ≤ *n*)%*nat* → *x* ^ *m* ≤ *x* ^ *n*.

Lemma *pow1* : ∀ *n*:*nat*, 1 ^ *n* = 1.

Lemma *pow_Rabs* : ∀ (*x*:*R*) (*n*:*nat*), *x* ^ *n* ≤ *Rabs x* ^ *n*.

Lemma *pow_maj_Rabs* : ∀ (*x y*:*R*) (*n*:*nat*), *Rabs y* ≤ *x* → *y* ^ *n* ≤ *x* ^ *n*.

115.3 PowerRZ

Ltac *case_eq name* :=
generalize (*refl_equal name*); *pattern name at* -1 in ⊢ ×; *case name*.

Definition *powerRZ* (*x*:*R*) (*n*:*Z*) :=
match *n* with
| *Z0* ⇒ 1
| *Zpos p* ⇒ *x* ^ *nat_of_P p*
| *Zneg p* ⇒ / *x* ^ *nat_of_P p*
end.

Infix Local "^*Z*" := *powerRZ* (*at level* 30, *right associativity*) : *R_scope*.

Lemma *Zpower_NR0* :
∀ (*x*:*Z*) (*n*:*nat*), (0 ≤ *x*)%*Z* → (0 ≤ *Zpower_nat x n*)%*Z*.

Lemma *powerRZ_O* : ∀ *x*:*R*, *x* ^*Z* 0 = 1.

Lemma *powerRZ_1* : ∀ *x*:*R*, *x* ^*Z Zsucc* 0 = *x*.

Lemma *powerRZ_NOR* : ∀ (*x*:*R*) (*z*:*Z*), *x* ≠ 0 → *x* ^*Z z* ≠ 0.

Lemma *powerRZ_add* :
∀ (*x*:*R*) (*n m*:*Z*), *x* ≠ 0 → *x* ^*Z* (*n* + *m*) = *x* ^*Z n* × *x* ^*Z m*.
Hint *Resolve powerRZ_O powerRZ_1 powerRZ_NOR powerRZ_add*: *real*.

Lemma *Zpower_nat_powerRZ* :
∀ *n m*:*nat*, *IZR* (*Zpower_nat* (*Z_of_nat n*) *m*) = *INR n* ^*Z Z_of_nat m*.

Lemma *powerRZ_lt* : ∀ (*x*:*R*) (*z*:*Z*), 0 < *x* → 0 < *x* ^*Z z*.
Hint *Resolve powerRZ_lt*: *real*.

Lemma *powerRZ_le* : ∀ (*x*:*R*) (*z*:*Z*), 0 < *x* → 0 ≤ *x* ^*Z z*.
Hint *Resolve powerRZ_le*: *real*.

Lemma *Zpower_nat_powerRZ_absolu* :
∀ *n m*:*Z*, (0 ≤ *m*)%*Z* → *IZR* (*Zpower_nat n* (*Zabs_nat m*)) = *IZR n* ^*Z m*.

Lemma *powerRZ_R1* : ∀ *n*:*Z*, 1 ^*Z n* = 1.

Definition *decimal_exp* (*r*:*R*) (*z*:*Z*) : *R* := (*r* × 10 ^*Z z*).

115.4 Sum of n first naturals

Definition $sum\_nat\_f\ (s\ n{:}nat)\ (f{:}nat \to nat) : nat :=$
$sum\_nat\_f\_O\ (\mathsf{fun}\ x{:}nat \Rightarrow f\ (x + s)\%nat)\ (n - s)$.

Definition $sum\_nat\_O\ (n{:}nat) : nat := sum\_nat\_f\_O\ (\mathsf{fun}\ x{:}nat \Rightarrow x)\ n$.

Definition $sum\_nat\ (s\ n{:}nat) : nat := sum\_nat\_f\ s\ n\ (\mathsf{fun}\ x{:}nat \Rightarrow x)$.

115.5 Sum

Fixpoint $sum\_f\_R0\ (f{:}nat \to R)\ (N{:}nat)\ \{struct\ N\} : R :=$
$\mathsf{match}\ N\ \mathsf{with}$
$\mid O \Rightarrow f\ 0\%nat$
$\mid S\ i \Rightarrow sum\_f\_R0\ f\ i + f\ (S\ i)$
end.

Definition $sum\_f\ (s\ n{:}nat)\ (f{:}nat \to R) : R :=$
$sum\_f\_R0\ (\mathsf{fun}\ x{:}nat \Rightarrow f\ (x + s)\%nat)\ (n - s)$.

Lemma GP_finite :
$\forall\ (x{:}R)\ (n{:}nat)$,
$sum\_f\_R0\ (\mathsf{fun}\ n{:}nat \Rightarrow x\ \hat{}\ n)\ n \times (x - 1) = x\ \hat{}\ (n + 1) - 1$.

Lemma $sum\_f\_R0\_triangle$:
$\forall\ (x{:}nat \to R)\ (n{:}nat)$,
$Rabs\ (sum\_f\_R0\ x\ n) \leq sum\_f\_R0\ (\mathsf{fun}\ i{:}nat \Rightarrow Rabs\ (x\ i))\ n$.

115.6 Distance in R

Definition $R\_dist\ (x\ y{:}R) : R := Rabs\ (x - y)$.

Lemma $R\_dist\_pos : \forall\ x\ y{:}R,\ R\_dist\ x\ y \geq 0$.

Lemma $R\_dist\_sym : \forall\ x\ y{:}R,\ R\_dist\ x\ y = R\_dist\ y\ x$.

Lemma $R\_dist\_refl : \forall\ x\ y{:}R,\ R\_dist\ x\ y = 0 \leftrightarrow x = y$.

Lemma $R\_dist\_eq : \forall\ x{:}R,\ R\_dist\ x\ x = 0$.

Lemma $R\_dist\_tri : \forall\ x\ y\ z{:}R,\ R\_dist\ x\ y \leq R\_dist\ x\ z + R\_dist\ z\ y$.

Lemma R_dist_plus :
$\forall\ a\ b\ c\ d{:}R,\ R\_dist\ (a + c)\ (b + d) \leq R\_dist\ a\ b + R\_dist\ c\ d$.

115.7 Infinit Sum

Definition $infinit\_sum\ (s{:}nat \to R)\ (l{:}R) : \mathsf{Prop} :=$
$\forall\ eps{:}R$,

$eps > 0 \rightarrow$
$\exists\ N : nat,$
$(\forall\ n{:}nat,\ (n \geq N)\%nat \rightarrow R\_dist\ (sum\_f\_R0\ \ s\ n)\ l < eps).$

Chapter 116

Module Coq.Reals.Rgeom

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo*.
Require Import *R_sqrt*. *Open* Local *Scope R_scope*.

116.1 Distance

Definition *dist_euc* (*x0 y0 x1 y1*:*R*) : *R* :=
  sqrt (*Rsqr* (*x0* - *x1*) + *Rsqr* (*y0* - *y1*)).

Lemma *distance_refl* : ∀ *x0 y0*:*R*, *dist_euc x0 y0 x0 y0* = 0.

Lemma *distance_symm* :
  ∀ *x0 y0 x1 y1*:*R*, *dist_euc x0 y0 x1 y1* = *dist_euc x1 y1 x0 y0*.

Lemma *law_cosines* :
  ∀ *x0 y0 x1 y1 x2 y2 ac*:*R*,
    let *a* := *dist_euc x1 y1 x0 y0* in
      let *b* := *dist_euc x2 y2 x0 y0* in
        let *c* := *dist_euc x2 y2 x1 y1* in
          a × *c* × *cos ac* = (*x0* - *x1*) × (*x2* - *x1*) + (*y0* - *y1*) × (*y2* - *y1*) →
          Rsqr b = *Rsqr c* + *Rsqr a* - 2 × (*a* × *c* × *cos ac*).

Lemma *triangle* :
  ∀ *x0 y0 x1 y1 x2 y2*:*R*,
    dist_euc x0 y0 x1 y1 ≤ *dist_euc x0 y0 x2 y2* + *dist_euc x2 y2 x1 y1*.

116.2 Translation

Definition *xt* (*x tx*:*R*) : *R* := *x* + *tx*.
Definition *yt* (*y ty*:*R*) : *R* := *y* + *ty*.

Lemma *translation_0* : ∀ *x y*:*R*, *xt x* 0 = *x* ∧ *yt y* 0 = *y*.

Lemma *isometric_translation* :
∀ *x1 x2 y1 y2 tx ty*:*R*,
Rsqr (*x1* - *x2*) + *Rsqr* (*y1* - *y2*) =
Rsqr (*xt x1 tx* - *xt x2 tx*) + *Rsqr* (*yt y1 ty* - *yt y2 ty*).

116.3 Rotation

Definition *xr* (*x y theta*:*R*) : *R* := *x* × *cos theta* + *y* × *sin theta*.
Definition *yr* (*x y theta*:*R*) : *R* := - *x* × *sin theta* + *y* × *cos theta*.

Lemma *rotation_0* : ∀ *x y*:*R*, *xr x y* 0 = *x* ∧ *yr x y* 0 = *y*.

Lemma *rotation_PI2* :
∀ *x y*:*R*, *xr x y* (*PI* / 2) = *y* ∧ *yr x y* (*PI* / 2) = - *x*.

Lemma *isometric_rotation_0* :
∀ *x1 y1 x2 y2 theta*:*R*,
Rsqr (*x1* - *x2*) + *Rsqr* (*y1* - *y2*) =
Rsqr (*xr x1 y1 theta* - *xr x2 y2 theta*) +
Rsqr (*yr x1 y1 theta* - *yr x2 y2 theta*).

Lemma *isometric_rotation* :
∀ *x1 y1 x2 y2 theta*:*R*,
dist_euc x1 y1 x2 y2 =
dist_euc (*xr x1 y1 theta*) (*yr x1 y1 theta*) (*xr x2 y2 theta*)
(*yr x2 y2 theta*).

116.4 Similarity

Lemma *isometric_rot_trans* :
∀ *x1 y1 x2 y2 tx ty theta*:*R*,
Rsqr (*x1* - *x2*) + *Rsqr* (*y1* - *y2*) =
Rsqr (*xr* (*xt x1 tx*) (*yt y1 ty*) *theta* - *xr* (*xt x2 tx*) (*yt y2 ty*) *theta*) +
Rsqr (*yr* (*xt x1 tx*) (*yt y1 ty*) *theta* - *yr* (*xt x2 tx*) (*yt y2 ty*) *theta*).

Lemma *isometric_trans_rot* :
∀ *x1 y1 x2 y2 tx ty theta*:*R*,
Rsqr (*x1* - *x2*) + *Rsqr* (*y1* - *y2*) =
Rsqr (*xt* (*xr x1 y1 theta*) *tx* - *xt* (*xr x2 y2 theta*) *tx*) +
Rsqr (*yt* (*yr x1 y1 theta*) *ty* - *yt* (*yr x2 y2 theta*) *ty*).

Chapter 117

Module Coq.Reals.RiemannInt_SF

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Ranalysis*.
Require Import *Classical_Prop*.
Open Local *Scope R_scope*.

117.1 Each bounded subset of N has a maximal element

Definition *Nbound* (*I*:*nat* → Prop) : Prop :=
  ∃ *n* : *nat*, (∀ *i*:*nat*, *I i* → (*i* ≤ *n*)%*nat*).

Lemma *IZN_var* : ∀ *z*:*Z*, (0 ≤ *z*)%*Z* → {*n* : *nat* | *z* = *Z_of_nat n*}.

Lemma *Nzorn* :
  ∀ *I*:*nat* → Prop,
    (∃ *n* : *nat*, *I n*) →
    Nbound I → *sigT* (fun *n*:*nat* ⇒ *I n* ∧ (∀ *i*:*nat*, *I i* → (*i* ≤ *n*)%*nat*)).

117.2 Step functions

Definition *open_interval* (*a b x*:*R*) : Prop := *a* < *x* < *b*.
Definition *co_interval* (*a b x*:*R*) : Prop := *a* ≤ *x* < *b*.

Definition *adapted_couple* (*f*:*R* → *R*) (*a b*:*R*) (*l lf*:*Rlist*) : Prop :=
  ordered_Rlist l ∧
  pos_Rl l 0 = *Rmin a b* ∧
  pos_Rl l (*pred* (*Rlength l*)) = *Rmax a b* ∧
  Rlength l = *S* (*Rlength lf*) ∧
  (∀ *i*:*nat*,
    (*i* < *pred* (*Rlength l*))%*nat* →
    constant_D_eq f (*open_interval* (*pos_Rl l i*) (*pos_Rl l* (*S i*)))
    (*pos_Rl lf i*)).

```
Definition adapted_couple_opt (f:R → R) (a b:R) (l lf:Rlist) :=
  adapted_couple f a b l lf ∧
  (∀ i:nat,
    (i < pred (Rlength lf))%nat →
    pos_Rl lf i ≠ pos_Rl lf (S i) ∨ f (pos_Rl l (S i)) ≠ pos_Rl lf i) ∧
  (∀ i:nat, (i < pred (Rlength l))%nat → pos_Rl l i ≠ pos_Rl l (S i)).

Definition is_subdivision (f:R → R) (a b:R) (l:Rlist) : Type :=
  sigT (fun l0:Rlist ⇒ adapted_couple f a b l l0).

Definition IsStepFun (f:R → R) (a b:R) : Type :=
  sigT (fun l:Rlist ⇒ is_subdivision f a b l).
```

117.2.1 Class of step functions

```
Record StepFun (a b:R) : Type := mkStepFun
  {fe :> R → R; pre : IsStepFun fe a b}.

Definition subdivision (a b:R) (f:StepFun a b) : Rlist := projT1 (pre f).

Definition subdivision_val (a b:R) (f:StepFun a b) : Rlist :=
  match projT2 (pre f) with
    | existT a b ⇒ a
  end.
```

117.2.2 Integral of step functions

```
Definition RiemannInt_SF (a b:R) (f:StepFun a b) : R :=
  match Rle_dec a b with
    | left _ ⇒ Int_SF (subdivision_val f) (subdivision f)
    | right _ ⇒ - Int_SF (subdivision_val f) (subdivision f)
  end.
```

117.2.3 Properties of step functions

```
Lemma StepFun_P1 :
  ∀ (a b:R) (f:StepFun a b),
    adapted_couple f a b (subdivision f) (subdivision_val f).

Lemma StepFun_P2 :
  ∀ (a b:R) (f:R → R) (l lf:Rlist),
    adapted_couple f a b l lf → adapted_couple f b a l lf.

Lemma StepFun_P3 :
  ∀ a b c:R,
    a ≤ b →
    adapted_couple (fct_cte c) a b (cons a (cons b nil)) (cons c nil).

Lemma StepFun_P4 : ∀ a b c:R, IsStepFun (fct_cte c) a b.
```

Lemma *StepFun_P5* :
$\forall\ (a\ b{:}R)\ (f{:}R \to R)\ (l{:}Rlist),$
$is\_subdivision\ f\ a\ b\ l \to is\_subdivision\ f\ b\ a\ l.$

Lemma *StepFun_P6* :
$\forall\ (f{:}R \to R)\ (a\ b{:}R),\ IsStepFun\ f\ a\ b \to IsStepFun\ f\ b\ a.$

Lemma *StepFun_P7* :
$\forall\ (a\ b\ r1\ r2\ r3{:}R)\ (f{:}R \to R)\ (l\ lf{:}Rlist),$
$a \leq b \to$
$adapted\_couple\ f\ a\ b\ (cons\ r1\ (cons\ r2\ l))\ (cons\ r3\ lf) \to$
$adapted\_couple\ f\ r2\ b\ (cons\ r2\ l)\ lf.$

Lemma *StepFun_P8* :
$\forall\ (f{:}R \to R)\ (l1\ lf1{:}Rlist)\ (a\ b{:}R),$
$adapted\_couple\ f\ a\ b\ l1\ lf1 \to a = b \to Int\_SF\ lf1\ l1 = 0.$

Lemma *StepFun_P9* :
$\forall\ (a\ b{:}R)\ (f{:}R \to R)\ (l\ lf{:}Rlist),$
$adapted\_couple\ f\ a\ b\ l\ lf \to a \neq b \to (2 \leq Rlength\ l)\%nat.$

Lemma *StepFun_P10* :
$\forall\ (f{:}R \to R)\ (l\ lf{:}Rlist)\ (a\ b{:}R),$
$a \leq b \to$
$adapted\_couple\ f\ a\ b\ l\ lf \to$
$\exists\ l' : Rlist,$
$(\exists\ lf' : Rlist,\ adapted\_couple\_opt\ f\ a\ b\ l'\ lf').$

Lemma *StepFun_P11* :
$\forall\ (a\ b\ r\ r1\ r3\ s1\ s2\ r4{:}R)\ (r2\ lf1\ s3\ lf2{:}Rlist)$
$(f{:}R \to R),$
$a < b \to$
$adapted\_couple\ f\ a\ b\ (cons\ r\ (cons\ r1\ r2))\ (cons\ r3\ lf1) \to$
$adapted\_couple\_opt\ f\ a\ b\ (cons\ s1\ (cons\ s2\ s3))\ (cons\ r4\ lf2) \to r1 \leq s2.$

Lemma *StepFun_P12* :
$\forall\ (a\ b{:}R)\ (f{:}R \to R)\ (l\ lf{:}Rlist),$
$adapted\_couple\_opt\ f\ a\ b\ l\ lf \to adapted\_couple\_opt\ f\ b\ a\ l\ lf.$

Lemma *StepFun_P13* :
$\forall\ (a\ b\ r\ r1\ r3\ s1\ s2\ r4{:}R)\ (r2\ lf1\ s3\ lf2{:}Rlist)$
$(f{:}R \to R),$
$a \neq b \to$
$adapted\_couple\ f\ a\ b\ (cons\ r\ (cons\ r1\ r2))\ (cons\ r3\ lf1) \to$
$adapted\_couple\_opt\ f\ a\ b\ (cons\ s1\ (cons\ s2\ s3))\ (cons\ r4\ lf2) \to r1 \leq s2.$

Lemma *StepFun_P14* :
$\forall\ (f{:}R \to R)\ (l1\ l2\ lf1\ lf2{:}Rlist)\ (a\ b{:}R),$
$a \leq b \to$
$adapted\_couple\ f\ a\ b\ l1\ lf1 \to$
$adapted\_couple\_opt\ f\ a\ b\ l2\ lf2 \to Int\_SF\ lf1\ l1 = Int\_SF\ lf2\ l2.$

Lemma *StepFun_P15* :
∀ (*f*:*R* → *R*) (*l1 l2 lf1 lf2*:*Rlist*) (*a b*:*R*),
adapted_couple f a b l1 lf1 →
adapted_couple_opt f a b l2 lf2 → *Int_SF lf1 l1* = *Int_SF lf2 l2*.

Lemma *StepFun_P16* :
∀ (*f*:*R* → *R*) (*l lf*:*Rlist*) (*a b*:*R*),
adapted_couple f a b l lf →
∃ *l'* : *Rlist*,
(∃ *lf'* : *Rlist*, *adapted_couple_opt f a b l' lf'*).

Lemma *StepFun_P17* :
∀ (*f*:*R* → *R*) (*l1 l2 lf1 lf2*:*Rlist*) (*a b*:*R*),
adapted_couple f a b l1 lf1 →
adapted_couple f a b l2 lf2 → *Int_SF lf1 l1* = *Int_SF lf2 l2*.

Lemma *StepFun_P18* :
∀ *a b c*:*R*, *RiemannInt_SF* (*mkStepFun* (*StepFun_P4 a b c*)) = *c* × (*b* - *a*).

Lemma *StepFun_P19* :
∀ (*l1*:*Rlist*) (*f g*:*R* → *R*) (*l*:*R*),
Int_SF (*FF l1* (fun *x*:*R* ⇒ *f x* + *l* × *g x*)) *l1* =
Int_SF (*FF l1 f*) *l1* + *l* × *Int_SF* (*FF l1 g*) *l1*.

Lemma *StepFun_P20* :
∀ (*l*:*Rlist*) (*f*:*R* → *R*),
(0 < *Rlength l*)%*nat* → *Rlength l* = *S* (*Rlength* (*FF l f*)).

Lemma *StepFun_P21* :
∀ (*a b*:*R*) (*f*:*R* → *R*) (*l*:*Rlist*),
is_subdivision f a b l → *adapted_couple f a b l* (*FF l f*).

Lemma *StepFun_P22* :
∀ (*a b*:*R*) (*f g*:*R* → *R*) (*lf lg*:*Rlist*),
a ≤ *b* →
is_subdivision f a b lf →
is_subdivision g a b lg → *is_subdivision f a b* (*cons_ORlist lf lg*).

Lemma *StepFun_P23* :
∀ (*a b*:*R*) (*f g*:*R* → *R*) (*lf lg*:*Rlist*),
is_subdivision f a b lf →
is_subdivision g a b lg → *is_subdivision f a b* (*cons_ORlist lf lg*).

Lemma *StepFun_P24* :
∀ (*a b*:*R*) (*f g*:*R* → *R*) (*lf lg*:*Rlist*),
a ≤ *b* →
is_subdivision f a b lf →
is_subdivision g a b lg → *is_subdivision g a b* (*cons_ORlist lf lg*).

Lemma *StepFun_P25* :
∀ (*a b*:*R*) (*f g*:*R* → *R*) (*lf lg*:*Rlist*),

is_subdivision f a b lf →
is_subdivision g a b lg → is_subdivision g a b (cons_ORlist lf lg).

Lemma *StepFun_P26* :
∀ (*a b l*:*R*) (*f g*:*R* → *R*) (*l1*:*Rlist*),
is_subdivision f a b l1 →
is_subdivision g a b l1 →
is_subdivision (fun *x*:*R* ⇒ *f x* + *l* × *g x*) *a b l1.*

Lemma *StepFun_P27* :
∀ (*a b l*:*R*) (*f g*:*R* → *R*) (*lf lg*:*Rlist*),
is_subdivision f a b lf →
is_subdivision g a b lg →
is_subdivision (fun *x*:*R* ⇒ *f x* + *l* × *g x*) *a b* (*cons_ORlist lf lg*).

The set of step functions on *a*,*b* is a vectorial space
Lemma *StepFun_P28* :
∀ (*a b l*:*R*) (*f g*:*StepFun a b*), *IsStepFun* (fun *x*:*R* ⇒ *f x* + *l* × *g x*) *a b.*

Lemma *StepFun_P29* :
∀ (*a b*:*R*) (*f*:*StepFun a b*), *is_subdivision f a b* (*subdivision f*).

Lemma *StepFun_P30* :
∀ (*a b l*:*R*) (*f g*:*StepFun a b*),
RiemannInt_SF (*mkStepFun* (*StepFun_P28 l f g*)) =
RiemannInt_SF f + *l* × *RiemannInt_SF g.*

Lemma *StepFun_P31* :
∀ (*a b*:*R*) (*f*:*R* → *R*) (*l lf*:*Rlist*),
adapted_couple f a b l lf →
adapted_couple (fun *x*:*R* ⇒ *Rabs* (*f x*)) *a b l* (*app_Rlist lf Rabs*).

Lemma *StepFun_P32* :
∀ (*a b*:*R*) (*f*:*StepFun a b*), *IsStepFun* (fun *x*:*R* ⇒ *Rabs* (*f x*)) *a b.*

Lemma *StepFun_P33* :
∀ *l2 l1*:*Rlist*,
ordered_Rlist l1 → *Rabs* (*Int_SF l2 l1*) ≤ *Int_SF* (*app_Rlist l2 Rabs*) *l1.*

Lemma *StepFun_P34* :
∀ (*a b*:*R*) (*f*:*StepFun a b*),
a ≤ *b →*
Rabs (*RiemannInt_SF f*) ≤ *RiemannInt_SF* (*mkStepFun* (*StepFun_P32 f*)).

Lemma *StepFun_P35* :
∀ (*l*:*Rlist*) (*a b*:*R*) (*f g*:*R* → *R*),
ordered_Rlist l →
pos_Rl l 0 = *a →*
pos_Rl l (*pred* (*Rlength l*)) = *b →*
(∀ *x*:*R*, *a* < *x* < *b* → *f x* ≤ *g x*) →
Int_SF (*FF l f*) *l* ≤ *Int_SF* (*FF l g*) *l.*

Lemma *StepFun_P36* :
∀ (*a b*:*R*) (*f g*:*StepFun a b*) (*l*:*Rlist*),
a ≤ *b* →
is_subdivision f a b l →
is_subdivision g a b l →
(∀ *x*:*R*, *a* < *x* < *b* → *f x* ≤ *g x*) →
RiemannInt_SF f ≤ *RiemannInt_SF g*.

Lemma *StepFun_P37* :
∀ (*a b*:*R*) (*f g*:*StepFun a b*),
a ≤ *b* →
(∀ *x*:*R*, *a* < *x* < *b* → *f x* ≤ *g x*) →
RiemannInt_SF f ≤ *RiemannInt_SF g*.

Lemma *StepFun_P38* :
∀ (*l*:*Rlist*) (*a b*:*R*) (*f*:*R* → *R*),
ordered_Rlist l →
pos_Rl l 0 = *a* →
pos_Rl l (*pred* (*Rlength l*)) = *b* →
sigT
(fun *g*:*StepFun a b* ⇒
g b = *f b* ∧
(∀ *i*:*nat*,
(*i* < *pred* (*Rlength l*))%*nat* →
constant_D_eq g (*co_interval* (*pos_Rl l i*) (*pos_Rl l* (*S i*)))
(*f* (*pos_Rl l i*)))).

Lemma *StepFun_P39* :
∀ (*a b*:*R*) (*f*:*StepFun a b*),
RiemannInt_SF f = - *RiemannInt_SF* (*mkStepFun* (*StepFun_P6* (*pre f*))).

Lemma *StepFun_P40* :
∀ (*f*:*R* → *R*) (*a b c*:*R*) (*l1 l2 lf1 lf2*:*Rlist*),
a < *b* →
b < *c* →
adapted_couple f a b l1 lf1 →
adapted_couple f b c l2 lf2 →
adapted_couple f a c (*cons_Rlist l1 l2*) (*FF* (*cons_Rlist l1 l2*) *f*).

Lemma *StepFun_P41* :
∀ (*f*:*R* → *R*) (*a b c*:*R*),
a ≤ *b* → *b* ≤ *c* → *IsStepFun f a b* → *IsStepFun f b c* → *IsStepFun f a c*.

Lemma *StepFun_P42* :
∀ (*l1 l2*:*Rlist*) (*f*:*R* → *R*),
pos_Rl l1 (*pred* (*Rlength l1*)) = *pos_Rl l2* 0 →
Int_SF (*FF* (*cons_Rlist l1 l2*) *f*) (*cons_Rlist l1 l2*) =
Int_SF (*FF l1 f*) *l1* + *Int_SF* (*FF l2 f*) *l2*.

Lemma *StepFun_P43* :

$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R)\ (pr1{:}IsStepFun\ f\ a\ b)$
$(pr2{:}IsStepFun\ f\ b\ c)\ (pr3{:}IsStepFun\ f\ a\ c),$
$RiemannInt\_SF\ (mkStepFun\ pr1) + RiemannInt\_SF\ (mkStepFun\ pr2) =$
$RiemannInt\_SF\ (mkStepFun\ pr3).$

Lemma *StepFun_P44* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R),$
$IsStepFun\ f\ a\ b \to a \leq c \leq b \to IsStepFun\ f\ a\ c.$

Lemma *StepFun_P45* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R),$
$IsStepFun\ f\ a\ b \to a \leq c \leq b \to IsStepFun\ f\ c\ b.$

Lemma *StepFun_P46* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R),$
$IsStepFun\ f\ a\ b \to IsStepFun\ f\ b\ c \to IsStepFun\ f\ a\ c.$

Chapter 118

Module Coq.Reals.RiemannInt

```
Require Import Rfunctions.
Require Import SeqSeries.
Require Import Ranalysis.
Require Import Rbase.
Require Import RiemannInt_SF.
Require Import Classical_Prop.
Require Import Classical_Pred_Type.
Require Import Max. Open Local Scope R_scope.
```

Riemann's Integral

```
Definition Riemann_integrable (f:R → R) (a b:R) : Type :=
  ∀ eps:posreal,
    sigT
    (fun phi:StepFun a b ⇒
      sigT
      (fun psi:StepFun a b ⇒
        (∀ t:R,
          Rmin a b ≤ t ≤ Rmax a b → Rabs (f t - phi t) ≤ psi t) ∧
        Rabs (RiemannInt_SF psi) < eps)).

Definition phi_sequence (un:nat → posreal) (f:R → R)
  (a b:R) (pr:Riemann_integrable f a b) (n:nat) :=
  projT1 (pr (un n)).

Lemma phi_sequence_prop :
  ∀ (un:nat → posreal) (f:R → R) (a b:R) (pr:Riemann_integrable f a b)
    (N:nat),
    sigT
    (fun psi:StepFun a b ⇒
      (∀ t:R,
        Rmin a b ≤ t ≤ Rmax a b →
        Rabs (f t - phi_sequence un pr N t) ≤ psi t) ∧
      Rabs (RiemannInt_SF psi) < un N).

Lemma RiemannInt_P1 :
```

∀ (*f*:*R* → *R*) (*a b*:*R*),
Riemann_integrable f a b → *Riemann_integrable f b a*.

Lemma *RiemannInt_P2* :
∀ (*f*:*R* → *R*) (*a b*:*R*) (*un*:*nat* → *posreal*) (*vn wn*:*nat* → *StepFun a b*),
Un_cv un 0 →
a ≤ *b* →
(∀ *n*:*nat*,
(∀ *t*:*R*, *Rmin a b* ≤ *t* ≤ *Rmax a b* → *Rabs* (*f t* - *vn n t*) ≤ *wn n t*) ∧
Rabs (*RiemannInt_SF* (*wn n*)) < *un n*) →
sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *RiemannInt_SF* (*vn N*)) *l*).

Lemma *RiemannInt_P3* :
∀ (*f*:*R* → *R*) (*a b*:*R*) (*un*:*nat* → *posreal*) (*vn wn*:*nat* → *StepFun a b*),
Un_cv un 0 →
(∀ *n*:*nat*,
(∀ *t*:*R*, *Rmin a b* ≤ *t* ≤ *Rmax a b* → *Rabs* (*f t* - *vn n t*) ≤ *wn n t*) ∧
Rabs (*RiemannInt_SF* (*wn n*)) < *un n*) →
sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *RiemannInt_SF* (*vn N*)) *l*).

Lemma *RiemannInt_exists* :
∀ (*f*:*R* → *R*) (*a b*:*R*) (*pr*:*Riemann_integrable f a b*)
(*un*:*nat* → *posreal*),
Un_cv un 0 →
sigT
(fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *RiemannInt_SF* (*phi_sequence un pr N*)) *l*).

Lemma *RiemannInt_P4* :
∀ (*f*:*R* → *R*) (*a b l*:*R*) (*pr1 pr2*:*Riemann_integrable f a b*)
(*un vn*:*nat* → *posreal*),
Un_cv un 0 →
Un_cv vn 0 →
Un_cv (fun *N*:*nat* ⇒ *RiemannInt_SF* (*phi_sequence un pr1 N*)) *l* →
Un_cv (fun *N*:*nat* ⇒ *RiemannInt_SF* (*phi_sequence vn pr2 N*)) *l*.

Lemma *RinvN_pos* : ∀ *n*:*nat*, 0 < / (*INR n* + 1).

Definition *RinvN* (*N*:*nat*) : *posreal* := *mkposreal* _ (*RinvN_pos N*).

Lemma *RinvN_cv* : *Un_cv RinvN* 0.

Definition *RiemannInt* (*f*:*R* → *R*) (*a b*:*R*) (*pr*:*Riemann_integrable f a b*) : *R* :=
match *RiemannInt_exists pr RinvN RinvN_cv* with
| *existT a' b'* ⇒ *a'*
end.

Lemma *RiemannInt_P5* :
∀ (*f*:*R* → *R*) (*a b*:*R*) (*pr1 pr2*:*Riemann_integrable f a b*),
RiemannInt pr1 = *RiemannInt pr2*.

C°(*a*,*b*) is included in L1(*a*,*b*)

```
Lemma maxN :
  ∀ (a b:R) (del:posreal),
    a < b →
    sigT (fun n:nat ⇒ a + INR n × del < b ∧ b ≤ a + INR (S n) × del).

Fixpoint SubEquiN (N:nat) (x y:R) (del:posreal) {struct N} : Rlist :=
  match N with
    | O ⇒ cons y nil
    | S p ⇒ cons x (SubEquiN p (x + del) y del)
  end.

Definition max_N (a b:R) (del:posreal) (h:a < b) : nat :=
  match maxN del h with
    | existT N H0 ⇒ N
  end.

Definition SubEqui (a b:R) (del:posreal) (h:a < b) : Rlist :=
  SubEquiN (S (max_N del h)) a b del.

Lemma Heine_cor1 :
  ∀ (f:R → R) (a b:R),
    a < b →
    (∀ x:R, a ≤ x ≤ b → continuity_pt f x) →
    ∀ eps:posreal,
      sigT
      (fun delta:posreal ⇒
        delta ≤ b - a ∧
        (∀ x y:R,
          a ≤ x ≤ b →
          a ≤ y ≤ b → Rabs (x - y) < delta → Rabs (f x - f y) < eps)).

Lemma Heine_cor2 :
  ∀ (f:R → R) (a b:R),
    (∀ x:R, a ≤ x ≤ b → continuity_pt f x) →
    ∀ eps:posreal,
      sigT
      (fun delta:posreal ⇒
        ∀ x y:R,
          a ≤ x ≤ b →
          a ≤ y ≤ b → Rabs (x - y) < delta → Rabs (f x - f y) < eps).

Lemma SubEqui_P1 :
  ∀ (a b:R) (del:posreal) (h:a < b), pos_Rl (SubEqui del h) 0 = a.

Lemma SubEqui_P2 :
  ∀ (a b:R) (del:posreal) (h:a < b),
    pos_Rl (SubEqui del h) (pred (Rlength (SubEqui del h))) = b.

Lemma SubEqui_P3 :
  ∀ (N:nat) (a b:R) (del:posreal), Rlength (SubEquiN N a b del) = S N.
```

Lemma *SubEqui_P4* :
∀ (*N*:*nat*) (*a b*:*R*) (*del*:*posreal*) (*i*:*nat*),
(*i* < *S N*)%*nat* → *pos_Rl* (*SubEquiN* (*S N*) *a b del*) *i* = *a* + *INR i* × *del*.

Lemma *SubEqui_P5* :
∀ (*a b*:*R*) (*del*:*posreal*) (*h*:*a* < *b*),
Rlength (*SubEqui del h*) = *S* (*S* (*max_N del h*)).

Lemma *SubEqui_P6* :
∀ (*a b*:*R*) (*del*:*posreal*) (*h*:*a* < *b*) (*i*:*nat*),
(*i* < *S* (*max_N del h*))%*nat* → *pos_Rl* (*SubEqui del h*) *i* = *a* + *INR i* × *del*.

Lemma *SubEqui_P7* :
∀ (*a b*:*R*) (*del*:*posreal*) (*h*:*a* < *b*), *ordered_Rlist* (*SubEqui del h*).

Lemma *SubEqui_P8* :
∀ (*a b*:*R*) (*del*:*posreal*) (*h*:*a* < *b*) (*i*:*nat*),
(*i* < *Rlength* (*SubEqui del h*))%*nat* → *a* ≤ *pos_Rl* (*SubEqui del h*) *i* ≤ *b*.

Lemma *SubEqui_P9* :
∀ (*a b*:*R*) (*del*:*posreal*) (*f*:*R* → *R*) (*h*:*a* < *b*),
sigT
(fun *g*:*StepFun a b* ⇒
g b = *f b* ∧
(∀ *i*:*nat*,
(*i* < *pred* (*Rlength* (*SubEqui del h*)))%*nat* →
constant_D_eq g
(*co_interval* (*pos_Rl* (*SubEqui del h*) *i*)
(*pos_Rl* (*SubEqui del h*) (*S i*)))
(*f* (*pos_Rl* (*SubEqui del h*) *i*)))).

Lemma *RiemannInt_P6* :
∀ (*f*:*R* → *R*) (*a b*:*R*),
a < *b* →
(∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*) → *Riemann_integrable f a b*.

Lemma *RiemannInt_P7* : ∀ (*f*:*R* → *R*) (*a*:*R*), *Riemann_integrable f a a*.

Lemma *continuity_implies_RiemannInt* :
∀ (*f*:*R* → *R*) (*a b*:*R*),
a ≤ *b* →
(∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*) → *Riemann_integrable f a b*.

Lemma *RiemannInt_P8* :
∀ (*f*:*R* → *R*) (*a b*:*R*) (*pr1*:*Riemann_integrable f a b*)
(*pr2*:*Riemann_integrable f b a*), *RiemannInt pr1* = - *RiemannInt pr2*.

Lemma *RiemannInt_P9* :
∀ (*f*:*R* → *R*) (*a*:*R*) (*pr*:*Riemann_integrable f a a*), *RiemannInt pr* = 0.

Lemma *Req_EM_T* : ∀ *r1 r2*:*R*, {*r1* = *r2*} + {*r1* ≠ *r2*}.

Lemma *RiemannInt_P10* :

$\forall\ (f\ g{:}R \to R)\ (a\ b\ l{:}R),$
$Riemann\_integrable\ f\ a\ b \to$
$Riemann\_integrable\ g\ a\ b \to$
$Riemann\_integrable\ (\mathsf{fun}\ x{:}R \Rightarrow f\ x + l \times g\ x)\ a\ b.$

Lemma *RiemannInt_P11* :
$\forall\ (f{:}R \to R)\ (a\ b\ l{:}R)\ (un{:}nat \to posreal)$
$(phi1\ phi2\ psi1\ psi2{:}nat \to StepFun\ a\ b),$
$Un\_cv\ un\ 0 \to$
$(\forall\ n{:}nat,$
$(\forall\ t{:}R,$
$Rmin\ a\ b \le t \le Rmax\ a\ b \to Rabs\ (f\ t - phi1\ n\ t) \le psi1\ n\ t) \wedge$
$Rabs\ (RiemannInt\_SF\ (psi1\ n)) < un\ n) \to$
$(\forall\ n{:}nat,$
$(\forall\ t{:}R,$
$Rmin\ a\ b \le t \le Rmax\ a\ b \to Rabs\ (f\ t - phi2\ n\ t) \le psi2\ n\ t) \wedge$
$Rabs\ (RiemannInt\_SF\ (psi2\ n)) < un\ n) \to$
$Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow RiemannInt\_SF\ (phi1\ N))\ l \to$
$Un\_cv\ (\mathsf{fun}\ N{:}nat \Rightarrow RiemannInt\_SF\ (phi2\ N))\ l.$

Lemma *RiemannInt_P12* :
$\forall\ (f\ g{:}R \to R)\ (a\ b\ l{:}R)\ (pr1{:}Riemann\_integrable\ f\ a\ b)$
$(pr2{:}Riemann\_integrable\ g\ a\ b)$
$(pr3{:}Riemann\_integrable\ (\mathsf{fun}\ x{:}R \Rightarrow f\ x + l \times g\ x)\ a\ b),$
$a \le b \to RiemannInt\ pr3 = RiemannInt\ pr1 + l \times RiemannInt\ pr2.$

Lemma *RiemannInt_P13* :
$\forall\ (f\ g{:}R \to R)\ (a\ b\ l{:}R)\ (pr1{:}Riemann\_integrable\ f\ a\ b)$
$(pr2{:}Riemann\_integrable\ g\ a\ b)$
$(pr3{:}Riemann\_integrable\ (\mathsf{fun}\ x{:}R \Rightarrow f\ x + l \times g\ x)\ a\ b),$
$RiemannInt\ pr3 = RiemannInt\ pr1 + l \times RiemannInt\ pr2.$

Lemma *RiemannInt_P14* : $\forall\ a\ b\ c{:}R,\ Riemann\_integrable\ (fct\_cte\ c)\ a\ b.$

Lemma *RiemannInt_P15* :
$\forall\ (a\ b\ c{:}R)\ (pr{:}Riemann\_integrable\ (fct\_cte\ c)\ a\ b),$
$RiemannInt\ pr = c \times (b - a).$

Lemma *RiemannInt_P16* :
$\forall\ (f{:}R \to R)\ (a\ b{:}R),$
$Riemann\_integrable\ f\ a\ b \to Riemann\_integrable\ (\mathsf{fun}\ x{:}R \Rightarrow Rabs\ (f\ x))\ a\ b.$

Lemma *Rle_cv_lim* :
$\forall\ (Un\ Vn{:}nat \to R)\ (l1\ l2{:}R),$
$(\forall\ n{:}nat,\ Un\ n \le Vn\ n) \to Un\_cv\ Un\ l1 \to Un\_cv\ Vn\ l2 \to l1 \le l2.$

Lemma *RiemannInt_P17* :
$\forall\ (f{:}R \to R)\ (a\ b{:}R)\ (pr1{:}Riemann\_integrable\ f\ a\ b)$
$(pr2{:}Riemann\_integrable\ (\mathsf{fun}\ x{:}R \Rightarrow Rabs\ (f\ x))\ a\ b),$
$a \le b \to Rabs\ (RiemannInt\ pr1) \le RiemannInt\ pr2.$

Lemma *RiemannInt_P18* :
$\forall\ (f\ g{:}R \to R)\ (a\ b{:}R)\ (pr1{:}Riemann\_integrable\ f\ a\ b)$
$(pr2{:}Riemann\_integrable\ g\ a\ b)$,
$a \leq b \to$
$(\forall\ x{:}R,\ a < x < b \to f\ x = g\ x) \to RiemannInt\ pr1 = RiemannInt\ pr2.$

Lemma *RiemannInt_P19* :
$\forall\ (f\ g{:}R \to R)\ (a\ b{:}R)\ (pr1{:}Riemann\_integrable\ f\ a\ b)$
$(pr2{:}Riemann\_integrable\ g\ a\ b)$,
$a \leq b \to$
$(\forall\ x{:}R,\ a < x < b \to f\ x \leq g\ x) \to RiemannInt\ pr1 \leq RiemannInt\ pr2.$

Lemma *FTC_P1* :
$\forall\ (f{:}R \to R)\ (a\ b{:}R)$,
$a \leq b \to$
$(\forall\ x{:}R,\ a \leq x \leq b \to continuity\_pt\ f\ x) \to$
$\forall\ x{:}R,\ a \leq x \to x \leq b \to Riemann\_integrable\ f\ a\ x.$

Definition *primitive* $(f{:}R \to R)\ (a\ b{:}R)\ (h{:}a \leq b)$
$(pr{:}\forall\ x{:}R,\ a \leq x \to x \leq b \to Riemann\_integrable\ f\ a\ x)$
$(x{:}R) : R :=$
match *Rle_dec* $a\ x$ with
| *left* $r \Rightarrow$
match *Rle_dec* $x\ b$ with
| *left* $r0 \Rightarrow RiemannInt\ (pr\ x\ r\ r0)$
| *right* $\_ \Rightarrow f\ b \times (x - b) + RiemannInt\ (pr\ b\ h\ (Rle\_refl\ b))$
end
| *right* $\_ \Rightarrow f\ a \times (x - a)$
end.

Lemma *RiemannInt_P20* :
$\forall\ (f{:}R \to R)\ (a\ b{:}R)\ (h{:}a \leq b)$
$(pr{:}\forall\ x{:}R,\ a \leq x \to x \leq b \to Riemann\_integrable\ f\ a\ x)$
$(pr0{:}Riemann\_integrable\ f\ a\ b)$,
$RiemannInt\ pr0 = primitive\ h\ pr\ b - primitive\ h\ pr\ a.$

Lemma *RiemannInt_P21* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R)$,
$a \leq b \to$
$b \leq c \to$
$Riemann\_integrable\ f\ a\ b \to$
$Riemann\_integrable\ f\ b\ c \to Riemann\_integrable\ f\ a\ c.$

Lemma *RiemannInt_P22* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R)$,
$Riemann\_integrable\ f\ a\ b \to a \leq c \leq b \to Riemann\_integrable\ f\ a\ c.$

Lemma *RiemannInt_P23* :
$\forall\ (f{:}R \to R)\ (a\ b\ c{:}R)$,
$Riemann\_integrable\ f\ a\ b \to a \leq c \leq b \to Riemann\_integrable\ f\ c\ b.$

Lemma *RiemannInt_P24* :
∀ (*f*:*R* → *R*) (*a b c*:*R*),
Riemann_integrable f a b →
Riemann_integrable f b c → *Riemann_integrable f a c*.

Lemma *RiemannInt_P25* :
∀ (*f*:*R* → *R*) (*a b c*:*R*) (*pr1*:*Riemann_integrable f a b*)
(*pr2*:*Riemann_integrable f b c*) (*pr3*:*Riemann_integrable f a c*),
a ≤ *b* → *b* ≤ *c* → *RiemannInt pr1* + *RiemannInt pr2* = *RiemannInt pr3*.

Lemma *RiemannInt_P26* :
∀ (*f*:*R* → *R*) (*a b c*:*R*) (*pr1*:*Riemann_integrable f a b*)
(*pr2*:*Riemann_integrable f b c*) (*pr3*:*Riemann_integrable f a c*),
RiemannInt pr1 + *RiemannInt pr2* = *RiemannInt pr3*.

Lemma *RiemannInt_P27* :
∀ (*f*:*R* → *R*) (*a b x*:*R*) (*h*:*a* ≤ *b*)
(*C0*:∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*),
a < *x* < *b* → *derivable_pt_lim* (*primitive h* (*FTC_P1 h C0*)) *x* (*f x*).

Lemma *RiemannInt_P28* :
∀ (*f*:*R* → *R*) (*a b x*:*R*) (*h*:*a* ≤ *b*)
(*C0*:∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*),
a ≤ *x* ≤ *b* → *derivable_pt_lim* (*primitive h* (*FTC_P1 h C0*)) *x* (*f x*).

Lemma *RiemannInt_P29* :
∀ (*f*:*R* → *R*) *a b* (*h*:*a* ≤ *b*)
(*C0*:∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*),
antiderivative f (*primitive h* (*FTC_P1 h C0*)) *a b*.

Lemma *RiemannInt_P30* :
∀ (*f*:*R* → *R*) (*a b*:*R*),
a ≤ *b* →
(∀ *x*:*R*, *a* ≤ *x* ≤ *b* → *continuity_pt f x*) →
sigT (fun *g*:*R* → *R* ⇒ *antiderivative f g a b*).

Record *C1_fun* : Type := *mkC1*
{*c1* :> *R* → *R*; *diff0* : *derivable c1*; *cont1* : *continuity* (*derive c1 diff0*)}.

Lemma *RiemannInt_P31* :
∀ (*f*:*C1_fun*) (*a b*:*R*),
a ≤ *b* → *antiderivative* (*derive f* (*diff0 f*)) *f a b*.

Lemma *RiemannInt_P32* :
∀ (*f*:*C1_fun*) (*a b*:*R*), *Riemann_integrable* (*derive f* (*diff0 f*)) *a b*.

Lemma *RiemannInt_P33* :
∀ (*f*:*C1_fun*) (*a b*:*R*) (*pr*:*Riemann_integrable* (*derive f* (*diff0 f*)) *a b*),
a ≤ *b* → *RiemannInt pr* = *f b* - *f a*.

Lemma *FTC_Riemann* :
∀ (*f*:*C1_fun*) (*a b*:*R*) (*pr*:*Riemann_integrable* (*derive f* (*diff0 f*)) *a b*),
RiemannInt pr = *f b* - *f a*.

Chapter 119

Module Coq.Reals.R_Ifp

Complements for the reals.Integer and fractional part
Require Import *Rbase*.
Require Import *Omega*.
Open Local *Scope R_scope*.

119.1 Fractional part

Definition *Int_part* $(r{:}R) : Z := (up\ r\ \text{-}\ 1)\%Z$.

Definition *frac_part* $(r{:}R) : R := r\ \text{-}\ IZR\ (Int\_part\ r)$.

Lemma *tech_up* : $\forall\ (r{:}R)\ (z{:}Z),\ r < IZR\ z \rightarrow IZR\ z \leq r + 1 \rightarrow z = up\ r$.

Lemma *up_tech* :
$\forall\ (r{:}R)\ (z{:}Z),\ IZR\ z \leq r \rightarrow r < IZR\ (z + 1) \rightarrow (z + 1)\%Z = up\ r$.

Lemma *fp_R0* : $frac\_part\ 0 = 0$.

Lemma *for_base_fp* : $\forall\ r{:}R,\ IZR\ (up\ r)\ \text{-}\ r > 0 \wedge IZR\ (up\ r)\ \text{-}\ r \leq 1$.

Lemma *base_fp* : $\forall\ r{:}R,\ frac\_part\ r \geq 0 \wedge frac\_part\ r < 1$.

119.2 Properties

Lemma *base_Int_part* :
$\forall\ r{:}R,\ IZR\ (Int\_part\ r) \leq r \wedge IZR\ (Int\_part\ r)\ \text{-}\ r > \text{-}1$.

Lemma *Int_part_INR* : $\forall\ n{:}nat,\ Int\_part\ (INR\ n) = Z\_of\_nat\ n$.

Lemma *fp_nat* : $\forall\ r{:}R,\ frac\_part\ r = 0 \rightarrow \exists\ c : Z,\ r = IZR\ c$.

Lemma *R0_fp_O* : $\forall\ r{:}R,\ 0 \neq frac\_part\ r \rightarrow 0 \neq r$.

Lemma *Rminus_Int_part1* :
$\forall\ r1\ r2{:}R,$
$frac\_part\ r1 \geq frac\_part\ r2 \rightarrow$
$Int\_part\ (r1\ \text{-}\ r2) = (Int\_part\ r1\ \text{-}\ Int\_part\ r2)\%Z$.

Lemma *Rminus_Int_part2* :
∀ *r1 r2*:*R*,
frac_part r1 < *frac_part r2* →
Int_part (*r1* - *r2*) = (*Int_part r1* - *Int_part r2* - 1)%*Z*.

Lemma *Rminus_fp1* :
∀ *r1 r2*:*R*,
frac_part r1 ≥ *frac_part r2* →
frac_part (*r1* - *r2*) = *frac_part r1* - *frac_part r2*.

Lemma *Rminus_fp2* :
∀ *r1 r2*:*R*,
frac_part r1 < *frac_part r2* →
frac_part (*r1* - *r2*) = *frac_part r1* - *frac_part r2* + 1.

Lemma *plus_Int_part1* :
∀ *r1 r2*:*R*,
frac_part r1 + *frac_part r2* ≥ 1 →
Int_part (*r1* + *r2*) = (*Int_part r1* + *Int_part r2* + 1)%*Z*.

Lemma *plus_Int_part2* :
∀ *r1 r2*:*R*,
frac_part r1 + *frac_part r2* < 1 →
Int_part (*r1* + *r2*) = (*Int_part r1* + *Int_part r2*)%*Z*.

Lemma *plus_frac_part1* :
∀ *r1 r2*:*R*,
frac_part r1 + *frac_part r2* ≥ 1 →
frac_part (*r1* + *r2*) = *frac_part r1* + *frac_part r2* - 1.

Lemma *plus_frac_part2* :
∀ *r1 r2*:*R*,
frac_part r1 + *frac_part r2* < 1 →
frac_part (*r1* + *r2*) = *frac_part r1* + *frac_part r2*.

Chapter 120

Module Coq.Reals.RIneq

Basic lemmas for the classical reals numbers
Require Export *Raxioms*.
Require Import *Rpow_def*.
Require Import *Zpower*.
Require Export *ZArithRing*.
Require Import *Omega*.
Require Export *RealField*.

Open Local *Scope Z_scope*.
Open Local *Scope R_scope*.

Implicit Type $r : R$.

120.1 Relation between orders and equality

Lemma *Rlt_irefl* : $\forall\ r, \neg\ r < r$.
Hint *Resolve Rlt_irefl*: *real*.

Lemma *Rle_refl* : $\forall\ r, r \leq r$.

Lemma *Rlt_not_eq* : $\forall\ r1\ r2, r1 < r2 \rightarrow r1 \neq r2$.

Lemma *Rgt_not_eq* : $\forall\ r1\ r2, r1 > r2 \rightarrow r1 \neq r2$.

Lemma *Rlt_dichotomy_converse* : $\forall\ r1\ r2, r1 < r2 \vee r1 > r2 \rightarrow r1 \neq r2$.
Hint *Resolve Rlt_dichotomy_converse*: *real*.

Reasoning by case on equalities and order

Lemma *Req_dec* : $\forall\ r1\ r2, r1 = r2 \vee r1 \neq r2$.
Hint *Resolve Req_dec*: *real*.

Lemma *Rtotal_order* : $\forall\ r1\ r2, r1 < r2 \vee r1 = r2 \vee r1 > r2$.

Lemma *Rdichotomy* : $\forall\ r1\ r2, r1 \neq r2 \rightarrow r1 < r2 \vee r1 > r2$.

120.2 Order Lemma : relating $<$, $>$, $\leq$ and $\geq$

Lemma *Rlt_le* : $\forall\ r1\ r2, r1 < r2 \to r1 \leq r2$.
Hint *Resolve Rlt_le*: *real*.

Lemma *Rle_ge* : $\forall\ r1\ r2, r1 \leq r2 \to r2 \geq r1$.

Hint Immediate *Rle_ge*: *real*.

Lemma *Rge_le* : $\forall\ r1\ r2, r1 \geq r2 \to r2 \leq r1$.

Hint *Resolve Rge_le*: *real*.

Lemma *Rnot_le_lt* : $\forall\ r1\ r2, \neg\ r1 \leq r2 \to r2 < r1$.

Hint Immediate *Rnot_le_lt*: *real*.

Lemma *Rnot_ge_lt* : $\forall\ r1\ r2, \neg\ r1 \geq r2 \to r1 < r2$.

Lemma *Rlt_not_le* : $\forall\ r1\ r2, r2 < r1 \to \neg\ r1 \leq r2$.

Lemma *Rgt_not_le* : $\forall\ r1\ r2, r1 > r2 \to \neg\ r1 \leq r2$.

Hint Immediate *Rlt_not_le*: *real*.

Lemma *Rle_not_lt* : $\forall\ r1\ r2, r2 \leq r1 \to \neg\ r1 < r2$.

Lemma *Rlt_not_ge* : $\forall\ r1\ r2, r1 < r2 \to \neg\ r1 \geq r2$.

Hint Immediate *Rlt_not_ge*: *real*.

Lemma *Req_le* : $\forall\ r1\ r2, r1 = r2 \to r1 \leq r2$.
Hint Immediate *Req_le*: *real*.

Lemma *Req_ge* : $\forall\ r1\ r2, r1 = r2 \to r1 \geq r2$.
Hint Immediate *Req_ge*: *real*.

Lemma *Req_le_sym* : $\forall\ r1\ r2, r2 = r1 \to r1 \leq r2$.
Hint Immediate *Req_le_sym*: *real*.

Lemma *Req_ge_sym* : $\forall\ r1\ r2, r2 = r1 \to r1 \geq r2$.
Hint Immediate *Req_ge_sym*: *real*.

Lemma *Rle_antisym* : $\forall\ r1\ r2, r1 \leq r2 \to r2 \leq r1 \to r1 = r2$.
Hint *Resolve Rle_antisym*: *real*.

Lemma *Rle_le_eq* : $\forall\ r1\ r2, r1 \leq r2 \wedge r2 \leq r1 \leftrightarrow r1 = r2$.

Lemma *Rlt_eq_compat* :
$\forall\ r1\ r2\ r3\ r4, r1 = r2 \to r2 < r4 \to r4 = r3 \to r1 < r3$.

Lemma *Rle_trans* : $\forall\ r1\ r2\ r3, r1 \leq r2 \to r2 \leq r3 \to r1 \leq r3$.

Lemma *Rle_lt_trans* : $\forall\ r1\ r2\ r3, r1 \leq r2 \to r2 < r3 \to r1 < r3$.

Lemma *Rlt_le_trans* : $\forall\ r1\ r2\ r3, r1 < r2 \to r2 \leq r3 \to r1 < r3$.

Decidability of the order
Lemma *Rlt_dec* : $\forall\ r1\ r2, \{r1 < r2\} + \{\tilde{}\ r1 < r2\}$.

Lemma *Rle_dec* : $\forall\ r1\ r2, \{r1 \leq r2\} + \{\tilde{}\ r1 \leq r2\}$.

Lemma *Rgt_dec* : $\forall\ r1\ r2, \{r1 > r2\} + \{\tilde{}\ r1 > r2\}$.

Lemma *Rge_dec* : $\forall\ r1\ r2, \{r1 \geq r2\} + \{\tilde{}\ r1 \geq r2\}$.

Lemma *Rlt_le_dec* : $\forall\ r1\ r2, \{r1 < r2\} + \{r2 \leq r1\}$.

Lemma *Rle_or_lt* : $\forall\ r1\ r2, r1 \leq r2 \vee r2 < r1$.

Lemma *Rle_lt_or_eq_dec* : $\forall\ r1\ r2, r1 \leq r2 \rightarrow \{r1 < r2\} + \{r1 = r2\}$.

Lemma *inser_trans_R* :
$\forall\ r1\ r2\ r3\ r4, r1 \leq r2 < r3 \rightarrow \{r1 \leq r2 < r4\} + \{r4 \leq r2 < r3\}$.

120.3 Field Lemmas

120.3.1 Addition

Lemma *Rplus_ne* : $\forall\ r, r + 0 = r \wedge 0 + r = r$.
Hint *Resolve Rplus_ne*: *real v62*.

Lemma *Rplus_0_r* : $\forall\ r, r + 0 = r$.
Hint *Resolve Rplus_0_r*: *real*.

Lemma *Rplus_opp_l* : $\forall\ r, - r + r = 0$.
Hint *Resolve Rplus_opp_l*: *real*.

Lemma *Rplus_opp_r_uniq* : $\forall\ r1\ r2, r1 + r2 = 0 \rightarrow r2 = - r1$.

Hint *Resolve* (*f_equal* (*A*:=*R*)): *real*.

Lemma *Rplus_eq_compat_l* : $\forall\ r\ r1\ r2, r1 = r2 \rightarrow r + r1 = r + r2$.

Hint *Resolve Rplus_eq_compat_l*: *v62*.

Lemma *Rplus_eq_reg_l* : $\forall\ r\ r1\ r2, r + r1 = r + r2 \rightarrow r1 = r2$.
Hint *Resolve Rplus_eq_reg_l*: *real*.

Lemma *Rplus_0_r_uniq* : $\forall\ r\ r1, r + r1 = r \rightarrow r1 = 0$.

120.3.2 Multiplication

Lemma *Rinv_r* : $\forall\ r, r \neq 0 \rightarrow r \times / r = 1$.
Hint *Resolve Rinv_r*: *real*.

Lemma *Rinv_l_sym* : $\forall\ r, r \neq 0 \rightarrow 1 = / r \times r$.

Lemma *Rinv_r_sym* : $\forall\ r, r \neq 0 \rightarrow 1 = r \times / r$.
Hint *Resolve Rinv_l_sym Rinv_r_sym*: *real*.

Lemma *Rmult_0_r* : $\forall\ r, r \times 0 = 0$.
Hint *Resolve Rmult_0_r*: *real v62*.

Lemma *Rmult_0_l* : $\forall\ r, 0 \times r = 0$.

Hint *Resolve Rmult_0_l*: *real v62.*

Lemma *Rmult_ne* : $\forall\ r,\ r \times 1 = r \wedge 1 \times r = r$.
Hint *Resolve Rmult_ne*: *real v62.*

Lemma *Rmult_1_r* : $\forall\ r,\ r \times 1 = r$.
Hint *Resolve Rmult_1_r*: *real.*

Lemma *Rmult_eq_compat_l* : $\forall\ r\ r1\ r2,\ r1 = r2 \rightarrow r \times r1 = r \times r2$.

Hint *Resolve Rmult_eq_compat_l*: *v62.*

Lemma *Rmult_eq_reg_l* : $\forall\ r\ r1\ r2,\ r \times r1 = r \times r2 \rightarrow r \neq 0 \rightarrow r1 = r2$.

Lemma *Rmult_integral* : $\forall\ r1\ r2,\ r1 \times r2 = 0 \rightarrow r1 = 0 \vee r2 = 0$.

Lemma *Rmult_eq_0_compat* : $\forall\ r1\ r2,\ r1 = 0 \vee r2 = 0 \rightarrow r1 \times r2 = 0$.

Hint *Resolve Rmult_eq_0_compat*: *real.*

Lemma *Rmult_eq_0_compat_r* : $\forall\ r1\ r2,\ r1 = 0 \rightarrow r1 \times r2 = 0$.

Lemma *Rmult_eq_0_compat_l* : $\forall\ r1\ r2,\ r2 = 0 \rightarrow r1 \times r2 = 0$.

Lemma *Rmult_neq_0_reg* : $\forall\ r1\ r2,\ r1 \times r2 \neq 0 \rightarrow r1 \neq 0 \wedge r2 \neq 0$.

Lemma *Rmult_integral_contrapositive* :
$\forall\ r1\ r2,\ r1 \neq 0 \wedge r2 \neq 0 \rightarrow r1 \times r2 \neq 0$.
Hint *Resolve Rmult_integral_contrapositive*: *real.*

Lemma *Rmult_plus_distr_r* :
$\forall\ r1\ r2\ r3,\ (r1 + r2) \times r3 = r1 \times r3 + r2 \times r3$.

120.3.3 Square function

Definition *Rsqr* $r : R := r \times r$.

Lemma *Rsqr_0* : $Rsqr\ 0 = 0$.

Lemma *Rsqr_0_uniq* : $\forall\ r,\ Rsqr\ r = 0 \rightarrow r = 0$.

120.3.4 Opposite

Lemma *Ropp_eq_compat* : $\forall\ r1\ r2,\ r1 = r2 \rightarrow -\ r1 = -\ r2$.
Hint *Resolve Ropp_eq_compat*: *real.*

Lemma *Ropp_0* : $-0 = 0$.
Hint *Resolve Ropp_0*: *real v62.*

Lemma *Ropp_eq_0_compat* : $\forall\ r,\ r = 0 \rightarrow -\ r = 0$.
Hint *Resolve Ropp_eq_0_compat*: *real.*

Lemma *Ropp_involutive* : $\forall\ r,\ - -\ r = r$.
Hint *Resolve Ropp_involutive*: *real.*

Lemma *Ropp_neq_0_compat* : $\forall\ r,\ r \neq 0 \rightarrow -\ r \neq 0$.

Hint *Resolve Ropp_neq_0_compat*: *real*.

Lemma *Ropp_plus_distr* : ∀ *r1 r2*, - (*r1* + *r2*) = - *r1* + - *r2*.
Hint *Resolve Ropp_plus_distr*: *real*.

120.3.5 Opposite and multiplication

Lemma *Ropp_mult_distr_l_reverse* : ∀ *r1 r2*, - *r1* × *r2* = - (*r1* × *r2*).
Hint *Resolve Ropp_mult_distr_l_reverse*: *real*.

Lemma *Rmult_opp_opp* : ∀ *r1 r2*, - *r1* × - *r2* = *r1* × *r2*.
Hint *Resolve Rmult_opp_opp*: *real*.

Lemma *Ropp_mult_distr_r_reverse* : ∀ *r1 r2*, *r1* × - *r2* = - (*r1* × *r2*).

120.3.6 Substraction

Lemma *Rminus_0_r* : ∀ *r*, *r* - 0 = *r*.
Hint *Resolve Rminus_0_r*: *real*.

Lemma *Rminus_0_l* : ∀ *r*, 0 - *r* = - *r*.
Hint *Resolve Rminus_0_l*: *real*.

Lemma *Ropp_minus_distr* : ∀ *r1 r2*, - (*r1* - *r2*) = *r2* - *r1*.
Hint *Resolve Ropp_minus_distr*: *real*.

Lemma *Ropp_minus_distr'* : ∀ *r1 r2*, - (*r2* - *r1*) = *r1* - *r2*.
Hint *Resolve Ropp_minus_distr'*: *real*.

Lemma *Rminus_diag_eq* : ∀ *r1 r2*, *r1* = *r2* → *r1* - *r2* = 0.
Hint *Resolve Rminus_diag_eq*: *real*.

Lemma *Rminus_diag_uniq* : ∀ *r1 r2*, *r1* - *r2* = 0 → *r1* = *r2*.
Hint Immediate *Rminus_diag_uniq*: *real*.

Lemma *Rminus_diag_uniq_sym* : ∀ *r1 r2*, *r2* - *r1* = 0 → *r1* = *r2*.
Hint Immediate *Rminus_diag_uniq_sym*: *real*.

Lemma *Rplus_minus* : ∀ *r1 r2*, *r1* + (*r2* - *r1*) = *r2*.
Hint *Resolve Rplus_minus*: *real*.

Lemma *Rminus_eq_contra* : ∀ *r1 r2*, *r1* ≠ *r2* → *r1* - *r2* ≠ 0.
Hint *Resolve Rminus_eq_contra*: *real*.

Lemma *Rminus_not_eq* : ∀ *r1 r2*, *r1* - *r2* ≠ 0 → *r1* ≠ *r2*.
Hint *Resolve Rminus_not_eq*: *real*.

Lemma *Rminus_not_eq_right* : ∀ *r1 r2*, *r2* - *r1* ≠ 0 → *r1* ≠ *r2*.
Hint *Resolve Rminus_not_eq_right*: *real*.

Lemma *Rmult_minus_distr_l* :
∀ *r1 r2 r3*, *r1* × (*r2* - *r3*) = *r1* × *r2* - *r1* × *r3*.

120.3.7 Inverse

Lemma *Rinv_1* : / 1 = 1.
Hint *Resolve Rinv_1*: *real*.

Lemma *Rinv_neq_0_compat* : ∀ *r*, *r* ≠ 0 → / *r* ≠ 0.
Hint *Resolve Rinv_neq_0_compat*: *real*.

Lemma *Rinv_involutive* : ∀ *r*, *r* ≠ 0 → / / *r* = *r*.
Hint *Resolve Rinv_involutive*: *real*.

Lemma *Rinv_mult_distr* :
∀ *r1 r2*, *r1* ≠ 0 → *r2* ≠ 0 → / (*r1* × *r2*) = / *r1* × / *r2*.

Lemma *Ropp_inv_permute* : ∀ *r*, *r* ≠ 0 → - / *r* = / - *r*.

Lemma *Rinv_r_simpl_r* : ∀ *r1 r2*, *r1* ≠ 0 → *r1* × / *r1* × *r2* = *r2*.

Lemma *Rinv_r_simpl_l* : ∀ *r1 r2*, *r1* ≠ 0 → *r2* × *r1* × / *r1* = *r2*.

Lemma *Rinv_r_simpl_m* : ∀ *r1 r2*, *r1* ≠ 0 → *r1* × *r2* × / *r1* = *r2*.
Hint *Resolve Rinv_r_simpl_l Rinv_r_simpl_r Rinv_r_simpl_m*: *real*.

Lemma *Rinv_mult_simpl* :
∀ *r1 r2 r3*, *r1* ≠ 0 → *r1* × / *r2* × (*r3* × / *r1*) = *r3* × / *r2*.

120.4 Field operations and order

120.4.1 Order and addition

Lemma *Rplus_lt_compat_r* : ∀ *r r1 r2*, *r1* < *r2* → *r1* + *r* < *r2* + *r*.

Hint *Resolve Rplus_lt_compat_r*: *real*.

Lemma *Rplus_lt_reg_r* : ∀ *r r1 r2*, *r* + *r1* < *r* + *r2* → *r1* < *r2*.

Lemma *Rplus_le_compat_l* : ∀ *r r1 r2*, *r1* ≤ *r2* → *r* + *r1* ≤ *r* + *r2*.

Lemma *Rplus_le_compat_r* : ∀ *r r1 r2*, *r1* ≤ *r2* → *r1* + *r* ≤ *r2* + *r*.

Hint *Resolve Rplus_le_compat_l Rplus_le_compat_r*: *real*.

Lemma *Rplus_le_reg_l* : ∀ *r r1 r2*, *r* + *r1* ≤ *r* + *r2* → *r1* ≤ *r2*.

Lemma *sum_inequa_Rle_lt* :
∀ *a x b c y d*:*R*,
a ≤ *x* → *x* < *b* → *c* < *y* → *y* ≤ *d* → *a* + *c* < *x* + *y* < *b* + *d*.

Lemma *Rplus_lt_compat* :
∀ *r1 r2 r3 r4*, *r1* < *r2* → *r3* < *r4* → *r1* + *r3* < *r2* + *r4*.

Lemma *Rplus_le_compat* :
∀ *r1 r2 r3 r4*, *r1* ≤ *r2* → *r3* ≤ *r4* → *r1* + *r3* ≤ *r2* + *r4*.

Lemma *Rplus_lt_le_compat* :
$\forall\ r1\ r2\ r3\ r4, r1 < r2 \rightarrow r3 \leq r4 \rightarrow r1 + r3 < r2 + r4.$

Lemma *Rplus_le_lt_compat* :
$\forall\ r1\ r2\ r3\ r4, r1 \leq r2 \rightarrow r3 < r4 \rightarrow r1 + r3 < r2 + r4.$

Hint Immediate *Rplus_lt_compat Rplus_le_compat Rplus_lt_le_compat Rplus_le_lt_compat*: *real*.

120.4.2 Order and Opposite

Lemma *Ropp_gt_lt_contravar* : $\forall\ r1\ r2, r1 > r2 \rightarrow -\ r1 < -\ r2.$
Hint *Resolve Ropp_gt_lt_contravar*.

Lemma *Ropp_lt_gt_contravar* : $\forall\ r1\ r2, r1 < r2 \rightarrow -\ r1 > -\ r2.$
Hint *Resolve Ropp_lt_gt_contravar*: *real*.

Lemma *Ropp_lt_cancel* : $\forall\ r1\ r2, -\ r2 < -\ r1 \rightarrow r1 < r2.$
Hint Immediate *Ropp_lt_cancel*: *real*.

Lemma *Ropp_lt_contravar* : $\forall\ r1\ r2, r2 < r1 \rightarrow -\ r1 < -\ r2.$
Hint *Resolve Ropp_lt_contravar*: *real*.

Lemma *Ropp_le_ge_contravar* : $\forall\ r1\ r2, r1 \leq r2 \rightarrow -\ r1 \geq -\ r2.$
Hint *Resolve Ropp_le_ge_contravar*: *real*.

Lemma *Ropp_le_cancel* : $\forall\ r1\ r2, -\ r2 \leq -\ r1 \rightarrow r1 \leq r2.$
Hint Immediate *Ropp_le_cancel*: *real*.

Lemma *Ropp_le_contravar* : $\forall\ r1\ r2, r2 \leq r1 \rightarrow -\ r1 \leq -\ r2.$
Hint *Resolve Ropp_le_contravar*: *real*.

Lemma *Ropp_ge_le_contravar* : $\forall\ r1\ r2, r1 \geq r2 \rightarrow -\ r1 \leq -\ r2.$
Hint *Resolve Ropp_ge_le_contravar*: *real*.

Lemma *Ropp_0_lt_gt_contravar* : $\forall\ r, 0 < r \rightarrow 0 > -\ r.$
Hint *Resolve Ropp_0_lt_gt_contravar*: *real*.

Lemma *Ropp_0_gt_lt_contravar* : $\forall\ r, 0 > r \rightarrow 0 < -\ r.$
Hint *Resolve Ropp_0_gt_lt_contravar*: *real*.

Lemma *Ropp_lt_gt_0_contravar* : $\forall\ r, r > 0 \rightarrow -\ r < 0.$

Lemma *Ropp_gt_lt_0_contravar* : $\forall\ r, r < 0 \rightarrow -\ r > 0.$
Hint *Resolve Ropp_lt_gt_0_contravar Ropp_gt_lt_0_contravar*: *real*.

Lemma *Ropp_0_le_ge_contravar* : $\forall\ r, 0 \leq r \rightarrow 0 \geq -\ r.$
Hint *Resolve Ropp_0_le_ge_contravar*: *real*.

Lemma *Ropp_0_ge_le_contravar* : $\forall\ r, 0 \geq r \rightarrow 0 \leq -\ r.$
Hint *Resolve Ropp_0_ge_le_contravar*: *real*.

120.4.3 Order and multiplication

Lemma *Rmult_lt_compat_r* : $\forall\ r\ r1\ r2, 0 < r \rightarrow r1 < r2 \rightarrow r1 \times r < r2 \times r$.
Hint *Resolve Rmult_lt_compat_r*.

Lemma *Rmult_lt_reg_l* : $\forall\ r\ r1\ r2, 0 < r \rightarrow r \times r1 < r \times r2 \rightarrow r1 < r2$.

Lemma *Rmult_lt_gt_compat_neg_l* :
$\forall\ r\ r1\ r2, r < 0 \rightarrow r1 < r2 \rightarrow r \times r1 > r \times r2$.

Lemma *Rmult_le_compat_l* :
$\forall\ r\ r1\ r2, 0 \leq r \rightarrow r1 \leq r2 \rightarrow r \times r1 \leq r \times r2$.
Hint *Resolve Rmult_le_compat_l*: *real*.

Lemma *Rmult_le_compat_r* :
$\forall\ r\ r1\ r2, 0 \leq r \rightarrow r1 \leq r2 \rightarrow r1 \times r \leq r2 \times r$.
Hint *Resolve Rmult_le_compat_r*: *real*.

Lemma *Rmult_le_reg_l* : $\forall\ r\ r1\ r2, 0 < r \rightarrow r \times r1 \leq r \times r2 \rightarrow r1 \leq r2$.

Lemma *Rmult_le_compat_neg_l* :
$\forall\ r\ r1\ r2, r \leq 0 \rightarrow r1 \leq r2 \rightarrow r \times r2 \leq r \times r1$.
Hint *Resolve Rmult_le_compat_neg_l*: *real*.

Lemma *Rmult_le_ge_compat_neg_l* :
$\forall\ r\ r1\ r2, r \leq 0 \rightarrow r1 \leq r2 \rightarrow r \times r1 \geq r \times r2$.
Hint *Resolve Rmult_le_ge_compat_neg_l*: *real*.

Lemma *Rmult_le_compat* :
$\forall\ r1\ r2\ r3\ r4$,
$0 \leq r1 \rightarrow 0 \leq r3 \rightarrow r1 \leq r2 \rightarrow r3 \leq r4 \rightarrow r1 \times r3 \leq r2 \times r4$.
Hint *Resolve Rmult_le_compat*: *real*.

Lemma *Rmult_gt_0_lt_compat* :
$\forall\ r1\ r2\ r3\ r4$,
$r3 > 0 \rightarrow r2 > 0 \rightarrow r1 < r2 \rightarrow r3 < r4 \rightarrow r1 \times r3 < r2 \times r4$.

Lemma *Rmult_ge_0_gt_0_lt_compat* :
$\forall\ r1\ r2\ r3\ r4$,
$r3 \geq 0 \rightarrow r2 > 0 \rightarrow r1 < r2 \rightarrow r3 < r4 \rightarrow r1 \times r3 < r2 \times r4$.

120.4.4 Order and Substractions

Lemma *Rlt_minus* : $\forall\ r1\ r2, r1 < r2 \rightarrow r1 - r2 < 0$.
Hint *Resolve Rlt_minus*: *real*.

Lemma *Rle_minus* : $\forall\ r1\ r2, r1 \leq r2 \rightarrow r1 - r2 \leq 0$.

Lemma *Rminus_lt* : $\forall\ r1\ r2, r1 - r2 < 0 \rightarrow r1 < r2$.

Lemma *Rminus_le* : $\forall\ r1\ r2, r1 - r2 \leq 0 \rightarrow r1 \leq r2$.

Lemma *tech_Rplus* : $\forall\ r\ (s{:}R), 0 \leq r \rightarrow 0 < s \rightarrow r + s \neq 0$.
Hint Immediate *tech_Rplus*: *real*.

120.4.5 Order and the square function

Lemma *Rle_0_sqr* : $\forall\ r, 0 \leq Rsqr\ r$.

Lemma *Rlt_0_sqr* : $\forall\ r, r \neq 0 \rightarrow 0 < Rsqr\ r$.
Hint *Resolve Rle_0_sqr Rlt_0_sqr*: *real*.

120.4.6 Zero is less than one

Lemma *Rlt_0_1* : $0 < 1$.
Hint *Resolve Rlt_0_1*: *real*.

Lemma *Rle_0_1* : $0 \leq 1$.

120.4.7 Order and inverse

Lemma *Rinv_0_lt_compat* : $\forall\ r, 0 < r \rightarrow 0 < / r$.
Hint *Resolve Rinv_0_lt_compat*: *real*.

Lemma *Rinv_lt_0_compat* : $\forall\ r, r < 0 \rightarrow / r < 0$.
Hint *Resolve Rinv_lt_0_compat*: *real*.

Lemma *Rinv_lt_contravar* : $\forall\ r1\ r2, 0 < r1 \times r2 \rightarrow r1 < r2 \rightarrow / r2 < / r1$.

Lemma *Rinv_1_lt_contravar* : $\forall\ r1\ r2, 1 \leq r1 \rightarrow r1 < r2 \rightarrow / r2 < / r1$.
Hint *Resolve Rinv_1_lt_contravar*: *real*.

120.5 Greater

Lemma *Rge_antisym* : $\forall\ r1\ r2, r1 \geq r2 \rightarrow r2 \geq r1 \rightarrow r1 = r2$.

Lemma *Rnot_lt_ge* : $\forall\ r1\ r2, \neg\ r1 < r2 \rightarrow r1 \geq r2$.

Lemma *Rnot_lt_le* : $\forall\ r1\ r2, \neg\ r1 < r2 \rightarrow r2 \leq r1$.

Lemma *Rnot_gt_le* : $\forall\ r1\ r2, \neg\ r1 > r2 \rightarrow r1 \leq r2$.

Lemma *Rgt_ge* : $\forall\ r1\ r2, r1 > r2 \rightarrow r1 \geq r2$.

Lemma *Rge_gt_trans* : $\forall\ r1\ r2\ r3, r1 \geq r2 \rightarrow r2 > r3 \rightarrow r1 > r3$.

Lemma *Rgt_ge_trans* : $\forall\ r1\ r2\ r3, r1 > r2 \rightarrow r2 \geq r3 \rightarrow r1 > r3$.

Lemma *Rgt_trans* : $\forall\ r1\ r2\ r3, r1 > r2 \rightarrow r2 > r3 \rightarrow r1 > r3$.

Lemma *Rge_trans* : $\forall\ r1\ r2\ r3, r1 \geq r2 \rightarrow r2 \geq r3 \rightarrow r1 \geq r3$.

Lemma *Rle_lt_0_plus_1* : $\forall\ r, 0 \leq r \rightarrow 0 < r + 1$.
Hint *Resolve Rle_lt_0_plus_1*: *real*.

Lemma *Rlt_plus_1* : $\forall\ r, r < r + 1$.
Hint *Resolve Rlt_plus_1*: *real*.

Lemma *tech_Rgt_minus* : $\forall\ r1\ r2, 0 < r2 \rightarrow r1 > r1$ - $r2$.

Lemma $Rplus\_gt\_compat\_l : \forall\ r\ r1\ r2,\ r1 > r2 \to r + r1 > r + r2.$
Hint *Resolve Rplus_gt_compat_l: real.*

Lemma $Rplus\_gt\_reg\_l : \forall\ r\ r1\ r2,\ r + r1 > r + r2 \to r1 > r2.$

Lemma $Rplus\_ge\_compat\_l : \forall\ r\ r1\ r2,\ r1 \geq r2 \to r + r1 \geq r + r2.$
Hint *Resolve Rplus_ge_compat_l: real.*

Lemma $Rplus\_ge\_reg\_l : \forall\ r\ r1\ r2,\ r + r1 \geq r + r2 \to r1 \geq r2.$

Lemma *Rmult_ge_compat_r* :
$\forall\ r\ r1\ r2,\ r \geq 0 \to r1 \geq r2 \to r1 \times r \geq r2 \times r.$

Lemma $Rgt\_minus : \forall\ r1\ r2,\ r1 > r2 \to r1 - r2 > 0.$

Lemma $minus\_Rgt : \forall\ r1\ r2,\ r1 - r2 > 0 \to r1 > r2.$

Lemma $Rge\_minus : \forall\ r1\ r2,\ r1 \geq r2 \to r1 - r2 \geq 0.$

Lemma $minus\_Rge : \forall\ r1\ r2,\ r1 - r2 \geq 0 \to r1 \geq r2.$

Lemma $Rmult\_gt\_0\_compat : \forall\ r1\ r2,\ r1 > 0 \to r2 > 0 \to r1 \times r2 > 0.$

Lemma $Rmult\_lt\_0\_compat : \forall\ r1\ r2,\ 0 < r1 \to 0 < r2 \to 0 < r1 \times r2.$

Lemma *Rplus_eq_0_l* :
$\forall\ r1\ r2,\ 0 \leq r1 \to 0 \leq r2 \to r1 + r2 = 0 \to r1 = 0.$

Lemma *Rplus_eq_R0* :
$\forall\ r1\ r2,\ 0 \leq r1 \to 0 \leq r2 \to r1 + r2 = 0 \to r1 = 0 \wedge r2 = 0.$

Lemma $Rplus\_sqr\_eq\_0\_l : \forall\ r1\ r2,\ Rsqr\ r1 + Rsqr\ r2 = 0 \to r1 = 0.$

Lemma *Rplus_sqr_eq_0* :
$\forall\ r1\ r2,\ Rsqr\ r1 + Rsqr\ r2 = 0 \to r1 = 0 \wedge r2 = 0.$

120.6 Injection from N to R

Lemma $S\_INR : \forall\ n{:}nat,\ INR\ (S\ n) = INR\ n + 1.$

Lemma $S\_O\_plus\_INR : \forall\ n{:}nat,\ INR\ (1 + n) = INR\ 1 + INR\ n.$

Lemma $plus\_INR : \forall\ n\ m{:}nat,\ INR\ (n + m) = INR\ n + INR\ m.$

Lemma $minus\_INR : \forall\ n\ m{:}nat,\ (m \leq n)\%nat \to INR\ (n - m) = INR\ n - INR\ m.$

Lemma $mult\_INR : \forall\ n\ m{:}nat,\ INR\ (n \times m) = INR\ n \times INR\ m.$

Hint *Resolve plus_INR minus_INR mult_INR: real.*

Lemma $lt\_INR\_0 : \forall\ n{:}nat,\ (0 < n)\%nat \to 0 < INR\ n.$
Hint *Resolve lt_INR_0: real.*

Lemma $lt\_INR : \forall\ n\ m{:}nat,\ (n < m)\%nat \to INR\ n < INR\ m.$
Hint *Resolve lt_INR: real.*

Lemma $INR\_lt\_1 : \forall\ n{:}nat,\ (1 < n)\%nat \to 1 < INR\ n.$

Hint *Resolve INR_lt_1*: *real*.

Lemma *INR_pos* : ∀ *p*:*positive*, 0 < *INR* (*nat_of_P p*).
Hint *Resolve INR_pos*: *real*.

Lemma *pos_INR* : ∀ *n*:*nat*, 0 ≤ *INR n*.
Hint *Resolve pos_INR*: *real*.

Lemma *INR_lt* : ∀ *n m*:*nat*, *INR n* < *INR m* → (*n* < *m*)%*nat*.
Hint *Resolve INR_lt*: *real*.

Lemma *le_INR* : ∀ *n m*:*nat*, (*n* ≤ *m*)%*nat* → *INR n* ≤ *INR m*.
Hint *Resolve le_INR*: *real*.

Lemma *not_INR_O* : ∀ *n*:*nat*, *INR n* ≠ 0 → *n* ≠ 0%*nat*.
Hint Immediate *not_INR_O*: *real*.

Lemma *not_O_INR* : ∀ *n*:*nat*, *n* ≠ 0%*nat* → *INR n* ≠ 0.
Hint *Resolve not_O_INR*: *real*.

Lemma *not_nm_INR* : ∀ *n m*:*nat*, *n* ≠ *m* → *INR n* ≠ *INR m*.
Hint *Resolve not_nm_INR*: *real*.

Lemma *INR_eq* : ∀ *n m*:*nat*, *INR n* = *INR m* → *n* = *m*.
Hint *Resolve INR_eq*: *real*.

Lemma *INR_le* : ∀ *n m*:*nat*, *INR n* ≤ *INR m* → (*n* ≤ *m*)%*nat*.
Hint *Resolve INR_le*: *real*.

Lemma *not_1_INR* : ∀ *n*:*nat*, *n* ≠ 1%*nat* → *INR n* ≠ 1.
Hint *Resolve not_1_INR*: *real*.

120.7 Injection from *Z* to *R*

Lemma *IZN* : ∀ *n*:*Z*, (0 ≤ *n*)%*Z* → ∃ *m* : *nat*, *n* = *Z_of_nat m*.

Lemma *INR_IZR_INZ* : ∀ *n*:*nat*, *INR n* = *IZR* (*Z_of_nat n*).

Lemma *plus_IZR_NEG_POS* :
∀ *p q*:*positive*, *IZR* (*Zpos p* + *Zneg q*) = *IZR* (*Zpos p*) + *IZR* (*Zneg q*).

Lemma *plus_IZR* : ∀ *n m*:*Z*, *IZR* (*n* + *m*) = *IZR n* + *IZR m*.

Lemma *mult_IZR* : ∀ *n m*:*Z*, *IZR* (*n* × *m*) = *IZR n* × *IZR m*.

Lemma *pow_IZR* : ∀ *z n*, *pow* (*IZR z*) *n* = *IZR* (*Zpower z* (*Z_of_nat n*)).

Lemma *Ropp_Ropp_IZR* : ∀ *n*:*Z*, *IZR* (- *n*) = - *IZR n*.

Lemma *Z_R_minus* : ∀ *n m*:*Z*, *IZR n* - *IZR m* = *IZR* (*n* - *m*).

Lemma *lt_O_IZR* : ∀ *n*:*Z*, 0 < *IZR n* → (0 < *n*)%*Z*.

Lemma *lt_IZR* : ∀ *n m*:*Z*, *IZR n* < *IZR m* → (*n* < *m*)%*Z*.

Lemma *eq_IZR_R0* : ∀ *n*:*Z*, *IZR n* = 0 → *n* = 0%*Z*.

Lemma $eq\_IZR : \forall\ n\ m{:}Z,\ IZR\ n = IZR\ m \rightarrow n = m.$

Lemma $not\_O\_IZR : \forall\ n{:}Z,\ n \neq 0\%Z \rightarrow IZR\ n \neq 0.$

Lemma $le\_O\_IZR : \forall\ n{:}Z,\ 0 \leq IZR\ n \rightarrow (0 \leq n)\%Z.$

Lemma $le\_IZR : \forall\ n\ m{:}Z,\ IZR\ n \leq IZR\ m \rightarrow (n \leq m)\%Z.$

Lemma $le\_IZR\_R1 : \forall\ n{:}Z,\ IZR\ n \leq 1 \rightarrow (n \leq 1)\%Z.$

Lemma $IZR\_ge : \forall\ n\ m{:}Z,\ (n \geq m)\%Z \rightarrow IZR\ n \geq IZR\ m.$

Lemma $IZR\_le : \forall\ n\ m{:}Z,\ (n \leq m)\%Z \rightarrow IZR\ n \leq IZR\ m.$

Lemma $IZR\_lt : \forall\ n\ m{:}Z,\ (n < m)\%Z \rightarrow IZR\ n < IZR\ m.$

Lemma $one\_IZR\_lt1 : \forall\ n{:}Z,\ \text{-}1 < IZR\ n < 1 \rightarrow n = 0\%Z.$

Lemma $one\_IZR\_r\_R1 :$
$\forall\ r\ (n\ m{:}Z),\ r < IZR\ n \leq r + 1 \rightarrow r < IZR\ m \leq r + 1 \rightarrow n = m.$

Lemma $single\_z\_r\_R1 :$
$\forall\ r\ (n\ m{:}Z),$
$r < IZR\ n \rightarrow IZR\ n \leq r + 1 \rightarrow r < IZR\ m \rightarrow IZR\ m \leq r + 1 \rightarrow n = m.$

Lemma $tech\_single\_z\_r\_R1 :$
$\forall\ r\ (n{:}Z),$
$r < IZR\ n \rightarrow$
$IZR\ n \leq r + 1 \rightarrow$
$(\exists\ s : Z,\ s \neq n \wedge r < IZR\ s \wedge IZR\ s \leq r + 1) \rightarrow False.$

120.8 Definitions of new types

Record $nonnegreal$: Type := $mknonnegreal$
$\{nonneg :> R;\ cond\_nonneg : 0 \leq nonneg\}.$

Record $posreal$: Type := $mkposreal\ \{pos :> R;\ cond\_pos : 0 < pos\}.$

Record $nonposreal$: Type := $mknonposreal$
$\{nonpos :> R;\ cond\_nonpos : nonpos \leq 0\}.$

Record $negreal$: Type := $mknegreal\ \{neg :> R;\ cond\_neg : neg < 0\}.$

Record $nonzeroreal$: Type := $mknonzeroreal$
$\{nonzero :> R;\ cond\_nonzero : nonzero \neq 0\}.$

Lemma $prod\_neq\_R0 : \forall\ r1\ r2,\ r1 \neq 0 \rightarrow r2 \neq 0 \rightarrow r1 \times r2 \neq 0.$

Lemma $Rmult\_le\_pos : \forall\ r1\ r2,\ 0 \leq r1 \rightarrow 0 \leq r2 \rightarrow 0 \leq r1 \times r2.$

Lemma $double : \forall\ r1,\ 2 \times r1 = r1 + r1.$

Lemma $double\_var : \forall\ r1,\ r1 = r1\ /\ 2 + r1\ /\ 2.$

120.9 Other rules about < and <=

Lemma *Rplus_lt_0_compat* : $\forall\ r1\ r2, 0 < r1 \rightarrow 0 < r2 \rightarrow 0 < r1 + r2$.

Lemma *Rplus_le_lt_0_compat* : $\forall\ r1\ r2, 0 \leq r1 \rightarrow 0 < r2 \rightarrow 0 < r1 + r2$.

Lemma *Rplus_lt_le_0_compat* : $\forall\ r1\ r2, 0 < r1 \rightarrow 0 \leq r2 \rightarrow 0 < r1 + r2$.

Lemma *Rplus_le_le_0_compat* : $\forall\ r1\ r2, 0 \leq r1 \rightarrow 0 \leq r2 \rightarrow 0 \leq r1 + r2$.

Lemma *plus_le_is_le* : $\forall\ r1\ r2\ r3, 0 \leq r2 \rightarrow r1 + r2 \leq r3 \rightarrow r1 \leq r3$.

Lemma *plus_lt_is_lt* : $\forall\ r1\ r2\ r3, 0 \leq r2 \rightarrow r1 + r2 < r3 \rightarrow r1 < r3$.

Lemma *Rmult_le_0_lt_compat* :
$\forall\ r1\ r2\ r3\ r4$,
$0 \leq r1 \rightarrow 0 \leq r3 \rightarrow r1 < r2 \rightarrow r3 < r4 \rightarrow r1 \times r3 < r2 \times r4$.

Lemma *le_epsilon* :
$\forall\ r1\ r2, (\forall\ eps{:}R, 0 < eps \rightarrow r1 \leq r2 + eps) \rightarrow r1 \leq r2$.

Lemma *completeness_weak* :
$\forall\ E{:}R \rightarrow$ Prop,
$bound\ E \rightarrow (\exists\ x : R, E\ x) \rightarrow \exists\ m : R, is\_lub\ E\ m$.

Chapter 121

Module Coq.Reals.Rlimit

Definition of the limit
Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Classical_Prop*.
Require Import *Fourier*. *Open* Local *Scope R_scope*.

121.1 Calculus

Lemma *eps2_Rgt_R0* : $\forall\ eps{:}R,\ eps > 0 \rightarrow eps \times / 2 > 0$.

Lemma *eps2* : $\forall\ eps{:}R,\ eps \times / 2 + eps \times / 2 = eps$.

Lemma *eps4* : $\forall\ eps{:}R,\ eps \times / (2 + 2) + eps \times / (2 + 2) = eps \times / 2$.

Lemma *Rlt_eps2_eps* : $\forall\ eps{:}R,\ eps > 0 \rightarrow eps \times / 2 < eps$.

Lemma *Rlt_eps4_eps* : $\forall\ eps{:}R,\ eps > 0 \rightarrow eps \times / (2 + 2) < eps$.

Lemma *prop_eps* : $\forall\ r{:}R,\ (\forall\ eps{:}R,\ eps > 0 \rightarrow r < eps) \rightarrow r \leq 0$.

Definition *mul_factor* $(l\ l'{:}R) := / (1 + (Rabs\ l + Rabs\ l'))$.

Lemma *mul_factor_wd* : $\forall\ l\ l'{:}R,\ 1 + (Rabs\ l + Rabs\ l') \neq 0$.

Lemma *mul_factor_gt* : $\forall\ eps\ l\ l'{:}R,\ eps > 0 \rightarrow eps \times mul\_factor\ l\ l' > 0$.

Lemma *mul_factor_gt_f* :
$\forall\ eps\ l\ l'{:}R,\ eps > 0 \rightarrow Rmin\ 1\ (eps \times mul\_factor\ l\ l') > 0$.

121.2 Metric space

Record *Metric_Space* : Type :=
{*Base* : Type;
dist : $Base \rightarrow Base \rightarrow R$;
dist_pos : $\forall\ x\ y{:}Base,\ dist\ x\ y \geq 0$;
dist_sym : $\forall\ x\ y{:}Base,\ dist\ x\ y = dist\ y\ x$;

dist_refl : $\forall\ x\ y$:*Base*, *dist* $x\ y = 0 \leftrightarrow x = y$;
dist_tri : $\forall\ x\ y\ z$:*Base*, *dist* $x\ y \leq$ *dist* $x\ z +$ *dist* $z\ y\}$.

121.2.1 Limit in Metric space

Definition *limit_in* (*X X'*:*Metric_Space*) (*f*:*Base X* → *Base X'*)
(*D*:*Base X* → Prop) (*x0*:*Base X*) (*l*:*Base X'*) :=
∀ *eps*:*R*,
eps > 0 →
∃ *alp* : *R*,
alp > 0 ∧
(∀ *x*:*Base X*, *D x* ∧ *dist X x x0* < *alp* → *dist X' (f x) l* < *eps*).

121.2.2 R is a metric space

Definition *R_met* : *Metric_Space* :=
Build_Metric_Space R R_dist R_dist_pos R_dist_sym R_dist_refl R_dist_tri.

121.3 Limit 1 arg

Definition *Dgf* (*Df Dg*:*R* → Prop) (*f*:*R* → *R*) (*x*:*R*) := *Df x* ∧ *Dg* (*f x*).

Definition *limit1_in* (*f*:*R* → *R*) (*D*:*R* → Prop) (*l x0*:*R*) : Prop :=
limit_in R_met R_met f D x0 l.

Lemma *tech_limit* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*l x0*:*R*),
D x0 → *limit1_in f D l x0* → *l* = *f x0*.

Lemma *tech_limit_contr* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*l x0*:*R*),
D x0 → *l* ≠ *f x0* → ¬ *limit1_in f D l x0*.

Lemma *lim_x* : ∀ (*D*:*R* → Prop) (*x0*:*R*), *limit1_in* (fun *x*:*R* ⇒ *x*) *D x0 x0*.

Lemma *limit_plus* :
∀ (*f g*:*R* → *R*) (*D*:*R* → Prop) (*l l' x0*:*R*),
limit1_in f D l x0 →
limit1_in g D l' x0 → *limit1_in* (fun *x*:*R* ⇒ *f x* + *g x*) *D* (*l* + *l'*) *x0*.

Lemma *limit_Ropp* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*l x0*:*R*),
limit1_in f D l x0 → *limit1_in* (fun *x*:*R* ⇒ - *f x*) *D* (- *l*) *x0*.

Lemma *limit_minus* :
∀ (*f g*:*R* → *R*) (*D*:*R* → Prop) (*l l' x0*:*R*),
limit1_in f D l x0 →
limit1_in g D l' x0 → *limit1_in* (fun *x*:*R* ⇒ *f x* - *g x*) *D* (*l* - *l'*) *x0*.

Lemma *limit_free* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*x x0*:*R*),
limit1_in (fun *h*:*R* ⇒ *f x*) *D* (*f x*) *x0*.

Lemma *limit_mul* :
∀ (*f g*:*R* → *R*) (*D*:*R* → Prop) (*l l'x0*:*R*),
limit1_in f D l x0 →
limit1_in g D l' x0 → *limit1_in* (fun *x*:*R* ⇒ *f x* × *g x*) *D* (*l* × *l'*) *x0*.

Definition *adhDa* (*D*:*R* → Prop) (*a*:*R*) : Prop :=
∀ *alp*:*R*, *alp* > 0 → ∃ *x* : *R*, *D x* ∧ *R_dist x a* < *alp*.

Lemma *single_limit* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*l l'x0*:*R*),
adhDa D x0 → *limit1_in f D l x0* → *limit1_in f D l' x0* → *l* = *l'*.

Lemma *limit_comp* :
∀ (*f g*:*R* → *R*) (*Df Dg*:*R* → Prop) (*l l'x0*:*R*),
limit1_in f Df l x0 →
limit1_in g Dg l' l → *limit1_in* (fun *x*:*R* ⇒ *g* (*f x*)) (*Dgf Df Dg f*) *l' x0*.

Lemma *limit_inv* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop) (*l x0*:*R*),
limit1_in f D l x0 → *l* ≠ 0 → *limit1_in* (fun *x*:*R* ⇒ / *f x*) *D* (/ *l*) *x0*.

Chapter 122

Module Coq.Reals.RList

```
Require Import Rbase.
Require Import Rfunctions.
Open Local Scope R_scope.

Inductive Rlist : Type :=
    | nil : Rlist
    | cons : R → Rlist → Rlist.

Fixpoint In (x:R) (l:Rlist) {struct l} : Prop :=
  match l with
    | nil ⇒ False
    | cons a l' ⇒ x = a ∨ In x l'
  end.

Fixpoint Rlength (l:Rlist) : nat :=
  match l with
    | nil ⇒ 0%nat
    | cons a l' ⇒ S (Rlength l')
  end.

Fixpoint MaxRlist (l:Rlist) : R :=
  match l with
    | nil ⇒ 0
    | cons a l1 ⇒
      match l1 with
        | nil ⇒ a
        | cons a' l2 ⇒ Rmax a (MaxRlist l1)
      end
  end.

Fixpoint MinRlist (l:Rlist) : R :=
  match l with
    | nil ⇒ 1
    | cons a l1 ⇒
      match l1 with
```

```
        | nil ⇒ a
        | cons a' l2 ⇒ Rmin a (MinRlist l1)
      end
  end.

Lemma MaxRlist_P1 : ∀ (l:Rlist) (x:R), In x l → x ≤ MaxRlist l.

Fixpoint AbsList (l:Rlist) (x:R) {struct l} : Rlist :=
  match l with
    | nil ⇒ nil
    | cons a l' ⇒ cons (Rabs (a - x) / 2) (AbsList l' x)
  end.

Lemma MinRlist_P1 : ∀ (l:Rlist) (x:R), In x l → MinRlist l ≤ x.

Lemma AbsList_P1 :
  ∀ (l:Rlist) (x y:R), In y l → In (Rabs (y - x) / 2) (AbsList l x).

Lemma MinRlist_P2 :
  ∀ l:Rlist, (∀ y:R, In y l → 0 < y) → 0 < MinRlist l.

Lemma AbsList_P2 :
  ∀ (l:Rlist) (x y:R),
    In y (AbsList l x) → ∃ z : R, In z l ∧ y = Rabs (z - x) / 2.

Lemma MaxRlist_P2 :
  ∀ l:Rlist, (∃ y : R, In y l) → In (MaxRlist l) l.

Fixpoint pos_Rl (l:Rlist) (i:nat) {struct l} : R :=
  match l with
    | nil ⇒ 0
    | cons a l' ⇒ match i with
                    | O ⇒ a
                    | S i' ⇒ pos_Rl l' i'
                  end
  end.

Lemma pos_Rl_P1 :
  ∀ (l:Rlist) (a:R),
    (0 < Rlength l)%nat →
    pos_Rl (cons a l) (Rlength l) = pos_Rl l (pred (Rlength l)).

Lemma pos_Rl_P2 :
  ∀ (l:Rlist) (x:R),
    In x l ↔ (∃ i : nat, (i < Rlength l)%nat ∧ x = pos_Rl l i).

Lemma Rlist_P1 :
  ∀ (l:Rlist) (P:R → R → Prop),
    (∀ x:R, In x l → ∃ y : R, P x y) →
    ∃ l' : Rlist,
      Rlength l = Rlength l' ∧
      (∀ i:nat, (i < Rlength l)%nat → P (pos_Rl l i) (pos_Rl l' i)).
```

```
Definition ordered_Rlist (l:Rlist) : Prop :=
  ∀ i:nat, (i < pred (Rlength l))%nat → pos_Rl l i ≤ pos_Rl l (S i).

Fixpoint insert (l:Rlist) (x:R) {struct l} : Rlist :=
  match l with
    | nil ⇒ cons x nil
    | cons a l' ⇒
      match Rle_dec a x with
        | left _ ⇒ cons a (insert l' x)
        | right _ ⇒ cons x l
      end
  end.

Fixpoint cons_Rlist (l k:Rlist) {struct l} : Rlist :=
  match l with
    | nil ⇒ k
    | cons a l' ⇒ cons a (cons_Rlist l' k)
  end.

Fixpoint cons_ORlist (k l:Rlist) {struct k} : Rlist :=
  match k with
    | nil ⇒ l
    | cons a k' ⇒ cons_ORlist k' (insert l a)
  end.

Fixpoint app_Rlist (l:Rlist) (f:R → R) {struct l} : Rlist :=
  match l with
    | nil ⇒ nil
    | cons a l' ⇒ cons (f a) (app_Rlist l' f)
  end.

Fixpoint mid_Rlist (l:Rlist) (x:R) {struct l} : Rlist :=
  match l with
    | nil ⇒ nil
    | cons a l' ⇒ cons ((x + a) / 2) (mid_Rlist l' a)
  end.

Definition Rtail (l:Rlist) : Rlist :=
  match l with
    | nil ⇒ nil
    | cons a l' ⇒ l'
  end.

Definition FF (l:Rlist) (f:R → R) : Rlist :=
  match l with
    | nil ⇒ nil
    | cons a l' ⇒ app_Rlist (mid_Rlist l' a) f
  end.

Lemma RList_P0 :
```

$\forall$ (l:*Rlist*) (a:R),
pos_Rl (*insert l a*) 0 = $a \vee$ *pos_Rl* (*insert l a*) 0 = *pos_Rl l* 0.

Lemma *RList_P1* :
$\forall$ (l:*Rlist*) (a:R), *ordered_Rlist l* $\rightarrow$ *ordered_Rlist* (*insert l a*).

Lemma *RList_P2* :
$\forall$ *l1 l2*:*Rlist*, *ordered_Rlist l2* $\rightarrow$ *ordered_Rlist* (*cons_ORlist l1 l2*).

Lemma *RList_P3* :
$\forall$ (l:*Rlist*) (x:R),
In x l $\leftrightarrow$ ($\exists$ i : *nat*, x = *pos_Rl l i* $\wedge$ (i < *Rlength l*)%*nat*).

Lemma *RList_P4* :
$\forall$ (*l1*:*Rlist*) (a:R), *ordered_Rlist* (*cons a l1*) $\rightarrow$ *ordered_Rlist l1*.

Lemma *RList_P5* :
$\forall$ (l:*Rlist*) (x:R), *ordered_Rlist l* $\rightarrow$ *In x l* $\rightarrow$ *pos_Rl l* 0 $\leq$ x.

Lemma *RList_P6* :
$\forall$ l:*Rlist*,
ordered_Rlist l $\leftrightarrow$
($\forall$ i j:*nat*,
($i \leq j$)%*nat* $\rightarrow$ (j < *Rlength l*)%*nat* $\rightarrow$ *pos_Rl l i* $\leq$ *pos_Rl l j*).

Lemma *RList_P7* :
$\forall$ (l:*Rlist*) (x:R),
ordered_Rlist l $\rightarrow$ *In x l* $\rightarrow$ $x \leq$ *pos_Rl l* (*pred* (*Rlength l*)).

Lemma *RList_P8* :
$\forall$ (l:*Rlist*) (a x:R), *In x* (*insert l a*) $\leftrightarrow$ x = $a \vee$ *In x l*.

Lemma *RList_P9* :
$\forall$ (*l1 l2*:*Rlist*) (x:R), *In x* (*cons_ORlist l1 l2*) $\leftrightarrow$ *In x l1* $\vee$ *In x l2*.

Lemma *RList_P10* :
$\forall$ (l:*Rlist*) (a:R), *Rlength* (*insert l a*) = *S* (*Rlength l*).

Lemma *RList_P11* :
$\forall$ *l1 l2*:*Rlist*,
Rlength (*cons_ORlist l1 l2*) = (*Rlength l1* + *Rlength l2*)%*nat*.

Lemma *RList_P12* :
$\forall$ (l:*Rlist*) (i:*nat*) (f:$R \rightarrow R$),
(i < *Rlength l*)%*nat* $\rightarrow$ *pos_Rl* (*app_Rlist l f*) *i* = *f* (*pos_Rl l i*).

Lemma *RList_P13* :
$\forall$ (l:*Rlist*) (i:*nat*) (a:R),
(i < *pred* (*Rlength l*))%*nat* $\rightarrow$
pos_Rl (*mid_Rlist l a*) (*S i*) = (*pos_Rl l i* + *pos_Rl l* (*S i*)) / 2.

Lemma *RList_P14* : $\forall$ (l:*Rlist*) (a:R), *Rlength* (*mid_Rlist l a*) = *Rlength l*.

Lemma *RList_P15* :

$\forall$ *l1 l2*:*Rlist*,
ordered_Rlist l1 $\rightarrow$
ordered_Rlist l2 $\rightarrow$
pos_Rl l1 0 = *pos_Rl l2* 0 $\rightarrow$ *pos_Rl* (*cons_ORlist l1 l2*) 0 = *pos_Rl l1* 0.

Lemma *RList_P16* :
$\forall$ *l1 l2*:*Rlist*,
ordered_Rlist l1 $\rightarrow$
ordered_Rlist l2 $\rightarrow$
pos_Rl l1 (*pred* (*Rlength l1*)) = *pos_Rl l2* (*pred* (*Rlength l2*)) $\rightarrow$
pos_Rl (*cons_ORlist l1 l2*) (*pred* (*Rlength* (*cons_ORlist l1 l2*))) =
pos_Rl l1 (*pred* (*Rlength l1*)).

Lemma *RList_P17* :
$\forall$ (*l1*:*Rlist*) (*x*:*R*) (*i*:*nat*),
ordered_Rlist l1 $\rightarrow$
In x l1 $\rightarrow$
pos_Rl l1 i < *x* $\rightarrow$ (*i* < *pred* (*Rlength l1*))%*nat* $\rightarrow$ *pos_Rl l1* (*S i*) $\leq$ *x*.

Lemma *RList_P18* :
$\forall$ (*l*:*Rlist*) (*f*:*R* $\rightarrow$ *R*), *Rlength* (*app_Rlist l f*) = *Rlength l*.

Lemma *RList_P19* :
$\forall$ *l*:*Rlist*,
l $\neq$ *nil* $\rightarrow$ $\exists$ *r* : *R*, ($\exists$ *r0* : *Rlist*, *l* = *cons r r0*).

Lemma *RList_P20* :
$\forall$ *l*:*Rlist*,
(2 $\leq$ *Rlength l*)%*nat* $\rightarrow$
$\exists$ *r* : *R*,
($\exists$ *r1* : *R*, ($\exists$ *l'* : *Rlist*, *l* = *cons r* (*cons r1 l'*))).

Lemma *RList_P21* : $\forall$ *l l'*:*Rlist*, *l* = *l'* $\rightarrow$ *Rtail l* = *Rtail l'*.

Lemma *RList_P22* :
$\forall$ *l1 l2*:*Rlist*, *l1* $\neq$ *nil* $\rightarrow$ *pos_Rl* (*cons_Rlist l1 l2*) 0 = *pos_Rl l1* 0.

Lemma *RList_P23* :
$\forall$ *l1 l2*:*Rlist*,
Rlength (*cons_Rlist l1 l2*) = (*Rlength l1* + *Rlength l2*)%*nat*.

Lemma *RList_P24* :
$\forall$ *l1 l2*:*Rlist*,
l2 $\neq$ *nil* $\rightarrow$
pos_Rl (*cons_Rlist l1 l2*) (*pred* (*Rlength* (*cons_Rlist l1 l2*))) =
pos_Rl l2 (*pred* (*Rlength l2*)).

Lemma *RList_P25* :
$\forall$ *l1 l2*:*Rlist*,
ordered_Rlist l1 $\rightarrow$
ordered_Rlist l2 $\rightarrow$

pos_Rl l1 (pred (Rlength l1)) ≤ pos_Rl l2 0 →
ordered_Rlist (cons_Rlist l1 l2).

Lemma *RList_P26* :
∀ *(l1 l2:Rlist) (i:nat),*
(i < Rlength l1)%nat → pos_Rl (cons_Rlist l1 l2) i = pos_Rl l1 i.

Lemma *RList_P27* :
∀ *l1 l2 l3:Rlist,*
cons_Rlist l1 (cons_Rlist l2 l3) = cons_Rlist (cons_Rlist l1 l2) l3.

Lemma *RList_P28* : ∀ *l:Rlist, cons_Rlist l nil = l.*

Lemma *RList_P29* :
∀ *(l2 l1:Rlist) (i:nat),*
(Rlength l1 ≤ i)%nat →
(i < Rlength (cons_Rlist l1 l2))%nat →
pos_Rl (cons_Rlist l1 l2) i = pos_Rl l2 (i - Rlength l1).

Chapter 123

Module Coq.Reals.Rpow_def

```
Require Import Rdefinitions.

Fixpoint pow (r:R) (n:nat) {struct n} : R :=
  match n with
    | O ⇒ R1
    | S n ⇒ Rmult r (pow r n)
  end.
```

Chapter 124

Module Coq.Reals.Rpower

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo*.
Require Import *Ranalysis1*.
Require Import *Exp_prop*.
Require Import *Rsqrt_def*.
Require Import *R_sqrt*.
Require Import *MVT*.
Require Import *Ranalysis4*. *Open* Local *Scope* *R_scope*.

Lemma *P_Rmin* : $\forall$ (*P*:*R* $\rightarrow$ Prop) (*x* *y*:*R*), *P* *x* $\rightarrow$ *P* *y* $\rightarrow$ *P* (*Rmin* *x* *y*).

Lemma *exp_le_3* : *exp* 1 $\leq$ 3.

124.1 Properties of Exp

Theorem *exp_increasing* : $\forall$ *x* *y*:*R*, *x* < *y* $\rightarrow$ *exp* *x* < *exp* *y*.

Theorem *exp_lt_inv* : $\forall$ *x* *y*:*R*, *exp* *x* < *exp* *y* $\rightarrow$ *x* < *y*.

Lemma *exp_ineq1* : $\forall$ *x*:*R*, 0 < *x* $\rightarrow$ 1 + *x* < *exp* *x*.

Lemma *ln_exists1* : $\forall$ *y*:*R*, 0 < *y* $\rightarrow$ 1 $\leq$ *y* $\rightarrow$ *sigT* (fun *z*:*R* $\Rightarrow$ *y* = *exp* *z*).

Lemma *ln_exists* : $\forall$ *y*:*R*, 0 < *y* $\rightarrow$ *sigT* (fun *z*:*R* $\Rightarrow$ *y* = *exp* *z*).

```
Definition Rln (y:posreal) : R :=
  match ln_exists (pos y) (cond_pos y) with
    | existT a b ⇒ a
  end.
```

```
Definition ln (x:R) : R :=
  match Rlt_dec 0 x with
    | left a ⇒ Rln (mkposreal x a)
    | right a ⇒ 0
```

end.

Lemma $exp\_ln : \forall\ x{:}R,\ 0 < x \rightarrow exp\ (ln\ x) = x.$

Theorem $exp\_inv : \forall\ x\ y{:}R,\ exp\ x = exp\ y \rightarrow x = y.$

Theorem $exp\_Ropp : \forall\ x{:}R,\ exp\ (-\ x) = /\ exp\ x.$

124.2 Properties of Ln

Theorem $ln\_increasing : \forall\ x\ y{:}R,\ 0 < x \rightarrow x < y \rightarrow ln\ x < ln\ y.$

Theorem $ln\_exp : \forall\ x{:}R,\ ln\ (exp\ x) = x.$

Theorem $ln\_1 : ln\ 1 = 0.$

Theorem $ln\_lt\_inv : \forall\ x\ y{:}R,\ 0 < x \rightarrow 0 < y \rightarrow ln\ x < ln\ y \rightarrow x < y.$

Theorem $ln\_inv : \forall\ x\ y{:}R,\ 0 < x \rightarrow 0 < y \rightarrow ln\ x = ln\ y \rightarrow x = y.$

Theorem $ln\_mult : \forall\ x\ y{:}R,\ 0 < x \rightarrow 0 < y \rightarrow ln\ (x \times y) = ln\ x + ln\ y.$

Theorem $ln\_Rinv : \forall\ x{:}R,\ 0 < x \rightarrow ln\ (/\ x) = -\ ln\ x.$

Theorem $ln\_continue$:
$\forall\ y{:}R,\ 0 < y \rightarrow continue\_in\ ln\ (\mathsf{fun}\ x{:}R \Rightarrow 0 < x)\ y.$

124.3 Definition of Rpower

Definition $Rpower\ (x\ y{:}R) := exp\ (y \times ln\ x).$

Infix Local "$\hat{}R$" := $Rpower\ (at\ level\ 30,\ right\ associativity) : R\_scope.$

124.4 Properties of Rpower

Theorem $Rpower\_plus : \forall\ x\ y\ z{:}R,\ z\ \hat{}R\ (x + y) = z\ \hat{}R\ x \times z\ \hat{}R\ y.$

Theorem $Rpower\_mult : \forall\ x\ y\ z{:}R,\ (x\ \hat{}R\ y)\ \hat{}R\ z = x\ \hat{}R\ (y \times z).$

Theorem $Rpower\_O : \forall\ x{:}R,\ 0 < x \rightarrow x\ \hat{}R\ 0 = 1.$

Theorem $Rpower\_1 : \forall\ x{:}R,\ 0 < x \rightarrow x\ \hat{}R\ 1 = x.$

Theorem $Rpower\_pow : \forall\ (n{:}nat)\ (x{:}R),\ 0 < x \rightarrow x\ \hat{}R\ INR\ n = x\ \hat{}\ n.$

Theorem $Rpower\_lt$:
$\forall\ x\ y\ z{:}R,\ 1 < x \rightarrow 0 \leq y \rightarrow y < z \rightarrow x\ \hat{}R\ y < x\ \hat{}R\ z.$

Theorem $Rpower\_sqrt : \forall\ x{:}R,\ 0 < x \rightarrow x\ \hat{}R\ (/\ 2) = sqrt\ x.$

Theorem $Rpower\_Ropp : \forall\ x\ y{:}R,\ x\ \hat{}R\ (-\ y) = /\ x\ \hat{}R\ y.$

Theorem Rle_Rpower :
$\forall\ e\ n\ m{:}R,\ 1 < e \rightarrow 0 \leq n \rightarrow n \leq m \rightarrow e\ \hat{}R\ n \leq e\ \hat{}R\ m.$

Theorem $ln\_lt\_2 : /\ 2 < ln\ 2.$

124.5 Differentiability of Ln and Rpower

Theorem *limit1_ext* :
∀ (*f g*:*R* → *R*) (*D*:*R* → Prop) (*l x*:*R*),
(∀ *x*:*R*, *D x* → *f x* = *g x*) → *limit1_in f D l x* → *limit1_in g D l x*.

Theorem *limit1_imp* :
∀ (*f*:*R* → *R*) (*D D1*:*R* → Prop) (*l x*:*R*),
(∀ *x*:*R*, *D1 x* → *D x*) → *limit1_in f D l x* → *limit1_in f D1 l x*.

Theorem *Rinv_Rdiv* : ∀ *x y*:*R*, *x* ≠ 0 → *y* ≠ 0 → / (*x* / *y*) = *y* / *x*.

Theorem *Dln* : ∀ *y*:*R*, 0 < *y* → *D_in ln Rinv* (fun *x*:*R* ⇒ 0 < *x*) *y*.

Lemma *derivable_pt_lim_ln* : ∀ *x*:*R*, 0 < *x* → *derivable_pt_lim ln x* (/ *x*).

Theorem *D_in_imp* :
∀ (*f g*:*R* → *R*) (*D D1*:*R* → Prop) (*x*:*R*),
(∀ *x*:*R*, *D1 x* → *D x*) → *D_in f g D x* → *D_in f g D1 x*.

Theorem *D_in_ext* :
∀ (*f g h*:*R* → *R*) (*D*:*R* → Prop) (*x*:*R*),
f x = *g x* → *D_in h f D x* → *D_in h g D x*.

Theorem *Dpower* :
∀ *y z*:*R*,
0 < *y* →
D_in (fun *x*:*R* ⇒ *x* ^*R z*) (fun *x*:*R* ⇒ *z* × *x* ^*R* (*z* - 1)) (
fun *x*:*R* ⇒ 0 < *x*) *y*.

Theorem *derivable_pt_lim_power* :
∀ *x y*:*R*,
0 < *x* → *derivable_pt_lim* (fun *x* ⇒ *x* ^*R y*) *x* (*y* × *x* ^*R* (*y* - 1)).

Chapter 125

Module Coq.Reals.Rprod

Require Import *Compare*.
Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *PartSum*.
Require Import *Binomial*.
Open Local *Scope R_scope*.

TT Ak; 1<=k<=N

Lemma *prod_SO_split* :
  $\forall$ (*An*:*nat* $\to$ *R*) (*n k*:*nat*),
    (*k* $\leq$ *n*)%*nat* $\to$
    prod_f_SO An n =
    prod_f_SO An k $\times$ *prod_f_SO* (fun *l*:*nat* $\Rightarrow$ *An* (*k* + *l*)%*nat*) (*n* - *k*).

Lemma *prod_SO_pos* :
  $\forall$ (*An*:*nat* $\to$ *R*) (*N*:*nat*),
    ($\forall$ *n*:*nat*, (*n* $\leq$ *N*)%*nat* $\to$ 0 $\leq$ *An n*) $\to$ 0 $\leq$ *prod_f_SO An N*.

Lemma *prod_SO_Rle* :
  $\forall$ (*An Bn*:*nat* $\to$ *R*) (*N*:*nat*),
    ($\forall$ *n*:*nat*, (*n* $\leq$ *N*)%*nat* $\to$ 0 $\leq$ *An n* $\leq$ *Bn n*) $\to$
    prod_f_SO An N $\leq$ *prod_f_SO Bn N*.

Application to factorial
Lemma *fact_prodSO* :
  $\forall$ *n*:*nat*, *INR* (*fact n*) = *prod_f_SO* (fun *k*:*nat* $\Rightarrow$ *INR k*) *n*.

Lemma *le_n_2n* : $\forall$ *n*:*nat*, (*n* $\leq$ 2 $\times$ *n*)%*nat*.

We prove that (N!)^2<=(2N-k)!*k! forall k in |*O*;2*N*|
Lemma *RfactN_fact2N_factk* :
  $\forall$ *N k*:*nat*,
    (*k* $\leq$ 2 $\times$ *N*)%*nat* $\to$
    Rsqr (*INR* (*fact N*)) $\leq$ *INR* (*fact* (2 $\times$ *N* - *k*)) $\times$ *INR* (*fact k*).

Lemma $INR\_fact\_lt\_0 : \forall\ n{:}nat,\ 0 < INR\ (fact\ n)$.

We have the following inequality : (C 2N k) <= (C 2N N) forall k in $|O;2N|$
Lemma $C\_maj : \forall\ N\ k{:}nat,\ (k \leq 2 \times N)\%nat \rightarrow C\ (2 \times N)\ k \leq C\ (2 \times N)\ N$.

Chapter 126

Module Coq.Reals.Rseries

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Classical*.
Require Import *Compare*.
Open Local *Scope R_scope*.

Implicit Type $r : R$.

126.1 Definition of sequence and properties

Section *sequence*.

Variable $Un : nat \to R$.

Definition $EUn\ r$: Prop := $\exists\ i : nat, r = Un\ i$.

Definition $Un\_cv\ (l{:}R)$: Prop :=
$\forall\ eps{:}R,$
$eps > 0 \to$
$\exists\ N : nat, (\forall\ n{:}nat, (n \geq N)\%nat \to R\_dist\ (Un\ n)\ l < eps)$.

Definition *Cauchy_crit* : Prop :=
$\forall\ eps{:}R,$
$eps > 0 \to$
$\exists\ N : nat,$
$(\forall\ n\ m{:}nat,$
$(n \geq N)\%nat \to (m \geq N)\%nat \to R\_dist\ (Un\ n)\ (Un\ m) < eps)$.

Definition *Un_growing* : Prop := $\forall\ n{:}nat, Un\ n \leq Un\ (S\ n)$.

Lemma *EUn_noempty* : $\exists\ r : R, EUn\ r$.

Lemma *Un_in_EUn* : $\forall\ n{:}nat, EUn\ (Un\ n)$.

Lemma *Un_bound_imp* :
$\forall\ x{:}R, (\forall\ n{:}nat, Un\ n \leq x) \to is\_upper\_bound\ EUn\ x$.

Lemma *growing_prop* :
$\forall\ n\ m{:}nat,\ Un\_growing \rightarrow (n \geq m)\%nat \rightarrow Un\ n \geq Un\ m.$

classical is needed: *not_all_not_ex*

Lemma *Un_cv_crit* : $Un\_growing \rightarrow bound\ EUn \rightarrow \exists\ l : R,\ Un\_cv\ l.$

Lemma *finite_greater* :
$\forall\ N{:}nat, \exists\ M : R, (\forall\ n{:}nat, (n \leq N)\%nat \rightarrow Un\ n \leq M).$

Lemma *cauchy_bound* : $Cauchy\_crit \rightarrow bound\ EUn.$

End *sequence*.

126.2 Definition of Power Series and properties

Section *Isequence*.

Variable $An : nat \rightarrow R.$

Definition $Pser\ (x\ l{:}R)$: Prop := $infinit\_sum\ (\text{fun}\ n{:}nat \Rightarrow An\ n \times x\ \hat{}\ n)\ l.$

End *Isequence*.

Lemma *GP_infinite* :
$\forall\ x{:}R,\ Rabs\ x < 1 \rightarrow Pser\ (\text{fun}\ n{:}nat \Rightarrow 1)\ x\ (/\ (1 - x)).$

Chapter 127

Module Coq.Reals.Rsigma

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rseries*.
Require Import *PartSum*.
Open Local *Scope R_scope*.

Section *Sigma*.

Variable $f : nat \to R$.

Definition $sigma\ (low\ high{:}nat) : R :=$
$sum\_f\_R0\ (\text{fun}\ k{:}nat \Rightarrow f\ (low + k))\ (high - low)$.

Theorem $sigma\_split$:
$\forall\ low\ high\ k{:}nat,$
$(low \leq k)\%nat \to$
$(k < high)\%nat \to sigma\ low\ high = sigma\ low\ k + sigma\ (S\ k)\ high$.

Theorem $sigma\_diff$:
$\forall\ low\ high\ k{:}nat,$
$(low \leq k)\%nat \to$
$(k < high)\%nat \to sigma\ low\ high - sigma\ low\ k = sigma\ (S\ k)\ high$.

Theorem $sigma\_diff\_neg$:
$\forall\ low\ high\ k{:}nat,$
$(low \leq k)\%nat \to$
$(k < high)\%nat \to sigma\ low\ k - sigma\ low\ high = - sigma\ (S\ k)\ high$.

Theorem $sigma\_first$:
$\forall\ low\ high{:}nat,$
$(low < high)\%nat \to sigma\ low\ high = f\ low + sigma\ (S\ low)\ high$.

Theorem $sigma\_last$:
$\forall\ low\ high{:}nat,$
$(low < high)\%nat \to sigma\ low\ high = f\ high + sigma\ low\ (pred\ high)$.

Theorem $sigma\_eq\_arg : \forall\ low{:}nat, sigma\ low\ low = f\ low$.

End *Sigma*.

Chapter 128

Module Coq.Reals.Rsqrt_def

Require Import *Sumbool*.
Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Ranalysis1*.
Open Local *Scope R_scope*.

Definition *dicho_lb* $(x\ y{:}R)\ (P{:}R \to bool)\ (N{:}nat) : R := Dichotomy\_lb\ x\ y\ P\ N$.
Definition *dicho_up* $(x\ y{:}R)\ (P{:}R \to bool)\ (N{:}nat) : R := Dichotomy\_ub\ x\ y\ P\ N$.

Lemma *dicho_comp* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool)\ (n{:}nat)$,
$x \leq y \to dicho\_lb\ x\ y\ P\ n \leq dicho\_up\ x\ y\ P\ n$.

Lemma *dicho_lb_growing* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to Un\_growing\ (dicho\_lb\ x\ y\ P)$.

Lemma *dicho_up_decreasing* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to Un\_decreasing\ (dicho\_up\ x\ y\ P)$.

Lemma *dicho_lb_maj_y* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to \forall\ n{:}nat,\ dicho\_lb\ x\ y\ P\ n \leq y$.

Lemma *dicho_lb_maj* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to has\_ub\ (dicho\_lb\ x\ y\ P)$.

Lemma *dicho_up_min_x* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to \forall\ n{:}nat,\ x \leq dicho\_up\ x\ y\ P\ n$.

Lemma *dicho_up_min* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool),\ x \leq y \to has\_lb\ (dicho\_up\ x\ y\ P)$.

Lemma *dicho_lb_cv* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool)$,
$x \leq y \to sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (dicho\_lb\ x\ y\ P)\ l)$.

Lemma *dicho_up_cv* :
$\forall\ (x\ y{:}R)\ (P{:}R \to bool)$,

$x \leq y \rightarrow sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ (dicho\_up\ x\ y\ P)\ l).$

Lemma *dicho_lb_dicho_up* :
$\forall\ (x\ y{:}R)\ (P{:}R \rightarrow bool)\ (n{:}nat),$
$x \leq y \rightarrow dicho\_up\ x\ y\ P\ n$ - $dicho\_lb\ x\ y\ P\ n = (y$ - $x)\ /\ 2\ \hat{}\ n.$

Definition *pow_2_n* $(n{:}nat) := 2\ \hat{}\ n.$

Lemma *pow_2_n_neq_R0* : $\forall\ n{:}nat,\ pow\_2\_n\ n \neq 0.$

Lemma *pow_2_n_growing* : $Un\_growing\ pow\_2\_n.$

Lemma *pow_2_n_infty* : $cv\_infty\ pow\_2\_n.$

Lemma *cv_dicho* :
$\forall\ (x\ y\ l1\ l2{:}R)\ (P{:}R \rightarrow bool),$
$x \leq y \rightarrow$
$Un\_cv\ (dicho\_lb\ x\ y\ P)\ l1 \rightarrow Un\_cv\ (dicho\_up\ x\ y\ P)\ l2 \rightarrow l1 = l2.$

Definition *cond_positivity* $(x{:}R) : bool :=$
match *Rle_dec* $0\ x$ with
| *left* _ $\Rightarrow$ *true*
| *right* _ $\Rightarrow$ *false*
end.

Sequential caracterisation of continuity
Lemma *continuity_seq* :
$\forall\ (f{:}R \rightarrow R)\ (Un{:}nat \rightarrow R)\ (l{:}R),$
$continuity\_pt\ f\ l \rightarrow Un\_cv\ Un\ l \rightarrow Un\_cv\ (\mathsf{fun}\ i{:}nat \Rightarrow f\ (Un\ i))\ (f\ l).$

Lemma *dicho_lb_car* :
$\forall\ (x\ y{:}R)\ (P{:}R \rightarrow bool)\ (n{:}nat),$
$P\ x = false \rightarrow P\ (dicho\_lb\ x\ y\ P\ n) = false.$

Lemma *dicho_up_car* :
$\forall\ (x\ y{:}R)\ (P{:}R \rightarrow bool)\ (n{:}nat),$
$P\ y = true \rightarrow P\ (dicho\_up\ x\ y\ P\ n) = true.$

Intermediate Value Theorem
Lemma *IVT* :
$\forall\ (f{:}R \rightarrow R)\ (x\ y{:}R),$
$continuity\ f \rightarrow$
$x < y \rightarrow f\ x < 0 \rightarrow 0 < f\ y \rightarrow sigT\ (\mathsf{fun}\ z{:}R \Rightarrow x \leq z \leq y \wedge f\ z = 0).$

Lemma *IVT_cor* :
$\forall\ (f{:}R \rightarrow R)\ (x\ y{:}R),$
$continuity\ f \rightarrow$
$x \leq y \rightarrow f\ x \times f\ y \leq 0 \rightarrow sigT\ (\mathsf{fun}\ z{:}R \Rightarrow x \leq z \leq y \wedge f\ z = 0).$

We can now define the square root function as the reciprocal transformation of the square root function
Lemma *Rsqrt_exists* :
$\forall\ y{:}R,\ 0 \leq y \rightarrow sigT\ (\mathsf{fun}\ z{:}R \Rightarrow 0 \leq z \wedge y = Rsqr\ z).$

Definition $Rsqrt\ (y{:}nonnegreal) : R :=$
  match $Rsqrt\_exists\ (nonneg\ y)\ (cond\_nonneg\ y)$ with
    | $existT\ a\ b \Rightarrow a$
  end.

Lemma $Rsqrt\_positivity : \forall\ x{:}nonnegreal, 0 \leq Rsqrt\ x.$

Lemma $Rsqrt\_Rsqrt : \forall\ x{:}nonnegreal, Rsqrt\ x \times Rsqrt\ x = x.$

Chapter 129

Module Coq.Reals.R_sqrt

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Rsqrt_def*. *Open* Local *Scope R_scope*.

129.1 Continuous extension of Rsqrt on R

Definition $sqrt\ (x{:}R) : R :=$
  match $Rcase\_abs\ x$ with
    | $left\ \_ \Rightarrow 0$
    | $right\ a \Rightarrow Rsqrt\ (mknonnegreal\ x\ (Rge\_le\ \_\ \_\ a))$
  end.

Lemma $sqrt\_positivity : \forall\ x{:}R, 0 \leq x \rightarrow 0 \leq sqrt\ x.$

Lemma $sqrt\_sqrt : \forall\ x{:}R, 0 \leq x \rightarrow sqrt\ x \times sqrt\ x = x.$

Lemma $sqrt\_0 : sqrt\ 0 = 0.$

Lemma $sqrt\_1 : sqrt\ 1 = 1.$

Lemma $sqrt\_eq\_0 : \forall\ x{:}R, 0 \leq x \rightarrow sqrt\ x = 0 \rightarrow x = 0.$

Lemma $sqrt\_lem\_0 : \forall\ x\ y{:}R, 0 \leq x \rightarrow 0 \leq y \rightarrow sqrt\ x = y \rightarrow y \times y = x.$

Lemma $sqrt\_lem\_1 : \forall\ x\ y{:}R, 0 \leq x \rightarrow 0 \leq y \rightarrow y \times y = x \rightarrow sqrt\ x = y.$

Lemma $sqrt\_def : \forall\ x{:}R, 0 \leq x \rightarrow sqrt\ x \times sqrt\ x = x.$

Lemma $sqrt\_square : \forall\ x{:}R, 0 \leq x \rightarrow sqrt\ (x \times x) = x.$

Lemma $sqrt\_Rsqr : \forall\ x{:}R, 0 \leq x \rightarrow sqrt\ (Rsqr\ x) = x.$

Lemma $sqrt\_Rsqr\_abs : \forall\ x{:}R, sqrt\ (Rsqr\ x) = Rabs\ x.$

Lemma $Rsqr\_sqrt : \forall\ x{:}R, 0 \leq x \rightarrow Rsqr\ (sqrt\ x) = x.$

Lemma $sqrt\_mult :$
  $\forall\ x\ y{:}R, 0 \leq x \rightarrow 0 \leq y \rightarrow sqrt\ (x \times y) = sqrt\ x \times sqrt\ y.$

Lemma $sqrt\_lt\_R0 : \forall\ x{:}R, 0 < x \rightarrow 0 < sqrt\ x.$

Lemma $sqrt\_div$:
$\forall\ x\ y{:}R,\ 0 \leq x \to 0 < y \to sqrt\ (x\ /\ y) = sqrt\ x\ /\ sqrt\ y.$

Lemma $sqrt\_lt\_0 : \forall\ x\ y{:}R,\ 0 \leq x \to 0 \leq y \to sqrt\ x < sqrt\ y \to x < y.$

Lemma $sqrt\_lt\_1 : \forall\ x\ y{:}R,\ 0 \leq x \to 0 \leq y \to x < y \to sqrt\ x < sqrt\ y.$

Lemma $sqrt\_le\_0$:
$\forall\ x\ y{:}R,\ 0 \leq x \to 0 \leq y \to sqrt\ x \leq sqrt\ y \to x \leq y.$

Lemma $sqrt\_le\_1$:
$\forall\ x\ y{:}R,\ 0 \leq x \to 0 \leq y \to x \leq y \to sqrt\ x \leq sqrt\ y.$

Lemma $sqrt\_inj : \forall\ x\ y{:}R,\ 0 \leq x \to 0 \leq y \to sqrt\ x = sqrt\ y \to x = y.$

Lemma $sqrt\_less : \forall\ x{:}R,\ 0 \leq x \to 1 < x \to sqrt\ x < x.$

Lemma $sqrt\_more : \forall\ x{:}R,\ 0 < x \to x < 1 \to x < sqrt\ x.$

Lemma $sqrt\_cauchy$:
$\forall\ a\ b\ c\ d{:}R,$
$a \times c + b \times d \leq sqrt\ (Rsqr\ a + Rsqr\ b) \times sqrt\ (Rsqr\ c + Rsqr\ d).$

129.2 Resolution of $a \times X\hat{}2 + b \times X + c = 0$

Definition $Delta\ (a{:}nonzeroreal)\ (b\ c{:}R) : R := Rsqr\ b - 4 \times a \times c.$

Definition $Delta\_is\_pos\ (a{:}nonzeroreal)\ (b\ c{:}R)$: Prop $:= 0 \leq Delta\ a\ b\ c.$

Definition $sol\_x1\ (a{:}nonzeroreal)\ (b\ c{:}R) : R :=$
$(-\ b + sqrt\ (Delta\ a\ b\ c))\ /\ (2 \times a).$

Definition $sol\_x2\ (a{:}nonzeroreal)\ (b\ c{:}R) : R :=$
$(-\ b - sqrt\ (Delta\ a\ b\ c))\ /\ (2 \times a).$

Lemma $Rsqr\_sol\_eq\_0\_1$:
$\forall\ (a{:}nonzeroreal)\ (b\ c\ x{:}R),$
$Delta\_is\_pos\ a\ b\ c \to$
$x = sol\_x1\ a\ b\ c \vee x = sol\_x2\ a\ b\ c \to a \times Rsqr\ x + b \times x + c = 0.$

Lemma $Rsqr\_sol\_eq\_0\_0$:
$\forall\ (a{:}nonzeroreal)\ (b\ c\ x{:}R),$
$Delta\_is\_pos\ a\ b\ c \to$
$a \times Rsqr\ x + b \times x + c = 0 \to x = sol\_x1\ a\ b\ c \vee x = sol\_x2\ a\ b\ c.$

Chapter 130

Module Coq.Reals.R_sqr

Require Import *Rbase*.
Require Import *Rbasic_fun*. *Open* Local *Scope R_scope*.

Rsqr : some results
Ltac *ring_Rsqr* := *unfold Rsqr* in ⊢ ×; *ring*.

Lemma *Rsqr_neg* : ∀ *x*:*R*, *Rsqr x* = *Rsqr* (- *x*).

Lemma *Rsqr_mult* : ∀ *x y*:*R*, *Rsqr* (*x* × *y*) = *Rsqr x* × *Rsqr y*.

Lemma *Rsqr_plus* : ∀ *x y*:*R*, *Rsqr* (*x* + *y*) = *Rsqr x* + *Rsqr y* + 2 × *x* × *y*.

Lemma *Rsqr_minus* : ∀ *x y*:*R*, *Rsqr* (*x* - *y*) = *Rsqr x* + *Rsqr y* - 2 × *x* × *y*.

Lemma *Rsqr_neg_minus* : ∀ *x y*:*R*, *Rsqr* (*x* - *y*) = *Rsqr* (*y* - *x*).

Lemma *Rsqr_1* : *Rsqr* 1 = 1.

Lemma *Rsqr_gt_0_0* : ∀ *x*:*R*, 0 < *Rsqr x* → *x* ≠ 0.

Lemma *Rsqr_pos_lt* : ∀ *x*:*R*, *x* ≠ 0 → 0 < *Rsqr x*.

Lemma *Rsqr_div* : ∀ *x y*:*R*, *y* ≠ 0 → *Rsqr* (*x* / *y*) = *Rsqr x* / *Rsqr y*.

Lemma *Rsqr_eq_0* : ∀ *x*:*R*, *Rsqr x* = 0 → *x* = 0.

Lemma *Rsqr_minus_plus* : ∀ *a b*:*R*, (*a* - *b*) × (*a* + *b*) = *Rsqr a* - *Rsqr b*.

Lemma *Rsqr_plus_minus* : ∀ *a b*:*R*, (*a* + *b*) × (*a* - *b*) = *Rsqr a* - *Rsqr b*.

Lemma *Rsqr_incr_0* :
∀ *x y*:*R*, *Rsqr x* ≤ *Rsqr y* → 0 ≤ *x* → 0 ≤ *y* → *x* ≤ *y*.

Lemma *Rsqr_incr_0_var* : ∀ *x y*:*R*, *Rsqr x* ≤ *Rsqr y* → 0 ≤ *y* → *x* ≤ *y*.

Lemma *Rsqr_incr_1* :
∀ *x y*:*R*, *x* ≤ *y* → 0 ≤ *x* → 0 ≤ *y* → *Rsqr x* ≤ *Rsqr y*.

Lemma *Rsqr_incrst_0* :
∀ *x y*:*R*, *Rsqr x* < *Rsqr y* → 0 ≤ *x* → 0 ≤ *y* → *x* < *y*.

Lemma *Rsqr_incrst_1* :
∀ *x y*:*R*, *x* < *y* → 0 ≤ *x* → 0 ≤ *y* → *Rsqr x* < *Rsqr y*.

Lemma $Rsqr\_neg\_pos\_le\_0$:
$\forall\ x\ y{:}R,\ Rsqr\ x \leq Rsqr\ y \rightarrow 0 \leq y \rightarrow -\ y \leq x.$

Lemma $Rsqr\_neg\_pos\_le\_1$:
$\forall\ x\ y{:}R,\ -\ y \leq x \rightarrow x \leq y \rightarrow 0 \leq y \rightarrow Rsqr\ x \leq Rsqr\ y.$

Lemma $neg\_pos\_Rsqr\_le : \forall\ x\ y{:}R,\ -\ y \leq x \rightarrow x \leq y \rightarrow Rsqr\ x \leq Rsqr\ y.$

Lemma $Rsqr\_abs : \forall\ x{:}R,\ Rsqr\ x = Rsqr\ (Rabs\ x).$

Lemma $Rsqr\_le\_abs\_0 : \forall\ x\ y{:}R,\ Rsqr\ x \leq Rsqr\ y \rightarrow Rabs\ x \leq Rabs\ y.$

Lemma $Rsqr\_le\_abs\_1 : \forall\ x\ y{:}R,\ Rabs\ x \leq Rabs\ y \rightarrow Rsqr\ x \leq Rsqr\ y.$

Lemma $Rsqr\_lt\_abs\_0 : \forall\ x\ y{:}R,\ Rsqr\ x < Rsqr\ y \rightarrow Rabs\ x < Rabs\ y.$

Lemma $Rsqr\_lt\_abs\_1 : \forall\ x\ y{:}R,\ Rabs\ x < Rabs\ y \rightarrow Rsqr\ x < Rsqr\ y.$

Lemma $Rsqr\_inj : \forall\ x\ y{:}R,\ 0 \leq x \rightarrow 0 \leq y \rightarrow Rsqr\ x = Rsqr\ y \rightarrow x = y.$

Lemma $Rsqr\_eq\_abs\_0 : \forall\ x\ y{:}R,\ Rsqr\ x = Rsqr\ y \rightarrow Rabs\ x = Rabs\ y.$

Lemma $Rsqr\_eq\_asb\_1 : \forall\ x\ y{:}R,\ Rabs\ x = Rabs\ y \rightarrow Rsqr\ x = Rsqr\ y.$

Lemma $triangle\_rectangle$:
$\forall\ x\ y\ z{:}R,$
$0 \leq z \rightarrow Rsqr\ x + Rsqr\ y \leq Rsqr\ z \rightarrow -\ z \leq x \leq z \wedge -\ z \leq y \leq z.$

Lemma $triangle\_rectangle\_lt$:
$\forall\ x\ y\ z{:}R,$
$Rsqr\ x + Rsqr\ y < Rsqr\ z \rightarrow Rabs\ x < Rabs\ z \wedge Rabs\ y < Rabs\ z.$

Lemma $triangle\_rectangle\_le$:
$\forall\ x\ y\ z{:}R,$
$Rsqr\ x + Rsqr\ y \leq Rsqr\ z \rightarrow Rabs\ x \leq Rabs\ z \wedge Rabs\ y \leq Rabs\ z.$

Lemma $Rsqr\_inv : \forall\ x{:}R,\ x \neq 0 \rightarrow Rsqr\ (/\ x) = /\ Rsqr\ x.$

Lemma $canonical\_Rsqr$:
$\forall\ (a{:}nonzeroreal)\ (b\ c\ x{:}R),$
$a \times Rsqr\ x + b \times x + c =$
$a \times Rsqr\ (x + b\ /\ (2 \times a)) + (4 \times a \times c\ -\ Rsqr\ b)\ /\ (4 \times a).$

Lemma $Rsqr\_eq : \forall\ x\ y{:}R,\ Rsqr\ x = Rsqr\ y \rightarrow x = y \vee x = -\ y.$

Chapter 131

Module Coq.Reals.Rtopology

```
Require Import Rbase.
Require Import Rfunctions.
Require Import Ranalysis1.
Require Import RList.
Require Import Classical_Prop.
Require Import Classical_Pred_Type. Open Local Scope R_scope.
```

131.1 General definitions and propositions

```
Definition included (D1 D2:R → Prop) : Prop := ∀ x:R, D1 x → D2 x.
Definition disc (x:R) (delta:posreal) (y:R) : Prop := Rabs (y - x) < delta.
Definition neighbourhood (V:R → Prop) (x:R) : Prop :=
  ∃ delta : posreal, included (disc x delta) V.
Definition open_set (D:R → Prop) : Prop :=
  ∀ x:R, D x → neighbourhood D x.
Definition complementary (D:R → Prop) (c:R) : Prop := ¬ D c.
Definition closed_set (D:R → Prop) : Prop := open_set (complementary D).
Definition intersection_domain (D1 D2:R → Prop) (c:R) : Prop := D1 c ∧ D2 c.
Definition union_domain (D1 D2:R → Prop) (c:R) : Prop := D1 c ∨ D2 c.
Definition interior (D:R → Prop) (x:R) : Prop := neighbourhood D x.

Lemma interior_P1 : ∀ D:R → Prop, included (interior D) D.

Lemma interior_P2 : ∀ D:R → Prop, open_set D → included D (interior D).

Definition point_adherent (D:R → Prop) (x:R) : Prop :=
  ∀ V:R → Prop,
    neighbourhood V x → ∃ y : R, intersection_domain V D y.
Definition adherence (D:R → Prop) (x:R) : Prop := point_adherent D x.

Lemma adherence_P1 : ∀ D:R → Prop, included D (adherence D).

Lemma included_trans :
  ∀ D1 D2 D3:R → Prop,
```

included D1 D2 → included D2 D3 → included D1 D3.

Lemma *interior_P3* : ∀ *D*:*R* → Prop, *open_set* (*interior D*).

Lemma *complementary_P1* :
∀ *D*:*R* → Prop,
¬ (∃ *y* : *R*, *intersection_domain D* (*complementary D*) *y*).

Lemma *adherence_P2* :
∀ *D*:*R* → Prop, *closed_set D* → *included* (*adherence D*) *D*.

Lemma *adherence_P3* : ∀ *D*:*R* → Prop, *closed_set* (*adherence D*).

Definition *eq_Dom* (*D1 D2*:*R* → Prop) : Prop :=
included D1 D2 ∧ *included D2 D1*.

Infix "=_D" := *eq_Dom* (*at level* 70, *no associativity*).

Lemma *open_set_P1* : ∀ *D*:*R* → Prop, *open_set D* ↔ *D* =_*D interior D*.

Lemma *closed_set_P1* : ∀ *D*:*R* → Prop, *closed_set D* ↔ *D* =_*D adherence D*.

Lemma *neighbourhood_P1* :
∀ (*D1 D2*:*R* → Prop) (*x*:*R*),
included D1 D2 → *neighbourhood D1 x* → *neighbourhood D2 x*.

Lemma *open_set_P2* :
∀ *D1 D2*:*R* → Prop,
open_set D1 → *open_set D2* → *open_set* (*union_domain D1 D2*).

Lemma *open_set_P3* :
∀ *D1 D2*:*R* → Prop,
open_set D1 → *open_set D2* → *open_set* (*intersection_domain D1 D2*).

Lemma *open_set_P4* : *open_set* (fun *x*:*R* ⇒ *False*).

Lemma *open_set_P5* : *open_set* (fun *x*:*R* ⇒ *True*).

Lemma *disc_P1* : ∀ (*x*:*R*) (*del*:*posreal*), *open_set* (*disc x del*).

Lemma *continuity_P1* :
∀ (*f*:*R* → *R*) (*x*:*R*),
continuity_pt f x ↔
(∀ *W*:*R* → Prop,
neighbourhood W (*f x*) →
∃ *V* : *R* → Prop,
neighbourhood V x ∧ (∀ *y*:*R*, *V y* → *W* (*f y*))).

Definition *image_rec* (*f*:*R* → *R*) (*D*:*R* → Prop) (*x*:*R*) : Prop := *D* (*f x*).

Lemma *continuity_P2* :
∀ (*f*:*R* → *R*) (*D*:*R* → Prop),
continuity f → *open_set D* → *open_set* (*image_rec f D*).

Lemma *continuity_P3* :
∀ *f*:*R* → *R*,

continuity f ↔
(∀ *D*:*R* → Prop, *open_set D* → *open_set* (*image_rec f D*)).

Theorem *Rsepare* :
∀ *x y*:*R*,
x ≠ *y* →
∃ *V* : *R* → Prop,
(∃ *W* : *R* → Prop,
neighbourhood V x ∧
neighbourhood W y ∧ ¬ (∃ *y* : *R*, *intersection_domain V W y*)).

Record *family* : Type := *mkfamily*
{*ind* : *R* → Prop;
f :> *R* → *R* → Prop;
cond_fam : ∀ *x*:*R*, (∃ *y* : *R*, *f x y*) → *ind x*}.

Definition *family_open_set* (*f*:*family*) : Prop := ∀ *x*:*R*, *open_set* (*f x*).

Definition *domain_finite* (*D*:*R* → Prop) : Prop :=
∃ *l* : *Rlist*, (∀ *x*:*R*, *D x* ↔ *In x l*).

Definition *family_finite* (*f*:*family*) : Prop := *domain_finite* (*ind f*).

Definition *covering* (*D*:*R* → Prop) (*f*:*family*) : Prop :=
∀ *x*:*R*, *D x* → ∃ *y* : *R*, *f y x*.

Definition *covering_open_set* (*D*:*R* → Prop) (*f*:*family*) : Prop :=
covering D f ∧ *family_open_set f*.

Definition *covering_finite* (*D*:*R* → Prop) (*f*:*family*) : Prop :=
covering D f ∧ *family_finite f*.

Lemma *restriction_family* :
∀ (*f*:*family*) (*D*:*R* → Prop) (*x*:*R*),
(∃ *y* : *R*, (fun *z1 z2*:*R* ⇒ *f z1 z2* ∧ *D z1*) *x y*) →
intersection_domain (*ind f*) *D x*.

Definition *subfamily* (*f*:*family*) (*D*:*R* → Prop) : *family* :=
mkfamily (*intersection_domain* (*ind f*) *D*) (fun *x y*:*R* ⇒ *f x y* ∧ *D x*)
(*restriction_family f D*).

Definition *compact* (*X*:*R* → Prop) : Prop :=
∀ *f*:*family*,
covering_open_set X f →
∃ *D* : *R* → Prop, *covering_finite X* (*subfamily f D*).

Lemma *family_P1* :
∀ (*f*:*family*) (*D*:*R* → Prop),
family_open_set f → *family_open_set* (*subfamily f D*).

Definition *bounded* (*D*:*R* → Prop) : Prop :=
∃ *m* : *R*, (∃ *M* : *R*, (∀ *x*:*R*, *D x* → *m* ≤ *x* ≤ *M*)).

Lemma *open_set_P6* :

$\forall$ *D1 D2*:*R* $\to$ Prop, *open_set D1* $\to$ *D1* =_*D D2* $\to$ *open_set D2*.

Lemma *compact_P1* : $\forall$ *X*:*R* $\to$ Prop, *compact X* $\to$ *bounded X*.

Lemma *compact_P2* : $\forall$ *X*:*R* $\to$ Prop, *compact X* $\to$ *closed_set X*.

Lemma *compact_EMP* : *compact* (fun _:*R* $\Rightarrow$ *False*).

Lemma *compact_eqDom* :
$\forall$ *X1 X2*:*R* $\to$ Prop, *compact X1* $\to$ *X1* =_*D X2* $\to$ *compact X2*.

Borel-Lebesgue's lemma
Lemma *compact_P3* : $\forall$ *a b*:*R*, *compact* (fun *c*:*R* $\Rightarrow$ $a \leq c \leq b$).

Lemma *compact_P4* :
$\forall$ *X F*:*R* $\to$ Prop, *compact X* $\to$ *closed_set F* $\to$ *included F X* $\to$ *compact F*.

Lemma *compact_P5* : $\forall$ *X*:*R* $\to$ Prop, *closed_set X* $\to$ *bounded X* $\to$ *compact X*.

Lemma *compact_carac* :
$\forall$ *X*:*R* $\to$ Prop, *compact X* $\leftrightarrow$ *closed_set X* $\wedge$ *bounded X*.

Definition *image_dir* (*f*:*R* $\to$ *R*) (*D*:*R* $\to$ Prop) (*x*:*R*) : Prop :=
$\exists$ *y* : *R*, *x* = *f y* $\wedge$ *D y*.

Lemma *continuity_compact* :
$\forall$ (*f*:*R* $\to$ *R*) (*X*:*R* $\to$ Prop),
(*continuity_pt f x*) $\to$ *compact X* $\to$ *compact* (*image_dir f X*).

Lemma *Rlt_Rminus* : $\forall$ *a b*:*R*, $a < b \to 0 < b$ - *a*.

Lemma *prolongement_C0* :
$\forall$ (*f*:*R* $\to$ *R*) (*a b*:*R*),
$a \leq b \to$
($\forall$ *c*:*R*, $a \leq c \leq b \to$ *continuity_pt f c*) $\to$
$\exists$ *g* : *R* $\to$ *R*,
continuity g $\wedge$ ($\forall$ *c*:*R*, $a \leq c \leq b \to g\ c = f\ c$).

Lemma *continuity_ab_maj* :
$\forall$ (*f*:*R* $\to$ *R*) (*a b*:*R*),
$a \leq b \to$
($\forall$ *c*:*R*, $a \leq c \leq b \to$ *continuity_pt f c*) $\to$
$\exists$ *Mx* : *R*, ($\forall$ *c*:*R*, $a \leq c \leq b \to f\ c \leq f\ Mx$) $\wedge$ $a \leq Mx \leq b$.

Lemma *continuity_ab_min* :
$\forall$ (*f*:*R* $\to$ *R*) (*a b*:*R*),
$a \leq b \to$
($\forall$ *c*:*R*, $a \leq c \leq b \to$ *continuity_pt f c*) $\to$
$\exists$ *mx* : *R*, ($\forall$ *c*:*R*, $a \leq c \leq b \to f\ mx \leq f\ c$) $\wedge$ $a \leq mx \leq b$.

131.2 Proof of Bolzano-Weierstrass theorem

Definition *ValAdh* (*un*:*nat* $\to$ *R*) (*x*:*R*) : Prop :=

```
  ∀ (V:R → Prop) (N:nat),
    neighbourhood V x → ∃ p : nat, (N ≤ p)%nat ∧ V (un p).

Definition intersection_family (f:family) (x:R) : Prop :=
  ∀ y:R, ind f y → f y x.

Lemma ValAdh_un_exists :
  ∀ (un:nat → R) (D:=fun x:R ⇒ ∃ n : nat, x = INR n)
    (f:=
      fun x:R ⇒
        adherence
        (fun y:R ⇒ (∃ p : nat, y = un p ∧ x ≤ INR p) ∧ D x))
    (x:R), (∃ y : R, f x y) → D x.

Definition ValAdh_un (un:nat → R) : R → Prop :=
  let D := fun x:R ⇒ ∃ n : nat, x = INR n in
    let f :=
      fun x:R ⇒
        adherence
        (fun y:R ⇒ (∃ p : nat, y = un p ∧ x ≤ INR p) ∧ D x) in
        intersection_family (mkfamily D f (ValAdh_un_exists un)).

Lemma ValAdh_un_prop :
  ∀ (un:nat → R) (x:R), ValAdh un x ↔ ValAdh_un un x.

Lemma adherence_P4 :
  ∀ F G:R → Prop, included F G → included (adherence F) (adherence G).

Definition family_closed_set (f:family) : Prop :=
  ∀ x:R, closed_set (f x).

Definition intersection_vide_in (D:R → Prop) (f:family) : Prop :=
  ∀ x:R,
    (ind f x → included (f x) D) ∧
    ¬ (∃ y : R, intersection_family f y).

Definition intersection_vide_finite_in (D:R → Prop)
  (f:family) : Prop := intersection_vide_in D f ∧ family_finite f.

Lemma compact_P6 :
  ∀ X:R → Prop,
    compact X →
    (∃ z : R, X z) →
    ∀ g:family,
      family_closed_set g →
      intersection_vide_in X g →
      ∃ D : R → Prop, intersection_vide_finite_in X (subfamily g D).

Theorem Bolzano_Weierstrass :
  ∀ (un:nat → R) (X:R → Prop),
    compact X → (∀ n:nat, X (un n)) → ∃ l : R, ValAdh un l.
```

131.3 Proof of Heine's theorem

Definition *uniform_continuity* $(f{:}R \to R)\ (X{:}R \to \mathsf{Prop}) : \mathsf{Prop} :=$
$\forall\ eps{:}posreal,$
$\exists\ delta : posreal,$
$(\forall\ x\ y{:}R,$
$X\ x \to X\ y \to Rabs\ (x - y) < delta \to Rabs\ (f\ x - f\ y) < eps).$

Lemma *is_lub_u* :
$\forall\ (E{:}R \to \mathsf{Prop})\ (x\ y{:}R),\ is\_lub\ E\ x \to is\_lub\ E\ y \to x = y.$

Lemma *domain_P1* :
$\forall\ X{:}R \to \mathsf{Prop},$
$\neg\ (\exists\ y : R,\ X\ y) \vee$
$(\exists\ y : R,\ X\ y \wedge (\forall\ x{:}R,\ X\ x \to x = y)) \vee$
$(\exists\ x : R,\ (\exists\ y : R,\ X\ x \wedge X\ y \wedge x \neq y)).$

Theorem *Heine* :
$\forall\ (f{:}R \to R)\ (X{:}R \to \mathsf{Prop}),$
$compact\ X \to$
$(\forall\ x{:}R,\ X\ x \to continuity\_pt\ f\ x) \to uniform\_continuity\ f\ X.$

Chapter 132

Module Coq.Reals.Rtrigo_alt

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo_def*.
Open Local *Scope R_scope*.

Using series definitions of cos and sin
Definition *sin_term* (*a*:*R*) (*i*:*nat*) : *R* :=
(-1) ^ *i* × (*a* ^ (2 × *i* + 1) / *INR* (*fact* (2 × *i* + 1))).

Definition *cos_term* (*a*:*R*) (*i*:*nat*) : *R* :=
(-1) ^ *i* × (*a* ^ (2 × *i*) / *INR* (*fact* (2 × *i*))).

Definition *sin_approx* (*a*:*R*) (*n*:*nat*) : *R* := *sum_f_R0* (*sin_term a*) *n*.

Definition *cos_approx* (*a*:*R*) (*n*:*nat*) : *R* := *sum_f_R0* (*cos_term a*) *n*.

Lemma *PI_4* : *PI* ≤ 4.

Theorem *sin_bound* :
∀ (*a*:*R*) (*n*:*nat*),
0 ≤ *a* →
a ≤ *PI* → *sin_approx a* (2 × *n* + 1) ≤ *sin a* ≤ *sin_approx a* (2 × (*n* + 1)).

Lemma *cos_bound* :
∀ (*a*:*R*) (*n*:*nat*),
- *PI* / 2 ≤ *a* →
a ≤ *PI* / 2 →
cos_approx a (2 × *n* + 1) ≤ *cos a* ≤ *cos_approx a* (2 × (*n* + 1)).

Chapter 133

Module Coq.Reals.Rtrigo_calc

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo*.
Require Import *R_sqrt*.
Open Local *Scope R_scope*.

Lemma *tan_PI* : $tan\ PI = 0$.

Lemma *sin_3PI2* : $sin\ (3 \times (PI\ /\ 2)) =$ -1.

Lemma *tan_2PI* : $tan\ (2 \times PI) = 0$.

Lemma *sin_cos_PI4* : $sin\ (PI\ /\ 4) = cos\ (PI\ /\ 4)$.

Lemma *sin_PI3_cos_PI6* : $sin\ (PI\ /\ 3) = cos\ (PI\ /\ 6)$.

Lemma *sin_PI6_cos_PI3* : $cos\ (PI\ /\ 3) = sin\ (PI\ /\ 6)$.

Lemma *PI6_RGT_0* : $0 < PI\ /\ 6$.

Lemma *PI6_RLT_PI2* : $PI\ /\ 6 < PI\ /\ 2$.

Lemma *sin_PI6* : $sin\ (PI\ /\ 6) = 1\ /\ 2$.

Lemma *sqrt2_neq_0* : $sqrt\ 2 \neq 0$.

Lemma *R1_sqrt2_neq_0* : $1\ /\ sqrt\ 2 \neq 0$.

Lemma *sqrt3_2_neq_0* : $2 \times sqrt\ 3 \neq 0$.

Lemma *Rlt_sqrt2_0* : $0 < sqrt\ 2$.

Lemma *Rlt_sqrt3_0* : $0 < sqrt\ 3$.

Lemma *PI4_RGT_0* : $0 < PI\ /\ 4$.

Lemma *cos_PI4* : $cos\ (PI\ /\ 4) = 1\ /\ sqrt\ 2$.

Lemma *sin_PI4* : $sin\ (PI\ /\ 4) = 1\ /\ sqrt\ 2$.

Lemma *tan_PI4* : $tan\ (PI\ /\ 4) = 1$.

Lemma *cos3PI4* : $cos\ (3 \times (PI\ /\ 4)) =$ -1 $/\ sqrt\ 2$.

Lemma $sin3PI4 : sin\ (3 \times (PI\ /\ 4)) = 1\ /\ sqrt\ 2.$

Lemma $cos\_PI6 : cos\ (PI\ /\ 6) = sqrt\ 3\ /\ 2.$

Lemma $tan\_PI6 : tan\ (PI\ /\ 6) = 1\ /\ sqrt\ 3.$

Lemma $sin\_PI3 : sin\ (PI\ /\ 3) = sqrt\ 3\ /\ 2.$

Lemma $cos\_PI3 : cos\ (PI\ /\ 3) = 1\ /\ 2.$

Lemma $tan\_PI3 : tan\ (PI\ /\ 3) = sqrt\ 3.$

Lemma $sin\_2PI3 : sin\ (2 \times (PI\ /\ 3)) = sqrt\ 3\ /\ 2.$

Lemma $cos\_2PI3 : cos\ (2 \times (PI\ /\ 3)) =$ -1 / 2.

Lemma $tan\_2PI3 : tan\ (2 \times (PI\ /\ 3)) =$ - $sqrt\ 3.$

Lemma $cos\_5PI4 : cos\ (5 \times (PI\ /\ 4)) =$ -1 / $sqrt\ 2.$

Lemma $sin\_5PI4 : sin\ (5 \times (PI\ /\ 4)) =$ -1 / $sqrt\ 2.$

Lemma $sin\_cos5PI4 : cos\ (5 \times (PI\ /\ 4)) = sin\ (5 \times (PI\ /\ 4)).$

Lemma $Rgt\_3PI2\_0 : 0 < 3 \times (PI\ /\ 2).$

Lemma $Rgt\_2PI\_0 : 0 < 2 \times PI.$

Lemma $Rlt\_PI\_3PI2 : PI < 3 \times (PI\ /\ 2).$

Lemma $Rlt\_3PI2\_2PI : 3 \times (PI\ /\ 2) < 2 \times PI.$

Radian -> Degree | Degree -> Radian
Definition $plat : R := 180.$
Definition $toRad\ (x{:}R) : R := x \times PI \times /\ plat.$
Definition $toDeg\ (x{:}R) : R := x \times plat \times /\ PI.$

Lemma $rad\_deg : \forall\ x{:}R,\ toRad\ (toDeg\ x) = x.$

Lemma $toRad\_inj : \forall\ x\ y{:}R,\ toRad\ x = toRad\ y \rightarrow x = y.$

Lemma $deg\_rad : \forall\ x{:}R,\ toDeg\ (toRad\ x) = x.$

Definition $sind\ (x{:}R) : R := sin\ (toRad\ x).$
Definition $cosd\ (x{:}R) : R := cos\ (toRad\ x).$
Definition $tand\ (x{:}R) : R := tan\ (toRad\ x).$

Lemma $Rsqr\_sin\_cos\_d\_one : \forall\ x{:}R,\ Rsqr\ (sind\ x) + Rsqr\ (cosd\ x) = 1.$

Other properties
Lemma $sin\_lb\_ge\_0 : \forall\ a{:}R,\ 0 \leq a \rightarrow a \leq PI\ /\ 2 \rightarrow 0 \leq sin\_lb\ a.$

Chapter 134

Module Coq.Reals.Rtrigo_def

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo_fun*.
Require Import *Max*.
Open Local *Scope R_scope*.

134.1 Definition of exponential

Definition *exp_in* (*x l*:*R*) : Prop :=
  infinit_sum (fun *i*:*nat* ⇒ / *INR* (*fact i*) × *x* ^ *i*) *l*.

Lemma *exp_cof_no_R0* : ∀ *n*:*nat*, / *INR* (*fact n*) ≠ 0.

Lemma *exist_exp* : ∀ *x*:*R*, *sigT* (fun *l*:*R* ⇒ *exp_in x l*).

Definition *exp* (*x*:*R*) : *R* := *projT1* (*exist_exp x*).

Lemma *pow_i* : ∀ *i*:*nat*, (0 < *i*)%*nat* → 0 ^ *i* = 0.

Lemma *exist_exp0* : *sigT* (fun *l*:*R* ⇒ *exp_in* 0 *l*).

Lemma *exp_0* : *exp* 0 = 1.

134.2 Definition of hyperbolic functions

Definition *cosh* (*x*:*R*) : *R* := (*exp x* + *exp* (- *x*)) / 2.
Definition *sinh* (*x*:*R*) : *R* := (*exp x* - *exp* (- *x*)) / 2.
Definition *tanh* (*x*:*R*) : *R* := *sinh x* / *cosh x*.

Lemma *cosh_0* : *cosh* 0 = 1.

Lemma *sinh_0* : *sinh* 0 = 0.

Definition *cos_n* (*n*:*nat*) : *R* := (-1) ^ *n* / *INR* (*fact* (2 × *n*)).

Lemma *simpl_cos_n* :

$\forall\ n{:}nat,\ cos\_n\ (S\ n)\ /\ cos\_n\ n = -\ /\ INR\ (2 \times S\ n \times (2 \times n + 1))$.

Lemma *archimed_cor1* :
$\forall\ eps{:}R,\ 0 < eps \rightarrow \exists\ N : nat,\ /\ INR\ N < eps \wedge (0 < N)\%nat$.

Lemma *Alembert_cos* : *Un_cv* (fun $n{:}nat \Rightarrow Rabs\ (cos\_n\ (S\ n)\ /\ cos\_n\ n))\ 0$.

Lemma *cosn_no_R0* : $\forall\ n{:}nat,\ cos\_n\ n \neq 0$.

Definition *cos_in* $(x\ l{:}R)$: Prop :=
infinit_sum (fun $i{:}nat \Rightarrow cos\_n\ i \times x\ \hat{}\ i)\ l$.

Lemma *exist_cos* : $\forall\ x{:}R,\ sigT$ (fun $l{:}R \Rightarrow cos\_in\ x\ l)$.

Definition of cosinus
Definition *cos* $(x{:}R)$: R :=
match *exist_cos* $(Rsqr\ x)$ with
| *existT* $a\ b \Rightarrow a$
end.

Definition *sin_n* $(n{:}nat)$: $R := (-1)\ \hat{}\ n\ /\ INR\ (fact\ (2 \times n + 1))$.

Lemma *simpl_sin_n* :
$\forall\ n{:}nat,\ sin\_n\ (S\ n)\ /\ sin\_n\ n = -\ /\ INR\ ((2 \times S\ n + 1) \times (2 \times S\ n))$.

Lemma *Alembert_sin* : *Un_cv* (fun $n{:}nat \Rightarrow Rabs\ (sin\_n\ (S\ n)\ /\ sin\_n\ n))\ 0$.

Lemma *sin_no_R0* : $\forall\ n{:}nat,\ sin\_n\ n \neq 0$.

Definition *sin_in* $(x\ l{:}R)$: Prop :=
infinit_sum (fun $i{:}nat \Rightarrow sin\_n\ i \times x\ \hat{}\ i)\ l$.

Lemma *exist_sin* : $\forall\ x{:}R,\ sigT$ (fun $l{:}R \Rightarrow sin\_in\ x\ l)$.

Definition *sin* $(x{:}R)$: R :=
match *exist_sin* $(Rsqr\ x)$ with
| *existT* $a\ b \Rightarrow x \times a$
end.

134.3 Properties

Lemma *cos_sym* : $\forall\ x{:}R,\ cos\ x = cos\ (-\ x)$.

Lemma *sin_antisym* : $\forall\ x{:}R,\ sin\ (-\ x) = -\ sin\ x$.

Lemma *sin_0* : $sin\ 0 = 0$.

Lemma *exist_cos0* : $sigT$ (fun $l{:}R \Rightarrow cos\_in\ 0\ l)$.

Lemma *cos_0* : $cos\ 0 = 1$.

Chapter 135

Module Coq.Reals.Rtrigo_fun

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Open Local *Scope R_scope*.

To define transcendental functions
for exponential function
Lemma *Alembert_exp* :
Un_cv (fun *n*:*nat* ⇒ *Rabs* (/ *INR* (*fact* (*S n*)) × / / *INR* (*fact n*))) 0.

Chapter 136

Module Coq.Reals.Rtrigo_reg

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Import *Rtrigo*.
Require Import *Ranalysis1*.
Require Import *PSeries_reg*.
Open Local *Scope nat_scope*.
Open Local *Scope R_scope*.

Lemma *CVN_R_cos* :
  ∀ *fn*:*nat* → *R* → *R*,
    fn = (fun (*N*:*nat*) (*x*:*R*) ⇒ (-1) ^ *N* / *INR* (*fact* (2 × *N*)) × *x* ^ (2 × *N*)) →
    CVN_R fn.

Lemma *continuity_cos* : *continuity cos*.

Lemma *continuity_sin* : *continuity sin*.

Lemma *CVN_R_sin* :
  ∀ *fn*:*nat* → *R* → *R*,
    fn =
    (fun (*N*:*nat*) (*x*:*R*) ⇒ (-1) ^ *N* / *INR* (*fact* (2 × *N* + 1)) × *x* ^ (2 × *N*)) →
    CVN_R fn.

(sin h)/h -> 1 when h -> 0
Lemma *derivable_pt_lim_sin_0* : *derivable_pt_lim sin* 0 1.

((cos h)-1)/h -> 0 when h -> 0
Lemma *derivable_pt_lim_cos_0* : *derivable_pt_lim cos* 0 0.

Theorem *derivable_pt_lim_sin* : ∀ *x*:*R*, *derivable_pt_lim sin x* (*cos x*).

Lemma *derivable_pt_lim_cos* : ∀ *x*:*R*, *derivable_pt_lim cos x* (- *sin x*).

Lemma *derivable_pt_sin* : ∀ *x*:*R*, *derivable_pt sin x*.

Lemma *derivable_pt_cos* : ∀ *x*:*R*, *derivable_pt cos x*.

Lemma *derivable_sin* : *derivable sin*.

Lemma *derivable_cos* : *derivable cos*.

Lemma *derive_pt_sin* :
∀ *x*:*R*, *derive_pt sin x* (*derivable_pt_sin* _) = *cos x*.

Lemma *derive_pt_cos* :
∀ *x*:*R*, *derive_pt cos x* (*derivable_pt_cos* _) = - *sin x*.

Chapter 137

Module Coq.Reals.Rtrigo

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *SeqSeries*.
Require Export *Rtrigo_fun*.
Require Export *Rtrigo_def*.
Require Export *Rtrigo_alt*.
Require Export *Cos_rel*.
Require Export *Cos_plus*.
Require Import *ZArith_base*.
Require Import *Zcomplements*.
Require Import *Classical_Prop*.
Open Local *Scope nat_scope*.
Open Local *Scope R_scope*.

sin_PI2 is the only remaining axiom
Axiom *sin_PI2* : $sin\ (PI\ /\ 2) = 1$.

Lemma *PI_neq0* : $PI \neq 0$.

Lemma *cos_minus* : $\forall\ x\ y{:}R,\ cos\ (x\ -\ y) = cos\ x \times cos\ y + sin\ x \times sin\ y$.

Lemma *sin2_cos2* : $\forall\ x{:}R,\ Rsqr\ (sin\ x) + Rsqr\ (cos\ x) = 1$.

Lemma *cos2* : $\forall\ x{:}R,\ Rsqr\ (cos\ x) = 1\ -\ Rsqr\ (sin\ x)$.

Lemma *cos_PI2* : $cos\ (PI\ /\ 2) = 0$.

Lemma *cos_PI* : $cos\ PI = -1$.

Lemma *sin_PI* : $sin\ PI = 0$.

Lemma *neg_cos* : $\forall\ x{:}R,\ cos\ (x + PI) = -\ cos\ x$.

Lemma *sin_cos* : $\forall\ x{:}R,\ sin\ x = -\ cos\ (PI\ /\ 2 + x)$.

Lemma *sin_plus* : $\forall\ x\ y{:}R,\ sin\ (x + y) = sin\ x \times cos\ y + cos\ x \times sin\ y$.

Lemma *sin_minus* : $\forall\ x\ y{:}R,\ sin\ (x\ -\ y) = sin\ x \times cos\ y\ -\ cos\ x \times sin\ y$.

Definition *tan* $(x{:}R) : R := sin\ x\ /\ cos\ x$.

Lemma *tan_plus* :
  $\forall\ x\ y{:}R,$
    $cos\ x \neq 0 \rightarrow$
    $cos\ y \neq 0 \rightarrow$
    $cos\ (x + y) \neq 0 \rightarrow$
    1 - $tan\ x \times tan\ y \neq 0 \rightarrow$
    $tan\ (x + y) = (tan\ x + tan\ y)\ /\ (1$ - $tan\ x \times tan\ y).$

137.1 Some properties of cos, sin and tan

Lemma *sin2* : $\forall\ x{:}R,\ Rsqr\ (sin\ x) = 1$ - $Rsqr\ (cos\ x).$

Lemma *sin_2a* : $\forall\ x{:}R,\ sin\ (2 \times x) = 2 \times sin\ x \times cos\ x.$

Lemma *cos_2a* : $\forall\ x{:}R,\ cos\ (2 \times x) = cos\ x \times cos\ x$ - $sin\ x \times sin\ x.$

Lemma *cos_2a_cos* : $\forall\ x{:}R,\ cos\ (2 \times x) = 2 \times cos\ x \times cos\ x$ - $1.$

Lemma *cos_2a_sin* : $\forall\ x{:}R,\ cos\ (2 \times x) = 1$ - $2 \times sin\ x \times sin\ x.$

Lemma *tan_2a* :
  $\forall\ x{:}R,$
    $cos\ x \neq 0 \rightarrow$
    $cos\ (2 \times x) \neq 0 \rightarrow$
    1 - $tan\ x \times tan\ x \neq 0 \rightarrow tan\ (2 \times x) = 2 \times tan\ x\ /\ (1$ - $tan\ x \times tan\ x).$

Lemma *sin_neg* : $\forall\ x{:}R,\ sin\ (-\ x) =$ - $sin\ x.$

Lemma *cos_neg* : $\forall\ x{:}R,\ cos\ (-\ x) = cos\ x.$

Lemma *tan_0* : $tan\ 0 = 0.$

Lemma *tan_neg* : $\forall\ x{:}R,\ tan\ (-\ x) =$ - $tan\ x.$

Lemma *tan_minus* :
  $\forall\ x\ y{:}R,$
    $cos\ x \neq 0 \rightarrow$
    $cos\ y \neq 0 \rightarrow$
    $cos\ (x$ - $y) \neq 0 \rightarrow$
    $1 + tan\ x \times tan\ y \neq 0 \rightarrow$
    $tan\ (x$ - $y) = (tan\ x$ - $tan\ y)\ /\ (1 + tan\ x \times tan\ y).$

Lemma *cos_3PI2* : $cos\ (3 \times (PI\ /\ 2)) = 0.$

Lemma *sin_2PI* : $sin\ (2 \times PI) = 0.$

Lemma *cos_2PI* : $cos\ (2 \times PI) = 1.$

Lemma *neg_sin* : $\forall\ x{:}R,\ sin\ (x + PI) =$ - $sin\ x.$

Lemma *sin_PI_x* : $\forall\ x{:}R,\ sin\ (PI$ - $x) = sin\ x.$

Lemma *sin_period* : $\forall\ (x{:}R)\ (k{:}nat),\ sin\ (x + 2 \times INR\ k \times PI) = sin\ x.$

Lemma *cos_period* : $\forall\ (x{:}R)\ (k{:}nat),\ cos\ (x + 2 \times INR\ k \times PI) = cos\ x.$

Lemma $sin\_shift : \forall\ x{:}R,\ sin\ (PI\ /\ 2\ -\ x) = cos\ x.$

Lemma $cos\_shift : \forall\ x{:}R,\ cos\ (PI\ /\ 2\ -\ x) = sin\ x.$

Lemma $cos\_sin : \forall\ x{:}R,\ cos\ x = sin\ (PI\ /\ 2\ +\ x).$

Lemma $PI2\_RGT\_0 : 0 < PI\ /\ 2.$

Lemma $SIN\_bound : \forall\ x{:}R,\ -1 \leq sin\ x \leq 1.$

Lemma $COS\_bound : \forall\ x{:}R,\ -1 \leq cos\ x \leq 1.$

Lemma $cos\_sin\_0 : \forall\ x{:}R,\ \neg\ (cos\ x = 0 \wedge sin\ x = 0).$

Lemma $cos\_sin\_0\_var : \forall\ x{:}R,\ cos\ x \neq 0 \vee sin\ x \neq 0.$

137.2 Using series definitions of cos and sin

Definition $sin\_lb\ (a{:}R) : R := sin\_approx\ a\ 3.$
Definition $sin\_ub\ (a{:}R) : R := sin\_approx\ a\ 4.$
Definition $cos\_lb\ (a{:}R) : R := cos\_approx\ a\ 3.$
Definition $cos\_ub\ (a{:}R) : R := cos\_approx\ a\ 4.$

Lemma $sin\_lb\_gt\_0 : \forall\ a{:}R,\ 0 < a \rightarrow a \leq PI\ /\ 2 \rightarrow 0 < sin\_lb\ a.$

Lemma $SIN : \forall\ a{:}R,\ 0 \leq a \rightarrow a \leq PI \rightarrow sin\_lb\ a \leq sin\ a \leq sin\_ub\ a.$

Lemma $COS :$
$\forall\ a{:}R,\ -\ PI\ /\ 2 \leq a \rightarrow a \leq PI\ /\ 2 \rightarrow cos\_lb\ a \leq cos\ a \leq cos\_ub\ a.$

Lemma $\_PI2\_RLT\_0 : -\ (PI\ /\ 2) < 0.$

Lemma $PI4\_RLT\_PI2 : PI\ /\ 4 < PI\ /\ 2.$

Lemma $PI2\_Rlt\_PI : PI\ /\ 2 < PI.$

137.3 Increasing and decreasing of *cos* and *sin*

Theorem $sin\_gt\_0 : \forall\ x{:}R,\ 0 < x \rightarrow x < PI \rightarrow 0 < sin\ x.$

Theorem $cos\_gt\_0 : \forall\ x{:}R,\ -\ (PI\ /\ 2) < x \rightarrow x < PI\ /\ 2 \rightarrow 0 < cos\ x.$

Lemma $sin\_ge\_0 : \forall\ x{:}R,\ 0 \leq x \rightarrow x \leq PI \rightarrow 0 \leq sin\ x.$

Lemma $cos\_ge\_0 : \forall\ x{:}R,\ -\ (PI\ /\ 2) \leq x \rightarrow x \leq PI\ /\ 2 \rightarrow 0 \leq cos\ x.$

Lemma $sin\_le\_0 : \forall\ x{:}R,\ PI \leq x \rightarrow x \leq 2 \times PI \rightarrow sin\ x \leq 0.$

Lemma $cos\_le\_0 : \forall\ x{:}R,\ PI\ /\ 2 \leq x \rightarrow x \leq 3 \times (PI\ /\ 2) \rightarrow cos\ x \leq 0.$

Lemma $sin\_lt\_0 : \forall\ x{:}R,\ PI < x \rightarrow x < 2 \times PI \rightarrow sin\ x < 0.$

Lemma $sin\_lt\_0\_var : \forall\ x{:}R,\ -\ PI < x \rightarrow x < 0 \rightarrow sin\ x < 0.$

Lemma $cos\_lt\_0 : \forall\ x{:}R,\ PI\ /\ 2 < x \rightarrow x < 3 \times (PI\ /\ 2) \rightarrow cos\ x < 0.$

Lemma $tan\_gt\_0 : \forall\ x{:}R,\ 0 < x \rightarrow x < PI\ /\ 2 \rightarrow 0 < tan\ x.$

Lemma *tan_lt_0* : $\forall\ x{:}R,\ -\ (PI\ /\ 2) < x \to x < 0 \to tan\ x < 0.$

Lemma *cos_ge_0_3PI2* :
$\forall\ x{:}R,\ 3 \times (PI\ /\ 2) \leq x \to x \leq 2 \times PI \to 0 \leq cos\ x.$

Lemma *form1* :
$\forall\ p\ q{:}R,\ cos\ p + cos\ q = 2 \times cos\ ((p\ -\ q)\ /\ 2) \times cos\ ((p + q)\ /\ 2).$

Lemma *form2* :
$\forall\ p\ q{:}R,\ cos\ p\ -\ cos\ q = -2 \times sin\ ((p\ -\ q)\ /\ 2) \times sin\ ((p + q)\ /\ 2).$

Lemma *form3* :
$\forall\ p\ q{:}R,\ sin\ p + sin\ q = 2 \times cos\ ((p\ -\ q)\ /\ 2) \times sin\ ((p + q)\ /\ 2).$

Lemma *form4* :
$\forall\ p\ q{:}R,\ sin\ p\ -\ sin\ q = 2 \times cos\ ((p + q)\ /\ 2) \times sin\ ((p\ -\ q)\ /\ 2).$

Lemma *sin_increasing_0* :
$\forall\ x\ y{:}R,$
$-\ (PI\ /\ 2) \leq x \to$
$x \leq PI\ /\ 2 \to -\ (PI\ /\ 2) \leq y \to y \leq PI\ /\ 2 \to sin\ x < sin\ y \to x < y.$

Lemma *sin_increasing_1* :
$\forall\ x\ y{:}R,$
$-\ (PI\ /\ 2) \leq x \to$
$x \leq PI\ /\ 2 \to -\ (PI\ /\ 2) \leq y \to y \leq PI\ /\ 2 \to x < y \to sin\ x < sin\ y.$

Lemma *sin_decreasing_0* :
$\forall\ x\ y{:}R,$
$x \leq 3 \times (PI\ /\ 2) \to$
$PI\ /\ 2 \leq x \to y \leq 3 \times (PI\ /\ 2) \to PI\ /\ 2 \leq y \to sin\ x < sin\ y \to y < x.$

Lemma *sin_decreasing_1* :
$\forall\ x\ y{:}R,$
$x \leq 3 \times (PI\ /\ 2) \to$
$PI\ /\ 2 \leq x \to y \leq 3 \times (PI\ /\ 2) \to PI\ /\ 2 \leq y \to x < y \to sin\ y < sin\ x.$

Lemma *cos_increasing_0* :
$\forall\ x\ y{:}R,$
$PI \leq x \to x \leq 2 \times PI \to PI \leq y \to y \leq 2 \times PI \to cos\ x < cos\ y \to x < y.$

Lemma *cos_increasing_1* :
$\forall\ x\ y{:}R,$
$PI \leq x \to x \leq 2 \times PI \to PI \leq y \to y \leq 2 \times PI \to x < y \to cos\ x < cos\ y.$

Lemma *cos_decreasing_0* :
$\forall\ x\ y{:}R,$
$0 \leq x \to x \leq PI \to 0 \leq y \to y \leq PI \to cos\ x < cos\ y \to y < x.$

Lemma *cos_decreasing_1* :
$\forall\ x\ y{:}R,$
$0 \leq x \to x \leq PI \to 0 \leq y \to y \leq PI \to x < y \to cos\ y < cos\ x.$

Lemma *tan_diff* :
$\forall\ x\ y{:}R,$
$cos\ x \neq 0 \rightarrow cos\ y \neq 0 \rightarrow tan\ x$ - $tan\ y = sin\ (x$ - $y)\ /\ (cos\ x \times cos\ y).$

Lemma *tan_increasing_0* :
$\forall\ x\ y{:}R,$
- $(PI\ /\ 4) \leq x \rightarrow$
$x \leq PI\ /\ 4 \rightarrow$ - $(PI\ /\ 4) \leq y \rightarrow y \leq PI\ /\ 4 \rightarrow tan\ x < tan\ y \rightarrow x < y.$

Lemma *tan_increasing_1* :
$\forall\ x\ y{:}R,$
- $(PI\ /\ 4) \leq x \rightarrow$
$x \leq PI\ /\ 4 \rightarrow$ - $(PI\ /\ 4) \leq y \rightarrow y \leq PI\ /\ 4 \rightarrow x < y \rightarrow tan\ x < tan\ y.$

Lemma *sin_incr_0* :
$\forall\ x\ y{:}R,$
- $(PI\ /\ 2) \leq x \rightarrow$
$x \leq PI\ /\ 2 \rightarrow$ - $(PI\ /\ 2) \leq y \rightarrow y \leq PI\ /\ 2 \rightarrow sin\ x \leq sin\ y \rightarrow x \leq y.$

Lemma *sin_incr_1* :
$\forall\ x\ y{:}R,$
- $(PI\ /\ 2) \leq x \rightarrow$
$x \leq PI\ /\ 2 \rightarrow$ - $(PI\ /\ 2) \leq y \rightarrow y \leq PI\ /\ 2 \rightarrow x \leq y \rightarrow sin\ x \leq sin\ y.$

Lemma *sin_decr_0* :
$\forall\ x\ y{:}R,$
$x \leq 3 \times (PI\ /\ 2) \rightarrow$
$PI\ /\ 2 \leq x \rightarrow$
$y \leq 3 \times (PI\ /\ 2) \rightarrow PI\ /\ 2 \leq y \rightarrow sin\ x \leq sin\ y \rightarrow y \leq x.$

Lemma *sin_decr_1* :
$\forall\ x\ y{:}R,$
$x \leq 3 \times (PI\ /\ 2) \rightarrow$
$PI\ /\ 2 \leq x \rightarrow$
$y \leq 3 \times (PI\ /\ 2) \rightarrow PI\ /\ 2 \leq y \rightarrow x \leq y \rightarrow sin\ y \leq sin\ x.$

Lemma *cos_incr_0* :
$\forall\ x\ y{:}R,$
$PI \leq x \rightarrow$
$x \leq 2 \times PI \rightarrow PI \leq y \rightarrow y \leq 2 \times PI \rightarrow cos\ x \leq cos\ y \rightarrow x \leq y.$

Lemma *cos_incr_1* :
$\forall\ x\ y{:}R,$
$PI \leq x \rightarrow$
$x \leq 2 \times PI \rightarrow PI \leq y \rightarrow y \leq 2 \times PI \rightarrow x \leq y \rightarrow cos\ x \leq cos\ y.$

Lemma *cos_decr_0* :
$\forall\ x\ y{:}R,$
$0 \leq x \rightarrow x \leq PI \rightarrow 0 \leq y \rightarrow y \leq PI \rightarrow cos\ x \leq cos\ y \rightarrow y \leq x.$

Lemma *cos_decr_1* :

$\forall\ x\ y{:}R,$
$0 \leq x \rightarrow x \leq PI \rightarrow 0 \leq y \rightarrow y \leq PI \rightarrow x \leq y \rightarrow cos\ y \leq cos\ x.$

Lemma *tan_incr_0* :
$\forall\ x\ y{:}R,$
$- (PI\ /\ 4) \leq x \rightarrow$
$x \leq PI\ /\ 4 \rightarrow - (PI\ /\ 4) \leq y \rightarrow y \leq PI\ /\ 4 \rightarrow tan\ x \leq tan\ y \rightarrow x \leq y.$

Lemma *tan_incr_1* :
$\forall\ x\ y{:}R,$
$- (PI\ /\ 4) \leq x \rightarrow$
$x \leq PI\ /\ 4 \rightarrow - (PI\ /\ 4) \leq y \rightarrow y \leq PI\ /\ 4 \rightarrow x \leq y \rightarrow tan\ x \leq tan\ y.$

Lemma *sin_eq_0_1* : $\forall\ x{:}R, (\exists\ k : Z, x = IZR\ k \times PI) \rightarrow sin\ x = 0.$

Lemma *sin_eq_0_0* : $\forall\ x{:}R, sin\ x = 0 \rightarrow \exists\ k : Z, x = IZR\ k \times PI.$

Lemma *cos_eq_0_0* :
$\forall\ x{:}R, cos\ x = 0 \rightarrow \exists\ k : Z, x = IZR\ k \times PI + PI\ /\ 2.$

ring_simplify. (* rewrite (Rmult_comm PI);

Lemma *cos_eq_0_1* :
$\forall\ x{:}R, (\exists\ k : Z, x = IZR\ k \times PI + PI\ /\ 2) \rightarrow cos\ x = 0.$

Lemma *sin_eq_O_2PI_0* :
$\forall\ x{:}R,$
$0 \leq x \rightarrow x \leq 2 \times PI \rightarrow sin\ x = 0 \rightarrow x = 0 \vee x = PI \vee x = 2 \times PI.$

Lemma *sin_eq_O_2PI_1* :
$\forall\ x{:}R,$
$0 \leq x \rightarrow x \leq 2 \times PI \rightarrow x = 0 \vee x = PI \vee x = 2 \times PI \rightarrow sin\ x = 0.$

Lemma *cos_eq_0_2PI_0* :
$\forall\ x{:}R,$
$0 \leq x \rightarrow x \leq 2 \times PI \rightarrow cos\ x = 0 \rightarrow x = PI\ /\ 2 \vee x = 3 \times (PI\ /\ 2).$

Lemma *cos_eq_0_2PI_1* :
$\forall\ x{:}R,$
$0 \leq x \rightarrow x \leq 2 \times PI \rightarrow x = PI\ /\ 2 \vee x = 3 \times (PI\ /\ 2) \rightarrow cos\ x = 0.$

Chapter 138

Module Coq.Reals.SeqProp

```
Require Import Rbase.
Require Import Rfunctions.
Require Import Rseries.
Require Import Classical.
Require Import Max.
Open Local Scope R_scope.

Definition Un_decreasing (Un:nat → R) : Prop :=
  ∀ n:nat, Un (S n) ≤ Un n.
Definition opp_seq (Un:nat → R) (n:nat) : R := - Un n.
Definition has_ub (Un:nat → R) : Prop := bound (EUn Un).
Definition has_lb (Un:nat → R) : Prop := bound (EUn (opp_seq Un)).

Lemma growing_cv :
  ∀ Un:nat → R,
    Un_growing Un → has_ub Un → sigT (fun l:R ⇒ Un_cv Un l).

Lemma decreasing_growing :
  ∀ Un:nat → R, Un_decreasing Un → Un_growing (opp_seq Un).

Lemma decreasing_cv :
  ∀ Un:nat → R,
    Un_decreasing Un → has_lb Un → sigT (fun l:R ⇒ Un_cv Un l).

Lemma maj_sup :
  ∀ Un:nat → R, has_ub Un → sigT (fun l:R ⇒ is_lub (EUn Un) l).

Lemma min_inf :
  ∀ Un:nat → R,
    has_lb Un → sigT (fun l:R ⇒ is_lub (EUn (opp_seq Un)) l).

Definition majorant (Un:nat → R) (pr:has_ub Un) : R :=
  match maj_sup Un pr with
    | existT a b ⇒ a
  end.

Definition minorant (Un:nat → R) (pr:has_lb Un) : R :=
```

```
  match min_inf Un pr with
    | existT a b ⇒ - a
  end.

Lemma maj_ss :
  ∀ (Un:nat → R) (k:nat),
    has_ub Un → has_ub (fun i:nat ⇒ Un (k + i)%nat).

Lemma min_ss :
  ∀ (Un:nat → R) (k:nat),
    has_lb Un → has_lb (fun i:nat ⇒ Un (k + i)%nat).

Definition sequence_majorant (Un:nat → R) (pr:has_ub Un)
  (i:nat) : R := majorant (fun k:nat ⇒ Un (i + k)%nat) (maj_ss Un i pr).

Definition sequence_minorant (Un:nat → R) (pr:has_lb Un)
  (i:nat) : R := minorant (fun k:nat ⇒ Un (i + k)%nat) (min_ss Un i pr).

Lemma Wn_decreasing :
  ∀ (Un:nat → R) (pr:has_ub Un), Un_decreasing (sequence_majorant Un pr).

Lemma Vn_growing :
  ∀ (Un:nat → R) (pr:has_lb Un), Un_growing (sequence_minorant Un pr).

Lemma Vn_Un_Wn_order :
  ∀ (Un:nat → R) (pr1:has_ub Un) (pr2:has_lb Un)
    (n:nat), sequence_minorant Un pr2 n ≤ Un n ≤ sequence_majorant Un pr1 n.

Lemma min_maj :
  ∀ (Un:nat → R) (pr1:has_ub Un) (pr2:has_lb Un),
    has_ub (sequence_minorant Un pr2).

Lemma maj_min :
  ∀ (Un:nat → R) (pr1:has_ub Un) (pr2:has_lb Un),
    has_lb (sequence_majorant Un pr1).

Lemma cauchy_maj : ∀ Un:nat → R, Cauchy_crit Un → has_ub Un.

Lemma cauchy_opp :
  ∀ Un:nat → R, Cauchy_crit Un → Cauchy_crit (opp_seq Un).

Lemma cauchy_min : ∀ Un:nat → R, Cauchy_crit Un → has_lb Un.

Lemma maj_cv :
  ∀ (Un:nat → R) (pr:Cauchy_crit Un),
    sigT (fun l:R ⇒ Un_cv (sequence_majorant Un (cauchy_maj Un pr)) l).

Lemma min_cv :
  ∀ (Un:nat → R) (pr:Cauchy_crit Un),
    sigT (fun l:R ⇒ Un_cv (sequence_minorant Un (cauchy_min Un pr)) l).

Lemma cond_eq :
  ∀ x y:R, (∀ eps:R, 0 < eps → Rabs (x - y) < eps) → x = y.

Lemma not_Rlt : ∀ r1 r2:R, ¬ r1 < r2 → r1 ≥ r2.
```

Lemma *approx_maj* :
$\forall\ (Un{:}nat \to R)\ (pr{:}has\_ub\ Un)\ (eps{:}R),$
$0 < eps \to \exists\ k : nat, Rabs\ (majorant\ Un\ pr\ -\ Un\ k) < eps.$

Lemma *approx_min* :
$\forall\ (Un{:}nat \to R)\ (pr{:}has\_lb\ Un)\ (eps{:}R),$
$0 < eps \to \exists\ k : nat, Rabs\ (minorant\ Un\ pr\ -\ Un\ k) < eps.$

Unicity of limit for convergent sequences
Lemma *UL_sequence* :
$\forall\ (Un{:}nat \to R)\ (l1\ l2{:}R), Un\_cv\ Un\ l1 \to Un\_cv\ Un\ l2 \to l1 = l2.$

Lemma *CV_plus* :
$\forall\ (An\ Bn{:}nat \to R)\ (l1\ l2{:}R),$
$Un\_cv\ An\ l1 \to Un\_cv\ Bn\ l2 \to Un\_cv\ (\mathsf{fun}\ i{:}nat \Rightarrow An\ i + Bn\ i)\ (l1 + l2).$

Lemma *cv_cvabs* :
$\forall\ (Un{:}nat \to R)\ (l{:}R),$
$Un\_cv\ Un\ l \to Un\_cv\ (\mathsf{fun}\ i{:}nat \Rightarrow Rabs\ (Un\ i))\ (Rabs\ l).$

Lemma *CV_Cauchy* :
$\forall\ Un{:}nat \to R, sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ Un\ l) \to Cauchy\_crit\ Un.$

Lemma *maj_by_pos* :
$\forall\ Un{:}nat \to R,$
$sigT\ (\mathsf{fun}\ l{:}R \Rightarrow Un\_cv\ Un\ l) \to$
$\exists\ l : R, 0 < l \wedge (\forall\ n{:}nat, Rabs\ (Un\ n) \leq l).$

Lemma *CV_mult* :
$\forall\ (An\ Bn{:}nat \to R)\ (l1\ l2{:}R),$
$Un\_cv\ An\ l1 \to Un\_cv\ Bn\ l2 \to Un\_cv\ (\mathsf{fun}\ i{:}nat \Rightarrow An\ i \times Bn\ i)\ (l1 \times l2).$

Lemma *tech9* :
$\forall\ Un{:}nat \to R,$
$Un\_growing\ Un \to \forall\ m\ n{:}nat, (m \leq n)\%nat \to Un\ m \leq Un\ n.$

Lemma *tech10* :
$\forall\ (Un{:}nat \to R)\ (x{:}R), Un\_growing\ Un \to is\_lub\ (EUn\ Un)\ x \to Un\_cv\ Un\ x.$

Lemma *tech13* :
$\forall\ (An{:}nat \to R)\ (k{:}R),$
$0 \leq k < 1 \to$
$Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow Rabs\ (An\ (S\ n)\ /\ An\ n))\ k \to$
$\exists\ k0 : R,$
$k < k0 < 1 \wedge$
$(\exists\ N : nat,$
$(\forall\ n{:}nat, (N \leq n)\%nat \to Rabs\ (An\ (S\ n)\ /\ An\ n) < k0)).$

Lemma *growing_ineq* :
$\forall\ (Un{:}nat \to R)\ (l{:}R),$
$Un\_growing\ Un \to Un\_cv\ Un\ l \to \forall\ n{:}nat, Un\ n \leq l.$

Un->l => (-Un) -> (-l)

Lemma *CV_opp* :
∀ (*An*:*nat* → *R*) (*l*:*R*), *Un_cv An l* → *Un_cv* (*opp_seq An*) (- *l*).

Lemma *decreasing_ineq* :
∀ (*Un*:*nat* → *R*) (*l*:*R*),
Un_decreasing Un → *Un_cv Un l* → ∀ *n*:*nat*, *l* ≤ *Un n*.

Lemma *CV_minus* :
∀ (*An Bn*:*nat* → *R*) (*l1 l2*:*R*),
Un_cv An l1 → *Un_cv Bn l2* → *Un_cv* (fun *i*:*nat* ⇒ *An i* - *Bn i*) (*l1* - *l2*).

Un -> +oo
Definition *cv_infty* (*Un*:*nat* → *R*) : Prop :=
∀ *M*:*R*, ∃ *N* : *nat*, (∀ *n*:*nat*, (*N* ≤ *n*)%*nat* → *M* < *Un n*).

Un -> +oo => /Un -> O
Lemma *cv_infty_cv_R0* :
∀ *Un*:*nat* → *R*,
(∀ *n*:*nat*, *Un n* ≠ 0) → *cv_infty Un* → *Un_cv* (fun *n*:*nat* ⇒ / *Un n*) 0.

Lemma *decreasing_prop* :
∀ (*Un*:*nat* → *R*) (*m n*:*nat*),
Un_decreasing Un → (*m* ≤ *n*)%*nat* → *Un n* ≤ *Un m*.

|x|^n/n! -> 0
Lemma *cv_speed_pow_fact* :
∀ *x*:*R*, *Un_cv* (fun *n*:*nat* ⇒ *x* ^ *n* / *INR* (*fact n*)) 0.

Chapter 139

Module Coq.Reals.SeqSeries

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Max*.
Require Export *Rseries*.
Require Export *SeqProp*.
Require Export *Rcomplete*.
Require Export *PartSum*.
Require Export *AltSeries*.
Require Export *Binomial*.
Require Export *Rsigma*.
Require Export *Rprod*.
Require Export *Cauchy_prod*.
Require Export *Alembert*.
Open Local *Scope R_scope*.

Lemma *sum_maj1* :
  ∀ (*fn*:*nat* → *R* → *R*) (*An*:*nat* → *R*) (*x l1 l2*:*R*)
    (*N*:*nat*),
    Un_cv (fun *n*:*nat* ⇒ *SP fn n x*) *l1* →
    Un_cv (fun *n*:*nat* ⇒ *sum_f_R0 An n*) *l2* →
    (∀ *n*:*nat*, *Rabs* (*fn n x*) ≤ *An n*) →
    Rabs (*l1* - *SP fn N x*) ≤ *l2* - *sum_f_R0 An N*.

Comparaison of convergence for series
Lemma *Rseries_CV_comp* :
  ∀ *An Bn*:*nat* → *R*,
    (∀ *n*:*nat*, 0 ≤ *An n* ≤ *Bn n*) →
    sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *sum_f_R0 Bn N*) *l*) →
    sigT (fun *l*:*R* ⇒ *Un_cv* (fun *N*:*nat* ⇒ *sum_f_R0 An N*) *l*).

Cesaro's theorem
Lemma *Cesaro* :
  ∀ (*An Bn*:*nat* → *R*) (*l*:*R*),
    Un_cv Bn l →

$(\forall\ n{:}nat,\ 0 < An\ n) \rightarrow$
$cv\_infty\ (\mathsf{fun}\ n{:}nat \Rightarrow sum\_f\_R0\ An\ n) \rightarrow$
$Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow sum\_f\_R0\ (\mathsf{fun}\ k{:}nat \Rightarrow An\ k \times Bn\ k)\ n\ /\ sum\_f\_R0\ An\ n)$
$l.$

Lemma *Cesaro_1* :
$\forall\ (An{:}nat \rightarrow R)\ (l{:}R),$
$Un\_cv\ An\ l \rightarrow Un\_cv\ (\mathsf{fun}\ n{:}nat \Rightarrow sum\_f\_R0\ An\ (pred\ n)\ /\ INR\ n)\ l.$

Chapter 140

Module Coq.Reals.SplitAbsolu

```
Require Import Rbasic_fun.

Ltac split_case_Rabs :=
  match goal with
    | ⊢ context [(Rcase_abs ?X1)] ⇒
      case (Rcase_abs X1); try split_case_Rabs
  end.

Ltac split_Rabs :=
  match goal with
    | id:context [(Rabs _)] ⊢ _ ⇒ generalize id; clear id; try split_Rabs
    | ⊢ context [(Rabs ?X1)] ⇒
      unfold Rabs in ⊢ ×; try split_case_Rabs; intros
  end.
```

Chapter 141

Module Coq.Reals.SplitRmult

```
Require Import Rbase.

Ltac split_Rmult :=
  match goal with
    | ⊢ ((?X1 × ?X2)%R ≠ 0%R) ⇒
      apply Rmult_integral_contrapositive; split; try split_Rmult
  end.
```

Chapter 142

Module Coq.Reals.Sqrt_reg

Require Import *Rbase*.
Require Import *Rfunctions*.
Require Import *Ranalysis1*.
Require Import *R_sqrt*. *Open* Local *Scope R_scope*.

Lemma *sqrt_var_maj* :
$\forall\ h{:}R,\ Rabs\ h \leq 1 \rightarrow Rabs\ (sqrt\ (1 + h)\ -\ 1) \leq Rabs\ h.$

sqrt is continuous in 1
Lemma *sqrt_continuity_pt_R1* : $continuity\_pt\ sqrt\ 1.$

sqrt is continuous forall x>0
Lemma *sqrt_continuity_pt* : $\forall\ x{:}R,\ 0 < x \rightarrow continuity\_pt\ sqrt\ x.$

sqrt is derivable for all x>0
Lemma *derivable_pt_lim_sqrt* :
$\forall\ x{:}R,\ 0 < x \rightarrow derivable\_pt\_lim\ sqrt\ x\ (/\ (2 \times sqrt\ x)).$

Lemma *derivable_pt_sqrt* : $\forall\ x{:}R,\ 0 < x \rightarrow derivable\_pt\ sqrt\ x.$

Lemma *derive_pt_sqrt* :
$\forall\ (x{:}R)\ (pr{:}0 < x),$
$derive\_pt\ sqrt\ x\ (derivable\_pt\_sqrt\ \_\ pr) = /\ (2 \times sqrt\ x).$

We show that sqrt is continuous for all x>=0
Remark : by definition of sqrt (as extension of Rsqrt on |R), we could also show that sqrt is continuous for all x
Lemma *continuity_pt_sqrt* : $\forall\ x{:}R,\ 0 \leq x \rightarrow continuity\_pt\ sqrt\ x.$

Chapter 143

Module Coq.Lists.ListSet

A Library for finite sets, implemented as lists

List is loaded, but not exported. This allow to "hide" the definitions, functions and theorems of List and to see only the ones of ListSet

```
Require Import List.

Section first_definitions.

  Variable A : Set.
  Hypothesis Aeq_dec : ∀ x y:A, {x = y} + {x ≠ y}.

  Definition set := list A.

  Definition empty_set : set := nil.

  Fixpoint set_add (a:A) (x:set) {struct x} : set :=
    match x with
    | nil ⇒ a :: nil
    | a1 :: x1 ⇒
        match Aeq_dec a a1 with
        | left _ ⇒ a1 :: x1
        | right _ ⇒ a1 :: set_add a x1
        end
    end.

  Fixpoint set_mem (a:A) (x:set) {struct x} : bool :=
    match x with
    | nil ⇒ false
    | a1 :: x1 ⇒
        match Aeq_dec a a1 with
        | left _ ⇒ true
        | right _ ⇒ set_mem a x1
        end
    end.
```

If a belongs to x, removes a from x. If not, does nothing

```
Fixpoint set_remove (a:A) (x:set) {struct x} : set :=
  match x with
  | nil ⇒ empty_set
  | a1 :: x1 ⇒
      match Aeq_dec a a1 with
      | left _ ⇒ x1
      | right _ ⇒ a1 :: set_remove a x1
      end
  end.

Fixpoint set_inter (x:set) : set → set :=
  match x with
  | nil ⇒ fun y ⇒ nil
  | a1 :: x1 ⇒
      fun y ⇒
        if set_mem a1 y then a1 :: set_inter x1 y else set_inter x1 y
  end.

Fixpoint set_union (x y:set) {struct y} : set :=
  match y with
  | nil ⇒ x
  | a1 :: y1 ⇒ set_add a1 (set_union x y1)
  end.
```

returns the set of all els of x that does not belong to y

```
Fixpoint set_diff (x y:set) {struct x} : set :=
  match x with
  | nil ⇒ nil
  | a1 :: x1 ⇒
      if set_mem a1 y then set_diff x1 y else set_add a1 (set_diff x1 y)
  end.

Definition set_In : A → set → Prop := In (A:=A).

Lemma set_In_dec : ∀ (a:A) (x:set), {set_In a x} + {~ set_In a x}.

Lemma set_mem_ind :
 ∀ (B:Set) (P:B → Prop) (y z:B) (a:A) (x:set),
   (set_In a x → P y) → P z → P (if set_mem a x then y else z).

Lemma set_mem_ind2 :
 ∀ (B:Set) (P:B → Prop) (y z:B) (a:A) (x:set),
   (set_In a x → P y) →
   (~ set_In a x → P z) → P (if set_mem a x then y else z).

Lemma set_mem_correct1 :
 ∀ (a:A) (x:set), set_mem a x = true → set_In a x.

Lemma set_mem_correct2 :
```

$\forall\ (a{:}A)\ (x{:}set),\ set\_In\ a\ x \rightarrow set\_mem\ a\ x = true.$

Lemma *set_mem_complete1* :
$\forall\ (a{:}A)\ (x{:}set),\ set\_mem\ a\ x = false \rightarrow \neg\ set\_In\ a\ x.$

Lemma *set_mem_complete2* :
$\forall\ (a{:}A)\ (x{:}set),\ \neg\ set\_In\ a\ x \rightarrow set\_mem\ a\ x = false.$

Lemma *set_add_intro1* :
$\forall\ (a\ b{:}A)\ (x{:}set),\ set\_In\ a\ x \rightarrow set\_In\ a\ (set\_add\ b\ x).$

Lemma *set_add_intro2* :
$\forall\ (a\ b{:}A)\ (x{:}set),\ a = b \rightarrow set\_In\ a\ (set\_add\ b\ x).$

Hint *Resolve set_add_intro1 set_add_intro2.*

Lemma *set_add_intro* :
$\forall\ (a\ b{:}A)\ (x{:}set),\ a = b \vee set\_In\ a\ x \rightarrow set\_In\ a\ (set\_add\ b\ x).$

Lemma *set_add_elim* :
$\forall\ (a\ b{:}A)\ (x{:}set),\ set\_In\ a\ (set\_add\ b\ x) \rightarrow a = b \vee set\_In\ a\ x.$

Lemma *set_add_elim2* :
$\forall\ (a\ b{:}A)\ (x{:}set),\ set\_In\ a\ (set\_add\ b\ x) \rightarrow a \neq b \rightarrow set\_In\ a\ x.$

Hint *Resolve set_add_intro set_add_elim set_add_elim2.*

Lemma *set_add_not_empty* : $\forall\ (a{:}A)\ (x{:}set),\ set\_add\ a\ x \neq empty\_set.$

Lemma *set_union_intro1* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ x \rightarrow set\_In\ a\ (set\_union\ x\ y).$

Lemma *set_union_intro2* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ y \rightarrow set\_In\ a\ (set\_union\ x\ y).$

Hint *Resolve set_union_intro2 set_union_intro1.*

Lemma *set_union_intro* :
$\forall\ (a{:}A)\ (x\ y{:}set),$
$set\_In\ a\ x \vee set\_In\ a\ y \rightarrow set\_In\ a\ (set\_union\ x\ y).$

Lemma *set_union_elim* :
$\forall\ (a{:}A)\ (x\ y{:}set),$
$set\_In\ a\ (set\_union\ x\ y) \rightarrow set\_In\ a\ x \vee set\_In\ a\ y.$

Lemma *set_union_emptyL* :
$\forall\ (a{:}A)\ (x{:}set),\ set\_In\ a\ (set\_union\ empty\_set\ x) \rightarrow set\_In\ a\ x.$

Lemma *set_union_emptyR* :
$\forall\ (a{:}A)\ (x{:}set),\ set\_In\ a\ (set\_union\ x\ empty\_set) \rightarrow set\_In\ a\ x.$

Lemma *set_inter_intro* :
$\forall\ (a{:}A)\ (x\ y{:}set),$
$set\_In\ a\ x \rightarrow set\_In\ a\ y \rightarrow set\_In\ a\ (set\_inter\ x\ y).$

Lemma *set_inter_elim1* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ (set\_inter\ x\ y) \to set\_In\ a\ x.$

Lemma *set_inter_elim2* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ (set\_inter\ x\ y) \to set\_In\ a\ y.$

Hint *Resolve set_inter_elim1 set_inter_elim2.*

Lemma *set_inter_elim* :
$\forall\ (a{:}A)\ (x\ y{:}set),$
$set\_In\ a\ (set\_inter\ x\ y) \to set\_In\ a\ x \wedge set\_In\ a\ y.$

Lemma *set_diff_intro* :
$\forall\ (a{:}A)\ (x\ y{:}set),$
$set\_In\ a\ x \to \neg\ set\_In\ a\ y \to set\_In\ a\ (set\_diff\ x\ y).$

Lemma *set_diff_elim1* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ (set\_diff\ x\ y) \to set\_In\ a\ x.$

Lemma *set_diff_elim2* :
$\forall\ (a{:}A)\ (x\ y{:}set),\ set\_In\ a\ (set\_diff\ x\ y) \to \neg\ set\_In\ a\ y.$

Lemma *set_diff_trivial* : $\forall\ (a{:}A)\ (x{:}set),\ \neg\ set\_In\ a\ (set\_diff\ x\ x).$

Hint *Resolve set_diff_intro set_diff_trivial.*

End *first_definitions.*

Section *other_definitions.*

Variables $A\ B$: Set.

Definition *set_prod* : $set\ A \to set\ B \to set\ (A \times B) :=$
$list\_prod\ (A{:=}A)\ (B{:=}B).$

$B\hat{}A$, set of applications from A to B
Definition *set_power* : $set\ A \to set\ B \to set\ (set\ (A \times B)) :=$
$list\_power\ (A{:=}A)\ (B{:=}B).$

Definition *set_map* : $(A \to B) \to set\ A \to set\ B := map\ (A{:=}A)\ (B{:=}B).$

Definition *set_fold_left* : $(B \to A \to B) \to set\ A \to B \to B :=$
$fold\_left\ (A{:=}B)\ (B{:=}A).$

Definition *set_fold_right* $(f{:}A \to B \to B)\ (x{:}set\ A)$
$(b{:}B) : B := fold\_right\ f\ b\ x.$

End *other_definitions.*

Chapter 144

Module Coq.Lists.ListTactics

```
Require Import BinPos.
Require Import List.

Ltac list_fold_right fcons fnil l :=
  match l with
  | (cons ?x ?tl) ⇒ fcons x ltac:(list_fold_right fcons fnil tl)
  | nil ⇒ fnil
  end.

Ltac lazy_list_fold_right fcons fnil l :=
  match l with
  | (cons ?x ?tl) ⇒
        let cont := lazy_list_fold_right fcons fnil in
        fcons x cont tl
  | nil ⇒ fnil
  end.

Ltac list_fold_left fcons fnil l :=
  match l with
  | (cons ?x ?tl) ⇒ list_fold_left fcons ltac:(fcons x fnil) tl
  | nil ⇒ fnil
  end.

Ltac list_iter f l :=
  match l with
  | (cons ?x ?tl) ⇒ f x; list_iter f tl
  | nil ⇒ idtac
  end.

Ltac list_iter_gen seq f l :=
  match l with
  | (cons ?x ?tl) ⇒
      let t1 _ := f x in
      let t2 _ := list_iter_gen seq f tl in
      seq t1 t2
```

```
  | nil ⇒ idtac
  end.

Ltac AddFvTail a l :=
 match l with
 | nil ⇒ constr:(cons a l)
 | (cons a _) ⇒ l
 | (cons ?x ?l) ⇒ let l' := AddFvTail a l in constr:(cons x l')
 end.

Ltac Find_at a l :=
 let rec find n l :=
    match l with
    | nil ⇒ fail 100 "anomaly: Find_at"
    | (cons a _) ⇒ eval compute in n
    | (cons _ ?l) ⇒ find (Psucc n) l
    end
 in find 1%positive l.

Ltac check_is_list t :=
  match t with
  | cons _ ?l ⇒ check_is_list l
  | nil ⇒ idtac
  | _ ⇒ fail 100 "anomaly: failed to build a canonical list"
  end.

Ltac check_fv l :=
  check_is_list l;
  match type of l with
  | list _ ⇒ idtac
  | _ ⇒ fail 100 "anomaly: built an ill-typed list"
  end.
```

Chapter 145

Module Coq.Lists.List

Require Import *Le Gt Minus Min Bool.*
Require Import *Setoid.*

145.1 Basics: definition of polymorphic lists and some operations

145.1.1 Definitions

Section *Lists.*

Variable A : Type.

Inductive *list* : Type :=
| *nil* : *list*
| *cons* : $A \rightarrow$ *list* $\rightarrow$ *list.*

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

Open Scope list_scope.

Head and tail

Definition *head* (*l*:*list*) :=
match *l* with
| *nil* $\Rightarrow$ *error*
| *x* :: _ $\Rightarrow$ *value x*
end.

Definition *hd* (*default*:*A*) (*l*:*list*) :=
match *l* with
| *nil* $\Rightarrow$ *default*
| *x* :: _ $\Rightarrow$ *x*
end.

Definition *tail* (*l*:*list*) : *list* :=
match *l* with

```
    | nil ⇒ nil
    | a :: m ⇒ m
  end.
```

Length of lists

```
  Fixpoint length (l:list) : nat :=
    match l with
      | nil ⇒ 0
      | _ :: m ⇒ S (length m)
    end.
```

The *In* predicate

```
  Fixpoint In (a:A) (l:list) {struct l} : Prop :=
    match l with
      | nil ⇒ False
      | b :: m ⇒ b = a ∨ In a m
    end.
```

Concatenation of two lists

```
  Fixpoint app (l m:list) {struct l} : list :=
    match l with
      | nil ⇒ m
      | a :: l1 ⇒ a :: app l1 m
    end.

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

End Lists.
```

Exporting list notations and tactics

```
Implicit Arguments nil [A].
Infix "::" := cons (at level 60, right associativity) : list_scope.
Infix "++" := app (right associativity, at level 60) : list_scope.

Ltac now_show c := change c in ⊢ ×.

Open Scope list_scope.

Delimit Scope list_scope with list.
```

145.1.2 Facts about lists

```
Section Facts.

  Variable A : Type.
```

Genereric facts

Discrimination

Theorem *nil_cons* : $\forall$ (*x*:*A*) (*l*:*list A*), *nil* $\neq$ *x* :: *l*.

Destruction

Theorem *destruct_list* : $\forall$ *l* : *list A*, {*x*:*A* & {*tl*:*list A* | *l* = *x*::*tl*}}+{*l* = *nil*}.

Head and tail

Theorem *head_nil* : *head* (@*nil A*) = *None*.

Theorem *head_cons* : $\forall$ (*l* : *list A*) (*x* : *A*), *head* (*x*::*l*) = *Some x*.

Facts about *In*

Characterization of *In*

Theorem *in_eq* : $\forall$ (*a*:*A*) (*l*:*list A*), *In a* (*a* :: *l*).

Theorem *in_cons* : $\forall$ (*a b*:*A*) (*l*:*list A*), *In b l* $\rightarrow$ *In b* (*a* :: *l*).

Theorem *in_nil* : $\forall$ *a*:*A*, $\neg$ *In a nil*.

Lemma *In_split* : $\forall$ *x* (*l*:*list A*), *In x l* $\rightarrow$ $\exists$ *l1*, $\exists$ *l2*, *l* = *l1*++*x*::*l2*.

Inversion

Theorem *in_inv* : $\forall$ (*a b*:*A*) (*l*:*list A*), *In b* (*a* :: *l*) $\rightarrow$ *a* = *b* $\vee$ *In b l*.

Decidability of *In*

Theorem *In_dec* :
($\forall$ *x y*:*A*, {*x* = *y*} + {*x* $\neq$ *y*}) $\rightarrow$
$\forall$ (*a*:*A*) (*l*:*list A*), {*In a l*} + {~ *In a l*}.

Facts about *app*

Discrimination

Theorem *app_cons_not_nil* : $\forall$ (*x y*:*list A*) (*a*:*A*), *nil* $\neq$ *x* ++ *a* :: *y*.

Concat with *nil*

Theorem *app_nil_end* : $\forall$ *l*:*list A*, *l* = *l* ++ *nil*.

app is associative

Theorem *app_ass* : $\forall$ *l m n*:*list A*, (*l* ++ *m*) ++ *n* = *l* ++ *m* ++ *n*.
Hint *Resolve app_ass*.

Theorem *ass_app* : $\forall$ *l m n*:*list A*, *l* ++ *m* ++ *n* = (*l* ++ *m*) ++ *n*.

app commutes with *cons*

Theorem *app_comm_cons* : $\forall$ (*x y*:*list A*) (*a*:*A*), *a* :: (*x* ++ *y*) = (*a* :: *x*) ++ *y*.

Facts deduced from the result of a concatenation

Theorem *app_eq_nil* : $\forall\ l\ l'$:*list* A, $l\ ++\ l' = nil \rightarrow l = nil \wedge l' = nil$.

Theorem *app_eq_unit* :
$\forall\ (x\ y$:*list* $A)\ (a$:$A)$,
$x\ ++\ y = a :: nil \rightarrow x = nil \wedge y = a :: nil \vee x = a :: nil \wedge y = nil$.

Lemma *app_inj_tail* :
$\forall\ (x\ y$:*list* $A)\ (a\ b$:$A)$, $x\ ++\ a :: nil = y\ ++\ b :: nil \rightarrow x = y \wedge a = b$.

Compatibility wtih other operations

Lemma *app_length* : $\forall\ l\ l'$: *list* A, *length* $(l{+}{+}l') =$ *length* $l +$ *length* l'.

Lemma *in_app_or* : $\forall\ (l\ m$:*list* $A)\ (a$:$A)$, *In* $a\ (l\ ++\ m) \rightarrow$ *In* $a\ l \vee$ *In* $a\ m$.

Lemma *in_or_app* : $\forall\ (l\ m$:*list* $A)\ (a$:$A)$, *In* $a\ l \vee$ *In* $a\ m \rightarrow$ *In* $a\ (l\ ++\ m)$.

End *Facts*.

Hint *Resolve app_nil_end ass_app app_ass*: *datatypes v62*.
Hint *Resolve app_comm_cons app_cons_not_nil*: *datatypes v62*.
Hint Immediate *app_eq_nil*: *datatypes v62*.
Hint *Resolve app_eq_unit app_inj_tail*: *datatypes v62*.
Hint *Resolve in_eq in_cons in_inv in_nil in_app_or in_or_app*: *datatypes v62*.

145.2 Operations on the elements of a list

Section *Elts*.

Variable A : Type.

145.2.1 Nth element of a list

Fixpoint *nth* $(n$:*nat*$)\ (l$:*list* $A)\ (default$:$A)\ \{struct\ l\}$: A :=
match n, l with
| $O, x :: l' \Rightarrow x$
| $O, other \Rightarrow default$
| $S\ m, nil \Rightarrow default$
| $S\ m, x :: t \Rightarrow nth\ m\ t\ default$
end.

Fixpoint *nth_ok* $(n$:*nat*$)\ (l$:*list* $A)\ (default$:$A)\ \{struct\ l\}$: *bool* :=
match n, l with
| $O, x :: l' \Rightarrow true$
| $O, other \Rightarrow false$
| $S\ m, nil \Rightarrow false$
| $S\ m, x :: t \Rightarrow nth\_ok\ m\ t\ default$
end.

Lemma *nth_in_or_default* :
$\forall\ (n$:*nat*$)\ (l$:*list* $A)\ (d$:$A)$, $\{In\ (nth\ n\ l\ d)\ l\} + \{nth\ n\ l\ d = d\}$.

```
Lemma nth_S_cons :
  ∀ (n:nat) (l:list A) (d a:A),
    In (nth n l d) l → In (nth (S n) (a :: l) d) (a :: l).

Fixpoint nth_error (l:list A) (n:nat) {struct n} : Exc A :=
  match n, l with
    | O, x :: _ ⇒ value x
    | S n, _ :: l ⇒ nth_error l n
    | _, _ ⇒ error
  end.

Definition nth_default (default:A) (l:list A) (n:nat) : A :=
  match nth_error l n with
    | Some x ⇒ x
    | None ⇒ default
  end.

Lemma nth_In :
  ∀ (n:nat) (l:list A) (d:A), n < length l → In (nth n l d) l.

Lemma nth_overflow : ∀ l n d, length l ≤ n → nth n l d = d.

Lemma nth_indep :
  ∀ l n d d', n < length l → nth n l d = nth n l d'.

Lemma app_nth1 :
  ∀ l l' d n, n < length l → nth n (l++l') d = nth n l d.

Lemma app_nth2 :
  ∀ l l' d n, n ≥ length l → nth n (l++l') d = nth (n-length l) l' d.
```

145.2.2 Remove

```
Section Remove.

  Hypothesis eq_dec : ∀ x y : A, {x = y}+{x ≠ y}.

  Fixpoint remove (x : A) (l : list A){struct l} : list A :=
    match l with
      | nil ⇒ nil
      | y::tl ⇒ if (eq_dec x y) then remove x tl else y::(remove x tl)
    end.

  Theorem remove_In : ∀ (l : list A) (x : A), ¬ In x (remove x l).

End Remove.
```

145.2.3 Last element of a list

last l d returns the last element of the list *l*, or the default value *d* if *l* is empty.

```
Fixpoint last (l:list A) (d:A) {struct l} : A :=
```

```
  match l with
    | nil ⇒ d
    | a :: nil ⇒ a
    | a :: l ⇒ last l d
  end.
```

removelast l remove the last element of *l*

```
  Fixpoint removelast (l:list A) {struct l} : list A :=
    match l with
      | nil ⇒ nil
      | a :: nil ⇒ nil
      | a :: l ⇒ a :: removelast l
    end.

  Lemma app_removelast_last :
    ∀ l d, l≠nil → l = removelast l ++ (last l d :: nil).

  Lemma exists_last :
    ∀ l, l≠nil → { l' : (list A) & { a : A | l = l'++a::nil}}.
```

145.2.4 Counting occurences of a element

```
  Hypotheses eqA_dec : ∀ x y : A, {x = y}+{x ≠ y}.

  Fixpoint count_occ (l : list A) (x : A){struct l} : nat :=
    match l with
      | nil ⇒ 0
      | y :: tl ⇒
        let n := count_occ tl x in
          if eqA_dec y x then S n else n
    end.
```

Compatibility of count_occ with operations on list

```
  Theorem count_occ_In : ∀ (l : list A) (x : A), In x l ↔ count_occ l x > 0.

  Theorem count_occ_inv_nil : ∀ (l : list A), (∀ x:A, count_occ l x = 0) ↔ l = nil.

  Lemma count_occ_nil : ∀ (x : A), count_occ nil x = 0.

  Lemma count_occ_cons_eq : ∀ (l : list A) (x y : A), x = y → count_occ (x::l) y = S (count_occ
l y).

  Lemma count_occ_cons_neq : ∀ (l : list A) (x y : A), x ≠ y → count_occ (x::l) y = count_occ l
y.

End Elts.
```

145.3 Manipulating whole lists

```
Section ListOps.

  Variable A : Type.
```

145.3.1 Reverse

```
Fixpoint rev (l:list A) : list A :=
  match l with
    | nil ⇒ nil
    | x :: l' ⇒ rev l' ++ x :: nil
  end.

Lemma distr_rev : ∀ x y:list A, rev (x ++ y) = rev y ++ rev x.

Remark rev_unit : ∀ (l:list A) (a:A), rev (l ++ a :: nil) = a :: rev l.

Lemma rev_involutive : ∀ l:list A, rev (rev l) = l.
```

Compatibility with other operations

```
Lemma In_rev : ∀ l x, In x l ↔ In x (rev l).

Lemma rev_length : ∀ l, length (rev l) = length l.

Lemma rev_nth : ∀ l d n, n < length l →
  nth n (rev l) d = nth (length l - S n) l d.
```

An alternative tail-recursive definition for reverse

```
Fixpoint rev_append (l l': list A) {struct l} : list A :=
  match l with
    | nil ⇒ l'
    | a::l ⇒ rev_append l (a::l')
  end.

Definition rev' l : list A := rev_append l nil.

Notation rev_acc := rev_append (only parsing).

Lemma rev_append_rev : ∀ l l', rev_acc l l' = rev l ++ l'.

Notation rev_acc_rev := rev_append_rev (only parsing).

Lemma rev_alt : ∀ l, rev l = rev_append l nil.
```

Reverse Induction Principle on Lists

```
Section Reverse_Induction.

  Lemma rev_list_ind :
    ∀ P:list A→ Prop,
      P nil →
      (∀ (a:A) (l:list A), P (rev l) → P (rev (a :: l))) →
      ∀ l:list A, P (rev l).

  Theorem rev_ind :
    ∀ P:list A → Prop,
      P nil →
      (∀ (x:A) (l:list A), P l → P (l ++ x :: nil)) → ∀ l:list A, P l.

End Reverse_Induction.
```

145.3.2 Lists modulo permutation

Section *Permutation*.

Inductive *Permutation* : *list A* → *list A* → Prop :=
| *perm_nil*: *Permutation nil nil*
| *perm_skip*: ∀ (*x*:*A*) (*l l'*:*list A*), *Permutation l l'* → *Permutation* (*cons x l*) (*cons x l'*)
| *perm_swap*: ∀ (*x y*:*A*) (*l*:*list A*), *Permutation* (*cons y* (*cons x l*)) (*cons x* (*cons y l*))
| *perm_trans*: ∀ (*l l' l''*:*list A*), *Permutation l l'* → *Permutation l' l''* → *Permutation l l''*.

Hint *Constructors Permutation*.

Some facts about *Permutation*

Theorem *Permutation_nil* : ∀ (*l* : *list A*), *Permutation nil l* → *l* = *nil*.

Theorem *Permutation_nil_cons* : ∀ (*l* : *list A*) (*x* : *A*), ¬ *Permutation nil* (*x*::*l*).

Permutation over lists is a equivalence relation

Theorem *Permutation_refl* : ∀ *l* : *list A*, *Permutation l l*.

Theorem *Permutation_sym* : ∀ *l l'* : *list A*, *Permutation l l'* → *Permutation l' l*.

Theorem *Permutation_trans* : ∀ *l l' l''* : *list A*, *Permutation l l'* → *Permutation l' l''* → *Permutation l l''*.

Hint *Resolve Permutation_refl Permutation_sym Permutation_trans*.

Compatibility with others operations on lists

Theorem *Permutation_in* : ∀ (*l l'* : *list A*) (*x* : *A*), *Permutation l l'* → *In x l* → *In x l'*.

Lemma *Permutation_app_tail* : ∀ (*l l' tl* : *list A*), *Permutation l l'* → *Permutation* (*l*++*tl*) (*l'*++*tl*).

Lemma *Permutation_app_head* : ∀ (*l tl tl'* : *list A*), *Permutation tl tl'* → *Permutation* (*l*++*tl*) (*l*++*tl'*).

Theorem *Permutation_app* : ∀ (*l m l' m'* : *list A*), *Permutation l l'* → *Permutation m m'* → *Permutation* (*l*++*m*) (*l'*++*m'*).

Theorem *Permutation_app_swap* : ∀ (*l l'* : *list A*), *Permutation* (*l*++*l'*) (*l'*++*l*).

Theorem *Permutation_cons_app* : ∀ (*l l1 l2*:*list A*) *a*,
Permutation l (*l1* ++ *l2*) → *Permutation* (*a* :: *l*) (*l1* ++ *a* :: *l2*).
Hint *Resolve Permutation_cons_app*.

Theorem *Permutation_length* : ∀ (*l l'* : *list A*), *Permutation l l'* → *length l* = *length l'*.

Theorem *Permutation_rev* : ∀ (*l* : *list A*), *Permutation l* (*rev l*).

Theorem *Permutation_ind_bis* :
∀ *P* : *list A* → *list A* → Prop,
P (@*nil A*) (@*nil A*) →
(∀ *x l l'*, *Permutation l l'* → *P l l'* → *P* (*x* :: *l*) (*x* :: *l'*)) →
(∀ *x y l l'*, *Permutation l l'* → *P l l'* → *P* (*y* :: *x* :: *l*) (*x* :: *y* :: *l'*)) →

$(\forall\ l\ l'\ l'', Permutation\ l\ l' \to P\ l\ l' \to Permutation\ l'\ l'' \to P\ l'\ l'' \to P\ l\ l'') \to$
$\forall\ l\ l', Permutation\ l\ l' \to P\ l\ l'.$

Ltac *break_list l x l' H* :=
destruct l as [|x l']; *simpl* in ×;
injection H; *intros*; *subst*; *clear H*.

Theorem *Permutation_app_inv* : $\forall\ (l1\ l2\ l3\ l4{:}list\ A)\ a,$
$Permutation\ (l1{+}{+}a{::}l2)\ (l3{+}{+}a{::}l4) \to Permutation\ (l1{+}{+}l2)\ (l3\ {+}{+}\ l4).$

Theorem *Permutation_cons_inv* :
$\forall\ l\ l'\ a, Permutation\ (a{::}l)\ (a{::}l') \to Permutation\ l\ l'.$

Theorem *Permutation_cons_app_inv* :
$\forall\ l\ l1\ l2\ a, Permutation\ (a\ ::\ l)\ (l1\ {+}{+}\ a\ ::\ l2) \to Permutation\ l\ (l1\ {+}{+}\ l2).$

Theorem *Permutation_app_inv_l* :
$\forall\ l\ l1\ l2, Permutation\ (l\ {+}{+}\ l1)\ (l\ {+}{+}\ l2) \to Permutation\ l1\ l2.$

Theorem *Permutation_app_inv_r* :
$\forall\ l\ l1\ l2, Permutation\ (l1\ {+}{+}\ l)\ (l2\ {+}{+}\ l) \to Permutation\ l1\ l2.$

End *Permutation*.

145.3.3 Decidable equality on lists

Hypotheses *eqA_dec* : $\forall\ (x\ y\ :\ A), \{x = y\}{+}\{x \neq y\}.$

Lemma *list_eq_dec* :
$\forall\ l\ l'{:}list\ A, \{l = l'\} + \{l \neq l'\}.$

End *ListOps*.

145.4 Applying functions to the elements of a list

145.4.1 Map

Section *Map*.
Variables $A\ B$: Type.
Variable $f\ :\ A \to B.$

Fixpoint $map\ (l{:}list\ A)\ :\ list\ B$:=
match l with
| $nil \Rightarrow nil$
| $cons\ a\ t \Rightarrow cons\ (f\ a)\ (map\ t)$
end.

Lemma *in_map* :
$\forall\ (l{:}list\ A)\ (x{:}A), In\ x\ l \to In\ (f\ x)\ (map\ l).$

Lemma *in_map_iff* : $\forall\ l\ y, In\ y\ (map\ l) \leftrightarrow \exists\ x, f\ x = y \wedge In\ x\ l.$

Lemma *map_length* : ∀ *l*, *length* (*map l*) = *length l*.

Lemma *map_nth* : ∀ *l d n*,
nth n (*map l*) (*f d*) = *f* (*nth n l d*).

Lemma *map_app* : ∀ *l l'*,
map (*l*++*l'*) = (*map l*)++(*map l'*).

Lemma *map_rev* : ∀ *l*, *map* (*rev l*) = *rev* (*map l*).

Hint *Constructors Permutation*.

Lemma *Permutation_map* :
∀ *l l'*, *Permutation l l'* → *Permutation* (*map l*) (*map l'*).

flat_map

Fixpoint *flat_map* (*f*:*A* → *list B*) (*l*:*list A*) {*struct l*} :
list B :=
match *l* with
| *nil* ⇒ *nil*
| *cons x t* ⇒ (*f x*)++(*flat_map f t*)
end.

Lemma *in_flat_map* : ∀ (*f*:*A*→*list B*)(*l*:*list A*)(*y*:*B*),
In y (*flat_map f l*) ↔ ∃ *x*, *In x l* ∧ *In y* (*f x*).

End *Map*.

Lemma *map_map* : ∀ (*A B C*:Type)(*f*:*A*→*B*)(*g*:*B*→*C*) *l*,
map g (*map f l*) = *map* (fun *x* ⇒ *g* (*f x*)) *l*.

Lemma *map_ext* :
∀ (*A B* : Type)(*f g*:*A*→*B*), (∀ *a*, *f a* = *g a*) → ∀ *l*, *map f l* = *map g l*.

Left-to-right iterator on lists
Section *Fold_Left_Recursor*.
Variables *A B* : Type.
Variable *f* : *A* → *B* → *A*.

Fixpoint *fold_left* (*l*:*list B*) (*a0*:*A*) {*struct l*} : *A* :=
match *l* with
| *nil* ⇒ *a0*
| *cons b t* ⇒ *fold_left t* (*f a0 b*)
end.

Lemma *fold_left_app* : ∀ (*l l'*:*list B*)(*i*:*A*),
fold_left (*l*++*l'*) *i* = *fold_left l'* (*fold_left l i*).

End *Fold_Left_Recursor*.

Lemma *fold_left_length* :
∀ (*A*:Type)(*l*:*list A*), *fold_left* (fun *x* _ ⇒ *S x*) *l* 0 = *length l*.

Right-to-left iterator on lists

```
Section Fold_Right_Recursor.
  Variables A B : Type.
  Variable f : B → A → A.
  Variable a0 : A.

  Fixpoint fold_right (l:list B) : A :=
    match l with
      | nil ⇒ a0
      | cons b t ⇒ f b (fold_right t)
    end.

End Fold_Right_Recursor.

  Lemma fold_right_app : ∀ (A B:Type)(f:A→B→B) l l' i,
    fold_right f i (l++l') = fold_right f (fold_right f i l') l.

  Lemma fold_left_rev_right : ∀ (A B:Type)(f:A→B→B) l i,
    fold_right f i (rev l) = fold_left (fun x y ⇒ f y x) l i.

  Theorem fold_symmetric :
    ∀ (A:Type) (f:A → A → A),
      (∀ x y z:A, f x (f y z) = f (f x y) z) →
      (∀ x y:A, f x y = f y x) →
      ∀ (a0:A) (l:list A), fold_left f l a0 = fold_right f a0 l.
```

(*list_power x y*) is y^x, or the set of sequences of elts of y indexed by elts of x, sorted in lexicographic order.

```
  Fixpoint list_power (A B:Type)(l:list A) (l':list B) {struct l} :
    list (list (A × B)) :=
    match l with
      | nil ⇒ cons nil nil
      | cons x t ⇒
        flat_map (fun f:list (A × B) ⇒ map (fun y:B ⇒ cons (x, y) f) l')
        (list_power t l')
    end.
```

145.4.2 Boolean operations over lists

```
  Section Bool.
    Variable A : Type.
    Variable f : A → bool.
```

find whether a boolean function can be satisfied by an elements of the list.

```
    Fixpoint existsb (l:list A) {struct l}: bool :=
      match l with
        | nil ⇒ false
        | a::l ⇒ f a || existsb l
      end.
```

Lemma *existsb_exists* :
$\forall\ l,\ existsb\ l = true \leftrightarrow \exists\ x,\ In\ x\ l \wedge f\ x = true.$

Lemma *existsb_nth* : $\forall\ l\ n\ d,\ n < length\ l \rightarrow$
$existsb\ l = false \rightarrow f\ (nth\ n\ l\ d) = false.$

find whether a boolean function is satisfied by all the elements of a list.

Fixpoint *forallb* (*l*:*list A*) {*struct l*} : *bool* :=
match *l* with
| *nil* ⇒ *true*
| *a*::*l* ⇒ *f a* && *forallb l*
end.

Lemma *forallb_forall* :
$\forall\ l,\ forallb\ l = true \leftrightarrow (\forall\ x,\ In\ x\ l \rightarrow f\ x = true).$

filter

Fixpoint *filter* (*l*:*list A*) : *list A* :=
match *l* with
| *nil* ⇒ *nil*
| *x* :: *l* ⇒ if *f x* then *x*::(*filter l*) else *filter l*
end.

Lemma *filter_In* : $\forall\ x\ l,\ In\ x\ (filter\ l) \leftrightarrow In\ x\ l \wedge f\ x = true.$

find

Fixpoint *find* (*l*:*list A*) : *option A* :=
match *l* with
| *nil* ⇒ *None*
| *x* :: *tl* ⇒ if *f x* then *Some x* else *find tl*
end.

partition

Fixpoint *partition* (*l*:*list A*) {*struct l*} : *list A* × *list A* :=
match *l* with
| *nil* ⇒ (*nil*, *nil*)
| *x* :: *tl* ⇒ let (*g*,*d*) := *partition tl* in
if *f x* then (*x*::*g*,*d*) else (*g*,*x*::*d*)
end.

End *Bool*.

145.4.3 Operations on lists of pairs or lists of lists

Section *ListPairs*.
Variables *A B* : Type.

split derives two lists from a list of pairs

Fixpoint *split* (*l*:*list* (*A*×*B*)) { *struct l* }: *list A* × *list B* :=

```
  match l with
    | nil ⇒ (nil, nil)
    | (x,y) :: tl ⇒ let (g,d) := split tl in (x::g, y::d)
  end.

Lemma in_split_l : ∀ (l:list (A×B))(p:A×B),
  In p l → In (fst p) (fst (split l)).

Lemma in_split_r : ∀ (l:list (A×B))(p:A×B),
  In p l → In (snd p) (snd (split l)).

Lemma split_nth : ∀ (l:list (A×B))(n:nat)(d:A×B),
  nth n l d = (nth n (fst (split l)) (fst d), nth n (snd (split l)) (snd d)).

Lemma split_length_l : ∀ (l:list (A×B)),
  length (fst (split l)) = length l.

Lemma split_length_r : ∀ (l:list (A×B)),
  length (snd (split l)) = length l.
```

combine is the opposite of *split*. Lists given to *combine* are meant to be of same length. If not, *combine* stops on the shorter list

```
Fixpoint combine (l : list A) (l' : list B){struct l} : list (A×B) :=
  match l,l' with
    | x::tl, y::tl' ⇒ (x,y)::(combine tl tl')
    | _, _ ⇒ nil
  end.

Lemma split_combine : ∀ (l: list (A×B)),
  let (l1,l2) := split l in combine l1 l2 = l.

Lemma combine_split : ∀ (l:list A)(l':list B), length l = length l' →
  split (combine l l') = (l,l').

Lemma in_combine_l : ∀ (l:list A)(l':list B)(x:A)(y:B),
  In (x,y) (combine l l') → In x l.

Lemma in_combine_r : ∀ (l:list A)(l':list B)(x:A)(y:B),
  In (x,y) (combine l l') → In y l'.

Lemma combine_length : ∀ (l:list A)(l':list B),
  length (combine l l') = min (length l) (length l').

Lemma combine_nth : ∀ (l:list A)(l':list B)(n:nat)(x:A)(y:B),
  length l = length l' →
  nth n (combine l l') (x,y) = (nth n l x, nth n l' y).
```

list_prod has the same signature as *combine*, but unlike *combine*, it adds every possible pairs, not only those at the same position.

```
Fixpoint list_prod (l:list A) (l':list B) {struct l} :
  list (A × B) :=
  match l with
```

```
    | nil ⇒ nil
    | cons x t ⇒ (map (fun y:B ⇒ (x, y)) l')++(list_prod t l')
  end.

Lemma in_prod_aux :
  ∀ (x:A) (y:B) (l:list B),
    In y l → In (x, y) (map (fun y0:B ⇒ (x, y0)) l).

Lemma in_prod :
  ∀ (l:list A) (l':list B) (x:A) (y:B),
    In x l → In y l' → In (x, y) (list_prod l l').

Lemma in_prod_iff :
  ∀ (l:list A)(l':list B)(x:A)(y:B),
    In (x,y) (list_prod l l') ↔ In x l ∧ In y l'.

Lemma prod_length : ∀ (l:list A)(l':list B),
  length (list_prod l l') = (length l) × (length l').

End ListPairs.
```

145.5 Miscelenous operations on lists

145.5.1 Length order of lists

```
Section length_order.
  Variable A : Type.

  Definition lel (l m:list A) := length l ≤ length m.

  Variables a b : A.
  Variables l m n : list A.

  Lemma lel_refl : lel l l.

  Lemma lel_trans : lel l m → lel m n → lel l n.

  Lemma lel_cons_cons : lel l m → lel (a :: l) (b :: m).

  Lemma lel_cons : lel l m → lel l (b :: m).

  Lemma lel_tail : lel (a :: l) (b :: m) → lel l m.

  Lemma lel_nil : ∀ l':list A, lel l' nil → nil = l'.
End length_order.

Hint Resolve lel_refl lel_cons_cons lel_cons lel_nil lel_nil nil_cons:
  datatypes v62.
```

145.5.2 Set inclusion on list

Section *SetIncl*.

Variable A : Type.

Definition *incl* $(l\ m{:}list\ A) := \forall\ a{:}A,\ In\ a\ l \to In\ a\ m$.
Hint *Unfold incl*.

Lemma *incl_refl* : $\forall\ l{:}list\ A,\ incl\ l\ l$.
Hint *Resolve incl_refl*.

Lemma *incl_tl* : $\forall\ (a{:}A)\ (l\ m{:}list\ A),\ incl\ l\ m \to incl\ l\ (a :: m)$.
Hint Immediate *incl_tl*.

Lemma *incl_tran* : $\forall\ l\ m\ n{:}list\ A,\ incl\ l\ m \to incl\ m\ n \to incl\ l\ n$.

Lemma *incl_appl* : $\forall\ l\ m\ n{:}list\ A,\ incl\ l\ n \to incl\ l\ (n\ ++\ m)$.
Hint Immediate *incl_appl*.

Lemma *incl_appr* : $\forall\ l\ m\ n{:}list\ A,\ incl\ l\ n \to incl\ l\ (m\ ++\ n)$.
Hint Immediate *incl_appr*.

Lemma *incl_cons* :
$\forall\ (a{:}A)\ (l\ m{:}list\ A),\ In\ a\ m \to incl\ l\ m \to incl\ (a :: l)\ m$.
Hint *Resolve incl_cons*.

Lemma *incl_app* : $\forall\ l\ m\ n{:}list\ A,\ incl\ l\ n \to incl\ m\ n \to incl\ (l\ ++\ m)\ n$.
Hint *Resolve incl_app*.

End *SetIncl*.

Hint *Resolve incl_refl incl_tl incl_tran incl_appl incl_appr incl_cons incl_app*: *datatypes v62*.

Section *Cutting*.

Variable A : Type.

```
Fixpoint firstn (n:nat)(l:list A) {struct n} : list A :=
  match n with
    | 0 ⇒ nil
    | S n ⇒ match l with
                | nil ⇒ nil
                | a::l ⇒ a::(firstn n l)
             end
  end.

Fixpoint skipn (n:nat)(l:list A) { struct n } : list A :=
  match n with
    | 0 ⇒ l
    | S n ⇒ match l with
                | nil ⇒ nil
                | a::l ⇒ skipn n l
             end
```

end.

Lemma *firstn_skipn* : $\forall\ n\ l$, *firstn* $n\ l$ ++ *skipn* $n\ l = l$.

End *Cutting*.

145.5.3 Lists without redundancy

Section *ReDun*.

Variable A : Type.

Inductive *NoDup* : *list* $A \rightarrow$ Prop :=
| *NoDup_nil* : *NoDup nil*
| *NoDup_cons* : $\forall\ x\ l, \neg$ *In* $x\ l \rightarrow$ *NoDup* $l \rightarrow$ *NoDup* $(x::l)$.

Lemma *NoDup_remove_1* : $\forall\ l\ l'\ a$, *NoDup* $(l{+}{+}a::l') \rightarrow$ *NoDup* $(l{+}{+}l')$.

Lemma *NoDup_remove_2* : $\forall\ l\ l'\ a$, *NoDup* $(l{+}{+}a::l') \rightarrow \neg$*In* $a\ (l{+}{+}l')$.

Lemma *NoDup_Permutation* : $\forall\ l\ l'$,
NoDup $l \rightarrow$ *NoDup* $l' \rightarrow (\forall\ x$, *In* $x\ l \leftrightarrow$ *In* $x\ l') \rightarrow$ *Permutation* $l\ l'$.

End *ReDun*.

145.5.4 Sequence of natural numbers

Section *NatSeq*.

seq computes the sequence of *len* contiguous integers that starts at *start*. For instance, *seq* 2 3 is 2::3::4::*nil*.

Fixpoint *seq* $(start\ len{:}nat)\ \{struct\ len\}$: *list nat* :=
match *len* with
| $0 \Rightarrow$ *nil*
| $S\ len \Rightarrow$ *start* :: *seq* $(S\ start)\ len$
end.

Lemma *seq_length* : $\forall\ len\ start$, *length* $(seq\ start\ len) = len$.

Lemma *seq_nth* : $\forall\ len\ start\ n\ d$,
$n < len \rightarrow$ *nth* $n\ (seq\ start\ len)\ d = start{+}n$.

Lemma *seq_shift* : $\forall\ len\ start$,
map $S\ (seq\ start\ len) = seq\ (S\ start)\ len$.

End *NatSeq*.

145.6 Exporting hints and tactics

Hint *Rewrite*
rev_involutive rev_unit map_nth map_length seq_length app_length rev_length : *list*.

Hint *Rewrite* ←
  app_nil_end : *list*.

Ltac *simpl_list* := *autorewrite* with *list*.
Ltac *ssimpl_list* := *autorewrite* with *list* using *simpl*.

Chapter 146

Module Coq.Lists.MonoList

THIS IS A OLD CONTRIB. IT IS NO LONGER MAINTAINED

```
Require Import Le.

Parameter List_Dom : Set.
Definition A := List_Dom.

Inductive list : Set :=
  | nil : list
  | cons : A → list → list.

Fixpoint app (l m:list) {struct l} : list :=
  match l return list with
  | nil ⇒ m
  | cons a l1 ⇒ cons a (app l1 m)
  end.

Lemma app_nil_end : ∀ l:list, l = app l nil.
Hint Resolve app_nil_end: list v62.

Lemma app_ass : ∀ l m n:list, app (app l m) n = app l (app m n).
Hint Resolve app_ass: list v62.

Lemma ass_app : ∀ l m n:list, app l (app m n) = app (app l m) n.
Hint Resolve ass_app: list v62.

Definition tail (l:list) : list :=
  match l return list with
  | cons _ m ⇒ m
  | _ ⇒ nil
  end.

Lemma nil_cons : ∀ (a:A) (m:list), nil ≠ cons a m.

Fixpoint length (l:list) : nat :=
  match l return nat with
  | cons _ m ⇒ S (length m)
  | _ ⇒ 0
```

end.

Section *length_order*.
Definition *lel* (*l m*:*list*) := *length l* ≤ *length m*.

Hint *Unfold lel*: *list*.

Variables *a b* : *A*.
Variables *l m n* : *list*.

Lemma *lel_refl* : *lel l l*.

Lemma *lel_trans* : *lel l m* → *lel m n* → *lel l n*.

Lemma *lel_cons_cons* : *lel l m* → *lel* (*cons a l*) (*cons b m*).

Lemma *lel_cons* : *lel l m* → *lel l* (*cons b m*).

Lemma *lel_tail* : *lel* (*cons a l*) (*cons b m*) → *lel l m*.

Lemma *lel_nil* : ∀ *l'*:*list*, *lel l' nil* → *nil* = *l'*.
End *length_order*.

Hint *Resolve lel_refl lel_cons_cons lel_cons lel_nil lel_nil nil_cons*: *list v62*.

Fixpoint *In* (*a*:*A*) (*l*:*list*) {*struct l*} : Prop :=
  match *l* with
  | *nil* ⇒ *False*
  | *cons b m* ⇒ *b* = *a* ∨ *In a m*
  end.

Lemma *in_eq* : ∀ (*a*:*A*) (*l*:*list*), *In a* (*cons a l*).
Hint *Resolve in_eq*: *list v62*.

Lemma *in_cons* : ∀ (*a b*:*A*) (*l*:*list*), *In b l* → *In b* (*cons a l*).
Hint *Resolve in_cons*: *list v62*.

Lemma *in_app_or* : ∀ (*l m*:*list*) (*a*:*A*), *In a* (*app l m*) → *In a l* ∨ *In a m*.
Hint Immediate *in_app_or*: *list v62*.

Lemma *in_or_app* : ∀ (*l m*:*list*) (*a*:*A*), *In a l* ∨ *In a m* → *In a* (*app l m*).
Hint *Resolve in_or_app*: *list v62*.

Definition *incl* (*l m*:*list*) := ∀ *a*:*A*, *In a l* → *In a m*.

Hint *Unfold incl*: *list v62*.

Lemma *incl_refl* : ∀ *l*:*list*, *incl l l*.
Hint *Resolve incl_refl*: *list v62*.

Lemma *incl_tl* : ∀ (*a*:*A*) (*l m*:*list*), *incl l m* → *incl l* (*cons a m*).
Hint Immediate *incl_tl*: *list v62*.

Lemma *incl_tran* : ∀ *l m n*:*list*, *incl l m* → *incl m n* → *incl l n*.

Lemma *incl_appl* : ∀ *l m n*:*list*, *incl l n* → *incl l* (*app n m*).
Hint Immediate *incl_appl*: *list v62*.

Lemma *incl_appr* : $\forall\ l\ m\ n$:*list*, *incl* $l\ n \rightarrow$ *incl* l (*app* $m\ n$).
Hint Immediate *incl_appr*: *list v62*.

Lemma *incl_cons* :
$\forall$ (a:A) ($l\ m$:*list*), *In* $a\ m \rightarrow$ *incl* $l\ m \rightarrow$ *incl* (*cons* $a\ l$) m.
Hint *Resolve incl_cons*: *list v62*.

Lemma *incl_app* : $\forall\ l\ m\ n$:*list*, *incl* $l\ n \rightarrow$ *incl* $m\ n \rightarrow$ *incl* (*app* $l\ m$) n.
Hint *Resolve incl_app*: *list v62*.

Chapter 147

Module Coq.Lists.SetoidList

Require Export *List*.
Require Export *Sorting*.
Require Export *Setoid*.

147.1 Logical relations over lists with respect to a setoid equality

or ordering.

This can be seen as a complement of predicate *lelistA* and *sort* found in *Sorting*.

Section *Type_with_equality*.
Variable A : Set.
Variable eqA : $A \to A \to$ Prop.

Being in a list modulo an equality relation over type A.

Inductive InA $(x : A)$: $list\ A \to$ Prop :=
  | InA_cons_hd : $\forall\ y\ l, eqA\ x\ y \to InA\ x\ (y :: l)$
  | InA_cons_tl : $\forall\ y\ l, InA\ x\ l \to InA\ x\ (y :: l)$.

Hint *Constructors InA*.

An alternative definition of InA.

Lemma InA_alt : $\forall\ x\ l, InA\ x\ l \leftrightarrow \exists\ y, eqA\ x\ y \wedge In\ y\ l$.

A list without redundancy modulo the equality over A.

Inductive $NoDupA$: $list\ A \to$ Prop :=
  | $NoDupA\_nil$: $NoDupA\ nil$
  | $NoDupA\_cons$: $\forall\ x\ l, \neg\ InA\ x\ l \to NoDupA\ l \to NoDupA\ (x::l)$.

Hint *Constructors NoDupA*.

lists with same elements modulo eqA

Definition $eqlistA\ l\ l' := \forall\ x, InA\ x\ l \leftrightarrow InA\ x\ l'$.

Results concerning lists modulo eqA

Hypothesis eqA_refl : $\forall\ x,\ eqA\ x\ x$.
Hypothesis eqA_sym : $\forall\ x\ y,\ eqA\ x\ y \rightarrow eqA\ y\ x$.
Hypothesis eqA_trans : $\forall\ x\ y\ z,\ eqA\ x\ y \rightarrow eqA\ y\ z \rightarrow eqA\ x\ z$.

Hint *Resolve* $eqA\_refl\ eqA\_trans$.
Hint Immediate eqA_sym.

Lemma InA_eqA : $\forall\ l\ x\ y,\ eqA\ x\ y \rightarrow InA\ x\ l \rightarrow InA\ y\ l$.
Hint Immediate InA_eqA.

Lemma In_InA : $\forall\ l\ x,\ In\ x\ l \rightarrow InA\ x\ l$.
Hint *Resolve* In_InA.

Lemma InA_split : $\forall\ l\ x,\ InA\ x\ l \rightarrow$
$\exists\ l1,\ \exists\ y,\ \exists\ l2,$
$eqA\ x\ y \wedge l = l1 +\!\!+ y::l2$.

Results concerning lists modulo eqA and ltA

Variable ltA : $A \rightarrow A \rightarrow$ Prop.

Hypothesis ltA_trans : $\forall\ x\ y\ z,\ ltA\ x\ y \rightarrow ltA\ y\ z \rightarrow ltA\ x\ z$.
Hypothesis ltA_not_eqA : $\forall\ x\ y,\ ltA\ x\ y \rightarrow \neg\ eqA\ x\ y$.
Hypothesis ltA_eqA : $\forall\ x\ y\ z,\ ltA\ x\ y \rightarrow eqA\ y\ z \rightarrow ltA\ x\ z$.
Hypothesis eqA_ltA : $\forall\ x\ y\ z,\ eqA\ x\ y \rightarrow ltA\ y\ z \rightarrow ltA\ x\ z$.

Hint *Resolve* ltA_trans.
Hint Immediate $ltA\_eqA\ eqA\_ltA$.

Notation $InfA$:=$(lelistA\ ltA)$.
Notation $SortA$:=$(sort\ ltA)$.

Lemma $InfA\_ltA$:
$\forall\ l\ x\ y,\ ltA\ x\ y \rightarrow InfA\ y\ l \rightarrow InfA\ x\ l$.

Lemma $InfA\_eqA$:
$\forall\ l\ x\ y,\ eqA\ x\ y \rightarrow InfA\ y\ l \rightarrow InfA\ x\ l$.
Hint Immediate $InfA\_ltA\ InfA\_eqA$.

Lemma $SortA\_InfA\_InA$:
$\forall\ l\ x\ a,\ SortA\ l \rightarrow InfA\ a\ l \rightarrow InA\ x\ l \rightarrow ltA\ a\ x$.

Lemma In_InfA :
$\forall\ l\ x,\ (\forall\ y,\ In\ y\ l \rightarrow ltA\ x\ y) \rightarrow InfA\ x\ l$.

Lemma InA_InfA :
$\forall\ l\ x,\ (\forall\ y,\ InA\ y\ l \rightarrow ltA\ x\ y) \rightarrow InfA\ x\ l$.

Lemma $InfA\_alt$:
$\forall\ l\ x,\ SortA\ l \rightarrow (InfA\ x\ l \leftrightarrow (\forall\ y,\ InA\ y\ l \rightarrow ltA\ x\ y))$.

Lemma $SortA\_NoDupA$: $\forall\ l,\ SortA\ l \rightarrow NoDupA\ l$.

Lemma $NoDupA\_app$: $\forall\ l\ l',\ NoDupA\ l \rightarrow NoDupA\ l' \rightarrow$
$(\forall\ x,\ InA\ x\ l \rightarrow InA\ x\ l' \rightarrow False) \rightarrow$

NoDupA (*l*++*l'*).

Lemma *NoDupA_rev* : ∀ *l*, *NoDupA l* → *NoDupA* (*rev l*).

Lemma *InA_app* : ∀ *l1 l2 x*,
InA x (*l1* ++ *l2*) → *InA x l1* ∨ *InA x l2*.

Hint *Constructors lelistA sort*.

Lemma *InfA_app* : ∀ *l1 l2 a*, *InfA a l1* → *InfA a l2* → *InfA a* (*l1*++*l2*).

Lemma *SortA_app* :
∀ *l1 l2*, *SortA l1* → *SortA l2* →
(∀ *x y*, *InA x l1* → *InA y l2* → *ltA x y*) →
SortA (*l1* ++ *l2*).

Section *Remove*.

Hypothesis *eqA_dec* : ∀ *x y* : *A*, {*eqA x y*}+{~(*eqA x y*)}.

Lemma *InA_dec* : ∀ *x l*, { *InA x l* } + { ¬ *InA x l* }.

Fixpoint *removeA* (*x* : *A*) (*l* : *list A*){*struct l*} : *list A* :=
match *l* with
| *nil* ⇒ *nil*
| *y*::*tl* ⇒ if (*eqA_dec x y*) then *removeA x tl* else *y*::(*removeA x tl*)
end.

Lemma *removeA_filter* : ∀ *x l*,
removeA x l = *filter* (fun *y* ⇒ if *eqA_dec x y* then *false* else *true*) *l*.

Lemma *removeA_InA* : ∀ *l x y*, *InA y* (*removeA x l*) ↔ *InA y l* ∧ ¬*eqA x y*.

Lemma *removeA_NoDupA* :
∀ *s x*, *NoDupA s* → *NoDupA* (*removeA x s*).

Lemma *removeA_eqlistA* : ∀ *l l' x*,
¬*InA x l* → *eqlistA* (*x* :: *l*) *l'* → *eqlistA l* (*removeA x l'*).

Let *addlistA x l l'* := ∀ *y*, *InA y l'* ↔ *eqA x y* ∨ *InA y l*.

Lemma *removeA_add* :
∀ *s s' x x'*, *NoDupA s* → *NoDupA* (*x'* :: *s'*) →
¬ *eqA x x'* → ¬ *InA x s* →
addlistA x s (*x'* :: *s'*) → *addlistA x* (*removeA x' s*) *s'*.

Section *Fold*.

Variable *B*:Set.
Variable *eqB*:*B*→*B*→Prop.

Two-argument functions that allow to reorder its arguments.
Definition *transpose* (*f* : *A* → *B* → *B*) :=
∀ (*x y* : *A*) (*z* : *B*), *eqB* (*f x* (*f y z*)) (*f y* (*f x z*)).

Compatibility of a two-argument function with respect to two equalities.

```
Definition compat_op (f : A → B → B) :=
  ∀ (x x' : A) (y y' : B), eqA x x' → eqB y y' → eqB (f x y) (f x' y').
```

Compatibility of a function upon natural numbers.

```
Definition compat_nat (f : A → nat) :=
  ∀ x x' : A, eqA x x' → f x = f x'.

Variable st:Setoid_Theory _ eqB.
Variable f:A→B→B.
Variable Comp:compat_op f.
Variable Ass:transpose f.
Variable i:B.

Lemma removeA_fold_right_0 :
  ∀ s x, ¬InA x s →
  eqB (fold_right f i s) (fold_right f i (removeA x s)).

Lemma removeA_fold_right :
  ∀ s x, NoDupA s → InA x s →
  eqB (fold_right f i s) (f x (fold_right f i (removeA x s))).

Lemma fold_right_equal :
  ∀ s s', NoDupA s → NoDupA s' →
  eqlistA s s' → eqB (fold_right f i s) (fold_right f i s').

Lemma fold_right_add :
  ∀ s' s x, NoDupA s → NoDupA s' → ¬ InA x s →
  addlistA x s s' → eqB (fold_right f i s') (f x (fold_right f i s)).

End Fold.

End Remove.

End Type_with_equality.

Hint Constructors InA.
Hint Constructors NoDupA.
Hint Constructors sort.
Hint Constructors lelistA.

Section Find.
Variable A B : Set.
Variable eqA : A → A → Prop.
Hypothesis eqA_sym : ∀ x y, eqA x y → eqA y x.
Hypothesis eqA_trans : ∀ x y z, eqA x y → eqA y z → eqA x z.
Hypothesis eqA_dec : ∀ x y : A, {eqA x y}+{~(eqA x y)}.

Fixpoint findA (f : A → bool) (l:list (A×B)) : option B :=
 match l with
  | nil ⇒ None
  | (a,b)::l ⇒ if f a then Some b else findA f l
 end.
```

Lemma *findA_NoDupA* :
 ∀ *l a b*,
 NoDupA (fun *p p'* ⇒ *eqA* (*fst p*) (*fst p'*)) *l* →
 (*InA* (fun *p p'* ⇒ *eqA* (*fst p*) (*fst p'*) ∧ *snd p* = *snd p'*) (*a*,*b*) *l* ↔
  findA (fun *a'* ⇒ if *eqA_dec a a'* then *true* else *false*) *l* = *Some b*).

End *Find*.

Chapter 148

Module Coq.Lists.Streams

Streams

Section *Streams*.

Variable A : Set.

CoInductive *Stream* : Set :=
  $Cons : A \to Stream \to Stream$.

Definition *hd* (x:*Stream*) := match x with
  | $Cons\ a\ \_ \Rightarrow a$
  end.

Definition *tl* (x:*Stream*) := match x with
  | $Cons\ \_\ s \Rightarrow s$
  end.

Fixpoint *Str_nth_tl* (n:*nat*) (s:*Stream*) {*struct* n} : *Stream* :=
  match n with
  | $O \Rightarrow s$
  | $S\ m \Rightarrow Str\_nth\_tl\ m\ (tl\ s)$
  end.

Definition *Str_nth* (n:*nat*) (s:*Stream*) : $A := hd\ (Str\_nth\_tl\ n\ s)$.

Lemma *unfold_Stream* :
 $\forall\ x$:*Stream*, $x =$ match x with
  | $Cons\ a\ s \Rightarrow Cons\ a\ s$
  end.

Lemma *tl_nth_tl* :
 $\forall\ (n$:*nat*$)\ (s$:*Stream*$)$, $tl\ (Str\_nth\_tl\ n\ s) = Str\_nth\_tl\ n\ (tl\ s)$.
Hint *Resolve tl_nth_tl*: *datatypes v62*.

Lemma *Str_nth_tl_plus* :
 $\forall\ (n\ m$:*nat*$)\ (s$:*Stream*$)$,
   $Str\_nth\_tl\ n\ (Str\_nth\_tl\ m\ s) = Str\_nth\_tl\ (n + m)\ s$.

Lemma *Str_nth_plus* :

$\forall\ (n\ m{:}nat)\ (s{:}Stream),\ Str\_nth\ n\ (Str\_nth\_tl\ m\ s) = Str\_nth\ (n + m)\ s.$

Extensional Equality between two streams

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
    eqst :
        hd s1 = hd s2 → EqSt (tl s1) (tl s2) → EqSt s1 s2.
```

A coinduction principle

```
Ltac coinduction proof :=
  cofix proof; intros; constructor;
   [ clear proof | try (apply proof; clear proof) ].
```

Extensional equality is an equivalence relation

```
Theorem EqSt_reflex : ∀ s:Stream, EqSt s s.
```

```
Theorem sym_EqSt : ∀ s1 s2:Stream, EqSt s1 s2 → EqSt s2 s1.
```

```
Theorem trans_EqSt :
 ∀ s1 s2 s3:Stream, EqSt s1 s2 → EqSt s2 s3 → EqSt s1 s3.
```

The definition given is equivalent to require the elements at each position to be equal

```
Theorem eqst_ntheq :
 ∀ (n:nat) (s1 s2:Stream), EqSt s1 s2 → Str_nth n s1 = Str_nth n s2.
```

```
Theorem ntheq_eqst :
 ∀ s1 s2:Stream,
    (∀ n:nat, Str_nth n s1 = Str_nth n s2) → EqSt s1 s2.
```

```
Section Stream_Properties.
```

```
Variable P : Stream → Prop.
```

```
Inductive Exists ( x: Stream ) : Prop :=
  | Here : P x → Exists x
  | Further : Exists (tl x) → Exists x.
```

```
CoInductive ForAll (x: Stream) : Prop :=
     HereAndFurther : P x → ForAll (tl x) → ForAll x.
```

```
Section Co_Induction_ForAll.
Variable Inv : Stream → Prop.
Hypothesis InvThenP : ∀ x:Stream, Inv x → P x.
Hypothesis InvIsStable : ∀ x:Stream, Inv x → Inv (tl x).
```

```
Theorem ForAll_coind : ∀ x:Stream, Inv x → ForAll x.
End Co_Induction_ForAll.
```

```
End Stream_Properties.
```

```
End Streams.
```

```
Section Map.
Variables A B : Set.
```

Variable f : $A \rightarrow B$.
CoFixpoint map (s:$Stream$ A) : $Stream$ B := $Cons$ (f (hd s)) (map (tl s)).
End Map.

Section $Constant\_Stream$.
Variable A : Set.
Variable a : A.
CoFixpoint $const$: $Stream$ A := $Cons$ a $const$.
End $Constant\_Stream$.

Chapter 149

Module Coq.Lists.TheoryList

Some programs and results about lists following CAML Manual

```
Require Export List.
Section Lists.

Variable A : Type.

The null function
Definition Isnil (l:list A) : Prop := nil = l.

Lemma Isnil_nil : Isnil nil.
Hint Resolve Isnil_nil.

Lemma not_Isnil_cons : ∀ (a:A) (l:list A), ¬ Isnil (a :: l).

Hint Resolve Isnil_nil not_Isnil_cons.

Lemma Isnil_dec : ∀ l:list A, {Isnil l} + {~ Isnil l}.

The Uncons function
Lemma Uncons :
 ∀ l:list A, {a : A & {m : list A | a :: m = l}} + {Isnil l}.

The head function
Lemma Hd :
 ∀ l:list A, {a : A | ∃ m : list A, a :: m = l} + {Isnil l}.

Lemma Tl :
 ∀ l:list A,
    {m : list A | (∃ a : A, a :: m = l) ∨ Isnil l ∧ Isnil m}.

Length of lists
Fixpoint Length_l (l:list A) (n:nat) {struct l} : nat :=
  match l with
  | nil ⇒ n
  | _ :: m ⇒ Length_l m (S n)
  end.

Lemma Length_l_pf : ∀ (l:list A) (n:nat), {m : nat | n + length l = m}.
```

```
Lemma Length : ∀ l:list A, {m : nat | length l = m}.
```

Members of lists

```
Inductive In_spec (a:A) : list A → Prop :=
  | in_hd : ∀ l:list A, In_spec a (a :: l)
  | in_tl : ∀ (l:list A) (b:A), In a l → In_spec a (b :: l).
Hint Resolve in_hd in_tl.
Hint Unfold In.
Hint Resolve in_cons.

Theorem In_In_spec : ∀ (a:A) (l:list A), In a l ↔ In_spec a l.

Inductive AllS (P:A → Prop) : list A → Prop :=
  | allS_nil : AllS P nil
  | allS_cons : ∀ (a:A) (l:list A), P a → AllS P l → AllS P (a :: l).
Hint Resolve allS_nil allS_cons.

Hypothesis eqA_dec : ∀ a b:A, {a = b} + {a ≠ b}.

Fixpoint mem (a:A) (l:list A) {struct l} : bool :=
  match l with
  | nil ⇒ false
  | b :: m ⇒ if eqA_dec a b then true else mem a m
  end.

Hint Unfold In.
Lemma Mem : ∀ (a:A) (l:list A), {In a l} + {AllS (fun b:A ⇒ b ≠ a) l}.
```

Index of elements

```
Require Import Le.
Require Import Lt.

Inductive nth_spec : list A → nat → A → Prop :=
  | nth_spec_O : ∀ (a:A) (l:list A), nth_spec (a :: l) 1 a
  | nth_spec_S :
      ∀ (n:nat) (a b:A) (l:list A),
        nth_spec l n a → nth_spec (b :: l) (S n) a.
Hint Resolve nth_spec_O nth_spec_S.

Inductive fst_nth_spec : list A → nat → A → Prop :=
  | fst_nth_O : ∀ (a:A) (l:list A), fst_nth_spec (a :: l) 1 a
  | fst_nth_S :
      ∀ (n:nat) (a b:A) (l:list A),
        a ≠ b → fst_nth_spec l n a → fst_nth_spec (b :: l) (S n) a.
Hint Resolve fst_nth_O fst_nth_S.

Lemma fst_nth_nth :
 ∀ (l:list A) (n:nat) (a:A), fst_nth_spec l n a → nth_spec l n a.
Hint Immediate fst_nth_nth.

Lemma nth_lt_O : ∀ (l:list A) (n:nat) (a:A), nth_spec l n a → 0 < n.

Lemma nth_le_length :
```

$\forall$ (l:*list* A) (n:*nat*) (a:A), *nth_spec* l n a $\rightarrow$ $n \leq$ *length* l.

Fixpoint *Nth_func* (l:*list* A) (n:*nat*) {*struct* l} : *Exc* A :=
  match l, n with
  | a :: _, S O $\Rightarrow$ *value* a
  | _ :: l', S (S p) $\Rightarrow$ *Nth_func* l' (S p)
  | _, _ $\Rightarrow$ *error*
  end.

Lemma *Nth* :
 $\forall$ (l:*list* A) (n:*nat*),
   {a : A | *nth_spec* l n a} + {$n = 0 \vee$ *length* $l < n$}.

Lemma *Item* :
 $\forall$ (l:*list* A) (n:*nat*), {a : A | *nth_spec* l (S n) a} + {*length* $l \leq n$}.

Require Import *Minus*.
Require Import *DecBool*.

Fixpoint *index_p* (a:A) (l:*list* A) {*struct* l} : *nat* $\rightarrow$ *Exc nat* :=
  match l with
  | *nil* $\Rightarrow$ fun p $\Rightarrow$ *error*
  | b :: m $\Rightarrow$ fun p $\Rightarrow$ *ifdec* (*eqA_dec* a b) (*value* p) (*index_p* a m (S p))
  end.

Lemma *Index_p* :
 $\forall$ (a:A) (l:*list* A) (p:*nat*),
   {n : *nat* | *fst_nth_spec* l (S n - p) a} + {*AllS* (fun b:A $\Rightarrow$ $a \neq b$) l}.

Lemma *Index* :
 $\forall$ (a:A) (l:*list* A),
   {n : *nat* | *fst_nth_spec* l n a} + {*AllS* (fun b:A $\Rightarrow$ $a \neq b$) l}.

Section *Find_sec*.
Variables R P : $A \rightarrow$ Prop.

Inductive *InR* : *list* $A \rightarrow$ Prop :=
  | *inR_hd* : $\forall$ (a:A) (l:*list* A), R $a \rightarrow$ *InR* (a :: l)
  | *inR_tl* : $\forall$ (a:A) (l:*list* A), *InR* $l \rightarrow$ *InR* (a :: l).
Hint *Resolve inR_hd inR_tl*.

Definition *InR_inv* (l:*list* A) :=
  match l with
  | *nil* $\Rightarrow$ *False*
  | b :: m $\Rightarrow$ R $b \vee$ *InR* m
  end.

Lemma *InR_INV* : $\forall$ l:*list* A, *InR* $l \rightarrow$ *InR_inv* l.

Lemma *InR_cons_inv* : $\forall$ (a:A) (l:*list* A), *InR* (a :: l) $\rightarrow$ R $a \vee$ *InR* l.

Lemma *InR_or_app* : $\forall$ l m:*list* A, *InR* $l \vee$ *InR* $m \rightarrow$ *InR* (l ++ m).

```
Lemma InR_app_or : ∀ l m:list A, InR (l ++ m) → InR l ∨ InR m.

Hypothesis RS_dec : ∀ a:A, {R a} + {P a}.

Fixpoint find (l:list A) : Exc A :=
  match l with
  | nil ⇒ error
  | a :: m ⇒ ifdec (RS_dec a) (value a) (find m)
  end.

Lemma Find : ∀ l:list A, {a : A | In a l & R a} + {AllS P l}.

Variable B : Type.
Variable T : A → B → Prop.

Variable TS_dec : ∀ a:A, {c : B | T a c} + {P a}.

Fixpoint try_find (l:list A) : Exc B :=
  match l with
  | nil ⇒ error
  | a :: l1 ⇒
      match TS_dec a with
      | inleft (exist c _) ⇒ value c
      | inright _ ⇒ try_find l1
      end
  end.

Lemma Try_find :
 ∀ l:list A, {c : B | exists2 a : A, In a l & T a c} + {AllS P l}.

End Find_sec.

Section Assoc_sec.

Variable B : Type.
Fixpoint assoc (a:A) (l:list (A × B)) {struct l} :
 Exc B :=
  match l with
  | nil ⇒ error
  | (a', b) :: m ⇒ ifdec (eqA_dec a a') (value b) (assoc a m)
  end.

Inductive AllS_assoc (P:A → Prop) : list (A × B) → Prop :=
  | allS_assoc_nil : AllS_assoc P nil
  | allS_assoc_cons :
      ∀ (a:A) (b:B) (l:list (A × B)),
        P a → AllS_assoc P l → AllS_assoc P ((a, b) :: l).

Hint Resolve allS_assoc_nil allS_assoc_cons.

Lemma Assoc :
 ∀ (a:A) (l:list (A × B)), B + {AllS_assoc (fun a':A ⇒ a ≠ a') l}.

End Assoc_sec.
```

End *Lists*.

Hint *Resolve Isnil_nil not_Isnil_cons in_hd in_tl in_cons allS_nil allS_cons*: *datatypes*.
Hint Immediate *fst_nth_nth*: *datatypes*.

Chapter 150

Module Coq.Sets.Classical_sets

Require Export *Ensembles*.
Require Export *Constructive_sets*.
Require Export *Classical_Type*.

Section *Ensembles_classical*.
Variable U : Type.

Lemma *not_included_empty_Inhabited* :
$\forall\ A$:*Ensemble* $U, \neg$ *Included* $U\ A$ (*Empty_set* $U) \rightarrow$ *Inhabited* $U\ A$.

Lemma *not_empty_Inhabited* :
$\forall\ A$:*Ensemble* $U, A \neq$ *Empty_set* $U \rightarrow$ *Inhabited* $U\ A$.

Lemma *Inhabited_Setminus* :
$\forall\ X\ Y$:*Ensemble* U,
Included $U\ X\ Y \rightarrow \neg$ *Included* $U\ Y\ X \rightarrow$ *Inhabited* U (*Setminus* $U\ Y\ X$).

Lemma *Strict_super_set_contains_new_element* :
$\forall\ X\ Y$:*Ensemble* U,
Included $U\ X\ Y \rightarrow X \neq Y \rightarrow$ *Inhabited* U (*Setminus* $U\ Y\ X$).

Lemma *Subtract_intro* :
$\forall$ (A:*Ensemble* U) ($x\ y$:U), *In* $U\ A\ y \rightarrow x \neq y \rightarrow$ *In* U (*Subtract* $U\ A\ x$) y.
Hint *Resolve Subtract_intro* : *sets*.

Lemma *Subtract_inv* :
$\forall$ (A:*Ensemble* U) ($x\ y$:U), *In* U (*Subtract* $U\ A\ x$) $y \rightarrow$ *In* $U\ A\ y \wedge x \neq y$.

Lemma *Included_Strict_Included* :
$\forall\ X\ Y$:*Ensemble* U, *Included* $U\ X\ Y \rightarrow$ *Strict_Included* $U\ X\ Y \vee X = Y$.

Lemma *Strict_Included_inv* :
$\forall\ X\ Y$:*Ensemble* U,
Strict_Included $U\ X\ Y \rightarrow$ *Included* $U\ X\ Y \wedge$ *Inhabited* U (*Setminus* $U\ Y\ X$).

Lemma *not_SIncl_empty* :
$\forall\ X$:*Ensemble* $U, \neg$ *Strict_Included* $U\ X$ (*Empty_set* U).

Lemma *Complement_Complement* :

∀ *A*:*Ensemble U*, *Complement U* (*Complement U A*) = *A*.

End *Ensembles_classical*.

Hint *Resolve Strict_super_set_contains_new_element Subtract_intro not_SIncl_empty*: *sets v62*.

Chapter 151

Module Coq.Sets.Constructive_sets

Require Export *Ensembles*.

Section *Ensembles_facts*.
Variable U : Type.

Lemma *Extension* : $\forall\ B\ C$:*Ensemble* U, $B = C \rightarrow$ *Same_set* $U\ B\ C$.

Lemma *Noone_in_empty* : $\forall\ x$:U, $\neg$ *In* U (*Empty_set* U) x.

Lemma *Included_Empty* : $\forall\ A$:*Ensemble* U, *Included* U (*Empty_set* U) A.

Lemma *Add_intro1* :
$\forall\ (A$:*Ensemble* $U)\ (x\ y$:$U)$, *In* $U\ A\ y \rightarrow$ *In* U (*Add* $U\ A\ x$) y.

Lemma *Add_intro2* : $\forall\ (A$:*Ensemble* $U)\ (x$:$U)$, *In* U (*Add* $U\ A\ x$) x.

Lemma *Inhabited_add* : $\forall\ (A$:*Ensemble* $U)\ (x$:$U)$, *Inhabited* U (*Add* $U\ A\ x$).

Lemma *Inhabited_not_empty* :
$\forall\ X$:*Ensemble* U, *Inhabited* $U\ X \rightarrow X \neq$ *Empty_set* U.

Lemma *Add_not_Empty* : $\forall\ (A$:*Ensemble* $U)\ (x$:$U)$, *Add* $U\ A\ x \neq$ *Empty_set* U.

Lemma *not_Empty_Add* : $\forall\ (A$:*Ensemble* $U)\ (x$:$U)$, *Empty_set* $U \neq$ *Add* $U\ A\ x$.

Lemma *Singleton_inv* : $\forall\ x\ y$:U, *In* U (*Singleton* $U\ x$) $y \rightarrow x = y$.

Lemma *Singleton_intro* : $\forall\ x\ y$:U, $x = y \rightarrow$ *In* U (*Singleton* $U\ x$) y.

Lemma *Union_inv* :
$\forall\ (B\ C$:*Ensemble* $U)\ (x$:$U)$, *In* U (*Union* $U\ B\ C$) $x \rightarrow$ *In* $U\ B\ x \vee$ *In* $U\ C\ x$.

Lemma *Add_inv* :
$\forall\ (A$:*Ensemble* $U)\ (x\ y$:$U)$, *In* U (*Add* $U\ A\ x$) $y \rightarrow$ *In* $U\ A\ y \vee x = y$.

Lemma *Intersection_inv* :
$\forall\ (B\ C$:*Ensemble* $U)\ (x$:$U)$,
In U (*Intersection* $U\ B\ C$) $x \rightarrow$ *In* $U\ B\ x \wedge$ *In* $U\ C\ x$.

Lemma *Couple_inv* : $\forall\ x\ y\ z$:U, *In* U (*Couple* $U\ x\ y$) $z \rightarrow z = x \vee z = y$.

Lemma *Setminus_intro* :

$\forall\ (A\ B{:}Ensemble\ U)\ (x{:}U),$
$In\ U\ A\ x \rightarrow \neg\ In\ U\ B\ x \rightarrow In\ U\ (Setminus\ U\ A\ B)\ x.$

Lemma *Strict_Included_intro* :
$\forall\ X\ Y{:}Ensemble\ U, Included\ U\ X\ Y \wedge X \neq Y \rightarrow Strict\_Included\ U\ X\ Y.$

Lemma *Strict_Included_strict* : $\forall\ X{:}Ensemble\ U, \neg\ Strict\_Included\ U\ X\ X.$

End *Ensembles_facts.*

Hint *Resolve Singleton_inv Singleton_intro Add_intro1 Add_intro2 Intersection_inv Couple_inv Setminus_intro Strict_Included_intro Strict_Included_strict Noone_in_empty Inhabited_not_empty Add_not_Empty not_Empty_Add Inhabited_add Included_Empty*: *sets v62.*

Chapter 152

Module Coq.Sets.Cpo

```
Require Export Ensembles.
Require Export Relations_1.
Require Export Partial_Order.

Section Bounds.
  Variable U : Type.
  Variable D : PO U.

  Let C := Carrier_of U D.

  Let R := Rel_of U D.

  Inductive Upper_Bound (B:Ensemble U) (x:U) : Prop :=
    Upper_Bound_definition :
    In U C x → (∀ y:U, In U B y → R y x) → Upper_Bound B x.

  Inductive Lower_Bound (B:Ensemble U) (x:U) : Prop :=
    Lower_Bound_definition :
    In U C x → (∀ y:U, In U B y → R x y) → Lower_Bound B x.

  Inductive Lub (B:Ensemble U) (x:U) : Prop :=
    Lub_definition :
    Upper_Bound B x → (∀ y:U, Upper_Bound B y → R x y) → Lub B x.

  Inductive Glb (B:Ensemble U) (x:U) : Prop :=
    Glb_definition :
    Lower_Bound B x → (∀ y:U, Lower_Bound B y → R y x) → Glb B x.

  Inductive Bottom (bot:U) : Prop :=
    Bottom_definition :
    In U C bot → (∀ y:U, In U C y → R bot y) → Bottom bot.

  Inductive Totally_ordered (B:Ensemble U) : Prop :=
    Totally_ordered_definition :
    (Included U B C →
      ∀ x y:U, Included U (Couple U x y) B → R x y ∨ R y x) →
    Totally_ordered B.

  Definition Compatible : Relation U :=
```

```
    fun x y:U ⇒
      In U C x →
      In U C y → ∃ z : _, In U C z ∧ Upper_Bound (Couple U x y) z.

  Inductive Directed (X:Ensemble U) : Prop :=
    Definition_of_Directed :
    Included U X C →
    Inhabited U X →
    (∀ x1 x2:U,
      Included U (Couple U x1 x2) X →
      ∃ x3 : _, In U X x3 ∧ Upper_Bound (Couple U x1 x2) x3) →
    Directed X.

  Inductive Complete : Prop :=
    Definition_of_Complete :
    (∃ bot : _, Bottom bot) →
    (∀ X:Ensemble U, Directed X → ∃ bsup : _, Lub X bsup) →
    Complete.

  Inductive Conditionally_complete : Prop :=
    Definition_of_Conditionally_complete :
    (∀ X:Ensemble U,
      Included U X C →
      (∃ maj : _, Upper_Bound X maj) →
      ∃ bsup : _, Lub X bsup) → Conditionally_complete.
End Bounds.

Hint Resolve Totally_ordered_definition Upper_Bound_definition
  Lower_Bound_definition Lub_definition Glb_definition Bottom_definition
  Definition_of_Complete Definition_of_Complete
  Definition_of_Conditionally_complete.

Section Specific_orders.
  Variable U : Type.

  Record Cpo : Type := Definition_of_cpo
    {PO_of_cpo : PO U; Cpo_cond : Complete U PO_of_cpo}.

  Record Chain : Type := Definition_of_chain
    {PO_of_chain : PO U;
    Chain_cond : Totally_ordered U PO_of_chain (Carrier_of U PO_of_chain)}.

End Specific_orders.
```

Chapter 153

Module Coq.Sets.Ensembles

Section *Ensembles*.
Variable U : Type.

Definition *Ensemble* := $U \rightarrow$ Prop.

Definition *In* (*A*:*Ensemble*) (*x*:*U*) : Prop := *A x*.

Definition *Included* (*B C*:*Ensemble*) : Prop := $\forall$ *x*:*U*, *In B x* $\rightarrow$ *In C x*.

Inductive *Empty_set* : *Ensemble* :=.

Inductive *Full_set* : *Ensemble* :=
Full_intro : $\forall$ *x*:*U*, *In Full_set x*.

NB: The following definition builds-in equality of elements in U as Leibniz equality.

This may have to be changed if we replace U by a Setoid on U with its own equality *eqs*, with *In_singleton*: (*y*: *U*)(*eqs x y*) $\rightarrow$ (*In* (*Singleton x*) *y*).

Inductive *Singleton* (*x*:*U*) : *Ensemble* :=
In_singleton : *In* (*Singleton x*) *x*.

Inductive *Union* (*B C*:*Ensemble*) : *Ensemble* :=
| *Union_introl* : $\forall$ *x*:*U*, *In B x* $\rightarrow$ *In* (*Union B C*) *x*
| *Union_intror* : $\forall$ *x*:*U*, *In C x* $\rightarrow$ *In* (*Union B C*) *x*.

Definition *Add* (*B*:*Ensemble*) (*x*:*U*) : *Ensemble* := *Union B* (*Singleton x*).

Inductive *Intersection* (*B C*:*Ensemble*) : *Ensemble* :=
Intersection_intro :
$\forall$ *x*:*U*, *In B x* $\rightarrow$ *In C x* $\rightarrow$ *In* (*Intersection B C*) *x*.

Inductive *Couple* (*x y*:*U*) : *Ensemble* :=
| *Couple_l* : *In* (*Couple x y*) *x*
| *Couple_r* : *In* (*Couple x y*) *y*.

Inductive *Triple* (*x y z*:*U*) : *Ensemble* :=
| *Triple_l* : *In* (*Triple x y z*) *x*
| *Triple_m* : *In* (*Triple x y z*) *y*
| *Triple_r* : *In* (*Triple x y z*) *z*.

Definition *Complement* (*A*:*Ensemble*) : *Ensemble* := fun *x*:*U* ⇒ ¬ *In A x*.

Definition *Setminus* (*B C*:*Ensemble*) : *Ensemble* :=
fun *x*:*U* ⇒ *In B x* ∧ ¬ *In C x*.

Definition *Subtract* (*B*:*Ensemble*) (*x*:*U*) : *Ensemble* := *Setminus B* (*Singleton x*).

Inductive *Disjoint* (*B C*:*Ensemble*) : Prop :=
Disjoint_intro : (∀ *x*:*U*, ¬ *In* (*Intersection B C*) *x*) → *Disjoint B C*.

Inductive *Inhabited* (*B*:*Ensemble*) : Prop :=
Inhabited_intro : ∀ *x*:*U*, *In B x* → *Inhabited B*.

Definition *Strict_Included* (*B C*:*Ensemble*) : Prop := *Included B C* ∧ *B* ≠ *C*.

Definition *Same_set* (*B C*:*Ensemble*) : Prop := *Included B C* ∧ *Included C B*.

Extensionality Axiom

Axiom *Extensionality_Ensembles* : ∀ *A B*:*Ensemble*, *Same_set A B* → *A* = *B*.

End *Ensembles*.

Hint *Unfold In Included Same_set Strict_Included Add Setminus Subtract*: *sets v62*.

Hint *Resolve Union_introl Union_intror Intersection_intro In_singleton Couple_l Couple_r Triple_l Triple_m Triple_r Disjoint_intro Extensionality_Ensembles*: *sets v62*.

Chapter 154

Module Coq.Sets.Finite_sets_facts

Require Export *Finite_sets*.
Require Export *Constructive_sets*.
Require Export *Classical_Type*.
Require Export *Classical_sets*.
Require Export *Powerset*.
Require Export *Powerset_facts*.
Require Export *Powerset_Classical_facts*.
Require Export *Gt*.
Require Export *Lt*.

Section *Finite_sets_facts*.

Variable U : Type.

Lemma *finite_cardinal* :
$\forall\ X$:*Ensemble* U, *Finite* $U\ X \to \exists\ n$: *nat*, *cardinal* $U\ X\ n$.

Lemma *cardinal_finite* :
$\forall\ (X$:*Ensemble* $U)\ (n$:*nat*), *cardinal* $U\ X\ n \to$ *Finite* $U\ X$.

Theorem *Add_preserves_Finite* :
$\forall\ (X$:*Ensemble* $U)\ (x$:$U)$, *Finite* $U\ X \to$ *Finite* U (*Add* $U\ X\ x$).

Theorem *Singleton_is_finite* : $\forall\ x$:U, *Finite* U (*Singleton* $U\ x$).

Theorem *Union_preserves_Finite* :
$\forall\ X\ Y$:*Ensemble* U, *Finite* $U\ X \to$ *Finite* $U\ Y \to$ *Finite* U (*Union* $U\ X\ Y$).

Lemma *Finite_downward_closed* :
$\forall\ A$:*Ensemble* U,
Finite $U\ A \to \forall\ X$:*Ensemble* U, *Included* $U\ X\ A \to$ *Finite* $U\ X$.

Lemma *Intersection_preserves_finite* :
$\forall\ A$:*Ensemble* U,
Finite $U\ A \to \forall\ X$:*Ensemble* U, *Finite* U (*Intersection* $U\ X\ A$).

Lemma *cardinalO_empty* :
$\forall\ X$:*Ensemble* U, *cardinal* $U\ X\ 0 \to X =$ *Empty_set* U.

Lemma *inh_card_gt_O* :

$\forall\ X$:*Ensemble U*, *Inhabited U X* $\rightarrow \forall\ n$:*nat*, *cardinal U X n* $\rightarrow n > 0$.

Lemma *card_soustr_1* :
$\forall\ (X$:*Ensemble U*$)\ (n$:*nat*$)$,
cardinal U X n $\rightarrow$
$\forall\ x$:*U*, *In U X x* $\rightarrow$ *cardinal U* (*Subtract U X x*) (*pred n*).

Lemma *cardinal_is_functional* :
$\forall\ (X$:*Ensemble U*$)\ (c1$:*nat*$)$,
cardinal U X c1 $\rightarrow$
$\forall\ (Y$:*Ensemble U*$)\ (c2$:*nat*$)$, *cardinal U Y c2* $\rightarrow X = Y \rightarrow c1 = c2$.

Lemma *cardinal_Empty* : $\forall\ m$:*nat*, *cardinal U* (*Empty_set U*) $m \rightarrow 0 = m$.

Lemma *cardinal_unicity* :
$\forall\ (X$:*Ensemble U*$)\ (n$:*nat*$)$,
cardinal U X n $\rightarrow \forall\ m$:*nat*, *cardinal U X m* $\rightarrow n = m$.

Lemma *card_Add_gen* :
$\forall\ (A$:*Ensemble U*$)\ (x$:*U*$)\ (n\ n'$:*nat*$)$,
cardinal U A n $\rightarrow$ *cardinal U* (*Add U A x*) $n' \rightarrow n' \leq S\ n$.

Lemma *incl_st_card_lt* :
$\forall\ (X$:*Ensemble U*$)\ (c1$:*nat*$)$,
cardinal U X c1 $\rightarrow$
$\forall\ (Y$:*Ensemble U*$)\ (c2$:*nat*$)$,
cardinal U Y c2 $\rightarrow$ *Strict_Included U X Y* $\rightarrow c2 > c1$.

Lemma *incl_card_le* :
$\forall\ (X\ Y$:*Ensemble U*$)\ (n\ m$:*nat*$)$,
cardinal U X n $\rightarrow$ *cardinal U Y m* $\rightarrow$ *Included U X Y* $\rightarrow n \leq m$.

Lemma *G_aux* :
$\forall\ P$:*Ensemble U* $\rightarrow$ Prop,
$(\forall\ X$:*Ensemble U*,
Finite U X $\rightarrow$
$(\forall\ Y$:*Ensemble U*, *Strict_Included U Y X* $\rightarrow$ *P Y*$) \rightarrow$ *P X*$) \rightarrow$
P (*Empty_set U*).

Lemma *Generalized_induction_on_finite_sets* :
$\forall\ P$:*Ensemble U* $\rightarrow$ Prop,
$(\forall\ X$:*Ensemble U*,
Finite U X $\rightarrow$
$(\forall\ Y$:*Ensemble U*, *Strict_Included U Y X* $\rightarrow$ *P Y*$) \rightarrow$ *P X*$) \rightarrow$
$\forall\ X$:*Ensemble U*, *Finite U X* $\rightarrow$ *P X*.

End *Finite_sets_facts*.

Chapter 155

Module Coq.Sets.Finite_sets

```
Require Import Ensembles.

Section Ensembles_finis.
  Variable U : Type.

  Inductive Finite : Ensemble U → Prop :=
    | Empty_is_finite : Finite (Empty_set U)
    | Union_is_finite :
      ∀ A:Ensemble U,
        Finite A → ∀ x:U, ¬ In U A x → Finite (Add U A x).

  Inductive cardinal : Ensemble U → nat → Prop :=
    | card_empty : cardinal (Empty_set U) 0
    | card_add :
      ∀ (A:Ensemble U) (n:nat),
        cardinal A n → ∀ x:U, ¬ In U A x → cardinal (Add U A x) (S n).

End Ensembles_finis.

Hint Resolve Empty_is_finite Union_is_finite: sets v62.
Hint Resolve card_empty card_add: sets v62.

Require Import Constructive_sets.

Section Ensembles_finis_facts.
  Variable U : Type.

  Lemma cardinal_invert :
    ∀ (X:Ensemble U) (p:nat),
      cardinal U X p →
      match p with
        | O ⇒ X = Empty_set U
        | S n ⇒
          ∃ A : _,
            (∃ x : _, X = Add U A x ∧ ¬ In U A x ∧ cardinal U A n)
      end.

  Lemma cardinal_elim :
```

```
∀ (X:Ensemble U) (p:nat),
  cardinal U X p →
  match p with
    | O ⇒ X = Empty_set U
    | S n ⇒ Inhabited U X
  end.

End Ensembles_finis_facts.
```

Chapter 156

Module Coq.Sets.Image

Require Export *Finite_sets*.
Require Export *Constructive_sets*.
Require Export *Classical_Type*.
Require Export *Classical_sets*.
Require Export *Powerset*.
Require Export *Powerset_facts*.
Require Export *Powerset_Classical_facts*.
Require Export *Gt*.
Require Export *Lt*.
Require Export *Le*.
Require Export *Finite_sets_facts*.

Section *Image*.

Variables U V : Type.

Inductive $Im\ (X{:}Ensemble\ U)\ (f{:}U \to V) : Ensemble\ V :=$
$Im\_intro : \forall\ x{:}U, In\ \_\ X\ x \to \forall\ y{:}V, y = f\ x \to In\ \_\ (Im\ X\ f)\ y.$

Lemma *Im_def* :
$\forall\ (X{:}Ensemble\ U)\ (f{:}U \to V)\ (x{:}U), In\ \_\ X\ x \to In\ \_\ (Im\ X\ f)\ (f\ x).$

Lemma *Im_add* :
$\forall\ (X{:}Ensemble\ U)\ (x{:}U)\ (f{:}U \to V),$
$Im\ (Add\ \_\ X\ x)\ f = Add\ \_\ (Im\ X\ f)\ (f\ x).$

Lemma *image_empty* : $\forall\ f{:}U \to V, Im\ (Empty\_set\ U)\ f = Empty\_set\ V.$

Lemma *finite_image* :
$\forall\ (X{:}Ensemble\ U)\ (f{:}U \to V), Finite\ \_\ X \to Finite\ \_\ (Im\ X\ f).$

Lemma *Im_inv* :
$\forall\ (X{:}Ensemble\ U)\ (f{:}U \to V)\ (y{:}V),$
$In\ \_\ (Im\ X\ f)\ y \to \exists\ x : U, In\ \_\ X\ x \wedge f\ x = y.$

Definition $injective\ (f{:}U \to V) := \forall\ x\ y{:}U, f\ x = f\ y \to x = y.$

Lemma *not_injective_elim* :
$\forall\ f{:}U \to V,$

$\neg$ *injective* $f \rightarrow \exists\ x : \_, (\exists\ y : \_, f\ x = f\ y \wedge x \neq y)$.

Lemma *cardinal_Im_intro* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat)$,
cardinal _ $A\ n \rightarrow \exists\ p : nat$, *cardinal* _ $(Im\ A\ f)\ p$.

Lemma *In_Image_elim* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)$,
injective $f \rightarrow \forall\ x{:}U$, *In* _ $(Im\ A\ f)\ (f\ x) \rightarrow In$ _ $A\ x$.

Lemma *injective_preserves_cardinal* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat)$,
injective $f \rightarrow$
cardinal _ $A\ n \rightarrow \forall\ n'{:}nat$, *cardinal* _ $(Im\ A\ f)\ n' \rightarrow n' = n$.

Lemma *cardinal_decreases* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat)$,
cardinal $U\ A\ n \rightarrow \forall\ n'{:}nat$, *cardinal* $V\ (Im\ A\ f)\ n' \rightarrow n' \leq n$.

Theorem *Pigeonhole* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat)$,
cardinal $U\ A\ n \rightarrow$
$\forall\ n'{:}nat$, *cardinal* $V\ (Im\ A\ f)\ n' \rightarrow n' < n \rightarrow \neg$ *injective* f.

Lemma *Pigeonhole_principle* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat)$,
cardinal _ $A\ n \rightarrow$
$\forall\ n'{:}nat$,
cardinal _ $(Im\ A\ f)\ n' \rightarrow$
$n' < n \rightarrow \exists\ x : \_, (\exists\ y : \_, f\ x = f\ y \wedge x \neq y)$.

End *Image*.

Hint *Resolve Im_def image_empty finite_image*: *sets v62*.

Chapter 157

Module Coq.Sets.Infinite_sets

```
Require Export Finite_sets.
Require Export Constructive_sets.
Require Export Classical_Type.
Require Export Classical_sets.
Require Export Powerset.
Require Export Powerset_facts.
Require Export Powerset_Classical_facts.
Require Export Gt.
Require Export Lt.
Require Export Le.
Require Export Finite_sets_facts.
Require Export Image.

Section Approx.
  Variable U : Type.

  Inductive Approximant (A X:Ensemble U) : Prop :=
    Defn_of_Approximant : Finite U X → Included U X A → Approximant A X.
End Approx.

Hint Resolve Defn_of_Approximant.

Section Infinite_sets.
  Variable U : Type.

  Lemma make_new_approximant :
    ∀ A X:Ensemble U,
      ¬ Finite U A → Approximant U A X → Inhabited U (Setminus U A X).

  Lemma approximants_grow :
    ∀ A X:Ensemble U,
      ¬ Finite U A →
      ∀ n:nat,
        cardinal U X n →
        Included U X A → ∃ Y : _, cardinal U Y (S n) ∧ Included U Y A.

  Lemma approximants_grow' :
```

$\forall\ A\ X{:}Ensemble\ U,$
$\neg\ Finite\ U\ A \rightarrow$
$\forall\ n{:}nat,$
$cardinal\ U\ X\ n \rightarrow$
$Approximant\ U\ A\ X \rightarrow$
$\exists\ Y : \_,\ cardinal\ U\ Y\ (S\ n) \wedge Approximant\ U\ A\ Y.$

Lemma *approximant_can_be_any_size* :
$\forall\ A\ X{:}Ensemble\ U,$
$\neg\ Finite\ U\ A \rightarrow$
$\forall\ n{:}nat, \exists\ Y : \_,\ cardinal\ U\ Y\ n \wedge Approximant\ U\ A\ Y.$

Variable V : Type.

Theorem *Image_set_continuous* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (X{:}Ensemble\ V),$
$Finite\ V\ X \rightarrow$
$Included\ V\ X\ (Im\ U\ V\ A\ f) \rightarrow$
$\exists\ n : \_,$
$(\exists\ Y : \_,\ (cardinal\ U\ Y\ n \wedge Included\ U\ Y\ A) \wedge Im\ U\ V\ Y\ f = X).$

Theorem *Image_set_continuous'* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (X{:}Ensemble\ V),$
$Approximant\ V\ (Im\ U\ V\ A\ f)\ X \rightarrow$
$\exists\ Y : \_,\ Approximant\ U\ A\ Y \wedge Im\ U\ V\ Y\ f = X.$

Theorem *Pigeonhole_bis* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V),$
$\neg\ Finite\ U\ A \rightarrow Finite\ V\ (Im\ U\ V\ A\ f) \rightarrow \neg\ injective\ U\ V\ f.$

Theorem *Pigeonhole_ter* :
$\forall\ (A{:}Ensemble\ U)\ (f{:}U \rightarrow V)\ (n{:}nat),$
$injective\ U\ V\ f \rightarrow Finite\ V\ (Im\ U\ V\ A\ f) \rightarrow Finite\ U\ A.$

End *Infinite_sets*.

Chapter 158

Module Coq.Sets.Integers

Require Export *Finite_sets*.
Require Export *Constructive_sets*.
Require Export *Classical_Type*.
Require Export *Classical_sets*.
Require Export *Powerset*.
Require Export *Powerset_facts*.
Require Export *Powerset_Classical_facts*.
Require Export *Gt*.
Require Export *Lt*.
Require Export *Le*.
Require Export *Finite_sets_facts*.
Require Export *Image*.
Require Export *Infinite_sets*.
Require Export *Compare_dec*.
Require Export *Relations_1*.
Require Export *Partial_Order*.
Require Export *Cpo*.

Section *Integers_sect*.

Inductive *Integers* : *Ensemble nat* :=
Integers_defn : ∀ *x*:*nat*, *In nat Integers x*.

Lemma *le_reflexive* : *Reflexive nat le*.

Lemma *le_antisym* : *Antisymmetric nat le*.

Lemma *le_trans* : *Transitive nat le*.

Lemma *le_Order* : *Order nat le*.

Lemma *triv_nat* : ∀ *n*:*nat*, *In nat Integers n*.

Definition *nat_po* : *PO nat*.

Lemma *le_total_order* : *Totally_ordered nat nat_po Integers*.

Lemma *Finite_subset_has_lub* :

$\forall$ *X*:*Ensemble nat*,
Finite nat X $\rightarrow$ $\exists$ *m* : *nat*, *Upper_Bound nat nat_po X m*.

Lemma *Integers_has_no_ub* :
$\neg$ ($\exists$ *m* : *nat*, *Upper_Bound nat nat_po Integers m*).

Lemma *Integers_infinite* : $\neg$ *Finite nat Integers*.

End *Integers_sect*.

Chapter 159

Module Coq.Sets.Multiset

```
Require Import Permut.

Section multiset_defs.

  Variable A : Set.
  Variable eqA : A → A → Prop.
  Hypothesis Aeq_dec : ∀ x y:A, {eqA x y} + {~ eqA x y}.

  Inductive multiset : Set :=
    Bag : (A → nat) → multiset.

  Definition EmptyBag := Bag (fun a:A ⇒ 0).
  Definition SingletonBag (a:A) :=
    Bag (fun a':A ⇒ match Aeq_dec a a' with
                      | left _ ⇒ 1
                      | right _ ⇒ 0
                    end).

  Definition multiplicity (m:multiset) (a:A) : nat := let (f) := m in f a.
```

multiset equality

```
  Definition meq (m1 m2:multiset) :=
    ∀ a:A, multiplicity m1 a = multiplicity m2 a.

  Lemma meq_refl : ∀ x:multiset, meq x x.

  Lemma meq_trans : ∀ x y z:multiset, meq x y → meq y z → meq x z.

  Lemma meq_sym : ∀ x y:multiset, meq x y → meq y x.
```

multiset union

```
  Definition munion (m1 m2:multiset) :=
    Bag (fun a:A ⇒ multiplicity m1 a + multiplicity m2 a).

  Lemma munion_empty_left : ∀ x:multiset, meq x (munion EmptyBag x).

  Lemma munion_empty_right : ∀ x:multiset, meq x (munion x EmptyBag).

  Require Plus.
```

Lemma *munion_comm* : ∀ *x y*:*multiset*, *meq* (*munion x y*) (*munion y x*).

Lemma *munion_ass* :
∀ *x y z*:*multiset*, *meq* (*munion* (*munion x y*) *z*) (*munion x* (*munion y z*)).

Lemma *meq_left* :
∀ *x y z*:*multiset*, *meq x y* → *meq* (*munion x z*) (*munion y z*).

Lemma *meq_right* :
∀ *x y z*:*multiset*, *meq x y* → *meq* (*munion z x*) (*munion z y*).

Here we should make multiset an abstract datatype, by hiding *Bag*, *munion*, *multiplicity*; all further properties are proved abstractly

Lemma *munion_rotate* :
∀ *x y z*:*multiset*, *meq* (*munion x* (*munion y z*)) (*munion z* (*munion x y*)).

Lemma *meq_congr* :
∀ *x y z t*:*multiset*, *meq x y* → *meq z t* → *meq* (*munion x z*) (*munion y t*).

Lemma *munion_perm_left* :
∀ *x y z*:*multiset*, *meq* (*munion x* (*munion y z*)) (*munion y* (*munion x z*)).

Lemma *multiset_twist1* :
∀ *x y z t*:*multiset*,
meq (*munion x* (*munion* (*munion y z*) *t*)) (*munion* (*munion y* (*munion x t*)) *z*).

Lemma *multiset_twist2* :
∀ *x y z t*:*multiset*,
meq (*munion x* (*munion* (*munion y z*) *t*)) (*munion* (*munion y* (*munion x z*)) *t*).

specific for treesort

Lemma *treesort_twist1* :
∀ *x y z t u*:*multiset*,
meq u (*munion y z*) →
meq (*munion x* (*munion u t*)) (*munion* (*munion y* (*munion x t*)) *z*).

Lemma *treesort_twist2* :
∀ *x y z t u*:*multiset*,
meq u (*munion y z*) →
meq (*munion x* (*munion u t*)) (*munion* (*munion y* (*munion x z*)) *t*).

End *multiset_defs*.

Hint *Unfold meq multiplicity*: *v62 datatypes*.
Hint *Resolve munion_empty_right munion_comm munion_ass meq_left meq_right munion_empty_left*: *v62 datatypes*.
Hint Immediate *meq_sym*: *v62 datatypes*.

Chapter 160

Module Coq.Sets.Partial_Order

```
Require Export Ensembles.
Require Export Relations_1.

Section Partial_orders.
  Variable U : Type.

  Definition Carrier := Ensemble U.

  Definition Rel := Relation U.

  Record PO : Type := Definition_of_PO
    { Carrier_of : Ensemble U;
      Rel_of : Relation U;
      PO_cond1 : Inhabited U Carrier_of;
      PO_cond2 : Order U Rel_of }.
  Variable p : PO.

  Definition Strict_Rel_of : Rel := fun x y:U ⇒ Rel_of p x y ∧ x ≠ y.

  Inductive covers (y x:U) : Prop :=
    Definition_of_covers :
    Strict_Rel_of x y →
    ¬ (∃ z : _, Strict_Rel_of x z ∧ Strict_Rel_of z y) →
    covers y x.

End Partial_orders.

Hint Unfold Carrier_of Rel_of Strict_Rel_of: sets v62.
Hint Resolve Definition_of_covers: sets v62.

Section Partial_order_facts.
  Variable U : Type.
  Variable D : PO U.

  Lemma Strict_Rel_Transitive_with_Rel :
    ∀ x y z:U,
      Strict_Rel_of U D x y → Rel_of U D y z → Strict_Rel_of U D x z.

  Lemma Strict_Rel_Transitive_with_Rel_left :
```

$\forall\ x\ y\ z{:}U,$
$\quad Rel\_of\ U\ D\ x\ y \to Strict\_Rel\_of\ U\ D\ y\ z \to Strict\_Rel\_of\ U\ D\ x\ z.$

Lemma *Strict_Rel_Transitive* : *Transitive U* (*Strict_Rel_of U D*).
End *Partial_order_facts*.

Chapter 161

Module Coq.Sets.Permut

We consider a Set U, given with a commutative-associative operator op, and a congruence $cong$; we show permutation lemmas

Section *Axiomatisation*.

Variable U : Set.
Variable op : $U \to U \to U$.
Variable $cong$: $U \to U \to$ Prop.

Hypothesis op_comm : $\forall\ x\ y{:}U,\ cong\ (op\ x\ y)\ (op\ y\ x)$.
Hypothesis op_ass : $\forall\ x\ y\ z{:}U,\ cong\ (op\ (op\ x\ y)\ z)\ (op\ x\ (op\ y\ z))$.

Hypothesis $cong\_left$: $\forall\ x\ y\ z{:}U,\ cong\ x\ y \to cong\ (op\ x\ z)\ (op\ y\ z)$.
Hypothesis $cong\_right$: $\forall\ x\ y\ z{:}U,\ cong\ x\ y \to cong\ (op\ z\ x)\ (op\ z\ y)$.
Hypothesis $cong\_trans$: $\forall\ x\ y\ z{:}U,\ cong\ x\ y \to cong\ y\ z \to cong\ x\ z$.
Hypothesis $cong\_sym$: $\forall\ x\ y{:}U,\ cong\ x\ y \to cong\ y\ x$.

Remark. we do not need: Hypothesis $cong\_refl$: $(x{:}U)(cong\ x\ x)$.

Lemma $cong\_congr$:
$\forall\ x\ y\ z\ t{:}U,\ cong\ x\ y \to cong\ z\ t \to cong\ (op\ x\ z)\ (op\ y\ t)$.

Lemma $comm\_right$: $\forall\ x\ y\ z{:}U,\ cong\ (op\ x\ (op\ y\ z))\ (op\ x\ (op\ z\ y))$.

Lemma $comm\_left$: $\forall\ x\ y\ z{:}U,\ cong\ (op\ (op\ x\ y)\ z)\ (op\ (op\ y\ x)\ z)$.

Lemma $perm\_right$: $\forall\ x\ y\ z{:}U,\ cong\ (op\ (op\ x\ y)\ z)\ (op\ (op\ x\ z)\ y)$.

Lemma $perm\_left$: $\forall\ x\ y\ z{:}U,\ cong\ (op\ x\ (op\ y\ z))\ (op\ y\ (op\ x\ z))$.

Lemma op_rotate : $\forall\ x\ y\ z\ t{:}U,\ cong\ (op\ x\ (op\ y\ z))\ (op\ z\ (op\ x\ y))$.

Needed for treesort ...
Lemma $twist$:
$\forall\ x\ y\ z\ t{:}U,\ cong\ (op\ x\ (op\ (op\ y\ z)\ t))\ (op\ (op\ y\ (op\ x\ t))\ z)$.

End *Axiomatisation*.

Chapter 162

Module Coq.Sets.Powerset_Classical_facts

Require Export *Ensembles*.
Require Export *Constructive_sets*.
Require Export *Relations_1*.
Require Export *Relations_1_facts*.
Require Export *Partial_Order*.
Require Export *Cpo*.
Require Export *Powerset*.
Require Export *Powerset_facts*.
Require Export *Classical_Type*.
Require Export *Classical_sets*.

Section *Sets_as_an_algebra*.

Variable U : Type.

Lemma *sincl_add_x* :
$\forall$ (A B:*Ensemble* U) (x:U),
$\neg$ *In* U A x $\rightarrow$
Strict_Included U (*Add* U A x) (*Add* U B x) $\rightarrow$ *Strict_Included* U A B.

Lemma *incl_soustr_in* :
$\forall$ (X:*Ensemble* U) (x:U), *In* U X x $\rightarrow$ *Included* U (*Subtract* U X x) X.

Lemma *incl_soustr* :
$\forall$ (X Y:*Ensemble* U) (x:U),
Included U X Y $\rightarrow$ *Included* U (*Subtract* U X x) (*Subtract* U Y x).

Lemma *incl_soustr_add_l* :
$\forall$ (X:*Ensemble* U) (x:U), *Included* U (*Subtract* U (*Add* U X x) x) X.

Lemma *incl_soustr_add_r* :
$\forall$ (X:*Ensemble* U) (x:U),
$\neg$ *In* U X x $\rightarrow$ *Included* U X (*Subtract* U (*Add* U X x) x).
Hint *Resolve incl_soustr_add_r*: *sets v62*.

Lemma *add_soustr_2* :
∀ (*X*:*Ensemble U*) (*x*:*U*),
In U X x → *Included U X* (*Add U* (*Subtract U X x*) *x*).

Lemma *add_soustr_1* :
∀ (*X*:*Ensemble U*) (*x*:*U*),
In U X x → *Included U* (*Add U* (*Subtract U X x*) *x*) *X*.

Lemma *add_soustr_xy* :
∀ (*X*:*Ensemble U*) (*x y*:*U*),
x ≠ *y* → *Subtract U* (*Add U X x*) *y* = *Add U* (*Subtract U X y*) *x*.

Lemma *incl_st_add_soustr* :
∀ (*X Y*:*Ensemble U*) (*x*:*U*),
¬ *In U X x* →
Strict_Included U (*Add U X x*) *Y* → *Strict_Included U X* (*Subtract U Y x*).

Lemma *Sub_Add_new* :
∀ (*X*:*Ensemble U*) (*x*:*U*), ¬ *In U X x* → *X* = *Subtract U* (*Add U X x*) *x*.

Lemma *Simplify_add* :
∀ (*X X0*:*Ensemble U*) (*x*:*U*),
¬ *In U X x* → ¬ *In U X0 x* → *Add U X x* = *Add U X0 x* → *X* = *X0*.

Lemma *Included_Add* :
∀ (*X A*:*Ensemble U*) (*x*:*U*),
Included U X (*Add U A x*) →
Included U X A ∨ (∃ *A'* : _, *X* = *Add U A' x* ∧ *Included U A' A*).

Lemma *setcover_inv* :
∀ *A x y*:*Ensemble U*,
covers (*Ensemble U*) (*Power_set_PO U A*) *y x* →
Strict_Included U x y ∧
(∀ *z*:*Ensemble U*, *Included U x z* → *Included U z y* → *x* = *z* ∨ *z* = *y*).

Theorem *Add_covers* :
∀ *A a*:*Ensemble U*,
Included U a A →
∀ *x*:*U*,
In U A x →
¬ *In U a x* → *covers* (*Ensemble U*) (*Power_set_PO U A*) (*Add U a x*) *a*.

Theorem *covers_Add* :
∀ *A a a'*:*Ensemble U*,
Included U a A →
Included U a' A →
covers (*Ensemble U*) (*Power_set_PO U A*) *a' a* →
∃ *x* : _, *a'* = *Add U a x* ∧ *In U A x* ∧ ¬ *In U a x*.

Theorem *covers_is_Add* :
∀ *A a a'*:*Ensemble U*,

Included U a A →
Included U a' A →
(covers (Ensemble U) (Power_set_PO U A) a' a ↔
(∃ x : _, a' = Add U a x ∧ In U A x ∧ ¬ In U a x)).

Theorem *Singleton_atomic* :
∀ *(x:U) (A:Ensemble U)*,
In U A x →
covers (Ensemble U) (Power_set_PO U A) (Singleton U x) (Empty_set U).

Lemma *less_than_singleton* :
∀ *(X:Ensemble U) (x:U)*,
Strict_Included U X (Singleton U x) → X = Empty_set U.

End *Sets_as_an_algebra.*

Hint *Resolve incl_soustr_in: sets v62.*
Hint *Resolve incl_soustr: sets v62.*
Hint *Resolve incl_soustr_add_l: sets v62.*
Hint *Resolve incl_soustr_add_r: sets v62.*
Hint *Resolve add_soustr_1 add_soustr_2: sets v62.*
Hint *Resolve add_soustr_xy: sets v62.*

Chapter 163

Module Coq.Sets.Powerset_facts

Require Export *Ensembles*.
Require Export *Constructive_sets*.
Require Export *Relations_1*.
Require Export *Relations_1_facts*.
Require Export *Partial_Order*.
Require Export *Cpo*.
Require Export *Powerset*.

Section *Sets_as_an_algebra*.
Variable U : Type.

Theorem *Empty_set_zero* : $\forall$ X:*Ensemble* U, *Union* U (*Empty_set* U) X = X.

Theorem *Empty_set_zero'* : $\forall$ x:U, *Add* U (*Empty_set* U) x = *Singleton* U x.

Lemma *less_than_empty* :
$\forall$ X:*Ensemble* U, *Included* U X (*Empty_set* U) $\rightarrow$ X = *Empty_set* U.

Theorem *Union_commutative* : $\forall$ A B:*Ensemble* U, *Union* U A B = *Union* U B A.

Theorem *Union_associative* :
$\forall$ A B C:*Ensemble* U, *Union* U (*Union* U A B) C = *Union* U A (*Union* U B C).

Theorem *Union_idempotent* : $\forall$ A:*Ensemble* U, *Union* U A A = A.

Lemma *Union_absorbs* :
$\forall$ A B:*Ensemble* U, *Included* U B A $\rightarrow$ *Union* U A B = A.

Theorem *Couple_as_union* :
$\forall$ x y:U, *Union* U (*Singleton* U x) (*Singleton* U y) = *Couple* U x y.

Theorem *Triple_as_union* :
$\forall$ x y z:U,
Union U (*Union* U (*Singleton* U x) (*Singleton* U y)) (*Singleton* U z) =
Triple U x y z.

Theorem *Triple_as_Couple* : $\forall$ x y:U, *Couple* U x y = *Triple* U x x y.

Theorem *Triple_as_Couple_Singleton* :

$\forall\ x\ y\ z{:}U,\ Triple\ U\ x\ y\ z = Union\ U\ (Couple\ U\ x\ y)\ (Singleton\ U\ z).$

Theorem *Intersection_commutative* :
$\forall\ A\ B{:}Ensemble\ U,\ Intersection\ U\ A\ B = Intersection\ U\ B\ A.$

Theorem *Distributivity* :
$\forall\ A\ B\ C{:}Ensemble\ U,$
$Intersection\ U\ A\ (Union\ U\ B\ C) =$
$Union\ U\ (Intersection\ U\ A\ B)\ (Intersection\ U\ A\ C).$

Theorem *Distributivity'* :
$\forall\ A\ B\ C{:}Ensemble\ U,$
$Union\ U\ A\ (Intersection\ U\ B\ C) =$
$Intersection\ U\ (Union\ U\ A\ B)\ (Union\ U\ A\ C).$

Theorem *Union_add* :
$\forall\ (A\ B{:}Ensemble\ U)\ (x{:}U),\ Add\ U\ (Union\ U\ A\ B)\ x = Union\ U\ A\ (Add\ U\ B\ x).$

Theorem *Non_disjoint_union* :
$\forall\ (X{:}Ensemble\ U)\ (x{:}U),\ In\ U\ X\ x \to Add\ U\ X\ x = X.$

Theorem *Non_disjoint_union'* :
$\forall\ (X{:}Ensemble\ U)\ (x{:}U),\ \neg\ In\ U\ X\ x \to Subtract\ U\ X\ x = X.$

Lemma *singlx* : $\forall\ x\ y{:}U,\ In\ U\ (Add\ U\ (Empty\_set\ U)\ x)\ y \to x = y.$

Lemma *incl_add* :
$\forall\ (A\ B{:}Ensemble\ U)\ (x{:}U),$
$Included\ U\ A\ B \to Included\ U\ (Add\ U\ A\ x)\ (Add\ U\ B\ x).$

Lemma *incl_add_x* :
$\forall\ (A\ B{:}Ensemble\ U)\ (x{:}U),$
$\neg\ In\ U\ A\ x \to Included\ U\ (Add\ U\ A\ x)\ (Add\ U\ B\ x) \to Included\ U\ A\ B.$

Lemma *Add_commutative* :
$\forall\ (A{:}Ensemble\ U)\ (x\ y{:}U),\ Add\ U\ (Add\ U\ A\ x)\ y = Add\ U\ (Add\ U\ A\ y)\ x.$

Lemma *Add_commutative'* :
$\forall\ (A{:}Ensemble\ U)\ (x\ y\ z{:}U),$
$Add\ U\ (Add\ U\ (Add\ U\ A\ x)\ y)\ z = Add\ U\ (Add\ U\ (Add\ U\ A\ z)\ x)\ y.$

Lemma *Add_distributes* :
$\forall\ (A\ B{:}Ensemble\ U)\ (x\ y{:}U),$
$Included\ U\ B\ A \to Add\ U\ (Add\ U\ A\ x)\ y = Union\ U\ (Add\ U\ A\ x)\ (Add\ U\ B\ y).$

Lemma *setcover_intro* :
$\forall\ (U{:}\mathsf{Type})\ (A\ x\ y{:}Ensemble\ U),$
$Strict\_Included\ U\ x\ y \to$
$\neg\ (\exists\ z : \_,\ Strict\_Included\ U\ x\ z \wedge Strict\_Included\ U\ z\ y) \to$
$covers\ (Ensemble\ U)\ (Power\_set\_PO\ U\ A)\ y\ x.$

End *Sets_as_an_algebra.*

Hint *Resolve Empty_set_zero Empty_set_zero' Union_associative Union_add singlx incl_add: sets v62.*

Chapter 164

Module Coq.Sets.Powerset

Require Export *Ensembles*.
Require Export *Relations_1*.
Require Export *Relations_1_facts*.
Require Export *Partial_Order*.
Require Export *Cpo*.

Section *The_power_set_partial_order*.
Variable U : Type.

Inductive *Power_set* (*A*:*Ensemble U*) : *Ensemble* (*Ensemble U*) :=
Definition_of_Power_set :
∀ *X*:*Ensemble U*, *Included U X A* → *In* (*Ensemble U*) (*Power_set A*) *X*.
Hint *Resolve Definition_of_Power_set*.

Theorem *Empty_set_minimal* : ∀ *X*:*Ensemble U*, *Included U* (*Empty_set U*) *X*.
Hint *Resolve Empty_set_minimal*.

Theorem *Power_set_Inhabited* :
∀ *X*:*Ensemble U*, *Inhabited* (*Ensemble U*) (*Power_set X*).
Hint *Resolve Power_set_Inhabited*.

Theorem *Inclusion_is_an_order* : *Order* (*Ensemble U*) (*Included U*).
Hint *Resolve Inclusion_is_an_order*.

Theorem *Inclusion_is_transitive* : *Transitive* (*Ensemble U*) (*Included U*).
Hint *Resolve Inclusion_is_transitive*.

Definition *Power_set_PO* : *Ensemble U* → *PO* (*Ensemble U*).
Hint *Unfold Power_set_PO*.

Theorem *Strict_Rel_is_Strict_Included* :
same_relation (*Ensemble U*) (*Strict_Included U*)
(*Strict_Rel_of* (*Ensemble U*) (*Power_set_PO* (*Full_set U*))).
Hint *Resolve Strict_Rel_Transitive Strict_Rel_is_Strict_Included*.

Lemma *Strict_inclusion_is_transitive_with_inclusion* :
∀ *x y z*:*Ensemble U*,
Strict_Included U x y → *Included U y z* → *Strict_Included U x z*.

Lemma *Strict_inclusion_is_transitive_with_inclusion_left* :
∀ *x y z*:*Ensemble U*,
Included U x y → *Strict_Included U y z* → *Strict_Included U x z*.

Lemma *Strict_inclusion_is_transitive* :
Transitive (*Ensemble U*) (*Strict_Included U*).

Theorem *Empty_set_is_Bottom* :
∀ *A*:*Ensemble U*, *Bottom* (*Ensemble U*) (*Power_set_PO A*) (*Empty_set U*).
Hint *Resolve Empty_set_is_Bottom*.

Theorem *Union_minimal* :
∀ *a b X*:*Ensemble U*,
Included U a X → *Included U b X* → *Included U* (*Union U a b*) *X*.
Hint *Resolve Union_minimal*.

Theorem *Intersection_maximal* :
∀ *a b X*:*Ensemble U*,
Included U X a → *Included U X b* → *Included U X* (*Intersection U a b*).

Theorem *Union_increases_l* : ∀ *a b*:*Ensemble U*, *Included U a* (*Union U a b*).

Theorem *Union_increases_r* : ∀ *a b*:*Ensemble U*, *Included U b* (*Union U a b*).

Theorem *Intersection_decreases_l* :
∀ *a b*:*Ensemble U*, *Included U* (*Intersection U a b*) *a*.

Theorem *Intersection_decreases_r* :
∀ *a b*:*Ensemble U*, *Included U* (*Intersection U a b*) *b*.
Hint *Resolve Union_increases_l Union_increases_r Intersection_decreases_l*
Intersection_decreases_r.

Theorem *Union_is_Lub* :
∀ *A a b*:*Ensemble U*,
Included U a A →
Included U b A →
Lub (*Ensemble U*) (*Power_set_PO A*) (*Couple* (*Ensemble U*) *a b*) (*Union U a b*).

Theorem *Intersection_is_Glb* :
∀ *A a b*:*Ensemble U*,
Included U a A →
Included U b A →
Glb (*Ensemble U*) (*Power_set_PO A*) (*Couple* (*Ensemble U*) *a b*)
(*Intersection U a b*).

End *The_power_set_partial_order*.

Hint *Resolve Empty_set_minimal*: *sets v62*.
Hint *Resolve Power_set_Inhabited*: *sets v62*.
Hint *Resolve Inclusion_is_an_order*: *sets v62*.
Hint *Resolve Inclusion_is_transitive*: *sets v62*.
Hint *Resolve Union_minimal*: *sets v62*.
Hint *Resolve Union_increases_l*: *sets v62*.

Hint *Resolve Union_increases_r*: *sets v62*.
Hint *Resolve Intersection_decreases_l*: *sets v62*.
Hint *Resolve Intersection_decreases_r*: *sets v62*.
Hint *Resolve Empty_set_is_Bottom*: *sets v62*.
Hint *Resolve Strict_inclusion_is_transitive*: *sets v62*.

Chapter 165

Module Coq.Sets.Relations_1_facts

Require Export *Relations_1*.

Definition *Complement* (*U*:Type) (*R*:*Relation U*) : *Relation U* :=
  fun *x y*:*U* ⇒ ¬ *R x y*.

Theorem *Rsym_imp_notRsym* :
 ∀ (*U*:Type) (*R*:*Relation U*),
   Symmetric U R → *Symmetric U* (*Complement U R*).

Theorem *Equiv_from_preorder* :
 ∀ (*U*:Type) (*R*:*Relation U*),
   Preorder U R → *Equivalence U* (fun *x y*:*U* ⇒ *R x y* ∧ *R y x*).
Hint *Resolve Equiv_from_preorder*.

Theorem *Equiv_from_order* :
 ∀ (*U*:Type) (*R*:*Relation U*),
   Order U R → *Equivalence U* (fun *x y*:*U* ⇒ *R x y* ∧ *R y x*).
Hint *Resolve Equiv_from_order*.

Theorem *contains_is_preorder* :
 ∀ *U*:Type, *Preorder* (*Relation U*) (*contains U*).
Hint *Resolve contains_is_preorder*.

Theorem *same_relation_is_equivalence* :
 ∀ *U*:Type, *Equivalence* (*Relation U*) (*same_relation U*).
Hint *Resolve same_relation_is_equivalence*.

Theorem *cong_reflexive_same_relation* :
 ∀ (*U*:Type) (*R R'*:*Relation U*),
   same_relation U R R' → *Reflexive U R* → *Reflexive U R'*.

Theorem *cong_symmetric_same_relation* :
 ∀ (*U*:Type) (*R R'*:*Relation U*),
   same_relation U R R' → *Symmetric U R* → *Symmetric U R'*.

Theorem *cong_antisymmetric_same_relation* :
 ∀ (*U*:Type) (*R R'*:*Relation U*),
   same_relation U R R' → *Antisymmetric U R* → *Antisymmetric U R'*.

Theorem *cong_transitive_same_relation* :
∀ (*U*:Type) (*R R'*:*Relation U*),
same_relation U R R' → *Transitive U R* → *Transitive U R'*.

Chapter 166

Module Coq.Sets.Relations_1

```
Section Relations_1.
  Variable U : Type.

  Definition Relation := U → U → Prop.
  Variable R : Relation.

  Definition Reflexive : Prop := ∀ x:U, R x x.

  Definition Transitive : Prop := ∀ x y z:U, R x y → R y z → R x z.

  Definition Symmetric : Prop := ∀ x y:U, R x y → R y x.

  Definition Antisymmetric : Prop := ∀ x y:U, R x y → R y x → x = y.

  Definition contains (R R':Relation) : Prop :=
    ∀ x y:U, R' x y → R x y.

  Definition same_relation (R R':Relation) : Prop :=
    contains R R' ∧ contains R' R.

  Inductive Preorder : Prop :=
      Definition_of_preorder : Reflexive → Transitive → Preorder.

  Inductive Order : Prop :=
      Definition_of_order :
        Reflexive → Transitive → Antisymmetric → Order.

  Inductive Equivalence : Prop :=
      Definition_of_equivalence :
        Reflexive → Transitive → Symmetric → Equivalence.

  Inductive PER : Prop :=
      Definition_of_PER : Symmetric → Transitive → PER.

End Relations_1.
Hint Unfold Reflexive Transitive Antisymmetric Symmetric contains
  same_relation: sets v62.
Hint Resolve Definition_of_preorder Definition_of_order
  Definition_of_equivalence Definition_of_PER: sets v62.
```

Chapter 167

Module Coq.Sets.Relations_2_facts

Require Export *Relations_1*.
Require Export *Relations_1_facts*.
Require Export *Relations_2*.

Theorem *Rstar_reflexive* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *Reflexive U* (*Rstar U R*).

Theorem *Rplus_contains_R* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *contains U* (*Rplus U R*) *R*.

Theorem *Rstar_contains_R* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *contains U* (*Rstar U R*) *R*.

Theorem *Rstar_contains_Rplus* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *contains U* (*Rstar U R*) (*Rplus U R*).

Theorem *Rstar_transitive* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *Transitive U* (*Rstar U R*).

Theorem *Rstar_cases* :
$\forall$ (*U*:Type) (*R*:*Relation U*) (*x y*:*U*),
Rstar U R x y $\rightarrow$ *x* = *y* $\vee$ ($\exists$ *u* : _, *R x u* $\wedge$ *Rstar U R u y*).

Theorem *Rstar_equiv_Rstar1* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *same_relation U* (*Rstar U R*) (*Rstar1 U R*).

Theorem *Rsym_imp_Rstarsym* :
$\forall$ (*U*:Type) (*R*:*Relation U*), *Symmetric U R* $\rightarrow$ *Symmetric U* (*Rstar U R*).

Theorem *Sstar_contains_Rstar* :
$\forall$ (*U*:Type) (*R S*:*Relation U*),
contains U (*Rstar U S*) *R* $\rightarrow$ *contains U* (*Rstar U S*) (*Rstar U R*).

Theorem *star_monotone* :
$\forall$ (*U*:Type) (*R S*:*Relation U*),
contains U S R $\rightarrow$ *contains U* (*Rstar U S*) (*Rstar U R*).

Theorem *RstarRplus_RRstar* :
$\forall$ (*U*:Type) (*R*:*Relation U*) (*x y z*:*U*),

Rstar U R x y → Rplus U R y z → ∃ u : _, R x u ∧ Rstar U R u z.

Theorem *Lemma1* :
∀ (*U*:Type) (*R*:*Relation U*),
Strongly_confluent U R →
∀ *x b*:*U*,
Rstar U R x b →
∀ *a*:*U*, *R x a* → ∃ *z* : _, *Rstar U R a z* ∧ *R b z.*

Chapter 168

Module Coq.Sets.Relations_2

Require Export *Relations_1*.

Section *Relations_2*.
Variable U : Type.
Variable R : *Relation* U.

Inductive *Rstar* $(x{:}U) : U \to$ Prop :=
  | *Rstar_0* : *Rstar* $x\ x$
  | *Rstar_n* : $\forall\ y\ z{:}U,\ R\ x\ y \to$ *Rstar* $y\ z \to$ *Rstar* $x\ z$.

Inductive *Rstar1* $(x{:}U) : U \to$ Prop :=
  | *Rstar1_0* : *Rstar1* $x\ x$
  | *Rstar1_1* : $\forall\ y{:}U,\ R\ x\ y \to$ *Rstar1* $x\ y$
  | *Rstar1_n* : $\forall\ y\ z{:}U,$ *Rstar1* $x\ y \to$ *Rstar1* $y\ z \to$ *Rstar1* $x\ z$.

Inductive *Rplus* $(x{:}U) : U \to$ Prop :=
  | *Rplus_0* : $\forall\ y{:}U,\ R\ x\ y \to$ *Rplus* $x\ y$
  | *Rplus_n* : $\forall\ y\ z{:}U,\ R\ x\ y \to$ *Rplus* $y\ z \to$ *Rplus* $x\ z$.

Definition *Strongly_confluent* : Prop :=
  $\forall\ x\ a\ b{:}U,\ R\ x\ a \to R\ x\ b \to ex$ (fun $z{:}U \Rightarrow R\ a\ z \wedge R\ b\ z$).

End *Relations_2*.

Hint *Resolve Rstar_0*: *sets v62*.
Hint *Resolve Rstar1_0*: *sets v62*.
Hint *Resolve Rstar1_1*: *sets v62*.
Hint *Resolve Rplus_0*: *sets v62*.

Chapter 169

Module Coq.Sets.Relations_3_facts

Require Export *Relations_1*.
Require Export *Relations_1_facts*.
Require Export *Relations_2*.
Require Export *Relations_2_facts*.
Require Export *Relations_3*.

Theorem *Rstar_imp_coherent* :
∀ (*U*:Type) (*R*:*Relation U*) (*x y*:*U*), *Rstar U R x y* → *coherent U R x y*.
Hint *Resolve Rstar_imp_coherent*.

Theorem *coherent_symmetric* :
∀ (*U*:Type) (*R*:*Relation U*), *Symmetric U* (*coherent U R*).

Theorem *Strong_confluence* :
∀ (*U*:Type) (*R*:*Relation U*), *Strongly_confluent U R* → *Confluent U R*.

Theorem *Strong_confluence_direct* :
∀ (*U*:Type) (*R*:*Relation U*), *Strongly_confluent U R* → *Confluent U R*.

Theorem *Noetherian_contains_Noetherian* :
∀ (*U*:Type) (*R R'*:*Relation U*),
Noetherian U R → *contains U R R'* → *Noetherian U R'*.

Theorem *Newman* :
∀ (*U*:Type) (*R*:*Relation U*),
Noetherian U R → *Locally_confluent U R* → *Confluent U R*.

Chapter 170

Module Coq.Sets.Relations_3

```
Require Export Relations_1.
Require Export Relations_2.

Section Relations_3.
  Variable U : Type.
  Variable R : Relation U.

  Definition coherent (x y:U) : Prop :=
    ∃ z : _, Rstar U R x z ∧ Rstar U R y z.

  Definition locally_confluent (x:U) : Prop :=
    ∀ y z:U, R x y → R x z → coherent y z.

  Definition Locally_confluent : Prop := ∀ x:U, locally_confluent x.

  Definition confluent (x:U) : Prop :=
    ∀ y z:U, Rstar U R x y → Rstar U R x z → coherent y z.

  Definition Confluent : Prop := ∀ x:U, confluent x.

  Inductive noetherian (x: U) : Prop :=
      definition_of_noetherian :
        (∀ y:U, R x y → noetherian y) → noetherian x.

  Definition Noetherian : Prop := ∀ x:U, noetherian x.

End Relations_3.
Hint Unfold coherent: sets v62.
Hint Unfold locally_confluent: sets v62.
Hint Unfold confluent: sets v62.
Hint Unfold Confluent: sets v62.
Hint Resolve definition_of_noetherian: sets v62.
Hint Unfold Noetherian: sets v62.
```

Chapter 171

Module Coq.Sets.Uniset

Sets as characteristic functions

```
Require Import Bool.

Section defs.

Variable A : Set.
Variable eqA : A → A → Prop.
Hypothesis eqA_dec : ∀ x y:A, {eqA x y} + {~ eqA x y}.

Inductive uniset : Set :=
    Charac : (A → bool) → uniset.

Definition charac (s:uniset) (a:A) : bool := let (f) := s in f a.

Definition Emptyset := Charac (fun a:A ⇒ false).

Definition Fullset := Charac (fun a:A ⇒ true).

Definition Singleton (a:A) :=
  Charac
    (fun a':A ⇒
       match eqA_dec a a' with
       | left h ⇒ true
       | right h ⇒ false
       end).

Definition In (s:uniset) (a:A) : Prop := charac s a = true.
Hint Unfold In.

uniset inclusion
Definition incl (s1 s2:uniset) := ∀ a:A, leb (charac s1 a) (charac s2 a).
Hint Unfold incl.

uniset equality
Definition seq (s1 s2:uniset) := ∀ a:A, charac s1 a = charac s2 a.
Hint Unfold seq.

Lemma leb_refl : ∀ b:bool, leb b b.
```

Hint *Resolve leb_refl.*

Lemma *incl_left* : ∀ *s1 s2*:*uniset*, *seq s1 s2* → *incl s1 s2.*

Lemma *incl_right* : ∀ *s1 s2*:*uniset*, *seq s1 s2* → *incl s2 s1.*

Lemma *seq_refl* : ∀ *x*:*uniset*, *seq x x.*
Hint *Resolve seq_refl.*

Lemma *seq_trans* : ∀ *x y z*:*uniset*, *seq x y* → *seq y z* → *seq x z.*

Lemma *seq_sym* : ∀ *x y*:*uniset*, *seq x y* → *seq y x.*

uniset union
Definition *union* (*m1 m2*:*uniset*) :=
Charac (fun *a*:*A* ⇒ *orb* (*charac m1 a*) (*charac m2 a*)).

Lemma *union_empty_left* : ∀ *x*:*uniset*, *seq x* (*union Emptyset x*).
Hint *Resolve union_empty_left.*

Lemma *union_empty_right* : ∀ *x*:*uniset*, *seq x* (*union x Emptyset*).
Hint *Resolve union_empty_right.*

Lemma *union_comm* : ∀ *x y*:*uniset*, *seq* (*union x y*) (*union y x*).
Hint *Resolve union_comm.*

Lemma *union_ass* :
∀ *x y z*:*uniset*, *seq* (*union* (*union x y*) *z*) (*union x* (*union y z*)).
Hint *Resolve union_ass.*

Lemma *seq_left* : ∀ *x y z*:*uniset*, *seq x y* → *seq* (*union x z*) (*union y z*).
Hint *Resolve seq_left.*

Lemma *seq_right* : ∀ *x y z*:*uniset*, *seq x y* → *seq* (*union z x*) (*union z y*).
Hint *Resolve seq_right.*

All the proofs that follow duplicate *Multiset_of_A*

Here we should make uniset an abstract datatype, by hiding *Charac*, *union*, *charac*; all further properties are proved abstractly

Require Import *Permut.*

Lemma *union_rotate* :
∀ *x y z*:*uniset*, *seq* (*union x* (*union y z*)) (*union z* (*union x y*)).

Lemma *seq_congr* :
∀ *x y z t*:*uniset*, *seq x y* → *seq z t* → *seq* (*union x z*) (*union y t*).

Lemma *union_perm_left* :
∀ *x y z*:*uniset*, *seq* (*union x* (*union y z*)) (*union y* (*union x z*)).

Lemma *uniset_twist1* :
∀ *x y z t*:*uniset*,
seq (*union x* (*union* (*union y z*) *t*)) (*union* (*union y* (*union x t*)) *z*).

Lemma *uniset_twist2* :

$\forall\ x\ y\ z\ t{:}uniset,$
$seq\ (union\ x\ (union\ (union\ y\ z)\ t))\ (union\ (union\ y\ (union\ x\ z))\ t).$

specific for treesort

Lemma *treesort_twist1* :
$\forall\ x\ y\ z\ t\ u{:}uniset,$
$seq\ u\ (union\ y\ z) \rightarrow$
$seq\ (union\ x\ (union\ u\ t))\ (union\ (union\ y\ (union\ x\ t))\ z).$

Lemma *treesort_twist2* :
$\forall\ x\ y\ z\ t\ u{:}uniset,$
$seq\ u\ (union\ y\ z) \rightarrow$
$seq\ (union\ x\ (union\ u\ t))\ (union\ (union\ y\ (union\ x\ z))\ t).$

End *defs*.

Chapter 172

Module Coq.Relations.Newman

Require Import *Rstar*.

Section *Newman*.

Variable A : Type.
Variable R : $A \to A \to$ Prop.

Let *Rstar* := *Rstar A R*.
Let *Rstar_reflexive* := *Rstar_reflexive A R*.
Let *Rstar_transitive* := *Rstar_transitive A R*.
Let *Rstar_Rstar'* := *Rstar_Rstar' A R*.

Definition *coherence* (x y:A) := *ex2* (*Rstar x*) (*Rstar y*).

Theorem *coherence_intro* :
$\forall$ x y z:A, *Rstar x z* $\to$ *Rstar y z* $\to$ *coherence x y*.

A very simple case of coherence :

Lemma *Rstar_coherence* : $\forall$ x y:A, *Rstar x y* $\to$ *coherence x y*.

coherence is symmetric
Lemma *coherence_sym* : $\forall$ x y:A, *coherence x y* $\to$ *coherence y x*.

Definition *confluence* (x:A) :=
$\forall$ y z:A, *Rstar x y* $\to$ *Rstar x z* $\to$ *coherence y z*.

Definition *local_confluence* (x:A) :=
$\forall$ y z:A, R x y $\to$ R x z $\to$ *coherence y z*.

Definition *noetherian* :=
$\forall$ (x:A) (P:$A \to$ Prop),
($\forall$ y:A, ($\forall$ z:A, R y z $\to$ P z) $\to$ P y) $\to$ P x.

Section *Newman_section*.

The general hypotheses of the theorem

Hypothesis *Hyp1* : *noetherian*.
Hypothesis *Hyp2* : $\forall$ x:A, *local_confluence x*.

The induction hypothesis

Section *Induct*.
Variable x : A.
Hypothesis *hyp_ind* : $\forall$ u:A, R x u $\rightarrow$ *confluence* u.

Confluence in x

Variables y z : A.
Hypothesis *h1* : *Rstar* x y.
Hypothesis *h2* : *Rstar* x z.

particular case $x \rightarrow u$ and u->*y

Section *Newman_*.
Variable u : A.
Hypothesis *t1* : R x u.
Hypothesis *t2* : *Rstar* u y.

In the usual diagram, we assume also $x \rightarrow v$ and v->*z

Theorem *Diagram* : $\forall$ (v:A) (*u1*:R x v) (*u2*:*Rstar* v z), *coherence* y z.

Theorem *caseRxy* : *coherence* y z.
End *Newman_*.

Theorem *Ind_proof* : *coherence* y z.
End *Induct*.

Theorem *Newman* : $\forall$ x:A, *confluence* x.

End *Newman_section*.

End *Newman*.

Chapter 173

Module Coq.Relations.Operators_Properties

```
Require Import Relation_Definitions.
Require Import Relation_Operators.
Section Properties.
  Variable A : Type.
  Variable R : relation A.
  Let incl (R1 R2:relation A) : Prop := ∀ x y:A, R1 x y → R2 x y.
  Section Clos_Refl_Trans.
    Lemma clos_rt_is_preorder : preorder A (clos_refl_trans A R).
    Lemma clos_rt_idempotent :
      incl (clos_refl_trans A (clos_refl_trans A R)) (clos_refl_trans A R).
    Lemma clos_refl_trans_ind_left :
      ∀ (A:Type) (R:A → A → Prop) (M:A) (P:A → Prop),
        P M →
        (∀ P0 N:A, clos_refl_trans A R M P0 → P P0 → R P0 N → P N) →
        ∀ a:A, clos_refl_trans A R M a → P a.
  End Clos_Refl_Trans.
  Section Clos_Refl_Sym_Trans.
    Lemma clos_rt_clos_rst :
      inclusion A (clos_refl_trans A R) (clos_refl_sym_trans A R).
    Lemma clos_rst_is_equiv : equivalence A (clos_refl_sym_trans A R).
    Lemma clos_rst_idempotent :
      incl (clos_refl_sym_trans A (clos_refl_sym_trans A R))
      (clos_refl_sym_trans A R).
  End Clos_Refl_Sym_Trans.
End Properties.
```

Chapter 174

Module Coq.Relations.Relation_Definitions

```
Section Relation_Definition.
  Variable A : Type.
  Definition relation := A → A → Prop.
  Variable R : relation.
  Section General_Properties_of_Relations.
    Definition reflexive : Prop := ∀ x:A, R x x.
    Definition transitive : Prop := ∀ x y z:A, R x y → R y z → R x z.
    Definition symmetric : Prop := ∀ x y:A, R x y → R y x.
    Definition antisymmetric : Prop := ∀ x y:A, R x y → R y x → x = y.
    Definition equiv := reflexive ∧ transitive ∧ symmetric.
  End General_Properties_of_Relations.
  Section Sets_of_Relations.
    Record preorder : Prop :=
      { preord_refl : reflexive; preord_trans : transitive}.
    Record order : Prop :=
      { ord_refl : reflexive;
        ord_trans : transitive;
        ord_antisym : antisymmetric}.
    Record equivalence : Prop :=
      { equiv_refl : reflexive;
        equiv_trans : transitive;
        equiv_sym : symmetric}.
    Record PER : Prop := {per_sym : symmetric; per_trans : transitive}.
  End Sets_of_Relations.
  Section Relations_of_Relations.
```

Definition *inclusion* (*R1 R2*:*relation*) : Prop :=
  ∀ *x y*:*A*, *R1 x y* → *R2 x y*.

Definition *same_relation* (*R1 R2*:*relation*) : Prop :=
  inclusion R1 R2 ∧ *inclusion R2 R1*.

Definition *commut* (*R1 R2*:*relation*) : Prop :=
  ∀ *x y*:*A*,
    R1 y x → ∀ *z*:*A*, *R2 z y* → *exists2 y'* : *A*, *R2 y' x* & *R1 z y'*.

End *Relations_of_Relations*.

End *Relation_Definition*.

Hint *Unfold reflexive transitive antisymmetric symmetric*: *sets v62*.

Hint *Resolve Build_preorder Build_order Build_equivalence Build_PER preord_refl preord_trans ord_refl ord_trans ord_antisym equiv_refl equiv_trans equiv_sym per_sym per_trans*: *sets v62*.

Hint *Unfold inclusion same_relation commut*: *sets v62*.

Chapter 175

Module Coq.Relations.Relation_Operators

```
Require Import Relation_Definitions.
Require Import List.
```

Some operators to build relations

```
Section Transitive_Closure.
  Variable A : Type.
  Variable R : relation A.

  Inductive clos_trans (x: A) : A → Prop :=
    | t_step : ∀ y:A, R x y → clos_trans x y
    | t_trans :
        ∀ y z:A, clos_trans x y → clos_trans y z → clos_trans x z.
End Transitive_Closure.

Section Reflexive_Transitive_Closure.
  Variable A : Type.
  Variable R : relation A.

  Inductive clos_refl_trans (x:A) : A → Prop:=
    | rt_step : ∀ y:A, R x y → clos_refl_trans x y
    | rt_refl : clos_refl_trans x x
    | rt_trans :
        ∀ y z:A,
          clos_refl_trans x y → clos_refl_trans y z → clos_refl_trans x z.
End Reflexive_Transitive_Closure.

Section Reflexive_Symetric_Transitive_Closure.
  Variable A : Type.
  Variable R : relation A.

  Inductive clos_refl_sym_trans : relation A :=
    | rst_step : ∀ x y:A, R x y → clos_refl_sym_trans x y
    | rst_refl : ∀ x:A, clos_refl_sym_trans x x
```

```
    | rst_sym :
      ∀ x y:A, clos_refl_sym_trans x y → clos_refl_sym_trans y x
    | rst_trans :
      ∀ x y z:A,
        clos_refl_sym_trans x y →
        clos_refl_sym_trans y z → clos_refl_sym_trans x z.
End Reflexive_Symetric_Transitive_Closure.

Section Transposee.
  Variable A : Type.
  Variable R : relation A.

  Definition transp (x y:A) := R y x.
End Transposee.

Section Union.
  Variable A : Type.
  Variables R1 R2 : relation A.

  Definition union (x y:A) := R1 x y ∨ R2 x y.
End Union.

Section Disjoint_Union.
Variables A B : Type.
Variable leA : A → A → Prop.
Variable leB : B → B → Prop.

Inductive le_AsB : A + B → A + B → Prop :=
  | le_aa : ∀ x y:A, leA x y → le_AsB (inl B x) (inl B y)
  | le_ab : ∀ (x:A) (y:B), le_AsB (inl B x) (inr A y)
  | le_bb : ∀ x y:B, leB x y → le_AsB (inr A x) (inr A y).

End Disjoint_Union.

Section Lexicographic_Product.
  Variable A : Type.
  Variable B : A → Type.
  Variable leA : A → A → Prop.
  Variable leB : ∀ x:A, B x → B x → Prop.

  Inductive lexprod : sigS B → sigS B → Prop :=
    | left_lex :
      ∀ (x x':A) (y:B x) (y':B x'),
        leA x x' → lexprod (existS B x y) (existS B x' y')
    | right_lex :
      ∀ (x:A) (y y':B x),
        leB x y y' → lexprod (existS B x y) (existS B x y').
End Lexicographic_Product.

Section Symmetric_Product.
  Variable A : Type.
```

```
  Variable B : Type.
  Variable leA : A → A → Prop.
  Variable leB : B → B → Prop.

  Inductive symprod : A × B → A × B → Prop :=
    | left_sym :
      ∀ x x':A, leA x x' → ∀ y:B, symprod (x, y) (x', y)
    | right_sym :
      ∀ y y':B, leB y y' → ∀ x:A, symprod (x, y) (x, y').

End Symmetric_Product.

Section Swap.
  Variable A : Type.
  Variable R : A → A → Prop.

  Inductive swapprod : A × A → A × A → Prop :=
    | sp_noswap : ∀ x x':A × A, symprod A A R R x x' → swapprod x x'
    | sp_swap :
      ∀ (x y:A) (p:A × A),
        symprod A A R R (x, y) p → swapprod (y, x) p.
End Swap.

Section Lexicographic_Exponentiation.

  Variable A : Set.
  Variable leA : A → A → Prop.
  Let Nil := nil (A:=A).
  Let List := list A.

  Inductive Ltl : List → List → Prop :=
    | Lt_nil : ∀ (a:A) (x:List), Ltl Nil (a :: x)
    | Lt_hd : ∀ a b:A, leA a b → ∀ x y:list A, Ltl (a :: x) (b :: y)
    | Lt_tl : ∀ (a:A) (x y:List), Ltl x y → Ltl (a :: x) (a :: y).

  Inductive Desc : List → Prop :=
    | d_nil : Desc Nil
    | d_one : ∀ x:A, Desc (x :: Nil)
    | d_conc :
      ∀ (x y:A) (l:List),
        leA x y → Desc (l ++ y :: Nil) → Desc ((l ++ y :: Nil) ++ x :: Nil).

  Definition Pow : Set := sig Desc.

  Definition lex_exp (a b:Pow) : Prop := Ltl (proj1_sig a) (proj1_sig b).

End Lexicographic_Exponentiation.

Hint Unfold transp union: sets v62.
Hint Resolve t_step rt_step rt_refl rst_step rst_refl: sets v62.
Hint Immediate rst_sym: sets v62.
```

Chapter 176

Module Coq.Relations.Relations

Require Export *Relation_Definitions*.
Require Export *Relation_Operators*.
Require Export *Operators_Properties*.

Lemma *inverse_image_of_equivalence* :
  $\forall$ (A B:Type) (f:$A \to B$) (r:*relation* B),
    equivalence B $r \to$ *equivalence* A (fun x y:$A \Rightarrow r$ (f x) (f y)).

Lemma *inverse_image_of_eq* :
  $\forall$ (A B:Type) (f:$A \to B$), *equivalence* A (fun x y:$A \Rightarrow f\ x = f\ y$).

Chapter 177

Module Coq.Relations.Rstar

Properties of a binary relation R on type A

Section *Rstar*.

Variable A : Type.
Variable R : $A \to A \to$ Prop.

Definition of the reflexive-transitive closure $R\times$ of R
Smallest reflexive P containing R *o* P

Definition *Rstar* (x y:A) :=
$\forall$ P:$A \to A \to$ Prop,
($\forall$ u:A, P u u) $\to$ ($\forall$ u v w:A, R u $v \to P$ v $w \to P$ u w) $\to P$ x y.

Theorem *Rstar_reflexive* : $\forall$ x:A, *Rstar* x x.

Theorem *Rstar_R* : $\forall$ x y z:A, R x $y \to$ *Rstar* y $z \to$ *Rstar* x z.

We conclude with transitivity of *Rstar* :

Theorem *Rstar_transitive* :
$\forall$ x y z:A, *Rstar* x $y \to$ *Rstar* y $z \to$ *Rstar* x z.

Another characterization of $R\times$
Smallest reflexive P containing R *o* $R\times$

Definition *Rstar'* (x y:A) :=
$\forall$ P:$A \to A \to$ Prop,
P x $x \to$ ($\forall$ u:A, R x $u \to$ *Rstar* u $y \to P$ x y) $\to P$ x y.

Theorem *Rstar'_reflexive* : $\forall$ x:A, *Rstar'* x x.

Theorem *Rstar'_R* : $\forall$ x y z:A, R x $z \to$ *Rstar* z $y \to$ *Rstar'* x y.

Equivalence of the two definitions:

Theorem *Rstar'_Rstar* : $\forall$ x y:A, *Rstar'* x $y \to$ *Rstar* x y.

Theorem *Rstar_Rstar'* : $\forall$ x y:A, *Rstar* x $y \to$ *Rstar'* x y.

Property of Commutativity of two relations

```
Definition commut (A:Set) (R1 R2:A → A → Prop) :=
  ∀ x y:A,
    R1 y x → ∀ z:A, R2 z y → exists2 y' : A, R2 y' x & R1 z y'.

End Rstar.
```

Chapter 178

Module Coq.Sorting.Heap

A development of Treesort on Heap trees

Require Import *List*.
Require Import *Multiset*.
Require Import *Permutation*.
Require Import *Relations*.
Require Import *Sorting*.

Section *defs*.

178.1 Trees and heap trees

178.1.1 Definition of trees over an ordered set

Variable A : Set.
Variable leA : $relation\ A$.
Variable eqA : $relation\ A$.

Let $gtA\ (x\ y{:}A) := \neg\ leA\ x\ y$.

Hypothesis leA_dec : $\forall\ x\ y{:}A, \{leA\ x\ y\} + \{leA\ y\ x\}$.
Hypothesis eqA_dec : $\forall\ x\ y{:}A, \{eqA\ x\ y\} + \{\tilde{}\ eqA\ x\ y\}$.
Hypothesis leA_refl : $\forall\ x\ y{:}A, eqA\ x\ y \rightarrow leA\ x\ y$.
Hypothesis leA_trans : $\forall\ x\ y\ z{:}A, leA\ x\ y \rightarrow leA\ y\ z \rightarrow leA\ x\ z$.
Hypothesis $leA\_antisym$: $\forall\ x\ y{:}A, leA\ x\ y \rightarrow leA\ y\ x \rightarrow eqA\ x\ y$.

Hint *Resolve* leA_refl.
Hint Immediate $eqA\_dec\ leA\_dec\ leA\_antisym$.

Let $emptyBag := EmptyBag\ A$.
Let $singletonBag := SingletonBag\ \_\ eqA\_dec$.

Inductive *Tree* : Set :=
  | *Tree_Leaf* : *Tree*

```
  | Tree_Node : A → Tree → Tree → Tree.
```

a is lower than a Tree T if T is a Leaf or T is a Node holding $b{>}a$

```
Definition leA_Tree (a:A) (t:Tree) :=
  match t with
    | Tree_Leaf ⇒ True
    | Tree_Node b T1 T2 ⇒ leA a b
  end.

Lemma leA_Tree_Leaf : ∀ a:A, leA_Tree a Tree_Leaf.

Lemma leA_Tree_Node :
  ∀ (a b:A) (G D:Tree), leA a b → leA_Tree a (Tree_Node b G D).
```

178.1.2 The heap property

```
Inductive is_heap : Tree → Prop :=
  | nil_is_heap : is_heap Tree_Leaf
  | node_is_heap :
    ∀ (a:A) (T1 T2:Tree),
      leA_Tree a T1 →
      leA_Tree a T2 →
      is_heap T1 → is_heap T2 → is_heap (Tree_Node a T1 T2).

Lemma invert_heap :
  ∀ (a:A) (T1 T2:Tree),
    is_heap (Tree_Node a T1 T2) →
    leA_Tree a T1 ∧ leA_Tree a T2 ∧ is_heap T1 ∧ is_heap T2.

Lemma is_heap_rec :
  ∀ P:Tree → Set,
    P Tree_Leaf →
    (∀ (a:A) (T1 T2:Tree),
      leA_Tree a T1 →
      leA_Tree a T2 →
      is_heap T1 → P T1 → is_heap T2 → P T2 → P (Tree_Node a T1 T2)) →
    ∀ T:Tree, is_heap T → P T.

Lemma low_trans :
  ∀ (T:Tree) (a b:A), leA a b → leA_Tree b T → leA_Tree a T.
```

178.1.3 From trees to multisets

contents of a tree as a multiset

Nota Bene : In what follows the definition of SingletonBag in not used. Actually, we could just take as postulate: Parameter *SingletonBag* : *A→multiset*.

```
Fixpoint contents (t:Tree) : multiset A :=
```

```
  match t with
    | Tree_Leaf ⇒ emptyBag
    | Tree_Node a t1 t2 ⇒
      munion (contents t1) (munion (contents t2) (singletonBag a))
  end.
```

equivalence of two trees is equality of corresponding multisets

```
Definition equiv_Tree (t1 t2:Tree) := meq (contents t1) (contents t2).
```

178.2 From lists to sorted lists

178.2.1 Specification of heap insertion

```
Inductive insert_spec (a:A) (T:Tree) : Set :=
  insert_exist :
  ∀ T1:Tree,
    is_heap T1 →
    meq (contents T1) (munion (contents T) (singletonBag a)) →
    (∀ b:A, leA b a → leA_Tree b T → leA_Tree b T1) →
    insert_spec a T.

Lemma insert : ∀ T:Tree, is_heap T → ∀ a:A, insert_spec a T.
```

178.2.2 Building a heap from a list

```
Inductive build_heap (l:list A) : Set :=
  heap_exist :
  ∀ T:Tree,
    is_heap T →
    meq (list_contents _ eqA_dec l) (contents T) → build_heap l.

Lemma list_to_heap : ∀ l:list A, build_heap l.
```

178.2.3 Building the sorted list

```
Inductive flat_spec (T:Tree) : Set :=
  flat_exist :
  ∀ l:list A,
    sort leA l →
    (∀ a:A, leA_Tree a T → lelistA leA a l) →
    meq (contents T) (list_contents _ eqA_dec l) → flat_spec T.

Lemma heap_to_list : ∀ T:Tree, is_heap T → flat_spec T.
```

178.3 Specification of treesort

Theorem *treesort* :
∀ *l*:*list A*, {*m* : *list A* | *sort leA m* & *permutation* _ *eqA*_*dec l m*}.

End *defs*.

Chapter 179

Module Coq.Sorting.Permutation

Require Import *Relations*.
Require Import *List*.
Require Import *Multiset*.
Require Import *Arith*.

This file define a notion of permutation for lists, based on multisets: there exists a permutation between two lists iff every elements have the same multiplicities in the two lists.

Unlike *List.Permutation*, the present notion of permutation requires a decidable equality. At the same time, this definition can be used with a non-standard equality, whereas *List.Permutation* cannot.

The present file contains basic results, obtained without any particular assumption on the decidable equality used.

File *PermutSetoid* contains additional results about permutations with respect to an setoid equality (i.e. an equivalence relation).

Finally, file *PermutEq* concerns Coq equality : this file is similar to the previous one, but proves in addition that *List.Permutation* and *permutation* are equivalent in this context. x

Section *defs*.

179.1 From lists to multisets

Variable A : Set.
Variable eqA : $relation\ A$.
Hypothesis eqA_dec : $\forall\ x\ y{:}A, \{eqA\ x\ y\} + \{\tilde{}\ eqA\ x\ y\}$.

Let $emptyBag$:= $EmptyBag\ A$.
Let $singletonBag$:= $SingletonBag\ \_\ eqA\_dec$.

contents of a list

Fixpoint $list\_contents\ (l{:}list\ A)$: $multiset\ A$:=
match l with

```
    | nil ⇒ emptyBag
    | a :: l ⇒ munion (singletonBag a) (list_contents l)
  end.

Lemma list_contents_app :
  ∀ l m:list A,
    meq (list_contents (l ++ m)) (munion (list_contents l) (list_contents m)).
```

179.2 *permutation*: definition and basic properties

```
Definition permutation (l m:list A) :=
  meq (list_contents l) (list_contents m).

Lemma permut_refl : ∀ l:list A, permutation l l.

Lemma permut_sym :
  ∀ l1 l2 : list A, permutation l1 l2 → permutation l2 l1.

Lemma permut_tran :
  ∀ l m n:list A, permutation l m → permutation m n → permutation l n.

Lemma permut_cons :
  ∀ l m:list A,
    permutation l m → ∀ a:A, permutation (a :: l) (a :: m).

Lemma permut_app :
  ∀ l l' m m':list A,
    permutation l l' → permutation m m' → permutation (l ++ m) (l' ++ m').

Lemma permut_add_inside :
  ∀ a l1 l2 l3 l4,
    permutation (l1 ++ l2) (l3 ++ l4) →
    permutation (l1 ++ a :: l2) (l3 ++ a :: l4).

Lemma permut_add_cons_inside :
  ∀ a l l1 l2,
    permutation l (l1 ++ l2) →
    permutation (a :: l) (l1 ++ a :: l2).

Lemma permut_middle :
  ∀ (l m:list A) (a:A), permutation (a :: l ++ m) (l ++ a :: m).

Lemma permut_sym_app :
  ∀ l1 l2, permutation (l1 ++ l2) (l2 ++ l1).

Lemma permut_rev :
  ∀ l, permutation l (rev l).
```

179.3 Some inversion results.

Lemma *permut_conv_inv* :
∀ *e l1 l2*, *permutation* (*e* :: *l1*) (*e* :: *l2*) → *permutation l1 l2*.

Lemma *permut_app_inv1* :
∀ *l l1 l2*, *permutation* (*l1* ++ *l*) (*l2* ++ *l*) → *permutation l1 l2*.

Lemma *permut_app_inv2* :
∀ *l l1 l2*, *permutation* (*l* ++ *l1*) (*l* ++ *l2*) → *permutation l1 l2*.

Lemma *permut_remove_hd* :
∀ *l l1 l2 a*,
permutation (*a* :: *l*) (*l1* ++ *a* :: *l2*) → *permutation l* (*l1* ++ *l2*).

End *defs*.

For compatibilty
Notation *permut_right* := *permut_cons*.

Chapter 180

Module Coq.Sorting.PermutEq

Require Import *Omega*.
Require Import *Relations*.
Require Import *Setoid*.
Require Import *List*.
Require Import *Multiset*.
Require Import *Permutation*.

This file is similar to *PermutSetoid*, except that the equality used here is Coq usual one instead of a setoid equality. In particular, we can then prove the equivalence between *List.Permutation* and *Permutation.permutation*.

Section *Perm*.

Variable A : Set.
Hypothesis *eq_dec* : $\forall\ x\ y{:}A,\ \{x{=}y\} + \{\tilde{}\ x{=}y\}$.

Notation *permutation* := (*permutation* _ *eq_dec*).
Notation *list_contents* := (*list_contents* _ *eq_dec*).

we can use *multiplicity* to define *In* and *NoDup*.

Lemma *multiplicity_In* :
$\forall\ l\ a,\ In\ a\ l \leftrightarrow 0 < multiplicity\ (list\_contents\ l)\ a$.

Lemma *multiplicity_In_O* :
$\forall\ l\ a,\ \neg\ In\ a\ l \rightarrow multiplicity\ (list\_contents\ l)\ a = 0$.

Lemma *multiplicity_In_S* :
$\forall\ l\ a,\ In\ a\ l \rightarrow multiplicity\ (list\_contents\ l)\ a \geq 1$.

Lemma *multiplicity_NoDup* :
$\forall\ l,\ NoDup\ l \leftrightarrow (\forall\ a,\ multiplicity\ (list\_contents\ l)\ a \leq 1)$.

Lemma *NoDup_permut* :
$\forall\ l\ l',\ NoDup\ l \rightarrow NoDup\ l' \rightarrow$
$(\forall\ x,\ In\ x\ l \leftrightarrow In\ x\ l') \rightarrow permutation\ l\ l'$.

Permutation is compatible with In.

Lemma *permut_In_In* :
∀ *l1 l2 e*, *permutation l1 l2* → *In e l1* → *In e l2*.

Lemma *permut_cons_In* :
∀ *l1 l2 e*, *permutation* (*e* :: *l1*) *l2* → *In e l2*.

Permutation of an empty list.

Lemma *permut_nil* :
∀ *l*, *permutation l nil* → *l* = *nil*.

When used with *eq*, this permutation notion is equivalent to the one defined in *List.v*.

Lemma *permutation_Permutation* :
∀ *l l'*, *Permutation l l'* ↔ *permutation l l'*.

Permutation for short lists.

Lemma *permut_length_1*:
∀ *a b*, *permutation* (*a* :: *nil*) (*b* :: *nil*) → *a*=*b*.

Lemma *permut_length_2* :
∀ *a1 b1 a2 b2*, *permutation* (*a1* :: *b1* :: *nil*) (*a2* :: *b2* :: *nil*) →
(*a1*=*a2*) ∧ (*b1*=*b2*) ∨ (*a1*=*b2*) ∧ (*a2*=*b1*).

Permutation is compatible with length.

Lemma *permut_length* :
∀ *l1 l2*, *permutation l1 l2* → *length l1* = *length l2*.

Variable *B* : Set.
Variable *eqB_dec* : ∀ *x y*:*B*, { *x*=*y* }+{ ¬*x*=*y* }.

Permutation is compatible with map.

Lemma *permutation_map* :
∀ *f l1 l2*, *permutation l1 l2* →
Permutation.permutation _ *eqB_dec* (*map f l1*) (*map f l2*).

End *Perm*.

Chapter 181

Module Coq.Sorting.PermutSetoid

Require Import *Omega*.
Require Import *Relations*.
Require Import *List*.
Require Import *Multiset*.
Require Import *Permutation*.
Require Import *SetoidList*.

This file contains additional results about permutations with respect to an setoid equality (i.e. an equivalence relation).

Section *Perm*.

Variable A : Set.
Variable eqA : $relation\ A$.
Hypothesis eqA_dec : $\forall\ x\ y{:}A, \{eqA\ x\ y\} + \{\tilde{}\ eqA\ x\ y\}$.

Notation $permutation := (permutation\ \_\ eqA\_dec)$.
Notation $list\_contents := (list\_contents\ \_\ eqA\_dec)$.

The following lemmas need some knowledge on eqA

Variable eqA_refl : $\forall\ x, eqA\ x\ x$.
Variable eqA_sym : $\forall\ x\ y, eqA\ x\ y \rightarrow eqA\ y\ x$.
Variable eqA_trans : $\forall\ x\ y\ z, eqA\ x\ y \rightarrow eqA\ y\ z \rightarrow eqA\ x\ z$.

we can use *multiplicity* to define InA and $NoDupA$.

Lemma $multiplicity\_InA$:
$\forall\ l\ a, InA\ eqA\ a\ l \leftrightarrow 0 < multiplicity\ (list\_contents\ l)\ a$.

Lemma $multiplicity\_InA\_O$:
$\forall\ l\ a, \neg\ InA\ eqA\ a\ l \rightarrow multiplicity\ (list\_contents\ l)\ a = 0$.

Lemma $multiplicity\_InA\_S$:
$\forall\ l\ a, InA\ eqA\ a\ l \rightarrow multiplicity\ (list\_contents\ l)\ a \geq 1$.

Lemma $multiplicity\_NoDupA$: $\forall\ l$,
$NoDupA\ eqA\ l \leftrightarrow (\forall\ a, multiplicity\ (list\_contents\ l)\ a \leq 1)$.

Permutation is compatible with InA.

Lemma *permut_InA_InA* :
  ∀ *l1 l2 e*, *permutation l1 l2* → *InA eqA e l1* → *InA eqA e l2*.

Lemma *permut_cons_InA* :
  ∀ *l1 l2 e*, *permutation* (*e* :: *l1*) *l2* → *InA eqA e l2*.

Permutation of an empty list.

Lemma *permut_nil* :
  ∀ *l*, *permutation l nil* → *l* = *nil*.

Permutation for short lists.

Lemma *permut_length_1*:
  ∀ *a b*, *permutation* (*a* :: *nil*) (*b* :: *nil*) → *eqA a b*.

Lemma *permut_length_2* :
  ∀ *a1 b1 a2 b2*, *permutation* (*a1* :: *b1* :: *nil*) (*a2* :: *b2* :: *nil*) →
    (*eqA a1 a2*) ∧ (*eqA b1 b2*) ∨ (*eqA a1 b2*) ∧ (*eqA a2 b1*).

Permutation is compatible with length.

Lemma *permut_length* :
  ∀ *l1 l2*, *permutation l1 l2* → *length l1* = *length l2*.

Lemma *NoDupA_eqlistA_permut* :
  ∀ *l l'*, *NoDupA eqA l* → *NoDupA eqA l'* →
    eqlistA eqA l l' → *permutation l l'*.

Variable *B* : Set.
Variable *eqB* : *B*→*B*→Prop.
Variable *eqB_dec* : ∀ *x y*:*B*, { *eqB x y* }+{ ¬*eqB x y* }.
Variable *eqB_trans* : ∀ *x y z*, *eqB x y* → *eqB y z* → *eqB x z*.

Permutation is compatible with map.

Lemma *permut_map* :
  ∀ *f*,
    (∀ *x y*, *eqA x y* → *eqB* (*f x*) (*f y*)) →
    ∀ *l1 l2*, *permutation l1 l2* →
      Permutation.permutation _ *eqB_dec* (*map f l1*) (*map f l2*).

End *Perm*.

Chapter 182

Module Coq.Sorting.Sorting

Require Import *List*.
Require Import *Multiset*.
Require Import *Permutation*.
Require Import *Relations*.

Section *defs*.

Variable A : Set.
Variable leA : $relation\ A$.
Variable eqA : $relation\ A$.

Let $gtA\ (x\ y{:}A) := \neg\ leA\ x\ y$.

Hypothesis leA_dec : $\forall\ x\ y{:}A, \{leA\ x\ y\} + \{leA\ y\ x\}$.
Hypothesis eqA_dec : $\forall\ x\ y{:}A, \{eqA\ x\ y\} + \{\tilde{}\ eqA\ x\ y\}$.
Hypothesis leA_refl : $\forall\ x\ y{:}A, eqA\ x\ y \rightarrow leA\ x\ y$.
Hypothesis leA_trans : $\forall\ x\ y\ z{:}A, leA\ x\ y \rightarrow leA\ y\ z \rightarrow leA\ x\ z$.
Hypothesis $leA\_antisym$: $\forall\ x\ y{:}A, leA\ x\ y \rightarrow leA\ y\ x \rightarrow eqA\ x\ y$.

Hint *Resolve* leA_refl.
Hint Immediate $eqA\_dec\ leA\_dec\ leA\_antisym$.

Let $emptyBag := EmptyBag\ A$.
Let $singletonBag := SingletonBag\ \_\ eqA\_dec$.

lelistA

Inductive $lelistA\ (a{:}A) : list\ A \rightarrow$ Prop :=
| $nil\_leA : lelistA\ a\ nil$
| $cons\_leA : \forall\ (b{:}A)\ (l{:}list\ A), leA\ a\ b \rightarrow lelistA\ a\ (b :: l)$.

Lemma $lelistA\_inv : \forall\ (a\ b{:}A)\ (l{:}list\ A), lelistA\ a\ (b :: l) \rightarrow leA\ a\ b$.

182.1 Definition for a list to be sorted

Inductive $sort : list\ A \rightarrow$ Prop :=

```
  | nil_sort : sort nil
  | cons_sort :
    ∀ (a:A) (l:list A), sort l → lelistA a l → sort (a :: l).

Lemma sort_inv :
  ∀ (a:A) (l:list A), sort (a :: l) → sort l ∧ lelistA a l.

Lemma sort_rec :
  ∀ P:list A → Set,
    P nil →
    (∀ (a:A) (l:list A), sort l → P l → lelistA a l → P (a :: l)) →
    ∀ y:list A, sort y → P y.
```

182.2 Merging two sorted lists

```
Inductive merge_lem (l1 l2:list A) : Set :=
  merge_exist :
  ∀ l:list A,
    sort l →
    meq (list_contents _ eqA_dec l)
    (munion (list_contents _ eqA_dec l1) (list_contents _ eqA_dec l2)) →
    (∀ a:A, lelistA a l1 → lelistA a l2 → lelistA a l) →
    merge_lem l1 l2.

Lemma merge :
  ∀ l1:list A, sort l1 → ∀ l2:list A, sort l2 → merge_lem l1 l2.

End defs.

Hint Constructors sort: datatypes v62.
Hint Constructors lelistA: datatypes v62.
```

Chapter 183

Module Coq.Wellfounded.Disjoint_Union

Author: Cristina Cornes From : Constructing Recursion Operators in Type Theory L. Paulson JSC (1986) 2, 325-355

```
Require Import Relation_Operators.

Section Wf_Disjoint_Union.
  Variables A B : Set.
  Variable leA : A → A → Prop.
  Variable leB : B → B → Prop.

  Notation Le_AsB := (le_AsB A B leA leB).

  Lemma acc_A_sum : ∀ x:A, Acc leA x → Acc Le_AsB (inl B x).

  Lemma acc_B_sum :
    well_founded leA → ∀ x:B, Acc leB x → Acc Le_AsB (inr A x).

  Lemma wf_disjoint_sum :
    well_founded leA → well_founded leB → well_founded Le_AsB.

End Wf_Disjoint_Union.
```

Chapter 184

Module Coq.Wellfounded.Inclusion

Author: Bruno Barras

Require Import *Relation_Definitions*.

Section *WfInclusion*.
  Variable A : Set.
  Variables $R1\ R2 : A \to A \to$ Prop.

  Lemma *Acc_incl* : $inclusion\ A\ R1\ R2 \to \forall\ z{:}A,\ Acc\ R2\ z \to Acc\ R1\ z$.

  Hint *Resolve Acc_incl*.

  Theorem *wf_incl* : $inclusion\ A\ R1\ R2 \to well\_founded\ R2 \to well\_founded\ R1$.

End *WfInclusion*.

Chapter 185

Module Coq.Wellfounded.Inverse_Image

Author: Bruno Barras

Section *Inverse_Image*.

Variables A B : Set.
Variable R : $B \to B \to$ Prop.
Variable f : $A \to B$.

Let Rof $(x\ y{:}A)$: Prop := $R\ (f\ x)\ (f\ y)$.

Remark Acc_lemma : $\forall\ y{:}B,\ Acc\ R\ y \to \forall\ x{:}A,\ y = f\ x \to Acc\ Rof\ x$.

Lemma $Acc\_inverse\_image$: $\forall\ x{:}A,\ Acc\ R\ (f\ x) \to Acc\ Rof\ x$.

Theorem $wf\_inverse\_image$: $well\_founded\ R \to well\_founded\ Rof$.

Variable F : $A \to B \to$ Prop.
Let RoF $(x\ y{:}A)$: Prop :=
$exists2\ b : B,\ F\ x\ b\ \&\ (\forall\ c{:}B,\ F\ y\ c \to R\ b\ c)$.

Lemma $Acc\_inverse\_rel$: $\forall\ b{:}B,\ Acc\ R\ b \to \forall\ x{:}A,\ F\ x\ b \to Acc\ RoF\ x$.

Theorem $wf\_inverse\_rel$: $well\_founded\ R \to well\_founded\ RoF$.

End *Inverse_Image*.

Chapter 186

Module Coq.Wellfounded.Lexicographic_Exponentiatio

Author: Cristina Cornes

From : Constructing Recursion Operators in Type Theory L. Paulson JSC (1986) 2, 325-355

```
Require Import Eqdep.
Require Import List.
Require Import Relation_Operators.
Require Import Transitive_Closure.

Section Wf_Lexicographic_Exponentiation.
  Variable A : Set.
  Variable leA : A → A → Prop.

  Notation Power := (Pow A leA).
  Notation Lex_Exp := (lex_exp A leA).
  Notation ltl := (Ltl A leA).
  Notation Descl := (Desc A leA).

  Notation List := (list A).
  Notation Nil := (nil (A:=A)).
  Notation Cons := (cons (A:=A)).
  Notation "« x , y »" := (exist Descl x y) (at level 0, x, y at level 100).

  Lemma left_prefix : ∀ x y z:List, ltl (x ++ y) z → ltl x z.

  Lemma right_prefix :
    ∀ x y z:List,
      ltl x (y ++ z) → ltl x y ∨ (∃ y' : List, x = y ++ y' ∧ ltl y' z).

  Lemma desc_prefix : ∀ (x:List) (a:A), Descl (x ++ Cons a Nil) → Descl x.

  Lemma desc_tail :
    ∀ (x:List) (a b:A),
      Descl (Cons b (x ++ Cons a Nil)) → clos_trans A leA a b.
```

Lemma *dist_aux* :
∀ *z*:*List*, *Descl z* → ∀ *x y*:*List*, *z* = *x* ++ *y* → *Descl x* ∧ *Descl y*.

Lemma *dist_Desc_concat* :
∀ *x y*:*List*, *Descl* (*x* ++ *y*) → *Descl x* ∧ *Descl y*.

Lemma *desc_end* :
∀ (*a b*:*A*) (*x*:*List*),
Descl (*x* ++ *Cons a Nil*) ∧ *ltl* (*x* ++ *Cons a Nil*) (*Cons b Nil*) →
clos_trans A leA a b.

Lemma *ltl_unit* :
∀ (*x*:*List*) (*a b*:*A*),
Descl (*x* ++ *Cons a Nil*) →
ltl (*x* ++ *Cons a Nil*) (*Cons b Nil*) → *ltl x* (*Cons b Nil*).

Lemma *acc_app* :
∀ (*x1 x2*:*List*) (*y1*:*Descl* (*x1* ++ *x2*)),
Acc Lex_Exp « *x1* ++ *x2*, *y1* » →
∀ (*x*:*List*) (*y*:*Descl x*), *ltl x* (*x1* ++ *x2*) → *Acc Lex_Exp* « *x*, *y* ».

Theorem *wf_lex_exp* : *well_founded leA* → *well_founded Lex_Exp*.

End *Wf_Lexicographic_Exponentiation*.

Chapter 187

Module Coq.Wellfounded.Lexicographic_Product

Authors: Bruno Barras, Cristina Cornes

```
Require Import Eqdep.
Require Import Relation_Operators.
Require Import Transitive_Closure.
```

From : Constructing Recursion Operators in Type Theory L. Paulson JSC (1986) 2, 325-355

```
Section WfLexicographic_Product.
  Variable A : Type.
  Variable B : A → Type.
  Variable leA : A → A → Prop.
  Variable leB : ∀ x:A, B x → B x → Prop.

  Notation LexProd := (lexprod A B leA leB).

  Lemma acc_A_B_lexprod :
    ∀ x:A,
      Acc leA x →
      (∀ x0:A, clos_trans A leA x0 x → well_founded (leB x0)) →
      ∀ y:B x, Acc (leB x) y → Acc LexProd (existS B x y).


  Theorem wf_lexprod :
    well_founded leA →
    (∀ x:A, well_founded (leB x)) → well_founded LexProd.

End WfLexicographic_Product.

Section Wf_Symmetric_Product.
  Variable A : Type.
  Variable B : Type.
  Variable leA : A → A → Prop.
  Variable leB : B → B → Prop.
```

Notation *Symprod* := (*symprod A B leA leB*).

Lemma *Acc_symprod* :
∀ *x*:*A*, *Acc leA x* → ∀ *y*:*B*, *Acc leB y* → *Acc Symprod* (*x*, *y*).

Lemma *wf_symprod* :
well_founded leA → *well_founded leB* → *well_founded Symprod*.

End *Wf_Symmetric_Product*.

Section *Swap*.

Variable *A* : Type.
Variable *R* : *A* → *A* → Prop.

Notation *SwapProd* := (*swapprod A R*).

Lemma *swap_Acc* : ∀ *x y*:*A*, *Acc SwapProd* (*x*, *y*) → *Acc SwapProd* (*y*, *x*).

Lemma *Acc_swapprod* :
∀ *x y*:*A*, *Acc R x* → *Acc R y* → *Acc SwapProd* (*x*, *y*).

Lemma *wf_swapprod* : *well_founded R* → *well_founded SwapProd*.

End *Swap*.

Chapter 188

Module Coq.Wellfounded.Transitive_Closure

Author: Bruno Barras

Require Import *Relation_Definitions*.
Require Import *Relation_Operators*.

Section *Wf_Transitive_Closure*.
Variable A : Type.
Variable R : *relation* A.

Notation *trans_clos* := (*clos_trans* A R).

Lemma *incl_clos_trans* : *inclusion* A R *trans_clos*.

Lemma *Acc_clos_trans* : $\forall$ x:A, *Acc* R x $\rightarrow$ *Acc trans_clos* x.

Hint *Resolve Acc_clos_trans*.

Lemma *Acc_inv_trans* : $\forall$ x y:A, *trans_clos* y x $\rightarrow$ *Acc* R x $\rightarrow$ *Acc* R y.

Theorem *wf_clos_trans* : *well_founded* R $\rightarrow$ *well_founded trans_clos*.

End *Wf_Transitive_Closure*.

Chapter 189

Module Coq.Wellfounded.Union

Author: Bruno Barras

```
Require Import Relation_Operators.
Require Import Relation_Definitions.
Require Import Transitive_Closure.

Section WfUnion.
  Variable A : Set.
  Variables R1 R2 : relation A.

  Notation Union := (union A R1 R2).

  Remark strip_commut :
    commut A R1 R2 →
    ∀ x y:A,
      clos_trans A R1 y x →
      ∀ z:A, R2 z y → exists2 y' : A, R2 y' x & clos_trans A R1 z y'.

  Lemma Acc_union :
    commut A R1 R2 →
    (∀ x:A, Acc R2 x → Acc R1 x) → ∀ a:A, Acc R2 a → Acc Union a.


  Theorem wf_union :
    commut A R1 R2 → well_founded R1 → well_founded R2 → well_founded Union.

End WfUnion.
```

Chapter 190

Module Coq.Wellfounded.Wellfounded

Require Export *Disjoint_Union*.
Require Export *Inclusion*.
Require Export *Inverse_Image*.
Require Export *Lexicographic_Exponentiation*.
Require Export *Lexicographic_Product*.
Require Export *Transitive_Closure*.
Require Export *Union*.
Require Export *Well_Ordering*.

Chapter 191

Module Coq.Wellfounded.Well_Ordering

Author: Cristina Cornes. From: Constructing Recursion Operators in Type Theory L. Paulson JSC (1986) 2, 325-355

```
Require Import Eqdep.

Section WellOrdering.
  Variable A : Type.
  Variable B : A → Type.

  Inductive WO : Type :=
    sup : ∀ (a:A) (f:B a → WO), WO.

  Inductive le_WO : WO → WO → Prop :=
    le_sup : ∀ (a:A) (f:B a → WO) (v:B a), le_WO (f v) (sup a f).

  Theorem wf_WO : well_founded le_WO.

End WellOrdering.

Section Characterisation_wf_relations.

Wellfounded relations are the inverse image of wellordering types
  Variable A : Type.
  Variable leA : A → A → Prop.

  Definition B (a:A) := {x : A | leA x a}.

  Definition wof : well_founded leA → A → WO A B.

End Characterisation_wf_relations.
```

Chapter 192

Module Coq.IntMap.Adalloc

```
Require Import Bool.
Require Import Sumbool.
Require Import Arith.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Nnat.
Require Import Map.
Require Import Fset.

Section AdAlloc.

  Variable A : Set.
```

Allocator: returns an address not in the domain of m. This allocator is optimal in that it returns the lowest possible address, in the usual ordering on integers. It is not the most efficient, however.

```
  Fixpoint ad_alloc_opt (m:Map A) : ad :=
    match m with
    | M0 ⇒ N0
    | M1 a _ ⇒ if Neqb a N0 then Npos 1 else N0
    | M2 m1 m2 ⇒
        Nmin (Ndouble (ad_alloc_opt m1))
          (Ndouble_plus_one (ad_alloc_opt m2))
    end.

  Lemma ad_alloc_opt_allocates_1 :
   ∀ m:Map A, MapGet A m (ad_alloc_opt m) = None.

  Lemma ad_alloc_opt_allocates :
   ∀ m:Map A, in_dom A (ad_alloc_opt m) m = false.
```

Moreover, this is optimal: all addresses below $(ad\_alloc\_opt\ m)$ are in $dom\ m$:

```
  Lemma ad_alloc_opt_optimal_1 :
   ∀ (m:Map A) (a:ad),
     Nle (ad_alloc_opt m) a = false → {y : A | MapGet A m a = Some y}.
```

Lemma $ad\_alloc\_opt\_optimal$:
$\forall\ (m{:}Map\ A)\ (a{:}ad),$
$Nle\ (ad\_alloc\_opt\ m)\ a = false \rightarrow in\_dom\ A\ a\ m = true.$

End $AdAlloc$.

Chapter 193

Module Coq.IntMap.Allmaps

Require Export *Map*.
Require Export *Fset*.
Require Export *Mapaxioms*.
Require Export *Mapiter*.
Require Export *Mapsubset*.
Require Export *Lsort*.
Require Export *Mapfold*.
Require Export *Mapcard*.
Require Export *Mapcanon*.
Require Export *Mapc*.
Require Export *Maplists*.
Require Export *Adalloc*.

Chapter 194

Module Coq.IntMap.Fset

```
Require Import Bool.
Require Import Sumbool.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.

Section Dom.

  Variables A B : Set.

  Fixpoint MapDomRestrTo (m:Map A) : Map B → Map A :=
    match m with
    | M0 ⇒ fun _:Map B ⇒ M0 A
    | M1 a y ⇒
        fun m':Map B ⇒ match MapGet B m' a with
                       | None ⇒ M0 A
                       | _ ⇒ m
                       end
    | M2 m1 m2 ⇒
        fun m':Map B ⇒
          match m' with
          | M0 ⇒ M0 A
          | M1 a' y' ⇒
              match MapGet A m a' with
              | None ⇒ M0 A
              | Some y ⇒ M1 A a' y
              end
          | M2 m'1 m'2 ⇒
              makeM2 A (MapDomRestrTo m1 m'1) (MapDomRestrTo m2 m'2)
          end
    end.

  Lemma MapDomRestrTo_semantics :
   ∀ (m:Map A) (m':Map B),
```

```
    eqm A (MapGet A (MapDomRestrTo m m'))
      (fun a0:ad ⇒
         match MapGet B m' a0 with
         | None ⇒ None
         | _ ⇒ MapGet A m a0
         end).

Fixpoint MapDomRestrBy (m:Map A) : Map B → Map A :=
  match m with
  | M0 ⇒ fun _:Map B ⇒ M0 A
  | M1 a y ⇒
      fun m':Map B ⇒ match MapGet B m' a with
                     | None ⇒ m
                     | _ ⇒ M0 A
                     end
  | M2 m1 m2 ⇒
      fun m':Map B ⇒
        match m' with
        | M0 ⇒ m
        | M1 a' y' ⇒ MapRemove A m a'
        | M2 m'1 m'2 ⇒
            makeM2 A (MapDomRestrBy m1 m'1) (MapDomRestrBy m2 m'2)
        end
  end.

Lemma MapDomRestrBy_semantics :
 ∀ (m:Map A) (m':Map B),
    eqm A (MapGet A (MapDomRestrBy m m'))
      (fun a0:ad ⇒
         match MapGet B m' a0 with
         | None ⇒ MapGet A m a0
         | _ ⇒ None
         end).

Definition in_dom (a:ad) (m:Map A) :=
  match MapGet A m a with
  | None ⇒ false
  | _ ⇒ true
  end.

Lemma in_dom_M0 : ∀ a:ad, in_dom a (M0 A) = false.

Lemma in_dom_M1 : ∀ (a a0:ad) (y:A), in_dom a0 (M1 A a y) = Neqb a a0.

Lemma in_dom_M1_1 : ∀ (a:ad) (y:A), in_dom a (M1 A a y) = true.

Lemma in_dom_M1_2 :
 ∀ (a a0:ad) (y:A), in_dom a0 (M1 A a y) = true → a = a0.

Lemma in_dom_some :
```

$\forall$ (m:*Map A*) (a:*ad*),
in_dom a m = *true* $\rightarrow$ {y : *A* | *MapGet A m a* = *Some y*}.

Lemma *in_dom_none* :
$\forall$ (m:*Map A*) (a:*ad*), *in_dom a m* = *false* $\rightarrow$ *MapGet A m a* = *None*.

Lemma *in_dom_put* :
$\forall$ (m:*Map A*) ($a0$:*ad*) ($y0$:*A*) (a:*ad*),
in_dom a (*MapPut A m a0 y0*) = *orb* (*Neqb a a0*) (*in_dom a m*).

Lemma *in_dom_put_behind* :
$\forall$ (m:*Map A*) ($a0$:*ad*) ($y0$:*A*) (a:*ad*),
in_dom a (*MapPut_behind A m a0 y0*) = *orb* (*Neqb a a0*) (*in_dom a m*).

Lemma *in_dom_remove* :
$\forall$ (m:*Map A*) ($a0$ a:*ad*),
in_dom a (*MapRemove A m a0*) = *andb* (*negb* (*Neqb a a0*)) (*in_dom a m*).

Lemma *in_dom_merge* :
$\forall$ (m m':*Map A*) (a:*ad*),
in_dom a (*MapMerge A m m'*) = *orb* (*in_dom a m*) (*in_dom a m'*).

Lemma *in_dom_delta* :
$\forall$ (m m':*Map A*) (a:*ad*),
in_dom a (*MapDelta A m m'*) = *xorb* (*in_dom a m*) (*in_dom a m'*).

End *Dom*.

Section *InDom*.

Variables *A B* : Set.

Lemma *in_dom_restrto* :
$\forall$ (m:*Map A*) (m':*Map B*) (a:*ad*),
in_dom A a (*MapDomRestrTo A B m m'*) =
andb (*in_dom A a m*) (*in_dom B a m'*).

Lemma *in_dom_restrby* :
$\forall$ (m:*Map A*) (m':*Map B*) (a:*ad*),
in_dom A a (*MapDomRestrBy A B m m'*) =
andb (*in_dom A a m*) (*negb* (*in_dom B a m'*)).

End *InDom*.

Definition *FSet* := *Map unit*.

Section *FSetDefs*.

Variable *A* : Set.

Definition *in_FSet* : *ad* $\rightarrow$ *FSet* $\rightarrow$ *bool* := *in_dom unit*.

Fixpoint *MapDom* (m:*Map A*) : *FSet* :=
match m with
| *M0* $\Rightarrow$ *M0 unit*

```
  | M1 a _ ⇒ M1 unit a tt
  | M2 m m' ⇒ M2 unit (MapDom m) (MapDom m')
  end.

Lemma MapDom_semantics_1 :
 ∀ (m:Map A) (a:ad) (y:A),
   MapGet A m a = Some y → in_FSet a (MapDom m) = true.

Lemma MapDom_semantics_2 :
 ∀ (m:Map A) (a:ad),
   in_FSet a (MapDom m) = true → {y : A | MapGet A m a = Some y}.

Lemma MapDom_semantics_3 :
 ∀ (m:Map A) (a:ad),
   MapGet A m a = None → in_FSet a (MapDom m) = false.

Lemma MapDom_semantics_4 :
 ∀ (m:Map A) (a:ad),
   in_FSet a (MapDom m) = false → MapGet A m a = None.

Lemma MapDom_Dom :
 ∀ (m:Map A) (a:ad), in_dom A a m = in_FSet a (MapDom m).

Definition FSetUnion (s s':FSet) : FSet := MapMerge unit s s'.

Lemma in_FSet_union :
 ∀ (s s':FSet) (a:ad),
   in_FSet a (FSetUnion s s') = orb (in_FSet a s) (in_FSet a s').

Definition FSetInter (s s':FSet) : FSet := MapDomRestrTo unit unit s s'.

Lemma in_FSet_inter :
 ∀ (s s':FSet) (a:ad),
   in_FSet a (FSetInter s s') = andb (in_FSet a s) (in_FSet a s').

Definition FSetDiff (s s':FSet) : FSet := MapDomRestrBy unit unit s s'.

Lemma in_FSet_diff :
 ∀ (s s':FSet) (a:ad),
   in_FSet a (FSetDiff s s') = andb (in_FSet a s) (negb (in_FSet a s')).

Definition FSetDelta (s s':FSet) : FSet := MapDelta unit s s'.

Lemma in_FSet_delta :
 ∀ (s s':FSet) (a:ad),
   in_FSet a (FSetDelta s s') = xorb (in_FSet a s) (in_FSet a s').

End FSetDefs.

Lemma FSet_Dom : ∀ s:FSet, MapDom unit s = s.
```

Chapter 195

Module Coq.IntMap.Lsort

```
Require Import Bool.
Require Import Sumbool.
Require Import Arith.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import List.
Require Import Mapiter.

Section LSort.

  Variable A : Set.

  Fixpoint alist_sorted (l:alist A) : bool :=
    match l with
    | nil ⇒ true
    | (a, _) :: l' ⇒
        match l' with
        | nil ⇒ true
        | (a', y') :: l'' ⇒ andb (Nless a a') (alist_sorted l')
        end
    end.

  Fixpoint alist_nth_ad (n:nat) (l:alist A) {struct l} : ad :=
    match l with
    | nil ⇒ N0        | (a, y) :: l' ⇒ match n with
                          | O ⇒ a
                          | S n' ⇒ alist_nth_ad n' l'
                          end
    end.

  Definition alist_sorted_1 (l:alist A) :=
    ∀ n:nat,
      S (S n) ≤ length l →
```

Nless (*alist_nth_ad n l*) (*alist_nth_ad* (*S n*) *l*) = *true*.

Lemma *alist_sorted_imp_1* :
∀ *l*:*alist A*, *alist_sorted l* = *true* → *alist_sorted_1 l*.

Definition *alist_sorted_2* (*l*:*alist A*) :=
∀ *m n*:*nat*,
m < *n* →
S n ≤ *length l* → *Nless* (*alist_nth_ad m l*) (*alist_nth_ad n l*) = *true*.

Lemma *alist_sorted_1_imp_2* :
∀ *l*:*alist A*, *alist_sorted_1 l* → *alist_sorted_2 l*.

Lemma *alist_sorted_2_imp* :
∀ *l*:*alist A*, *alist_sorted_2 l* → *alist_sorted l* = *true*.

Lemma *app_length* :
∀ (*C*:Set) (*l l'*:*list C*), *length* (*l* ++ *l'*) = *length l* + *length l'*.

Lemma *aapp_length* :
∀ *l l'*:*alist A*, *length* (*aapp A l l'*) = *length l* + *length l'*.

Lemma *alist_nth_ad_aapp_1* :
∀ (*l l'*:*alist A*) (*n*:*nat*),
S n ≤ *length l* → *alist_nth_ad n* (*aapp A l l'*) = *alist_nth_ad n l*.

Lemma *alist_nth_ad_aapp_2* :
∀ (*l l'*:*alist A*) (*n*:*nat*),
S n ≤ *length l'* →
alist_nth_ad (*length l* + *n*) (*aapp A l l'*) = *alist_nth_ad n l'*.

Lemma *interval_split* :
∀ *p q n*:*nat*,
S n ≤ *p* + *q* → {*n'* : *nat* | *S n'* ≤ *q* ∧ *n* = *p* + *n'*} + {*S n* ≤ *p*}.

Lemma *alist_conc_sorted* :
∀ *l l'*:*alist A*,
alist_sorted_2 l →
alist_sorted_2 l' →
(∀ *n n'*:*nat*,
S n ≤ *length l* →
S n' ≤ *length l'* →
Nless (*alist_nth_ad n l*) (*alist_nth_ad n' l'*) = *true*) →
alist_sorted_2 (*aapp A l l'*).

Lemma *alist_nth_ad_semantics* :
∀ (*l*:*alist A*) (*n*:*nat*),
S n ≤ *length l* →
{*y* : *A* | *alist_semantics A l* (*alist_nth_ad n l*) = *Some y*}.

Lemma *alist_of_Map_nth_ad* :
∀ (*m*:*Map A*) (*pf*:*ad* → *ad*) (*l*:*alist A*),
l =

MapFold1 A (alist A) (anil A) (aapp A)
(fun (a0:ad) (y:A) ⇒ acons A (a0, y) (anil A)) pf m →
∀ n:nat, S n ≤ length l → {a' : ad | alist_nth_ad n l = pf a'}.

Definition *ad_monotonic (pf:ad → ad) :=*
∀ a a':ad, Nless a a' = true → Nless (pf a) (pf a') = true.

Lemma *Ndouble_monotonic : ad_monotonic Ndouble.*

Lemma *Ndouble_plus_one_monotonic : ad_monotonic Ndouble_plus_one.*

Lemma *ad_comp_monotonic :*
∀ pf pf':ad → ad,
ad_monotonic pf →
ad_monotonic pf' → ad_monotonic (fun a0:ad ⇒ pf (pf' a0)).

Lemma *ad_comp_double_monotonic :*
∀ pf:ad → ad,
ad_monotonic pf → ad_monotonic (fun a0:ad ⇒ pf (Ndouble a0)).

Lemma *ad_comp_double_plus_un_monotonic :*
∀ pf:ad → ad,
ad_monotonic pf → ad_monotonic (fun a0:ad ⇒ pf (Ndouble_plus_one a0)).

Lemma *alist_of_Map_sorts_1 :*
∀ (m:Map A) (pf:ad → ad),
ad_monotonic pf →
alist_sorted_2
(MapFold1 A (alist A) (anil A) (aapp A)
(fun (a:ad) (y:A) ⇒ acons A (a, y) (anil A)) pf m).

Lemma *alist_of_Map_sorts :*
∀ m:Map A, alist_sorted (alist_of_Map A m) = true.

Lemma *alist_of_Map_sorts1 :*
∀ m:Map A, alist_sorted_1 (alist_of_Map A m).

Lemma *alist_of_Map_sorts2 :*
∀ m:Map A, alist_sorted_2 (alist_of_Map A m).

Lemma *alist_too_low :*
∀ (l:alist A) (a a':ad) (y:A),
Nless a a' = true →
alist_sorted_2 ((a', y) :: l) →
alist_semantics A ((a', y) :: l) a = None.

Lemma *alist_semantics_nth_ad :*
∀ (l:alist A) (a:ad) (y:A),
alist_semantics A l a = Some y →
{n : nat | S n ≤ length l ∧ alist_nth_ad n l = a}.

Lemma *alist_semantics_tail :*
∀ (l:alist A) (a:ad) (y:A),

alist_sorted_2 ((*a*, *y*) :: *l*) →
eqm A (*alist_semantics A l*)
(fun *a0*:*ad* ⇒
if *Neqb a a0* then *None* else *alist_semantics A* ((*a*, *y*) :: *l*) *a0*).

Lemma *alist_semantics_same_tail* :
∀ (*l l'*:*alist A*) (*a*:*ad*) (*y*:*A*),
alist_sorted_2 ((*a*, *y*) :: *l*) →
alist_sorted_2 ((*a*, *y*) :: *l'*) →
eqm A (*alist_semantics A* ((*a*, *y*) :: *l*))
(*alist_semantics A* ((*a*, *y*) :: *l'*)) →
eqm A (*alist_semantics A l*) (*alist_semantics A l'*).

Lemma *alist_sorted_tail* :
∀ (*l*:*alist A*) (*a*:*ad*) (*y*:*A*),
alist_sorted_2 ((*a*, *y*) :: *l*) → *alist_sorted_2 l*.

Lemma *alist_canonical* :
∀ *l l'*:*alist A*,
eqm A (*alist_semantics A l*) (*alist_semantics A l'*) →
alist_sorted_2 l → *alist_sorted_2 l'* → *l* = *l'*.

End *LSort*.

Chapter 196

Module Coq.IntMap.Mapaxioms

```
Require Import Bool.
Require Import Sumbool.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import Fset.

Section MapAxioms.

  Variables A B C : Set.

  Lemma eqm_sym : ∀ f f':ad → option A, eqm A f f' → eqm A f' f.

  Lemma eqm_refl : ∀ f:ad → option A, eqm A f f.

  Lemma eqm_trans :
   ∀ f f' f":ad → option A, eqm A f f' → eqm A f' f" → eqm A f f".

  Definition eqmap (m m':Map A) := eqm A (MapGet A m) (MapGet A m').

  Lemma eqmap_sym : ∀ m m':Map A, eqmap m m' → eqmap m' m.

  Lemma eqmap_refl : ∀ m:Map A, eqmap m m.

  Lemma eqmap_trans :
   ∀ m m' m":Map A, eqmap m m' → eqmap m' m" → eqmap m m".

  Lemma MapPut_as_Merge :
   ∀ (m:Map A) (a:ad) (y:A),
     eqmap (MapPut A m a y) (MapMerge A m (M1 A a y)).

  Lemma MapPut_ext :
   ∀ m m':Map A,
     eqmap m m' →
     ∀ (a:ad) (y:A), eqmap (MapPut A m a y) (MapPut A m' a y).

  Lemma MapPut_behind_as_Merge :
   ∀ (m:Map A) (a:ad) (y:A),
```

$eqmap\ (MapPut\_behind\ A\ m\ a\ y)\ (MapMerge\ A\ (M1\ A\ a\ y)\ m)$.

Lemma $MapPut\_behind\_ext$:
$\forall\ m\ m'{:}Map\ A,$
$eqmap\ m\ m' \rightarrow$
$\forall\ (a{:}ad)\ (y{:}A),$
$eqmap\ (MapPut\_behind\ A\ m\ a\ y)\ (MapPut\_behind\ A\ m'\ a\ y)$.

Lemma $MapMerge\_empty\_m\_1 : \forall\ m{:}Map\ A,\ MapMerge\ A\ (M0\ A)\ m = m$.

Lemma $MapMerge\_empty\_m : \forall\ m{:}Map\ A,\ eqmap\ (MapMerge\ A\ (M0\ A)\ m)\ m$.

Lemma $MapMerge\_m\_empty\_1 : \forall\ m{:}Map\ A,\ MapMerge\ A\ m\ (M0\ A) = m$.

Lemma $MapMerge\_m\_empty : \forall\ m{:}Map\ A,\ eqmap\ (MapMerge\ A\ m\ (M0\ A))\ m$.

Lemma $MapMerge\_empty\_l$:
$\forall\ m\ m'{:}Map\ A,\ eqmap\ (MapMerge\ A\ m\ m')\ (M0\ A) \rightarrow eqmap\ m\ (M0\ A)$.

Lemma $MapMerge\_empty\_r$:
$\forall\ m\ m'{:}Map\ A,\ eqmap\ (MapMerge\ A\ m\ m')\ (M0\ A) \rightarrow eqmap\ m'\ (M0\ A)$.

Lemma $MapMerge\_assoc$:
$\forall\ m\ m'\ m''{:}Map\ A,$
$eqmap\ (MapMerge\ A\ (MapMerge\ A\ m\ m')\ m'')$
$(MapMerge\ A\ m\ (MapMerge\ A\ m'\ m''))$.

Lemma $MapMerge\_idempotent : \forall\ m{:}Map\ A,\ eqmap\ (MapMerge\ A\ m\ m)\ m$.

Lemma $MapMerge\_ext$:
$\forall\ m1\ m2\ m'1\ m'2{:}Map\ A,$
$eqmap\ m1\ m'1 \rightarrow$
$eqmap\ m2\ m'2 \rightarrow eqmap\ (MapMerge\ A\ m1\ m2)\ (MapMerge\ A\ m'1\ m'2)$.

Lemma $MapMerge\_ext\_l$:
$\forall\ m1\ m'1\ m2{:}Map\ A,$
$eqmap\ m1\ m'1 \rightarrow eqmap\ (MapMerge\ A\ m1\ m2)\ (MapMerge\ A\ m'1\ m2)$.

Lemma $MapMerge\_ext\_r$:
$\forall\ m1\ m2\ m'2{:}Map\ A,$
$eqmap\ m2\ m'2 \rightarrow eqmap\ (MapMerge\ A\ m1\ m2)\ (MapMerge\ A\ m1\ m'2)$.

Lemma $MapMerge\_RestrTo\_l$:
$\forall\ m\ m'\ m''{:}Map\ A,$
$eqmap\ (MapMerge\ A\ (MapDomRestrTo\ A\ A\ m\ m')\ m'')$
$(MapDomRestrTo\ A\ A\ (MapMerge\ A\ m\ m'')\ (MapMerge\ A\ m'\ m''))$.

Lemma $MapRemove\_as\_RestrBy$:
$\forall\ (m{:}Map\ A)\ (a{:}ad)\ (y{:}B),$
$eqmap\ (MapRemove\ A\ m\ a)\ (MapDomRestrBy\ A\ B\ m\ (M1\ B\ a\ y))$.

Lemma $MapRemove\_ext$:
$\forall\ m\ m'{:}Map\ A,$
$eqmap\ m\ m' \rightarrow \forall\ a{:}ad,\ eqmap\ (MapRemove\ A\ m\ a)\ (MapRemove\ A\ m'\ a)$.

Lemma *MapDomRestrTo_empty_m_1* :
$\forall$ *m*:*Map B*, *MapDomRestrTo A B* (*M0 A*) *m* = *M0 A*.

Lemma *MapDomRestrTo_empty_m* :
$\forall$ *m*:*Map B*, *eqmap* (*MapDomRestrTo A B* (*M0 A*) *m*) (*M0 A*).

Lemma *MapDomRestrTo_m_empty_1* :
$\forall$ *m*:*Map A*, *MapDomRestrTo A B m* (*M0 B*) = *M0 A*.

Lemma *MapDomRestrTo_m_empty* :
$\forall$ *m*:*Map A*, *eqmap* (*MapDomRestrTo A B m* (*M0 B*)) (*M0 A*).

Lemma *MapDomRestrTo_assoc* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrTo A C* (*MapDomRestrTo A B m m'*) *m"*)
(*MapDomRestrTo A B m* (*MapDomRestrTo B C m' m"*)).

Lemma *MapDomRestrTo_idempotent* :
$\forall$ *m*:*Map A*, *eqmap* (*MapDomRestrTo A A m m*) *m*.

Lemma *MapDomRestrTo_Dom* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
eqmap (*MapDomRestrTo A B m m'*) (*MapDomRestrTo A unit m* (*MapDom B m'*)).

Lemma *MapDomRestrBy_empty_m_1* :
$\forall$ *m*:*Map B*, *MapDomRestrBy A B* (*M0 A*) *m* = *M0 A*.

Lemma *MapDomRestrBy_empty_m* :
$\forall$ *m*:*Map B*, *eqmap* (*MapDomRestrBy A B* (*M0 A*) *m*) (*M0 A*).

Lemma *MapDomRestrBy_m_empty_1* :
$\forall$ *m*:*Map A*, *MapDomRestrBy A B m* (*M0 B*) = *m*.

Lemma *MapDomRestrBy_m_empty* :
$\forall$ *m*:*Map A*, *eqmap* (*MapDomRestrBy A B m* (*M0 B*)) *m*.

Lemma *MapDomRestrBy_Dom* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
eqmap (*MapDomRestrBy A B m m'*) (*MapDomRestrBy A unit m* (*MapDom B m'*)).

Lemma *MapDomRestrBy_m_m_1* :
$\forall$ *m*:*Map A*, *eqmap* (*MapDomRestrBy A A m m*) (*M0 A*).

Lemma *MapDomRestrBy_By* :
$\forall$ (*m*:*Map A*) (*m' m"*:*Map B*),
eqmap (*MapDomRestrBy A B* (*MapDomRestrBy A B m m'*) *m"*)
(*MapDomRestrBy A B m* (*MapMerge B m' m"*)).

Lemma *MapDomRestrBy_By_comm* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrBy A C* (*MapDomRestrBy A B m m'*) *m"*)
(*MapDomRestrBy A B* (*MapDomRestrBy A C m m"*) *m'*).

Lemma *MapDomRestrBy_To* :

$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrBy A C* (*MapDomRestrTo A B m m'*) *m"*)
(*MapDomRestrTo A B m* (*MapDomRestrBy B C m' m"*)).

Lemma *MapDomRestrBy_To_comm* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrBy A C* (*MapDomRestrTo A B m m'*) *m"*)
(*MapDomRestrTo A B* (*MapDomRestrBy A C m m"*) *m'*).

Lemma *MapDomRestrTo_By* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrTo A C* (*MapDomRestrBy A B m m'*) *m"*)
(*MapDomRestrTo A C m* (*MapDomRestrBy C B m" m'*)).

Lemma *MapDomRestrTo_By_comm* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrTo A C* (*MapDomRestrBy A B m m'*) *m"*)
(*MapDomRestrBy A B* (*MapDomRestrTo A C m m"*) *m'*).

Lemma *MapDomRestrTo_To_comm* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
eqmap (*MapDomRestrTo A C* (*MapDomRestrTo A B m m'*) *m"*)
(*MapDomRestrTo A B* (*MapDomRestrTo A C m m"*) *m'*).

Lemma *MapMerge_DomRestrTo* :
$\forall$ (*m m'*:*Map A*) (*m"*:*Map B*),
eqmap (*MapDomRestrTo A B* (*MapMerge A m m'*) *m"*)
(*MapMerge A* (*MapDomRestrTo A B m m"*) (*MapDomRestrTo A B m' m"*)).

Lemma *MapMerge_DomRestrBy* :
$\forall$ (*m m'*:*Map A*) (*m"*:*Map B*),
eqmap (*MapDomRestrBy A B* (*MapMerge A m m'*) *m"*)
(*MapMerge A* (*MapDomRestrBy A B m m"*) (*MapDomRestrBy A B m' m"*)).

Lemma *MapDelta_empty_m_1* : $\forall$ *m*:*Map A*, *MapDelta A* (*M0 A*) *m* = *m*.

Lemma *MapDelta_empty_m* : $\forall$ *m*:*Map A*, *eqmap* (*MapDelta A* (*M0 A*) *m*) *m*.

Lemma *MapDelta_m_empty_1* : $\forall$ *m*:*Map A*, *MapDelta A m* (*M0 A*) = *m*.

Lemma *MapDelta_m_empty* : $\forall$ *m*:*Map A*, *eqmap* (*MapDelta A m* (*M0 A*)) *m*.

Lemma *MapDelta_nilpotent* : $\forall$ *m*:*Map A*, *eqmap* (*MapDelta A m m*) (*M0 A*).

Lemma *MapDelta_as_Merge* :
$\forall$ *m m'*:*Map A*,
eqmap (*MapDelta A m m'*)
(*MapMerge A* (*MapDomRestrBy A A m m'*) (*MapDomRestrBy A A m' m*)).

Lemma *MapDelta_as_DomRestrBy* :
$\forall$ *m m'*:*Map A*,
eqmap (*MapDelta A m m'*)
(*MapDomRestrBy A A* (*MapMerge A m m'*) (*MapDomRestrTo A A m m'*)).

Lemma *MapDelta_as_DomRestrBy_2* :
∀ *m m'*:*Map A*,
eqmap (*MapDelta A m m'*)
(*MapDomRestrBy A A* (*MapMerge A m m'*) (*MapDomRestrTo A A m' m*)).

Lemma *MapDelta_sym* :
∀ *m m'*:*Map A*, *eqmap* (*MapDelta A m m'*) (*MapDelta A m' m*).

Lemma *MapDelta_ext* :
∀ *m1 m2 m'1 m'2*:*Map A*,
eqmap m1 m'1 →
eqmap m2 m'2 → *eqmap* (*MapDelta A m1 m2*) (*MapDelta A m'1 m'2*).

Lemma *MapDelta_ext_l* :
∀ *m1 m'1 m2*:*Map A*,
eqmap m1 m'1 → *eqmap* (*MapDelta A m1 m2*) (*MapDelta A m'1 m2*).

Lemma *MapDelta_ext_r* :
∀ *m1 m2 m'2*:*Map A*,
eqmap m2 m'2 → *eqmap* (*MapDelta A m1 m2*) (*MapDelta A m1 m'2*).

Lemma *MapDom_Split_1* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
eqmap m (*MapMerge A* (*MapDomRestrTo A B m m'*) (*MapDomRestrBy A B m m'*)).

Lemma *MapDom_Split_2* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
eqmap m (*MapMerge A* (*MapDomRestrBy A B m m'*) (*MapDomRestrTo A B m m'*)).

Lemma *MapDom_Split_3* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
eqmap
(*MapDomRestrTo A A* (*MapDomRestrTo A B m m'*) (*MapDomRestrBy A B m m'*))
(*M0 A*).

End *MapAxioms*.

Lemma *MapDomRestrTo_ext* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2*:*Map B*) (*m'1*:*Map A*)
(*m'2*:*Map B*),
eqmap A m1 m'1 →
eqmap B m2 m'2 →
eqmap A (*MapDomRestrTo A B m1 m2*) (*MapDomRestrTo A B m'1 m'2*).

Lemma *MapDomRestrTo_ext_l* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2*:*Map B*) (*m'1*:*Map A*),
eqmap A m1 m'1 →
eqmap A (*MapDomRestrTo A B m1 m2*) (*MapDomRestrTo A B m'1 m2*).

Lemma *MapDomRestrTo_ext_r* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2 m'2*:*Map B*),
eqmap B m2 m'2 →

eqmap A (MapDomRestrTo A B m1 m2) (MapDomRestrTo A B m1 m'2).

Lemma *MapDomRestrBy_ext* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2*:*Map B*) (*m'1*:*Map A*)
(*m'2*:*Map B*),
eqmap A m1 m'1 →
eqmap B m2 m'2 →
eqmap A (MapDomRestrBy A B m1 m2) (MapDomRestrBy A B m'1 m'2).

Lemma *MapDomRestrBy_ext_l* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2*:*Map B*) (*m'1*:*Map A*),
eqmap A m1 m'1 →
eqmap A (MapDomRestrBy A B m1 m2) (MapDomRestrBy A B m'1 m2).

Lemma *MapDomRestrBy_ext_r* :
∀ (*A B*:Set) (*m1*:*Map A*) (*m2 m'2*:*Map B*),
eqmap B m2 m'2 →
eqmap A (MapDomRestrBy A B m1 m2) (MapDomRestrBy A B m1 m'2).

Lemma *MapDomRestrBy_m_m* :
∀ (*A*:Set) (*m*:*Map A*),
eqmap A (MapDomRestrBy A unit m (MapDom A m)) (M0 A).

Lemma *FSetDelta_assoc* :
∀ *s s' s"*:*FSet*,
eqmap unit (MapDelta _ (MapDelta _ s s') s")
(MapDelta _ s (MapDelta _ s' s")).

Lemma *FSet_ext* :
∀ *s s'*:*FSet*,
(∀ *a*:*ad*, *in_FSet a s* = *in_FSet a s'*) → *eqmap unit s s'.*

Lemma *FSetUnion_comm* :
∀ *s s'*:*FSet*, *eqmap unit (FSetUnion s s') (FSetUnion s' s).*

Lemma *FSetUnion_assoc* :
∀ *s s' s"*:*FSet*,
eqmap unit (FSetUnion (FSetUnion s s') s")
(FSetUnion s (FSetUnion s' s")).

Lemma *FSetUnion_M0_s* : ∀ *s*:*FSet*, *eqmap unit (FSetUnion (M0 unit) s) s.*

Lemma *FSetUnion_s_M0* : ∀ *s*:*FSet*, *eqmap unit (FSetUnion s (M0 unit)) s.*

Lemma *FSetUnion_idempotent* : ∀ *s*:*FSet*, *eqmap unit (FSetUnion s s) s.*

Lemma *FSetInter_comm* :
∀ *s s'*:*FSet*, *eqmap unit (FSetInter s s') (FSetInter s' s).*

Lemma *FSetInter_assoc* :
∀ *s s' s"*:*FSet*,
eqmap unit (FSetInter (FSetInter s s') s")
(FSetInter s (FSetInter s' s")).

Lemma *FSetInter_M0_s* :
∀ *s*:*FSet*, *eqmap unit* (*FSetInter* (*M0 unit*) *s*) (*M0 unit*).

Lemma *FSetInter_s_M0* :
∀ *s*:*FSet*, *eqmap unit* (*FSetInter s* (*M0 unit*)) (*M0 unit*).

Lemma *FSetInter_idempotent* : ∀ *s*:*FSet*, *eqmap unit* (*FSetInter s s*) *s*.

Lemma *FSetUnion_Inter_l* :
∀ *s s' s"*:*FSet*,
eqmap unit (*FSetUnion* (*FSetInter s s'*) *s"*)
(*FSetInter* (*FSetUnion s s"*) (*FSetUnion s' s"*)).

Lemma *FSetUnion_Inter_r* :
∀ *s s' s"*:*FSet*,
eqmap unit (*FSetUnion s* (*FSetInter s' s"*))
(*FSetInter* (*FSetUnion s s'*) (*FSetUnion s s"*)).

Lemma *FSetInter_Union_l* :
∀ *s s' s"*:*FSet*,
eqmap unit (*FSetInter* (*FSetUnion s s'*) *s"*)
(*FSetUnion* (*FSetInter s s"*) (*FSetInter s' s"*)).

Lemma *FSetInter_Union_r* :
∀ *s s' s"*:*FSet*,
eqmap unit (*FSetInter s* (*FSetUnion s' s"*))
(*FSetUnion* (*FSetInter s s'*) (*FSetInter s s"*)).

Chapter 197

Module Coq.IntMap.Mapcanon

```
Require Import Bool.
Require Import Sumbool.
Require Import Arith.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import Mapaxioms.
Require Import Mapiter.
Require Import Fset.
Require Import List.
Require Import Lsort.
Require Import Mapsubset.
Require Import Mapcard.

Section MapCanon.

  Variable A : Set.

  Inductive mapcanon : Map A → Prop :=
    | M0_canon : mapcanon (M0 A)
    | M1_canon : ∀ (a:ad) (y:A), mapcanon (M1 A a y)
    | M2_canon :
        ∀ m1 m2:Map A,
          mapcanon m1 →
          mapcanon m2 → 2 ≤ MapCard A (M2 A m1 m2) → mapcanon (M2 A m1 m2).

  Lemma mapcanon_M2 :
   ∀ m1 m2:Map A, mapcanon (M2 A m1 m2) → 2 ≤ MapCard A (M2 A m1 m2).

  Lemma mapcanon_M2_1 :
   ∀ m1 m2:Map A, mapcanon (M2 A m1 m2) → mapcanon m1.

  Lemma mapcanon_M2_2 :
   ∀ m1 m2:Map A, mapcanon (M2 A m1 m2) → mapcanon m2.

  Lemma M2_eqmap_1 :
```

$\forall$ *m0 m1 m2 m3*:*Map A*,
eqmap A (*M2 A m0 m1*) (*M2 A m2 m3*) → *eqmap A m0 m2*.

Lemma *M2_eqmap_2* :
$\forall$ *m0 m1 m2 m3*:*Map A*,
eqmap A (*M2 A m0 m1*) (*M2 A m2 m3*) → *eqmap A m1 m3*.

Lemma *mapcanon_unique* :
$\forall$ *m m'*:*Map A*, *mapcanon m* → *mapcanon m'* → *eqmap A m m'* → *m* = *m'*.

Lemma *MapPut1_canon* :
$\forall$ (*p*:*positive*) (*a a'*:*ad*) (*y y'*:*A*), *mapcanon* (*MapPut1 A a y a' y' p*).

Lemma *MapPut_canon* :
$\forall$ *m*:*Map A*,
mapcanon m → $\forall$ (*a*:*ad*) (*y*:*A*), *mapcanon* (*MapPut A m a y*).

Lemma *MapPut_behind_canon* :
$\forall$ *m*:*Map A*,
mapcanon m → $\forall$ (*a*:*ad*) (*y*:*A*), *mapcanon* (*MapPut_behind A m a y*).

Lemma *makeM2_canon* :
$\forall$ *m m'*:*Map A*, *mapcanon m* → *mapcanon m'* → *mapcanon* (*makeM2 A m m'*).

Fixpoint *MapCanonicalize* (*m*:*Map A*) : *Map A* :=
match *m* with
| *M2 m0 m1* ⇒ *makeM2 A* (*MapCanonicalize m0*) (*MapCanonicalize m1*)
| _ ⇒ *m*
end.

Lemma *mapcanon_exists_1* : $\forall$ *m*:*Map A*, *eqmap A m* (*MapCanonicalize m*).

Lemma *mapcanon_exists_2* : $\forall$ *m*:*Map A*, *mapcanon* (*MapCanonicalize m*).

Lemma *mapcanon_exists* :
$\forall$ *m*:*Map A*, {*m'* : *Map A* | *eqmap A m m'* ∧ *mapcanon m'*}.

Lemma *MapRemove_canon* :
$\forall$ *m*:*Map A*, *mapcanon m* → $\forall$ *a*:*ad*, *mapcanon* (*MapRemove A m a*).

Lemma *MapMerge_canon* :
$\forall$ *m m'*:*Map A*, *mapcanon m* → *mapcanon m'* → *mapcanon* (*MapMerge A m m'*).

Lemma *MapDelta_canon* :
$\forall$ *m m'*:*Map A*, *mapcanon m* → *mapcanon m'* → *mapcanon* (*MapDelta A m m'*).

Variable *B* : Set.

Lemma *MapDomRestrTo_canon* :
$\forall$ *m*:*Map A*,
mapcanon m → $\forall$ *m'*:*Map B*, *mapcanon* (*MapDomRestrTo A B m m'*).

Lemma *MapDomRestrBy_canon* :
$\forall$ *m*:*Map A*,
mapcanon m → $\forall$ *m'*:*Map B*, *mapcanon* (*MapDomRestrBy A B m m'*).

Lemma *Map_of_alist_canon* : ∀ *l*:*alist A*, *mapcanon* (*Map_of_alist A l*).

Lemma *MapSubset_c_1* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
mapcanon m → *MapSubset A B m m'* → *MapDomRestrBy A B m m'* = *M0 A*.

Lemma *MapSubset_c_2* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
MapDomRestrBy A B m m' = *M0 A* → *MapSubset A B m m'*.

End *MapCanon*.

Section *FSetCanon*.

Variable *A* : Set.

Lemma *MapDom_canon* :
∀ *m*:*Map A*, *mapcanon A m* → *mapcanon unit* (*MapDom A m*).

End *FSetCanon*.

Section *MapFoldCanon*.

Variables *A B* : Set.

Lemma *MapFold_canon_1* :
∀ *m0*:*Map B*,
mapcanon B m0 →
∀ *op*:*Map B* → *Map B* → *Map B*,
(∀ *m1*:*Map B*,
mapcanon B m1 →
∀ *m2*:*Map B*, *mapcanon B m2* → *mapcanon B* (*op m1 m2*)) →
∀ *f*:*ad* → *A* → *Map B*,
(∀ (*a*:*ad*) (*y*:*A*), *mapcanon B* (*f a y*)) →
∀ (*m*:*Map A*) (*pf*:*ad* → *ad*),
mapcanon B (*MapFold1 A* (*Map B*) *m0 op f pf m*).

Lemma *MapFold_canon* :
∀ *m0*:*Map B*,
mapcanon B m0 →
∀ *op*:*Map B* → *Map B* → *Map B*,
(∀ *m1*:*Map B*,
mapcanon B m1 →
∀ *m2*:*Map B*, *mapcanon B m2* → *mapcanon B* (*op m1 m2*)) →
∀ *f*:*ad* → *A* → *Map B*,
(∀ (*a*:*ad*) (*y*:*A*), *mapcanon B* (*f a y*)) →
∀ *m*:*Map A*, *mapcanon B* (*MapFold A* (*Map B*) *m0 op f m*).

Lemma *MapCollect_canon* :
∀ *f*:*ad* → *A* → *Map B*,
(∀ (*a*:*ad*) (*y*:*A*), *mapcanon B* (*f a y*)) →
∀ *m*:*Map A*, *mapcanon B* (*MapCollect A B f m*).

End *MapFoldCanon*.

Chapter 198

Module Coq.IntMap.Mapcard

Require Import *Bool*.
Require Import *Sumbool*.
Require Import *Arith*.
Require Import *NArith*.
Require Import *Ndigits*.
Require Import *Ndec*.
Require Import *Map*.
Require Import *Mapaxioms*.
Require Import *Mapiter*.
Require Import *Fset*.
Require Import *Mapsubset*.
Require Import *List*.
Require Import *Lsort*.
Require Import *Peano_dec*.

Section *MapCard*.

Variables A B : Set.

Lemma *MapCard_M0* : $MapCard\ A\ (M0\ A) = 0$.

Lemma *MapCard_M1* : $\forall\ (a{:}ad)\ (y{:}A),\ MapCard\ A\ (M1\ A\ a\ y) = 1$.

Lemma *MapCard_is_O* :
$\forall\ m{:}Map\ A,\ MapCard\ A\ m = 0 \rightarrow \forall\ a{:}ad,\ MapGet\ A\ m\ a = None$.

Lemma *MapCard_is_not_O* :
$\forall\ (m{:}Map\ A)\ (a{:}ad)\ (y{:}A),$
$MapGet\ A\ m\ a = Some\ y \rightarrow \{n : nat \mid MapCard\ A\ m = S\ n\}$.

Lemma *MapCard_is_one* :
$\forall\ m{:}Map\ A,$
$MapCard\ A\ m = 1 \rightarrow \{a : ad\ \&\ \{y : A \mid MapGet\ A\ m\ a = Some\ y\}\}$.

Lemma *MapCard_is_one_unique* :
$\forall\ m{:}Map\ A,$
$MapCard\ A\ m = 1 \rightarrow$

$\forall$ (a a':ad) (y y':A),
$MapGet\ A\ m\ a = Some\ y \rightarrow$
$MapGet\ A\ m\ a' = Some\ y' \rightarrow a = a' \wedge y = y'$.

Lemma *length_as_fold* :
$\forall$ (C:Set) (l:$list\ C$),
$length\ l = fold\_right$ (fun (_:C) (n:nat) $\Rightarrow S\ n$) $0\ l$.

Lemma *length_as_fold_2* :
$\forall\ l$:$alist\ A$,
$length\ l =$
$fold\_right$ (fun (r:$ad \times A$) (n:nat) $\Rightarrow$ let (a, y) := r in $1 + n$) $0\ l$.

Lemma *MapCard_as_Fold_1* :
$\forall$ (m:$Map\ A$) (pf:$ad \rightarrow ad$),
$MapCard\ A\ m = MapFold1\ A\ nat\ 0\ plus$ (fun (_:ad) (_:A) $\Rightarrow$ 1) $pf\ m$.

Lemma *MapCard_as_Fold* :
$\forall\ m$:$Map\ A$,
$MapCard\ A\ m = MapFold\ A\ nat\ 0\ plus$ (fun (_:ad) (_:A) $\Rightarrow$ 1) m.

Lemma *MapCard_as_length* :
$\forall\ m$:$Map\ A$, $MapCard\ A\ m = length\ (alist\_of\_Map\ A\ m)$.

Lemma *MapCard_Put1_equals_2* :
$\forall$ (p:$positive$) (a a':ad) (y y':A),
$MapCard\ A\ (MapPut1\ A\ a\ y\ a'\ y'\ p) = 2$.

Lemma *MapCard_Put_sum* :
$\forall$ (m m':$Map\ A$) (a:ad) (y:A) (n n':nat),
$m' = MapPut\ A\ m\ a\ y \rightarrow$
$n = MapCard\ A\ m \rightarrow n' = MapCard\ A\ m' \rightarrow \{n' = n\} + \{n' = S\ n\}$.

Lemma *MapCard_Put_lb* :
$\forall$ (m:$Map\ A$) (a:ad) (y:A), $MapCard\ A\ (MapPut\ A\ m\ a\ y) \geq MapCard\ A\ m$.

Lemma *MapCard_Put_ub* :
$\forall$ (m:$Map\ A$) (a:ad) (y:A),
$MapCard\ A\ (MapPut\ A\ m\ a\ y) \leq S\ (MapCard\ A\ m)$.

Lemma *MapCard_Put_1* :
$\forall$ (m:$Map\ A$) (a:ad) (y:A),
$MapCard\ A\ (MapPut\ A\ m\ a\ y) = MapCard\ A\ m \rightarrow$
$\{y : A \mid MapGet\ A\ m\ a = Some\ y\}$.

Lemma *MapCard_Put_2* :
$\forall$ (m:$Map\ A$) (a:ad) (y:A),
$MapCard\ A\ (MapPut\ A\ m\ a\ y) = S\ (MapCard\ A\ m) \rightarrow MapGet\ A\ m\ a = None$.

Lemma *MapCard_Put_1_conv* :
$\forall$ (m:$Map\ A$) (a:ad) (y y':A),
$MapGet\ A\ m\ a = Some\ y \rightarrow MapCard\ A\ (MapPut\ A\ m\ a\ y') = MapCard\ A\ m$.

Lemma *MapCard_Put_2_conv* :
$\forall$ (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapGet A m a = *None* $\rightarrow$ *MapCard A* (*MapPut A m a y*) = *S* (*MapCard A m*).

Lemma *MapCard_ext* :
$\forall$ *m m'*:*Map A*,
eqm A (*MapGet A m*) (*MapGet A m'*) $\rightarrow$ *MapCard A m* = *MapCard A m'*.

Lemma *MapCard_Dom* : $\forall$ *m*:*Map A*, *MapCard A m* = *MapCard unit* (*MapDom A m*).

Lemma *MapCard_Dom_Put_behind* :
$\forall$ (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapDom A (*MapPut_behind A m a y*) = *MapDom A* (*MapPut A m a y*).

Lemma *MapCard_Put_behind_Put* :
$\forall$ (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapCard A (*MapPut_behind A m a y*) = *MapCard A* (*MapPut A m a y*).

Lemma *MapCard_Put_behind_sum* :
$\forall$ (*m m'*:*Map A*) (*a*:*ad*) (*y*:*A*) (*n n'*:*nat*),
m' = *MapPut_behind A m a y* $\rightarrow$
n = *MapCard A m* $\rightarrow$ *n'* = *MapCard A m'* $\rightarrow$ {*n'* = *n*} + {*n'* = *S n*}.

Lemma *MapCard_makeM2* :
$\forall$ *m m'*:*Map A*, *MapCard A* (*makeM2 A m m'*) = *MapCard A m* + *MapCard A m'*.

Lemma *MapCard_Remove_sum* :
$\forall$ (*m m'*:*Map A*) (*a*:*ad*) (*n n'*:*nat*),
m' = *MapRemove A m a* $\rightarrow$
n = *MapCard A m* $\rightarrow$ *n'* = *MapCard A m'* $\rightarrow$ {*n* = *n'*} + {*n* = *S n'*}.

Lemma *MapCard_Remove_ub* :
$\forall$ (*m*:*Map A*) (*a*:*ad*), *MapCard A* (*MapRemove A m a*) $\leq$ *MapCard A m*.

Lemma *MapCard_Remove_lb* :
$\forall$ (*m*:*Map A*) (*a*:*ad*), *S* (*MapCard A* (*MapRemove A m a*)) $\geq$ *MapCard A m*.

Lemma *MapCard_Remove_1* :
$\forall$ (*m*:*Map A*) (*a*:*ad*),
MapCard A (*MapRemove A m a*) = *MapCard A m* $\rightarrow$ *MapGet A m a* = *None*.

Lemma *MapCard_Remove_2* :
$\forall$ (*m*:*Map A*) (*a*:*ad*),
S (*MapCard A* (*MapRemove A m a*)) = *MapCard A m* $\rightarrow$
{*y* : *A* | *MapGet A m a* = *Some y*}.

Lemma *MapCard_Remove_1_conv* :
$\forall$ (*m*:*Map A*) (*a*:*ad*),
MapGet A m a = *None* $\rightarrow$ *MapCard A* (*MapRemove A m a*) = *MapCard A m*.

Lemma *MapCard_Remove_2_conv* :
$\forall$ (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapGet A m a = *Some y* $\rightarrow$ *S* (*MapCard A* (*MapRemove A m a*)) = *MapCard A m*.

Lemma *MapMerge_Restr_Card* :
∀ *m m'*:*Map A*,
MapCard A m + *MapCard A m'* =
MapCard A (*MapMerge A m m'*) + *MapCard A* (*MapDomRestrTo A A m m'*).

Lemma *MapMerge_disjoint_Card* :
∀ *m m'*:*Map A*,
MapDisjoint A A m m' →
MapCard A (*MapMerge A m m'*) = *MapCard A m* + *MapCard A m'*.

Lemma *MapSplit_Card* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
MapCard A m =
MapCard A (*MapDomRestrTo A B m m'*) + *MapCard A* (*MapDomRestrBy A B m m'*).

Lemma *MapMerge_Card_ub* :
∀ *m m'*:*Map A*,
MapCard A (*MapMerge A m m'*) ≤ *MapCard A m* + *MapCard A m'*.

Lemma *MapDomRestrTo_Card_ub_l* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
MapCard A (*MapDomRestrTo A B m m'*) ≤ *MapCard A m*.

Lemma *MapDomRestrBy_Card_ub_l* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
MapCard A (*MapDomRestrBy A B m m'*) ≤ *MapCard A m*.

Lemma *MapMerge_Card_disjoint* :
∀ *m m'*:*Map A*,
MapCard A (*MapMerge A m m'*) = *MapCard A m* + *MapCard A m'* →
MapDisjoint A A m m'.

Lemma *MapCard_is_Sn* :
∀ (*m*:*Map A*) (*n*:*nat*),
MapCard _ m = *S n* → {*a* : *ad* | *in_dom _ a m* = *true*}.

End *MapCard*.

Section *MapCard2*.

Variables *A B* : Set.

Lemma *MapSubset_card_eq_1* :
∀ (*n*:*nat*) (*m*:*Map A*) (*m'*:*Map B*),
MapSubset _ _ m m' →
MapCard _ m = *n* → *MapCard _ m'* = *n* → *MapSubset _ _ m' m*.

Lemma *MapDomRestrTo_Card_ub_r* :
∀ (*m*:*Map A*) (*m'*:*Map B*),
MapCard A (*MapDomRestrTo A B m m'*) ≤ *MapCard B m'*.

End *MapCard2*.

Section *MapCard3*.

Variables A B : Set.

Lemma *MapMerge_Card_lb_l* :
$\forall\ m\ m'$:*Map* A, *MapCard* A (*MapMerge* $A\ m\ m'$) $\geq$ *MapCard* $A\ m$.

Lemma *MapMerge_Card_lb_r* :
$\forall\ m\ m'$:*Map* A, *MapCard* A (*MapMerge* $A\ m\ m'$) $\geq$ *MapCard* $A\ m'$.

Lemma *MapDomRestrBy_Card_lb* :
$\forall\ (m$:*Map* $A)\ (m'$:*Map* $B)$,
MapCard $B\ m' +$ *MapCard* A (*MapDomRestrBy* $A\ B\ m\ m'$) $\geq$ *MapCard* $A\ m$.

Lemma *MapSubset_Card_le* :
$\forall\ (m$:*Map* $A)\ (m'$:*Map* $B)$,
MapSubset $A\ B\ m\ m' \rightarrow$ *MapCard* $A\ m \leq$ *MapCard* $B\ m'$.

Lemma *MapSubset_card_eq* :
$\forall\ (m$:*Map* $A)\ (m'$:*Map* $B)$,
MapSubset $\_\ \_\ m\ m' \rightarrow$
MapCard $\_\ m' \leq$ *MapCard* $\_\ m \rightarrow$ *eqmap* $\_$ (*MapDom* $\_\ m$) (*MapDom* $\_\ m'$).

End *MapCard3*.

Chapter 199

Module Coq.IntMap.Mapc

```
Require Import Bool.
Require Import Sumbool.
Require Import Arith.
Require Import NArith.
Require Import Map.
Require Import Mapaxioms.
Require Import Fset.
Require Import Mapiter.
Require Import Mapsubset.
Require Import List.
Require Import Lsort.
Require Import Mapcard.
Require Import Mapcanon.

Section MapC.

  Variables A B C : Set.

  Lemma MapPut_as_Merge_c :
   ∀ m:Map A,
     mapcanon A m →
     ∀ (a:ad) (y:A), MapPut A m a y = MapMerge A m (M1 A a y).

  Lemma MapPut_behind_as_Merge_c :
   ∀ m:Map A,
     mapcanon A m →
     ∀ (a:ad) (y:A), MapPut_behind A m a y = MapMerge A (M1 A a y) m.

  Lemma MapMerge_empty_m_c : ∀ m:Map A, MapMerge A (M0 A) m = m.

  Lemma MapMerge_assoc_c :
   ∀ m m' m'':Map A,
     mapcanon A m →
     mapcanon A m' →
     mapcanon A m'' →
     MapMerge A (MapMerge A m m') m'' = MapMerge A m (MapMerge A m' m'').
```

Lemma *MapMerge_idempotent_c* :
$\forall$ *m*:*Map A*, *mapcanon A m* $\rightarrow$ *MapMerge A m m* = *m*.

Lemma *MapMerge_RestrTo_l_c* :
$\forall$ *m m' m"*:*Map A*,
mapcanon A m $\rightarrow$
mapcanon A m" $\rightarrow$
MapMerge A (*MapDomRestrTo A A m m'*) *m"* =
MapDomRestrTo A A (*MapMerge A m m"*) (*MapMerge A m' m"*).

Lemma *MapRemove_as_RestrBy_c* :
$\forall$ *m*:*Map A*,
mapcanon A m $\rightarrow$
$\forall$ (*a*:*ad*) (*y*:*B*), *MapRemove A m a* = *MapDomRestrBy A B m* (*M1 B a y*).

Lemma *MapDomRestrTo_assoc_c* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m $\rightarrow$
MapDomRestrTo A C (*MapDomRestrTo A B m m'*) *m"* =
MapDomRestrTo A B m (*MapDomRestrTo B C m' m"*).

Lemma *MapDomRestrTo_idempotent_c* :
$\forall$ *m*:*Map A*, *mapcanon A m* $\rightarrow$ *MapDomRestrTo A A m m* = *m*.

Lemma *MapDomRestrTo_Dom_c* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
mapcanon A m $\rightarrow$
MapDomRestrTo A B m m' = *MapDomRestrTo A unit m* (*MapDom B m'*).

Lemma *MapDomRestrBy_Dom_c* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
mapcanon A m $\rightarrow$
MapDomRestrBy A B m m' = *MapDomRestrBy A unit m* (*MapDom B m'*).

Lemma *MapDomRestrBy_By_c* :
$\forall$ (*m*:*Map A*) (*m' m"*:*Map B*),
mapcanon A m $\rightarrow$
MapDomRestrBy A B (*MapDomRestrBy A B m m'*) *m"* =
MapDomRestrBy A B m (*MapMerge B m' m"*).

Lemma *MapDomRestrBy_By_comm_c* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m $\rightarrow$
MapDomRestrBy A C (*MapDomRestrBy A B m m'*) *m"* =
MapDomRestrBy A B (*MapDomRestrBy A C m m"*) *m'*.

Lemma *MapDomRestrBy_To_c* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m $\rightarrow$
MapDomRestrBy A C (*MapDomRestrTo A B m m'*) *m"* =
MapDomRestrTo A B m (*MapDomRestrBy B C m' m"*).

Lemma *MapDomRestrBy_To_comm_c* :
∀ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m →
MapDomRestrBy A C (*MapDomRestrTo A B m m'*) *m"* =
MapDomRestrTo A B (*MapDomRestrBy A C m m"*) *m'*.

Lemma *MapDomRestrTo_By_c* :
∀ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m →
MapDomRestrTo A C (*MapDomRestrBy A B m m'*) *m"* =
MapDomRestrTo A C m (*MapDomRestrBy C B m" m'*).

Lemma *MapDomRestrTo_By_comm_c* :
∀ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m →
MapDomRestrTo A C (*MapDomRestrBy A B m m'*) *m"* =
MapDomRestrBy A B (*MapDomRestrTo A C m m"*) *m'*.

Lemma *MapDomRestrTo_To_comm_c* :
∀ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
mapcanon A m →
MapDomRestrTo A C (*MapDomRestrTo A B m m'*) *m"* =
MapDomRestrTo A B (*MapDomRestrTo A C m m"*) *m'*.

Lemma *MapMerge_DomRestrTo_c* :
∀ (*m m'*:*Map A*) (*m"*:*Map B*),
mapcanon A m →
mapcanon A m' →
MapDomRestrTo A B (*MapMerge A m m'*) *m"* =
MapMerge A (*MapDomRestrTo A B m m"*) (*MapDomRestrTo A B m' m"*).

Lemma *MapMerge_DomRestrBy_c* :
∀ (*m m'*:*Map A*) (*m"*:*Map B*),
mapcanon A m →
mapcanon A m' →
MapDomRestrBy A B (*MapMerge A m m'*) *m"* =
MapMerge A (*MapDomRestrBy A B m m"*) (*MapDomRestrBy A B m' m"*).

Lemma *MapDelta_nilpotent_c* :
∀ *m*:*Map A*, *mapcanon A m* → *MapDelta A m m* = *M0 A*.

Lemma *MapDelta_as_Merge_c* :
∀ *m m'*:*Map A*,
mapcanon A m →
mapcanon A m' →
MapDelta A m m' =
MapMerge A (*MapDomRestrBy A A m m'*) (*MapDomRestrBy A A m' m*).

Lemma *MapDelta_as_DomRestrBy_c* :
∀ *m m'*:*Map A*,

mapcanon A m →
mapcanon A m' →
MapDelta A m m' =
MapDomRestrBy A A (MapMerge A m m') (MapDomRestrTo A A m m').

Lemma *MapDelta_as_DomRestrBy_2_c :*
∀ m m':Map A,
mapcanon A m →
mapcanon A m' →
MapDelta A m m' =
MapDomRestrBy A A (MapMerge A m m') (MapDomRestrTo A A m' m).

Lemma *MapDelta_sym_c :*
∀ m m':Map A,
mapcanon A m → mapcanon A m' → MapDelta A m m' = MapDelta A m' m.

Lemma *MapDom_Split_1_c :*
∀ (m:Map A) (m':Map B),
mapcanon A m →
m = MapMerge A (MapDomRestrTo A B m m') (MapDomRestrBy A B m m').

Lemma *MapDom_Split_2_c :*
∀ (m:Map A) (m':Map B),
mapcanon A m →
m = MapMerge A (MapDomRestrBy A B m m') (MapDomRestrTo A B m m').

Lemma *MapDom_Split_3_c :*
∀ (m:Map A) (m':Map B),
mapcanon A m →
MapDomRestrTo A A (MapDomRestrTo A B m m') (MapDomRestrBy A B m m') =
M0 A.

Lemma *Map_of_alist_of_Map_c :*
∀ m:Map A, mapcanon A m → Map_of_alist A (alist_of_Map A m) = m.

Lemma *alist_of_Map_of_alist_c :*
∀ l:alist A,
alist_sorted_2 A l → alist_of_Map A (Map_of_alist A l) = l.

Lemma *MapSubset_antisym_c :*
∀ (m:Map A) (m':Map B),
mapcanon A m →
mapcanon B m' →
MapSubset A B m m' → MapSubset B A m' m → MapDom A m = MapDom B m'.

Lemma *FSubset_antisym_c :*
∀ s s':FSet,
mapcanon unit s →
mapcanon unit s' → MapSubset _ _ s s' → MapSubset _ _ s' s → s = s'.

Lemma *MapDisjoint_empty_c :*

$\forall\ m{:}Map\ A,\ mapcanon\ A\ m \to MapDisjoint\ A\ A\ m\ m \to m = M0\ A.$

Lemma $MapDelta\_disjoint\_c$:
$\forall\ m\ m'{:}Map\ A,$
$mapcanon\ A\ m \to$
$mapcanon\ A\ m' \to$
$MapDisjoint\ A\ A\ m\ m' \to MapDelta\ A\ m\ m' = MapMerge\ A\ m\ m'.$

End $MapC.$

Lemma $FSetDelta\_assoc\_c$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \to$
$mapcanon\ unit\ s' \to$
$mapcanon\ unit\ s'' \to$
$MapDelta\ \_\ (MapDelta\ \_\ s\ s')\ s'' = MapDelta\ \_\ s\ (MapDelta\ \_\ s'\ s'').$

Lemma $FSet\_ext\_c$:
$\forall\ s\ s'{:}FSet,$
$mapcanon\ unit\ s \to$
$mapcanon\ unit\ s' \to (\forall\ a{:}ad,\ in\_FSet\ a\ s = in\_FSet\ a\ s') \to s = s'.$

Lemma $FSetUnion\_comm\_c$:
$\forall\ s\ s'{:}FSet,$
$mapcanon\ unit\ s \to mapcanon\ unit\ s' \to FSetUnion\ s\ s' = FSetUnion\ s'\ s.$

Lemma $FSetUnion\_assoc\_c$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \to$
$mapcanon\ unit\ s' \to$
$mapcanon\ unit\ s'' \to$
$FSetUnion\ (FSetUnion\ s\ s')\ s'' = FSetUnion\ s\ (FSetUnion\ s'\ s'').$

Lemma $FSetUnion\_M0\_s\_c : \forall\ s{:}FSet,\ FSetUnion\ (M0\ unit)\ s = s.$

Lemma $FSetUnion\_s\_M0\_c : \forall\ s{:}FSet,\ FSetUnion\ s\ (M0\ unit) = s.$

Lemma $FSetUnion\_idempotent$:
$\forall\ s{:}FSet,\ mapcanon\ unit\ s \to FSetUnion\ s\ s = s.$

Lemma $FSetInter\_comm\_c$:
$\forall\ s\ s'{:}FSet,$
$mapcanon\ unit\ s \to mapcanon\ unit\ s' \to FSetInter\ s\ s' = FSetInter\ s'\ s.$

Lemma $FSetInter\_assoc\_c$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \to$
$FSetInter\ (FSetInter\ s\ s')\ s'' = FSetInter\ s\ (FSetInter\ s'\ s'').$

Lemma $FSetInter\_M0\_s\_c : \forall\ s{:}FSet,\ FSetInter\ (M0\ unit)\ s = M0\ unit.$

Lemma $FSetInter\_s\_M0\_c : \forall\ s{:}FSet,\ FSetInter\ s\ (M0\ unit) = M0\ unit.$

Lemma $FSetInter\_idempotent$:

$\forall\ s{:}FSet,\ mapcanon\ unit\ s \rightarrow FSetInter\ s\ s = s.$

Lemma $FSetUnion\_Inter\_l\_c$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \rightarrow$
$mapcanon\ unit\ s'' \rightarrow$
$FSetUnion\ (FSetInter\ s\ s')\ s'' =$
$FSetInter\ (FSetUnion\ s\ s'')\ (FSetUnion\ s'\ s'').$

Lemma $FSetUnion\_Inter\_r$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \rightarrow$
$mapcanon\ unit\ s' \rightarrow$
$FSetUnion\ s\ (FSetInter\ s'\ s'') =$
$FSetInter\ (FSetUnion\ s\ s')\ (FSetUnion\ s\ s'').$

Lemma $FSetInter\_Union\_l\_c$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \rightarrow$
$mapcanon\ unit\ s' \rightarrow$
$FSetInter\ (FSetUnion\ s\ s')\ s'' =$
$FSetUnion\ (FSetInter\ s\ s'')\ (FSetInter\ s'\ s'').$

Lemma $FSetInter\_Union\_r$:
$\forall\ s\ s'\ s''{:}FSet,$
$mapcanon\ unit\ s \rightarrow$
$mapcanon\ unit\ s' \rightarrow$
$FSetInter\ s\ (FSetUnion\ s'\ s'') =$
$FSetUnion\ (FSetInter\ s\ s')\ (FSetInter\ s\ s'').$

Chapter 200

Module Coq.IntMap.Mapfold

Require Import *Bool*.
Require Import *Sumbool*.
Require Import *NArith*.
Require Import *Ndigits*.
Require Import *Ndec*.
Require Import *Map*.
Require Import *Fset*.
Require Import *Mapaxioms*.
Require Import *Mapiter*.
Require Import *Lsort*.
Require Import *Mapsubset*.
Require Import *List*.

Section *MapFoldResults*.

Variable A : Set.

Variable M : Set.
Variable *neutral* : M.
Variable *op* : $M \to M \to M$.

Variable *nleft* : $\forall\ a{:}M,\ op\ neutral\ a = a$.
Variable *nright* : $\forall\ a{:}M,\ op\ a\ neutral = a$.
Variable *assoc* : $\forall\ a\ b\ c{:}M,\ op\ (op\ a\ b)\ c = op\ a\ (op\ b\ c)$.

Lemma *MapFold_ext* :
$\forall\ (f{:}ad \to A \to M)\ (m\ m'{:}Map\ A),$
$eqmap\ A\ m\ m' \to MapFold\ \_\ \_\ neutral\ op\ f\ m = MapFold\ \_\ \_\ neutral\ op\ f\ m'$.

Lemma *MapFold_ext_f_1* :
$\forall\ (m{:}Map\ A)\ (f\ g{:}ad \to A \to M)\ (pf{:}ad \to ad),$
$(\forall\ (a{:}ad)\ (y{:}A),\ MapGet\ \_\ m\ a = Some\ y \to f\ (pf\ a)\ y = g\ (pf\ a)\ y) \to$
$MapFold1\ \_\ \_\ neutral\ op\ f\ pf\ m = MapFold1\ \_\ \_\ neutral\ op\ g\ pf\ m$.

Lemma *MapFold_ext_f* :
$\forall\ (f\ g{:}ad \to A \to M)\ (m{:}Map\ A),$
$(\forall\ (a{:}ad)\ (y{:}A),\ MapGet\ \_\ m\ a = Some\ y \to f\ a\ y = g\ a\ y) \to$

MapFold _ _ neutral op f m = MapFold _ _ neutral op g m.

Lemma *MapFold1_as_Fold_1* :
∀ (*m*:*Map A*) (*f f'*:*ad* → *A* → *M*) (*pf pf'*:*ad* → *ad*),
(∀ (*a*:*ad*) (*y*:*A*), *f* (*pf a*) *y* = *f'* (*pf' a*) *y*) →
MapFold1 _ _ neutral op f pf m = MapFold1 _ _ neutral op f' pf' m.

Lemma *MapFold1_as_Fold* :
∀ (*f*:*ad* → *A* → *M*) (*pf*:*ad* → *ad*) (*m*:*Map A*),
MapFold1 _ _ neutral op f pf m =
MapFold _ _ neutral op (fun (*a*:*ad*) (*y*:*A*) ⇒ *f* (*pf a*) *y*) *m.*

Lemma *MapFold1_ext* :
∀ (*f*:*ad* → *A* → *M*) (*m m'*:*Map A*),
eqmap A m m' →
∀ *pf*:*ad* → *ad*,
MapFold1 _ _ neutral op f pf m = MapFold1 _ _ neutral op f pf m'.

Variable *comm* : ∀ *a b*:*M*, *op a b* = *op b a.*

Lemma *MapFold_Put_disjoint_1* :
∀ (*p*:*positive*) (*f*:*ad* → *A* → *M*) (*pf*:*ad* → *ad*)
(*a1 a2*:*ad*) (*y1 y2*:*A*),
Nxor a1 a2 = *Npos p* →
MapFold1 A M neutral op f pf (*MapPut1 A a1 y1 a2 y2 p*) =
op (*f* (*pf a1*) *y1*) (*f* (*pf a2*) *y2*).

Lemma *MapFold_Put_disjoint_2* :
∀ (*f*:*ad* → *A* → *M*) (*m*:*Map A*) (*a*:*ad*) (*y*:*A*) (*pf*:*ad* → *ad*),
MapGet A m a = *None* →
MapFold1 A M neutral op f pf (*MapPut A m a y*) =
op (*f* (*pf a*) *y*) (*MapFold1 A M neutral op f pf m*).

Lemma *MapFold_Put_disjoint* :
∀ (*f*:*ad* → *A* → *M*) (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapGet A m a = *None* →
MapFold A M neutral op f (*MapPut A m a y*) =
op (*f a y*) (*MapFold A M neutral op f m*).

Lemma *MapFold_Put_behind_disjoint_2* :
∀ (*f*:*ad* → *A* → *M*) (*m*:*Map A*) (*a*:*ad*) (*y*:*A*) (*pf*:*ad* → *ad*),
MapGet A m a = *None* →
MapFold1 A M neutral op f pf (*MapPut_behind A m a y*) =
op (*f* (*pf a*) *y*) (*MapFold1 A M neutral op f pf m*).

Lemma *MapFold_Put_behind_disjoint* :
∀ (*f*:*ad* → *A* → *M*) (*m*:*Map A*) (*a*:*ad*) (*y*:*A*),
MapGet A m a = *None* →
MapFold A M neutral op f (*MapPut_behind A m a y*) =
op (*f a y*) (*MapFold A M neutral op f m*).

Lemma *MapFold_Merge_disjoint_1* :

$\forall\ (f{:}ad \to A \to M)\ (m1\ m2{:}Map\ A)\ (pf{:}ad \to ad),$
$MapDisjoint\ A\ A\ m1\ m2 \to$
$MapFold1\ A\ M\ neutral\ op\ f\ pf\ (MapMerge\ A\ m1\ m2) =$
$op\ (MapFold1\ A\ M\ neutral\ op\ f\ pf\ m1)\ (MapFold1\ A\ M\ neutral\ op\ f\ pf\ m2).$

Lemma *MapFold_Merge_disjoint* :
$\forall\ (f{:}ad \to A \to M)\ (m1\ m2{:}Map\ A),$
$MapDisjoint\ A\ A\ m1\ m2 \to$
$MapFold\ A\ M\ neutral\ op\ f\ (MapMerge\ A\ m1\ m2) =$
$op\ (MapFold\ A\ M\ neutral\ op\ f\ m1)\ (MapFold\ A\ M\ neutral\ op\ f\ m2).$

End *MapFoldResults*.

Section *MapFoldDistr*.

Variable A : Set.

Variable M : Set.
Variable $neutral$: M.
Variable op : $M \to M \to M$.

Variable M' : Set.
Variable $neutral'$: M'.
Variable op' : $M' \to M' \to M'$.

Variable N : Set.

Variable $times$: $M \to N \to M'$.

Variable $absorb$: $\forall\ c{:}N,\ times\ neutral\ c = neutral'$.
Variable
$distr$:
$\forall\ (a\ b{:}M)\ (c{:}N),\ times\ (op\ a\ b)\ c = op'\ (times\ a\ c)\ (times\ b\ c).$

Lemma *MapFold_distr_r_1* :
$\forall\ (f{:}ad \to A \to M)\ (m{:}Map\ A)\ (c{:}N)\ (pf{:}ad \to ad),$
$times\ (MapFold1\ A\ M\ neutral\ op\ f\ pf\ m)\ c =$
$MapFold1\ A\ M'\ neutral'\ op'$ (fun $(a{:}ad)\ (y{:}A) \Rightarrow times\ (f\ a\ y)\ c)\ pf\ m.$

Lemma *MapFold_distr_r* :
$\forall\ (f{:}ad \to A \to M)\ (m{:}Map\ A)\ (c{:}N),$
$times\ (MapFold\ A\ M\ neutral\ op\ f\ m)\ c =$
$MapFold\ A\ M'\ neutral'\ op'$ (fun $(a{:}ad)\ (y{:}A) \Rightarrow times\ (f\ a\ y)\ c)\ m.$

End *MapFoldDistr*.

Section *MapFoldDistrL*.

Variable A : Set.

Variable M : Set.
Variable $neutral$: M.
Variable op : $M \to M \to M$.

Variable M' : Set.

```
  Variable neutral' : M'.
  Variable op' : M' → M' → M'.

  Variable N : Set.

  Variable times : N → M → M'.

  Variable absorb : ∀ c:N, times c neutral = neutral'.
  Variable
    distr :
      ∀ (a b:M) (c:N), times c (op a b) = op' (times c a) (times c b).

  Lemma MapFold_distr_l :
   ∀ (f:ad → A → M) (m:Map A) (c:N),
     times c (MapFold A M neutral op f m) =
     MapFold A M' neutral' op' (fun (a:ad) (y:A) ⇒ times c (f a y)) m.

End MapFoldDistrL.

Section MapFoldExists.

  Variable A : Set.

  Lemma MapFold_orb_1 :
   ∀ (f:ad → A → bool) (m:Map A) (pf:ad → ad),
     MapFold1 A bool false orb f pf m =
     match MapSweep1 A f pf m with
     | Some _ ⇒ true
     | _ ⇒ false
     end.

  Lemma MapFold_orb :
   ∀ (f:ad → A → bool) (m:Map A),
     MapFold A bool false orb f m =
     match MapSweep A f m with
     | Some _ ⇒ true
     | _ ⇒ false
     end.

End MapFoldExists.

Section DMergeDef.

  Variable A : Set.

  Definition DMerge :=
    MapFold (Map A) (Map A) (M0 A) (MapMerge A) (fun (_:ad) (m:Map A) ⇒ m).

  Lemma in_dom_DMerge_1 :
   ∀ (m:Map (Map A)) (a:ad),
     in_dom A a (DMerge m) =
     match MapSweep _ (fun (_:ad) (m0:Map A) ⇒ in_dom A a m0) m with
     | Some _ ⇒ true
     | _ ⇒ false
```

end.

Lemma *in_dom_DMerge_2* :
∀ (*m*:*Map* (*Map A*)) (*a*:*ad*),
in_dom A a (*DMerge m*) = *true* →
{*b* : *ad* &
{*m0* : *Map A* | *MapGet* _ *m b* = *Some m0* ∧ *in_dom A a m0* = *true*}}.

Lemma *in_dom_DMerge_3* :
∀ (*m*:*Map* (*Map A*)) (*a b*:*ad*) (*m0*:*Map A*),
MapGet _ *m a* = *Some m0* →
in_dom A b m0 = *true* → *in_dom A b* (*DMerge m*) = *true*.

End *DMergeDef*.

Chapter 201

Module Coq.IntMap.Mapiter

```
Require Import Bool.
Require Import Sumbool.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import Mapaxioms.
Require Import Fset.
Require Import List.

Section MapIter.

  Variable A : Set.

  Section MapSweepDef.

  Variable f : ad → A → bool.

  Definition MapSweep2 (a0:ad) (y:A) :=
    if f a0 y then Some (a0, y) else None.

  Fixpoint MapSweep1 (pf:ad → ad) (m:Map A) {struct m} :
   option (ad × A) :=
    match m with
    | M0 ⇒ None
    | M1 a y ⇒ MapSweep2 (pf a) y
    | M2 m m' ⇒
        match MapSweep1 (fun a:ad ⇒ pf (Ndouble a)) m with
        | Some r ⇒ Some r
        | None ⇒ MapSweep1 (fun a:ad ⇒ pf (Ndouble_plus_one a)) m'
        end
    end.

  Definition MapSweep (m:Map A) := MapSweep1 (fun a:ad ⇒ a) m.

  Lemma MapSweep_semantics_1_1 :
   ∀ (m:Map A) (pf:ad → ad) (a:ad) (y:A),
```

```
    MapSweep1 pf m = Some (a, y) → f a y = true.

Lemma MapSweep_semantics_1 :
 ∀ (m:Map A) (a:ad) (y:A), MapSweep m = Some (a, y) → f a y = true.

Lemma MapSweep_semantics_2_1 :
 ∀ (m:Map A) (pf:ad → ad) (a:ad) (y:A),
    MapSweep1 pf m = Some (a, y) → {a' : ad | a = pf a'}.

Lemma MapSweep_semantics_2_2 :
 ∀ (m:Map A) (pf fp:ad → ad),
    (∀ a0:ad, fp (pf a0) = a0) →
    ∀ (a:ad) (y:A),
       MapSweep1 pf m = Some (a, y) → MapGet A m (fp a) = Some y.

Lemma MapSweep_semantics_2 :
 ∀ (m:Map A) (a:ad) (y:A),
    MapSweep m = Some (a, y) → MapGet A m a = Some y.

Lemma MapSweep_semantics_3_1 :
 ∀ (m:Map A) (pf:ad → ad),
    MapSweep1 pf m = None →
    ∀ (a:ad) (y:A), MapGet A m a = Some y → f (pf a) y = false.

Lemma MapSweep_semantics_3 :
 ∀ m:Map A,
    MapSweep m = None →
    ∀ (a:ad) (y:A), MapGet A m a = Some y → f a y = false.

Lemma MapSweep_semantics_4_1 :
 ∀ (m:Map A) (pf:ad → ad) (a:ad) (y:A),
    MapGet A m a = Some y →
    f (pf a) y = true →
    {a' : ad & {y' : A | MapSweep1 pf m = Some (a', y')}}.


Lemma MapSweep_semantics_4 :
 ∀ (m:Map A) (a:ad) (y:A),
    MapGet A m a = Some y →
    f a y = true → {a' : ad & {y' : A | MapSweep m = Some (a', y')}}.

End MapSweepDef.

Variable B : Set.

Fixpoint MapCollect1 (f:ad → A → Map B) (pf:ad → ad)
 (m:Map A) {struct m} : Map B :=
  match m with
  | M0 ⇒ M0 B
  | M1 a y ⇒ f (pf a) y
  | M2 m1 m2 ⇒
       MapMerge B (MapCollect1 f (fun a0:ad ⇒ pf (Ndouble a0)) m1)
```

```
        (MapCollect1 f (fun a0:ad ⇒ pf (Ndouble_plus_one a0)) m2)
  end.

Definition MapCollect (f:ad → A → Map B) (m:Map A) :=
  MapCollect1 f (fun a:ad ⇒ a) m.

Section MapFoldDef.

  Variable M : Set.
  Variable neutral : M.
  Variable op : M → M → M.

  Fixpoint MapFold1 (f:ad → A → M) (pf:ad → ad)
   (m:Map A) {struct m} : M :=
    match m with
    | M0 ⇒ neutral
    | M1 a y ⇒ f (pf a) y
    | M2 m1 m2 ⇒
        op (MapFold1 f (fun a0:ad ⇒ pf (Ndouble a0)) m1)
          (MapFold1 f (fun a0:ad ⇒ pf (Ndouble_plus_one a0)) m2)
    end.

  Definition MapFold (f:ad → A → M) (m:Map A) :=
    MapFold1 f (fun a:ad ⇒ a) m.

  Lemma MapFold_empty : ∀ f:ad → A → M, MapFold f (M0 A) = neutral.

  Lemma MapFold_M1 :
   ∀ (f:ad → A → M) (a:ad) (y:A), MapFold f (M1 A a y) = f a y.

  Variable State : Set.
  Variable f : State → ad → A → State × M.

  Fixpoint MapFold1_state (state:State) (pf:ad → ad)
   (m:Map A) {struct m} : State × M :=
    match m with
    | M0 ⇒ (state, neutral)
    | M1 a y ⇒ f state (pf a) y
    | M2 m1 m2 ⇒
        match MapFold1_state state (fun a0:ad ⇒ pf (Ndouble a0)) m1 with
        | (state1, x1) ⇒
            match
              MapFold1_state state1
                (fun a0:ad ⇒ pf (Ndouble_plus_one a0)) m2
            with
            | (state2, x2) ⇒ (state2, op x1 x2)
            end
        end
    end.

  Definition MapFold_state (state:State) :=
```

```
    MapFold1_state state (fun a:ad ⇒ a).

  Lemma pair_sp : ∀ (B C:Set) (x:B × C), x = (fst x, snd x).

  Lemma MapFold_state_stateless_1 :
   ∀ (m:Map A) (g:ad → A → M) (pf:ad → ad),
     (∀ (state:State) (a:ad) (y:A), snd (f state a y) = g a y) →
     ∀ state:State, snd (MapFold1_state state pf m) = MapFold1 g pf m.

  Lemma MapFold_state_stateless :
   ∀ g:ad → A → M,
     (∀ (state:State) (a:ad) (y:A), snd (f state a y) = g a y) →
     ∀ (state:State) (m:Map A),
       snd (MapFold_state state m) = MapFold g m.

End MapFoldDef.

Lemma MapCollect_as_Fold :
 ∀ (f:ad → A → Map B) (m:Map A),
   MapCollect f m = MapFold (Map B) (M0 B) (MapMerge B) f m.

Definition alist := list (ad × A).
Definition anil := nil (A:=(ad × A)).
Definition acons := cons (A:=(ad × A)).
Definition aapp := app (A:=(ad × A)).

Definition alist_of_Map :=
  MapFold alist anil aapp (fun (a:ad) (y:A) ⇒ acons (a, y) anil).

Fixpoint alist_semantics (l:alist) : ad → option A :=
  match l with
  | nil ⇒ fun _:ad ⇒ None
  | (a, y) :: l' ⇒
      fun a0:ad ⇒ if Neqb a a0 then Some y else alist_semantics l' a0
  end.

Lemma alist_semantics_app :
 ∀ (l l':alist) (a:ad),
   alist_semantics (aapp l l') a =
   match alist_semantics l a with
   | None ⇒ alist_semantics l' a
   | Some y ⇒ Some y
   end.

Lemma alist_of_Map_semantics_1_1 :
 ∀ (m:Map A) (pf:ad → ad) (a:ad) (y:A),
   alist_semantics
     (MapFold1 alist anil aapp (fun (a0:ad) (y:A) ⇒ acons (a0, y) anil) pf
        m) a = Some y → {a' : ad | a = pf a'}.

Definition ad_inj (pf:ad → ad) :=
  ∀ a0 a1:ad, pf a0 = pf a1 → a0 = a1.
```

```
Lemma ad_comp_double_inj :
 ∀ pf:ad → ad, ad_inj pf → ad_inj (fun a0:ad ⇒ pf (Ndouble a0)).

Lemma ad_comp_double_plus_un_inj :
 ∀ pf:ad → ad,
   ad_inj pf → ad_inj (fun a0:ad ⇒ pf (Ndouble_plus_one a0)).

Lemma alist_of_Map_semantics_1 :
 ∀ (m:Map A) (pf:ad → ad),
   ad_inj pf →
   ∀ a:ad,
     MapGet A m a =
     alist_semantics
       (MapFold1 alist anil aapp (fun (a0:ad) (y:A) ⇒ acons (a0, y) anil)
          pf m) (pf a).

Lemma alist_of_Map_semantics :
 ∀ m:Map A, eqm A (MapGet A m) (alist_semantics (alist_of_Map m)).

Fixpoint Map_of_alist (l:alist) : Map A :=
  match l with
  | nil ⇒ M0 A
  | (a, y) :: l' ⇒ MapPut A (Map_of_alist l') a y
  end.

Lemma Map_of_alist_semantics :
 ∀ l:alist, eqm A (alist_semantics l) (MapGet A (Map_of_alist l)).

Lemma Map_of_alist_of_Map :
 ∀ m:Map A, eqmap A (Map_of_alist (alist_of_Map m)) m.

Lemma alist_of_Map_of_alist :
 ∀ l:alist,
   eqm A (alist_semantics (alist_of_Map (Map_of_alist l)))
     (alist_semantics l).

Lemma fold_right_aapp :
 ∀ (M:Set) (neutral:M) (op:M → M → M),
   (∀ a b c:M, op (op a b) c = op a (op b c)) →
   (∀ a:M, op neutral a = a) →
   ∀ (f:ad → A → M) (l l':alist),
     fold_right (fun (r:ad × A) (m:M) ⇒ let (a, y) := r in op (f a y) m)
       neutral (aapp l l') =
     op
       (fold_right
          (fun (r:ad × A) (m:M) ⇒ let (a, y) := r in op (f a y) m) neutral
          l)
       (fold_right
          (fun (r:ad × A) (m:M) ⇒ let (a, y) := r in op (f a y) m) neutral
          l').
```

```
Lemma MapFold_as_fold_1 :
 ∀ (M:Set) (neutral:M) (op:M → M → M),
   (∀ a b c:M, op (op a b) c = op a (op b c)) →
   (∀ a:M, op neutral a = a) →
   (∀ a:M, op a neutral = a) →
   ∀ (f:ad → A → M) (m:Map A) (pf:ad → ad),
     MapFold1 M neutral op f pf m =
     fold_right (fun (r:ad × A) (m:M) ⇒ let (a, y) := r in op (f a y) m)
       neutral
       (MapFold1 alist anil aapp (fun (a:ad) (y:A) ⇒ acons (a, y) anil) pf
          m).

Lemma MapFold_as_fold :
 ∀ (M:Set) (neutral:M) (op:M → M → M),
   (∀ a b c:M, op (op a b) c = op a (op b c)) →
   (∀ a:M, op neutral a = a) →
   (∀ a:M, op a neutral = a) →
   ∀ (f:ad → A → M) (m:Map A),
     MapFold M neutral op f m =
     fold_right (fun (r:ad × A) (m:M) ⇒ let (a, y) := r in op (f a y) m)
       neutral (alist_of_Map m).

Lemma alist_MapMerge_semantics :
 ∀ m m':Map A,
   eqm A (alist_semantics (aapp (alist_of_Map m') (alist_of_Map m)))
     (alist_semantics (alist_of_Map (MapMerge A m m'))).

Lemma alist_MapMerge_semantics_disjoint :
 ∀ m m':Map A,
   eqmap A (MapDomRestrTo A A m m') (M0 A) →
   eqm A (alist_semantics (aapp (alist_of_Map m) (alist_of_Map m')))
     (alist_semantics (alist_of_Map (MapMerge A m m'))).

Lemma alist_semantics_disjoint_comm :
 ∀ l l':alist,
   eqmap A (MapDomRestrTo A A (Map_of_alist l) (Map_of_alist l')) (M0 A) →
   eqm A (alist_semantics (aapp l l')) (alist_semantics (aapp l' l)).

End MapIter.
```

Chapter 202

Module Coq.IntMap.Maplists

```
Require Import BinNat.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import Fset.
Require Import Mapaxioms.
Require Import Mapsubset.
Require Import Mapcard.
Require Import Mapcanon.
Require Import Mapc.
Require Import Bool.
Require Import Sumbool.
Require Import List.
Require Import Arith.
Require Import Mapiter.
Require Import Mapfold.

Section MapLists.

  Fixpoint ad_in_list (a:ad) (l:list ad) {struct l} : bool :=
    match l with
    | nil ⇒ false
    | a' :: l' ⇒ orb (Neqb a a') (ad_in_list a l')
    end.

  Fixpoint ad_list_stutters (l:list ad) : bool :=
    match l with
    | nil ⇒ false
    | a :: l' ⇒ orb (ad_in_list a l') (ad_list_stutters l')
    end.

  Lemma ad_in_list_forms_circuit :
   ∀ (x:ad) (l:list ad),
      ad_in_list x l = true →
      {l1 : list ad & {l2 : list ad | l = l1 ++ x :: l2}}.
```

Lemma $ad\_list\_stutters\_has\_circuit$:
$\forall$ l:$list\ ad$,
$ad\_list\_stutters\ l = true \rightarrow$
$\{x : ad\ \&$
$\{l0 : list\ ad\ \&$
$\{l1 : list\ ad\ \&\ \{l2 : list\ ad \mid l = l0 +\!\!+ x :: l1 +\!\!+ x :: l2\}\}\}\}$.

Fixpoint $Elems\ (l{:}list\ ad) : FSet :=$
match l with
$\mid nil \Rightarrow M0\ unit$
$\mid a :: l' \Rightarrow MapPut\ \_\ (Elems\ l')\ a\ tt$
end.

Lemma $Elems\_canon : \forall\ l{:}list\ ad,\ mapcanon\ \_\ (Elems\ l)$.

Lemma $Elems\_app$:
$\forall\ l\ l'{:}list\ ad,\ Elems\ (l +\!\!+ l') = FSetUnion\ (Elems\ l)\ (Elems\ l')$.

Lemma $Elems\_rev : \forall\ l{:}list\ ad,\ Elems\ (rev\ l) = Elems\ l$.

Lemma $ad\_in\_elems\_in\_list$:
$\forall\ (l{:}list\ ad)\ (a{:}ad),\ in\_FSet\ a\ (Elems\ l) = ad\_in\_list\ a\ l$.

Lemma $ad\_list\_not\_stutters\_card$:
$\forall\ l{:}list\ ad,$
$ad\_list\_stutters\ l = false \rightarrow length\ l = MapCard\ \_\ (Elems\ l)$.

Lemma $ad\_list\_card : \forall\ l{:}list\ ad,\ MapCard\ \_\ (Elems\ l) \leq length\ l$.

Lemma $ad\_list\_stutters\_card$:
$\forall\ l{:}list\ ad,$
$ad\_list\_stutters\ l = true \rightarrow MapCard\ \_\ (Elems\ l) < length\ l$.

Lemma $ad\_list\_not\_stutters\_card\_conv$:
$\forall\ l{:}list\ ad,$
$length\ l = MapCard\ \_\ (Elems\ l) \rightarrow ad\_list\_stutters\ l = false$.

Lemma $ad\_list\_stutters\_card\_conv$:
$\forall\ l{:}list\ ad,$
$MapCard\ \_\ (Elems\ l) < length\ l \rightarrow ad\_list\_stutters\ l = true$.

Lemma $ad\_in\_list\_l$:
$\forall\ (l\ l'{:}list\ ad)\ (a{:}ad),$
$ad\_in\_list\ a\ l = true \rightarrow ad\_in\_list\ a\ (l +\!\!+ l') = true$.

Lemma $ad\_list\_stutters\_app\_l$:
$\forall\ l\ l'{:}list\ ad,$
$ad\_list\_stutters\ l = true \rightarrow ad\_list\_stutters\ (l +\!\!+ l') = true$.

Lemma $ad\_in\_list\_r$:
$\forall\ (l\ l'{:}list\ ad)\ (a{:}ad),$
$ad\_in\_list\ a\ l' = true \rightarrow ad\_in\_list\ a\ (l +\!\!+ l') = true$.

Lemma $ad\_list\_stutters\_app\_r$:

$\forall\ l\ l'$:$list\ ad$,
$ad\_list\_stutters\ l' = true \rightarrow ad\_list\_stutters\ (l\ ++\ l') = true.$

Lemma $ad\_list\_stutters\_app\_conv\_l$:
$\forall\ l\ l'$:$list\ ad$,
$ad\_list\_stutters\ (l\ ++\ l') = false \rightarrow ad\_list\_stutters\ l = false.$

Lemma $ad\_list\_stutters\_app\_conv\_r$:
$\forall\ l\ l'$:$list\ ad$,
$ad\_list\_stutters\ (l\ ++\ l') = false \rightarrow ad\_list\_stutters\ l' = false.$

Lemma $ad\_in\_list\_app\_1$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$, $ad\_in\_list\ x\ (l\ ++\ x\ ::\ l') = true.$

Lemma $ad\_in\_list\_app$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$,
$ad\_in\_list\ x\ (l\ ++\ l') = orb\ (ad\_in\_list\ x\ l)\ (ad\_in\_list\ x\ l').$

Lemma $ad\_in\_list\_rev$:
$\forall\ (l$:$list\ ad)\ (x$:$ad)$, $ad\_in\_list\ x\ (rev\ l) = ad\_in\_list\ x\ l.$

Lemma $ad\_list\_has\_circuit\_stutters$:
$\forall\ (l0\ l1\ l2$:$list\ ad)\ (x$:$ad)$,
$ad\_list\_stutters\ (l0\ ++\ x\ ::\ l1\ ++\ x\ ::\ l2) = true.$

Lemma $ad\_list\_stutters\_prev\_l$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$,
$ad\_in\_list\ x\ l = true \rightarrow ad\_list\_stutters\ (l\ ++\ x\ ::\ l') = true.$

Lemma $ad\_list\_stutters\_prev\_conv\_l$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$,
$ad\_list\_stutters\ (l\ ++\ x\ ::\ l') = false \rightarrow ad\_in\_list\ x\ l = false.$

Lemma $ad\_list\_stutters\_prev\_r$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$,
$ad\_in\_list\ x\ l' = true \rightarrow ad\_list\_stutters\ (l\ ++\ x\ ::\ l') = true.$

Lemma $ad\_list\_stutters\_prev\_conv\_r$:
$\forall\ (l\ l'$:$list\ ad)\ (x$:$ad)$,
$ad\_list\_stutters\ (l\ ++\ x\ ::\ l') = false \rightarrow ad\_in\_list\ x\ l' = false.$

Lemma ad_list_Elems :
$\forall\ l\ l'$:$list\ ad$,
$MapCard\ \_\ (Elems\ l) = MapCard\ \_\ (Elems\ l') \rightarrow$
$length\ l = length\ l' \rightarrow ad\_list\_stutters\ l = ad\_list\_stutters\ l'.$

Lemma $ad\_list\_app\_length$:
$\forall\ l\ l'$:$list\ ad$, $length\ (l\ ++\ l') = length\ l + length\ l'.$

Lemma $ad\_list\_stutters\_permute$:
$\forall\ l\ l'$:$list\ ad$,
$ad\_list\_stutters\ (l\ ++\ l') = ad\_list\_stutters\ (l'\ ++\ l).$

Lemma $ad\_list\_rev\_length$: $\forall\ l$:$list\ ad$, $length\ (rev\ l) = length\ l.$

Lemma *ad_list_stutters_rev* :
∀ *l*:*list ad*, *ad_list_stutters* (*rev l*) = *ad_list_stutters l*.

Lemma *ad_list_app_rev* :
∀ (*l l'*:*list ad*) (*x*:*ad*), *rev l* ++ *x* :: *l'* = *rev* (*x* :: *l*) ++ *l'*.

Section *ListOfDomDef*.

Variable *A* : Set.

Definition *ad_list_of_dom* :=
MapFold A (*list ad*) *nil* (*app* (*A*:=*ad*)) (fun (*a*:*ad*) (_:*A*) ⇒ *a* :: *nil*).

Lemma *ad_in_list_of_dom_in_dom* :
∀ (*m*:*Map A*) (*a*:*ad*), *ad_in_list a* (*ad_list_of_dom m*) = *in_dom A a m*.

Lemma *Elems_of_list_of_dom* :
∀ *m*:*Map A*, *eqmap unit* (*Elems* (*ad_list_of_dom m*)) (*MapDom A m*).

Lemma *Elems_of_list_of_dom_c* :
∀ *m*:*Map A*, *mapcanon A m* → *Elems* (*ad_list_of_dom m*) = *MapDom A m*.

Lemma *ad_list_of_dom_card_1* :
∀ (*m*:*Map A*) (*pf*:*ad* → *ad*),
length
(*MapFold1 A* (*list ad*) *nil* (*app* (*A*:=*ad*)) (fun (*a*:*ad*) (_:*A*) ⇒ *a* :: *nil*)
pf m) = *MapCard A m*.

Lemma *ad_list_of_dom_card* :
∀ *m*:*Map A*, *length* (*ad_list_of_dom m*) = *MapCard A m*.

Lemma *ad_list_of_dom_not_stutters* :
∀ *m*:*Map A*, *ad_list_stutters* (*ad_list_of_dom m*) = *false*.

End *ListOfDomDef*.

Lemma *ad_list_of_dom_Dom_1* :
∀ (*A*:Set) (*m*:*Map A*) (*pf*:*ad* → *ad*),
MapFold1 A (*list ad*) *nil* (*app* (*A*:=*ad*)) (fun (*a*:*ad*) (_:*A*) ⇒ *a* :: *nil*) *pf*
m =
MapFold1 unit (*list ad*) *nil* (*app* (*A*:=*ad*))
(fun (*a*:*ad*) (_:*unit*) ⇒ *a* :: *nil*) *pf* (*MapDom A m*).

Lemma *ad_list_of_dom_Dom* :
∀ (*A*:Set) (*m*:*Map A*),
ad_list_of_dom A m = *ad_list_of_dom unit* (*MapDom A m*).

End *MapLists*.

Chapter 203

Module Coq.IntMap.Mapsubset

```
Require Import Bool.
Require Import Sumbool.
Require Import Arith.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.
Require Import Map.
Require Import Fset.
Require Import Mapaxioms.
Require Import Mapiter.

Section MapSubsetDef.

  Variables A B : Set.

  Definition MapSubset (m:Map A) (m':Map B) :=
    ∀ a:ad, in_dom A a m = true → in_dom B a m' = true.

  Definition MapSubset_1 (m:Map A) (m':Map B) :=
    match MapSweep A (fun (a:ad) (_:A) ⇒ negb (in_dom B a m')) m with
    | None ⇒ true
    | _ ⇒ false
    end.

  Definition MapSubset_2 (m:Map A) (m':Map B) :=
    eqmap A (MapDomRestrBy A B m m') (M0 A).

  Lemma MapSubset_imp_1 :
   ∀ (m:Map A) (m':Map B), MapSubset m m' → MapSubset_1 m m' = true.

  Lemma MapSubset_1_imp :
   ∀ (m:Map A) (m':Map B), MapSubset_1 m m' = true → MapSubset m m'.

  Lemma map_dom_empty_1 :
   ∀ m:Map A, eqmap A m (M0 A) → ∀ a:ad, in_dom _ a m = false.

  Lemma map_dom_empty_2 :
   ∀ m:Map A, (∀ a:ad, in_dom _ a m = false) → eqmap A m (M0 A).
```

Lemma *MapSubset_imp_2* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*), *MapSubset m m'* $\rightarrow$ *MapSubset_2 m m'*.

Lemma *MapSubset_2_imp* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*), *MapSubset_2 m m'* $\rightarrow$ *MapSubset m m'*.

End *MapSubsetDef*.

Section *MapSubsetOrder*.

Variables *A B C* : Set.

Lemma *MapSubset_refl* : $\forall$ *m*:*Map A*, *MapSubset A A m m*.

Lemma *MapSubset_antisym* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
MapSubset A B m m' $\rightarrow$
MapSubset B A m' m $\rightarrow$ *eqmap unit* (*MapDom A m*) (*MapDom B m'*).

Lemma *MapSubset_trans* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*),
MapSubset A B m m' $\rightarrow$ *MapSubset B C m' m"* $\rightarrow$ *MapSubset A C m m"*.

End *MapSubsetOrder*.

Section *FSubsetOrder*.

Lemma *FSubset_refl* : $\forall$ *s*:*FSet*, *MapSubset _ _ s s*.

Lemma *FSubset_antisym* :
$\forall$ *s s'*:*FSet*,
MapSubset _ _ s s' $\rightarrow$ *MapSubset _ _ s' s* $\rightarrow$ *eqmap unit s s'*.

Lemma *FSubset_trans* :
$\forall$ *s s' s"*:*FSet*,
MapSubset _ _ s s' $\rightarrow$ *MapSubset _ _ s' s"* $\rightarrow$ *MapSubset _ _ s s"*.

End *FSubsetOrder*.

Section *MapSubsetExtra*.

Variables *A B* : Set.

Lemma *MapSubset_Dom_1* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
MapSubset A B m m' $\rightarrow$ *MapSubset unit unit* (*MapDom A m*) (*MapDom B m'*).

Lemma *MapSubset_Dom_2* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
MapSubset unit unit (*MapDom A m*) (*MapDom B m'*) $\rightarrow$ *MapSubset A B m m'*.

Lemma *MapSubset_1_Dom* :
$\forall$ (*m*:*Map A*) (*m'*:*Map B*),
MapSubset_1 A B m m' = *MapSubset_1 unit unit* (*MapDom A m*) (*MapDom B m'*).

Lemma *MapSubset_Put* :

$\forall\ (m{:}Map\ A)\ (a{:}ad)\ (y{:}A),\ MapSubset\ A\ A\ m\ (MapPut\ A\ m\ a\ y).$

Lemma *MapSubset_Put_mono* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (a{:}ad)\ (y{:}A)\ (y'{:}B),$
$MapSubset\ A\ B\ m\ m' \rightarrow MapSubset\ A\ B\ (MapPut\ A\ m\ a\ y)\ (MapPut\ B\ m'\ a\ y').$

Lemma *MapSubset_Put_behind* :
$\forall\ (m{:}Map\ A)\ (a{:}ad)\ (y{:}A),\ MapSubset\ A\ A\ m\ (MapPut\_behind\ A\ m\ a\ y).$

Lemma *MapSubset_Put_behind_mono* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (a{:}ad)\ (y{:}A)\ (y'{:}B),$
$MapSubset\ A\ B\ m\ m' \rightarrow$
$MapSubset\ A\ B\ (MapPut\_behind\ A\ m\ a\ y)\ (MapPut\_behind\ B\ m'\ a\ y').$

Lemma *MapSubset_Remove* :
$\forall\ (m{:}Map\ A)\ (a{:}ad),\ MapSubset\ A\ A\ (MapRemove\ A\ m\ a)\ m.$

Lemma *MapSubset_Remove_mono* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (a{:}ad),$
$MapSubset\ A\ B\ m\ m' \rightarrow MapSubset\ A\ B\ (MapRemove\ A\ m\ a)\ (MapRemove\ B\ m'\ a).$

Lemma *MapSubset_Merge_l* :
$\forall\ m\ m'{:}Map\ A,\ MapSubset\ A\ A\ m\ (MapMerge\ A\ m\ m').$

Lemma *MapSubset_Merge_r* :
$\forall\ m\ m'{:}Map\ A,\ MapSubset\ A\ A\ m'\ (MapMerge\ A\ m\ m').$

Lemma *MapSubset_Merge_mono* :
$\forall\ (m\ m'{:}Map\ A)\ (m''\ m'''{:}Map\ B),$
$MapSubset\ A\ B\ m\ m'' \rightarrow$
$MapSubset\ A\ B\ m'\ m''' \rightarrow$
$MapSubset\ A\ B\ (MapMerge\ A\ m\ m')\ (MapMerge\ B\ m''\ m''').$

Lemma *MapSubset_DomRestrTo_l* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B),\ MapSubset\ A\ A\ (MapDomRestrTo\ A\ B\ m\ m')\ m.$

Lemma *MapSubset_DomRestrTo_r* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B),\ MapSubset\ A\ B\ (MapDomRestrTo\ A\ B\ m\ m')\ m'.$

Lemma *MapSubset_ext* :
$\forall\ (m0\ m1{:}Map\ A)\ (m2\ m3{:}Map\ B),$
$eqmap\ A\ m0\ m1 \rightarrow$
$eqmap\ B\ m2\ m3 \rightarrow MapSubset\ A\ B\ m0\ m2 \rightarrow MapSubset\ A\ B\ m1\ m3.$

Variables $C\ D$: Set.

Lemma *MapSubset_DomRestrTo_mono* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (m''{:}Map\ C)\ (m'''{:}Map\ D),$
$MapSubset\ \_\ \_\ m\ m'' \rightarrow$
$MapSubset\ \_\ \_\ m'\ m''' \rightarrow$
$MapSubset\ \_\ \_\ (MapDomRestrTo\ \_\ \_\ m\ m')\ (MapDomRestrTo\ \_\ \_\ m''\ m''').$

Lemma *MapSubset_DomRestrBy_l* :
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B),\ MapSubset\ A\ A\ (MapDomRestrBy\ A\ B\ m\ m')\ m.$

Lemma *MapSubset_DomRestrBy_mono* :
∀ (*m*:*Map A*) (*m'*:*Map B*) (*m"*:*Map C*) (*m"'*:*Map D*),
MapSubset _ _ *m m"* →
MapSubset _ _ *m"' m'* →
MapSubset _ _ (*MapDomRestrBy* _ _ *m m'*) (*MapDomRestrBy* _ _ *m" m"'*).

End *MapSubsetExtra*.

Section *MapDisjointDef*.

Variables *A B* : Set.

Definition *MapDisjoint* (*m*:*Map A*) (*m'*:*Map B*) :=
∀ *a*:*ad*, *in_dom A a m* = *true* → *in_dom B a m'* = *true* → *False*.

Definition *MapDisjoint_1* (*m*:*Map A*) (*m'*:*Map B*) :=
match *MapSweep A* (fun (*a*:*ad*) (_:*A*) ⇒ *in_dom B a m'*) *m* with
| *None* ⇒ *true*
| _ ⇒ *false*
end.

Definition *MapDisjoint_2* (*m*:*Map A*) (*m'*:*Map B*) :=
eqmap A (*MapDomRestrTo A B m m'*) (*M0 A*).

Lemma *MapDisjoint_imp_1* :
∀ (*m*:*Map A*) (*m'*:*Map B*), *MapDisjoint m m'* → *MapDisjoint_1 m m'* = *true*.

Lemma *MapDisjoint_1_imp* :
∀ (*m*:*Map A*) (*m'*:*Map B*), *MapDisjoint_1 m m'* = *true* → *MapDisjoint m m'*.

Lemma *MapDisjoint_imp_2* :
∀ (*m*:*Map A*) (*m'*:*Map B*), *MapDisjoint m m'* → *MapDisjoint_2 m m'*.

Lemma *MapDisjoint_2_imp* :
∀ (*m*:*Map A*) (*m'*:*Map B*), *MapDisjoint_2 m m'* → *MapDisjoint m m'*.

Lemma *Map_M0_disjoint* : ∀ *m*:*Map B*, *MapDisjoint* (*M0 A*) *m*.

Lemma *Map_disjoint_M0* : ∀ *m*:*Map A*, *MapDisjoint m* (*M0 B*).

End *MapDisjointDef*.

Section *MapDisjointExtra*.

Variables *A B* : Set.

Lemma *MapDisjoint_ext* :
∀ (*m0 m1*:*Map A*) (*m2 m3*:*Map B*),
eqmap A m0 m1 →
eqmap B m2 m3 → *MapDisjoint A B m0 m2* → *MapDisjoint A B m1 m3*.

Lemma *MapMerge_disjoint* :
∀ *m m'*:*Map A*,
MapDisjoint A A m m' →
∀ *a*:*ad*,

in_dom A a (MapMerge A m m') =
orb (andb (in_dom A a m) (negb (in_dom A a m')))
(andb (in_dom A a m') (negb (in_dom A a m))).

Lemma *MapDisjoint_M2_l* :
∀ (m0 m1:Map A) (m2 m3:Map B),
MapDisjoint A B (M2 A m0 m1) (M2 B m2 m3) → MapDisjoint A B m0 m2.

Lemma *MapDisjoint_M2_r* :
∀ (m0 m1:Map A) (m2 m3:Map B),
MapDisjoint A B (M2 A m0 m1) (M2 B m2 m3) → MapDisjoint A B m1 m3.

Lemma *MapDisjoint_M2* :
∀ (m0 m1:Map A) (m2 m3:Map B),
MapDisjoint A B m0 m2 →
MapDisjoint A B m1 m3 → MapDisjoint A B (M2 A m0 m1) (M2 B m2 m3).

Lemma *MapDisjoint_M1_l* :
∀ (m:Map A) (a:ad) (y:B),
MapDisjoint B A (M1 B a y) m → in_dom A a m = false.

Lemma *MapDisjoint_M1_r* :
∀ (m:Map A) (a:ad) (y:B),
MapDisjoint A B m (M1 B a y) → in_dom A a m = false.

Lemma *MapDisjoint_M1_conv_l* :
∀ (m:Map A) (a:ad) (y:B),
in_dom A a m = false → MapDisjoint B A (M1 B a y) m.

Lemma *MapDisjoint_M1_conv_r* :
∀ (m:Map A) (a:ad) (y:B),
in_dom A a m = false → MapDisjoint A B m (M1 B a y).

Lemma *MapDisjoint_sym* :
∀ (m:Map A) (m':Map B), MapDisjoint A B m m' → MapDisjoint B A m' m.

Lemma *MapDisjoint_empty* :
∀ m:Map A, MapDisjoint A A m m → eqmap A m (M0 A).

Lemma *MapDelta_disjoint* :
∀ m m':Map A,
MapDisjoint A A m m' → eqmap A (MapDelta A m m') (MapMerge A m m').

Variable *C* : Set.

Lemma *MapDomRestr_disjoint* :
∀ (m:Map A) (m':Map B) (m":Map C),
MapDisjoint A B (MapDomRestrTo A C m m") (MapDomRestrBy B C m' m").

Lemma *MapDelta_RestrTo_disjoint* :
∀ m m':Map A,
MapDisjoint A A (MapDelta A m m') (MapDomRestrTo A A m m').

Lemma *MapDelta_RestrTo_disjoint_2* :

$\forall\ m\ m'{:}Map\ A,$
$MapDisjoint\ A\ A\ (MapDelta\ A\ m\ m')\ (MapDomRestrTo\ A\ A\ m'\ m).$

Variable D : Set.

Lemma $MapSubset\_Disjoint$:
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (m''{:}Map\ C)\ (m'''{:}Map\ D),$
$MapSubset\ \_\ \_\ m\ m' \rightarrow$
$MapSubset\ \_\ \_\ m''\ m''' \rightarrow$
$MapDisjoint\ \_\ \_\ m'\ m''' \rightarrow MapDisjoint\ \_\ \_\ m\ m''.$

Lemma $MapSubset\_Disjoint\_l$:
$\forall\ (m{:}Map\ A)\ (m'{:}Map\ B)\ (m''{:}Map\ C),$
$MapSubset\ \_\ \_\ m\ m' \rightarrow MapDisjoint\ \_\ \_\ m'\ m'' \rightarrow MapDisjoint\ \_\ \_\ m\ m''.$

Lemma $MapSubset\_Disjoint\_r$:
$\forall\ (m{:}Map\ A)\ (m''{:}Map\ C)\ (m'''{:}Map\ D),$
$MapSubset\ \_\ \_\ m''\ m''' \rightarrow$
$MapDisjoint\ \_\ \_\ m\ m''' \rightarrow MapDisjoint\ \_\ \_\ m\ m''.$

End $MapDisjointExtra$.

Chapter 204

Module Coq.IntMap.Map

Definition of finite sets as trees indexed by adresses

```
Require Import Bool.
Require Import Sumbool.
Require Import NArith.
Require Import Ndigits.
Require Import Ndec.

Definition ad := N.

Section MapDefs.
```

We now define maps from ad to A.

```
  Variable A : Set.

  Inductive Map : Set :=
    | M0 : Map
    | M1 : ad → A → Map
    | M2 : Map → Map → Map.

  Lemma option_sum : ∀ o:option A, {y : A | o = Some y} + {o = None}.
```

The semantics of maps is given by the function *MapGet*. The semantics of a map m is a partial, finite function from ad to A:

```
  Fixpoint MapGet (m:Map) : ad → option A :=
    match m with
    | M0 ⇒ fun a:ad ⇒ None
    | M1 x y ⇒ fun a:ad ⇒ if Neqb x a then Some y else None
    | M2 m1 m2 ⇒
        fun a:ad ⇒
          match a with
          | N0 ⇒ MapGet m1 N0
          | Npos xH ⇒ MapGet m2 N0
          | Npos (xO p) ⇒ MapGet m1 (Npos p)
          | Npos (xI p) ⇒ MapGet m2 (Npos p)
          end
```

```
  end.

Definition newMap := M0.

Definition MapSingleton := M1.

Definition eqm (g g':ad → option A) := ∀ a:ad, g a = g' a.

Lemma newMap_semantics : eqm (MapGet newMap) (fun a:ad ⇒ None).

Lemma MapSingleton_semantics :
 ∀ (a:ad) (y:A),
   eqm (MapGet (MapSingleton a y))
     (fun a':ad ⇒ if Neqb a a' then Some y else None).

Lemma M1_semantics_1 : ∀ (a:ad) (y:A), MapGet (M1 a y) a = Some y.

Lemma M1_semantics_2 :
 ∀ (a a':ad) (y:A), Neqb a a' = false → MapGet (M1 a y) a' = None.

Lemma Map2_semantics_1 :
 ∀ m m':Map,
   eqm (MapGet m) (fun a:ad ⇒ MapGet (M2 m m') (Ndouble a)).

Lemma Map2_semantics_1_eq :
 ∀ (m m':Map) (f:ad → option A),
   eqm (MapGet (M2 m m')) f → eqm (MapGet m) (fun a:ad ⇒ f (Ndouble a)).

Lemma Map2_semantics_2 :
 ∀ m m':Map,
   eqm (MapGet m') (fun a:ad ⇒ MapGet (M2 m m') (Ndouble_plus_one a)).

Lemma Map2_semantics_2_eq :
 ∀ (m m':Map) (f:ad → option A),
   eqm (MapGet (M2 m m')) f →
   eqm (MapGet m') (fun a:ad ⇒ f (Ndouble_plus_one a)).

Lemma MapGet_M2_bit_0_0 :
 ∀ a:ad,
   Nbit0 a = false →
   ∀ m m':Map, MapGet (M2 m m') a = MapGet m (Ndiv2 a).

Lemma MapGet_M2_bit_0_1 :
 ∀ a:ad,
   Nbit0 a = true →
   ∀ m m':Map, MapGet (M2 m m') a = MapGet m' (Ndiv2 a).

Lemma MapGet_M2_bit_0_if :
 ∀ (m m':Map) (a:ad),
   MapGet (M2 m m') a =
   (if Nbit0 a then MapGet m' (Ndiv2 a) else MapGet m (Ndiv2 a)).

Lemma MapGet_M2_bit_0 :
 ∀ (m m' m'':Map) (a:ad),
```

```
    (if Nbit0 a then MapGet (M2 m' m) a else MapGet (M2 m m") a) =
    MapGet m (Ndiv2 a).

Lemma Map2_semantics_3 :
 ∀ m m':Map,
    eqm (MapGet (M2 m m'))
      (fun a:ad ⇒
         match Nbit0 a with
         | false ⇒ MapGet m (Ndiv2 a)
         | true ⇒ MapGet m' (Ndiv2 a)
         end).

Lemma Map2_semantics_3_eq :
 ∀ (m m':Map) (f f':ad → option A),
    eqm (MapGet m) f →
    eqm (MapGet m') f' →
    eqm (MapGet (M2 m m'))
      (fun a:ad ⇒
         match Nbit0 a with
         | false ⇒ f (Ndiv2 a)
         | true ⇒ f' (Ndiv2 a)
         end).

Fixpoint MapPut1 (a:ad) (y:A) (a':ad) (y':A) (p:positive) {struct p} :
 Map :=
  match p with
  | xO p' ⇒
      let m := MapPut1 (Ndiv2 a) y (Ndiv2 a') y' p' in
      match Nbit0 a with
      | false ⇒ M2 m M0
      | true ⇒ M2 M0 m
      end
  | _ ⇒
      match Nbit0 a with
      | false ⇒ M2 (M1 (Ndiv2 a) y) (M1 (Ndiv2 a') y')
      | true ⇒ M2 (M1 (Ndiv2 a') y') (M1 (Ndiv2 a) y)
      end
  end.

Lemma MapGet_if_commute :
 ∀ (b:bool) (m m':Map) (a:ad),
    MapGet (if b then m else m') a = (if b then MapGet m a else MapGet m' a).

Lemma MapGet_if_same :
 ∀ (m:Map) (b:bool) (a:ad), MapGet (if b then m else m) a = MapGet m a.

Lemma MapGet_M2_bit_0_2 :
 ∀ (m m' m":Map) (a:ad),
    MapGet (if Nbit0 a then M2 m m' else M2 m' m") a =
```

```
    MapGet m' (Ndiv2 a).

Lemma MapPut1_semantics_1 :
 ∀ (p:positive) (a a':ad) (y y':A),
    Nxor a a' = Npos p → MapGet (MapPut1 a y a' y' p) a = Some y.

Lemma MapPut1_semantics_2 :
 ∀ (p:positive) (a a':ad) (y y':A),
    Nxor a a' = Npos p → MapGet (MapPut1 a y a' y' p) a' = Some y'.

Lemma MapGet_M2_both_None :
 ∀ (m m':Map) (a:ad),
    MapGet m (Ndiv2 a) = None →
    MapGet m' (Ndiv2 a) = None → MapGet (M2 m m') a = None.

Lemma MapPut1_semantics_3 :
 ∀ (p:positive) (a a' a0:ad) (y y':A),
    Nxor a a' = Npos p →
    Neqb a a0 = false →
    Neqb a' a0 = false → MapGet (MapPut1 a y a' y' p) a0 = None.

Lemma MapPut1_semantics :
 ∀ (p:positive) (a a':ad) (y y':A),
    Nxor a a' = Npos p →
    eqm (MapGet (MapPut1 a y a' y' p))
      (fun a0:ad ⇒
         if Neqb a a0
         then Some y
         else if Neqb a' a0 then Some y' else None).

Lemma MapPut1_semantics' :
 ∀ (p:positive) (a a':ad) (y y':A),
    Nxor a a' = Npos p →
    eqm (MapGet (MapPut1 a y a' y' p))
      (fun a0:ad ⇒
         if Neqb a' a0
         then Some y'
         else if Neqb a a0 then Some y else None).

Fixpoint MapPut (m:Map) : ad → A → Map :=
  match m with
  | M0 ⇒ M1
  | M1 a y ⇒
      fun (a':ad) (y':A) ⇒
        match Nxor a a' with
        | N0 ⇒ M1 a' y'
        | Npos p ⇒ MapPut1 a y a' y' p
        end
  | M2 m1 m2 ⇒
      fun (a:ad) (y:A) ⇒
```

```
        match a with
        | N0 ⇒ M2 (MapPut m1 N0 y) m2
        | Npos xH ⇒ M2 m1 (MapPut m2 N0 y)
        | Npos (xO p) ⇒ M2 (MapPut m1 (Npos p) y) m2
        | Npos (xI p) ⇒ M2 m1 (MapPut m2 (Npos p) y)
        end
  end.

Lemma MapPut_semantics_1 :
 ∀ (a:ad) (y:A) (a0:ad),
   MapGet (MapPut M0 a y) a0 = MapGet (M1 a y) a0.

Lemma MapPut_semantics_2_1 :
 ∀ (a:ad) (y y':A) (a0:ad),
   MapGet (MapPut (M1 a y) a y') a0 =
   (if Neqb a a0 then Some y' else None).

Lemma MapPut_semantics_2_2 :
 ∀ (a a':ad) (y y':A) (a0 a'':ad),
   Nxor a a' = a'' →
   MapGet (MapPut (M1 a y) a' y') a0 =
   (if Neqb a' a0 then Some y' else if Neqb a a0 then Some y else None).

Lemma MapPut_semantics_2 :
 ∀ (a a':ad) (y y':A) (a0:ad),
   MapGet (MapPut (M1 a y) a' y') a0 =
   (if Neqb a' a0 then Some y' else if Neqb a a0 then Some y else None).

Lemma MapPut_semantics_3_1 :
 ∀ (m m':Map) (a:ad) (y:A),
   MapPut (M2 m m') a y =
   (if Nbit0 a
    then M2 m (MapPut m' (Ndiv2 a) y)
    else M2 (MapPut m (Ndiv2 a) y) m').

Lemma MapPut_semantics :
 ∀ (m:Map) (a:ad) (y:A),
   eqm (MapGet (MapPut m a y))
     (fun a':ad ⇒ if Neqb a a' then Some y else MapGet m a').

Fixpoint MapPut_behind (m:Map) : ad → A → Map :=
  match m with
  | M0 ⇒ M1
  | M1 a y ⇒
      fun (a':ad) (y':A) ⇒
        match Nxor a a' with
        | N0 ⇒ m
        | Npos p ⇒ MapPut1 a y a' y' p
        end
  | M2 m1 m2 ⇒
```

```
      fun (a:ad) (y:A) =>
        match a with
        | N0 => M2 (MapPut_behind m1 N0 y) m2
        | Npos xH => M2 m1 (MapPut_behind m2 N0 y)
        | Npos (xO p) => M2 (MapPut_behind m1 (Npos p) y) m2
        | Npos (xI p) => M2 m1 (MapPut_behind m2 (Npos p) y)
        end
  end.

Lemma MapPut_behind_semantics_3_1 :
 ∀ (m m':Map) (a:ad) (y:A),
   MapPut_behind (M2 m m') a y =
   (if Nbit0 a
    then M2 m (MapPut_behind m' (Ndiv2 a) y)
    else M2 (MapPut_behind m (Ndiv2 a) y) m').

Lemma MapPut_behind_as_before_1 :
 ∀ a a' a0:ad,
   Neqb a' a0 = false →
   ∀ y y':A,
     MapGet (MapPut (M1 a y) a' y') a0 =
     MapGet (MapPut_behind (M1 a y) a' y') a0.

Lemma MapPut_behind_as_before :
 ∀ (m:Map) (a:ad) (y:A) (a0:ad),
   Neqb a a0 = false →
   MapGet (MapPut m a y) a0 = MapGet (MapPut_behind m a y) a0.

Lemma MapPut_behind_new :
 ∀ (m:Map) (a:ad) (y:A),
   MapGet (MapPut_behind m a y) a =
   match MapGet m a with
   | Some y' => Some y'
   | _ => Some y
   end.

Lemma MapPut_behind_semantics :
 ∀ (m:Map) (a:ad) (y:A),
   eqm (MapGet (MapPut_behind m a y))
     (fun a':ad =>
        match MapGet m a' with
        | Some y' => Some y'
        | _ => if Neqb a a' then Some y else None
        end).

Definition makeM2 (m m':Map) :=
  match m, m' with
  | M0, M0 => M0
  | M0, M1 a y => M1 (Ndouble_plus_one a) y
```

```
  | M1 a y, M0 ⇒ M1 (Ndouble a) y
  | _, _ ⇒ M2 m m'
  end.

Lemma makeM2_M2 :
 ∀ m m':Map, eqm (MapGet (makeM2 m m')) (MapGet (M2 m m')).

Fixpoint MapRemove (m:Map) : ad → Map :=
  match m with
  | M0 ⇒ fun _:ad ⇒ M0
  | M1 a y ⇒
      fun a':ad ⇒ match Neqb a a' with
                  | true ⇒ M0
                  | false ⇒ m
                  end
  | M2 m1 m2 ⇒
      fun a:ad ⇒
        if Nbit0 a
        then makeM2 m1 (MapRemove m2 (Ndiv2 a))
        else makeM2 (MapRemove m1 (Ndiv2 a)) m2
  end.

Lemma MapRemove_semantics :
 ∀ (m:Map) (a:ad),
   eqm (MapGet (MapRemove m a))
     (fun a':ad ⇒ if Neqb a a' then None else MapGet m a').

Fixpoint MapCard (m:Map) : nat :=
  match m with
  | M0 ⇒ 0
  | M1 _ _ ⇒ 1
  | M2 m m' ⇒ MapCard m + MapCard m'
  end.

Fixpoint MapMerge (m:Map) : Map → Map :=
  match m with
  | M0 ⇒ fun m':Map ⇒ m'
  | M1 a y ⇒ fun m':Map ⇒ MapPut_behind m' a y
  | M2 m1 m2 ⇒
      fun m':Map ⇒
        match m' with
        | M0 ⇒ m
        | M1 a' y' ⇒ MapPut m a' y'
        | M2 m'1 m'2 ⇒ M2 (MapMerge m1 m'1) (MapMerge m2 m'2)
        end
  end.

Lemma MapMerge_semantics :
 ∀ m m':Map,
```

```
  eqm (MapGet (MapMerge m m'))
    (fun a0:ad ⇒
       match MapGet m' a0 with
       | Some y' ⇒ Some y'
       | None ⇒ MapGet m a0
       end).
```

MapInter, *MapRngRestrTo*, *MapRngRestrBy*, *MapInverse* not implemented: need a decidable equality on A.

```
Fixpoint MapDelta (m:Map) : Map → Map :=
  match m with
  | M0 ⇒ fun m':Map ⇒ m'
  | M1 a y ⇒
      fun m':Map ⇒
        match MapGet m' a with
        | None ⇒ MapPut m' a y
        | _ ⇒ MapRemove m' a
        end
  | M2 m1 m2 ⇒
      fun m':Map ⇒
        match m' with
        | M0 ⇒ m
        | M1 a' y' ⇒
            match MapGet m a' with
            | None ⇒ MapPut m a' y'
            | _ ⇒ MapRemove m a'
            end
        | M2 m'1 m'2 ⇒ makeM2 (MapDelta m1 m'1) (MapDelta m2 m'2)
        end
  end.

Lemma MapDelta_semantics_comm :
 ∀ m m':Map, eqm (MapGet (MapDelta m m')) (MapGet (MapDelta m' m)).

Lemma MapDelta_semantics_1_1 :
 ∀ (a:ad) (y:A) (m':Map) (a0:ad),
    MapGet (M1 a y) a0 = None →
    MapGet m' a0 = None → MapGet (MapDelta (M1 a y) m') a0 = None.

Lemma MapDelta_semantics_1 :
 ∀ (m m':Map) (a:ad),
    MapGet m a = None →
    MapGet m' a = None → MapGet (MapDelta m m') a = None.

Lemma MapDelta_semantics_2_1 :
 ∀ (a:ad) (y:A) (m':Map) (a0:ad) (y0:A),
    MapGet (M1 a y) a0 = None →
    MapGet m' a0 = Some y0 → MapGet (MapDelta (M1 a y) m') a0 = Some y0.
```

```
Lemma MapDelta_semantics_2_2 :
 ∀ (a:ad) (y:A) (m':Map) (a0:ad) (y0:A),
   MapGet (M1 a y) a0 = Some y0 →
   MapGet m' a0 = None → MapGet (MapDelta (M1 a y) m') a0 = Some y0.

Lemma MapDelta_semantics_2 :
 ∀ (m m':Map) (a:ad) (y:A),
   MapGet m a = None →
   MapGet m' a = Some y → MapGet (MapDelta m m') a = Some y.

Lemma MapDelta_semantics_3_1 :
 ∀ (a0:ad) (y0:A) (m':Map) (a:ad) (y y':A),
   MapGet (M1 a0 y0) a = Some y →
   MapGet m' a = Some y' → MapGet (MapDelta (M1 a0 y0) m') a = None.

Lemma MapDelta_semantics_3 :
 ∀ (m m':Map) (a:ad) (y y':A),
   MapGet m a = Some y →
   MapGet m' a = Some y' → MapGet (MapDelta m m') a = None.

Lemma MapDelta_semantics :
 ∀ m m':Map,
   eqm (MapGet (MapDelta m m'))
     (fun a0:ad ⇒
        match MapGet m a0, MapGet m' a0 with
        | None, Some y' ⇒ Some y'
        | Some y, None ⇒ Some y
        | _, _ ⇒ None
        end).

Definition MapEmptyp (m:Map) := match m with
                                | M0 ⇒ true
                                | _ ⇒ false
                                end.

Lemma MapEmptyp_correct : MapEmptyp M0 = true.

Lemma MapEmptyp_complete : ∀ m:Map, MapEmptyp m = true → m = M0.
```

MapSplit not implemented: not the preferred way of recursing over Maps (use *MapSweep*, *MapCollect*, or *MapFold* in Mapiter.v.

```
End MapDefs.
```

Chapter 205

Module Coq.FSets.FMapAVL

This module implements map using AVL trees. It follows the implementation from Ocaml's standard library.

Require Import *FSetInterface*.
Require Import *FMapInterface*.
Require Import *FMapList*.

Require Import *ZArith*.
Require Import *Int*.

Module *Raw* (*I*:*Int*)(*X*: *OrderedType*).
Import *I*.
Module *II*:=*MoreInt*(*I*).
Import *II*.
Open Local *Scope Int_scope*.

Module *E* := *X*.
Module *MX* := *OrderedTypeFacts X*.
Module *PX* := *KeyOrderedType X*.
Module *L* := *FMapList.Raw X*.
Import *MX*.
Import *PX*.

Definition *key* := *X.t*.

205.1 Trees

Section *Elt*.

Variable *elt* : Set.

Notation *eqk* := (*eqk* (*elt*:= *elt*)).
Notation *eqke* := (*eqke* (*elt*:= *elt*)).
Notation *ltk* := (*ltk* (*elt*:= *elt*)).

```
Inductive tree : Set :=
  | Leaf : tree
  | Node : tree → key → elt → tree → int → tree.

Notation t := tree.
```

The Sixth field of *Node* is the height of the tree

205.2 Occurrence in a tree

```
Inductive MapsTo (x : key)(e : elt) : tree → Prop :=
  | MapsRoot : ∀ l r h y,
      X.eq x y → MapsTo x e (Node l y e r h)
  | MapsLeft : ∀ l r h y e',
      MapsTo x e l → MapsTo x e (Node l y e' r h)
  | MapsRight : ∀ l r h y e',
      MapsTo x e r → MapsTo x e (Node l y e' r h).

Inductive In (x : key) : tree → Prop :=
  | InRoot : ∀ l r h y e,
      X.eq x y → In x (Node l y e r h)
  | InLeft : ∀ l r h y e',
      In x l → In x (Node l y e' r h)
  | InRight : ∀ l r h y e',
      In x r → In x (Node l y e' r h).

Definition In0 (k:key)(m:t) : Prop := ∃ e:elt, MapsTo k e m.
```

205.3 Binary search trees

lt_tree x s: all elements in *s* are smaller than *x* (resp. greater for *gt_tree*)

```
Definition lt_tree x s := ∀ y:key, In y s → X.lt y x.
Definition gt_tree x s := ∀ y:key, In y s → X.lt x y.
```

bst t : *t* is a binary search tree

```
Inductive bst : tree → Prop :=
  | BSLeaf : bst Leaf
  | BSNode : ∀ x e l r h,
      bst l → bst r → lt_tree x l → gt_tree x r → bst (Node l x e r h).
```

205.4 AVL trees

avl s : *s* is a properly balanced AVL tree, i.e. for any node the heights of the two children differ by at most 2

```
Definition height (s : tree) : int :=
  match s with
  | Leaf ⇒ 0
  | Node _ _ _ _ h ⇒ h
  end.

Inductive avl : tree → Prop :=
  | RBLeaf : avl Leaf
  | RBNode : ∀ x e l r h,
      avl l →
      avl r →
      -(2) ≤ height l - height r ≤ 2 →
      h = max (height l) (height r) + 1 →
      avl (Node l x e r h).

End Elt.
```

Some helpful hints and tactics.

```
Notation t := tree.
Hint Constructors tree.
Hint Constructors MapsTo.
Hint Constructors In.
Hint Constructors bst.
Hint Constructors avl.
Hint Unfold lt_tree gt_tree.

Ltac inv f :=
  match goal with
      | H:f (Leaf _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ (Leaf _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ _ (Leaf _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ _ _ (Leaf _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f (Node _ _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ (Node _ _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ _ (Node _ _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ _ _ (Node _ _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
      | _ ⇒ idtac
  end.

Ltac safe_inv f := match goal with
  | H:f (Node _ _ _ _ _) ⊢ _ ⇒
        generalize H; inversion_clear H; safe_inv f
  | H:f _ (Node _ _ _ _ _) ⊢ _ ⇒
        generalize H; inversion_clear H; safe_inv f
  | _ ⇒ intros
 end.

Ltac inv_all f :=
  match goal with
```

```
    | H: f _ ⊢ _ ⇒ inversion_clear H; inv f
    | H: f _ _ ⊢ _ ⇒ inversion_clear H; inv f
    | H: f _ _ _ ⊢ _ ⇒ inversion_clear H; inv f
    | H: f _ _ _ _ ⊢ _ ⇒ inversion_clear H; inv f
    | _ ⇒ idtac
  end.

Ltac order := match goal with
  | H: lt_tree ?x ?s, H1: In ?y ?s ⊢ _ ⇒ generalize (H _ H1); clear H; order
  | H: gt_tree ?x ?s, H1: In ?y ?s ⊢ _ ⇒ generalize (H _ H1); clear H; order
  | _ ⇒ MX.order
    end.

Ltac intuition_in := repeat progress (intuition; inv In; inv MapsTo).
Ltac firstorder_in := repeat progress (firstorder; inv In; inv MapsTo).

Lemma height_non_negative : ∀ elt (s : t elt), avl s → height s ≥ 0.

Ltac avl_nn_hyp H :=
      let nz := fresh "nz" in assert (nz := height_non_negative H).

Ltac avl_nn h :=
  let t := type of h in
  match type of t with
    | Prop ⇒ avl_nn_hyp h
    | _ ⇒ match goal with H : avl h ⊢ _ ⇒ avl_nn_hyp H end
  end.

Ltac avl_nns :=
  match goal with
      | H:avl _ ⊢ _ ⇒ avl_nn_hyp H; clear H; avl_nns
      | _ ⇒ idtac
  end.
```

Facts about *MapsTo* and *In*.

```
Lemma MapsTo_In : ∀ elt k e (m:t elt), MapsTo k e m → In k m.
Hint Resolve MapsTo_In.

Lemma In_MapsTo : ∀ elt k (m:t elt), In k m → ∃ e, MapsTo k e m.

Lemma In_alt : ∀ elt k (m:t elt), In0 k m ↔ In k m.

Lemma MapsTo_1 :
 ∀ elt (m:t elt) x y e, X.eq x y → MapsTo x e m → MapsTo y e m.
Hint Immediate MapsTo_1.

Lemma In_1 :
 ∀ elt (m:t elt) x y, X.eq x y → In x m → In y m.
```

Results about *lt_tree* and *gt_tree*

```
Lemma lt_leaf : ∀ elt x, lt_tree x (Leaf elt).
```

Lemma *gt_leaf* : ∀ *elt x*, *gt_tree x* (*Leaf elt*).

Lemma *lt_tree_node* : ∀ *elt x y* (*l*:*t elt*) *r e h*,
lt_tree x l → *lt_tree x r* → *X.lt y x* → *lt_tree x* (*Node l y e r h*).

Lemma *gt_tree_node* : ∀ *elt x y* (*l*:*t elt*) *r e h*,
gt_tree x l → *gt_tree x r* → *X.lt x y* → *gt_tree x* (*Node l y e r h*).

Hint *Resolve lt_leaf gt_leaf lt_tree_node gt_tree_node.*

Lemma *lt_left* : ∀ *elt x y* (*l*: *t elt*) *r e h*,
lt_tree x (*Node l y e r h*) → *lt_tree x l.*

Lemma *lt_right* : ∀ *elt x y* (*l*:*t elt*) *r e h*,
lt_tree x (*Node l y e r h*) → *lt_tree x r.*

Lemma *gt_left* : ∀ *elt x y* (*l*:*t elt*) *r e h*,
gt_tree x (*Node l y e r h*) → *gt_tree x l.*

Lemma *gt_right* : ∀ *elt x y* (*l*:*t elt*) *r e h*,
gt_tree x (*Node l y e r h*) → *gt_tree x r.*

Hint *Resolve lt_left lt_right gt_left gt_right.*

Lemma *lt_tree_not_in* :
∀ *elt x* (*t* : *t elt*), *lt_tree x t* → ¬ *In x t.*

Lemma *lt_tree_trans* :
∀ *elt x y*, *X.lt x y* → ∀ (*t*:*t elt*), *lt_tree x t* → *lt_tree y t.*

Lemma *gt_tree_not_in* :
∀ *elt x* (*t* : *t elt*), *gt_tree x t* → ¬ *In x t.*

Lemma *gt_tree_trans* :
∀ *elt x y*, *X.lt y x* → ∀ (*t*:*t elt*), *gt_tree x t* → *gt_tree y t.*

Hint *Resolve lt_tree_not_in lt_tree_trans gt_tree_not_in gt_tree_trans.*

Results about *avl*

Lemma *avl_node* : ∀ *elt x e* (*l*:*t elt*) *r*,
avl l →
avl r →
-(2) ≤ *height l* - *height r* ≤ 2 →
avl (*Node l x e r* (*max* (*height l*) (*height r*) + 1)).
Hint *Resolve avl_node.*

205.5 Helper functions

create l x r creates a node, assuming *l* and *r* to be balanced and |*height l* - *height r*| ≤ 2.

Definition *create elt* (*l*:*t elt*) *x e r* :=
Node l x e r (*max* (*height l*) (*height r*) + 1).

```
Lemma create_bst :
 ∀ elt (l:t elt) x e r, bst l → bst r → lt_tree x l → gt_tree x r →
 bst (create l x e r).
Hint Resolve create_bst.

Lemma create_avl :
 ∀ elt (l:t elt) x e r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 avl (create l x e r).

Lemma create_height :
 ∀ elt (l:t elt) x e r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 height (create l x e r) = max (height l) (height r) + 1.

Lemma create_in :
 ∀ elt (l:t elt) x e r y, In y (create l x e r) ↔ X.eq y x ∨ In y l ∨ In y r.
```

trick for emulating *assert false* in Coq

```
Notation assert_false := Leaf.
```

bal l x e r acts as *create*, but performs one step of rebalancing if necessary, i.e. assumes $|height\ l - height\ r| \leq 3$.

```
Definition bal elt (l: tree elt) x e r :=
  let hl := height l in
  let hr := height r in
  if gt_le_dec hl (hr+2) then
    match l with
     | Leaf ⇒ assert_false _
     | Node ll lx le lr _ ⇒
       if ge_lt_dec (height ll) (height lr) then
          create ll lx le (create lr x e r)
       else
          match lr with
           | Leaf ⇒ assert_false _
           | Node lrl lrx lre lrr _ ⇒
                create (create ll lx le lrl) lrx lre (create lrr x e r)
          end
    end
  else
    if gt_le_dec hr (hl+2) then
      match r with
       | Leaf ⇒ assert_false _
       | Node rl rx re rr _ ⇒
          if ge_lt_dec (height rr) (height rl) then
              create (create l x e rl) rx re rr
          else
            match rl with
             | Leaf ⇒ assert_false _
             | Node rll rlx rle rlr _ ⇒
```

```
                create (create l x e rll) rlx rle (create rlr rx re rr)
            end
        end
      else
        create l x e r.

Ltac bal_tac :=
 intros elt l x e r;
 unfold bal;
 destruct (gt_le_dec (height l) (height r + 2));
   [ destruct l as [ |ll lx le lr lh];
     [ | destruct (ge_lt_dec (height ll) (height lr));
         [ | destruct lr ] ]
   | destruct (gt_le_dec (height r) (height l + 2));
     [ destruct r as [ |rl rx re rr rh];
         [ | destruct (ge_lt_dec (height rr) (height rl));
             [ | destruct rl ] ]
     | ] ]; intros.

Ltac bal_tac_imp := match goal with
  | ⊢ context [ assert_false ] ⇒
      inv avl; avl_nns; simpl in ×; false_omega
  | _ ⇒ idtac
  end.

Lemma bal_bst : ∀ elt (l:t elt) x e r, bst l → bst r →
 lt_tree x l → gt_tree x r → bst (bal l x e r).

Lemma bal_avl : ∀ elt (l:t elt) x e r, avl l → avl r →
 -(3) ≤ height l - height r ≤ 3 → avl (bal l x e r).

Lemma bal_height_1 : ∀ elt (l:t elt) x e r, avl l → avl r →
 -(3) ≤ height l - height r ≤ 3 →
 0 ≤ height (bal l x e r) - max (height l) (height r) ≤ 1.

Lemma bal_height_2 :
 ∀ elt (l:t elt) x e r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 height (bal l x e r) == max (height l) (height r) +1.

Lemma bal_in : ∀ elt (l:t elt) x e r y, avl l → avl r →
 (In y (bal l x e r) ↔ X.eq y x ∨ In y l ∨ In y r).

Lemma bal_mapsto : ∀ elt (l:t elt) x e r y e', avl l → avl r →
 (MapsTo y e' (bal l x e r) ↔ MapsTo y e' (create l x e r)).

Ltac omega_bal := match goal with
  | H:avl ?l, H':avl ?r ⊢ context [ bal ?l ?x ?e ?r ] ⇒
     generalize (bal_height_1 x e H H') (bal_height_2 x e H H');
     omega_max
  end.
```

205.6 Insertion

Lemma *add_avl_1* : ∀ *elt* (*m*:*t elt*) *x e*, *avl m* →
avl (*add x e m*) ∧ 0 ≤ *height* (*add x e m*) - *height m* ≤ 1.

Lemma *add_avl* : ∀ *elt* (*m*:*t elt*) *x e*, *avl m* → *avl* (*add x e m*).
Hint *Resolve add_avl.*

Lemma *add_in* : ∀ *elt* (*m*:*t elt*) *x y e*, *avl m* →
(*In y* (*add x e m*) ↔ *X.eq y x* ∨ *In y m*).

Lemma *add_bst* : ∀ *elt* (*m*:*t elt*) *x e*, *bst m* → *avl m* → *bst* (*add x e m*).

Lemma *add_1* : ∀ *elt* (*m*:*t elt*) *x y e*, *avl m* → *X.eq x y* → *MapsTo y e* (*add x e m*).

Lemma *add_2* : ∀ *elt* (*m*:*t elt*) *x y e e'*, *avl m* → ¬*X.eq x y* →
MapsTo y e m → *MapsTo y e* (*add x e' m*).

Lemma *add_3* : ∀ *elt* (*m*:*t elt*) *x y e e'*, *avl m* → ¬*X.eq x y* →
MapsTo y e (*add x e' m*) → *MapsTo y e m*.

205.7 Extraction of minimum binding

morally, *remove_min* is to be applied to a non-empty tree *t* = *Node l x e r h*. Since we can't deal here with *assert false* for *t*=*Leaf*, we pre-unpack *t* (and forget about *h*).

Lemma *remove_min_avl_1* : ∀ *elt* (*l*:*t elt*) *x e r h*, *avl* (*Node l x e r h*) →
avl (*fst* (*remove_min l x e r*)) ∧
0 ≤ *height* (*Node l x e r h*) - *height* (*fst* (*remove_min l x e r*)) ≤ 1.

Lemma *remove_min_avl* : ∀ *elt* (*l*:*t elt*) *x e r h*, *avl* (*Node l x e r h*) →
avl (*fst* (*remove_min l x e r*)).

Lemma *remove_min_in* : ∀ *elt* (*l*:*t elt*) *x e r h y*, *avl* (*Node l x e r h*) →
(*In y* (*Node l x e r h*) ↔
X.eq y (*fst* (*snd* (*remove_min l x e r*))) ∨ *In y* (*fst* (*remove_min l x e r*))).

Lemma *remove_min_mapsto* : ∀ *elt* (*l*:*t elt*) *x e r h y e'*, *avl* (*Node l x e r h*) →
(*MapsTo y e'* (*Node l x e r h*) ↔
((*X.eq y* (*fst* (*snd* (*remove_min l x e r*))) ∧ *e'* = (*snd* (*snd* (*remove_min l x e r*))))
∨ *MapsTo y e'* (*fst* (*remove_min l x e r*)))).

Lemma *remove_min_bst* : ∀ *elt* (*l*:*t elt*) *x e r h*,
bst (*Node l x e r h*) → *avl* (*Node l x e r h*) → *bst* (*fst* (*remove_min l x e r*)).

Lemma *remove_min_gt_tree* : ∀ *elt* (*l*:*t elt*) *x e r h*,
bst (*Node l x e r h*) → *avl* (*Node l x e r h*) →
gt_tree (*fst* (*snd* (*remove_min l x e r*))) (*fst* (*remove_min l x e r*)).

205.8 Merging two trees

merge t1 t2 builds the union of *t1* and *t2* assuming all elements of *t1* to be smaller than all elements of *t2*, and $|height\ t1 - height\ t2| \leq 2$.

Lemma *merge_avl_1* : $\forall$ *elt* (*s1 s2*:*t elt*), *avl s1* $\rightarrow$ *avl s2* $\rightarrow$
$-(2) \leq height\ s1 - height\ s2 \leq 2 \rightarrow$
avl (*merge s1 s2*) $\wedge$
$0 \leq height\ (merge\ s1\ s2) - max\ (height\ s1)\ (height\ s2) \leq 1$.

Lemma *merge_avl* : $\forall$ *elt* (*s1 s2*:*t elt*), *avl s1* $\rightarrow$ *avl s2* $\rightarrow$
$-(2) \leq height\ s1 - height\ s2 \leq 2 \rightarrow$ *avl* (*merge s1 s2*).

Lemma *merge_in* : $\forall$ *elt* (*s1 s2*:*t elt*) *y*, *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
(*In y* (*merge s1 s2*) $\leftrightarrow$ *In y s1* $\vee$ *In y s2*).

Lemma *merge_mapsto* : $\forall$ *elt* (*s1 s2*:*t elt*) *y e*, *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
(*MapsTo y e* (*merge s1 s2*) $\leftrightarrow$ *MapsTo y e s1* $\vee$ *MapsTo y e s2*).

Lemma *merge_bst* : $\forall$ *elt* (*s1 s2*:*t elt*), *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
($\forall$ *y1 y2* : *key*, *In y1 s1* $\rightarrow$ *In y2 s2* $\rightarrow$ *X.lt y1 y2*) $\rightarrow$
bst (*merge s1 s2*).

205.9 Deletion

Lemma *remove_avl_1* : $\forall$ *elt* (*s*:*t elt*) *x*, *avl s* $\rightarrow$
avl (*remove x s*) $\wedge$ $0 \leq height\ s - height\ (remove\ x\ s) \leq 1$.

Lemma *remove_avl* : $\forall$ *elt* (*s*:*t elt*) *x*, *avl s* $\rightarrow$ *avl* (*remove x s*).
Hint *Resolve remove_avl*.

Lemma *remove_in* : $\forall$ *elt* (*s*:*t elt*) *x y*, *bst s* $\rightarrow$ *avl s* $\rightarrow$
(*In y* (*remove x s*) $\leftrightarrow$ $\neg$ *X.eq y x* $\wedge$ *In y s*).

Lemma *remove_bst* : $\forall$ *elt* (*s*:*t elt*) *x*, *bst s* $\rightarrow$ *avl s* $\rightarrow$ *bst* (*remove x s*).

Lemma *remove_1* : $\forall$ *elt* (*m*:*t elt*) *x y*, *bst m* $\rightarrow$ *avl m* $\rightarrow$ *X.eq x y* $\rightarrow$ $\neg$ *In y* (*remove x m*).

Lemma *remove_2* : $\forall$ *elt* (*m*:*t elt*) *x y e*, *bst m* $\rightarrow$ *avl m* $\rightarrow$ $\neg$*X.eq x y* $\rightarrow$
MapsTo y e m $\rightarrow$ *MapsTo y e* (*remove x m*).

Lemma *remove_3* : $\forall$ *elt* (*m*:*t elt*) *x y e*, *bst m* $\rightarrow$ *avl m* $\rightarrow$
MapsTo y e (*remove x m*) $\rightarrow$ *MapsTo y e m*.

Section *Elt2*.

Variable *elt*:Set.

Notation *eqk* := (*eqk* (*elt*:= *elt*)).
Notation *eqke* := (*eqke* (*elt*:= *elt*)).
Notation *ltk* := (*ltk* (*elt*:= *elt*)).

205.10 Empty map

Definition $Empty\ m := \forall\ (a : key)(e{:}elt)\ ,\ \neg\ MapsTo\ a\ e\ m.$

Definition $empty := (Leaf\ elt).$

Lemma $empty\_bst : bst\ empty.$

Lemma $empty\_avl : avl\ empty.$

Lemma $empty\_1 : Empty\ empty.$

205.11 Emptyness test

Definition $is\_empty\ (s{:}t\ elt) :=$ match s with $Leaf \Rightarrow true \mid \_ \Rightarrow false$ end.

Lemma $is\_empty\_1 : \forall\ s, Empty\ s \rightarrow is\_empty\ s = true.$

Lemma $is\_empty\_2 : \forall\ s, is\_empty\ s = true \rightarrow Empty\ s.$

205.12 Appartness

The *mem* function is deciding appartness. It exploits the *bst* property to achieve logarithmic complexity.

Implicit *Arguments mem.*

Lemma $mem\_1 : \forall\ s\ x, bst\ s \rightarrow In\ x\ s \rightarrow mem\ x\ s = true.$

Lemma $mem\_2 : \forall\ s\ x, mem\ x\ s = true \rightarrow In\ x\ s.$

Lemma $find\_1 : \forall\ m\ x\ e, bst\ m \rightarrow MapsTo\ x\ e\ m \rightarrow find\ x\ m = Some\ e.$

Lemma $find\_2 : \forall\ m\ x\ e, find\ x\ m = Some\ e \rightarrow MapsTo\ x\ e\ m.$

An all-in-one spec for *add* used later in the naive *map2*

Lemma $add\_spec : \forall\ m\ x\ y\ e\ ,\ bst\ m \rightarrow avl\ m \rightarrow$
$find\ x\ (add\ y\ e\ m) =$ if $eq\_dec\ x\ y$ then $Some\ e$ else $find\ x\ m.$

205.13 Elements

elements_tree_aux acc t catenates the elements of *t* in infix order to the list *acc*

Fixpoint $elements\_aux\ (acc : list\ (key{\times}elt))\ (t : t\ elt)\ \{struct\ t\} : list\ (key{\times}elt) :=$
match t with
$\mid Leaf \Rightarrow acc$
$\mid Node\ l\ x\ e\ r\ \_ \Rightarrow elements\_aux\ ((x,e) :: elements\_aux\ acc\ r)\ l$
end.

then *elements* is an instanciation with an empty *acc*

Definition *elements* := *elements_aux nil*.

Lemma *elements_aux_mapsto* : ∀ *s acc x e*,
InA eqke (*x*,*e*) (*elements_aux acc s*) ↔ *MapsTo x e s* ∨ *InA eqke* (*x*,*e*) *acc*.

Lemma *elements_mapsto* : ∀ *s x e*, *InA eqke* (*x*,*e*) (*elements s*) ↔ *MapsTo x e s*.

Lemma *elements_in* : ∀ *s x*, *L.PX.In x* (*elements s*) ↔ *In x s*.

Lemma *elements_aux_sort* : ∀ *s acc*, *bst s* → *sort ltk acc* →
(∀ *x e y*, *InA eqke* (*x*,*e*) *acc* → *In y s* → *X.lt y x*) →
sort ltk (*elements_aux acc s*).

Lemma *elements_sort* : ∀ *s* : *t elt*, *bst s* → *sort ltk* (*elements s*).
Hint *Resolve elements_sort*.

205.14 Fold

Fixpoint *fold* (*A* : Set) (*f* : *key* → *elt* → *A* → *A*)(*s* : *t elt*) {*struct s*} : *A* → *A* :=
fun *a* ⇒ match *s* with
| *Leaf* ⇒ *a*
| *Node l x e r* _ ⇒ *fold f r* (*f x e* (*fold f l a*))
end.

Definition *fold'* (*A* : Set) (*f* : *key* → *elt* → *A* → *A*)(*s* : *t elt*) :=
L.fold f (*elements s*).

Lemma *fold_equiv_aux* :
∀ (*A* : Set) (*s* : *t elt*) (*f* : *key* → *elt* → *A* → *A*) (*a* : *A*) *acc*,
L.fold f (*elements_aux acc s*) *a* = *L.fold f acc* (*fold f s a*).

Lemma *fold_equiv* :
∀ (*A* : Set) (*s* : *t elt*) (*f* : *key* → *elt* → *A* → *A*) (*a* : *A*),
fold f s a = *fold' f s a*.

Lemma *fold_1* :
∀ (*s*:*t elt*)(*Hs*:*bst s*)(*A* : Set)(*i*:*A*)(*f* : *key* → *elt* → *A* → *A*),
fold f s i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements s*) *i*.

205.15 Comparison

Definition *Equal* (*cmp*:*elt*→*elt*→*bool*) *m m'* :=
(∀ *k*, *In k m* ↔ *In k m'*) ∧
(∀ *k e e'*, *MapsTo k e m* → *MapsTo k e' m'* → *cmp e e'* = *true*).

205.15.1 Enumeration of the elements of a tree

```
Inductive enumeration : Set :=
  | End : enumeration
  | More : key → elt → t elt → enumeration → enumeration.
```

flatten_e e returns the list of elements of *e* i.e. the list of elements actually compared

```
Fixpoint flatten_e (e : enumeration) : list (key×elt) := match e with
   | End ⇒ nil
   | More x e t r ⇒ (x,e) :: elements t ++ flatten_e r
 end.
```

sorted_e e expresses that elements in the enumeration *e* are sorted, and that all trees in *e* are binary search trees.

```
Inductive In_e (p:key×elt) : enumeration → Prop :=
   | InEHd1 :
        ∀ (y : key)(d:elt) (s : t elt) (e : enumeration),
        eqke p (y,d) → In_e p (More y d s e)
   | InEHd2 :
        ∀ (y : key) (d:elt) (s : t elt) (e : enumeration),
        MapsTo (fst p) (snd p) s → In_e p (More y d s e)
   | InETl :
        ∀ (y : key) (d:elt) (s : t elt) (e : enumeration),
        In_e p e → In_e p (More y d s e).

Hint Constructors In_e.

Inductive sorted_e : enumeration → Prop :=
   | SortedEEnd : sorted_e End
   | SortedEMore :
        ∀ (x : key) (d:elt) (s : t elt) (e : enumeration),
        bst s →
        (gt_tree x s) →
        sorted_e e →
        (∀ p, In_e p e → ltk (x,d) p) →
        (∀ p,
          MapsTo (fst p) (snd p) s → ∀ q, In_e q e → ltk p q) →
        sorted_e (More x d s e).

Hint Constructors sorted_e.

Lemma in_flatten_e :
 ∀ p e, InA eqke p (flatten_e e) → In_e p e.

Lemma sorted_flatten_e :
 ∀ e : enumeration, sorted_e e → sort ltk (flatten_e e).

Lemma elements_app :
 ∀ s acc, elements_aux acc s = elements s ++ acc.
```

```
Lemma compare_flatten_1 :
 ∀ t1 t2 x e z l,
 elements t1 ++ (x,e) :: elements t2 ++ l =
 elements (Node t1 x e t2 z) ++ l.
```

key lemma for correctness

```
Lemma flatten_e_elements :
 ∀ l r x d z e,
 elements l ++ flatten_e (More x d r e) =
 elements (Node l x d r z) ++ flatten_e e.

Open Local Scope Z_scope.
```

termination of *compare_aux*

```
Fixpoint measure_e_t (s : t elt) : Z := match s with
   | Leaf ⇒ 0
   | Node l _ _ r _ ⇒ 1 + measure_e_t l + measure_e_t r
 end.

Fixpoint measure_e (e : enumeration) : Z := match e with
   | End ⇒ 0
   | More _ _ s r ⇒ 1 + measure_e_t s + measure_e r
 end.

Ltac Measure_e_t := unfold measure_e_t in ⊢ ×; fold measure_e_t in ⊢ ×.
Ltac Measure_e := unfold measure_e in ⊢ ×; fold measure_e in ⊢ ×.

Lemma measure_e_t_0 : ∀ s : t elt, measure_e_t s ≥ 0.

Ltac Measure_e_t_0 s := generalize (@measure_e_t_0 s); intro.

Lemma measure_e_0 : ∀ e : enumeration, measure_e e ≥ 0.

Ltac Measure_e_0 e := generalize (@measure_e_0 e); intro.
```

Induction principle over the sum of the measures for two lists

```
Definition compare_rec2 :
 ∀ P : enumeration → enumeration → Set,
   (∀ x x' : enumeration,
     (∀ y y' : enumeration,
       measure_e y + measure_e y' < measure_e x + measure_e x' → P y y') →
     P x x') →
 ∀ x x' : enumeration, P x x'.
```

cons t e adds the elements of tree *t* on the head of enumeration *e*. Code:

```
    let rec cons s e = match s with | Empty -> e | Node(l, k, d, r, _) -> cons l (More(k, d, r, e))
```

```
Definition cons : ∀ s e, bst s → sorted_e e →
   (∀ x y, MapsTo (fst x) (snd x) s → In_e y e → ltk x y) →
   { r : enumeration
   | sorted_e r ∧
```

```
    measure_e r = measure_e_t s + measure_e e ∧
    flatten_e r = elements s ++ flatten_e e
  }.

Definition equal_aux :
 ∀ (cmp: elt → elt → bool)(e1 e2:enumeration),
 sorted_e e1 → sorted_e e2 →
 { L.Equal cmp (flatten_e e1) (flatten_e e2) } +
 { ¬ L.Equal cmp (flatten_e e1) (flatten_e e2) }.

Lemma Equal_elements : ∀ cmp s s',
 Equal cmp s s' ↔ L.Equal cmp (elements s) (elements s').

Definition equal : ∀ cmp s s', bst s → bst s' →
  {Equal cmp s s'} + {~ Equal cmp s s'}.

End Elt2.

Section Elts.

Variable elt elt' elt" : Set.

Section Map.
Variable f : elt → elt'.

Fixpoint map (m:t elt) {struct m} : t elt' :=
  match m with
    | Leaf ⇒ Leaf _
    | Node l v d r h ⇒ Node (map l) v (f d) (map r) h
  end.

Lemma map_height : ∀ m, height (map m) = height m.

Lemma map_avl : ∀ m, avl m → avl (map m).

Lemma map_1 : ∀ (m: tree elt)(x:key)(e:elt),
     MapsTo x e m → MapsTo x (f e) (map m).

Lemma map_2 : ∀ (m: t elt)(x:key),
     In x (map m) → In x m.

Lemma map_bst : ∀ m, bst m → bst (map m).

End Map.
Section Mapi.
Variable f : key → elt → elt'.

Fixpoint mapi (m:t elt) {struct m} : t elt' :=
  match m with
    | Leaf ⇒ Leaf _
    | Node l v d r h ⇒ Node (mapi l) v (f v d) (mapi r) h
  end.

Lemma mapi_height : ∀ m, height (mapi m) = height m.

Lemma mapi_avl : ∀ m, avl m → avl (mapi m).
```

Lemma *mapi_1* : ∀ (*m*: *tree elt*)(*x*:*key*)(*e*:*elt*),
MapsTo x e m → ∃ *y*, *X.eq y x* ∧ *MapsTo x* (*f y e*) (*mapi m*).

Lemma *mapi_2* : ∀ (*m*: *t elt*)(*x*:*key*),
In x (*mapi m*) → *In x m*.

Lemma *mapi_bst* : ∀ *m*, *bst m* → *bst* (*mapi m*).

End *Mapi*.

Section *Map2*.
Variable *f* : *option elt* → *option elt'* → *option elt"*.

Definition *anti_elements* (*l*:*list* (*key*×*elt"*)) := *L.fold* (@*add* _) *l* (*empty* _).

Definition *map2* (*m*:*t elt*)(*m'*:*t elt'*) : *t elt"* :=
anti_elements (*L.map2 f* (*elements m*) (*elements m'*)).

Lemma *anti_elements_avl_aux* : ∀ (*l*:*list* (*key*×*elt"*))(*m*:*t elt"*),
avl m → *avl* (*L.fold* (@*add* _) *l m*).

Lemma *anti_elements_avl* : ∀ *l*, *avl* (*anti_elements l*).

Lemma *anti_elements_bst_aux* : ∀ (*l*:*list* (*key*×*elt"*))(*m*:*t elt"*),
bst m → *avl m* → *bst* (*L.fold* (@*add* _) *l m*).

Lemma *anti_elements_bst* : ∀ *l*, *bst* (*anti_elements l*).

Lemma *anti_elements_mapsto_aux* : ∀ (*l*:*list* (*key*×*elt"*)) *m k e*,
bst m → *avl m* → *NoDupA* (*eqk* (*elt*:=*elt"*)) *l* →
(∀ *x*, *L.PX.In x l* → *In x m* → *False*) →
(*MapsTo k e* (*L.fold* (@*add* _) *l m*) ↔ *L.PX.MapsTo k e l* ∨ *MapsTo k e m*).

Lemma *anti_elements_mapsto* : ∀ *l k e*, *NoDupA* (*eqk* (*elt*:=*elt"*)) *l* →
(*MapsTo k e* (*anti_elements l*) ↔ *L.PX.MapsTo k e l*).

Lemma *map2_avl* : ∀ (*m*: *t elt*)(*m'*: *t elt'*), *avl* (*map2 m m'*).

Lemma *map2_bst* : ∀ (*m*: *t elt*)(*m'*: *t elt'*), *bst* (*map2 m m'*).

Lemma *find_elements* : ∀ (*elt*:Set)(*m*: *t elt*) *x*, *bst m* →
L.find x (*elements m*) = *find x m*.

Lemma *find_anti_elements* : ∀ (*l*: *list* (*key*×*elt"*)) *x*, *sort* (@*ltk* _) *l* →
find x (*anti_elements l*) = *L.find x l*.

Lemma *map2_1* : ∀ (*m*: *t elt*)(*m'*: *t elt'*)(*x*:*key*), *bst m* → *bst m'* →
In x m ∨ *In x m'* → *find x* (*map2 m m'*) = *f* (*find x m*) (*find x m'*).

Lemma *map2_2* : ∀ (*m*: *t elt*)(*m'*: *t elt'*)(*x*:*key*), *bst m* → *bst m'* →
In x (*map2 m m'*) → *In x m* ∨ *In x m'*.

End *Map2*.
End *Elts*.
End *Raw*.

205.16 Encapsulation

Now, in order to really provide a functor implementing S, we need to encapsulate everything into a type of balanced binary search trees.

```
Module IntMake (I:Int)(X: OrderedType) <: S with Module E := X.

 Module E := X.
 Module Raw := Raw I X.

 Record bbst (elt:Set) : Set :=
  Bbst {this :> Raw.tree elt; is_bst : Raw.bst this; is_avl: Raw.avl this}.

 Definition t := bbst.
 Definition key := E.t.

 Section Elt.
 Variable elt elt' elt": Set.

 Implicit Types m : t elt.
 Implicit Types x y : key.
 Implicit Types e : elt.

 Definition empty : t elt := Bbst (Raw.empty_bst elt) (Raw.empty_avl elt).
 Definition is_empty m : bool := Raw.is_empty m.(this).
 Definition add x e m : t elt :=
  Bbst (Raw.add_bst x e m.(is_bst) m.(is_avl)) (Raw.add_avl x e m.(is_avl)).
 Definition remove x m : t elt :=
  Bbst (Raw.remove_bst x m.(is_bst) m.(is_avl)) (Raw.remove_avl x m.(is_avl)).
 Definition mem x m : bool := Raw.mem x m.(this).
 Definition find x m : option elt := Raw.find x m.(this).
 Definition map f m : t elt' :=
  Bbst (Raw.map_bst f m.(is_bst)) (Raw.map_avl f m.(is_avl)).
 Definition mapi (f:key→elt→elt') m : t elt' :=
  Bbst (Raw.mapi_bst f m.(is_bst)) (Raw.mapi_avl f m.(is_avl)).
 Definition map2 f m (m':t elt') : t elt" :=
  Bbst (Raw.map2_bst f m m') (Raw.map2_avl f m m').
 Definition elements m : list (key×elt) := Raw.elements m.(this).
 Definition fold (A:Set) (f:key→elt→A→A) m i := Raw.fold (A:=A) f m.(this) i.
 Definition equal cmp m m' : bool :=
   if (Raw.equal cmp m.(is_bst) m'.(is_bst)) then true else false.

 Definition MapsTo x e m : Prop := Raw.MapsTo x e m.(this).
 Definition In x m : Prop := Raw.In0 x m.(this).
 Definition Empty m : Prop := Raw.Empty m.(this).

 Definition eq_key : (key×elt) → (key×elt) → Prop := @Raw.PX.eqk elt.
 Definition eq_key_elt : (key×elt) → (key×elt) → Prop := @Raw.PX.eqke elt.
 Definition lt_key : (key×elt) → (key×elt) → Prop := @Raw.PX.ltk elt.

 Lemma MapsTo_1 : ∀ m x y e, E.eq x y → MapsTo x e m → MapsTo y e m.
```

Lemma *mem_1* : ∀ *m x*, *In x m* → *mem x m* = *true*.

Lemma *mem_2* : ∀ *m x*, *mem x m* = *true* → *In x m*.

Lemma *empty_1* : *Empty empty*.

Lemma *is_empty_1* : ∀ *m*, *Empty m* → *is_empty m* = *true*.
Lemma *is_empty_2* : ∀ *m*, *is_empty m* = *true* → *Empty m*.

Lemma *add_1* : ∀ *m x y e*, *E.eq x y* → *MapsTo y e* (*add x e m*).
Lemma *add_2* : ∀ *m x y e e'*, ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*add x e' m*).
Lemma *add_3* : ∀ *m x y e e'*, ¬ *E.eq x y* → *MapsTo y e* (*add x e' m*) → *MapsTo y e m*.

Lemma *remove_1* : ∀ *m x y*, *E.eq x y* → ¬ *In y* (*remove x m*).
Lemma *remove_2* : ∀ *m x y e*, ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*remove x m*).
Lemma *remove_3* : ∀ *m x y e*, *MapsTo y e* (*remove x m*) → *MapsTo y e m*.

Lemma *find_1* : ∀ *m x e*, *MapsTo x e m* → *find x m* = *Some e*.
Lemma *find_2* : ∀ *m x e*, *find x m* = *Some e* → *MapsTo x e m*.

Lemma *fold_1* : ∀ *m* (*A* : Set) (*i* : *A*) (*f* : *key* → *elt* → *A* → *A*),
fold f m i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

Lemma *elements_1* : ∀ *m x e*,
MapsTo x e m → *InA eq_key_elt* (*x*,*e*) (*elements m*).

Lemma *elements_2* : ∀ *m x e*,
InA eq_key_elt (*x*,*e*) (*elements m*) → *MapsTo x e m*.

Lemma *elements_3* : ∀ *m*, *sort lt_key* (*elements m*).

Definition *Equal cmp m m'* :=
(∀ *k*, *In k m* ↔ *In k m'*) ∧
(∀ *k e e'*, *MapsTo k e m* → *MapsTo k e' m'* → *cmp e e'* = *true*).

Lemma *Equal_Equal* : ∀ *cmp m m'*, *Equal cmp m m'* ↔ *Raw.Equal cmp m m'*.

Lemma *equal_1* : ∀ *m m' cmp*,
Equal cmp m m' → *equal cmp m m'* = *true*.

Lemma *equal_2* : ∀ *m m' cmp*,
equal cmp m m' = *true* → *Equal cmp m m'*.

End *Elt*.

Lemma *map_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*elt*→*elt'*),
MapsTo x e m → *MapsTo x* (*f e*) (*map f m*).

Lemma *map_2* : ∀ (*elt elt'*:Set)(*m*:*t elt*)(*x*:*key*)(*f*:*elt*→*elt'*), *In x* (*map f m*) → *In x m*.

Lemma *mapi_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)
(*f*:*key*→*elt*→*elt'*), *MapsTo x e m* →
∃ *y*, *E.eq y x* ∧ *MapsTo x* (*f y e*) (*mapi f m*).
Lemma *mapi_2* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)
(*f*:*key*→*elt*→*elt'*), *In x* (*mapi f m*) → *In x m*.

Lemma *map2_1* : ∀ (*elt elt' elt''*:Set)(*m*: *t elt*)(*m'*: *t elt'*)

```
    (x:key)(f:option elt→option elt'→option elt"),
    In x m ∨ In x m' →
    find x (map2 f m m') = f (find x m) (find x m').

  Lemma map2_2 : ∀ (elt elt' elt":Set)(m: t elt)(m': t elt')
     (x:key)(f:option elt→option elt'→option elt"),
     In x (map2 f m m') → In x m ∨ In x m'.

End IntMake.

Module IntMake_ord (I:Int)(X: OrderedType)(D : OrderedType) <:
    Sord with Module Data := D
              with Module MapS.E := X.

  Module Data := D.
  Module MapS := IntMake(I)(X).
  Import MapS.

  Module MD := OrderedTypeFacts(D).
  Import MD.

  Module LO := FMapList.Make_ord(X)(D).

  Definition t := MapS.t D.t.

  Definition cmp e e' := match D.compare e e' with EQ _ ⇒ true | _ ⇒ false end.

  Definition elements (m:t) :=
    LO.MapS.Build_slist (Raw.elements_sort m.(is_bst)).

  Definition eq : t → t → Prop :=
    fun m1 m2 ⇒ LO.eq (elements m1) (elements m2).

  Definition lt : t → t → Prop :=
    fun m1 m2 ⇒ LO.lt (elements m1) (elements m2).

  Lemma eq_1 : ∀ m m', Equal cmp m m' → eq m m'.

  Lemma eq_2 : ∀ m m', eq m m' → Equal cmp m m'.

  Lemma eq_refl : ∀ m : t, eq m m.

  Lemma eq_sym : ∀ m1 m2 : t, eq m1 m2 → eq m2 m1.

  Lemma eq_trans : ∀ m1 m2 m3 : t, eq m1 m2 → eq m2 m3 → eq m1 m3.

  Lemma lt_trans : ∀ m1 m2 m3 : t, lt m1 m2 → lt m2 m3 → lt m1 m3.

  Lemma lt_not_eq : ∀ m1 m2 : t, lt m1 m2 → ¬ eq m1 m2.

  Import Raw.

  Definition flatten_slist (e:enumeration D.t)(He:sorted_e e) :=
   LO.MapS.Build_slist (sorted_flatten_e He).

  Open Local Scope Z_scope.

  Definition compare_aux :
```

$\forall$ (*e1 e2*:*enumeration D.t*)(*He1*:*sorted_e e1*)(*He2*: *sorted_e e2*),
Compare LO.lt LO.eq (*flatten_slist He1*) (*flatten_slist He2*).

Definition *compare* : $\forall$ *m1 m2*, *Compare lt eq m1 m2*.

End *IntMake_ord*.

Module *Make* (*X*: *OrderedType*) <: *S* with Module *E* := *X*
:=*IntMake*(*Z_as_Int*)(*X*).

Module *Make_ord* (*X*: *OrderedType*)(*D*: *OrderedType*)
<: *Sord* with Module *Data* := *D*
with Module *MapS.E* := *X*
:=*IntMake_ord*(*Z_as_Int*)(*X*)(*D*).

Chapter 206

Module Coq.FSets.FMapFacts

206.1 Finite maps library

This functor derives additional facts from *FMapInterface.S*. These facts are mainly the specifications of *FMapInterface.S* written using different styles: equivalence and boolean equalities.

Require Import *Bool*.
Require Import *OrderedType*.
Require Export *FMapInterface*.

Module *Facts* (*M*: *S*).
Module *ME* := *OrderedTypeFacts M.E*.
Import *ME*.
Import *M*.
Import *Logic*.
Import *Peano*.
Lemma *MapsTo_fun* : ∀ (*elt*:Set) *m x* (*e e'*:*elt*),
MapsTo x e m → *MapsTo x e' m* → *e*=*e'*.

206.2 Specifications written using equivalences

Section *IffSpec*.
Variable *elt elt' elt''*: Set.
Implicit Type *m*: *t elt*.
Implicit Type *x y z*: *key*.
Implicit Type *e*: *elt*.

Lemma *MapsTo_iff* : ∀ *m x y e*, *E.eq x y* → (*MapsTo x e m* ↔ *MapsTo y e m*).

Lemma *In_iff* : ∀ *m x y*, *E.eq x y* → (*In x m* ↔ *In y m*).

Lemma *find_mapsto_iff* : ∀ *m x e*, *MapsTo x e m* ↔ *find x m* = *Some e*.

Lemma *not_find_mapsto_iff* : ∀ *m x*, ¬*In x m* ↔ *find x m* = *None*.

Lemma $mem\_in\_iff : \forall\ m\ x,\ In\ x\ m \leftrightarrow mem\ x\ m = true.$

Lemma $not\_mem\_in\_iff : \forall\ m\ x,\ \neg In\ x\ m \leftrightarrow mem\ x\ m = false.$

Lemma $equal\_iff : \forall\ m\ m'\ cmp,\ Equal\ cmp\ m\ m' \leftrightarrow equal\ cmp\ m\ m' = true.$

Lemma $empty\_mapsto\_iff : \forall\ x\ e,\ MapsTo\ x\ e\ (empty\ elt) \leftrightarrow False.$

Lemma $empty\_in\_iff : \forall\ x,\ In\ x\ (empty\ elt) \leftrightarrow False.$

Lemma $is\_empty\_iff : \forall\ m,\ Empty\ m \leftrightarrow is\_empty\ m = true.$

Lemma $add\_mapsto\_iff : \forall\ m\ x\ y\ e\ e',$
$MapsTo\ y\ e'\ (add\ x\ e\ m) \leftrightarrow$
$(E.eq\ x\ y \wedge e{=}e') \vee$
$(\tilde{}E.eq\ x\ y \wedge MapsTo\ y\ e'\ m).$

Lemma $add\_in\_iff : \forall\ m\ x\ y\ e,\ In\ y\ (add\ x\ e\ m) \leftrightarrow E.eq\ x\ y \vee In\ y\ m.$

Lemma $add\_neq\_mapsto\_iff : \forall\ m\ x\ y\ e\ e',$
$\neg\ E.eq\ x\ y \rightarrow (MapsTo\ y\ e'\ (add\ x\ e\ m) \leftrightarrow MapsTo\ y\ e'\ m).$

Lemma $add\_neq\_in\_iff : \forall\ m\ x\ y\ e,$
$\neg\ E.eq\ x\ y \rightarrow (In\ y\ (add\ x\ e\ m) \leftrightarrow In\ y\ m).$

Lemma $remove\_mapsto\_iff : \forall\ m\ x\ y\ e,$
$MapsTo\ y\ e\ (remove\ x\ m) \leftrightarrow \neg E.eq\ x\ y \wedge MapsTo\ y\ e\ m.$

Lemma $remove\_in\_iff : \forall\ m\ x\ y,\ In\ y\ (remove\ x\ m) \leftrightarrow \neg E.eq\ x\ y \wedge In\ y\ m.$

Lemma $remove\_neq\_mapsto\_iff : \forall\ m\ x\ y\ e,$
$\neg\ E.eq\ x\ y \rightarrow (MapsTo\ y\ e\ (remove\ x\ m) \leftrightarrow MapsTo\ y\ e\ m).$

Lemma $remove\_neq\_in\_iff : \forall\ m\ x\ y,$
$\neg\ E.eq\ x\ y \rightarrow (In\ y\ (remove\ x\ m) \leftrightarrow In\ y\ m).$

Lemma $elements\_mapsto\_iff : \forall\ m\ x\ e,$
$MapsTo\ x\ e\ m \leftrightarrow InA\ (@eq\_key\_elt\ \_)\ (x,e)\ (elements\ m).$

Lemma $elements\_in\_iff : \forall\ m\ x,$
$In\ x\ m \leftrightarrow \exists\ e,\ InA\ (@eq\_key\_elt\ \_)\ (x,e)\ (elements\ m).$

Lemma $map\_mapsto\_iff : \forall\ m\ x\ b\ (f : elt \rightarrow elt'),$
$MapsTo\ x\ b\ (map\ f\ m) \leftrightarrow \exists\ a,\ b = f\ a \wedge MapsTo\ x\ a\ m.$

Lemma $map\_in\_iff : \forall\ m\ x\ (f : elt \rightarrow elt'),$
$In\ x\ (map\ f\ m) \leftrightarrow In\ x\ m.$

Lemma $mapi\_in\_iff : \forall\ m\ x\ (f{:}key{\rightarrow}elt{\rightarrow}elt'),$
$In\ x\ (mapi\ f\ m) \leftrightarrow In\ x\ m.$

Lemma $mapi\_inv : \forall\ m\ x\ b\ (f : key \rightarrow elt \rightarrow elt'),$
$MapsTo\ x\ b\ (mapi\ f\ m) \rightarrow$
$\exists\ a, \exists\ y, E.eq\ y\ x \wedge b = f\ y\ a \wedge MapsTo\ x\ a\ m.$

Lemma $mapi\_1bis : \forall\ m\ x\ e\ (f{:}key{\rightarrow}elt{\rightarrow}elt'),$
$(\forall\ x\ y\ e, E.eq\ x\ y \rightarrow f\ x\ e = f\ y\ e) \rightarrow$

MapsTo x e m → MapsTo x (f x e) (mapi f m).

Lemma *mapi_mapsto_iff* : ∀ *m x b* (*f*:*key*→*elt*→*elt'*),
(∀ *x y e*, *E.eq x y* → *f x e* = *f y e*) →
(*MapsTo x b* (*mapi f m*) ↔ ∃ *a*, *b* = *f x a* ∧ *MapsTo x a m*).

Things are even worse for *map2* : we don't try to state any equivalence, see instead boolean results below.

End *IffSpec*.

Useful tactic for simplifying expressions like *In y* (*add x e* (*remove z m*))

Ltac *map_iff* :=
repeat (*progress* (
rewrite add_mapsto_iff || *rewrite add_in_iff* ||
rewrite remove_mapsto_iff || *rewrite remove_in_iff* ||
rewrite empty_mapsto_iff || *rewrite empty_in_iff* ||
rewrite map_mapsto_iff || *rewrite map_in_iff* ||
rewrite mapi_in_iff)).

206.3 Specifications written using boolean predicates

Section *BoolSpec*.

Lemma *mem_find_b* : ∀ (*elt*:Set)(*m*:*t elt*)(*x*:*key*), *mem x m* = if *find x m* then *true* else *false*.

Variable *elt elt' elt"* : Set.
Implicit *Types m* : *t elt*.
Implicit *Types x y z* : *key*.
Implicit *Types e* : *elt*.

Lemma *mem_b* : ∀ *m x y*, *E.eq x y* → *mem x m* = *mem y m*.

Lemma *find_o* : ∀ *m x y*, *E.eq x y* → *find x m* = *find y m*.

Lemma *empty_o* : ∀ *x*, *find x* (*empty elt*) = *None*.

Lemma *empty_a* : ∀ *x*, *mem x* (*empty elt*) = *false*.

Lemma *add_eq_o* : ∀ *m x y e*,
E.eq x y → *find y* (*add x e m*) = *Some e*.

Lemma *add_neq_o* : ∀ *m x y e*,
¬ *E.eq x y* → *find y* (*add x e m*) = *find y m*.
Hint *Resolve add_neq_o*.

Lemma *add_o* : ∀ *m x y e*,
find y (*add x e m*) = if *eq_dec x y* then *Some e* else *find y m*.

Lemma *add_eq_b* : ∀ *m x y e*,
E.eq x y → *mem y* (*add x e m*) = *true*.

Lemma *add_neq_b* : ∀ *m x y e*,

```
 ¬E.eq x y → mem y (add x e m) = mem y m.

Lemma add_b : ∀ m x y e,
 mem y (add x e m) = eqb x y || mem y m.

Lemma remove_eq_o : ∀ m x y,
 E.eq x y → find y (remove x m) = None.
Hint Resolve remove_eq_o.

Lemma remove_neq_o : ∀ m x y,
 ¬ E.eq x y → find y (remove x m) = find y m.
Hint Resolve remove_neq_o.

Lemma remove_o : ∀ m x y,
 find y (remove x m) = if eq_dec x y then None else find y m.

Lemma remove_eq_b : ∀ m x y,
 E.eq x y → mem y (remove x m) = false.

Lemma remove_neq_b : ∀ m x y,
 ¬ E.eq x y → mem y (remove x m) = mem y m.

Lemma remove_b : ∀ m x y,
 mem y (remove x m) = negb (eqb x y) && mem y m.

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

Lemma map_o : ∀ m x (f:elt→elt'),
 find x (map f m) = option_map f (find x m).

Lemma map_b : ∀ m x (f:elt→elt'),
 mem x (map f m) = mem x m.

Lemma mapi_b : ∀ m x (f:key→elt→elt'),
 mem x (mapi f m) = mem x m.

Lemma mapi_o : ∀ m x (f:key→elt→elt'),
 (∀ x y e, E.eq x y → f x e = f y e) →
 find x (mapi f m) = option_map (f x) (find x m).

Lemma map2_1bis : ∀ (m: t elt)(m': t elt') x
 (f:option elt→option elt'→option elt"),
 f None None = None →
 find x (map2 f m m') = f (find x m) (find x m').

Lemma elements_o : ∀ m x,
 find x m = findA (eqb x) (elements m).

Lemma elements_b : ∀ m x, mem x m = existsb (fun p ⇒ eqb x (fst p)) (elements m).

End BoolSpec.

End Facts.
```

Chapter 207

Module Coq.FSets.FMapInterface

207.1 Finite map library

This file proposes an interface for finite maps

When compared with Ocaml Map, this signature has been split in two:

- The first part S contains the usual operators (add, find, ...) It only requires a ordered key type, the data type can be arbitrary. The only function that asks more is *equal*, whose first argument should be an equality on data.
- Then, *Sord* extends S with a complete comparison function. For that, the data type should have a decidable total ordering.

Module Type S.

Declare Module E : *OrderedType*.

Definition $key := E.t$.

Parameter t : Set → Set.
the abstract type of maps

Section *Types*.

Variable *elt*:Set.

Parameter *empty* : $t\ elt$.
The empty map.

Parameter *is_empty* : $t\ elt \rightarrow bool$.
Test whether a map is empty or not.

Parameter *add* : $key \rightarrow elt \rightarrow t\ elt \rightarrow t\ elt$.
$add\ x\ y\ m$ returns a map containing the same bindings as m, plus a binding of x to y. If x was already bound in m, its previous binding disappears.

Parameter *find* : *key* → *t elt* → *option elt*.
find x m returns the current binding of *x* in *m*, or raises *Not_found* if no such binding exists. NB: in Coq, the exception mechanism becomes a option type.

Parameter *remove* : *key* → *t elt* → *t elt*.
remove x m returns a map containing the same bindings as *m*, except for *x* which is unbound in the returned map.

Parameter *mem* : *key* → *t elt* → *bool*.
mem x m returns *true* if *m* contains a binding for *x*, and *false* otherwise.

Coq comment: *iter* is useless in a purely functional world
val iter : (key -> 'a -> unit) -> 'a t -> unit
iter f m applies f to all bindings in map m. f receives the key as first argument, and the associated value as second argument. The bindings are passed to f in increasing order with respect to the ordering over the type of the keys. Only current bindings are presented to f: bindings hidden by more recent bindings are not passed to f.

Variable *elt'* : Set.
Variable *elt"*: Set.

Parameter *map* : (*elt* → *elt'*) → *t elt* → *t elt'*.
map f m returns a map with same domain as *m*, where the associated value a of all bindings of *m* has been replaced by the result of the application of *f* to *a*. The bindings are passed to *f* in increasing order with respect to the ordering over the type of the keys.

Parameter *mapi* : (*key* → *elt* → *elt'*) → *t elt* → *t elt'*.
Same as *S.map*, but the function receives as arguments both the key and the associated value for each binding of the map.

Parameter *map2* : (*option elt* → *option elt'* → *option elt"*) → *t elt* → *t elt'* → *t elt"*.
Not present in Ocaml. *map f m m'* creates a new map whose bindings belong to the ones of either *m* or *m'*. The presence and value for a key *k* is determined by *f e e'* where *e* and *e'* are the (optional) bindings of *k* in *m* and *m'*.

Parameter *elements* : *t elt* → *list* (*key*×*elt*).
Not present in Ocaml. *elements m* returns an assoc list corresponding to the bindings of *m*. Elements of this list are sorted with respect to their first components. Useful to specify *fold* ...

Parameter *fold* : ∀ *A*: Set, (*key* → *elt* → *A* → *A*) → *t elt* → *A* → *A*.
fold f m a computes (*f kN dN* ... (*f k1 d1 a*)...), where *k1* ... *kN* are the keys of all bindings in *m* (in increasing order), and *d1* ... *dN* are the associated data.

Parameter *equal* : (*elt* → *elt* → *bool*) → *t elt* → *t elt* → *bool*.
equal cmp m1 m2 tests whether the maps *m1* and *m2* are equal, that is, contain equal keys and associate them with equal data. *cmp* is the equality predicate used to compare the data associated with the keys.

Section *Spec*.

Variable *m m' m"* : *t elt*.
Variable *x y z* : *key*.
Variable *e e'* : *elt*.

Parameter $MapsTo$: $key \to elt \to t\ elt \to$ Prop.

Definition $In\ (k{:}key)(m{:}\ t\ elt)$: Prop := $\exists\ e{:}elt,\ MapsTo\ k\ e\ m$.

Definition $Empty\ m := \forall\ (a : key)(e{:}elt)\ ,\ \neg\ MapsTo\ a\ e\ m$.

Definition $eq\_key\ (p\ p'{:}key \times elt) := E.eq\ (fst\ p)\ (fst\ p')$.

Definition $eq\_key\_elt\ (p\ p'{:}key \times elt) :=$
$E.eq\ (fst\ p)\ (fst\ p') \wedge (snd\ p) = (snd\ p')$.

Definition $lt\_key\ (p\ p'{:}key \times elt) := E.lt\ (fst\ p)\ (fst\ p')$.

Specification of $MapsTo$

Parameter $MapsTo\_1$: $E.eq\ x\ y \to MapsTo\ x\ e\ m \to MapsTo\ y\ e\ m$.

Specification of mem

Parameter mem_1 : $In\ x\ m \to mem\ x\ m = true$.
Parameter mem_2 : $mem\ x\ m = true \to In\ x\ m$.

Specification of $empty$

Parameter $empty\_1$: $Empty\ empty$.

Specification of is_empty

Parameter is_empty_1 : $Empty\ m \to is\_empty\ m = true$.
Parameter is_empty_2 : $is\_empty\ m = true \to Empty\ m$.

Specification of add

Parameter add_1 : $E.eq\ x\ y \to MapsTo\ y\ e\ (add\ x\ e\ m)$.
Parameter add_2 : $\neg\ E.eq\ x\ y \to MapsTo\ y\ e\ m \to MapsTo\ y\ e\ (add\ x\ e'\ m)$.
Parameter add_3 : $\neg\ E.eq\ x\ y \to MapsTo\ y\ e\ (add\ x\ e'\ m) \to MapsTo\ y\ e\ m$.

Specification of $remove$

Parameter $remove\_1$: $E.eq\ x\ y \to \neg\ In\ y\ (remove\ x\ m)$.
Parameter $remove\_2$: $\neg\ E.eq\ x\ y \to MapsTo\ y\ e\ m \to MapsTo\ y\ e\ (remove\ x\ m)$.
Parameter $remove\_3$: $MapsTo\ y\ e\ (remove\ x\ m) \to MapsTo\ y\ e\ m$.

Specification of $find$

Parameter $find\_1$: $MapsTo\ x\ e\ m \to find\ x\ m = Some\ e$.
Parameter $find\_2$: $find\ x\ m = Some\ e \to MapsTo\ x\ e\ m$.

Specification of $elements$

Parameter $elements\_1$:
$MapsTo\ x\ e\ m \to InA\ eq\_key\_elt\ (x,e)\ (elements\ m)$.
Parameter $elements\_2$:
$InA\ eq\_key\_elt\ (x,e)\ (elements\ m) \to MapsTo\ x\ e\ m$.
Parameter $elements\_3$: $sort\ lt\_key\ (elements\ m)$.

Specification of $fold$

Parameter $fold\_1$:
$\forall\ (A : \mathsf{Set})\ (i : A)\ (f : key \to elt \to A \to A)$,
$fold\ f\ m\ i = fold\_left\ (\mathsf{fun}\ a\ p \Rightarrow f\ (fst\ p)\ (snd\ p)\ a)\ (elements\ m)\ i$.

Definition $Equal\ cmp\ m\ m'$:=

$(\forall\ k,\ In\ k\ m \leftrightarrow In\ k\ m') \wedge$
$(\forall\ k\ e\ e',\ MapsTo\ k\ e\ m \rightarrow MapsTo\ k\ e'\ m' \rightarrow cmp\ e\ e' = true).$

Variable $cmp : elt \rightarrow elt \rightarrow bool.$

Specification of *equal*

Parameter $equal\_1 : Equal\ cmp\ m\ m' \rightarrow equal\ cmp\ m\ m' = true.$
Parameter $equal\_2 : equal\ cmp\ m\ m' = true \rightarrow Equal\ cmp\ m\ m'.$

End *Spec*.
End *Types*.

Specification of *map*

Parameter $map\_1 : \forall\ (elt\ elt'{:}\mathsf{Set})(m{:}\ t\ elt)(x{:}key)(e{:}elt)(f{:}elt{\rightarrow}elt'),$
$MapsTo\ x\ e\ m \rightarrow MapsTo\ x\ (f\ e)\ (map\ f\ m).$
Parameter $map\_2 : \forall\ (elt\ elt'{:}\mathsf{Set})(m{:}\ t\ elt)(x{:}key)(f{:}elt{\rightarrow}elt'),$
$In\ x\ (map\ f\ m) \rightarrow In\ x\ m.$

Specification of *mapi*

Parameter $mapi\_1 : \forall\ (elt\ elt'{:}\mathsf{Set})(m{:}\ t\ elt)(x{:}key)(e{:}elt)$
$(f{:}key{\rightarrow}elt{\rightarrow}elt'),\ MapsTo\ x\ e\ m \rightarrow$
$\exists\ y,\ E.eq\ y\ x \wedge MapsTo\ x\ (f\ y\ e)\ (mapi\ f\ m).$
Parameter $mapi\_2 : \forall\ (elt\ elt'{:}\mathsf{Set})(m{:}\ t\ elt)(x{:}key)$
$(f{:}key{\rightarrow}elt{\rightarrow}elt'),\ In\ x\ (mapi\ f\ m) \rightarrow In\ x\ m.$

Specification of *map2*

Parameter $map2\_1 : \forall\ (elt\ elt'\ elt''{:}\mathsf{Set})(m{:}\ t\ elt)(m'{:}\ t\ elt')$
$(x{:}key)(f{:}option\ elt{\rightarrow}option\ elt'{\rightarrow}option\ elt''),$
$In\ x\ m \vee In\ x\ m' \rightarrow$
$find\ x\ (map2\ f\ m\ m') = f\ (find\ x\ m)\ (find\ x\ m').$

Parameter $map2\_2 : \forall\ (elt\ elt'\ elt''{:}\mathsf{Set})(m{:}\ t\ elt)(m'{:}\ t\ elt')$
$(x{:}key)(f{:}option\ elt{\rightarrow}option\ elt'{\rightarrow}option\ elt''),$
$In\ x\ (map2\ f\ m\ m') \rightarrow In\ x\ m \vee In\ x\ m'.$

End *S*.

Module Type *Sord*.

Declare Module *Data* : *OrderedType*.
Declare Module *MapS* : *S*.
Import *MapS*.

Definition $t := MapS.t\ Data.t.$

Parameter $eq : t \rightarrow t \rightarrow \mathsf{Prop}.$
Parameter $lt : t \rightarrow t \rightarrow \mathsf{Prop}.$

Axiom $eq\_refl : \forall\ m : t,\ eq\ m\ m.$
Axiom $eq\_sym : \forall\ m1\ m2 : t,\ eq\ m1\ m2 \rightarrow eq\ m2\ m1.$
Axiom $eq\_trans : \forall\ m1\ m2\ m3 : t,\ eq\ m1\ m2 \rightarrow eq\ m2\ m3 \rightarrow eq\ m1\ m3.$
Axiom $lt\_trans : \forall\ m1\ m2\ m3 : t,\ lt\ m1\ m2 \rightarrow lt\ m2\ m3 \rightarrow lt\ m1\ m3.$
Axiom $lt\_not\_eq : \forall\ m1\ m2 : t,\ lt\ m1\ m2 \rightarrow \neg\ eq\ m1\ m2.$

Definition *cmp e e'* := match *Data.compare e e'* with *EQ _* ⇒ *true* | *_* ⇒ *false* end.

Parameter *eq_1* : ∀ *m m'*, *Equal cmp m m'* → *eq m m'*.
Parameter *eq_2* : ∀ *m m'*, *eq m m'* → *Equal cmp m m'*.

Parameter *compare* : ∀ *m1 m2*, *Compare lt eq m1 m2*.
Total ordering between maps. The first argument (in Coq: Data.compare) is a total ordering used to compare data associated with equal keys in the two maps.

End *Sord*.

Chapter 208

Module Coq.FSets.FMapIntMap

Require Import *Bool*.
Require Import *NArith Ndigits Ndec Nnat*.
Require Import *Allmaps*.
Require Import *OrderedType*.
Require Import *OrderedTypeEx*.
Require Import *FMapInterface FMapList*.

208.1 An implementation of *FMapInterface.S* based on *IntMap*

Keys are of type *N*. The main functions are directly taken from *IntMap*. Since they have no exact counterpart in *IntMap*, functions *fold*, *map2* and *equal* are for now obtained by translation to sorted lists.

N is an ordered type, using not the usual order on numbers, but lexicographic ordering on bits (lower bit considered first).

Module *NUsualOrderedType* <: *UsualOrderedType*.
  Definition *t*:=*N*.
  Definition *eq*:=@*eq N*.
  Definition *eq_refl* := @*refl_equal t*.
  Definition *eq_sym* := @*sym_eq t*.
  Definition *eq_trans* := @*trans_eq t*.

  Definition *lt p q*:= *Nless p q* = *true*.

  Definition *lt_trans* := *Nless_trans*.

  Lemma *lt_not_eq* : ∀ *x y* : *t*, *lt x y* → ¬ *eq x y*.

  Definition *compare* : ∀ *x y* : *t*, *Compare lt eq x y*.

End *NUsualOrderedType*.

The module of maps over *N* keys based on *IntMap*

Module *MapIntMap* <: *S* with Module *E*:=*NUsualOrderedType*.

Module E:=$NUsualOrderedType$.
Module ME:=$OrderedTypeFacts(E)$.
Module PE:=$KeyOrderedType(E)$.

Definition $key := N$.

Definition $t := Map$.

Section A.
Variable A:Set.

Definition $empty : t\ A := M0\ A$.

Definition $is\_empty\ (m : t\ A) : bool :=$
$MapEmptyp\ \_\ (MapCanonicalize\ \_\ m)$.

Definition $find\ (x{:}key)(m{:}\ t\ A) : option\ A := MapGet\ \_\ m\ x$.

Definition $mem\ (x{:}key)(m{:}\ t\ A) : bool :=$
match $find\ x\ m$ with
$\mid Some\ \_ \Rightarrow true$
$\mid None \Rightarrow false$
end.

Definition $add\ (x{:}key)(v{:}A)(m{:}t\ A) : t\ A := MapPut\ \_\ m\ x\ v$.

Definition $remove\ (x{:}key)(m{:}t\ A) : t\ A := MapRemove\ \_\ m\ x$.

Definition $elements\ (m : t\ A) : list\ (N{\times}A) := alist\_of\_Map\ \_\ m$.

Definition $MapsTo\ (x{:}key)(v{:}A)(m{:}t\ A) := find\ x\ m = Some\ v$.

Definition $In\ (x{:}key)(m{:}t\ A) := \exists\ e{:}A,\ MapsTo\ x\ e\ m$.

Definition $Empty\ m := \forall\ (a : key)(e{:}A)\ , \neg\ MapsTo\ a\ e\ m$.

Definition $eq\_key\ (p\ p'{:}key{\times}A) := E.eq\ (fst\ p)\ (fst\ p')$.

Definition $eq\_key\_elt\ (p\ p'{:}key{\times}A) :=$
$E.eq\ (fst\ p)\ (fst\ p') \wedge (snd\ p) = (snd\ p')$.

Definition $lt\_key\ (p\ p'{:}key{\times}A) := E.lt\ (fst\ p)\ (fst\ p')$.

Lemma $Empty\_alt : \forall\ m,\ Empty\ m \leftrightarrow \forall\ a,\ find\ a\ m = None$.

Section $Spec$.
Variable $m\ m'\ m'' : t\ A$.
Variable $x\ y\ z : key$.
Variable $e\ e' : A$.

Lemma $MapsTo\_1 : E.eq\ x\ y \rightarrow MapsTo\ x\ e\ m \rightarrow MapsTo\ y\ e\ m$.

Lemma $find\_1 : MapsTo\ x\ e\ m \rightarrow find\ x\ m = Some\ e$.

Lemma $find\_2 : find\ x\ m = Some\ e \rightarrow MapsTo\ x\ e\ m$.

Lemma $empty\_1 : Empty\ empty$.

Lemma $is\_empty\_1 : Empty\ m \rightarrow is\_empty\ m = true$.

Lemma *is_empty_2* : *is_empty m = true → Empty m.*

Lemma *mem_1* : *In x m → mem x m = true.*

Lemma *mem_2* : ∀ *m x*, *mem x m = true → In x m.*

Lemma *add_1* : *E.eq x y → MapsTo y e (add x e m).*

Lemma *add_2* : ¬ *E.eq x y → MapsTo y e m → MapsTo y e (add x e' m).*

Lemma *add_3* : ¬ *E.eq x y → MapsTo y e (add x e' m) → MapsTo y e m.*

Lemma *remove_1* : *E.eq x y → ¬ In y (remove x m).*

Lemma *remove_2* : ¬ *E.eq x y → MapsTo y e m → MapsTo y e (remove x m).*

Lemma *remove_3* : *MapsTo y e (remove x m) → MapsTo y e m.*

Lemma *alist_sorted_sort* : ∀ *l*, *alist_sorted A l=true → sort lt_key l.*

Lemma *elements_3* : *sort lt_key (elements m).*

Lemma *elements_1* :
MapsTo x e m → InA eq_key_elt (x,e) (elements m).

Lemma *elements_2* :
InA eq_key_elt (x,e) (elements m) → MapsTo x e m.

Definition *Equal cmp m m'* :=
(∀ *k*, *In k m ↔ In k m'*) ∧
(∀ *k e e'*, *MapsTo k e m → MapsTo k e' m' → cmp e e' = true*).

unfortunately, the *MapFold* of *IntMap* isn't compatible with the FMap interface. We use a naive version for now :

Definition *fold* (*B*:Set)(*f*:*key → A → B → B*)(*m*:*t A*)(*i*:*B*) : *B* :=
fold_left (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

Lemma *fold_1* :
∀ (*B*:Set) (*i* : *B*) (*f* : *key → A → B → B*),
fold f m i = fold_left (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

End *Spec*.

Variable *B* : Set.

Fixpoint *mapi_aux* (*pf*:*N→N*)(*f* : *N → A → B*)(*m*:*t A*) { *struct m* }: *t B* :=
match *m* with
| *M0* ⇒ *M0* _
| *M1 x y* ⇒ *M1* _ *x* (*f* (*pf x*) *y*)
| *M2 m0 m1* ⇒ *M2* _ (*mapi_aux* (fun *n* ⇒ *pf* (*Ndouble n*)) *f m0*)
(*mapi_aux* (fun *n* ⇒ *pf* (*Ndouble_plus_one n*)) *f m1*)
end.

Definition *mapi* := *mapi_aux* (fun *n* ⇒ *n*).

Definition *map* (*f*:*A→B*) := *mapi* (fun _ ⇒ *f*).

End A.

Lemma $mapi\_aux\_1$: $\forall$ (elt elt':Set)(m: t elt)(pf:$N\to N$)(x:key)(e:elt)
(f:$key\to elt\to elt'$), $MapsTo$ x e m $\to$
$\exists$ y, $E.eq$ y x $\wedge$ $MapsTo$ x (f (pf y) e) ($mapi\_aux$ pf f m).

Lemma $mapi\_1$: $\forall$ (elt elt':Set)(m: t elt)(x:key)(e:elt)
(f:$key\to elt\to elt'$), $MapsTo$ x e m $\to$
$\exists$ y, $E.eq$ y x $\wedge$ $MapsTo$ x (f y e) ($mapi$ f m).

Lemma $mapi\_aux\_2$: $\forall$ (elt elt':Set)(m: t elt)(pf:$N\to N$)(x:key)
(f:$key\to elt\to elt'$), In x ($mapi\_aux$ pf f m) $\to$ In x m.

Lemma $mapi\_2$: $\forall$ (elt elt':Set)(m: t elt)(x:key)
(f:$key\to elt\to elt'$), In x ($mapi$ f m) $\to$ In x m.

Lemma map_1 : $\forall$ (elt elt':Set)(m: t elt)(x:key)(e:elt)(f:$elt\to elt'$),
$MapsTo$ x e m $\to$ $MapsTo$ x (f e) (map f m).

Lemma map_2 : $\forall$ (elt elt':Set)(m: t elt)(x:key)(f:$elt\to elt'$),
In x (map f m) $\to$ In x m.

Module L := $FMapList.Raw$ E.

Not exactly pretty nor perfect, but should suffice as a first naive implem. Anyway, map2 isn't in Ocaml...

Definition $anti\_elements$ (A:Set)(l:$list$ ($key\times A$)) := $L.fold$ (@add _) l ($empty$ _).

Definition $map2$ (A B C:Set)(f:$option$ $A\to option$ B $\to$ $option$ C)(m:t A)(m':t B) : t C :=
$anti\_elements$ ($L.map2$ f ($elements$ m) ($elements$ m')).

Lemma add_spec : $\forall$ (A:Set)(m:t A) x y e,
$find$ x (add y e m) = if $ME.eq\_dec$ x y then $Some$ e else $find$ x m.

Lemma $anti\_elements\_mapsto\_aux$: $\forall$ (A:Set)(l:$list$ ($key\times A$)) m k e,
$NoDupA$ (eq_key (A:=A)) l $\to$
($\forall$ x, $L.PX.In$ x l $\to$ In x m $\to$ $False$) $\to$
($MapsTo$ k e ($L.fold$ (@add _) l m) $\leftrightarrow$ $L.PX.MapsTo$ k e l $\vee$ $MapsTo$ k e m).

Lemma $anti\_elements\_mapsto$: $\forall$ (A:Set) l k e, $NoDupA$ (eq_key (A:=A)) l $\to$
($MapsTo$ k e ($anti\_elements$ l) $\leftrightarrow$ $L.PX.MapsTo$ k e l).

Lemma $find\_anti\_elements$: $\forall$ (A:Set)(l: $list$ ($key\times A$)) x, $sort$ (@lt_key _) l $\to$
$find$ x ($anti\_elements$ l) = $L.find$ x l.

Lemma $find\_elements$: $\forall$ (A:Set)(m: t A) x,
$L.find$ x ($elements$ m) = $find$ x m.

Lemma $elements\_in$: $\forall$ (A:Set)(s:t A) x, $L.PX.In$ x ($elements$ s) $\leftrightarrow$ In x s.

Lemma $map2\_1$: $\forall$ (A B C:Set)(m: t A)(m': t B)(x:key)
(f:$option$ $A\to option$ B $\to option$ C),
In x m $\vee$ In x m' $\to$ $find$ x ($map2$ f m m') = f ($find$ x m) ($find$ x m').

Lemma *map2_2* : ∀ (*A B C*: Set)(*m*: *t A*)(*m'*: *t B*)(*x*:*key*)
(*f*:*option A*→*option B* →*option C*),
In x (*map2 f m m'*) → *In x m* ∨ *In x m'*.

same trick for *equal*

Definition *equal* (*A*:Set)(*cmp*:*A* → *A* → *bool*)(*m m'* : *t A*) : *bool* :=
L.equal cmp (*elements m*) (*elements m'*).

Lemma *equal_1* :
∀ (*A*:Set)(*m*: *t A*)(*m'*: *t A*)(*cmp*: *A* → *A* → *bool*),
Equal cmp m m' → *equal cmp m m'* = *true*.

Lemma *equal_2* :
∀ (*A*:Set)(*m*: *t A*)(*m'*: *t A*)(*cmp*: *A* → *A* → *bool*),
equal cmp m m' = *true* → *Equal cmp m m'*.

End *MapIntMap*.

Chapter 209

Module Coq.FSets.FMapList

209.1 Finite map library

This file proposes an implementation of the non-dependant interface *FMapInterface.S* using lists of pairs ordered (increasing) with respect to left projection.

Require Import *FSetInterface*.
Require Import *FMapInterface*.

Module *Raw* (*X*:*OrderedType*).

Module *E* := *X*.
Module *MX* := *OrderedTypeFacts X*.
Module *PX* := *KeyOrderedType X*.
Import *MX*.
Import *PX*.

Definition *key* := *X.t*.
Definition *t* (*elt*:Set) := *list* (*X.t* × *elt*).

Section *Elt*.
Variable *elt* : Set.

Notation *eqk* := (*eqk* (*elt*:=*elt*)).
Notation *eqke* := (*eqke* (*elt*:=*elt*)).
Notation *ltk* := (*ltk* (*elt*:=*elt*)).
Notation *MapsTo* := (*MapsTo* (*elt*:=*elt*)).
Notation *In* := (*In* (*elt*:=*elt*)).
Notation *Sort* := (*sort ltk*).
Notation *Inf* := (*lelistA* (*ltk*)).

209.2 *empty*

Definition *empty* : *t elt* := *nil*.

Definition $Empty\ m := \forall\ (a : key)(e{:}elt)\ ,\ \neg\ MapsTo\ a\ e\ m.$

Lemma $empty\_1\ :\ Empty\ empty.$
Hint $Resolve\ empty\_1.$

Lemma $empty\_sorted\ :\ Sort\ empty.$

209.3 *is_empty*

Definition $is\_empty\ (l\ :\ t\ elt)\ :\ bool\ :=$ if l then $true$ else $false.$

Lemma $is\_empty\_1\ :\forall\ m,\ Empty\ m \rightarrow is\_empty\ m = true.$

Lemma $is\_empty\_2\ :\ \forall\ m,\ is\_empty\ m = true \rightarrow Empty\ m.$

209.4 *mem*

Lemma $mem\_1\ :\ \forall\ m\ (Hm{:}Sort\ m)\ x,\ In\ x\ m \rightarrow mem\ x\ m = true.$

Lemma $mem\_2\ :\ \forall\ m\ (Hm{:}Sort\ m)\ x,\ mem\ x\ m = true \rightarrow In\ x\ m.$

209.5 *find*

Lemma $find\_2\ :\ \forall\ m\ x\ e,\ find\ x\ m = Some\ e \rightarrow MapsTo\ x\ e\ m.$

Lemma $find\_1\ :\ \forall\ m\ (Hm{:}Sort\ m)\ x\ e,\ MapsTo\ x\ e\ m \rightarrow find\ x\ m = Some\ e.$

209.6 *add*

Lemma $add\_1\ :\ \forall\ m\ x\ y\ e,\ X.eq\ x\ y \rightarrow MapsTo\ y\ e\ (add\ x\ e\ m).$

Lemma $add\_2\ :\ \forall\ m\ x\ y\ e\ e',$
$\neg\ X.eq\ x\ y \rightarrow MapsTo\ y\ e\ m \rightarrow MapsTo\ y\ e\ (add\ x\ e'\ m).$

Lemma $add\_3\ :\ \forall\ m\ x\ y\ e\ e',$
$\neg\ X.eq\ x\ y \rightarrow MapsTo\ y\ e\ (add\ x\ e'\ m) \rightarrow MapsTo\ y\ e\ m.$

Lemma $add\_Inf\ :\ \forall\ (m{:}t\ elt)(x\ x'{:}key)(e\ e'{:}elt),$
$Inf\ (x',e')\ m \rightarrow ltk\ (x',e')\ (x,e) \rightarrow Inf\ (x',e')\ (add\ x\ e\ m).$
Hint $Resolve\ add\_Inf.$

Lemma $add\_sorted\ :\ \forall\ m\ (Hm{:}Sort\ m)\ x\ e,\ Sort\ (add\ x\ e\ m).$

209.7 *remove*

Lemma *remove_1* : ∀ *m* (*Hm*:*Sort m*) *x y*, *X.eq x y* → ¬ *In y* (*remove x m*).

Lemma *remove_2* : ∀ *m* (*Hm*:*Sort m*) *x y e*,
¬ *X.eq x y* → *MapsTo y e m* → *MapsTo y e* (*remove x m*).

Lemma *remove_3* : ∀ *m* (*Hm*:*Sort m*) *x y e*,
MapsTo y e (*remove x m*) → *MapsTo y e m*.

Lemma *remove_Inf* : ∀ (*m*:*t elt*)(*Hm* : *Sort m*)(*x x'*:*key*)(*e'*:*elt*),
Inf (*x'*,*e'*) *m* → *Inf* (*x'*,*e'*) (*remove x m*).
Hint *Resolve remove_Inf*.

Lemma *remove_sorted* : ∀ *m* (*Hm*:*Sort m*) *x*, *Sort* (*remove x m*).

209.8 *elements*

Definition *elements* (*m*: *t elt*) := *m*.

Lemma *elements_1* : ∀ *m x e*,
MapsTo x e m → *InA eqke* (*x*,*e*) (*elements m*).

Lemma *elements_2* : ∀ *m x e*,
InA eqke (*x*,*e*) (*elements m*) → *MapsTo x e m*.

Lemma *elements_3* : ∀ *m* (*Hm*:*Sort m*), *sort ltk* (*elements m*).

209.9 *fold*

Lemma *fold_1* : ∀ *m* (*A*:Set)(*i*:*A*)(*f*:*key*→*elt*→*A*→*A*),
fold f m i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

209.10 *equal*

Definition *Equal cmp m m'* :=
(∀ *k*, *In k m* ↔ *In k m'*) ∧
(∀ *k e e'*, *MapsTo k e m* → *MapsTo k e' m'* → *cmp e e'* = *true*).

Lemma *equal_1* : ∀ *m* (*Hm*:*Sort m*) *m'* (*Hm'*: *Sort m'*) *cmp*,
Equal cmp m m' → *equal cmp m m'* = *true*.

Lemma *equal_2* : ∀ *m* (*Hm*:*Sort m*) *m'* (*Hm*:*Sort m'*) *cmp*,
equal cmp m m' = *true* → *Equal cmp m m'*.

This lemma isn't part of the spec of *Equal*, but is used in *FMapAVL*

Lemma *equal_cons* : ∀ *cmp l1 l2 x y*, *Sort* (*x*::*l1*) → *Sort* (*y*::*l2*) →
eqk x y → *cmp* (*snd x*) (*snd y*) = *true* →
(*Equal cmp l1 l2* ↔ *Equal cmp* (*x* :: *l1*) (*y* :: *l2*)).

Variable *elt'*:Set.

209.11 *map* and *mapi*

Fixpoint *map* (*f*:*elt* → *elt'*) (*m*:*t elt*) {*struct m*} : *t elt'* :=
match *m* with
| *nil* ⇒ *nil*
| (*k*,*e*)::*m'* ⇒ (*k*,*f e*) :: *map f m'*
end.

Fixpoint *mapi* (*f*: *key* → *elt* → *elt'*) (*m*:*t elt*) {*struct m*} : *t elt'* :=
match *m* with
| *nil* ⇒ *nil*
| (*k*,*e*)::*m'* ⇒ (*k*,*f k e*) :: *mapi f m'*
end.

End *Elt*.
Section *Elt2*.

Variable *elt elt'* : Set.

Specification of *map*

Lemma *map_1* : ∀ (*m*:*t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*elt*→*elt'*),
MapsTo x e m → *MapsTo x* (*f e*) (*map f m*).

Lemma *map_2* : ∀ (*m*:*t elt*)(*x*:*key*)(*f*:*elt*→*elt'*),
In x (*map f m*) → *In x m*.

Lemma *map_lelistA* : ∀ (*m*: *t elt*)(*x*:*key*)(*e*:*elt*)(*e'*:*elt'*)(*f*:*elt*→*elt'*),
lelistA (@*ltk elt*) (*x*,*e*) *m* →
lelistA (@*ltk elt'*) (*x*,*e'*) (*map f m*).

Hint *Resolve map_lelistA*.

Lemma *map_sorted* : ∀ (*m*: *t elt*)(*Hm* : *sort* (@*ltk elt*) *m*)(*f*:*elt* → *elt'*),
sort (@*ltk elt'*) (*map f m*).

Specification of *mapi*

Lemma *mapi_1* : ∀ (*m*:*t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*key*→*elt*→*elt'*),

MapsTo x e m →
∃ y, X.eq y x ∧ MapsTo x (f y e) (mapi f m).

Lemma *mapi_2* : ∀ *(m:t elt)(x:key)(f:key→elt→elt'),*
In x (mapi f m) → In x m.

Lemma *mapi_lelistA* : ∀ *(m: t elt)(x:key)(e:elt)(f:key→elt→elt'),*
lelistA (@ltk elt) (x,e) m →
lelistA (@ltk elt') (x,f x e) (mapi f m).

Hint *Resolve mapi_lelistA.*

Lemma *mapi_sorted* : ∀ *m (Hm : sort (@ltk elt) m)(f: key →elt → elt'),*
sort (@ltk elt') (mapi f m).

End *Elt2.*
Section *Elt3.*

209.12 *map2*

Variable *elt elt' elt"* : Set.
Variable *f : option elt → option elt' → option elt".*

Definition *option_cons (A*:Set*)(k:key)(o:option A)(l:list (key×A)) :=*
match *o* with
| *Some e ⇒ (k,e)::l*
| *None ⇒ l*
end.

Fixpoint *map2_l (m : t elt) : t elt" :=*
match *m* with
| *nil ⇒ nil*
| *(k,e)::l ⇒ option_cons k (f (Some e) None) (map2_l l)*
end.

Fixpoint *map2_r (m' : t elt') : t elt" :=*
match *m'* with
| *nil ⇒ nil*
| *(k,e')::l' ⇒ option_cons k (f None (Some e')) (map2_r l')*
end.

Fixpoint *map2 (m : t elt) : t elt' → t elt" :=*
match *m* with
| *nil ⇒ map2_r*
| *(k,e) :: l ⇒*
fix map2_aux (m' : t elt') : t elt" :=
match *m'* with
| *nil ⇒ map2_l m*
| *(k',e') :: l' ⇒*

```
            match X.compare k k' with
             | LT _ ⇒ option_cons k (f (Some e) None) (map2 l m')
             | EQ _ ⇒ option_cons k (f (Some e) (Some e')) (map2 l l')
             | GT _ ⇒ option_cons k' (f None (Some e')) (map2_aux l')
            end
        end
  end.

Notation oee' := (option elt × option elt')%type.

Fixpoint combine (m : t elt) : t elt' → t oee' :=
  match m with
   | nil ⇒ map (fun e' ⇒ (None,Some e'))
   | (k,e) :: l ⇒
      fix combine_aux (m':t elt') : list (key × oee') :=
        match m' with
         | nil ⇒ map (fun e ⇒ (Some e,None)) m
         | (k',e') :: l' ⇒
            match X.compare k k' with
             | LT _ ⇒ (k,(Some e, None))::combine l m'
             | EQ _ ⇒ (k,(Some e, Some e'))::combine l l'
             | GT _ ⇒ (k',(None,Some e'))::combine_aux l'
            end
        end
  end.

Definition fold_right_pair (A B C:Set)(f: A→B→C→C)(l:list (A×B))(i:C) :=
  List.fold_right (fun p ⇒ f (fst p) (snd p)) i l.

Definition map2_alt m m' :=
  let m0 : t oee' := combine m m' in
  let m1 : t (option elt") := map (fun p ⇒ f (fst p) (snd p)) m0 in
  fold_right_pair (option_cons (A:=elt")) m1 nil.

Lemma map2_alt_equiv : ∀ m m', map2_alt m m' = map2 m m'.

Lemma combine_lelistA :
  ∀ m m' (x:key)(e:elt)(e':elt')(e":oee'),
  lelistA (@ltk elt) (x,e) m →
  lelistA (@ltk elt') (x,e') m' →
  lelistA (@ltk oee') (x,e") (combine m m').
Hint Resolve combine_lelistA.

Lemma combine_sorted :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m'),
  sort (@ltk oee') (combine m m').

Lemma map2_sorted :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m'),
  sort (@ltk elt") (map2 m m').
```

```
Definition at_least_one (o:option elt)(o':option elt') :=
  match o, o' with
   | None, None ⇒ None
   | _, _ ⇒ Some (o,o')
  end.

Lemma combine_1 :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m') (x:key),
  find x (combine m m') = at_least_one (find x m) (find x m').

Definition at_least_one_then_f (o:option elt)(o':option elt') :=
  match o, o' with
   | None, None ⇒ None
   | _, _ ⇒ f o o'
  end.

Lemma map2_0 :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m') (x:key),
  find x (map2 m m') = at_least_one_then_f (find x m) (find x m').
```

Specification of *map2*

```
Lemma map2_1 :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m')(x:key),
  In x m ∨ In x m' →
  find x (map2 m m') = f (find x m) (find x m').

Lemma map2_2 :
  ∀ m (Hm : sort (@ltk elt) m) m' (Hm' : sort (@ltk elt') m')(x:key),
  In x (map2 m m') → In x m ∨ In x m'.

End Elt3.
End Raw.

Module Make (X: OrderedType) <: S with Module E := X.
Module Raw := Raw X.
Module E := X.

Definition key := E.t.

Record slist (elt:Set) : Set :=
  {this :> Raw.t elt; sorted : sort (@Raw.PX.ltk elt) this}.
Definition t (elt:Set) : Set := slist elt.

Section Elt.
 Variable elt elt' elt'':Set.

 Implicit Types m : t elt.
 Implicit Types x y : key.
 Implicit Types e : elt.

 Definition empty : t elt := Build_slist (Raw.empty_sorted elt).
 Definition is_empty m : bool := Raw.is_empty m.(this).
```

Definition *add x e m : t elt := Build_slist (Raw.add_sorted m.(sorted) x e).*
Definition *find x m : option elt := Raw.find x m.(this).*
Definition *remove x m : t elt := Build_slist (Raw.remove_sorted m.(sorted) x).*
Definition *mem x m : bool := Raw.mem x m.(this).*
Definition *map f m : t elt' := Build_slist (Raw.map_sorted m.(sorted) f).*
Definition *mapi (f:key→elt→elt') m : t elt' := Build_slist (Raw.mapi_sorted m.(sorted) f).*
Definition *map2 f m (m':t elt') : t elt" :=*
Build_slist (Raw.map2_sorted f m.(sorted) m'.(sorted)).
Definition *elements m : list (key×elt) := @Raw.elements elt m.(this).*
Definition *fold (A:Set)(f:key→elt→A→A) m (i:A) : A := @Raw.fold elt A f m.(this) i.*
Definition *equal cmp m m' : bool := @Raw.equal elt cmp m.(this) m'.(this).*

Definition *MapsTo x e m* : Prop := *Raw.PX.MapsTo x e m.(this).*
Definition *In x m* : Prop := *Raw.PX.In x m.(this).*
Definition *Empty m* : Prop := *Raw.Empty m.(this).*
Definition *Equal cmp m m'* : Prop := *@Raw.Equal elt cmp m.(this) m'.(this).*

Definition *eq_key* : *(key×elt) → (key×elt) →* Prop := *@Raw.PX.eqk elt.*
Definition *eq_key_elt* : *(key×elt) → (key×elt) →* Prop:= *@Raw.PX.eqke elt.*
Definition *lt_key* : *(key×elt) → (key×elt) →* Prop := *@Raw.PX.ltk elt.*

Lemma *MapsTo_1* : *∀ m x y e, E.eq x y → MapsTo x e m → MapsTo y e m.*

Lemma *mem_1* : *∀ m x, In x m → mem x m = true.*
Lemma *mem_2* : *∀ m x, mem x m = true → In x m.*

Lemma *empty_1* : *Empty empty.*

Lemma *is_empty_1* : *∀ m, Empty m → is_empty m = true.*
Lemma *is_empty_2* : *∀ m, is_empty m = true → Empty m.*

Lemma *add_1* : *∀ m x y e, E.eq x y → MapsTo y e (add x e m).*
Lemma *add_2* : *∀ m x y e e', ¬ E.eq x y → MapsTo y e m → MapsTo y e (add x e' m).*
Lemma *add_3* : *∀ m x y e e', ¬ E.eq x y → MapsTo y e (add x e' m) → MapsTo y e m.*

Lemma *remove_1* : *∀ m x y, E.eq x y → ¬ In y (remove x m).*
Lemma *remove_2* : *∀ m x y e, ¬ E.eq x y → MapsTo y e m → MapsTo y e (remove x m).*
Lemma *remove_3* : *∀ m x y e, MapsTo y e (remove x m) → MapsTo y e m.*

Lemma *find_1* : *∀ m x e, MapsTo x e m → find x m = Some e.*
Lemma *find_2* : *∀ m x e, find x m = Some e → MapsTo x e m.*

Lemma *elements_1* : *∀ m x e, MapsTo x e m → InA eq_key_elt (x,e) (elements m).*
Lemma *elements_2* : *∀ m x e, InA eq_key_elt (x,e) (elements m) → MapsTo x e m.*
Lemma *elements_3* : *∀ m, sort lt_key (elements m).*

Lemma *fold_1* : *∀ m (A : Set) (i : A) (f : key → elt → A → A),*
fold f m i = fold_left (fun *a p ⇒ f (fst p) (snd p) a) (elements m) i.*

Lemma *equal_1* : *∀ m m' cmp, Equal cmp m m' → equal cmp m m' = true.*
Lemma *equal_2* : *∀ m m' cmp, equal cmp m m' = true → Equal cmp m m'.*

End *Elt.*

```
Lemma map_1 : ∀ (elt elt':Set)(m: t elt)(x:key)(e:elt)(f:elt→elt'),
        MapsTo x e m → MapsTo x (f e) (map f m).
Lemma map_2 : ∀ (elt elt':Set)(m: t elt)(x:key)(f:elt→elt'),
        In x (map f m) → In x m.

Lemma mapi_1 : ∀ (elt elt':Set)(m: t elt)(x:key)(e:elt)
        (f:key→elt→elt'), MapsTo x e m →
        ∃ y, E.eq y x ∧ MapsTo x (f y e) (mapi f m).
Lemma mapi_2 : ∀ (elt elt':Set)(m: t elt)(x:key)
        (f:key→elt→elt'), In x (mapi f m) → In x m.

Lemma map2_1 : ∀ (elt elt' elt":Set)(m: t elt)(m': t elt')
        (x:key)(f:option elt→option elt'→option elt"),
        In x m ∨ In x m' →
        find x (map2 f m m') = f (find x m) (find x m').
Lemma map2_2 : ∀ (elt elt' elt":Set)(m: t elt)(m': t elt')
        (x:key)(f:option elt→option elt'→option elt"),
        In x (map2 f m m') → In x m ∨ In x m'.

End Make.

Module Make_ord (X: OrderedType)(D : OrderedType) <:
    Sord with Module Data := D
          with Module MapS.E := X.

Module Data := D.
Module MapS := Make(X).
Import MapS.

Module MD := OrderedTypeFacts(D).
Import MD.

Definition t := MapS.t D.t.

Definition cmp e e' := match D.compare e e' with EQ _ ⇒ true | _ ⇒ false end.

Fixpoint eq_list (m m' : list (X.t × D.t)) { struct m } : Prop :=
  match m, m' with
   | nil, nil ⇒ True
   | (x,e)::l, (x',e')::l' ⇒
       match X.compare x x' with
        | EQ _ ⇒ D.eq e e' ∧ eq_list l l'
        | _ ⇒ False
       end
   | _, _ ⇒ False
  end.

Definition eq m m' := eq_list m.(this) m'.(this).

Fixpoint lt_list (m m' : list (X.t × D.t)) {struct m} : Prop :=
  match m, m' with
   | nil, nil ⇒ False
```

```
    | nil, _ ⇒ True
    | _, nil ⇒ False
    | (x,e)::l, (x',e')::l' ⇒
      match X.compare x x' with
        | LT _ ⇒ True
        | GT _ ⇒ False
        | EQ _ ⇒ D.lt e e' ∨ (D.eq e e' ∧ lt_list l l')
      end
  end.
```

Definition *lt m m'* := *lt_list m.(this) m'.(this)*.

Lemma *eq_equal* : ∀ *m m'*, *eq m m'* ↔ *equal cmp m m'* = *true*.

Lemma *eq_1* : ∀ *m m'*, *Equal cmp m m'* → *eq m m'*.

Lemma *eq_2* : ∀ *m m'*, *eq m m'* → *Equal cmp m m'*.

Lemma *eq_refl* : ∀ *m* : *t*, *eq m m*.

Lemma *eq_sym* : ∀ *m1 m2* : *t*, *eq m1 m2* → *eq m2 m1*.

Lemma *eq_trans* : ∀ *m1 m2 m3* : *t*, *eq m1 m2* → *eq m2 m3* → *eq m1 m3*.

Lemma *lt_trans* : ∀ *m1 m2 m3* : *t*, *lt m1 m2* → *lt m2 m3* → *lt m1 m3*.

Lemma *lt_not_eq* : ∀ *m1 m2* : *t*, *lt m1 m2* → ¬ *eq m1 m2*.

Ltac *cmp_solve* := *unfold eq*, *lt*; *simpl*; *try Raw.MX.elim_comp*; *auto*.

Definition *compare* : ∀ *m1 m2*, *Compare lt eq m1 m2*.

End *Make_ord*.

Chapter 210

Module Coq.FSets.FMapPositive

```
Require Import Bool.
Require Import ZArith.
Require Import OrderedType.
Require Import FMapInterface.

Open Local Scope positive_scope.
```

210.1 An implementation of *FMapInterface.S* for positive keys.

This file is an adaptation to the *FMap* framework of a work by Xavier Leroy and Sandrine Blazy (used for building certified compilers). Keys are of type *positive*, and maps are binary trees: the sequence of binary digits of a positive number corresponds to a path in such a tree. This is quite similar to the *IntMap* library, except that no path compression is implemented, and that the current file is simple enough to be self-contained.

Even if *positive* can be seen as an ordered type with respect to the usual order (see *OrderedTypeEx*), we use here a lexicographic order over bits, which is more natural here (lower bits are considered first).

```
Module PositiveOrderedTypeBits <: OrderedType.
  Definition t:=positive.
  Definition eq:=@eq positive.

  Fixpoint bits_lt (p q:positive) { struct p } : Prop :=
   match p, q with
   | xH, xI _ ⇒ True
   | xH, _ ⇒ False
   | xO p, xO q ⇒ bits_lt p q
   | xO _, _ ⇒ True
   | xI p, xI q ⇒ bits_lt p q
   | xI _, _ ⇒ False
   end.

  Definition lt:=bits_lt.
```

Lemma $eq\_refl : \forall\ x : t,\ eq\ x\ x.$

Lemma $eq\_sym : \forall\ x\ y : t,\ eq\ x\ y \rightarrow eq\ y\ x.$

Lemma $eq\_trans : \forall\ x\ y\ z : t,\ eq\ x\ y \rightarrow eq\ y\ z \rightarrow eq\ x\ z.$

Lemma $bits\_lt\_trans : \forall\ x\ y\ z : positive,\ bits\_lt\ x\ y \rightarrow bits\_lt\ y\ z \rightarrow bits\_lt\ x\ z.$

Lemma $lt\_trans : \forall\ x\ y\ z : t,\ lt\ x\ y \rightarrow lt\ y\ z \rightarrow lt\ x\ z.$

Lemma $bits\_lt\_antirefl : \forall\ x : positive,\ \neg\ bits\_lt\ x\ x.$

Lemma $lt\_not\_eq : \forall\ x\ y : t,\ lt\ x\ y \rightarrow \neg\ eq\ x\ y.$

Definition $compare : \forall\ x\ y : t,\ Compare\ lt\ eq\ x\ y.$

End $PositiveOrderedTypeBits.$

Other positive stuff

Lemma $peq\_dec\ (x\ y{:}\ positive){:}\ \{x = y\} + \{x \neq y\}.$

```
Fixpoint append (i j : positive) {struct i} : positive :=
  match i with
    | xH ⇒ j
    | xI ii ⇒ xI (append ii j)
    | xO ii ⇒ xO (append ii j)
  end.
```

```
Lemma append_assoc_0 :
  ∀ (i j : positive), append i (xO j) = append (append i (xO xH)) j.
```

```
Lemma append_assoc_1 :
  ∀ (i j : positive), append i (xI j) = append (append i (xI xH)) j.
```

Lemma $append\_neutral\_r : \forall\ (i : positive),\ append\ i\ xH = i.$

Lemma $append\_neutral\_l : \forall\ (i : positive),\ append\ xH\ i = i.$

The module of maps over positive keys

Module $PositiveMap <: S$ with Module $E{:=}PositiveOrderedTypeBits.$

Module $E{:=}PositiveOrderedTypeBits.$

Definition $key := positive.$

```
Inductive tree (A : Set) : Set :=
  | Leaf : tree A
  | Node : tree A → option A → tree A → tree A.
```

Definition $t := tree.$

Section $A.$
Variable A:Set.

Implicit $Arguments\ Leaf$ [A].

Definition $empty : t\ A := Leaf.$

```
Fixpoint is_empty (m : t A) {struct m} : bool :=
 match m with
  | Leaf ⇒ true
  | Node l None r ⇒ (is_empty l) && (is_empty r)
  | _ ⇒ false
 end.

Fixpoint find (i : positive) (m : t A) {struct i} : option A :=
  match m with
  | Leaf ⇒ None
  | Node l o r ⇒
      match i with
      | xH ⇒ o
      | xO ii ⇒ find ii l
      | xI ii ⇒ find ii r
      end
  end.

Fixpoint mem (i : positive) (m : t A) {struct i} : bool :=
  match m with
  | Leaf ⇒ false
  | Node l o r ⇒
      match i with
      | xH ⇒ match o with None ⇒ false | _ ⇒ true end
      | xO ii ⇒ mem ii l
      | xI ii ⇒ mem ii r
      end
  end.

Fixpoint add (i : positive) (v : A) (m : t A) {struct i} : t A :=
  match m with
  | Leaf ⇒
      match i with
      | xH ⇒ Node Leaf (Some v) Leaf
      | xO ii ⇒ Node (add ii v Leaf) None Leaf
      | xI ii ⇒ Node Leaf None (add ii v Leaf)
      end
  | Node l o r ⇒
      match i with
      | xH ⇒ Node l (Some v) r
      | xO ii ⇒ Node (add ii v l) o r
      | xI ii ⇒ Node l o (add ii v r)
      end
  end.

Fixpoint remove (i : positive) (m : t A) {struct i} : t A :=
  match i with
  | xH ⇒
```

```
        match m with
        | Leaf ⇒ Leaf
        | Node Leaf o Leaf ⇒ Leaf
        | Node l o r ⇒ Node l None r
        end
    | xO ii ⇒
        match m with
        | Leaf ⇒ Leaf
        | Node l None Leaf ⇒
            match remove ii l with
            | Leaf ⇒ Leaf
            | mm ⇒ Node mm None Leaf
            end
        | Node l o r ⇒ Node (remove ii l) o r
        end
    | xI ii ⇒
        match m with
        | Leaf ⇒ Leaf
        | Node Leaf None r ⇒
            match remove ii r with
            | Leaf ⇒ Leaf
            | mm ⇒ Node Leaf None mm
            end
        | Node l o r ⇒ Node l o (remove ii r)
        end
    end.
```

elements

```
    Fixpoint xelements (m : t A) (i : positive) {struct m}
               : list (positive × A) :=
      match m with
      | Leaf ⇒ nil
      | Node l None r ⇒
          (xelements l (append i (xO xH))) ++ (xelements r (append i (xI xH)))
      | Node l (Some x) r ⇒
          (xelements l (append i (xO xH)))
          ++ ((i, x) :: xelements r (append i (xI xH)))
      end.

  Definition elements (m : t A) := xelements m xH.

  Section CompcertSpec.

  Theorem gempty:
    ∀ (i: positive), find i empty = None.

  Theorem gss:
    ∀ (i: positive) (x: A) (m: t A), find i (add i x m) = Some x.
```

Lemma *gleaf* : ∀ (*i* : *positive*), *find i* (*Leaf* : *t A*) = *None*.

Theorem *gso*:
∀ (*i j*: *positive*) (*x*: *A*) (*m*: *t A*),
i ≠ *j* → *find i* (*add j x m*) = *find i m*.

Lemma *rleaf* : ∀ (*i* : *positive*), *remove i* (*Leaf* : *t A*) = *Leaf*.

Theorem *grs*:
∀ (*i*: *positive*) (*m*: *t A*), *find i* (*remove i m*) = *None*.

Theorem *gro*:
∀ (*i j*: *positive*) (*m*: *t A*),
i ≠ *j* → *find i* (*remove j m*) = *find i m*.

Lemma *xelements_correct*:
∀ (*m*: *t A*) (*i j* : *positive*) (*v*: *A*),
find i m = *Some v* → *List.In* (*append j i*, *v*) (*xelements m j*).

Theorem *elements_correct*:
∀ (*m*: *t A*) (*i*: *positive*) (*v*: *A*),
find i m = *Some v* → *List.In* (*i*, *v*) (*elements m*).

Fixpoint *xfind* (*i j* : *positive*) (*m* : *t A*) {*struct j*} : *option A* :=
match *i*, *j* with
| _, *xH* ⇒ *find i m*
| *xO ii*, *xO jj* ⇒ *xfind ii jj m*
| *xI ii*, *xI jj* ⇒ *xfind ii jj m*
| _, _ ⇒ *None*
end.

Lemma *xfind_left* :
∀ (*j i* : *positive*) (*m1 m2* : *t A*) (*o* : *option A*) (*v* : *A*),
xfind i (*append j* (*xO xH*)) *m1* = *Some v* → *xfind i j* (*Node m1 o m2*) = *Some v*.

Lemma *xelements_ii* :
∀ (*m*: *t A*) (*i j* : *positive*) (*v*: *A*),
List.In (*xI i*, *v*) (*xelements m* (*xI j*)) → *List.In* (*i*, *v*) (*xelements m j*).

Lemma *xelements_io* :
∀ (*m*: *t A*) (*i j* : *positive*) (*v*: *A*),
¬*List.In* (*xI i*, *v*) (*xelements m* (*xO j*)).

Lemma *xelements_oo* :
∀ (*m*: *t A*) (*i j* : *positive*) (*v*: *A*),
List.In (*xO i*, *v*) (*xelements m* (*xO j*)) → *List.In* (*i*, *v*) (*xelements m j*).

Lemma *xelements_oi* :
∀ (*m*: *t A*) (*i j* : *positive*) (*v*: *A*),
¬*List.In* (*xO i*, *v*) (*xelements m* (*xI j*)).

Lemma *xelements_ih* :
∀ (*m1 m2*: *t A*) (*o*: *option A*) (*i* : *positive*) (*v*: *A*),

List.In (*xI i*, *v*) (*xelements* (*Node m1 o m2*) *xH*) → *List.In* (*i*, *v*) (*xelements m2 xH*).

Lemma *xelements_oh* :
∀ (*m1 m2*: *t A*) (*o*: *option A*) (*i* : *positive*) (*v*: *A*),
List.In (*xO i*, *v*) (*xelements* (*Node m1 o m2*) *xH*) → *List.In* (*i*, *v*) (*xelements m1 xH*).

Lemma *xelements_hi* :
∀ (*m*: *t A*) (*i* : *positive*) (*v*: *A*),
¬*List.In* (*xH*, *v*) (*xelements m* (*xI i*)).

Lemma *xelements_ho* :
∀ (*m*: *t A*) (*i* : *positive*) (*v*: *A*),
¬*List.In* (*xH*, *v*) (*xelements m* (*xO i*)).

Lemma *find_xfind_h* :
∀ (*m*: *t A*) (*i*: *positive*), *find i m* = *xfind i xH m*.

Lemma *xelements_complete*:
∀ (*i j* : *positive*) (*m*: *t A*) (*v*: *A*),
List.In (*i*, *v*) (*xelements m j*) → *xfind i j m* = *Some v*.

Theorem *elements_complete*:
∀ (*m*: *t A*) (*i*: *positive*) (*v*: *A*),
List.In (*i*, *v*) (*elements m*) → *find i m* = *Some v*.

End *CompcertSpec*.

Definition *MapsTo* (*i*:*positive*)(*v*:*A*)(*m*:*t A*) := *find i m* = *Some v*.

Definition *In* (*i*:*positive*)(*m*:*t A*) := ∃ *e*:*A*, *MapsTo i e m*.

Definition *Empty m* := ∀ (*a* : *positive*)(*e*:*A*) , ¬ *MapsTo a e m*.

Definition *eq_key* (*p p'*:*positive*×*A*) := *E.eq* (*fst p*) (*fst p'*).

Definition *eq_key_elt* (*p p'*:*positive*×*A*) :=
E.eq (*fst p*) (*fst p'*) ∧ (*snd p*) = (*snd p'*).

Definition *lt_key* (*p p'*:*positive*×*A*) := *E.lt* (*fst p*) (*fst p'*).

Lemma *mem_find* :
∀ *m x*, *mem x m* = match *find x m* with *None* ⇒ *false* | _ ⇒ *true* end.

Lemma *Empty_alt* : ∀ *m*, *Empty m* ↔ ∀ *a*, *find a m* = *None*.

Lemma *Empty_Node* : ∀ *l o r*, *Empty* (*Node l o r*) ↔ *o*=*None* ∧ *Empty l* ∧ *Empty r*.

Section *FMapSpec*.

Lemma *mem_1* : ∀ *m x*, *In x m* → *mem x m* = *true*.

Lemma *mem_2* : ∀ *m x*, *mem x m* = *true* → *In x m*.

Variable *m m' m"* : *t A*.
Variable *x y z* : *key*.
Variable *e e'* : *A*.

Lemma *MapsTo_1* : *E.eq x y* → *MapsTo x e m* → *MapsTo y e m*.

```
Lemma find_1 : MapsTo x e m → find x m = Some e.

Lemma find_2 : find x m = Some e → MapsTo x e m.

Lemma empty_1 : Empty empty.

Lemma is_empty_1 : Empty m → is_empty m = true.

Lemma is_empty_2 : is_empty m = true → Empty m.

Lemma add_1 : E.eq x y → MapsTo y e (add x e m).

Lemma add_2 : ¬ E.eq x y → MapsTo y e m → MapsTo y e (add x e' m).

Lemma add_3 : ¬ E.eq x y → MapsTo y e (add x e' m) → MapsTo y e m.

Lemma remove_1 : E.eq x y → ¬ In y (remove x m).

Lemma remove_2 : ¬ E.eq x y → MapsTo y e m → MapsTo y e (remove x m).

Lemma remove_3 : MapsTo y e (remove x m) → MapsTo y e m.

Lemma elements_1 :
  MapsTo x e m → InA eq_key_elt (x,e) (elements m).

Lemma elements_2 :
  InA eq_key_elt (x,e) (elements m) → MapsTo x e m.

Lemma xelements_bits_lt_1 : ∀ p p0 q m v,
  List.In (p0,v) (xelements m (append p (xO q))) → E.bits_lt p0 p.

Lemma xelements_bits_lt_2 : ∀ p p0 q m v,
  List.In (p0,v) (xelements m (append p (xI q))) → E.bits_lt p p0.

Lemma xelements_sort : ∀ p, sort lt_key (xelements m p).

Lemma elements_3 : sort lt_key (elements m).

End FMapSpec.
```

map and *mapi*

```
Variable B : Set.

Fixpoint xmapi (f : positive → A → B) (m : t A) (i : positive)
           {struct m} : t B :=
  match m with
   | Leaf ⇒ @Leaf B
   | Node l o r ⇒ Node (xmapi f l (append i (xO xH)))
                       (option_map (f i) o)
                       (xmapi f r (append i (xI xH)))
  end.

Definition mapi (f : positive → A → B) m := xmapi f m xH.

Definition map (f : A → B) m := mapi (fun _ ⇒ f) m.

End A.

Lemma xgmapi:
```

```
    ∀ (A B: Set) (f: positive → A → B) (i j : positive) (m: t A),
    find i (xmapi f m j) = option_map (f (append j i)) (find i m).

Theorem gmapi:
  ∀ (A B: Set) (f: positive → A → B) (i: positive) (m: t A),
  find i (mapi f m) = option_map (f i) (find i m).

Lemma mapi_1 :
  ∀ (elt elt':Set)(m: t elt)(x:key)(e:elt)(f:key→elt→elt'),
  MapsTo x e m →
  ∃ y, E.eq y x ∧ MapsTo x (f y e) (mapi f m).

Lemma mapi_2 :
  ∀ (elt elt':Set)(m: t elt)(x:key)(f:key→elt→elt'),
  In x (mapi f m) → In x m.

Lemma map_1 : ∀ (elt elt':Set)(m: t elt)(x:key)(e:elt)(f:elt→elt'),
  MapsTo x e m → MapsTo x (f e) (map f m).

Lemma map_2 : ∀ (elt elt':Set)(m: t elt)(x:key)(f:elt→elt'),
  In x (map f m) → In x m.

Section map2.
Variable A B C : Set.
Variable f : option A → option B → option C.

Implicit Arguments Leaf [A].

Fixpoint xmap2_l (m : t A) {struct m} : t C :=
    match m with
    | Leaf ⇒ Leaf
    | Node l o r ⇒ Node (xmap2_l l) (f o None) (xmap2_l r)
    end.

Lemma xgmap2_l : ∀ (i : positive) (m : t A),
        f None None = None → find i (xmap2_l m) = f (find i m) None.

Fixpoint xmap2_r (m : t B) {struct m} : t C :=
    match m with
    | Leaf ⇒ Leaf
    | Node l o r ⇒ Node (xmap2_r l) (f None o) (xmap2_r r)
    end.

Lemma xgmap2_r : ∀ (i : positive) (m : t B),
        f None None = None → find i (xmap2_r m) = f None (find i m).

Fixpoint _map2 (m1 : t A)(m2 : t B) {struct m1} : t C :=
  match m1 with
  | Leaf ⇒ xmap2_r m2
  | Node l1 o1 r1 ⇒
      match m2 with
      | Leaf ⇒ xmap2_l m1
      | Node l2 o2 r2 ⇒ Node (_map2 l1 l2) (f o1 o2) (_map2 r1 r2)
```

```
      end
  end.

  Lemma gmap2: ∀ (i: positive)(m1:t A)(m2: t B),
    f None None = None →
    find i (_map2 m1 m2) = f (find i m1) (find i m2).

End map2.

Definition map2 (elt elt' elt":Set)(f:option elt→option elt'→option elt") :=
 _map2 (fun o1 o2 ⇒ match o1,o2 with None,None ⇒ None | _, _ ⇒ f o1 o2 end).

Lemma map2_1 : ∀ (elt elt' elt":Set)(m: t elt)(m': t elt')
  (x:key)(f:option elt→option elt'→option elt"),
  In x m ∨ In x m' →
  find x (map2 f m m') = f (find x m) (find x m').

Lemma map2_2 : ∀ (elt elt' elt":Set)(m: t elt)(m': t elt')
  (x:key)(f:option elt→option elt'→option elt"),
  In x (map2 f m m') → In x m ∨ In x m'.

Definition fold (A B : Set) (f: positive → A → B → B) (tr: t A) (v: B) :=
   List.fold_left (fun a p ⇒ f (fst p) (snd p) a) (elements tr) v.

Lemma fold_1 :
  ∀ (A:Set)(m:t A)(B:Set)(i : B) (f : key → A → B → B),
  fold f m i = fold_left (fun a p ⇒ f (fst p) (snd p) a) (elements m) i.

Fixpoint equal (A:Set)(cmp : A → A → bool)(m1 m2 : t A) {struct m1} : bool :=
  match m1, m2 with
    | Leaf, _ ⇒ is_empty m2
    | _, Leaf ⇒ is_empty m1
    | Node l1 o1 r1, Node l2 o2 r2 ⇒
        (match o1, o2 with
          | None, None ⇒ true
          | Some v1, Some v2 ⇒ cmp v1 v2
          | _, _ ⇒ false
         end)
        && equal cmp l1 l2 && equal cmp r1 r2
   end.

Definition Equal (A:Set)(cmp:A→A→bool)(m m':t A) :=
  (∀ k, In k m ↔ In k m') ∧
  (∀ k e e', MapsTo k e m → MapsTo k e' m' → cmp e e' = true).

Lemma equal_1 : ∀ (A:Set)(m m':t A)(cmp:A→A→bool),
  Equal cmp m m' → equal cmp m m' = true.

Lemma equal_2 : ∀ (A:Set)(m m':t A)(cmp:A→A→bool),
  equal cmp m m' = true → Equal cmp m m'.

End PositiveMap.
```

Here come some additionnal facts about this implementation. Most are facts that cannot be derivable from the general interface.

Module *PositiveMapAdditionalFacts*.
Import *PositiveMap*.

Theorem *gsspec*:
∀ (*A*:Set)(*i j*: *positive*) (*x*: *A*) (*m*: *t A*),
find i (*add j x m*) = if *peq_dec i j* then *Some x* else *find i m*.

Theorem *gsident*:
∀ (*A*:Set)(*i*: *positive*) (*m*: *t A*) (*v*: *A*),
find i m = *Some v* → *add i v m* = *m*.

Lemma *xmap2_lr* :
∀ (*A B* : Set)(*f g*: *option A* → *option A* → *option B*)(*m* : *t A*),
(∀ (*i j* : *option A*), *f i j* = *g j i*) →
xmap2_l f m = *xmap2_r g m*.

Theorem *map2_commut*:
∀ (*A B*: Set) (*f g*: *option A* → *option A* → *option B*),
(∀ (*i j*: *option A*), *f i j* = *g j i*) →
∀ (*m1 m2*: *t A*),
_map2 f m1 m2 = *_map2 g m2 m1*.

End *PositiveMapAdditionalFacts*.

Chapter 211

Module Coq.FSets.FMaps

Require Export *OrderedType*.
Require Export *OrderedTypeEx*.
Require Export *OrderedTypeAlt*.
Require Export *FMapInterface*.
Require Export *FMapList*.
Require Export *FMapPositive*.
Require Export *FMapIntMap*.
Require Export *FMapFacts*.

Chapter 212

Module Coq.FSets.FMapWeakFacts

212.1 Finite maps library

This functor derives additional facts from *FMapWeakInterface.S*. These facts are mainly the specifications of *FMapWeakInterface.S* written using different styles: equivalence and boolean equalities.

Require Import *Bool*.
Require Import *OrderedType*.
Require Export *FMapWeakInterface*.

Module *Facts* (*M*: *S*).
Import *M*.
Import *Logic*.
Import *Peano*.
Lemma *MapsTo_fun* : ∀ (*elt*:Set) *m x* (*e e'*:*elt*),
MapsTo x e m → *MapsTo x e' m* → *e*=*e'*.

212.2 Specifications written using equivalences

Section *IffSpec*.
Variable *elt elt' elt''*: Set.
Implicit Type *m*: *t elt*.
Implicit Type *x y z*: *key*.
Implicit Type *e*: *elt*.

Lemma *MapsTo_iff* : ∀ *m x y e*, *E.eq x y* → (*MapsTo x e m* ↔ *MapsTo y e m*).

Lemma *In_iff* : ∀ *m x y*, *E.eq x y* → (*In x m* ↔ *In y m*).

Lemma *find_mapsto_iff* : ∀ *m x e*, *MapsTo x e m* ↔ *find x m* = *Some e*.

Lemma *not_find_mapsto_iff* : ∀ *m x*, ¬*In x m* ↔ *find x m* = *None*.

Lemma *mem_in_iff* : ∀ *m x*, *In x m* ↔ *mem x m* = *true*.

Lemma $not\_mem\_in\_iff : \forall\ m\ x, \neg In\ x\ m \leftrightarrow mem\ x\ m = false.$

Lemma $equal\_iff : \forall\ m\ m'\ cmp, Equal\ cmp\ m\ m' \leftrightarrow equal\ cmp\ m\ m' = true.$

Lemma $empty\_mapsto\_iff : \forall\ x\ e, MapsTo\ x\ e\ (empty\ elt) \leftrightarrow False.$

Lemma $empty\_in\_iff : \forall\ x, In\ x\ (empty\ elt) \leftrightarrow False.$

Lemma $is\_empty\_iff : \forall\ m, Empty\ m \leftrightarrow is\_empty\ m = true.$

Lemma $add\_mapsto\_iff : \forall\ m\ x\ y\ e\ e',$
$MapsTo\ y\ e'\ (add\ x\ e\ m) \leftrightarrow$
$(E.eq\ x\ y \wedge e{=}e') \vee$
$(\tilde{}E.eq\ x\ y \wedge MapsTo\ y\ e'\ m).$

Lemma $add\_in\_iff : \forall\ m\ x\ y\ e, In\ y\ (add\ x\ e\ m) \leftrightarrow E.eq\ x\ y \vee In\ y\ m.$

Lemma $add\_neq\_mapsto\_iff : \forall\ m\ x\ y\ e\ e',$
$\neg\ E.eq\ x\ y \rightarrow (MapsTo\ y\ e'\ (add\ x\ e\ m) \leftrightarrow MapsTo\ y\ e'\ m).$

Lemma $add\_neq\_in\_iff : \forall\ m\ x\ y\ e,$
$\neg\ E.eq\ x\ y \rightarrow (In\ y\ (add\ x\ e\ m) \leftrightarrow In\ y\ m).$

Lemma $remove\_mapsto\_iff : \forall\ m\ x\ y\ e,$
$MapsTo\ y\ e\ (remove\ x\ m) \leftrightarrow \neg E.eq\ x\ y \wedge MapsTo\ y\ e\ m.$

Lemma $remove\_in\_iff : \forall\ m\ x\ y, In\ y\ (remove\ x\ m) \leftrightarrow \neg E.eq\ x\ y \wedge In\ y\ m.$

Lemma $remove\_neq\_mapsto\_iff : \forall\ m\ x\ y\ e,$
$\neg\ E.eq\ x\ y \rightarrow (MapsTo\ y\ e\ (remove\ x\ m) \leftrightarrow MapsTo\ y\ e\ m).$

Lemma $remove\_neq\_in\_iff : \forall\ m\ x\ y,$
$\neg\ E.eq\ x\ y \rightarrow (In\ y\ (remove\ x\ m) \leftrightarrow In\ y\ m).$

Lemma $elements\_mapsto\_iff : \forall\ m\ x\ e,$
$MapsTo\ x\ e\ m \leftrightarrow InA\ (@eq\_key\_elt\ \_)\ (x,e)\ (elements\ m).$

Lemma $elements\_in\_iff : \forall\ m\ x,$
$In\ x\ m \leftrightarrow \exists\ e, InA\ (@eq\_key\_elt\ \_)\ (x,e)\ (elements\ m).$

Lemma $map\_mapsto\_iff : \forall\ m\ x\ b\ (f : elt \rightarrow elt'),$
$MapsTo\ x\ b\ (map\ f\ m) \leftrightarrow \exists\ a, b = f\ a \wedge MapsTo\ x\ a\ m.$

Lemma $map\_in\_iff : \forall\ m\ x\ (f : elt \rightarrow elt'),$
$In\ x\ (map\ f\ m) \leftrightarrow In\ x\ m.$

Lemma $mapi\_in\_iff : \forall\ m\ x\ (f{:}key{\rightarrow}elt{\rightarrow}elt'),$
$In\ x\ (mapi\ f\ m) \leftrightarrow In\ x\ m.$

Lemma $mapi\_inv : \forall\ m\ x\ b\ (f : key \rightarrow elt \rightarrow elt'),$
$MapsTo\ x\ b\ (mapi\ f\ m) \rightarrow$
$\exists\ a, \exists\ y, E.eq\ y\ x \wedge b = f\ y\ a \wedge MapsTo\ x\ a\ m.$

Lemma $mapi\_1bis : \forall\ m\ x\ e\ (f{:}key{\rightarrow}elt{\rightarrow}elt'),$
$(\forall\ x\ y\ e, E.eq\ x\ y \rightarrow f\ x\ e = f\ y\ e) \rightarrow$
$MapsTo\ x\ e\ m \rightarrow MapsTo\ x\ (f\ x\ e)\ (mapi\ f\ m).$

Lemma *mapi_mapsto_iff* : ∀ *m x b* (*f*:*key*→*elt*→*elt'*),
(∀ *x y e*, *E.eq x y* → *f x e* = *f y e*) →
(*MapsTo x b* (*mapi f m*) ↔ ∃ *a*, *b* = *f x a* ∧ *MapsTo x a m*).

Things are even worse for *map2* : we don't try to state any equivalence, see instead boolean results below.

End *IffSpec*.

Useful tactic for simplifying expressions like *In y* (*add x e* (*remove z m*))

Ltac *map_iff* :=
repeat (*progress* (
rewrite add_mapsto_iff || *rewrite add_in_iff* ||
rewrite remove_mapsto_iff || *rewrite remove_in_iff* ||
rewrite empty_mapsto_iff || *rewrite empty_in_iff* ||
rewrite map_mapsto_iff || *rewrite map_in_iff* ||
rewrite mapi_in_iff)).

212.3 Specifications written using boolean predicates

Section *BoolSpec*.

Definition *eqb x y* := if *E.eq_dec x y* then *true* else *false*.

Lemma *mem_find_b* : ∀ (*elt*:Set)(*m*:*t elt*)(*x*:*key*), *mem x m* = if *find x m* then *true* else *false*.

Variable *elt elt' elt"* : Set.
Implicit *Types m* : *t elt*.
Implicit *Types x y z* : *key*.
Implicit *Types e* : *elt*.

Lemma *mem_b* : ∀ *m x y*, *E.eq x y* → *mem x m* = *mem y m*.

Lemma *find_o* : ∀ *m x y*, *E.eq x y* → *find x m* = *find y m*.

Lemma *empty_o* : ∀ *x*, *find x* (*empty elt*) = *None*.

Lemma *empty_a* : ∀ *x*, *mem x* (*empty elt*) = *false*.

Lemma *add_eq_o* : ∀ *m x y e*,
E.eq x y → *find y* (*add x e m*) = *Some e*.

Lemma *add_neq_o* : ∀ *m x y e*,
¬ *E.eq x y* → *find y* (*add x e m*) = *find y m*.
Hint *Resolve add_neq_o*.

Lemma *add_o* : ∀ *m x y e*,
find y (*add x e m*) = if *E.eq_dec x y* then *Some e* else *find y m*.

Lemma *add_eq_b* : ∀ *m x y e*,
E.eq x y → *mem y* (*add x e m*) = *true*.

Lemma *add_neq_b* : ∀ *m x y e*,

```
 ¬E.eq x y → mem y (add x e m) = mem y m.

Lemma add_b : ∀ m x y e,
 mem y (add x e m) = eqb x y || mem y m.

Lemma remove_eq_o : ∀ m x y,
 E.eq x y → find y (remove x m) = None.
Hint Resolve remove_eq_o.

Lemma remove_neq_o : ∀ m x y,
 ¬ E.eq x y → find y (remove x m) = find y m.
Hint Resolve remove_neq_o.

Lemma remove_o : ∀ m x y,
 find y (remove x m) = if E.eq_dec x y then None else find y m.

Lemma remove_eq_b : ∀ m x y,
 E.eq x y → mem y (remove x m) = false.

Lemma remove_neq_b : ∀ m x y,
 ¬ E.eq x y → mem y (remove x m) = mem y m.

Lemma remove_b : ∀ m x y,
 mem y (remove x m) = negb (eqb x y) && mem y m.

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

Lemma map_o : ∀ m x (f:elt→elt'),
 find x (map f m) = option_map f (find x m).

Lemma map_b : ∀ m x (f:elt→elt'),
 mem x (map f m) = mem x m.

Lemma mapi_b : ∀ m x (f:key→elt→elt'),
 mem x (mapi f m) = mem x m.

Lemma mapi_o : ∀ m x (f:key→elt→elt'),
 (∀ x y e, E.eq x y → f x e = f y e) →
 find x (mapi f m) = option_map (f x) (find x m).

Lemma map2_1bis : ∀ (m: t elt)(m': t elt') x
 (f:option elt→option elt'→option elt"),
 f None None = None →
 find x (map2 f m m') = f (find x m) (find x m').

Fixpoint findA (A B:Set)(f : A → bool) (l:list (A×B)) : option B :=
 match l with
  | nil ⇒ None
  | (a,b)::l ⇒ if f a then Some b else findA f l
 end.
```

Lemma *findA_NoDupA* :
∀ (*A B*:Set)
(*eqA*:*A*→*A*→Prop)
(*eqA_sym*: ∀ *a b*, *eqA a b* → *eqA b a*)
(*eqA_trans*: ∀ *a b c*, *eqA a b* → *eqA b c* → *eqA a c*)
(*eqA_dec* : ∀ *a a'*, { *eqA a a'* }+{~ *eqA a a'* })
(*l*:*list* (*A*×*B*))(*x*:*A*)(*e*:*B*),
NoDupA (fun *p p'* ⇒ *eqA* (*fst p*) (*fst p'*)) *l* →
(*InA* (fun *p p'* ⇒ *eqA* (*fst p*) (*fst p'*) ∧ *snd p* = *snd p'*) (*x*,*e*) *l* ↔
findA (fun *y*:*A* ⇒ if *eqA_dec x y* then *true* else *false*) *l* = *Some e*).

Lemma *elements_o* : ∀ *m x*,
find x m = *findA* (*eqb x*) (*elements m*).

Lemma *elements_b* : ∀ *m x*, *mem x m* = *existsb* (fun *p* ⇒ *eqb x* (*fst p*)) (*elements m*).

End *BoolSpec*.

End *Facts*.

Chapter 213

Module Coq.FSets.FMapWeakInterface

213.1 Finite map library

This file proposes an interface for finite maps over keys with decidable equality, but no decidable order.

Require Import *FSetInterface*.
Require Import *FSetWeakInterface*.

Module Type *S*.

Declare Module *E* : *DecidableType*.

Definition *key* := *E.t*.

Parameter *t* : Set → Set.
the abstract type of maps

Section *Types*.

Variable *elt*:Set.

Parameter *empty* : *t elt*.
The empty map.

Parameter *is_empty* : *t elt* → *bool*.
Test whether a map is empty or not.

Parameter *add* : *key* → *elt* → *t elt* → *t elt*.
add x y m returns a map containing the same bindings as m, plus a binding of x to y. If x was already bound in m, its previous binding disappears.

Parameter *find* : *key* → *t elt* → *option elt*.
find x m returns the current binding of x in m, or raises *Not_found* if no such binding exists. NB: in Coq, the exception mechanism becomes a option type.

Parameter *remove* : *key* → *t elt* → *t elt*.
remove x m returns a map containing the same bindings as m, except for x which is unbound in the returned map.

Parameter *mem* : *key* → *t elt* → *bool*.
mem x m returns *true* if *m* contains a binding for *x*, and *false* otherwise.

Coq comment: *iter* is useless in a purely functional world
val iter : (key -> 'a -> unit) -> 'a t -> unit
iter f m applies f to all bindings in map m. f receives the key as first argument, and the associated value as second argument. The bindings are passed to f in increasing order with respect to the ordering over the type of the keys. Only current bindings are presented to f: bindings hidden by more recent bindings are not passed to f.

Variable *elt'* : Set.
Variable *elt"*: Set.

Parameter *map* : (*elt* → *elt'*) → *t elt* → *t elt'*.
map f m returns a map with same domain as *m*, where the associated value a of all bindings of *m* has been replaced by the result of the application of *f* to *a*. The bindings are passed to *f* in increasing order with respect to the ordering over the type of the keys.

Parameter *mapi* : (*key* → *elt* → *elt'*) → *t elt* → *t elt'*.
Same as *S.map*, but the function receives as arguments both the key and the associated value for each binding of the map.

Parameter *map2* : (*option elt* → *option elt'* → *option elt"*) → *t elt* → *t elt'* → *t elt"*.
Not present in Ocaml. *map f m m'* creates a new map whose bindings belong to the ones of either *m* or *m'*. The presence and value for a key *k* is determined by *f e e'* where *e* and *e'* are the (optional) bindings of *k* in *m* and *m'*.

Parameter *elements* : *t elt* → *list* (*key*×*elt*).
Not present in Ocaml. *elements m* returns an assoc list corresponding to the bindings of *m*. Elements of this list are sorted with respect to their first components. Useful to specify *fold* ...

Parameter *fold* : ∀ *A*: Set, (*key* → *elt* → *A* → *A*) → *t elt* → *A* → *A*.
fold f m a computes (*f kN dN* ... (*f k1 d1 a*)...), where *k1* ... *kN* are the keys of all bindings in *m* (in increasing order), and *d1* ... *dN* are the associated data.

Parameter *equal* : (*elt* → *elt* → *bool*) → *t elt* → *t elt* → *bool*.
equal cmp m1 m2 tests whether the maps *m1* and *m2* are equal, that is, contain equal keys and associate them with equal data. *cmp* is the equality predicate used to compare the data associated with the keys.

Section *Spec*.

Variable *m m' m"* : *t elt*.
Variable *x y z* : *key*.
Variable *e e'* : *elt*.

Parameter *MapsTo* : *key* → *elt* → *t elt* → Prop.

Definition *In* (*k*:*key*)(*m*: *t elt*) : Prop := ∃ *e*:*elt*, *MapsTo k e m*.

Definition *Empty m* := ∀ (*a* : *key*)(*e*:*elt*) , ¬ *MapsTo a e m*.

Definition *eq_key* (*p p'*:*key*×*elt*) := *E.eq* (*fst p*) (*fst p'*).

Definition *eq_key_elt* (*p p'*:*key*×*elt*) :=
E.eq (*fst p*) (*fst p'*) ∧ (*snd p*) = (*snd p'*).

Specification of *MapsTo*
Parameter *MapsTo_1* : *E.eq x y* → *MapsTo x e m* → *MapsTo y e m*.

Specification of *mem*
Parameter *mem_1* : *In x m* → *mem x m* = *true*.
Parameter *mem_2* : *mem x m* = *true* → *In x m*.

Specification of *empty*
Parameter *empty_1* : *Empty empty*.

Specification of *is_empty*
Parameter *is_empty_1* : *Empty m* → *is_empty m* = *true*.
Parameter *is_empty_2* : *is_empty m* = *true* → *Empty m*.

Specification of *add*
Parameter *add_1* : *E.eq x y* → *MapsTo y e* (*add x e m*).
Parameter *add_2* : ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*add x e' m*).
Parameter *add_3* : ¬ *E.eq x y* → *MapsTo y e* (*add x e' m*) → *MapsTo y e m*.

Specification of *remove*
Parameter *remove_1* : *E.eq x y* → ¬ *In y* (*remove x m*).
Parameter *remove_2* : ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*remove x m*).
Parameter *remove_3* : *MapsTo y e* (*remove x m*) → *MapsTo y e m*.

Specification of *find*
Parameter *find_1* : *MapsTo x e m* → *find x m* = *Some e*.
Parameter *find_2* : *find x m* = *Some e* → *MapsTo x e m*.

Specification of *elements*
Parameter *elements_1* :
MapsTo x e m → *InA eq_key_elt* (*x*,*e*) (*elements m*).
Parameter *elements_2* :
InA eq_key_elt (*x*,*e*) (*elements m*) → *MapsTo x e m*.
Parameter *elements_3* : *NoDupA eq_key* (*elements m*).

Specification of *fold*
Parameter *fold_1* :
∀ (*A* : Set) (*i* : *A*) (*f* : *key* → *elt* → *A* → *A*),
fold f m i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

Definition *Equal cmp m m'* :=
(∀ *k*, *In k m* ↔ *In k m'*) ∧
(∀ *k e e'*, *MapsTo k e m* → *MapsTo k e' m'* → *cmp e e'* = *true*).

Variable *cmp* : *elt* → *elt* → *bool*.

Specification of *equal*
Parameter *equal_1* : *Equal cmp m m'* → *equal cmp m m'* = *true*.
Parameter *equal_2* : *equal cmp m m'* = *true* → *Equal cmp m m'*.

End *Spec*.
End *Types*.

Specification of *map*

Parameter *map_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*elt*→*elt'*),
MapsTo x e m → *MapsTo x* (*f e*) (*map f m*).
Parameter *map_2* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*f*:*elt*→*elt'*),
In x (*map f m*) → *In x m*.

Specification of *mapi*

Parameter *mapi_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)
(*f*:*key*→*elt*→*elt'*), *MapsTo x e m* →
∃ *y*, *E.eq y x* ∧ *MapsTo x* (*f y e*) (*mapi f m*).
Parameter *mapi_2* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)
(*f*:*key*→*elt*→*elt'*), *In x* (*mapi f m*) → *In x m*.

Specification of *map2*

Parameter *map2_1* : ∀ (*elt elt' elt"*:Set)(*m*: *t elt*)(*m'*: *t elt'*)
(*x*:*key*)(*f*:*option elt*→*option elt'*→*option elt"*),
In x m ∨ *In x m'* →
find x (*map2 f m m'*) = *f* (*find x m*) (*find x m'*).

Parameter *map2_2* : ∀ (*elt elt' elt"*:Set)(*m*: *t elt*)(*m'*: *t elt'*)
(*x*:*key*)(*f*:*option elt*→*option elt'*→*option elt"*),
In x (*map2 f m m'*) → *In x m* ∨ *In x m'*.

Hint Immediate *MapsTo_1 mem_2 is_empty_2*.

Hint *Resolve mem_1 is_empty_1 is_empty_2 add_1 add_2 add_3 remove_1 remove_2 remove_3 find_1 find_2 fold_1 map_1 map_2 mapi_1 mapi_2*.

End *S*.

Chapter 214

Module Coq.FSets.FMapWeakList

214.1 Finite map library

This file proposes an implementation of the non-dependant interface *FMapInterface.S* using lists of pairs, unordered but without redundancy.

Require Import *FSetInterface*.
Require Import *FSetWeakInterface*.
Require Import *FMapWeakInterface*.

Module *Raw* (*X*:*DecidableType*).

Module *PX* := *KeyDecidableType X*.
Import *PX*.

Definition *key* := *X.t*.
Definition *t* (*elt*:Set) := *list* (*X.t* × *elt*).

Section *Elt*.

Variable *elt* : Set.

Notation *eqk* := (*eqk* (*elt*:=*elt*)).
Notation *eqke* := (*eqke* (*elt*:=*elt*)).
Notation *MapsTo* := (*MapsTo* (*elt*:=*elt*)).
Notation *In* := (*In* (*elt*:=*elt*)).
Notation *NoDupA* := (*NoDupA eqk*).

214.2 *empty*

Definition *empty* : *t elt* := *nil*.

Definition *Empty m* := ∀ (*a* : *key*)(*e*:*elt*), ¬ *MapsTo a e m*.

Lemma *empty_1* : *Empty empty*.

Hint *Resolve empty_1.*

Lemma *empty_NoDup* : *NoDupA empty.*

214.3 *is_empty*

Definition *is_empty* (*l* : *t elt*) : *bool* := if *l* then *true* else *false.*

Lemma *is_empty_1* :∀ *m*, *Empty m* → *is_empty m* = *true.*

Lemma *is_empty_2* : ∀ *m*, *is_empty m* = *true* → *Empty m.*

214.4 *mem*

Lemma *mem_1* : ∀ *m* (*Hm*:*NoDupA m*) *x*, *In x m* → *mem x m* = *true.*

Lemma *mem_2* : ∀ *m* (*Hm*:*NoDupA m*) *x*, *mem x m* = *true* → *In x m.*

214.5 *find*

Lemma *find_2* : ∀ *m x e*, *find x m* = *Some e* → *MapsTo x e m.*

Lemma *find_1* : ∀ *m* (*Hm*:*NoDupA m*) *x e*,
MapsTo x e m → *find x m* = *Some e.*

Lemma *find_eq* : ∀ *m* (*Hm*:*NoDupA m*) *x x'*,
X.eq x x' → *find x m* = *find x' m.*

214.6 *add*

Lemma *add_1* : ∀ *m x y e*, *X.eq x y* → *MapsTo y e* (*add x e m*).

Lemma *add_2* : ∀ *m x y e e'*,
¬ *X.eq x y* → *MapsTo y e m* → *MapsTo y e* (*add x e' m*).

Lemma *add_3* : ∀ *m x y e e'*,
¬ *X.eq x y* → *MapsTo y e* (*add x e' m*) → *MapsTo y e m.*

Lemma *add_3'* : ∀ *m x y e e'*,
¬ *X.eq x y* → *InA eqk* (*y*,*e*) (*add x e' m*) → *InA eqk* (*y*,*e*) *m.*

Lemma *add_NoDup* : ∀ *m* (*Hm*:*NoDupA m*) *x e*, *NoDupA* (*add x e m*).

Lemma *add_eq* : ∀ *m* (*Hm*:*NoDupA m*) *x a e*,
X.eq x a → *find x* (*add a e m*) = *Some e.*

Lemma *add_not_eq* : ∀ *m* (*Hm*:*NoDupA m*) *x a e*,
¬*X.eq x a* → *find x* (*add a e m*) = *find x m.*

214.7 *remove*

Lemma *remove_1* : ∀ *m* (*Hm*:*NoDupA m*) *x y*, *X.eq x y* → ¬ *In y* (*remove x m*).

Lemma *remove_2* : ∀ *m* (*Hm*:*NoDupA m*) *x y e*,
¬ *X.eq x y* → *MapsTo y e m* → *MapsTo y e* (*remove x m*).

Lemma *remove_3* : ∀ *m* (*Hm*:*NoDupA m*) *x y e*,
MapsTo y e (*remove x m*) → *MapsTo y e m*.

Lemma *remove_3'* : ∀ *m* (*Hm*:*NoDupA m*) *x y e*,
InA eqk (*y*,*e*) (*remove x m*) → *InA eqk* (*y*,*e*) *m*.

Lemma *remove_NoDup* : ∀ *m* (*Hm*:*NoDupA m*) *x*, *NoDupA* (*remove x m*).

214.8 *elements*

Definition *elements* (*m*: *t elt*) := *m*.

Lemma *elements_1* : ∀ *m x e*, *MapsTo x e m* → *InA eqke* (*x*,*e*) (*elements m*).

Lemma *elements_2* : ∀ *m x e*, *InA eqke* (*x*,*e*) (*elements m*) → *MapsTo x e m*.

Lemma *elements_3* : ∀ *m* (*Hm*:*NoDupA m*), *NoDupA* (*elements m*).

214.9 *fold*

Lemma *fold_1* : ∀ *m* (*A*:Set)(*i*:*A*)(*f*:*key*→*elt*→*A*→*A*),
fold f m i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

214.10 *equal*

Definition *check* (*cmp* : *elt* → *elt* → *bool*)(*k*:*key*)(*e*:*elt*)(*m'*: *t elt*) :=
match *find k m'* with
| *None* ⇒ *false*
| *Some e'* ⇒ *cmp e e'*
end.

Definition *submap* (*cmp* : *elt* → *elt* → *bool*)(*m m'* : *t elt*) : *bool* :=
fold (fun *k e b* ⇒ *andb* (*check cmp k e m'*) *b*) *m true*.

Definition *equal* (*cmp* : *elt* → *elt* → *bool*)(*m m'* : *t elt*) : *bool* :=
andb (*submap cmp m m'*) (*submap* (fun *e' e* ⇒ *cmp e e'*) *m' m*).

Definition *Submap cmp m m'* :=

$(\forall\ k,\ In\ k\ m \rightarrow In\ k\ m') \wedge$
$(\forall\ k\ e\ e',\ MapsTo\ k\ e\ m \rightarrow MapsTo\ k\ e'\ m' \rightarrow cmp\ e\ e' = true).$

Definition $Equal\ cmp\ m\ m' :=$
$(\forall\ k,\ In\ k\ m \leftrightarrow In\ k\ m') \wedge$
$(\forall\ k\ e\ e',\ MapsTo\ k\ e\ m \rightarrow MapsTo\ k\ e'\ m' \rightarrow cmp\ e\ e' = true).$

Lemma $submap\_1 : \forall\ m\ (Hm{:}NoDupA\ m)\ m'\ (Hm'{:}\ NoDupA\ m')\ cmp,$
$Submap\ cmp\ m\ m' \rightarrow submap\ cmp\ m\ m' = true.$

Lemma $submap\_2 : \forall\ m\ (Hm{:}NoDupA\ m)\ m'\ (Hm'{:}\ NoDupA\ m')\ cmp,$
$submap\ cmp\ m\ m' = true \rightarrow Submap\ cmp\ m\ m'.$

Specification of *equal*

Lemma $equal\_1 : \forall\ m\ (Hm{:}NoDupA\ m)\ m'\ (Hm'{:}\ NoDupA\ m')\ cmp,$
$Equal\ cmp\ m\ m' \rightarrow equal\ cmp\ m\ m' = true.$

Lemma $equal\_2 : \forall\ m\ (Hm{:}NoDupA\ m)\ m'\ (Hm'{:}NoDupA\ m')\ cmp,$
$equal\ cmp\ m\ m' = true \rightarrow Equal\ cmp\ m\ m'.$

Variable elt':Set.

214.11 *map* and *mapi*

Fixpoint $map\ (f{:}elt \rightarrow elt')\ (m{:}t\ elt)\ \{struct\ m\} : t\ elt' :=$
match m with
$\mid nil \Rightarrow nil$
$\mid (k,e)::m' \Rightarrow (k,f\ e) :: map\ f\ m'$
end.

Fixpoint $mapi\ (f{:}\ key \rightarrow elt \rightarrow elt')\ (m{:}t\ elt)\ \{struct\ m\} : t\ elt' :=$
match m with
$\mid nil \Rightarrow nil$
$\mid (k,e)::m' \Rightarrow (k,f\ k\ e) :: mapi\ f\ m'$
end.

End *Elt*.
Section *Elt2*.

Variable $elt\ elt'$: Set.

Specification of *map*

Lemma $map\_1 : \forall\ (m{:}t\ elt)(x{:}key)(e{:}elt)(f{:}elt{\rightarrow}elt'),$
$MapsTo\ x\ e\ m \rightarrow MapsTo\ x\ (f\ e)\ (map\ f\ m).$

Lemma $map\_2 : \forall\ (m{:}t\ elt)(x{:}key)(f{:}elt{\rightarrow}elt'),$
$In\ x\ (map\ f\ m) \rightarrow In\ x\ m.$

Lemma $map\_NoDup : \forall\ m\ (Hm : NoDupA\ (@eqk\ elt)\ m)(f{:}elt{\rightarrow}elt'),$

NoDupA (@eqk elt') (map f m).

Specification of *mapi*

Lemma *mapi_1* : ∀ (*m*:*t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*key*→*elt*→*elt'*),
MapsTo x e m →
∃ *y*, *X.eq y x* ∧ *MapsTo x* (*f y e*) (*mapi f m*).

Lemma *mapi_2* : ∀ (*m*:*t elt*)(*x*:*key*)(*f*:*key*→*elt*→*elt'*),
In x (*mapi f m*) → *In x m*.

Lemma *mapi_NoDup* : ∀ *m* (*Hm* : *NoDupA* (@*eqk elt*) *m*)(*f*: *key*→*elt*→*elt'*),
NoDupA (@*eqk elt'*) (*mapi f m*).

End *Elt2*.
Section *Elt3*.

Variable *elt elt' elt"* : Set.

Notation *oee'* := (*option elt* × *option elt'*)%*type*.

Definition *combine_l* (*m*:*t elt*)(*m'*:*t elt'*) : *t oee'* :=
mapi (fun *k e* ⇒ (*Some e*, *find k m'*)) *m*.

Definition *combine_r* (*m*:*t elt*)(*m'*:*t elt'*) : *t oee'* :=
mapi (fun *k e'* ⇒ (*find k m*, *Some e'*)) *m'*.

Definition *fold_right_pair* (*A B C*:Set)(*f*:*A*→*B*→*C*→*C*)(*l*:*list* (*A*×*B*))(*i*:*C*) :=
List.fold_right (fun *p* ⇒ *f* (*fst p*) (*snd p*)) *i l*.

Definition *combine* (*m*:*t elt*)(*m'*:*t elt'*) : *t oee'* :=
let *l* := *combine_l m m'* in
let *r* := *combine_r m m'* in
fold_right_pair (*add* (*elt*:=*oee'*)) *l r*.

Lemma *fold_right_pair_NoDup* :
∀ *l r* (*Hl*: *NoDupA* (*eqk* (*elt*:=*oee'*)) *l*)
(*Hl*: *NoDupA* (*eqk* (*elt*:=*oee'*)) *r*),
NoDupA (*eqk* (*elt*:=*oee'*)) (*fold_right_pair* (*add* (*elt*:=*oee'*)) *l r*).
Hint *Resolve fold_right_pair_NoDup*.

Lemma *combine_NoDup* :
∀ *m* (*Hm*:*NoDupA* (@*eqk elt*) *m*) *m'* (*Hm'*:*NoDupA* (@*eqk elt'*) *m'*),
NoDupA (@*eqk oee'*) (*combine m m'*).

Definition *at_least_left* (*o*:*option elt*)(*o'*:*option elt'*) :=
match *o* with
| *None* ⇒ *None*
| _ ⇒ *Some* (*o*,*o'*)
end.

Definition *at_least_right* (*o*:*option elt*)(*o'*:*option elt'*) :=
match *o'* with

```
  | None ⇒ None
  | _ ⇒ Some (o,o')
 end.

Lemma combine_l_1 :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m')(x:key),
 find x (combine_l m m') = at_least_left (find x m) (find x m').

Lemma combine_r_1 :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m')(x:key),
 find x (combine_r m m') = at_least_right (find x m) (find x m').

Definition at_least_one (o:option elt)(o':option elt') :=
 match o, o' with
  | None, None ⇒ None
  | _, _ ⇒ Some (o,o')
 end.

Lemma combine_1 :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m')(x:key),
 find x (combine m m') = at_least_one (find x m) (find x m').

Variable f : option elt → option elt' → option elt".

Definition option_cons (A:Set)(k:key)(o:option A)(l:list (key×A)) :=
 match o with
  | Some e ⇒ (k,e)::l
  | None ⇒ l
 end.

Definition map2 m m' :=
 let m0 : t oee' := combine m m' in
 let m1 : t (option elt") := map (fun p ⇒ f (fst p) (snd p)) m0 in
 fold_right_pair (option_cons (A:=elt")) m1 nil.

Lemma map2_NoDup :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m'),
 NoDupA (@eqk elt") (map2 m m').

Definition at_least_one_then_f (o:option elt)(o':option elt') :=
 match o, o' with
  | None, None ⇒ None
  | _, _ ⇒ f o o'
 end.

Lemma map2_0 :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m')(x:key),
 find x (map2 m m') = at_least_one_then_f (find x m) (find x m').
```

Specification of *map2*

```
Lemma map2_1 :
 ∀ m (Hm:NoDupA (@eqk elt) m) m' (Hm':NoDupA (@eqk elt') m')(x:key),
```

In x m ∨ In x m' →
find x (map2 m m') = f (find x m) (find x m').

Lemma *map2_2* :
∀ *m* (*Hm*:*NoDupA* (@*eqk elt*) *m*) *m'* (*Hm'*:*NoDupA* (@*eqk elt'*) *m'*)(*x*:*key*),
In x (*map2 m m'*) → *In x m* ∨ *In x m'*.

End *Elt3*.
End *Raw*.

Module *Make* (*X*: *DecidableType*) <: *S* with Module *E*:=*X*.
Module *Raw* := *Raw X*.

Module *E* := *X*.
Definition *key* := *E.t*.

Record *slist* (*elt*:Set) : Set :=
{*this* :> *Raw.t elt*; *NoDup* : *NoDupA* (@*Raw.PX.eqk elt*) *this*}.
Definition *t* (*elt*:Set) := *slist elt*.

Section *Elt*.
Variable *elt elt' elt''*:Set.

Implicit *Types m* : *t elt*.
Implicit *Types x y* : *key*.
Implicit *Types e* : *elt*.

Definition *empty* : *t elt* := *Build_slist* (*Raw.empty_NoDup elt*).
Definition *is_empty m* : *bool* := *Raw.is_empty m*.(*this*).
Definition *add x e m* : *t elt* := *Build_slist* (*Raw.add_NoDup m*.(*NoDup*) *x e*).
Definition *find x m* : *option elt* := *Raw.find x m*.(*this*).
Definition *remove x m* : *t elt* := *Build_slist* (*Raw.remove_NoDup m*.(*NoDup*) *x*).
Definition *mem x m* : *bool* := *Raw.mem x m*.(*this*).
Definition *map f m* : *t elt'* := *Build_slist* (*Raw.map_NoDup m*.(*NoDup*) *f*).
Definition *mapi* (*f*:*key*→*elt*→*elt'*) *m* : *t elt'* := *Build_slist* (*Raw.mapi_NoDup m*.(*NoDup*) *f*).
Definition *map2 f m* (*m'*:*t elt'*) : *t elt''* :=
Build_slist (*Raw.map2_NoDup f m*.(*NoDup*) *m'*.(*NoDup*)).
Definition *elements m* : *list* (*key*×*elt*) := @*Raw.elements elt m*.(*this*).
Definition *fold* (*A*:Set)(*f*:*key*→*elt*→*A*→*A*) *m* (*i*:*A*) : *A* := @*Raw.fold elt A f m*.(*this*) *i*.
Definition *equal cmp m m'* : *bool* := @*Raw.equal elt cmp m*.(*this*) *m'*.(*this*).

Definition *MapsTo x e m* : Prop := *Raw.PX.MapsTo x e m*.(*this*).
Definition *In x m* : Prop := *Raw.PX.In x m*.(*this*).
Definition *Empty m* : Prop := *Raw.Empty m*.(*this*).
Definition *Equal cmp m m'* : Prop := @*Raw.Equal elt cmp m*.(*this*) *m'*.(*this*).

Definition *eq_key* : (*key*×*elt*) → (*key*×*elt*) → Prop := @*Raw.PX.eqk elt*.
Definition *eq_key_elt* : (*key*×*elt*) → (*key*×*elt*) → Prop:= @*Raw.PX.eqke elt*.

Lemma *MapsTo_1* : ∀ *m x y e*, *E.eq x y* → *MapsTo x e m* → *MapsTo y e m*.

Lemma *mem_1* : ∀ *m x*, *In x m* → *mem x m* = *true*.
Lemma *mem_2* : ∀ *m x*, *mem x m* = *true* → *In x m*.

Lemma *empty_1* : *Empty empty*.

Lemma *is_empty_1* : ∀ *m*, *Empty m* → *is_empty m* = *true*.
Lemma *is_empty_2* : ∀ *m*, *is_empty m* = *true* → *Empty m*.

Lemma *add_1* : ∀ *m x y e*, *E.eq x y* → *MapsTo y e* (*add x e m*).
Lemma *add_2* : ∀ *m x y e e'*, ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*add x e' m*).
Lemma *add_3* : ∀ *m x y e e'*, ¬ *E.eq x y* → *MapsTo y e* (*add x e' m*) → *MapsTo y e m*.

Lemma *remove_1* : ∀ *m x y*, *E.eq x y* → ¬ *In y* (*remove x m*).
Lemma *remove_2* : ∀ *m x y e*, ¬ *E.eq x y* → *MapsTo y e m* → *MapsTo y e* (*remove x m*).
Lemma *remove_3* : ∀ *m x y e*, *MapsTo y e* (*remove x m*) → *MapsTo y e m*.

Lemma *find_1* : ∀ *m x e*, *MapsTo x e m* → *find x m* = *Some e*.
Lemma *find_2* : ∀ *m x e*, *find x m* = *Some e* → *MapsTo x e m*.

Lemma *elements_1* : ∀ *m x e*, *MapsTo x e m* → *InA eq_key_elt* (*x*,*e*) (*elements m*).
Lemma *elements_2* : ∀ *m x e*, *InA eq_key_elt* (*x*,*e*) (*elements m*) → *MapsTo x e m*.
Lemma *elements_3* : ∀ *m*, *NoDupA eq_key* (*elements m*).

Lemma *fold_1* : ∀ *m* (*A* : Set) (*i* : *A*) (*f* : *key* → *elt* → *A* → *A*),
fold f m i = *fold_left* (fun *a p* ⇒ *f* (*fst p*) (*snd p*) *a*) (*elements m*) *i*.

Lemma *equal_1* : ∀ *m m' cmp*, *Equal cmp m m'* → *equal cmp m m'* = *true*.
Lemma *equal_2* : ∀ *m m' cmp*, *equal cmp m m'* = *true* → *Equal cmp m m'*.

End *Elt*.

Lemma *map_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)(*f*:*elt*→*elt'*),
MapsTo x e m → *MapsTo x* (*f e*) (*map f m*).
Lemma *map_2* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*f*:*elt*→*elt'*),
In x (*map f m*) → *In x m*.

Lemma *mapi_1* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)(*e*:*elt*)
(*f*:*key*→*elt*→*elt'*), *MapsTo x e m* →
∃ *y*, *E.eq y x* ∧ *MapsTo x* (*f y e*) (*mapi f m*).
Lemma *mapi_2* : ∀ (*elt elt'*:Set)(*m*: *t elt*)(*x*:*key*)
(*f*:*key*→*elt*→*elt'*), *In x* (*mapi f m*) → *In x m*.

Lemma *map2_1* : ∀ (*elt elt' elt"*:Set)(*m*: *t elt*)(*m'*: *t elt'*)
(*x*:*key*)(*f*:*option elt*→*option elt'*→*option elt"*),
In x m ∨ *In x m'* →
find x (*map2 f m m'*) = *f* (*find x m*) (*find x m'*).
Lemma *map2_2* : ∀ (*elt elt' elt"*:Set)(*m*: *t elt*)(*m'*: *t elt'*)
(*x*:*key*)(*f*:*option elt*→*option elt'*→*option elt"*),
In x (*map2 f m m'*) → *In x m* ∨ *In x m'*.

End *Make*.

Chapter 215

Module Coq.FSets.FMapWeak

Require Export *DecidableType*.
Require Export *DecidableTypeEx*.
Require Export *FMapWeakInterface*.
Require Export *FMapWeakList*.
Require Export *FMapWeakFacts*.

Chapter 216

Module Coq.FSets.FSetAVL

This module implements sets using AVL trees. It follows the implementation from Ocaml's standard library.

```
Require Import FSetInterface.
Require Import FSetList.
Require Import ZArith.
Require Import Int.

Module Raw (I:Int)(X:OrderedType).
Import I.
Module II:=MoreInt(I).
Import II.
Open Local Scope Int_scope.

Module E := X.
Module MX := OrderedTypeFacts X.

Definition elt := X.t.
```

216.1 Trees

```
Inductive tree : Set :=
  | Leaf : tree
  | Node : tree → X.t → tree → int → tree.

Notation t := tree.
```

The fourth field of *Node* is the height of the tree

A tactic to repeat *inversion_clear* on all hyps of the form (*f* (*Node* _ _ _ _))

```
Ltac inv f :=
  match goal with
      | H:f Leaf ⊢ _ ⇒ inversion_clear H; inv f
      | H:f _ Leaf ⊢ _ ⇒ inversion_clear H; inv f
```

```
    | H:f (Node _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
    | H:f _ (Node _ _ _ _) ⊢ _ ⇒ inversion_clear H; inv f
    | _ ⇒ idtac
  end.
```

Same, but with a backup of the original hypothesis.

```
Ltac safe_inv f := match goal with
  | H:f (Node _ _ _ _) ⊢ _ ⇒
        generalize H; inversion_clear H; safe_inv f
  | _ ⇒ intros
 end.
```

216.2 Occurrence in a tree

```
Inductive In (x : elt) : tree → Prop :=
  | IsRoot :
      ∀ (l r : tree) (h : int) (y : elt),
      X.eq x y → In x (Node l y r h)
  | InLeft :
      ∀ (l r : tree) (h : int) (y : elt),
      In x l → In x (Node l y r h)
  | InRight :
      ∀ (l r : tree) (h : int) (y : elt),
      In x r → In x (Node l y r h).

Hint Constructors In.

Ltac intuition_in := repeat progress (intuition; inv In).
```

In is compatible with *X.eq*

```
Lemma In_1 :
 ∀ s x y, X.eq x y → In x s → In y s.
Hint Immediate In_1.
```

216.3 Binary search trees

lt_tree x s: all elements in *s* are smaller than *x* (resp. greater for *gt_tree*)

```
Definition lt_tree (x : elt) (s : tree) :=
 ∀ y:elt, In y s → X.lt y x.
Definition gt_tree (x : elt) (s : tree) :=
 ∀ y:elt, In y s → X.lt x y.

Hint Unfold lt_tree gt_tree.

Ltac order := match goal with
```

```
| H: lt_tree ?x ?s, H1: In ?y ?s ⊢ _ ⇒ generalize (H _ H1); clear H; order
| H: gt_tree ?x ?s, H1: In ?y ?s ⊢ _ ⇒ generalize (H _ H1); clear H; order
| _ ⇒ MX.order
  end.
```

Results about *lt_tree* and *gt_tree*

```
Lemma lt_leaf : ∀ x : elt, lt_tree x Leaf.
```

```
Lemma gt_leaf : ∀ x : elt, gt_tree x Leaf.
```

```
Lemma lt_tree_node :
 ∀ (x y : elt) (l r : tree) (h : int),
 lt_tree x l → lt_tree x r → X.lt y x → lt_tree x (Node l y r h).
```

```
Lemma gt_tree_node :
 ∀ (x y : elt) (l r : tree) (h : int),
 gt_tree x l → gt_tree x r → X.lt x y → gt_tree x (Node l y r h).
```

```
Hint Resolve lt_leaf gt_leaf lt_tree_node gt_tree_node.
```

```
Lemma lt_tree_not_in :
 ∀ (x : elt) (t : tree), lt_tree x t → ¬ In x t.
```

```
Lemma lt_tree_trans :
 ∀ x y, X.lt x y → ∀ t, lt_tree x t → lt_tree y t.
```

```
Lemma gt_tree_not_in :
 ∀ (x : elt) (t : tree), gt_tree x t → ¬ In x t.
```

```
Lemma gt_tree_trans :
 ∀ x y, X.lt y x → ∀ t, gt_tree x t → gt_tree y t.
```

```
Hint Resolve lt_tree_not_in lt_tree_trans gt_tree_not_in gt_tree_trans.
```

bst t : *t* is a binary search tree

```
Inductive bst : tree → Prop :=
  | BSLeaf : bst Leaf
  | BSNode :
      ∀ (x : elt) (l r : tree) (h : int),
      bst l → bst r → lt_tree x l → gt_tree x r → bst (Node l x r h).
```

```
Hint Constructors bst.
```

216.4 AVL trees

avl s : *s* is a properly balanced AVL tree, i.e. for any node the heights of the two children differ by at most 2

```
Definition height (s : tree) : int :=
  match s with
  | Leaf ⇒ 0
```

```
  | Node _ _ _ h ⇒ h
  end.

Inductive avl : tree → Prop :=
  | RBLeaf : avl Leaf
  | RBNode :
      ∀ (x : elt) (l r : tree) (h : int),
      avl l →
      avl r →
      -(2) ≤ height l - height r ≤ 2 →
      h = max (height l) (height r) + 1 →
      avl (Node l x r h).

Hint Constructors avl.
```

Results about *avl*

```
Lemma avl_node :
 ∀ (x : elt) (l r : tree),
 avl l →
 avl r →
 -(2) ≤ height l - height r ≤ 2 →
 avl (Node l x r (max (height l) (height r) + 1)).
Hint Resolve avl_node.
```

The tactics

```
Lemma height_non_negative : ∀ s : tree, avl s → height s ≥ 0.
Implicit Arguments height_non_negative.
```

When *H*:*avl r*, typing *avl_nn H* or *avl_nn r* adds *height r*≥0

```
Ltac avl_nn_hyp H :=
      let nz := fresh "nz" in assert (nz := height_non_negative H).

Ltac avl_nn h :=
  let t := type of h in
  match type of t with
    | Prop ⇒ avl_nn_hyp h
    | _ ⇒ match goal with H : avl h ⊢ _ ⇒ avl_nn_hyp H end
  end.

Ltac avl_nns :=
  match goal with
      | H:avl _ ⊢ _ ⇒ avl_nn_hyp H; clear H; avl_nns
      | _ ⇒ idtac
  end.
```

216.5 Some shortcuts.

```
Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
```

Definition $Subset\ s\ s' := \forall\ a : elt, In\ a\ s \rightarrow In\ a\ s'$.
Definition $Empty\ s := \forall\ a : elt, \neg\ In\ a\ s$.
Definition $For\_all\ (P : elt \rightarrow \mathsf{Prop})\ s := \forall\ x, In\ x\ s \rightarrow P\ x$.
Definition $Exists\ (P : elt \rightarrow \mathsf{Prop})\ s := \exists\ x, In\ x\ s \wedge P\ x$.

216.6 Empty set

Definition $empty := Leaf$.

Lemma $empty\_bst : bst\ empty$.

Lemma $empty\_avl : avl\ empty$.

Lemma $empty\_1 : Empty\ empty$.

216.7 Emptyness test

Definition $is\_empty\ (s{:}t) :=$ match s with $Leaf \Rightarrow true \mid \_ \Rightarrow false$ end.

Lemma $is\_empty\_1 : \forall\ s, Empty\ s \rightarrow is\_empty\ s = true$.

Lemma $is\_empty\_2 : \forall\ s, is\_empty\ s = true \rightarrow Empty\ s$.

216.8 Appartness

The *mem* function is deciding appartness. It exploits the *bst* property to achieve logarithmic complexity.

Lemma $mem\_1 : \forall\ s\ x, bst\ s \rightarrow In\ x\ s \rightarrow mem\ x\ s = true$.

Lemma $mem\_2 : \forall\ s\ x, mem\ x\ s = true \rightarrow In\ x\ s$.

216.9 Singleton set

Definition $singleton\ (x : elt) := Node\ Leaf\ x\ Leaf\ 1$.

Lemma $singleton\_bst : \forall\ x : elt, bst\ (singleton\ x)$.

Lemma $singleton\_avl : \forall\ x : elt, avl\ (singleton\ x)$.

Lemma $singleton\_1 : \forall\ x\ y, In\ y\ (singleton\ x) \rightarrow X.eq\ x\ y$.

Lemma $singleton\_2 : \forall\ x\ y, X.eq\ x\ y \rightarrow In\ y\ (singleton\ x)$.

216.10 Helper functions

create l x r creates a node, assuming l and r to be balanced and $|height\ l$ - $height\ r| \leq 2$.

```
Definition create l x r :=
  Node l x r (max (height l) (height r) + 1).

Lemma create_bst :
 ∀ l x r, bst l → bst r → lt_tree x l → gt_tree x r →
 bst (create l x r).
Hint Resolve create_bst.

Lemma create_avl :
 ∀ l x r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 avl (create l x r).

Lemma create_height :
 ∀ l x r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 height (create l x r) = max (height l) (height r) + 1.

Lemma create_in :
 ∀ l x r y, In y (create l x r) ↔ X.eq y x ∨ In y l ∨ In y r.
```

trick for emulating *assert false* in Coq

```
Definition assert_false := Leaf.
```

bal l x r acts as *create*, but performs one step of rebalancing if necessary, i.e. assumes $|height\ l$ - $height\ r| \leq 3$.

```
Definition bal l x r :=
  let hl := height l in
  let hr := height r in
  if gt_le_dec hl (hr+2) then
    match l with
     | Leaf ⇒ assert_false
     | Node ll lx lr _ ⇒
       if ge_lt_dec (height ll) (height lr) then
         create ll lx (create lr x r)
       else
         match lr with
          | Leaf ⇒ assert_false
          | Node lrl lrx lrr _ ⇒
               create (create ll lx lrl) lrx (create lrr x r)
         end
    end
  else
    if gt_le_dec hr (hl+2) then
      match r with
       | Leaf ⇒ assert_false
       | Node rl rx rr _ ⇒
```

```
          if ge_lt_dec (height rr) (height rl) then
            create (create l x rl) rx rr
          else
            match rl with
             | Leaf ⇒ assert_false
             | Node rll rlx rlr _ ⇒
                 create (create l x rll) rlx (create rlr rx rr)
            end
       end
     else
       create l x r.

Ltac bal_tac :=
 intros l x r;
 unfold bal;
 destruct (gt_le_dec (height l) (height r + 2));
   [ destruct l as [ |ll lx lr lh];
     [ | destruct (ge_lt_dec (height ll) (height lr));
           [ | destruct lr ] ]
   | destruct (gt_le_dec (height r) (height l + 2));
     [ destruct r as [ |rl rx rr rh];
           [ | destruct (ge_lt_dec (height rr) (height rl));
                 [ | destruct rl ] ]
     | ] ]; intros.

Lemma bal_bst : ∀ l x r, bst l → bst r →
 lt_tree x l → gt_tree x r → bst (bal l x r).

Lemma bal_avl : ∀ l x r, avl l → avl r →
 -(3) ≤ height l - height r ≤ 3 → avl (bal l x r).

Lemma bal_height_1 : ∀ l x r, avl l → avl r →
 -(3) ≤ height l - height r ≤ 3 →
 0 ≤ height (bal l x r) - max (height l) (height r) ≤ 1.

Lemma bal_height_2 :
 ∀ l x r, avl l → avl r → -(2) ≤ height l - height r ≤ 2 →
 height (bal l x r) == max (height l) (height r) +1.

Lemma bal_in : ∀ l x r y, avl l → avl r →
 (In y (bal l x r) ↔ X.eq y x ∨ In y l ∨ In y r).

Ltac omega_bal := match goal with
  | H:avl ?l, H':avl ?r ⊢ context [ bal ?l ?x ?r ] ⇒
     generalize (bal_height_1 l x r H H') (bal_height_2 l x r H H');
     omega_max
  end.
```

216.11 Insertion

```
Lemma add_avl_1 : ∀ s x, avl s →
 avl (add x s) ∧ 0 ≤ height (add x s) - height s ≤ 1.

Lemma add_avl : ∀ s x, avl s → avl (add x s).
Hint Resolve add_avl.

Lemma add_in : ∀ s x y, avl s →
 (In y (add x s) ↔ X.eq y x ∨ In y s).

Lemma add_bst : ∀ s x, bst s → avl s → bst (add x s).
```

216.12 Join

Same as *bal* but does not assume anything regarding heights of l and r.

```
Fixpoint join (l:t) : elt → t → t :=
  match l with
    | Leaf ⇒ add
    | Node ll lx lr lh ⇒ fun x ⇒
       fix join_aux (r:t) : t := match r with
           | Leaf ⇒ add x l
           | Node rl rx rr rh ⇒
                 if gt_le_dec lh (rh+2) then bal ll lx (join lr x r)
                 else if gt_le_dec rh (lh+2) then bal (join_aux rl) rx rr
                 else create l x r
           end
  end.

Ltac join_tac :=
 intro l; induction l as [| ll _lx lr Hlr lh];
   [ | intros x r; induction r as [| rl Hrl rx rr _rh]; unfold join;
     [ | destruct (gt_le_dec lh (rh+2));
       [ match goal with ⊢ context b [ bal ?a ?b ?c] ⇒
           replace (bal a b c)
           with (bal ll lx (join lr x (Node rl rx rr rh))); [ | auto]
         end
       | destruct (gt_le_dec rh (lh+2));
         [ match goal with ⊢ context b [ bal ?a ?b ?c] ⇒
             replace (bal a b c)
             with (bal (join (Node ll lx lr lh) x rl) rx rr); [ | auto]
           end
         | ] ] ] ]; intros.

Lemma join_avl_1 : ∀ l x r, avl l → avl r → avl (join l x r) ∧
 0≤ height (join l x r) - max (height l) (height r) ≤ 1.
```

Lemma *join_avl* : ∀ *l x r*, *avl l* → *avl r* → *avl* (*join l x r*).
Hint *Resolve join_avl*.

Lemma *join_in* : ∀ *l x r y*, *avl l* → *avl r* →
(*In y* (*join l x r*) ↔ *X.eq y x* ∨ *In y l* ∨ *In y r*).

Lemma *join_bst* : ∀ *l x r*, *bst l* → *avl l* → *bst r* → *avl r* →
lt_tree x l → *gt_tree x r* → *bst* (*join l x r*).

216.13 Extraction of minimum element

morally, *remove_min* is to be applied to a non-empty tree *t* = *Node l x r h*. Since we can't deal here with *assert false* for *t*=*Leaf*, we pre-unpack *t* (and forget about *h*).

Lemma *remove_min_avl_1* : ∀ *l x r h*, *avl* (*Node l x r h*) →
avl (*fst* (*remove_min l x r*)) ∧
0 ≤ *height* (*Node l x r h*) - *height* (*fst* (*remove_min l x r*)) ≤ 1.

Lemma *remove_min_avl* : ∀ *l x r h*, *avl* (*Node l x r h*) →
avl (*fst* (*remove_min l x r*)).

Lemma *remove_min_in* : ∀ *l x r h y*, *avl* (*Node l x r h*) →
(*In y* (*Node l x r h*) ↔
X.eq y (*snd* (*remove_min l x r*)) ∨ *In y* (*fst* (*remove_min l x r*))).

Lemma *remove_min_bst* : ∀ *l x r h*,
bst (*Node l x r h*) → *avl* (*Node l x r h*) → *bst* (*fst* (*remove_min l x r*)).

Lemma *remove_min_gt_tree* : ∀ *l x r h*,
bst (*Node l x r h*) → *avl* (*Node l x r h*) →
gt_tree (*snd* (*remove_min l x r*)) (*fst* (*remove_min l x r*)).

216.14 Merging two trees

merge t1 t2 builds the union of *t1* and *t2* assuming all elements of *t1* to be smaller than all elements of *t2*, and |*height t1* - *height t2*| ≤ 2.

Lemma *merge_avl_1* : ∀ *s1 s2*, *avl s1* → *avl s2* →
-(2) ≤ *height s1* - *height s2* ≤ 2 →
avl (*merge s1 s2*) ∧
0≤ *height* (*merge s1 s2*) - *max* (*height s1*) (*height s2*) ≤1.

Lemma *merge_avl* : ∀ *s1 s2*, *avl s1* → *avl s2* →
-(2) ≤ *height s1* - *height s2* ≤ 2 → *avl* (*merge s1 s2*).

Lemma *merge_in* : ∀ *s1 s2 y*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
(*In y* (*merge s1 s2*) ↔ *In y s1* ∨ *In y s2*).

Lemma *merge_bst* : ∀ *s1 s2*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
(∀ *y1 y2* : *elt*, *In y1 s1* → *In y2 s2* → *X.lt y1 y2*) →
bst (*merge s1 s2*).

216.15 Deletion

Lemma *remove_avl_1* : ∀ *s x*, *avl s* →
avl (*remove x s*) ∧ 0 ≤ *height s* - *height* (*remove x s*) ≤ 1.

Lemma *remove_avl* : ∀ *s x*, *avl s* → *avl* (*remove x s*).
Hint *Resolve remove_avl*.

Lemma *remove_in* : ∀ *s x y*, *bst s* → *avl s* →
(*In y* (*remove x s*) ↔ ¬ *X.eq y x* ∧ *In y s*).

Lemma *remove_bst* : ∀ *s x*, *bst s* → *avl s* → *bst* (*remove x s*).

216.16 Minimum element

Lemma *min_elt_1* : ∀ *s x*, *min_elt s* = *Some x* → *In x s*.

Lemma *min_elt_2* : ∀ *s x y*, *bst s* →
min_elt s = *Some x* → *In y s* → ¬ *X.lt y x*.

Lemma *min_elt_3* : ∀ *s*, *min_elt s* = *None* → *Empty s*.

216.17 Maximum element

Lemma *max_elt_1* : ∀ *s x*, *max_elt s* = *Some x* → *In x s*.

Lemma *max_elt_2* : ∀ *s x y*, *bst s* →
max_elt s = *Some x* → *In y s* → ¬ *X.lt x y*.

Lemma *max_elt_3* : ∀ *s*, *max_elt s* = *None* → *Empty s*.

216.18 Any element

Definition *choose* := *min_elt*.

Lemma *choose_1* : ∀ *s x*, *choose s* = *Some x* → *In x s*.

Lemma *choose_2* : ∀ *s*, *choose s* = *None* → *Empty s*.

216.19 Concatenation

Same as *merge* but does not assume anything about heights.

Lemma *concat_avl* : $\forall$ *s1 s2*, *avl s1* $\rightarrow$ *avl s2* $\rightarrow$ *avl* (*concat s1 s2*).

Lemma *concat_bst* : $\forall$ *s1 s2*, *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
($\forall$ *y1 y2* : *elt*, *In y1 s1* $\rightarrow$ *In y2 s2* $\rightarrow$ *X.lt y1 y2*) $\rightarrow$
bst (*concat s1 s2*).

Lemma *concat_in* : $\forall$ *s1 s2 y*, *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
($\forall$ *y1 y2* : *elt*, *In y1 s1* $\rightarrow$ *In y2 s2* $\rightarrow$ *X.lt y1 y2*) $\rightarrow$
(*In y* (*concat s1 s2*) $\leftrightarrow$ *In y s1* $\vee$ *In y s2*).

216.20 Splitting

split x s returns a triple (*l*, *present*, *r*) where

- *l* is the set of elements of *s* that are $< x$
- *r* is the set of elements of *s* that are $> x$
- *present* is *true* if and only if *s* contains *x*.

Lemma *split_avl* : $\forall$ *s x*, *avl s* $\rightarrow$
avl (*fst* (*split x s*)) $\wedge$ *avl* (*snd* (*snd* (*split x s*))).

Lemma *split_in_1* : $\forall$ *s x y*, *bst s* $\rightarrow$ *avl s* $\rightarrow$
(*In y* (*fst* (*split x s*)) $\leftrightarrow$ *In y s* $\wedge$ *X.lt y x*).

Lemma *split_in_2* : $\forall$ *s x y*, *bst s* $\rightarrow$ *avl s* $\rightarrow$
(*In y* (*snd* (*snd* (*split x s*))) $\leftrightarrow$ *In y s* $\wedge$ *X.lt x y*).

Lemma *split_in_3* : $\forall$ *s x*, *bst s* $\rightarrow$ *avl s* $\rightarrow$
(*fst* (*snd* (*split x s*)) = *true* $\leftrightarrow$ *In x s*).

Lemma *split_bst* : $\forall$ *s x*, *bst s* $\rightarrow$ *avl s* $\rightarrow$
bst (*fst* (*split x s*)) $\wedge$ *bst* (*snd* (*snd* (*split x s*))).

216.21 Intersection

```
Fixpoint inter (s1 s2 : t) {struct s1} : t := match s1, s2 with
    | Leaf,_ ⇒ Leaf
    | _,Leaf ⇒ Leaf
    | Node l1 x1 r1 h1, _ ⇒
            match split x1 s2 with
                | (l2',(true,r2')) ⇒ join (inter l1 l2') x1 (inter r1 r2')
```

```
                | (l2',(false,r2')) ⇒ concat (inter l1 l2') (inter r1 r2')
            end
    end.
```

Lemma *inter_avl* : ∀ *s1 s2*, *avl s1* → *avl s2* → *avl* (*inter s1 s2*).

Lemma *inter_bst_in* : ∀ *s1 s2*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
bst (*inter s1 s2*) ∧ (∀ *y*, *In y* (*inter s1 s2*) ↔ *In y s1* ∧ *In y s2*).

Lemma *inter_bst* : ∀ *s1 s2*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
bst (*inter s1 s2*).

Lemma *inter_in* : ∀ *s1 s2 y*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
(*In y* (*inter s1 s2*) ↔ *In y s1* ∧ *In y s2*).

216.22 Difference

```
Fixpoint diff (s1 s2 : t) { struct s1 } : t := match s1, s2 with
 | Leaf, _ ⇒ Leaf
 | _, Leaf ⇒ s1
 | Node l1 x1 r1 h1, _ ⇒
     match split x1 s2 with
       | (l2',(true,r2')) ⇒ concat (diff l1 l2') (diff r1 r2')
       | (l2',(false,r2')) ⇒ join (diff l1 l2') x1 (diff r1 r2')
     end
   end.
```

Lemma *diff_avl* : ∀ *s1 s2*, *avl s1* → *avl s2* → *avl* (*diff s1 s2*).

Lemma *diff_bst_in* : ∀ *s1 s2*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
bst (*diff s1 s2*) ∧ (∀ *y*, *In y* (*diff s1 s2*) ↔ *In y s1* ∧ ¬*In y s2*).

Lemma *diff_bst* : ∀ *s1 s2*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
bst (*diff s1 s2*).

Lemma *diff_in* : ∀ *s1 s2 y*, *bst s1* → *avl s1* → *bst s2* → *avl s2* →
(*In y* (*diff s1 s2*) ↔ *In y s1* ∧ ¬*In y s2*).

216.23 Elements

elements_tree_aux acc t catenates the elements of *t* in infix order to the list *acc*

```
Fixpoint elements_aux (acc : list X.t) (t : tree) {struct t} : list X.t :=
  match t with
   | Leaf ⇒ acc
   | Node l x r _ ⇒ elements_aux (x :: elements_aux acc r) l
  end.
```

then *elements* is an instanciation with an empty *acc*

Definition *elements* := *elements_aux nil*.

Lemma *elements_aux_in* : ∀ *s acc x*,
InA X.eq x (*elements_aux acc s*) ↔ *In x s* ∨ *InA X.eq x acc*.

Lemma *elements_in* : ∀ *s x*, *InA X.eq x* (*elements s*) ↔ *In x s*.

Lemma *elements_aux_sort* : ∀ *s acc*, *bst s* → *sort X.lt acc* →
(∀ *x y* : *elt*, *InA X.eq x acc* → *In y s* → *X.lt y x*) →
sort X.lt (*elements_aux acc s*).

Lemma *elements_sort* : ∀ *s* : *tree*, *bst s* → *sort X.lt* (*elements s*).
Hint *Resolve elements_sort*.

216.24 Filter

Section *F*.
Variable *f* : *elt* → *bool*.

Fixpoint *filter_acc* (*acc*:*t*)(*s*:*t*) { *struct s* } : *t* := match *s* with
| *Leaf* ⇒ *acc*
| *Node l x r h* ⇒
filter_acc (*filter_acc* (if *f x* then *add x acc* else *acc*) *l*) *r*
end.

Definition *filter* := *filter_acc Leaf*.

Lemma *filter_acc_avl* : ∀ *s acc*, *avl s* → *avl acc* →
avl (*filter_acc acc s*).
Hint *Resolve filter_acc_avl*.

Lemma *filter_acc_bst* : ∀ *s acc*, *bst s* → *avl s* → *bst acc* → *avl acc* →
bst (*filter_acc acc s*).

Lemma *filter_acc_in* : ∀ *s acc*, *avl s* → *avl acc* →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*filter_acc acc s*) ↔ *In x acc* ∨ *In x s* ∧ *f x* = *true*.

Lemma *filter_avl* : ∀ *s*, *avl s* → *avl* (*filter s*).

Lemma *filter_bst* : ∀ *s*, *bst s* → *avl s* → *bst* (*filter s*).

Lemma *filter_in* : ∀ *s*, *avl s* →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*filter s*) ↔ *In x s* ∧ *f x* = *true*.

216.25 Partition

Fixpoint *partition_acc* (*acc* : *t*×*t*)(*s* : *t*) { *struct s* } : *t*×*t* :=
match *s* with

```
  | Leaf ⇒ acc
  | Node l x r _ ⇒
    let (acct,accf) := acc in
    partition_acc
      (partition_acc
        (if f x then (add x acct, accf) else (acct, add x accf)) l) r
end.
```

Definition *partition* := *partition_acc* (*Leaf*,*Leaf*).

Lemma *partition_acc_avl_1* : ∀ *s acc*, *avl s* →
avl (*fst acc*) → *avl* (*fst* (*partition_acc acc s*)).

Lemma *partition_acc_avl_2* : ∀ *s acc*, *avl s* →
avl (*snd acc*) → *avl* (*snd* (*partition_acc acc s*)).
Hint *Resolve partition_acc_avl_1 partition_acc_avl_2*.

Lemma *partition_acc_bst_1* : ∀ *s acc*, *bst s* → *avl s* →
bst (*fst acc*) → *avl* (*fst acc*) →
bst (*fst* (*partition_acc acc s*)).

Lemma *partition_acc_bst_2* : ∀ *s acc*, *bst s* → *avl s* →
bst (*snd acc*) → *avl* (*snd acc*) →
bst (*snd* (*partition_acc acc s*)).

Lemma *partition_acc_in_1* : ∀ *s acc*, *avl s* → *avl* (*fst acc*) →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*fst* (*partition_acc acc s*)) ↔
In x (*fst acc*) ∨ *In x s* ∧ *f x* = *true*.

Lemma *partition_acc_in_2* : ∀ *s acc*, *avl s* → *avl* (*snd acc*) →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*snd* (*partition_acc acc s*)) ↔
In x (*snd acc*) ∨ *In x s* ∧ *f x* = *false*.

Lemma *partition_avl_1* : ∀ *s*, *avl s* → *avl* (*fst* (*partition s*)).

Lemma *partition_avl_2* : ∀ *s*, *avl s* → *avl* (*snd* (*partition s*)).

Lemma *partition_bst_1* : ∀ *s*, *bst s* → *avl s* →
bst (*fst* (*partition s*)).

Lemma *partition_bst_2* : ∀ *s*, *bst s* → *avl s* →
bst (*snd* (*partition s*)).

Lemma *partition_in_1* : ∀ *s*, *avl s* →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*fst* (*partition s*)) ↔ *In x s* ∧ *f x* = *true*.

Lemma *partition_in_2* : ∀ *s*, *avl s* →
compat_bool X.eq f → ∀ *x* : *elt*,
In x (*snd* (*partition s*)) ↔ *In x s* ∧ *f x* = *false*.

for_all and ∃

```
Fixpoint for_all (s:t) : bool := match s with
  | Leaf ⇒ true
  | Node l x r _ ⇒ f x && for_all l && for_all r
  end.

Lemma for_all_1 : ∀ s, compat_bool E.eq f →
 For_all (fun x ⇒ f x = true) s → for_all s = true.

Lemma for_all_2 : ∀ s, compat_bool E.eq f →
 for_all s = true → For_all (fun x ⇒ f x = true) s.

Fixpoint exists_ (s:t) : bool := match s with
  | Leaf ⇒ false
  | Node l x r _ ⇒ f x || exists_ l || exists_ r
  end.

Lemma exists_1 : ∀ s, compat_bool E.eq f →
 Exists (fun x ⇒ f x = true) s → exists_ s = true.

Lemma exists_2 : ∀ s, compat_bool E.eq f →
 exists_ s = true → Exists (fun x ⇒ f x = true) s.

End F.
```

216.26 Fold

```
Module L := FSetList.Raw X.

Fixpoint fold (A : Set) (f : elt → A → A)(s : tree) {struct s} : A → A :=
 fun a ⇒ match s with
  | Leaf ⇒ a
  | Node l x r _ ⇒ fold A f r (f x (fold A f l a))
 end.
Implicit Arguments fold [A].

Definition fold' (A : Set) (f : elt → A → A)(s : tree) :=
  L.fold f (elements s).
Implicit Arguments fold' [A].

Lemma fold_equiv_aux :
 ∀ (A : Set) (s : tree) (f : elt → A → A) (a : A) (acc : list elt),
 L.fold f (elements_aux acc s) a = L.fold f acc (fold f s a).

Lemma fold_equiv :
 ∀ (A : Set) (s : tree) (f : elt → A → A) (a : A),
 fold f s a = fold' f s a.

Lemma fold_1 :
 ∀ (s:t)(Hs:bst s)(A : Set)(f : elt → A → A)(i : A),
 fold f s i = fold_left (fun a e ⇒ f e a) (elements s) i.
```

216.27 Cardinal

Fixpoint *cardinal* (*s* : *tree*) : *nat* :=
  match *s* with
    | *Leaf* ⇒ 0%*nat*
    | *Node l _ r _* ⇒ *S* (*cardinal l* + *cardinal r*)
  end.

Lemma *cardinal_elements_aux_1* :
∀ *s acc*, (*length acc* + *cardinal s*)%*nat* = *length* (*elements_aux acc s*).

Lemma *cardinal_elements_1* : ∀ *s* : *tree*, *cardinal s* = *length* (*elements s*).

NB: the remaining functions (union, subset, compare) are still defined in a dependent style, due to the use of well-founded induction.

Induction over cardinals

Lemma *sorted_subset_cardinal* : ∀ *l' l* : *list X.t*,
sort X.lt l → *sort X.lt l'* →
(∀ *x* : *elt*, *InA X.eq x l* → *InA X.eq x l'*) → (*length l* ≤ *length l'*)%*nat*.

Lemma *cardinal_subset* : ∀ *a b* : *tree*, *bst a* → *bst b* →
(∀ *y* : *elt*, *In y a* → *In y b*) →
(*cardinal a* ≤ *cardinal b*)%*nat*.

Lemma *cardinal_left* : ∀ (*l r* : *tree*) (*x* : *elt*) (*h* : *int*),
(*cardinal l* < *cardinal* (*Node l x r h*))%*nat*.

Lemma *cardinal_right* :
∀ (*l r* : *tree*) (*x* : *elt*) (*h* : *int*),
(*cardinal r* < *cardinal* (*Node l x r h*))%*nat*.

Lemma *cardinal_rec2* : ∀ *P* : *tree* → *tree* → Set,
(∀ *s1 s2* : *tree*,
  (∀ *t1 t2* : *tree*,
    (*cardinal t1* + *cardinal t2* < *cardinal s1* + *cardinal s2*)%*nat* → *P t1 t2*)
  → *P s1 s2*) →
∀ *s1 s2* : *tree*, *P s1 s2*.

Lemma *height_0* : ∀ *s*, *avl s* → *height s* = 0 → *s* = *Leaf*.

216.28 Union

union s1 s2 does an induction over the sum of the cardinals of *s1* and *s2*. Code is

```
let rec union s1 s2 =
  match (s1, s2) with
    (Empty, t2) -> t2
  | (t1, Empty) -> t1
```

```
    | (Node(l1, v1, r1, h1), Node(l2, v2, r2, h2)) ->
        if h1 >= h2 then
          if h2 = 1 then add v2 s1 else begin
            let (l2', _, r2') = split v1 s2 in
            join (union l1 l2') v1 (union r1 r2')
          end
        else
          if h1 = 1 then add v1 s2 else begin
            let (l1', _, r1') = split v2 s1 in
            join (union l1' l2) v2 (union r1' r2)
          end
```

Definition *union* :
$\forall$ *s1 s2*, *bst s1* $\rightarrow$ *avl s1* $\rightarrow$ *bst s2* $\rightarrow$ *avl s2* $\rightarrow$
$\{s' : t \mid bst\ s' \wedge avl\ s' \wedge \forall\ x : elt, In\ x\ s' \leftrightarrow In\ x\ s1 \vee In\ x\ s2\}$.

216.29 Subset

```
  let rec subset s1 s2 =
    match (s1, s2) with
      Empty, _ -> true
    | _, Empty -> false
    | Node (l1, v1, r1, _), (Node (l2, v2, r2, _) as t2) ->
        let c = Ord.compare v1 v2 in
        if c = 0 then
          subset l1 l2 && subset r1 r2
        else if c < 0 then
          subset (Node (l1, v1, Empty, 0)) l2 && subset r1 t2
        else
          subset (Node (Empty, v1, r1, 0)) r2 && subset l1 t2
```

Definition *subset* : $\forall$ *s1 s2* : *t*, *bst s1* $\rightarrow$ *bst s2* $\rightarrow$
$\{Subset\ s1\ s2\} + \{\tilde{}\ Subset\ s1\ s2\}$.

216.30 Comparison

216.30.1 Relations *eq* and *lt* over trees

Definition *eq* : $t \rightarrow t \rightarrow$ Prop := *Equal*.

Lemma *eq_refl* : $\forall$ *s* : *t*, *eq s s*.

Lemma *eq_sym* : $\forall$ *s s'* : *t*, *eq s s'* $\rightarrow$ *eq s' s*.

Lemma *eq_trans* : ∀ *s s' s"* : *t*, *eq s s'* → *eq s' s"* → *eq s s"*.

Lemma *eq_L_eq* :
∀ *s s'* : *t*, *eq s s'* → *L.eq* (*elements s*) (*elements s'*).

Lemma *L_eq_eq* :
∀ *s s'* : *t*, *L.eq* (*elements s*) (*elements s'*) → *eq s s'*.
Hint *Resolve eq_L_eq L_eq_eq.*

Definition *lt* (*s1 s2* : *t*) : Prop := *L.lt* (*elements s1*) (*elements s2*).

Definition *lt_trans* (*s s' s"* : *t*) (*h* : *lt s s'*)
(*h'* : *lt s' s"*) : *lt s s"* := *L.lt_trans h h'*.

Lemma *lt_not_eq* : ∀ *s s'* : *t*, *bst s* → *bst s'* → *lt s s'* → ¬ *eq s s'*.

A new comparison algorithm suggested by Xavier Leroy:

type enumeration = End | More of elt * t * enumeration

let rec cons s e = match s with | Empty -> e | Node(l, v, r, _) -> cons l (More(v, r, e))

let rec compare_aux e1 e2 = match (e1, e2) with | (End, End) -> 0 | (End, More _) -> -1 | (More _, End) -> 1 | (More(v1, r1, k1), More(v2, r2, k2)) -> let c = Ord.compare v1 v2 in if c <> 0 then c else compare_aux (cons r1 k1) (cons r2 k2)

let compare s1 s2 = compare_aux (cons s1 End) (cons s2 End)

216.30.2 Enumeration of the elements of a tree

Inductive *enumeration* : Set :=
| End : *enumeration*
| *More* : *elt* → *tree* → *enumeration* → *enumeration*.

flatten_e e returns the list of elements of *e* i.e. the list of elements actually compared

Fixpoint *flatten_e* (*e* : *enumeration*) : *list elt* := match *e* with
| End ⇒ *nil*
| *More x t r* ⇒ *x* :: *elements t* ++ *flatten_e r*
end.

sorted_e e expresses that elements in the enumeration *e* are sorted, and that all trees in *e* are binary search trees.

Inductive *In_e* (*x*:*elt*) : *enumeration* → Prop :=
| *InEHd1* :
∀ (*y* : *elt*) (*s* : *tree*) (*e* : *enumeration*),
X.eq x y → *In_e x* (*More y s e*)
| *InEHd2* :
∀ (*y* : *elt*) (*s* : *tree*) (*e* : *enumeration*),
In x s → *In_e x* (*More y s e*)
| *InETl* :
∀ (*y* : *elt*) (*s* : *tree*) (*e* : *enumeration*),

In_e x e → In_e x (More y s e).

Hint *Constructors In_e.*

Inductive *sorted_e : enumeration → Prop :=*
| *SortedEEnd : sorted_e* End
| *SortedEMore :*
∀ (x : elt) (s : tree) (e : enumeration),
bst s →
(gt_tree x s) →
sorted_e e →
(∀ y : elt, In_e y e → X.lt x y) →
(∀ y : elt,
In y s → ∀ z : elt, In_e z e → X.lt y z) →
sorted_e (More x s e).

Hint *Constructors sorted_e.*

Lemma *in_app :*
∀ (x : elt) (l1 l2 : list elt),
InA X.eq x (l1 ++ l2) → InA X.eq x l1 ∨ InA X.eq x l2.

Lemma *in_flatten_e :*
∀ (x : elt) (e : enumeration), InA X.eq x (flatten_e e) → In_e x e.

Lemma *sort_app :*
∀ l1 l2 : list elt, sort X.lt l1 → sort X.lt l2 →
(∀ x y : elt, InA X.eq x l1 → InA X.eq y l2 → X.lt x y) →
sort X.lt (l1 ++ l2).

Lemma *sorted_flatten_e :*
∀ e : enumeration, sorted_e e → sort X.lt (flatten_e e).

Lemma *elements_app :*
∀ (s : tree) (acc : list elt), elements_aux acc s = elements s ++ acc.

Lemma *compare_flatten_1 :*
∀ (t0 t2 : tree) (t1 : elt) (z : int) (l : list elt),
elements t0 ++ t1 :: elements t2 ++ l =
elements (Node t0 t1 t2 z) ++ l.

key lemma for correctness

Lemma *flatten_e_elements :*
∀ (x : elt) (l r : tree) (z : int) (e : enumeration),
elements l ++ flatten_e (More x r e) = elements (Node l x r z) ++ flatten_e e.

termination of *compare_aux*

Open Local *Scope Z_scope.*

Fixpoint *measure_e_t (s : tree) : Z :=* match *s* with
| *Leaf ⇒ 0*
| *Node l _ r _ ⇒ 1 + measure_e_t l + measure_e_t r*

```
 end.

Fixpoint measure_e (e : enumeration) : Z := match e with
  | End ⇒ 0
  | More _ s r ⇒ 1 + measure_e_t s + measure_e r
 end.

Ltac Measure_e_t := unfold measure_e_t in ⊢ ×; fold measure_e_t in ⊢ ×.
Ltac Measure_e := unfold measure_e in ⊢ ×; fold measure_e in ⊢ ×.

Lemma measure_e_t_0 : ∀ s : tree, measure_e_t s ≥ 0.

Ltac Measure_e_t_0 s := generalize (measure_e_t_0 s); intro.

Lemma measure_e_0 : ∀ e : enumeration, measure_e e ≥ 0.

Ltac Measure_e_0 e := generalize (measure_e_0 e); intro.
```

Induction principle over the sum of the measures for two lists

```
Definition compare_rec2 :
 ∀ P : enumeration → enumeration → Set,
  (∀ x x' : enumeration,
   (∀ y y' : enumeration,
    measure_e y + measure_e y' < measure_e x + measure_e x' → P y y') →
   P x x') →
 ∀ x x' : enumeration, P x x'.
```

cons t e adds the elements of tree *t* on the head of enumeration *e*. Code:

```
    let rec cons s e = match s with | Empty -> e | Node(l, v, r, _) -> cons l (More(v, r, e))
```

```
Definition cons : ∀ (s : tree) (e : enumeration), bst s → sorted_e e →
  (∀ (x y : elt), In x s → In_e y e → X.lt x y) →
  { r : enumeration
  | sorted_e r ∧
    measure_e r = measure_e_t s + measure_e e ∧
    flatten_e r = elements s ++ flatten_e e
  }.

Lemma l_eq_cons :
 ∀ (l1 l2 : list elt) (x y : elt),
 X.eq x y → L.eq l1 l2 → L.eq (x :: l1) (y :: l2).

Definition compare_aux :
 ∀ e1 e2 : enumeration, sorted_e e1 → sorted_e e2 →
 Compare L.lt L.eq (flatten_e e1) (flatten_e e2).

Definition compare : ∀ s1 s2, bst s1 → bst s2 → Compare lt eq s1 s2.
```

216.31 Equality test

```
Definition equal : ∀ s s' : t, bst s → bst s' → {Equal s s'} + {~ Equal s s'}.
```

We provide additionally a different implementation for union, subset and equal, which is less efficient, but uses structural induction, hence computes within Coq.

Alternative union based on fold. Complexity : $min(|s|,|s'|)*log(max(|s|,|s'|))$

Definition *union' s s'* :=
  if *ge_lt_dec* (*height s*) (*height s'*) then *fold add s' s* else *fold add s s'*.

Lemma *fold_add_avl* : ∀ *s s'*, *avl s* → *avl s'* → *avl* (*fold add s s'*).
Hint *Resolve fold_add_avl*.

Lemma *union'_avl* : ∀ *s s'*, *avl s* → *avl s'* → *avl* (*union' s s'*).

Lemma *fold_add_bst* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 bst (*fold add s s'*).

Lemma *union'_bst* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 bst (*union' s s'*).

Lemma *fold_add_in* : ∀ *s s' y*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 (*In y* (*fold add s s'*) ↔ *In y s* ∨ *In y s'*).

Lemma *union'_in* : ∀ *s s' y*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 (*In y* (*union' s s'*) ↔ *In y s* ∨ *In y s'*).

Alternative subset based on diff.

Definition *subset' s s'* := *is_empty* (*diff s s'*).

Lemma *subset'_1* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 Subset s s' → *subset' s s'* = *true*.

Lemma *subset'_2* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 subset' s s' = *true* → *Subset s s'*.

Alternative equal based on subset

Definition *equal' s s'* := *subset' s s'* && *subset' s' s*.

Lemma *equal'_1* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 Equal s s' → *equal' s s'* = *true*.

Lemma *equal'_2* : ∀ *s s'*, *bst s* → *avl s* → *bst s'* → *avl s'* →
 equal' s s' = *true* → *Equal s s'*.

End *Raw*.

216.32 Encapsulation

Now, in order to really provide a functor implementing S, we need to encapsulate everything into a type of balanced binary search trees.

Module *IntMake* (*I*:*Int*)(*X*: *OrderedType*) <: *S* with Module *E* := *X*.

  Module *E* := *X*.

```
Module Raw := Raw I X.

Record bbst : Set := Bbst {this :> Raw.t; is_bst : Raw.bst this; is_avl: Raw.avl this}.
Definition t := bbst.
Definition elt := E.t.

Definition In (x : elt) (s : t) : Prop := Raw.In x s.
Definition Equal (s s':t) : Prop := ∀ a : elt, In a s ↔ In a s'.
Definition Subset (s s':t) : Prop := ∀ a : elt, In a s → In a s'.
Definition Empty (s:t) : Prop := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) (s:t) : Prop := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) (s:t) : Prop := ∃ x, In x s ∧ P x.

Lemma In_1 : ∀ (s:t)(x y:elt), E.eq x y → In x s → In y s.

Definition mem (x:elt)(s:t) : bool := Raw.mem x s.

Definition empty : t := Bbst _ Raw.empty_bst Raw.empty_avl.
Definition is_empty (s:t) : bool := Raw.is_empty s.
Definition singleton (x:elt) : t := Bbst _ (Raw.singleton_bst x) (Raw.singleton_avl x).
Definition add (x:elt)(s:t) : t :=
  Bbst _ (Raw.add_bst s x (is_bst s) (is_avl s))
         (Raw.add_avl s x (is_avl s)).
Definition remove (x:elt)(s:t) : t :=
  Bbst _ (Raw.remove_bst s x (is_bst s) (is_avl s))
         (Raw.remove_avl s x (is_avl s)).
Definition inter (s s':t) : t :=
  Bbst _ (Raw.inter_bst _ _ (is_bst s) (is_avl s) (is_bst s') (is_avl s'))
         (Raw.inter_avl _ _ (is_avl s) (is_avl s')).
Definition diff (s s':t) : t :=
  Bbst _ (Raw.diff_bst _ _ (is_bst s) (is_avl s) (is_bst s') (is_avl s'))
         (Raw.diff_avl _ _ (is_avl s) (is_avl s')).
Definition elements (s:t) : list elt := Raw.elements s.
Definition min_elt (s:t) : option elt := Raw.min_elt s.
Definition max_elt (s:t) : option elt := Raw.max_elt s.
Definition choose (s:t) : option elt := Raw.choose s.
Definition fold (B : Set) (f : elt → B → B) (s:t) : B → B := Raw.fold f s.
Definition cardinal (s:t) : nat := Raw.cardinal s.
Definition filter (f : elt → bool) (s:t) : t :=
  Bbst _ (Raw.filter_bst f _ (is_bst s) (is_avl s))
         (Raw.filter_avl f _ (is_avl s)).
Definition for_all (f : elt → bool) (s:t) : bool := Raw.for_all f s.
Definition exists_ (f : elt → bool) (s:t) : bool := Raw.exists_ f s.
Definition partition (f : elt → bool) (s:t) : t × t :=
  let p := Raw.partition f s in
  (Bbst (fst p) (Raw.partition_bst_1 f _ (is_bst s) (is_avl s))
                (Raw.partition_avl_1 f _ (is_avl s)),
   Bbst (snd p) (Raw.partition_bst_2 f _ (is_bst s) (is_avl s))
                (Raw.partition_avl_2 f _ (is_avl s))).
```

```
Definition union (s s':t) : t :=
  let (u,p) := Raw.union _ _ (is_bst s) (is_avl s) (is_bst s') (is_avl s') in
  let (b,p) := p in
  let (a,_) := p in
  Bbst u b a.

Definition union' (s s' : t) : t :=
  Bbst _ (Raw.union'_bst _ _ (is_bst s) (is_avl s) (is_bst s') (is_avl s'))
         (Raw.union'_avl _ _ (is_avl s) (is_avl s')).

Definition equal (s s': t) : bool := if Raw.equal _ _ (is_bst s) (is_bst s') then true else false.
Definition equal' (s s':t) : bool := Raw.equal' s s'.

Definition subset (s s':t) : bool := if Raw.subset _ _ (is_bst s) (is_bst s') then true else false.
Definition subset' (s s':t) : bool := Raw.subset' s s'.

Definition eq (s s':t) : Prop := Raw.eq s s'.
Definition lt (s s':t) : Prop := Raw.lt s s'.

Definition compare (s s':t) : Compare lt eq s s'.

Section Specs.
Variable s s' s'': t.
Variable x y : elt.

Hint Resolve is_bst is_avl.

Lemma mem_1 : In x s → mem x s = true.
Lemma mem_2 : mem x s = true → In x s.

Lemma equal_1 : Equal s s' → equal s s' = true.

Lemma equal_2 : equal s s' = true → Equal s s'.

Lemma equal'_1 : Equal s s' → equal' s s' = true.
Lemma equal'_2 : equal' s s' = true → Equal s s'.

Lemma subset_1 : Subset s s' → subset s s' = true.

Lemma subset_2 : subset s s' = true → Subset s s'.

Lemma subset'_1 : Subset s s' → subset' s s' = true.
Lemma subset'_2 : subset' s s' = true → Subset s s'.

Lemma empty_1 : Empty empty.

Lemma is_empty_1 : Empty s → is_empty s = true.
Lemma is_empty_2 : is_empty s = true → Empty s.

Lemma add_1 : E.eq x y → In y (add x s).

Lemma add_2 : In y s → In y (add x s).

Lemma add_3 : ¬ E.eq x y → In y (add x s) → In y s.

Lemma remove_1 : E.eq x y → ¬ In y (remove x s).

Lemma remove_2 : ¬ E.eq x y → In y s → In y (remove x s).
```

Lemma *remove_3* : *In y* (*remove x s*) → *In y s*.

Lemma *singleton_1* : *In y* (*singleton x*) → *E.eq x y*.
Lemma *singleton_2* : *E.eq x y* → *In y* (*singleton x*).

Lemma *union_1* : *In x* (*union s s'*) → *In x s* ∨ *In x s'*.

Lemma *union_2* : *In x s* → *In x* (*union s s'*).

Lemma *union_3* : *In x s'* → *In x* (*union s s'*).

Lemma *union'_1* : *In x* (*union' s s'*) → *In x s* ∨ *In x s'*.

Lemma *union'_2* : *In x s* → *In x* (*union' s s'*).

Lemma *union'_3* : *In x s'* → *In x* (*union' s s'*).

Lemma *inter_1* : *In x* (*inter s s'*) → *In x s*.

Lemma *inter_2* : *In x* (*inter s s'*) → *In x s'*.

Lemma *inter_3* : *In x s* → *In x s'* → *In x* (*inter s s'*).

Lemma *diff_1* : *In x* (*diff s s'*) → *In x s*.

Lemma *diff_2* : *In x* (*diff s s'*) → ¬ *In x s'*.

Lemma *diff_3* : *In x s* → ¬ *In x s'* → *In x* (*diff s s'*).

Lemma *fold_1* : ∀ (*A* : Set) (*i* : *A*) (*f* : *elt* → *A* → *A*),
fold A f s i = *fold_left* (fun *a e* ⇒ *f e a*) (*elements s*) *i*.

Lemma *cardinal_1* : *cardinal s* = *length* (*elements s*).

Section *Filter*.
Variable *f* : *elt* → *bool*.

Lemma *filter_1* : *compat_bool E.eq f* → *In x* (*filter f s*) → *In x s*.

Lemma *filter_2* : *compat_bool E.eq f* → *In x* (*filter f s*) → *f x* = *true*.

Lemma *filter_3* : *compat_bool E.eq f* → *In x s* → *f x* = *true* → *In x* (*filter f s*).

Lemma *for_all_1* : *compat_bool E.eq f* → *For_all* (fun *x* ⇒ *f x* = *true*) *s* → *for_all f s* = *true*.
Lemma *for_all_2* : *compat_bool E.eq f* → *for_all f s* = *true* → *For_all* (fun *x* ⇒ *f x* = *true*) *s*.

Lemma *exists_1* : *compat_bool E.eq f* → *Exists* (fun *x* ⇒ *f x* = *true*) *s* → *exists_ f s* = *true*.
Lemma *exists_2* : *compat_bool E.eq f* → *exists_ f s* = *true* → *Exists* (fun *x* ⇒ *f x* = *true*) *s*.

Lemma *partition_1* : *compat_bool E.eq f* →
Equal (*fst* (*partition f s*)) (*filter f s*).

Lemma *partition_2* : *compat_bool E.eq f* →
Equal (*snd* (*partition f s*)) (*filter* (fun *x* ⇒ *negb* (*f x*)) *s*).

End *Filter*.

Lemma *elements_1* : *In x s* → *InA E.eq x* (*elements s*).

Lemma *elements_2* : *InA E.eq x* (*elements s*) → *In x s*.

Lemma *elements_3* : *sort E.lt* (*elements s*).

Lemma *min_elt_1* : *min_elt s* = *Some x* → *In x s*.
Lemma *min_elt_2* : *min_elt s* = *Some x* → *In y s* → ¬ *E.lt y x*.
Lemma *min_elt_3* : *min_elt s* = *None* → *Empty s*.

Lemma *max_elt_1* : *max_elt s* = *Some x* → *In x s*.
Lemma *max_elt_2* : *max_elt s* = *Some x* → *In y s* → ¬ *E.lt x y*.
Lemma *max_elt_3* : *max_elt s* = *None* → *Empty s*.

Lemma *choose_1* : *choose s* = *Some x* → *In x s*.
Lemma *choose_2* : *choose s* = *None* → *Empty s*.

Lemma *eq_refl* : *eq s s*.
Lemma *eq_sym* : *eq s s'* → *eq s' s*.
Lemma *eq_trans* : *eq s s'* → *eq s' s"* → *eq s s"*.

Lemma *lt_trans* : *lt s s'* → *lt s' s"* → *lt s s"*.
Lemma *lt_not_eq* : *lt s s'* → ¬*eq s s'*.

End *Specs*.
End *IntMake*.

Module *Make* (*X*: *OrderedType*) <: *S* with Module *E* := *X*
 :=*IntMake*(*Z_as_Int*)(*X*).

Chapter 217

Module Coq.FSets.FSetBridge

217.1 Finite sets library

This module implements bridges (as functors) from dependent to/from non-dependent set signature.

Require Export *FSetInterface*.

217.2 From non-dependent signature *S* to dependent signature *Sdep*.

Module *DepOfNodep* (*M*: *S*) <: *Sdep* with Module *E* := *M.E*.
  Import *M*.

  Module *ME* := *OrderedTypeFacts E*.

  Definition *empty* : {*s* : *t* | *Empty s*}.

  Definition *is_empty* : ∀ *s* : *t*, {*Empty s*} + {~ *Empty s*}.

  Definition *mem* : ∀ (*x* : *elt*) (*s* : *t*), {*In x s*} + {~ *In x s*}.

  Definition *Add* (*x* : *elt*) (*s s'* : *t*) :=
    ∀ *y* : *elt*, *In y s'* ↔ *E.eq x y* ∨ *In y s*.

  Definition *add* : ∀ (*x* : *elt*) (*s* : *t*), {*s'* : *t* | *Add x s s'*}.

  Definition *singleton* :
    ∀ *x* : *elt*, {*s* : *t* | ∀ *y* : *elt*, *In y s* ↔ *E.eq x y*}.

  Definition *remove* :
    ∀ (*x* : *elt*) (*s* : *t*),
    {*s'* : *t* | ∀ *y* : *elt*, *In y s'* ↔ ¬ *E.eq x y* ∧ *In y s*}.

  Definition *union* :
    ∀ *s s'* : *t*, {*s"* : *t* | ∀ *x* : *elt*, *In x s"* ↔ *In x s* ∨ *In x s'*}.

  Definition *inter* :
    ∀ *s s'* : *t*, {*s"* : *t* | ∀ *x* : *elt*, *In x s"* ↔ *In x s* ∧ *In x s'*}.

Definition *diff* :
$\forall\ s\ s' : t, \{s'' : t \mid \forall\ x : elt, In\ x\ s'' \leftrightarrow In\ x\ s \wedge \neg\ In\ x\ s'\}$.

Definition *equal* : $\forall\ s\ s' : t, \{Equal\ s\ s'\} + \{\tilde{}\ Equal\ s\ s'\}$.

Definition *subset* : $\forall\ s\ s' : t, \{Subset\ s\ s'\} + \{\tilde{}Subset\ s\ s'\}$.

Definition *elements* :
$\forall\ s : t,$
$\{l : list\ elt \mid sort\ E.lt\ l \wedge (\forall\ x : elt, In\ x\ s \leftrightarrow InA\ E.eq\ x\ l)\}$.

Definition *fold* :
$\forall\ (A : \mathsf{Set})\ (f : elt \to A \to A)\ (s : t)\ (i : A),$
$\{r : A \mid \mathsf{let}\ (l,\_) := elements\ s\ \mathsf{in}$
$r = fold\_left\ (\mathsf{fun}\ a\ e \Rightarrow f\ e\ a)\ l\ i\}$.

Definition *cardinal* :
$\forall\ s : t,$
$\{r : nat \mid \mathsf{let}\ (l,\_) := elements\ s\ \mathsf{in}\ r = length\ l\ \}$.

Definition *fdec* $(P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})$
$(x : elt) := \mathsf{if}\ Pdec\ x\ \mathsf{then}\ true\ \mathsf{else}\ false$.

Lemma *compat_P_aux* :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\}),$
$compat\_P\ E.eq\ P \to compat\_bool\ E.eq\ (fdec\ Pdec)$.

Hint *Resolve compat_P_aux*.

Definition *filter* :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})\ (s : t),$
$\{s' : t \mid compat\_P\ E.eq\ P \to \forall\ x : elt, In\ x\ s' \leftrightarrow In\ x\ s \wedge P\ x\}$.

Definition *for_all* :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})\ (s : t),$
$\{compat\_P\ E.eq\ P \to For\_all\ P\ s\} + \{compat\_P\ E.eq\ P \to \neg\ For\_all\ P\ s\}$.

Definition *exists_* :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})\ (s : t),$
$\{compat\_P\ E.eq\ P \to Exists\ P\ s\} + \{compat\_P\ E.eq\ P \to \neg\ Exists\ P\ s\}$.

Definition *partition* :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})\ (s : t),$
$\{partition : t \times t \mid$
$\mathsf{let}\ (s1, s2) := partition\ \mathsf{in}$
$compat\_P\ E.eq\ P \to$
$For\_all\ P\ s1 \wedge$
$For\_all\ (\mathsf{fun}\ x \Rightarrow \neg\ P\ x)\ s2 \wedge$
$(\forall\ x : elt, In\ x\ s \leftrightarrow In\ x\ s1 \vee In\ x\ s2)\}$.

Definition *choose* : $\forall\ s : t, \{x : elt \mid In\ x\ s\} + \{Empty\ s\}$.

Definition *min_elt* :
$\forall\ s : t,$

$\{x : elt \mid In\ x\ s \wedge For\_all\ (\mathsf{fun}\ y \Rightarrow \neg\ E.lt\ y\ x)\ s\} + \{Empty\ s\}.$

Definition max_elt :
$\forall\ s : t,$
$\{x : elt \mid In\ x\ s \wedge For\_all\ (\mathsf{fun}\ y \Rightarrow \neg\ E.lt\ x\ y)\ s\} + \{Empty\ s\}.$

Module $E := E.$

Definition $elt := elt.$
Definition $t := t.$

Definition $In := In.$
Definition $Equal\ s\ s' := \forall\ a : elt, In\ a\ s \leftrightarrow In\ a\ s'.$
Definition $Subset\ s\ s' := \forall\ a : elt, In\ a\ s \rightarrow In\ a\ s'.$
Definition $Empty\ s := \forall\ a : elt, \neg\ In\ a\ s.$
Definition $For\_all\ (P : elt \rightarrow \mathsf{Prop})\ (s : t) :=$
$\forall\ x : elt, In\ x\ s \rightarrow P\ x.$
Definition $Exists\ (P : elt \rightarrow \mathsf{Prop})\ (s : t) :=$
$\exists\ x : elt, In\ x\ s \wedge P\ x.$

Definition $eq\_In := In\_1.$

Definition $eq := Equal.$
Definition $lt := lt.$
Definition $eq\_refl := eq\_refl.$
Definition $eq\_sym := eq\_sym.$
Definition $eq\_trans := eq\_trans.$
Definition $lt\_trans := lt\_trans.$
Definition $lt\_not\_eq := lt\_not\_eq.$
Definition $compare := compare.$

End $DepOfNodep.$

217.3 From dependent signature *Sdep* to non-dependent signature *S*.

Module $NodepOfDep\ (M{:}\ Sdep) <: S$ with Module $E := M.E.$
Import $M.$

Module $ME := OrderedTypeFacts\ E.$

Definition $empty : t :=$ let $(s, \_) := empty$ in $s.$

Lemma $empty\_1 : Empty\ empty.$

Definition $is\_empty\ (s : t) : bool :=$
if $is\_empty\ s$ then $true$ else $false.$

Lemma $is\_empty\_1 : \forall\ s : t, Empty\ s \rightarrow is\_empty\ s = true.$

Lemma $is\_empty\_2 : \forall\ s : t, is\_empty\ s = true \rightarrow Empty\ s.$

```
Definition mem (x : elt) (s : t) : bool :=
  if mem x s then true else false.

Lemma mem_1 : ∀ (s : t) (x : elt), In x s → mem x s = true.

Lemma mem_2 : ∀ (s : t) (x : elt), mem x s = true → In x s.

Definition equal (s s' : t) : bool :=
  if equal s s' then true else false.

Lemma equal_1 : ∀ s s' : t, Equal s s' → equal s s' = true.

Lemma equal_2 : ∀ s s' : t, equal s s' = true → Equal s s'.

Definition subset (s s' : t) : bool :=
  if subset s s' then true else false.

Lemma subset_1 : ∀ s s' : t, Subset s s' → subset s s' = true.

Lemma subset_2 : ∀ s s' : t, subset s s' = true → Subset s s'.

Definition choose (s : t) : option elt :=
  match choose s with
  | inleft (exist x _) ⇒ Some x
  | inright _ ⇒ None
  end.

Lemma choose_1 : ∀ (s : t) (x : elt), choose s = Some x → In x s.

Lemma choose_2 : ∀ s : t, choose s = None → Empty s.

Definition elements (s : t) : list elt := let (l, _) := elements s in l.

Lemma elements_1 : ∀ (s : t) (x : elt), In x s → InA E.eq x (elements s).

Lemma elements_2 : ∀ (s : t) (x : elt), InA E.eq x (elements s) → In x s.

Lemma elements_3 : ∀ s : t, sort E.lt (elements s).

Definition min_elt (s : t) : option elt :=
  match min_elt s with
  | inleft (exist x _) ⇒ Some x
  | inright _ ⇒ None
  end.

Lemma min_elt_1 : ∀ (s : t) (x : elt), min_elt s = Some x → In x s.

Lemma min_elt_2 :
 ∀ (s : t) (x y : elt), min_elt s = Some x → In y s → ¬ E.lt y x.

Lemma min_elt_3 : ∀ s : t, min_elt s = None → Empty s.

Definition max_elt (s : t) : option elt :=
  match max_elt s with
  | inleft (exist x _) ⇒ Some x
  | inright _ ⇒ None
  end.
```

Lemma $max\_elt\_1 : \forall\ (s : t)\ (x : elt),\ max\_elt\ s = Some\ x \rightarrow In\ x\ s.$

Lemma $max\_elt\_2 :$
$\forall\ (s : t)\ (x\ y : elt),\ max\_elt\ s = Some\ x \rightarrow In\ y\ s \rightarrow \neg\ E.lt\ x\ y.$

Lemma $max\_elt\_3 : \forall\ s : t,\ max\_elt\ s = None \rightarrow Empty\ s.$

Definition $add\ (x : elt)\ (s : t) : t :=$ let $(s', \_) := add\ x\ s$ in $s'.$

Lemma $add\_1 : \forall\ (s : t)\ (x\ y : elt),\ E.eq\ x\ y \rightarrow In\ y\ (add\ x\ s).$

Lemma $add\_2 : \forall\ (s : t)\ (x\ y : elt),\ In\ y\ s \rightarrow In\ y\ (add\ x\ s).$

Lemma $add\_3 :$
$\forall\ (s : t)\ (x\ y : elt),\ \neg\ E.eq\ x\ y \rightarrow In\ y\ (add\ x\ s) \rightarrow In\ y\ s.$

Definition $remove\ (x : elt)\ (s : t) : t :=$ let $(s', \_) := remove\ x\ s$ in $s'.$

Lemma $remove\_1 : \forall\ (s : t)\ (x\ y : elt),\ E.eq\ x\ y \rightarrow \neg\ In\ y\ (remove\ x\ s).$

Lemma $remove\_2 :$
$\forall\ (s : t)\ (x\ y : elt),\ \neg\ E.eq\ x\ y \rightarrow In\ y\ s \rightarrow In\ y\ (remove\ x\ s).$

Lemma $remove\_3 : \forall\ (s : t)\ (x\ y : elt),\ In\ y\ (remove\ x\ s) \rightarrow In\ y\ s.$

Definition $singleton\ (x : elt) : t :=$ let $(s, \_) := singleton\ x$ in $s.$

Lemma $singleton\_1 : \forall\ x\ y : elt,\ In\ y\ (singleton\ x) \rightarrow E.eq\ x\ y.$

Lemma $singleton\_2 : \forall\ x\ y : elt,\ E.eq\ x\ y \rightarrow In\ y\ (singleton\ x).$

Definition $union\ (s\ s' : t) : t :=$ let $(s'', \_) := union\ s\ s'$ in $s''.$

Lemma $union\_1 :$
$\forall\ (s\ s' : t)\ (x : elt),\ In\ x\ (union\ s\ s') \rightarrow In\ x\ s \vee In\ x\ s'.$

Lemma $union\_2 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ s \rightarrow In\ x\ (union\ s\ s').$

Lemma $union\_3 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ s' \rightarrow In\ x\ (union\ s\ s').$

Definition $inter\ (s\ s' : t) : t :=$ let $(s'', \_) := inter\ s\ s'$ in $s''.$

Lemma $inter\_1 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ (inter\ s\ s') \rightarrow In\ x\ s.$

Lemma $inter\_2 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ (inter\ s\ s') \rightarrow In\ x\ s'.$

Lemma $inter\_3 :$
$\forall\ (s\ s' : t)\ (x : elt),\ In\ x\ s \rightarrow In\ x\ s' \rightarrow In\ x\ (inter\ s\ s').$

Definition $diff\ (s\ s' : t) : t :=$ let $(s'', \_) := diff\ s\ s'$ in $s''.$

Lemma $diff\_1 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ (diff\ s\ s') \rightarrow In\ x\ s.$

Lemma $diff\_2 : \forall\ (s\ s' : t)\ (x : elt),\ In\ x\ (diff\ s\ s') \rightarrow \neg\ In\ x\ s'.$

Lemma $diff\_3 :$
$\forall\ (s\ s' : t)\ (x : elt),\ In\ x\ s \rightarrow \neg\ In\ x\ s' \rightarrow In\ x\ (diff\ s\ s').$

Definition $cardinal\ (s : t) : nat :=$ let $(f, \_) := cardinal\ s$ in $f.$

Lemma $cardinal\_1 : \forall\ s,\ cardinal\ s = length\ (elements\ s).$

```
Definition fold (B : Set) (f : elt → B → B) (i : t)
  (s : B) : B := let (fold, _) := fold f i s in fold.

Lemma fold_1 :
 ∀ (s : t) (A : Set) (i : A) (f : elt → A → A),
 fold f s i = fold_left (fun a e ⇒ f e a) (elements s) i.

Definition f_dec :
  ∀ (f : elt → bool) (x : elt), {f x = true} + {f x ≠ true}.

Lemma compat_P_aux :
 ∀ f : elt → bool,
 compat_bool E.eq f → compat_P E.eq (fun x ⇒ f x = true).

Hint Resolve compat_P_aux.

Definition filter (f : elt → bool) (s : t) : t :=
  let (s', _) := filter (P:=fun x ⇒ f x = true) (f_dec f) s in s'.

Lemma filter_1 :
 ∀ (s : t) (x : elt) (f : elt → bool),
 compat_bool E.eq f → In x (filter f s) → In x s.

Lemma filter_2 :
 ∀ (s : t) (x : elt) (f : elt → bool),
 compat_bool E.eq f → In x (filter f s) → f x = true.

Lemma filter_3 :
 ∀ (s : t) (x : elt) (f : elt → bool),
 compat_bool E.eq f → In x s → f x = true → In x (filter f s).

Definition for_all (f : elt → bool) (s : t) : bool :=
  if for_all (P:=fun x ⇒ f x = true) (f_dec f) s
  then true
  else false.

Lemma for_all_1 :
 ∀ (s : t) (f : elt → bool),
 compat_bool E.eq f →
 For_all (fun x ⇒ f x = true) s → for_all f s = true.

Lemma for_all_2 :
 ∀ (s : t) (f : elt → bool),
 compat_bool E.eq f →
 for_all f s = true → For_all (fun x ⇒ f x = true) s.

Definition exists_ (f : elt → bool) (s : t) : bool :=
  if exists_ (P:=fun x ⇒ f x = true) (f_dec f) s
  then true
  else false.

Lemma exists_1 :
 ∀ (s : t) (f : elt → bool),
```

compat_bool E.eq f → *Exists* (fun *x* ⇒ *f x* = *true*) *s* → *exists_ f s* = *true*.

Lemma *exists_2* :
∀ (*s* : *t*) (*f* : *elt* → *bool*),
compat_bool E.eq f → *exists_ f s* = *true* → *Exists* (fun *x* ⇒ *f x* = *true*) *s*.

Definition *partition* (*f* : *elt* → *bool*) (*s* : *t*) :
t × *t* :=
let (*p*, _) := *partition* (*P*:=fun *x* ⇒ *f x* = *true*) (*f_dec f*) *s* in *p*.

Lemma *partition_1* :
∀ (*s* : *t*) (*f* : *elt* → *bool*),
compat_bool E.eq f → *Equal* (*fst* (*partition f s*)) (*filter f s*).

Lemma *partition_2* :
∀ (*s* : *t*) (*f* : *elt* → *bool*),
compat_bool E.eq f → *Equal* (*snd* (*partition f s*)) (*filter* (fun *x* ⇒ *negb* (*f x*)) *s*).

Module *E* := *E*.
Definition *elt* := *elt*.
Definition *t* := *t*.

Definition *In* := *In*.
Definition *Equal s s'* := ∀ *a* : *elt*, *In a s* ↔ *In a s'*.
Definition *Subset s s'* := ∀ *a* : *elt*, *In a s* → *In a s'*.
Definition *Add* (*x* : *elt*) (*s s'* : *t*) :=
∀ *y* : *elt*, *In y s'* ↔ *E.eq y x* ∨ *In y s*.
Definition *Empty s* := ∀ *a* : *elt*, ¬ *In a s*.
Definition *For_all* (*P* : *elt* → Prop) (*s* : *t*) :=
∀ *x* : *elt*, *In x s* → *P x*.
Definition *Exists* (*P* : *elt* → Prop) (*s* : *t*) :=
∃ *x* : *elt*, *In x s* ∧ *P x*.

Definition *In_1* := *eq_In*.

Definition *eq* := *Equal*.
Definition *lt* := *lt*.
Definition *eq_refl* := *eq_refl*.
Definition *eq_sym* := *eq_sym*.
Definition *eq_trans* := *eq_trans*.
Definition *lt_trans* := *lt_trans*.
Definition *lt_not_eq* := *lt_not_eq*.
Definition *compare* := *compare*.

End *NodepOfDep*.

Chapter 218

Module Coq.FSets.FSetEqProperties

218.1 Finite sets library

This module proves many properties of finite sets that are consequences of the axiomatization in *FsetInterface* Contrary to the functor in *FsetProperties* it uses sets operations instead of predicates over sets, i.e. *mem x s=true* instead of *In x s*, *equal s s'=true* instead of *Equal s s'*, etc.

Require Import *FSetProperties*.
Require Import *Zerob*.
Require Import *Sumbool*.
Require Import *Omega*.

Module *EqProperties* (*M*:*S*).
Import *M*.
Import *Logic*.
Import *Peano*.

Module *ME* := *OrderedTypeFacts E*.
Module *MP* := *Properties M*.
Import *MP*.
Import *MP.FM*.

Definition *Add* := *MP.Add*.

Section *BasicProperties*.

Some old specifications written with boolean equalities.

Variable *s s' s"*: *t*.
Variable *x y z* : *elt*.

Lemma *mem_eq*:
E.eq x y → *mem x s=mem y s*.

Lemma *equal_mem_1*:
(∀ *a*, *mem a s=mem a s'*) → *equal s s'=true*.

Lemma *equal_mem_2*:

equal s s'=true → ∀ a, mem a s=mem a s'.

Lemma *subset_mem_1*:
(∀ a, mem a s=true→mem a s'=true) → subset s s'=true.

Lemma *subset_mem_2*:
subset s s'=true → ∀ a, mem a s=true → mem a s'=true.

Lemma *empty_mem*: *mem x empty=false.*

Lemma *is_empty_equal_empty*: *is_empty s = equal s empty.*

Lemma *choose_mem_1*: *choose s=Some x → mem x s=true.*

Lemma *choose_mem_2*: *choose s=None → is_empty s=true.*

Lemma *add_mem_1*: *mem x (add x s)=true.*

Lemma *add_mem_2*: *¬E.eq x y → mem y (add x s)=mem y s.*

Lemma *remove_mem_1*: *mem x (remove x s)=false.*

Lemma *remove_mem_2*: *¬E.eq x y → mem y (remove x s)=mem y s.*

Lemma *singleton_equal_add*:
equal (singleton x) (add x empty)=true.

Lemma *union_mem*:
mem x (union s s')=mem x s || mem x s'.

Lemma *inter_mem*:
mem x (inter s s')=mem x s && mem x s'.

Lemma *diff_mem*:
mem x (diff s s')=mem x s && negb (mem x s').

properties of *mem*

Lemma *mem_3* : *¬In x s → mem x s=false.*

Lemma *mem_4* : *mem x s=false → ¬In x s.*

Properties of *equal*

Lemma *equal_refl*: *equal s s=true.*

Lemma *equal_sym*: *equal s s'=equal s' s.*

Lemma *equal_trans*:
equal s s'=true → equal s' s"=true → equal s s"=true.

Lemma *equal_equal*:
equal s s'=true → equal s s"=equal s' s".

Lemma *equal_cardinal*:
equal s s'=true → cardinal s=cardinal s'.

Lemma *subset_refl*: *subset s s=true.*

Lemma *subset_antisym*:

subset s s'=true → subset s' s=true → equal s s'=true.

Lemma *subset_trans*:
subset s s'=true → subset s' s"=true → subset s s"=true.

Lemma *subset_equal*:
equal s s'=true → subset s s'=true.

Properties of *choose*

Lemma *choose_mem_3*:
is_empty s=false → {x:elt|choose s=Some x ∧ mem x s=true}.

Lemma *choose_mem_4*: *choose empty=None.*

Properties of *add*

Lemma *add_mem_3*:
mem y s=true → mem y (add x s)=true.

Lemma *add_equal*:
mem x s=true → equal (add x s) s=true.

Properties of *remove*

Lemma *remove_mem_3*:
mem y (remove x s)=true → mem y s=true.

Lemma *remove_equal*:
mem x s=false → equal (remove x s) s=true.

Lemma *add_remove*:
mem x s=true → equal (add x (remove x s)) s=true.

Lemma *remove_add*:
mem x s=false → equal (remove x (add x s)) s=true.

Properties of *is_empty*

Lemma *is_empty_cardinal*: *is_empty s = zerob (cardinal s).*

Properties of *singleton*

Lemma *singleton_mem_1*: *mem x (singleton x)=true.*

Lemma *singleton_mem_2*: *¬E.eq x y → mem y (singleton x)=false.*

Lemma *singleton_mem_3*: *mem y (singleton x)=true → E.eq x y.*

Properties of *union*

Lemma *union_sym*:
equal (union s s') (union s' s)=true.

Lemma *union_subset_equal*:
subset s s'=true → equal (union s s') s'=true.

Lemma *union_equal_1*:
equal s s'=true→ equal (union s s") (union s' s")=true.

Lemma *union_equal_2*:
equal s' s"=true→ equal (union s s') (union s s")=true.

Lemma *union_assoc*:
equal (union (union s s') s") (union s (union s' s"))=true.

Lemma *add_union_singleton*:
equal (add x s) (union (singleton x) s)=true.

Lemma *union_add*:
equal (union (add x s) s') (add x (union s s'))=true.

Lemma *union_subset_1*: *subset s (union s s')=true.*

Lemma *union_subset_2*: *subset s' (union s s')=true.*

Lemma *union_subset_3*:
subset s s"=true → subset s' s"=true →
subset (union s s') s"=true.

Properties of *inter*

Lemma *inter_sym*: *equal (inter s s') (inter s' s)=true.*

Lemma *inter_subset_equal*:
subset s s'=true → equal (inter s s') s=true.

Lemma *inter_equal_1*:
equal s s'=true → equal (inter s s") (inter s' s")=true.

Lemma *inter_equal_2*:
equal s' s"=true → equal (inter s s') (inter s s")=true.

Lemma *inter_assoc*:
equal (inter (inter s s') s") (inter s (inter s' s"))=true.

Lemma *union_inter_1*:
equal (inter (union s s') s") (union (inter s s") (inter s' s"))=true.

Lemma *union_inter_2*:
equal (union (inter s s') s") (inter (union s s") (union s' s"))=true.

Lemma *inter_add_1*: *mem x s'=true →*
equal (inter (add x s) s') (add x (inter s s'))=true.

Lemma *inter_add_2*: *mem x s'=false →*
equal (inter (add x s) s') (inter s s')=true.

Lemma *inter_subset_1*: *subset (inter s s') s=true.*

Lemma *inter_subset_2*: *subset (inter s s') s'=true.*

Lemma *inter_subset_3*:
subset s" s=true → subset s" s'=true →
subset s" (inter s s')=true.

Properties of *diff*

Lemma *diff_subset*: *subset (diff s s') s=true.*

Lemma *diff_subset_equal*:
subset s s'=true → equal (diff s s') empty=true.

Lemma *remove_inter_singleton*:
equal (remove x s) (diff s (singleton x))=true.

Lemma *diff_inter_empty*:
equal (inter (diff s s') (inter s s')) empty=true.

Lemma *diff_inter_all*:
equal (union (diff s s') (inter s s')) s=true.

End *BasicProperties.*

Hint Immediate *empty_mem is_empty_equal_empty add_mem_1*
remove_mem_1 singleton_equal_add union_mem inter_mem
diff_mem equal_sym add_remove remove_add : set.
Hint *Resolve equal_mem_1 subset_mem_1 choose_mem_1*
choose_mem_2 add_mem_2 remove_mem_2 equal_refl equal_equal
subset_refl subset_equal subset_antisym
add_mem_3 add_equal remove_mem_3 remove_equal : set.

General recursion principes based on *cardinal*

Lemma *cardinal_set_rec*: ∀ (*P*:*t*→Type),
(∀ *s s'*, *equal s s'=true → P s → P s'*) →
(∀ *s x*, *mem x s=false → P s → P (add x s)*) →
P empty → ∀ *n s*, *cardinal s=n → P s.*

Lemma *set_rec*: ∀ (*P*:*t*→Type),
(∀ *s s'*, *equal s s'=true → P s → P s'*) →
(∀ *s x*, *mem x s=false → P s → P (add x s)*) →
P empty → ∀ *s*, *P s.*

Properties of *fold*

Lemma *exclusive_set* : ∀ *s s' x*,
¬*In x s*\/˜*In x s'* ↔ *mem x s* && *mem x s'=false.*

Section *Fold.*
Variables (*A*:Set)(*eqA*:*A*→*A*→Prop)(*st*:*Setoid_Theory _ eqA*).
Variables (*f*:*elt*→*A*→*A*)(*Comp*:*compat_op E.eq eqA f*)(*Ass*:*transpose eqA f*).
Variables (*i*:*A*).
Variables (*s s'*:*t*)(*x*:*elt*).

Lemma *fold_empty*: *eqA (fold f empty i) i.*

Lemma *fold_equal*:
equal s s'=true → eqA (fold f s i) (fold f s' i).

Lemma *fold_add*:
mem x s=false → eqA (fold f (add x s) i) (f x (fold f s i)).

Lemma *add_fold*:
mem x s=true → eqA (fold f (add x s) i) (fold f s i).

Lemma *remove_fold_1*:
mem x s=true → eqA (f x (fold f (remove x s) i)) (fold f s i).

Lemma *remove_fold_2*:
mem x s=false → eqA (fold f (remove x s) i) (fold f s i).

Lemma *fold_union*:
(∀ x, mem x s && mem x s'=false) →
eqA (fold f (union s s') i) (fold f s (fold f s' i)).

End *Fold.*

Properties of *cardinal*

Lemma *add_cardinal_1*:
∀ s x, mem x s=true → cardinal (add x s)=cardinal s.

Lemma *add_cardinal_2*:
∀ s x, mem x s=false → cardinal (add x s)=S (cardinal s).

Lemma *remove_cardinal_1*:
∀ s x, mem x s=true → S (cardinal (remove x s))=cardinal s.

Lemma *remove_cardinal_2*:
∀ s x, mem x s=false → cardinal (remove x s)=cardinal s.

Lemma *union_cardinal*:
∀ s s', (∀ x, mem x s && mem x s'=false) →
cardinal (union s s')=cardinal s+cardinal s'.

Lemma *subset_cardinal*:
∀ s s', subset s s'=true → cardinal s≤cardinal s'.

Section *Bool.*

Properties of *filter*

Variable *f*:*elt→bool.*
Variable *Comp*: *compat_bool E.eq f.*

Let *Comp'* : *compat_bool E.eq* (fun *x* ⇒*negb* (*f x*)).

Lemma *filter_mem*: *∀ s x, mem x (filter f s)=mem x s && f x.*

Lemma *for_all_filter*:
∀ s, for_all f s=is_empty (filter (fun *x* ⇒ *negb* (*f x*)) *s).*

Lemma *exists_filter* :
∀ s, exists_ f s=negb (is_empty (filter f s)).

Lemma *partition_filter_1*:
∀ s, equal (fst (partition f s)) (filter f s)=true.

Lemma *partition_filter_2*:

$\forall\ s,\ equal\ (snd\ (partition\ f\ s))\ (filter\ (\mathsf{fun}\ x \Rightarrow negb\ (f\ x))\ s){=}true.$

Lemma *add_filter_1* : $\forall\ s\ s'\ x,$
$f\ x{=}true \rightarrow (Add\ x\ s\ s') \rightarrow (Add\ x\ (filter\ f\ s)\ (filter\ f\ s')).$

Lemma *add_filter_2* : $\forall\ s\ s'\ x,$
$f\ x{=}false \rightarrow (Add\ x\ s\ s') \rightarrow filter\ f\ s\ [=]\ filter\ f\ s'.$

Lemma *union_filter*: $\forall\ f\ g, (compat\_bool\ E.eq\ f) \rightarrow (compat\_bool\ E.eq\ g) \rightarrow$
$\forall\ s,\ union\ (filter\ f\ s)\ (filter\ g\ s)\ [=]\ filter\ (\mathsf{fun}\ x{\Rightarrow}orb\ (f\ x)\ (g\ x))\ s.$

Lemma *filter_union*: $\forall\ s\ s', filter\ f\ (union\ s\ s')\ [=]\ union\ (filter\ f\ s)\ (filter\ f\ s').$

Properties of *for_all*

Lemma *for_all_mem_1*: $\forall\ s,$
$(\forall\ x, (mem\ x\ s){=}true\text{->}(f\ x){=}true) \rightarrow (for\_all\ f\ s){=}true.$

Lemma *for_all_mem_2*: $\forall\ s,$
$(for\_all\ f\ s){=}true \rightarrow \forall\ x,(mem\ x\ s){=}true \rightarrow (f\ x){=}true.$

Lemma *for_all_mem_3*:
$\forall\ s\ x,(mem\ x\ s){=}true \rightarrow (f\ x){=}false \rightarrow (for\_all\ f\ s){=}false.$

Lemma *for_all_mem_4*:
$\forall\ s, for\_all\ f\ s{=}false \rightarrow \{x{:}elt \mid mem\ x\ s{=}true \wedge f\ x{=}false\}.$

Properties of $\exists$

Lemma *for_all_exists*:
$\forall\ s,\ exists\_\ f\ s\ =\ negb\ (for\_all\ (\mathsf{fun}\ x\ {\Rightarrow}negb\ (f\ x))\ s).$

End *Bool*.
Section *Bool'*.

Variable $f{:}elt{\rightarrow}bool.$
Variable *Comp*: $compat\_bool\ E.eq\ f.$

Let *Comp'* : $compat\_bool\ E.eq\ (\mathsf{fun}\ x\ {\Rightarrow}negb\ (f\ x)).$

Lemma *exists_mem_1*:
$\forall\ s, (\forall\ x, mem\ x\ s{=}true{\rightarrow}f\ x{=}false) \rightarrow exists\_\ f\ s{=}false.$

Lemma *exists_mem_2*:
$\forall\ s, exists\_\ f\ s{=}false \rightarrow \forall\ x, mem\ x\ s{=}true \rightarrow f\ x{=}false.$

Lemma *exists_mem_3*:
$\forall\ s\ x, mem\ x\ s{=}true \rightarrow f\ x{=}true \rightarrow exists\_\ f\ s{=}true.$

Lemma *exists_mem_4*:
$\forall\ s, exists\_\ f\ s{=}true \rightarrow \{x{:}elt \mid (mem\ x\ s){=}true \wedge (f\ x){=}true\}.$

End *Bool'*.

Section *Sum*.

Adding a valuation function on all elements of a set.

Definition $sum\ (f{:}elt \to nat)(s{:}t) := fold\ (\mathsf{fun}\ x \Rightarrow plus\ (f\ x))\ s\ 0.$

Lemma sum_plus :
$\forall\ f\ g,\ compat\_nat\ E.eq\ f \to compat\_nat\ E.eq\ g \to$
$\forall\ s,\ sum\ (\mathsf{fun}\ x \Rightarrow f\ x + g\ x)\ s = sum\ f\ s + sum\ g\ s.$

Lemma sum_filter : $\forall\ f,\ (compat\_bool\ E.eq\ f) \to$
$\forall\ s,\ (sum\ (\mathsf{fun}\ x \Rightarrow \mathsf{if}\ f\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ s) = (cardinal\ (filter\ f\ s)).$

Lemma $fold\_compat$:
$\forall\ (A{:}\mathsf{Set})(eqA{:}A \to A \to \mathsf{Prop})(st{:}(Setoid\_Theory\ \_\ eqA))$
$(f\ g{:}elt \to A \to A),$
$(compat\_op\ E.eq\ eqA\ f) \to (transpose\ eqA\ f) \to$
$(compat\_op\ E.eq\ eqA\ g) \to (transpose\ eqA\ g) \to$
$\forall\ (i{:}A)(s{:}t),(\forall\ x{:}elt,\ (In\ x\ s) \to \forall\ y,\ (eqA\ (f\ x\ y)\ (g\ x\ y))) \to$
$(eqA\ (fold\ f\ s\ i)\ (fold\ g\ s\ i)).$

Lemma sum_compat :
$\forall\ f\ g,\ compat\_nat\ E.eq\ f \to compat\_nat\ E.eq\ g \to$
$\forall\ s,\ (\forall\ x,\ In\ x\ s \to f\ x = g\ x) \to sum\ f\ s = sum\ g\ s.$

End $Sum.$

End $EqProperties.$

Chapter 219

Module Coq.FSets.FSetFacts

219.1 Finite sets library

This functor derives additional facts from *FSetInterface.S*. These facts are mainly the specifications of *FSetInterface.S* written using different styles: equivalence and boolean equalities. Moreover, we prove that *E.Eq* and *Equal* are setoid equalities.

Require Export *FSetInterface*.

Module *Facts* (*M*: *S*).
Module *ME* := *OrderedTypeFacts M.E*.
Import *ME*.
Import *M*.
Import *Logic*.
Import *Peano*.

219.2 Specifications written using equivalences

Section *IffSpec*.
Variable *s s' s"* : *t*.
Variable *x y z* : *elt*.

Lemma *In_eq_iff* : *E.eq x y* → (*In x s* ↔ *In y s*).

Lemma *mem_iff* : *In x s* ↔ *mem x s* = *true*.

Lemma *not_mem_iff* : ¬*In x s* ↔ *mem x s* = *false*.

Lemma *equal_iff* : *s*[=]*s'* ↔ *equal s s'* = *true*.

Lemma *subset_iff* : *s*[<=]*s'* ↔ *subset s s'* = *true*.

Lemma *empty_iff* : *In x empty* ↔ *False*.

Lemma *is_empty_iff* : *Empty s* ↔ *is_empty s* = *true*.

Lemma *singleton_iff* : *In y* (*singleton x*) ↔ *E.eq x y*.

Lemma *add_iff* : *In y (add x s)* ↔ *E.eq x y* ∨ *In y s*.

Lemma *add_neq_iff* : ¬ *E.eq x y* → (*In y (add x s)* ↔ *In y s*).

Lemma *remove_iff* : *In y (remove x s)* ↔ *In y s* ∧ ¬*E.eq x y*.

Lemma *remove_neq_iff* : ¬ *E.eq x y* → (*In y (remove x s)* ↔ *In y s*).

Lemma *union_iff* : *In x (union s s')* ↔ *In x s* ∨ *In x s'*.

Lemma *inter_iff* : *In x (inter s s')* ↔ *In x s* ∧ *In x s'*.

Lemma *diff_iff* : *In x (diff s s')* ↔ *In x s* ∧ ¬ *In x s'*.

Variable *f* : *elt*→*bool*.

Lemma *filter_iff* : *compat_bool E.eq f* → (*In x (filter f s)* ↔ *In x s* ∧ *f x* = *true*).

Lemma *for_all_iff* : *compat_bool E.eq f* →
(*For_all* (fun *x* ⇒ *f x* = *true*) *s* ↔ *for_all f s* = *true*).

Lemma *exists_iff* : *compat_bool E.eq f* →
(*Exists* (fun *x* ⇒ *f x* = *true*) *s* ↔ *exists_ f s* = *true*).

Lemma *elements_iff* : *In x s* ↔ *InA E.eq x (elements s)*.

End *IffSpec*.

Useful tactic for simplifying expressions like *In y (add x (union s s'))*

Ltac *set_iff* :=
repeat (progress (
rewrite add_iff || *rewrite remove_iff* || *rewrite singleton_iff*
|| *rewrite union_iff* || *rewrite inter_iff* || *rewrite diff_iff*
|| *rewrite empty_iff*)).

219.3 Specifications written using boolean predicates

Section *BoolSpec*.
Variable *s s' s"* : *t*.
Variable *x y z* : *elt*.

Lemma *mem_b* : *E.eq x y* → *mem x s* = *mem y s*.

Lemma *empty_b* : *mem y empty* = *false*.

Lemma *add_b* : *mem y (add x s)* = *eqb x y* || *mem y s*.

Lemma *add_neq_b* : ¬ *E.eq x y* → *mem y (add x s)* = *mem y s*.

Lemma *remove_b* : *mem y (remove x s)* = *mem y s* && *negb (eqb x y)*.

Lemma *remove_neq_b* : ¬ *E.eq x y* → *mem y (remove x s)* = *mem y s*.

Lemma *singleton_b* : *mem y (singleton x)* = *eqb x y*.

Lemma *union_b* : *mem x (union s s')* = *mem x s* || *mem x s'*.

Lemma *inter_b* : *mem x (inter s s') = mem x s && mem x s'.*

Lemma *diff_b* : *mem x (diff s s') = mem x s && negb (mem x s').*

Lemma *elements_b* : *mem x s = existsb (eqb x) (elements s).*

Variable *f* : *elt→bool.*

Lemma *filter_b* : *compat_bool E.eq f → mem x (filter f s) = mem x s && f x.*

Lemma *for_all_b* : *compat_bool E.eq f →*
for_all f s = forallb f (elements s).

Lemma *exists_b* : *compat_bool E.eq f →*
exists_ f s = existsb f (elements s).

End *BoolSpec.*

219.4 *E.eq* and *Equal* are setoid equalities

Definition *E_ST* : *Setoid_Theory elt E.eq.*

Add Setoid elt E.eq E_ST as *EltSetoid.*

Definition *Equal_ST* : *Setoid_Theory t Equal.*

Add Setoid t Equal Equal_ST as *EqualSetoid.*

Add Morphism In with *signature E.eq ==> Equal ==> iff* as *In_m.*

Add Morphism is_empty : *is_empty_m.*

Add Morphism Empty with *signature Equal ==> iff* as *Empty_m.*

Add Morphism mem : *mem_m.*

Add Morphism singleton : *singleton_m.*

Add Morphism add : *add_m.*

Add Morphism remove : *remove_m.*

Add Morphism union : *union_m.*

Add Morphism inter : *inter_m.*

Add Morphism diff : *diff_m.*

Add Morphism Subset with *signature Equal ==> Equal ==> iff* as *Subset_m.*

Add Morphism subset : *subset_m.*

Add Morphism equal : *equal_m.*

Lemma *filter_equal* : *∀ f, compat_bool E.eq f →*
∀ s s', s[=]s' → filter f s [=] filter f s'.

End *Facts.*

Chapter 220

Module Coq.FSets.FSetInterface

220.1 Finite set library

Set interfaces

Compatibility of a boolean function with respect to an equality.
Definition *compat_bool* (*A*:Set)(*eqA*: *A*→*A*→Prop)(*f*: *A*→ *bool*) :=
  ∀ *x y* : *A*, *eqA x y* → *f x* = *f y*.

Compatibility of a predicate with respect to an equality.
Definition *compat_P* (*A*:Set)(*eqA*: *A*→*A*→Prop)(*P* : *A* → Prop) :=
  ∀ *x y* : *A*, *eqA x y* → *P x* → *P y*.

Hint *Unfold compat_bool compat_P*.

220.2 Non-dependent signature

Signature *S* presents sets as purely informative programs together with axioms

Module Type *S*.

  Declare Module *E* : *OrderedType*.
  Definition *elt* := *E.t*.

  Parameter *t* : Set.
the abstract type of sets

Logical predicates
  Parameter *In* : *elt* → *t* → Prop.
  Definition *Equal s s'* := ∀ *a* : *elt*, *In a s* ↔ *In a s'*.
  Definition *Subset s s'* := ∀ *a* : *elt*, *In a s* → *In a s'*.
  Definition *Empty s* := ∀ *a* : *elt*, ¬ *In a s*.
  Definition *For_all* (*P* : *elt* → Prop) *s* := ∀ *x*, *In x s* → *P x*.
  Definition *Exists* (*P* : *elt* → Prop) *s* := ∃ *x*, *In x s* ∧ *P x*.

Notation "s [=] t" := ($Equal$ s t) ($at\ level$ 70, $no\ associativity$).
Notation "s [<=] t" := ($Subset$ s t) ($at\ level$ 70, $no\ associativity$).

Parameter $empty$: t.
The empty set.

Parameter is_empty : $t \to bool$.
Test whether a set is empty or not.

Parameter mem : $elt \to t \to bool$.
$mem\ x\ s$ tests whether x belongs to the set s.

Parameter add : $elt \to t \to t$.
$add\ x\ s$ returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged.

Parameter $singleton$: $elt \to t$.
$singleton\ x$ returns the one-element set containing only x.

Parameter $remove$: $elt \to t \to t$.
$remove\ x\ s$ returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged.

Parameter $union$: $t \to t \to t$.
Set union.

Parameter $inter$: $t \to t \to t$.
Set intersection.

Parameter $diff$: $t \to t \to t$.
Set difference.

Definition eq : $t \to t \to$ Prop := $Equal$.
Parameter lt : $t \to t \to$ Prop.
Parameter $compare$: $\forall\ s\ s'$: t, $Compare\ lt\ eq\ s\ s'$.
Total ordering between sets. Can be used as the ordering function for doing sets of sets.

Parameter $equal$: $t \to t \to bool$.
$equal\ s1\ s2$ tests whether the sets $s1$ and $s2$ are equal, that is, contain equal elements.

Parameter $subset$: $t \to t \to bool$.
$subset\ s1\ s2$ tests whether the set $s1$ is a subset of the set $s2$.

Coq comment: $iter$ is useless in a purely functional world
iter: (elt -> unit) -> set -> unit. i
$iter\ f\ s$ applies f in turn to all elements of s. The order in which the elements of s are presented to f is unspecified.

Parameter $fold$: $\forall\ A$: Set, $(elt \to A \to A) \to t \to A \to A$.
$fold\ f\ s\ a$ computes $(f\ xN\ \ldots\ (f\ x2\ (f\ x1\ a))\ldots)$, where $x1\ \ldots\ xN$ are the elements of s, in increasing order.

Parameter for_all : $(elt \to bool) \to t \to bool$.
$for\_all\ p\ s$ checks if all elements of the set satisfy the predicate p.

Parameter *exists_* : $(elt \to bool) \to t \to bool$.
$\exists\ p\ s$ checks if at least one element of the set satisfies the predicate p.

Parameter *filter* : $(elt \to bool) \to t \to t$.
filter $p\ s$ returns the set of all elements in s that satisfy predicate p.

Parameter *partition* : $(elt \to bool) \to t \to t \times t$.
partition $p\ s$ returns a pair of sets $(s1, s2)$, where $s1$ is the set of all the elements of s that satisfy the predicate p, and $s2$ is the set of all the elements of s that do not satisfy p.

Parameter *cardinal* : $t \to nat$.
Return the number of elements of a set.
Coq comment: nat instead of int ...

Parameter *elements* : $t \to list\ elt$.
Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering *Ord.compare*, where *Ord* is the argument given to {!Set.Make}.

Parameter *min_elt* : $t \to option\ elt$.
Return the smallest element of the given set (with respect to the *Ord.compare* ordering), or raise *Not_found* if the set is empty.
Coq comment: *Not_found* is represented by the option type

Parameter *max_elt* : $t \to option\ elt$.
Same as {!Set.S.min_elt}, but returns the largest element of the given set.
Coq comment: *Not_found* is represented by the option type

Parameter *choose* : $t \to option\ elt$.
Return one element of the given set, or raise *Not_found* if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.
Coq comment: *Not_found* is represented by the option type

Section *Spec*.

Variable $s\ s'\ s''$: t.
Variable $x\ y$: elt.

Specification of *In*
Parameter *In_1* : $E.eq\ x\ y \to In\ x\ s \to In\ y\ s$.

Specification of *eq*
Parameter *eq_refl* : $eq\ s\ s$.
Parameter *eq_sym* : $eq\ s\ s' \to eq\ s'\ s$.
Parameter *eq_trans* : $eq\ s\ s' \to eq\ s'\ s'' \to eq\ s\ s''$.

Specification of *lt*
Parameter *lt_trans* : $lt\ s\ s' \to lt\ s'\ s'' \to lt\ s\ s''$.
Parameter *lt_not_eq* : $lt\ s\ s' \to \neg\ eq\ s\ s'$.

Specification of *mem*
Parameter *mem_1* : $In\ x\ s \to mem\ x\ s = true$.
Parameter *mem_2* : $mem\ x\ s = true \to In\ x\ s$.

Specification of *equal*

Parameter $equal\_1 : s[=]s' \to equal\ s\ s' = true$.
Parameter $equal\_2 : equal\ s\ s' = true \to s[=]s'$.

Specification of *subset*
Parameter $subset\_1 : s[<=]s' \to subset\ s\ s' = true$.
Parameter $subset\_2 : subset\ s\ s' = true \to s[<=]s'$.

Specification of *empty*
Parameter $empty\_1 : Empty\ empty$.

Specification of *is_empty*
Parameter $is\_empty\_1 : Empty\ s \to is\_empty\ s = true$.
Parameter $is\_empty\_2 : is\_empty\ s = true \to Empty\ s$.

Specification of *add*
Parameter $add\_1 : E.eq\ x\ y \to In\ y\ (add\ x\ s)$.
Parameter $add\_2 : In\ y\ s \to In\ y\ (add\ x\ s)$.
Parameter $add\_3 : \neg\ E.eq\ x\ y \to In\ y\ (add\ x\ s) \to In\ y\ s$.

Specification of *remove*
Parameter $remove\_1 : E.eq\ x\ y \to \neg\ In\ y\ (remove\ x\ s)$.
Parameter $remove\_2 : \neg\ E.eq\ x\ y \to In\ y\ s \to In\ y\ (remove\ x\ s)$.
Parameter $remove\_3 : In\ y\ (remove\ x\ s) \to In\ y\ s$.

Specification of *singleton*
Parameter $singleton\_1 : In\ y\ (singleton\ x) \to E.eq\ x\ y$.
Parameter $singleton\_2 : E.eq\ x\ y \to In\ y\ (singleton\ x)$.

Specification of *union*
Parameter $union\_1 : In\ x\ (union\ s\ s') \to In\ x\ s \vee In\ x\ s'$.
Parameter $union\_2 : In\ x\ s \to In\ x\ (union\ s\ s')$.
Parameter $union\_3 : In\ x\ s' \to In\ x\ (union\ s\ s')$.

Specification of *inter*
Parameter $inter\_1 : In\ x\ (inter\ s\ s') \to In\ x\ s$.
Parameter $inter\_2 : In\ x\ (inter\ s\ s') \to In\ x\ s'$.
Parameter $inter\_3 : In\ x\ s \to In\ x\ s' \to In\ x\ (inter\ s\ s')$.

Specification of *diff*
Parameter $diff\_1 : In\ x\ (diff\ s\ s') \to In\ x\ s$.
Parameter $diff\_2 : In\ x\ (diff\ s\ s') \to \neg\ In\ x\ s'$.
Parameter $diff\_3 : In\ x\ s \to \neg\ In\ x\ s' \to In\ x\ (diff\ s\ s')$.

Specification of *fold*
Parameter $fold\_1 : \forall\ (A : \mathsf{Set})\ (i : A)\ (f : elt \to A \to A),$
$fold\ f\ s\ i = fold\_left\ (\mathsf{fun}\ a\ e \Rightarrow f\ e\ a)\ (elements\ s)\ i$.

Specification of *cardinal*
Parameter $cardinal\_1 : cardinal\ s = length\ (elements\ s)$.

Section *Filter*.

Variable $f : elt \to bool$.

Specification of *filter*

Parameter *filter_1* : *compat_bool E.eq f* → *In x* (*filter f s*) → *In x s*.
Parameter *filter_2* : *compat_bool E.eq f* → *In x* (*filter f s*) → *f x* = *true*.
Parameter *filter_3* :
compat_bool E.eq f → *In x s* → *f x* = *true* → *In x* (*filter f s*).

Specification of *for_all*

Parameter *for_all_1* :
compat_bool E.eq f →
For_all (fun *x* ⇒ *f x* = *true*) *s* → *for_all f s* = *true*.
Parameter *for_all_2* :
compat_bool E.eq f →
for_all f s = *true* → *For_all* (fun *x* ⇒ *f x* = *true*) *s*.

Specification of ∃

Parameter *exists_1* :
compat_bool E.eq f →
Exists (fun *x* ⇒ *f x* = *true*) *s* → *exists_ f s* = *true*.
Parameter *exists_2* :
compat_bool E.eq f →
exists_ f s = *true* → *Exists* (fun *x* ⇒ *f x* = *true*) *s*.

Specification of *partition*

Parameter *partition_1* : *compat_bool E.eq f* →
fst (*partition f s*) [=] *filter f s*.
Parameter *partition_2* : *compat_bool E.eq f* →
snd (*partition f s*) [=] *filter* (fun *x* ⇒ *negb* (*f x*)) *s*.

End *Filter*.

Specification of *elements*

Parameter *elements_1* : *In x s* → *InA E.eq x* (*elements s*).
Parameter *elements_2* : *InA E.eq x* (*elements s*) → *In x s*.
Parameter *elements_3* : *sort E.lt* (*elements s*).

Specification of *min_elt*

Parameter *min_elt_1* : *min_elt s* = *Some x* → *In x s*.
Parameter *min_elt_2* : *min_elt s* = *Some x* → *In y s* → ¬ *E.lt y x*.
Parameter *min_elt_3* : *min_elt s* = *None* → *Empty s*.

Specification of *max_elt*

Parameter *max_elt_1* : *max_elt s* = *Some x* → *In x s*.
Parameter *max_elt_2* : *max_elt s* = *Some x* → *In y s* → ¬ *E.lt x y*.
Parameter *max_elt_3* : *max_elt s* = *None* → *Empty s*.

Specification of *choose*

Parameter *choose_1* : *choose s* = *Some x* → *In x s*.
Parameter *choose_2* : *choose s* = *None* → *Empty s*.
End *Spec*.

End *S*.

220.3 Dependent signature

Signature *Sdep* presents sets using dependent types

Module Type *Sdep*.

Declare Module E : *OrderedType*.
Definition $elt := E.t$.

Parameter t : Set.

Parameter In : $elt \to t \to$ Prop.
Definition $Equal\ s\ s' := \forall\ a : elt, In\ a\ s \leftrightarrow In\ a\ s'$.
Definition $Subset\ s\ s' := \forall\ a : elt, In\ a\ s \to In\ a\ s'$.
Definition $Add\ x\ s\ s' := \forall\ y, In\ y\ s' \leftrightarrow E.eq\ x\ y \vee In\ y\ s$.
Definition $Empty\ s := \forall\ a : elt, \neg\ In\ a\ s$.
Definition $For\_all\ (P : elt \to \text{Prop})\ s := \forall\ x, In\ x\ s \to P\ x$.
Definition $Exists\ (P : elt \to \text{Prop})\ s := \exists\ x, In\ x\ s \wedge P\ x$.

Notation "s [=] t" := ($Equal\ s\ t$) (*at level* 70, *no associativity*).

Definition $eq : t \to t \to \text{Prop} := Equal$.
Parameter $lt : t \to t \to$ Prop.
Parameter $compare : \forall\ s\ s' : t, Compare\ lt\ eq\ s\ s'$.

Parameter $eq\_refl : \forall\ s : t, eq\ s\ s$.
Parameter $eq\_sym : \forall\ s\ s' : t, eq\ s\ s' \to eq\ s'\ s$.
Parameter $eq\_trans : \forall\ s\ s'\ s'' : t, eq\ s\ s' \to eq\ s'\ s'' \to eq\ s\ s''$.
Parameter $lt\_trans : \forall\ s\ s'\ s'' : t, lt\ s\ s' \to lt\ s'\ s'' \to lt\ s\ s''$.
Parameter $lt\_not\_eq : \forall\ s\ s' : t, lt\ s\ s' \to \neg\ eq\ s\ s'$.

Parameter $eq\_In : \forall\ (s : t)\ (x\ y : elt), E.eq\ x\ y \to In\ x\ s \to In\ y\ s$.

Parameter $empty : \{s : t \mid Empty\ s\}$.

Parameter $is\_empty : \forall\ s : t, \{Empty\ s\} + \{\tilde{}\ Empty\ s\}$.

Parameter $mem : \forall\ (x : elt)\ (s : t), \{In\ x\ s\} + \{\tilde{}\ In\ x\ s\}$.

Parameter $add : \forall\ (x : elt)\ (s : t), \{s' : t \mid Add\ x\ s\ s'\}$.

Parameter
$singleton : \forall\ x : elt, \{s : t \mid \forall\ y : elt, In\ y\ s \leftrightarrow E.eq\ x\ y\}$.

Parameter
$remove$:
$\forall\ (x : elt)\ (s : t)$,
$\{s' : t \mid \forall\ y : elt, In\ y\ s' \leftrightarrow \neg\ E.eq\ x\ y \wedge In\ y\ s\}$.

Parameter
$union$:
$\forall\ s\ s' : t$,
$\{s'' : t \mid \forall\ x : elt, In\ x\ s'' \leftrightarrow In\ x\ s \vee In\ x\ s'\}$.

Parameter

$inter$:
$\forall\ s\ s' : t,$
$\{s'' : t \mid \forall\ x : elt, In\ x\ s'' \leftrightarrow In\ x\ s \wedge In\ x\ s'\}.$

Parameter
$diff$:
$\forall\ s\ s' : t,$
$\{s'' : t \mid \forall\ x : elt, In\ x\ s'' \leftrightarrow In\ x\ s \wedge \neg\ In\ x\ s'\}.$

Parameter $equal : \forall\ s\ s' : t, \{s[=]s'\} + \{\tilde{}\ s[=]s'\}.$

Parameter $subset : \forall\ s\ s' : t, \{Subset\ s\ s'\} + \{\tilde{}\ Subset\ s\ s'\}.$

Parameter
$filter$:
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})$
$(s : t),$
$\{s' : t \mid compat\_P\ E.eq\ P \to \forall\ x : elt, In\ x\ s' \leftrightarrow In\ x\ s \wedge P\ x\}.$

Parameter
for_all :
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})$
$(s : t),$
$\{compat\_P\ E.eq\ P \to For\_all\ P\ s\} + \{compat\_P\ E.eq\ P \to \neg\ For\_all\ P\ s\}.$

Parameter
$exists\_$:
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})$
$(s : t),$
$\{compat\_P\ E.eq\ P \to Exists\ P\ s\} + \{compat\_P\ E.eq\ P \to \neg\ Exists\ P\ s\}.$

Parameter
$partition$:
$\forall\ (P : elt \to \mathsf{Prop})\ (Pdec : \forall\ x : elt, \{P\ x\} + \{\tilde{}\ P\ x\})$
$(s : t),$
$\{partition : t \times t \mid$
$\mathsf{let}\ (s1, s2) := partition\ \mathsf{in}$
$compat\_P\ E.eq\ P \to$
$For\_all\ P\ s1 \wedge$
$For\_all\ (\mathsf{fun}\ x \Rightarrow \neg\ P\ x)\ s2 \wedge$
$(\forall\ x : elt, In\ x\ s \leftrightarrow In\ x\ s1 \vee In\ x\ s2)\}.$

Parameter
$elements$:
$\forall\ s : t,$
$\{l : list\ elt \mid$
$sort\ E.lt\ l \wedge (\forall\ x : elt, In\ x\ s \leftrightarrow InA\ E.eq\ x\ l)\}.$

Parameter
$fold$:
$\forall\ (A : \mathsf{Set})\ (f : elt \to A \to A)\ (s : t)\ (i : A),$

$\{r : A \mid \mathsf{let}\ (l,\_) := elements\ s\ \mathsf{in}$
$r = fold\_left\ (\mathsf{fun}\ a\ e \Rightarrow f\ e\ a)\ l\ i\}.$

Parameter
$cardinal :$
$\forall\ s : t,$
$\{r : nat \mid \mathsf{let}\ (l,\_) := elements\ s\ \mathsf{in}\ r = length\ l\ \}.$

Parameter
$min\_elt :$
$\forall\ s : t,$
$\{x : elt \mid In\ x\ s \wedge For\_all\ (\mathsf{fun}\ y \Rightarrow \neg\ E.lt\ y\ x)\ s\} + \{Empty\ s\}.$

Parameter
$max\_elt :$
$\forall\ s : t,$
$\{x : elt \mid In\ x\ s \wedge For\_all\ (\mathsf{fun}\ y \Rightarrow \neg\ E.lt\ x\ y)\ s\} + \{Empty\ s\}.$

Parameter $choose : \forall\ s : t, \{x : elt \mid In\ x\ s\} + \{Empty\ s\}.$

End *Sdep*.

Chapter 221

Module Coq.FSets.FSetList

221.1 Finite sets library

This file proposes an implementation of the non-dependant interface *FSetInterface.S* using strictly ordered list.

Require Export *FSetInterface*.

221.2 Functions over lists

First, we provide sets as lists which are not necessarily sorted. The specs are proved under the additional condition of being sorted. And the functions returning sets are proved to preserve this invariant.

Module *Raw* (*X*: *OrderedType*).

Module *E* := *X*.
Module *MX* := *OrderedTypeFacts X*.
Import *MX*.

Definition *elt* := *X.t*.
Definition *t* := *list elt*.

Definition *empty* : *t* := *nil*.

Definition *is_empty* (*l* : *t*) : *bool* := if *l* then *true* else *false*.

221.2.1 The set operations.

Fixpoint *mem* (*x* : *elt*) (*s* : *t*) {*struct s*} : *bool* :=
match *s* with
| *nil* ⇒ *false*
| *y* :: *l* ⇒

```
      match X.compare x y with
      | LT _ ⇒ false
      | EQ _ ⇒ true
      | GT _ ⇒ mem x l
      end
  end.

Fixpoint add (x : elt) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ x :: nil
  | y :: l ⇒
      match X.compare x y with
      | LT _ ⇒ x :: s
      | EQ _ ⇒ s
      | GT _ ⇒ y :: add x l
      end
  end.

Definition singleton (x : elt) : t := x :: nil.

Fixpoint remove (x : elt) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ nil
  | y :: l ⇒
      match X.compare x y with
      | LT _ ⇒ s
      | EQ _ ⇒ l
      | GT _ ⇒ y :: remove x l
      end
  end.

Fixpoint union (s : t) : t → t :=
  match s with
  | nil ⇒ fun s' ⇒ s'
  | x :: l ⇒
      (fix union_aux (s' : t) : t :=
         match s' with
         | nil ⇒ s
         | x' :: l' ⇒
             match X.compare x x' with
             | LT _ ⇒ x :: union l s'
             | EQ _ ⇒ x :: union l l'
             | GT _ ⇒ x' :: union_aux l'
             end
         end)
  end.

Fixpoint inter (s : t) : t → t :=
  match s with
```

```
  | nil ⇒ fun _ ⇒ nil
  | x :: l ⇒
      (fix inter_aux (s' : t) : t :=
         match s' with
         | nil ⇒ nil
         | x' :: l' ⇒
             match X.compare x x' with
             | LT _ ⇒ inter l s'
             | EQ _ ⇒ x :: inter l l'
             | GT _ ⇒ inter_aux l'
             end
         end)
  end.

Fixpoint diff (s : t) : t → t :=
  match s with
  | nil ⇒ fun _ ⇒ nil
  | x :: l ⇒
      (fix diff_aux (s' : t) : t :=
         match s' with
         | nil ⇒ s
         | x' :: l' ⇒
             match X.compare x x' with
             | LT _ ⇒ x :: diff l s'
             | EQ _ ⇒ diff l l'
             | GT _ ⇒ diff_aux l'
             end
         end)
  end.

Fixpoint equal (s : t) : t → bool :=
  fun s' : t ⇒
  match s, s' with
  | nil, nil ⇒ true
  | x :: l, x' :: l' ⇒
      match X.compare x x' with
      | EQ _ ⇒ equal l l'
      | _ ⇒ false
      end
  | _, _ ⇒ false
  end.

Fixpoint subset (s s' : t) {struct s'} : bool :=
  match s, s' with
  | nil, _ ⇒ true
  | x :: l, x' :: l' ⇒
      match X.compare x x' with
```

```
      | LT _ ⇒ false
      | EQ _ ⇒ subset l l'
      | GT _ ⇒ subset s l'
      end
  | _, _ ⇒ false
  end.

Fixpoint fold (B : Set) (f : elt → B → B) (s : t) {struct s} :
 B → B := fun i ⇒ match s with
                  | nil ⇒ i
                  | x :: l ⇒ fold f l (f x i)
                  end.

Fixpoint filter (f : elt → bool) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ nil
  | x :: l ⇒ if f x then x :: filter f l else filter f l
  end.

Fixpoint for_all (f : elt → bool) (s : t) {struct s} : bool :=
  match s with
  | nil ⇒ true
  | x :: l ⇒ if f x then for_all f l else false
  end.

Fixpoint exists_ (f : elt → bool) (s : t) {struct s} : bool :=
  match s with
  | nil ⇒ false
  | x :: l ⇒ if f x then true else exists_ f l
  end.

Fixpoint partition (f : elt → bool) (s : t) {struct s} :
 t × t :=
  match s with
  | nil ⇒ (nil, nil)
  | x :: l ⇒
      let (s1, s2) := partition f l in
      if f x then (x :: s1, s2) else (s1, x :: s2)
  end.

Definition cardinal (s : t) : nat := length s.

Definition elements (x : t) : list elt := x.

Definition min_elt (s : t) : option elt :=
  match s with
  | nil ⇒ None
  | x :: _ ⇒ Some x
  end.

Fixpoint max_elt (s : t) : option elt :=
```

```
  match s with
  | nil ⇒ None
  | x :: nil ⇒ Some x
  | _ :: l ⇒ max_elt l
  end.

Definition choose := min_elt.
```

221.2.2 Proofs of set operation specifications.

```
Section ForNotations.

Notation Sort := (sort X.lt).
Notation Inf := (lelistA X.lt).
Notation In := (InA X.eq).

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) (s : t) := ∃ x, In x s ∧ P x.

Lemma mem_1 :
 ∀ (s : t) (Hs : Sort s) (x : elt), In x s → mem x s = true.

Lemma mem_2 : ∀ (s : t) (x : elt), mem x s = true → In x s.

Lemma add_Inf :
 ∀ (s : t) (x a : elt), Inf a s → X.lt a x → Inf a (add x s).
Hint Resolve add_Inf.

Lemma add_sort : ∀ (s : t) (Hs : Sort s) (x : elt), Sort (add x s).

Lemma add_1 :
 ∀ (s : t) (Hs : Sort s) (x y : elt), X.eq x y → In y (add x s).

Lemma add_2 :
 ∀ (s : t) (Hs : Sort s) (x y : elt), In y s → In y (add x s).

Lemma add_3 :
 ∀ (s : t) (Hs : Sort s) (x y : elt),
 ¬ X.eq x y → In y (add x s) → In y s.

Lemma remove_Inf :
 ∀ (s : t) (Hs : Sort s) (x a : elt), Inf a s → Inf a (remove x s).
Hint Resolve remove_Inf.

Lemma remove_sort :
 ∀ (s : t) (Hs : Sort s) (x : elt), Sort (remove x s).

Lemma remove_1 :
 ∀ (s : t) (Hs : Sort s) (x y : elt), X.eq x y → ¬ In y (remove x s).
```

```
Lemma remove_2 :
 ∀ (s : t) (Hs : Sort s) (x y : elt),
 ¬ X.eq x y → In y s → In y (remove x s).

Lemma remove_3 :
 ∀ (s : t) (Hs : Sort s) (x y : elt), In y (remove x s) → In y s.

Lemma singleton_sort : ∀ x : elt, Sort (singleton x).

Lemma singleton_1 : ∀ x y : elt, In y (singleton x) → X.eq x y.

Lemma singleton_2 : ∀ x y : elt, X.eq x y → In y (singleton x).

Ltac DoubleInd :=
  simple induction s;
   [ simpl; auto; try solve [ intros; inversion H ]
   | intros x l Hrec; simple induction s';
       [ simpl; auto; try solve [ intros; inversion H ]
       | intros x' l' Hrec' Hs Hs'; inversion Hs; inversion Hs'; subst;
           simpl ] ].

Lemma union_Inf :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (a : elt),
 Inf a s → Inf a s' → Inf a (union s s').
Hint Resolve union_Inf.

Lemma union_sort :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s'), Sort (union s s').

Lemma union_1 :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (x : elt),
 In x (union s s') → In x s ∨ In x s'.

Lemma union_2 :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (x : elt),
 In x s → In x (union s s').

Lemma union_3 :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (x : elt),
 In x s' → In x (union s s').

Lemma inter_Inf :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (a : elt),
 Inf a s → Inf a s' → Inf a (inter s s').
Hint Resolve inter_Inf.

Lemma inter_sort :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s'), Sort (inter s s').

Lemma inter_1 :
 ∀ (s s' : t) (Hs : Sort s) (Hs' : Sort s') (x : elt),
 In x (inter s s') → In x s.

Lemma inter_2 :
```

$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (x : elt),$
$In\ x\ (inter\ s\ s') \to In\ x\ s'.$

Lemma *inter_3* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (x : elt),$
$In\ x\ s \to In\ x\ s' \to In\ x\ (inter\ s\ s').$

Lemma *diff_Inf* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (a : elt),$
$Inf\ a\ s \to Inf\ a\ s' \to Inf\ a\ (diff\ s\ s').$
Hint *Resolve diff_Inf.*

Lemma *diff_sort* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s'), Sort\ (diff\ s\ s').$

Lemma *diff_1* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (x : elt),$
$In\ x\ (diff\ s\ s') \to In\ x\ s.$

Lemma *diff_2* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (x : elt),$
$In\ x\ (diff\ s\ s') \to \neg\ In\ x\ s'.$

Lemma *diff_3* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s')\ (x : elt),$
$In\ x\ s \to \neg\ In\ x\ s' \to In\ x\ (diff\ s\ s').$

Lemma *equal_1* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s'),$
$Equal\ s\ s' \to equal\ s\ s' = true.$

Lemma *equal_2* : $\forall\ s\ s' : t, equal\ s\ s' = true \to Equal\ s\ s'.$

Lemma *subset_1* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s'),$
$Subset\ s\ s' \to subset\ s\ s' = true.$

Lemma *subset_2* : $\forall\ s\ s' : t, subset\ s\ s' = true \to Subset\ s\ s'.$

Lemma *empty_sort* : $Sort\ empty.$

Lemma *empty_1* : $Empty\ empty.$

Lemma *is_empty_1* : $\forall\ s : t, Empty\ s \to is\_empty\ s = true.$

Lemma *is_empty_2* : $\forall\ s : t, is\_empty\ s = true \to Empty\ s.$

Lemma *elements_1* : $\forall\ (s : t)\ (x : elt), In\ x\ s \to In\ x\ (elements\ s).$

Lemma *elements_2* : $\forall\ (s : t)\ (x : elt), In\ x\ (elements\ s) \to In\ x\ s.$

Lemma *elements_3* : ∀ (*s* : *t*) (*Hs* : *Sort s*), *Sort* (*elements s*).

Lemma *min_elt_1* : ∀ (*s* : *t*) (*x* : *elt*), *min_elt s* = *Some x* → *In x s*.

Lemma *min_elt_2* :
∀ (*s* : *t*) (*Hs* : *Sort s*) (*x y* : *elt*),
min_elt s = *Some x* → *In y s* → ¬ *X.lt y x*.

Lemma *min_elt_3* : ∀ *s* : *t*, *min_elt s* = *None* → *Empty s*.

Lemma *max_elt_1* : ∀ (*s* : *t*) (*x* : *elt*), *max_elt s* = *Some x* → *In x s*.

Lemma *max_elt_2* :
∀ (*s* : *t*) (*Hs* : *Sort s*) (*x y* : *elt*),
max_elt s = *Some x* → *In y s* → ¬ *X.lt x y*.

Lemma *max_elt_3* : ∀ *s* : *t*, *max_elt s* = *None* → *Empty s*.

Definition *choose_1* :
∀ (*s* : *t*) (*x* : *elt*), *choose s* = *Some x* → *In x s* := *min_elt_1*.

Definition *choose_2* : ∀ *s* : *t*, *choose s* = *None* → *Empty s* := *min_elt_3*.

Lemma *fold_1* :
∀ (*s* : *t*) (*Hs* : *Sort s*) (*A* : Set) (*i* : *A*) (*f* : *elt* → *A* → *A*),
fold f s i = *fold_left* (fun *a e* ⇒ *f e a*) (*elements s*) *i*.

Lemma *cardinal_1* :
∀ (*s* : *t*) (*Hs* : *Sort s*),
cardinal s = *length* (*elements s*).

Lemma *filter_Inf* :
∀ (*s* : *t*) (*Hs* : *Sort s*) (*x* : *elt*) (*f* : *elt* → *bool*),
Inf x s → *Inf x* (*filter f s*).

Lemma *filter_sort* :
∀ (*s* : *t*) (*Hs* : *Sort s*) (*f* : *elt* → *bool*), *Sort* (*filter f s*).

Lemma *filter_1* :
∀ (*s* : *t*) (*x* : *elt*) (*f* : *elt* → *bool*),
compat_bool X.eq f → *In x* (*filter f s*) → *In x s*.

Lemma *filter_2* :
∀ (*s* : *t*) (*x* : *elt*) (*f* : *elt* → *bool*),
compat_bool X.eq f → *In x* (*filter f s*) → *f x* = *true*.

Lemma *filter_3* :
∀ (*s* : *t*) (*x* : *elt*) (*f* : *elt* → *bool*),
compat_bool X.eq f → *In x s* → *f x* = *true* → *In x* (*filter f s*).

Lemma *for_all_1* :
∀ (*s* : *t*) (*f* : *elt* → *bool*),
compat_bool X.eq f →
For_all (fun *x* ⇒ *f x* = *true*) *s* → *for_all f s* = *true*.

Lemma *for_all_2* :

$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to$
$for\_all\ f\ s = true \to For\_all\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s.$

Lemma *exists_1* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to Exists\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s \to exists\_\ f\ s = true.$

Lemma *exists_2* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to exists\_\ f\ s = true \to Exists\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s.$

Lemma *partition_Inf_1* :
$\forall\ (s : t)\ (Hs : Sort\ s)\ (f : elt \to bool)\ (x : elt),$
$Inf\ x\ s \to Inf\ x\ (fst\ (partition\ f\ s)).$

Lemma *partition_Inf_2* :
$\forall\ (s : t)\ (Hs : Sort\ s)\ (f : elt \to bool)\ (x : elt),$
$Inf\ x\ s \to Inf\ x\ (snd\ (partition\ f\ s)).$

Lemma *partition_sort_1* :
$\forall\ (s : t)\ (Hs : Sort\ s)\ (f : elt \to bool),\ Sort\ (fst\ (partition\ f\ s)).$

Lemma *partition_sort_2* :
$\forall\ (s : t)\ (Hs : Sort\ s)\ (f : elt \to bool),\ Sort\ (snd\ (partition\ f\ s)).$

Lemma *partition_1* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to Equal\ (fst\ (partition\ f\ s))\ (filter\ f\ s).$

Lemma *partition_2* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to$
$Equal\ (snd\ (partition\ f\ s))\ (filter\ (\mathsf{fun}\ x \Rightarrow negb\ (f\ x))\ s).$

Definition $eq : t \to t \to \mathsf{Prop} := Equal.$

Lemma $eq\_refl : \forall\ s : t,\ eq\ s\ s.$

Lemma $eq\_sym : \forall\ s\ s' : t,\ eq\ s\ s' \to eq\ s'\ s.$

Lemma $eq\_trans : \forall\ s\ s'\ s'' : t,\ eq\ s\ s' \to eq\ s'\ s'' \to eq\ s\ s''.$

Inductive $lt : t \to t \to \mathsf{Prop} :=$
| $lt\_nil : \forall\ (x : elt)\ (s : t),\ lt\ nil\ (x :: s)$
| $lt\_cons\_lt :$
$\forall\ (x\ y : elt)\ (s\ s' : t),\ X.lt\ x\ y \to lt\ (x :: s)\ (y :: s')$
| $lt\_cons\_eq :$
$\forall\ (x\ y : elt)\ (s\ s' : t),$
$X.eq\ x\ y \to lt\ s\ s' \to lt\ (x :: s)\ (y :: s').$

Hint *Constructors lt.*

Lemma $lt\_trans : \forall\ s\ s'\ s'' : t,\ lt\ s\ s' \to lt\ s'\ s'' \to lt\ s\ s''.$

Lemma *lt_not_eq* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s'),\ lt\ s\ s' \rightarrow \neg\ eq\ s\ s'.$

Definition *compare* :
$\forall\ (s\ s' : t)\ (Hs : Sort\ s)\ (Hs' : Sort\ s'),\ Compare\ lt\ eq\ s\ s'.$

End *ForNotations*.
Hint *Constructors lt*.

End *Raw*.

221.3 Encapsulation

Now, in order to really provide a functor implementing S, we need to encapsulate everything into a type of strictly ordered lists.

Module *Make* $(X:\ OrderedType) <:\ S$ with Module $E := X$.

Module *Raw* := *Raw X*.
Module $E := X$.

Record *slist* : Set := $\{this :> Raw.t;\ sorted : sort\ E.lt\ this\}$.
Definition $t := slist$.
Definition $elt := E.t$.

Definition $In\ (x : elt)\ (s : t) :$ Prop := $InA\ E.eq\ x\ s.(this)$.
Definition $Equal\ (s\ s':t) :$ Prop := $\forall\ a : elt,\ In\ a\ s \leftrightarrow In\ a\ s'$.
Definition $Subset\ (s\ s':t) :$ Prop := $\forall\ a : elt,\ In\ a\ s \rightarrow In\ a\ s'$.
Definition $Empty\ (s:t) :$ Prop := $\forall\ a : elt,\ \neg\ In\ a\ s$.
Definition $For\_all\ (P : elt \rightarrow$ Prop$)(s:t) :$ Prop := $\forall\ x,\ In\ x\ s \rightarrow P\ x$.
Definition $Exists\ (P : elt \rightarrow$ Prop$)(s:t) :$ Prop := $\exists\ x,\ In\ x\ s \wedge P\ x$.

Definition $mem\ (x : elt)\ (s : t) : bool := Raw.mem\ x\ s$.
Definition $add\ (x : elt)(s : t) : t := Build\_slist\ (Raw.add\_sort\ (sorted\ s)\ x)$.
Definition $remove\ (x : elt)(s : t) : t := Build\_slist\ (Raw.remove\_sort\ (sorted\ s)\ x)$.
Definition $singleton\ (x : elt) : t := Build\_slist\ (Raw.singleton\_sort\ x)$.
Definition $union\ (s\ s' : t) : t :=$
$Build\_slist\ (Raw.union\_sort\ (sorted\ s)\ (sorted\ s'))$.
Definition $inter\ (s\ s' : t) : t :=$
$Build\_slist\ (Raw.inter\_sort\ (sorted\ s)\ (sorted\ s'))$.
Definition $diff\ (s\ s' : t) : t :=$
$Build\_slist\ (Raw.diff\_sort\ (sorted\ s)\ (sorted\ s'))$.
Definition $equal\ (s\ s' : t) : bool := Raw.equal\ s\ s'$.
Definition $subset\ (s\ s' : t) : bool := Raw.subset\ s\ s'$.
Definition $empty : t := Build\_slist\ Raw.empty\_sort$.
Definition $is\_empty\ (s : t) : bool := Raw.is\_empty\ s$.
Definition $elements\ (s : t) : list\ elt := Raw.elements\ s$.
Definition $min\_elt\ (s : t) : option\ elt := Raw.min\_elt\ s$.
Definition $max\_elt\ (s : t) : option\ elt := Raw.max\_elt\ s$.

```
Definition choose (s : t) : option elt := Raw.choose s.
Definition fold (B : Set) (f : elt → B → B) (s : t) : B → B := Raw.fold (B:=B) f s.
Definition cardinal (s : t) : nat := Raw.cardinal s.
Definition filter (f : elt → bool) (s : t) : t :=
  Build_slist (Raw.filter_sort (sorted s) f).
Definition for_all (f : elt → bool) (s : t) : bool := Raw.for_all f s.
Definition exists_ (f : elt → bool) (s : t) : bool := Raw.exists_ f s.
Definition partition (f : elt → bool) (s : t) : t × t :=
  let p := Raw.partition f s in
  (Build_slist (this:=fst p) (Raw.partition_sort_1 (sorted s) f),
  Build_slist (this:=snd p) (Raw.partition_sort_2 (sorted s) f)).
Definition eq (s s' : t) : Prop := Raw.eq s s'.
Definition lt (s s' : t) : Prop := Raw.lt s s'.

Section Spec.
 Variable s s' s'': t.
 Variable x y : elt.

 Lemma In_1 : E.eq x y → In x s → In y s.

 Lemma mem_1 : In x s → mem x s = true.
 Lemma mem_2 : mem x s = true → In x s.

 Lemma equal_1 : Equal s s' → equal s s' = true.
 Lemma equal_2 : equal s s' = true → Equal s s'.

 Lemma subset_1 : Subset s s' → subset s s' = true.
 Lemma subset_2 : subset s s' = true → Subset s s'.

 Lemma empty_1 : Empty empty.

 Lemma is_empty_1 : Empty s → is_empty s = true.
 Lemma is_empty_2 : is_empty s = true → Empty s.

 Lemma add_1 : E.eq x y → In y (add x s).
 Lemma add_2 : In y s → In y (add x s).
 Lemma add_3 : ¬ E.eq x y → In y (add x s) → In y s.

 Lemma remove_1 : E.eq x y → ¬ In y (remove x s).
 Lemma remove_2 : ¬ E.eq x y → In y s → In y (remove x s).
 Lemma remove_3 : In y (remove x s) → In y s.

 Lemma singleton_1 : In y (singleton x) → E.eq x y.
 Lemma singleton_2 : E.eq x y → In y (singleton x).

 Lemma union_1 : In x (union s s') → In x s ∨ In x s'.
 Lemma union_2 : In x s → In x (union s s').
 Lemma union_3 : In x s' → In x (union s s').

 Lemma inter_1 : In x (inter s s') → In x s.
 Lemma inter_2 : In x (inter s s') → In x s'.
 Lemma inter_3 : In x s → In x s' → In x (inter s s').
```

Lemma *diff_1* : *In x (diff s s') → In x s.*
Lemma *diff_2* : *In x (diff s s') → ¬ In x s'.*
Lemma *diff_3* : *In x s → ¬ In x s' → In x (diff s s').*

Lemma *fold_1* : ∀ (*A* : Set) (*i* : *A*) (*f* : *elt → A → A*),
fold f s i = fold_left (fun *a e* ⇒ *f e a*) (*elements s*) *i.*

Lemma *cardinal_1* : *cardinal s = length* (*elements s*).

Section *Filter.*

Variable *f* : *elt → bool.*

Lemma *filter_1* : *compat_bool E.eq f → In x* (*filter f s*) *→ In x s.*
Lemma *filter_2* : *compat_bool E.eq f → In x* (*filter f s*) *→ f x = true.*
Lemma *filter_3* :
compat_bool E.eq f → In x s → f x = true → In x (*filter f s*).

Lemma *for_all_1* :
compat_bool E.eq f →
For_all (fun *x* ⇒ *f x = true*) *s → for_all f s = true.*
Lemma *for_all_2* :
compat_bool E.eq f →
for_all f s = true → For_all (fun *x* ⇒ *f x = true*) *s.*

Lemma *exists_1* :
compat_bool E.eq f →
Exists (fun *x* ⇒ *f x = true*) *s → exists_ f s = true.*
Lemma *exists_2* :
compat_bool E.eq f →
exists_ f s = true → Exists (fun *x* ⇒ *f x = true*) *s.*

Lemma *partition_1* :
compat_bool E.eq f → Equal (*fst* (*partition f s*)) (*filter f s*).
Lemma *partition_2* :
compat_bool E.eq f →
Equal (*snd* (*partition f s*)) (*filter* (fun *x* ⇒ *negb* (*f x*)) *s*).

End *Filter.*

Lemma *elements_1* : *In x s → InA E.eq x* (*elements s*).
Lemma *elements_2* : *InA E.eq x* (*elements s*) *→ In x s.*
Lemma *elements_3* : *sort E.lt* (*elements s*).

Lemma *min_elt_1* : *min_elt s = Some x → In x s.*
Lemma *min_elt_2* : *min_elt s = Some x → In y s → ¬ E.lt y x.*
Lemma *min_elt_3* : *min_elt s = None → Empty s.*

Lemma *max_elt_1* : *max_elt s = Some x → In x s.*
Lemma *max_elt_2* : *max_elt s = Some x → In y s → ¬ E.lt x y.*
Lemma *max_elt_3* : *max_elt s = None → Empty s.*

Lemma *choose_1* : *choose s = Some x → In x s.*

Lemma *choose_2* : *choose s = None → Empty s.*

Lemma *eq_refl* : *eq s s.*
Lemma *eq_sym* : *eq s s' → eq s' s.*
Lemma *eq_trans* : *eq s s' → eq s' s" → eq s s".*

Lemma *lt_trans* : *lt s s' → lt s' s" → lt s s".*
Lemma *lt_not_eq* : *lt s s' → ¬ eq s s'.*

Definition *compare* : *Compare lt eq s s'.*

End *Spec.*

End *Make.*

Chapter 222

Module Coq.FSets.FSetProperties

222.1 Finite sets library

This functor derives additional properties from *FSetInterface.S*. Contrary to the functor in *FSetEqProperties* it uses predicates over sets instead of sets operations, i.e. *In x s* instead of *mem x s=true*, *Equal s s'* instead of *equal s s'=true*, etc.

Require Export *FSetInterface*.
Require Import *FSetFacts*.

Hint *Unfold transpose compat_op compat_nat*.
Hint *Extern* 1 (*Setoid_Theory* _ _) ⇒ *constructor*; *congruence*.

Module *Properties* (*M*: *S*).
  Module *ME*:=*OrderedTypeFacts*(*M.E*).
  Import *ME*.
  Import *M.E*.
  Import *M*.
  Import *Logic*.
  Import *Peano*.

Results about lists without duplicates

  Module *FM* := *Facts M*.
  Import *FM*.

  Definition *Add* (*x* : *elt*) (*s s'* : *t*) :=
    ∀ *y* : *elt*, *In y s'* ↔ *E.eq x y* ∨ *In y s*.

  Lemma *In_dec* : ∀ *x s*, {*In x s*} + {~ *In x s*}.

  Section *BasicProperties*.

properties of *Equal*

  Lemma *equal_refl* : ∀ *s*, *s*[=]*s*.

  Lemma *equal_sym* : ∀ *s s'*, *s*[=]*s'* → *s'*[=]*s*.

Lemma *equal_trans* : ∀ *s1 s2 s3*, *s1*[=]*s2* → *s2*[=]*s3* → *s1*[=]*s3*.

Variable *s s' s" s1 s2 s3* : *t*.
Variable *x x'* : *elt*.

properties of *Subset*

Lemma *subset_refl* : *s*[<=]*s*.

Lemma *subset_antisym* : *s*[<=]*s'* → *s'*[<=]*s* → *s*[=]*s'*.

Lemma *subset_trans* : *s1*[<=]*s2* → *s2*[<=]*s3* → *s1*[<=]*s3*.

Lemma *subset_equal* : *s*[=]*s'* → *s*[<=]*s'*.

Lemma *subset_empty* : *empty*[<=]*s*.

Lemma *subset_remove_3* : *s1*[<=]*s2* → *remove x s1* [<=] *s2*.

Lemma *subset_diff* : *s1*[<=]*s3* → *diff s1 s2* [<=] *s3*.

Lemma *subset_add_3* : *In x s2* → *s1*[<=]*s2* → *add x s1* [<=] *s2*.

Lemma *subset_add_2* : *s1*[<=]*s2* → *s1*[<=] *add x s2*.

Lemma *in_subset* : *In x s1* → *s1*[<=]*s2* → *In x s2*.

Lemma *double_inclusion* : *s1*[=]*s2* ↔ *s1*[<=]*s2* ∧ *s2*[<=]*s1*.

properties of *empty*

Lemma *empty_is_empty_1* : *Empty s* → *s*[=]*empty*.

Lemma *empty_is_empty_2* : *s*[=]*empty* → *Empty s*.

properties of *add*

Lemma *add_equal* : *In x s* → *add x s* [=] *s*.

Lemma *add_add* : *add x* (*add x' s*) [=] *add x'* (*add x s*).

properties of *remove*

Lemma *remove_equal* : ¬ *In x s* → *remove x s* [=] *s*.

Lemma *Equal_remove* : *s*[=]*s'* → *remove x s* [=] *remove x s'*.

properties of *add* and *remove*

Lemma *add_remove* : *In x s* → *add x* (*remove x s*) [=] *s*.

Lemma *remove_add* : ¬*In x s* → *remove x* (*add x s*) [=] *s*.

properties of *singleton*

Lemma *singleton_equal_add* : *singleton x* [=] *add x empty*.

properties of *union*

Lemma *union_sym* : *union s s'* [=] *union s' s*.

Lemma *union_subset_equal* : *s*[<=]*s'* → *union s s'* [=] *s'*.

Lemma *union_equal_1* : *s*[=]*s'* → *union s s"* [=] *union s' s"*.

Lemma *union_equal_2* : *s'*[=]*s"* → *union s s'* [=] *union s s"*.

Lemma *union_assoc* : *union* (*union s s'*) *s"* [=] *union s* (*union s' s"*).

Lemma *add_union_singleton* : *add x s* [=] *union* (*singleton x*) *s*.

Lemma *union_add* : *union* (*add x s*) *s'* [=] *add x* (*union s s'*).

Lemma *union_subset_1* : *s* [<=] *union s s'*.

Lemma *union_subset_2* : *s'* [<=] *union s s'*.

Lemma *union_subset_3* : *s*[<=]*s"* → *s'*[<=]*s"* → *union s s'* [<=] *s"*.

Lemma *union_subset_4* : *s*[<=]*s'* → *union s s"* [<=] *union s' s"*.

Lemma *union_subset_5* : *s*[<=]*s'* → *union s" s* [<=] *union s" s'*.

Lemma *empty_union_1* : *Empty s* → *union s s'* [=] *s'*.

Lemma *empty_union_2* : *Empty s* → *union s' s* [=] *s'*.

Lemma *not_in_union* : ¬*In x s* → ¬*In x s'* → ¬*In x* (*union s s'*).

properties of *inter*

Lemma *inter_sym* : *inter s s'* [=] *inter s' s*.

Lemma *inter_subset_equal* : *s*[<=]*s'* → *inter s s'* [=] *s*.

Lemma *inter_equal_1* : *s*[=]*s'* → *inter s s"* [=] *inter s' s"*.

Lemma *inter_equal_2* : *s'*[=]*s"* → *inter s s'* [=] *inter s s"*.

Lemma *inter_assoc* : *inter* (*inter s s'*) *s"* [=] *inter s* (*inter s' s"*).

Lemma *union_inter_1* : *inter* (*union s s'*) *s"* [=] *union* (*inter s s"*) (*inter s' s"*).

Lemma *union_inter_2* : *union* (*inter s s'*) *s"* [=] *inter* (*union s s"*) (*union s' s"*).

Lemma *inter_add_1* : *In x s'* → *inter* (*add x s*) *s'* [=] *add x* (*inter s s'*).

Lemma *inter_add_2* : ¬ *In x s'* → *inter* (*add x s*) *s'* [=] *inter s s'*.

Lemma *empty_inter_1* : *Empty s* → *Empty* (*inter s s'*).

Lemma *empty_inter_2* : *Empty s'* → *Empty* (*inter s s'*).

Lemma *inter_subset_1* : *inter s s'* [<=] *s*.

Lemma *inter_subset_2* : *inter s s'* [<=] *s'*.

Lemma *inter_subset_3* :
s"[<=]*s* → *s"*[<=]*s'* → *s"*[<=] *inter s s'*.

properties of *diff*

Lemma *empty_diff_1* : *Empty s* → *Empty* (*diff s s'*).

Lemma *empty_diff_2* : *Empty s* → *diff s' s* [=] *s'*.

Lemma *diff_subset* : *diff s s'* [<=] *s*.

Lemma *diff_subset_equal* : *s*[<=]*s'* → *diff s s'* [=] *empty*.

Lemma *remove_diff_singleton* :
remove x s [=] *diff s* (*singleton x*).

Lemma *diff_inter_empty* : *inter* (*diff s s'*) (*inter s s'*) [=] *empty*.

Lemma *diff_inter_all* : *union* (*diff s s'*) (*inter s s'*) [=] *s*.

properties of *Add*

Lemma *Add_add* : *Add x s* (*add x s*).

Lemma *Add_remove* : *In x s* → *Add x* (*remove x s*) *s*.

Lemma *union_Add* : *Add x s s'* → *Add x* (*union s s"*) (*union s' s"*).

Lemma *inter_Add* :
In x s" → *Add x s s'* → *Add x* (*inter s s"*) (*inter s' s"*).

Lemma *union_Equal* :
In x s" → *Add x s s'* → *union s s"* [=] *union s' s"*.

Lemma *inter_Add_2* :
¬*In x s"* → *Add x s s'* → *inter s s"* [=] *inter s' s"*.

End *BasicProperties*.

Hint Immediate *equal_sym*: *set*.
Hint *Resolve equal_refl equal_trans* : *set*.

Hint Immediate *add_remove remove_add union_sym inter_sym*: *set*.
Hint *Resolve subset_refl subset_equal subset_antisym*
subset_trans subset_empty subset_remove_3 subset_diff subset_add_3
subset_add_2 in_subset empty_is_empty_1 empty_is_empty_2 add_equal
remove_equal singleton_equal_add union_subset_equal union_equal_1
union_equal_2 union_assoc add_union_singleton union_add union_subset_1
union_subset_2 union_subset_3 inter_subset_equal inter_equal_1 inter_equal_2
inter_assoc union_inter_1 union_inter_2 inter_add_1 inter_add_2
empty_inter_1 empty_inter_2 empty_union_1 empty_union_2 empty_diff_1
empty_diff_2 union_Add inter_Add union_Equal inter_Add_2 not_in_union
inter_subset_1 inter_subset_2 inter_subset_3 diff_subset diff_subset_equal
remove_diff_singleton diff_inter_empty diff_inter_all Add_add Add_remove
Equal_remove add_add : *set*.

222.2 Alternative (weaker) specifications for *fold*

Section *Old_Spec_Now_Properties*.

Notation *NoDup* := (*NoDupA E.eq*).

When *FSets* was first designed, the order in which Ocaml's *Set.fold* takes the set elements was unspecified. This specification reflects this fact:

Lemma *fold_0* :
∀ *s* (*A* : Set) (*i* : *A*) (*f* : *elt* → *A* → *A*),
∃ *l* : *list elt*,
NoDup l ∧
(∀ *x* : *elt*, *In x s* ↔ *InA E.eq x l*) ∧
fold f s i = *fold_right f i l*.

An alternate (and previous) specification for *fold* was based on the recursive structure of a set. It is now lemmas *fold_1* and *fold_2*.

Lemma *fold_1* :
∀ *s* (*A* : Set) (*eqA* : *A* → *A* → Prop)
(*st* : *Setoid_Theory A eqA*) (*i* : *A*) (*f* : *elt* → *A* → *A*),
Empty s → *eqA* (*fold f s i*) *i*.

Lemma *fold_2* :
∀ *s s' x* (*A* : Set) (*eqA* : *A* → *A* → Prop)
(*st* : *Setoid_Theory A eqA*) (*i* : *A*) (*f* : *elt* → *A* → *A*),
compat_op E.eq eqA f →
transpose eqA f →
¬ *In x s* → *Add x s s'* → *eqA* (*fold f s' i*) (*f x* (*fold f s i*)).

Similar specifications for *cardinal*.

Lemma *cardinal_fold* : ∀ *s*, *cardinal s* = *fold* (fun _ ⇒ *S*) *s* 0.

Lemma *cardinal_0* :
∀ *s*, ∃ *l* : *list elt*,
NoDupA E.eq l ∧
(∀ *x* : *elt*, *In x s* ↔ *InA E.eq x l*) ∧
cardinal s = *length l*.

Lemma *cardinal_1* : ∀ *s*, *Empty s* → *cardinal s* = 0.

Lemma *cardinal_2* :
∀ *s s' x*, ¬ *In x s* → *Add x s s'* → *cardinal s'* = *S* (*cardinal s*).

End *Old_Spec_Now_Properties*.

222.3 Induction principle over sets

Lemma *cardinal_inv_1* : ∀ *s*, *cardinal s* = 0 → *Empty s*.
Hint *Resolve cardinal_inv_1*.

Lemma *cardinal_inv_2* :
∀ *s n*, *cardinal s* = *S n* → { *x* : *elt* | *In x s* }.

Lemma *Equal_cardinal_aux* :

$\forall\ n\ s\ s'$, $cardinal\ s = n \rightarrow s[=]s' \rightarrow cardinal\ s = cardinal\ s'$.

Lemma $Equal\_cardinal : \forall\ s\ s', s[=]s' \rightarrow cardinal\ s = cardinal\ s'$.

$Add\ Morphism\ cardinal : cardinal\_m$.

Hint $Resolve\ Add\_add\ Add\_remove\ Equal\_remove\ cardinal\_inv\_1\ Equal\_cardinal$.

Lemma $cardinal\_induction$:
$\forall\ P : t \rightarrow \mathsf{Type}$,
$(\forall\ s, Empty\ s \rightarrow P\ s) \rightarrow$
$(\forall\ s\ s', P\ s \rightarrow \forall\ x, \neg In\ x\ s \rightarrow Add\ x\ s\ s' \rightarrow P\ s') \rightarrow$
$\forall\ n\ s, cardinal\ s = n \rightarrow P\ s$.

Lemma $set\_induction$:
$\forall\ P : t \rightarrow \mathsf{Type}$,
$(\forall\ s : t, Empty\ s \rightarrow P\ s) \rightarrow$
$(\forall\ s\ s' : t, P\ s \rightarrow \forall\ x : elt, \neg In\ x\ s \rightarrow Add\ x\ s\ s' \rightarrow P\ s') \rightarrow$
$\forall\ s : t, P\ s$.

Other properties of *fold*.

Section $Fold$.
Variables $(A{:}\mathsf{Set})(eqA{:}A{\rightarrow}A{\rightarrow}\mathsf{Prop})(st{:}Setoid\_Theory\ \_\ eqA)$.
Variables $(f{:}elt{\rightarrow}A{\rightarrow}A)(Comp{:}compat\_op\ E.eq\ eqA\ f)(Ass{:}transpose\ eqA\ f)$.

Section $Fold\_1$.
Variable $i\ i'{:}A$.

Lemma $fold\_empty : eqA\ (fold\ f\ empty\ i)\ i$.

Lemma $fold\_equal$:
$\forall\ s\ s', s[=]s' \rightarrow eqA\ (fold\ f\ s\ i)\ (fold\ f\ s'\ i)$.

Lemma $fold\_add : \forall\ s\ x, \neg In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (add\ x\ s)\ i)\ (f\ x\ (fold\ f\ s\ i))$.

Lemma $add\_fold : \forall\ s\ x, In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (add\ x\ s)\ i)\ (fold\ f\ s\ i)$.

Lemma $remove\_fold\_1 : \forall\ s\ x, In\ x\ s \rightarrow$
$eqA\ (f\ x\ (fold\ f\ (remove\ x\ s)\ i))\ (fold\ f\ s\ i)$.

Lemma $remove\_fold\_2 : \forall\ s\ x, \neg In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (remove\ x\ s)\ i)\ (fold\ f\ s\ i)$.

Lemma $fold\_commutes : \forall\ s\ x$,
$eqA\ (fold\ f\ s\ (f\ x\ i))\ (f\ x\ (fold\ f\ s\ i))$.

Lemma $fold\_init : \forall\ s, eqA\ i\ i' \rightarrow$
$eqA\ (fold\ f\ s\ i)\ (fold\ f\ s\ i')$.

End $Fold\_1$.
Section $Fold\_2$.
Variable $i{:}A$.

Lemma *fold_union_inter* : ∀ *s s'*,
eqA (*fold f* (*union s s'*) (*fold f* (*inter s s'*) *i*))
(*fold f s* (*fold f s'* *i*)).

End *Fold_2*.
Section *Fold_3*.
Variable *i*:*A*.

Lemma *fold_diff_inter* : ∀ *s s'*,
eqA (*fold f* (*diff s s'*) (*fold f* (*inter s s'*) *i*)) (*fold f s i*).

Lemma *fold_union*: ∀ *s s'*, (∀ *x*, ¬*In x s*\/~*In x s'*) →
eqA (*fold f* (*union s s'*) *i*) (*fold f s* (*fold f s'* *i*)).

End *Fold_3*.
End *Fold*.

Lemma *fold_plus* :
∀ *s p*, *fold* (fun _ ⇒ *S*) *s p* = *fold* (fun _ ⇒ *S*) *s* 0 + *p*.

properties of *cardinal*

Lemma *empty_cardinal* : *cardinal empty* = 0.

Hint Immediate *empty_cardinal cardinal_1* : *set*.

Lemma *singleton_cardinal* : ∀ *x*, *cardinal* (*singleton x*) = 1.

Hint *Resolve singleton_cardinal*: *set*.

Lemma *diff_inter_cardinal* :
∀ *s s'*, *cardinal* (*diff s s'*) + *cardinal* (*inter s s'*) = *cardinal s* .

Lemma *union_cardinal*:
∀ *s s'*, (∀ *x*, ¬*In x s*\/~*In x s'*) →
cardinal (*union s s'*)=*cardinal s*+*cardinal s'*.

Lemma *subset_cardinal* :
∀ *s s'*, *s*[<=]*s'* → *cardinal s* ≤ *cardinal s'* .

Lemma *subset_cardinal_lt* :
∀ *s s' x*, *s*[<=]*s'* → *In x s'* → ¬*In x s* → *cardinal s* < *cardinal s'*.

Theorem *union_inter_cardinal* :
∀ *s s'*, *cardinal* (*union s s'*) + *cardinal* (*inter s s'*) = *cardinal s* + *cardinal s'* .

Lemma *union_cardinal_inter* :
∀ *s s'*, *cardinal* (*union s s'*) = *cardinal s* + *cardinal s'* - *cardinal* (*inter s s'*).

Lemma *union_cardinal_le* :
∀ *s s'*, *cardinal* (*union s s'*) ≤ *cardinal s* + *cardinal s'*.

Lemma *add_cardinal_1* :
∀ *s x*, *In x s* → *cardinal* (*add x s*) = *cardinal s*.

Lemma *add_cardinal_2* :

$\forall\ s\ x, \neg In\ x\ s \rightarrow cardinal\ (add\ x\ s) = S\ (cardinal\ s)$.

Lemma *remove_cardinal_1* :
$\forall\ s\ x, In\ x\ s \rightarrow S\ (cardinal\ (remove\ x\ s)) = cardinal\ s$.

Lemma *remove_cardinal_2* :
$\forall\ s\ x, \neg In\ x\ s \rightarrow cardinal\ (remove\ x\ s) = cardinal\ s$.

Hint *Resolve subset_cardinal union_cardinal add_cardinal_1 add_cardinal_2*.

End *Properties*.

Chapter 223

Module Coq.FSets.FSets

Require Export *OrderedType*.
Require Export *OrderedTypeEx*.
Require Export *OrderedTypeAlt*.
Require Export *FSetInterface*.
Require Export *FSetBridge*.
Require Export *FSetProperties*.
Require Export *FSetEqProperties*.
Require Export *FSetList*.

Chapter 224

Module Coq.FSets.FSetToFiniteSet

Require Import *Ensembles Finite_sets*.
Require Import *FSetInterface FSetProperties OrderedTypeEx*.

224.1 Going from *FSets* with usual equality

to the old *Ensembles* and *Finite_sets* theory.

Module *S_to_Finite_set* (*U*:*UsualOrderedType*)(*M*:*S* with Module *E* := *U*).
 Module *MP*:= *Properties*(*M*).
 Import *M MP FM Ensembles Finite_sets*.

 Definition *mkEns* : *M.t* → *Ensemble M.elt* :=
  fun *s x* ⇒ *M.In x s*.

 Notation " !! " := *mkEns*.

 Lemma *In_In* : ∀ *s x*, *M.In x s* ↔ *In* _ (!!*s*) *x*.

 Lemma *Subset_Included* : ∀ *s s'*, *s*[<=]*s'* ↔ *Included* _ (!!*s*) (!!*s'*).

 Notation " *a* === *b* " := (*Same_set M.elt a b*) (*at level* 70, *no associativity*).

 Lemma *Equal_Same_set* : ∀ *s s'*, *s*[=]*s'* ↔ !!*s* === !!*s'*.

 Lemma *empty_Empty_Set* : !!*M.empty* === *Empty_set* _.

 Lemma *Empty_Empty_set* : ∀ *s*, *Empty s* → !!*s* === *Empty_set* _.

 Lemma *singleton_Singleton* : ∀ *x*, !!(*M.singleton x*) === *Singleton* _ *x* .

 Lemma *union_Union* : ∀ *s s'*, !!(*union s s'*) === *Union* _ (!!*s*) (!!*s'*).

 Lemma *inter_Intersection* : ∀ *s s'*, !!(*inter s s'*) === *Intersection* _ (!!*s*) (!!*s'*).

 Lemma *add_Add* : ∀ *x s*, !!(*add x s*) === *Add* _ (!!*s*) *x*.

 Lemma *Add_Add* : ∀ *x s s'*, *MP.Add x s s'* → !!*s'* === *Add* _ (!!*s*) *x*.

 Lemma *remove_Subtract* : ∀ *x s*, !!(*remove x s*) === *Subtract* _ (!!*s*) *x*.

 Lemma *mkEns_Finite* : ∀ *s*, *Finite* _ (!!*s*).

Lemma *mkEns_cardinal* : ∀ *s*, *cardinal* _ (!!*s*) (*M.cardinal s*).

End *S_to_Finite_set*.

Chapter 225

Module Coq.FSets.FSetWeakFacts

225.1 Finite sets library

This functor derives additional facts from *FSetInterface.S*. These facts are mainly the specifications of *FSetInterface.S* written using different styles: equivalence and boolean equalities. Moreover, we prove that *E.Eq* and *Equal* are setoid equalities.

Require Export *FSetWeakInterface*.

Module *Facts* (*M*: *S*).
Import *M.E*.
Import *M*.
Import *Logic*.

225.2 Specifications written using equivalences

Section *IffSpec*.
Variable *s s' s''* : *t*.
Variable *x y z* : *elt*.

Lemma *In_eq_iff* : *E.eq x y* → (*In x s* ↔ *In y s*).

Lemma *mem_iff* : *In x s* ↔ *mem x s* = *true*.

Lemma *not_mem_iff* : ¬*In x s* ↔ *mem x s* = *false*.

Lemma *equal_iff* : *s*[=]*s'* ↔ *equal s s'* = *true*.

Lemma *subset_iff* : *s*[<=]*s'* ↔ *subset s s'* = *true*.

Lemma *empty_iff* : *In x empty* ↔ *False*.

Lemma *is_empty_iff* : *Empty s* ↔ *is_empty s* = *true*.

Lemma *singleton_iff* : *In y* (*singleton x*) ↔ *E.eq x y*.

Lemma *add_iff* : *In y* (*add x s*) ↔ *E.eq x y* ∨ *In y s*.

Lemma *add_neq_iff* : ¬ *E.eq x y* → (*In y* (*add x s*) ↔ *In y s*).

Lemma *remove_iff* : *In y* (*remove x s*) ↔ *In y s* ∧ ¬*E.eq x y*.

Lemma *remove_neq_iff* : ¬ *E.eq x y* → (*In y* (*remove x s*) ↔ *In y s*).

Lemma *union_iff* : *In x* (*union s s'*) ↔ *In x s* ∨ *In x s'*.

Lemma *inter_iff* : *In x* (*inter s s'*) ↔ *In x s* ∧ *In x s'*.

Lemma *diff_iff* : *In x* (*diff s s'*) ↔ *In x s* ∧ ¬ *In x s'*.

Variable *f* : *elt*→*bool*.

Lemma *filter_iff* : *compat_bool E.eq f* → (*In x* (*filter f s*) ↔ *In x s* ∧ *f x* = *true*).

Lemma *for_all_iff* : *compat_bool E.eq f* →
(*For_all* (fun *x* ⇒ *f x* = *true*) *s* ↔ *for_all f s* = *true*).

Lemma *exists_iff* : *compat_bool E.eq f* →
(*Exists* (fun *x* ⇒ *f x* = *true*) *s* ↔ *exists_ f s* = *true*).

Lemma *elements_iff* : *In x s* ↔ *InA E.eq x* (*elements s*).

End *IffSpec*.

Useful tactic for simplifying expressions like *In y* (*add x* (*union s s'*))

Ltac *set_iff* :=
repeat (*progress* (
rewrite add_iff || *rewrite remove_iff* || *rewrite singleton_iff*
|| *rewrite union_iff* || *rewrite inter_iff* || *rewrite diff_iff*
|| *rewrite empty_iff*)).

225.3 Specifications written using boolean predicates

Definition *eqb x y* := if *eq_dec x y* then *true* else *false*.

Section *BoolSpec*.
Variable *s s' s"* : *t*.
Variable *x y z* : *elt*.

Lemma *mem_b* : *E.eq x y* → *mem x s* = *mem y s*.

Lemma *empty_b* : *mem y empty* = *false*.

Lemma *add_b* : *mem y* (*add x s*) = *eqb x y* || *mem y s*.

Lemma *add_neq_b* : ¬ *E.eq x y* → *mem y* (*add x s*) = *mem y s*.

Lemma *remove_b* : *mem y* (*remove x s*) = *mem y s* && *negb* (*eqb x y*).

Lemma *remove_neq_b* : ¬ *E.eq x y* → *mem y* (*remove x s*) = *mem y s*.

Lemma *singleton_b* : *mem y* (*singleton x*) = *eqb x y*.

Lemma *union_b* : *mem x* (*union s s'*) = *mem x s* || *mem x s'*.

Lemma *inter_b* : *mem x (inter s s') = mem x s && mem x s'*.

Lemma *diff_b* : *mem x (diff s s') = mem x s && negb (mem x s')*.

Lemma *elements_b* : *mem x s = existsb (eqb x) (elements s)*.

Variable *f* : *elt→bool*.

Lemma *filter_b* : *compat_bool E.eq f → mem x (filter f s) = mem x s && f x*.

Lemma *for_all_b* : *compat_bool E.eq f →*
for_all f s = forallb f (elements s).

Lemma *exists_b* : *compat_bool E.eq f →*
exists_ f s = existsb f (elements s).

End *BoolSpec*.

225.4 *E.eq* and *Equal* are setoid equalities

Definition *E_ST* : *Setoid_Theory elt E.eq*.

Add Setoid elt E.eq E_ST as *EltSetoid*.

Definition *Equal_ST* : *Setoid_Theory t Equal*.

Add Setoid t Equal Equal_ST as *EqualSetoid*.

Add Morphism In with *signature E.eq ==> Equal ==> iff* as *In_m*.

Add Morphism is_empty : *is_empty_m*.

Add Morphism Empty with *signature Equal ==> iff* as *Empty_m*.

Add Morphism mem : *mem_m*.

Add Morphism singleton : *singleton_m*.

Add Morphism add : *add_m*.

Add Morphism remove : *remove_m*.

Add Morphism union : *union_m*.

Add Morphism inter : *inter_m*.

Add Morphism diff : *diff_m*.

Add Morphism Subset with *signature Equal ==> Equal ==> iff* as *Subset_m*.

Add Morphism subset : *subset_m*.

Add Morphism equal : *equal_m*.

Lemma *filter_equal* : ∀ *f*, *compat_bool E.eq f →*
∀ *s s'*, *s[=]s' → filter f s [=] filter f s'*.

End *Facts*.

Chapter 226

Module Coq.FSets.FSetWeakInterface

226.1 Finite sets library

Set interfaces for types with only a decidable equality, but no ordering

Require Export *Bool*.
Require Export *DecidableType*.

Compatibility of a boolean function with respect to an equality.
Definition *compat_bool* (*A*:Set)(*eqA*: *A*→*A*→Prop)(*f*: *A*→ *bool*) :=
  ∀ *x y* : *A*, *eqA x y* → *f x* = *f y*.

Compatibility of a predicate with respect to an equality.
Definition *compat_P* (*A*:Set)(*eqA*: *A*→*A*→Prop)(*P* : *A* → Prop) :=
  ∀ *x y* : *A*, *eqA x y* → *P x* → *P y*.

Hint *Unfold compat_bool compat_P*.

226.2 Non-dependent signature

Signature *S* presents sets as purely informative programs together with axioms

Module Type *S*.

  Declare Module *E* : *DecidableType*.
  Definition *elt* := *E.t*.

  Parameter *t* : Set.
the abstract type of sets

Logical predicates
  Parameter *In* : *elt* → *t* → Prop.
  Definition *Equal s s'* := ∀ *a* : *elt*, *In a s* ↔ *In a s'*.
  Definition *Subset s s'* := ∀ *a* : *elt*, *In a s* → *In a s'*.
  Definition *Empty s* := ∀ *a* : *elt*, ¬ *In a s*.

Definition *For_all* $(P : elt \to \mathsf{Prop})\ s := \forall\ x,\ In\ x\ s \to P\ x.$
Definition *Exists* $(P : elt \to \mathsf{Prop})\ s := \exists\ x,\ In\ x\ s \wedge P\ x.$

Notation "s [=] t" := $(Equal\ s\ t)$ ($at\ level$ 70, $no\ associativity$).
Notation "s [<=] t" := $(Subset\ s\ t)$ ($at\ level$ 70, $no\ associativity$).

Parameter *empty* : t.
The empty set.

Parameter *is_empty* : $t \to bool$.
Test whether a set is empty or not.

Parameter *mem* : $elt \to t \to bool$.
mem x s tests whether x belongs to the set s.

Parameter *add* : $elt \to t \to t$.
add x s returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged.

Parameter *singleton* : $elt \to t$.
singleton x returns the one-element set containing only x.

Parameter *remove* : $elt \to t \to t$.
remove x s returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged.

Parameter *union* : $t \to t \to t$.
Set union.

Parameter *inter* : $t \to t \to t$.
Set intersection.

Parameter *diff* : $t \to t \to t$.
Set difference.

Parameter *equal* : $t \to t \to bool$.
equal *s1* *s2* tests whether the sets *s1* and *s2* are equal, that is, contain equal elements.

Parameter *subset* : $t \to t \to bool$.
subset *s1* *s2* tests whether the set *s1* is a subset of the set *s2*.

Coq comment: *iter* is useless in a purely functional world
iter: (elt -> unit) -> set -> unit. i
iter f s applies f in turn to all elements of s. The order in which the elements of s are presented to f is unspecified.

Parameter *fold* : $\forall\ A : \mathsf{Set}, (elt \to A \to A) \to t \to A \to A$.
fold f s a computes $(f\ xN\ \ldots\ (f\ x2\ (f\ x1\ a))\ldots)$, where *x1* ... *xN* are the elements of s. The order in which elements of s are presented to f is unspecified.

Parameter *for_all* : $(elt \to bool) \to t \to bool$.
for_all p s checks if all elements of the set satisfy the predicate p.

Parameter *exists_* : $(elt \to bool) \to t \to bool$.
$\exists$ p s checks if at least one element of the set satisfies the predicate p.

Parameter *filter* : $(elt \to bool) \to t \to t$.
filter p s returns the set of all elements in s that satisfy predicate p.

Parameter *partition* : $(elt \to bool) \to t \to t \times t$.
partition p s returns a pair of sets $(s1, s2)$, where $s1$ is the set of all the elements of s that satisfy the predicate p, and $s2$ is the set of all the elements of s that do not satisfy p.

Parameter *cardinal* : $t \to nat$.
Return the number of elements of a set.
Coq comment: nat instead of int ...

Parameter *elements* : $t \to list\ elt$.
Return the list of all elements of the given set, in any order.

Parameter *choose* : $t \to option\ elt$.
Return one element of the given set, or raise *Not_found* if the set is empty. Which element is chosen is unspecified. Equal sets could return different elements.
Coq comment: *Not_found* is represented by the option type

Section *Spec*.

Variable $s\ s'$: t.
Variable $x\ y$: elt.

Specification of *In*
Parameter *In_1* : $E.eq\ x\ y \to In\ x\ s \to In\ y\ s$.

Specification of *mem*
Parameter *mem_1* : $In\ x\ s \to mem\ x\ s = true$.
Parameter *mem_2* : $mem\ x\ s = true \to In\ x\ s$.

Specification of *equal*
Parameter *equal_1* : $Equal\ s\ s' \to equal\ s\ s' = true$.
Parameter *equal_2* : $equal\ s\ s' = true \to Equal\ s\ s'$.

Specification of *subset*
Parameter *subset_1* : $Subset\ s\ s' \to subset\ s\ s' = true$.
Parameter *subset_2* : $subset\ s\ s' = true \to Subset\ s\ s'$.

Specification of *empty*
Parameter *empty_1* : $Empty\ empty$.

Specification of *is_empty*
Parameter *is_empty_1* : $Empty\ s \to is\_empty\ s = true$.
Parameter *is_empty_2* : $is\_empty\ s = true \to Empty\ s$.

Specification of *add*
Parameter *add_1* : $E.eq\ x\ y \to In\ y\ (add\ x\ s)$.
Parameter *add_2* : $In\ y\ s \to In\ y\ (add\ x\ s)$.
Parameter *add_3* : $\neg\ E.eq\ x\ y \to In\ y\ (add\ x\ s) \to In\ y\ s$.

Specification of *remove*
Parameter *remove_1* : $E.eq\ x\ y \to \neg\ In\ y\ (remove\ x\ s)$.
Parameter *remove_2* : $\neg\ E.eq\ x\ y \to In\ y\ s \to In\ y\ (remove\ x\ s)$.

Parameter $remove\_3$: $In\ y\ (remove\ x\ s) \to In\ y\ s$.

Specification of *singleton*

Parameter $singleton\_1$: $In\ y\ (singleton\ x) \to E.eq\ x\ y$.
Parameter $singleton\_2$: $E.eq\ x\ y \to In\ y\ (singleton\ x)$.

Specification of *union*

Parameter $union\_1$: $In\ x\ (union\ s\ s') \to In\ x\ s \vee In\ x\ s'$.
Parameter $union\_2$: $In\ x\ s \to In\ x\ (union\ s\ s')$.
Parameter $union\_3$: $In\ x\ s' \to In\ x\ (union\ s\ s')$.

Specification of *inter*

Parameter $inter\_1$: $In\ x\ (inter\ s\ s') \to In\ x\ s$.
Parameter $inter\_2$: $In\ x\ (inter\ s\ s') \to In\ x\ s'$.
Parameter $inter\_3$: $In\ x\ s \to In\ x\ s' \to In\ x\ (inter\ s\ s')$.

Specification of *diff*

Parameter $diff\_1$: $In\ x\ (diff\ s\ s') \to In\ x\ s$.
Parameter $diff\_2$: $In\ x\ (diff\ s\ s') \to \neg\ In\ x\ s'$.
Parameter $diff\_3$: $In\ x\ s \to \neg\ In\ x\ s' \to In\ x\ (diff\ s\ s')$.

Specification of *fold*

Parameter $fold\_1$: $\forall\ (A : \mathsf{Set})\ (i : A)\ (f : elt \to A \to A)$,
$fold\ f\ s\ i = fold\_left\ (\mathsf{fun}\ a\ e \Rightarrow f\ e\ a)\ (elements\ s)\ i$.

Specification of *cardinal*

Parameter $cardinal\_1$: $cardinal\ s = length\ (elements\ s)$.

Section *Filter*.

Variable f : $elt \to bool$.

Specification of *filter*

Parameter $filter\_1$: $compat\_bool\ E.eq\ f \to In\ x\ (filter\ f\ s) \to In\ x\ s$.
Parameter $filter\_2$: $compat\_bool\ E.eq\ f \to In\ x\ (filter\ f\ s) \to f\ x = true$.
Parameter $filter\_3$:
$compat\_bool\ E.eq\ f \to In\ x\ s \to f\ x = true \to In\ x\ (filter\ f\ s)$.

Specification of *for_all*

Parameter for_all_1 :
$compat\_bool\ E.eq\ f \to$
$For\_all\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s \to for\_all\ f\ s = true$.
Parameter for_all_2 :
$compat\_bool\ E.eq\ f \to$
$for\_all\ f\ s = true \to For\_all\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s$.

Specification of $\exists$

Parameter $exists\_1$:
$compat\_bool\ E.eq\ f \to$
$Exists\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s \to exists\_\ f\ s = true$.
Parameter $exists\_2$:
$compat\_bool\ E.eq\ f \to$

exists_ f s = true → Exists (fun *x* ⇒ *f x = true*) *s*.

Specification of *partition*

Parameter *partition_1* :
compat_bool E.eq f → Equal (*fst* (*partition f s*)) (*filter f s*).
Parameter *partition_2* :
compat_bool E.eq f →
Equal (*snd* (*partition f s*)) (*filter* (fun *x* ⇒ *negb* (*f x*)) *s*).

End *Filter*.

Specification of *elements*

Parameter *elements_1* : *In x s → InA E.eq x* (*elements s*).
Parameter *elements_2* : *InA E.eq x* (*elements s*) → *In x s*.
Parameter *elements_3* : *NoDupA E.eq* (*elements s*).

Specification of *choose*

Parameter *choose_1* : *choose s = Some x → In x s*.
Parameter *choose_2* : *choose s = None → Empty s*.

End *Spec*.

Hint Immediate *In_1*.

Hint *Resolve mem_1 mem_2 equal_1 equal_2 subset_1 subset_2 empty_1 is_empty_1 is_empty_2 choose_1 choose_2 add_1 add_2 add_3 remove_1 remove_2 remove_3 singleton_1 singleton_2 union_1 union_2 union_3 inter_1 inter_2 inter_3 diff_1 diff_2 diff_3 filter_1 filter_2 filter_3 for_all_1 for_all_2 exists_1 exists_2 partition_1 partition_2 elements_1 elements_2 elements_3*.

End *S*.

Chapter 227

Module Coq.FSets.FSetWeakList

227.1 Finite sets library

This file proposes an implementation of the non-dependant interface *FSetWeakInterface.S* using lists without redundancy.

Require Import *FSetWeakInterface*.

227.2 Functions over lists

First, we provide sets as lists which are (morally) without redundancy. The specs are proved under the additional condition of no redundancy. And the functions returning sets are proved to preserve this invariant.

Module *Raw* (*X*: *DecidableType*).

Module *E* := *X*.

Definition *elt* := *X.t*.
Definition *t* := *list elt*.

Definition *empty* : *t* := *nil*.

Definition *is_empty* (*l* : *t*) : *bool* := if *l* then *true* else *false*.

227.2.1 The set operations.

```
Fixpoint mem (x : elt) (s : t) {struct s} : bool :=
  match s with
  | nil ⇒ false
  | y :: l ⇒
          if X.eq_dec x y then true else mem x l
  end.
```

```
Fixpoint add (x : elt) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ x :: nil
  | y :: l ⇒
      if X.eq_dec x y then s else y :: add x l
  end.

Definition singleton (x : elt) : t := x :: nil.

Fixpoint remove (x : elt) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ nil
  | y :: l ⇒
      if X.eq_dec x y then l else y :: remove x l
  end.

Fixpoint fold (B : Set) (f : elt → B → B) (s : t) {struct s} :
 B → B := fun i ⇒ match s with
                  | nil ⇒ i
                  | x :: l ⇒ fold f l (f x i)
                  end.

Definition union (s : t) : t → t := fold add s.

Definition diff (s s' : t) : t := fold remove s' s.

Definition inter (s s': t) : t :=
  fold (fun x s ⇒ if mem x s' then add x s else s) s nil.

Definition subset (s s' : t) : bool := is_empty (diff s s').

Definition equal (s s' : t) : bool := andb (subset s s') (subset s' s).

Fixpoint filter (f : elt → bool) (s : t) {struct s} : t :=
  match s with
  | nil ⇒ nil
  | x :: l ⇒ if f x then x :: filter f l else filter f l
  end.

Fixpoint for_all (f : elt → bool) (s : t) {struct s} : bool :=
  match s with
  | nil ⇒ true
  | x :: l ⇒ if f x then for_all f l else false
  end.

Fixpoint exists_ (f : elt → bool) (s : t) {struct s} : bool :=
  match s with
  | nil ⇒ false
  | x :: l ⇒ if f x then true else exists_ f l
  end.

Fixpoint partition (f : elt → bool) (s : t) {struct s} :
 t × t :=
```

```
  match s with
  | nil ⇒ (nil, nil)
  | x :: l ⇒
      let (s1, s2) := partition f l in
      if f x then (x :: s1, s2) else (s1, x :: s2)
  end.

Definition cardinal (s : t) : nat := length s.

Definition elements (s : t) : list elt := s.

Definition choose (s : t) : option elt :=
  match s with
  | nil ⇒ None
  | x::_ ⇒ Some x
  end.
```

227.2.2 Proofs of set operation specifications.

```
Section ForNotations.
Notation NoDup := (NoDupA X.eq).
Notation In := (InA X.eq).

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) s := ∃ x, In x s ∧ P x.

Lemma In_eq :
  ∀ (s : t) (x y : elt), X.eq x y → In x s → In y s.
Hint Immediate In_eq.

Lemma mem_1 :
 ∀ (s : t)(x : elt), In x s → mem x s = true.

Lemma mem_2 : ∀ (s : t) (x : elt), mem x s = true → In x s.

Lemma add_1 :
 ∀ (s : t) (Hs : NoDup s) (x y : elt), X.eq x y → In y (add x s).

Lemma add_2 :
 ∀ (s : t) (Hs : NoDup s) (x y : elt), In y s → In y (add x s).

Lemma add_3 :
 ∀ (s : t) (Hs : NoDup s) (x y : elt),
 ¬ X.eq x y → In y (add x s) → In y s.

Lemma add_unique :
 ∀ (s : t) (Hs : NoDup s)(x:elt), NoDup (add x s).

Lemma remove_1 :
```

$\forall\ (s : t)\ (Hs : NoDup\ s)\ (x\ y : elt),\ X.eq\ x\ y \to \neg\ In\ y\ (remove\ x\ s).$

Lemma *remove_2* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (x\ y : elt),$
$\neg\ X.eq\ x\ y \to In\ y\ s \to In\ y\ (remove\ x\ s).$

Lemma *remove_3* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (x\ y : elt),\ In\ y\ (remove\ x\ s) \to In\ y\ s.$

Lemma *remove_unique* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (x : elt),\ NoDup\ (remove\ x\ s).$

Lemma *singleton_unique* : $\forall\ x : elt,\ NoDup\ (singleton\ x).$

Lemma *singleton_1* : $\forall\ x\ y : elt,\ In\ y\ (singleton\ x) \to X.eq\ x\ y.$

Lemma *singleton_2* : $\forall\ x\ y : elt,\ X.eq\ x\ y \to In\ y\ (singleton\ x).$

Lemma *empty_unique* : $NoDup\ empty.$

Lemma *empty_1* : $Empty\ empty.$

Lemma *is_empty_1* : $\forall\ s : t,\ Empty\ s \to is\_empty\ s = true.$

Lemma *is_empty_2* : $\forall\ s : t,\ is\_empty\ s = true \to Empty\ s.$

Lemma *elements_1* : $\forall\ (s : t)\ (x : elt),\ In\ x\ s \to In\ x\ (elements\ s).$

Lemma *elements_2* : $\forall\ (s : t)\ (x : elt),\ In\ x\ (elements\ s) \to In\ x\ s.$

Lemma *elements_3* : $\forall\ (s : t)\ (Hs : NoDup\ s),\ NoDup\ (elements\ s).$

Lemma *fold_1* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (A : \mathsf{Set})\ (i : A)\ (f : elt \to A \to A),$
$fold\ f\ s\ i = fold\_left\ (\mathsf{fun}\ a\ e \Rightarrow f\ e\ a)\ (elements\ s)\ i.$

Lemma *union_unique* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s'),\ NoDup\ (union\ s\ s').$

Lemma *union_1* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (union\ s\ s') \to In\ x\ s \vee In\ x\ s'.$

Lemma *union_0* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ s \vee In\ x\ s' \to In\ x\ (union\ s\ s').$

Lemma *union_2* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ s \to In\ x\ (union\ s\ s').$

Lemma *union_3* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ s' \to In\ x\ (union\ s\ s').$

Lemma *inter_unique* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s'),\ NoDup\ (inter\ s\ s').$

Lemma *inter_0* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (inter\ s\ s') \rightarrow In\ x\ s \wedge In\ x\ s'.$

Lemma *inter_1* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (inter\ s\ s') \rightarrow In\ x\ s.$

Lemma *inter_2* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (inter\ s\ s') \rightarrow In\ x\ s'.$

Lemma *inter_3* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ s \rightarrow In\ x\ s' \rightarrow In\ x\ (inter\ s\ s').$

Lemma *diff_unique* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s'),\ NoDup\ (diff\ s\ s').$

Lemma *diff_0* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (diff\ s\ s') \rightarrow In\ x\ s \wedge \neg\ In\ x\ s'.$

Lemma *diff_1* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (diff\ s\ s') \rightarrow In\ x\ s.$

Lemma *diff_2* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ (diff\ s\ s') \rightarrow \neg\ In\ x\ s'.$

Lemma *diff_3* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s')\ (x : elt),$
$In\ x\ s \rightarrow \neg\ In\ x\ s' \rightarrow In\ x\ (diff\ s\ s').$

Lemma *subset_1* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s'),$
$Subset\ s\ s' \rightarrow subset\ s\ s' = true.$

Lemma *subset_2* : $\forall\ (s\ s' : t)(Hs : NoDup\ s)\ (Hs' : NoDup\ s'),$
$subset\ s\ s' = true \rightarrow Subset\ s\ s'.$

Lemma *equal_1* :
$\forall\ (s\ s' : t)\ (Hs : NoDup\ s)\ (Hs' : NoDup\ s'),$
$Equal\ s\ s' \rightarrow equal\ s\ s' = true.$

Lemma *equal_2* : $\forall\ (s\ s' : t)(Hs : NoDup\ s)\ (Hs' : NoDup\ s'),$
$equal\ s\ s' = true \rightarrow Equal\ s\ s'.$

Definition *choose_1* :
$\forall\ (s : t)\ (x : elt),\ choose\ s = Some\ x \rightarrow In\ x\ s.$

Definition *choose_2* : $\forall\ s : t,\ choose\ s = None \rightarrow Empty\ s.$

Lemma *cardinal_1* :
$\forall\ (s : t)\ (Hs : NoDup\ s),\ cardinal\ s = length\ (elements\ s).$

Lemma *filter_1* :
$\forall\ (s : t)\ (x : elt)\ (f : elt \to bool),$
$In\ x\ (filter\ f\ s) \to In\ x\ s.$

Lemma *filter_2* :
$\forall\ (s : t)\ (x : elt)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to In\ x\ (filter\ f\ s) \to f\ x = true.$

Lemma *filter_3* :
$\forall\ (s : t)\ (x : elt)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to In\ x\ s \to f\ x = true \to In\ x\ (filter\ f\ s).$

Lemma *filter_unique* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (f : elt \to bool),\ NoDup\ (filter\ f\ s).$

Lemma *for_all_1* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to$
$For\_all\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s \to for\_all\ f\ s = true.$

Lemma *for_all_2* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to$
$for\_all\ f\ s = true \to For\_all\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s.$

Lemma *exists_1* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to Exists\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s \to exists\_\ f\ s = true.$

Lemma *exists_2* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to exists\_\ f\ s = true \to Exists\ (\mathsf{fun}\ x \Rightarrow f\ x = true)\ s.$

Lemma *partition_1* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to Equal\ (fst\ (partition\ f\ s))\ (filter\ f\ s).$

Lemma *partition_2* :
$\forall\ (s : t)\ (f : elt \to bool),$
$compat\_bool\ X.eq\ f \to$
$Equal\ (snd\ (partition\ f\ s))\ (filter\ (\mathsf{fun}\ x \Rightarrow negb\ (f\ x))\ s).$

Lemma *partition_aux_1* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (f : elt \to bool)(x{:}elt),$
$In\ x\ (fst\ (partition\ f\ s)) \to In\ x\ s.$

Lemma *partition_aux_2* :
$\forall\ (s : t)\ (Hs : NoDup\ s)\ (f : elt \to bool)(x{:}elt),$
$In\ x\ (snd\ (partition\ f\ s)) \to In\ x\ s.$

```
  Lemma partition_unique_1 :
   ∀ (s : t) (Hs : NoDup s) (f : elt → bool), NoDup (fst (partition f s)).

  Lemma partition_unique_2 :
   ∀ (s : t) (Hs : NoDup s) (f : elt → bool), NoDup (snd (partition f s)).

  Definition eq : t → t → Prop := Equal.

  Lemma eq_refl : ∀ s : t, eq s s.

  Lemma eq_sym : ∀ s s' : t, eq s s' → eq s' s.

  Lemma eq_trans : ∀ s s' s'' : t, eq s s' → eq s' s'' → eq s s''.

  End ForNotations.
End Raw.
```

227.3 Encapsulation

Now, in order to really provide a functor implementing S, we need to encapsulate everything into a type of lists without redundancy.

```
Module Make (X: DecidableType) <: S with Module E := X.

  Module Raw := Raw X.
  Module E := X.

  Record slist : Set := {this :> Raw.t; unique : NoDupA E.eq this}.
  Definition t := slist.
  Definition elt := E.t.

  Definition In (x : elt) (s : t) : Prop := InA E.eq x s.(this).
  Definition Equal (s s':t) : Prop := ∀ a : elt, In a s ↔ In a s'.
  Definition Subset (s s':t) : Prop := ∀ a : elt, In a s → In a s'.
  Definition Empty (s:t) : Prop := ∀ a : elt, ¬ In a s.
  Definition For_all (P : elt → Prop) (s : t) : Prop :=
    ∀ x : elt, In x s → P x.
  Definition Exists (P : elt → Prop) (s : t) : Prop := ∃ x : elt, In x s ∧ P x.

  Definition mem (x : elt) (s : t) : bool := Raw.mem x s.
  Definition add (x : elt)(s : t) : t := Build_slist (Raw.add_unique (unique s) x).
  Definition remove (x : elt)(s : t) : t := Build_slist (Raw.remove_unique (unique s) x).
  Definition singleton (x : elt) : t := Build_slist (Raw.singleton_unique x).
  Definition union (s s' : t) : t :=
    Build_slist (Raw.union_unique (unique s) (unique s')).
  Definition inter (s s' : t) : t :=
    Build_slist (Raw.inter_unique (unique s) (unique s')).
  Definition diff (s s' : t) : t :=
    Build_slist (Raw.diff_unique (unique s) (unique s')).
  Definition equal (s s' : t) : bool := Raw.equal s s'.
  Definition subset (s s' : t) : bool := Raw.subset s s'.
```

```
Definition empty : t := Build_slist Raw.empty_unique.
Definition is_empty (s : t) : bool := Raw.is_empty s.
Definition elements (s : t) : list elt := Raw.elements s.
Definition choose (s:t) : option elt := Raw.choose s.
Definition fold (B : Set) (f : elt → B → B) (s : t) : B → B := Raw.fold (B:=B) f s.
Definition cardinal (s : t) : nat := Raw.cardinal s.
Definition filter (f : elt → bool) (s : t) : t :=
  Build_slist (Raw.filter_unique (unique s) f).
Definition for_all (f : elt → bool) (s : t) : bool := Raw.for_all f s.
Definition exists_ (f : elt → bool) (s : t) : bool := Raw.exists_ f s.
Definition partition (f : elt → bool) (s : t) : t × t :=
  let p := Raw.partition f s in
  (Build_slist (this:=fst p) (Raw.partition_unique_1 (unique s) f),
   Build_slist (this:=snd p) (Raw.partition_unique_2 (unique s) f)).

Section Spec.
 Variable s s' : t.
 Variable x y : elt.

 Lemma In_1 : E.eq x y → In x s → In y s.

 Lemma mem_1 : In x s → mem x s = true.
 Lemma mem_2 : mem x s = true → In x s.

 Lemma equal_1 : Equal s s' → equal s s' = true.
 Lemma equal_2 : equal s s' = true → Equal s s'.

 Lemma subset_1 : Subset s s' → subset s s' = true.
 Lemma subset_2 : subset s s' = true → Subset s s'.

 Lemma empty_1 : Empty empty.

 Lemma is_empty_1 : Empty s → is_empty s = true.
 Lemma is_empty_2 : is_empty s = true → Empty s.

 Lemma add_1 : E.eq x y → In y (add x s).
 Lemma add_2 : In y s → In y (add x s).
 Lemma add_3 : ¬ E.eq x y → In y (add x s) → In y s.

 Lemma remove_1 : E.eq x y → ¬ In y (remove x s).
 Lemma remove_2 : ¬ E.eq x y → In y s → In y (remove x s).
 Lemma remove_3 : In y (remove x s) → In y s.

 Lemma singleton_1 : In y (singleton x) → E.eq x y.
 Lemma singleton_2 : E.eq x y → In y (singleton x).

 Lemma union_1 : In x (union s s') → In x s ∨ In x s'.
 Lemma union_2 : In x s → In x (union s s').
 Lemma union_3 : In x s' → In x (union s s').

 Lemma inter_1 : In x (inter s s') → In x s.
 Lemma inter_2 : In x (inter s s') → In x s'.
 Lemma inter_3 : In x s → In x s' → In x (inter s s').
```

Lemma *diff_1* : *In x (diff s s')* → *In x s*.
Lemma *diff_2* : *In x (diff s s')* → ¬ *In x s'*.
Lemma *diff_3* : *In x s* → ¬ *In x s'* → *In x (diff s s')*.

Lemma *fold_1* : ∀ (*A* : Set) (*i* : *A*) (*f* : *elt* → *A* → *A*),
fold f s i = *fold_left* (fun *a e* ⇒ *f e a*) (*elements s*) *i*.

Lemma *cardinal_1* : *cardinal s* = *length* (*elements s*).

Section *Filter*.

Variable *f* : *elt* → *bool*.

Lemma *filter_1* : *compat_bool E.eq f* → *In x* (*filter f s*) → *In x s*.
Lemma *filter_2* : *compat_bool E.eq f* → *In x* (*filter f s*) → *f x* = *true*.
Lemma *filter_3* :
compat_bool E.eq f → *In x s* → *f x* = *true* → *In x* (*filter f s*).

Lemma *for_all_1* :
compat_bool E.eq f →
For_all (fun *x* ⇒ *f x* = *true*) *s* → *for_all f s* = *true*.
Lemma *for_all_2* :
compat_bool E.eq f →
for_all f s = *true* → *For_all* (fun *x* ⇒ *f x* = *true*) *s*.

Lemma *exists_1* :
compat_bool E.eq f →
Exists (fun *x* ⇒ *f x* = *true*) *s* → *exists_ f s* = *true*.
Lemma *exists_2* :
compat_bool E.eq f →
exists_ f s = *true* → *Exists* (fun *x* ⇒ *f x* = *true*) *s*.

Lemma *partition_1* :
compat_bool E.eq f → *Equal* (*fst* (*partition f s*)) (*filter f s*).
Lemma *partition_2* :
compat_bool E.eq f →
Equal (*snd* (*partition f s*)) (*filter* (fun *x* ⇒ *negb* (*f x*)) *s*).

End *Filter*.

Lemma *elements_1* : *In x s* → *InA E.eq x* (*elements s*).
Lemma *elements_2* : *InA E.eq x* (*elements s*) → *In x s*.
Lemma *elements_3* : *NoDupA E.eq* (*elements s*).

Lemma *choose_1* : *choose s* = *Some x* → *In x s*.
Lemma *choose_2* : *choose s* = *None* → *Empty s*.

End *Spec*.

End *Make*.

Chapter 228

Module Coq.FSets.FSetWeakProperties

228.1 Finite sets library

NB: this file is a clone of *FSetProperties* for weak sets and should remain so until we find a way to share the two.

This functor derives additional properties from *FSetWeakInterface.S*. Contrary to the functor in *FSetWeakEqProperties* it uses predicates over sets instead of sets operations, i.e. *In x s* instead of *mem x s=true*, *Equal s s'* instead of *equal s s'=true*, etc.

Require Export *FSetWeakInterface*.
Require Import *FSetWeakFacts*.

Hint *Unfold transpose compat_op*.
Hint *Extern* 1 (*Setoid_Theory* _ _) ⇒ *constructor*; *congruence*.

Module *Properties* (*M*: *S*).
  Import *M.E*.
  Import *M*.
  Import *Logic*.
  Import *Peano*.

Results about lists without duplicates

  Module *FM* := *Facts M*.
  Import *FM*.

  Definition *Add* (*x* : *elt*) (*s s'* : *t*) :=
    ∀ *y* : *elt*, *In y s'* ↔ *E.eq x y* ∨ *In y s*.

  Lemma *In_dec* : ∀ *x s*, {*In x s*} + {~ *In x s*}.

  Section *BasicProperties*.

properties of *Equal*

  Lemma *equal_refl* : ∀ *s*, *s*[=]*s*.

  Lemma *equal_sym* : ∀ *s s'*, *s*[=]*s'* → *s'*[=]*s*.

Lemma *equal_trans* : ∀ *s1 s2 s3*, *s1*[=]*s2* → *s2*[=]*s3* → *s1*[=]*s3*.

Variable *s s' s" s1 s2 s3* : *t*.
Variable *x x'* : *elt*.

properties of *Subset*

Lemma *subset_refl* : *s*[<=]*s*.

Lemma *subset_antisym* : *s*[<=]*s'* → *s'*[<=]*s* → *s*[=]*s'*.

Lemma *subset_trans* : *s1*[<=]*s2* → *s2*[<=]*s3* → *s1*[<=]*s3*.

Lemma *subset_equal* : *s*[=]*s'* → *s*[<=]*s'*.

Lemma *subset_empty* : *empty*[<=]*s*.

Lemma *subset_remove_3* : *s1*[<=]*s2* → *remove x s1* [<=] *s2*.

Lemma *subset_diff* : *s1*[<=]*s3* → *diff s1 s2* [<=] *s3*.

Lemma *subset_add_3* : *In x s2* → *s1*[<=]*s2* → *add x s1* [<=] *s2*.

Lemma *subset_add_2* : *s1*[<=]*s2* → *s1*[<=] *add x s2*.

Lemma *in_subset* : *In x s1* → *s1*[<=]*s2* → *In x s2*.

Lemma *double_inclusion* : *s1*[=]*s2* ↔ *s1*[<=]*s2* ∧ *s2*[<=]*s1*.

properties of *empty*

Lemma *empty_is_empty_1* : *Empty s* → *s*[=]*empty*.

Lemma *empty_is_empty_2* : *s*[=]*empty* → *Empty s*.

properties of *add*

Lemma *add_equal* : *In x s* → *add x s* [=] *s*.

Lemma *add_add* : *add x* (*add x' s*) [=] *add x'* (*add x s*).

properties of *remove*

Lemma *remove_equal* : ¬ *In x s* → *remove x s* [=] *s*.

Lemma *Equal_remove* : *s*[=]*s'* → *remove x s* [=] *remove x s'*.

properties of *add* and *remove*

Lemma *add_remove* : *In x s* → *add x* (*remove x s*) [=] *s*.

Lemma *remove_add* : ¬*In x s* → *remove x* (*add x s*) [=] *s*.

properties of *singleton*

Lemma *singleton_equal_add* : *singleton x* [=] *add x empty*.

properties of *union*

Lemma *union_sym* : *union s s'* [=] *union s' s*.

Lemma *union_subset_equal* : *s*[<=]*s'* → *union s s'* [=] *s'*.

Lemma *union_equal_1* : *s*[=]*s'* → *union s s"* [=] *union s' s"*.

Lemma *union_equal_2* : *s'*[=]*s"* → *union s s'* [=] *union s s"*.

Lemma *union_assoc* : *union* (*union s s'*) *s"* [=] *union s* (*union s' s"*).

Lemma *add_union_singleton* : *add x s* [=] *union* (*singleton x*) *s*.

Lemma *union_add* : *union* (*add x s*) *s'* [=] *add x* (*union s s'*).

Lemma *union_subset_1* : *s* [<=] *union s s'*.

Lemma *union_subset_2* : *s'* [<=] *union s s'*.

Lemma *union_subset_3* : *s*[<=]*s"* → *s'*[<=]*s"* → *union s s'* [<=] *s"*.

Lemma *union_subset_4* : *s*[<=]*s'* → *union s s"* [<=] *union s' s"*.

Lemma *union_subset_5* : *s*[<=]*s'* → *union s" s* [<=] *union s" s'*.

Lemma *empty_union_1* : *Empty s* → *union s s'* [=] *s'*.

Lemma *empty_union_2* : *Empty s* → *union s' s* [=] *s'*.

Lemma *not_in_union* : ¬*In x s* → ¬*In x s'* → ¬*In x* (*union s s'*).

properties of *inter*

Lemma *inter_sym* : *inter s s'* [=] *inter s' s*.

Lemma *inter_subset_equal* : *s*[<=]*s'* → *inter s s'* [=] *s*.

Lemma *inter_equal_1* : *s*[=]*s'* → *inter s s"* [=] *inter s' s"*.

Lemma *inter_equal_2* : *s'*[=]*s"* → *inter s s'* [=] *inter s s"*.

Lemma *inter_assoc* : *inter* (*inter s s'*) *s"* [=] *inter s* (*inter s' s"*).

Lemma *union_inter_1* : *inter* (*union s s'*) *s"* [=] *union* (*inter s s"*) (*inter s' s"*).

Lemma *union_inter_2* : *union* (*inter s s'*) *s"* [=] *inter* (*union s s"*) (*union s' s"*).

Lemma *inter_add_1* : *In x s'* → *inter* (*add x s*) *s'* [=] *add x* (*inter s s'*).

Lemma *inter_add_2* : ¬ *In x s'* → *inter* (*add x s*) *s'* [=] *inter s s'*.

Lemma *empty_inter_1* : *Empty s* → *Empty* (*inter s s'*).

Lemma *empty_inter_2* : *Empty s'* → *Empty* (*inter s s'*).

Lemma *inter_subset_1* : *inter s s'* [<=] *s*.

Lemma *inter_subset_2* : *inter s s'* [<=] *s'*.

Lemma *inter_subset_3* :
s"[<=]*s* → *s"*[<=]*s'* → *s"*[<=] *inter s s'*.

properties of *diff*

Lemma *empty_diff_1* : *Empty s* → *Empty* (*diff s s'*).

Lemma *empty_diff_2* : *Empty s* → *diff s' s* [=] *s'*.

Lemma *diff_subset* : *diff s s'* [<=] *s*.

Lemma *diff_subset_equal* : *s*[<=]*s'* → *diff s s'* [=] *empty*.

Lemma *remove_diff_singleton* :
remove x s [=] *diff s* (*singleton x*).

Lemma *diff_inter_empty* : *inter* (*diff s s'*) (*inter s s'*) [=] *empty*.

Lemma *diff_inter_all* : *union* (*diff s s'*) (*inter s s'*) [=] *s*.

properties of *Add*

Lemma *Add_add* : *Add x s* (*add x s*).

Lemma *Add_remove* : *In x s* → *Add x* (*remove x s*) *s*.

Lemma *union_Add* : *Add x s s'* → *Add x* (*union s s"*) (*union s' s"*).

Lemma *inter_Add* :
In x s" → *Add x s s'* → *Add x* (*inter s s"*) (*inter s' s"*).

Lemma *union_Equal* :
In x s" → *Add x s s'* → *union s s"* [=] *union s' s"*.

Lemma *inter_Add_2* :
¬*In x s"* → *Add x s s'* → *inter s s"* [=] *inter s' s"*.

End *BasicProperties*.

Hint Immediate *equal_sym*: *set*.
Hint *Resolve equal_refl equal_trans* : *set*.

Hint Immediate *add_remove remove_add union_sym inter_sym*: *set*.
Hint *Resolve subset_refl subset_equal subset_antisym*
subset_trans subset_empty subset_remove_3 subset_diff subset_add_3
subset_add_2 in_subset empty_is_empty_1 empty_is_empty_2 add_equal
remove_equal singleton_equal_add union_subset_equal union_equal_1
union_equal_2 union_assoc add_union_singleton union_add union_subset_1
union_subset_2 union_subset_3 inter_subset_equal inter_equal_1 inter_equal_2
inter_assoc union_inter_1 union_inter_2 inter_add_1 inter_add_2
empty_inter_1 empty_inter_2 empty_union_1 empty_union_2 empty_diff_1
empty_diff_2 union_Add inter_Add union_Equal inter_Add_2 not_in_union
inter_subset_1 inter_subset_2 inter_subset_3 diff_subset diff_subset_equal
remove_diff_singleton diff_inter_empty diff_inter_all Add_add Add_remove
Equal_remove add_add : *set*.

228.2 Alternative (weaker) specifications for *fold*

Section *Old_Spec_Now_Properties*.

Notation *NoDup* := (*NoDupA E.eq*).

When *FSets* was first designed, the order in which Ocaml's *Set.fold* takes the set elements was unspecified. This specification reflects this fact:

Lemma *fold_0* :
∀ *s* (*A* : Set) (*i* : *A*) (*f* : *elt* → *A* → *A*),
∃ *l* : *list elt*,
NoDup l ∧
(∀ *x* : *elt*, *In x s* ↔ *InA E.eq x l*) ∧
fold f s i = *fold_right f i l*.

An alternate (and previous) specification for *fold* was based on the recursive structure of a set. It is now lemmas *fold_1* and *fold_2*.

Lemma *fold_1* :
∀ *s* (*A* : Set) (*eqA* : *A* → *A* → Prop)
(*st* : *Setoid_Theory A eqA*) (*i* : *A*) (*f* : *elt* → *A* → *A*),
Empty s → *eqA* (*fold f s i*) *i*.

Lemma *fold_2* :
∀ *s s' x* (*A* : Set) (*eqA* : *A* → *A* → Prop)
(*st* : *Setoid_Theory A eqA*) (*i* : *A*) (*f* : *elt* → *A* → *A*),
compat_op E.eq eqA f →
transpose eqA f →
¬ *In x s* → *Add x s s'* → *eqA* (*fold f s' i*) (*f x* (*fold f s i*)).

Similar specifications for *cardinal*.

Lemma *cardinal_fold* : ∀ *s*, *cardinal s* = *fold* (fun _ ⇒ *S*) *s* 0.

Lemma *cardinal_0* :
∀ *s*, ∃ *l* : *list elt*,
NoDupA E.eq l ∧
(∀ *x* : *elt*, *In x s* ↔ *InA E.eq x l*) ∧
cardinal s = *length l*.

Lemma *cardinal_1* : ∀ *s*, *Empty s* → *cardinal s* = 0.

Lemma *cardinal_2* :
∀ *s s' x*, ¬ *In x s* → *Add x s s'* → *cardinal s'* = *S* (*cardinal s*).

End *Old_Spec_Now_Properties*.

228.3 Induction principle over sets

Lemma *cardinal_inv_1* : ∀ *s*, *cardinal s* = 0 → *Empty s*.
Hint *Resolve cardinal_inv_1*.

Lemma *cardinal_inv_2* :
∀ *s n*, *cardinal s* = *S n* → { *x* : *elt* | *In x s* }.

Lemma *Equal_cardinal_aux* :

$\forall\ n\ s\ s'$, $cardinal\ s = n \rightarrow s[=]s' \rightarrow cardinal\ s = cardinal\ s'$.

Lemma $Equal\_cardinal : \forall\ s\ s', s[=]s' \rightarrow cardinal\ s = cardinal\ s'$.

$Add\ Morphism\ cardinal : cardinal\_m$.

Hint $Resolve\ Add\_add\ Add\_remove\ Equal\_remove\ cardinal\_inv\_1\ Equal\_cardinal$.

Lemma $cardinal\_induction$:
$\forall\ P : t \rightarrow \mathsf{Type}$,
$(\forall\ s, Empty\ s \rightarrow P\ s) \rightarrow$
$(\forall\ s\ s', P\ s \rightarrow \forall\ x, \neg In\ x\ s \rightarrow Add\ x\ s\ s' \rightarrow P\ s') \rightarrow$
$\forall\ n\ s, cardinal\ s = n \rightarrow P\ s$.

Lemma $set\_induction$:
$\forall\ P : t \rightarrow \mathsf{Type}$,
$(\forall\ s : t, Empty\ s \rightarrow P\ s) \rightarrow$
$(\forall\ s\ s' : t, P\ s \rightarrow \forall\ x : elt, \neg In\ x\ s \rightarrow Add\ x\ s\ s' \rightarrow P\ s') \rightarrow$
$\forall\ s : t, P\ s$.

Other properties of *fold*.

Section $Fold$.
Variables $(A{:}\mathsf{Set})(eqA{:}A{\rightarrow}A{\rightarrow}\mathsf{Prop})(st{:}Setoid\_Theory\ \_\ eqA)$.
Variables $(f{:}elt{\rightarrow}A{\rightarrow}A)(Comp{:}compat\_op\ E.eq\ eqA\ f)(Ass{:}transpose\ eqA\ f)$.

Section $Fold\_1$.
Variable $i\ i'{:}A$.

Lemma $fold\_empty : eqA\ (fold\ f\ empty\ i)\ i$.

Lemma $fold\_equal$:
$\forall\ s\ s', s[=]s' \rightarrow eqA\ (fold\ f\ s\ i)\ (fold\ f\ s'\ i)$.

Lemma $fold\_add : \forall\ s\ x, \neg In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (add\ x\ s)\ i)\ (f\ x\ (fold\ f\ s\ i))$.

Lemma $add\_fold : \forall\ s\ x, In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (add\ x\ s)\ i)\ (fold\ f\ s\ i)$.

Lemma $remove\_fold\_1 : \forall\ s\ x, In\ x\ s \rightarrow$
$eqA\ (f\ x\ (fold\ f\ (remove\ x\ s)\ i))\ (fold\ f\ s\ i)$.

Lemma $remove\_fold\_2 : \forall\ s\ x, \neg In\ x\ s \rightarrow$
$eqA\ (fold\ f\ (remove\ x\ s)\ i)\ (fold\ f\ s\ i)$.

Lemma $fold\_commutes : \forall\ s\ x$,
$eqA\ (fold\ f\ s\ (f\ x\ i))\ (f\ x\ (fold\ f\ s\ i))$.

Lemma $fold\_init : \forall\ s, eqA\ i\ i' \rightarrow$
$eqA\ (fold\ f\ s\ i)\ (fold\ f\ s\ i')$.

End $Fold\_1$.
Section $Fold\_2$.
Variable $i{:}A$.

Lemma *fold_union_inter* : ∀ *s s'*,
eqA (*fold f* (*union s s'*) (*fold f* (*inter s s'*) *i*))
(*fold f s* (*fold f s'* *i*)).

End *Fold_2*.
Section *Fold_3*.
Variable *i*:*A*.

Lemma *fold_diff_inter* : ∀ *s s'*,
eqA (*fold f* (*diff s s'*) (*fold f* (*inter s s'*) *i*)) (*fold f s i*).

Lemma *fold_union*: ∀ *s s'*, (∀ *x*, ¬*In x s*\/~*In x s'*) →
eqA (*fold f* (*union s s'*) *i*) (*fold f s* (*fold f s'* *i*)).

End *Fold_3*.
End *Fold*.

Lemma *fold_plus* :
∀ *s p*, *fold* (fun _ ⇒ *S*) *s p* = *fold* (fun _ ⇒ *S*) *s* 0 + *p*.

properties of *cardinal*

Lemma *empty_cardinal* : *cardinal empty* = 0.

Hint Immediate *empty_cardinal cardinal_1* : *set*.

Lemma *singleton_cardinal* : ∀ *x*, *cardinal* (*singleton x*) = 1.

Hint *Resolve singleton_cardinal*: *set*.

Lemma *diff_inter_cardinal* :
∀ *s s'*, *cardinal* (*diff s s'*) + *cardinal* (*inter s s'*) = *cardinal s* .

Lemma *union_cardinal*:
∀ *s s'*, (∀ *x*, ¬*In x s*\/~*In x s'*) →
cardinal (*union s s'*)=*cardinal s*+*cardinal s'*.

Lemma *subset_cardinal* :
∀ *s s'*, *s*[<=]*s'* → *cardinal s* ≤ *cardinal s'* .

Lemma *subset_cardinal_lt* :
∀ *s s' x*, *s*[<=]*s'* → *In x s'* → ¬*In x s* → *cardinal s* < *cardinal s'*.

Theorem *union_inter_cardinal* :
∀ *s s'*, *cardinal* (*union s s'*) + *cardinal* (*inter s s'*) = *cardinal s* + *cardinal s'* .

Lemma *union_cardinal_inter* :
∀ *s s'*, *cardinal* (*union s s'*) = *cardinal s* + *cardinal s'* - *cardinal* (*inter s s'*).

Lemma *union_cardinal_le* :
∀ *s s'*, *cardinal* (*union s s'*) ≤ *cardinal s* + *cardinal s'*.

Lemma *add_cardinal_1* :
∀ *s x*, *In x s* → *cardinal* (*add x s*) = *cardinal s*.

Lemma *add_cardinal_2* :

$\forall\ s\ x,\ \neg In\ x\ s \rightarrow cardinal\ (add\ x\ s) = S\ (cardinal\ s)$.

Lemma *remove_cardinal_1* :
$\forall\ s\ x,\ In\ x\ s \rightarrow S\ (cardinal\ (remove\ x\ s)) = cardinal\ s$.

Lemma *remove_cardinal_2* :
$\forall\ s\ x,\ \neg In\ x\ s \rightarrow cardinal\ (remove\ x\ s) = cardinal\ s$.

Hint *Resolve subset_cardinal union_cardinal add_cardinal_1 add_cardinal_2*.

End *Properties*.

Chapter 229

Module Coq.FSets.FSetWeak

Require Export *DecidableType*.
Require Export *DecidableTypeEx*.
Require Export *FSetWeakInterface*.
Require Export *FSetWeakFacts*.
Require Export *FSetWeakProperties*.
Require Export *FSetWeakList*.

Chapter 230

Module Coq.FSets.OrderedTypeAlt

Require Import *OrderedType*.

230.1 An alternative (but equivalent) presentation for an Ordered Type inferface.

NB: *comparison*, defined in *theories/Init/datatypes.v* is $Eq|Lt|Gt$ whereas *compare*, defined in *theories/FSets/OrderedType.v* is $EQ\ \_ \mid LT\ \_ \mid GT\ \_$

Module Type *OrderedTypeAlt*.

Parameter t : Set.

Parameter *compare* : $t \to t \to comparison$.

Infix "?=" := *compare* (*at level* 70, *no associativity*).

Parameter *compare_sym* :
$\forall\ x\ y, (y?{=}x) = CompOpp\ (x?{=}y)$.
Parameter *compare_trans* :
$\forall\ c\ x\ y\ z, (x?{=}y) = c \to (y?{=}z) = c \to (x?{=}z) = c$.

End *OrderedTypeAlt*.

From this new presentation to the original one.

Module *OrderedType_from_Alt* (*O*:*OrderedTypeAlt*) <: *OrderedType*.
Import *O*.

Definition $t := t$.

Definition $eq\ x\ y := (x?{=}y) = Eq$.
Definition $lt\ x\ y := (x?{=}y) = Lt$.

Lemma *eq_refl* : $\forall\ x, eq\ x\ x$.

Lemma *eq_sym* : $\forall\ x\ y, eq\ x\ y \to eq\ y\ x$.

Definition *eq_trans* := (*compare_trans Eq*).

```
 Definition lt_trans := (compare_trans Lt).

 Lemma lt_not_eq : ∀ x y, lt x y → ¬eq x y.

 Definition compare : ∀ x y, Compare lt eq x y.

End OrderedType_from_Alt.
```

From the original presentation to this alternative one.

```
Module OrderedType_to_Alt (O:OrderedType) <: OrderedTypeAlt.
 Import O.
 Module MO:=OrderedTypeFacts(O).
 Import MO.

 Definition t := t.

 Definition compare x y := match compare x y with
   | LT _ ⇒ Lt
   | EQ _ ⇒ Eq
   | GT _ ⇒ Gt
  end.

 Infix "?=" := compare (at level 70, no associativity).

 Lemma compare_sym :
   ∀ x y, (y?=x) = CompOpp (x?=y).

 Lemma compare_trans :
   ∀ c x y z, (x?=y) = c → (y?=z) = c → (x?=z) = c.

End OrderedType_to_Alt.
```

Chapter 231

Module Coq.FSets.OrderedTypeEx

```
Require Import OrderedType.
Require Import ZArith.
Require Import Omega.
Require Import NArith Ndec.
Require Import Compare_dec.
```

231.1 Examples of Ordered Type structures.

First, a particular case of *OrderedType* where the equality is the usual one of Coq.

```
Module Type UsualOrderedType.
  Parameter t : Set.
  Definition eq := @eq t.
  Parameter lt : t → t → Prop.
  Definition eq_refl := @refl_equal t.
  Definition eq_sym := @sym_eq t.
  Definition eq_trans := @trans_eq t.
  Axiom lt_trans : ∀ x y z : t, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀ x y : t, lt x y → ¬ eq x y.
  Parameter compare : ∀ x y : t, Compare lt eq x y.
End UsualOrderedType.
```

a *UsualOrderedType* is in particular an *OrderedType*.

```
Module UOT_to_OT (U:UsualOrderedType) <: OrderedType := U.
```

nat is an ordered type with respect to the usual order on natural numbers.

```
Module Nat_as_OT <: UsualOrderedType.

  Definition t := nat.

  Definition eq := @eq nat.
  Definition eq_refl := @refl_equal t.
  Definition eq_sym := @sym_eq t.
```

Definition *eq_trans* := @*trans_eq t*.

Definition *lt* := *lt*.

Lemma *lt_trans* : ∀ *x y z* : *t*, *lt x y* → *lt y z* → *lt x z*.

Lemma *lt_not_eq* : ∀ *x y* : *t*, *lt x y* → ¬ *eq x y*.

Definition *compare* : ∀ *x y* : *t*, *Compare lt eq x y*.

End *Nat_as_OT*.

Z is an ordered type with respect to the usual order on integers.

Open Local *Scope Z_scope*.

Module *Z_as_OT* <: *UsualOrderedType*.

Definition *t* := *Z*.
Definition *eq* := @*eq Z*.
Definition *eq_refl* := @*refl_equal t*.
Definition *eq_sym* := @*sym_eq t*.
Definition *eq_trans* := @*trans_eq t*.

Definition *lt* (*x y*:*Z*) := (*x*<*y*).

Lemma *lt_trans* : ∀ *x y z*, *x*<*y* → *y*<*z* → *x*<*z*.

Lemma *lt_not_eq* : ∀ *x y*, *x*<*y* → ¬ *x*=*y*.

Definition *compare* : ∀ *x y*, *Compare lt eq x y*.

End *Z_as_OT*.

positive is an ordered type with respect to the usual order on natural numbers.

Open Local *Scope positive_scope*.

Module *Positive_as_OT* <: *UsualOrderedType*.
Definition *t*:=*positive*.
Definition *eq*:=@*eq positive*.
Definition *eq_refl* := @*refl_equal t*.
Definition *eq_sym* := @*sym_eq t*.
Definition *eq_trans* := @*trans_eq t*.

Definition *lt p q*:= (*p* ?= *q*) *Eq* = *Lt*.

Lemma *lt_trans* : ∀ *x y z* : *t*, *lt x y* → *lt y z* → *lt x z*.

Lemma *lt_not_eq* : ∀ *x y* : *t*, *lt x y* → ¬ *eq x y*.

Definition *compare* : ∀ *x y* : *t*, *Compare lt eq x y*.

End *Positive_as_OT*.

N is an ordered type with respect to the usual order on natural numbers.

Open Local *Scope positive_scope*.

Module *N_as_OT* <: *UsualOrderedType*.

Definition t:=N.
Definition eq:=@eq N.
Definition eq_refl := @$refl\_equal$ t.
Definition eq_sym := @sym_eq t.
Definition eq_trans := @$trans\_eq$ t.

Definition lt p q:= Nle q p = $false$.

Definition lt_trans := Nlt_trans.

Lemma lt_not_eq : $\forall$ x y : t, lt x y $\rightarrow$ $\neg$ eq x y.

Definition $compare$: $\forall$ x y : t, $Compare$ lt eq x y.

End N_as_OT.

From two ordered types, we can build a new OrderedType over their cartesian product, using the lexicographic order.

Module $PairOrderedType(O1$ $O2$:$OrderedType)$ <: $OrderedType$.
Module $MO1$:=$OrderedTypeFacts(O1)$.
Module $MO2$:=$OrderedTypeFacts(O2)$.

Definition t := $prod$ $O1.t$ $O2.t$.

Definition eq x y := $O1.eq$ $(fst$ $x)$ $(fst$ $y)$ $\wedge$ $O2.eq$ $(snd$ $x)$ $(snd$ $y)$.

Definition lt x y :=
$O1.lt$ $(fst$ $x)$ $(fst$ $y)$ $\vee$
$(O1.eq$ $(fst$ $x)$ $(fst$ $y)$ $\wedge$ $O2.lt$ $(snd$ $x)$ $(snd$ $y))$.

Lemma eq_refl : $\forall$ x : t, eq x x.

Lemma eq_sym : $\forall$ x y : t, eq x y $\rightarrow$ eq y x.

Lemma eq_trans : $\forall$ x y z : t, eq x y $\rightarrow$ eq y z $\rightarrow$ eq x z.

Lemma lt_trans : $\forall$ x y z : t, lt x y $\rightarrow$ lt y z $\rightarrow$ lt x z.

Lemma lt_not_eq : $\forall$ x y : t, lt x y $\rightarrow$ $\neg$ eq x y.

Definition $compare$: $\forall$ x y : t, $Compare$ lt eq x y.

End $PairOrderedType$.

Chapter 232

Module Coq.FSets.OrderedType

Require Export *SetoidList*.

232.1 Ordered types

Inductive *Compare* (X : Set) (lt eq : $X \to X \to$ Prop) (x y : X) : Set :=
| LT : $lt\ x\ y \to Compare\ lt\ eq\ x\ y$
| EQ : $eq\ x\ y \to Compare\ lt\ eq\ x\ y$
| GT : $lt\ y\ x \to Compare\ lt\ eq\ x\ y$.

Module Type *OrderedType*.

Parameter t : Set.

Parameter eq : $t \to t \to$ Prop.
Parameter lt : $t \to t \to$ Prop.

Axiom eq_refl : $\forall\ x : t, eq\ x\ x$.
Axiom eq_sym : $\forall\ x\ y : t, eq\ x\ y \to eq\ y\ x$.
Axiom eq_trans : $\forall\ x\ y\ z : t, eq\ x\ y \to eq\ y\ z \to eq\ x\ z$.

Axiom lt_trans : $\forall\ x\ y\ z : t, lt\ x\ y \to lt\ y\ z \to lt\ x\ z$.
Axiom lt_not_eq : $\forall\ x\ y : t, lt\ x\ y \to \neg\ eq\ x\ y$.

Parameter $compare$: $\forall\ x\ y : t, Compare\ lt\ eq\ x\ y$.

Hint Immediate eq_sym.
Hint *Resolve* $eq\_refl\ eq\_trans\ lt\_not\_eq\ lt\_trans$.

End *OrderedType*.

232.2 Ordered types properties

Additional properties that can be derived from signature *OrderedType*.

Module *OrderedTypeFacts* (O: *OrderedType*).

```
  Import O.
  Lemma lt_antirefl : ∀ x, ¬ lt x x.
  Lemma lt_eq : ∀ x y z, lt x y → eq y z → lt x z.
  Lemma eq_lt : ∀ x y z, eq x y → lt y z → lt x z.
  Lemma le_eq : ∀ x y z, ¬lt x y → eq y z → ¬lt x z.
  Lemma eq_le : ∀ x y z, eq x y → ¬lt y z → ¬lt x z.
  Lemma neq_eq : ∀ x y z, ¬eq x y → eq y z → ¬eq x z.
  Lemma eq_neq : ∀ x y z, eq x y → ¬eq y z → ¬eq x z.
  Hint Immediate eq_lt lt_eq le_eq eq_le neq_eq eq_neq.
  Lemma le_lt_trans : ∀ x y z, ¬lt y x → lt y z → lt x z.
  Lemma lt_le_trans : ∀ x y z, lt x y → ¬lt z y → lt x z.
  Lemma le_neq : ∀ x y, ¬lt x y → ¬eq x y → lt y x.
  Lemma neq_sym : ∀ x y, ¬eq x y → ¬eq y x.
Ltac abstraction := match goal with
  | H : False ⊢ _ ⇒ elim H
 | H : lt ?x ?x ⊢ _ ⇒ elim (lt_antirefl H)
 | H : ¬eq ?x ?x ⊢ _ ⇒ elim (H (eq_refl x))
 | H : eq ?x ?x ⊢ _ ⇒ clear H; abstraction
 | H : ¬lt ?x ?x ⊢ _ ⇒ clear H; abstraction
 | ⊢ eq ?x ?x ⇒ exact (eq_refl x)
 | ⊢ lt ?x ?x ⇒ elimtype False; abstraction
 | ⊢ ¬ _ ⇒ intro; abstraction
 | H1: ¬lt ?x ?y, H2: ¬eq ?x ?y ⊢ _ ⇒
     generalize (le_neq H1 H2); clear H1 H2; intro; abstraction
 | H1: ¬lt ?x ?y, H2: ¬eq ?y ?x ⊢ _ ⇒
     generalize (le_neq H1 (neq_sym H2)); clear H1 H2; intro; abstraction
  | H : lt ?x ?y ⊢ _ ⇒ revert H; abstraction
 | H : ¬lt ?x ?y ⊢ _ ⇒ revert H; abstraction
 | H : ¬eq ?x ?y ⊢ _ ⇒ revert H; abstraction
 | H : eq ?x ?y ⊢ _ ⇒ revert H; abstraction
 | _ ⇒ idtac
   end.
Ltac do_eq a b EQ := match goal with
 | ⊢ lt ?x ?y → _ ⇒ let H := fresh "H" in
     (intro H;
       (generalize (eq_lt (eq_sym EQ) H); clear H; intro H) ||
       (generalize (lt_eq H EQ); clear H; intro H) ||
       idtac);
       do_eq a b EQ
 | ⊢ ¬lt ?x ?y → _ ⇒ let H := fresh "H" in
```

```
      (intro H;
       (generalize (eq_le (eq_sym EQ) H); clear H; intro H) ||
       (generalize (le_eq H EQ); clear H; intro H) ||
       idtac);
       do_eq a b EQ
 | ⊢ eq ?x ?y → _ ⇒ let H := fresh "H" in
      (intro H;
       (generalize (eq_trans (eq_sym EQ) H); clear H; intro H) ||
       (generalize (eq_trans H EQ); clear H; intro H) ||
       idtac);
       do_eq a b EQ
 | ⊢ ¬eq ?x ?y → _ ⇒ let H := fresh "H" in
      (intro H;
       (generalize (eq_neq (eq_sym EQ) H); clear H; intro H) ||
       (generalize (neq_eq H EQ); clear H; intro H) ||
       idtac);
       do_eq a b EQ
 | ⊢ lt a ?y ⇒ apply eq_lt with b; [exact EQ|]
 | ⊢ lt ?y a ⇒ apply lt_eq with b; [|exact (eq_sym EQ)]
 | ⊢ eq a ?y ⇒ apply eq_trans with b; [exact EQ|]
 | ⊢ eq ?y a ⇒ apply eq_trans with b; [|exact (eq_sym EQ)]
 | _ ⇒ idtac
 end.

Ltac propagate_eq := abstraction; clear; match goal with
   | ⊢ eq ?a ?b → _ ⇒
       let EQ := fresh "EQ" in (intro EQ; do_eq a b EQ; clear EQ);
       propagate_eq
 | _ ⇒ idtac
    end.

Ltac do_lt x y LT := match goal with
   | ⊢ lt x y → _ ⇒ intros _; do_lt x y LT
 | ⊢ lt y ?z → _ ⇒ let H := fresh "H" in
       (intro H; generalize (lt_trans LT H); intro); do_lt x y LT
 | ⊢ lt ?z x → _ ⇒ let H := fresh "H" in
       (intro H; generalize (lt_trans H LT); intro); do_lt x y LT
 | ⊢ lt _ _ → _ ⇒ intro; do_lt x y LT
   | ⊢ ¬lt y x → _ ⇒ intros _; do_lt x y LT
 | ⊢ ¬lt x ?z → _ ⇒ let H := fresh "H" in
       (intro H; generalize (le_lt_trans H LT); intro); do_lt x y LT
 | ⊢ ¬lt ?z y → _ ⇒ let H := fresh "H" in
       (intro H; generalize (lt_le_trans LT H); intro); do_lt x y LT
 | ⊢ ¬lt _ _ → _ ⇒ intro; do_lt x y LT
 | _ ⇒ idtac
 end.
```

```
Definition hide_lt := lt.

Ltac propagate_lt := abstraction; match goal with
  | ⊢ lt ?x ?y → _ ⇒
      let LT := fresh "LT" in (intro LT; do_lt x y LT;
      change (hide_lt x y) in LT);
      propagate_lt
  | _ ⇒ unfold hide_lt in ×
  end.

Ltac order :=
 intros;
 propagate_eq;
 propagate_lt;
 auto;
 propagate_lt;
 eauto.

Ltac false_order := elimtype False; order.

  Lemma gt_not_eq : ∀ x y, lt y x → ¬ eq x y.

  Lemma eq_not_lt : ∀ x y : t, eq x y → ¬ lt x y.

  Hint Resolve gt_not_eq eq_not_lt.

  Lemma eq_not_gt : ∀ x y : t, eq x y → ¬ lt y x.

  Lemma lt_not_gt : ∀ x y : t, lt x y → ¬ lt y x.

  Hint Resolve eq_not_gt lt_antirefl lt_not_gt.

  Lemma elim_compare_eq :
   ∀ x y : t,
   eq x y → ∃ H : eq x y, compare x y = EQ _ H.

  Lemma elim_compare_lt :
   ∀ x y : t,
   lt x y → ∃ H : lt x y, compare x y = LT _ H.

  Lemma elim_compare_gt :
   ∀ x y : t,
   lt y x → ∃ H : lt y x, compare x y = GT _ H.

  Ltac elim_comp :=
    match goal with
      | ⊢ ?e ⇒ match e with
            | context ctx [ compare ?a ?b ] ⇒
                  let H := fresh in
                  (destruct (compare a b) as [H|H|H];
                   try solve [ intros; false_order])
          end
    end.
```

Ltac *elim_comp_eq x y* :=
elim (*elim_compare_eq* (*x*:=*x*) (*y*:=*y*));
[intros _1 _2; rewrite _2; clear _1 _2 | auto].

Ltac *elim_comp_lt x y* :=
elim (*elim_compare_lt* (*x*:=*x*) (*y*:=*y*));
[intros _1 _2; rewrite _2; clear _1 _2 | auto].

Ltac *elim_comp_gt x y* :=
elim (*elim_compare_gt* (*x*:=*x*) (*y*:=*y*));
[intros _1 _2; rewrite _2; clear _1 _2 | auto].

Lemma *eq_dec* : ∀ *x y* : *t*, {*eq x y*} + {~ *eq x y*}.

Lemma *lt_dec* : ∀ *x y* : *t*, {*lt x y*} + {~ *lt x y*}.

Definition *eqb x y* : *bool* := if *eq_dec x y* then *true* else *false*.

Lemma *eqb_alt* :
∀ *x y*, *eqb x y* = match *compare x y* with *EQ* _ ⇒ *true* | _ ⇒ *false* end.

Section *ForNotations*.

Notation *In*:=(*InA eq*).
Notation *Inf*:=(*lelistA lt*).
Notation *Sort*:=(*sort lt*).
Notation *NoDup*:=(*NoDupA eq*).

Lemma *In_eq* : ∀ *l x y*, *eq x y* → *In x l* → *In y l*.

Lemma *ListIn_In* : ∀ *l x*, *List.In x l* → *In x l*.

Lemma *Inf_lt* : ∀ *l x y*, *lt x y* → *Inf y l* → *Inf x l*.

Lemma *Inf_eq* : ∀ *l x y*, *eq x y* → *Inf y l* → *Inf x l*.

Lemma *Sort_Inf_In* : ∀ *l x a*, *Sort l* → *Inf a l* → *In x l* → *lt a x*.

Lemma *ListIn_Inf* : ∀ *l x*, (∀ *y*, *List.In y l* → *lt x y*) → *Inf x l*.

Lemma *In_Inf* : ∀ *l x*, (∀ *y*, *In y l* → *lt x y*) → *Inf x l*.

Lemma *Inf_alt* :
∀ *l x*, *Sort l* → (*Inf x l* ↔ (∀ *y*, *In y l* → *lt x y*)).

Lemma *Sort_NoDup* : ∀ *l*, *Sort l* → *NoDup l*.

End *ForNotations*.

Hint *Resolve ListIn_In Sort_NoDup Inf_lt*.
Hint Immediate *In_eq Inf_lt*.

End *OrderedTypeFacts*.

Module *KeyOrderedType*(*O*:*OrderedType*).
Import *O*.
Module *MO*:=*OrderedTypeFacts*(*O*).
Import *MO*.

Section *Elt*.
Variable *elt* : Set.
Notation *key*:=*t*.

Definition *eqk* (*p p'*:*key*×*elt*) := *eq* (*fst p*) (*fst p'*).
Definition *eqke* (*p p'*:*key*×*elt*) :=
eq (*fst p*) (*fst p'*) ∧ (*snd p*) = (*snd p'*).
Definition *ltk* (*p p'*:*key*×*elt*) := *lt* (*fst p*) (*fst p'*).

Hint *Unfold eqk eqke ltk*.
Hint *Extern* 2 (*eqke* ?*a* ?*b*) ⇒ *split*.

Lemma *eqke_eqk* : ∀ *x x'*, *eqke x x'* → *eqk x x'*.

Lemma *ltk_right_r* : ∀ *x k e e'*, *ltk x* (*k*,*e*) → *ltk x* (*k*,*e'*).

Lemma *ltk_right_l* : ∀ *x k e e'*, *ltk* (*k*,*e*) *x* → *ltk* (*k*,*e'*) *x*.
Hint Immediate *ltk_right_r ltk_right_l*.

Lemma *eqk_refl* : ∀ *e*, *eqk e e*.

Lemma *eqke_refl* : ∀ *e*, *eqke e e*.

Lemma *eqk_sym* : ∀ *e e'*, *eqk e e'* → *eqk e' e*.

Lemma *eqke_sym* : ∀ *e e'*, *eqke e e'* → *eqke e' e*.

Lemma *eqk_trans* : ∀ *e e' e"*, *eqk e e'* → *eqk e' e"* → *eqk e e"*.

Lemma *eqke_trans* : ∀ *e e' e"*, *eqke e e'* → *eqke e' e"* → *eqke e e"*.

Lemma *ltk_trans* : ∀ *e e' e"*, *ltk e e'* → *ltk e' e"* → *ltk e e"*.

Lemma *ltk_not_eqk* : ∀ *e e'*, *ltk e e'* → ¬ *eqk e e'*.

Lemma *ltk_not_eqke* : ∀ *e e'*, *ltk e e'* → ¬*eqke e e'*.

Hint *Resolve eqk_trans eqke_trans eqk_refl eqke_refl*.
Hint *Resolve ltk_trans ltk_not_eqk ltk_not_eqke*.
Hint Immediate *eqk_sym eqke_sym*.

Lemma *eqk_not_ltk* : ∀ *x x'*, *eqk x x'* → ¬*ltk x x'*.

Lemma *ltk_eqk* : ∀ *e e' e"*, *ltk e e'* → *eqk e' e"* → *ltk e e"*.

Lemma *eqk_ltk* : ∀ *e e' e"*, *eqk e e'* → *ltk e' e"* → *ltk e e"*.
Hint *Resolve eqk_not_ltk*.
Hint Immediate *ltk_eqk eqk_ltk*.

Lemma *InA_eqke_eqk* :
∀ *x m*, *InA eqke x m* → *InA eqk x m*.
Hint *Resolve InA_eqke_eqk*.

Definition *MapsTo* (*k*:*key*)(*e*:*elt*):= *InA eqke* (*k*,*e*).
Definition *In k m* := ∃ *e*:*elt*, *MapsTo k e m*.
Notation *Sort* := (*sort ltk*).

Notation *Inf* := (*lelistA ltk*).

Hint *Unfold MapsTo In.*

Lemma *In_alt* : ∀ *k l*, *In k l* ↔ ∃ *e*, *InA eqk* (*k*,*e*) *l*.

Lemma *MapsTo_eq* : ∀ *l x y e*, *eq x y* → *MapsTo x e l* → *MapsTo y e l*.

Lemma *In_eq* : ∀ *l x y*, *eq x y* → *In x l* → *In y l*.

Lemma *Inf_eq* : ∀ *l x x'*, *eqk x x'* → *Inf x' l* → *Inf x l*.

Lemma *Inf_lt* : ∀ *l x x'*, *ltk x x'* → *Inf x' l* → *Inf x l*.

Hint Immediate *Inf_eq*.
Hint *Resolve Inf_lt*.

Lemma *Sort_Inf_In* :
∀ *l p q*, *Sort l* → *Inf q l* → *InA eqk p l* → *ltk q p*.

Lemma *Sort_Inf_NotIn* :
∀ *l k e*, *Sort l* → *Inf* (*k*,*e*) *l* → ¬*In k l*.

Lemma *Sort_NoDupA*: ∀ *l*, *Sort l* → *NoDupA eqk l*.

Lemma *Sort_In_cons_1* : ∀ *e l e'*, *Sort* (*e*::*l*) → *InA eqk e' l* → *ltk e e'*.

Lemma *Sort_In_cons_2* : ∀ *l e e'*, *Sort* (*e*::*l*) → *InA eqk e'* (*e*::*l*) →
ltk e e' ∨ *eqk e e'*.

Lemma *Sort_In_cons_3* :
∀ *x l k e*, *Sort* ((*k*,*e*)::*l*) → *In x l* → ¬*eq x k*.

Lemma *In_inv* : ∀ *k k' e l*, *In k* ((*k'*,*e*) :: *l*) → *eq k k'* ∨ *In k l*.

Lemma *In_inv_2* : ∀ *k k' e e' l*,
InA eqk (*k*, *e*) ((*k'*, *e'*) :: *l*) → ¬ *eq k k'* → *InA eqk* (*k*, *e*) *l*.

Lemma *In_inv_3* : ∀ *x x' l*,
InA eqke x (*x'* :: *l*) → ¬ *eqk x x'* → *InA eqke x l*.

End *Elt*.

Hint *Unfold eqk eqke ltk*.
Hint *Extern* 2 (*eqke* ?*a* ?*b*) ⇒ *split*.
Hint *Resolve eqk_trans eqke_trans eqk_refl eqke_refl*.
Hint *Resolve ltk_trans ltk_not_eqk ltk_not_eqke*.
Hint Immediate *eqk_sym eqke_sym*.
Hint *Resolve eqk_not_ltk*.
Hint Immediate *ltk_eqk eqk_ltk*.
Hint *Resolve InA_eqke_eqk*.
Hint *Unfold MapsTo In*.
Hint Immediate *Inf_eq*.
Hint *Resolve Inf_lt*.
Hint *Resolve Sort_Inf_NotIn*.
Hint *Resolve In_inv_2 In_inv_3*.

End *KeyOrderedType*.