Module FSetInterface

Set interfaces

Require Export *Bool*.
Require Export *PolyList*.
Require Export *Sorting*.
Require Export *Setoid*.
Set Implicit Arguments.

Misc properties used in specifications.

Section *Misc*.
Variable A,B : *Set*.
Variable *eqA* : $A \to A \to$ *Prop*.
Variable *eqB* : $B \to B \to$ *Prop*.

Compatibility of a boolean function with respect to an equality.
Definition *compat_bool* := $[f{:}A{\to}\mathit{bool}]\ (x{,}y{:}A)(\mathit{eqA}\ x\ y) \to (f\ x){=}(f\ y)$.

Compatibility of a predicate with respect to an equality.
Definition *compat_P* := $[P{:}A{\to}\mathit{Prop}](x{,}y{:}A)(\mathit{eqA}\ x\ y) \to (P\ x) \to (P\ y)$.

Being in a list modulo an equality relation.
Inductive *InList* $[x{:}A]$: $(\mathit{list}\ A) \to \mathit{Prop}$:=
  | *InList_cons_hd* : $(y{:}A)(l{:}(\mathit{list}\ A))(\mathit{eqA}\ x\ y) \to (\mathit{InList}\ x\ (\mathit{cons}\ y\ l))$
  | *InList_cons_tl* : $(y{:}A)(l{:}(\mathit{list}\ A))(\mathit{InList}\ x\ l) \to (\mathit{InList}\ x\ (\mathit{cons}\ y\ l))$.

A list without redondancy.
Inductive *Unique* : $(\mathit{list}\ A) \to \mathit{Prop}$:=
  | *Unique_nil* : $(\mathit{Unique}\ (\mathit{nil}\ A))$
  | *Unique_cons* : $(x{:}A)(l{:}(\mathit{list}\ A))\ \neg(\mathit{InList}\ x\ l) \to (\mathit{Unique}\ l) \to (\mathit{Unique}\ (\mathit{cons}\ x\ l))$.

End *Misc*.

Hint *InList* := *Constructors InList*.
Hint *InList* := *Constructors Unique*.
Hint *sort* := *Constructors sort*.
Hint *lelistA* := *Constructors lelistA*.
Hints *Unfold compat_bool compat_P*.

1 Ordered types

Inductive *Compare* $[X{:}\mathit{Set};\ \mathit{lt}{,}\mathit{eq}{:}X{\to}X{\to}\mathit{Prop};\ x{,}y{:}X]$: *Set* :=
  | *Lt* : $(\mathit{lt}\ x\ y) \to (\mathit{Compare}\ \mathit{lt}\ \mathit{eq}\ x\ y)$
  | *Eq* : $(\mathit{eq}\ x\ y) \to (\mathit{Compare}\ \mathit{lt}\ \mathit{eq}\ x\ y)$
  | *Gt* : $(\mathit{lt}\ y\ x) \to (\mathit{Compare}\ \mathit{lt}\ \mathit{eq}\ x\ y)$.

Module Type *OrderedType*.

Parameter t : Set.

Parameter eq : $t \rightarrow t \rightarrow Prop$.
Parameter lt : $t \rightarrow t \rightarrow Prop$.

Axiom eq_refl : $(x{:}t)$ $(eq\ x\ x)$.
Axiom eq_sym : $(x,y{:}t)$ $(eq\ x\ y) \rightarrow (eq\ y\ x)$.
Axiom eq_trans : $(x,y,z{:}t)$ $(eq\ x\ y) \rightarrow (eq\ y\ z) \rightarrow (eq\ x\ z)$.

Axiom lt_trans : $(x,y,z{:}t)$ $(lt\ x\ y) \rightarrow (lt\ y\ z) \rightarrow (lt\ x\ z)$.
Axiom lt_not_eq : $(x,y{:}t)$ $(lt\ x\ y) \rightarrow \neg(eq\ x\ y)$.

Parameter $compare$: $(x,y{:}t)(Compare\ lt\ eq\ x\ y)$.

Hints Immediate eq_sym.
Hints $Resolve$ eq_refl eq_trans lt_not_eq lt_trans.

End $OrderedType$.

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

Parameter $empty$: t.
The empty set.

Parameter is_empty: $t \rightarrow bool$.
Test whether a set is empty or not.

Parameter mem: $elt \rightarrow t \rightarrow bool$.
$mem\ x\ s$ tests whether x belongs to the set s.

Parameter add: $elt \rightarrow t \rightarrow t$.
$add\ x\ s$ returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged.

Parameter $singleton$: $elt \rightarrow t$.
$singleton\ x$ returns the one-element set containing only x.

Parameter $remove$: $elt \rightarrow t \rightarrow t$.
$remove\ x\ s$ returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged.

Parameter $union$: $t \rightarrow t \rightarrow t$.

Set union.

Parameter *inter*: $t \to t \to t$.
Set intersection.

Parameter *diff*: $t \to t \to t$.
Set difference.

Parameter *eq* : $t \to t \to Prop$.
Parameter *lt* : $t \to t \to Prop$.
Parameter *compare*: (*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*: (*A*:*Set*)($elt \to A \to A$) $\to t \to A \to A$.
fold f s a computes (*f xN* ... (*f x2* (*f x1 a*))...), 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. 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
Coq comment: We do not necessary choose equal elements from equal sets.

Section *Spec.*

Variable *s,s',s"* : *t.*
Variable *x,y,z* : *elt.*

Parameter *In* : $elt \to t \to Prop$.
Definition *Equal* := $[s,s'](a{:}elt)(In\ a\ s) \leftrightarrow (In\ a\ s')$.
Definition *Subset* := $[s,s'](a{:}elt)(In\ a\ s) \to (In\ a\ s')$.
Definition Add := $[x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x) \vee (In\ y\ s))$.
Definition *Empty* := $[s](a{:}elt)\neg(In\ a\ s)$.
Definition *For_all* := $[P{:}elt \to Prop;\ s{:}t](x{:}elt)(In\ x\ s) \to (P\ x)$.
Definition *Exists* := $[P{:}elt \to Prop;\ s{:}t](EX\ x{:}elt \mid (In\ x\ s) \wedge (P\ x))$.

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*: $(Equal\ s\ s') \to (equal\ s\ s')=true$.
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: $(Empty\ s) \rightarrow (is\_empty\ s){=}true$.
Parameter is_empty_2: $(is\_empty\ s){=}true \rightarrow (Empty\ s)$.

Specification of *add*
Parameter add_1: $(E.eq\ y\ x) \rightarrow (In\ y\ (add\ x\ s))$.
Parameter add_2: $(In\ y\ s) \rightarrow (In\ y\ (add\ x\ s))$.
Parameter add_3: $\neg(E.eq\ x\ y) \rightarrow (In\ y\ (add\ x\ s)) \rightarrow (In\ y\ s)$.

Specification of *remove*
Parameter $remove\_1$: $(E.eq\ y\ x) \rightarrow \neg(In\ y\ (remove\ x\ s))$.
Parameter $remove\_2$: $\neg(E.eq\ x\ y) \rightarrow (In\ y\ s) \rightarrow (In\ y\ (remove\ x\ s))$.
Parameter $remove\_3$: $(In\ y\ (remove\ x\ s)) \rightarrow (In\ y\ s)$.

Specification of *singleton*
Parameter $singleton\_1$: $(In\ y\ (singleton\ x)) \rightarrow (E.eq\ x\ y)$.
Parameter $singleton\_2$: $(E.eq\ x\ y) \rightarrow (In\ y\ (singleton\ x))$.

Specification of *union*
Parameter $union\_1$: $(In\ x\ (union\ s\ s')) \rightarrow (In\ x\ s){\vee}(In\ x\ s')$.
Parameter $union\_2$: $(In\ x\ s) \rightarrow (In\ x\ (union\ s\ s'))$.
Parameter $union\_3$: $(In\ x\ s') \rightarrow (In\ x\ (union\ s\ s'))$.

Specification of *inter*
Parameter $inter\_1$: $(In\ x\ (inter\ s\ s')) \rightarrow (In\ x\ s)$.
Parameter $inter\_2$: $(In\ x\ (inter\ s\ s')) \rightarrow (In\ x\ s')$.
Parameter $inter\_3$: $(In\ x\ s) \rightarrow (In\ x\ s') \rightarrow (In\ x\ (inter\ s\ s'))$.

Specification of *diff*
Parameter $diff\_1$: $(In\ x\ (diff\ s\ s')) \rightarrow (In\ x\ s)$.
Parameter $diff\_2$: $(In\ x\ (diff\ s\ s')) \rightarrow \neg(In\ x\ s')$.
Parameter $diff\_3$: $(In\ x\ s) \rightarrow \neg(In\ x\ s') \rightarrow (In\ x\ (diff\ s\ s'))$.

Specification of *fold*
Parameter $fold\_1$:
$(A{:}Set)(i{:}A)(f{:}elt{\rightarrow}A{\rightarrow}A)$
$(EX\ l{:}(list\ elt)\ |$
$(Unique\ E.eq\ l) \wedge$
$((x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)) \wedge$
$(fold\ f\ s\ i){=}(fold\_right\ f\ i\ l))$.

Specification of *cardinal*
Parameter $cardinal\_1$:
$(EX\ l{:}(list\ elt)\ |$
$(Unique\ E.eq\ l) \wedge$
$((x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)) \wedge$
$(cardinal\ s){=}(length\ l))$.

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*:
$(compat\_bool\ E.eq\ f) \to (For\_all\ [x](f\ x){=}true\ s) \to (for\_all\ f\ s){=}true$.
Parameter *for_all_2*:
$(compat\_bool\ E.eq\ f) \to (for\_all\ f\ s){=}true \to (For\_all\ [x](f\ x){=}true\ s)$.

Specification of *exists*

Parameter *exists_1*:
$(compat\_bool\ E.eq\ f) \to (Exists\ [x](f\ x){=}true\ s) \to (exists\ f\ s){=}true$.
Parameter *exists_2*:
$(compat\_bool\ E.eq\ f) \to (exists\ f\ s){=}true \to (Exists\ [x](f\ x){=}true\ s)$.

Specification of *partition*

Parameter *partition_1*:
$(compat\_bool\ E.eq\ f) \to (Equal\ (fst\ ?\ ?\ (partition\ f\ s))\ (filter\ f\ s))$.
Parameter *partition_2*:
$(compat\_bool\ E.eq\ f) \to$
$(Equal\ (snd\ ?\ ?\ (partition\ f\ s))\ (filter\ [x](negb\ (f\ x))\ s))$.

Specification of *elements*

Parameter *elements_1*: $(In\ x\ s) \to (InList\ E.eq\ x\ (elements\ s))$.
Parameter *elements_2*: $(InList\ E.eq\ x\ (elements\ s)) \to (In\ x\ s)$.
Parameter *elements_3*: $(sort\ E.lt\ (elements\ s))$.

Specification of *min_elt*

Parameter *min_elt_1*: $(min\_elt\ s) = (Some\ ?\ x) \to (In\ x\ s)$.
Parameter *min_elt_2*: $(min\_elt\ s) = (Some\ ?\ x) \to (In\ y\ s) \to \neg(E.lt\ y\ x)$.
Parameter *min_elt_3*: $(min\_elt\ s) = (None\ ?) \to (Empty\ s)$.

Specification of *max_elt*

Parameter *max_elt_1*: $(max\_elt\ s) = (Some\ ?\ x) \to (In\ x\ s)$.
Parameter *max_elt_2*: $(max\_elt\ s) = (Some\ ?\ x) \to (In\ y\ s) \to \neg(E.lt\ x\ y)$.
Parameter *max_elt_3*: $(max\_elt\ s) = (None\ ?) \to (Empty\ s)$.

Specification of *choose*

Parameter *choose_1*: $(choose\ s){=}(Some\ ?\ x) \to (In\ x\ s)$.
Parameter *choose_2*: $(choose\ s){=}(None\ ?) \to (Empty\ s)$.
End *Filter*.
End *Spec*.

Notation "âĹĚ" := *empty*.
Notation "a âŃČ b" := $(union\ a\ b)$ $(at\ level\ 1)$.

Notation "*a* âŃĆ *b*" := (*inter a b*) (*at level* 1).
Notation "âĹě *a* âĹě" := (*cardinal a*) (*at level* 1).
Notation "*a* âĹĹ *b*" := (*In a b*) (*at level* 1).
Notation "*a* âĹĽ *b*" := ¬(*In a b*) (*at level* 1).
Notation "*a* âĽą *b*" := (*Equal a b*) (*at level* 1).
Notation "*a* âĽć *b*" := ¬(*Equal a b*) (*at level* 1).
Notation "*a* âŁĘ *b*" := (*Subset a b*) (*at level* 1).
Notation "*a* âŁĹ *b*" := ¬(*Subset a b*) (*at level* 1).

Hints Immediate
In_1.

Hints *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
min_elt_1 min_elt_2 min_elt_3
max_elt_1 max_elt_2 max_elt_3.

End *S*.

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 $eq : t \to t \to Prop.$
Parameter $lt : t \to t \to Prop.$
Parameter $compare$: $(s,s':t)(Compare\ lt\ eq\ s\ s')$.

Parameter eq_refl: $(s:t)(eq\ s\ s)$.
Parameter eq_sym: $(s,s':t)(eq\ s\ s') \to (eq\ s'\ s)$.
Parameter eq_trans: $(s,s',s'':t)(eq\ s\ s') \to (eq\ s'\ s'') \to (eq\ s\ s'')$.
Parameter lt_trans : $(s,s',s'':t)(lt\ s\ s') \to (lt\ s'\ s'') \to (lt\ s\ s'')$.
Parameter lt_not_eq : $(s,s':t)(lt\ s\ s') \to \neg(eq\ s\ s')$.

Parameter $In : elt \to t \to Prop.$
Definition $Equal := [s,s'](a:elt)(In\ a\ s) \leftrightarrow (In\ a\ s')$.
Definition $Subset := [s,s'](a:elt)(In\ a\ s) \to (In\ a\ s')$.
Definition Add $:= [x:elt;s,s':t](y:elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x) \lor (In\ y\ s))$.
Definition $Empty := [s](a:elt)\neg(In\ a\ s)$.
Definition $For\_all := [P:elt \to Prop;\ s:t](x:elt)(In\ x\ s) \to (P\ x)$.
Definition $Exists := [P:elt \to Prop;\ s:t](EX\ x:elt \mid (In\ x\ s) \land (P\ x))$.

Parameter eq_In: $(s:t)(x,y:elt)(E.eq\ x\ y) \to (In\ x\ s) \to (In\ y\ s)$.

Parameter $empty$: $\{\ s:t \mid Empty\ s\ \}$.

Parameter is_empty : $(s:t)\{\ Empty\ s\ \}+\{\ \neg(Empty\ s)\ \}$.

Parameter mem : $(x:elt)\ (s:t)\ \{\ (In\ x\ s)\ \} + \{\ \neg(In\ x\ s)\ \}$.

Parameter add : $(x:elt)\ (s:t)\ \{\ s':t \mid (\mathsf{Add}\ x\ s\ s')\ \}$.

Parameter $singleton$: $(x:elt)\{\ s:t \mid (y:elt)(In\ y\ s) \leftrightarrow (E.eq\ x\ y)\}$.

Parameter $remove$: $(x:elt)(s:t)$
$\{\ s':t \mid (y:elt)(In\ y\ s') \leftrightarrow (\neg(E.eq\ y\ x) \land (In\ y\ s))\}$.

Parameter $union$: $(s,s':t)$
$\{\ s'':t \mid (x:elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s) \lor (In\ x\ s'))\}$.

Parameter $inter$: $(s,s':t)$
$\{\ s'':t \mid (x:elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s) \land (In\ x\ s'))\}$.

Parameter $diff$: $(s,s':t)$
$\{\ s'':t \mid (x:elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s)/\backslash\tilde{}(In\ x\ s'))\}$.

Parameter $equal$: $(s,s':t)\{\ Equal\ s\ s'\ \} + \{\ \neg(Equal\ s\ s')\ \}$.

Parameter $subset$: $(s,s':t)\ \{\ Subset\ s\ s'\ \} + \{\ \neg(Subset\ s\ s')\ \}$.

Parameter $fold$:
$(A:Set)(f:elt \to A \to A)(s:t)(i:A)$
$\{\ r : A \mid (EX\ l:(list\ elt) \mid$
$(Unique\ E.eq\ l) \land$
$((x:elt)(In\ x\ s) \leftrightarrow (InList\ E.eq\ x\ l)) \land$
$r = (fold\_right\ f\ i\ l))\ \}$.

Parameter *cardinal* :
  $(s{:}t)$ { r : nat | $(EX\ l{:}(list\ elt)$ |
  $(Unique\ E.eq\ l) \wedge$
  $((x{:}elt)(In\ x\ s) \leftrightarrow (InList\ E.eq\ x\ l)) \wedge$
  $r = (length\ l))$ }.

Parameter *filter* : $(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
  $\{\ s'{:}t \mid (compat\_P\ E.eq\ P) \rightarrow (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s) \wedge (P\ x))\ \}$.

Parameter *for_all* : $(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
  $\{\ (compat\_P\ E.eq\ P) \rightarrow (For\_all\ P\ s)\ \}\ +$
  $\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(For\_all\ P\ s)\ \}$.

Parameter *exists* : $(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
  $\{\ (compat\_P\ E.eq\ P) \rightarrow (Exists\ P\ s)\ \}\ +$
  $\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(Exists\ P\ s)\ \}$.

Parameter *partition* : $(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
  $\{\ partition : t \times t \mid$
    $let\ (s1,s2) = partition\ in$
    $(compat\_P\ E.eq\ P) \rightarrow$
    $((For\_all\ P\ s1) \wedge$
    $(For\_all\ ([x]\tilde{}(P\ x))\ s2) \wedge$
    $(x{:}elt)(In\ x\ s) \leftrightarrow ((In\ x\ s1) \vee (In\ x\ s2)))\ \}$.

Parameter *elements* :
  $(s{:}t)\{\ l{:}(list\ elt) \mid (sort\ E.lt\ l) \wedge (x{:}elt)(In\ x\ s) \leftrightarrow (InList\ E.eq\ x\ l)\}$.

Parameter *min_elt* :
  $(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ y\ x)\ s)\ \} + \{\ Empty\ s\ \}$.

Parameter *max_elt* :
  $(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ x\ y)\ s)\ \} + \{\ Empty\ s\ \}$.

Parameter *choose* : $(s{:}t)\ \{\ x{:}elt \mid (In\ x\ s)\} + \{\ Empty\ s\ \}$.

Notation "âĹĚ" := *empty*.
Notation "*a* âŃČ *b*" := (*union a b*) (*at level* 1).
Notation "*a* âŃĆ *b*" := (*inter a b*) (*at level* 1).
Notation "*a* âĹĹ *b*" := (*In a b*) (*at level* 1).
Notation "*a* âĹĽ *b*" := ¬(*In a b*) (*at level* 1).
Notation "*a* âĽą *b*" := (*Equal a b*) (*at level* 1).
Notation "*a* âĽć *b*" := ¬(*Equal a b*) (*at level* 1).
Notation "*a* âŁĘ *b*" := (*Subset a b*) (*at level* 1).
Notation "*a* âŁĹ *b*" := ¬(*Subset a b*) (*at level* 1).

End *Sdep*.

4 Ordered types properties

Additional properties that can be derived from signature *OrderedType*.

Module *MoreOrderedType* [*O*:*OrderedType*].

Lemma *gt_not_eq* : $(x,y{:}O.t)(O.lt\ y\ x) \rightarrow \neg(O.eq\ x\ y)$.

Lemma *eq_not_lt* : $(x,y{:}O.t)(O.eq\ x\ y) \rightarrow \neg(O.lt\ x\ y)$.

Hints *Resolve gt_not_eq eq_not_lt.*

Lemma *eq_not_gt* : $(x,y{:}O.t)(O.eq\ x\ y) \rightarrow \neg(O.lt\ y\ x)$.

Lemma *lt_antirefl* : $(x{:}O.t)\neg(O.lt\ x\ x)$.

Lemma *lt_not_gt* : $(x,y{:}O.t)(O.lt\ x\ y) \rightarrow \neg(O.lt\ y\ x)$.

Hints *Resolve eq_not_gt lt_antirefl lt_not_gt.*

Lemma *lt_eq* : $(x,y,z{:}O.t)(O.lt\ x\ y) \rightarrow (O.eq\ y\ z) \rightarrow (O.lt\ x\ z)$.

Lemma *eq_lt* : $(x,y,z{:}O.t)(O.eq\ x\ y) \rightarrow (O.lt\ y\ z) \rightarrow (O.lt\ x\ z)$.

Hints Immediate *eq_lt lt_eq.*

Lemma *elim_compare_eq*:
$(x,y{:}O.t)\ (O.eq\ x\ y) \rightarrow$
$(EX\ H : (O.eq\ x\ y)\ |\ (O.compare\ x\ y) = (Eq\ ?\ H))$.

Lemma *elim_compare_lt*:
$(x,y{:}O.t)\ (O.lt\ x\ y) \rightarrow$
$(EX\ H : (O.lt\ x\ y)\ |\ (O.compare\ x\ y) = (Lt\ ?\ H))$.

Lemma *elim_compare_gt*:
$(x,y{:}O.t)\ (O.lt\ y\ x) \rightarrow$
$(EX\ H : (O.lt\ y\ x)\ |\ (O.compare\ x\ y) = (Gt\ ?\ H))$.

Tactic Definition *Comp_eq x y* :=
Elim (!*elim_compare_eq x y*);
[*Intros _1 _2*; *Rewrite _2*; *Clear _1 _2* | *Auto*].

Tactic Definition *Comp_lt x y* :=
Elim (!*elim_compare_lt x y*);
[*Intros _1 _2*; *Rewrite _2*; *Clear _1 _2* | *Auto*].

Tactic Definition *Comp_gt x y* :=
Elim (!*elim_compare_gt x y*);
[*Intros _1 _2*; *Rewrite _2*; *Clear _1 _2* | *Auto*].

Lemma *eq_dec* : $(x,y{:}O.t)\{(O.eq\ x\ y)\}+\{\tilde{}(O.eq\ x\ y)\}$.

Lemma *lt_dec* : $(x,y{:}O.t)\{(O.lt\ x\ y)\}+\{\tilde{}(O.lt\ x\ y)\}$.

Results concerning lists modulo E.eq

Notation "'*Sort*' l" := (*sort O.lt l*) (*at level* 10, *l at level* 8).
Notation "'*Inf*' x l" := (*lelistA O.lt x l*) (*at level* 10, *x,l at level* 8).
Notation "'*In*' x l" := (*InList O.eq x l*) (*at level* 10, *x,l at level* 8).

Lemma *In_eq*: (*s*:(*list O.t*))(*x*,*y*:*O.t*)(*O.eq x y*) → (*In x s*) → (*In y s*).
Hints Immediate *In_eq*.

Lemma *Inf_lt* : (*s*:(*list O.t*))(*x*,*y*:*O.t*)(*O.lt x y*) → (*Inf y s*) → (*Inf x s*).

Lemma *Inf_eq* : (*s*:(*list O.t*))(*x*,*y*:*O.t*)(*O.eq x y*) → (*Inf y s*) → (*Inf x s*).
Hints *Resolve Inf_lt Inf_eq*.

Lemma *In_sort*: (*s*:(*list O.t*))(*x*,*a*:*O.t*)(*Inf a s*) → (*In x s*) → (*Sort s*) → (*O.lt a x*).

Lemma *Inf_In* : (*l*:(*list O.t*))(*x*:*O.t*)
((*y*:*O.t*)(*PolyList.In y l*) → (*O.lt x y*)) → (*Inf x l*).

Lemma *Inf_In_2* : (*l*:(*list O.t*))(*x*:*O.t*)
((*y*:*O.t*)(*In y l*) → (*O.lt x y*)) → (*Inf x l*).

Lemma *In_InList* : (*l*:(*list O.t*))(*x*:*O.t*)(*PolyList.In x l*) → (*In x l*).
Hints *Resolve In_InList*.

Lemma *Sort_Unique* : (*l*:(*list O.t*))(*Sort l*) → (*Unique O.eq l*).

End *MoreOrderedType*.

Module FSetBridge

This module implements bridges (as functors) from dependent to/from non-dependent set signature.

Require Export *FSetInterface*.
Set Implicit Arguments.
Set Ground Depth 2.

5 From non-dependent signature S to dependent signature $Sdep$.

Module *DepOfNodep* [*M*:*S*] <: *Sdep with Module E* := *M.E*.
Import *M*.

Module *ME* := *MoreOrderedType E*.

Definition $empty$: { s:t | $Empty\ s$ }.

Definition is_empty: (s:t){ $Empty\ s$ }+{ $\neg$($Empty\ s$) }.

Definition mem : (x:elt) (s:t) { ($In\ x\ s$) } + { $\neg$($In\ x\ s$) }.

Definition add : (x:elt) (s:t){ s':t | (Add $x\ s\ s'$) }.

Definition $singleton$: (x:elt){ s:t | (y:elt)($In\ y\ s$) $\leftrightarrow$ ($E.eq\ x\ y$)}.

Definition $remove$: (x:elt)(s:t)
{ s':t | (y:elt)($In\ y\ s'$) $\leftrightarrow$ ($\neg$($E.eq\ y\ x$) $\wedge$ ($In\ y\ s$))}.

Definition $union$: (s,s':t)
{ s'':t | (x:elt)($In\ x\ s''$) $\leftrightarrow$ (($In\ x\ s$)$\vee$($In\ x\ s'$))}.

Definition $inter$: (s,s':t)
{ s'':t | (x:elt)($In\ x\ s''$) $\leftrightarrow$ (($In\ x\ s$)$\wedge$($In\ x\ s'$))}.

Definition $diff$: (s,s':t)
{ s'':t | (x:elt)($In\ x\ s''$) $\leftrightarrow$ (($In\ x\ s$)/\~($In\ x\ s'$))}.

Definition $equal$: (s,s':t){ $Equal\ s\ s'$ } + { $\neg$($Equal\ s\ s'$) }.

Definition $subset$: (s,s':t) { $Subset\ s\ s'$ } + { $\neg$($Subset\ s\ s'$) }.

Definition $fold$:
(A:Set)(f:$elt$$\rightarrow$$A$$\rightarrow$$A$)($s$:$t$)($i$:$A$)
{ r : A | ($EX\ l$:($list\ elt$) |
($Unique\ E.eq\ l$) $\wedge$
((x:elt)($In\ x\ s$)$\leftrightarrow$($InList\ E.eq\ x\ l$)) $\wedge$
r = ($fold\_right\ f\ i\ l$)) }.

Definition $cardinal$:
(s:t) { r : nat | ($EX\ l$:($list\ elt$) |
($Unique\ E.eq\ l$) $\wedge$
((x:elt)($In\ x\ s$)$\leftrightarrow$($InList\ E.eq\ x\ l$)) $\wedge$
r = ($length\ l$)) }.

Definition $fdec$:=
[P:$elt$$\rightarrow$$Prop$; $Pdec$:(x:elt){$P\ x$}+{~($P\ x$)}; x:elt]
if ($Pdec\ x$) $then$ [_]$true$ $else$ [_]$false$.

Lemma $compat\_P\_aux$:
(P:$elt$$\rightarrow$$Prop$)($Pdec$:($x$:$elt$){$P\ x$}+{~($P\ x$)})($compat_P\ E.eq\ P$) $\rightarrow$
($compat\_bool\ E.eq$ ($fdec\ Pdec$)).

Hints $Resolve\ compat\_P\_aux$.

Definition $filter$: (P:$elt$$\rightarrow$$Prop$)($Pdec$:($x$:$elt$){$P\ x$}+{~($P\ x$)})($s$:$t$)
{ s':t | ($compat\_P\ E.eq\ P$) $\rightarrow$ (x:elt)($In\ x\ s'$) $\leftrightarrow$ (($In\ x\ s$)$\wedge$($P\ x$)) }.

Definition for_all : $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (For\_all\ P\ s)\ \} +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(For\_all\ P\ s)\ \}$.

Definition $exists$: $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (Exists\ P\ s)\ \} +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(Exists\ P\ s)\ \}$.

Definition $partition$: $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ partition : t{\times}t \mid$
$let\ (s1,s2) = partition\ in$
$(compat\_P\ E.eq\ P) \rightarrow$
$((For\_all\ P\ s1) \wedge$
$(For\_all\ ([x]\tilde{}(P\ x))\ s2) \wedge$
$(x{:}elt)(In\ x\ s){\leftrightarrow}((In\ x\ s1){\vee}(In\ x\ s2)))\ \}$.

Definition $choose$: $(s{:}t)\ \{x{:}elt \mid (In\ x\ s)\} + \{\ Empty\ s\ \}$.

Definition $elements$:
$(s{:}t)\{\ l{:}(list\ elt) \mid (sort\ E.lt\ l){\wedge}(x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)\}$.

Definition min_elt :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ y\ x)\ s)\ \} + \{\ Empty\ s\ \}$.

Definition max_elt :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ x\ y)\ s)\ \} + \{\ Empty\ s\ \}$.

Module $E{:=}E$.

Definition $elt := elt$.
Definition $t := t$.

Definition $In := In$.
Definition $Equal := [s,s'](a{:}elt)(In\ a\ s){\leftrightarrow}(In\ a\ s')$.
Definition $Subset := [s,s'](a{:}elt)(In\ a\ s){\rightarrow}(In\ a\ s')$.
Definition Add $:= [x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x){\vee}(In\ y\ s))$.
Definition $Empty := [s](a{:}elt)\neg(In\ a\ s)$.
Definition $For\_all := [P{:}elt{\rightarrow}Prop;\ s{:}t](x{:}elt)(In\ x\ s){\rightarrow}(P\ x)$.
Definition $Exists := [P{:}elt{\rightarrow}Prop;\ s{:}t](EX\ x{:}elt \mid (In\ x\ s){\wedge}(P\ x))$.

Definition $eq\_In := In\_1$.

Definition $eq := eq$.
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 $DepOfNodep$.

6 From dependent signature *Sdep* to non-dependent signature *S*.

```
Module NodepOfDep [M:Sdep] <: S with Module E := M.E.
  Import M.

  Module ME := MoreOrderedType E.

  Definition empty : t := let (s,_) = M.empty in s.

  Lemma empty_1 : (Empty empty).

  Definition is_empty : t → bool :=
    [s:t]if (M.is_empty s) then [_]true else [_]false.

  Lemma is_empty_1 : (s:t)(Empty s) → (is_empty s)=true.

  Lemma is_empty_2 : (s:t)(is_empty s)=true → (Empty s).

  Definition mem : elt → t → bool :=
    [x:elt][s:t]if (M.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 : t → t → bool :=
    [s,s']if (M.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 : t → t → bool :=
    [s,s']if (M.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 : t → (option elt) :=
    [s:t]Cases (M.choose s) of (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 : t → (list elt) := [s] let (l,_) = (M.elements s) in l.

  Lemma elements_1: (s:t)(x:elt)(In x s) → (InList E.eq x (elements s)).

  Lemma elements_2: (s:t)(x:elt)(InList E.eq x (elements s)) → (In x s).
```

Lemma *elements_3*: $(s{:}t)(sort\ E.lt\ (elements\ s))$.

Definition *min_elt* : $t \to (option\ elt) :=$
$[s{:}t]Cases\ (M.min\_elt\ s)\ of\ (inleft\ (exist\ x\ \_)) \Rightarrow (Some\ ?\ x)$
$\mid (inright\ \_) \Rightarrow (None\ ?)\ end.$

Lemma *min_elt_1*: $(s{:}t)(x{:}elt)(min\_elt\ s) = (Some\ ?\ x) \to (In\ x\ s)$.

Lemma *min_elt_2*: $(s{:}t)(x{,}y{:}elt)(min\_elt\ s) = (Some\ ?\ x) \to (In\ y\ s) \to \neg(E.lt\ y\ x)$.

Lemma *min_elt_3* : $(s{:}t)(min\_elt\ s) = (None\ ?) \to (Empty\ s)$.

Definition *max_elt* : $t \to (option\ elt) :=$
$[s{:}t]Cases\ (M.max\_elt\ s)\ of\ (inleft\ (exist\ x\ \_)) \Rightarrow (Some\ ?\ x)$
$\mid (inright\ \_) \Rightarrow (None\ ?)\ end.$

Lemma *max_elt_1*: $(s{:}t)(x{:}elt)(max\_elt\ s) = (Some\ ?\ x) \to (In\ x\ s)$.

Lemma *max_elt_2*: $(s{:}t)(x{,}y{:}elt)(max\_elt\ s) = (Some\ ?\ x) \to (In\ y\ s) \to \neg(E.lt\ x\ y)$.

Lemma *max_elt_3* : $(s{:}t)(max\_elt\ s) = (None\ ?) \to (Empty\ s)$.

Definition *add* : $elt \to t \to t :=$
$[x{:}elt][s{:}t]let\ (s',\_) = (M.add\ x\ s)\ in\ s'.$

Lemma *add_1* : $(s{:}t)(x{,}y{:}elt)(E.eq\ y\ x) \to (In\ y\ (add\ x\ s))$.

Lemma *add_2* : $(s{:}t)(x{,}y{:}elt)(In\ y\ s) \to (In\ y\ (add\ x\ s))$.

Lemma *add_3* : $(s{:}t)(x{,}y{:}elt)\neg(E.eq\ x\ y) \to (In\ y\ (add\ x\ s)) \to (In\ y\ s)$.

Definition *remove* : $elt \to t \to t :=$
$[x{:}elt][s{:}t]let\ (s',\_) = (M.remove\ x\ s)\ in\ s'.$

Lemma *remove_1* : $(s{:}t)(x{,}y{:}elt)(E.eq\ y\ x) \to \neg(In\ y\ (remove\ x\ s))$.

Lemma *remove_2* : $(s{:}t)(x{,}y{:}elt)$
$\neg(E.eq\ x\ y) \to (In\ y\ s) \to (In\ y\ (remove\ x\ s))$.

Lemma *remove_3* : $(s{:}t)(x{,}y{:}elt)(In\ y\ (remove\ x\ s)) \to (In\ y\ s)$.

Definition *singleton* : $elt \to t := [x]let\ (s,\_) = (M.singleton\ x)\ in\ s.$

Lemma *singleton_1* : $(x{,}y{:}elt)(In\ y\ (singleton\ x)) \to (E.eq\ x\ y)$.

Lemma *singleton_2*: $(x{,}y{:}elt)(E.eq\ x\ y) \to (In\ y\ (singleton\ x))$.

Definition *union* : $t \to t \to t :=$
$[s,s']let\ (s'',\_) = (M.union\ s\ s')\ in\ s''.$

Lemma *union_1*: $(s,s'{:}t)(x{:}elt)(In\ x\ (union\ s\ s')) \to (In\ x\ s)\vee(In\ x\ s')$.

Lemma *union_2*: $(s,s'{:}t)(x{:}elt)(In\ x\ s) \to (In\ x\ (union\ s\ s'))$.

Lemma *union_3*: $(s,s'{:}t)(x{:}elt)(In\ x\ s') \to (In\ x\ (union\ s\ s'))$.

Definition *inter* : $t \to t \to t :=$

[s,s']let (s",_) = (M.inter s s') in s".

Lemma inter_1: (s,s':t)(x:elt)(In x (inter s s')) → (In x s).

Lemma inter_2: (s,s':t)(x:elt)(In x (inter s s')) → (In x s').

Lemma inter_3: (s,s':t)(x:elt)(In x s) → (In x s') → (In x (inter s s')).

Definition diff : t → t → t :=
[s,s']let (s",_) = (M.diff s s') in s".

Lemma diff_1: (s,s':t)(x:elt)(In x (diff s s')) → (In x s).

Lemma diff_2: (s,s':t)(x:elt)(In x (diff s s')) → ¬(In x s').

Lemma diff_3: (s,s':t)(x:elt)(In x s) → ¬(In x s') → (In x (diff s s')).

Definition cardinal : t → nat := [s]let (f,_)= (M.cardinal s) in f.

Lemma cardinal_1 :
(s:t)(EX l:(list elt) |
(Unique E.eq l) ∧
((x:elt)(In x s)↔(InList E.eq x l)) ∧
(cardinal s)=(length l)).

Definition fold : (B:Set)(elt→B→B)→t→B→B :=
[B,f,i,s]let (fold,_)= (M.fold f i s) in fold.

Lemma fold_1:
(s:t)(A:Set)(i:A)(f:elt→A→A)
(EX l:(list elt) |
(Unique E.eq l) ∧
((x:elt)(In x s)↔(InList E.eq x l)) ∧
(fold f s i)=(fold_right f i l)).

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 [x](f x)=true).

Hints Resolve compat_P_aux.

Definition filter : (elt → bool) → t → t :=
[f,s]let (s',_) = (!M.filter [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:

$(s{:}t)(x{:}elt)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(In\ x\ s) \rightarrow (f\ x){=}true \rightarrow (In\ x\ (filter\ f\ s))$.

Definition for_all: $(elt \rightarrow bool) \rightarrow t \rightarrow bool :=$
$[f,s]if\ (!M.for\_all\ [x](f\ x){=}true\ (f\_dec\ f)\ s)\ then\ [\_]true\ else\ [\_]false$.

Lemma for_all_1:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(For\_all\ [x](f\ x){=}true\ s) \rightarrow (for\_all\ f\ s){=}true$.

Lemma for_all_2:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (for\_all\ f\ s){=}true \rightarrow (For\_all\ [x](f\ x){=}true\ s)$.

Definition $exists$: $(elt \rightarrow bool) \rightarrow t \rightarrow bool :=$
$[f,s]if\ (!M.exists\ [x](f\ x){=}true\ (f\_dec\ f)\ s)\ then\ [\_]true\ else\ [\_]false$.

Lemma $exists\_1$:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(Exists\ [x](f\ x){=}true\ s) \rightarrow (exists\ f\ s){=}true$.

Lemma $exists\_2$:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(exists\ f\ s){=}true \rightarrow (Exists\ [x](f\ x){=}true\ s)$.

Definition $partition$: $(elt \rightarrow bool) \rightarrow t \rightarrow t{\times}t :=$
$[f,s]let\ (p,\_) = (!M.partition\ [x](f\ x){=}true\ (f\_dec\ f)\ s)\ in\ p$.

Lemma $partition\_1$:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (Equal\ (fst\ ?\ ?\ (partition\ f\ s))\ (filter\ f\ s))$.

Lemma $partition\_2$:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(Equal\ (snd\ ?\ ?\ (partition\ f\ s))\ (filter\ [x](negb\ (f\ x))\ s))$.

Module $E{:=}E$.
Definition $elt := elt$.
Definition $t := t$.

Definition $In := In$.
Definition $Equal := [s,s'](a{:}elt)(In\ a\ s){\leftrightarrow}(In\ a\ s')$.
Definition $Subset := [s,s'](a{:}elt)(In\ a\ s){\rightarrow}(In\ a\ s')$.
Definition Add $:= [x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x){\vee}(In\ y\ s))$.
Definition $Empty := [s](a{:}elt)\neg(In\ a\ s)$.
Definition $For\_all := [P{:}elt{\rightarrow}Prop;\ s{:}t](x{:}elt)(In\ x\ s){\rightarrow}(P\ x)$.
Definition $Exists := [P{:}elt{\rightarrow}Prop;\ s{:}t](EX\ x{:}elt \mid (In\ x\ s){\wedge}(P\ x))$.

Definition $In\_1 := eq\_In$.

Definition *eq* := *eq*.
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*.

Module FSetProperties

This functor derives additional properties from *FSetInterface.S*. Contrary to the functor in *FSetProperties* 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 *Omega*.

Import *nat_scope*.
Open Scope nat_scope.

Require Export *FSetInterface*.
Require Export *Bool*.
Require Export *Sumbool*.
Require Export *Zerob*.
Set Implicit Arguments.

Section *Misc*.
Variable *A*,*B* : *Set*.
Variable *eqA* : *A* → *A* → *Prop*.
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'*).

End *Misc*.
Hints *Unfold transpose compat_op compat_nat*.

To prove (*Setoid_Theory* ? (*eq* ?))
Tactic Definition *ST* :=
Constructor; *Intros*;[*Trivial* | *Symmetry*; *Trivial* | *EApply trans_eq*; *EAuto*].

Hint *st* : *core* := *Extern* 1 (*Setoid_Theory* ? (*eq* ?)) *ST*.

Definition *gen_st* : (*A*:*Set*)(*Setoid_Theory* ? (*eq* *A*)).

Usual syntax for lists. CAVEAT: the Coq cast "::" will no longer be available.
Notation "[]" := (*nil* ?) (*at level* 1).
Notation "*a* :: *l*" := (*cons* *a* *l*) (*at level* 1, *l* *at next level*).

Module *Properties* [*M*:*S*].
Import *M*.
Import *Logic*.
Import *Peano*.

Module *ME* := *MoreOrderedType* *E*.

7 Alternative (weaker) specification for *fold*,

based on *empty* (*fold_1*) and *add* (*fold_2*).

Section *Old_Spec_Now_Properties*.

Results about lists without duplicates

Section *Unique_Remove*.

Fixpoint *remove_list* [*x*:*elt*;*s*:(*list* *elt*)] : (*list* *elt*) := *Cases* *s* *of*
nil ⇒ []
| (*cons* *y* *l*) ⇒ *if* (*ME.eq_dec* *x* *y*) *then* [_]*l* *else* [_]*y*::(*remove_list* *x* *l*)
end.

Lemma *remove_list_correct* :
(*s*:(*list* *elt*))(*x*:*elt*)(*Unique* *E.eq* *s*) →
(*Unique* *E.eq* (*remove_list* *x* *s*)) ∧
((*y*:*elt*)(*InList* *E.eq* *y* (*remove_list* *x* *s*))↔(*InList* *E.eq* *y* *s*)/\~(*E.eq* *x* *y*)).

Local *ListEq* := [*l*,*l'*](*y*:*elt*)(*InList* *E.eq* *y* *l*)↔(*InList* *E.eq* *y* *l'*).
Local *ListAdd* := [*x*,*l*,*l'*](*y*:*elt*)(*InList* *E.eq* *y* *l'*)↔(*E.eq* *y* *x*)∨(*InList* *E.eq* *y* *l*).

Lemma *remove_list_equal* :
(*s*,*s'*:(*list* *elt*))(*x*:*elt*)(*Unique* *E.eq* *x*::*s*) → (*Unique* *E.eq* *s'*) →
(*ListEq* *x*::*s* *s'*) → (*ListEq* *s* (*remove_list* *x* *s'*)).

Lemma *remove_list_add* :
(*s*,*s'*:(*list* *elt*))(*x*,*x'*:*elt*)(*Unique* *E.eq* *s*) → (*Unique* *E.eq* *x'*::*s'*) →
¬(*E.eq* *x* *x'*) → ¬(*InList* *E.eq* *x* *s*) →
(*ListAdd* *x* *s* *x'*::*s'*) → (*ListAdd* *x* (*remove_list* *x'* *s*) *s'*).

Lemma *remove_list_fold_right* :
(*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*) →
(*s*:(*list* *elt*))(*x*:*elt*)(*Unique* *E.eq* *s*) → (*InList* *E.eq* *x* *s*) →

$(eqA\ (fold\_right\ f\ i\ s)\ (f\ x\ (fold\_right\ f\ i\ (remove\_list\ x\ s))))$.

Lemma $fold\_right\_equal$:
$(A{:}Set)(eqA{:}A\to A\to Prop)(st{:}(Setoid\_Theory\ A\ eqA))$
$(i{:}A)(f{:}elt\to A\to A)(compat\_op\ E.eq\ eqA\ f) \to (transpose\ eqA\ f) \to$
$(s,s'{:}(list\ elt))(Unique\ E.eq\ s) \to (Unique\ E.eq\ s') \to (ListEq\ s\ s') \to$
$(eqA\ (fold\_right\ f\ i\ s)\ (fold\_right\ f\ i\ s'))$.

Lemma $fold\_right\_add$:
$(A{:}Set)(eqA{:}A\to A\to Prop)(st{:}(Setoid\_Theory\ A\ eqA))$
$(i{:}A)(f{:}elt\to A\to A)(compat\_op\ E.eq\ eqA\ f) \to (transpose\ eqA\ f) \to$
$(s',s{:}(list\ elt))(x{:}elt)(Unique\ E.eq\ s) \to (Unique\ E.eq\ s') \to \neg(InList\ E.eq\ x\ s) \to$
$(ListAdd\ x\ s\ s') \to$
$(eqA\ (fold\_right\ f\ i\ s')\ (f\ x\ (fold\_right\ f\ i\ s)))$.

End $Unique\_Remove$.

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 $fold\_1$:
$(s{:}t)(A{:}Set)(eqA{:}A\to A\to Prop)(st{:}(Setoid\_Theory\ A\ eqA))(i{:}A)(f{:}elt\to A\to A)$
$(Empty\ s) \to (eqA\ (fold\ f\ s\ i)\ i)$.

Lemma $fold\_2$:
$(s,s'{:}t)(x{:}elt)(A{:}Set)(eqA{:}A\to A\to Prop)(st{:}(Setoid\_Theory\ A\ eqA))$
$(i{:}A)(f{:}elt\to A\to A)(compat\_op\ E.eq\ eqA\ f) \to (transpose\ eqA\ f) \to \neg(In\ x\ s) \to$
$(\mathsf{Add}\ x\ s\ s') \to (eqA\ (fold\ f\ s'\ i)\ (f\ x\ (fold\ f\ s\ i)))$.

Similar specification for *cardinal*.

Lemma $cardinal\_fold$: $(s{:}t)(cardinal\ s)=(fold\ [\_]S\ s\ O)$.

Lemma $cardinal\_1$: $(s{:}t)(Empty\ s) \to (cardinal\ s)=O$.

Lemma $cardinal\_2$:
$(s,s'{:}t)(x{:}elt)\neg(In\ x\ s) \to (\mathsf{Add}\ x\ s\ s') \to (cardinal\ s') = (S\ (cardinal\ s))$.

Hints $Resolve\ cardinal\_1\ cardinal\_2$.

End $Old\_Spec\_Now\_Properties$.

8 Induction principle over sets

Lemma Add_add :
$(s{:}t)(x{:}elt)(\mathsf{Add}\ x\ s\ (add\ x\ s))$.

Lemma Add_remove : $(s{:}t)(x{:}elt)(In\ x\ s) \to (\mathsf{Add}\ x\ (remove\ x\ s)\ s)$.

Hints $Resolve\ Add\_add\ Add\_remove$.

Lemma $Equal\_remove$: $(s,s'{:}t)(x{:}elt)(In\ x\ s)\to(Equal\ s\ s')\to$

(*Equal* (*remove* *x* *s*) (*remove* *x* *s'*)).

Hints *Resolve Equal_remove*.

Lemma *cardinal_inv_1* : (*s*:*t*)(*cardinal* *s*)=*O* → (*Empty* *s*).
Hints *Resolve cardinal_inv_1*.

Lemma *cardinal_inv_2* :
(*s*:*t*)(*n*:*nat*)(*cardinal* *s*)=(*S* *n*) → (*EX* *x*:*elt* | (*In* *x* *s*)).

Lemma *Equal_cardinal_aux* : (*n*:*nat*)(*s*,*s'*:*t*)(*cardinal* *s*)=*n* →
(*Equal* *s* *s'*)→(*cardinal* *s*)=(*cardinal* *s'*).

Lemma *Equal_cardinal* : (*s*,*s'*:*t*)(*Equal* *s* *s'*)→(*cardinal* *s*)=(*cardinal* *s'*).

Hints *Resolve Add_add Add_remove Equal_remove*
cardinal_inv_1 Equal_cardinal.

Lemma *empty_cardinal*: (*cardinal* *empty*)=*O*.

Lemma *empty_cardinal_2* : (*s*:*t*)(*Empty* *s*) → (*cardinal* *s*)=*O*.

Hints Immediate *empty_cardinal empty_cardinal_2* : *set*.

Lemma *cardinal_induction* : (*P* : *t* → *Prop*)
((*s*:*t*)(*Empty* *s*)→(*P* *s*)) →
((*s*,*s'*:*t*)(*P* *s*) → (*x*:*elt*)¬(*In* *x* *s*) → (Add *x* *s* *s'*) → (*P* *s'*)) →
(*n*:*nat*)(*s*:*t*)(*cardinal* *s*)=*n* → (*P* *s*).

Lemma *set_induction* : (*P* : *t* → *Prop*)
((*s*:*t*)(*Empty* *s*)→(*P* *s*)) →
((*s*,*s'*:*t*)(*P* *s*) → (*x*:*elt*)¬(*In* *x* *s*) → (Add *x* *s* *s'*) → (*P* *s'*)) →
(*s*:*t*)(*P* *s*).

Other formulation of *fold_1* and *fold_2*.

Section *Fold*.

Variable *s*,*s'*:*t*.
Variable *x*:*elt*.

Variable *A*:*Set*.
Variable *eqA*:*A*→*A*→*Prop*.
Variable *st*:(*Setoid_Theory* ? *eqA*).
Variable *i*:*A*.
Variable *f*:*elt*→*A*→*A*.
Variable *Comp*:(*compat_op* *E.eq* *eqA* *f*).
Variable *Assoc*:(*transpose* *eqA* *f*).

Lemma *fold_empty*: (*eqA* (*fold* *f* *empty* *i*) *i*).

Lemma *fold_equal*:
(*Equal* *s* *s'*) → (*eqA* (*fold* *f* *s* *i*) (*fold* *f* *s'* *i*)).

Lemma *fold_add*:
  ¬(*In x s*) → (*eqA* (*fold f* (*add x s*) *i*) (*f x* (*fold f s i*))).

End *Fold*.

properties of *Equal*

Lemma *equal_refl* : (*s*:*t*)(*Equal s s*).

Lemma *equal_sym* : (*s*,*s'*:*t*)(*Equal s s'*) → (*Equal s' s*).

Lemma *equal_trans* :
  (*s*,*u*,*v*:*t*)(*Equal s u*) → (*Equal u v*) → (*Equal s v*).

Hints *Resolve equal_refl equal_trans* : *set*.
Hints Immediate *equal_sym* : *set*.

properties of *Subset*

Lemma *subset_refl* : (*s*:*t*)(*Subset s s*).

Lemma *subset_antisym* :
  (*s*,*s'*:*t*)(*Subset s s'*) → (*Subset s' s*) → (*Equal s s'*).

Lemma *subset_trans* :
  (*s*,*t_*,*u*:*t*)(*Subset s t_*) → (*Subset t_ u*) → (*Subset s u*).

Lemma *subset_equal* :
  (*s*,*s'*:*t*)(*Equal s s'*) → (*Subset s s'*).

Lemma *subset_empty* :
  (*s*:*t*)(*Subset empty s*).

Lemma *subset_remove_3* :
  (*s1*,*s2*:*t*)(*x*:*elt*)
  (*Subset s1 s2*) → (*Subset* (*remove x s1*) *s2*).

Lemma *subset_diff* :
  (*s1*,*s2*,*s3*:*t*)
  (*Subset s1 s3*) → (*Subset* (*diff s1 s2*) *s3*).

Lemma *subset_add_3* :
  (*s1*,*s2*:*t*)(*x*:*elt*)
  (*In x s2*) → (*Subset s1 s2*) → (*Subset* (*add x s1*) *s2*).

Lemma *subset_add_2* :
  (*s1*,*s2*:*t*)(*x*:*elt*)
  (*Subset s1 s2*) → (*Subset s1* (*add x s2*)).

Lemma *in_subset* :
  (*s1*,*s2*:*t*)(*x*:*elt*)
  (*In x s1*) → (*Subset s1 s2*) → (*In x s2*).

Hints *Resolve subset_refl subset_equal subset_antisym*

subset_trans subset_empty subset_remove_3
subset_diff subset_add_3 subset_add_2 in_subset
: *set.*

properties of *empty*

Lemma *empty_is_empty_1* : $(s{:}t)(Empty\ s) \rightarrow (Equal\ s\ empty)$.

Lemma *empty_is_empty_2* : $(s{:}t)(Equal\ s\ empty) \rightarrow (Empty\ s)$.

Hints *Resolve empty_is_empty_1 empty_is_empty_2 : set.*

properties of *add*

Tactic Definition *Add_ y x s* :=
Generalize (!*add_1 s x y*) (!*add_2 s x y*) (!*add_3 s x y*).

Lemma *add_equal* :
$(s{:}t)(x{:}elt)(In\ x\ s) \rightarrow (Equal\ (add\ x\ s)\ s)$.

Hints *Resolve add_equal :set.*

properties of *remove*

Tactic Definition *Remove_ a x s* :=
Generalize (!*remove_1 s x a*) (!*remove_2 s x a*) (!*remove_3 s x a*).

Lemma *remove_equal*:
$(s{:}t)(x{:}elt)\neg(In\ x\ s) \rightarrow (Equal\ (remove\ x\ s)\ s)$.

Hints *Resolve remove_equal : set.*

properties of *add* and *remove*

Lemma *add_remove*:
$(s{:}t)(x{:}elt)(In\ x\ s) \rightarrow (Equal\ (add\ x\ (remove\ x\ s))\ s)$.

Lemma *remove_add*:
$(s{:}t)(x{:}elt)\neg(In\ x\ s) \rightarrow (Equal\ (remove\ x\ (add\ x\ s))\ s)$.

Hints Immediate *add_remove remove_add :set.*

properties of *singleton*

Tactic Definition *Singleton a x* :=
Generalize (!*singleton_1 x a*) (!*singleton_2 x a*).

Lemma *singleton_equal_add* : $(x{:}elt)$
$(Equal\ (singleton\ x)\ (add\ x\ empty))$.

Lemma *singleton_cardinal*: $(x{:}elt)(cardinal\ (singleton\ x))=(S\ O)$.

Hints *Resolve singleton_equal_add singleton_cardinal : set.*

properties of *union*

Tactic Definition *Union a s s'* :=
Generalize (!*union_1 s s' a*) (!*union_2 s s' a*) (!*union_3 s s' a*).

Lemma *union_sym* :
(s,s':t)(Equal (union s s') (union s' s)).

Lemma *union_subset_equal* :
(s,s':t)(Subset s s') → *(Equal (union s s') s').*

Lemma *union_equal_1* :
(s,s',s":t)(Equal s s') →
(Equal (union s s") (union s' s")).

Lemma *union_equal_2* :
(s,s',s":t)(Equal s' s") →
(Equal (union s s') (union s s")).

Lemma *union_assoc* :
(s,s',s":t)
(Equal (union (union s s') s") (union s (union s' s"))).

Hints *Resolve union_subset_equal union_equal_1 union_equal_2*
union_assoc : set.
Hints Immediate *union_sym.*

Lemma *add_union_singleton*:
(s:t)(x:elt)(Equal (add x s) (union (singleton x) s)).

Lemma *union_add*:
(s,s':t)(x:elt)
(Equal (union (add x s) s') (add x (union s s'))).

Lemma *union_subset_1* :
(s,s':t)(Subset s (union s s')).

Lemma *union_subset_2*:
(s,s':t)(Subset s' (union s s')).

Lemma *subset_union* :
(s,s',s":t)(Subset s s") → *(Subset s' s")* →
(Subset (union s s') s").

Hints *Resolve add_union_singleton union_add union_subset_1*
union_subset_2 subset_union : set.

properties of *inter*

Tactic Definition *Inter a s s'* :=
Generalize (!*inter_1 s s' a*) (!*inter_2 s s' a*) (!*inter_3 s s' a*).

Lemma *inter_sym*:
(s,s':t)(Equal (inter s s') (inter s' s)).

Lemma *inter_subset_equal*:
(s,s':t)(Subset s s') → *(Equal (inter s s') s).*

Lemma *inter_equal_1*:

(s,s',s":t)(Equal s s') →
(Equal (inter s s") (inter s' s")).

Lemma *inter_equal_2*:
(s,s',s":t)(Equal s' s") →
(Equal (inter s s') (inter s s")).

Hints Immediate *inter_sym* : *set*.
Hints *Resolve inter_subset_equal inter_equal_1 inter_equal_2* : *set*.

Lemma *inter_assoc*:
(s,s',s":t)
(Equal (inter (inter s s') s") (inter s (inter s' s"))).

Lemma *union_inter_1*:
(s,s',s":t)
(Equal (inter (union s s') s") (union (inter s s") (inter s' s"))).

Lemma *union_inter_2*:
(s,s',s":t)
(Equal (union (inter s s') s") (inter (union s s") (union s' s"))).

Lemma *inter_add_1*:
(s,s':t)(x:elt)(In x s') →
(Equal (inter (add x s) s') (add x (inter s s'))).

Lemma *inter_add_2*:
(s,s':t)(x:elt)¬*(In x s')* →
(Equal (inter (add x s) s') (inter s s')).

Hints *Resolve inter_assoc union_inter_1 union_inter_2*
inter_add_1 inter_add_2 : *set*.

properties of *Subset*

Lemma *inter_subset_1*:
(s,s':t)(Subset (inter s s') s).

Lemma *inter_subset_2*:
(s,s':t)(Subset (inter s s') s').

Lemma *inter_subset_3*:
(s,s',s":t)(Subset s" s) → *(Subset s" s')* →
(Subset s" (inter s s')).

Hints *Resolve inter_subset_1 inter_subset_2 inter_subset_3* : *set*.

Properties of *diff*

Tactic Definition *Diff a s s'* :=
Generalize (!*diff_1 s s' a*) (!*diff_2 s s' a*) (!*diff_3 s s' a*).

Lemma *diff_subset*:
(s,s':t)(Subset (diff s s') s).

Lemma *diff_subset_equal*:
(*s*,*s'*:*t*)(*Subset s s'*) → (*Equal* (*diff s s'*) *empty*).

Lemma *remove_inter_singleton*:
(*s*:*t*)(*x*:*elt*)(*Equal* (*remove x s*) (*diff s* (*singleton x*))).

Lemma *diff_inter_empty*:
(*s*,*s'*:*t*)(*Equal* (*inter* (*diff s s'*) (*inter s s'*)) *empty*).

Lemma *In_dec* : (*x*:*elt*)(*s*:*t*){ (*In x s*) }+ { ¬(*In x s*) }.

Lemma *diff_inter_all*:
(*s*,*s'*:*t*)(*Equal* (*union* (*diff s s'*) (*inter s s'*)) *s*).

Hints *Resolve diff_subset diff_subset_equal*
remove_inter_singleton diff_inter_empty
diff_inter_all : *set*.

Lemma *fold_plus*:
(*s*:*t*)(*p*:*nat*)(*fold* [_]*S s p*)=(*fold* [_]*S s O*)+*p*.

properties of *cardinal*

Lemma *empty_inter_1* : (*s*,*s'*:*t*)(*Empty s*) → (*Empty* (*inter s s'*)).

Lemma *empty_inter_2* : (*s*,*s'*:*t*)(*Empty s'*) → (*Empty* (*inter s s'*)).

Lemma *empty_union_1* : (*s*,*s'*:*t*)(*Empty s*) → (*Equal* (*union s s'*) *s'*).

Lemma *empty_union_2* : (*s*,*s'*:*t*)(*Empty s*) → (*Equal* (*union s' s*) *s'*).

Lemma *empty_diff_1* : (*s*,*s'*:*t*)(*Empty s*) → (*Empty* (*diff s s'*)).

Lemma *empty_diff_2* : (*s*,*s'*:*t*)(*Empty s*) → (*Equal* (*diff s' s*) *s'*).

Hints *Resolve empty_inter_1 empty_inter_2*
empty_union_1 empty_union_2
empty_diff_1 empty_diff_2 : *set*.

Lemma *union_Add* : (*s*,*s'*,*s"*:*t*)(*x*:*elt*)
(Add *x s s'*) → (Add *x* (*union s s"*) (*union s' s"*)).

Lemma *inter_Add* : (*s*,*s'*,*s"*:*t*)(*x*:*elt*)
(*In x s"*) → (Add *x s s'*) → (Add *x* (*inter s s"*) (*inter s' s"*)).

Lemma *union_Equal* : (*s*,*s'*,*s"*:*t*)(*x*:*elt*)
(*In x s"*) → (Add *x s s'*) → (*Equal* (*union s s"*) (*union s' s"*)).

Lemma *inter_Add_2* :(*s*,*s'*,*s"*:*t*)(*x*:*elt*)
¬(*In x s"*) → (Add *x s s'*) → (*Equal* (*inter s s"*) (*inter s' s"*)).

Lemma *not_in_union* : (*x*:*elt*)(*s*,*s'*:*t*)
¬(*In x s*) → ¬(*In x s'*) → ¬(*In x* (*union s s'*)).

Hints *Resolve union_Add inter_Add union_Equal inter_Add_2*
not_in_union : *set*.

Lemma *fold_commutes*:
$(A{:}Set)$
$(f{:}elt{\rightarrow}A{\rightarrow}A)(i{:}A)(compat\_op\ E.eq\ (eq\ ?)\ f) \rightarrow (transpose\ (eq\ ?)\ f) \rightarrow (s{:}t)(x{:}elt)$
$(fold\ f\ s\ (f\ x\ i)) = (f\ x\ (fold\ f\ s\ i))$.

Lemma *fold_union_inter*:
$(A{:}Set)$
$(f{:}elt{\rightarrow}A{\rightarrow}A)(i{:}A)(compat\_op\ E.eq\ (eq\ ?)\ f) \rightarrow (transpose\ (eq\ ?)\ f) \rightarrow$
$(s{,}s'{:}t)(fold\ f\ (union\ s\ s')\ (fold\ f\ (inter\ s\ s')\ i))$
$= (fold\ f\ s\ (fold\ f\ s'\ i))$.

Lemma *fold_diff_inter*:
$(A{:}Set)(f{:}elt{\rightarrow}A{\rightarrow}A)(i{:}A)(compat\_op\ E.eq\ (eq\ A)\ f) \rightarrow (transpose\ (eq\ A)\ f) \rightarrow$
$(s{,}s'{:}t)(fold\ f\ (diff\ s\ s')\ (fold\ f\ (inter\ s\ s')\ i)){=}(fold\ f\ s\ i)$.

Lemma *diff_inter_cardinal*:
$(s{,}s'{:}t)(cardinal\ (diff\ s\ s')){+}(cardinal\ (inter\ s\ s')){=}(cardinal\ s)$.

Lemma *subset_cardinal*:
$(s{,}s'{:}t)(Subset\ s\ s') \rightarrow (le\ (cardinal\ s)\ (cardinal\ s'))$.

Lemma *union_inter_cardinal*:
$(s{,}s'{:}t)(cardinal\ (union\ s\ s')){+}(cardinal\ (inter\ s\ s'))$
$= (cardinal\ s){+}(cardinal\ s')$.

Lemma *union_cardinal* :
$(s{,}s'{:}t)(le\ (cardinal\ (union\ s\ s'))\ (plus\ (cardinal\ s)\ (cardinal\ s')))$.

Lemma *add_cardinal_1*:
$(s{:}t)(x{:}elt)(In\ x\ s) \rightarrow (cardinal\ (add\ x\ s)){=}(cardinal\ s)$.

Lemma *add_cardinal_2*:
$(s{:}t)(x{:}elt)\neg(In\ x\ s) \rightarrow (cardinal\ (add\ x\ s)){=}(S\ (cardinal\ s))$.

Hints *Resolve subset_cardinal union_cardinal*
add_cardinal_1 add_cardinal_2.

End *Properties.*

Module FSetList

This file proposes an implementation of the non-dependant interface *FSetInterface.S* using strictly ordered list.

Require Export *FSetInterface.*

Set Implicit Arguments.

Usual syntax for lists. CAVEAT: the Coq cast "::" will no longer be available.
Notation "[]" := $(nil\ ?)$ $(at\ level\ 1)$.
Notation "$a :: l$" := $(cons\ a\ l)$ $(at\ level\ 1,\ l\ at\ next\ level)$.

9 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 ME := MoreOrderedType X.

  Definition elt := E.t.
  Definition t := (list elt).

  Definition empty : t := [].

  Definition is_empty : t → bool := [l]if l then true else [_,_]false.
```

9.1 The set operations.

```
  Fixpoint mem [x:elt; s:t] : bool := Cases s of
      nil ⇒ false
    | (cons y l) ⇒ Cases (E.compare x y) of
          (Lt _) ⇒ false
        | (Eq _) ⇒ true
        | (Gt _) ⇒ (mem x l)
        end
    end.

  Fixpoint add [x:elt; s:t] : t := Cases s of
      nil ⇒ x::[]
   | (cons y l) ⇒ Cases (E.compare x y) of
          (Lt _) ⇒ x::s
        | (Eq _) ⇒ s
        | (Gt _) ⇒ y::(add x l)
      end
  end.

  Definition singleton : elt → t := [x] x::[].

  Fixpoint remove [x:elt; s:t] : t := Cases s of
      nil ⇒ []
   | (cons y l) ⇒ Cases (E.compare x y) of
          (Lt _) ⇒ s
        | (Eq _) ⇒ l
        | (Gt _) ⇒ y::(remove x l)
      end
  end.
```

```
Fixpoint union [s:t] : t → t := Cases s of
   nil ⇒ [s']s'
| (cons x l) ⇒
       Fix union_aux { union_aux / 1 : t → t := [s']Cases s' of
            nil ⇒ s
          | (cons x' l') ⇒ Cases (E.compare x x') of
                 (Lt _ ) ⇒ x::(union l s')
               | (Eq _ ) ⇒ x::(union l l')
               | (Gt _) ⇒ x'::(union_aux l')
               end
          end}
 end.

Fixpoint inter [s:t] : t → t := Cases s of
   nil ⇒ [_] []
| (cons x l) ⇒
       Fix inter_aux { inter_aux / 1 : t → t := [s']Cases s' of
            nil ⇒ []
          | (cons x' l') ⇒ Cases (E.compare x x') of
                 (Lt _ ) ⇒ (inter l s')
               | (Eq _ ) ⇒ x::(inter l l')
               | (Gt _) ⇒ (inter_aux l')
               end
          end}
 end.

Fixpoint diff [s:t] : t → t := Cases s of
   nil ⇒ [_] []
| (cons x l) ⇒
       Fix diff_aux { diff_aux / 1 : t → t := [s']Cases s' of
            nil ⇒ s
          | (cons x' l') ⇒ Cases (E.compare x x') of
                 (Lt _ ) ⇒ x::(diff l s')
               | (Eq _ ) ⇒ (diff l l')
               | (Gt _) ⇒ (diff_aux l')
               end
          end}
 end.

Fixpoint equal [s:t] : t → bool := [s':t]Cases s s' of
   nil nil ⇒ true
| (cons x l) (cons x' l') ⇒ Cases (E.compare x x') of
       (Eq _) ⇒ (equal l l')
     | _ ⇒ false
```

```
   end
| _ _ ⇒ false
end.

Fixpoint subset [s,s':t] : bool := Cases s s' of
   nil _ ⇒ true
| (cons x l) (cons x' l') ⇒ Cases (E.compare x x') of
      (Lt _) ⇒ false
    | (Eq _) ⇒ (subset l l')
    | (Gt _) ⇒ (subset s l')
   end
| _ _ ⇒ false
end.

Fixpoint fold [B:Set; f:elt→B→B; s:t] : B → B := [i]Cases s of
   nil ⇒ i
| (cons x l) ⇒ (f x (fold f l i))
end.

Fixpoint filter [f:elt→bool; s:t] : t := Cases s of
   nil ⇒ []
| (cons x l) ⇒ if (f x) then x::(filter f l) else (filter f l)
end.

Fixpoint for_all [f:elt→bool; s:t] : bool := Cases s of
   nil ⇒ true
| (cons x l) ⇒ if (f x) then (for_all f l) else false
end.

Fixpoint exists [f:elt→bool; s:t] : bool := Cases s of
   nil ⇒ false
| (cons x l) ⇒ if (f x) then true else (exists f l)
end.

Fixpoint partition [f:elt→bool; s:t] : t×t := Cases s of
    nil ⇒ ([],[])
| (cons x l) ⇒ let (s1,s2) = (partition f l) in
                      if (f x) then (x::s1,s2) else (s1,x::s2)
end.

Definition cardinal : t → nat := [s](fold [_]S s O).

Definition elements : t → (list elt) := [x]x.

Definition min_elt : t → (option elt) := [s]Cases s of
    nil ⇒ (None ?)
 | (cons x _) ⇒ (Some ? x)
end.

Fixpoint max_elt [s:t] : (option elt) := Cases s of
```

$nil \Rightarrow (None\ ?)$
$\mid (cons\ x\ nil) \Rightarrow (Some\ ?\ x)$
$\mid (cons\ \_\ l) \Rightarrow (max\_elt\ l)$
end.

Definition $choose := min\_elt$.

9.2 Proofs of set operation specifications.

Definition $In : elt \rightarrow t \rightarrow Prop := (InList\ E.eq)$.
Notation "'$Sort$' l" := $(sort\ E.lt\ l)$ $(at\ level\ 10,\ l\ at\ level\ 8)$.
Notation "'Inf' x l" := $(lelistA\ E.lt\ x\ l)$ $(at\ level\ 10,\ x{,}l\ at\ level\ 8)$.
Notation "'In' x l" := $(InList\ E.eq\ x\ l)$ $(at\ level\ 10,\ x{,}l\ at\ level\ 8)$.

Definition $Equal := [s,s'](a{:}elt)(In\ a\ s)\leftrightarrow(In\ a\ s')$.
Definition $Subset := [s,s'](a{:}elt)(In\ a\ s)\rightarrow(In\ a\ s')$.
Definition Add $:= [x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x)\vee(In\ y\ s))$.
Definition $Empty := [s](a{:}elt)\neg(In\ a\ s)$.
Definition $For\_all := [P{:}elt\rightarrow Prop;\ s{:}t](x{:}elt)(In\ x\ s)\rightarrow(P\ x)$.
Definition $Exists := [P{:}elt\rightarrow Prop;\ s{:}t](EX\ x{:}elt \mid (In\ x\ s)\wedge(P\ x))$.

Definition In_eq: $(s{:}t)(x,y{:}elt)(E.eq\ x\ y) \rightarrow (In\ x\ s) \rightarrow (In\ y\ s) := ME.In\_eq$.
Definition Inf_lt : $(s{:}t)(x,y{:}elt)(E.lt\ x\ y) \rightarrow (Inf\ y\ s) \rightarrow (Inf\ x\ s) := ME.Inf\_lt$.
Definition Inf_eq : $(s{:}t)(x,y{:}elt)(E.eq\ x\ y) \rightarrow (Inf\ y\ s) \rightarrow (Inf\ x\ s) := ME.Inf\_eq$.
Definition In_sort : $(s{:}t)(x,a{:}elt)(Inf\ a\ s) \rightarrow (In\ x\ s) \rightarrow (Sort\ s) \rightarrow (E.lt\ a\ x) := ME.In\_sort$.
Hints $Resolve\ Inf\_lt\ Inf\_eq$.
Hints Immediate In_eq.

Lemma mem_1: $(s{:}t)(Hs{:}(Sort\ s))(x{:}elt)(In\ x\ s) \rightarrow (mem\ x\ s)=true$.

Lemma mem_2: $(s{:}t)(x{:}elt)(mem\ x\ s)=true \rightarrow (In\ x\ s)$.

Lemma add_Inf : $(s{:}t)(x,a{:}elt)(Inf\ a\ s)\rightarrow(E.lt\ a\ x)\rightarrow(Inf\ a\ (add\ x\ s))$.
Hints $Resolve\ add\_Inf$.

Lemma add_sort : $(s{:}t)(Hs{:}(Sort\ s))(x{:}elt)(Sort\ (add\ x\ s))$.

Lemma add_1 : $(s{:}t)(Hs{:}(Sort\ s))(x,y{:}elt)(E.eq\ y\ x) \rightarrow (In\ y\ (add\ x\ s))$.

Lemma add_2 : $(s{:}t)(Hs{:}(Sort\ s))(x,y{:}elt)(In\ y\ s) \rightarrow (In\ y\ (add\ x\ s))$.

Lemma add_3 : $(s{:}t)(Hs{:}(Sort\ s))(x,y{:}elt)$
$\neg(E.eq\ x\ y) \rightarrow (In\ y\ (add\ x\ s)) \rightarrow (In\ y\ s)$.

Lemma $remove\_Inf$: $(s{:}t)(Hs{:}(Sort\ s))(x,a{:}elt)(Inf\ a\ s)\rightarrow$
$(Inf\ a\ (remove\ x\ s))$.
Hints $Resolve\ remove\_Inf$.

Lemma $remove\_sort$: $(s{:}t)(Hs{:}(Sort\ s))(x{:}elt)(Sort\ (remove\ x\ s))$.

Lemma *remove_1* : $(s{:}t)(Hs{:}(Sort\ s))(x{,}y{:}elt)(E.eq\ y\ x) \to \neg(In\ y\ (remove\ x\ s))$.

Lemma *remove_2* : $(s{:}t)(Hs{:}(Sort\ s))(x{,}y{:}elt)$
$\neg(E.eq\ x\ y) \to (In\ y\ s) \to (In\ y\ (remove\ x\ s))$.

Lemma *remove_3* : $(s{:}t)(Hs{:}(Sort\ s))(x{,}y{:}elt)(In\ y\ (remove\ x\ s)) \to (In\ y\ s)$.

Lemma *singleton_sort* : $(x{:}elt)(Sort\ (singleton\ x))$.

Lemma *singleton_1* : $(x{,}y{:}elt)(In\ y\ (singleton\ x)) \to (E.eq\ x\ y)$.

Lemma *singleton_2* : $(x{,}y{:}elt)(E.eq\ x\ y) \to (In\ y\ (singleton\ x))$.

Tactic Definition *DoubleInd* :=
Induction s;
[*Simpl*; *Auto*; *Try Solve* [*Intros*; *Inversion H*] |
Intros x l Hrec;
Induction s';
[*Simpl*; *Auto*; *Try Solve* [*Intros*; *Inversion H*] |
Intros x' l' Hrec' Hs Hs'; *Inversion Hs*; *Inversion Hs'*; *Subst*; *Simpl*]].

Lemma *union_Inf* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(a{:}elt)$
$(Inf\ a\ s) \to (Inf\ a\ s') \to (Inf\ a\ (union\ s\ s'))$.
Hints *Resolve union_Inf*.

Lemma *union_sort* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(Sort\ (union\ s\ s'))$.

Lemma *union_1* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ (union\ s\ s')) \to (In\ x\ s)\vee(In\ x\ s')$.

Lemma *union_2* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ s) \to (In\ x\ (union\ s\ s'))$.

Lemma *union_3* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ s') \to (In\ x\ (union\ s\ s'))$.

Lemma *inter_Inf* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(a{:}elt)$
$(Inf\ a\ s) \to (Inf\ a\ s') \to (Inf\ a\ (inter\ s\ s'))$.
Hints *Resolve inter_Inf*.

Lemma *inter_sort* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(Sort\ (inter\ s\ s'))$.

Lemma *inter_1* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ (inter\ s\ s')) \to (In\ x\ s)$.

Lemma *inter_2* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ (inter\ s\ s')) \to (In\ x\ s')$.

Lemma *inter_3* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(x{:}elt)$
$(In\ x\ s) \to (In\ x\ s') \to (In\ x\ (inter\ s\ s'))$.

Lemma *diff_Inf* : $(s{,}s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(a{:}elt)$

(Inf a s) → (Inf a s') → (Inf a (diff s s')).
Hints *Resolve diff_Inf.*

Lemma *diff_sort : (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(Sort (diff s s')).*

Lemma *diff_1 : (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(x:elt)*
(In x (diff s s')) → (In x s).

Lemma *diff_2 : (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(x:elt)*
(In x (diff s s')) → ¬(In x s').

Lemma *diff_3 : (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))(x:elt)*
(In x s) → ¬(In x s') → (In x (diff s s')).

Lemma *equal_1: (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))*
(Equal s s') → (equal s s')=true.

Lemma *equal_2: (s,s':t)(equal s s')=true → (Equal s s').*

Lemma *subset_1: (s,s':t)(Hs:(Sort s))(Hs':(Sort s'))*
(Subset s s') → (subset s s')=true.

Lemma *subset_2: (s,s':t)(subset s s')=true → (Subset s s').*

Lemma *empty_sort : (Sort empty).*

Lemma *empty_1 : (Empty empty).*

Lemma *is_empty_1 : (s:t)(Empty s) → (is_empty s)=true.*

Lemma *is_empty_2 : (s:t)(is_empty s)=true → (Empty s).*

Lemma *elements_1: (s:t)(x:elt)(In x s) → (InList E.eq x (elements s)).*

Lemma *elements_2: (s:t)(x:elt)(InList E.eq x (elements s)) → (In x s).*

Lemma *elements_3: (s:t)(Hs:(Sort s))(sort E.lt (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) → ¬(E.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) → ¬(E.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*: $(s{:}t)(choose\ s)=(None\ ?) \rightarrow (Empty\ s)$
:= *min_elt_3*.

Lemma *fold_1* :
$(s{:}t)(Hs{:}(Sort\ s))(A{:}Set)(i{:}A)(f{:}elt{\rightarrow}A{\rightarrow}A)$
$(EX\ l{:}(list\ elt) \mid (Unique\ E.eq\ l) \wedge$
$((x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)) \wedge (fold\ f\ s\ i)=(fold\_right$
$f\ i\ l))$.

Lemma *cardinal_1* :
$(s{:}t)(Hs{:}(Sort\ s))(EX\ l{:}(list\ elt) \mid (Unique\ E.eq\ l) \wedge$
$((x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)) \wedge (cardinal\ s)=(length$
$l))$.

Lemma *filter_Inf* :
$(s{:}t)(Hs{:}(Sort\ s))(x{:}elt)(f{:}elt{\rightarrow}bool)(Inf\ x\ s) \rightarrow (Inf\ x\ (filter\ f\ s))$.

Lemma *filter_sort* : $(s{:}t)(Hs{:}(Sort\ s))(f{:}elt{\rightarrow}bool)(Sort\ (filter\ f\ s))$.

Lemma *filter_1*: $(s{:}t)(x{:}elt)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(In\ x\ (filter\ f\ s)) \rightarrow (In\ x\ s)$.

Lemma *filter_2*:
$(s{:}t)(x{:}elt)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (In\ x\ (filter\ f\ s)) \rightarrow$
$(f\ x)=true$.

Lemma *filter_3*:
$(s{:}t)(x{:}elt)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(In\ x\ s) \rightarrow (f\ x)=true \rightarrow (In\ x\ (filter\ f\ s))$.

Lemma *for_all_1*:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
$(For\_all\ [x](f\ x)=true\ s) \rightarrow (for\_all\ f\ s)=true$.

Lemma *for_all_2*:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (for\_all\ f\ s)=true \rightarrow$
$(For\_all\ [x](f\ x)=true\ s)$.

Lemma *exists_1*:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (Exists\ [x](f\ x)=true\ s) \rightarrow$
$(exists\ f\ s)=true$.

Lemma *exists_2*:
$(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow (exists\ f\ s)=true \rightarrow$
$(Exists\ [x](f\ x)=true\ s)$.

Lemma *partition_Inf_1* :
$(s{:}t)(Hs{:}(Sort\ s))(f{:}elt{\rightarrow}bool)(x{:}elt)(Inf\ x\ s) \rightarrow$
$(Inf\ x\ (fst\ ?\ ?\ (partition\ f\ s)))$.

Lemma *partition_Inf_2* :
  $(s{:}t)(Hs{:}(Sort\ s))(f{:}elt{\rightarrow}bool)(x{:}elt)(Inf\ x\ s) \rightarrow$
  $(Inf\ x\ (snd\ ?\ ?\ (partition\ f\ s)))$.

Lemma *partition_sort_1* : $(s{:}t)(Hs{:}(Sort\ s))(f{:}elt{\rightarrow}bool)$
  $(Sort\ (fst\ ?\ ?\ (partition\ f\ s)))$.

Lemma *partition_sort_2* : $(s{:}t)(Hs{:}(Sort\ s))(f{:}elt{\rightarrow}bool)$
  $(Sort\ (snd\ ?\ ?\ (partition\ f\ s)))$.

Lemma *partition_1*:
  $(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
  $(Equal\ (fst\ ?\ ?\ (partition\ f\ s))\ (filter\ f\ s))$.

Lemma *partition_2*:
  $(s{:}t)(f{:}elt{\rightarrow}bool)(compat\_bool\ E.eq\ f) \rightarrow$
  $(Equal\ (snd\ ?\ ?\ (partition\ f\ s))\ (filter\ [x](negb\ (f\ x))\ s))$.

Definition *eq* : $t \rightarrow t \rightarrow Prop := Equal$.

Lemma *eq_refl*: $(s{:}t)(eq\ s\ s)$.

Lemma *eq_sym*: $(s,s'{:}t)(eq\ s\ s') \rightarrow (eq\ s'\ s)$.

Lemma *eq_trans*: $(s,s',s''{:}t)(eq\ s\ s') \rightarrow (eq\ s'\ s'') \rightarrow (eq\ s\ s'')$.

Inductive *lt* : $t \rightarrow t \rightarrow Prop :=$
  $lt\_nil : (x{:}elt)(s{:}t)(lt\ []\ (x{::}s))$
  $\mid lt\_cons\_lt : (x,y{:}elt)(s,s'{:}t)(E.lt\ x\ y) \rightarrow (lt\ (x{::}s)\ (y{::}s'))$
  $\mid lt\_cons\_eq : (x,y{:}elt)(s,s'{:}t)(E.eq\ x\ y) \rightarrow (lt\ s\ s') \rightarrow (lt\ (x{::}s)\ (y{::}s'))$.
Hint *lt* := *Constructors lt.*

Lemma *lt_trans* : $(s,s',s''{:}t)(lt\ s\ s') \rightarrow (lt\ s'\ s'') \rightarrow (lt\ s\ s'')$.

Lemma *lt_not_eq* : $(s,s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(lt\ s\ s') \rightarrow \neg(eq\ s\ s')$.

Definition *compare* : $(s,s'{:}t)(Hs{:}(Sort\ s))(Hs'{:}(Sort\ s'))(Compare\ lt\ eq\ s\ s')$.

End *Raw.*

10 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 $E := X$.
Module *Raw* := $(Raw\ X)$.

Record *slist* : *Set* := { *this* :> *Raw.t* ; *sorted* : (*sort E.lt this*) }.
Definition *t* := *slist*.
Definition *elt* := *X.t*.

Definition *In* := [*x*:*elt*;*s*:*t*](*Raw.In x s*).
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*](*EX x*:*elt* | (*In x s*)∧(*P x*)).

Definition *In_1* := [*s*:*t*](!*Raw.In_eq s*).

Definition *mem* := [*x*:*elt*;*s*:*t*](*Raw.mem x s*).
Definition *mem_1* := [*s*:*t*](*Raw.mem_1* (*sorted s*)).
Definition *mem_2* := [*s*:*t*](!*Raw.mem_2 s*).

Definition *add* := [*x*,*s*](*Build_slist* (*Raw.add_sort* (*sorted s*) *x*)).
Definition *add_1* := [*s*:*t*](*Raw.add_1* (*sorted s*)).
Definition *add_2* := [*s*:*t*](*Raw.add_2* (*sorted s*)).
Definition *add_3* := [*s*:*t*](*Raw.add_3* (*sorted s*)).

Definition *remove* := [*x*,*s*](*Build_slist* (*Raw.remove_sort* (*sorted s*) *x*)).
Definition *remove_1* := [*s*:*t*](*Raw.remove_1* (*sorted s*)).
Definition *remove_2* := [*s*:*t*](*Raw.remove_2* (*sorted s*)).
Definition *remove_3* := [*s*:*t*](*Raw.remove_3* (*sorted s*)).

Definition *singleton* := [*x*](*Build_slist* (*Raw.singleton_sort x*)).
Definition *singleton_1* := *Raw.singleton_1*.
Definition *singleton_2* := *Raw.singleton_2*.

Definition *union* := [*s*,*s'*:*t*](*Build_slist* (*Raw.union_sort* (*sorted s*) (*sorted s'*))).
Definition *union_1* := [*s*,*s'*:*t*](*Raw.union_1* (*sorted s*) (*sorted s'*)).
Definition *union_2* := [*s*,*s'*:*t*](*Raw.union_2* (*sorted s*) (*sorted s'*)).
Definition *union_3* := [*s*,*s'*:*t*](*Raw.union_3* (*sorted s*) (*sorted s'*)).

Definition *inter* :=[*s*,*s'*:*t*](*Build_slist* (*Raw.inter_sort* (*sorted s*) (*sorted s'*))).

End *Make*.

Module FSetRBT

This module implements sets using red-black trees

Require *FSetInterface*.
Require *FSetList*.

```
Require FSetBridge.

Require Omega.
Require ZArith.

Import Z_scope.
Open Scope Z_scope.

Notation "[]" := (nil ?) (at level 1).
Notation "a :: l" := (cons a l) (at level 1, l at next level).

Set Ground Depth 3.

Module Make [X : OrderedType] <: Sdep with Module E := X.

  Module E := X.
  Module ME := MoreOrderedType X.

  Definition elt := X.t.
```

11 Red-black trees

```
  Inductive color : Set := red : color | black : color.

  Inductive tree : Set :=
  | Leaf : tree
  | Node : color → tree → X.t → tree → tree.
```

12 Occurrence in a tree

```
  Inductive In_tree [x:elt] : tree → Prop :=
  | IsRoot : (l,r:tree)(c:color)(y:elt)
                 (X.eq x y) → (In_tree x (Node c l y r))
  | InLeft : (l,r:tree)(c:color)(y:elt)
                 (In_tree x l) → (In_tree x (Node c l y r))
  | InRight : (l,r:tree)(c:color)(y:elt)
                 (In_tree x r) → (In_tree x (Node c l y r)).

  Hint In_tree := Constructors In_tree.
```

In_tree is color-insensitive

```
  Lemma In_color : (c,c':color)(x,y:elt)(l,r:tree)
    (In_tree y (Node c l x r)) → (In_tree y (Node c' l x r)).
  Hints Resolve In_color.
```

13 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_tree* y s) → (*X.lt* y x).
Definition *gt_tree* [x:*elt*; s:*tree*] := (y:*elt*)(*In_tree* y s) → (*X.lt* x y).

Hints *Unfold lt_tree gt_tree.*

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*)(c:*color*)
(*lt_tree* x l) → (*lt_tree* x r) → (*X.lt* y x) →
(*lt_tree* x (*Node* c l y r)).

Lemma *gt_tree_node* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*gt_tree* x l) → (*gt_tree* x r) → (*E.lt* x y) →
(*gt_tree* x (*Node* c l y r)).

Hints *Resolve lt_leaf gt_leaf lt_tree_node gt_tree_node.*

Lemma *lt_node_lt* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*lt_tree* x (*Node* c l y r)) → (*E.lt* y x).

Lemma *gt_node_gt* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*gt_tree* x (*Node* c l y r)) → (*E.lt* x y).

Lemma *lt_left* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*lt_tree* x (*Node* c l y r)) → (*lt_tree* x l).

Lemma *lt_right* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*lt_tree* x (*Node* c l y r)) → (*lt_tree* x r).

Lemma *gt_left* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*gt_tree* x (*Node* c l y r)) → (*gt_tree* x l).

Lemma *gt_right* : (x,y:*elt*)(l,r:*tree*)(c:*color*)
(*gt_tree* x (*Node* c l y r)) → (*gt_tree* x r).

Hints *Resolve lt_node_lt gt_node_gt*
lt_left lt_right gt_left gt_right.

Lemma *lt_tree_not_in* :
(x:*elt*)(t:*tree*)(*lt_tree* x t) → ¬(*In_tree* x t).

Lemma *lt_tree_trans* :
(x,y:*elt*)(*X.lt* x y) → (t:*tree*)(*lt_tree* x t) → (*lt_tree* y t).

Lemma *gt_tree_not_in* :
(x:*elt*)(t:*tree*)(*gt_tree* x t) → ¬(*In_tree* x t).

Lemma *gt_tree_trans* :

$(x,y{:}elt)(X.lt\ y\ x) \to (t{:}tree)(gt\_tree\ x\ t) \to (gt\_tree\ y\ t).$

Hints *Resolve lt_tree_not_in lt_tree_trans gt_tree_not_in gt_tree_trans.*

bst t : *t* is a binary search tree

Inductive $bst : tree \to Prop :=$
$|\ BSLeaf :$
$(bst\ Leaf)$
$|\ BSNode : (x{:}elt)(l,r{:}tree)(c{:}color)$
$(bst\ l) \to (bst\ r) \to$
$(lt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to$
$(bst\ (Node\ c\ l\ x\ r)).$

Hint $bst :=$ *Constructors bst.*

Results about *bst*

Lemma $bst\_left : (x{:}elt)(l,r{:}tree)(c{:}color)$
$(bst\ (Node\ c\ l\ x\ r)) \to (bst\ l).$

Lemma $bst\_right : (x{:}elt)(l,r{:}tree)(c{:}color)$
$(bst\ (Node\ c\ l\ x\ r)) \to (bst\ r).$

Implicits *bst_left.* Implicits *bst_right.*
Hints *Resolve bst_left bst_right.*

Lemma $bst\_color : (c,c'{:}color)(x{:}elt)(l,r{:}tree)$
$(bst\ (Node\ c\ l\ x\ r)) \to (bst\ (Node\ c'\ l\ x\ r)).$
Hints *Resolve bst_color.*

Key fact about binary search trees: rotations preserve the *bst* property

Lemma $rotate\_left : (x,y{:}elt)(a,b,c{:}tree)(c1,c2,c3,c4{:}color)$
$(bst\ (Node\ c1\ a\ x\ (Node\ c2\ b\ y\ c))) \to$
$(bst\ (Node\ c3\ (Node\ c4\ a\ x\ b)\ y\ c)).$

Lemma $rotate\_right : (x,y{:}elt)(a,b,c{:}tree)(c1,c2,c3,c4{:}color)$
$(bst\ (Node\ c3\ (Node\ c4\ a\ x\ b)\ y\ c)) \to$
$(bst\ (Node\ c1\ a\ x\ (Node\ c2\ b\ y\ c))).$

Hints *Resolve rotate_left rotate_right.*

14 Balanced red-black trees

rbtree n t: *t* is a properly balanced red-black tree, i.e. it satisfies the following two invariants:

- a red node has no red son
- any path from the root to a leaf has exactly n black nodes

```
Definition is_not_red [t:tree] := Cases t of
| Leaf ⇒ True
| (Node black x0 x1 x2) ⇒ True
| (Node red x0 x1 x2) ⇒ False
end.

Hints Unfold is_not_red.

Inductive rbtree : nat → tree → Prop :=
| RBLeaf :
    (rbtree O Leaf)
| RBRed : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) → (rbtree n r) →
    (is_not_red l) → (is_not_red r) →
    (rbtree n (Node red l x r))
| RBBlack : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) → (rbtree n r) →
    (rbtree (S n) (Node black l x r)).

Hint rbtree := Constructors rbtree.
```

Results about *rbtree*

```
Lemma rbtree_left :
  (x:elt)(l,r:tree)(c:color)
  (EX n:nat | (rbtree n (Node c l x r))) →
  (EX n:nat | (rbtree n l)).

Lemma rbtree_right :
  (x:elt)(l,r:tree)(c:color)
  (EX n:nat | (rbtree n (Node c l x r))) →
  (EX n:nat | (rbtree n r)).

Implicits rbtree_left. Implicits rbtree_right.
Hints Resolve rbtree_left rbtree_right.
```

15 Sets as red-black trees

A set is implement as a record *t*, containing a tree, a proof that it is a binary search tree and a proof that it is a properly balanced red-black tree

```
Record t_ : Set := t_intro {
  the_tree :> tree;
  is_bst : (bst the_tree);
  is_rbtree : (EX n:nat | (rbtree n the_tree)) }.
Definition t := t_.
```

Projections

Lemma t_is_bst : $(s{:}t)(bst\ s)$.
Hints $Resolve\ t\_is\_bst$.

16 Logical appartness

Definition In : $elt \to t \to Prop$:= $[x{:}elt][s{:}t](In\_tree\ x\ s)$.

Definition $Equal$:= $[s,s'](a{:}elt)(In\ a\ s)\leftrightarrow(In\ a\ s')$.
Definition $Subset$:= $[s,s'](a{:}elt)(In\ a\ s)\to(In\ a\ s')$.
Definition Add := $[x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x)\vee(In\ y\ s))$.
Definition $Empty$:= $[s](a{:}elt)\neg(In\ a\ s)$.
Definition For_all := $[P{:}elt\to Prop;\ s{:}t](x{:}elt)(In\ x\ s)\to(P\ x)$.
Definition $Exists$:= $[P{:}elt\to Prop;\ s{:}t](EX\ x{:}elt \mid (In\ x\ s)\wedge(P\ x))$.

Lemma eq_In: $(s{:}t)(x,y{:}elt)(E.eq\ x\ y) \to (In\ x\ s) \to (In\ y\ s)$.

Hints $Resolve\ eq\_In$.

17 Empty set

Definition t_empty : t.

Definition $empty$: $\{\ s{:}t \mid (x{:}elt)\neg(In\ x\ s)\ \}$.

18 Emptyness test

Definition is_empty : $(s{:}t)\{\ Empty\ s\ \}+\{\ \neg(Empty\ s)\ \}$.

19 Appartness

The *mem* function is deciding appartness. It exploits the *bst* property to achieve logarithmic complexity.

Definition mem : $(x{:}elt)\ (s{:}t)\ \{\ (In\ x\ s)\ \} + \{\ \neg(In\ x\ s)\ \}$.

20 Singleton set

Definition $singleton\_tree$ $[x{:}elt]$:= $(Node\ red\ Leaf\ x\ Leaf)$.

Lemma $singleton\_bst$: $(x{:}elt)(bst\ (singleton\_tree\ x))$.

Lemma $singleton\_rbtree$: $(x{:}elt)(EX\ n{:}nat \mid (rbtree\ n\ (singleton\_tree\ x)))$.

Definition $singleton$: $(x{:}elt)\{\ s{:}t \mid (y{:}elt)(In\ y\ s) \leftrightarrow (E.eq\ x\ y)\}$.

21 Insertion

almost_rbtree n t: *t* may have one red-red conflict at its root; it satisfies *rbtree n* everywhere else

Inductive $almost\_rbtree : nat \rightarrow tree \rightarrow Prop :=$
$\mid ARBLeaf :$
$\quad (almost\_rbtree\ O\ Leaf)$
$\mid ARBRed : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$\quad (rbtree\ n\ l) \rightarrow (rbtree\ n\ r) \rightarrow$
$\quad (almost\_rbtree\ n\ (Node\ red\ l\ x\ r))$
$\mid ARBBlack : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$\quad (rbtree\ n\ l) \rightarrow (rbtree\ n\ r) \rightarrow$
$\quad (almost\_rbtree\ (S\ n)\ (Node\ black\ l\ x\ r)).$

Hint $almost\_rbtree := Constructors\ almost\_rbtree.$

Results about *almost_rbtree*

Lemma $rbtree\_almost\_rbtree :$
$\quad (n{:}nat)(t{:}tree)(rbtree\ n\ t) \rightarrow (almost\_rbtree\ n\ t).$

Hints $Resolve\ rbtree\_almost\_rbtree.$

Lemma $rbtree\_almost\_rbtree\_ex : (s{:}tree)$
$\quad (EX\ n{:}nat \mid (rbtree\ n\ s)) \rightarrow (EX\ n{:}nat \mid (almost\_rbtree\ n\ s)).$

Hints $Resolve\ rbtree\_almost\_rbtree\_ex.$

Lemma $almost\_rbtree\_rbtree\_black : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$\quad (almost\_rbtree\ n\ (Node\ black\ l\ x\ r)) \rightarrow$
$\quad (rbtree\ n\ (Node\ black\ l\ x\ r)).$
Hints $Resolve\ almost\_rbtree\_rbtree\_black.$

Balancing functions *lbalance* and *rbalance* (see below) require a rather complex pattern-matching; it is realized by the following function *conflict* which returns in the sum type *Conflict*

Inductive $Conflict\ [s{:}tree] : Set :=$
$\mid NoConflict :$
$\quad ((n{:}nat)\ (almost\_rbtree\ n\ s) \rightarrow (rbtree\ n\ s)) \rightarrow$
$\quad (Conflict\ s)$
$\mid RedRedConflict :$
$\quad (a,b,c{:}tree)(x,y{:}elt)$
$\quad (bst\ a) \rightarrow (bst\ b) \rightarrow (bst\ c) \rightarrow$
$\quad (lt\_tree\ x\ a) \rightarrow (gt\_tree\ x\ b) \rightarrow$
$\quad (lt\_tree\ y\ b) \rightarrow (gt\_tree\ y\ c) \rightarrow (X.lt\ x\ y) \rightarrow$
$\quad ((z{:}elt)(In\_tree\ z\ s) \leftrightarrow$
$\quad\quad ((X.eq\ z\ x) \vee (X.eq\ z\ y) \vee$
$\quad\quad (In\_tree\ z\ a) \vee (In\_tree\ z\ b) \vee (In\_tree\ z\ c))) \rightarrow$

$((n{:}nat)(almost\_rbtree\ n\ s) \rightarrow$
$\quad((rbtree\ n\ a) \wedge (rbtree\ n\ b) \wedge (rbtree\ n\ c))) \rightarrow$
$(Conflict\ s).$

Definition $conflict$: $(s{:}tree)(bst\ s) \rightarrow (Conflict\ s)$.

$lbalance\ c\ x\ l\ r$ acts as a black node constructor, solving a possible red-red conflict on l, using the following schema:

```
B (R (R a x b) y c) z d
B (R a x (R b y c)) z d -> R (B a x b) y (R c z d)
B a x b -> B a x b
```

The result is not necessarily a black node.

Definition $lbalance$:
$(l{:}tree)(x{:}elt)(r{:}tree)$
$(lt\_tree\ x\ l) \rightarrow (gt\_tree\ x\ r) \rightarrow$
$(bst\ l) \rightarrow (bst\ r) \rightarrow$
$\{\ s{:}tree\ |$
$\quad(bst\ s) \wedge$
$\quad((n{:}nat)((almost\_rbtree\ n\ l) \wedge (rbtree\ n\ r)) \rightarrow$
$\qquad(rbtree\ (S\ n)\ s)) \wedge$
$\quad(y{:}elt)(In\_tree\ y\ s) \leftrightarrow ((E.eq\ y\ x) \vee (In\_tree\ y\ l) \vee (In\_tree\ y\ r))\ \}.$

$rbalance\ l\ x\ r$ is similar to $lbalance$, solving a possible red-red conflict on r. The balancing schema is similar:

```
B a x (R (R b y c) z d)
B a x (R b y (R c z d)) -> R (B a x b) y (R c z d)
B a x b -> B a x b
```

Definition $rbalance$:
$(l{:}tree)(x{:}elt)(r{:}tree)$
$(lt\_tree\ x\ l) \rightarrow (gt\_tree\ x\ r) \rightarrow$
$(bst\ l) \rightarrow (bst\ r) \rightarrow$
$\{\ s{:}tree\ |$
$\quad(bst\ s) \wedge$
$\quad((n{:}nat)((almost\_rbtree\ n\ r) \wedge (rbtree\ n\ l)) \rightarrow$
$\qquad(rbtree\ (S\ n)\ s)) \wedge$
$\quad(y{:}elt)(In\_tree\ y\ s) \leftrightarrow ((E.eq\ y\ x) \vee (In\_tree\ y\ l) \vee (In\_tree\ y\ r))\ \}.$

$insert\ x\ s$ inserts x in tree s, resulting in a possible top red-red conflict when s is red. Its code is:

```
insert x Empty =
  Node red Empty x Empty
insert x (Node red a y b) =
```

```
    if lt x y then Node red (insert x a) y b
    else if lt y x then Node red a y (insert x b)
    else (Node c a y b)
  insert x (Node black a y b) =
    if lt x y then lbalance (insert x a) y b
    else if lt y x then rbalance a y (insert x b)
    else (Node c a y b)
```

Definition *insert* :
$(x{:}elt)\ (s{:}t)$
$\{\ s'{:}tree \mid (bst\ s') \wedge$
$((n{:}nat)(rbtree\ n\ s) \rightarrow$
$((almost\_rbtree\ n\ s') \wedge$
$((is\_not\_red\ s) \rightarrow (rbtree\ n\ s')))) \wedge$
$(y{:}elt)(In\_tree\ y\ s') \leftrightarrow ((E.eq\ y\ x) \vee (In\_tree\ y\ s))\ \}.$

Finally *add x s* calls *insert* and recolors the root as black:

```
add x s = match insert x s with
  | Node _ a y b -> Node black a y b
  | Empty -> assert false (* absurd *)
```

Definition *add* : $(x{:}elt)\ (s{:}t)\ \{\ s'{:}t \mid (\mathsf{Add}\ x\ s\ s')\ \}.$

22 Deletion

UnbalancedLeft n t: *t* is a tree of black height $S\ n$ on its left side and n on its right side (the root color is taken into account, whathever it is)

Inductive *UnbalancedLeft* : $nat \rightarrow tree \rightarrow Prop :=$
$\mid ULRed : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$(rbtree\ (S\ n)\ l) \rightarrow (rbtree\ n\ r) \rightarrow$
$(is\_not\_red\ l) \rightarrow$
$(UnbalancedLeft\ n\ (Node\ red\ l\ x\ r))$
$\mid ULBlack : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$(rbtree\ (S\ n)\ l) \rightarrow (rbtree\ n\ r) \rightarrow$
$(UnbalancedLeft\ (S\ n)\ (Node\ black\ l\ x\ r)).$

when a tree is unbalanced on its left, it can be repared

Definition *unbalanced_left* :
$(s{:}tree)(bst\ s) \rightarrow$
$\{\ r : tree \times bool \mid$
$let\ (s',b) = r\ in$
$(bst\ s') \wedge$

$((is\_not\_red\ s) \wedge b{=}false \rightarrow (is\_not\_red\ s')) \wedge$
$((n{:}nat)(UnbalancedLeft\ n\ s) \rightarrow$
$\quad (if\ b\ then\ (rbtree\ n\ s')\ else\ (rbtree\ (S\ n)\ s'))) \wedge$
$((y{:}elt)(In\_tree\ y\ s') \leftrightarrow (In\_tree\ y\ s))\ \}.$

UnbalancedRight n t: *t* is a tree of black height $S\ n$ on its right side and n on its left side (the root color is taken into account, whathever it is)

Inductive $UnbalancedRight : nat \rightarrow tree \rightarrow Prop :=$
$\mid URRed : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$\quad (rbtree\ n\ l) \rightarrow (rbtree\ (S\ n)\ r) \rightarrow$
$\quad (is\_not\_red\ r) \rightarrow$
$\quad (UnbalancedRight\ n\ (Node\ red\ l\ x\ r))$
$\mid URBlack : (x{:}elt)(l,r{:}tree)(n{:}nat)$
$\quad (rbtree\ n\ l) \rightarrow (rbtree\ (S\ n)\ r) \rightarrow$
$\quad (UnbalancedRight\ (S\ n)\ (Node\ black\ l\ x\ r)).$

when a tree is unbalanced on its right, it can be repared

Definition $unbalanced\_right$:
$(s{:}tree)(bst\ s) \rightarrow$
$\{\ r : tree \times bool \mid$
$\quad let\ (s',b) = r\ in$
$\quad (bst\ s') \wedge$
$\quad ((is\_not\_red\ s) \wedge b{=}false \rightarrow (is\_not\_red\ s')) \wedge$
$\quad ((n{:}nat)(UnbalancedRight\ n\ s) \rightarrow$
$\quad\quad (if\ b\ then\ (rbtree\ n\ s')\ else\ (rbtree\ (S\ n)\ s'))) \wedge$
$\quad ((y{:}elt)(In\_tree\ y\ s') \leftrightarrow (In\_tree\ y\ s))\ \}.$

remove_min s extracts the minimum elements of *s* and indicates whether the black height has decreased.

Definition $remove\_min$:
$(s{:}tree)(bst\ s) \rightarrow \neg s{=}Leaf \rightarrow$
$\{\ r : tree \times elt \times bool \mid$
$\quad let\ (s',r') = r\ in$
$\quad let\ (m,b) = r'\ in$
$\quad (bst\ s') \wedge$
$\quad ((is\_not\_red\ s) \wedge b{=}false \rightarrow (is\_not\_red\ s')) \wedge$
$\quad ((n{:}nat)\ (rbtree\ n\ s) \rightarrow$
$\quad\quad (if\ b\ then\ (lt\ O\ n) \wedge (rbtree\ (pred\ n)\ s')\ else\ (rbtree\ n\ s'))) \wedge$
$\quad ((y{:}elt)(In\_tree\ y\ s') \rightarrow (E.lt\ m\ y)) \wedge$
$\quad ((y{:}elt)(In\_tree\ y\ s) \leftrightarrow (E.eq\ y\ m) \vee (In\_tree\ y\ s'))\ \}.$

blackify colors the root node in *Black*

Definition $blackify$:
$(s{:}tree)(bst\ s) \rightarrow$
$\{\ r : tree \times bool \mid$

let (*s'*,*b*) = *r* *in*
(*is_not_red s'*) ∧ (*bst s'*) ∧
((*n*:*nat*)(*rbtree n s*) →
if b then (*rbtree n s'*) *else* (*rbtree* (*S n*) *s'*)) ∧
((*y*:*elt*)(*In_tree y s*) ↔ (*In_tree y s'*)) }.

remove_aux x s removes *x* from *s* and indicates whether the black height has decreased.

Definition *remove_aux* :
(*x*:*elt*)(*s*:*tree*)(*bst s*) →
{ *r* : *tree* × *bool* |
let (*s'*,*b*) = *r* *in*
(*bst s'*) ∧
((*is_not_red s*) ∧ *b*=*false* → (*is_not_red s'*)) ∧
((*n*:*nat*) (*rbtree n s*) →
(*if b then* (*lt O n*) ∧ (*rbtree* (*pred n*) *s'*) *else* (*rbtree n s'*))) ∧
((*y*:*elt*)(*In_tree y s'*) ↔ (¬(*E.eq y x*) ∧ (*In_tree y s*))) }.

remove is just a call to *remove_aux*

Definition *remove* : (*x*:*elt*)(*s*:*t*)
{ *s'*:*t* | (*y*:*elt*)(*In y s'*) ↔ (¬(*E.eq y x*) ∧ (*In y s*))}.

23 From lists to red-black trees

of_list builds a red-black tree from a sorted list

Set Implicit Arguments.
Record *of_list_aux_Invariant* [*k*,*n*:*Z*; *l*,*l'*:(*list elt*); *r*:*tree*] : *Prop* :=
make_olai
{ *olai_bst* : (*bst r*);
olai_rb : (*EX kn*:*nat* | (*inject_nat kn*)=*k* ∧ (*rbtree kn r*));
olai_sort : (*sort E.lt l'*);
olai_length : (*Zlength l'*)=(*Zlength l*)-*n*;
olai_same :
((*x*:*elt*)(*InList E.eq x l*) ↔ (*In_tree x r*) ∨ (*InList E.eq x l'*));
olai_order :
((*x*,*y*:*elt*)(*In_tree x r*) → (*InList E.eq y l'*) → (*E.lt x y*)) }.
Unset *Implicit Arguments.*

Lemma *power_invariant* :
(*n*,*k*:*Z*)0<*k* →
(*two_p k*) ≤ *n* +1 ≤ (*two_p* (*Zs k*)) →
let (*nn*,_) = (*Zsplit2* (*n*-1)) *in let* (*n1*,*n2*) = *nn in*
(*two_p* (*Zpred k*)) ≤ *n1*+1 ≤ (*two_p k*) ∧
(*two_p* (*Zpred k*)) ≤ *n2*+1 ≤ (*two_p k*).

```
Definition of_list_aux :
  (k:Z) 0≤k →
  (n:Z) (two_p k) ≤ n+1 ≤ (two_p (Zs k)) →
  (l:(list elt)) (sort E.lt l) → n ≤ (Zlength l) →
  { rl' : tree × (list elt)
  | let (r,l')=rl' in (of_list_aux_Invariant k n l l' r) }.
Definition of_list : (l:(list elt))(sort E.lt l) →
  { s : t | (x:elt)(In x s) ↔ (InList E.eq x l) }.
```

24 Elements

elements_tree_aux acc t catenates the elements of *t* in infix order to the list *acc*

```
Fixpoint elements_tree_aux [acc:(list X.t); t:tree] : (list X.t) :=
  Cases t of
  | Leaf ⇒
      acc
  | (Node _ l x r) ⇒
      (elements_tree_aux (x :: (elements_tree_aux acc r)) l)
  end.
```

then *elements_tree* is an instanciation with an empty *acc*

```
Definition elements_tree := (elements_tree_aux []).
Lemma elements_tree_aux_acc_1 :
  (s:tree)(acc:(list elt))
  (x:elt)(InList E.eq x acc)→(InList E.eq x (elements_tree_aux acc s)).
Hints Resolve elements_tree_aux_acc_1.
Lemma elements_tree_aux_1 :
  (s:tree)(acc:(list elt))
  (x:elt)(In_tree x s)→(InList E.eq x (elements_tree_aux acc s)).
Lemma elements_tree_1 :
  (s:tree)
  (x:elt)(In_tree x s)→(InList E.eq x (elements_tree s)).
Hints Resolve elements_tree_1.
Lemma elements_tree_aux_acc_2 :
  (s:tree)(acc:(list elt))
  (x:elt)(InList E.eq x (elements_tree_aux acc s)) →
    (InList E.eq x acc) ∨ (In_tree x s).
Hints Resolve elements_tree_aux_acc_2.
Lemma elements_tree_2 :
 (s:tree)
```

$(x{:}elt)(InList\ E.eq\ x\ (elements\_tree\ s)) \rightarrow (In\_tree\ x\ s)$.
Hints *Resolve elements_tree_2.*

Lemma *elements_tree_aux_sort* :
$(s{:}tree)(bst\ s) \rightarrow (acc{:}(list\ elt))$
$(sort\ E.lt\ acc) \rightarrow$
$((x{:}elt)(InList\ E.eq\ x\ acc) \rightarrow (y{:}elt)(In\_tree\ y\ s) \rightarrow (E.lt\ y\ x)) \rightarrow$
$(sort\ E.lt\ (elements\_tree\_aux\ acc\ s))$.

Lemma *elements_tree_sort* :
$(s{:}tree)(bst\ s) \rightarrow (sort\ E.lt\ (elements\_tree\ s))$.
Hints *Resolve elements_tree_sort.*

Definition *elements* :
$(s{:}t)\{\ l{:}(list\ elt)\ |\ (sort\ E.lt\ l) \wedge (x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)\}$.

25 Isomorphism with *FSetList*

Module $Lists := FSetList.Make(X)$.

Definition *of_slist* : $(l{:}Lists.t)\ \{\ s\ :\ t\ |\ (x{:}elt)(Lists.In\ x\ l){\leftrightarrow}(In\ x\ s)\ \}$.

Definition *to_slist* : $(s{:}t)\ \{\ l\ :\ Lists.t\ |\ (x{:}elt)(In\ x\ s){\leftrightarrow}(Lists.In\ x\ l)\ \}$.

26 Union

Definition *union* : $(s,s'{:}t)\{\ s''{:}t\ |\ (x{:}elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s){\vee}(In\ x\ s'))\}$.

27 Intersection

Lemma *inter* : $(s,s'{:}t)\{\ s''{:}t\ |\ (x{:}elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s){\wedge}(In\ x\ s'))\}$.

28 Difference

Lemma *diff* : $(s,s'{:}t)\{\ s''{:}t\ |\ (x{:}elt)(In\ x\ s'') \leftrightarrow ((In\ x\ s)/\backslash\tilde{}(In\ x\ s'))\}$.

29 Equality test

Set Ground Depth 5.

Lemma *equal* : $(s,s'{:}t)\{\ Equal\ s\ s'\ \} + \{\ \neg(Equal\ s\ s')\ \}$.

30 Inclusion test

```
Lemma subset : (s,s':t){ Subset s s' } + { ¬(Subset s s') }.
```

31 Fold

```
Fixpoint fold_tree [A:Set; f:elt→A→A; s:tree] : A → A :=
  Cases s of
  | Leaf ⇒ [a]a
  | (Node _ l x r) ⇒ [a](fold_tree A f l (f x (fold_tree A f r a)))
  end.
Implicits fold_tree [1].

Definition fold_tree' :=
 [A:Set; f:elt→A→A; s:tree] (Lists.Raw.fold f (elements_tree s)).
Implicits fold_tree' [1].

Lemma fold_tree_equiv_aux :
   (A:Set)(s:tree)(f:elt→A→A)(a:A; acc : (list elt))
   (Lists.Raw.fold f (elements_tree_aux acc s) a)
   = (fold_tree f s (Lists.Raw.fold f acc a)).

Lemma fold_tree_equiv :
   (A:Set)(s:tree)(f:elt→A→A; a:A)
   (fold_tree f s a)=(fold_tree' f s a).

Definition fold :
 (A:Set)(f:elt→A→A)(s:t)(i:A)
 { r : A | (EX l:(list elt) |
                  (Unique E.eq l) ∧
                  ((x:elt)(In x s)↔(InList E.eq x l)) ∧
                  r = (fold_right f i l)) }.
```

32 Cardinal

```
Definition cardinal :
   (s:t) { r : nat | (EX l:(list elt) |
                           (Unique E.eq l) ∧
                           ((x:elt)(In x s)↔(InList E.eq x l)) ∧
                           r = (length l)) }.
```

33 Filter

Module $DLists := DepOfNodep(Lists)$.

Definition $filter$: $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ s'{:}t \mid (compat\_P\ E.eq\ P) \rightarrow (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s){\wedge}(P\ x))\ \}$.

Lemma for_all : $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (For\_all\ P\ s)\ \}\ +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(For\_all\ P\ s)\ \}$.

Lemma $exists$: $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (Exists\ P\ s)\ \}\ +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(Exists\ P\ s)\ \}$.

Lemma $partition$: $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}{+}\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ partition : t{\times}t \mid$
$let\ (s1,s2) = partition\ in$
$(compat\_P\ E.eq\ P) \rightarrow$
$((For\_all\ P\ s1) \wedge$
$(For\_all\ ([x]\tilde{}(P\ x))\ s2) \wedge$
$(x{:}elt)(In\ x\ s){\leftrightarrow}((In\ x\ s1){\vee}(In\ x\ s2)))\ \}$.

34 Minimum element

Definition min_elt :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ y\ x)\ s)\ \} + \{\ Empty\ s\ \}$.

35 Maximum element

Definition max_elt :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ x\ y)\ s)\ \} + \{\ Empty\ s\ \}$.

36 Any element

Definition $choose$: $(s{:}t)\ \{\ x{:}elt \mid (In\ x\ s)\} + \{\ Empty\ s\ \}$.

37 Comparison

Definition $eq : t \rightarrow t \rightarrow Prop := Equal$.

Definition lt : $t \to t \to Prop$:=
  $[s,s':t]let\ (l,\_) = (to\_slist\ s)\ in$
    $let\ (l',\_) = (to\_slist\ s')\ in$
    $(Lists.lt\ l\ l')$.

Lemma eq_refl: $(s:t)(eq\ s\ s)$.

Lemma eq_sym: $(s,s':t)(eq\ s\ s') \to (eq\ s'\ s)$.

Lemma eq_trans: $(s,s',s'':t)(eq\ s\ s') \to (eq\ s'\ s'') \to (eq\ s\ s'')$.

Lemma lt_trans : $(s,s',s'':t)(lt\ s\ s') \to (lt\ s'\ s'') \to (lt\ s\ s'')$.

Definition eql : $t \to t \to Prop$:=
  $[s,s':t]let\ (l,\_) = (to\_slist\ s)\ in$
    $let\ (l',\_) = (to\_slist\ s')\ in$
    $(Lists.eq\ l\ l')$.

Lemma eq_eql : $(s,s':t)(eq\ s\ s') \to (eql\ s\ s')$.

Lemma eql_eq : $(s,s':t)(eql\ s\ s') \to (eq\ s\ s')$.

Lemma lt_not_eq : $(s,s':t)(lt\ s\ s') \to \neg(eq\ s\ s')$.

Definition $compare$: $(s,s':t)(Compare\ lt\ eq\ s\ s')$.

End $Make$.

Module FSetAVL

This module implements sets using AVL trees. It follows the implementation from Ocaml's standard library.

Require $FSetInterface$.
Require $FSetList$.

Require $ZArith$.
Import Z_scope.
$Open\ Scope\ Z\_scope$.

Notation "[]" := $(nil\ ?)$ $(at\ level\ 1)$.
Notation "a :: l" := $(cons\ a\ l)$ $(at\ level\ 1,\ l\ at\ next\ level)$.

$Set\ Ground\ Depth\ 3$.

Module $Make$ $[X : OrderedType]$ <: $Sdep\ with\ Module\ E := X$.

Module E := X.
Module ME := $MoreOrderedType\ X$.

Definition elt := $X.t$.

38 Trees

Inductive *tree* : *Set* :=
| *Leaf* : *tree*
| *Node* : *tree* → *X.t* → *tree* → *Z* → *tree.*

The fourth field of *Node* is the height of the tree

39 Occurrence in a tree

Inductive *In_tree* [*x*:*elt*] : *tree* → *Prop* :=
| *IsRoot* : (*l*,*r*:*tree*)(*h*:*Z*)(*y*:*elt*)
    (*X.eq x y*) → (*In_tree x* (*Node l y r h*))
| *InLeft* : (*l*,*r*:*tree*)(*h*:*Z*)(*y*:*elt*)
    (*In_tree x l*) → (*In_tree x* (*Node l y r h*))
| *InRight* : (*l*,*r*:*tree*)(*h*:*Z*)(*y*:*elt*)
    (*In_tree x r*) → (*In_tree x* (*Node l y r h*)).

Hint *In_tree* := *Constructors In_tree.*

In_tree is compatible with *X.eq*

Lemma *eq_In_tree*: (*s*:*tree*)(*x*,*y*:*elt*)(*E.eq x y*) → (*In_tree x s*) → (*In_tree y s*).

In_tree is height-insensitive

Lemma *In_height* : (*h*,*h'*:*Z*)(*x*,*y*:*elt*)(*l*,*r*:*tree*)
  (*In_tree y* (*Node l x r h*)) → (*In_tree y* (*Node l x r h'*)).
Hints *Resolve In_height.*

40 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_tree y s*) → (*X.lt y x*).
Definition *gt_tree* [*x*:*elt*; *s*:*tree*] := (*y*:*elt*)(*In_tree y s*) → (*X.lt x y*).

Hints *Unfold lt_tree gt_tree.*

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*:*Z*)
  (*lt_tree x l*) → (*lt_tree x r*) → (*X.lt y x*) →
  (*lt_tree x* (*Node l y r h*)).

Lemma gt_tree_node : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(gt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to (E.lt\ x\ y) \to$
$(gt\_tree\ x\ (Node\ l\ y\ r\ h))$.

Hints *Resolve lt_leaf gt_leaf lt_tree_node gt_tree_node.*

Lemma lt_node_lt : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(lt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (E.lt\ y\ x)$.

Lemma gt_node_gt : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(gt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (E.lt\ x\ y)$.

Lemma lt_left : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(lt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (lt\_tree\ x\ l)$.

Lemma lt_right : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(lt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (lt\_tree\ x\ r)$.

Lemma gt_left : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(gt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (gt\_tree\ x\ l)$.

Lemma gt_right : $(x,y{:}elt)(l,r{:}tree)(h{:}Z)$
$(gt\_tree\ x\ (Node\ l\ y\ r\ h)) \to (gt\_tree\ x\ r)$.

Hints *Resolve lt_node_lt gt_node_gt*
lt_left lt_right gt_left gt_right.

Lemma $lt\_tree\_not\_in$:
$(x{:}elt)(t{:}tree)(lt\_tree\ x\ t) \to \neg(In\_tree\ x\ t)$.

Lemma lt_tree_trans :
$(x,y{:}elt)(X.lt\ x\ y) \to (t{:}tree)(lt\_tree\ x\ t) \to (lt\_tree\ y\ t)$.

Lemma $gt\_tree\_not\_in$:
$(x{:}elt)(t{:}tree)(gt\_tree\ x\ t) \to \neg(In\_tree\ x\ t)$.

Lemma gt_tree_trans :
$(x,y{:}elt)(X.lt\ y\ x) \to (t{:}tree)(gt\_tree\ x\ t) \to (gt\_tree\ y\ t)$.

Hints *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 : $tree \to Prop :=$
| $BSLeaf$:
$(bst\ Leaf)$
| $BSNode$: $(x{:}elt)(l,r{:}tree)(h{:}Z)$
$(bst\ l) \to (bst\ r) \to$
$(lt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to$
$(bst\ (Node\ l\ x\ r\ h))$.

Hint bst := *Constructors bst.*

Results about bst

Lemma *bst_left* : $(x{:}elt)(l{,}r{:}tree)(h{:}Z)$
$(bst\ (Node\ l\ x\ r\ h)) \rightarrow (bst\ l)$.

Lemma *bst_right* : $(x{:}elt)(l{,}r{:}tree)(h{:}Z)$
$(bst\ (Node\ l\ x\ r\ h)) \rightarrow (bst\ r)$.

Implicits *bst_left*. Implicits *bst_right*.
Hints *Resolve bst_left bst_right*.

Lemma *bst_height* : $(h{,}h'{:}Z)(x{:}elt)(l{,}r{:}tree)$
$(bst\ (Node\ l\ x\ r\ h)) \rightarrow (bst\ (Node\ l\ x\ r\ h'))$.
Hints *Resolve bst_height*.

Key fact about binary search trees: rotations preserve the *bst* property

Lemma *rotate_left* : $(x{,}y{:}elt)(a{,}b{,}c{:}tree)(h1{,}h2{,}h3{,}h4{:}Z)$
$(bst\ (Node\ a\ x\ (Node\ b\ y\ c\ h2)\ h1)) \rightarrow$
$(bst\ (Node\ (Node\ a\ x\ b\ h4)\ y\ c\ h3))$.

Lemma *rotate_right* : $(x{,}y{:}elt)(a{,}b{,}c{:}tree)(h1{,}h2{,}h3{,}h4{:}Z)$
$(bst\ (Node\ (Node\ a\ x\ b\ h4)\ y\ c\ h3)) \rightarrow$
$(bst\ (Node\ a\ x\ (Node\ b\ y\ c\ h2)\ h1))$.

Hints *Resolve rotate_left rotate_right*.

41 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* : $tree \rightarrow Z$:=
$[s{:}tree]$ *Cases s of*
| *Leaf* $\Rightarrow 0$
| $(Node\ \_\ \_\ \_\ h) \Rightarrow h$ *end*.

Definition *max* $[x{,}y{:}Z] : Z$:=
if $(Z\_lt\_ge\_dec\ x\ y)$ *then* $[\_]y$ *else* $[\_]x$.

Instead of writing $h = 1 + (max\ (height\ l)\ (height\ r))$ we prefer the following relation *height_of_node l r h* to ease the use of *Omega*

Definition *height_of_node* $[l{,}r{:}tree;\ h{:}Z]$:=
$((height\ l) \geq (height\ r) \wedge h = (height\ l) + 1) \vee$
$((height\ r) \geq (height\ l) \wedge h = (height\ r) + 1)$.

Inductive *avl* : $tree \rightarrow Prop$:=
| *RBLeaf* :
(avl Leaf)
| *RBNode* : $(x{:}elt)(l{,}r{:}tree)(h{:}Z)$

$(avl\ l) \rightarrow (avl\ r) \rightarrow$
‘-2 $\leq$ $(height\ l)$ - $(height\ r)$ $\leq$ 2‘ $\rightarrow$
$(height\_of\_node\ l\ r\ h) \rightarrow$
$(avl\ (Node\ l\ x\ r\ h))$.

Hint *avl* := *Constructors avl.*

Results about *avl*

Lemma *avl_left* :
$(x{:}elt)(l{,}r{:}tree)(h{:}Z)$
$(avl\ (Node\ l\ x\ r\ h)) \rightarrow (avl\ l)$.

Lemma *avl_right* :
$(x{:}elt)(l{,}r{:}tree)(h{:}Z)$
$(avl\ (Node\ l\ x\ r\ h)) \rightarrow (avl\ l)$.

Implicits *avl_left.* Implicits *avl_right.*
Hints *Resolve avl_left avl_right.*

Tactic Definition *MaxCase x y* :=
Unfold max; *Case* (*Z_lt_ge_dec x y*); *Simpl.*

Lemma *avl_node*: $(x{:}elt)(l{,}r{:}tree)$
$(avl\ l) \rightarrow (avl\ r) \rightarrow$
‘-2 $\leq$ $(height\ l)$ - $(height\ r)$ $\leq$ 2‘ $\rightarrow$
$(avl\ (Node\ l\ x\ r\ ((max\ (height\ l)\ (height\ r)) + 1)))$.
Hints *Resolve avl_node.*

The *AVL* tactic

Lemma *height_non_negative* :
$(s{:}tree)(avl\ s) \rightarrow (height\ s) \geq 0$.

Lemma *height_equation* :
$(l{,}r{:}tree)(x{:}elt)(h{:}Z)$
$(avl\ (Node\ l\ x\ r\ h)) \rightarrow$
‘-2 $\leq$ $(height\ l)$ - $(height\ r)$ $\leq$ 2‘ $\wedge$
$(((height\ l) \geq (height\ r) \wedge h = (height\ l) + 1) \vee$
$((height\ r) \geq (height\ l) \wedge h = (height\ r) + 1))$.

Implicits *height_non_negative.*
Implicits *height_equation.*

If h is a proof of $avl\ (Node\ l\ x\ r\ h)$, then tactic *AVL* h is adding all information about h to the context

Tactic Definition *AVL h* :=
(*Generalize* (*height_non_negative h*); *Try Simpl*);
(*Try Generalize* (*height_equation h*)); *Intros.*

42 Sets as AVL trees

A set is implement as a record t, containing a tree, a proof that it is a binary search tree and a proof that it is a properly balanced AVL tree

Record $t\_$: Set := t_intro {
  the_tree :> $tree$;
  is_bst : $(bst\ the\_tree)$;
  is_avl : $(avl\ the\_tree)$ }.
Definition t := $t\_$.

Projections

Lemma t_is_bst : $(s{:}t)(bst\ s)$.
Hints $Resolve\ t\_is\_bst$.

Lemma t_is_avl : $(s{:}t)(avl\ s)$.
Hints $Resolve\ t\_is\_avl$.

43 Logical appartness

Definition In : $elt \to t \to Prop$:= $[x{:}elt][s{:}t](In\_tree\ x\ s)$.

Definition $Equal$:= $[s,s'](a{:}elt)(In\ a\ s)\leftrightarrow(In\ a\ s')$.
Definition $Subset$:= $[s,s'](a{:}elt)(In\ a\ s)\to(In\ a\ s')$.
Definition Add := $[x{:}elt;s,s'{:}t](y{:}elt)(In\ y\ s') \leftrightarrow ((E.eq\ y\ x)\vee(In\ y\ s))$.
Definition $Empty$:= $[s](a{:}elt)\neg(In\ a\ s)$.
Definition For_all := $[P{:}elt\to Prop;\ s{:}t](x{:}elt)(In\ x\ s)\to(P\ x)$.
Definition $Exists$:= $[P{:}elt\to Prop;\ s{:}t](EX\ x{:}elt\ |\ (In\ x\ s)\wedge(P\ x))$.

Lemma eq_In: $(s{:}t)(x,y{:}elt)(E.eq\ x\ y) \to (In\ x\ s) \to (In\ y\ s)$.

Hints $Resolve\ eq\_In$.

44 Empty set

Definition t_empty : t.

Definition $empty$: $\{\ s{:}t\ |\ (x{:}elt)\neg(In\ x\ s)\ \}$.

45 Emptyness test

Definition is_empty : $(s{:}t)\{\ Empty\ s\ \}+\{\ \neg(Empty\ s)\ \}$.

46 Appartness

The *mem* function is deciding appartness. It exploits the *bst* property to achieve logarithmic complexity.

Definition $mem : (x{:}elt)\ (s{:}t)\ \{\ (In\ x\ s)\ \} + \{\ \neg(In\ x\ s)\ \}.$

47 Singleton set

Definition $singleton\_tree\ [x{:}elt] := (Node\ Leaf\ x\ Leaf\ 1).$

Lemma $singleton\_bst : (x{:}elt)(bst\ (singleton\_tree\ x)).$

Lemma $singleton\_avl : (x{:}elt)(avl\ (singleton\_tree\ x)).$

Definition $singleton : (x{:}elt)\{\ s{:}t\ |\ (y{:}elt)(In\ y\ s) \leftrightarrow (E.eq\ x\ y)\}.$

48 Helper functions

create l x r creates a node, assuming *l* and *r* to be balanced and $|height\ l\ \text{-}\ height\ r| \leq 2$.

Definition $create$:
$(l{:}tree)(x{:}elt)(r{:}tree)$
$(bst\ l) \to (avl\ l) \to (bst\ r) \to (avl\ r) \to$
$(lt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to$
‘$-2 \leq (height\ l)\ \text{-}\ (height\ r) \leq 2$‘ $\to$
$\{\ s{:}tree\ |$
$(bst\ s) \land$
$(avl\ s) \land$
$(height\_of\_node\ l\ r\ (height\ s)) \land$
$(y{:}elt)(In\_tree\ y\ s) \leftrightarrow$
$((X.eq\ y\ x) \lor (In\_tree\ y\ l) \lor (In\_tree\ y\ r))\ \}.$

bal l x r acts as *create*, but performs one step of rebalancing if necessary, i.e. assumes $|height\ l\ \text{-}\ height\ r| \leq 3$.

Definition bal :
$(l{:}tree)(x{:}elt)(r{:}tree)$
$(bst\ l) \to (avl\ l) \to (bst\ r) \to (avl\ r) \to$
$(lt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to$
‘$-3 \leq (height\ l)\ \text{-}\ (height\ r) \leq 3$‘ $\to$
$\{\ s{:}tree\ |$
$(bst\ s) \land (avl\ s) \land$
$(((height\_of\_node\ l\ r\ (height\ s)) \lor$
$(height\_of\_node\ l\ r\ ((height\ s) + 1))) \land$
(‘$-2 \leq (height\ l)\ \text{-}\ (height\ r) \leq 2$‘ $\to$

$(height\_of\_node\ l\ r\ (height\ s)))) \wedge$
$(y{:}elt)(In\_tree\ y\ s) \leftrightarrow$
$((X.eq\ y\ x) \vee (In\_tree\ y\ l) \vee (In\_tree\ y\ r))\ \}.$

49 Insertion

Definition add_tree :
$(x{:}elt)(s{:}tree)(bst\ s) \to (avl\ s) \to$
$\{\ s'{:}tree \mid (bst\ s') \wedge (avl\ s') \wedge$
$0 \leq (height\ s')\text{-}(height\ s) \leq 1 \wedge$
$((y{:}elt)(In\_tree\ y\ s') \leftrightarrow ((E.eq\ y\ x) \vee (In\_tree\ y\ s)))\ \}.$

Definition add : $(x{:}elt)\ (s{:}t)\ \{\ s'{:}t \mid (\mathsf{Add}\ x\ s\ s')\ \}.$

50 Join

Same as *bal* but does not assumme anything regarding heights of l and r. Code is

```
let rec join l v r =
  match (l, r) with
    (Empty, _) -> add v r
  | (_, Empty) -> add v l
  | (Node(ll, lv, lr, lh), Node(rl, rv, rr, rh)) ->
      if lh > rh + 2 then bal ll lv (join lr v r) else
      if rh > lh + 2 then bal (join l v rl) rv rr else
      create l v r
```

Definition $join$:
$(l{:}tree)(x{:}elt)(r{:}tree)$
$(bst\ l) \to (avl\ l) \to (bst\ r) \to (avl\ r) \to$
$(lt\_tree\ x\ l) \to (gt\_tree\ x\ r) \to$
$\{\ s{:}tree \mid (bst\ s) \wedge (avl\ s) \wedge$
$((height\_of\_node\ l\ r\ (height\ s)) \vee$
$(height\_of\_node\ l\ r\ ((height\ s)+1))) \wedge$
$((y{:}elt)(In\_tree\ y\ s) \leftrightarrow$
$((X.eq\ y\ x) \vee (In\_tree\ y\ l) \vee (In\_tree\ y\ r)))\ \}.$

51 Extraction of minimum and maximum element

Definition $remove\_min$:
$(s{:}tree)(bst\ s) \to (avl\ s) \to \neg s{=}Leaf \to$

{ r : $tree \times elt$ |
  let (s',m) = r in
  (bst s') $\wedge$ (avl s') $\wedge$
  (($height$ s') = ($height$ s) $\vee$ ($height$ s')=($height$ s)-1) $\wedge$
  ((y:elt)(In_tree y s') $\rightarrow$ ($E.lt$ m y)) $\wedge$
  ((y:elt)(In_tree y s) $\leftrightarrow$ ($E.eq$ y m) $\vee$ (In_tree y s')) }.

Definition *remove_max* :
  (s:$tree$)(bst s) $\rightarrow$ (avl s) $\rightarrow$ $\neg s$=$Leaf$ $\rightarrow$
  { r : $tree \times elt$ |
  let (s',m) = r in
  (bst s') $\wedge$ (avl s') $\wedge$
  (($height$ s') = ($height$ s) $\vee$ ($height$ s')=($height$ s)-1) $\wedge$
  ((y:elt)(In_tree y s') $\rightarrow$ ($E.lt$ y m)) $\wedge$
  ((y:elt)(In_tree y s) $\leftrightarrow$ ($E.eq$ y m) $\vee$ (In_tree y s')) }.

52 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. Code is

```
let merge t1 t2 =
  match (t1, t2) with
    (Empty, t) -> t
  | (t, Empty) -> t
  | (_, _) -> let (m,t'2) = remove_min t2 in bal t1 m t'2
```

Definition *merge* :
  ($s1$,$s2$:$tree$)(bst $s1$) $\rightarrow$ (avl $s1$) $\rightarrow$ (bst $s2$) $\rightarrow$ (avl $s2$) $\rightarrow$
  (($y1$,$y2$:elt)(In_tree $y1$ $s1$) $\rightarrow$ (In_tree $y2$ $s2$) $\rightarrow$ ($X.lt$ $y1$ $y2$)) $\rightarrow$
  ‘-2 $\leq$ ($height$ $s1$) - ($height$ $s2$) $\leq$ 2‘ $\rightarrow$
  { s:$tree$ | (bst s) $\wedge$ (avl s) $\wedge$
    (($height\_of\_node$ $s1$ $s2$ ($height$ s)) $\vee$
     ($height\_of\_node$ $s1$ $s2$ (($height$ s)+1))) $\wedge$
    ((y:elt)(In_tree y s) $\leftrightarrow$
      ((In_tree y $s1$) $\vee$ (In_tree y $s2$))) }.

Variant where we remove from the biggest subtree

Definition *merge_bis* :
  ($s1$,$s2$:$tree$)(bst $s1$) $\rightarrow$ (avl $s1$) $\rightarrow$ (bst $s2$) $\rightarrow$ (avl $s2$) $\rightarrow$
  (($y1$,$y2$:elt)(In_tree $y1$ $s1$) $\rightarrow$ (In_tree $y2$ $s2$) $\rightarrow$ ($X.lt$ $y1$ $y2$)) $\rightarrow$
  ‘-2 $\leq$ ($height$ $s1$) - ($height$ $s2$) $\leq$ 2‘ $\rightarrow$
  { s:$tree$ | (bst s) $\wedge$ (avl s) $\wedge$
    (($height\_of\_node$ $s1$ $s2$ ($height$ s)) $\vee$

$(height\_of\_node\ s1\ s2\ ((height\ s){+}1)))\ \wedge$
$((y{:}elt)(In\_tree\ y\ s) \leftrightarrow$
$((In\_tree\ y\ s1) \vee (In\_tree\ y\ s2)))\ \}.$

53 Deletion

Definition *remove_tree* :
$(x{:}elt)(s{:}tree)(bst\ s) \rightarrow (avl\ s) \rightarrow$
$\{\ s'{:}tree \mid (bst\ s') \wedge (avl\ s') \wedge$
$((height\ s') = (height\ s) \vee (height\ s') = (height\ s)$ - 1$) \wedge$
$((y{:}elt)(In\_tree\ y\ s') \leftrightarrow (\neg(E.eq\ y\ x) \wedge (In\_tree\ y\ s)))\ \}.$

Definition *remove* : $(x{:}elt)(s{:}t)$
$\{\ s'{:}t \mid (y{:}elt)(In\ y\ s') \leftrightarrow (\neg(E.eq\ y\ x) \wedge (In\ y\ s))\}.$

54 Minimum element

Definition *min_elt* :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ y\ x)\ s)\ \} + \{\ Empty\ s\ \}.$

55 Maximum element

Definition *max_elt* :
$(s{:}t)\{\ x{:}elt \mid (In\ x\ s) \wedge (For\_all\ [y]\tilde{}(E.lt\ x\ y)\ s)\ \} + \{\ Empty\ s\ \}.$

56 Any element

Definition *choose* : $(s{:}t)\ \{\ x{:}elt \mid (In\ x\ s)\} + \{\ Empty\ s\ \}.$

57 Concatenation

Same as *merge* but does not assume anything about heights. Code is

```
let concat t1 t2 =
  match (t1, t2) with
    (Empty, t) -> t
  | (t, Empty) -> t
  | (_, _) -> join t1 (min_elt t2) (remove_min_elt t2)
```

Definition *concat* :
$(s1,s2{:}tree)(bst\ s1) \to (avl\ s1) \to (bst\ s2) \to (avl\ s2) \to$
$((y1,y2{:}elt)(In\_tree\ y1\ s1) \to (In\_tree\ y2\ s2) \to (X.lt\ y1\ y2)) \to$
$\{\ s{:}tree \mid (bst\ s) \wedge (avl\ s) \wedge$
$((y{:}elt)(In\_tree\ y\ s) \leftrightarrow$
$((In\_tree\ y\ s1) \vee (In\_tree\ y\ s2)))\ \}.$

58 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.

```
let rec split x = function
    Empty ->
      (Empty, false, Empty)
  | Node(l, v, r, _) ->
      let c = Ord.compare x v in
      if c = 0 then (l, true, r)
      else if c < 0 then
        let (ll, pres, rl) = split x l in (ll, pres, join rl v r)
      else
        let (lr, pres, rr) = split x r in (join l v lr, pres, rr)
```

Definition *split* :
$(x{:}elt)(s{:}tree)(bst\ s) \to (avl\ s) \to$
$\{\ res{:}tree \times bool \times tree \mid$
let (l,res') = *res in*
let (b,r) = *res' in*
$(bst\ l) \wedge (avl\ l) \wedge (bst\ r) \wedge (avl\ r) \wedge$
$((y{:}elt)(In\_tree\ y\ l) \leftrightarrow ((In\_tree\ y\ s) \wedge (X.lt\ y\ x))) \wedge$
$((y{:}elt)(In\_tree\ y\ r) \leftrightarrow ((In\_tree\ y\ s) \wedge (X.lt\ x\ y))) \wedge$
$(b{=}true \leftrightarrow (In\_tree\ x\ s))\ \}.$

59 Intersection

```
let rec inter s1 s2 =
  match (s1, s2) with
    (Empty, t2) -> Empty
  | (t1, Empty) -> Empty
  | (Node(l1, v1, r1, _), t2) ->
      match split v1 t2 with
        (l2, false, r2) ->
          concat (inter l1 l2) (inter r1 r2)
      | (l2, true, r2) ->
          join (inter l1 l2) v1 (inter r1 r2)
```

Definition *inter* :
  $(s1,s2{:}t)$
  $\{\ s'{:}t \mid (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s1) \wedge (In\ x\ s2))\}$.

60 Difference

```
 let rec diff s1 s2 =
  match (s1, s2) with
    (Empty, t2) -> Empty
  | (t1, Empty) -> t1
  | (Node(l1, v1, r1, _), t2) ->
      match split v1 t2 with
        (l2, false, r2) ->
          join (diff l1 l2) v1 (diff r1 r2)
      | (l2, true, r2) ->
          concat (diff l1 l2) (diff r1 r2)
```

Definition *diff* :
  $(s1,s2{:}t)$
  $\{\ s'{:}t \mid (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s1) \wedge \neg(In\ x\ s2))\}$.

61 Elements

elements_tree_aux acc t catenates the elements of t in infix order to the list *acc*

Fixpoint *elements_tree_aux* [*acc*:(*list X.t*); *t*:*tree*] : (*list X.t*) :=
  Cases t of
  | *Leaf* ⇒

acc
$\mid$ $(Node\ l\ x\ r\ \_) \Rightarrow$
$(elements\_tree\_aux\ (x :: (elements\_tree\_aux\ acc\ r))\ l)$
$end.$

then $elements\_tree$ is an instanciation with an empty acc

Definition $elements\_tree := (elements\_tree\_aux\ [])$.

Lemma $elements\_tree\_aux\_acc\_1$:
$(s{:}tree)(acc{:}(list\ elt))$
$(x{:}elt)(InList\ E.eq\ x\ acc) \rightarrow (InList\ E.eq\ x\ (elements\_tree\_aux\ acc\ s))$.
Hints $Resolve\ elements\_tree\_aux\_acc\_1$.

Lemma $elements\_tree\_aux\_1$:
$(s{:}tree)(acc{:}(list\ elt))$
$(x{:}elt)(In\_tree\ x\ s) \rightarrow (InList\ E.eq\ x\ (elements\_tree\_aux\ acc\ s))$.

Lemma $elements\_tree\_1$:
$(s{:}tree)$
$(x{:}elt)(In\_tree\ x\ s) \rightarrow (InList\ E.eq\ x\ (elements\_tree\ s))$.
Hints $Resolve\ elements\_tree\_1$.

Lemma $elements\_tree\_aux\_acc\_2$:
$(s{:}tree)(acc{:}(list\ elt))$
$(x{:}elt)(InList\ E.eq\ x\ (elements\_tree\_aux\ acc\ s)) \rightarrow$
$(InList\ E.eq\ x\ acc) \vee (In\_tree\ x\ s)$.
Hints $Resolve\ elements\_tree\_aux\_acc\_2$.

Lemma $elements\_tree\_2$:
$(s{:}tree)$
$(x{:}elt)(InList\ E.eq\ x\ (elements\_tree\ s)) \rightarrow (In\_tree\ x\ s)$.
Hints $Resolve\ elements\_tree\_2$.

Lemma $elements\_tree\_aux\_sort$:
$(s{:}tree)(bst\ s) \rightarrow (acc{:}(list\ elt))$
$(sort\ E.lt\ acc) \rightarrow$
$((x{:}elt)(InList\ E.eq\ x\ acc) \rightarrow (y{:}elt)(In\_tree\ y\ s) \rightarrow (E.lt\ y\ x)) \rightarrow$
$(sort\ E.lt\ (elements\_tree\_aux\ acc\ s))$.

Lemma $elements\_tree\_sort$:
$(s{:}tree)(bst\ s) \rightarrow (sort\ E.lt\ (elements\_tree\ s))$.
Hints $Resolve\ elements\_tree\_sort$.

Definition $elements$:
$(s{:}t)\{\ l{:}(list\ elt) \mid (sort\ E.lt\ l) \wedge (x{:}elt)(In\ x\ s) \leftrightarrow (InList\ E.eq\ x\ l)\}$.

62 Cardinal

Fixpoint $cardinal\_tree\ [s{:}tree] : nat := Cases\ s\ of$

| *Leaf* $\Rightarrow$ *O*
| (*Node l _ r _*) $\Rightarrow$ (*S* (*plus* (*cardinal_tree l*) (*cardinal_tree r*))) *end.*

Lemma *cardinal_elements_1* :
(*s*:*tree*)(*acc*:(*list X.t*))
(*plus* (*length acc*) (*cardinal_tree s*)) =
(*length* (*elements_tree_aux acc s*)).

Lemma *cardinal_elements_2* :
(*s*:*tree*)(*cardinal_tree s*)=(*length* (*elements_tree s*)).

Definition *cardinal* :
(*s*:*t*) { *r* : *nat* | (*EX l*:(*list elt*) |
(*Unique E.eq l*) $\wedge$
((*x*:*elt*)(*In x s*)$\leftrightarrow$(*InList E.eq x l*)) $\wedge$
r = (*length l*)) }.

Induction over cardinals

Lemma *sorted_subset_cardinal* :
(*l'*,*l*:(*list X.t*))(*sort X.lt l*) $\rightarrow$ (*sort X.lt l'*) $\rightarrow$
((*x*:*elt*)(*InList X.eq x l*) $\rightarrow$ (*InList X.eq x l'*)) $\rightarrow$
(*le* (*length l*) (*length l'*)).

Lemma *cardinal_subset* :
(*a*,*b*:*tree*)(*bst a*) $\rightarrow$ (*bst b*) $\rightarrow$
((*y*:*elt*)(*In_tree y a*) $\rightarrow$ (*In_tree y b*)) $\rightarrow$
(*le* (*cardinal_tree a*) (*cardinal_tree b*)).

Lemma *cardinal_left* :
(*l*,*r*:*tree*)(*x*:*elt*)(*h*:*Z*)
(*lt* (*cardinal_tree l*) (*cardinal_tree* (*Node l x r h*))).

Lemma *cardinal_right* :
(*l*,*r*:*tree*)(*x*:*elt*)(*h*:*Z*)
(*lt* (*cardinal_tree r*) (*cardinal_tree* (*Node l x r h*))).

Lemma *cardinal_rec2* :
(*P*:*tree*$\rightarrow$*tree*$\rightarrow$*Set*)
((*x*,*x'*:*tree*)
((*y*,*y'*:*tree*)(*lt* (*plus* (*cardinal_tree y*) (*cardinal_tree y'*))
(*plus* (*cardinal_tree x*) (*cardinal_tree x'*))) $\rightarrow$(*P y y'*))
$\rightarrow$ (*P x x'*)) $\rightarrow$
(*x*,*x'*:*tree*)(*P x x'*).

Lemma *height_0* : (*s*:*tree*)(*avl s*) $\rightarrow$ (*height s*)=0 $\rightarrow$ (*x*:*elt*)$\neg$(*In_tree x s*).
Implicits *height_0*.

63 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* : $(s1,s2{:}t)$
$\{\ s'{:}t \mid (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s1) \vee (In\ x\ s2))\}$.

64 Filter

```
let filter p s =
  let rec filt accu = function
    | Empty -> accu
    | Node(l, v, r, _) ->
        filt (filt (if p v then add v accu else accu) l) r in
  filt Empty s
```

Definition *filter_acc* :
$(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})$
$(s,acc{:}tree)(bst\ s) \rightarrow (avl\ s) \rightarrow (bst\ acc) \rightarrow (avl\ acc) \rightarrow$
$\{\ s'{:}tree \mid$
$(bst\ s') \wedge (avl\ s') \wedge$
$((compat\_P\ E.eq\ P) \rightarrow$
$(x{:}elt)(In\_tree\ x\ s') \leftrightarrow$
$((In\_tree\ x\ acc) \vee ((In\_tree\ x\ s) \wedge (P\ x))))\ \}$.

Definition *filter* : $(P{:}elt \rightarrow Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ s'{:}t \mid (compat\_P\ E.eq\ P) \rightarrow (x{:}elt)(In\ x\ s') \leftrightarrow ((In\ x\ s) \wedge (P\ x))\ \}$.

65 Partition

```
let partition p s =
  let rec part (t, f as accu) = function
    | Empty -> accu
    | Node(l, v, r, _) ->
        part (part (if p v then (add v t, f) else (t, add v f)) l) r in
  part (Empty, Empty) s
```

Definition *partition_acc* :

$(P{:}elt{\to}Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})$
$(s{,}acct{,}accf{:}tree)$
$(bst\ s) \to (avl\ s) \to$
$(bst\ acct) \to (avl\ acct) \to (bst\ accf) \to (avl\ accf) \to$
$\{\ partition\ :\ tree{\times}tree\ |$
$\quad let\ (s1{,}s2) = partition\ in$
$\quad (bst\ s1) \land (avl\ s1) \land (bst\ s2) \land (avl\ s2) \land$
$\quad ((compat\_P\ E.eq\ P) \to$
$\quad\ (x{:}elt)(((In\_tree\ x\ s1) \leftrightarrow$
$\qquad\qquad ((In\_tree\ x\ acct) \lor ((In\_tree\ x\ s) \land (P\ x)))) \land$
$\qquad\quad (((In\_tree\ x\ s2) \leftrightarrow$
$\qquad\qquad ((In\_tree\ x\ accf) \lor ((In\_tree\ x\ s) \land \neg(P\ x))))))\ \}.$

Definition *partition* :

$(P{:}elt{\to}Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ partition\ :\ t{\times}t\ |$
$\quad let\ (s1{,}s2) = partition\ in$
$\quad (compat\_P\ E.eq\ P) \to$
$\quad ((For\_all\ P\ s1) \land$
$\quad\ (For\_all\ ([x]\tilde{}(P\ x))\ s2) \land$
$\quad\ (x{:}elt)(In\ x\ s){\leftrightarrow}((In\ x\ s1){\lor}(In\ x\ s2)))\ \}.$

66 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* :
$(s1,s2{:}t)\{\ Subset\ s1\ s2\ \} + \{\ \neg(Subset\ s1\ s2)\ \}$.

for_all and *exists*

Definition *for_all* : $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (For\_all\ P\ s)\ \} +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(For\_all\ P\ s)\ \}$.

Definition *exists* : $(P{:}elt{\rightarrow}Prop)(Pdec{:}(x{:}elt)\{P\ x\}+\{\tilde{}(P\ x)\})(s{:}t)$
$\{\ (compat\_P\ E.eq\ P) \rightarrow (Exists\ P\ s)\ \} +$
$\{\ (compat\_P\ E.eq\ P) \rightarrow \neg(Exists\ P\ s)\ \}$.

67 Fold

Module $L := Raw(E)$.

Fixpoint *fold_tree* $[A{:}Set;\ f{:}elt{\rightarrow}A{\rightarrow}A;\ s{:}tree] : A \rightarrow A :=$
$Cases\ s\ of$
$\mid Leaf \Rightarrow [a]a$
$\mid (Node\ l\ x\ r\ \_) \Rightarrow [a](fold\_tree\ A\ f\ l\ (f\ x\ (fold\_tree\ A\ f\ r\ a)))$
$end.$
Implicits *fold_tree* [1].

Definition *fold_tree'* :=
$[A{:}Set;\ f{:}elt{\rightarrow}A{\rightarrow}A;\ s{:}tree]\ (L.fold\ f\ (elements\_tree\ s))$.
Implicits *fold_tree'* [1].

Lemma *fold_tree_equiv_aux* :
$(A{:}Set)(s{:}tree)(f{:}elt{\rightarrow}A{\rightarrow}A)(a{:}A;\ acc : (list\ elt))$
$(L.fold\ f\ (elements\_tree\_aux\ acc\ s)\ a)$
$= (fold\_tree\ f\ s\ (L.fold\ f\ acc\ a))$.

Lemma *fold_tree_equiv* :
$(A{:}Set)(s{:}tree)(f{:}elt{\rightarrow}A{\rightarrow}A;\ a{:}A)$
$(fold\_tree\ f\ s\ a)=(fold\_tree'\ f\ s\ a)$.

Definition *fold* :
$(A{:}Set)(f{:}elt{\rightarrow}A{\rightarrow}A)(s{:}t)(i{:}A)$
$\{\ r : A \mid (EX\ l{:}(list\ elt) \mid$
$(Unique\ E.eq\ l) \land$
$((x{:}elt)(In\ x\ s){\leftrightarrow}(InList\ E.eq\ x\ l)) \land$
$r = (fold\_right\ f\ i\ l))\ \}$.

68 Comparison

68.1 Relations *eq* and *lt* over trees

Definition *eq* : $t \rightarrow t \rightarrow Prop$:= *Equal*.

Lemma *eq_refl*: (*s*:*t*)(*eq s s*).

Lemma *eq_sym*: (*s*,*s'*:*t*)(*eq s s'*) $\rightarrow$ (*eq s' s*).

Lemma *eq_trans*: (*s*,*s'*,*s"*:*t*)(*eq s s'*) $\rightarrow$ (*eq s' s"*) $\rightarrow$ (*eq s s"*).

Lemma *eq_L_eq* : (*s*,*s'*:*t*)
(*eq s s'*) $\rightarrow$ (*L.eq* (*elements_tree s*) (*elements_tree s'*)).

Lemma *L_eq_eq* : (*s*,*s'*:*t*)
(*L.eq* (*elements_tree s*) (*elements_tree s'*)) $\rightarrow$ (*eq s s'*).
Hints *Resolve eq_L_eq L_eq_eq*.

Definition *lt* : $t \rightarrow t \rightarrow Prop$:=
[*s1*,*s2*:*t*](*L.lt* (*elements_tree s1*) (*elements_tree s2*)).

Definition *lt_trans* : (*s*,*s'*,*s"*:*t*)(*lt s s'*) $\rightarrow$ (*lt s' s"*) $\rightarrow$ (*lt s s"*)
:= [*s*,*s'*,*s"*:*t*][*h*][*h'*](*L.lt_trans h h'*).

Lemma *lt_not_eq* : (*s*,*s'*:*t*)(*lt s s'*) $\rightarrow$ $\neg$(*eq s s'*).

68.2 Comparison algorithm

The code for *compare s1 s2* is basically comparing *elements s1* and *elements s2* in lexicographic order. But it builds these lists lazily (it is doing deforestation). It uses a function *compare_aux* comparing two lists of trees.

We slightly modify the original code to get a simplest termination argument. The resulting code is even a bit more efficient.

```
let rec compare_aux l1 l2 =
    match (l1, l2) with
    ([], []) -> 0
  | ([], _)  -> -1
  | (_, []) -> 1
  | (Empty :: t1, Empty :: t2) ->
      compare_aux t1 t2
  | (Node(Empty, v1, r1, _) :: t1, Node(Empty, v2, r2, _) :: t2) ->
      let c = Ord.compare v1 v2 in
      if c <> 0 then c else compare_aux (r1::t1) (r2::t2)
  | (Node(Empty, v1, r1, _) :: t1, Empty :: t2) ->
```

```
      compare_aux (Node(Empty, v1, r1, 0) :: t1) t2
  | (Node(l1, v1, r1, _) :: t1, t2) when l1 <> Empty ->
      compare_aux (l1 :: Node(Empty, v1, r1, 0) :: t1) t2
  | (Empty :: t1, Node(Empty, v2, r2, _) :: t2) ->
      compare_aux t1 (Node(Empty, v2, r2, 0) :: t2)
  | (t1, Node(l2, v2, r2, _) :: t2) (* when l2 <> Empty *) ->
      compare_aux t1 (l2 :: Node(Empty, v2, r2, 0) :: t2)
```

68.3 Lists of trees

flatten l returns the list of elements of l i.e. the list of elements actually compared

Fixpoint *flatten* $[l{:}(list\ tree)]$: $(list\ elt)$:=
Cases l *of*
$\mid$ *nil* $\Rightarrow$ []
$\mid$ $(cons\ t\ r) \Rightarrow (app\ (elements\_tree\ t)\ (flatten\ r))$ *end.*

sorted_l l expresses that elements in the trees of l are sorted, and that all trees in l are binary search trees.

Inductive *In_tree_l* : $elt \to (list\ tree) \to Prop$:=
$\mid InHd : (x{:}elt)(s{:}tree)(l{:}(list\ tree))(In\_tree\ x\ s) \to$
$(In\_tree\_l\ x\ (cons\ s\ l))$
$\mid InTl : (x{:}elt)(s{:}tree)(l{:}(list\ tree))(In\_tree\_l\ x\ l) \to$
$(In\_tree\_l\ x\ (cons\ s\ l))$.

Hint *In_tree_l* := *Constructors In_tree_l.*

Inductive *sorted_l* : $(list\ tree) \to Prop$:=
$\mid SortedLNil$:
$(sorted\_l\ [])$
$\mid SortedLCons$:
$(s{:}tree)(l{:}(list\ tree))$
$(bst\ s) \to (sorted\_l\ l) \to$
$((x{:}elt)(In\_tree\ x\ s) \to (y{:}elt)(In\_tree\_l\ y\ l) \to (X.lt\ x\ y)) \to$
$(sorted\_l\ (cons\ s\ l))$.

Hint *sorted_l* := *Constructors sorted_l.*

Lemma *sort_app* :
$(l1,l2{:}(list\ elt))(sort\ E.lt\ l1) \to (sort\ E.lt\ l2) \to$
$((x,y{:}elt)(InList\ E.eq\ x\ l1) \to (InList\ E.eq\ y\ l2) \to (X.lt\ x\ y)) \to$
$(sort\ E.lt\ (app\ l1\ l2))$.

Lemma *in_app* :
$(x{:}elt)(l1,l2{:}(list\ elt))$
$(InList\ E.eq\ x\ (app\ l1\ l2)) \to$

$((InList\ E.eq\ x\ l1) \vee (InList\ E.eq\ x\ l2))$.

Lemma *in_flatten* :
$(x{:}elt)(l{:}(list\ tree))$
$(InList\ E.eq\ x\ (flatten\ l)) \rightarrow (In\_tree\_l\ x\ l)$.

Lemma *sorted_flatten* :
$(l{:}(list\ tree))(sorted\_l\ l) \rightarrow (sort\ E.lt\ (flatten\ l))$.

68.4 Termination of *compare_aux*

The measure is given by $m([]) = 0$, $m(s :: l) = m(s) + m(l)$, $m(Leaf) = 1$, and $m(Node(l,\_,r,\_) = 1 + 3m(l) + m(r)$

Fixpoint *measure_t* $[s{:}tree]$: Z :=
$Cases\ s\ of\ Leaf \Rightarrow 1$
$\mid (Node\ l\ \_\ r\ \_) \Rightarrow 1{+}3{\times}(measure\_t\ l){+}(measure\_t\ r)\ end.$

Fixpoint *measure_l* $[l{:}(list\ tree)]$: Z :=
$Cases\ l\ of\ nil \Rightarrow 0$
$\mid (cons\ s\ l) \Rightarrow (measure\_t\ s){+}(measure\_l\ l)\ end.$

Tactic Definition *Measure_t* := *Unfold measure_t*; *Fold measure_t*.
Tactic Definition *Measure_l* := *Unfold measure_l*; *Fold measure_l*.

Lemma *measure_t_1* : $(s{:}tree)(measure\_t\ s) \geq 1$.

Tactic Definition *Measure_t_1 s* := *Generalize* (*measure_t_1 s*); *Intro*.

Lemma *measure_t_3* : $(l,r{:}tree)(x{:}elt)(z{:}Z)$
$(measure\_t\ (Node\ l\ x\ r\ z)) \geq 3$.

Tactic Definition *Measure_t_3 l x r z* := *Generalize* (*measure_t_3 l x r z*); *Intro*.

Lemma *measure_l_0* : $(l{:}(list\ tree))(measure\_l\ l){\geq}0$.

Tactic Definition *Measure_l_0 l* := *Generalize* (*measure_l_0 l*); *Intro*.

Induction principle over the sum of the measures for two lists

Definition *compare_rec2* :
$(P{:}(list\ tree){\rightarrow}(list\ tree){\rightarrow}Set)$
$((x,x'{:}(list\ tree))$
$((y,y'{:}(list\ tree))$
$(measure\_l\ y){+}(measure\_l\ y') < (measure\_l\ x){+}(measure\_l\ x')$
${\rightarrow}(P\ y\ y'))$
$\rightarrow (P\ x\ x')) \rightarrow$
$(x,x'{:}(list\ tree))(P\ x\ x')$.

68.5 Lemmas for the correctness of *compare*

Lemma *lt_nil_elements_tree_Node* :
  (*l*,*r*:*tree*)(*x*:*elt*)(*z*:*Z*)(*L.lt* [] (*elements_tree* (*Node l x r z*))).

Lemma *lt_app* : (*l1*,*l2*,*l3*:(*list elt*))
  (*L.lt l1 l2*) → (*L.lt l1* (*app l2 l3*)).

Lemma *lt_app_eq* : (*l1*,*l2*,*l3*:(*list elt*))
  (*L.lt l2 l3*) → (*L.lt* (*app l1 l2*) (*app l1 l3*)).

Lemma *lt_eq_1* : (*l1*,*l2*,*l3*:(*list elt*))
  l1=*l2* → (*L.lt l1 l3*) → (*L.lt l2 l3*).

Lemma *lt_eq_2* : (*l1*,*l2*,*l3*:(*list elt*))
  l2=*l3* → (*L.lt l1 l2*) → (*L.lt l1 l3*).

Lemma *eq_eq_1* : (*l1*,*l2*,*l3*:(*list elt*))
  l1=*l2* → (*L.eq l1 l3*) → (*L.eq l2 l3*).

Lemma *eq_eq_2* : (*l1*,*l2*,*l3*:(*list elt*))
  l2=*l3* → (*L.eq l1 l2*) → (*L.eq l1 l3*).

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*)).

Hints *Resolve lt_nil_elements_tree_Node lt_app lt_app_eq*
  lt_eq_1 lt_eq_2 eq_eq_1 eq_eq_2 l_eq_cons.

Lemma *elements_app* : (*s*:*tree*)(*acc*:(*list elt*))
  (*elements_tree_aux acc s*) = (*app* (*elements_tree s*) *acc*).

main lemma for correctness of *compare*

Lemma *compare_flatten* :
  (*l*,*r*:*tree*)(*x*:*elt*)(*z*:*Z*)(*tl*:(*list tree*))
  (*flatten* ((*Node l x r z*) :: *tl*))
= (*flatten* (*l* :: ((*Node Leaf x r z*) :: *tl*))).

Hints *Resolve compare_flatten.*

same lemma, expressed differently

Lemma *compare_flatten_1* :
  (*t0*,*t2*:*tree*)(*t1*:*elt*)(*z*:*Z*)(*l*:(*list elt*))
  (*app* (*elements_tree t0*) (*t1* :: (*app* (*elements_tree t2*) *l*)))
= (*app* (*elements_tree* (*Node t0 t1 t2 z*)) *l*).

Hints *Resolve compare_flatten_1.*

invariant for *compare l1 l2*: *Leaf* may only occur on head of *l1* and *l2*, and only when the other list is non-empty

Fixpoint *no_leaf* [*l*:(*list tree*)] : *Prop* := *Cases l of*
  | *nil* ⇒ *True*

| (*cons Leaf _*) ⇒ *False*
| (*cons _ r*) ⇒ (*no_leaf r*) *end.*

Inductive *leaf_invariant* : (*list tree*) → (*list tree*) → *Prop* :=
| *LI_nil_l* : (*l*:(*list tree*)) (*no_leaf l*) → (*leaf_invariant* [] *l*)
| *LI_l_nil* : (*l*:(*list tree*)) (*no_leaf l*) → (*leaf_invariant l* [])
| *LI_leaf_leaf* : (*l1*,*l2*:(*list tree*))(*no_leaf l1*) → (*no_leaf l2*) →
(*leaf_invariant* (*Leaf* :: *l1*) (*Leaf* :: *l2*))
| *LI_leaf_l* : (*l1*,*l2*:(*list tree*))(*no_leaf l1*) → (*no_leaf l2*) →
¬*l2*=[] → (*leaf_invariant* (*Leaf* :: *l1*) *l2*)
| *LI_l_leaf* : (*l1*,*l2*:(*list tree*))(*no_leaf l1*) → (*no_leaf l2*) →
¬*l1*=[] → (*leaf_invariant l1* (*Leaf* :: *l2*))
| *LI_l_l* : (*l1*,*l2*:(*list tree*))(*no_leaf l1*) → (*no_leaf l2*) →
¬*l1*=[] → ¬*l2*=[] → (*leaf_invariant l1 l2*).

Hint *leaf_invariant* := *Constructors leaf_invariant.*

Lemma *no_leaf_invariant* : (*l1*,*l2*:(*list tree*))
(*no_leaf l1*) → (*no_leaf l2*) → (*leaf_invariant l1 l2*).

Hints *Resolve no_leaf_invariant.*

68.6 *compare_aux* and *compare*

Definition *compare_aux* :
(*l1*,*l2*:(*list tree*))(*sorted_l l1*) → (*sorted_l l2*) →
(*leaf_invariant l1 l2*) →
(*Compare L.lt L.eq* (*flatten l1*) (*flatten l2*)).

Lemma *flatten_elements* :
(*s*:*tree*)(*flatten* (*s*::[])) = (*elements_tree s*).

Definition *compare* : (*s1*,*s2*:*t*)(*Compare lt eq s1 s2*).

69 Equality test

Definition *equal* : (*s*,*s'*:*t*){ *Equal s s'* } + { ¬(*Equal s s'*) }.

Write State toto.

End *Make.*

Module FSet

Require Export *FSetInterface.*
Require Export *FSetBridge.*

Require Export *FSetProperties*.
Require Export *FSetList*.
Require Export *FSetRBT*.
Require Export *FSetAVL*.