Contents

Chapter 1

Library MSetsExtra.MSetFoldWithAbort

1.1 Fold with abort for sets

This file provided an efficient fold operation for set interfaces. The standard fold iterates over all elements of the set. The efficient one - called foldWithAbort - is allowed to skip certain elements and thereby abort early.

```
Require Export MSetInterface.
Require Import ssreflect.
Require Import MSetWithDups.
Require Import Int.
Require Import MSetGenTree MSetAVL MSetRBT.
Require Import MSetList MSetWeakList.
```

1.1.1 Fold With Abort Operations

We want to provide an efficient folding operation. Efficieny is gained by aborting the folding early, if we know that continuing would not have an effect any more. Formalising this leads to the following specification of foldWithAbort.

Definition foldWithAbortType
  elt element type of set t type of set A return type :=
  $(elt \to A \to A) \to$ f $(elt \to A \to$ **bool**$) \to$ f_abort $t \to$ input set A
$\to$ base value A.

Definition foldWithAbortSpecPred $\{elt\ t$: Type$\}$
  $(In : elt \to t \to$ Prop$)$
  (fold : $\forall\ \{A$: Type$\}$, $(elt \to A \to A) \to t \to A \to A)$
  $(foldWithAbort$: $\forall\ \{A$: Type$\}$, foldWithAbortType $elt\ t\ A)$: Prop :=

$\forall$

$(A : \texttt{Type})$
result type
$(i\ i' : A)$
base values for foldWithAbort and fold
$(f : elt \to A \to A)\ (f' : elt \to A \to A)$
fold functions for foldWithAbort and fold
$(f\_abort : elt \to A \to \textsf{bool})$
abort function
$(s : t)$ sets to fold over
$(P : A \to A \to \texttt{Prop})$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P $(\forall\ st\ st'\ e,$ $In\ e\ s \to P\ st\ st' \to P\ (f\ e\ st)\ (f\ e\ st')) \to$

f and f' agree for the elements of s $(\forall\ e\ st,\ In\ e\ s \to (P\ (f\ e\ st)\ (f'\ e\ st))) \to$

f_abort is OK, i.e. all other elements can be skipped without leaving the equivalence relation. $(\forall\ e1\ st,$
$In\ e1\ s \to f\_abort\ e1\ st = \textsf{true} \to$
$(\forall\ st'\ e2,\ P\ st\ st' \to$
$In\ e2\ s \to e2 \neq e1 \to$
$P\ st\ (f\ e2\ st'))) \to$

The base values are in equivalence relation $P\ i\ i' \to$

The results are in equivalence relation $P\ (foldWithAbort\ f\ f\_abort\ s\ i)\ (fold\ f'\ s\ i')$.

The specification of folding for ordered sets (as represented by interface *Sets*) demands that elements are visited in increasing order. For ordered sets we can therefore abort folding based on the weaker knowledge that greater elements have no effect on the result. The following definition captures this.

`Definition` foldWithAbortGtType
elt element type of set t type of set A return type :=
$(elt \to A \to A) \to$ f $(elt \to A \to \textsf{bool}) \to$ f_gt $t \to$ input set $A \to$ base value A.

`Definition` foldWithAbortGtSpecPred $\{elt\ t : \texttt{Type}\}$
$(lt : elt \to elt \to \texttt{Prop})$
$(In : elt \to t \to \texttt{Prop})$
$(\texttt{fold} : \forall\ \{A : \texttt{Type}\},\ (elt \to A \to A) \to t \to A \to A)$

($foldWithAbortGt$: $\forall$ {A : `Type`}, foldWithAbortType elt t A) : `Prop` :=

$\forall$

(A : `Type`)
result type
(i i' : A)
base values for foldWithAbort and fold
(f : $elt \to A \to A$) (f' : $elt \to A \to A$)
fold functions for foldWithAbort and fold
(f_gt : $elt \to A \to$ **bool**)
abort function
(s : t) sets to fold over
(P : $A \to A \to$ `Prop`) equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P ($\forall$ st st' e, In e $s \to P$ st $st' \to P$ (f e st) (f e st')) $\to$

f and f' agree for the elements of s ($\forall$ e st, In e $s \to$ (P (f e st) (f' e st))) $\to$

f_abort is OK, i.e. all other elements can be skipped without leaving the equivalence relation. ($\forall$ $e1$ st,
In $e1$ $s \to f\_gt$ $e1$ st = true $\to$
($\forall$ st' $e2$, P st $st' \to$
In $e2$ $s \to lt$ $e1$ $e2 \to$
P st (f $e2$ st'))) $\to$

The base values are in equivalence relation P i $i' \to$

The results are in equivalence relation P ($foldWithAbortGt$ f f_gt s i) ($fold$ f' s i').

For ordered sets, we can safely skip elements at the end based on the knowledge that they are all greater than the current element. This leads to serious performance improvements for operations like filtering. It is tempting to try the symmetric operation and skip elements at the beginning based on the knowledge that they are too small to be interesting. So, we would like to start late as well as abort early.

This is indeed a very natural and efficient operation for set implementations based on binary search trees (i.e. the AVL and RBT sets). We can completely symmetrically to skipping greater elements also skip smaller elements. This leads to the following specification.

`Definition` foldWithAbortGtLtType

elt element type of set t type of set A return type :=
$(elt \to A \to$ **bool**$) \to$ f_lt $(elt \to A \to A) \to$ f $(elt \to A \to$ **bool**$) \to$ f_gt
$t \to$ input set $A \to$ base value A.

`Definition` foldWithAbortGtLtSpecPred $\{elt\ t : \texttt{Type}\}$
$(lt : elt \to elt \to \texttt{Prop})$
$(In : elt \to t \to \texttt{Prop})$
$(\texttt{fold} : \forall \{A : \texttt{Type}\}, (elt \to A \to A) \to t \to A \to A)$
$(foldWithAbortGtLt : \forall \{A : \texttt{Type}\}, \text{foldWithAbortGtLtType}\ elt\ t\ A) : \texttt{Prop} :=$

$\forall$
$(A : \texttt{Type})$
result type
$(i\ i' : A)$
base values for foldWithAbort and fold
$(f : elt \to A \to A)\ (f' : elt \to A \to A)$
fold functions for foldWithAbort and fold
$(f\_lt\ f\_gt : elt \to A \to$ **bool**$)$
abort functions
$(s : t)$ sets to fold over
$(P : A \to A \to \texttt{Prop})$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P $(\forall\ st\ st'\ e,$
$In\ e\ s \to P\ st\ st' \to P\ (f\ e\ st)\ (f\ e\ st')) \to$

f and f' agree for the elements of s $(\forall\ e\ st, In\ e\ s \to (P\ (f\ e\ st)\ (f'\ e\ st))) \to$

f_lt is OK, i.e. smaller elements can be skipped without leaving the equivalence relation.
$(\forall\ e1\ st,$
$In\ e1\ s \to f\_lt\ e1\ st = \mathsf{true} \to$
$(\forall\ st'\ e2, P\ st\ st' \to$
$In\ e2\ s \to lt\ e2\ e1 \to$
$P\ st\ (f\ e2\ st'))) \to$

f_gt is OK, i.e. greater elements can be skipped without leaving the equivalence relation.
$(\forall\ e1\ st,$
$In\ e1\ s \to f\_gt\ e1\ st = \mathsf{true} \to$
$(\forall\ st'\ e2, P\ st\ st' \to$
$In\ e2\ s \to lt\ e1\ e2 \to$
$P\ st\ (f\ e2\ st'))) \to$

The base values are in equivalence relation $P\ i\ i' \rightarrow$

The results are in equivalence relation P (*foldWithAbortGtLt f_lt f f_gt s i*) (*fold f' s i'*).

We are interested in folding with abort mainly for runtime performance reasons of extracted code. The argument functions *f_lt*, *f_gt* and *f* of foldWithAbortGtLt often share a large, comparably expensive part of their computation.

In order to further improve runtime performance, therefore another version foldWithAbortPrecompute *f_precompute f_lt f f_gt* that uses an extra function *f_precompute* to allows to compute the commonly used parts of these functions only once. This leads to the following definitions.

`Definition` foldWithAbortPrecomputeType

elt element type of set *t* type of set *A* return type *B* type of precomputed results :=

$(elt \rightarrow B) \rightarrow$ f_precompute $(elt \rightarrow B \rightarrow A \rightarrow$ **bool**$) \rightarrow$ f_lt $(elt \rightarrow B \rightarrow A \rightarrow A) \rightarrow$ f $(elt \rightarrow B \rightarrow A \rightarrow$ **bool**$) \rightarrow$ f_gt $t \rightarrow$ input set $A \rightarrow$ base value A.

The specification is similar to the one without precompute, but uses *f_precompute* so avoid doing computations multiple times `Definition` foldWithAbortPrecomputeSpecPred $\{elt\ t$: `Type`$\}$

$(lt : elt \rightarrow elt \rightarrow$ `Prop`$)$

$(In : elt \rightarrow t \rightarrow$ `Prop`$)$

(`fold` : $\forall$ $\{A$: `Type`$\}$, $(elt \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$

(*foldWithAbortPrecompute* : $\forall$ $\{A\ B$: `Type`$\}$, foldWithAbortPrecomputeType *elt t A B*) : `Prop` :=

$\forall$

(*A B* : `Type`)

result type

$(i\ i' : A)$

base values for foldWithAbortPrecompute and fold

$(f\_precompute : elt \rightarrow B)$

precompute function

$(f : elt \rightarrow B \rightarrow A \rightarrow A)$ $(f' : elt \rightarrow A \rightarrow A)$

fold functions for foldWithAbortPrecompute and fold

$(f\_lt\ f\_gt : elt \rightarrow B \rightarrow A \rightarrow$ **bool**$)$

abort functions

$(s : t)$ sets to fold over

$(P : A \rightarrow A \rightarrow$ `Prop`$)$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \rightarrow$

f is for the elements of s compatible with the equivalence relation P ($\forall$ *st st' e*, *In e s* $\rightarrow$ *P st st'* $\rightarrow$ *P (f e (f_precompute e) st) (f e (f_precompute e) st'))* $\rightarrow$

f and *f'* agree for the elements of s ($\forall$ *e st*, *In e s* $\rightarrow$ *(P (f e (f_precompute e) st) (f' e st)))* $\rightarrow$

f_lt is OK, i.e. smaller elements can be skipped without leaving the equivalence relation. ($\forall$ *e1 st*,
In e1 s $\rightarrow$ *f_lt e1 (f_precompute e1) st* = true $\rightarrow$
($\forall$ *st' e2*, *P st st'* $\rightarrow$
In e2 s $\rightarrow$ *lt e2 e1* $\rightarrow$
P st (f e2 (f_precompute e2) st'))) $\rightarrow$

f_gt is OK, i.e. greater elements can be skipped without leaving the equivalence relation. ($\forall$ *e1 st*,
In e1 s $\rightarrow$ *f_gt e1 (f_precompute e1) st* = true $\rightarrow$
($\forall$ *st' e2*, *P st st'* $\rightarrow$
In e2 s $\rightarrow$ *lt e1 e2* $\rightarrow$
P st (f e2 (f_precompute e2) st'))) $\rightarrow$

The base values are in equivalence relation *P i i'* $\rightarrow$

The results are in equivalence relation *P (foldWithAbortPrecompute f_precompute f_lt f f_gt s i) (fold f' s i')*.

Module Types

We now define a module type for foldWithAbort. This module type demands only the existence of the precompute version, since the other ones can be easily defined via this most efficient one.

```
Module Type HasFoldWithAbort (E : OrderedType) (Import C : WSetsOnWithDups E).
  Parameter foldWithAbortPrecompute : ∀ {A B : Type},
    foldWithAbortPrecomputeType elt t A B.
  Parameter foldWithAbortPrecomputeSpec :
     foldWithAbortPrecomputeSpecPred E.lt In (@fold) (@foldWithAbortPrecompute).
End HasFoldWithAbort.
```

1.1.2 Derived operations

Using these efficient fold operations, many operations can be implemented efficiently. We provide lemmata and efficient implementations of useful algorithms via module HASFOLDWITHABORTOPS.

```
Module HasFoldWithAbortOps (E : OrderedType) (C : WSetsOnWithDups E)
                           (FT : HasFoldWithAbort E C).

  Import FT.
  Import C.
```

First lets define the other folding with abort variants

```
  Definition foldWithAbortGtLt {A} f_lt (f : (elt → A → A)) f_gt :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun e _ st ⇒ f_lt e st)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_gt e st).

  Lemma foldWithAbortGtLtSpec :
     foldWithAbortGtLtSpecPred E.lt In (@fold) (@foldWithAbortGtLt).
  Proof.
    rewrite /foldWithAbortGtLt /foldWithAbortGtLtSpecPred.
    intros A i i' f f' f_lt f_gt s P.
    move ⇒ H_f_compat H_ff' H_lt H_gt H_ii'.
    apply foldWithAbortPrecomputeSpec ⇒ //.
  Qed.

  Definition foldWithAbortGt {A} (f : (elt → A → A)) f_gt :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun _ _ _ ⇒ false)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_gt e st).

  Lemma foldWithAbortGtSpec :
     foldWithAbortGtSpecPred E.lt In (@fold) (@foldWithAbortGt).
  Proof.
    rewrite /foldWithAbortGt /foldWithAbortGtSpecPred.
    intros A i i' f f' f_gt s P.
    move ⇒ H_f_compat H_ff' H_gt H_ii'.
    apply foldWithAbortPrecomputeSpec ⇒ //.
  Qed.

  Definition foldWithAbort {A} (f : (elt → A → A)) f_abort :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun e _ st ⇒ f_abort e st)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_abort e st).

  Lemma foldWithAbortSpec :
     foldWithAbortSpecPred In (@fold) (@foldWithAbort).
  Proof.
    rewrite /foldWithAbort /foldWithAbortGtSpecPred.
```

```
  intros A i i' f f' f_abort s P.
  move ⇒ H_equiv_P H_f_compat H_ff' H_abort H_ii'.
  have H_lt_neq: (∀ e1 e2, E.lt e1 e2 → e1 ≠ e2). {
    move ⇒ e1 e2 H_lt H_e12_eq.
    rewrite H_e12_eq in H_lt.
    have : ( Irreflexive E.lt) by apply StrictOrder_Irreflexive.
    rewrite /Irreflexive /Reflexive /complement ⇒ H.
    eapply H, H_lt.
  }
  apply foldWithAbortPrecomputeSpec ⇒ //; (
    move ⇒ e1 st H_in_e1 H_abort_e1 st' e2 H_P H_in_e2 /H_lt_neq H_lt;
    apply (H_abort e1 st H_in_e1 H_abort_e1 st' e2 H_P H_in_e2);
    by auto
  ).
Qed.
```

Specialisations for equality

Let's provide simplified specifications, which use equality instead of an arbitrary equivalence relation on results. Lemma foldWithAbortPrecomputeSpec_Equal : ∀ (A B : Type) (i : A) (f_pre : elt → B)
(f : elt → B → A → A) (f' : elt → A → A) (f_lt f_gt : elt → B → A → bool) (s : t),

```
    (∀ e st, In e s → (f e (f_pre e) st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_lt e1 (f_pre e1) st = true →
        (∀ e2, In e2 s → E.lt e2 e1 →
                    (f e2 (f_pre e2) st = st))) →


    (∀ e1 st,
        In e1 s → f_gt e1 (f_pre e1) st = true →
        (∀ e2, In e2 s → E.lt e1 e2 →
                    (f e2 (f_pre e2) st = st))) →

    (foldWithAbortPrecompute f_pre f_lt f f_gt s i) = (fold f' s i).
Proof.
  intros A B i f_pre f f' f_lt f_gt s H_f' H_lt H_gt.
   eapply (foldWithAbortPrecomputeSpec A B i i f_pre f f'); eauto. {
     apply eq_equivalence.
```

```
    } {
      move => st1 st2 e H_in -> //.
    } {
      move => e1 st H_e1_in H_do_smaller st' e2 <-.
      move : (H_lt e1 st H_e1_in H_do_smaller e2).
      intuition.
    } {
      move => e1 st H_e1_in H_do_greater st' e2 <-.
      move : (H_gt e1 st H_e1_in H_do_greater e2).
      intuition.
    }
Qed.

Lemma foldWithAbortGtLtSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt -> A -> A) (f' : elt -> A -> A) (f_lt f_gt : elt -> A -> bool) (s : t),

    (∀ e st, In e s -> (f e st = f' e st)) ->


    (∀ e1 st,
        In e1 s -> f_lt e1 st = true ->
        (∀ e2, In e2 s -> E.lt e2 e1 ->
                  (f e2 st = st))) ->


    (∀ e1 st,
        In e1 s -> f_gt e1 st = true ->
        (∀ e2, In e2 s -> E.lt e1 e2 ->
                  (f e2 st = st))) ->

    (foldWithAbortGtLt f_lt f f_gt s i) = (fold f' s i).
Proof.
  intros A i f f' f_lt f_gt s H_f' H_lt H_gt.
   eapply (foldWithAbortGtLtSpec A i i f f'); eauto. {
      apply eq_equivalence.
    } {
      move => st1 st2 e H_in -> //.
    } {
      move => e1 st H_e1_in H_do_smaller st' e2 <-.
      move : (H_lt e1 st H_e1_in H_do_smaller e2).
      intuition.
    } {
      move => e1 st H_e1_in H_do_greater st' e2 <-.
```

```
      move : (H_gt e1 st H_e1_in H_do_greater e2).
      intuition.
    }
Qed.

Lemma foldWithAbortGtSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt → A → A) (f' : elt → A → A) (f_gt : elt → A → bool) (s : t),

    (∀ e st, In e s → (f e st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_gt e1 st = true →
        (∀ e2, In e2 s → E.lt e1 e2 →
                       (f e2 st = st))) →

    (foldWithAbortGt f f_gt s i) = (fold f' s i).
Proof.
  intros A i f f' f_gt s H_f' H_gt.
   eapply (foldWithAbortGtSpec A i i f f'); eauto. {
      apply eq_equivalence.
    } {
      move ⇒ st1 st2 e H_in → //.
    } {
      move ⇒ e1 st H_e1_in H_do_greater st' e2 ←.
      move : (H_gt e1 st H_e1_in H_do_greater e2).
      intuition.
    }
Qed.

Lemma foldWithAbortSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt → A → A) (f' : elt → A → A) (f_abort : elt → A → bool) (s : t),

    (∀ e st, In e s → (f e st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_abort e1 st = true →
        (∀ e2, In e2 s → e1 ≠ e2 →
                       (f e2 st = st))) →

    (foldWithAbort f f_abort s i) = (fold f' s i).
Proof.
   intros A i f f' f_abort s H_f' H_abort.
```

```
    eapply (foldWithAbortSpec A i i f f'); eauto. {
      apply eq_equivalence.
    } {
      move => st1 st2 e H_in -> //.
    } {
      move => e1 st H_e1_in H_do_abort st' e2 <-.
      move : (H_abort e1 st H_e1_in H_do_abort e2).
      intuition.
    }
  Qed.
```

FoldWithAbortSpecArgs

While folding, we are often interested in skipping elements that do not satisfy a certain property P. This needs expressing in terms of skips of smaller of larger elements in order to be done efficiently by our folding functions. Formally, this leads to the definition of *foldWithAbortSpecForPred*.

Given a **FoldWithAbortSpecArg** for a predicate P and a set s, many operations can be implemented efficiently. Below we will provide efficient versions of filter, choose, ∃, ∀ and more. `Record` **FoldWithAbortSpecArg** $\{B\}$:= {

fwasa_f_pre : (elt → B); The precompute function fwasa_f_lt : (elt → B → **bool**); f_lt without state argument fwasa_f_gt : (elt → B → **bool**); f_gt without state argument fwasa_P' : (elt → B → **bool**) the predicate P }.

foldWithAbortSpecForPred s P fwasa holds, if the argument *fwasa* fits the predicate P for set s. `Definition` foldWithAbortSpecArgsForPred $\{A$: `Type`$\}$
(s : *t*) (P : elt → **bool**) (*fwasa* : @**FoldWithAbortSpecArg** A) :=

the predicate P' coincides for s and the given precomputation with P (∀ e, *In* e s → (fwasa_P' *fwasa* e (fwasa_f_pre *fwasa* e) = P e)) ∧

If fwasa_f_lt holds, all elements smaller than the current one don't satisfy predicate P. (∀ $e1$,
In $e1$ s → fwasa_f_lt *fwasa* $e1$ (fwasa_f_pre *fwasa* $e1$) = true →
(∀ $e2$, *In* $e2$ s → *E.lt* $e2$ $e1$ → (P $e2$ = false))) ∧

If fwasa_f_gt holds, all elements greater than the current one don't satisfy predicate P. (∀ $e1$,
In $e1$ s → fwasa_f_gt *fwasa* $e1$ (fwasa_f_pre *fwasa* $e1$) = true →
(∀ $e2$, *In* $e2$ s → *E.lt* $e1$ $e2$ → (P $e2$ = false))).

Filter with abort

`Definition` filter_with_abort $\{B\}$ (*fwasa* : @**FoldWithAbortSpecArg** B) s :=

```
    @foldWithAbortPrecompute t B (fwasa_f_pre fwasa) (fun e p _ ⇒ fwasa_f_lt fwasa e
p)
      (fun e e_pre s ⇒ if fwasa_P' fwasa e e_pre then add e s else s)
      (fun e p _ ⇒ fwasa_f_gt fwasa e p) s empty.
Lemma filter_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  Equal (filter_with_abort fwasa s)
        (filter P s).
Proof.
  unfold foldWithAbortSpecArgsForPred.
  move ⇒ [] f_pre f_lt f_gt P' P s /=.
  move ⇒ [H_f'] [H_lt] H_gt H_proper.
  rewrite /filter_with_abort /=.
  have → : (foldWithAbortPrecompute f_pre (fun e p _ ⇒ f_lt e p)
   (fun (e : elt) (e_pre : B) (s0 : t) ⇒
    if P' e e_pre then add e s0 else s0) (fun e p _ ⇒ f_gt e p) s empty =
   (fold (fun e s0 ⇒ if P e then add e s0 else s0) s empty)). {
    apply foldWithAbortPrecomputeSpec_Equal. {
      intros e st H_e_in.
      rewrite H_f' //.
    } {
      intros e1 st H_e1_in H_f_lt_eq e2 H_e2_in H_lt_e2_e1.
      rewrite (H_f' _ H_e2_in).
      suff → : (P e2 = false) by done.
      apply (H_lt e1); eauto.
    } {
      intros e1 st H_e1_in H_f_gt_eq e2 H_e2_in H_gt_e2_e1.
      rewrite (H_f' _ H_e2_in).
      suff → : (P e2 = false) by done.
      apply (H_gt e1); eauto.
    }
  }

  rewrite /Equal ⇒ e.
  rewrite fold_spec.
  setoid_rewrite filter_spec ⇒ //.
  suff → : ∀ acc, In e
    (fold_left
       (flip (fun (e0 : elt) (s0 : t) ⇒ if P e0 then add e0 s0 else s0))
       (elements s) acc) ↔ (InA E.eq e (elements s) ∧ P e = true) ∨ (In e acc). {
    rewrite elements_spec1.
```

```
    suff : (¬In e empty) by tauto.
    apply empty_spec.
  }
  induction (elements s) as [| x xs IH] ⇒ acc. {
    rewrite /= InA_nil. tauto.
  } {
    rewrite /= /flip IH InA_cons.
    case_eq (P x). {
      rewrite add_spec.
      intuition.
      left.
      rewrite H0.
      split ⇒ //.
      left.
      apply Equivalence_Reflexive.
    } {
      intuition.
      contradict H2.
      setoid_rewrite H1.
      by rewrite H.
    }
  }
Qed.
```

Choose with abort

```
Definition choose_with_abort {B} (fwasa : @FoldWithAbortSpecArg B) s :=
   foldWithAbortPrecompute (fwasa_f_pre fwasa)
     (fun e p st ⇒ match st with None ⇒ (fwasa_f_lt fwasa e p) | _ ⇒ true end)
     (fun e e_pre st ⇒ match st with None ⇒
        if (fwasa_P' fwasa e e_pre) then Some e else None | _ ⇒ st end)

     (fun e p st ⇒ match st with None ⇒ (fwasa_f_gt fwasa e p) | _ ⇒ true end)
     s None.

Lemma choose_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (match (choose_with_abort fwasa s) with
     | None ⇒ (∀ e, In e s → P e = false)
     | Some e ⇒ In e s ∧ (P e = true)
   end).
Proof.
  rewrite /foldWithAbortSpecArgsForPred.
```

```
move => [] f_pre f_lt f_gt P' P s /=.
move => [H_f'] [H_lt] H_gt H_proper.

set fwasa := {|
    fwasa_f_pre := f_pre;
    fwasa_f_lt := f_lt;
    fwasa_f_gt := f_gt;
    fwasa_P' := P' |}.

suff : (match (choose_with_abort fwasa s) with
    | None => (∀ e, InA E.eq e (elements s) → P e = false)
    | Some e => InA E.eq e (elements s) ∧ (P e = true)
  end). {
    case (choose_with_abort fwasa s). {
      move => e.
      rewrite elements_spec1 //.
    } {
      move => H e H_in.
      apply H.
      rewrite elements_spec1 //.
    }
}

have → : (choose_with_abort fwasa s =
   (fold (fun e st =>
     match st with
       | None => if P e then Some e else None
       | _ => st end) s None)). {
   apply foldWithAbortPrecomputeSpec_Equal. {
     intros e st H_e_in.
     case st => //=.
     rewrite H_f' //.
   } {
     move => e1 [] //= H_e1_in H_f_lt_eq e2 H_e2_in H_lt_e2_e1.
     rewrite (H_f' _ H_e2_in).
     case_eq (P e2) => // H_P_e2.
     contradict H_P_e2.
     apply not_true_iff_false, (H_lt e1); auto.
   } {
     move => e1 [] //= H_e1_in H_f_gt_eq e2 H_e2_in H_gt_e2_e1.
     rewrite (H_f' _ H_e2_in).
     case_eq (P e2) => // H_P_e2.
     contradict H_P_e2.
     apply not_true_iff_false, (H_gt e1); auto.
```

```
    }
  }

  rewrite fold_spec /flip.
  induction (elements s) as [| x xs IH]. {
    rewrite /=.
    move => e /InA_nil //.
  } {
    case_eq (P x) => H_Px; rewrite /= H_Px. {
      have -> : ∀ xs, fold_left (fun (x0 : option elt) (y : elt) =>
                  match x0 with | Some _ => x0 | None => if P y then Some y else
None
                  end) xs (Some x) = Some x. {
        move => ys.
        induction ys => //.
      }
      split; last assumption.
      apply InA_cons_hd.
      apply E.eq_equiv.
    } {
      move : IH.
      case (fold_left
        (fun (x0 : option elt) (y : elt) =>
          match x0 with | Some _ => x0 | None => if P y then Some y else None
          end) xs None). {

          move => e [H_e_in] H_Pe.
          split; last assumption.
          apply InA_cons_tl => //.
      } {
        move => H_e_nin e H_e_in.
        have : (InA E.eq e xs ∨ (E.eq e x)). {
          inversion H_e_in; tauto.
        }
        move => []. {
          apply H_e_nin.
        } {
          move => -> //.
        }
      }
    }
  }
```

```
Qed.
```

Exists and Forall with abort

```
Definition exists_with_abort {B} (fwasa : @FoldWithAbortSpecArg B) s :=
  match choose_with_abort fwasa s with
    | None ⇒ false
    | Some _ ⇒ true
  end.

Lemma exists_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (exists_with_abort fwasa s =
   exists_ P s).
Proof.
  intros fwasa P s H_fwasa H_proper.
  apply Logic.eq_sym.
  rewrite /exists_with_abort.
  move : (choose_with_abort_spec _ _ _ H_fwasa H_proper).
  case (choose_with_abort fwasa s). {
    move ⇒ e [H_e_in] H_Pe.
    rewrite exists_spec /Exists.
    by ∃ e.
  } {
    move ⇒ H_not_ex.
    apply not_true_iff_false.
    rewrite exists_spec /Exists.
    move ⇒ [e] [H_in] H_pe.
    move : (H_not_ex e H_in).
    rewrite H_pe //.
  }
Qed.
```

Negation leads to forall.

```
Definition forall_with_abort {B} fwasa s :=
  negb (@exists_with_abort B fwasa s).

Lemma forall_with_abort_spec {B} : ∀ fwasa s P,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (forall_with_abort fwasa s =
   for_all (fun e ⇒ negb (P e)) s).
Proof.
  intros fwasa s P H_ok H_proper.
  rewrite /forall_with_abort exists_with_abort_spec; auto.
```

```
    rewrite eq_iff_eq_true negb_true_iff -not_true_iff_false.
    rewrite exists_spec.
    setoid_rewrite for_all_spec; last solve_proper.
    rewrite /Exists /For_all.
    split. {
      move ⇒ H_pre x H_x_in.
      rewrite negb_true_iff -not_true_iff_false ⇒ H_Px.
      apply H_pre.
      by ∃ x.
    } {
      move ⇒ H_pre [x] [H_x_in] H_P_x.
      move : (H_pre x H_x_in).
      rewrite H_P_x.
      done.
    }
  Qed.
End HasFoldWithAbortOps.
```

1.1.3 Modules Types For Sets with Fold with Abort

```
Module Type WSetsWithDupsFoldA.
  Declare Module E : OrderedType.
  Include WSetsOnWithDups E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End WSetsWithDupsFoldA.

Module Type WSetsWithFoldA <: WSets.
  Declare Module E : OrderedType.
  Include WSetsOn E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End WSetsWithFoldA.

Module Type SetsWithFoldA <: Sets.
  Declare Module E : OrderedType.
  Include SetsOn E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End SetsWithFoldA.
```

1.1.4 Implementations

GenTree implementation

Finally, provide such a fold with abort operation for generic trees. `Module` MAKEGENTREEFOLDA (`Import` E : ORDEREDTYPE) (`Import` I:INFOTYP)
(`Import` Raw:OPS E I)
(M : MSETGENTREE.PROPS E I RAW).

```
Fixpoint foldWithAbort_Raw {A B: Type} (f_pre : E.t → B) f_lt (f: E.t → B → A →
A) f_gt t (base: A) : A :=
    match t with
    | Raw.Leaf ⇒ base
    | Raw.Node _ l x r ⇒
        let x_pre := f_pre x in
        let st0 := if f_lt x x_pre base then base else foldWithAbort_Raw f_pre f_lt f f_gt
l base in
        let st1 := f x x_pre st0 in
        let st2 := if f_gt x x_pre st1 then st1 else foldWithAbort_Raw f_pre f_lt f f_gt
r st1 in
        st2
    end.

Lemma foldWithAbort_RawSpec : ∀ (A B : Type) (i i' : A) (f_pre : E.t → B)
      (f : E.t → B → A → A) (f' : E.t → A → A) (f_lt f_gt : E.t → B → A → bool) (s
: Raw.tree)
      (P : A → A → Prop),

      (M.bst s) →
      Equivalence P →
      (∀ st st' e, M.In e s → P st st' → P (f e (f_pre e) st) (f e (f_pre e) st')) →
      (∀ e st, M.In e s → P (f e (f_pre e) st) (f' e st)) →


      (∀ e1 st,
          M.In e1 s → f_lt e1 (f_pre e1) st = true →
          (∀ st' e2, P st st' →
                          M.In e2 s → E.lt e2 e1 →
                          P st (f e2 (f_pre e2) st'))) →


      (∀ e1 st,
          M.In e1 s → f_gt e1 (f_pre e1) st = true →
          (∀ st' e2, P st st' →
                          M.In e2 s → E.lt e1 e2 →
```

```
                           P st (f e2 (f_pre e2) st'))) →
    P i i' →
    P (foldWithAbort_Raw f_pre f_lt f f_gt s i) (fold f' s i').
Proof.
  intros A B i i' f_pre f f' f_lt f_gt s P.
  move ⇒ H_bst H_equiv_P H_P_f H_f' H_RL H_RG.
  set base := s.
  move : i i'.
  have : (∀ e, M.In e base → M.In e s). {
    rewrite /In /base //.
  }
  have : M.bst base. {
    apply H_bst.
  }
  move : base.
  clear H_bst.
  induction base as [|c l IHl e r IHr] using M.tree_ind. {
    rewrite /foldWithAbort_Raw /Raw.fold.
    move ⇒ _ _ i i' //.
  } {
    move ⇒ H_bst H_sub i i' H_P_ii'.
    have [H_bst_l [H_bst_r [H_lt_tree_l H_gt_tree_r]]]:
      M.bst l ∧ M.bst r ∧ M.lt_tree e l ∧ M.gt_tree e r. {
      inversion H_bst. done.
    }

    have H_sub_l : (∀ e0 : E.t, M.In e0 l → M.In e0 s ∧ E.lt e0 e). {
      intros e0 H_in_l.
      split; last by apply H_lt_tree_l.
      eapply H_sub.
      rewrite /M.In M.In_node_iff.
      tauto.
    }
    move : (IHl H_bst_l) ⇒ {IHl} IHl {H_bst_l} {H_lt_tree_l}.
    have H_sub_r : (∀ e0 : E.t, M.In e0 r → M.In e0 s ∧ E.lt e e0). {
      intros e0 H_in_r.
      split; last by apply H_gt_tree_r.
      eapply H_sub.
      rewrite /M.In M.In_node_iff.
      tauto.
```

```
}
move : (IHr H_bst_r) => {IHr} IHr {H_bst_r} {H_gt_tree_r}.
have H_in_e : M.In e s. {
  eapply H_sub.
  rewrite /M.In M.In_node_iff.
  right; left.
  apply Equivalence_Reflexive.
}
move => {H_sub}.
rewrite /=.
set st0 := if f_lt e (f_pre e) i then i else foldWithAbort_Raw f_pre f_lt f f_gt l i.
set st0' := Raw.fold f' l i'.
set st1 := f e (f_pre e) st0.
set st1' := f' e st0'.
set st2 := if f_gt e (f_pre e) st1 then st1 else foldWithAbort_Raw f_pre f_lt f f_gt
r st1.
set st2' := Raw.fold f' r st1'.
have H_P_st0 : P st0 st0'. {
  rewrite /st0 /st0'.
  case_eq (f_lt e (f_pre e) i). {
    move => H_fl_eq.
    move : H_P_ii' H_sub_l.
    move : H_equiv_P H_f' (H_RL _ _ H_in_e H_fl_eq).
    rewrite /M.lt_tree. clear.
    move => H_equiv_P H_f' H_RL.
    move : i'.
    induction l as [|c l IHl e' r IHr] using M.tree_ind. {
      done.
    } {
      intros i' H_P_ii' H_sub_l.
      rewrite /=.
      apply IHr; last first. {
        move => y H_y_in.
        apply H_sub_l.
        rewrite /M.In M.In_node_iff. tauto.
      }
      have [] : (M.In e' s /\ E.lt e' e). {
        apply H_sub_l.
        rewrite /M.In M.In_node_iff.
        right; left.
        apply Equivalence_Reflexive.
      }
```

```
      move => H_e'_in H_lt_in.
      suff H_P_i : (P i (f e' (f_pre e') (fold f' l i'))). {
        eapply Equivalence_Transitive; first apply H_P_i.
        by apply H_f'.
      }
      eapply H_RL => //.
      apply IHl; last first. {
        move => y H_y_in.
        apply H_sub_l.
        rewrite /M.In M.In_node_iff. tauto.
      }
      assumption.
    }
  } {
    move => _.
    apply IHl => //.
    eapply H_sub_l.
  }
}
have H_P_st1 : P st1 st1'. {
  rewrite /st1 /st1'.
  rewrite -H_f' //.
  apply H_P_f => //.
}
have H_P_st2 : P st2 st2'. {
  rewrite /st2 /st2'.
  clearbody st1 st1'.
  case_eq (f_gt e (f_pre e) st1). {
    move => H_gt_eq.
    move : H_P_st1 H_sub_r.
    move : H_equiv_P (H_RG _ _ H_in_e H_gt_eq) H_f'.
    unfold M.gt_tree. clear.
    move => H_equiv_P H_RG H_f'.
    move : st1'.
    induction r as [|c l IHl e' r IHr] using M.tree_ind. {
      done.
    } {
      intros st1' H_P_st1 H_sub_r.
      rewrite /=.
      apply IHr; last first. {
        move => y H_y_in.
        apply H_sub_r.
```

```
              rewrite /M.In M.In_node_iff. tauto.
            }
            have [] : (M.In e' s ∧ E.lt e e'). {
              apply H_sub_r.
              rewrite /M.In M.In_node_iff.
              right; left.
              apply Equivalence_Reflexive.
            }
            move ⇒ H_e'_in H_lt_ee'.
            suff H_P_st1_aux : (P st1 (f e' (f_pre e') (fold f' l st1'))). {
              eapply Equivalence_Transitive; first apply H_P_st1_aux.
              by apply H_f'.
            }
            eapply H_RG ⇒ //.
            apply IHl; last first. {
              move ⇒ y H_y_in.
              apply H_sub_r.
              rewrite /M.In M.In_node_iff. tauto.
            }
            assumption.
          }
        } {
          move ⇒ _.
          apply IHr ⇒ //.
          eapply H_sub_r.
        }
      }
      done.
    }
  Qed.
End MakeGenTreeFoldA.
```

AVL implementation

The generic tree implementation naturally leads to an AVL one.

```
Module MakeAVLSetsWithFoldA (X : OrderedType) <: SetsWithFoldA with
Module E := X.
  Include MSetAVL.Make X.
  Include MakeGenTreeFoldA X Z_as_Int Raw Raw.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt (f: elt → B → A → A) f_gt
t (base: A) : A :=
    foldWithAbort_Raw f_pre f_lt f f_gt (t.(this)) base.
```

Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred *X.lt* In fold (@foldWithAbortPrecompute).

Proof.

intros *A B i i' f_pre f f' f_lt f_gt s P*.

move ⇒ *H_P_f H_f' H_RL H_RG H_P_ii'*.

rewrite /foldWithAbortPrecompute /fold.

apply foldWithAbort_RawSpec ⇒ //.

case *s*. rewrite /this /**Raw.Ok** //.

Qed.

Include HasFoldWithAbortOps X.

End MakeAVLSetsWithFoldA.

RBT implementation

The generic tree implementation naturally leads to an RBT one. Module MakeRBTSetsWithFoldA (X : OrderedType) <: SetsWithFoldA with Module E := X.

Include MSetRBT.Make X.

Include MakeGenTreeFoldA X Color Raw Raw.

Definition foldWithAbortPrecompute $\{A\ B\text{: Type}\}$ *f_pre f_lt* (f: elt → B → A → A) *f_gt* t (*base*: A) : A :=

foldWithAbort_Raw *f_pre f_lt f f_gt* (t.(this)) *base*.

Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred *X.lt* In fold (@foldWithAbortPrecompute).

Proof.

intros *A B i i' f_pre f f' f_lt f_gt s P*.

move ⇒ *H_P_f H_f' H_RL H_RG H_P_ii'*.

rewrite /foldWithAbortPrecompute /fold.

apply foldWithAbort_RawSpec ⇒ //.

case *s*. rewrite /this /**Raw.Ok** //.

Qed.

Include HasFoldWithAbortOps X.

End MakeRBTSetsWithFoldA.

1.1.5 Sorted Lists Implementation

Module MakeListSetsWithFoldA (X : OrderedType) <: SetsWithFoldA with Module E := X.

Include MSetList.Make X.

Fixpoint foldWithAbortRaw $\{A\ B\text{: Type}\}$ (*f_pre* : *X.t* → B) (*f_lt* : *X.t* → B → A → **bool**)

```
    (f: X.t → B → A → A) (f_gt : X.t → B → A → bool) (t : list X.t) (acc : A) : A :=
  match t with
    | nil ⇒ acc
    | x :: xs ⇒ (
        let pre_x := f_pre x in
        let acc := f x (pre_x) acc in
        if (f_gt x pre_x acc) then
          acc
        else
          foldWithAbortRaw f_pre f_lt f f_gt xs acc
      )
  end.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt f f_gt t acc :=
    @foldWithAbortRaw A B f_pre f_lt f f_gt t.(this) acc.

  Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred X.lt In fold (@foldWithAbortPrecompute).
  Proof.
    intros A B i i' f_pre f f' f_lt f_gt.
    move ⇒ [] l H_is_ok_l P H_equiv_P.
    rewrite /fold /foldWithAbortPrecompute /In /this /Raw.In /Raw.fold.
    move ⇒ H_P_f H_f' H_RL H_RG.

    set base := l.
    move : i i'.
    have : (∀ e, InA X.eq e base → InA X.eq e l). {
      rewrite /base //.
    }
    have : sort X.lt base. {
      rewrite Raw.isok_iff /base //.
    }
    clear H_is_ok_l.

    induction base as [| x xs IH]. {
      by simpl.
    }
    move ⇒ H_sort H_in_xxs i i' Pii' /=.

    have [H_sort_xs H_hd_rel {H_sort}] : Sorted X.lt xs ∧ HdRel X.lt x xs. {
      by inversion H_sort.
    }

    move : H_hd_rel.
    rewrite (Raw.ML.Inf_alt x H_sort_xs) ⇒ H_lt_xs.

    have H_x_in_l : InA X.eq x l. {
```

```
    apply H_in_xxs.
    apply InA_cons_hd.
    apply X.eq_equiv.
  }
  have H_in_xs : (∀ e : X.t, InA X.eq e xs → InA X.eq e l). {
    intros e H_in.
    apply H_in_xxs, InA_cons_tl ⇒ //.
  }

  have H_P_next : P (f x (f_pre x) i) (flip f' i' x). {
    rewrite /flip -H_f' //.
    apply H_P_f ⇒ //.
  }

  case_eq (f_gt x (f_pre x) (f x (f_pre x) i)); last first. {
    move ⇒ _.
    apply IH ⇒ //.
  } {
    move ⇒ H_gt.
    suff H_suff : (∀ st, P (f x (f_pre x) i) st →
        P (f x (f_pre x) i) (fold_left (flip f') xs st)). {
        apply H_suff ⇒ //.
    }
    move : H_in_xs H_lt_xs.
    clear IH H_in_xxs H_sort_xs.
    move : (H_RG x _ H_x_in_l H_gt) ⇒ H_RG_x.
    induction xs as [| x' xs' IH']. {
      done.
    } {
      intros H_in_xs H_lt_xs st H_P_st.
      rewrite /=.
      have H_x'_in_l : InA X.eq x' l. {
        apply H_in_xs.
        apply InA_cons_hd, X.eq_equiv.
      }
      apply IH'. {
        intros e H.
        apply H_in_xs, InA_cons_tl ⇒ //.
      } {
        intros e H.
        apply H_lt_xs, InA_cons_tl ⇒ //.
      } {
```

```
        rewrite /flip -H_f' //.
        apply H_RG_x => //.
        apply H_lt_xs.
        apply InA_cons_hd, X.eq_equiv.
      }
    }
  }
Qed.

Include HasFoldWithAbortOps X.
End MakeListSetsWithFoldA.
```

Unsorted Lists without Dups Implementation

```
Module MakeWeakListSetsWithFoldA (X : OrderedType) <: WSetsWithFoldA
with Module E := X.
  Module Raw := MSetWeakList.MakeRaw X.
  Module E := X.
  Include WRaw2SetsOn E Raw.

  Fixpoint foldWithAbortRaw {A B: Type} (f_pre : X.t → B) (f_lt : X.t → B → A →
bool)
    (f: X.t → B → A → A) (f_gt : X.t → B → A → bool) (t : list X.t) (acc : A) : A :=
  match t with
    | nil ⇒ acc
    | x :: xs ⇒ (
        let pre_x := f_pre x in
        let acc := f x (pre_x) acc in
        if (f_gt x pre_x acc) && (f_lt x pre_x acc) then
          acc
        else
          foldWithAbortRaw f_pre f_lt f f_gt xs acc
      )
  end.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt f f_gt t acc :=
    @foldWithAbortRaw A B f_pre f_lt f f_gt t.(this) acc.

  Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred X.lt In fold (@fold-
WithAbortPrecompute).
  Proof.
    intros A B i i' f_pre f f' f_lt f_gt.
    move ⇒ [] l H_is_ok_l P H_P_equiv.
    rewrite /fold /foldWithAbortPrecompute /In /this /Raw.In /Raw.fold.
    move ⇒ H_P_f H_f' H_RL H_RG.
```

```
set base := l.
move : i i'.
have : (∀ e, InA X.eq e base → InA X.eq e l). {
  rewrite /base //.
}
have : NoDupA X.eq base. {
  apply H_is_ok_l.
}
clear H_is_ok_l.

induction base as [| x xs IH]. {
  by simpl.
}
move ⇒ H_nodup_xxs H_in_xxs i i' Pii' /=.

have [H_nin_x_xs H_nodup_xs {H_nodup_xxs}] : ¬ InA X.eq x xs ∧ NoDupA X.eq xs.
{
  by inversion H_nodup_xxs.
}

have H_x_in_l : InA X.eq x l. {
  apply H_in_xxs.
  apply InA_cons_hd.
  apply X.eq_equiv.
}
have H_in_xs : (∀ e : X.t, InA X.eq e xs → InA X.eq e l). {
  intros e H_in.
  apply H_in_xxs, InA_cons_tl ⇒ //.
}

have H_P_next : P (f x (f_pre x) i) (flip f' i' x). {
  rewrite /flip -H_f' //.
  apply H_P_f ⇒ //.
}

case_eq (f_gt x (f_pre x) (f x (f_pre x) i) &&
          f_lt x (f_pre x) (f x (f_pre x) i)); last first. {
  move ⇒ _.
  apply IH ⇒ //.
} {
  move ⇒ /andb_true_iff [H_gt H_lt].
  suff H_suff : (∀ st, P (f x (f_pre x) i) st →
      P (f x (f_pre x) i) (fold_left (flip f') xs st)). {
      apply H_suff ⇒ //.
```

```
}

have H_neq_xs : ∀ e, List.In e xs → X.lt x e ∨ X.lt e x. {
  intros e H_in.
  move : (X.compare_spec x e).
  case (X.compare x e) ⇒ H_cmp; inversion H_cmp. {
    contradict H_nin_x_xs.
    rewrite InA_alt.
    by ∃ e.
  } {
    by left.
  } {
    by right.
  }
}
move : H_in_xs H_neq_xs.
clear IH H_in_xxs H_nodup_xs.
move : (H_RG x _ H_x_in_l H_gt) ⇒ H_RG_x.
move : (H_RL x _ H_x_in_l H_lt) ⇒ H_RL_x.
induction xs as [| x' xs' IH']. {
  done.
} {
  intros H_in_xs H_neq_xs st H_P_st.
  rewrite /=.
  have H_x'_in_xxs' : List.In x' (x' :: xs'). {
    simpl; by left.
  }
  have H_x'_in_l : InA X.eq x' l. {
    apply H_in_xs.
    apply InA_cons_hd, X.eq_equiv.
  }
  apply IH'. {
    intros H.
    apply H_nin_x_xs, InA_cons_tl ⇒ //.
  } {
    intros e H.
    apply H_in_xs, InA_cons_tl ⇒ //.
  } {
    intros e H.
    apply H_neq_xs, in_cons ⇒ //.
  } {
    rewrite /flip -H_f' //.
```

```
          move : (H_neq_xs x' H_x'_in_xxs') => [] H_cmp. {
            apply H_RG_x => //.
          } {
            apply H_RL_x => //.
          }
        }
      }
    }
  Qed.

  Include HasFoldWithAbortOps X.
End MakeWeakListSetsWithFoldA.
```

Chapter 2

Library MSetsExtra.MSetIntervals

2.1 Weak sets implemented by interval lists

This file contains an implementation of the set interface *SetsOn* which uses internally intervals of Z. This allows some large sets, which naturally map to intervals of integers to be represented very efficiently.

Internally intervals of Z are used. However, via an encoding and decoding layer, other types of elements can be handled as well. There are instantiations for Z, N and nat currently. More can be easily added.

```
Require Import MSetInterface OrdersFacts OrdersLists.
Require Import BinNat.
Require Import ssreflect.
Require Import NArith.
Require Import ZArith.
Require Import NOrder.
Require Import DecidableTypeEx.
Module Import NOP := NOrderProp N.
Open Scope Z_scope.
```

2.1.1 Auxiliary stuff

Simple auxiliary lemmata

```
Lemma Z_le_add_r : ∀ (z : Z) (n : N),
  z ≤ z + Z.of_N n.
Proof.
  intros z n.
  suff : (z + 0 ≤ z + Z.of_N n). {
    rewrite Z.add_0_r //.
  }
  apply Zplus_le_compat_l.
  apply N2Z.is_nonneg.
```

```
Qed.
Lemma Z_lt_add_r : ∀ (z : Z) (n : N),
  (n ≠ 0)%N →
  z < z + Z.of_N n.
Proof.
  move ⇒ z n H_neq_0.
  suff : (z + Z.of_N 0 < z + Z.of_N n). {
    rewrite Z.add_0_r //.
  }
  apply Z.add_lt_mono_l, N2Z.inj_lt.
  by apply N.neq_0_lt_0.
Qed.
Lemma Z_lt_le_add_r : ∀ y1 y2 c,
  y1 < y2 →
  y1 ≤ y2 + Z.of_N c.
Proof.
  intros y1 y2 c H.
  apply Z.le_trans with (m := y2). {
    by apply Z.lt_le_incl.
  } {
    apply Z_le_add_r.
  }
Qed.
Lemma Z_to_N_minus_neq_0 : ∀ (x y : Z),
    y < x →
    Z.to_N (x - y) ≠ 0%N.
Proof.
  intros x y H_y_lt.
  apply N.neq_0_lt_0.
  apply N2Z.inj_lt.
  suff H : 0 < x - y. {
    rewrite Z2N.id ⇒ //.
    by apply Z.lt_le_incl.
  }
  by apply Z.lt_0_sub.
Qed.
Lemma add_add_sub_eq : ∀ (x y : Z), (x + (y - x) = y).
Proof.
  intros x y.
  rewrite Z.add_sub_assoc ⇒ //.
  rewrite Z.add_sub_swap Z.sub_diag Z.add_0_l //.
```

```
Qed.
Lemma NoDupA_map {A B} : ∀ (eqA : A → A → Prop) (eqB : B → B → Prop) (f : A →
B) l,
  NoDupA eqA l →
  (∀ x1 x2, List.In x1 l → List.In x2 l →
                    eqB (f x1) (f x2) → eqA x1 x2) →
  NoDupA eqB (map f l).
Proof.
  intros eqA eqB f.
  induction l as [| x xs IH]. {
    move ⇒ _ _; rewrite /=.
    apply NoDupA_nil.
  } {
    move ⇒ H_pre H_eqA_impl.
    have [H_nin_x H_no_dup_xs] : ¬ InA eqA x xs ∧ NoDupA eqA xs. {
      by inversion_clear H_pre.
    }
    simpl.
    apply NoDupA_cons; last first. {
      apply IH ⇒ //.
      intros x1 x2 H_in_x1 H_in_x2 H_eqB.
      apply H_eqA_impl ⇒ //=; by right.
    }
    move ⇒ H_in_map; apply H_nin_x.
    move : H_in_map.
    rewrite !InA_alt ⇒ [[y] [H_eqB_y]].
    rewrite in_map_iff ⇒ [[y'] [H_y_eq] H_y'_in].
    subst.
    ∃ y'.
    split ⇒ //.
    apply H_eqA_impl ⇒ //. {
      by simpl; left.
    } {
      by simpl; right.
    }
  }
Qed.
```

rev_map

rev_map is used for efficiency.

```
Fixpoint rev_map_aux {A B} (f : A → B) (acc : list B) (l
: list A) :=
  match l with
```

```
  | nil ⇒ acc
  | x :: xs ⇒ rev_map_aux f ((f x) :: acc) xs
  end.
Definition rev_map {A B} (f : A → B) (l : list A) : list B := rev_map_aux f nil l.
```

Lemmata about rev_map `Lemma rev_map_aux_alt_def {A B} : ∀ (f : A → B) l acc,`
```
  rev_map_aux f acc l = List.rev_append (List.map f l) acc.
Proof.
  intro f.
  induction l as [|x xs IH]. {
    intros acc.
    by simpl.
  } {
    intros acc.
    rewrite /= IH //.
  }
Qed.

Lemma rev_map_alt_def {A B} : ∀ (f : A → B) l,
  rev_map f l = List.rev (List.map f l).
Proof.
  intros f l.
  rewrite /rev_map rev_map_aux_alt_def -rev_alt //.
Qed.
```

2.1.2 Encoding Elements

We want to encode not only elements of type Z, but other types as well. In order to do so, an encoding / decoding layer is used. This layer is represented by module type ElementEncode. It provides encode and decode function.

```
Module Type ElementEncode.
  Declare Module E : OrderedType.

  Parameter encode : E.t → Z.
  Parameter decode : Z → E.t.
```

Decoding is the inverse of encoding. Notice that the reverse is not demanded. This means that we do need to provide for all integers z an element e with encode $v = z$. `Axiom decode_encode_ok: ∀ (e : E.t),`
```
    decode (encode e) = e.
```

Encoding is compatible with the equality of elements. `Axiom encode_eq : ∀ (e1 e2 : E.t),`
```
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
```

Encoding is compatible with the order of elements. `Axiom encode_lt : ∀ (e1 e2 : E.t),` (Z.lt (encode $e1$) (encode $e2$)) ↔ E.lt $e1$ $e2$.

End ElementEncode.

2.1.3 Set Operations

We represent sets of Z via lists of intervals. The intervals are all in increasing order and non-overlapping. Moreover, we require the most compact representation, i.e. no two intervals can be merged. For example

0-2, 4-4, 6-8 is a valid interval list for the set {0;1;2;4;6;7;8}

In contrast

4-4, 0-2, 6-8 is a invalid because the intervals are not ordered andb 0-2, 4-5, 6-8 is a invalid because it is not compact (0-2, 4–8 is valid).

Intervals we represent by tuples (**Z**, **N**). The tuple (z, c) represents the interval z-(z+c).

We apply the encode function before adding an element to such interval sets and the decode function when checking it. This allows for sets with other element types than Z.

```
Module Ops (Enc : ElementEncode) <: Ops Enc.E.
  Definition elt := Enc.E.t.
  Definition t := list (Z × N).
```

The empty list is trivial to define and check for.

```
  Definition empty : t := nil.
  Definition is_empty (l : t) := match l with nil ⇒ true | _ ⇒ false end.
```

Defining the list of elements, is much more tricky, especially, if it needs to be executable.

```
Lemma acc_pred : ∀ n p, n = Npos p → Acc N.lt n → Acc N.lt (N.pred n).
  Proof.
    intros n p H0 H1.
    apply H1.
    rewrite H0.
    apply N.lt_pred_l.
    discriminate.
  Defined.

  Fixpoint fold_elementsZ_aux {A} (f : A → Z → option A) (acc : A) (x : Z) (c : N) (H
: Acc N.lt c) { struct H } : (bool × A) :=
    match c as c0 return c = c0 → (bool × A) with
    | N0 ⇒ fun _ ⇒ (false, acc)
    | c ⇒ fun Heq ⇒ match (f acc x) with
        | None ⇒ (true, acc)
        | Some acc' ⇒
          fold_elementsZ_aux f acc' (Z.succ x) (N.pred c) (acc_pred _ _ Heq H) end
    end (refl_equal _).

  Definition fold_elementsZ_single {A} f (acc : A) x c := fold_elementsZ_aux f acc x c
(lt_wf_0 _).

  Fixpoint fold_elementsZ {A} f (acc : A) (s : t) : (bool × A) :=
```

```
  match s with
  | nil ⇒ (false, acc)
  | (x, c) :: s' ⇒
    match fold_elementsZ_single f acc x c with
        (false, acc') ⇒ fold_elementsZ f acc' s'
      | (true, acc') ⇒ (true, acc')
    end
  end.
Definition elementsZ (s : t) : list Z :=
  snd (fold_elementsZ (fun l x ⇒ Some (x :: l)) nil s).
Definition elements (s : t) : list elt :=
  rev_map Enc.decode (elementsZ s).
```

membership is easily defined

```
Fixpoint memZ (x : Z) (s : t) :=
  match s with
  | nil ⇒ false
  | (y, c) :: l ⇒
      if (Z.ltb x y) then false else
      if (Z.ltb x (y+Z.of_N c)) then true else
      memZ x l
  end.
Definition mem (x : elt) (s : t) := memZ (Enc.encode x) s.
```

Comparing intervals

```
Inductive interval_compare_result :=
    ICR_before
  | ICR_before_touch
  | ICR_overlap_before
  | ICR_overlap_after
  | ICR_equal
  | ICR_subsume_1
  | ICR_subsume_2
  | ICR_after
  | ICR_after_touch.
Definition interval_compare (i1 i2 : (Z × N)) : interval_compare_result :=
  match (i1, i2) with ((y1, c1), (y2, c2)) ⇒
    let yc2 := (y2+Z.of_N c2) in
    match (Z.compare yc2 y1) with
      | Lt ⇒ ICR_after
      | Eq ⇒ ICR_after_touch
      | Gt ⇒ let yc1 := (y1+Z.of_N c1) in
             match (Z.compare yc1 y2) with
             | Lt ⇒ ICR_before
             | Eq ⇒ ICR_before_touch
```

```
                | Gt ⇒
                        match (Z.compare y1 y2, Z.compare yc1 yc2) with
                        | (Lt, Lt) ⇒ ICR_overlap_before
                        | (Lt, _) ⇒ ICR_subsume_2
                        | (Eq, Lt) ⇒ ICR_subsume_1
                        | (Eq, Gt) ⇒ ICR_subsume_2
                        | (Eq, Eq) ⇒ ICR_equal
                        | (Gt, Gt) ⇒ ICR_overlap_after
                        | (Gt, _) ⇒ ICR_subsume_1
                        end
                end
    end
  end.

Definition interval_1_compare (y1 : Z) (i : (Z × N)) : interval_compare_result :=
  match i with (y2, c2) ⇒
    let yc2 := (y2+Z.of_N c2) in
    match (Z.compare yc2 y1) with
      | Lt ⇒ ICR_after
      | Eq ⇒ ICR_after_touch
      | Gt ⇒ match (Z.compare (Z.succ y1) y2) with
                | Lt ⇒ ICR_before
                | Eq ⇒ ICR_before_touch
                | Gt ⇒ ICR_subsume_1
                end
    end
  end.

Fixpoint compare (s1 s2 : t) :=
  match (s1, s2) with
    | (nil, nil) ⇒ Eq
    | (nil, _ :: _) ⇒ Lt
    | (_ :: _, nil) ⇒ Gt
    | ((y1, c1)::s1', (y2, c2)::s2') ⇒
      match (Z.compare y1 y2) with
        | Lt ⇒ Lt
        | Gt ⇒ Gt
        | Eq ⇒ match N.compare c1 c2 with
                   | Lt ⇒ Lt
                   | Gt ⇒ Gt
                   | Eq ⇒ compare s1' s2'
                 end
      end
  end.
```

Auxiliary functions for inserting at front and merging intervals `Definition merge_interval_size (x1 : Z) (c1 : N) (x2 : Z) (c2 : N) : N :=`
```
  (N.max c1 (Z.to_N (x2 + Z.of_N c2 - x1))).
```

```
Fixpoint insert_interval_begin (x : Z) (c : N) (l : t) :=
  match l with
  | nil ⇒ (x,c)::nil
  | (y, c')::l' ⇒
      match (Z.compare (x + Z.of_N c) y) with
      | Lt ⇒ (x, c) :: l
      | Eq ⇒ (x, (c+c')%N) :: l'
      | Gt ⇒ insert_interval_begin x (merge_interval_size x c y c') l'
      end
  end.
```

adding an element needs to be defined carefully again in order to generate efficient code

```
Fixpoint addZ_aux (acc : list (Z × N)) (x : Z) (s : t) :=
  match s with
  | nil ⇒ List.rev' ((x, (1%N))::acc)
  | (y, c) :: l ⇒
     match (interval_1_compare x (y,c)) with
       | ICR_before ⇒ List.rev_append ((x, (1%N))::acc) s
       | ICR_before_touch ⇒ List.rev_append ((x, N.succ c)::acc) l
       | ICR_after ⇒ addZ_aux ((y,c) :: acc) x l
       | ICR_after_touch ⇒ List.rev_append acc (insert_interval_begin y (N.succ c) l)
       | _ ⇒ List.rev_append ((y, c)::acc) l
     end
  end.
```

```
Definition addZ x s := addZ_aux nil x s.
Definition add x s := addZ (Enc.encode x) s.
```

add_list is a simple extension to add many elements. This is used to define the function from_elements. `Definition add_list (l : list elt) (s : t) : t :=`
```
   List.fold_left (fun s x ⇒ add x s) l s.
```

```
Definition from_elements (l : list elt) : t := add_list l empty.
```

singleton is trivial to define `Definition singleton (x : elt) : t := (Enc.encode x, 1%N) :: nil.`

```
Lemma singleton_alt_def : ∀ x, singleton x = add x empty.
Proof. by []. Qed.
```

removing needs to be done with code extraction in mind again. `Definition insert_intervalZ_guarded (x : Z) (c : N) s :=`
```
   if (N.eqb c 0) then s else (x, c) :: s.
```

```
Fixpoint removeZ_aux (acc : list (Z × N)) (x : Z) (s : t) : t :=
```

```
    match s with
    | nil ⇒ List.rev' acc
    | (y, c) :: l ⇒
        if (Z.ltb x y) then List.rev_append acc s else
        if (Z.ltb x (y+Z.of_N c)) then (
          List.rev_append (insert_intervalZ_guarded (Z.succ x)
            (Z.to_N ((y+Z.of_N c)- (Z.succ x)))
           (insert_intervalZ_guarded y (Z.to_N (x-y)) acc)) l
        ) else removeZ_aux ((y,c)::acc) x l
    end.
```

```
Definition removeZ (x : Z) (s : t) : t := removeZ_aux nil x s.
Definition remove (x : elt) (s : t) : t := removeZ (Enc.encode x) s.
```

```
Definition remove_list (l : list elt) (s : t) : t :=
  List.fold_left (fun s x ⇒ remove x s) l s.
```

union

```
Fixpoint union_aux (s1 : t) :=
    fix aux (s2 : t) (acc : list (Z × N)) :=
    match (s1, s2) with
    | (nil, _) ⇒ List.rev_append acc s2
    | (_, nil) ⇒ List.rev_append acc s1
    | ((y1, c1) :: l1, (y2, c2) :: l2) ⇒
        match (interval_compare (y1, c1) (y2,c2)) with
          | ICR_before ⇒ union_aux l1 s2 ((y1, c1)::acc)
          | ICR_before_touch ⇒
              union_aux l1 (
               insert_interval_begin y1 ((c1+c2)%N) l2) acc
          | ICR_after ⇒ aux l2 ((y2, c2)::acc)
          | ICR_after_touch ⇒ union_aux l1 (
              insert_interval_begin y2 ((c1+c2)%N) l2) acc
          | ICR_overlap_before ⇒
              union_aux l1 (insert_interval_begin y1 (merge_interval_size y1 c1 y2 c2) l2)
acc
          | ICR_overlap_after ⇒
              union_aux l1 (insert_interval_begin y2 (merge_interval_size y2 c2 y1 c1) l2)
acc
          | ICR_equal ⇒ union_aux l1 s2 acc
          | ICR_subsume_1 ⇒ union_aux l1 s2 acc
          | ICR_subsume_2 ⇒ aux l2 acc
        end
    end.
```

```
Definition union s1 s2 := union_aux s1 s2 nil.
```

diff

```
Fixpoint diff_aux (y2 : Z) (c2 : N) (acc : list (Z × N)) (s : t) : (list (Z × N) × t) :=
  match s with
  | nil ⇒ (acc, nil)
  | ((y1, c1) :: l1) ⇒
      match (interval_compare (y1, c1) (y2,c2)) with
        | ICR_before ⇒ diff_aux y2 c2 ((y1, c1)::acc) l1
        | ICR_before_touch ⇒ diff_aux y2 c2 ((y1, c1)::acc) l1
        | ICR_after ⇒ (acc, s)
        | ICR_after_touch ⇒ (acc, s)
        | ICR_overlap_before ⇒ diff_aux y2 c2 ((y1, Z.to_N (y2 - y1))::acc) l1
        | ICR_overlap_after ⇒ (acc, (y2+Z.of_N c2, Z.to_N ((y1 + Z.of_N c1) - (y2 +
Z.of_N c2))) :: l1)
        | ICR_equal ⇒ (acc, l1)
        | ICR_subsume_1 ⇒ diff_aux y2 c2 acc l1
        | ICR_subsume_2 ⇒ ((insert_intervalZ_guarded y1
              (Z.to_N (y2 - y1)) acc),
            insert_intervalZ_guarded (y2+Z.of_N c2) (Z.to_N ((y1 + Z.of_N c1) - (y2 +
Z.of_N c2))) l1)
      end
  end.

Fixpoint diff_aux2 (acc : list (Z × N)) (s1 s2 : t) : (list (Z × N)) :=
  match (s1, s2) with
  | (nil, _) ⇒ rev_append acc s1
  | (_, nil) ⇒ rev_append acc s1
  | (_, (y2, c2) :: l2) ⇒
    match diff_aux y2 c2 acc s1 with
      (acc', s1') ⇒ diff_aux2 acc' s1' l2
    end
  end.

Definition diff s1 s2 := diff_aux2 nil s1 s2.

 subset    Fixpoint subset (s1 : t) :=
  fix aux (s2 : t) :=
  match (s1, s2) with
  | (nil, _) ⇒ true
  | (_ :: _, nil) ⇒ false
  | ((y1, c1) :: l1, (y2, c2) :: l2) ⇒
      match (interval_compare (y1, c1) (y2,c2)) with
        | ICR_before ⇒ false
        | ICR_before_touch ⇒ false
        | ICR_after ⇒ aux l2
        | ICR_after_touch ⇒ false
        | ICR_overlap_before ⇒ false
```

```
        | ICR_overlap_after ⇒ false
        | ICR_equal ⇒ subset l1 l2
        | ICR_subsume_1 ⇒ subset l1 s2
        | ICR_subsume_2 ⇒ false
      end
  end.
```

equal

```
Fixpoint equal (s s' : t) : bool := match s, s' with
  | nil, nil ⇒ true
  | ((x , cx) :: xs), ((y , cy) :: ys) ⇒ andb (Z.eqb x y) (andb (N.eqb cx cy) (equal xs ys))
  | _, _ ⇒ false
end.
```

inter

```
Fixpoint inter_aux (y2 : Z) (c2 : N) (acc : list (Z × N)) (s : t) : (list (Z × N) × t) :=
  match s with
  | nil ⇒ (acc , nil)
  | ((y1 , c1) :: l1) ⇒
      match (interval_compare (y1 , c1) (y2 , c2)) with
        | ICR_before ⇒ inter_aux y2 c2 acc l1
        | ICR_before_touch ⇒ inter_aux y2 c2 acc l1
        | ICR_after ⇒ (acc , s)
        | ICR_after_touch ⇒ (acc , s)
        | ICR_overlap_before ⇒ inter_aux y2 c2 ((y2 , Z.to_N (y1 + Z.of_N c1 - y2)) :: acc) l1
        | ICR_overlap_after ⇒ ((y1 , Z.to_N (y2 + Z.of_N c2 - y1)) :: acc , s)
        | ICR_equal ⇒ ((y1 , c1) :: acc , l1)
        | ICR_subsume_1 ⇒ inter_aux y2 c2 ((y1 , c1) :: acc) l1
        | ICR_subsume_2 ⇒ ((y2 , c2) :: acc , s)
      end
  end.

Fixpoint inter_aux2 (acc : list (Z × N)) (s1 s2 : t) : (list (Z × N)) :=
  match (s1 , s2) with
  | (nil , _) ⇒ List.rev' acc
  | (_ , nil) ⇒ List.rev' acc
  | (_ , (y2 , c2) :: l2) ⇒
    match inter_aux y2 c2 acc s1 with
      (acc' , s1') ⇒ inter_aux2 acc' s1' l2
    end
  end.

Definition inter s1 s2 := inter_aux2 nil s1 s2.
```

Partition and filter

```
Definition partitionZ_fold_insert
```

```
              (cur : option (Z × N)) (x : Z) :=
  match cur with
     None ⇒ (x, 1%N)
   | Some (y, c) ⇒ (y, N.succ c)
  end.

Definition partitionZ_fold_skip (acc : list (Z × N))
              (cur : option (Z × N)) : (list (Z × N)) :=
  match cur with
     None ⇒ acc
   | Some yc ⇒ yc::acc
  end.

Definition partitionZ_fold_fun f st (x : Z) :=
  match st with ((acc_t, c_t), (acc_f, c_f)) ⇒
    if (f x) then
      ((acc_t, Some (partitionZ_fold_insert c_t x)),
       (partitionZ_fold_skip acc_f c_f, None))
    else
      ((partitionZ_fold_skip acc_t c_t, None),
       (acc_f, Some (partitionZ_fold_insert c_f x)))
  end.

Definition partitionZ_single_aux f st (x : Z) (c : N) :=
  snd (fold_elementsZ_single (fun st x ⇒ Some (partitionZ_fold_fun f st x)) st x c).

Definition partitionZ_single f acc_t acc_f x c :=
  match partitionZ_single_aux f ((acc_t, None), (acc_f, None)) x c with
  | ((acc_t, c_t), (acc_f, c_f)) ⇒
      (partitionZ_fold_skip acc_t c_t,
       partitionZ_fold_skip acc_f c_f)
  end.

Fixpoint partitionZ_aux acc_t acc_f f s :=
  match s with
  | nil ⇒ (List.rev acc_t, List.rev acc_f)
  | (y, c) :: s' ⇒
    match partitionZ_single f acc_t acc_f y c with
    | (acc_t', acc_f') ⇒ partitionZ_aux acc_t' acc_f' f s'
    end
  end.

Definition partitionZ := partitionZ_aux nil nil.

Definition partition (f : elt → bool) : t → (t × t) :=
  partitionZ (fun z ⇒ f (Enc.decode z)).

Definition filterZ_fold_fun f st (x : Z) :=
```

```
  match st with (acc_t, c_t) ⇒
    if (f x) then
      (acc_t, Some (partitionZ_fold_insert c_t x))
    else
      (partitionZ_fold_skip acc_t c_t, None)
  end.

Definition filterZ_single_aux f st (x : Z) (c : N) :=
  snd (fold_elementsZ_single (fun st x ⇒ Some (filterZ_fold_fun f st x)) st x c).

Definition filterZ_single f acc x c :=
  match filterZ_single_aux f (acc, None) x c with
  | (acc, c) ⇒
      (partitionZ_fold_skip acc c)
  end.

Fixpoint filterZ_aux acc f s :=
  match s with
  | nil ⇒ (List.rev acc)
  | (y, c) :: s' ⇒
    filterZ_aux (filterZ_single f acc y c) f s'
  end.

Definition filterZ := filterZ_aux nil.

Definition filter (f : elt → bool) : t → t :=
  filterZ (fun z ⇒ f (Enc.decode z)).
```

Simple wrappers

```
Definition fold {B : Type} (f : elt → B → B) (s : t) (i : B) : B :=
  snd (fold_elementsZ (fun b z ⇒ Some (f (Enc.decode z) b)) i s).

Definition for_all (f : elt → bool) (s : t) : bool :=
  snd (fold_elementsZ (fun b z ⇒
    if b then
      Some (f (Enc.decode z))
    else None) true s).

Definition exists_ (f : elt → bool) (s : t) : bool :=
  snd (fold_elementsZ (fun b z ⇒
    if b then
      None
    else Some (f (Enc.decode z))) false s).

Fixpoint cardinalN c (s : t) : N := match s with
  | nil ⇒ c
  | (_,cx)::xs ⇒ cardinalN (c + cx)%N xs
end.
```

```
Definition cardinal (s : t) : nat := N.to_nat (cardinalN (0%N) s).

Definition min_eltZ (s : t) : option Z :=
  match s with
  | nil ⇒ None
  | (x , _) :: _ ⇒ Some x
  end.

Definition min_elt (s : t) : option elt :=
  match (min_eltZ s) with
  | None ⇒ None
  | Some x ⇒ Some (Enc.decode x)
  end.

Definition choose := min_elt.

Fixpoint max_eltZ (s : t) : option Z :=
  match s with
  | nil ⇒ None
  | (x , c) :: nil ⇒ Some (Z.pred (x + Z.of_N c))
  | (x , _) :: s' ⇒ max_eltZ s'
  end.

Definition max_elt (s : t) : option elt :=
  match (max_eltZ s) with
  | None ⇒ None
  | Some x ⇒ Some (Enc.decode x)
  end.

End Ops.
```

2.1.4 Raw Module

Following the idea of *MSetInterface.RawSets*, we first define a module Raw proves all the required properties with respect to an explicitly provided invariant. In a next step, this invariant is then moved into the set type. This allows to instantiate the *WSetsOn* interface.

```
Module Raw (Enc : ElementEncode).
  Include (Ops Enc).
```

Defining invariant IsOk

```
  Definition is_encoded_elems_list (l : list Z) : Prop :=
    (∀ x, List.In x l → ∃ e, Enc.encode e = x).

  Definition interval_list_elements_greater (x : Z) (l : t) : bool :=
    match l with
      | nil ⇒ true
      | (y, _)::_ ⇒ Z.ltb x y
```

```
    end.
  Fixpoint interval_list_invariant (l : t) :=
   match l with
   | nil => true
   | (x, c) :: l' =>
        interval_list_elements_greater (x + (Z.of_N c)) l' && negb (N.eqb c 0) && interval_list_invariant
l'
    end.
  Definition IsOk s := (interval_list_invariant s = true /\ is_encoded_elems_list (elementsZ
s)).
```

Defining notations

```
  Section ForNotations.
    Class Ok (s:t) : Prop := ok : IsOk s.
    Hint Resolve @ok.
    Hint Unfold Ok.
    Instance IsOk_Ok s `(Hs : IsOk s) : Ok s := { ok := Hs }.
    Definition In x s := (SetoidList.InA Enc.E.eq x (elements s)).
    Definition InZ x s := (List.In x (elementsZ s)).
    Definition Equal s s' := forall a : elt, In a s <-> In a s'.
    Definition Subset s s' := forall a : elt, In a s -> In a s'.
    Definition Empty s := forall a : elt, ~ In a s.
    Definition For_all (P : elt -> Prop) s := forall x, In x s -> P x.
    Definition Exists (P : elt -> Prop) (s : t) := exists x, In x s /\ P x.
  End ForNotations.
```

elements list properties

The functions elementsZ, elementsZ_single, elements and elements_single are crucial and used everywhere. Therefore, we first establish a few properties of these important functions.

```
  Lemma elementsZ_nil : (elementsZ (nil : t) = nil).
  Proof. done. Qed.

  Lemma elements_nil : (elements (nil : t) = nil).
  Proof. done. Qed.

  Definition elementsZ_single (x:Z) (c:N) :=
      List.rev' (N.peano_rec (fun _ => list Z)
                      nil (fun n ls => (x+Z.of_N n)%Z :: ls) c).

  Definition elements_single x c :=
    List.map Enc.decode (elementsZ_single x c).
```

```
Lemma elementsZ_single_base : ∀ x,
  elementsZ_single x (0%N) = nil.
Proof. done. Qed.

Lemma elementsZ_single_succ : ∀ x c,
  elementsZ_single x (N.succ c) =
  elementsZ_single x c ++ (x+Z.of_N c)::nil.
Proof.
  intros x c.
  rewrite /elementsZ_single N.peano_rec_succ /rev' -!rev_alt //.
Qed.

Lemma elementsZ_single_add : ∀ x c2 c1,
  elementsZ_single x (c1 + c2)%N =
  elementsZ_single x c1 ++ elementsZ_single (x+Z.of_N c1) c2.
Proof.
  intros x.
  induction c2 as [| c2' IH] using N.peano_ind. {
    move ⇒ c1.
    rewrite elementsZ_single_base /= app_nil_r N.add_0_r //.
  } {
    move ⇒ c1.
    rewrite N.add_succ_r !elementsZ_single_succ IH app_assoc N2Z.inj_add Z.add_assoc
//.
  }
Qed.

Lemma elementsZ_single_succ_front : ∀ x c,
  elementsZ_single x (N.succ c) =
  x :: elementsZ_single (Z.succ x) c.
Proof.
  intros x c.
  rewrite -N.add_1_l elementsZ_single_add.
  rewrite N.one_succ elementsZ_single_succ elementsZ_single_base /= Z.add_0_r.
  by rewrite Z.add_1_r.
Qed.

Lemma In_elementsZ_single : ∀ c y x,
  List.In y (elementsZ_single x c) ↔
  (x ≤ y) ∧ (y < (x+Z.of_N c)).
Proof.
  induction c as [| c' IH] using N.peano_ind. {
    intros y x.
    rewrite elementsZ_single_base Z.add_0_r /=.
    omega.
```

```
  } {
    intros y x.
    rewrite elementsZ_single_succ in_app_iff IH /= N2Z.inj_succ Z.add_succ_r Z.lt_succ_r.
    split. {
      move => [ | []] //. {
        move => [H_x_le H_y_le].
        omega.
      } {
        move => <-.
        split.
          - by apply Z_le_add_r.
          - by apply Z.le_refl.
      }
    } {
      move => [H_x_le] H_y_lt.
      omega.
    }
  }
Qed.

Lemma In_elementsZ_single1 : ∀ y x,
  List.In y (elementsZ_single x (1%N)) <->
  (x = y).
Proof.
  intros y x.
  rewrite In_elementsZ_single /= Z.add_1_r Z.lt_succ_r.
  omega.
Qed.

Lemma length_elementsZ_single : ∀ cx x,
  length (elementsZ_single x cx) = N.to_nat cx.
Proof.
  induction cx as [| cx' IH] using N.peano_ind. {
    by simpl.
  } {
    intros x.
    rewrite elementsZ_single_succ_front /=.
    rewrite IH N2Nat.inj_succ //.
  }
Qed.

Lemma fold_elementsZ_aux_irrel {A} :
  ∀ f c (acc : A) x H1 H2,
    fold_elementsZ_aux f acc x c H1 =
    fold_elementsZ_aux f acc x c H2.
```

```
Proof.
  intros f c.
  induction c as [c IH] using (well_founded_ind lt_wf_0).
  case_eq c. {
    intros H_c acc x; case; intro H_H1; case; intro H_H2.
    reflexivity.
  } {
    intros p H_c acc x; case; intro H_H1; case; intro H_H2.
    unfold fold_elementsZ_aux; fold (@fold_elementsZ_aux A).
    case (f acc x) => // acc'.
    apply IH.
    rewrite H_c.
    apply N.lt_pred_l.
    discriminate.
  }
Qed.

Lemma fold_elementsZ_single_pos {A} : ∀ f (acc : A) x p,
  fold_elementsZ_single f acc x (N.pos p) =
  match f acc x with
  | Some acc' ⇒
      fold_elementsZ_single f acc' (Z.succ x)
       (N.pred (N.pos p))
  | None ⇒ (true, acc)
  end.
Proof.
  intros f acc x p.
  unfold fold_elementsZ_single.
  unfold fold_elementsZ_aux.
  case: (lt_wf_0 _).
  fold (@fold_elementsZ_aux A).
  intro.
  case (f acc x) ⇒ // acc'.
  apply fold_elementsZ_aux_irrel.
Qed.

Lemma fold_elementsZ_single_zero {A} : ∀ f (acc : A) x,
    fold_elementsZ_single f acc x (0%N) = (false, acc).
Proof.
  intros f acc x.
  unfold fold_elementsZ_single.
  case (lt_wf_0 (0%N)); intro.
  unfold fold_elementsZ_aux.
  reflexivity.
```

```
Qed.

Lemma fold_elementsZ_single_succ {A} : ∀ f (acc : A) x c,
  fold_elementsZ_single f acc x (N.succ c) =
  match f acc x with
    | Some acc' ⇒
        fold_elementsZ_single f acc' (Z.succ x) c
    | None ⇒ (true, acc)
  end.
Proof.
  intros f acc x c.
  case c. {
    by rewrite fold_elementsZ_single_pos.
  } {
    intro p; simpl.
    rewrite fold_elementsZ_single_pos.
    case (f acc x) ⇒ // acc' /=.
    by rewrite Pos.pred_N_succ.
  }
Qed.

Fixpoint fold_opt {A B} f (acc : A) (bs : list B) : (bool × A) :=
  match bs with
    | nil ⇒ (false, acc)
    | (b :: bs') ⇒
      match f acc b with
      | Some acc' ⇒ fold_opt f acc' bs'
      | None ⇒ (true, acc)
      end
  end.

Lemma fold_opt_list_cons : ∀ {A} (bs : list A) (acc : list A),
  fold_opt (fun l x ⇒ Some (x :: l)) acc bs =
  (false, List.rev bs ++ acc).
Proof.
  induction bs as [| b bs' IH] ⇒ acc. {
    by simpl.
  } {
    rewrite /= IH -app_assoc //.
  }
Qed.

Lemma fold_opt_app {A B} : ∀ f (acc : A) (l1 l2 : list B),
  fold_opt f acc (l1 ++ l2) =
  (let (ab, acc') := fold_opt f acc l1 in
```

```
    if ab then (true, acc') else fold_opt f acc' l2).
Proof.
  intros f acc l1 l2.
  move : acc.
  induction l1 as [| b l1' IH] ⇒ acc. {
    rewrite app_nil_l //.
  } {
    rewrite /=.
    case (f acc b); last done.
    intro acc'.
    rewrite IH //.
  }
Qed.

Lemma fold_elementsZ_single_alt_def {A} : ∀ f c (acc : A) x,
    fold_elementsZ_single f acc x c =
    fold_opt f acc (elementsZ_single x c).
Proof.
  intro f.
  induction c as [| c' IH] using N.peano_ind. {
    intros acc x.
    rewrite fold_elementsZ_single_zero
            elementsZ_single_base /fold_opt //.
  } {
    intros acc x.
    rewrite fold_elementsZ_single_succ
            elementsZ_single_succ_front /=.
    case (f acc x); last reflexivity.
    intro acc'.
    apply IH.
  }
Qed.

Lemma fold_elementsZ_nil {A} : ∀ f (acc : A),
    fold_elementsZ f acc nil = (false, acc).
Proof. done. Qed.

Lemma fold_elementsZ_cons {A} : ∀ f (acc : A) y c s,
  fold_elementsZ f acc ((y, c)::s) =
  (let (ab, acc') := fold_elementsZ_single f acc y c in
    if ab then (true, acc') else fold_elementsZ f acc' s).
Proof.
  intros f acc y c s.
  done.
```

```
Qed.

Lemma fold_elementsZ_alt_def_aux : ∀ (s : t) base,
  (snd (fold_elementsZ
    (fun (l : list Z) (x : Z) ⇒ Some (x :: l)) base s)) =
  elementsZ s ++ base.
Proof.
  induction s as [|| [y1 c1] s' IH] ⇒ base. {
    rewrite elementsZ_nil /fold_elementsZ /fold_opt /snd
      app_nil_l //.
  } {
    rewrite /elementsZ !fold_elementsZ_cons.
    rewrite !fold_elementsZ_single_alt_def !fold_opt_list_cons.
    rewrite !IH app_nil_r app_assoc //.
  }
Qed.

Lemma fold_elementsZ_alt_def {A} : ∀ f s (acc : A),
   fold_elementsZ f acc s =
   fold_opt f acc (rev (elementsZ s)).
Proof.
  intro f.
  induction s as [|| [y1 c1] s' IH] ⇒ acc. {
    by simpl.
  } {
    rewrite /elementsZ !fold_elementsZ_cons.
    rewrite !fold_elementsZ_single_alt_def
            fold_opt_list_cons app_nil_r
            fold_elementsZ_alt_def_aux rev_app_distr
            rev_involutive fold_opt_app.
    case (fold_opt f acc (elementsZ_single y1 c1)).
    move ⇒ [] //.
  }
Qed.

Lemma elementsZ_cons : ∀ x c s, elementsZ (((x, c) :: s) : t) =
   ((elementsZ s) ++ (List.rev (elementsZ_single x c))).
Proof.
  intros x c s.
  rewrite /elementsZ fold_elementsZ_cons
          !fold_elementsZ_alt_def
          fold_elementsZ_single_alt_def.
  rewrite !fold_opt_list_cons.
  rewrite !app_nil_r !rev_involutive /=.
```

```
  rewrite fold_elementsZ_alt_def_aux //.
Qed.

Lemma elements_cons : ∀ x c s, elements (((x, c) :: s) : t) =
    ((elements_single x c) ++ elements s).
Proof.
  intros x c s.
  rewrite /elements /elements_single elementsZ_cons.
  rewrite !rev_map_alt_def map_app rev_app_distr map_rev rev_involutive //.
Qed.

Lemma elementsZ_app : ∀ (s1 s2 : t), elementsZ (s1 ++ s2) =
    ((elementsZ s2) ++ (elementsZ s1)).
Proof.
  induction s1 as [|[x1 c1] s1 IH1]. {
    move ⇒ s2.
    rewrite elementsZ_nil app_nil_r //.
  }
  move ⇒ s2.
  rewrite -app_comm_cons !elementsZ_cons IH1 -app_assoc //.
Qed.

Lemma InZ_nil : ∀ y, InZ y nil ↔ False.
Proof.
  intro y.
  done.
Qed.

Lemma InZ_cons : ∀ y x c s, InZ y (((x, c) :: s) : t) ↔
    List.In y (elementsZ_single x c) ∨ InZ y s.
Proof.
  intros y x c s.
  rewrite /InZ elementsZ_cons in_app_iff -in_rev.
  firstorder.
Qed.

Lemma InZ_app : ∀ s1 s2 y,
    InZ y (s1 ++ s2) ↔ InZ y s1 ∨ InZ y s2.
Proof.
  intros s1 s2 y.
  rewrite /InZ elementsZ_app in_app_iff.
  tauto.
Qed.

Lemma InZ_rev : ∀ s y,
    InZ y (List.rev s) ↔ InZ y s.
Proof.
```

```
  intros s x.
  rewrite /InZ.
  induction s as [| [y c] s' IH]; first done.
  simpl.
  rewrite elementsZ_app in_app_iff IH.
  rewrite !elementsZ_cons !in_app_iff elementsZ_nil
          -!in_rev /=.
  tauto.
Qed.

Lemma In_elementsZ_single_dec : ∀ y x c,
  {List.In y (elementsZ_single x c)} +
  {¬ List.In y (elementsZ_single x c)}.
Proof.
  intros y x c.
  case (Z_le_dec x y); last first. {
    right; rewrite In_elementsZ_single; tauto.
  }
  case (Z_lt_dec y (x + Z.of_N c)); last first. {
    right; rewrite In_elementsZ_single; tauto.
  }
  left; rewrite In_elementsZ_single; tauto.
Qed.

Lemma InZ_dec : ∀ y s,
   {InZ y s} + {¬InZ y s}.
Proof.
  intros y.
  induction s as [| [x c] s IH]. {
    by right.
  }
  move : IH ⇒ [] IH. {
    by left; rewrite InZ_cons; right.
  }
  case (In_elementsZ_single_dec y x c). {
    by left; rewrite InZ_cons; left.
  } {
    by right; rewrite InZ_cons; tauto.
  }
Qed.

Lemma In_elementsZ_single_hd : ∀ (c : N) x, (c ≠ 0)%N → List.In x (elementsZ_single x
c).
Proof.
  intros c x H_c_neq.
```

```
    rewrite In_elementsZ_single.
    split. {
      apply Z.le_refl.
    } {
      apply Z.lt_add_pos_r.
      have -> : 0 = Z.of_N (0%N) by [].
      apply N2Z.inj_lt.
      by apply N.neq_0_lt_0.
    }
  Qed.
```

comparing intervals

```
  Ltac Z_named_compare_cases H := match goal with
    | [ ⊢ context [Z.compare ?z1 ?z2] ] ⇒
      case_eq (Z.compare z1 z2); [move ⇒ /Z.compare_eq_iff | move ⇒ /Z.compare_lt_iff |
move ⇒ /Z.compare_gt_iff]; move ⇒ H //
  end.

  Ltac Z_compare_cases := let H := fresh "H" in Z_named_compare_cases H.

  Lemma interval_compare_elim : ∀ (y1 : Z) (c1 : N) (y2 : Z) (c2 : N),
    match (interval_compare (y1, c1) (y2, c2)) with
      | ICR_before ⇒ (y1 + Z.of_N c1) < y2
      | ICR_before_touch ⇒ (y1 + Z.of_N c1) = y2
      | ICR_after ⇒ (y2 + Z.of_N c2) < y1
      | ICR_after_touch ⇒ (y2 + Z.of_N c2) = y1
      | ICR_equal ⇒ (y1 = y2) ∧ (c1 = c2)
      | ICR_overlap_before ⇒ (y1 < y2) ∧ (y2 < y1 + Z.of_N c1) ∧ (y1 + Z.of_N c1 < y2
+ Z.of_N c2)
      | ICR_overlap_after ⇒ (y2 < y1) ∧ (y1 < y2 + Z.of_N c2) ∧ (y2 + Z.of_N c2 < y1 +
Z.of_N c1)
      | ICR_subsume_1 ⇒ (y2 ≤ y1) ∧ (y1 + Z.of_N c1 ≤ y2 + Z.of_N c2) ∧ (y2 < y1 ∨
y1 + Z.of_N c1 < y2 + Z.of_N c2)
      | ICR_subsume_2 ⇒ (y1 ≤ y2) ∧ (y2 + Z.of_N c2 ≤ y1 + Z.of_N c1) ∧ (y1 < y2 ∨
y2 + Z.of_N c2 < y1 + Z.of_N c1)
    end.
  Proof.
    intros y1 c1 y2 c2.
    rewrite /interval_compare.
    (repeat Z_compare_cases); subst; repeat split;
       try (by apply Z.eq_le_incl);
       try (by apply Z.lt_le_incl);
       try (by left); try (by right).
```

```
  apply Z.add_reg_l in H2.
  by apply N2Z.inj.
Qed.

Lemma interval_compare_swap : ∀ (y1 : Z) (c1 : N) (y2 : Z) (c2 : N),
  (c1 ≠ 0%N) ∨ (c2 ≠ 0%N) →
  interval_compare (y2, c2) (y1, c1) =
  match (interval_compare (y1, c1) (y2, c2)) with
    | ICR_before ⇒ ICR_after
    | ICR_before_touch ⇒ ICR_after_touch
    | ICR_after ⇒ ICR_before
    | ICR_after_touch ⇒ ICR_before_touch
    | ICR_equal ⇒ ICR_equal
    | ICR_overlap_before ⇒ ICR_overlap_after
    | ICR_overlap_after ⇒ ICR_overlap_before
    | ICR_subsume_1 ⇒ ICR_subsume_2
    | ICR_subsume_2 ⇒ ICR_subsume_1
  end.
Proof.
  intros y1 c1 y2 c2 H_c12_neq_0.
  rewrite /interval_compare.
  move : (Z.compare_antisym y1 y2) ⇒ →.
  move : (Z.compare_antisym (y1 + Z.of_N c1) (y2 + Z.of_N c2)) ⇒ →.
  have H_suff : y1 + Z.of_N c1 ≤ y2 → y2 + Z.of_N c2 ≤ y1 → False. {
    move ⇒ H1 H2.
    case H_c12_neq_0 ⇒ H_c_neq_0. {
      suff : (y1 + Z.of_N c1 ≤ y1). {
        apply Z.nle_gt.
        by apply Z_lt_add_r.
      }
      eapply Z.le_trans; eauto.
      eapply Z.le_trans; eauto.
      apply Z_le_add_r.
    } {
      suff : (y2 + Z.of_N c2 ≤ y2). {
        apply Z.nle_gt.
        by apply Z_lt_add_r.
      }
      eapply Z.le_trans; eauto.
      eapply Z.le_trans; eauto.
      apply Z_le_add_r.
    }
  }
```

```
  repeat Z_compare_cases. {
    exfalso; apply H_suff.
      - by rewrite H; apply Z.le_refl.
      - by rewrite H0; apply Z.le_refl.
  } {
    exfalso; apply H_suff.
      - by rewrite H; apply Z.le_refl.
      - by apply Z.lt_le_incl.
  } {
    exfalso; apply H_suff.
      - by apply Z.lt_le_incl.
      - by rewrite H0; apply Z.le_refl.
  } {
    exfalso; apply H_suff.
      - by apply Z.lt_le_incl.
      - by apply Z.lt_le_incl.
  }
Qed.

Lemma interval_1_compare_alt_def : ∀ (y : Z) (i : (Z × N)),
  interval_1_compare y i = match (interval_compare (y, (1%N)) i) with
    | ICR_equal ⇒ ICR_subsume_1
    | ICR_subsume_1 ⇒ ICR_subsume_1
    | ICR_subsume_2 ⇒ ICR_subsume_1
    | r ⇒ r
  end.
Proof.
  move ⇒ y1 [y2 c2].
  rewrite /interval_1_compare /interval_compare.
  replace (y1 + Z.of_N 1) with (Z.succ y1); last done.
  repeat Z_compare_cases. {
    contradict H1.
    by apply Zle_not_lt, Zlt_succ_le.
  } {
    contradict H.
    by apply Zle_not_lt, Zlt_succ_le.
  }
Qed.

Lemma interval_1_compare_elim : ∀ (y1 : Z) (y2 : Z) (c2 : N),
  match (interval_1_compare y1 (y2, c2)) with
    | ICR_before ⇒ Z.succ y1 < y2
    | ICR_before_touch ⇒ y2 = Z.succ y1
    | ICR_after ⇒ (y2 + Z.of_N c2) < y1
```

```
    | ICR_after_touch ⇒ (y2 + Z.of_N c2) = y1
    | ICR_equal ⇒ False
    | ICR_overlap_before ⇒ False
    | ICR_overlap_after ⇒ False
    | ICR_subsume_1 ⇒ (c2 = 0%N) ∨ ((y2 ≤ y1) ∧ (y1 < y2 + Z.of_N c2))
    | ICR_subsume_2 ⇒ False
  end.
Proof.
  intros y1 y2 c2.
  move : (interval_compare_elim y1 (1%N) y2 c2).
  rewrite interval_1_compare_alt_def.
  have H_succ: ∀ z, z + Z.of_N 1 = Z.succ z by done.
  case_eq (interval_compare (y1 , 1%N) (y2 , c2)) ⇒ H_comp;
    rewrite ?H_succ ?Z.lt_succ_r ?Z.le_succ_l //. {
    move ⇒ [H_lt] [H_le] _.
    contradict H_lt.
    by apply Zle_not_lt.
  } {
    move ⇒ [_] [H_lt] H_le.
    contradict H_lt.
    by apply Zle_not_lt.
  } {
    move ⇒ [->] ←.
    rewrite ?Z.lt_succ_r.
    right.
    split; apply Z.le_refl.
  } {
    tauto.
  } {
    case (N.zero_or_succ c2). {
      move ⇒ → _; by left.
    } {
      move ⇒ [c2'] →.
      rewrite !N2Z.inj_succ Z.add_succ_r -Z.succ_le_mono Z.le_succ_l.
      move ⇒ [H_y1_le] [H_le_y1].
      suff → : y1 = y2. {
        move ⇒ [] H_pre; contradict H_pre. {
          apply Z.lt_irrefl.
        } {
          apply Zle_not_lt, Z_le_add_r.
        }
      }
```

```
      apply Z.le_antisymm => //.
      eapply Z.le_trans; last apply H_le_y1.
      apply Z_le_add_r.
    }
  }
Qed.
```

Alternative definition of addZ

```
Lemma addZ_aux_alt_def : ∀ x s acc,
  addZ_aux acc x s = (List.rev acc) ++ addZ x s.
Proof.
  intros y1 s.
  unfold addZ.
  induction s as [| [y2 c2] s' IH] => acc. {
    rewrite /addZ_aux /addZ /= /rev' !rev_append_rev /= app_nil_r //.
  } {
    unfold addZ_aux.
    case (interval_1_compare y1 (y2, c2)); fold addZ_aux;
      rewrite ?rev_append_rev /= ?app_assoc_reverse //.
    rewrite (IH ((y2,c2)::acc)) (IH ((y2,c2)::nil)).
    rewrite /= app_assoc_reverse //.
  }
Qed.

Lemma addZ_alt_def : ∀ x s,
  addZ x s =
  match s with
  | nil => (x, (1%N))::nil
  | (y, c) :: l =>
      match (interval_1_compare x (y,c)) with
        | ICR_before => (x, (1%N))::s
        | ICR_before_touch => (x, N.succ c)::l
        | ICR_after => (y,c)::(addZ x l)
        | ICR_after_touch => insert_interval_begin y (N.succ c) l
        | _ => (y, c)::l
      end
  end.
Proof.
  intros x s.
  rewrite /addZ.
  case s => //.
  move => [y c] s'.
  unfold addZ_aux.
```

```
    case (interval_1_compare x (y, c)); fold addZ_aux;
      rewrite ?rev_append_rev /= ?app_assoc_reverse //.
    rewrite addZ_aux_alt_def //.
Qed.
```

Auxiliary Lemmata about Invariant

```
Lemma interval_list_elements_greater_cons : ∀ z x c s,
  interval_list_elements_greater z ((x, c) :: s) = true ↔
  (z < x).
Proof.
  intros z x c s.
  rewrite /=.
  apply Z.ltb_lt.
Qed.

Lemma interval_list_elements_greater_impl : ∀ x y s,
  (y ≤ x) →
  interval_list_elements_greater x s = true →
  interval_list_elements_greater y s = true.
Proof.
  intros x y s.
  case s ⇒ //.
  move ⇒ [z c] s'.
  rewrite /interval_list_elements_greater.
  move ⇒ H_y_leq /Z.ltb_lt H_x_lt.
  apply Z.ltb_lt.
  eapply Z.le_lt_trans; eauto.
Qed.

Lemma interval_list_invariant_nil : interval_list_invariant nil = true.
Proof.
  by [].
Qed.

Lemma Ok_nil : Ok nil ↔ True.
Proof.
  rewrite /Ok /IsOk /interval_list_invariant /is_encoded_elems_list //.
Qed.

Lemma is_encoded_elems_list_app : ∀ l1 l2,
  is_encoded_elems_list (l1 ++ l2) ↔
  (is_encoded_elems_list l1 ∧ is_encoded_elems_list l2).
Proof.
  intros l1 l2.
```

```
  rewrite /is_encoded_elems_list.
  setoid_rewrite in_app_iff.
  split; firstorder.
Qed.

Lemma is_encoded_elems_list_rev : ∀ l,
  is_encoded_elems_list (List.rev l) ↔
  is_encoded_elems_list l.
Proof.
  intros l.
  rewrite /is_encoded_elems_list.
  split; (
    move ⇒ H x H_in;
    apply H;
    move : H_in;
    rewrite -in_rev ⇒ //
  ).
Qed.

Lemma interval_list_invariant_cons : ∀ y c s',
  interval_list_invariant ((y, c) :: s') = true ↔
  (interval_list_elements_greater (y+Z.of_N c) s' = true ∧
    ((c ≠ 0)%N) ∧ interval_list_invariant s' = true).
Proof.
  rewrite /interval_list_invariant -/interval_list_invariant.
  intros y c s'.
  rewrite !Bool.andb_true_iff negb_true_iff.
  split. {
    move ⇒ [] [H_inf] /N.eqb_neq H_c H_s'. tauto.
  } {
    move ⇒ [H_inf] [/N.eqb_neq H_c] H_s'. tauto.
  }
Qed.

Lemma interval_list_invariant_sing : ∀ x c,
  interval_list_invariant ((x, c)::nil) = true ↔ (c ≠ 0)%N.
Proof.
  intros x c.
  rewrite interval_list_invariant_cons.
  split; tauto.
Qed.

Lemma Ok_cons : ∀ y c s', Ok ((y, c) :: s') ↔
  (interval_list_elements_greater (y+Z.of_N c) s' = true ∧ ((c ≠ 0)%N) ∧
   is_encoded_elems_list (elementsZ_single y c) ∧ Ok s').
```

```
Proof.
  intros y c s'.
  rewrite /Ok /IsOk interval_list_invariant_cons elementsZ_cons is_encoded_elems_list_app
     is_encoded_elems_list_rev.
  tauto.
Qed.

Lemma Nin_elements_greater : ∀ s y,
    interval_list_elements_greater y s = true →
    interval_list_invariant s = true →
    ∀ x, x ≤ y → ~(InZ x s).
Proof.
  induction s as [|[z c] s' IH]. {
    intros y _ _ x _.
    by simpl.
  } {
    move ⇒ y /interval_list_elements_greater_cons H_y_lt
      /interval_list_invariant_cons [H_gr] [H_c] H_s'
      x H_x_le.
    rewrite InZ_cons In_elementsZ_single.
    have H_x_lt : x < z by eapply Z.le_lt_trans; eauto.

    move ⇒ []. {
      move ⇒ [H_z_leq] _; contradict H_z_leq.
      by apply Z.nle_gt.
    } {
      eapply IH; eauto.
      by apply Z_lt_le_add_r.
    }
  }
Qed.

Lemma Nin_elements_greater_equal :
    ∀ x s,
      interval_list_elements_greater x s = true →
      interval_list_invariant s = true →
      ¬ (InZ x s).
Proof.
  move ⇒ x s H_inv H_gr.
  apply (Nin_elements_greater s x) ⇒ //.
  apply Z.le_refl.
Qed.

Lemma interval_list_elements_greater_alt_def : ∀ s y,
    interval_list_invariant s = true →
```

```
    (interval_list_elements_greater y s = true ↔
      (∀ x, x ≤ y → ~(InZ x s))).
Proof.
  intros s y H_inv.
  split. {
    move ⇒ H_gr.
    apply Nin_elements_greater ⇒ //.
  } {
    move : H_inv.
    case s as [|[x2 c] s'] ⇒ //.
    rewrite interval_list_invariant_cons interval_list_elements_greater_cons.
    move ⇒ [_] [H_c_neq] _ H.
    apply Z.nle_gt ⇒ H_ge.
    apply (H x2) ⇒ //.
    rewrite InZ_cons; left.
    apply In_elementsZ_single_hd ⇒ //.
  }
Qed.

Lemma interval_list_elements_greater_alt2_def : ∀ s y,
    interval_list_invariant s = true →

    (interval_list_elements_greater y s = true ↔
      (∀ x, InZ x s → y < x)).
Proof.
  intros s y H.
  rewrite interval_list_elements_greater_alt_def //.
  firstorder.
  apply Z.nle_gt.
  move ⇒ H_lt.
  eapply H0; eauto.
Qed.

Lemma interval_list_elements_greater_intro : ∀ s y,
    interval_list_invariant s = true →
    (∀ x, InZ x s → y < x) →
    interval_list_elements_greater y s = true.
Proof.
  intros s y H1 H2.
  rewrite interval_list_elements_greater_alt2_def //.
Qed.

Lemma interval_list_elements_greater_app_elim_1 : ∀ s1 s2 y,
  interval_list_elements_greater y (s1 ++ s2) = true →
```

```
  interval_list_elements_greater y s1 = true.
Proof.
  intros s1 s2 y.
  case s1 => //.
Qed.

Lemma interval_list_invariant_app_intro : ∀ s1 s2,
    interval_list_invariant s1 = true →
    interval_list_invariant s2 = true →
    (∀ (x1 x2 : Z), InZ x1 s1 → InZ x2 s2 → Z.succ x1 < x2) →
    interval_list_invariant (s1 ++ s2) = true.
Proof.
  induction s1 as [|[y1 c1] s1' IH]. {
    move => s2 _ //.
  } {
    move => s2.
    rewrite -app_comm_cons !interval_list_invariant_cons.
    move => [H_gr] [H_c1_neq] H_inv_s1' H_inv_s2 H_inz_s2.
    split; last split. {
      move : H_gr H_inz_s2.
      case s1' as [|[y1' c1'] s1'']; last done.
      move => _ H_inz_s2.
      rewrite app_nil_l.
      apply interval_list_elements_greater_intro => //.
      move => x H_x_in_s2.
      suff H_inz : InZ (Z.pred (y1 + Z.of_N c1)) ((y1, c1) :: nil). {
        move : (H_inz_s2 _ _ H_inz H_x_in_s2).
        by rewrite Z.succ_pred.
      }
      rewrite InZ_cons In_elementsZ_single -Z.lt_le_pred; left.
      split. {
        by apply Z_lt_add_r.
      } {
        apply Z.lt_pred_l.
      }
    } {
      assumption.
    } {
      apply IH => //.
      intros x1 x2 H_in_x1 H_in_x2.
      apply H_inz_s2 => //.
      rewrite InZ_cons; by right.
    }
```

```
  }
Qed.

Lemma interval_list_invariant_app_elim : ∀ s1 s2,
    interval_list_invariant (s1 ++ s2) = true →
    interval_list_invariant s1 = true ∧
    interval_list_invariant s2 = true ∧
    (∀ (x1 x2 : Z), InZ x1 s1 → InZ x2 s2 → Z.succ x1 < x2).
Proof.
  move ⇒ s1 s2.
  induction s1 as [|[y1 c1] s1' IH]; first done.
  rewrite -app_comm_cons !interval_list_invariant_cons.
  move ⇒ [H_gr] [H_c1_neq_0] /IH [H_inv_s1'] [H_inv_s2] H_in_s1'_s2.
  repeat split; try assumption. {
    move : H_gr.
    case s1'; first done.
    move ⇒ [y2 c2] s1''.
    rewrite interval_list_elements_greater_cons //.
  } {
    move ⇒ x1 x2.
    rewrite InZ_cons In_elementsZ_single.
    move ⇒ []; last by apply H_in_s1'_s2.
    move ⇒ [] H_y1_le H_x1_lt H_x2_in.
    move : H_gr.
    rewrite interval_list_elements_greater_alt2_def; last first. {
        by apply interval_list_invariant_app_intro.
    }
    move ⇒ H_in_s12'.
    have : (y1 + Z.of_N c1 < x2). {
      apply H_in_s12'.
      rewrite InZ_app.
      by right.
    }
    move ⇒ H_lt_x2.
    apply Z.le_lt_trans with (m := y1 + Z.of_N c1) ⇒ //.
    by apply Zlt_le_succ.
  }
Qed.

Lemma interval_list_invariant_app_iff : ∀ s1 s2,
    interval_list_invariant (s1 ++ s2) = true ↔
    (interval_list_invariant s1 = true ∧
    interval_list_invariant s2 = true ∧
    (∀ (x1 x2 : Z), InZ x1 s1 → InZ x2 s2 → Z.succ x1 < x2)).
```

```
Proof.
  intros s1 s2.
  split. {
    by apply interval_list_invariant_app_elim.
  } {
    move => [] H_inv_s1 [].
    by apply interval_list_invariant_app_intro.
  }
Qed.

Lemma interval_list_invariant_snoc_intro : ∀ s1 y2 c2,
    interval_list_invariant s1 = true →
    (c2 ≠ 0)%N →
    (∀ x, InZ x s1 → Z.succ x < y2) →
    interval_list_invariant (s1 ++ ((y2 , c2)::nil)) = true.
Proof.
  intros s1 y2 c2 H_inv_s1 H_c2_neq H_in_s1.
  apply interval_list_invariant_app_intro => //. {
    rewrite interval_list_invariant_cons; done.
  } {
    intros x1 x2 H_in_x1.
    rewrite InZ_cons.
    move => [] //.
    rewrite In_elementsZ_single.
    move => [H_y2_le] _.
    eapply Z.lt_le_trans; eauto.
  }
Qed.
```

Properties of In and InZ

```
Lemma encode_decode_eq : ∀ x s, Ok s → InZ x s →
  (Enc.encode (Enc.decode x) = x).
Proof.
  intros x s.
  rewrite /Ok /IsOk /InZ.
  move => [_] H_enc H_in_x.
  move : (H_enc _ H_in_x) => [x'] ←.
  rewrite Enc.decode_encode_ok //.
Qed.

Lemma In_alt_def : ∀ x s, Ok s →
  (In x s ↔ List.In x (elements s)).
Proof.
```

```
  intros x s H_ok.
  rewrite /In InA_alt /elements rev_map_alt_def.
  split. {
    move => [y] [H_y_eq].
    rewrite -!in_rev !in_map_iff.
    move => [x'] [H_y_eq'] H_x'_in.
    suff H_x'_eq : (Enc.encode x = x'). {
      ∃ x'.
      split => //.
      rewrite -H_x'_eq Enc.decode_encode_ok //.
    }
    have H_enc_list : is_encoded_elems_list (elementsZ s). {
      move : H_ok.
      rewrite /Ok /IsOk => [] [] //.
    }
    move : (H_enc_list _ H_x'_in) => [x''] H_x'_eq.
    move : H_y_eq'.
    rewrite -!H_x'_eq Enc.decode_encode_ok => H_y_eq'.
    subst.
    suff → : Z.eq (Enc.encode x) (Enc.encode y) by done.
    by rewrite Enc.encode_eq.
  } {
    move => H_enc_in.
    ∃ x.
    split => //.
    apply Enc.E.eq_equiv.
  }
Qed.

Lemma In_InZ : ∀ x s, Ok s →
  (In x s ↔ InZ (Enc.encode x) s).
Proof.
  intros x s H_ok.
  rewrite /InZ In_alt_def /elements rev_map_alt_def -in_rev in_map_iff.
  split; last first. {
    ∃ (Enc.encode x).
    by rewrite Enc.decode_encode_ok.
  }
  move => [y] [<-] H_y_in.
  suff : ∃ z, (Enc.encode z = y). {
    move => [z] H_y_eq.
    move : H_y_in.
    by rewrite -!H_y_eq Enc.decode_encode_ok.
```

```
    }
    suff H_enc_list : is_encoded_elems_list (elementsZ s). {
      by apply H_enc_list.
    }
    apply H_ok.
  Qed.

  Lemma InZ_In : ∀ x s, Ok s →
    (InZ x s → In (Enc.decode x) s).
  Proof.
    intros x s H_ok.
    rewrite In_InZ /InZ.
    move : H_ok.
    rewrite /Ok /IsOk /is_encoded_elems_list.
    move ⇒ [_] H_enc.
    move ⇒ H_in.
    move : (H_enc _ H_in) ⇒ [e] H_x.
    subst.
    by rewrite Enc.decode_encode_ok.
  Qed.
```

Membership specification

```
  Lemma memZ_spec :
    ∀ (s : t) (x : Z) (Hs : Ok s), memZ x s = true ↔ InZ x s.
  Proof.
    induction s as [|[y c] s' IH]. {
      intros x _.
      rewrite /InZ elementsZ_nil //.
    } {
      move ⇒ x /Ok_cons [H_inf] [H_c] [H_is_enc] H_s'.
      rewrite /InZ /memZ elementsZ_cons -/memZ.
      rewrite in_app_iff -!in_rev In_elementsZ_single.

      case_eq (x <? y). {
        move ⇒ /Z.ltb_lt H_x_lt.
        split; first done.
        move ⇒ []. {
          move ⇒ H_x_in; contradict H_x_in.
          apply Nin_elements_greater with (y := (y + Z.of_N c)) ⇒ //. {
            apply H_s'.
          } {
            apply Z_lt_le_add_r ⇒ //.
          }
```

```
        } {
          move ⇒ [H_y_le]; contradict H_y_le.
          by apply Z.nle_gt.
        }
      } {
        move ⇒ /Z.ltb_ge H_y_le.
        case_eq (x <? y + Z.of_N c). {
          move ⇒ /Z.ltb_lt H_x_lt.
          split; last done.
          move ⇒ _.
          by right.
        } {
          move ⇒ /Z.ltb_ge H_yc_le.
          rewrite IH.
          split; first tauto.
          move ⇒ [] //.
          move ⇒ [_] H_x_lt; contradict H_x_lt.
          by apply Z.nlt_ge.
        }
      }
    }
Qed.

Lemma mem_spec :
 ∀ (s : t) (x : elt) (Hs : Ok s), mem x s = true ↔ In x s.
Proof.
  intros s x Hs.
  rewrite /mem memZ_spec In_InZ //.
Qed.

Lemma merge_interval_size_neq_0 : ∀ x1 c1 x2 c2,
    (c1 ≠ 0%N) →
    (merge_interval_size x1 c1 x2 c2 ≠ 0)%N.
Proof.
  intros x1 c1 x2 c2.
  rewrite /merge_interval_size !N.neq_0_lt_0 N.max_lt_iff.
  by left.
Qed.
```

insert if length not 0

```
Lemma interval_list_invariant_insert_intervalZ_guarded : ∀ x c s,
  interval_list_invariant s = true →
  interval_list_elements_greater (x + Z.of_N c) s = true →
```

```
    interval_list_invariant (insert_intervalZ_guarded x c s) = true.
  Proof.
    intros x c s.
    rewrite /insert_intervalZ_guarded.
    case_eq (c =? 0)%N => //.
    move => /N.eqb_neq.
    rewrite interval_list_invariant_cons.
    tauto.
  Qed.

  Lemma interval_list_elements_greater_insert_intervalZ_guarded : ∀ x c y s,
    interval_list_elements_greater y (insert_intervalZ_guarded x c s) = true ↔
    (if (c =? 0)%N then (interval_list_elements_greater y s = true) else (y < x)).
  Proof.
    intros x c y s.
    rewrite /insert_intervalZ_guarded.
    case (c =? 0)%N => //.
    rewrite /interval_list_elements_greater Z.ltb_lt //.
  Qed.

  Lemma insert_intervalZ_guarded_app : ∀ x c s1 s2,
    (insert_intervalZ_guarded x c s1) ++ s2 =
    insert_intervalZ_guarded x c (s1 ++ s2).
  Proof.
    intros x c s1 s2.
    rewrite /insert_intervalZ_guarded.
    case (N.eqb c 0) => //.
  Qed.

  Lemma insert_intervalZ_guarded_rev_nil_app : ∀ x c s,
    rev (insert_intervalZ_guarded x c nil) ++ s =
    insert_intervalZ_guarded x c s.
  Proof.
    intros x c s.
    rewrite /insert_intervalZ_guarded.
    case (N.eqb c 0) => //.
  Qed.

Lemma elementsZ_insert_intervalZ_guarded : ∀ x c s,
    elementsZ (insert_intervalZ_guarded x c s) = elementsZ ((x, c) :: s).
  Proof.
    intros x c s.
    rewrite /insert_intervalZ_guarded.
    case_eq (c =? 0)%N => //.
    move => /N.eqb_eq →.
```

```
  rewrite elementsZ_cons elementsZ_single_base /= app_nil_r //.
Qed.

Lemma InZ_insert_intervalZ_guarded : ∀ y x c s,
  InZ y (insert_intervalZ_guarded x c s) = InZ y ((x, c) :: s).
Proof.
  intros y x c s.
  rewrite /InZ elementsZ_insert_intervalZ_guarded //.
Qed.
```

Merging intervals

```
Lemma merge_interval_size_add : ∀ x c1 c2,
   (merge_interval_size x c1 (x + Z.of_N c1) c2 = (c1 + c2))%N.
Proof.
  intros x c1 c2.
  rewrite /merge_interval_size.
  replace (x + Z.of_N c1 + Z.of_N c2 - x) with
          (Z.of_N c1 + Z.of_N c2) by omega.
  rewrite -N2Z.inj_add N2Z.id.
  apply N.max_r, N.le_add_r.
Qed.

Lemma merge_interval_size_eq_max : ∀ y1 c1 y2 c2,
   y1 ≤ y2 + Z.of_N c2 →
   y1 + Z.of_N (merge_interval_size y1 c1 y2 c2) =
   Z.max (y1 + Z.of_N c1) (y2 + Z.of_N c2).
Proof.
  intros y1 c1 y2 c2 H_y1_le.
  rewrite /merge_interval_size N2Z.inj_max Z2N.id; last first. {
    by apply Zle_minus_le_0.
  }
  rewrite -Z.add_max_distr_l.
  replace (y1 + (y2 + Z.of_N c2 - y1)) with (y2 + Z.of_N c2) by omega.
  done.
Qed.

Lemma merge_interval_size_invariant : ∀ y1 c1 y2 c2 z s,
  interval_list_invariant s = true →
  y1 + Z.of_N c1 ≤ y2 + Z.of_N c2 →
  y2 + Z.of_N c2 ≤ z →
  interval_list_elements_greater z s = true →
  (c1 ≠ 0)%N →
  interval_list_invariant ((y1, merge_interval_size y1 c1 y2 c2) :: s) =
 true.
```

```
Proof.
  intros y1 c1 y2 c2 z s H_inv H_le H_le_z H_gr H_c1_neq_0.
  rewrite interval_list_invariant_cons.
  split; last split. {
    rewrite merge_interval_size_eq_max; last first. {
      eapply Z.le_trans; last apply H_le.
      apply Z_le_add_r.
    } {
      rewrite Z.max_r => //.
      eapply interval_list_elements_greater_impl; first apply H_le_z.
      done.
    }
  } {
    apply merge_interval_size_neq_0.
    assumption.
  } {
    assumption.
  }
Qed.

Lemma In_merge_interval : ∀ x1 c1 x2 c2 y,
  x1 ≤ x2 →
  x2 ≤ x1 + Z.of_N c1 → (
  List.In y (elementsZ_single x1 (merge_interval_size x1 c1 x2 c2)) ↔
  List.In y (elementsZ_single x1 c1) ∨ List.In y (elementsZ_single x2 c2)).
Proof.
  intros x1 c1 x2 c2 y H_x1_le H_x2_le.
  rewrite !In_elementsZ_single merge_interval_size_eq_max;
    last first. {
    eapply Z.le_trans; eauto.
    by apply Z_le_add_r.
  }
  rewrite Z.max_lt_iff.
  split. {
    move ⇒ [H_x_le] [] H_y_lt. {
      by left.
    } {
      case_eq (Z.leb x2 y). {
        move ⇒ /Z.leb_le H_y'_le.
        by right.
      } {
        move ⇒ /Z.leb_gt H_y_lt_x2.
        left.
```

```
        split => //.
        eapply Z.lt_le_trans; eauto.
      }
    }
  } {
    move => []. {
      tauto.
    } {
      move => [H_x2_le'] H_y_lt.
      split. {
        eapply Z.le_trans; eauto.
      } {
        by right.
      }
    }
  }
Qed.

Lemma insert_interval_begin_spec : ∀ y s x c,
   interval_list_invariant s = true →
   interval_list_elements_greater x s = true →
   (c ≠ 0)%N →
    (
   interval_list_invariant (insert_interval_begin x c s) = true ∧
   (InZ y (insert_interval_begin x c s) ↔
   (List.In y (elementsZ_single x c) ∨ InZ y s))).
Proof.
  intros y.
  induction s as [| [y' c'] s' IH]. {
    intros x c _ H_c_neq H_z_lt.
    rewrite /insert_interval_begin InZ_cons interval_list_invariant_cons //.
  } {
    intros x c.
    rewrite interval_list_invariant_cons
     interval_list_elements_greater_cons.
    move => [H_gr] [H_c'_neq_0] H_inv_s' H_x_lt H_c_neq_0.
    unfold insert_interval_begin.
    Z_named_compare_cases H_y'; fold insert_interval_begin. {
      subst.
      rewrite !InZ_cons elementsZ_single_add in_app_iff.
      split; last tauto.
      rewrite interval_list_invariant_cons N2Z.inj_add
        Z.add_assoc N.eq_add_0.
```

```
        tauto.
      } {
        rewrite !InZ_cons !interval_list_invariant_cons
          interval_list_elements_greater_cons.
        repeat split => //.
      } {
        set c'' := merge_interval_size x c y' c'.
        have H_x_lt' : x < y' + Z.of_N c'. {
          eapply Z.lt_le_trans with (m := y') => //.
          by apply Z_le_add_r.
        }

        have H_pre : interval_list_elements_greater x s' = true. {
          eapply interval_list_elements_greater_impl; eauto.
          by apply Z.lt_le_incl.
        }
        have H_pre2 : c'' ≠ 0%N. {
          by apply merge_interval_size_neq_0.
        }
        move : (IH x c'' H_inv_s' H_pre H_pre2) => {IH} {H_pre} {H_pre2} [->] →.
        split; first reflexivity.
        unfold c''; clear c''.
        rewrite In_merge_interval. {
          rewrite InZ_cons.
          tauto.
        } {
          by apply Z.lt_le_incl.
        } {
          by apply Z.lt_le_incl.
        }
      }
    }
  Qed.
```

add specification

```
  Lemma addZ_InZ :
   ∀ (s : t) (x y : Z),
    interval_list_invariant s = true →
    (InZ y (addZ x s) ↔ x = y ∨ InZ y s).
  Proof.
    move => s x y.
    induction s as [|[z c] s' IH]. {
```

```
  move => _.
  rewrite /InZ addZ_alt_def
          elementsZ_cons elementsZ_nil app_nil_l -in_rev
          In_elementsZ_single1 /=.
  firstorder.
} {
  move => /interval_list_invariant_cons [H_greater] [H_c_neq_0] H_inv_c'.
  move : (IH H_inv_c') => {IH} IH.
  rewrite addZ_alt_def.
  have H_succ : ∀ z, z + Z.of_N 1 = Z.succ z by done.
  move : (interval_1_compare_elim x z c).
  case (interval_1_compare x (z, c));
    rewrite ?InZ_cons ?In_elementsZ_single1 ?H_succ ?Z.lt_succ_r //. {
    move => ->.
    rewrite elementsZ_single_succ_front /=.
    tauto.
  } {
    move => [] // H_x_in.
    split; first tauto.
    move => [] // <-.
    left.
    by rewrite In_elementsZ_single.
  } {
    rewrite IH.
    tauto.
  } {
    move => H_x_eq.
    have -> : (InZ y (insert_interval_begin z (N.succ c) s') <->
               List.In y (elementsZ_single z (N.succ c)) \/ InZ y s'). {
      eapply insert_interval_begin_spec. {
        by apply H_inv_c'.
      } {
        eapply interval_list_elements_greater_impl; eauto.
        apply Z_le_add_r.
      } {
        by apply N.neq_succ_0.
      }
    }
    rewrite -H_x_eq elementsZ_single_succ in_app_iff /=.
    tauto.
  }
}
```

```
Qed.
Lemma addZ_invariant : ∀ s x,
  interval_list_invariant s = true →
  interval_list_invariant (addZ x s) = true.
Proof.
  move ⇒ s x.
  induction s as [|[z c] s' IH]. {
    move ⇒ _.
    by simpl.
  } {
    move ⇒ /interval_list_invariant_cons [H_greater] [H_c_neq_0]
              H_inv_c'.
    move : (IH H_inv_c') ⇒ {IH} IH.
    rewrite addZ_alt_def.
    have H_succ : ∀ z, z + Z.of_N 1 = Z.succ z by done.
    move : (interval_1_compare_elim x z c).
    case_eq (interval_1_compare x (z, c)) ⇒ H_comp;
      rewrite ?InZ_cons ?In_elementsZ_single1 ?H_succ ?Z.lt_succ_r //. {
      move ⇒ H_z_gt.
      rewrite interval_list_invariant_cons /= !andb_true_iff !H_succ.
      repeat split ⇒ //. {
        by apply Z.ltb_lt.
      } {
        apply negb_true_iff, N.eqb_neq ⇒ //.
      }
    } {
      move ⇒ ?; subst.
      rewrite /= !andb_true_iff.
      repeat split ⇒ //. {
        move : H_greater.
        rewrite Z.add_succ_l -Z.add_succ_r N2Z.inj_succ //.
      } {
        apply negb_true_iff, N.eqb_neq ⇒ //.
        apply N.neq_succ_0.
      }
    } {
      move ⇒ [] // _.
      rewrite interval_list_invariant_cons /=.
      tauto.
    } {
      rewrite interval_list_invariant_cons.
      move ⇒ H_lt_x.
```

```
      repeat split => //.
      apply interval_list_elements_greater_intro => //.
      move => xx.
      rewrite addZ_InZ => //.
      move => [<- //|].
      apply interval_list_elements_greater_alt2_def => //.
    } {
      move => H_x_eq.
      apply insert_interval_begin_spec => //. {
        eapply interval_list_elements_greater_impl; eauto.
        apply Z_le_add_r.
      } {
        by apply N.neq_succ_0.
      }
    }
  }
Qed.

Global Instance add_ok s x : ∀ `(Ok s), Ok (add x s).
Proof.
  move => H_ok_s.
  move : (H_ok_s).
  rewrite /Ok /IsOk /is_encoded_elems_list /add.
  move => [H_isok_s] H_pre.
  split. {
    apply addZ_invariant => //.
  } {
    intros y.
    move : (addZ_InZ s (Enc.encode x) y H_isok_s).
    rewrite /InZ => ->.
    move => []. {
      move => <-.
      by ∃ x.
    } {
      move => /H_pre //.
    }
  }
Qed.

Lemma add_spec :
 ∀ (s : t) (x y : elt) (Hs : Ok s),
   In y (add x s) ↔ Enc.E.eq y x ∨ In y s.
Proof.
  intros s x y Hs.
```

```
    have Hs' := (add_ok s x Hs).
    rewrite !In_InZ.
    rewrite /add addZ_InZ. {
      rewrite -Enc.encode_eq; firstorder.
    } {
      apply Hs.
    }
  Qed.
```

empty specification

```
  Global Instance empty_ok : Ok empty.
  Proof.
    rewrite /empty Ok_nil //.
  Qed.

  Lemma empty_spec' : ∀ x, (In x empty ↔ False).
    rewrite /Empty /empty /In elements_nil.
    intros a.
    rewrite InA_nil //.
  Qed.

  Lemma empty_spec : Empty empty.
  Proof.
    rewrite /Empty ⇒ a.
    rewrite empty_spec' //.
  Qed.
```

is_empty specification

```
  Lemma is_empty_spec : ∀ (s : t) (Hs : Ok s), is_empty s = true ↔ Empty s.
  Proof.
    intros [ | [x c] s]. {
      split ⇒ // _.
      apply empty_spec.
    } {
      rewrite /= /Empty Ok_cons.
      move ⇒ [_] [H_c_neq] [H_enc] _.
      split ⇒ //.
      move ⇒ H.
      contradiction (H (Enc.decode x)) ⇒ {H}.
      rewrite /In InA_alt elements_cons.
      ∃ (Enc.decode x).
      split; first by apply Enc.E.eq_equiv.
```

```
    rewrite in_app_iff; left.
    rewrite /elements_single in_map_iff.
    ∃ x.
    split ⇒ //.
    apply In_elementsZ_single_hd ⇒ //.
  }
Qed.
```

singleton specification

```
Global Instance singleton_ok x : Ok (singleton x).
Proof.
  rewrite singleton_alt_def.
  apply add_ok.
  apply empty_ok.
Qed.

Lemma singleton_spec : ∀ x y : elt, In y (singleton x) ↔ Enc.E.eq y x.
Proof.
  intros x y.
  rewrite singleton_alt_def.
  rewrite (add_spec empty x y) /empty /In elements_nil InA_nil.
  split. {
    move ⇒ [] //.
  } {
    by left.
  }
Qed.
```

add_list specification

```
Lemma add_list_ok : ∀ l s, Ok s → Ok (add_list l s).
Proof.
  induction l as [| x l' IH]. {
    done.
  } {
    move ⇒ s H_s_ok /=.
    apply IH.
    by apply add_ok.
  }
Qed.

Lemma add_list_spec : ∀ x l s, Ok s →
   (In x (add_list l s) ↔ (SetoidList.InA Enc.E.eq x l) ∨ In x s).
```

```
Proof.
  move => x.
  induction l as [| y l' IH]. {
    intros s H.
    rewrite /= InA_nil.
    tauto.
  } {
    move => s H_ok /=.
    rewrite IH add_spec InA_cons.
    tauto.
  }
Qed.
```

union specification

```
Lemma union_aux_flatten_alt_def : ∀ (s1 s2 : t) acc,
  union_aux s1 s2 acc =
  match (s1, s2) with
  | (nil, _) => List.rev_append acc s2
  | (_, nil) => List.rev_append acc s1
  | ((y1, c1) :: l1, (y2, c2) :: l2) =>
      match (interval_compare (y1, c1) (y2,c2)) with
        | ICR_before => union_aux l1 s2 ((y1, c1)::acc)
        | ICR_before_touch =>
            union_aux l1 (
              insert_interval_begin y1 ((c1+c2)%N) l2) acc
        | ICR_after => union_aux s1 l2 ((y2, c2)::acc)
        | ICR_after_touch => union_aux l1 (
            insert_interval_begin y2 ((c1+c2)%N) l2) acc
        | ICR_overlap_before =>
            union_aux l1 (
              insert_interval_begin y1
                (merge_interval_size y1 c1 y2 c2) l2) acc
        | ICR_overlap_after =>
            union_aux l1 (
              insert_interval_begin y2
                (merge_interval_size y2 c2 y1 c1) l2) acc
        | ICR_equal => union_aux l1 s2 acc
        | ICR_subsume_1 => union_aux l1 s2 acc
        | ICR_subsume_2 => union_aux s1 l2 acc
      end
  end.
Proof.
```

```
  intros s1 s2 acc.
  case s1, s2 => //.
Qed.

Lemma union_aux_alt_def : ∀ (s1 s2 : t) acc,
  union_aux s1 s2 acc =
  List.rev_append acc (union s1 s2).
Proof.
  rewrite /union.
  intros s1 s2 acc.
  move : acc s2.
  induction s1 as [|| [y1 c1] l1 IH1]. {
    intros acc s2.
    rewrite !union_aux_flatten_alt_def.
    rewrite !rev_append_rev //.
  }
  intros acc s2; move : acc.
  induction s2 as [|| [y2 c2] l2 IH2]; first by simpl.
  move => acc.
  rewrite !(union_aux_flatten_alt_def ((y1, c1)::l1) ((y2, c2)::l2)).
  case (interval_compare (y1, c1) (y2, c2));
    rewrite ?(IH1 ((y1, c1) :: acc)) ?(IH1 ((y1, c1) :: nil))
            ?(IH2 ((y2, c2) :: acc)) ?(IH2 ((y2, c2) :: nil))
            ?(IH1 acc) //.
Qed.

Lemma union_alt_def : ∀ (s1 s2 : t),
  union s1 s2 =
  match (s1, s2) with
  | (nil, _) => s2
  | (_, nil) => s1
  | ((y1, c1) :: l1, (y2, c2) :: l2) =>
      match (interval_compare (y1, c1) (y2,c2)) with
        | ICR_before => (y1, c1) :: (union l1 s2)
        | ICR_before_touch =>
            union l1 (insert_interval_begin y1 ((c1+c2)%N) l2)
        | ICR_after => (y2, c2) :: union s1 l2
        | ICR_after_touch => union l1
            (insert_interval_begin y2 ((c1+c2)%N) l2)
        | ICR_overlap_before =>
            union l1 (insert_interval_begin y1 (merge_interval_size y1 c1 y2 c2) l2)
        | ICR_overlap_after =>
            union l1 (insert_interval_begin y2 (merge_interval_size y2 c2 y1 c1) l2)
        | ICR_equal => union l1 s2
```

```
          | ICR_subsume_1 ⇒ union l1 s2
          | ICR_subsume_2 ⇒ union s1 l2
        end
    end.
Proof.
  intros s1 s2.
  rewrite /union union_aux_flatten_alt_def.
  case s1 as [|| [y1 c1] l1] ⇒ //.
  case s2 as [|| [y2 c2] l2] ⇒ //.
  case (interval_compare (y1, c1) (y2, c2));
    rewrite union_aux_alt_def //.
Qed.

Lemma union_InZ :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  ∀ y, (InZ y (union s1 s2) ↔ InZ y s1 ∨ InZ y s2).
Proof.
  intro s1.
  induction s1 as [|| [y1 c1] l1 IH1]. {
    intros s2 _ _ y.
    rewrite union_alt_def /InZ /=.
    tauto.
  } {
    induction s2 as [|| [y2 c2] l2 IH2]. {
      intros _ _ y.
      rewrite union_alt_def /InZ /=.
      tauto.
    } {
      move ⇒ H_inv_s1 H_inv_s2.
      move : (H_inv_s1) (H_inv_s2).
      rewrite !interval_list_invariant_cons.
      move ⇒ [H_gr_l1] [H_c1_neq_0] H_inv_l1.
      move ⇒ [H_gr_l2] [H_c2_neq_0] H_inv_l2.
      move : (IH2 H_inv_s1 H_inv_l2) ⇒ {IH2} IH2.
      have : ∀ s2 : t,
        interval_list_invariant s2 = true →
        ∀ y : Z, InZ y (union l1 s2) ↔ InZ y l1 ∨ InZ y s2. {
        intros. by apply IH1.
      }
      move ⇒ {IH1} IH1 y.
      rewrite union_alt_def.
```

```
move : (interval_compare_elim y1 c1 y2 c2).
case (interval_compare (y1, c1) (y2, c2)). {
  rewrite !InZ_cons IH1 // InZ_cons.
  tauto.
} {
  move ⇒ H_y2_eq.
  replace (c1 + c2)%N with (merge_interval_size y1 c1 y2 c2);
    last first. {
    rewrite -H_y2_eq merge_interval_size_add //.
  }
  set c'' := merge_interval_size y1 c1 y2 c2.
  have [H_inv_insert H_InZ_insert] :
       interval_list_invariant (insert_interval_begin y1 c'' l2) = true ∧
        (InZ y (insert_interval_begin y1 c'' l2) ↔
        List.In y (elementsZ_single y1 c'') ∨ InZ y l2). {
    apply insert_interval_begin_spec ⇒ //. {
      eapply interval_list_elements_greater_impl; eauto.
      rewrite -H_y2_eq -Z.add_assoc -N2Z.inj_add.
      apply Z_le_add_r.
    } {
      by apply merge_interval_size_neq_0.
    }
  }

  rewrite IH1 ⇒ //.
  rewrite H_InZ_insert !InZ_cons /c''.
  rewrite -H_y2_eq In_merge_interval. {
    tauto.
  } {
    apply Z_le_add_r.
  } {
    by apply Z.le_refl.
  }
} {
  move ⇒ [H_y1_lt] [H_y2_lt] H_y1_c1_lt.
  set c'' := merge_interval_size y1 c1 y2 c2.
  have [H_inv_insert H_InZ_insert] :
       interval_list_invariant (insert_interval_begin y1 c'' l2) = true ∧
        (InZ y (insert_interval_begin y1 c'' l2) ↔
        List.In y (elementsZ_single y1 c'') ∨ InZ y l2). {
    apply insert_interval_begin_spec ⇒ //. {
      eapply interval_list_elements_greater_impl; eauto.
```

```
        apply Z_lt_le_add_r => //.
      } {
        by apply merge_interval_size_neq_0.
      }
    }
    rewrite IH1 => //.
    rewrite H_InZ_insert !InZ_cons /c''.
    rewrite In_merge_interval. {
      tauto.
    } {
      by apply Z.lt_le_incl.
    } {
      by apply Z.lt_le_incl.
    }
  } {
    move => [H_y2_lt] [H_y1_lt] H_y2_c_lt.
    set c'' := merge_interval_size y2 c2 y1 c1.
    have [H_inv_insert H_InZ_insert] :
         interval_list_invariant (insert_interval_begin y2 c'' l2) = true /\
          (InZ y (insert_interval_begin y2 c'' l2) <->
          List.In y (elementsZ_single y2 c'') \/ InZ y l2). {
      apply insert_interval_begin_spec => //. {
        eapply interval_list_elements_greater_impl; eauto.
        apply Z_le_add_r.
      } {
        by apply merge_interval_size_neq_0.
      }
    }

    rewrite IH1 => //.
    rewrite H_InZ_insert !InZ_cons /c''.
    rewrite In_merge_interval. {
      tauto.
    } {
      by apply Z.lt_le_incl.
    } {
      by apply Z.lt_le_incl.
    }
  } {
    move => [? ?]; subst.
    rewrite IH1 => //.
    rewrite !InZ_cons.
```

```
    tauto.
  } {
    move ⇒ [H_y2_le] [H_y1_c1_le] _.
    rewrite IH1 ⇒ //.
    rewrite !InZ_cons.
    suff : (List.In y (elementsZ_single y1 c1) →
              List.In y (elementsZ_single y2 c2)). {
      tauto.
    }
    rewrite !In_elementsZ_single.
    move ⇒ [H_y1_le H_y_lt].
    split; omega.
  } {
    move ⇒ [H_y1_le] [H_y2_c2_le] _.
    rewrite IH2.
    rewrite !InZ_cons.
    suff : (List.In y (elementsZ_single y2 c2) →
              List.In y (elementsZ_single y1 c1)). {
      tauto.
    }
    rewrite !In_elementsZ_single.
    move ⇒ [H_y2_le H_y_lt].
    split; omega.
  } {
    rewrite !InZ_cons IH2 !InZ_cons.
    tauto.
  } {
    move ⇒ H_y1_eq.
    replace (c1 + c2)%N with (merge_interval_size y2 c2 y1 c1);
      last first. {
      rewrite -H_y1_eq merge_interval_size_add N.add_comm //.
    }
    set c'' := merge_interval_size y2 c2 y1 c1.
    have [H_inv_insert H_InZ_insert] :
         interval_list_invariant (insert_interval_begin y2 c'' l2) = true ∧
          (InZ y (insert_interval_begin y2 c'' l2) ↔
          List.In y (elementsZ_single y2 c'') ∨ InZ y l2). {
      apply insert_interval_begin_spec ⇒ //. {
        eapply interval_list_elements_greater_impl; eauto.
        apply Z_le_add_r.
      } {
        by apply merge_interval_size_neq_0.
```

```
        }
      }

      rewrite IH1 => //.
      rewrite H_InZ_insert !InZ_cons /c''.
      rewrite -H_y1_eq In_merge_interval. {
        tauto.
      } {
        apply Z_le_add_r.
      } {
        by apply Z.le_refl.
      }
    }
  }
}
Qed.

Lemma union_invariant :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  interval_list_invariant (union s1 s2) = true.
Proof.
  intro s1.
  induction s1 as [|| [y1 c1] l1 IH1]. {
    intros s2 _ H_inv_s2.
    rewrite union_alt_def /InZ //.
  } {
    induction s2 as [|| [y2 c2] l2 IH2]. {
      intros H_inv_s1 _.
      rewrite union_alt_def /InZ //.
    } {
      move => H_inv_s1 H_inv_s2.
      move : (H_inv_s1) (H_inv_s2).
      rewrite !interval_list_invariant_cons.
      move => [H_gr_l1] [H_c1_neq_0] H_inv_l1.
      move => [H_gr_l2] [H_c2_neq_0] H_inv_l2.
      move : (IH2 H_inv_s1 H_inv_l2) => {IH2} IH2.
      have : ∀ s2 : t,
        interval_list_invariant s2 = true →
        interval_list_invariant (union l1 s2) = true. {
        intros. by apply IH1.
      }
```

```
move => {IH1} IH1.
rewrite union_alt_def.
move : (interval_compare_elim y1 c1 y2 c2).
case (interval_compare (y1, c1) (y2, c2)). {
  move => H_lt_y2.
  have H_inv' : interval_list_invariant (union l1 ((y2, c2) :: l2)) = true. {
    by apply IH1.
  }

  rewrite interval_list_invariant_cons.
  repeat split => //.
  apply interval_list_elements_greater_intro => // x.
  rewrite union_InZ => //.
  move => []. {
    apply interval_list_elements_greater_alt2_def => //.
  } {
    apply interval_list_elements_greater_alt2_def => //.
    rewrite interval_list_elements_greater_cons //.
  }
} {
  move => H_y2_eq.
  apply IH1.
  apply insert_interval_begin_spec => //. {
    eapply interval_list_elements_greater_impl; last apply H_gr_l2.
    rewrite -H_y2_eq -Z.add_assoc -N2Z.inj_add.
    apply Z_le_add_r.
  } {
    rewrite N.eq_add_0.
    tauto.
  }
} {
  move => [H_y1_lt] _.
  apply IH1.
  apply insert_interval_begin_spec => //. {
    eapply interval_list_elements_greater_impl; last apply H_gr_l2.
    apply Z_lt_le_add_r => //.
  } {
    apply merge_interval_size_neq_0 => //.
  }
} {
  move => [H_y2_lt] _.
  apply IH1.
```

```
        apply insert_interval_begin_spec => //. {
          eapply interval_list_elements_greater_impl; last apply H_gr_l2.
          apply Z_le_add_r => //.
        } {
          apply merge_interval_size_neq_0 => //.
        }
      } {
        move => [? ?]; subst.
        apply IH1 => //.
      } {
        move => _.
        apply IH1 => //.
      } {
        move => _.
        apply IH2 => //.
      } {
        move => H_lt_y1.
        rewrite interval_list_invariant_cons => //.
        repeat split => //.
        apply interval_list_elements_greater_intro => // x.
        rewrite union_InZ => //.
        move => []. {
          apply interval_list_elements_greater_alt2_def => //.
          rewrite interval_list_elements_greater_cons //.
        } {
          apply interval_list_elements_greater_alt2_def => //.
        }
      } {
        move => H_y1_eq.
        apply IH1 => //.
        apply insert_interval_begin_spec => //. {
          eapply interval_list_elements_greater_impl; last apply H_gr_l2.
          apply Z_le_add_r.
        } {
          rewrite N.eq_add_0.
          tauto.
        }
      }
    }
  }
Qed.

Global Instance union_ok s1 s2 : ∀ `(Ok s1, Ok s2), Ok (union s1 s2).
```

```
Proof.
  move => H_ok_s1 H_ok_s2.
  move : (H_ok_s1) (H_ok_s2).
  rewrite /Ok /IsOk /is_encoded_elems_list /add.
  move => [H_inv_s1] H_pre1.
  move => [H_inv_s2] H_pre2.
  split. {
    apply union_invariant => //.
  } {
    intros y.
    move : (union_InZ s1 s2 H_inv_s1 H_inv_s2).
    rewrite /InZ => ->.
    move => []. {
      apply H_pre1.
    } {
      apply H_pre2.
    }
  }
Qed.

Lemma union_spec :
 ∀ (s s' : t) (x : elt) (Hs : Ok s) (Hs' : Ok s'),
 In x (union s s') ↔ In x s ∨ In x s'.
Proof.
  intros s s' x H_ok H_ok'.
  rewrite !In_InZ.
  rewrite union_InZ => //. {
    apply H_ok.
  } {
    apply H_ok'.
  }
Qed.
```

inter specification

```
Lemma inter_aux_alt_def :
  ∀ (y2 : Z) (c2 : N) (s : t) acc,
    inter_aux y2 c2 acc s = match inter_aux y2 c2 nil s with
                              (acc', s') => (acc' ++ acc, s')
                            end.
Proof.
  intros y2 c2.
  induction s as [| [y1 c1] s' IH] => acc. {
```

```
    rewrite /inter_aux app_nil_l //.
  } {
    simpl.
    case_eq (inter_aux y2 c2 nil s') => acc'' s'' H_eq.
    case (interval_compare (y1, c1) (y2, c2));
      rewrite ?(IH acc)
              ?(IH ((y2, Z.to_N (y1 + Z.of_N c1 - y2)) :: acc))
              ?(IH ((y2, Z.to_N (y1 + Z.of_N c1 - y2)) :: nil))
              ?(IH ((y1, Z.to_N (y2 + Z.of_N c2 - y1)) :: acc))
              ?(IH ((y1, Z.to_N (y2 + Z.of_N c2 - y1)) :: nil))
              ?(IH ((y1, c1) :: acc))
              ?(IH ((y1, c1) :: nil))

              ?H_eq -?app_assoc -?app_comm_cons //.
  }
Qed.

Lemma inter_aux_props :
  ∀ (y2 : Z) (c2 : N) (s : t) acc,
    interval_list_invariant (rev acc) = true →
    interval_list_invariant s = true →
    (∀ x1 x2, InZ x1 acc → InZ x2 s →
                        List.In x2 (elementsZ_single y2 c2) →
                        Z.succ x1 < x2) →
    (c2 ≠ 0%N) →
    match (inter_aux y2 c2 acc s) with (acc', s') ⇒
      (∀ y, (InZ y acc' ↔
                 (InZ y acc ∨ (InZ y s ∧ (List.In y (elementsZ_single y2 c2)))))) ∧
      (∀ y, InZ y s' → InZ y s) ∧
      (∀ y, InZ y s → y2 + Z.of_N c2 < y → InZ y s') ∧
      interval_list_invariant (rev acc') = true ∧
      interval_list_invariant s' = true
    end.
Proof.
  intros y2 c2.
  induction s as [|[y1 c1] s1' IH] ⇒ acc. {
    rewrite /inter_aux.
    move ⇒ H_inv_acc _ _ _.
    split; last split; try done.
    move ⇒ y. rewrite InZ_nil.
    tauto.
  } {
    rewrite interval_list_invariant_cons.
```

```
move ⇒ H_inv_acc [H_gr_s1'] [H_c1_neq_0] H_inv_s1'.
move ⇒ H_in_acc_lt H_c2_neq_0.

rewrite inter_aux_alt_def.
case_eq (inter_aux y2 c2 nil ((y1 , c1) :: s1')).
move ⇒ acc' s' H_inter_aux_eq.

set P1 := ∀ y : Z,
  (InZ y acc' ↔
  ((InZ y ((y1 , c1) :: s1') ∧ List.In y (elementsZ_single y2 c2)))).
set P2 := (∀ y,
             (InZ y s' → InZ y ((y1 , c1) :: s1')) ∧
             (InZ y ((y1 , c1) :: s1') →
                y2 + Z.of_N c2 < y → InZ y s')).
set P3 := interval_list_invariant (rev acc') = true.
set P4 := interval_list_invariant s' = true.

suff : (P1 ∧ P2 ∧ P3 ∧ P4). {
  move ⇒ [H_P1] [H_P2] [H_P3] H_P4.
  split; last split; last split; last split. {
    move ⇒ y.
    move : (H_P1 y).
    rewrite !InZ_app InZ_cons !In_elementsZ_single.
    move ⇒ ←.
    tauto.
  } {
    move ⇒ y H_y_in.
    by apply H_P2.
  } {
    move ⇒ y H_y_in.
    by apply H_P2.
  } {
    rewrite rev_app_distr.
    apply interval_list_invariant_app_intro ⇒ //.
    move ⇒ x1 x2.
    rewrite !InZ_rev.
    move ⇒ H_x1_in /H_P1 [H_x2_in1] H_x2_in2.
    apply H_in_acc_lt ⇒ //.
  } {
    apply H_P4.
  }
}

move : (H_gr_s1').
rewrite interval_list_elements_greater_alt2_def ⇒ // ⇒ H_gr_s1'_alt.
```

```
have : ∀ (acc : list (Z × N)),
  interval_list_invariant (rev acc) = true →
  (∀ y, InZ y acc ↔ (
      y1 ≤ y < y1 + Z.of_N c1 ∧
      y2 ≤ y < y2 + Z.of_N c2)) →
  (y1 + Z.of_N c1 ≤ y2 + Z.of_N c2) →
  (inter_aux y2 c2 acc s1' = (acc', s')) →
  P1 ∧ P2 ∧ P3 ∧ P4. {

  intros acc0 H_inv_acc0 H_in_acc0 H_y2c_lt H_inter_aux_eq0.
  have H_in_acc0_lt : (∀ x1 x2 : Z,
    InZ x1 acc0 →
    InZ x2 s1' →
    List.In x2 (elementsZ_single y2 c2) →
    Z.succ x1 < x2). {

    intros x1 x2 H_x1_in_acc0 H_x2_in_s1' H_x2_in_yc2.
    suff H_yc1_lt_x2 : Z.succ x1 ≤ y1 + Z.of_N c1. {
      apply Z.le_lt_trans with (m := y1 + Z.of_N c1) ⇒ //.
      by apply H_gr_s1'_alt.
    }
    move : (H_x1_in_acc0).
    rewrite H_in_acc0 Z.le_succ_l.
    tauto.
  }

  move : (IH acc0 H_inv_acc0 H_inv_s1' H_in_acc0_lt H_c2_neq_0).
  rewrite H_inter_aux_eq0.
  move ⇒ [H1] [H2] [H3] [H4] H5.
  split; last split ⇒ //. {
    move ⇒ y.
    rewrite (H1 y).
    rewrite InZ_cons !In_elementsZ_single
            H_in_acc0.
    tauto.
  } {
    move ⇒ y.
    split. {
      move ⇒ /H2.
      rewrite InZ_cons.
      by right.
    } {
```

```
        rewrite InZ_cons In_elementsZ_single.
        move => []. {
          move => [_] H_y_lt H_lt_y.
          exfalso.
          suff : (y < y) by apply Z.lt_irrefl.
          apply Z.lt_le_trans with (m := y1 + Z.of_N c1) => //.
          apply Z.le_trans with (m := y2 + Z.of_N c2) => //.
          by apply Z.lt_le_incl.
        } {
          apply H3.
        }
      }
    }
}
move => {IH} IH.
clear H_inv_acc H_in_acc_lt acc.
move : (interval_compare_elim y1 c1 y2 c2) H_inter_aux_eq.
unfold inter_aux.
case_eq (interval_compare (y1, c1) (y2, c2)) => H_comp;
   fold inter_aux. {
  move => H_lt_y2.
  apply IH. {
    done.
  } {
    move => x.
    rewrite InZ_nil.
    split => //.
    omega.
  } {
    apply Z.le_trans with (m := y2). {
      by apply Z.lt_le_incl.
    } {
      apply Z_le_add_r.
    }
  }
} {
  move => H_eq_y2.
  apply IH. {
    done.
  } {
    move => x.
    rewrite InZ_nil.
```

```
      split => //.
      omega.
    } {
      rewrite H_eq_y2.
      apply Z_le_add_r.
    }
  } {
    move => [H_y1_lt_y2] [H_y2_lt_yc1] H_yc1_lt_yc2.
    apply IH. {
      rewrite interval_list_invariant_sing.
      by apply Z_to_N_minus_neq_0.
    } {
      move => x.
      rewrite InZ_cons InZ_nil In_elementsZ_single Z2N.id; last omega.
      replace (y1 + (y2 - y1)) with y2 by omega.
      split; omega.
    } {
      by apply Z.lt_le_incl.
    }
  } {
    rewrite /P1 /P2 /P3 /P4.
    move => [H_y2_lt] [H_y1_lt] H_yc1_lt.
    move => [] H_acc' H_s'.
    clear IH P1 P2 P3 P4 H_comp.
    subst s' acc'.
    split; last split; last split. {
      move => y.
      have H_yc2_intro : y1 + Z.of_N (Z.to_N (y2 + Z.of_N c2 - y1)) =
                         y2 + Z.of_N c2. {
        rewrite Z2N.id; omega.
      }

      rewrite !InZ_cons !In_elementsZ_single InZ_nil H_yc2_intro.
      split. {
        move => [] //.
        move => [H_y1_le] H_y_lt.
        split; last by omega.
        left. omega.
      } {
        move => [H_in_s] [H_y2_le] H_y_lt.
        left.
        split; last assumption.
```

```
      move : H_in_s => []. {
        tauto.
      } {
        move => /H_gr_s1'_alt H_lt_y.
        apply Z.le_trans with (m := y1 + Z.of_N c1). {
          by apply Z_le_add_r.
        } {
          by apply Z.lt_le_incl.
        }
      }
    }
  } {
    move => y.
    split; done.
  } {
    rewrite interval_list_invariant_sing.
    by apply Z_to_N_minus_neq_0.
  } {
    by rewrite interval_list_invariant_cons.
  }
} {
  rewrite /P1 /P2 /P3 /P4.
  move => [H_y12_eq] H_c12_eq [] H_acc' H_s'.
  clear IH P1 P2 P3 P4 H_comp.
  subst.
  split; last split; last split. {
    move => y.
    rewrite !InZ_cons InZ_nil In_elementsZ_single.
    split; last by tauto. {
      move => [] //.
      tauto.
    }
  } {
    move => y.
    rewrite InZ_cons In_elementsZ_single.
    split; first by right.
    move => [] //.
    move => [_] H_y_lt H_lt_y.
    exfalso.
    suff : (y2 + Z.of_N c2 < y2 + Z.of_N c2) by
        apply Z.lt_irrefl.
    apply Z.lt_trans with (m := y) => //.
```

```
    } {
      rewrite interval_list_invariant_sing //.
    } {
      assumption.
    }
  } {
    move => [H_y2_le_y1] [H_yc1_le_yc2] _.
    apply IH. {
      by rewrite interval_list_invariant_sing.
    } {
      move => y.
      rewrite InZ_cons InZ_nil In_elementsZ_single.
      split. {
        move => [] //.
        move => [H_y1_le] H_y_lt.
        split; first done.
        split; omega.
      } {
        move => [?] _.
        by left.
      }
    } {
      assumption.
    }
  } {
    rewrite /P1 /P2 /P3 /P4.
    move => [H_y1_le] [H_yc2_le] _.
    move => [] H_acc' H_s'.
    clear IH P1 P2 P3 P4 H_comp.
    subst.
    split; last split; last split. {
      move => y.
      rewrite !InZ_cons !In_elementsZ_single InZ_nil.
      split. {
        move => [] //.
        move => [H_y2_le] H_y_lt.
        split; last by omega.
        left. omega.
      } {
        move => [H_in_s] [H_y2_le] H_y_lt.
        by left.
      }
```

```
    } {
      tauto.
    } {
      by rewrite interval_list_invariant_sing.
    } {
      by rewrite interval_list_invariant_cons.
    }
  } {
    rewrite /P1 /P2 /P3 /P4.
    move => H_yc2_lt [] H_acc' H_s'.
    clear IH P1 P2 P3 P4 H_comp.
    subst.
    split; last split; last split. {
      move => y.
      rewrite InZ_cons !In_elementsZ_single InZ_nil.
      split; first done.
      move => [] []. {
        move => [H_y1_le_y] H_y_lt_yc1.
        move => [H_y2_le_y] H_y_lt_yc2.
        omega.
      } {
        move => /H_gr_s1'_alt H_lt_y [_] H_y_lt.
        suff : (y1 + Z.of_N c1 < y1). {
          apply Z.nlt_ge.
          apply Z_le_add_r.
        }
        omega.
      }
    } {
      tauto.
    } {
      done.
    } {
      by rewrite interval_list_invariant_cons.
    }
  } {
    rewrite /P1 /P2 /P3 /P4.
    move => H_y1_eq [] H_acc' H_s'.
    clear IH P1 P2 P3 P4 H_comp.
    subst acc' s'.
    split; last split; last split. {
      move => y.
```

```
        rewrite InZ_cons !In_elementsZ_single InZ_nil.
        split; first done.
        move => [] []. {
          move => [H_y1_le_y] H_y_lt_yc1.
          move => [H_y2_le_y] H_y_lt_yc2.
          omega.
        } {
          move => /H_gr_s1'_alt H_lt_y [_] H_y_lt.
          suff : (y1 + Z.of_N c1 < y1). {
            apply Z.nlt_ge.
            apply Z_le_add_r.
          }
          omega.
        }
      } {
        tauto.
      } {
        done.
      } {
        by rewrite interval_list_invariant_cons.
      }
    }
  }
Qed.

Lemma inter_aux2_props :
 ∀ (s2 s1 acc : t),
  interval_list_invariant (rev acc) = true →
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  (∀ x1 x2, InZ x1 acc → InZ x2 s1 → InZ x2 s2 → Z.succ x1 < x2) →
  ((∀ y, (InZ y (inter_aux2 acc s1 s2) ↔
               (InZ y acc) ∨ ((InZ y s1) ∧ InZ y s2))) ∧
  (interval_list_invariant (inter_aux2 acc s1 s2) = true)).
Proof.
  induction s2 as [|[y2 c2] s2' IH]. {
    move => s1 acc.
    move => H_inv_acc _ _ _.
    unfold inter_aux2.
    replace (match s1 with
      | nil => rev' acc
      | _ :: _ => rev' acc
           end) with (rev' acc); last by case s1.
```

```
    rewrite /rev' rev_append_rev app_nil_r.
    split; last done.
    move => y.
    rewrite InZ_nil InZ_rev.
    tauto.
  } {
    intros s1 acc H_inv_acc H_inv_s1.
    rewrite interval_list_invariant_cons.
    move => [H_gr_s2'] [H_c2_neq_0] H_inv_s2'.
    move => H_acc_s12.

    move : H_gr_s2'.
    rewrite interval_list_elements_greater_alt2_def //.
    move => H_gr_s2'.

    rewrite /inter_aux2; fold inter_aux2.
    case_eq s1. {
      move => H_s1_eq.
      split. {
        move => y.
        rewrite /rev' rev_append_rev app_nil_r InZ_nil
                InZ_rev.
        tauto.
      } {
        rewrite /rev' rev_append_rev app_nil_r //.
      }
    } {
      move => [_ _] _ <-.
      case_eq (inter_aux y2 c2 acc s1).
      move => acc' s1' H_inter_aux_eq.

      have H_acc_s1_yc2 : ∀ x1 x2 : Z,
        InZ x1 acc ->
        InZ x2 s1 ->
        List.In x2 (elementsZ_single y2 c2) ->
        Z.succ x1 < x2. {
        intros x1 x2 H_x1_in H_x2_in1 H_x2_in2.
        apply H_acc_s12 => //.
        rewrite InZ_cons; by left.
      }

      move : (inter_aux_props y2 c2 s1 acc H_inv_acc H_inv_s1 H_acc_s1_yc2 H_c2_neq_0).
      rewrite H_inter_aux_eq.
      move => [H_in_acc'] [H_in_s1'_elim] [H_in_s1'_intro]
              [H_inv_acc'] H_inv_s1'.
```

```
have H_in_acc'_s2' : ∀ x1 x2 : Z,
    InZ x1 acc' → InZ x2 s1' → InZ x2 s2' → Z.succ x1 < x2. {
  move ⇒ x1 x2 /H_in_acc'.
  move ⇒ []. {
    move ⇒ H_in_acc H_in_s1' H_in_s2'.
    apply H_acc_s12 ⇒ //. {
      by apply H_in_s1'_elim.
    } {
      rewrite InZ_cons; by right.
    }
  } {
    rewrite In_elementsZ_single.
    move ⇒ [H_in_s1] [_] H_x1_lt _.
    move ⇒ /H_gr_s2' H_lt_x2.
    apply Z.le_lt_trans with (m := y2 + Z.of_N c2) ⇒ //.
    by apply Z.le_succ_l.
  }
}

move : (IH s1' acc' H_inv_acc' H_inv_s1' H_inv_s2' H_in_acc'_s2').
move ⇒ [H_inZ_res] H_inv_res.
split; last assumption.
move ⇒ y.
rewrite H_inZ_res H_in_acc' InZ_cons
        In_elementsZ_single.
split. {
  move ⇒ []; first by tauto.
  move ⇒ [H_y_in_s1' H_y_in_s2'].
  right.
  split; last by right.
  by apply H_in_s1'_elim.
} {
  move ⇒ []. {
    move ⇒ H_y_in_acc.
    by left; left.
  } {
    move ⇒ [H_y_in_s1].
    move ⇒ []. {
      move ⇒ H_in_yc2.
      by left; right.
    } {
      right.
```

```
            split; last assumption.
            apply H_in_s1'_intro => //.
            by apply H_gr_s2'.
          }
        }
      }
    }
  }
Qed.

Lemma inter_InZ :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  ∀ y, (InZ y (inter s1 s2) ↔ InZ y s1 ∧ InZ y s2).
Proof.
  intros s1 s2 H_inv_s1 H_inv_s2 y.
  rewrite /inter.

  move : (inter_aux2_props s2 s1 nil).
  move => [] //.
  move => H_in_inter _.
  rewrite H_in_inter InZ_nil.
  tauto.
Qed.

Lemma inter_invariant :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  interval_list_invariant (inter s1 s2) = true.
Proof.
  intros s1 s2 H_inv_s1 H_inv_s2.
  rewrite /inter.

  move : (inter_aux2_props s2 s1 nil).
  move => [] //.
Qed.

Global Instance inter_ok s1 s2 : ∀ `(Ok s1, Ok s2), Ok (inter s1 s2).
Proof.
  move => H_ok_s1 H_ok_s2.
  move : (H_ok_s1) (H_ok_s2).
  rewrite /Ok /IsOk /is_encoded_elems_list /add.
  move => [H_inv_s1] H_pre1.
  move => [H_inv_s2] H_pre2.
```

```
    split. {
      apply inter_invariant ⇒ //.
    } {
      intros y.
      move : (inter_InZ s1 s2 H_inv_s1 H_inv_s2).
      rewrite /InZ ⇒ →.
      move ⇒ [].
      move ⇒ /H_pre1 //.
    }
  Qed.

  Lemma inter_spec :
   ∀ (s s' : t) (x : elt) (Hs : Ok s) (Hs' : Ok s'),
   In x (inter s s') ↔ In x s ∧ In x s'.
  Proof.
    intros s s' x H_ok H_ok'.
    rewrite !In_InZ.
    rewrite inter_InZ ⇒ //. {
      apply H_ok.
    } {
      apply H_ok'.
    }
  Qed.
```

diff specification

```
  Lemma diff_aux_alt_def :
    ∀ (y2 : Z) (c2 : N) (s : t) acc,
      diff_aux y2 c2 acc s = match diff_aux y2 c2 nil s with
                                 (acc', s') ⇒ (acc' ++ acc, s')
                               end.
  Proof.
    intros y2 c2.
    induction s as [|| [y1 c1] acc' IH] ⇒ acc. {
      rewrite /diff_aux app_nil_l //.
    } {
      simpl.
      case_eq (diff_aux y2 c2 nil acc') ⇒ acc'' s'' H_eq.
      case (interval_compare (y1, c1) (y2, c2));
        rewrite ?(IH ((y1, c1)::acc)) ?(IH ((y1, c1)::nil))
                ?(IH acc) ?(IH ((y1, Z.to_N (y2 - y1)) :: acc))
                ?(IH ((y1, Z.to_N (y2 - y1)) :: nil)) ?H_eq;
        rewrite ?insert_intervalZ_guarded_app -?app_assoc -?app_comm_cons //.
```

```
    }
  Qed.

  Lemma diff_aux_props :
    ∀ (y2 : Z) (c2 : N) (s : t) acc,
      interval_list_invariant (List.rev acc) = true →
      interval_list_invariant s = true →
      (∀ x1 x2, InZ x1 acc → InZ x2 s → Z.succ x1 < x2) →
      (∀ x, InZ x acc → x < y2) →
      (c2 ≠ 0%N) →
      match (diff_aux y2 c2 acc s) with
        (acc', s') ⇒ (∀ y, InZ y (List.rev_append acc' s') ↔
                                    InZ y (List.rev_append acc s) ∧ ~(List.In y (elementsZ_single
y2 c2))) ∧
                      (interval_list_invariant (List.rev_append acc' s') = true) ∧
                      (∀ x, InZ x acc' → x < y2 + Z.of_N c2)
      end.
  Proof.
    intros y2 c2.
    induction s as [|[y1 c1] s1' IH] ⇒ acc. {
      rewrite /diff_aux -rev_alt.
      move ⇒ H_inv_acc _ _ H_in_acc H_c2_neq.
      split; last split. {
        move ⇒ y; split; last by move ⇒ [] //.
        rewrite InZ_rev.
        move ⇒ H_in. split ⇒ //.
        move ⇒ /In_elementsZ_single ⇒ [] [] /Z.nlt_ge H_neq.
        contradict H_neq.
        by apply H_in_acc.
      } {
        assumption.
      } {
        intros x H_in_acc'.
        apply Z.lt_le_trans with (m := y2). {
          by apply H_in_acc.
        } {
          by apply Z_le_add_r.
        }
      }
    } {
      rewrite interval_list_invariant_cons.
      move ⇒ H_inv_acc [H_gr_s1'] [H_c1_neq_0] H_inv_s1'.
      move ⇒ H_in_s1 H_in_acc H_c2_neq_0.
```

```
rewrite diff_aux_alt_def.
case_eq (diff_aux y2 c2 nil ((y1 , c1) :: s1')).
move ⇒ acc' s' H_diff_aux_eq.
set P1 := ∀ y : Z,
  (InZ y acc' ∨ InZ y s') ↔
  InZ y ((y1 , c1) :: s1') ∧ ¬ List.In y (elementsZ_single y2 c2).
set P2 := interval_list_invariant (rev acc' ++ s') = true.
set P3 := ∀ x : Z, InZ x acc' → (x < y2 + Z.of_N c2).
suff : (P1 ∧ P2 ∧ P3). {
  move ⇒ [H_P1] [H_P2] H_P3.
  split; last split. {
    move ⇒ y.
    move : (H_P1 y).
    rewrite !rev_append_rev rev_app_distr !InZ_app
            !InZ_rev In_elementsZ_single.
    suff : (InZ y acc → ¬ y2 ≤ y < y2 + Z.of_N c2). {
      tauto.
    }
    move ⇒ /H_in_acc H_y_lt [H_y_ge] _.
    contradict H_y_ge.
    by apply Zlt_not_le.
  } {
    rewrite rev_append_rev rev_app_distr -app_assoc.
    apply interval_list_invariant_app_intro ⇒ //.
    move ⇒ x1 x2.
    rewrite InZ_app !InZ_rev.
    move ⇒ H_in_acc' H_x2_in_s'.
    suff : (InZ x2 ((y1 , c1)::s1')). {
      by apply H_in_s1.
    }
    move : (H_P1 x2).
    tauto.
  } {
    move ⇒ x.
    rewrite InZ_app.
    move ⇒ []. {
      apply H_P3.
    } {
      move ⇒ /H_in_acc H_x_lt.
      eapply Z.lt_trans; eauto.
      by apply Z_lt_add_r.
    }
```

```
    }
  }

  move : (H_gr_s1').
  rewrite interval_list_elements_greater_alt2_def => // => H_gr_s1'_alt.

  have : ∀ (acc : list (Z × N)),
    interval_list_invariant (rev acc) = true →
    (∀ x : Z,
        InZ x acc ↔
        ((y1 ≤ x < y1 + Z.of_N c1) ∧ (x < y2))) →
    (y1 + Z.of_N c1 ≤ y2 + Z.of_N c2) →
    (diff_aux y2 c2 acc s1' = (acc', s')) →
    P1 ∧ P2 ∧ P3. {

    intros acc0 H_inv_acc0 H_in_acc0 H_c1_before H_diff_aux_eq0.
    have H_in_s1' : (∀ x1 x2 : Z,
                         InZ x1 acc0 → InZ x2 s1' → Z.succ x1 < x2). {
      intros x1 x2 H_x1_in_acc0.
      move => /H_gr_s1'_alt.
      eapply Z.le_lt_trans.
      move : H_x1_in_acc0.
      rewrite Z.le_succ_l H_in_acc0.
      tauto.
    }
    have H_in_acc0' : (∀ x : Z, InZ x acc0 → x < y2). {
      move => x.
      rewrite H_in_acc0.
      move => [_] //.
    }
    move : (IH acc0 H_inv_acc0 H_inv_s1' H_in_s1' H_in_acc0' H_c2_neq_0).
    rewrite H_diff_aux_eq0 !rev_append_rev.
    move => [H1] [H2] H3.
    split; last split => //. {
      move => y.
      move : (H1 y).
      rewrite !InZ_app !InZ_rev In_elementsZ_single.
      move => →.
      rewrite InZ_cons In_elementsZ_single.
      split. {
        rewrite H_in_acc0 -(Z.nle_gt y2 y).
        tauto.
      } {
```

```
        rewrite H_in_acc0 -(Z.nle_gt y2 y).
        move ⇒ [] H_in H_nin_i2.
        split; last by assumption.
        move : H_in ⇒ [] H_in; last by right.
        left.
        omega.
      }
    }
  }
  move ⇒ {IH} IH.

  clear H_inv_acc H_in_s1 H_in_acc acc.
  move : (interval_compare_elim y1 c1 y2 c2) H_diff_aux_eq.
  unfold diff_aux.
  case_eq (interval_compare (y1 , c1) (y2 , c2)) ⇒ H_comp;
                                                    fold diff_aux. {
    move ⇒ H_lt_y2.
    apply IH. {
      by rewrite interval_list_invariant_sing.
    } {
      move ⇒ x.
      rewrite InZ_cons In_elementsZ_single.
      split; last by tauto.
      move ⇒ []; last done.
      move ⇒ [H_y1_le H_x_lt].
      split; first done.
      eapply Z.lt_trans; eauto.
    } {
      apply Z.le_trans with (m := y2).
        - by apply Z.lt_le_incl.
        - by apply Z_le_add_r.
    }
  } {
    move ⇒ H_eq_y2.
    apply IH. {
      by rewrite interval_list_invariant_sing.
    } {
      move ⇒ x.
      rewrite InZ_cons In_elementsZ_single -H_eq_y2.
      split; last by tauto.
      move ⇒ []; last done.
      move ⇒ []. done.
    } {
```

```
      rewrite H_eq_y2.
      by apply Z_le_add_r.
    }
  } {
    move ⇒ [H_y1_lt_y2] [H_y2_lt_yc1] H_yc1_lt_yc2.
    apply IH. {
      rewrite interval_list_invariant_sing.
      by apply Z_to_N_minus_neq_0.
    } {
      move ⇒ x.
      rewrite InZ_cons In_elementsZ_single Z2N.id; last omega.
      replace (y1 + (y2 - y1)) with y2 by omega.
      split; last tauto.
      move ⇒ [] //.
      move ⇒ [H_y1_le] H_x_lt.
      repeat split ⇒ //.
      apply Z.lt_trans with (m := y2) ⇒ //.
    } {
      by apply Z.lt_le_incl.
    }
  } {
    rewrite /P1 /P2 /P3.
    move ⇒ [H_y2_lt] [H_y1_lt] H_yc1_lt [H_acc'] H_s'.
    clear IH P1 P2 P3 H_comp.
    subst.
    have H_yc1_intro : y2 + Z.of_N c2 + Z.of_N (Z.to_N (y1 + Z.of_N c1 - (y2 +
Z.of_N c2))) = y1 + Z.of_N c1. {
      rewrite Z2N.id; omega.
    }
    have H_nin_yc2 : ∀ y,
        InZ y s1' → ¬ y2 ≤ y < y2 + Z.of_N c2. {
      move ⇒ y /H_gr_s1'_alt H_lt_y.
      move ⇒ [H_y2_le_y].
      apply Z.le_ngt, Z.lt_le_incl.
      by apply Z.lt_trans with (m := y1 + Z.of_N c1).
    }
    split; last split. {
      move ⇒ y.
      rewrite !InZ_cons !In_elementsZ_single H_yc1_intro.
      split. {
        move ⇒ [] //.
        move ⇒ []. {
```

```
          move => [H_le_y] H_y_lt.
          split. {
            left; omega.
          } {
            move => [_].
            by apply Z.nlt_ge.
          }
        } {
          move : (H_nin_yc2 y). tauto.
        }
      } {
        move => [] []; last by right; right.
        move => [H_y_ge] H_y_lt_yc1 H_nin_yc2'.
        right; left. omega.
      }
    } {
      rewrite interval_list_invariant_cons H_yc1_intro.
      split => //.
      split => //.
      by apply Z_to_N_minus_neq_0.
    } {
      move => [] //.
    }
  } {
    rewrite /P1 /P2 /P3.
    move => [H_y12_eq] H_c12_eq [] H_acc' H_s'.
    clear IH P1 P2 P3 H_comp.
    subst.
    split; last split. {
      move => y.
      rewrite InZ_cons In_elementsZ_single.
      split; last by tauto. {
        move => [] //.
        move => H_y_in.
        split; first by right.
        move => [] _.
        by apply Z.nlt_ge, Z.lt_le_incl, H_gr_s1'_alt.
      }
    } {
      apply H_inv_s1'.
    } {
      move => x [] //.
```

```
      }
    } {
      move => [H_y2_le_y1] [H_yc1_le_yc2] _.
      apply IH. {
        done.
      } {
        move => x.
        split; first done.
        omega.
      } {
        assumption.
      }
    } {
      rewrite /P1 /P2 /P3.
      move => [H_y1_le] [H_yc2_le_yc1] _ [] H_acc' H_s'.
      clear IH P1 P2 P3 H_comp.
      subst.
      have H_yc1_intro : y2 + Z.of_N c2 + Z.of_N (Z.to_N (y1 + Z.of_N c1 - (y2 +
Z.of_N c2))) = y1 + Z.of_N c1. {
        rewrite Z2N.id; omega.
      }
      have H_y1_intro : y1 + Z.of_N (Z.to_N (y2 - y1)) = y2. {
        rewrite Z2N.id; omega.
      }
      split; last split. {
        move => y.
        rewrite !InZ_insert_intervalZ_guarded
                !InZ_cons !In_elementsZ_single
                H_yc1_intro H_y1_intro InZ_nil.
        split. {
          rewrite -!or_assoc.
          move => [[[]|]|] //. {
            move => [H_y1_le_y] H_y_lt.
            split. {
              left.
              split => //.
              apply Z.lt_le_trans with (m := y2 + Z.of_N c2) => //.
              apply Z.lt_trans with (m := y2) => //.
              by apply Z_lt_add_r.
            } {
              move => [] /Z.le_ngt //.
            }
```

```
      } {
        move => [H_y2c_le_y] H_y_lt_yc1.
        split. {
          left.
          split => //.
          apply Z.le_trans with (m := y2 + Z.of_N c2) => //.
          apply Z.le_trans with (m := y2) => //.
          apply Z_le_add_r.
        } {
          move => [] _ /Z.lt_nge //.
        }
      } {
        move => H_y_in_s1'.
        split; first by right.
        suff H_suff : y2 + Z.of_N c2 <= y. {
          move => [] _ /Z.lt_nge //.
        }
        apply Z.le_trans with (m := y1 + Z.of_N c1) => //.
        apply Z.lt_le_incl.
        by apply H_gr_s1'_alt.
      }
    } {
      move => [] []; last by tauto.
      move => [H_y1_le_y] H_y_lt H_neq_y2.
      apply not_and in H_neq_y2; last by apply Z.le_decidable.
      case H_neq_y2. {
        move => /Z.nle_gt H_y_lt'.
        left; left; done.
      } {
        move => /Z.nlt_ge H_le_y.
        right; left; done.
      }
    }
  } {
    rewrite insert_intervalZ_guarded_rev_nil_app.
    rewrite !interval_list_invariant_insert_intervalZ_guarded => //. {
      rewrite H_yc1_intro => //.
    } {
      rewrite /insert_intervalZ_guarded.
      case_eq ((Z.to_N (y1 + Z.of_N c1 - (y2 + Z.of_N c2)) =? 0)%N). {
        rewrite H_y1_intro.
        move => /N.eqb_eq /N2Z.inj_iff.
```

```
          rewrite Z2N.id; last first. {
            by apply Z.le_0_sub.
          }
          move => /Zminus_eq H_yc1_eq.
          eapply interval_list_elements_greater_impl;
            last apply H_gr_s1'.
          rewrite H_yc1_eq.
          apply Z_le_add_r.
        } {
          move => _.
          rewrite interval_list_elements_greater_cons
                    H_y1_intro.
          by apply Z_lt_add_r.
        }
      }
    } {
      move => x.
      rewrite InZ_insert_intervalZ_guarded InZ_cons In_elementsZ_single H_y1_intro InZ_nil.
      move => [] //.
      move => [_] H_x_lt.
      apply Z.lt_le_trans with (m := y2) => //.
      apply Z_le_add_r.
    }
  } {
    rewrite /P1 /P2 /P3.
    move => H_yc2_lt [] H_acc' H_s'.
    clear IH P1 P2 P3 H_comp.
    subst.
    split; last split. {
      move => y.
      rewrite InZ_cons In_elementsZ_single.
      split; last by tauto. {
        move => [] //.
        move => H_y_in.
        split; first assumption.
        rewrite In_elementsZ_single.
        move => [] H_y2_le H_y_lt.
        case H_y_in; first by omega.
        move => /H_gr_s1'_alt H_lt_y.
        suff : y2 + Z.of_N c2 < y. {
          move => ?. omega.
        }
```

```
          apply Z.lt_trans with (m := y1 + Z.of_N c1) => //.
          apply Z.lt_le_trans with (m := y1) => //.
          apply Z_le_add_r.
        }
      } {
        by rewrite interval_list_invariant_cons.
      } {
        done.
      }
    } {
      rewrite /P1 /P2 /P3.
      move => H_yc2_eq [] H_acc' H_s'.
      clear IH P1 P2 P3 H_comp.
      subst.
      split; last split. {
        move => y.
        rewrite InZ_cons In_elementsZ_single.
        split; last by tauto. {
          move => [] //.
          move => H_y_in.
          split; first assumption.
          rewrite In_elementsZ_single.
          move => [] H_y2_le H_y_lt.
          case H_y_in; first by omega.
          move => /H_gr_s1'_alt H_lt_y.
          suff : y2 + Z.of_N c2 < y. {
            move => ?. omega.
          }
          apply Z.lt_trans with (m := (y2 + Z.of_N c2) + Z.of_N c1) => //.
          by apply Z_lt_add_r.
        }
      } {
        by rewrite interval_list_invariant_cons.
      } {
        done.
      }
    }
  }
Qed.

Lemma diff_aux2_props :
  ∀ (s2 s1 acc : t),
    interval_list_invariant (rev_append acc s1) = true →
```

```
  interval_list_invariant s2 = true →
  (∀ x1 x2, InZ x1 acc → InZ x2 s2 → Z.succ x1 < x2) →
  ((∀ y, (InZ y (diff_aux2 acc s1 s2) ↔
               ((InZ y acc) ∨ (InZ y s1)) ∧ ¬InZ y s2)) ∧
  (interval_list_invariant (diff_aux2 acc s1 s2) = true)).
Proof.
  induction s2 as [|| [y2 c2] s2' IH]. {
    move ⇒ s1 acc H_inv_acc_s1 _ _.
    rewrite /diff_aux2.
    replace (match s1 with
      | nil ⇒ rev_append acc s1
      | _ :: _ ⇒ rev_append acc s1
     end) with (rev_append acc s1); last by case s1.
    split. {
      move ⇒ y.
      rewrite rev_append_rev InZ_app InZ_rev InZ_nil.
      tauto.
    } {
      assumption.
    }
  } {
    intros s1 acc H_inv_acc_s1.
    rewrite interval_list_invariant_cons.
    move ⇒ [H_gr_s2'] [H_c2_neq_0] H_inv_s2'.
    move ⇒ H_acc_s2.
    rewrite /diff_aux2; fold diff_aux2.
    case_eq s1. {
      move ⇒ H_s1_eq.
      split. {
        move ⇒ y.
        rewrite rev_append_rev InZ_app InZ_nil InZ_rev.
        split; last tauto.
        move ⇒ [] // H_y_in.
        split; first by left.
        move ⇒ H_y_in'.
        move : (H_acc_s2 _ _ H_y_in H_y_in').
        apply Z.nlt_succ_diag_l.
      } {
        move : H_inv_acc_s1.
         by rewrite H_s1_eq.
      }
    } {
```

```
    move => [_ _] _ <-.
    case_eq (diff_aux y2 c2 acc s1).
    move => acc' s1' H_diff_aux_eq.
    have H_acc_lt_y2 : (∀ x : Z, InZ x acc → x < y2). {
      move => x H_x_in.
      have H_y2_in: (InZ y2 ((y2,c2) :: s2')). {
        rewrite InZ_cons.
        left.
        by apply In_elementsZ_single_hd.
      }
      move : (H_acc_s2 _ _ H_x_in H_y2_in).
      apply Z.lt_trans, Z.lt_succ_diag_r.
    }

    have [H_inv_acc [H_inv_s1 H_acc_s1]] :
      interval_list_invariant (rev acc) = true ∧
      interval_list_invariant s1 = true ∧
      (∀ x1 x2 : Z,
         InZ x1 acc → InZ x2 s1 → Z.succ x1 < x2). {

      move : H_inv_acc_s1.
      rewrite rev_append_rev.
      move => /interval_list_invariant_app_elim.
      move => [?] [?] H_x.
      split; first assumption.
      split; first assumption.
      move => x1 x2 H_in_x1.
      apply H_x.
      by rewrite InZ_rev.
    }

    move : (diff_aux_props y2 c2 s1 acc H_inv_acc H_inv_s1 H_acc_s1 H_acc_lt_y2
H_c2_neq_0).
    rewrite !H_diff_aux_eq.
    move => [H_inZ_res] [H_inv_res] H_inZ_acc'.
    have H_acc'_s2' : (∀ x1 x2 : Z,
                                InZ x1 acc' → InZ x2 s2' → Z.succ x1 < x2). {
      move => x1 x2 H_x1_in H_x2_in.
      apply Z.le_lt_trans with (m := y2 + Z.of_N c2). {
        apply Z.le_succ_l.
        by apply H_inZ_acc'.
      } {
```

```
        move : H_gr_s2'.
        rewrite interval_list_elements_greater_alt2_def //.
        move ⇒ H_gr_s2'.
        by apply H_gr_s2'.
      }
    }

    move : (IH s1' acc' H_inv_res H_inv_s2' H_acc'_s2').
    move ⇒ [] H_inZ_diff_res →.
    split; last done.
    move ⇒ y.
    rewrite H_inZ_diff_res.
    move : (H_inZ_res y).
    rewrite !rev_append_rev !InZ_app !InZ_rev InZ_cons.
    move ⇒ →.
    tauto.
  }
}
Qed.

Lemma diff_InZ :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  ∀ y, (InZ y (diff s1 s2) ↔ InZ y s1 ∧ ¬InZ y s2).
Proof.
  intros s1 s2 H_inv_s1 H_inv_s2 y.
  rewrite /diff.
  move : (diff_aux2_props s2 s1 nil).
  move ⇒ [] //.
  move ⇒ H_in_diff _.
  rewrite H_in_diff InZ_nil.
  tauto.
Qed.

Lemma diff_invariant :
 ∀ (s1 s2 : t),
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  interval_list_invariant (diff s1 s2) = true.
Proof.
  intros s1 s2 H_inv_s1 H_inv_s2.
  rewrite /diff.
```

```
    move : (diff_aux2_props s2 s1 nil).
    move => [] //.
  Qed.

  Global Instance diff_ok s1 s2 : ∀ `(Ok s1, Ok s2), Ok (diff s1 s2).
  Proof.
    move => H_ok_s1 H_ok_s2.
    move : (H_ok_s1) (H_ok_s2).
    rewrite /Ok /IsOk /is_encoded_elems_list /add.
    move => [H_inv_s1] H_pre1.
    move => [H_inv_s2] H_pre2.
    split. {
      apply diff_invariant => //.
    } {
      intros y.
      move : (diff_InZ s1 s2 H_inv_s1 H_inv_s2).
      rewrite /InZ => ->.
      move => [].
      move => /H_pre1 //.
    }
  Qed.

  Lemma diff_spec :
   ∀ (s s' : t) (x : elt) (Hs : Ok s) (Hs' : Ok s'),
   In x (diff s s') ↔ In x s ∧ ¬In x s'.
  Proof.
    intros s s' x H_ok H_ok'.
    rewrite !In_InZ.
    rewrite diff_InZ => //. {
      apply H_ok.
    } {
      apply H_ok'.
    }
  Qed.
```

remove specification

```
  Lemma removeZ_alt_def : ∀ x s acc,
    interval_list_invariant s = true →
    removeZ_aux acc x s = match diff_aux x (1%N) acc s with
                          (acc', s') ⇒ rev_append acc' s'
                        end.
  Proof.
    intros y2.
```

```
induction s as [| [y1 c1] s' IH]; first done.
move ⇒ acc.
rewrite interval_list_invariant_cons /=.
move ⇒ [H_gr] [H_c1_neq_0] H_inv_s'.
move : (interval_1_compare_elim y2 y1 c1).
rewrite interval_1_compare_alt_def.
rewrite !(interval_compare_swap y1 c1 y2); last first. {
  right. done.
}
move : (interval_compare_elim y1 c1 y2 (1%N)).
case_eq (interval_compare (y1, c1) (y2, (1%N))) ⇒ H_eq. {
  move ⇒ H_lt_y2 _.
  have H_yc2_nlt : ~(y2 < y1 + Z.of_N c1). {
    by apply Z.nlt_ge, Z.lt_le_incl.
  }
  have H_y2_nlt : ~(y2 < y1). {
    move ⇒ H_y2_y1.
    apply H_yc2_nlt.
    apply Z.lt_le_trans with (m := y1) ⇒ //.
    apply Z_le_add_r.
  }
  move : (H_y2_nlt) (H_yc2_nlt) ⇒ /Z.ltb_nlt → /Z.ltb_nlt →.
  rewrite IH //.
} {
  move ⇒ H_lt_y2 _.
  have H_yc2_nlt : ~(y2 < y1 + Z.of_N c1). {
    apply Z.nlt_ge.
    rewrite H_lt_y2.
    apply Z.le_refl.
  }
  have H_y2_nlt : ~(y2 < y1). {
    move ⇒ H_y2_y1.
    apply H_yc2_nlt.
    apply Z.lt_le_trans with (m := y1) ⇒ //.
    apply Z_le_add_r.
  }
  move : (H_y2_nlt) (H_yc2_nlt) ⇒ /Z.ltb_nlt → /Z.ltb_nlt →.
  rewrite IH //.
} {
  done.
} {
  done.
```

```
} {
  move => [H_y1_eq] H_c1_eq.
  move => [] // .
  move => [H_lt_y2] H_y2_lt.
  have H_y2_nlt : ~(y2 < y1). {
    apply Z.nlt_ge => //.
  }
  move : (H_y2_nlt) (H_y2_lt) => /Z.ltb_nlt -> /Z.ltb_lt ->.
  rewrite /insert_intervalZ_guarded.
  have -> : (Z.to_N (y1 + Z.of_N c1 - Z.succ y2) =? 0 = true)%N. {
    rewrite H_y1_eq H_c1_eq Z.add_1_r Z.sub_diag //.
  }

  have -> : (Z.to_N (y2 - y1) =? 0 = true)%N. {
    rewrite H_y1_eq Z.sub_diag //.
  }
  done.
} {
  move => [H_y2_le] [H_yc1_le] _.
  move => [] //.
  move => [H_y1_le] H_y2_lt.
  have H_y2_nlt : ~(y2 < y1). {
    apply Z.nlt_ge => //.
  }
  move : (H_y2_nlt) (H_y2_lt) => /Z.ltb_nlt -> /Z.ltb_lt ->.
  have H_y1_eq : (y1 = y2) by omega.
  have H_yc1_eq : (y1 + Z.of_N c1 = Z.succ y2). {
    apply Z.le_antisymm. {
      move : H_yc1_le.
      rewrite Z.add_1_r //.
    } {
      by apply Z.le_succ_l.
    }
  }

  rewrite /insert_intervalZ_guarded.
  have -> : (Z.to_N (y1 + Z.of_N c1 - Z.succ y2) =? 0 = true)%N. {
    rewrite H_yc1_eq Z.sub_diag //.
  }

  have -> : (Z.to_N (y2 - y1) =? 0 = true)%N. {
    rewrite H_y1_eq Z.sub_diag //.
```

```
    }

    suff -> : diff_aux y2 (1%N) acc s' = (acc, s') by done.

    move : H_gr.
    rewrite H_yc1_eq.
    case s' as [| [y' c'] s'']. {
      done.
    } {
      rewrite interval_list_elements_greater_cons /=
                /interval_compare.
      move => H_lt_y'.
      have -> : y2 + Z.of_N 1 ?= y' = Lt. {
        apply Z.compare_lt_iff.
        by rewrite Z.add_1_r.
      }
      done.
    }
  } {
    move => [H_y1_le] [H_yc2_le] _.
    move => [] //.
    move => [_] H_y2_lt.
    have H_y2_nlt : ~(y2 < y1). {
      apply Z.nlt_ge => //.
    }
    move : (H_y2_nlt) (H_y2_lt) => /Z.ltb_nlt -> /Z.ltb_lt ->.
    rewrite !rev_append_rev /insert_intervalZ_guarded Z.add_1_r.
    case ((Z.to_N (y2 - y1) =? 0)%N), (Z.to_N (y1 + Z.of_N c1 - Z.succ y2) =? 0)%N.
{
      reflexivity.
    } {
      rewrite /= -!app_assoc //.
    } {
      reflexivity.
    } {
      rewrite /= -!app_assoc //.
    }
  } {
    move => _ H_y2_lt'.
    have H_y2_lt : (y2 < y1). {
      apply Z.lt_trans with (m := Z.succ y2) => //.
      apply Z.lt_succ_diag_r.
    }
```

```
      move : (H_y2_lt) ⇒ /Z.ltb_lt → //.
    } {
      move ⇒ _ H_y1_eq.
      have H_y2_lt : (y2 < y1). {
        rewrite H_y1_eq.
        apply Z.lt_succ_diag_r.
      }
      move : (H_y2_lt) ⇒ /Z.ltb_lt → //.
    }
  Qed.
```

```
  Lemma removeZ_interval_list_invariant : ∀ s x, interval_list_invariant s = true → interval_list_invariant
(removeZ x s) = true.
  Proof.
    intros s x H_inv.
    rewrite /removeZ removeZ_alt_def //.
    move : (diff_aux_props x (1%N) s nil).
    case_eq (diff_aux x 1%N nil s).
    move ⇒ acc' s' H_eq [] //.
    move ⇒ _ [] //.
  Qed.
```

```
  Lemma removeZ_spec :
   ∀ (s : t) (x y : Z) (Hs : interval_list_invariant s = true),
    InZ y (removeZ x s) ↔ InZ y s ∧ ¬Z.eq y x.
  Proof.
    intros s x y H_inv.
    rewrite /removeZ removeZ_alt_def //.
    move : (diff_aux_props x (1%N) s nil).
    case_eq (diff_aux x 1%N nil s).
    move ⇒ acc' s' H_eq [] //.
    move ⇒ → _.
    rewrite rev_append_rev InZ_app InZ_rev InZ_nil
            In_elementsZ_single1.
    split; move ⇒ [H1 H2]; split ⇒ //;
      move ⇒ H3; apply H2; by rewrite H3.
  Qed.
```

```
  Global Instance remove_ok s x : ∀ `(Ok s), Ok (remove x s).
  Proof.
    rewrite /Ok /interval_list_invariant /remove.
    move ⇒ [H_is_ok_s H_enc_s].
    split. {
      by apply removeZ_interval_list_invariant.
    } {
```

```
    rewrite /is_encoded_elems_list => y.
    move : (removeZ_spec s (Enc.encode x) y H_is_ok_s).
    rewrite /InZ => -> [] H_y_in _.
    apply H_enc_s => //.
  }
Qed.

Lemma remove_spec :
 ∀ (s : t) (x y : elt) (Hs : Ok s),
  In y (remove x s) ↔ In y s ∧ ¬Enc.E.eq y x.
Proof.
  intros s x y Hs.
  have H_rs := (remove_ok s x Hs).
  rewrite /remove !In_InZ.
  rewrite removeZ_spec. {
    rewrite Enc.encode_eq //.
  } {
    apply Hs.
  }
Qed.
```

remove_list specification

```
Lemma remove_list_ok : ∀ l s, Ok s → Ok (remove_list l s).
Proof.
  induction l as [| x l' IH]. {
    done.
  } {
    move => s H_s_ok /=.
    apply IH.
    by apply remove_ok.
  }
Qed.

Lemma remove_list_spec : ∀ x l s, Ok s →
   (In x (remove_list l s) ↔ ~(InA Enc.E.eq x l) ∧ In x s).
Proof.
  move => x.
  induction l as [| y l' IH]. {
    intros s H.
    rewrite /= InA_nil.
    tauto.
  } {
    move => s H_ok /=.
```

```
      rewrite IH remove_spec InA_cons.
      tauto.
    }
  Qed.
```

subset specification

```
  Lemma subset_flatten_alt_def : ∀ (s1 s2 : t),
    subset s1 s2 =
    match (s1, s2) with
    | (nil, _) ⇒ true
    | (_ :: _, nil) ⇒ false
    | ((y1, c1) :: l1, (y2, c2) :: l2) ⇒
        match (interval_compare (y1, c1) (y2,c2)) with
          | ICR_before ⇒ false
          | ICR_before_touch ⇒ false
          | ICR_after ⇒ subset s1 l2
          | ICR_after_touch ⇒ false
          | ICR_overlap_before ⇒ false
          | ICR_overlap_after ⇒ false
          | ICR_equal ⇒ subset l1 l2
          | ICR_subsume_1 ⇒ subset l1 s2
          | ICR_subsume_2 ⇒ false
        end
    end.
  Proof.
    intros s1 s2.
    case s1, s2 ⇒ //.
  Qed.

  Lemma subset_props_aux : ∀ y1 c1 l1 y2 c2 l2,
    (∃ y, InZ y ((y1, c1) :: l1) ∧ ¬InZ y ((y2, c2) :: l2)) →
    (false = true ↔
    (∀ y : Z,
        InZ y ((y1, c1) :: l1) → InZ y ((y2, c2) :: l2))).
  Proof.
    intros y1 c1 l1 y2 c2 l2.
    move ⇒ [y] [H_y_in H_y_nin].
    split; first done.
    move ⇒ H.
    contradict H_y_nin.
    by apply H.
  Qed.
```

```
Lemma subset_props_aux_before : ∀ y1 c1 l1 y2 c2 l2,
  (c1 ≠ 0%N) →
  interval_list_invariant ((y2 , c2) :: l2) = true →
  (y1 < y2) →
  (false = true ↔
  (∀ y : Z,
      InZ y ((y1 , c1) :: l1) → InZ y ((y2 , c2) :: l2))).
Proof.
  intros y1 c1 l1 y2 c2 l2.
  rewrite interval_list_invariant_cons.
  move ⇒ H_c1_neq_0 [H_gr] [H_inv_l2] H_c2_neq_0 H_y1_lt.
  apply subset_props_aux.
  ∃ y1.
  split. {
    rewrite InZ_cons.
    left.
    by apply In_elementsZ_single_hd.
  } {
    rewrite InZ_cons.
    suff : ¬ (List.In y1 (elementsZ_single y2 c2)) ∧ ¬ InZ y1 l2 by tauto.
    split. {
      rewrite In_elementsZ_single.
      move ⇒ [] /Z.le_ngt //.
    } {
      eapply Nin_elements_greater; eauto.
      apply Z.le_trans with (m := y2). {
        by apply Z.lt_le_incl.
      } {
        apply Z_le_add_r.
      }
    }
  }
Qed.

Lemma subset_props : ∀ s1 s2 : t,
  interval_list_invariant s1 = true →
  interval_list_invariant s2 = true →
  (subset s1 s2 = true ↔
   (∀ y, InZ y s1 → InZ y s2)).
Proof.
  induction s1 as [|[y1 c1] l1 IH1]. {
    move ⇒ s2 _ _.
    rewrite subset_flatten_alt_def.
```

```
  split; done.
} {
  induction s2 as [| [y2 c2] l2 IH2]. {
    rewrite interval_list_invariant_cons
            subset_flatten_alt_def.
    move => [_] [H_c1_neq_0] _ _.
    split => //.
    move => H; move : (H y1).
    rewrite InZ_nil => {H} H.
    contradict H.
    rewrite InZ_cons; left.
    by apply In_elementsZ_single_hd.
  } {
    move => H_inv_s1 H_inv_s2.
    move : (H_inv_s1) (H_inv_s2).
    rewrite !interval_list_invariant_cons.
    move => [H_gr_l1] [H_c1_neq_0] H_inv_l1.
    move => [H_gr_l2] [H_c2_neq_0] H_inv_l2.
    move : (IH2 H_inv_s1 H_inv_l2) => {IH2} IH2.
    have : ∀ s2 : t,
      interval_list_invariant s2 = true →
      (subset l1 s2 = true ↔
      (∀ y : Z, InZ y l1 → InZ y s2)). {
      intros. by apply IH1.
    }
    move => {IH1} IH1.

    have H_yc2_nin : ¬ InZ (y2 + Z.of_N c2) ((y2 , c2) :: l2). {
      rewrite !InZ_cons !In_elementsZ_single.
      move => []. {
        move => [_] /Z.lt_irrefl //.
      } {
        eapply Nin_elements_greater; eauto.
        apply Z.le_refl.
      }
    }

    rewrite subset_flatten_alt_def.
    move : (interval_compare_elim y1 c1 y2 c2).
    case (interval_compare (y1 , c1) (y2 , c2)). {
      move => H_lt_y2.
      apply subset_props_aux_before => //.
      apply Z.le_lt_trans with (m := y1 + Z.of_N c1) => //.
```

```
  apply Z_le_add_r.
} {
  move => H_y2_eq.
  apply subset_props_aux_before => //.
  rewrite -H_y2_eq.
  by apply Z_lt_add_r.
} {
  move => [H_y1_lt] _.
  apply subset_props_aux_before => //.
} {
  move => [H_y2_lt] [H_y1_lt] H_yc2_lt.
  apply subset_props_aux.
  ∃ (y2 + Z.of_N c2).
  split => //.
  rewrite !InZ_cons !In_elementsZ_single.
  left.
  split => //.
  by apply Z.lt_le_incl.
} {
  move => [H_y1_eq] H_c1_eq; subst.
  rewrite IH1 => //.
  split; move => H_pre y; move : (H_pre y) => {H_pre};
    rewrite !InZ_cons. {
    tauto.
  } {
    move => H_pre H_y_in_l1.
    suff : ~(List.In y (elementsZ_single y2 c2)). {
      tauto.
    }
    move : H_gr_l1.
    rewrite interval_list_elements_greater_alt2_def
      // In_elementsZ_single.
    move => H; move : (H y H_y_in_l1) => {H}.
    move => /Z.lt_ngt H_neq [_] //.
  }
} {
  move => [H_y2_lt_y1] [H_yc1_le] _.
  rewrite IH1.
  split; move => H_pre y; move : (H_pre y) => {H_pre};
    rewrite !InZ_cons. {
    move => H []; last apply H. move => {H}.
    rewrite !In_elementsZ_single.
```

```
    move => [H_y1_le] H_y_lt.
    left.
    omega.
  } {
    move => H_pre H_y_in_l1.
    apply H_pre.
    by right.
  } {
    assumption.
  }
} {
  move => [H_y1_le] [_] []. {
    apply subset_props_aux_before => //.
  } {
    move => H_yc2_lt.
    apply subset_props_aux.
    ∃ (y2 + Z.of_N c2).
    split => //.
    rewrite !InZ_cons !In_elementsZ_single.
    left.
    split => //.
    apply Z.le_trans with (m := y2) => //.
    apply Z_le_add_r.
  }
} {
  move => H_yc2_lt_y1.
  rewrite IH2.
  split; move => H_pre y; move : (H_pre y) => {H_pre};
                                              rewrite !InZ_cons. {
    tauto.
  } {
    rewrite !In_elementsZ_single.
    move => H_pre H_y_in.
    suff : ~(y2 ≤ y < y2 + Z.of_N c2). {
      tauto.
    }
    move => [H_y2_le H_y_lt].
    move : H_y_in => []. {
      move => [H_y1_le] H_y_lt'.
      omega.
    } {
      eapply Nin_elements_greater; eauto.
```

```
          apply Z.le_trans with (m := y1); last first. {
            apply Z_le_add_r.
          }
          apply Z.lt_le_incl.
          apply Z.lt_trans with (m := y2 + Z.of_N c2) => //.
        }
      }
    } {
      move => H_y1_eq.
      apply subset_props_aux.
      ∃ y1.
      rewrite !InZ_cons.
      split. {
        left.
        by apply In_elementsZ_single_hd.
      } {
        rewrite !In_elementsZ_single H_y1_eq.
        move => []. {
          move => [_] /Z.lt_irrefl //.
        } {
          eapply Nin_elements_greater; eauto.
          rewrite H_y1_eq.
          apply Z.le_refl.
        }
      }
    }
  }
}
Qed.

Lemma subset_spec :
 ∀ (s s' : t) (Hs : Ok s) (Hs' : Ok s'),
 subset s s' = true ↔ Subset s s'.
Proof.
  intros s s' Hs Hs'.
  move : (Hs) (Hs').
  rewrite /Ok /IsOk.
  move => [H_inv_s H_enc_s] [H_inv_s' H_enc_s'].
  rewrite (subset_props s s' H_inv_s H_inv_s').
  rewrite /Subset.
  split. {
    move => H_pre enc_y.
    rewrite !In_InZ.
```

```
    apply H_pre.
  } {
    move => H_pre y H_y_in.
    move : (H_enc_s _ H_y_in) => [e] H_e. subst.
    move : (H_pre e) H_y_in.
    rewrite !In_InZ //.
  }
Qed.
```

elements and elementsZ specification

```
Lemma elements_spec1 : ∀ (s : t) (x : elt) (Hs : Ok s), List.In x (elements s) ↔ In x s.
Proof.
  intros s x Hs.
  by rewrite In_alt_def.
Qed.

Lemma NoDupA_elementsZ_single: ∀ c x,
  NoDupA Z.eq (elementsZ_single x c).
Proof.
  induction c as [| c' IH] using N.peano_ind. {
    intros x.
    rewrite elementsZ_single_base //.
  } {
    intros x.
    rewrite elementsZ_single_succ.
    apply NoDupA_app. {
      apply Z.eq_equiv.
    } {
      apply IH.
    } {
      apply NoDupA_singleton.
    } {
      move => y.
      rewrite !InA_alt.
      move => [_] [<-] H_y_in.
      move => [_] [<-] H_y_in'.
      move : H_y_in H_y_in'.
      rewrite In_elementsZ_single /=.
      move => [H_x_le] H_y_lt [] // H_y_eq.
      contradict H_y_lt.
      rewrite H_y_eq.
      apply Z.lt_irrefl.
```

```
    }
  }
Qed.
```

Lemma elementsZ_spec2w : ∀ $(s : t)$ $(Hs :$ **Ok** $s)$, **NoDupA** Z.eq (elementsZ s).

```
Proof.
  induction s as [|[x c] s' IH]. {
    move => _.
    rewrite elementsZ_nil.
    apply NoDupA_nil.
  } {
    move => H_ok_s.
    move : (H_ok_s) => /Ok_cons [H_interval_list_elements_greater] [H_c] [H_enc] H_s'.
    rewrite elementsZ_cons.
    apply NoDupA_app. {
      apply Z.eq_equiv.
    } {
      by apply IH.
    } {
      apply NoDupA_rev. {
        apply Z.eq_equiv.
      } {
        apply NoDupA_elementsZ_single.
      }
    } {
      move => y.
      rewrite !InA_alt.
      move => [_] [<-] H_y_in.
      move => [_] [<-] H_y_in'.
      move : H_y_in'.
      rewrite -in_rev In_elementsZ_single /=.
      move => [H_x_le] H_y_lt.
      eapply (Nin_elements_greater s' (x + Z.of_N c)) => //. {
        apply H_s'.
      } {
        apply Z.lt_le_incl, H_y_lt.
      } {
        apply H_y_in.
      }
    }
  }
Qed.
```

Lemma elements_spec2w : ∀ $(s : t)$ $(Hs :$ **Ok** $s)$, **NoDupA** *Enc.E.eq* (elements s).

```
Proof.
  intros s Hs.
  rewrite /elements rev_map_alt_def.
  apply NoDupA_rev. {
    apply Enc.E.eq_equiv.
  } {
    eapply NoDupA_map; first by apply elementsZ_spec2w.

    intros x1 x2 H_x1_in H_x2_in H_dec_eq.
    have H_is_enc: is_encoded_elems_list (elementsZ s). {
      apply Hs.
    }
    move : (H_is_enc _ H_x1_in) ⇒ [y1 H_x1_eq].
    move : (H_is_enc _ H_x2_in) ⇒ [y2 H_x2_eq].
    move : H_dec_eq.
    rewrite -H_x1_eq -H_x2_eq !Enc.decode_encode_ok Enc.encode_eq //.
  }
Qed.
```

equal specification

```
Lemma equal_alt_def : ∀ s1 s2,
  equal s1 s2 = true ↔ (s1 = s2).
Proof.
  induction s1 as [|| [x cx] xs IH]. {
    move ⇒ [] //.
  } {
    move ⇒ [] //=.
    move ⇒ [y cy] ys.
    rewrite !andb_true_iff IH N.eqb_eq Z.eqb_eq.
    split. {
      move ⇒ [->] [->] → //.
    } {
      move ⇒ [->] → → //.
    }
  }
Qed.

Lemma equal_elementsZ :
  ∀ (s s' : t) {Hs : Ok s} {Hs' : Ok s'},
  (∀ x, (InZ x s ↔ InZ x s')) → (s = s').
Proof.
  intros s s'.
  move ⇒ H_ok_s H_ok_s' H_InZ_eq.
```

```
  have [] : ((subset s s' = true) ∧ (subset s' s = true)). {
    rewrite !subset_spec /Subset.
    split ⇒ x; rewrite !In_InZ H_InZ_eq //.
  }
  have : interval_list_invariant s' = true by apply H_ok_s'.
  have : interval_list_invariant s = true by apply H_ok_s.
  clear H_ok_s H_ok_s' H_InZ_eq.
  move : s s'.
  induction s as [|[x1 c1] s1 IH];
    case s' as [|[x2 c2] s2] ⇒ //.
  rewrite !interval_list_invariant_cons.
  move ⇒ [H_gr_s1] [H_c1_neq_0] H_inv_s1.
  move ⇒ [H_gr_s2] [H_c2_neq_0] H_inv_s2.
  rewrite subset_flatten_alt_def
            (subset_flatten_alt_def ((x2, c2)::s2)).
  rewrite (interval_compare_swap x1 c1); last by left.
  move : (interval_compare_elim x1 c1 x2 c2).
  case (interval_compare (x1, c1) (x2, c2)) ⇒ //.
  move ⇒ [->] → H_sub_s12 H_sub_s21.
  suff → : s1 = s2 by done.
  by apply IH.
Qed.

Lemma equal_spec :
  ∀ (s s' : t) {Hs : Ok s} {Hs' : Ok s'},
  equal s s' = true ↔ Equal s s'.
Proof.
  intros s s' Hs Hs'.
  rewrite equal_alt_def /Equal.
  split. {
    move ⇒ → //.
  } {
    move ⇒ H.
    apply equal_elementsZ ⇒ // x.
    move : (H (Enc.decode x)).
    rewrite !In_InZ.
    suff H_ex : (∀ s'', Ok s'' → InZ x s'' → (∃ z, Enc.encode z = x)). {
      move ⇒ HH.
      split. {
        move ⇒ H3.
        move : HH (H3).
        move : (H_ex s Hs H3) ⇒ [z] ←.
        rewrite Enc.decode_encode_ok ⇒ ← //.
```

```
      } {
        move => H3.
        move : HH (H3).
        move : (H_ex s' Hs' H3) => [z] <-.
        rewrite Enc.decode_encode_ok => <- //.
      }
    }
    clear.
    intros s'' H_ok H_in_x.
    have H_enc : is_encoded_elems_list (elementsZ s''). {
      apply H_ok.
    }
    apply H_enc.
    apply H_in_x.
  }
Qed.
```

compare

```
Definition lt (s1 s2 : t) : Prop := (compare s1 s2 = Lt).

Lemma compare_eq_Eq : ∀ s1 s2,
  (compare s1 s2 = Eq ↔ equal s1 s2 = true).
Proof.
  induction s1 as [|| [y1 c1] s1' IH];
    case s2 as [|| [y2 c2] s2'] => //.
  rewrite /= !andb_true_iff -IH Z.eqb_eq N.eqb_eq.
  move : (Z.compare_eq_iff y1 y2).
  case (Z.compare y1 y2). {
    move => H.
    have -> : y1 = y2. by apply H.
    clear H.

    move : (N.compare_eq_iff c1 c2).
    case (N.compare c1 c2). {
      move => H.
      have -> : c1 = c2. by apply H.
      tauto.
    } {
      move => H.
      have H_neq : ~(c1 = c2). by rewrite -H => {H}.
      tauto.
    } {
      move => H.
```

```
      have H_neq : ~(c1 = c2). by rewrite -H => {H}.
      tauto.
    }
  } {
    move => H.
    have H_neq : ~(y1 = y2). by rewrite -H => {H}.
    tauto.
  } {
    move => H.
    have H_neq : ~(y1 = y2). by rewrite -H => {H}.
    tauto.
  }
Qed.

Lemma compare_eq_Lt_nil_l : ∀ s,
  compare nil s = Lt ↔ s ≠ nil.
Proof.
  intros s.
  case s => //=.
  split => //.
Qed.

Lemma compare_eq_Lt_nil_r : ∀ s,
  ~(compare s nil = Lt).
Proof.
  intros s.
  case s as [| [y1 c1] s'] => //=.
Qed.

Lemma compare_eq_Lt_cons : ∀ y1 y2 c1 c2 s1 s2,
  compare ((y1, c1)::s1) ((y2, c2)::s2) = Lt ↔
  (y1 < y2) ∨ ((y1 = y2) ∧ (c1 < c2)%N) ∨
  ((y1 = y2) ∧ (c1 = c2) ∧ compare s1 s2 = Lt).
Proof.
  intros y1 y2 c1 c2 s1 s2.
  rewrite /=.
  case_eq (Z.compare y1 y2). {
    move => /Z.compare_eq_iff →.
    case_eq (N.compare c1 c2). {
      move => /N.compare_eq_iff →.
      split. {
        move => H.
        right; right.
        done.
      } {
```

```
      move => [| []]. {
        move => /Z.lt_irrefl //.
      } {
        move => [_] /N.lt_irrefl //.
      } {
        move => [_] [_] -> //.
      }
    }
  } {
    move => /N.compare_lt_iff H_c1_lt.
    split => //.
    move => _.
    right; left. done.
  } {
    move => /N.compare_gt_iff H_c2_lt.
    split => //.
    move => [| []]. {
      move => /Z.lt_irrefl //.
    } {
      move => [_] /N.lt_asymm //.
    } {
      move => [_] [] H_c1_eq.
      contradict H_c2_lt.
      subst c1.
      by apply N.lt_irrefl.
    }
  }
} {
  move => /Z.compare_lt_iff.
  tauto.
} {
  move => /Z.compare_gt_iff H_y2_lt.
  split => //.
  move => [| []]. {
    move => /Z.lt_asymm //.
  } {
    move => [] H_y1_eq.
    exfalso. omega.
  } {
    move => [] H_y1_eq.
    exfalso. omega.
  }
```

```
  }
Qed.

Lemma compare_antisym: ∀ (s1 s2 : t),
  (compare s1 s2) = CompOpp (compare s2 s1).
Proof.
  induction s1 as [| [y1 c1] s1' IH];
    case s2 as [| [y2 c2] s2'] => //.
  rewrite /= (Z.compare_antisym y1 y2) (N.compare_antisym c1 c2).
  case (Z.compare y1 y2) => //=.
  case (N.compare c1 c2) => //=.
Qed.

Lemma compare_spec : ∀ s1 s2,
  CompSpec eq lt s1 s2 (compare s1 s2).
Proof.
  intros s1 s2.
  rewrite /CompSpec /lt (compare_antisym s2 s1).
  case_eq (compare s1 s2). {
    rewrite compare_eq_Eq equal_alt_def => ->.
    by apply CompEq.
  } {
    move => _.
    by apply CompLt.
  } {
    move => _.
    by apply CompGt.
  }
Qed.

Lemma lt_Irreflexive : Irreflexive lt.
Proof.
  rewrite /Irreflexive /Reflexive /complement /lt.
  intros x.
  suff -> : compare x x = Eq by done.
  rewrite compare_eq_Eq equal_alt_def //.
Qed.

Lemma lt_Transitive : Transitive lt.
Proof.
  rewrite /Transitive /lt.
  induction x as [| [y1 c1] s1' IH];
    case y as [| [y2 c2] s2'];
    case z as [| [y3 c3] s3'] => //.

  rewrite !compare_eq_Lt_cons.
```

```
  move => [H_y1_lt | [[->] H_c1_lt | [->] [->] H_comp]]
 [H_y2_lt | [[<-] H_c2_lt | [<-] [<-] H_comp']]. {
    left.
    by apply Z.lt_trans with (m := y2).
  } {
    by left.
  } {
    by left.
  } {
    by left.
  } {
    right; left.
    split => //.
    by apply N.lt_trans with (m := c2).
  } {
    by right; left.
  } {
    by left.
  } {
    by right; left.
  } {
    right; right.
    split => //.
    split => //.
    by apply (IH s2').
  }
Qed.
```

elements is sorted

```
Lemma elementsZ_single_sorted : ∀ c x,
  sort Z.lt (elementsZ_single x c).
Proof.
  induction c as [| c' IH] using N.peano_ind. {
    intro x.
    rewrite elementsZ_single_base.
    apply Sorted_nil.
  } {
    intro x.
    rewrite elementsZ_single_succ_front.
    apply Sorted_cons. {
      apply IH.
    } {
```

```
      case (N.zero_or_succ c'). {
        move => ->.
        rewrite elementsZ_single_base //.
      } {
        move => [c''] ->.
        rewrite elementsZ_single_succ_front.
        constructor.
        apply Z.lt_succ_diag_r.
      }
    }
  }
Qed.

Lemma elementsZ_sorted : ∀ s,
  interval_list_invariant s = true ->
  sort Z.lt (rev (elementsZ s)).
Proof.
  induction s as [| [y c] s' IH]. {
    move => _.
    rewrite elementsZ_nil.
    apply Sorted_nil.
  } {
    rewrite interval_list_invariant_cons elementsZ_cons
            rev_app_distr rev_involutive.
    move => [H_gr] [H_c_neq_0] H_inv_s'.
    apply SortA_app with (eqA := Logic.eq). {
      apply eq_equivalence.
    } {
      apply Z.lt_strorder.
    } {
      apply elementsZ_single_sorted.
    } {
      by apply IH.
    } {
      intros x1 x2.
      move => /InA_alt [_] [<-] /In_elementsZ_single [_ H_x1_lt].
      move => /InA_alt [_] [<-].
      rewrite -In_rev => H_x2_in.

      apply Z.lt_trans with (m := (y + Z.of_N c)) => //.
      eapply interval_list_elements_greater_alt2_def;
        eauto.
    }
  }
```

```
Qed.

Lemma elements_sorted : ∀ s,
  Ok s →
  sort Enc.E.lt (elements s).
Proof.
  move ⇒ s [H_inv] H_enc.
  rewrite /elements rev_map_alt_def -map_rev.
  have : (∀ x : Z, List.In x (rev (elementsZ s)) →
           ∃ e : Enc.E.t, Enc.encode e = x). {
    move ⇒ x.
    move : (H_enc x).
    rewrite In_rev //.
  }
  move : (elementsZ_sorted s H_inv) ⇒ {H_enc}.
  generalize (rev (elementsZ s)).
  induction l as [| x xs IH]. {
    rewrite /= ⇒ _ _.
    apply Sorted_nil.
  } {
    move ⇒ H_sort H_enc.
    apply Sorted_inv in H_sort as [H_sort H_hd_rel].
    simpl.
    apply Sorted_cons. {
      apply IH ⇒ //.
      move ⇒ xx H_xx_in.
      apply H_enc.
      by apply in_cons.
    } {
      move : H_hd_rel H_enc.
      case xs ⇒ //=.
      move ⇒ x' xs' H_hd_rel H_enc.
      apply HdRel_inv in H_hd_rel.
      apply HdRel_cons.
      rewrite -Enc.encode_lt.
      have [y H_y] : (∃ y, Enc.encode y = x). {
        apply H_enc. by left.
      }
      have [y' H_y'] : (∃ y', Enc.encode y' = x'). {
        apply H_enc. by right; left.
      }
      move : H_hd_rel.
      rewrite -!H_y -!H_y' !Enc.decode_encode_ok //.
```

```
    }
  }
Qed.
```

choose specification

```
Definition min_eltZ_spec1 :
  ∀ (s : t) (x : Z),
    interval_list_invariant s = true →
    min_eltZ s = Some x → InZ x s.
Proof.
  intros s x.
  case s as [| [x' c] s']. {
    rewrite /min_eltZ //.
  } {
    rewrite /min_eltZ InZ_cons interval_list_invariant_cons.
    move ⇒ [_] [H_c_neq] _ [->].
    left.
    by apply In_elementsZ_single_hd.
  }
Qed.

Lemma min_eltZ_spec2 :
  ∀ (s : t) (x y : Z) (Hs : Ok s),
  min_eltZ s = Some x → InZ y s → ¬ Z.lt y x.
Proof.
  intros s x y H_ok H_min H_in H_y_lt_x.
  eapply (Nin_elements_greater s (Z.pred x)) ⇒ //; last apply H_in. {
    move : H_ok H_min.
    case s ⇒ //.
    move ⇒ [z c] s' _ [<-].
    rewrite interval_list_elements_greater_cons.
    apply Z.lt_pred_l.
  } {
    apply H_ok.
  } {
    by apply Z.lt_le_pred.
  }
Qed.

Definition min_eltZ_spec3 :
  ∀ (s : t),
    min_eltZ s = None → ∀ x, ¬InZ x s.
Proof.
```

```
  intros s.
  case s as [| [x' c] s'];
    rewrite /min_eltZ //.
  move => _ x //.
Qed.

Definition min_elt_spec1 :
  ∀ (s : t) (x : elt) (Hs : Ok s), min_elt s = Some x → In x s.
Proof.
  rewrite /min_elt.
  move => s x H_ok.
  case_eq (min_eltZ s) => //.
  move => z H_min_elt [<-].
  apply InZ_In => //.
  apply min_eltZ_spec1 => //.
  apply H_ok.
Qed.

Definition min_elt_spec2 :
  ∀ (s : t) (x y : elt) (Hs : Ok s), min_elt s = Some x → In y s → ~(Enc.E.lt y x).
Proof.
  rewrite /min_elt.
  move => s x y H_ok.
  case_eq (min_eltZ s) => //.
  move => z H_min_elt [<-].
  rewrite In_InZ => H_inZ.
  have H_y_eq : y = Enc.decode (Enc.encode y). {
    by rewrite Enc.decode_encode_ok.
  }
  rewrite H_y_eq -Enc.encode_lt.
  apply (min_eltZ_spec2 _ _ _ H_ok); last first. {
    by rewrite Enc.decode_encode_ok.
  }
  suff → : Enc.encode (Enc.decode z) = z by assumption.
  apply encode_decode_eq with (s := s) => //.
  apply min_eltZ_spec1 => //.
  apply H_ok.
Qed.

Definition min_elt_spec3 :
  ∀ s : t, min_elt s = None → Empty s.
Proof.
  rewrite /min_elt /min_eltZ /Empty /In.
  case s as [| [x' c] s'] => //.
  move => _ e.
```

```
  rewrite elements_nil InA_nil //.
Qed.

Definition choose_spec1 :
  ∀ (s : t) (x : elt) (Hs : Ok s), choose s = Some x → In x s.
Proof.
  rewrite /choose.
  apply min_elt_spec1.
Qed.

Definition choose_spec2 :
  ∀ s : t, choose s = None → Empty s.
Proof.
  rewrite /choose.
  apply min_elt_spec3.
Qed.

Lemma choose_spec3: ∀ s s' x x', Ok s → Ok s' →
  choose s = Some x → choose s' = Some x' → Equal s s' → x = x'.
Proof.
  intros s s' x x' Hs Hs' Hx Hx'.
  rewrite -equal_spec equal_alt_def ⇒ H_s_eq.
  move : Hx Hx'.
  rewrite H_s_eq ⇒ → [] //.
Qed.

Definition max_eltZ_spec1 :
  ∀ (s : t) (x : Z),
    interval_list_invariant s = true →
    max_eltZ s = Some x → InZ x s.
Proof.
  intros s x.
  induction s as [|[x' c] s' IH]. {
    rewrite /max_eltZ //.
  } {
    rewrite InZ_cons interval_list_invariant_cons /=.
    move ⇒ [_] [H_c_neq].
    case s' as [|[y' c'] s'']. {
      move ⇒ _ [<-].
      left. {
        rewrite In_elementsZ_single.
        split. {
          rewrite -Z.lt_le_pred.
          by apply Z_lt_add_r.
        } {
```

```
          apply Z.lt_pred_l.
        }
      }
    } {
      move => H_inv H_max_eq.
      right.
      by apply IH.
    }
  }
Qed.

Lemma max_eltZ_spec2 :
  ∀ (s : t) (x y : Z),
  interval_list_invariant s = true →
  max_eltZ s = Some x → InZ y s → ¬ Z.lt x y.
Proof.
  induction s as [| [y c] s' IH]. {
    done.
  } {
    move => x x'.
    rewrite interval_list_invariant_cons.
    move => [H_gr] [H_c_neq_0] H_inv_s'.
    have H_gr' : (∀ xx : Z, InZ xx (s') → y + Z.of_N c < xx). {
      apply interval_list_elements_greater_alt2_def => //.
    }

    case s' as [| [y' c'] s'']. {
      move => [<-].
      rewrite InZ_cons InZ_nil In_elementsZ_single.
      omega.
    } {
      move => H_max_eq.
      rewrite InZ_cons.
      move => []; last by apply IH.
      rewrite In_elementsZ_single.
      move => [_] H_x'_lt H_lt_x'.
      have H_x_in : InZ x ((y', c')::s''). {
        by apply max_eltZ_spec1.
      }

      move : (H_gr' _ H_x_in).
      apply Z.nlt_ge, Z.lt_le_incl.
      by apply Z.lt_trans with (m := x').
```

```
    }
  }
Qed.

Lemma max_eltZ_eq_None :
  ∀ (s : t),
    max_eltZ s = None → s = nil.
Proof.
  induction s as [| [x' c] s' IH] ⇒ //.
  move : IH.
  case s' as [| [y' c'] s''] ⇒ //=.
  move ⇒ H H_pre.
  move : (H H_pre) ⇒ //.
Qed.

Definition max_eltZ_spec3 :
  ∀ (s : t),
    max_eltZ s = None → ∀ x, ¬InZ x s.
Proof.
  move ⇒ s /max_eltZ_eq_None → x /InZ_nil //.
Qed.

Definition max_elt_spec1 :
  ∀ (s : t) (x : elt) (Hs : Ok s), max_elt s = Some x → In x s.
Proof.
  rewrite /max_elt.
  move ⇒ s x H_ok.
  case_eq (max_eltZ s) ⇒ //.
  move ⇒ z H_max_elt [<-].
  apply InZ_In ⇒ //.
  apply max_eltZ_spec1 ⇒ //.
  apply H_ok.
Qed.

Definition max_elt_spec2 :
  ∀ (s : t) (x y : elt) (Hs : Ok s), max_elt s = Some x → In y s → ~(Enc.E.lt x y).
Proof.
  rewrite /max_elt.
  move ⇒ s x y H_ok.
  move : (H_ok) ⇒ [H_inv] _.
  case_eq (max_eltZ s) ⇒ //.
  move ⇒ z H_max_elt [<-].
  rewrite In_InZ ⇒ H_inZ.
  rewrite -Enc.encode_lt.
  apply (max_eltZ_spec2 _ _ _ H_inv) ⇒ //.
```

```
    suff -> : Enc.encode (Enc.decode z) = z => //.
    apply encode_decode_eq with (s := s) => //.
    apply max_eltZ_spec1 => //.
Qed.

Definition max_elt_spec3 :
    ∀ s : t, max_elt s = None -> Empty s.
Proof.
    intro s.
    rewrite /max_elt /Empty.
    case_eq (max_eltZ s) => //.
    move => /max_eltZ_eq_None -> _ x.
    rewrite /In elements_nil InA_nil //.
Qed.
```

fold specification

```
Lemma fold_spec :
  ∀ (s : t) (A : Type) (i : A) (f : elt -> A -> A),
  fold f s i = fold_left (flip f) (elements s) i.
Proof.
    intros s A i f.
    rewrite /fold fold_elementsZ_alt_def /elements
            rev_map_alt_def -map_rev.
    move : i.
    generalize (rev (elementsZ s)).
    induction l as [| x xs IH]. {
      done.
    } {
      move => i.
      rewrite /= IH //.
    }
Qed.
```

cardinal specification

```
Lemma cardinalN_spec : ∀ (s : t) (c : N),
    cardinalN c s = (c + N.of_nat (length (elements s)))%N.
Proof.
    induction s as [| [x cx] xs IH]. {
      intros c.
      rewrite elements_nil /= N.add_0_r //.
    } {
```

```
    intros c.
    rewrite /= IH.
    rewrite /elements !rev_map_alt_def !rev_length !map_length.
    rewrite elementsZ_cons app_length Nat2N.inj_add rev_length.
    rewrite length_elementsZ_single N2Nat.id.
    ring.
  }
Qed.
```

```
Lemma cardinal_spec :
 ∀ (s : t),
 cardinal s = length (elements s).
Proof.
  intros s.
  rewrite /cardinal cardinalN_spec N.add_0_l Nat2N.id //.
Qed.
```

for_all specification

```
Lemma for_all_spec :
 ∀ (s : t) (f : elt → bool) (Hs : Ok s),
 Proper (Enc.E.eq==>eq) f →
 (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
Proof.
  intros s f Hs H.
  rewrite /for_all /For_all /In fold_elementsZ_alt_def
          /elements rev_map_alt_def -map_rev.
  generalize (rev (elementsZ s)).
  induction l as [|x xs IH]. {
    split ⇒ // _ x /= /InA_nil //.
  } {
    rewrite /=.
    case_eq (f (Enc.decode x)) ⇒ H_f_eq. {
      rewrite IH.
      split. {
        move ⇒ HH x' /InA_cons []. {
          by move ⇒ →.
        } {
          apply HH.
        }
      } {
        move ⇒ HH x' H_in.
        apply HH.
```

```
          apply InA_cons.
          by right.
        }
      } {
        split; move ⇒ HH. {
          contradict HH.
          case xs ⇒ //.
        } {
          exfalso.
          have H_in: (InA Enc.E.eq (Enc.decode x) (Enc.decode x :: map Enc.decode xs)).
{
            apply InA_cons.
            left.
            apply Enc.E.eq_equiv.
          }
          move : (HH _ H_in).
          rewrite H_f_eq ⇒ //.
        }
      }
    }
  Qed.
```

exists specification

```
  Lemma exists_spec :
   ∀ (s : t) (f : elt → bool) (Hs : Ok s),
   Proper (Enc.E.eq==>eq) f →
   (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
  Proof.
    intros s f Hs H.
    rewrite /exists_ /Exists /In fold_elementsZ_alt_def
            /elements rev_map_alt_def -map_rev.
    generalize (rev (elementsZ s)).
    induction l as [| x xs IH]. {
      split ⇒ //.
      move ⇒ [x] /= [] /InA_nil //.
    } {
      rewrite /=.
      case_eq (f (Enc.decode x)) ⇒ H_f_eq. {
        split ⇒ _. {
          ∃ (Enc.decode x).
          split ⇒ //.
          apply InA_cons.
```

```
        left.
        apply Enc.E.eq_equiv.
      } {
        case xs ⇒ //.
      }
    } {
      rewrite IH.
      split. {
        move ⇒ [x0] [H_in] H_f_x0.
        ∃ x0.
        split ⇒ //.
        apply InA_cons.
        by right.
      } {
        move ⇒ [x0] [] /InA_cons H_in H_f_x0.
        ∃ x0.
        split ⇒ //.
        move : H_in ⇒ [] // H_in.
        contradict H_f_x0.
        rewrite H_in H_f_eq //.
      }
    }
  }
Qed.
```

filter specification

```
Definition partitionZ_aux_invariant (x : Z) acc c :=
  interval_list_invariant (List.rev (partitionZ_fold_skip acc c)) = true ∧
  match c with
    None ⇒ (∀ y', InZ y' acc → Z.succ y' < x)
  | Some (y, c') ⇒ (x = y + Z.of_N c')
  end.

Lemma partitionZ_aux_invariant_insert : ∀ x acc c,
  partitionZ_aux_invariant x acc c →
  partitionZ_aux_invariant (Z.succ x) acc
    (Some (partitionZ_fold_insert c x)).
Proof.
  intros x acc c.
  rewrite /partitionZ_fold_insert /partitionZ_aux_invariant
          /partitionZ_fold_skip.
  case c; last first. {
```

```
    move => [H_inv] H_in.
    rewrite /= interval_list_invariant_app_iff Z.add_1_r.
    split; last done.
    split; first done.
    split; first done.
    move => x1 x2.
    rewrite InZ_rev InZ_cons InZ_nil In_elementsZ_single1.
    move => H_x1_in [] // <-.
    by apply H_in.
  } {
    move => [y c'].
    rewrite /= !interval_list_invariant_app_iff
      N2Z.inj_succ Z.add_succ_r .
    rewrite !interval_list_invariant_cons !interval_list_invariant_nil.
    move => [] [H_inv_acc] [] [] _ [H_c_neq_0] _
      H_in_c ->.
    split; last done.
    split; first done.
    split. {
      split; first done.
      split; last done.
      apply N.neq_succ_0.
    } {
      move => x1 x2.
      rewrite InZ_cons InZ_nil In_elementsZ_single.
      move => H_x1_in [] // [H_y_le] H_x2_lt.
      apply Z.lt_le_trans with (m := y) => //.
      apply H_in_c => //.
      rewrite InZ_cons In_elementsZ_single.
      left.
      split. {
        apply Z.le_refl.
      } {
        by apply Z_lt_add_r.
      }
    }
  }
Qed.

Lemma partitionZ_aux_invariant_skip : ∀ x acc c,
  partitionZ_aux_invariant x acc c ->
  partitionZ_aux_invariant (Z.succ x) (partitionZ_fold_skip acc c) None.
Proof.
```

```
  intros x acc c.
  rewrite /partitionZ_fold_skip /partitionZ_aux_invariant
          /partitionZ_fold_skip.
  case c; last first. {
    move => [H_inv] H_in.
    split; first done.
    move => y' H_y'_in.
    apply Z.lt_trans with (m := x). {
      by apply H_in.
    } {
      apply Z.lt_succ_diag_r.
    }
  } {
    move => [y c'] [H_inv] ->.
    split => //.
    move => y'.
    rewrite InZ_cons In_elementsZ_single.
    move => []. {
      move => [_].
      rewrite -Z.succ_lt_mono //.
    } {
      move : H_inv.
      rewrite /= !interval_list_invariant_app_iff interval_list_invariant_cons.
      move => [_] [] [_] [H_c'_neq] _ H_pre H_y'_in.
      apply Z.lt_trans with (m := y). {
        apply H_pre. {
          by rewrite InZ_rev.
        } {
          rewrite InZ_cons.
          left.
          by apply In_elementsZ_single_hd.
        }
      }
      apply Z.lt_succ_r, Z_le_add_r.
    }
  }
Qed.

Definition partitionZ_fold_current (c : option (Z × N)) :=
  match c with
      None => nil
    | Some yc => yc::nil
  end.
```

```
Lemma InZ_partitionZ_fold_current_Some : ∀ yc y,
    InZ y (partitionZ_fold_current (Some yc)) ↔
    InZ y (yc :: nil).
Proof. done. Qed.

Lemma InZ_partitionZ_fold_insert : ∀ c x y l,
  match c with
  | Some (y, c') ⇒ x = y + Z.of_N c'
  | None ⇒ True
  end → (
  InZ y (partitionZ_fold_insert c x :: l) ↔
   ((x = y) ∨ InZ y (partitionZ_fold_current c) ∨
      InZ y l)).
Proof.
  intros c x y l.
  rewrite /partitionZ_fold_insert /partitionZ_fold_current
          /partitionZ_fold_skip.
  case c. {
    move ⇒ [y' c'] →.
    rewrite !InZ_cons elementsZ_single_succ in_app_iff
            InZ_nil /=.
    tauto.
  } {
    rewrite InZ_cons InZ_nil In_elementsZ_single1.
    tauto.
  }
Qed.

Lemma InZ_partitionZ_fold_skip : ∀ c acc y,
  InZ y (partitionZ_fold_skip acc c) ↔
  (InZ y (partitionZ_fold_current c) ∨ InZ y acc).
Proof.
  intros c acc y.
  rewrite /partitionZ_fold_skip /partitionZ_fold_current
          /partitionZ_fold_skip.
  case c. {
    move ⇒ [y' c'].
    rewrite !InZ_cons InZ_nil /=.
    tauto.
  } {
    rewrite InZ_nil.
    tauto.
  }
Qed.
```

```
Lemma filterZ_single_aux_props :
  ∀ f c x acc cur,
    partitionZ_aux_invariant x acc cur →
    match (filterZ_single_aux f (acc, cur) x c) with
      (acc', c') ⇒
      let r := partitionZ_fold_skip acc' c' in
      interval_list_invariant (List.rev r) = true ∧
      (∀ y', InZ y' r ↔ (InZ y' (partitionZ_fold_skip acc cur) ∨
                                         (f y' = true ∧ List.In y' (elementsZ_single x c))))

    end.
Proof.
  intro f.
  induction c as [| c' IH] using N.peano_ind. {
    intros x acc cur.
    rewrite /partitionZ_aux_invariant.
    move ⇒ [H_inv] _.
    rewrite /filterZ_single_aux fold_elementsZ_single_zero /=.
    tauto.
  }
  intros x acc cur H_inv.
  have → : filterZ_single_aux f (acc, cur) x (N.succ c') =
             filterZ_single_aux f (filterZ_fold_fun f (acc, cur) x) (Z.succ x) c'. {
      by rewrite /filterZ_single_aux fold_elementsZ_single_succ.
  }
  case_eq (filterZ_fold_fun f (acc, cur) x).
  move ⇒ acc' cur' H_fold_eq.

  case_eq (filterZ_single_aux f (acc', cur') (Z.succ x) c').
  move ⇒ acc'' cur'' H_succ_eq.

  have H_inv' : partitionZ_aux_invariant (Z.succ x) acc' cur'. {
    move : H_fold_eq H_inv.
    rewrite /filterZ_fold_fun.
    case (f x); move ⇒ [<-] ←. {
      apply partitionZ_aux_invariant_insert.
    } {
      apply partitionZ_aux_invariant_skip.
    }
  }

  move : (IH (Z.succ x) acc' cur' H_inv') ⇒ {IH}.
  rewrite H_succ_eq /=.
  set r := partitionZ_fold_skip acc'' cur''.
```

```
    move ⇒ [H_inv_r] H_in_r.
    split; first assumption.
    move ⇒ y'.
    move : H_fold_eq.
    rewrite H_in_r /filterZ_fold_fun.
    case_eq (f x) ⇒ H_fx [<-] ←. {
      rewrite InZ_partitionZ_fold_skip InZ_partitionZ_fold_current_Some InZ_partitionZ_fold_skip
elementsZ_single_succ_front.
      rewrite InZ_partitionZ_fold_insert; last first. {
        move : H_inv.
        rewrite /partitionZ_aux_invariant ⇒ [[_]].
        case cur ⇒ //.
      }
      rewrite InZ_nil /=.
      split; last by tauto.
      move ⇒ []; last by tauto.
      move ⇒ []; last by tauto.
      move ⇒ []. {
        move ⇒ ←.
        tauto.
      } {
        tauto.
      }
    } {
      rewrite InZ_partitionZ_fold_skip /partitionZ_fold_current InZ_partitionZ_fold_skip ele-
mentsZ_single_succ_front !InZ_nil /=.
      split; first by tauto.
      move ⇒ []; first by tauto.
      move ⇒ [] H_fy' []. {
        move ⇒ H_x_eq; subst y'.
        contradict H_fy'.
        by rewrite H_fx.
      } {
        tauto.
      }
    }
  Qed.

  Lemma filterZ_single_props :
    ∀ f c x acc,
      interval_list_invariant (rev acc) = true →
      (∀ y' : Z, InZ y' acc → Z.succ y' < x) →
      match (filterZ_single f acc x c) with
```

```
      r ⇒
      interval_list_invariant (List.rev r) = true ∧
      (∀ y', InZ y' r ↔ (InZ y' acc ∨
                              (f y' = true ∧ List.In y' (elementsZ_single x c))))

    end.
Proof.
  intros f c x acc.
  move ⇒ H_inv H_acc.
  rewrite /filterZ_single.
  have H_inv' : partitionZ_aux_invariant x acc None. {
    by rewrite /partitionZ_aux_invariant /=.
  }
  move : (filterZ_single_aux_props f c x acc None H_inv').
  case_eq (filterZ_single_aux f (acc, None) x c).
  move ⇒ acc' cur' /= H_res.
  tauto.
Qed.

Lemma filterZ_aux_props :
  ∀ f s acc,
    interval_list_invariant s = true →
    interval_list_invariant (rev acc) = true →
    (∀ x1 x2 : Z, InZ x1 acc → InZ x2 s → Z.succ x1 < x2) →
    match (filterZ_aux acc f s) with
      r ⇒
      interval_list_invariant r = true ∧
      (∀ y', InZ y' r ↔ (InZ y' acc ∨
                              (f y' = true ∧ InZ y' s)))
    end.
Proof.
  intro f.
  induction s as [|[y c] s' IH]. {
    intros acc.
    move ⇒ _ H_inv _.
    rewrite /filterZ_aux.
    split; first assumption.
    move ⇒ y'; rewrite InZ_rev InZ_nil; tauto.
  } {
    intros acc.
    rewrite interval_list_invariant_cons.
    move ⇒ [H_gr] [H_c_neq_0] H_inv_s' H_inv H_in_acc /=.
    move : H_gr.
```

```
    rewrite interval_list_elements_greater_alt2_def => // H_gr.
    have H_pre : (∀ y' : Z, InZ y' acc → Z.succ y' < y). {
      move => x1 H_x1_in.
      apply H_in_acc => //.
      rewrite InZ_cons.
      by left; apply In_elementsZ_single_hd.
    }

    move : (filterZ_single_props f c y acc H_inv H_pre) => {H_pre}.
    set acc' := filterZ_single f acc y c.
    move => [H_inv'] H_in_acc'.
    have H_pre : (∀ x1 x2 : Z,
                    InZ x1 acc' → InZ x2 s' → Z.succ x1 < x2). {
      move => x1 x2.
      rewrite H_in_acc' In_elementsZ_single.
      move => []. {
        move => H_x1_in H_x2_in.
        apply H_in_acc => //.
        rewrite InZ_cons.
        by right.
      } {
        move => [_] [_] H_x1_lt H_x2_in.
        apply Z.le_lt_trans with (m := y + Z.of_N c).
          - by apply Z.le_succ_l.
          - by apply H_gr.
      }
    }
    move : (IH acc' H_inv_s' H_inv' H_pre) => {H_pre}.
    move => [H_inv_r] H_in_r.
    split; first assumption.
    move => y'.
    rewrite H_in_r H_in_acc' InZ_cons.
    tauto.
  }
Qed.

Lemma filterZ_props :
  ∀ f s,
    interval_list_invariant s = true →
    match (filterZ f s) with r ⇒
      interval_list_invariant r = true ∧
      (∀ y', InZ y' r ↔ (f y' = true ∧ InZ y' s))
```

```
    end.
Proof.
  intros f s H_inv_s.
  rewrite /filterZ.
  have H_pre_1 : interval_list_invariant (rev nil) = true by done.
  have H_pre_2 : (∀ x1 x2 : Z, InZ x1 nil → InZ x2 s → Z.succ x1 < x2) by done.
  move : (filterZ_aux_props f s nil H_inv_s H_pre_1 H_pre_2) ⇒ {H_pre_1} {H_pre_2}.
  move ⇒ [H_inv'] H_in_r.
  split; first assumption.
  move ⇒ y'.
  rewrite H_in_r InZ_nil.
  tauto.
Qed.
Global Instance filter_ok s f : ∀ `(Ok s), Ok (filter f s).
Proof.
  move ⇒ [H_inv H_enc].
  rewrite /filter.
  set f' := (fun z : Z ⇒ f (Enc.decode z)).
  move : (filterZ_props f' s H_inv).
  move ⇒ [H_inv'] H_in_r.
  rewrite /Ok /IsOk /is_encoded_elems_list.
  split; first assumption.
  move ⇒ x /H_in_r [_] H_x_in.
  by apply H_enc.
Qed.
Lemma filter_spec :
 ∀ (s : t) (x : elt) (f : elt → bool),
 Ok s →
 (In x (filter f s) ↔ In x s ∧ f x = true).
Proof.
  move ⇒ s x f H_ok.
  suff H_suff :
    (∀ x, (InZ x (filter f s)) ↔
                InZ x s ∧ f (Enc.decode x) = true). {
    rewrite !In_alt_def /elements !rev_map_alt_def
            -!in_rev !in_map_iff.
    setoid_rewrite H_suff.
    rewrite /InZ.
    split. {
      move ⇒ [y] [<-] [?] ?.
      split ⇒ //.
```

```
        by ∃ y.
      } {
        move ⇒ [] [y] [<-] ? ?.
        by ∃ y.
      }
    }
    rewrite /filter.
    set f' := (fun z : Z ⇒ f (Enc.decode z)).
    move : (H_ok) ⇒ [H_inv _].
    move : (filterZ_props f' s H_inv).
    move ⇒ [H_inv'] H_in_r.
    move ⇒ y; rewrite H_in_r; tauto.
  Qed.
```

partition specification

```
  Lemma partitionZ_single_aux_alt_def : ∀ f  c  y  acc_t  c_t  acc_f  c_f,
    partitionZ_single_aux f ((acc_t, c_t), (acc_f, c_f)) y c =
    (filterZ_single_aux f (acc_t, c_t) y c,
     filterZ_single_aux (fun x : Z ⇒ negb (f x)) (acc_f, c_f) y c).
  Proof.
    intros f.
    rewrite /partitionZ_single_aux /filterZ_single_aux.
    induction c as [| c' IH] using N.peano_ind. {
      intros y acc_t c_t acc_f c_f.
      rewrite !fold_elementsZ_single_zero //. } {
      intros y acc_t c_t acc_f c_f.
      rewrite !fold_elementsZ_single_succ.
      case_eq (partitionZ_fold_fun f (acc_t, c_t, (acc_f, c_f)) y) ⇒ [] [acc_t' c_t'] [acc_f'
c_f'] H_fold_eq.
      rewrite IH ⇒ {IH}.
      suff : (filterZ_fold_fun f (acc_t, c_t) y = (acc_t', c_t')) ∧
             (filterZ_fold_fun (fun x : Z ⇒ negb (f x)) (acc_f, c_f) y = (acc_f', c_f')).
{
        move ⇒ [->] → //.
      }
      move : H_fold_eq.
      rewrite /partitionZ_fold_fun /filterZ_fold_fun.
      case (f y); move ⇒ [<-] ← ← ← //.
    }
  Qed.

  Lemma partitionZ_aux_alt_def : ∀ f  s  acc_t  acc_f,
```

```
  partitionZ_aux acc_t acc_f f s =
  (filterZ_aux acc_t f s,
   filterZ_aux acc_f (fun x : Z ⇒ negb (f x)) s).
Proof.
  intros f.
  induction s as [| [y c] s' IH]. {
    done.
  } {
    intros acc_t acc_f.
    rewrite /= /partitionZ_single /filterZ_single
            partitionZ_single_aux_alt_def.
    case (filterZ_single_aux f (acc_t, None) y c) ⇒ acc_t' c_t'.
    case (filterZ_single_aux (fun x : Z ⇒ negb (f x)) (acc_f, None) y c) ⇒ acc_f' c_f'.
    rewrite IH //.
  }
Qed.

Lemma partitionZ_alt_def : ∀ f s,
  partitionZ f s = (filterZ f s,
                    filterZ (fun x ⇒ negb (f x)) s).
Proof.
  intros f s.
  rewrite /partitionZ /filterZ
          partitionZ_aux_alt_def //.
Qed.

Lemma partition_alt_def : ∀ f s,
  partition f s = (filter f s,
                   filter (fun x ⇒ negb (f x)) s).
Proof.
  intros f s.
  rewrite /partition /filter partitionZ_alt_def.
  done.
Qed.

Global Instance partition_ok1 s f : ∀ `(Ok s), Ok (fst (partition f s)).
Proof.
  move ⇒ H_ok.
  rewrite partition_alt_def /fst.
  by apply filter_ok.
Qed.

Global Instance partition_ok2 s f : ∀ `(Ok s), Ok (snd (partition f s)).
Proof.
  move ⇒ H_ok.
```

```
    rewrite partition_alt_def /snd.
    by apply filter_ok.
  Qed.

  Lemma partition_spec1 :
   ∀ (s : t) (f : elt → bool),
   Equal (fst (partition f s)) (filter f s).
  Proof.
    intros s f.
    rewrite partition_alt_def /fst /Equal //.
  Qed.

  Lemma partition_spec2 :
   ∀ (s : t) (f : elt → bool),
   Ok s →
   Equal (snd (partition f s)) (filter (fun x ⇒ negb (f x)) s).
  Proof.
    intros s f.
    rewrite partition_alt_def /snd /Equal //.
  Qed.

End Raw.
```

2.1.5 Main Module

We can now build the invariant into the set type to obtain an instantiation of module type *WSetsOn*.

```
Module MSetIntervals (Enc : ElementEncode) <: SetsOn Enc.E.
  Module E := Enc.E.
  Module Raw := Raw Enc.

 Local Unset Elimination Schemes.
 Local Unset Case Analysis Schemes.

 Definition elt := Raw.elt.
 Record t_ := Mkt {this :> Raw.t; is_ok : Raw.Ok this}.
 Definition t := t_.
 Arguments Mkt this {is_ok}.
 Hint Resolve is_ok : typeclass_instances.

 Definition In (x : elt)(s : t) := Raw.In x s.(this).
 Definition Equal (s s' : t) := ∀ a : elt, In a s ↔ In a s'.
 Definition Subset (s s' : t) := ∀ a : elt, In a s → In a s'.
 Definition Empty (s : t) := ∀ a : elt, ¬ In a s.
 Definition For_all (P : elt → Prop)(s : t) := ∀ x, In x s → P x.
 Definition Exists (P : elt → Prop)(s : t) := ∃ x, In x s ∧ P x.
```

```
Definition mem (x : elt)(s : t) := Raw.mem x s.(this).
Definition add (x : elt)(s : t) : t := Mkt (Raw.add x s.(this)).
Definition remove (x : elt)(s : t) : t := Mkt (Raw.remove x s.(this)).
Definition singleton (x : elt) : t := Mkt (Raw.singleton x).
Definition union (s s' : t) : t := Mkt (Raw.union s s').
Definition inter (s s' : t) : t := Mkt (Raw.inter s s').
Definition diff (s s' : t) : t := Mkt (Raw.diff s s').
Definition equal (s s' : t) := Raw.equal s s'.
Definition subset (s s' : t) := Raw.subset s s'.(this).
Definition empty : t := Mkt Raw.empty.
Definition is_empty (s : t) := Raw.is_empty s.
Definition elements (s : t) : list elt := Raw.elements s.
Definition min_elt (s : t) : option elt := Raw.min_elt s.
Definition max_elt (s : t) : option elt := Raw.max_elt s.
Definition choose (s : t) : option elt := Raw.choose s.
Definition compare (s1 s2 : t) : comparison := Raw.compare s1 s2.
Definition fold {A : Type}(f : elt → A → A)(s : t) : A → A := Raw.fold f s.
Definition cardinal (s : t) := Raw.cardinal s.
Definition filter (f : elt → bool)(s : t) : t := Mkt (Raw.filter f s).
Definition for_all (f : elt → bool)(s : t) := Raw.for_all f s.
Definition exists_ (f : elt → bool)(s : t) := Raw.exists_ f s.
Definition partition (f : elt → bool)(s : t) : t × t :=
  let p := Raw.partition f s in (Mkt (fst p), Mkt (snd p)).

Instance In_compat : Proper (E.eq==>eq==>iff) In.
Proof.
  repeat red.
  move ⇒ x y H_eq_xy x' y' →.
  rewrite /In /Raw.In.
  setoid_rewrite H_eq_xy.
  done.
Qed.

Definition eq : t → t → Prop := Equal.

Instance eq_equiv : Equivalence eq.
Proof. firstorder. Qed.

Definition eq_dec : ∀ (s s':t), { eq s s' }+{ ¬eq s s' }.
Proof.
 intros (s,Hs) (s',Hs').
 change ({Raw.Equal s s'}+{¬Raw.Equal s s'}).
 destruct (Raw.equal s s') eqn:H; [left|right];
  rewrite ← Raw.equal_spec; congruence.
Defined.
```

```
Definition lt : t → t → Prop := Raw.lt.
Instance lt_strorder : StrictOrder lt.
Proof.
  unfold lt.
  constructor. {
    move : Raw.lt_Irreflexive.
    rewrite /Irreflexive /complement /Reflexive.
    move ⇒ H x.
    apply H.
  } {
    move : Raw.lt_Transitive.
    rewrite /Transitive.
    move ⇒ H x y z.
    apply H.
  }
Qed.
Instance lt_compat : Proper (eq==>eq==>iff) lt.
Proof.
  repeat red.
  move ⇒ [x1 H_x1_ok] [y1 H_y1_ok] H_eq.
  move ⇒ [x2 H_x2_ok] [y2 H_y2_ok].
  move : H_eq.
  rewrite /eq /lt /Equal /In /=.
  replace (∀ a : elt, Raw.In a x1 ↔ Raw.In a y1) with
    (Raw.Equal x1 y1) by done.
  replace (∀ a : elt, Raw.In a x2 ↔ Raw.In a y2) with
    (Raw.Equal x2 y2) by done.
  rewrite -!Raw.equal_spec !Raw.equal_alt_def.
  move ⇒ → → //.
Qed.
Section Spec.
 Variable s s' : t.
 Variable x y : elt.
 Variable f : elt → bool.
 Notation compatb := (Proper (E.eq==>Logic.eq)) (only parsing).
 Lemma mem_spec : mem x s = true ↔ In x s.
 Proof. exact (Raw.mem_spec _ _ _). Qed.
 Lemma equal_spec : equal s s' = true ↔ Equal s s'.
 Proof. rewrite Raw.equal_spec //. Qed.
 Lemma subset_spec : subset s s' = true ↔ Subset s s'.
 Proof. exact (Raw.subset_spec _ _ _ _). Qed.
```

```
Lemma empty_spec : Empty empty.
Proof. exact Raw.empty_spec. Qed.
Lemma is_empty_spec : is_empty s = true ↔ Empty s.
Proof. rewrite Raw.is_empty_spec //. Qed.
Lemma add_spec : In y (add x s) ↔ E.eq y x ∨ In y s.
Proof. exact (Raw.add_spec _ _ _ _). Qed.
Lemma remove_spec : In y (remove x s) ↔ In y s ∧ ¬E.eq y x.
Proof. exact (Raw.remove_spec _ _ _ _). Qed.
Lemma singleton_spec : In y (singleton x) ↔ E.eq y x.
Proof. exact (Raw.singleton_spec _ _). Qed.
Lemma union_spec : In x (union s s') ↔ In x s ∨ In x s'.
Proof. exact (Raw.union_spec _ _ _ _ _). Qed.
Lemma inter_spec : In x (inter s s') ↔ In x s ∧ In x s'.
Proof. exact (Raw.inter_spec _ _ _ _ _). Qed.
Lemma diff_spec : In x (diff s s') ↔ In x s ∧ ¬In x s'.
Proof. exact (Raw.diff_spec _ _ _ _ _). Qed.
Lemma fold_spec : ∀ (A : Type) (i : A) (f : elt → A → A),
    fold f s i = fold_left (fun a e ⇒ f e a) (elements s) i.
Proof. exact (@Raw.fold_spec _). Qed.
Lemma cardinal_spec : cardinal s = length (elements s).
Proof. exact (@Raw.cardinal_spec s). Qed.
Lemma filter_spec : compatb f →
  (In x (filter f s) ↔ In x s ∧ f x = true).
Proof. move ⇒ _; exact (@Raw.filter_spec _ _ _ _). Qed.
Lemma for_all_spec : compatb f →
  (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
Proof. exact (@Raw.for_all_spec _ _ _). Qed.
Lemma exists_spec : compatb f →
  (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
Proof. exact (@Raw.exists_spec _ _ _). Qed.
Lemma partition_spec1 : compatb f → Equal (fst (partition f s)) (filter f s).
Proof. move ⇒ _; exact (@Raw.partition_spec1 _ _). Qed.
Lemma partition_spec2 : compatb f →
    Equal (snd (partition f s)) (filter (fun x ⇒ negb (f x)) s).
Proof. move ⇒ _; exact (@Raw.partition_spec2 _ _ _). Qed.
Lemma elements_spec1 : InA E.eq x (elements s) ↔ In x s.
Proof. rewrite /In /Raw.In /elements //. Qed.
Lemma elements_spec2w : NoDupA E.eq (elements s).
Proof. exact (Raw.elements_spec2w _ _). Qed.
Lemma elements_spec2 : sort E.lt (elements s).
Proof. exact (Raw.elements_sorted _ _). Qed.
Lemma choose_spec1 : choose s = Some x → In x s.
```

```
  Proof. exact (Raw.choose_spec1 _ _ _). Qed.
  Lemma choose_spec2 : choose s = None → Empty s.
  Proof. exact (Raw.choose_spec2 _). Qed.
  Lemma choose_spec3 : choose s = Some x → choose s' = Some y →
    Equal s s' → E.eq x y.
  Proof.
    intros H1 H2 H3.
    suff → : x = y. {
      apply E.eq_equiv.
    }
    move : H1 H2 H3.
    exact (Raw.choose_spec3 _ _ _ _ _ _ _).
  Qed.

  Lemma min_elt_spec1 : choose s = Some x → In x s.
  Proof. exact (Raw.min_elt_spec1 _ _ _). Qed.
  Lemma min_elt_spec2 : min_elt s = Some x → In y s → ¬ E.lt y x.
  Proof. exact (Raw.min_elt_spec2 _ _ _ _). Qed.
  Lemma min_elt_spec3 : choose s = None → Empty s.
  Proof. exact (Raw.min_elt_spec3 _). Qed.

  Lemma max_elt_spec1 : max_elt s = Some x → In x s.
  Proof. exact (Raw.max_elt_spec1 _ _ _). Qed.
  Lemma max_elt_spec2 : max_elt s = Some x → In y s → ¬ E.lt x y.
  Proof. exact (Raw.max_elt_spec2 _ _ _ _). Qed.
  Lemma max_elt_spec3 : max_elt s = None → Empty s.
  Proof. exact (Raw.max_elt_spec3 _). Qed.

  Lemma compare_spec : CompSpec eq lt s s' (compare s s').
  Proof.
    generalize s s'.
    move ⇒ [s1 H_ok_s1] [s2 H_ok_s2].
    move : (Raw.compare_spec s1 s2).
    rewrite /CompSpec /eq /Equal /In /lt /compare /=.
    replace (∀ a : elt, Raw.In a s1 ↔ Raw.In a s2) with
    (Raw.Equal s1 s2) by done.
    suff H_eq : (Raw.Equal s1 s2) ↔ (s1 = s2). {
      move ⇒ [] H; constructor ⇒ //.
      by rewrite H_eq.
    }
    rewrite -Raw.equal_spec Raw.equal_alt_def //.
  Qed.

 End Spec.

End MSetIntervals.
```

2.1.6 Instantiations

It remains to provide instantiations for commonly used datatypes.

Z

```
Module ElementEncodeZ <: ElementEncode.
  Module E := Z.

  Definition encode (z : Z) := z.
  Definition decode (z : Z) := z.

  Lemma decode_encode_ok: ∀ (e : E.t),
    decode (encode e) = e.
  Proof. by []. Qed.

  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
  Proof. by []. Qed.

  Lemma encode_lt : ∀ (e1 e2 : E.t),
    (Z.lt (encode e1) (encode e2)) ↔ E.lt e1 e2.
  Proof. by []. Qed.
End ElementEncodeZ.

Module MSetIntervalsZ <: SetsOn Z := MSetIntervals ElementEncodeZ.
```

N

```
Module ElementEncodeN <: ElementEncode.
  Module E := N.

  Definition encode (n : N) := Z.of_N n.
  Definition decode (z : Z) := Z.to_N z.

  Lemma decode_encode_ok: ∀ (e : E.t),
    decode (encode e) = e.
  Proof.
    intros e.
    rewrite /encode /decode N2Z.id //.
  Qed.

  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
  Proof.
    intros e1 e2.
    rewrite /encode /Z.eq N2Z.inj_iff /E.eq //.
  Qed.
```

```
  Lemma encode_lt : ∀ (e1 e2 : E.t),
    (Z.lt (encode e1) (encode e2)) ↔ E.lt e1 e2.
  Proof.
    intros e1 e2.
    rewrite /encode -N2Z.inj_lt //.
  Qed.
End ElementEncodeN.
Module MSetIntervalsN <: SetsOn N := MSetIntervals ElementEncodeN.
```

nat

```
Module ElementEncodeNat <: ElementEncode.
  Module E := NPeano.Nat.

  Definition encode (n : nat) := Z.of_nat n.
  Definition decode (z : Z) := Z.to_nat z.

  Lemma decode_encode_ok: ∀ (e : E.t),
    decode (encode e) = e.
  Proof.
    intros e.
    rewrite /encode /decode Nat2Z.id //.
  Qed.

  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
  Proof.
    intros e1 e2.
    rewrite /encode /Z.eq Nat2Z.inj_iff /E.eq //.
  Qed.

  Lemma encode_lt : ∀ (e1 e2 : E.t),
    (Z.lt (encode e1) (encode e2)) ↔ E.lt e1 e2.
  Proof.
    intros e1 e2.
    rewrite /encode -Nat2Z.inj_lt //.
  Qed.
End ElementEncodeNat.

Module MSetIntervalsNat <: SetsOn NPeano.Nat := MSetIntervals ElementEncodeNat.
```

Chapter 3

Library MSetsExtra.MSetIterator

3.1 Fold with abort for sets

This file provided an efficient fold operation for set interfaces. The standard fold iterates over all elements of the set. The efficient one - called foldWithAbort - is allowed to skip certain elements and thereby abort early.

```
Require Export MSetInterface.
Require Import ssreflect.
Require Import MSetWithDups.
Require Import Int.
Require Import MSetGenTree MSetAVL MSetRBT.
Require Import MSetList MSetWeakList.
```

3.1.1 Fold With Abort Operations

We want to provide an efficient folding operation. Efficieny is gained by aborting the folding early, if we know that continuing would not have an effect any more. Formalising this leads to the following specification of foldWithAbort.

Definition foldWithAbortType
  elt element type of set t type of set A return type :=
  $(elt \to A \to A) \to$ f $(elt \to A \to$ **bool**$) \to$ f_abort $t \to$ input set A
$\to$ base value A.

Definition foldWithAbortSpecPred $\{elt\ t$: Type$\}$
  $(In : elt \to t \to$ Prop$)$
  $($fold : $\forall \{A$: Type$\}, (elt \to A \to A) \to t \to A \to A)$
  $(foldWithAbort : \forall \{A$: Type$\}$, foldWithAbortType $elt\ t\ A)$: Prop :=

  $\forall$
    $(A$: Type$)$

result type
$(i\ i' : A)$
base values for foldWithAbort and fold
$(f : elt \to A \to A)$ $(f' : elt \to A \to A)$
fold functions for foldWithAbort and fold
$(f\_abort : elt \to A \to$ **bool**$)$
abort function
$(s : t)$ sets to fold over
$(P : A \to A \to \texttt{Prop})$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P $(\forall\ st\ st'\ e,$ $In\ e\ s \to P\ st\ st' \to P\ (f\ e\ st)\ (f\ e\ st')) \to$

f and f' agree for the elements of s $(\forall\ e\ st,\ In\ e\ s \to (P\ (f\ e\ st)\ (f'\ e\ st))) \to$

f_abort is OK, i.e. all other elements can be skipped without leaving the equivalence relation. $(\forall\ e1\ st,$
$In\ e1\ s \to f\_abort\ e1\ st = \mathsf{true} \to$
$(\forall\ st'\ e2,\ P\ st\ st' \to$
$In\ e2\ s \to e2 \neq e1 \to$
$P\ st\ (f\ e2\ st'))) \to$

The base values are in equivalence relation $P\ i\ i' \to$

The results are in equivalence relation $P\ (foldWithAbort\ f\ f\_abort\ s\ i)\ (fold\ f'\ s\ i')$.

The specification of folding for ordered sets (as represented by interface *Sets*) demands that elements are visited in increasing order. For ordered sets we can therefore abort folding based on the weaker knowledge that greater elements have no effect on the result. The following definition captures this.

`Definition` foldWithAbortGtType
elt element type of set t type of set A return type :=
$(elt \to A \to A) \to$ f $(elt \to A \to$ **bool**$) \to$ f_gt $t \to$ input set $A \to$ base value A.

`Definition` foldWithAbortGtSpecPred $\{elt\ t : \texttt{Type}\}$
$(lt : elt \to elt \to \texttt{Prop})$
$(In : elt \to t \to \texttt{Prop})$
$(\texttt{fold} : \forall\ \{A : \texttt{Type}\},\ (elt \to A \to A) \to t \to A \to A)$
$(foldWithAbortGt : \forall\ \{A : \texttt{Type}\},$ foldWithAbortType $elt\ t\ A) : \texttt{Prop} :=$

$\forall$

$(A : \texttt{Type})$
result type
$(i\ i' : A)$
base values for foldWithAbort and fold
$(f : elt \to A \to A)\ (f' : elt \to A \to A)$
fold functions for foldWithAbort and fold
$(f\_gt : elt \to A \to \textbf{bool})$
abort function
$(s : t)$ sets to fold over
$(P : A \to A \to \texttt{Prop})$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P $(\forall\ st\ st'\ e,$ $In\ e\ s \to P\ st\ st' \to P\ (f\ e\ st)\ (f\ e\ st')) \to$

f and f' agree for the elements of s $(\forall\ e\ st,\ In\ e\ s \to (P\ (f\ e\ st)\ (f'\ e\ st))) \to$

f_abort is OK, i.e. all other elements can be skipped without leaving the equivalence relation. $(\forall\ e1\ st,$
$In\ e1\ s \to f\_gt\ e1\ st = \textsf{true} \to$
$(\forall\ st'\ e2,\ P\ st\ st' \to$
$In\ e2\ s \to lt\ e1\ e2 \to$
$P\ st\ (f\ e2\ st'))) \to$

The base values are in equivalence relation $P\ i\ i' \to$

The results are in equivalence relation $P\ (foldWithAbortGt\ f\ f\_gt\ s\ i)\ (fold\ f'\ s\ i')$.

For ordered sets, we can safely skip elements at the end based on the knowledge that they are all greater than the current element. This leads to serious performance improvements for operations like filtering. It is tempting to try the symmetric operation and skip elements at the beginning based on the knowledge that they are too small to be interesting. So, we would like to start late as well as abort early.

This is indeed a very natural and efficient operation for set implementations based on binary search trees (i.e. the AVL and RBT sets). We can completely symmetrically to skipping greater elements also skip smaller elements. This leads to the following specification.

`Definition` foldWithAbortGtLtType
elt element type of set t type of set A return type :=
$(elt \to A \to \textbf{bool}) \to$ f_lt $(elt \to A \to A) \to$ f $(elt \to A \to \textbf{bool}) \to$ f_gt

$t \rightarrow$ input set $A \rightarrow$ base value A.

`Definition` foldWithAbortGtLtSpecPred $\{elt\ t : \texttt{Type}\}$

$(lt : elt \rightarrow elt \rightarrow \texttt{Prop})$

$(In : elt \rightarrow t \rightarrow \texttt{Prop})$

$(\texttt{fold} : \forall \{A : \texttt{Type}\}, (elt \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$

$(foldWithAbortGtLt : \forall \{A : \texttt{Type}\}, \text{foldWithAbortGtLtType}\ elt\ t\ A) : \texttt{Prop} :=$

$\forall$

$(A : \texttt{Type})$

result type

$(i\ i' : A)$

base values for foldWithAbort and fold

$(f : elt \rightarrow A \rightarrow A)\ (f' : elt \rightarrow A \rightarrow A)$

fold functions for foldWithAbort and fold

$(f\_lt\ f\_gt : elt \rightarrow A \rightarrow$ **bool**$)$

abort functions

$(s : t)$ sets to fold over

$(P : A \rightarrow A \rightarrow \texttt{Prop})$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \rightarrow$

f is for the elements of s compatible with the equivalence relation P $(\forall\ st\ st'\ e,$
$In\ e\ s \rightarrow P\ st\ st' \rightarrow P\ (f\ e\ st)\ (f\ e\ st')) \rightarrow$

f and f' agree for the elements of s $(\forall\ e\ st, In\ e\ s \rightarrow (P\ (f\ e\ st)\ (f'\ e\ st))) \rightarrow$

f_lt is OK, i.e. smaller elements can be skipped without leaving the equivalence relation.
$(\forall\ e1\ st,$
$In\ e1\ s \rightarrow f\_lt\ e1\ st = \text{true} \rightarrow$
$(\forall\ st'\ e2, P\ st\ st' \rightarrow$
$In\ e2\ s \rightarrow lt\ e2\ e1 \rightarrow$
$P\ st\ (f\ e2\ st'))) \rightarrow$

f_gt is OK, i.e. greater elements can be skipped without leaving the equivalence relation.
$(\forall\ e1\ st,$
$In\ e1\ s \rightarrow f\_gt\ e1\ st = \text{true} \rightarrow$
$(\forall\ st'\ e2, P\ st\ st' \rightarrow$
$In\ e2\ s \rightarrow lt\ e1\ e2 \rightarrow$
$P\ st\ (f\ e2\ st'))) \rightarrow$

The base values are in equivalence relation $P\ i\ i' \rightarrow$

The results are in equivalence relation P (*foldWithAbortGtLt f_lt f f_gt s i*) (*fold f' s i'*).

We are interested in folding with abort mainly for runtime performance reasons of extracted code. The argument functions *f_lt*, *f_gt* and *f* of foldWithAbortGtLt often share a large, comparably expensive part of their computation.

In order to further improve runtime performance, therefore another version foldWithAbortPrecompute *f_precompute f_lt f f_gt* that uses an extra function *f_precompute* to allows to compute the commonly used parts of these functions only once. This leads to the following definitions.

`Definition` foldWithAbortPrecomputeType
elt element type of set *t* type of set *A* return type *B* type of precomputed results :=

$(elt \to B) \to$ f_precompute $(elt \to B \to A \to$ **bool**$) \to$ f_lt $(elt \to B \to A \to A) \to$ f $(elt \to B \to A \to$ **bool**$) \to$ f_gt $t \to$ input set $A \to$ base value A.

The specification is similar to the one without precompute, but uses *f_precompute* so avoid doing computations multiple times `Definition` foldWithAbortPrecomputeSpecPred $\{elt\ t :$ `Type`$\}$
$(lt : elt \to elt \to$ `Prop`$)$
$(In : elt \to t \to$ `Prop`$)$
$($`fold` $: \forall \{A :$ `Type`$\}, (elt \to A \to A) \to t \to A \to A)$
$(foldWithAbortPrecompute : \forall \{A\ B :$ `Type`$\},$ foldWithAbortPrecomputeType $elt\ t\ A\ B)$
: `Prop` :=

$\forall$
$(A\ B :$ `Type`$)$
result type
$(i\ i' : A)$
base values for foldWithAbortPrecompute and fold
$(f\_precompute : elt \to B)$
precompute function
$(f : elt \to B \to A \to A)$ $(f' : elt \to A \to A)$
fold functions for foldWithAbortPrecompute and fold
$(f\_lt\ f\_gt : elt \to B \to A \to$ **bool**$)$
abort functions
$(s : t)$ sets to fold over
$(P : A \to A \to$ `Prop`$)$ equivalence relation on results ,

P is an equivalence relation **Equivalence** $P \to$

f is for the elements of s compatible with the equivalence relation P $(\forall$ *st st' e*, *In e s* $\rightarrow$ *P st st'* $\rightarrow$ *P* (*f e* (*f_precompute e*) *st*) (*f e* (*f_precompute e*) *st'*)) $\rightarrow$

f and f' agree for the elements of s $(\forall$ *e st*, *In e s* $\rightarrow$ (*P* (*f e* (*f_precompute e*) *st*) (*f' e st*))) $\rightarrow$

f_lt is OK, i.e. smaller elements can be skipped without leaving the equivalence relation. ($\forall$ *e1 st*,
In e1 s $\rightarrow$ *f_lt e1* (*f_precompute e1*) *st* = true $\rightarrow$
($\forall$ *st' e2*, *P st st'* $\rightarrow$
In e2 s $\rightarrow$ *lt e2 e1* $\rightarrow$
P st (*f e2* (*f_precompute e2*) *st'*))) $\rightarrow$

f_gt is OK, i.e. greater elements can be skipped without leaving the equivalence relation. ($\forall$ *e1 st*,
In e1 s $\rightarrow$ *f_gt e1* (*f_precompute e1*) *st* = true $\rightarrow$
($\forall$ *st' e2*, *P st st'* $\rightarrow$
In e2 s $\rightarrow$ *lt e1 e2* $\rightarrow$
P st (*f e2* (*f_precompute e2*) *st'*))) $\rightarrow$

The base values are in equivalence relation *P i i'* $\rightarrow$

The results are in equivalence relation *P* (*foldWithAbortPrecompute f_precompute f_lt f f_gt s i*) (*fold f' s i'*).

Module Types

We now define a module type for foldWithAbort. This module type demands only the existence of the precompute version, since the other ones can be easily defined via this most efficient one.

`Module Type` HASFOLDWITHABORT (*E* : ORDEREDTYPE) (`Import` *C* : WSETSONWITHDUPS E).

`Parameter` *foldWithAbortPrecompute* : $\forall$ {*A B* : `Type`},
foldWithAbortPrecomputeType elt *t A B*.

`Parameter` *foldWithAbortPrecomputeSpec* :
foldWithAbortPrecomputeSpecPred *E.lt In* (@*fold*) (@*foldWithAbortPrecompute*).

`End` HASFOLDWITHABORT.

3.1.2 Derived operations

Using these efficient fold operations, many operations can be implemented efficiently. We provide lemmata and efficient implementations of useful algorithms via module HasFoldWithAbortOps.

Module HasFoldWithAbortOps (E : OrderedType) (C : WSetsOnWithDups E)
(FT : HasFoldWithAbort E C).

```
  Import FT.
  Import C.
```

First lets define the other folding with abort variants

```
  Definition foldWithAbortGtLt {A} f_lt (f : (elt → A → A)) f_gt :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun e _ st ⇒ f_lt e st)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_gt e st).

  Lemma foldWithAbortGtLtSpec :
     foldWithAbortGtLtSpecPred E.lt In (@fold) (@foldWithAbortGtLt).
  Proof.
    rewrite /foldWithAbortGtLt /foldWithAbortGtLtSpecPred.
    intros A i i' f f' f_lt f_gt s P.
    move ⇒ H_f_compat H_ff' H_lt H_gt H_ii'.
    apply foldWithAbortPrecomputeSpec ⇒ //.
  Qed.

  Definition foldWithAbortGt {A} (f : (elt → A → A)) f_gt :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun _ _ _ ⇒ false)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_gt e st).

  Lemma foldWithAbortGtSpec :
     foldWithAbortGtSpecPred E.lt In (@fold) (@foldWithAbortGt).
  Proof.
    rewrite /foldWithAbortGt /foldWithAbortGtSpecPred.
    intros A i i' f f' f_gt s P.
    move ⇒ H_f_compat H_ff' H_gt H_ii'.
    apply foldWithAbortPrecomputeSpec ⇒ //.
  Qed.

  Definition foldWithAbort {A} (f : (elt → A → A)) f_abort :=
    foldWithAbortPrecompute (fun _ ⇒ tt) (fun e _ st ⇒ f_abort e st)
      (fun e _ st ⇒ f e st) (fun e _ st ⇒ f_abort e st).

  Lemma foldWithAbortSpec :
     foldWithAbortSpecPred In (@fold) (@foldWithAbort).
  Proof.
    rewrite /foldWithAbort /foldWithAbortGtSpecPred.
```

```
  intros A i i' f f' f_abort s P.
  move ⇒ H_equiv_P H_f_compat H_ff' H_abort H_ii'.
  have H_lt_neq: (∀ e1 e2, E.lt e1 e2 → e1 ≠ e2). {
    move ⇒ e1 e2 H_lt H_e12_eq.
    rewrite H_e12_eq in H_lt.
    have : ( Irreflexive E.lt) by apply StrictOrder_Irreflexive.
    rewrite /Irreflexive /Reflexive /complement ⇒ H.
    eapply H, H_lt.
  }
  apply foldWithAbortPrecomputeSpec ⇒ //; (
    move ⇒ e1 st H_in_e1 H_abort_e1 st' e2 H_P H_in_e2 /H_lt_neq H_lt;
    apply (H_abort e1 st H_in_e1 H_abort_e1 st' e2 H_P H_in_e2);
    by auto
  ).
Qed.
```

Specialisations for equality

Let's provide simplified specifications, which use equality instead of an arbitrary equivalence relation on results.   Lemma foldWithAbortPrecomputeSpec_Equal : ∀ (A B : Type) (i : A) (f_pre : elt → B)

```
    (f : elt → B → A → A) (f' : elt → A → A) (f_lt f_gt : elt → B → A → bool) (s : t),

    (∀ e st, In e s → (f e (f_pre e) st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_lt e1 (f_pre e1) st = true →
        (∀ e2, In e2 s → E.lt e2 e1 →
                    (f e2 (f_pre e2) st = st))) →


    (∀ e1 st,
        In e1 s → f_gt e1 (f_pre e1) st = true →
        (∀ e2, In e2 s → E.lt e1 e2 →
                    (f e2 (f_pre e2) st = st))) →

    (foldWithAbortPrecompute f_pre f_lt f f_gt s i) = (fold f' s i).
 Proof.
  intros A B i f_pre f f' f_lt f_gt s H_f' H_lt H_gt.
   eapply (foldWithAbortPrecomputeSpec A B i i f_pre f f'); eauto. {
     apply eq_equivalence.
```

```
    } {
      move => st1 st2 e H_in -> //.
    } {
      move => e1 st H_e1_in H_do_smaller st' e2 <-.
      move : (H_lt e1 st H_e1_in H_do_smaller e2).
      intuition.
    } {
      move => e1 st H_e1_in H_do_greater st' e2 <-.
      move : (H_gt e1 st H_e1_in H_do_greater e2).
      intuition.
    }
Qed.

Lemma foldWithAbortGtLtSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt → A → A) (f' : elt → A → A) (f_lt f_gt : elt → A → bool) (s : t),

    (∀ e st, In e s → (f e st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_lt e1 st = true →
        (∀ e2, In e2 s → E.lt e2 e1 →
                    (f e2 st = st))) →


    (∀ e1 st,
        In e1 s → f_gt e1 st = true →
        (∀ e2, In e2 s → E.lt e1 e2 →
                    (f e2 st = st))) →

    (foldWithAbortGtLt f_lt f f_gt s i) = (fold f' s i).
Proof.
  intros A i f f' f_lt f_gt s H_f' H_lt H_gt.
   eapply (foldWithAbortGtLtSpec A i i f f'); eauto. {
      apply eq_equivalence.
    } {
      move => st1 st2 e H_in -> //.
    } {
      move => e1 st H_e1_in H_do_smaller st' e2 <-.
      move : (H_lt e1 st H_e1_in H_do_smaller e2).
      intuition.
    } {
      move => e1 st H_e1_in H_do_greater st' e2 <-.
```

```
    move : (H_gt e1 st H_e1_in H_do_greater e2).
    intuition.
  }
Qed.

Lemma foldWithAbortGtSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt → A → A) (f' : elt → A → A) (f_gt : elt → A → bool) (s : t),

    (∀ e st, In e s → (f e st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_gt e1 st = true →
        (∀ e2, In e2 s → E.lt e1 e2 →
                  (f e2 st = st))) →

    (foldWithAbortGt f f_gt s i) = (fold f' s i).
Proof.
  intros A i f f' f_gt s H_f' H_gt.
  eapply (foldWithAbortGtSpec A i i f f'); eauto. {
    apply eq_equivalence.
  } {
    move ⇒ st1 st2 e H_in → //.
  } {
    move ⇒ e1 st H_e1_in H_do_greater st' e2 ←.
    move : (H_gt e1 st H_e1_in H_do_greater e2).
    intuition.
  }
Qed.

Lemma foldWithAbortSpec_Equal : ∀ (A : Type) (i : A)
    (f : elt → A → A) (f' : elt → A → A) (f_abort : elt → A → bool) (s : t),

    (∀ e st, In e s → (f e st = f' e st)) →


    (∀ e1 st,
        In e1 s → f_abort e1 st = true →
        (∀ e2, In e2 s → e1 ≠ e2 →
                  (f e2 st = st))) →

    (foldWithAbort f f_abort s i) = (fold f' s i).
Proof.
  intros A i f f' f_abort s H_f' H_abort.
```

```
    eapply (foldWithAbortSpec A i i f f'); eauto. {
      apply eq_equivalence.
    } {
      move ⇒ st1 st2 e H_in → //.
    } {
      move ⇒ e1 st H_e1_in H_do_abort st' e2 ←.
      move : (H_abort e1 st H_e1_in H_do_abort e2).
      intuition.
    }
  Qed.
```

FoldWithAbortSpecArgs

While folding, we are often interested in skipping elements that do not satisfy a certain property P. This needs expressing in terms of skips of smaller of larger elements in order to be done efficiently by our folding functions. Formally, this leads to the definition of *foldWithAbortSpecForPred*.

Given a **FoldWithAbortSpecArg** for a predicate P and a set s, many operations can be implemented efficiently. Below we will provide efficient versions of filter, choose, ∃, ∀ and more. `Record` **FoldWithAbortSpecArg** $\{B\}$:= {

fwasa_f_pre : (elt → B); The precompute function fwasa_f_lt : (elt → B → **bool**); f_lt without state argument fwasa_f_gt : (elt → B → **bool**); f_gt without state argument fwasa_P' : (elt → B → **bool**) the predicate P }.

foldWithAbortSpecForPred s P fwasa holds, if the argument *fwasa* fits the predicate P for set s. `Definition` foldWithAbortSpecArgsForPred $\{A$: Type$\}$
(s : t) (P : elt → **bool**) (*fwasa* : @**FoldWithAbortSpecArg** A) :=

the predicate P' coincides for s and the given precomputation with P (∀ e, In e s → (fwasa_P' *fwasa* e (fwasa_f_pre *fwasa* e) = P e)) ∧

If fwasa_f_lt holds, all elements smaller than the current one don't satisfy predicate P. (∀ $e1$,
In $e1$ s → fwasa_f_lt *fwasa* $e1$ (fwasa_f_pre *fwasa* $e1$) = true →
(∀ $e2$, In $e2$ s → E.lt $e2$ $e1$ → (P $e2$ = false))) ∧

If fwasa_f_gt holds, all elements greater than the current one don't satisfy predicate P. (∀ $e1$,
In $e1$ s → fwasa_f_gt *fwasa* $e1$ (fwasa_f_pre *fwasa* $e1$) = true →
(∀ $e2$, In $e2$ s → E.lt $e1$ $e2$ → (P $e2$ = false))).

Filter with abort

`Definition` filter_with_abort $\{B\}$ (*fwasa* : @**FoldWithAbortSpecArg** B) s :=

```
    @foldWithAbortPrecompute t B (fwasa_f_pre fwasa) (fun e p _ ⇒ fwasa_f_lt fwasa e
p)
      (fun e e_pre s ⇒ if fwasa_P' fwasa e e_pre then add e s else s)
      (fun e p _ ⇒ fwasa_f_gt fwasa e p) s empty.
Lemma filter_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  Equal (filter_with_abort fwasa s)
        (filter P s).
Proof.
  unfold foldWithAbortSpecArgsForPred.
  move ⇒ [] f_pre f_lt f_gt P' P s /=.
  move ⇒ [H_f'] [H_lt] H_gt H_proper.
  rewrite /filter_with_abort /=.
  have → : (foldWithAbortPrecompute f_pre (fun e p _ ⇒ f_lt e p)
   (fun (e : elt) (e_pre : B) (s0 : t) ⇒
    if P' e e_pre then add e s0 else s0) (fun e p _ ⇒ f_gt e p) s empty =
   (fold (fun e s0 ⇒ if P e then add e s0 else s0) s empty)). {
    apply foldWithAbortPrecomputeSpec_Equal. {
      intros e st H_e_in.
      rewrite H_f' //.
    } {
      intros e1 st H_e1_in H_f_lt_eq e2 H_e2_in H_lt_e2_e1.
      rewrite (H_f' _ H_e2_in).
      suff → : (P e2 = false) by done.
      apply (H_lt e1); eauto.
    } {
      intros e1 st H_e1_in H_f_gt_eq e2 H_e2_in H_gt_e2_e1.
      rewrite (H_f' _ H_e2_in).
      suff → : (P e2 = false) by done.
      apply (H_gt e1); eauto.
    }
  }

  rewrite /Equal ⇒ e.
  rewrite fold_spec.
  setoid_rewrite filter_spec ⇒ //.
  suff → : ∀ acc, In e
    (fold_left
       (flip (fun (e0 : elt) (s0 : t) ⇒ if P e0 then add e0 s0 else s0))
       (elements s) acc) ↔ (InA E.eq e (elements s) ∧ P e = true) ∨ (In e acc). {
    rewrite elements_spec1.
```

```
    suff : (¬In e empty) by tauto.
    apply empty_spec.
  }
  induction (elements s) as [| x xs IH] ⇒ acc. {
    rewrite /= InA_nil. tauto.
  } {
    rewrite /= /flip IH InA_cons.
    case_eq (P x). {
      rewrite add_spec.
      intuition.
      left.
      rewrite H0.
      split ⇒ //.
      left.
      apply Equivalence_Reflexive.
    } {
      intuition.
      contradict H2.
      setoid_rewrite H1.
      by rewrite H.
    }
  }
Qed.
```

Choose with abort

```
Definition choose_with_abort {B} (fwasa : @FoldWithAbortSpecArg B) s :=
  foldWithAbortPrecompute (fwasa_f_pre fwasa)
    (fun e p st ⇒ match st with None ⇒ (fwasa_f_lt fwasa e p) | _ ⇒ true end)
    (fun e e_pre st ⇒ match st with None ⇒
       if (fwasa_P' fwasa e e_pre) then Some e else None | _ ⇒ st end)

    (fun e p st ⇒ match st with None ⇒ (fwasa_f_gt fwasa e p) | _ ⇒ true end)
    s None.

Lemma choose_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (match (choose_with_abort fwasa s) with
     | None ⇒ (∀ e, In e s → P e = false)
     | Some e ⇒ In e s ∧ (P e = true)
   end).
Proof.
  rewrite /foldWithAbortSpecArgsForPred.
```

```
move => [] f_pre f_lt f_gt P' P s /=.
move => [H_f'] [H_lt] H_gt H_proper.

set fwasa := {|
    fwasa_f_pre := f_pre;
    fwasa_f_lt := f_lt;
    fwasa_f_gt := f_gt;
    fwasa_P' := P' |}.

suff : (match (choose_with_abort fwasa s) with
    | None => (∀ e, InA E.eq e (elements s) → P e = false)
    | Some e => InA E.eq e (elements s) ∧ (P e = true)
  end). {
    case (choose_with_abort fwasa s). {
      move => e.
      rewrite elements_spec1 //.
    } {
      move => H e H_in.
      apply H.
      rewrite elements_spec1 //.
    }
}

have → : (choose_with_abort fwasa s =
   (fold (fun e st =>
     match st with
       | None => if P e then Some e else None
       | _ => st end) s None)). {
  apply foldWithAbortPrecomputeSpec_Equal. {
    intros e st H_e_in.
    case st => //=.
    rewrite H_f' //.
  } {
    move => e1 [] //= H_e1_in H_f_lt_eq e2 H_e2_in H_lt_e2_e1.
    rewrite (H_f' _ H_e2_in).
    case_eq (P e2) => // H_P_e2.
    contradict H_P_e2.
    apply not_true_iff_false, (H_lt e1); auto.
  } {
    move => e1 [] //= H_e1_in H_f_gt_eq e2 H_e2_in H_gt_e2_e1.
    rewrite (H_f' _ H_e2_in).
    case_eq (P e2) => // H_P_e2.
    contradict H_P_e2.
    apply not_true_iff_false, (H_gt e1); auto.
```

```
      }
    }

    rewrite fold_spec /flip.
    induction (elements s) as [| x xs IH]. {
      rewrite /=.
      move => e /InA_nil //.
    } {
      case_eq (P x) => H_Px; rewrite /= H_Px. {
        have -> : ∀ xs, fold_left (fun (x0 : option elt) (y : elt) =>
                      match x0 with | Some _ => x0 | None => if P y then Some y else
None
                      end) xs (Some x) = Some x. {
          move => ys.
          induction ys => //.
        }
        split; last assumption.
        apply InA_cons_hd.
        apply E.eq_equiv.
      } {
        move : IH.
        case (fold_left
          (fun (x0 : option elt) (y : elt) =>
             match x0 with | Some _ => x0 | None => if P y then Some y else None
             end) xs None). {

             move => e [H_e_in] H_Pe.
             split; last assumption.
             apply InA_cons_tl => //.
        } {
          move => H_e_nin e H_e_in.
          have : (InA E.eq e xs ∨ (E.eq e x)). {
            inversion H_e_in; tauto.
          }
          move => []. {
            apply H_e_nin.
          } {
            move => -> //.
          }
        }
      }
    }
```

```
Qed.
```

Exists and Forall with abort

```
Definition exists_with_abort {B} (fwasa : @FoldWithAbortSpecArg B) s :=
  match choose_with_abort fwasa s with
    | None ⇒ false
    | Some _ ⇒ true
  end.

Lemma exists_with_abort_spec {B} : ∀ fwasa P s,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (exists_with_abort fwasa s =
   exists_ P s).
Proof.
  intros fwasa P s H_fwasa H_proper.
  apply Logic.eq_sym.
  rewrite /exists_with_abort.
  move : (choose_with_abort_spec _ _ _ H_fwasa H_proper).
  case (choose_with_abort fwasa s). {
    move ⇒ e [H_e_in] H_Pe.
    rewrite exists_spec /Exists.
    by ∃ e.
  } {
    move ⇒ H_not_ex.
    apply not_true_iff_false.
    rewrite exists_spec /Exists.
    move ⇒ [e] [H_in] H_pe.
    move : (H_not_ex e H_in).
    rewrite H_pe //.
  }
Qed.
```

Negation leads to forall.

```
Definition forall_with_abort {B} fwasa s :=
  negb (@exists_with_abort B fwasa s).

Lemma forall_with_abort_spec {B} : ∀ fwasa s P,
  @foldWithAbortSpecArgsForPred B s P fwasa →
  Proper (E.eq ==> Logic.eq) P →
  (forall_with_abort fwasa s =
   for_all (fun e ⇒ negb (P e)) s).
Proof.
  intros fwasa s P H_ok H_proper.
  rewrite /forall_with_abort exists_with_abort_spec; auto.
```

```
    rewrite eq_iff_eq_true negb_true_iff -not_true_iff_false.
    rewrite exists_spec.
    setoid_rewrite for_all_spec; last solve_proper.
    rewrite /Exists /For_all.
    split. {
      move ⇒ H_pre x H_x_in.
      rewrite negb_true_iff -not_true_iff_false ⇒ H_Px.
      apply H_pre.
      by ∃ x.
    } {
      move ⇒ H_pre [x] [H_x_in] H_P_x.
      move : (H_pre x H_x_in).
      rewrite H_P_x.
      done.
    }
  Qed.
End HasFoldWithAbortOps.
```

3.1.3 Modules Types For Sets with Fold with Abort

```
Module Type WSetsWithDupsFoldA.
  Declare Module E : OrderedType.
  Include WSetsOnWithDups E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End WSetsWithDupsFoldA.

Module Type WSetsWithFoldA <: WSets.
  Declare Module E : OrderedType.
  Include WSetsOn E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End WSetsWithFoldA.

Module Type SetsWithFoldA <: Sets.
  Declare Module E : OrderedType.
  Include SetsOn E.
  Include HasFoldWithAbort E.
  Include HasFoldWithAbortOps E.
End SetsWithFoldA.
```

3.1.4 Implementations

GenTree implementation

Finally, provide such a fold with abort operation for generic trees. `Module` MAKEGENTREEFOLDA (`Import` E : ORDEREDTYPE) (`Import` I:INFOTYP)
(`Import` Raw:OPS E I)
(M : MSETGENTREE.PROPS E I RAW).

```
Fixpoint foldWithAbort_Raw {A B: Type} (f_pre : E.t → B) f_lt (f: E.t → B → A →
A) f_gt t (base: A) : A :=
    match t with
    | Raw.Leaf ⇒ base
    | Raw.Node _ l x r ⇒
        let x_pre := f_pre x in
        let st0 := if f_lt x x_pre base then base else foldWithAbort_Raw f_pre f_lt f f_gt
l base in
        let st1 := f x x_pre st0 in
        let st2 := if f_gt x x_pre st1 then st1 else foldWithAbort_Raw f_pre f_lt f f_gt
r st1 in
        st2
    end.
```

```
Lemma foldWithAbort_RawSpec : ∀ (A B : Type) (i i' : A) (f_pre : E.t → B)
      (f : E.t → B → A → A) (f' : E.t → A → A) (f_lt f_gt : E.t → B → A → bool) (s
: Raw.tree)
      (P : A → A → Prop),

      (M.bst s) →
      Equivalence P →
      (∀ st st' e, M.In e s → P st st' → P (f e (f_pre e) st) (f e (f_pre e) st')) →
      (∀ e st, M.In e s → P (f e (f_pre e) st) (f' e st)) →


      (∀ e1 st,
          M.In e1 s → f_lt e1 (f_pre e1) st = true →
          (∀ st' e2, P st st' →
                              M.In e2 s → E.lt e2 e1 →
                              P st (f e2 (f_pre e2) st'))) →


      (∀ e1 st,
          M.In e1 s → f_gt e1 (f_pre e1) st = true →
          (∀ st' e2, P st st' →
                              M.In e2 s → E.lt e1 e2 →
```

```
                    P st (f e2 (f_pre e2) st'))) →

    P i i' →
    P (foldWithAbort_Raw f_pre f_lt f f_gt s i) (fold f' s i').
Proof.
  intros A B i i' f_pre f f' f_lt f_gt s P.
  move ⇒ H_bst H_equiv_P H_P_f H_f' H_RL H_RG.

  set base := s.

  move : i i'.
  have : (∀ e, M.In e base → M.In e s). {
    rewrite /In /base //.
  }
  have : M.bst base. {
    apply H_bst.
  }
  move : base.
  clear H_bst.

  induction base as [|c l IHl e r IHr] using M.tree_ind. {
    rewrite /foldWithAbort_Raw /Raw.fold.
    move ⇒ _ _ i i' //.
  } {
    move ⇒ H_bst H_sub i i' H_P_ii'.

    have [H_bst_l [H_bst_r [H_lt_tree_l H_gt_tree_r]]]:
      M.bst l ∧ M.bst r ∧ M.lt_tree e l ∧ M.gt_tree e r. {
      inversion H_bst. done.
    }

    have H_sub_l : (∀ e0 : E.t, M.In e0 l → M.In e0 s ∧ E.lt e0 e). {
      intros e0 H_in_l.
      split; last by apply H_lt_tree_l.
      eapply H_sub.
      rewrite /M.In M.In_node_iff.
      tauto.
    }
    move : (IHl H_bst_l) ⇒ {IHl} IHl {H_bst_l} {H_lt_tree_l}.
    have H_sub_r : (∀ e0 : E.t, M.In e0 r → M.In e0 s ∧ E.lt e e0). {
      intros e0 H_in_r.
      split; last by apply H_gt_tree_r.
      eapply H_sub.
      rewrite /M.In M.In_node_iff.
      tauto.
```

```
    }
    move : (IHr H_bst_r) => {IHr} IHr {H_bst_r} {H_gt_tree_r}.
    have H_in_e : M.In e s. {
      eapply H_sub.
      rewrite /M.In M.In_node_iff.
      right; left.
      apply Equivalence_Reflexive.
    }
    move => {H_sub}.
    rewrite /=.
    set st0 := if f_lt e (f_pre e) i then i else foldWithAbort_Raw f_pre f_lt f f_gt l i.
    set st0' := Raw.fold f' l i'.
    set st1 := f e (f_pre e) st0.
    set st1' := f' e st0'.
    set st2 := if f_gt e (f_pre e) st1 then st1 else foldWithAbort_Raw f_pre f_lt f f_gt
r st1.
    set st2' := Raw.fold f' r st1'.
    have H_P_st0 : P st0 st0'. {
      rewrite /st0 /st0'.
      case_eq (f_lt e (f_pre e) i). {
        move => H_fl_eq.
        move : H_P_ii' H_sub_l.
        move : H_equiv_P H_f' (H_RL _ _ H_in_e H_fl_eq).
        rewrite /M.lt_tree. clear.
        move => H_equiv_P H_f' H_RL.
        move : i'.
        induction l as [|c l IHl e' r IHr] using M.tree_ind. {
          done.
        } {
          intros i' H_P_ii' H_sub_l.
          rewrite /=.
          apply IHr; last first. {
            move => y H_y_in.
            apply H_sub_l.
            rewrite /M.In M.In_node_iff. tauto.
          }
          have [] : (M.In e' s ∧ E.lt e' e). {
            apply H_sub_l.
            rewrite /M.In M.In_node_iff.
            right; left.
            apply Equivalence_Reflexive.
          }
```

```
      move => H_e'_in H_lt_in.
      suff H_P_i : (P i (f e' (f_pre e') (fold f' l i'))). {
        eapply Equivalence_Transitive; first apply H_P_i.
        by apply H_f'.
      }
      eapply H_RL => //.
      apply IHl; last first. {
        move => y H_y_in.
        apply H_sub_l.
        rewrite /M.In M.In_node_iff. tauto.
      }
      assumption.
    }
  } {
    move => _.
    apply IHl => //.
    eapply H_sub_l.
  }
}
have H_P_st1 : P st1 st1'. {
  rewrite /st1 /st1'.
  rewrite -H_f' //.
  apply H_P_f => //.
}
have H_P_st2 : P st2 st2'. {
  rewrite /st2 /st2'.
  clearbody st1 st1'.
  case_eq (f_gt e (f_pre e) st1). {
    move => H_gt_eq.
    move : H_P_st1 H_sub_r.
    move : H_equiv_P (H_RG _ _ H_in_e H_gt_eq) H_f'.
    unfold M.gt_tree. clear.
    move => H_equiv_P H_RG H_f'.
    move : st1'.
    induction r as [|c l IHl e' r IHr] using M.tree_ind. {
      done.
    } {
      intros st1' H_P_st1 H_sub_r.
      rewrite /=.
      apply IHr; last first. {
        move => y H_y_in.
        apply H_sub_r.
```

```
          rewrite /M.In M.In_node_iff. tauto.
        }
        have [] : (M.In e' s ∧ E.lt e e'). {
          apply H_sub_r.
          rewrite /M.In M.In_node_iff.
          right; left.
          apply Equivalence_Reflexive.
        }
        move ⇒ H_e'_in H_lt_ee'.
        suff H_P_st1_aux : (P st1 (f e' (f_pre e') (fold f' l st1'))). {
          eapply Equivalence_Transitive; first apply H_P_st1_aux.
          by apply H_f'.
        }
        eapply H_RG ⇒ //.
        apply IHl; last first. {
          move ⇒ y H_y_in.
          apply H_sub_r.
          rewrite /M.In M.In_node_iff. tauto.
        }
        assumption.
      }
    } {
      move ⇒ _.
      apply IHr ⇒ //.
      eapply H_sub_r.
    }
  }
  done.
}
Qed.
End MakeGenTreeFoldA.
```

AVL implementation

The generic tree implementation naturally leads to an AVL one.

```
Module MakeAVLSetsWithFoldA (X : OrderedType) <: SetsWithFoldA with
Module E := X.
  Include MSetAVL.Make X.
  Include MakeGenTreeFoldA X Z_as_Int Raw Raw.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt (f: elt → B → A → A) f_gt
t (base: A) : A :=
    foldWithAbort_Raw f_pre f_lt f f_gt (t.(this)) base.
```

Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred *X.lt* In fold (@foldWithAbortPrecompute).

Proof.

intros *A B i i' f_pre f f' f_lt f_gt s P*.

move ⇒ *H_P_f H_f' H_RL H_RG H_P_ii'*.

rewrite /foldWithAbortPrecompute /fold.

apply foldWithAbort_RawSpec ⇒ //.

case *s*. rewrite /this /**Raw.Ok** //.

Qed.

Include HasFoldWithAbortOps X.

End MakeAVLSetsWithFoldA.

RBT implementation

The generic tree implementation naturally leads to an RBT one. Module MakeRBTSetsWithFoldA (*X* : OrderedType) <: SetsWithFoldA with Module *E* := X.

Include MSetRBT.Make X.

Include MakeGenTreeFoldA X Color Raw Raw.

Definition foldWithAbortPrecompute {*A B*: Type} *f_pre f_lt* (*f*: elt → *B* → *A* → *A*) *f_gt* *t* (*base*: *A*) : *A* :=

foldWithAbort_Raw *f_pre f_lt f f_gt* (*t*.(this)) *base*.

Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred *X.lt* In fold (@foldWithAbortPrecompute).

Proof.

intros *A B i i' f_pre f f' f_lt f_gt s P*.

move ⇒ *H_P_f H_f' H_RL H_RG H_P_ii'*.

rewrite /foldWithAbortPrecompute /fold.

apply foldWithAbort_RawSpec ⇒ //.

case *s*. rewrite /this /**Raw.Ok** //.

Qed.

Include HasFoldWithAbortOps X.

End MakeRBTSetsWithFoldA.

3.1.5 Sorted Lists Implementation

Module MakeListSetsWithFoldA (*X* : OrderedType) <: SetsWithFoldA with Module *E* := X.

Include MSetList.Make X.

Fixpoint foldWithAbortRaw {*A B*: Type} (*f_pre* : *X.t* → *B*) (*f_lt* : *X.t* → *B* → *A* → **bool**)

```
    (f: X.t → B → A → A) (f_gt : X.t → B → A → bool) (t : list X.t) (acc : A) : A :=
  match t with
    | nil ⇒ acc
    | x :: xs ⇒ (
        let pre_x := f_pre x in
        let acc := f x (pre_x) acc in
        if (f_gt x pre_x acc) then
          acc
        else
          foldWithAbortRaw f_pre f_lt f f_gt xs acc
      )
  end.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt f f_gt t acc :=
    @foldWithAbortRaw A B f_pre f_lt f f_gt t.(this) acc.

  Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred X.lt In fold (@fold-
WithAbortPrecompute).
  Proof.
    intros A B i i' f_pre f f' f_lt f_gt.
    move ⇒ [] l H_is_ok_l P H_equiv_P.
    rewrite /fold /foldWithAbortPrecompute /In /this /Raw.In /Raw.fold.
    move ⇒ H_P_f H_f' H_RL H_RG.

    set base := l.
    move : i i'.
    have : (∀ e, InA X.eq e base → InA X.eq e l). {
      rewrite /base //.
    }
    have : sort X.lt base. {
      rewrite Raw.isok_iff /base //.
    }
    clear H_is_ok_l.

    induction base as [|x xs IH]. {
      by simpl.
    }
    move ⇒ H_sort H_in_xxs i i' Pii' /=.

    have [H_sort_xs H_hd_rel {H_sort}] : Sorted X.lt xs ∧ HdRel X.lt x xs. {
      by inversion H_sort.
    }

    move : H_hd_rel.
    rewrite (Raw.ML.Inf_alt x H_sort_xs) ⇒ H_lt_xs.

    have H_x_in_l : InA X.eq x l. {
```

```
    apply H_in_xxs.
    apply InA_cons_hd.
    apply X.eq_equiv.
  }
  have H_in_xs : (∀ e : X.t, InA X.eq e xs → InA X.eq e l). {
    intros e H_in.
    apply H_in_xxs, InA_cons_tl ⇒ //.
  }

  have H_P_next : P (f x (f_pre x) i) (flip f' i' x). {
    rewrite /flip -H_f' //.
    apply H_P_f ⇒ //.
  }

  case_eq (f_gt x (f_pre x) (f x (f_pre x) i)); last first. {
    move ⇒ _.
    apply IH ⇒ //.
  } {
    move ⇒ H_gt.
    suff H_suff : (∀ st, P (f x (f_pre x) i) st →
        P (f x (f_pre x) i) (fold_left (flip f') xs st)). {
        apply H_suff ⇒ //.
    }
    move : H_in_xs H_lt_xs.

    clear IH H_in_xxs H_sort_xs.
    move : (H_RG x _ H_x_in_l H_gt) ⇒ H_RG_x.
    induction xs as [| x' xs' IH']. {
      done.
    } {
      intros H_in_xs H_lt_xs st H_P_st.
      rewrite /=.
      have H_x'_in_l : InA X.eq x' l. {
        apply H_in_xs.
        apply InA_cons_hd, X.eq_equiv.
      }
      apply IH'. {
        intros e H.
        apply H_in_xs, InA_cons_tl ⇒ //.
      } {
        intros e H.
        apply H_lt_xs, InA_cons_tl ⇒ //.
      } {
```

```
        rewrite /flip -H_f' //.
        apply H_RG_x => //.
        apply H_lt_xs.
        apply InA_cons_hd, X.eq_equiv.
      }
    }
  }
Qed.

Include HasFoldWithAbortOps X.
End MakeListSetsWithFoldA.
```

Unsorted Lists without Dups Implementation

```
Module MakeWeakListSetsWithFoldA (X : OrderedType) <: WSetsWithFoldA
with Module E := X.
  Module Raw := MSetWeakList.MakeRaw X.
  Module E := X.
  Include WRaw2SetsOn E Raw.

  Fixpoint foldWithAbortRaw {A B: Type} (f_pre : X.t → B) (f_lt : X.t → B → A →
bool)
    (f: X.t → B → A → A) (f_gt : X.t → B → A → bool) (t : list X.t) (acc : A) : A :=
  match t with
    | nil ⇒ acc
    | x :: xs ⇒ (
        let pre_x := f_pre x in
        let acc := f x (pre_x) acc in
        if (f_gt x pre_x acc) && (f_lt x pre_x acc) then
          acc
        else
          foldWithAbortRaw f_pre f_lt f f_gt xs acc
      )
  end.

  Definition foldWithAbortPrecompute {A B: Type} f_pre f_lt f f_gt t acc :=
    @foldWithAbortRaw A B f_pre f_lt f f_gt t.(this) acc.

  Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred X.lt In fold (@fold-
WithAbortPrecompute).
  Proof.
    intros A B i i' f_pre f f' f_lt f_gt.
    move ⇒ [] l H_is_ok_l P H_P_equiv.
    rewrite /fold /foldWithAbortPrecompute /In /this /Raw.In /Raw.fold.
    move ⇒ H_P_f H_f' H_RL H_RG.
```

```
set base := l.
move : i i'.
have : (∀ e, InA X.eq e base → InA X.eq e l). {
  rewrite /base //.
}
have : NoDupA X.eq base. {
  apply H_is_ok_l.
}
clear H_is_ok_l.

induction base as [| x xs IH]. {
  by simpl.
}
move ⇒ H_nodup_xxs H_in_xxs i i' Pii' /=.

have [H_nin_x_xs H_nodup_xs {H_nodup_xxs}] : ¬ InA X.eq x xs ∧ NoDupA X.eq xs.
{
  by inversion H_nodup_xxs.
}

have H_x_in_l : InA X.eq x l. {
  apply H_in_xxs.
  apply InA_cons_hd.
  apply X.eq_equiv.
}
have H_in_xs : (∀ e : X.t, InA X.eq e xs → InA X.eq e l). {
  intros e H_in.
  apply H_in_xxs, InA_cons_tl ⇒ //.
}

have H_P_next : P (f x (f_pre x) i) (flip f' i' x). {
  rewrite /flip -H_f' //.
  apply H_P_f ⇒ //.
}

case_eq (f_gt x (f_pre x) (f x (f_pre x) i) &&
         f_lt x (f_pre x) (f x (f_pre x) i)); last first. {
  move ⇒ _.
  apply IH ⇒ //.
} {
  move ⇒ /andb_true_iff [H_gt H_lt].
  suff H_suff : (∀ st, P (f x (f_pre x) i) st →
      P (f x (f_pre x) i) (fold_left (flip f') xs st)). {
      apply H_suff ⇒ //.
```

```
}

have H_neq_xs : ∀ e, List.In e xs → X.lt x e ∨ X.lt e x. {
  intros e H_in.
  move : (X.compare_spec x e).
  case (X.compare x e) ⇒ H_cmp; inversion H_cmp. {
    contradict H_nin_x_xs.
    rewrite InA_alt.
    by ∃ e.
  } {
    by left.
  } {
    by right.
  }
}
move : H_in_xs H_neq_xs.
clear IH H_in_xxs H_nodup_xs.
move : (H_RG x _ H_x_in_l H_gt) ⇒ H_RG_x.
move : (H_RL x _ H_x_in_l H_lt) ⇒ H_RL_x.
induction xs as [| x' xs' IH']. {
  done.
} {
  intros H_in_xs H_neq_xs st H_P_st.
  rewrite /=.
  have H_x'_in_xxs' : List.In x' (x' :: xs'). {
    simpl; by left.
  }
  have H_x'_in_l : InA X.eq x' l. {
    apply H_in_xs.
    apply InA_cons_hd, X.eq_equiv.
  }
  apply IH'. {
    intros H.
    apply H_nin_x_xs, InA_cons_tl ⇒ //.
  } {
    intros e H.
    apply H_in_xs, InA_cons_tl ⇒ //.
  } {
    intros e H.
    apply H_neq_xs, in_cons ⇒ //.
  } {
    rewrite /flip -H_f' //.
```

```
          move : (H_neq_xs x' H_x'_in_xxs') => [] H_cmp. {
            apply H_RG_x => //.
          } {
            apply H_RL_x => //.
          }
        }
      }
    }
  Qed.

  Include HasFoldWithAbortOps X.
End MakeWeakListSetsWithFoldA.
```

Chapter 4

Library MSetsExtra.MSetListWithDups

4.1 Weak sets implemented as lists with duplicates

This file contains an implementation of the weak set interface WSetsOnWithDupsExtra. As a datatype unsorted lists are used that might contain duplicates.

This implementation is useful, if one needs very efficient insert and union operation, and can guarantee that one does not add too many duplicates. The operation elements_dist is implemented by sorting the list first. Therefore this instantiation can only be used if the element type is ordered.

```
Require Export MSetInterface.
Require Import ssreflect.
Require Import List OrdersFacts OrdersLists.
Require Import Sorting Permutation.
Require Import MSetWithDups.
```

4.1.1 Removing duplicates from sorted lists

The following module RemoveDupsFromSorted defines an operation remove_dups_from_sortedA that removes duplicates from a sorted list. In order to talk about sorted lists, the element type needs to be ordered.

This function is combined with a sort function to get a function remove_dups_by_sortingA to sort unsorted lists and then remove duplicates. `Module RemoveDupsFromSorted (Import X:OrderedType).`

First, we need some infrastructure for our ordered type `Module Import MX := OrderedTypeFacts X.`

```
  Module Import XTotalLeBool <: TotalLeBool.
    Definition t := X.t.
```

```
Definition leb x y :=
  match X.compare x y with
    | Lt ⇒ true
    | Eq ⇒ true
    | Gt ⇒ false
  end.

Infix "<=?" := leb (at level 35).

Theorem leb_total : ∀ (a1 a2 : t), (a1 <=? a2 = true) ∨ (a2 <=? a1 = true).
Proof.
  intros a1 a2.
  unfold leb.
  rewrite (compare_antisym a1 a2).
  case (X.compare a1 a2); rewrite /=; tauto.
Qed.

Definition le x y := (leb x y = true).
```

End XTotalLeBool.

```
Lemma eqb_eq_alt : ∀ x y, eqb x y = true ↔ eq x y.
Proof.
  intros x y.
  rewrite eqb_alt -compare_eq_iff.
  case (compare x y) ⇒ //.
Qed.
```

Now we can define our main function `Fixpoint remove_dups_from_sortedA_aux (acc : list t) (l : list t) : list t :=`

```
  match l with
  | nil ⇒ List.rev' acc
  | x :: xs ⇒
     match xs with
     | nil ⇒ List.rev' (x :: acc)
     | y :: ys ⇒
         if eqb x y then
           remove_dups_from_sortedA_aux acc xs
         else
           remove_dups_from_sortedA_aux (x::acc) xs
     end
  end.

Definition remove_dups_from_sortedA := remove_dups_from_sortedA_aux (nil : list t).
```

We can prove some technical lemmata `Lemma remove_dups_from_sortedA_aux_alt : ∀ (l : list X.t) acc,`

```
  remove_dups_from_sortedA_aux acc l =
  List.rev acc ++ (remove_dups_from_sortedA l).
```

```
Proof.
  unfold remove_dups_from_sortedA.
  induction l as [| x xs IH] => acc. {
    rewrite /remove_dups_from_sortedA_aux /rev' -!rev_alt /= app_nil_r //.
  } {
    rewrite /=.
    case_eq xs. {
      rewrite /rev' -!rev_alt //.
    } {
      move => y ys H_xs_eq.
      rewrite -!H_xs_eq !(IH acc) !(IH (x :: acc)) (IH (x::nil)).
      case (eqb x y) => //.
      rewrite /= -app_assoc //.
    }
  }
Qed.

Lemma remove_dups_from_sortedA_alt :
  ∀ (l : list t),
  remove_dups_from_sortedA l =
  match l with
  | nil => nil
  | x :: xs =>
     match xs with
     | nil => l
     | y :: ys =>
         if eqb x y then
           remove_dups_from_sortedA xs
         else
           x :: remove_dups_from_sortedA xs
     end
  end.
Proof.
  case. {
    done.
  } {
    intros x xs.
    rewrite /remove_dups_from_sortedA /= /rev' /=.
    case xs => //.
    move => y ys.
    rewrite !remove_dups_from_sortedA_aux_alt /= //.
  }
Qed.
```

```
Lemma remove_dups_from_sortedA_hd :
    ∀ x xs,
    ∃ (x':t) xs',
      remove_dups_from_sortedA (x :: xs) =
      (x' :: xs') ∧ (eqb x x' = true).
Proof.
  intros x xs.
  move : x;
  induction xs as [| y ys IH] ⇒ x. {
    rewrite remove_dups_from_sortedA_alt.
    ∃ x, nil.
    split; first reflexivity.
    rewrite eqb_alt compare_refl //.
  } {
    rewrite remove_dups_from_sortedA_alt.
    case_eq (eqb x y); last first. {
      move ⇒ _.
      ∃ x, (remove_dups_from_sortedA (y :: ys)).
      split; first reflexivity.
      rewrite eqb_alt compare_refl //.
    } {
      move ⇒ H_eqb_xy.
      move : (IH y) ⇒ {IH} [x'] [xs'] [->] H_eqb_yx'.
      ∃ x', xs'.
      split; first done.
      move : H_eqb_xy H_eqb_yx'.
      rewrite !eqb_eq_alt.
      apply MX.eq_trans.
    }
  }
Qed.
```

Finally we get our main result for removing duplicates from sorted lists Lemma remove_dups_from_sorte
:

```
  ∀ (l : list t),
    Sorted le l →
    let l' := remove_dups_from_sortedA l in (

    Sorted lt l' ∧
    NoDupA eq l' ∧
    (∀ x, InA eq x l ↔ InA eq x l')).
Proof.
  simpl.
```

```
induction l as [| x xs IH]. {
  rewrite remove_dups_from_sortedA_alt.
  done.
} {
  rewrite remove_dups_from_sortedA_alt.
  move : IH.
  case xs => {xs}. {
    move => _.
    split; last split. {
      apply Sorted_cons => //.
    } {
      apply NoDupA_singleton.
    } {
      done.
    }
  } {
    move => y ys IH H_sorted_x_y_ys.
    apply Sorted_inv in H_sorted_x_y_ys as [H_sorted_y_ys H_hd_rel].
    apply HdRel_inv in H_hd_rel.
    have : ∃ y' ys',
      remove_dups_from_sortedA (y :: ys) = y' :: ys' ∧
      eqb y y' = true. {
      apply remove_dups_from_sortedA_hd => //.
    }
    move => [y'] [ys'] [H_yys'_intro] /eqb_eq_alt H_eq_y_y'.
    move : (IH H_sorted_y_ys).
    rewrite !H_yys'_intro.
    move => {IH} [IH1] [IH2] IH3.
    case_eq (eqb x y). {
      rewrite eqb_eq_alt => H_eq_x_y.
      split => //.
      split => //.
      move => x'.
      rewrite InA_cons IH3.
      split; last tauto.
      move => [] //.
      move => H_eq_x'_x.
      apply InA_cons_hd.
      apply eq_trans with (y := x) => //.
      apply eq_trans with (y := y) => //.
    }
    move => H_neqb_x_y.
```

```
        have H_sorted : Sorted lt (x :: y' :: ys'). {
          apply Sorted_cons => //.
          apply HdRel_cons.
          rewrite -compare_lt_iff.
          suff : (compare x y = Lt). {
            setoid_rewrite compare_compat; eauto;
              apply eq_refl.
          }
          move : H_hd_rel H_neqb_x_y.
          rewrite eqb_alt /le /leb.
          case (compare x y) => //.
        }
        split; last split. {
          assumption.
        } {
          apply NoDupA_cons => //.
          move => /InA_alt [x'] [H_eq_xx'] H_in_x'.
          have : Forall (lt x) (y' :: ys'). {
            apply Sorted_extends => //.
            rewrite /Relations_1.Transitive.
            by apply lt_trans.
          }
          rewrite Forall_forall => H_forall.
          move : (H_forall _ H_in_x') => {H_forall}.
          move : H_eq_xx'.
          rewrite -compare_lt_iff -compare_eq_iff.
          move => -> //.
        } {
          move => x0.
          rewrite !(InA_cons eq x0 x) IH3 //.
        }
      }
    }
Qed.
```

Next, we combine it with sorting `Module Import XSort := Sort XTotalLeBool.`

```
Definition remove_dups_by_sortingA (l : list t) : list t :=
  remove_dups_from_sortedA (XSort.sort l).

Lemma remove_dups_by_sortingA_spec :
  ∀ (l : list t),
    let l' := remove_dups_by_sortingA l in (
```

```
      Sorted lt l' ∧
      NoDupA eq l' ∧
      (∀ x, InA eq x l ↔ InA eq x l')).
  Proof.
    intro l.
    suff : (∀ x : X.t, InA eq x (sort l) ↔ InA eq x l) ∧
           Sorted le (sort l). {

      unfold remove_dups_by_sortingA.
      move : (remove_dups_from_sortedA_spec (sort l)).
      simpl.
      move ⇒ H_spec [H_in_sort H_sorted_sort].
      move : (H_spec H_sorted_sort).
      move ⇒ [H1] [H2] H3.
      split ⇒ //.
      split ⇒ //.
      move ⇒ x.
      rewrite -H_in_sort H3 //.
    }

    split. {
      have H_in_sort : ∀ x, List.In x (XSort.sort l) ↔ List.In x l. {
        intros x.
        have H_perm := (XSort.Permuted_sort l).
        split; apply Permutation_in ⇒ //.
        by apply Permutation_sym.
      }

      intros x.
      rewrite !InA_alt.
      setoid_rewrite H_in_sort ⇒ //.
    } {

      move : (LocallySorted_sort l).
      rewrite /is_true /le /leb //.
    }
  Qed.
End RemoveDupsFromSorted.
```

4.1.2 Operations Module

With removing duplicates defined, we can implement the operations for our set implementation easily.

```
Module Ops (X:OrderedType) <: WOps X.
  Module RDFS := RemoveDupsFromSorted X.
  Module Import MX := OrderedTypeFacts X.

  Definition elt := X.t.
  Definition t := list elt.

  Definition empty : t := nil.

  Definition is_empty (l : t) := match l with nil ⇒ true | _ ⇒ false end.
  Fixpoint mem (x : elt) (s : t) : bool :=
    match s with
    | nil ⇒ false
    | y :: l ⇒
           match X.compare x y with
               Eq ⇒ true
             | _ ⇒ mem x l
           end
    end.

  Definition add x (s : t) := x :: s.
  Definition singleton (x : elt) := x :: nil.

  Fixpoint rev_filter_aux acc (f : elt → bool) s :=
    match s with
       nil ⇒ acc
     | (x :: xs) ⇒ rev_filter_aux (if (f x) then (x :: acc) else acc) f xs
    end.
  Definition rev_filter := rev_filter_aux nil.

  Definition filter (f : elt → bool) (s : t) : t := rev_filter f s.

  Definition remove x s :=
    rev_filter (fun y ⇒ match X.compare x y with Eq ⇒ false | _ ⇒ true end) s.

  Definition union (s1 s2 : t) : t :=
    List.rev_append s2 s1.

  Definition inter (s1 s2 : t) : t :=
    rev_filter (fun y ⇒ mem y s2) s1.

  Definition elements (x : t) : list elt := x.

  Definition elements_dist (x : t) : list elt :=
    RDFS.remove_dups_by_sortingA x.

  Definition fold {B : Type} (f : elt → B → B) (s : t) (i : B) : B :=
```

```
    fold_left (flip f) (elements s) i.
  Definition diff (s s' : t) : t := fold remove s' s.
  Definition subset (s s' : t) : bool :=
    List.forallb (fun x ⇒ mem x s') s.
  Definition equal (s s' : t) : bool := andb (subset s s') (subset s' s).
  Fixpoint for_all (f : elt → bool) (s : t) : bool :=
    match s with
    | nil ⇒ true
    | x :: l ⇒ if f x then for_all f l else false
    end.
  Fixpoint exists_ (f : elt → bool) (s : t) : bool :=
    match s with
    | nil ⇒ false
    | x :: l ⇒ if f x then true else exists_ f l
    end.
  Fixpoint partition_aux (a1 a2 : t) (f : elt → bool) (s : t) : t × t :=
    match s with
    | nil ⇒ (a1, a2)
    | x :: l ⇒
        if f x then partition_aux (x :: a1) a2 f l else
                    partition_aux a1 (x :: a2) f l
    end.
  Definition partition := partition_aux nil nil.
  Definition cardinal (s : t) : nat := length (elements_dist s).
  Definition choose (s : t) : option elt :=
     match s with
      | nil ⇒ None
      | x::_ ⇒ Some x
     end.
End Ops.
```

4.1.3 Main Module

Using these operations, we can define the main functor. For this, we need to prove that the provided operations do indeed satisfy the weak set interface. This is mostly straightforward and unsurprising. The only interesting part is that removing duplicates from a sorted list behaves as expected. This has however already been proved in module RemoveDupsFromSorted. Module Make (E:OrderedType) <: WSetsOnWithDupsExtra E.

```
  Include Ops E.
  Import MX.
```

4.1.4 Proofs of set operation specifications.

Logical predicates

```
Definition In x (s : t) := SetoidList.InA E.eq x s.
Instance In_compat : Proper (E.eq==>eq==>iff) In.
Proof. repeat red. intros. rewrite H H0. auto. Qed.

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) s := ∃ x, In x s ∧ P x.

Notation "s [=] t" := (Equal s t) (at level 70, no associativity).
Notation "s [<=] t" := (Subset s t) (at level 70, no associativity).

Definition eq : t → t → Prop := Equal.
Lemma eq_equiv : Equivalence eq.
Proof.
  constructor. {
    done.
  } {
    by constructor; rewrite H.
  } {
    by constructor; rewrite H H0.
  }
Qed.
```

Specifications of set operators

```
Notation compatb := (Proper (E.eq==>Logic.eq)) (only parsing).

Lemma mem_spec : ∀ s x, mem x s = true ↔ In x s.
Proof.
  induction s as [| y s' IH]. {
    move ⇒ x.
    rewrite /= /In InA_nil.
    split ⇒ //.
  } {
    move ⇒ x.
    rewrite /= /In InA_cons.
    move : (MX.compare_eq_iff x y).
    case (E.compare x y). {
      tauto.
    } {
      rewrite IH; intuition; inversion H.
    } {
      rewrite IH; intuition; inversion H.
```

```
      }
    }
  Qed.

  Lemma subset_spec : ∀ s s', subset s s' = true ↔ s [<=] s'.
  Proof.
    intros s s'.
    rewrite /subset forallb_forall /Subset /In.
    split. {
      move ⇒ H z /InA_alt [] x [H_z_eq] H_in.
      move : (H _ H_in).
      rewrite mem_spec.
      setoid_replace z with x ⇒ //.
    } {
      move ⇒ H z H_in.
      rewrite mem_spec.
      apply H, In_InA ⇒ //.
      apply E.eq_equiv.
    }
  Qed.

  Lemma equal_spec : ∀ s s', equal s s' = true ↔ s [=] s'.
  Proof.
    intros s s'.
    rewrite /Equal /equal Bool.andb_true_iff !subset_spec /Subset.
    split. {
      move ⇒ [H1 H2] a.
      split.
        - by apply H1.
        - by apply H2.
    } {
      move ⇒ H.
      split; move ⇒ a; rewrite H //.
    }
  Qed.

  Lemma eq_dec : ∀ x y : t, {eq x y} + {¬ eq x y}.
  Proof.
    intros x y.
    change ({Equal x y}+{¬Equal x y}).
    destruct (equal x y) eqn:H; [left|right];
     rewrite ← equal_spec; congruence.
  Qed.

  Lemma empty_spec : Empty empty.
```

```
Proof. rewrite /Empty /empty /In. move ⇒ a /InA_nil //. Qed.
Lemma is_empty_spec : ∀ s, is_empty s = true ↔ Empty s.
Proof.
  rewrite /is_empty /Empty /In.
  case; split ⇒ //. {
    move ⇒ _ a.
    rewrite InA_nil //.
  } {
    move ⇒ H; contradiction (H a).
    apply InA_cons_hd.
    apply Equivalence_Reflexive.
  }
Qed.
Lemma add_spec : ∀ s x y, In y (add x s) ↔ E.eq y x ∨ In y s.
Proof.
  intros s x y.
  rewrite /add /In InA_cons //.
Qed.
Lemma singleton_spec : ∀ x y, In y (singleton x) ↔ E.eq y x.
Proof.
  intros x y.
  rewrite /singleton /In InA_cons.
  split. {
    move ⇒ [] // /InA_nil //.
  } {
    by left.
  }
Qed.
Hint Resolve (@Equivalence_Reflexive _ _ E.eq_equiv).
Hint Immediate (@Equivalence_Symmetric _ _ E.eq_equiv).
Hint Resolve (@Equivalence_Transitive _ _ E.eq_equiv).
Lemma rev_filter_aux_spec : ∀ s acc x f, compatb f →
  (In x (rev_filter_aux acc f s) ↔ (In x s ∧ f x = true) ∨ (In x acc)).
Proof.
  intros s acc x f H_compat.
  move : x acc.
  induction s as [| y s' IH]. {
    intros x acc.
    rewrite /rev_filter_aux /In InA_nil.
    tauto.
  } {
```

```
    intros x acc.
    rewrite /= IH /In.
    case_eq (f y) => H_fy; rewrite !InA_cons; intuition. {
      left.
      split; first by left.
      setoid_replace x with y => //.
    } {
      contradict H1.
      setoid_replace x with y => //.
      by rewrite H_fy.
    }
  }
Qed.

Lemma filter_spec : ∀ s x f, compatb f →
  (In x (filter f s) ↔ In x s ∧ f x = true).
Proof.
  intros s x f H_compat.
  rewrite /filter /rev_filter rev_filter_aux_spec /In InA_nil.
  tauto.
Qed.

Lemma remove_spec : ∀ s x y, In y (remove x s) ↔ In y s ∧ ¬E.eq y x.
Proof.
  intros s x y.
  rewrite /remove /rev_filter.
  have H_compat : compatb ((fun y0 : elt ⇒
       match E.compare x y0 with
       | Eq ⇒ false
       | _ ⇒ true
       end)). {
    repeat red; intros.
    setoid_replace x0 with y0 => //.
  }
  rewrite rev_filter_aux_spec /In InA_nil.
  have → : (E.eq y x ↔ E.eq x y). {
    split; move => ?; by apply Equivalence_Symmetric.
  }
  rewrite -compare_eq_iff.
  case (E.compare x y). {
    intuition.
  } {
    intuition.
    inversion H0.
```

```
  } {
    intuition.
    inversion H0.
  }
Qed.
```

```
Lemma union_spec : ∀ s s' x, In x (union s s') ↔ In x s ∨ In x s'.
Proof.
  intros s s' x.
  rewrite /union /In rev_append_rev InA_app_iff InA_rev; tauto.
Qed.
```

```
Lemma inter_spec : ∀ s s' x, In x (inter s s') ↔ In x s ∧ In x s'.
Proof.
  intros s s' x.
  have H_compat : compatb (fun y : elt ⇒ mem y s'). {
    repeat red; intros.
    suff : ( mem x0 s' = true ↔ mem y s' = true). {
      case (mem y s'), (mem x0 s'); intuition.
    }
    rewrite !mem_spec /In.
    setoid_replace x0 with y ⇒ //.
  }
  rewrite /inter rev_filter_aux_spec mem_spec /In InA_nil.
  tauto.
Qed.
```

```
Lemma fold_spec : ∀ s (A : Type) (i : A) (f : elt → A → A),
  fold f s i = fold_left (flip f) (elements s) i.
Proof. done. Qed.
```

```
Lemma elements_spec1 : ∀ s x, InA E.eq x (elements s) ↔ In x s.
Proof.
  intros s x.
  rewrite /elements /In //.
Qed.
```

```
Lemma diff_spec : ∀ s s' x, In x (diff s s') ↔ In x s ∧ ¬In x s'.
Proof.
  intros s s' x.
  rewrite /diff fold_spec -(elements_spec1 s').
  move : s.
  induction (elements s') as [| y ys IH] ⇒ s. {
    rewrite InA_nil /=; tauto.
  } {
    rewrite /= IH InA_cons /flip remove_spec.
```

```
    tauto.
  }
Qed.

Lemma cardinal_spec : ∀ s, cardinal s = length (elements_dist s).
Proof. rewrite /cardinal //. Qed.

Lemma for_all_spec : ∀ s f, compatb f →
  (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
Proof.
  intros s f H_compat.
  rewrite /For_all.
  induction s as [| x xs IH]. {
    rewrite /= /In.
    split ⇒ //.
    move ⇒ _ x /InA_nil //.
  } {
    rewrite /=.
    case_eq (f x) ⇒ H_fx. {
      rewrite IH.
      split. {
        move ⇒ H x' /InA_cons []. {
          move ⇒ → //.
        } {
          apply H.
        }
      } {
        move ⇒ H x' H_in.
        apply H.
        apply InA_cons.
        by right.
      }
    } {
      split ⇒ //.
      move ⇒ H.
      suff : f x = true. {
        rewrite H_fx //.
      }
      apply H.
      apply InA_cons_hd.
      done.
    }
  }
Qed.
```

```
Lemma exists_spec : ∀ s f, compatb f →
  (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
Proof.
  intros s f H_compat.
  rewrite /Exists.
  induction s as [| x xs IH]. {
    rewrite /= /In.
    split ⇒ //.
    move ⇒ [x] [] /InA_nil //.
  } {
    rewrite /=.
    case_eq (f x) ⇒ H_fx. {
      split ⇒ // _.
      ∃ x.
      split ⇒ //.
      apply InA_cons_hd.
      done.
    } {
      rewrite IH.
      split. {
        move ⇒ [x'] [H_in] H_fx'.
        ∃ x'.
        split ⇒ //.
        apply InA_cons.
        by right.
      } {
        move ⇒ [x'] [] /InA_cons []. {
          move ⇒ →.
          rewrite H_fx //.
        } {
          by ∃ x'.
        }
      }
    }
  }
Qed.

Lemma partition_aux_spec : ∀ a1 a2 s f,
  (partition_aux a1 a2 f s = (rev_filter_aux a1 f s, rev_filter_aux a2 (fun x ⇒ negb (f
x)) s)).
Proof.
  move ⇒ a1 a2 s f.
  move : a1 a2.
```

```
  induction s as [| x xs IH]. {
    rewrite /partition_aux /rev_filter_aux //.
  } {
    intros a1 a2.
    rewrite /= IH.
    case (f x) => //.
  }
Qed.

Lemma partition_spec1 : ∀ s f, compatb f →
  fst (partition f s) [=] filter f s.
Proof.
  move => s f _.
  rewrite /partition partition_aux_spec /fst /filter /rev_filter //.
Qed.

Lemma partition_spec2 : ∀ s f, compatb f →
  snd (partition f s) [=] filter (fun x => negb (f x)) s.
Proof.
  move => s f _.
  rewrite /partition partition_aux_spec /snd /filter /rev_filter //.
Qed.

Lemma choose_spec1 : ∀ s x, choose s = Some x → In x s.
Proof.
 move => [] // y s' x [->].
 rewrite /In.
 apply InA_cons_hd.
 apply Equivalence_Reflexive.
Qed.

Lemma choose_spec2 : ∀ s, choose s = None → Empty s.
Proof. move => [] // _ a. rewrite /In InA_nil //. Qed.

Lemma elements_dist_spec_full :
  ∀ s,
    Sorted E.lt (elements_dist s) ∧
    NoDupA E.eq (elements_dist s) ∧
    (∀ x, InA E.eq x (elements_dist s) ↔ InA E.eq x (elements s)).
Proof.
  move => s.
  rewrite /elements_dist /elements.
  move : (RDFS.remove_dups_by_sortingA_spec s).
  simpl.
  firstorder.
Qed.
```

Lemma elements_dist_spec1 : $\forall\ x\ s$, **InA** *E.eq* x (elements_dist s) $\leftrightarrow$
InA *E.eq* x (elements s).

Proof. intros; apply elements_dist_spec_full. Qed.

Lemma elements_dist_spec2w : $\forall\ s$, **NoDupA** *E.eq* (elements_dist s).

Proof. intros; apply elements_dist_spec_full. Qed.

End MAKE.

Chapter 5

Library MSetsExtra.MSetWithDups

5.1 Signature for weak sets which may contain duplicates

The interface *WSetsOn* demands that elements returns a list without duplicates and that the fold function iterates over this result. Another potential problem is that the function cardinal is supposed to return the length of the elements list.

Therefore, implementations that store duplicates internally and for which the fold function would visit elements multiple times are ruled out. Such implementations might be desirable for performance reasons, though. One such (sometimes useful) example are unsorted lists with duplicates. They have a very efficient insert and union operation. If they are used in such a way that not too many membership tests happen and that not too many duplicates accumulate, it might be a very efficient datastructure.

In order to allow efficient weak set implementations that use duplicates internally, we provide new module types in this file. There is WSetsOnWithDups, which is a proper subset of *WSetsOn*. It just removes the problematic properties of elements and cardinal.

Since one is of course interested in specifying the cardinality and in computing a list of elements without duplicates, there is also an extension WSetsOnWithDupsExtra of WSetsOnWithDups. This extension introduces a new operation elements_dist, which is a version of elements without duplicates. This allows to specify *cardinality* with respect to elements_dist.

```
Require Import Coq.MSets.MSetInterface.
Require Import ssreflect.
```

5.1.1 WSetsOnWithDups

The module type *WSetOnWithDups* is a proper subset of *WSetsOn*; the problematic parameters *cardinal_spec* and elements_spec2w are missing.

We use this approach to be as noninvasive as possible. If we had the liberty to modify the existing MSet library, it might be better to define WSetsOnWithDups as below and define

WSetOn by adding the two extra parameters. Module Type WSETSONWITHDUPS (E : DECIDABLETYPE).

Include WOPS E.

```
Parameter In : elt → t → Prop.
Declare Instance In_compat : Proper (E.eq==>eq==>iff) In.

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) s := ∃ x, In x s ∧ P x.

Notation "s [=] t" := (Equal s t) (at level 70, no associativity).
Notation "s [<=] t" := (Subset s t) (at level 70, no associativity).

Definition eq : t → t → Prop := Equal.
```

Include ISEQ. eq is obviously an equivalence, for subtyping only Include HASEQDEC.

```
Section Spec.
Variable s s': t.
Variable x y : elt.
Variable f : elt → bool.
Notation compatb := (Proper (E.eq==>Logic.eq)) (only parsing).

Parameter mem_spec : mem x s = true ↔ In x s.
Parameter equal_spec : equal s s' = true ↔ s[=]s'.
Parameter subset_spec : subset s s' = true ↔ s[<=]s'.
Parameter empty_spec : Empty empty.
Parameter is_empty_spec : is_empty s = true ↔ Empty s.
Parameter add_spec : In y (add x s) ↔ E.eq y x ∨ In y s.
Parameter remove_spec : In y (remove x s) ↔ In y s ∧ ¬E.eq y x.
Parameter singleton_spec : In y (singleton x) ↔ E.eq y x.
Parameter union_spec : In x (union s s') ↔ In x s ∨ In x s'.
Parameter inter_spec : In x (inter s s') ↔ In x s ∧ In x s'.
Parameter diff_spec : In x (diff s s') ↔ In x s ∧ ¬In x s'.
Parameter fold_spec : ∀ (A : Type) (i : A) (f : elt → A → A),
  fold f s i = fold_left (flip f) (elements s) i.
Parameter filter_spec : compatb f →
  (In x (filter f s) ↔ In x s ∧ f x = true).
Parameter for_all_spec : compatb f →
  (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
Parameter exists_spec : compatb f →
  (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
Parameter partition_spec1 : compatb f →
  fst (partition f s) [=] filter f s.
```

Parameter *partition_spec2* : compatb f →
  snd (*partition* f s) [=] *filter* (fun x ⇒ negb (f x)) s.
Parameter *elements_spec1* : **InA** *E.eq* x (*elements* s) ↔ *In* x s.
Parameter *choose_spec1* : *choose* s = Some x → *In* x s.
Parameter *choose_spec2* : *choose* s = None → Empty s.

End Spec.

End WSetsOnWithDups.

5.1.2 WSetsOnWithDupsExtra

WSetsOnWithDupsExtra introduces elements_dist in order to specify cardinality and in order to get an operation similar to the original behavior of elements. Module Type WSetsOnWithDupsExtra (E : DecidableType).
Include WSetsOnWithDups E.

An operation for getting an elements list without duplicates Parameter *elements_dist* : *t* → **list** elt.

Parameter *elements_dist_spec1* : ∀ x s, **InA** *E.eq* x (*elements_dist* s) ↔ **InA** *E.eq* x (*elements* s).

Parameter *elements_dist_spec2w* : ∀ s, **NoDupA** *E.eq* (*elements_dist* s).

Cardinality can then be specified with respect to elements_dist. Parameter *cardinal_spec* : ∀ s, *cardinal* s = length (*elements_dist* s).
End WSetsOnWithDupsExtra.

5.1.3 WSetOn to WSetsOnWithDupsExtra

Since WSetsOnWithDupsExtra is morally a weaker version of *WSetsOn* that allows the fold operation to visit elements multiple time, we can write then following conversion.

Module WSetsOn_TO_WSetsOnWithDupsExtra (E : DecidableType) (W : WSetsOn E) <:
  WSetsOnWithDupsExtra E.

Include W.

Definition elements_dist := *W.elements*.

Lemma elements_dist_spec1 : ∀ x s, **InA** *E.eq* x (elements_dist s) ↔ **InA** *E.eq* x (*elements* s).

Proof. *done*. Qed.

Lemma elements_dist_spec2w : ∀ s, **NoDupA** *E.eq* (elements_dist s).
Proof. apply *elements_spec2w*. Qed.

End WSetsOn_TO_WSetsOnWithDupsExtra.