

# Contents

<b>1</b>	<b>Library <code>MSetsExtra.MSetWithDups</code></b>	<b>2</b>
1.1	Signature for weak sets which may contain duplicates . . . . .	2
1.1.1	<code>WSetsOnWithDups</code> . . . . .	2
1.1.2	<code>WSetsOnWithDupsExtra</code> . . . . .	4
1.1.3	<code>WSetOn</code> to <code>WSetsOnWithDupsExtra</code> . . . . .	4
<b>2</b>	<b>Library <code>MSetsExtra.MSetListWithDups</code></b>	<b>5</b>
2.1	Weak sets implemented as lists with duplicates . . . . .	5
2.1.1	Removing duplicates from sorted lists . . . . .	5
2.1.2	Operations Module . . . . .	12
2.1.3	Main Module . . . . .	13
2.1.4	Proofs of set operation specifications. . . . .	14
<b>3</b>	<b>Library <code>MSetsExtra.MSetIntervals</code></b>	<b>23</b>
3.1	Weak sets implemented by interval lists . . . . .	23
3.1.1	Auxiliary stuff . . . . .	23
3.1.2	Encoding Elements . . . . .	26
3.1.3	Set Operations . . . . .	27
3.1.4	Raw Module . . . . .	36
3.1.5	Main Module . . . . .	149
3.1.6	Instantiations . . . . .	154
<b>4</b>	<b>Library <code>MSetsExtra.MSetFoldWithAbort</code></b>	<b>156</b>
4.1	Fold with abort for sets . . . . .	156
4.1.1	Fold With Abort Operations . . . . .	156
4.1.2	Derived operations . . . . .	162
4.1.3	Modules Types For Sets with Fold with Abort . . . . .	172
4.1.4	Implementations . . . . .	173
4.1.5	Sorted Lists Implementation . . . . .	178

# Chapter 1

## Library MSetsExtra.MSetWithDups

### 1.1 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 mathcomp.ssreflect.ssreflect.
```

#### 1.1.1 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 ln : elt → t → Prop.`

`Declare Instance ln_compat : Proper (E.eq==>eq==>iff) ln.`

`Definition Equal s s' := ∀ a : elt, ln a s ↔ ln a s'.`

`Definition Subset s s' := ∀ a : elt, ln a s → ln a s'.`

`Definition Empty s := ∀ a : elt, ¬ ln a s.`

`Definition For_all (P : elt → Prop) s := ∀ x, ln x s → P x.`

`Definition Exists (P : elt → Prop) s := ∃ x, ln 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 HASE-QDEC.`

`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 ↔ ln 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 : ln y (add x s) ↔ E.eq y x ∨ ln y s.`

`Parameter remove_spec : ln y (remove x s) ↔ ln y s ∧ ¬E.eq y x.`

`Parameter singleton_spec : ln y (singleton x) ↔ E.eq y x.`

`Parameter union_spec : ln x (union s s') ↔ ln x s ∨ ln x s'.`

`Parameter inter_spec : ln x (inter s s') ↔ ln x s ∧ ln x s'.`

`Parameter diff_spec : ln x (diff s s') ↔ ln x s ∧ ¬ln 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 →`

`(ln x (filter f s) ↔ ln 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.

```

### 1.1.2 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.

### 1.1.3 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.

# Chapter 2

## Library

## MSetExtra.MSetListWithDups

### 2.1 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 mathcomp.ssreflect.ssreflect.
Require Import List OrdersFacts OrdersLists.
Require Import Sorting Permutation.
Require Import MSetWithDups.
```

#### 2.1.1 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\_sortedA :

:

```

  ∀ (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  $\Rightarrow$  {xs}. {
    move  $\Rightarrow$  ..
    split; last split. {
      apply Sorted_cons  $\Rightarrow$  //.
    } {
      apply NoDupA_singleton.
    } {
      done.
    }
  }
} {
  move  $\Rightarrow$  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 :  $\exists$  y' ys',
    remove_dups_from_sortedA (y :: ys) = y' :: ys'  $\wedge$ 
    eqb y y' = true. {
    apply remove_dups_from_sortedA_hd  $\Rightarrow$  //.
  }
  move  $\Rightarrow$  [y'] [ys'] [H_yys'_intro] /eqb_eq_alt H_eq_y_y'.
  move : (IH H_sorted_y_ys).
  rewrite !H_yys'_intro.
  move  $\Rightarrow$  {IH} [IH1] [IH2] IH3.
  case_eq (eqb x y). {
    rewrite eqb_eq_alt  $\Rightarrow$  H_eq_x_y.
    split  $\Rightarrow$  //.
    split  $\Rightarrow$  //.
    move  $\Rightarrow$  x'.
    rewrite InA_cons IH3.
    split; last tauto.
    move  $\Rightarrow$  [] //.
    move  $\Rightarrow$  H_eq_x'_x.
    apply InA_cons_hd.
    apply eq_trans with (y := x)  $\Rightarrow$  //.
    apply eq_trans with (y := y)  $\Rightarrow$  //.
  }
  move  $\Rightarrow$  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.

```

## 2.1.2 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.

### 2.1.3 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.`

## 2.1.4 Proofs of set operation specifications.

Logical predicates      Definition  $\text{In } x (s : t) := \text{SetoidList.InA } E.\text{eq } x s$ .

Instance  $\text{In\_compat} : \text{Proper } (E.\text{eq} ==> \text{eq} ==> \text{iff}) \text{ In}$ .

Proof. repeat red. intros. rewrite  $H \ H0$ . auto. Qed.

Definition  $\text{Equal } s s' := \forall a : \text{elt}, \text{In } a s \leftrightarrow \text{In } a s'$ .

Definition  $\text{Subset } s s' := \forall a : \text{elt}, \text{In } a s \rightarrow \text{In } a s'$ .

Definition  $\text{Empty } s := \forall a : \text{elt}, \neg \text{In } a s$ .

Definition  $\text{For\_all } (P : \text{elt} \rightarrow \text{Prop}) s := \forall x, \text{In } x s \rightarrow P x$ .

Definition  $\text{Exists } (P : \text{elt} \rightarrow \text{Prop}) s := \exists x, \text{In } x s \wedge P x$ .

Notation " $s [=] t$ " :=  $(\text{Equal } s t)$  (at level 70, no associativity).

Notation " $s [<=] t$ " :=  $(\text{Subset } s t)$  (at level 70, no associativity).

Definition  $\text{eq} : t \rightarrow t \rightarrow \text{Prop} := \text{Equal}$ .

Lemma  $\text{eq\_equiv} : \text{Equivalence eq}$ .

Proof.

```

    constructor. {
      done.
    } {
      by constructor; rewrite  $H$ .
    } {
      by constructor; rewrite  $H \ H0$ .
    }

```

Qed.

Specifications of set operators

Notation  $\text{compatb} := (\text{Proper } (E.\text{eq} ==> \text{Logic.eq}))$  (*only parsing*).

Lemma  $\text{mem\_spec} : \forall s x, \text{mem } x s = \text{true} \leftrightarrow \text{In } x s$ .

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 :  $\forall s s', \text{subset } s s' = \text{true} \leftrightarrow s [=] s'$ .

Proof.

```

  intros s s'.
  rewrite /subset forallb_forall /Subset /In.
  split. {
    move  $\Rightarrow$   $H \ z \ / \text{InA\_alt} \ [] \ x \ [H\_z\_eq] \ H\_in$ .
    move : ( $H \_ H\_in$ ).
    rewrite mem_spec.
    setoid_replace z with x  $\Rightarrow$  //.
  } {
    move  $\Rightarrow$   $H \ z \ H\_in$ .
    rewrite mem_spec.
    apply H, In_InA  $\Rightarrow$  //.
    apply E.eq_equiv.
  }
}

```

Qed.

Lemma equal\_spec :  $\forall s s', \text{equal } s s' = \text{true} \leftrightarrow s [=] s'$ .

Proof.

```

  intros s s'.
  rewrite /Equal /equal Bool.andb_true_iff !subset_spec /Subset.
  split. {
    move  $\Rightarrow$  [ $H1 \ H2$ ] a.
    split.
      - by apply H1.
      - by apply H2.
  } {
    move  $\Rightarrow$  H.
    split; move  $\Rightarrow$  a; rewrite H //.
  }
}

```

Qed.

Lemma eq\_dec :  $\forall x y : t, \{\text{eq } x y\} + \{\neg \text{eq } x y\}$ .

Proof.

```

  intros x y.
  change ( $\{\text{Equal } x y\} + \{\neg \text{Equal } x y\}$ ).
  destruct (equal x y) eqn:H; [left|right];
  rewrite  $\leftarrow$  equal_spec; congruence.

```

Qed.

Lemma empty\_spec : Empty empty.

Proof. rewrite /Empty /empty /ln. move  $\Rightarrow a$  /lnA\_nil //. Qed.

Lemma is\_empty\_spec :  $\forall s, \text{is\_empty } s = \text{true} \leftrightarrow \text{Empty } s$ .

Proof.

```
rewrite /is_empty /Empty /ln.
case; split  $\Rightarrow$  //. {
  move  $\Rightarrow$  - a.
  rewrite lnA_nil //.
} {
  move  $\Rightarrow$  H; contradiction (H a).
  apply lnA_cons_hd.
  apply Equivalence_Reflexive.
}
```

Qed.

Lemma add\_spec :  $\forall s \ x \ y, \text{ln } y \ (\text{add } x \ s) \leftrightarrow E.\text{eq } y \ x \vee \text{ln } y \ s$ .

Proof.

```
intros s x y.
rewrite /add /ln lnA_cons //.
```

Qed.

Lemma singleton\_spec :  $\forall x \ y, \text{ln } y \ (\text{singleton } x) \leftrightarrow E.\text{eq } y \ x$ .

Proof.

```
intros x y.
rewrite /singleton /ln lnA_cons.
split. {
  move  $\Rightarrow$  [] // /lnA_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 :  $\forall s \ acc \ x \ f, \text{compatb } f \rightarrow$   
 $(\text{ln } x \ (\text{rev\_filter\_aux } acc \ f \ s) \leftrightarrow (\text{ln } x \ s \wedge f \ x = \text{true}) \vee (\text{ln } x \ acc))$ .

Proof.

```
intros s acc x f H_compat.
move : x acc.
induction s as [y s' IH]. {
  intros x acc.
  rewrite /rev_filter_aux /ln lnA_nil.
  tauto.
} {
```



```

intros x acc.
rewrite /= IH /ln.
case_eq (f y) => H_fy; rewrite !lnA_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 →
  (ln x (filter f s) ↔ ln x s ∧ f x = true).
Proof.
  intros s x f H_compat.
  rewrite /filter /rev_filter rev_filter_aux_spec /ln lnA_nil.
  tauto.
Qed.

Lemma remove_spec : ∀ s x y, ln y (remove x s) ↔ ln 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 /ln lnA_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 :  $\forall s s' x, \text{In } x (\text{union } s s') \leftrightarrow \text{In } x s \vee \text{In } x s'$ .
Proof.
  intros s s' x.
  rewrite /union /In rev_append_rev InA_app_iff InA_rev; tauto.
Qed.

Lemma inter_spec :  $\forall s s' x, \text{In } x (\text{inter } s s') \leftrightarrow \text{In } x s \wedge \text{In } x s'$ .
Proof.
  intros s s' x.
  have H_compat : compatb (fun y : elt  $\Rightarrow$  mem y s'). {
    repeat red; intros.
    suff : ( mem x0 s' = true  $\leftrightarrow$  mem y s' = true ). {
      case (mem y s'), (mem x0 s'); intuition.
    }
    rewrite !mem_spec /In.
    setoid_replace x0 with y  $\Rightarrow$  //.
  }
  rewrite /inter rev_filter_aux_spec mem_spec /In InA_nil.
  tauto.
Qed.

Lemma fold_spec :  $\forall s (A : \text{Type}) (i : A) (f : \text{elt} \rightarrow A \rightarrow A),$ 
  fold f s i = fold_left (flip f) (elements s) i.
Proof. done. Qed.

Lemma elements_spec1 :  $\forall s x, \text{InA } E.\text{eq } x (\text{elements } s) \leftrightarrow \text{In } x s$ .
Proof.
  intros s x.
  rewrite /elements /In //.
Qed.

Lemma diff_spec :  $\forall s s' x, \text{In } x (\text{diff } s s') \leftrightarrow \text{In } x s \wedge \neg \text{In } x s'$ .
Proof.
  intros s s' x.
  rewrite /diff fold_spec -(elements_spec1 s').
  move : s.
  induction (elements s') as [| y ys IH]  $\Rightarrow$  s. {
    rewrite InA_nil /=; tauto.
  } {
    rewrite /= IH InA_cons /flip remove_spec.
  }

```

```

    tauto.
  }
Qed.

Lemma cardinal_spec :  $\forall s$ , cardinal  $s$  = length (elements_dist  $s$ ).
Proof. rewrite /cardinal //. Qed.

Lemma for_all_spec :  $\forall s f$ , compatb  $f \rightarrow$ 
  ( $\text{for\_all } f \ s = \text{true} \leftrightarrow \text{For\_all } (\text{fun } x \Rightarrow f \ x = \text{true}) \ s$ ).
Proof.
  intros  $s f$   $H\_compat$ .
  rewrite /For_all.
  induction  $s$  as [|  $x$   $xs$   $IH$ ]. {
    rewrite /= /ln.
    split  $\Rightarrow$  //.
    move  $\Rightarrow$  -  $x$  /lnA_nil //.
  } {
    rewrite /=.
    case_eq ( $f \ x$ )  $\Rightarrow H\_fx$ . {
      rewrite  $IH$ .
      split. {
        move  $\Rightarrow H \ x' /lnA_cons []$ . {
          move  $\Rightarrow \rightarrow$  //.
        } {
          apply  $H$ .
        }
      }
    } {
      move  $\Rightarrow H \ x' H\_in$ .
      apply  $H$ .
      apply lnA_cons.
      by right.
    }
  } {
    split  $\Rightarrow$  //.
    move  $\Rightarrow H$ .
    suff :  $f \ x = \text{true}$ . {
      rewrite  $H\_fx$  //.
    }
    apply  $H$ .
    apply lnA_cons_hd.
    done.
  }
}
Qed.

```

Lemma exists\_spec :  $\forall s f, \text{compatb } f \rightarrow$   
 $(\text{exists\_} f \ s = \text{true} \leftrightarrow \text{Exists } (\text{fun } x \Rightarrow f \ x = \text{true}) \ s).$

Proof.

```

intros s f H_compat.
rewrite /Exists.
induction s as [| x xs IH]. {
  rewrite /= /ln.
  split  $\Rightarrow$  //.
  move  $\Rightarrow$  [x] [] /lnA_nil //.
} {
  rewrite /=.
  case_eq (f x)  $\Rightarrow$  H_fx. {
    split  $\Rightarrow$  // ..
     $\exists$  x.
    split  $\Rightarrow$  //.
    apply lnA_cons_hd.
    done.
  } {
    rewrite IH.
    split. {
      move  $\Rightarrow$  [x'] [H_in] H_fx'.
       $\exists$  x'.
      split  $\Rightarrow$  //.
      apply lnA_cons.
      by right.
    } {
      move  $\Rightarrow$  [x'] [] /lnA_cons []. {
        move  $\Rightarrow$   $\rightarrow$ .
        rewrite H_fx //.
      } {
        by  $\exists$  x'.
      }
    }
  }
}

```

Qed.

Lemma partition\_aux\_spec :  $\forall a1 \ a2 \ s \ f,$   
 $(\text{partition\_aux } a1 \ a2 \ f \ s = (\text{rev\_filter\_aux } a1 \ f \ s, \text{rev\_filter\_aux } a2 \ (\text{fun } x \Rightarrow \text{negb } (f \ x)) \ s)).$

Proof.

```

move  $\Rightarrow$  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 :  $\forall s f$ , compatb  $f \rightarrow$ 
  fst (partition  $f s$ ) [=] filter  $f s$ .
Proof.
  move => s f -.
  rewrite /partition partition_aux_spec /fst /filter /rev_filter //.
Qed.

Lemma partition_spec2 :  $\forall s f$ , compatb  $f \rightarrow$ 
  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 :  $\forall s x$ , choose  $s = \text{Some } x \rightarrow \text{In } x s$ .
Proof.
  move => [| // y s' x [->].
  rewrite /In.
  apply InA_cons_hd.
  apply Equivalence_Reflexive.
Qed.

Lemma choose_spec2 :  $\forall s$ , choose  $s = \text{None} \rightarrow \text{Empty } s$ .
Proof. move => [| // - a. rewrite /In InA_nil //. Qed.

Lemma elements_dist_spec_full :
   $\forall s$ ,
    Sorted  $E.lt$  (elements_dist  $s$ )  $\wedge$ 
    NoDupA  $E.eq$  (elements_dist  $s$ )  $\wedge$ 
    ( $\forall x$ , InA  $E.eq x$  (elements_dist  $s$ )  $\leftrightarrow$  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 :  $\forall x s, \text{InA } E.eq x (\text{elements\_dist } s) \leftrightarrow \text{InA } E.eq x (\text{elements } s).$

Proof. intros; apply elements\_dist\_spec\_full. Qed.

Lemma elements\_dist\_spec2w :  $\forall s, \text{NoDupA } E.eq (\text{elements\_dist } s).$

Proof. intros; apply elements\_dist\_spec\_full. Qed.

End MAKE.

# Chapter 3

## Library MSetExtra.MSetIntervals

### 3.1 Weak sets implemented by interval lists

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

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

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

#### 3.1.1 Auxiliary stuff

Simple auxiliary lemmata Lemma  $Z\_le\_add\_r : \forall (z : \mathbb{Z}) (n : \mathbb{N}),$   
 $z \leq z + Z.of\_N n.$

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 :  $\forall (z : \mathbf{Z}) (n : \mathbf{N}),$   
  $(n \neq 0) \% N \rightarrow$   
  $z < z + \mathbf{Z.of\_N} \ n.$

Proof.

move  $\Rightarrow z \ n \ H\_neq\_0.$   
 suff :  $(z + \mathbf{Z.of\_N} \ 0 < z + \mathbf{Z.of\_N} \ n).$  {  
 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 :  $\forall y1 \ y2 \ c,$   
  $y1 < y2 \rightarrow$   
  $y1 \leq y2 + \mathbf{Z.of\_N} \ c.$

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 :  $\forall (x \ y : \mathbf{Z}),$   
  $y < x \rightarrow$   
  $\mathbf{Z.to\_N} \ (x - y) \neq 0 \% N.$

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  $\Rightarrow$  //.  
 by apply Z.lt\_le\_incl.  
 }  
 by apply Z.lt\_0\_sub.

Qed.

Lemma add\_add\_sub\_eq :  $\forall (x \ y : \mathbf{Z}), (x + (y - x) = y).$

Proof.

intros x y.  
 rewrite Z.add\_sub\_assoc  $\Rightarrow$  //.  
 rewrite Z.add\_sub\_swap Z.sub\_diag Z.add\_0\_l //.



Qed.

Lemma NoDupA\_map {A B} :  $\forall (eqA : A \rightarrow A \rightarrow \text{Prop}) (eqB : B \rightarrow B \rightarrow \text{Prop}) (f : A \rightarrow B) l,$

**NoDupA** eqA l  $\rightarrow$   
 $(\forall x1\ x2, \text{List.In } x1\ l \rightarrow \text{List.In } x2\ l \rightarrow$   
 $eqB (f\ x1) (f\ x2) \rightarrow eqA\ x1\ x2) \rightarrow$   
**NoDupA** eqB (map f l).

Proof.

```

intros eqA eqB f.
induction l as [| x xs IH]. {
  move  $\Rightarrow$  _ _; rewrite /=.
  apply NoDupA_nil.
} {
  move  $\Rightarrow$  H_pre H_eqA_impl.
  have [H_nin_x H_no_dup_xs] :  $\neg \text{InA } eqA\ x\ xs \wedge \text{NoDupA } eqA\ xs$ . {
    by inversion_clear H_pre.
  }
  simpl.
  apply NoDupA_cons; last first. {
    apply IH  $\Rightarrow$  //.
    intros x1 x2 H_in_x1 H_in_x2 H_eqB.
    apply H_eqA_impl  $\Rightarrow$  /=; by right.
  }
  move  $\Rightarrow$  H_in_map; apply H_nin_x.
  move : H_in_map.
  rewrite !InA_alt  $\Rightarrow$  [[y] [H_eqB_y]].
  rewrite in_map_iff  $\Rightarrow$  [[y'] [H_y_eq] H_y'_in].
  subst.
   $\exists y'$ .
  split  $\Rightarrow$  //.
  apply H_eqA_impl  $\Rightarrow$  //. {
    by simpl; left.
  } {
    by simpl; right.
  }
}

```

Qed.

## rev\_map

rev\_map is used for efficiency. Fixpoint rev\_map\_aux {A B} (f : A  $\rightarrow$  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`  $\{A\ B\}$   $(f : A \rightarrow B)$   $(l : \text{list } A) : \text{list } B := \text{rev\_map\_aux } f \text{ nil } l$ .

Lemmata about `rev_map` Lemma `rev_map_aux_alt_def`  $\{A\ B\} : \forall (f : A \rightarrow B) l \text{ acc},$   
`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`  $\{A\ B\} : \forall (f : A \rightarrow B) l,$   
`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.

### 3.1.2 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 `ELEMENT_ENCODE`. It provides encode and decode function.

Module Type `ELEMENT_ENCODE`.

Declare Module `E` : `ORDEREDTYPE`.

Parameter `encode` :  $E.t \rightarrow \mathbf{Z}$ .

Parameter `decode` :  $\mathbf{Z} \rightarrow E.t$ .

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`:  $\forall (e : E.t),$   
`decode (encode e) = e.`

Encoding is compatible with the equality of elements. Axiom `encode_eq`:  $\forall (e1\ e2 : E.t),$

`(Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.`

Encoding is compatible with the order of elements. Axiom `encode_lt`:  $\forall (e1\ e2 : E.t),$

**(Z.lt (encode e1) (encode e2)) ↔ E.lt e1 e2.**

End ELEMENTENCODE.

### 3.1.3 Set Operations

We represent sets of  $\mathbb{Z}$  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  $\mathbb{Z}$ .

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
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
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
end.

```

Auxiliary functions for inserting at front and merging intervals      Definition merge\_interval\_size  
 $(x1 : \mathbf{Z}) (c1 : \mathbf{N}) (x2 : \mathbf{Z}) (c2 : \mathbf{N}) : \mathbf{N} :=$   
 $(\mathbf{N.max} \ c1 \ (\mathbf{Z.to\_N} \ (x2 + \mathbf{Z.of\_N} \ c2 - x1)))$ .

Fixpoint insert\_interval\_begin  $(x : \mathbf{Z}) (c : \mathbf{N}) (l : \mathbf{t}) :=$   
 $\text{match } l \text{ with}$   
 $| \text{nil} \Rightarrow (x, c) :: \text{nil}$   
 $| (y, c') :: l' \Rightarrow$   
 $\quad \text{match } (\mathbf{Z.compare} \ (x + \mathbf{Z.of\_N} \ c) \ y) \text{ with}$   
 $\quad | \text{Lt} \Rightarrow (x, c) :: l$   
 $\quad | \text{Eq} \Rightarrow (x, (c+c')\%N) :: l'$   
 $\quad | \text{Gt} \Rightarrow \text{insert\_interval\_begin } x \ (\text{merge\_interval\_size } x \ c \ y \ c') \ l'$   
 $\quad \text{end}$   
 $\text{end.}$

adding an element needs to be defined carefully again in order to generate efficient code  
 Fixpoint addZ\_aux  $(acc : \text{list } (\mathbf{Z} \times \mathbf{N})) (x : \mathbf{Z}) (s : \mathbf{t}) :=$

$\text{match } s \text{ with}$   
 $| \text{nil} \Rightarrow \text{List.rev'} \ ((x, (1\%N)) :: acc)$   
 $| (y, c) :: l \Rightarrow$   
 $\quad \text{match } (\text{interval\_1\_compare } x \ (y, c)) \text{ with}$   
 $\quad | \text{ICR\_before} \Rightarrow \text{List.rev\_append} \ ((x, (1\%N)) :: acc) \ s$   
 $\quad | \text{ICR\_before\_touch} \Rightarrow \text{List.rev\_append} \ ((x, \mathbf{N.succ} \ c) :: acc) \ l$   
 $\quad | \text{ICR\_after} \Rightarrow \text{addZ\_aux} \ ((y, c) :: acc) \ x \ l$   
 $\quad | \text{ICR\_after\_touch} \Rightarrow \text{List.rev\_append} \ acc \ (\text{insert\_interval\_begin } y \ (\mathbf{N.succ} \ c) \ l)$   
 $\quad | \_ \Rightarrow \text{List.rev\_append} \ ((y, c) :: acc) \ l$   
 $\quad \text{end}$   
 $\text{end.}$

Definition addZ  $x \ s := \text{addZ\_aux nil } x \ s$ .

Definition add  $x \ s := \text{addZ } (\text{Enc.encode } x) \ s$ .

add\_list is a simple extension to add many elements. This is used to define the function from\_elements.      Definition add\_list  $(l : \text{list elt}) (s : \mathbf{t}) : \mathbf{t} :=$

$\text{List.fold\_left} \ (\text{fun } s \ x \Rightarrow \text{add } x \ s) \ l \ s$ .

Definition from\_elements  $(l : \text{list elt}) : \mathbf{t} := \text{add\_list } l \ \text{empty}$ .

singleton is trivial to define      Definition singleton  $(x : \text{elt}) : \mathbf{t} := (\text{Enc.encode } x, 1\%N) :: \text{nil}$ .

Lemma singleton\_alt\_def :  $\forall x, \text{singleton } x = \text{add } x \ \text{empty}$ .

Proof. by []. Qed.

removing needs to be done with code extraction in mind again.      Definition insert\_intervalZ\_guarded  $(x : \mathbf{Z}) (c : \mathbf{N}) s :=$

$\text{if } (\mathbf{N.eqb} \ c \ 0) \text{ then } s \text{ else } (x, c) :: s$ .

Fixpoint removeZ\_aux  $(acc : \text{list } (\mathbf{Z} \times \mathbf{N})) (x : \mathbf{Z}) (s : \mathbf{t}) : \mathbf{t} :=$

```

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.
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.
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
    end
  end
end.

```



```

    | 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.

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.

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**  $\Rightarrow$  **None**  
 | ( $x, \_$ ) ::  $\_ \Rightarrow$  **Some**  $x$   
 end.

Definition min\_elt ( $s : t$ ) : **option elt** :=  
 match (min\_eltZ  $s$ ) with  
 | **None**  $\Rightarrow$  **None**  
 | **Some**  $x \Rightarrow$  **Some** (*Enc.decode*  $x$ )  
 end.

Definition choose := min\_elt.

Fixpoint max\_eltZ ( $s : t$ ) : **option Z** :=  
 match  $s$  with  
 | **nil**  $\Rightarrow$  **None**  
 | ( $x, c$ ) :: **nil**  $\Rightarrow$  **Some** (**Z.pred** ( $x + \mathbf{Z.of\_N}$   $c$ ))  
 | ( $x, \_$ ) ::  $s' \Rightarrow$  max\_eltZ  $s'$   
 end.

Definition max\_elt ( $s : t$ ) : **option elt** :=  
 match (max\_eltZ  $s$ ) with  
 | **None**  $\Rightarrow$  **None**  
 | **Some**  $x \Rightarrow$  **Some** (*Enc.decode*  $x$ )  
 end.

End OPS.

### 3.1.4 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 ( $l : \mathbf{list\ Z}$ ) : Prop :=  
 ( $\forall x, \mathbf{List.In}$   $x\ l \rightarrow \exists e, \mathbf{Enc.encode}\ e = x$ ).

Definition interval\_list\_elements\_greater ( $x : \mathbf{Z}$ ) ( $l : t$ ) : **bool** :=  
 match  $l$  with  
 | **nil**  $\Rightarrow$  **true**  
 | ( $y, \_$ ) ::  $\_ \Rightarrow \mathbf{Z.ltb}$   $x\ y$

```

    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 lsOk s := (interval_list_invariant s = true ∧ is_encoded_elems_list (elementsZ
s)).

```

## Defining notations

Section ForNotations.

```

Class Ok (s:t) : Prop := ok : lsOk s.
Hint Resolve @ok.
Hint Unfold Ok.
Instance lsOk_Ok s '(Hs : lsOk s) : Ok s := { ok := Hs }.
Definition ln x s := (SetoidList.InA Enc.E.eq x (elements s)).
Definition lnZ x s := (List.In x (elementsZ s)).
Definition Equal s s' := ∀ a : elt, ln a s ↔ ln a s'.
Definition Subset s s' := ∀ a : elt, ln a s → ln a s'.
Definition Empty s := ∀ a : elt, ¬ ln a s.
Definition For_all (P : elt → Prop) s := ∀ x, ln x s → P x.
Definition Exists (P : elt → Prop) (s : t) := ∃ x, ln 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 :  $\forall x$ ,  
 elementsZ\_single  $x$  ( $0\%N$ ) = nil.

Proof. *done*. Qed.

Lemma elementsZ\_single\_succ :  $\forall x\ c$ ,  
 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 :  $\forall x\ c2\ c1$ ,  
 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 :  $\forall x\ c$ ,  
 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 :  $\forall c\ y\ x$ ,  
 $List.In\ y\ (elementsZ\_single\ x\ c) \leftrightarrow$   
 $(x \leq y) \wedge (y < (x+Z.of\_N\ c))$ .

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 :  $\forall y x$ ,  
 $\text{List.In } y \text{ (elementsZ\_single } x \text{ (1\%N))} \leftrightarrow (x = y)$ .

Proof.

```

  intros y x.
  rewrite In_elementsZ_single /= Z.add_1_r Z.lt_succ_r.
  omega.

```

Qed.

Lemma length\_elementsZ\_single :  $\forall cx x$ ,  
 $\text{length (elementsZ\_single } x \text{ } cx) = \text{N.to\_nat } cx$ .

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} :  
 $\forall f c (acc : A) x H1 H2$ ,  
 $\text{fold\_elementsZ\_aux } f \text{ } acc \text{ } x \text{ } c \text{ } H1 = \text{fold\_elementsZ\_aux } f \text{ } acc \text{ } x \text{ } c \text{ } H2$ .

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  $\{A\} : \forall f (acc : A) x p,$   
 $\text{fold\_elementsZ\_single } f \text{ acc } x \text{ (N.pos } p) =$   
 $\text{match } f \text{ acc } x \text{ with}$   
 $| \text{Some } acc' \Rightarrow$   
 $\text{fold\_elementsZ\_single } f \text{ acc' (Z.succ } x)$   
 $\text{(N.pred (N.pos } p))$   
 $| \text{None} \Rightarrow (\text{true}, acc)$   
 $\text{end.}$

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  $\{A\} : \forall f (acc : A) x,$   
 $\text{fold\_elementsZ\_single } f \text{ acc } x \text{ (0\%N)} = (\text{false}, acc).$

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 :  $\forall (s : t) \text{ base},$   
 $(\text{snd } (\text{fold\_elementsZ } (\text{fun } (l : \text{list } Z) (x : Z) \Rightarrow \text{Some } (x :: l)) \text{ base } s)) =$   
 $\text{elementsZ } s ++ \text{base}.$

Proof.

```
induction s as [| [y1 c1] s' IH]  $\Rightarrow$  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} :  $\forall f s (\text{acc} : A),$   
 $\text{fold\_elementsZ } f \text{ acc } s =$   
 $\text{fold\_opt } f \text{ acc } (\text{rev } (\text{elementsZ } s)).$

Proof.

```
intro f.
induction s as [| [y1 c1] s' IH]  $\Rightarrow$  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  $\Rightarrow$  [] //.
}
```

Qed.

Lemma elementsZ\_cons :  $\forall x c s, \text{elementsZ } ((x, c) :: s) : t =$   
 $((\text{elementsZ } s) ++ (\text{List.rev } (\text{elementsZ\_single } x c))).$

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 :  $\forall x c s, \text{elements } ((x, c) :: s) : t =$ 
     $((\text{elements\_single } x c) ++ \text{elements } s).$ 
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 :  $\forall (s1 s2 : t), \text{elementsZ } (s1 ++ s2) =$ 
     $((\text{elementsZ } s2) ++ (\text{elementsZ } s1)).$ 
Proof.
    induction s1 as [| [x1 c1] s1 IH1]. {
        move  $\Rightarrow s2$ .
        rewrite elementsZ_nil app_nil_r //.
    }
    move  $\Rightarrow s2$ .
    rewrite -app_comm_cons !elementsZ_cons IH1 -app_assoc //.
Qed.

Lemma lnZ_nil :  $\forall y, \text{lnZ } y \text{ nil} \leftrightarrow \text{False}.$ 
Proof.
    intro y.
    done.
Qed.

Lemma lnZ_cons :  $\forall y x c s, \text{lnZ } y ((x, c) :: s) : t \leftrightarrow$ 
     $\text{List.In } y (\text{elementsZ\_single } x c) \vee \text{lnZ } y s.$ 
Proof.
    intros y x c s.
    rewrite /lnZ elementsZ_cons in_app_iff -in_rev.
    firstorder.
Qed.

Lemma lnZ_app :  $\forall s1 s2 y,$ 
     $\text{lnZ } y (s1 ++ s2) \leftrightarrow \text{lnZ } y s1 \vee \text{lnZ } y s2.$ 
Proof.
    intros s1 s2 y.
    rewrite /lnZ elementsZ_app in_app_iff.
    tauto.
Qed.

Lemma lnZ_rev :  $\forall s y,$ 
     $\text{lnZ } y (\text{List.rev } s) \leftrightarrow \text{lnZ } y s.$ 
Proof.

```

```

intros s x.
rewrite /lnZ.
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 ln_elementsZ_single_dec :  $\forall y x c,$ 
  {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 ln_elementsZ_single; tauto.
  }
  case (Z.lt_dec y (x + Z.of_N c)); last first. {
    right; rewrite ln_elementsZ_single; tauto.
  }
  left; rewrite ln_elementsZ_single; tauto.
Qed.

Lemma lnZ_dec :  $\forall y s,$ 
  {lnZ y s} + {¬lnZ y s}.
Proof.
  intros y.
  induction s as [| [x c] s IH]. {
    by right.
  }
  move : IH  $\Rightarrow$  [| IH]. {
    by left; rewrite lnZ_cons; right.
  }
  case (ln_elementsZ_single_dec y x c). {
    by left; rewrite lnZ_cons; left.
  } {
    by right; rewrite lnZ_cons; tauto.
  }
Qed.

Lemma ln_elementsZ_single_hd :  $\forall (c : \mathbf{N}) x, (c \neq 0) \% N \rightarrow \text{List.In } x \text{ (elementsZ\_single } x$ 
 $c)$ .
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` :  $\forall (y1 : \mathbf{Z}) (c1 : \mathbf{N}) (y2 : \mathbf{Z}) (c2 : \mathbf{N})$ ,

$(c1 \neq 0\%N) \vee (c2 \neq 0\%N) \rightarrow$

`interval_compare`  $(y2, c2) (y1, c1) =$

`match` (`interval_compare`  $(y1, c1) (y2, c2)$ ) `with`

| `ICR_before`  $\Rightarrow$  `ICR_after`

| `ICR_before_touch`  $\Rightarrow$  `ICR_after_touch`

| `ICR_after`  $\Rightarrow$  `ICR_before`

| `ICR_after_touch`  $\Rightarrow$  `ICR_before_touch`

| `ICR_equal`  $\Rightarrow$  `ICR_equal`

| `ICR_overlap_before`  $\Rightarrow$  `ICR_overlap_after`

| `ICR_overlap_after`  $\Rightarrow$  `ICR_overlap_before`

| `ICR_subsume_1`  $\Rightarrow$  `ICR_subsume_2`

| `ICR_subsume_2`  $\Rightarrow$  `ICR_subsume_1`

`end`.

Proof.

`intros`  $y1\ c1\ y2\ c2\ H\_c12\_neq\_0$ .

`rewrite` `/interval_compare`.

`move` :  $(\mathbf{Z.compare\_antisym}\ y1\ y2) \Rightarrow \rightarrow$ .

`move` :  $(\mathbf{Z.compare\_antisym}\ (y1 + \mathbf{Z.of\_N}\ c1)\ (y2 + \mathbf{Z.of\_N}\ c2)) \Rightarrow \rightarrow$ .

`have`  $H\_suff : y1 + \mathbf{Z.of\_N}\ c1 \leq y2 \rightarrow y2 + \mathbf{Z.of\_N}\ c2 \leq y1 \rightarrow \mathbf{False}$ . {

`move`  $\Rightarrow H1\ H2$ .

`case`  $H\_c12\_neq\_0 \Rightarrow H\_c\_neq\_0$ . {

$suff : (y1 + \mathbf{Z.of\_N}\ c1 \leq y1)$ . {

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`.

} {

$suff : (y2 + \mathbf{Z.of\_N}\ c2 \leq y2)$ . {

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 :  $\forall (y : \mathbb{Z}) (i : (\mathbb{Z} \times \mathbb{N}))$ ,
  interval_1_compare y i = match (interval_compare (y, (1%N)) i) with
  | ICR_equal  $\Rightarrow$  ICR_subsume_1
  | ICR_subsume_1  $\Rightarrow$  ICR_subsume_1
  | ICR_subsume_2  $\Rightarrow$  ICR_subsume_1
  | r  $\Rightarrow$  r
  end.

Proof.
move  $\Rightarrow$  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 :  $\forall (y1 : \mathbb{Z}) (y2 : \mathbb{Z}) (c2 : \mathbb{N})$ ,
  match (interval_1_compare y1 (y2, c2)) with
  | ICR_before  $\Rightarrow$  Z.succ y1 < y2
  | ICR_before_touch  $\Rightarrow$  y2 = Z.succ y1
  | ICR_after  $\Rightarrow$  (y2 + Z.of_N c2) < y1

```



```

| ICR_after_touch  $\Rightarrow (y2 + \text{Z.of\_N } c2) = y1$ 
| ICR_equal  $\Rightarrow \text{False}$ 
| ICR_overlap_before  $\Rightarrow \text{False}$ 
| ICR_overlap_after  $\Rightarrow \text{False}$ 
| ICR_subsume_1  $\Rightarrow (c2 = 0\%N) \vee ((y2 \leq y1) \wedge (y1 < y2 + \text{Z.of\_N } c2))$ 
| ICR_subsume_2  $\Rightarrow \text{False}$ 
end.
Proof.
  intros y1 y2 c2.
  move : (interval_compare_elim y1 (1%N) y2 c2).
  rewrite interval_1_compare_alt_def.
  have H_succ:  $\forall z, z + \text{Z.of\_N } 1 = \text{Z.succ } z$  by done.
  case_eq (interval_compare (y1, 1%N) (y2, c2))  $\Rightarrow H\_comp$ ;
    rewrite ?H_succ ?Z.lt_succ_r ?Z.le_succ_l //. {
      move  $\Rightarrow [H\_lt] [H\_le] \_.$ 
      contradict H_lt.
      by apply Zle_not_lt.
    } {
      move  $\Rightarrow [-] [H\_lt] H\_le.$ 
      contradict H_lt.
      by apply Zle_not_lt.
    } {
      move  $\Rightarrow [->] \leftarrow.$ 
      rewrite ?Z.lt_succ_r.
      right.
      split; apply Z.le_refl.
    } {
      tauto.
    } {
      case (N.zero_or_succ c2). {
        move  $\Rightarrow \rightarrow \_;$  by left.
      } {
        move  $\Rightarrow [c2'] \rightarrow.$ 
        rewrite !N2Z.inj_succ Z.add_succ_r -Z.succ_le_mono Z.le_succ_l.
        move  $\Rightarrow [H\_y1\_le] [H\_le\_y1].$ 
        suff  $\rightarrow : y1 = y2.$  {
          move  $\Rightarrow [] H\_pre;$  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 :  $\forall z x c s$ ,  
 $\text{interval\_list\_elements\_greater } z ((x, c) :: s) = \text{true} \leftrightarrow (z < x)$ .

Proof.

```

  intros z x c s.
  rewrite /.
  apply Z.lt_lt.

```

Qed.

Lemma interval\_list\_elements\_greater\_impl :  $\forall x y s$ ,  
 $(y \leq x) \rightarrow \text{interval\_list\_elements\_greater } x s = \text{true} \rightarrow \text{interval\_list\_elements\_greater } y s = \text{true}$ .

Proof.

```

  intros x y s.
  case s => //.
  move => [z c] s'.
  rewrite /interval_list_elements_greater.
  move => H_y_leq /Z.lt_lt H_x_lt.
  apply Z.lt_lt.
  eapply Z.le_lt_trans; eauto.

```

Qed.

Lemma interval\_list\_invariant\_nil :  $\text{interval\_list\_invariant nil} = \text{true}$ .

Proof.

```

  by [].

```

Qed.

Lemma Ok\_nil :  $\text{Ok nil} \leftrightarrow \text{True}$ .

Proof.

```

  rewrite /Ok /isOk /interval_list_invariant /is_encoded_elems_list //.

```

Qed.

Lemma is\_encoded\_elems\_list\_app :  $\forall l1 l2$ ,  
 $\text{is\_encoded\_elems\_list } (l1 ++ l2) \leftrightarrow (\text{is\_encoded\_elems\_list } l1 \wedge \text{is\_encoded\_elems\_list } l2)$ .

Proof.

```

  intros l1 l2.

```

```

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

Lemma is_encoded_elems_list_rev :  $\forall l$ ,
  is_encoded_elems_list (List.rev l)  $\leftrightarrow$ 
  is_encoded_elems_list l.
Proof.
  intros l.
  rewrite /is_encoded_elems_list.
  split; (
    move  $\Rightarrow$  H x H_in;
    apply H;
    move : H_in;
    rewrite -in_rev  $\Rightarrow$  //
  ).
Qed.

Lemma interval_list_invariant_cons :  $\forall y\ c\ s'$ ,
  interval_list_invariant ((y, c) :: s') = true  $\leftrightarrow$ 
  (interval_list_elements_greater (y+Z.of_N c) s' = true  $\wedge$ 
    ((c  $\neq$  0)%N)  $\wedge$  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  $\Rightarrow$  [| [H_inf] /N.eqb_neq H_c H_s'. tauto.
  } {
    move  $\Rightarrow$  [H_inf] [/N.eqb_neq H_c] H_s'. tauto.
  }
Qed.

Lemma interval_list_invariant_sing :  $\forall x\ c$ ,
  interval_list_invariant ((x, c) :: nil) = true  $\leftrightarrow$  (c  $\neq$  0)%N.
Proof.
  intros x c.
  rewrite interval_list_invariant_cons.
  split; tauto.
Qed.

Lemma Ok_cons :  $\forall y\ c\ s'$ , Ok ((y, c) :: s')  $\leftrightarrow$ 
  (interval_list_elements_greater (y+Z.of_N c) s' = true  $\wedge$  ((c  $\neq$  0)%N)  $\wedge$ 
  is_encoded_elems_list (elementsZ_single y c)  $\wedge$  Ok s').

```

Proof.

```

intros y c s'.
rewrite /Ok /!sOk interval_list_invariant_cons elementsZ_cons is_encoded_elems_list_app
is_encoded_elems_list_rev.
tauto.

```

Qed.

Lemma Nin\_elements\_greater :  $\forall s y,$   
 $\text{interval\_list\_elements\_greater } y \ s = \text{true} \rightarrow$   
 $\text{interval\_list\_invariant } s = \text{true} \rightarrow$   
 $\forall x, x \leq y \rightarrow \sim (\text{InZ } x \ s).$

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 :  
 $\forall x s,$   
 $\text{interval\_list\_elements\_greater } x \ s = \text{true} \rightarrow$   
 $\text{interval\_list\_invariant } s = \text{true} \rightarrow$   
 $\neg (\text{InZ } x \ s).$

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 :  $\forall s y,$   
 $\text{interval\_list\_invariant } s = \text{true} \rightarrow$

(interval\_list\_elements\_greater y s = true  $\leftrightarrow$   
 ( $\forall x, x \leq y \rightarrow \sim (\text{InZ } x \ s)$ )).

Proof.

```

intros s y H_inv.
split. {
  move  $\Rightarrow$  H_gr.
  apply Nin_elements_greater  $\Rightarrow$  //.
} {
  move : H_inv.
  case s as [| [x2 c] s']  $\Rightarrow$  //.
  rewrite interval_list_invariant_cons interval_list_elements_greater_cons.
  move  $\Rightarrow$  [-] [H_c_neq] - H.
  apply Z.nle_gt  $\Rightarrow$  H_ge.
  apply (H x2)  $\Rightarrow$  //.
  rewrite InZ_cons; left.
  apply In_elementsZ_single_hd  $\Rightarrow$  //.
}

```

Qed.

Lemma interval\_list\_elements\_greater\_alt2\_def :  $\forall s y$ ,  
 interval\_list\_invariant s = true  $\rightarrow$

(interval\_list\_elements\_greater y s = true  $\leftrightarrow$   
 ( $\forall x, \text{InZ } x \ s \rightarrow y < x$ )).

Proof.

```

intros s y H.
rewrite interval_list_elements_greater_alt_def //.
firstorder.
apply Z.nle_gt.
move  $\Rightarrow$  H_lt.
eapply H0; eauto.

```

Qed.

Lemma interval\_list\_elements\_greater\_intro :  $\forall s y$ ,  
 interval\_list\_invariant s = true  $\rightarrow$   
 ( $\forall x, \text{InZ } x \ s \rightarrow y < x$ )  $\rightarrow$   
 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 :  $\forall s1 \ s2 \ y$ ,  
 interval\_list\_elements\_greater y (s1 ++ s2) = true  $\rightarrow$

interval\_list\_elements\_greater y s1 = true.

Proof.

intros s1 s2 y.

case s1  $\Rightarrow$  //.

Qed.

Lemma interval\_list\_invariant\_app\_intro :  $\forall$  s1 s2,

interval\_list\_invariant s1 = true  $\rightarrow$

interval\_list\_invariant s2 = true  $\rightarrow$

( $\forall$  (x1 x2 :  $\mathbb{Z}$ ),  $\text{InZ } x1 \ s1 \rightarrow \text{InZ } x2 \ s2 \rightarrow \mathbb{Z}.\text{succ } x1 < x2$ )  $\rightarrow$

interval\_list\_invariant (s1 ++ s2) = true.

Proof.

induction s1 as [| [y1 c1] s1' IH]. {

move  $\Rightarrow$  s2 - //.

} {

move  $\Rightarrow$  s2.

rewrite -app\_comm\_cons !interval\_list\_invariant\_cons.

move  $\Rightarrow$  [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  $\Rightarrow$  - H\_inz\_s2.

rewrite app\_nil\_l.

apply interval\_list\_elements\_greater\_intro  $\Rightarrow$  //.

move  $\Rightarrow$  x H\_x\_in\_s2.

suff H\_inz :  $\text{InZ } (\mathbb{Z}.\text{pred } (y1 + \mathbb{Z}.\text{of\_N } c1)) \ ((y1, c1) :: \text{nil})$ . {

move : (H\_inz\_s2 - - H\_inz H\_x\_in\_s2).

by rewrite  $\mathbb{Z}.\text{succ\_pred}$ .

}

rewrite  $\text{InZ\_cons In\_elementsZ\_single } -\mathbb{Z}.\text{lt\_le\_pred}$ ; left.

split. {

by apply  $\mathbb{Z}.\text{lt\_add\_r}$ .

} {

apply  $\mathbb{Z}.\text{lt\_pred\_l}$ .

}

} {

assumption.

} {

apply IH  $\Rightarrow$  //.

intros x1 x2 H\_in\_x1 H\_in\_x2.

apply H\_inz\_s2  $\Rightarrow$  //.

rewrite  $\text{InZ\_cons}$ ; by right.

}

}

Qed.

Lemma interval\_list\_invariant\_app\_elim :  $\forall s1\ s2,$   
 $\text{interval\_list\_invariant } (s1 ++ s2) = \text{true} \rightarrow$   
 $\text{interval\_list\_invariant } s1 = \text{true} \wedge$   
 $\text{interval\_list\_invariant } s2 = \text{true} \wedge$   
 $(\forall (x1\ x2 : \mathbb{Z}), \text{InZ } x1\ s1 \rightarrow \text{InZ } x2\ s2 \rightarrow \mathbb{Z}.\text{succ } x1 < x2).$

Proof.

move  $\Rightarrow s1\ s2.$   
induction  $s1$  as [| [y1 c1] s1' IH]; first done.  
rewrite -app\_comm\_cons !interval\_list\_invariant\_cons.  
move  $\Rightarrow [H\_gr] [H\_c1\_neq\_0] / IH [H\_inv\_s1'] [H\_inv\_s2] H\_in\_s1'_s2.$   
repeat split; try assumption. {  
move :  $H\_gr.$   
case  $s1'$ ; first done.  
move  $\Rightarrow [y2\ c2] s1''.$   
rewrite interval\_list\_elements\_greater\_cons //.  
} {  
move  $\Rightarrow x1\ x2.$   
rewrite InZ\_cons In\_elementsZ\_single.  
move  $\Rightarrow []$ ; last by apply  $H\_in\_s1'_s2.$   
move  $\Rightarrow []\ H\_y1\_le\ H\_x1\_lt\ H\_x2\_in.$   
move :  $H\_gr.$   
rewrite interval\_list\_elements\_greater\_alt2\_def; last first. {  
by apply interval\_list\_invariant\_app\_intro.  
}  
move  $\Rightarrow H\_in\_s12'.$   
have :  $(y1 + \mathbb{Z}.\text{of\_N } c1 < x2).$  {  
apply  $H\_in\_s12'.$   
rewrite InZ\_app.  
by right.  
}  
move  $\Rightarrow H\_lt\_x2.$   
apply  $\mathbb{Z}.\text{le\_lt\_trans}$  with  $(m := y1 + \mathbb{Z}.\text{of\_N } c1) \Rightarrow //.$   
by apply  $\mathbb{Z}.\text{lt\_le\_succ}.$   
}

Qed.

Lemma interval\_list\_invariant\_app\_iff :  $\forall s1\ s2,$   
 $\text{interval\_list\_invariant } (s1 ++ s2) = \text{true} \leftrightarrow$   
 $(\text{interval\_list\_invariant } s1 = \text{true} \wedge$   
 $\text{interval\_list\_invariant } s2 = \text{true} \wedge$   
 $(\forall (x1\ x2 : \mathbb{Z}), \text{InZ } x1\ s1 \rightarrow \text{InZ } x2\ s2 \rightarrow \mathbb{Z}.\text{succ } x1 < x2)).$



Proof.

```

intros s1 s2.
split. {
  by apply interval_list_invariant_app_elim.
} {
  move  $\Rightarrow$  []  $H\_inv\_s1$  [].
  by apply interval_list_invariant_app_intro.
}

```

Qed.

Lemma interval\_list\_invariant\_snoc\_intro :  $\forall s1\ y2\ c2,$   
 $interval\_list\_invariant\ s1 = \text{true} \rightarrow$   
 $(c2 \neq 0) \% N \rightarrow$   
 $(\forall x, \text{InZ}\ x\ s1 \rightarrow \text{Z.succ}\ x < y2) \rightarrow$   
 $interval\_list\_invariant\ (s1 ++ ((y2, c2) :: nil)) = \text{true}.$

Proof.

```

intros s1 y2 c2 H_inv_s1 H_c2_neq H_in_s1.
apply interval_list_invariant_app_intro  $\Rightarrow$  //. {
  rewrite interval_list_invariant_cons; done.
} {
  intros x1 x2 H_in_x1.
  rewrite InZ_cons.
  move  $\Rightarrow$  [] //.
  rewrite In_elementsZ_single.
  move  $\Rightarrow$  [H_y2_le] ..
  eapply Z.lt.le_trans; eauto.
}

```

Qed.

## Properties of In and InZ

Lemma encode\_decode\_eq :  $\forall x\ s, \text{Ok}\ s \rightarrow \text{InZ}\ x\ s \rightarrow$   
 $(\text{Enc.encode}\ (\text{Enc.decode}\ x) = x).$

Proof.

```

intros x s.
rewrite /Ok /IsOk /InZ.
move  $\Rightarrow$  [-] H_enc H_in_x.
move : (H_enc - H_in_x)  $\Rightarrow$  [x']  $\leftarrow$ .
rewrite Enc.decode_encode_ok //.

```

Qed.

Lemma In\_alt\_def :  $\forall x\ s, \text{Ok}\ s \rightarrow$   
 $(\text{In}\ x\ s \leftrightarrow \text{List.In}\ x\ (\text{elements}\ s)).$

Proof.

```

intros x s H_ok.
rewrite /ln lnA_alt /elements rev_map_alt_def.
split. {
  move  $\Rightarrow$  [y] [H_y_eq].
  rewrite -!in_rev !in_map_iff.
  move  $\Rightarrow$  [x'] [H_y_eq'] H_x'_in.
  suff H_x'_eq : (Enc.encode x = x'). {
     $\exists$  x'.
    split  $\Rightarrow$  //.
    rewrite -H_x'_eq Enc.decode_encode_ok //.
  }
  have H_enc_list : is_encoded_elems_list (elementsZ s). {
    move : H_ok.
    rewrite /Ok /IsOk  $\Rightarrow$  [] [] //.
  }
  move : (H_enc_list _ H_x'_in)  $\Rightarrow$  [x''] H_x'_eq.
  move : H_y_eq'.
  rewrite -!H_x'_eq Enc.decode_encode_ok  $\Rightarrow$  H_y_eq'.
  subst.
  suff  $\rightarrow$  : Z.eq (Enc.encode x) (Enc.encode y) by done.
  by rewrite Enc.encode_eq.
} {
  move  $\Rightarrow$  H_enc_in.
   $\exists$  x.
  split  $\Rightarrow$  //.
  apply Enc.E.eq_equiv.
}

```

Qed.

Lemma ln\_lnZ :  $\forall x s, \mathbf{Ok} s \rightarrow$   
 $(\ln x s \leftrightarrow \ln Z (Enc.encode x) s).$

Proof.

```

intros x s H_ok.
rewrite /lnZ ln_alt_def /elements rev_map_alt_def -in_rev in_map_iff.
split; last first. {
   $\exists$  (Enc.encode x).
  by rewrite Enc.decode_encode_ok.
}
move  $\Rightarrow$  [y] [<-] H_y_in.
suff :  $\exists z, (Enc.encode z = y).$  {
  move  $\Rightarrow$  [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 lnZ\_ln :  $\forall x\ s, \mathbf{Ok}\ s \rightarrow$   
 $(\text{lnZ } x\ s \rightarrow \text{ln } (\text{Enc.decode } x)\ s).$

Proof.

```

intros x s H_ok.
rewrite ln_lnZ /lnZ.
move : H_ok.
rewrite /Ok /isOk /is_encoded_elems_list.
move  $\Rightarrow$  [-] H_enc.
move  $\Rightarrow$  H_in.
move : (H_enc - H_in)  $\Rightarrow$  [e] H_x.
subst.
by rewrite Enc.decode_encode_ok.
Qed.

```

## Membership specification

Lemma memZ\_spec :  
 $\forall (s : \mathbf{t}) (x : \mathbf{Z}) (Hs : \mathbf{Ok}\ s), \text{memZ } x\ s = \mathbf{true} \leftrightarrow \text{lnZ } x\ s.$

Proof.

```

induction s as [| [y c] s' IH]. {
  intros x -.
  rewrite /lnZ elementsZ_nil //.
} {
  move  $\Rightarrow$  x /Ok_cons [H_inf] [H_c] [H_is_enc] H_s'.
  rewrite /lnZ /memZ elementsZ_cons -/memZ.
  rewrite in_app_iff -!in_rev ln_elementsZ_single.
  case_eq (x <? y). {
    move  $\Rightarrow$  /Z.ltb_lt H_x_lt.
    split; first done.
    move  $\Rightarrow$  []. {
      move  $\Rightarrow$  H_x_in; contradict H_x_in.
      apply Nin_elements_greater with (y := (y + Z.of_N c))  $\Rightarrow$  //. {
        apply H_s'.
      } {
        apply Z.lt_le_add_r  $\Rightarrow$  //.
      }
    }
  }

```

```

    } {
      move  $\Rightarrow$   $[H\_y\_le]$ ; contradict  $H\_y\_le$ .
      by apply Z.nle_gt.
    }
  } {
    move  $\Rightarrow$  /Z.ltb_ge  $H\_y\_le$ .
    case_eq (x <? y + Z.of_N c). {
      move  $\Rightarrow$  /Z.ltb_lt  $H\_x\_lt$ .
      split; last done.
      move  $\Rightarrow$  -.
      by right.
    } {
      move  $\Rightarrow$  /Z.ltb_ge  $H\_yc\_le$ .
      rewrite IH.
      split; first tauto.
      move  $\Rightarrow$  [] //.
      move  $\Rightarrow$  [-]  $H\_x\_lt$ ; contradict  $H\_x\_lt$ .
      by apply Z.nlt_ge.
    }
  }
}
}

```

Qed.

Lemma mem\_spec :

$\forall (s : t) (x : \text{elt}) (Hs : \mathbf{Ok} \ s), \text{mem } x \ s = \mathbf{true} \leftrightarrow \text{In } x \ s.$

Proof.

intros s x Hs.

rewrite **/mem memZ\_spec** **In\_InZ** //.

Qed.

Lemma merge\_interval\_size\_neq\_0 :  $\forall x1 \ c1 \ x2 \ c2,$

$(c1 \neq 0\%N) \rightarrow$

$(\text{merge\_interval\_size } x1 \ c1 \ x2 \ c2 \neq 0)\%N.$

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 :  $\forall x \ c \ s,$

$\text{interval\_list\_invariant } s = \mathbf{true} \rightarrow$

$\text{interval\_list\_elements\_greater } (x + \mathbf{Z.of\_N} \ c) \ s = \mathbf{true} \rightarrow$

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 :  $\forall x$   $c$   $y$   $s$ ,  
interval\_list\_elements\_greater  $y$  (insert\_intervalZ\_guarded  $x$   $c$   $s$ ) = true  $\leftrightarrow$   
(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 :  $\forall x$   $c$   $s1$   $s2$ ,  
(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 :  $\forall x$   $c$   $s$ ,  
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 :  $\forall x$   $c$   $s$ ,  
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 lnZ_insert_intervalZ_guarded : ∀ y x c s,
  lnZ y (insert_intervalZ_guarded x c s) = lnZ y ((x, c) :: s).
Proof.
  intros y x c s.
  rewrite /lnZ 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 Z.le_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 `ln_merge_interval` :  $\forall x1\ c1\ x2\ c2\ y,$   
 $x1 \leq x2 \rightarrow$   
 $x2 \leq x1 + \text{Z.of\_N } c1 \rightarrow$   
 $\text{List.In } y\ (\text{elementsZ\_single } x1\ (\text{merge\_interval\_size } x1\ c1\ x2\ c2)) \leftrightarrow$   
 $\text{List.In } y\ (\text{elementsZ\_single } x1\ c1) \vee \text{List.In } y\ (\text{elementsZ\_single } x2\ c2).$

Proof.

```

intros x1 c1 x2 c2 y H_x1_le H_x2_le.
rewrite !ln_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 :  $\forall y \ s \ x \ c$ ,  
 interval\_list\_invariant  $s = \text{true} \rightarrow$   
 interval\_list\_elements\_greater  $x \ s = \text{true} \rightarrow$   
 $(c \neq 0) \% N \rightarrow$   
 (  $\text{interval\_list\_invariant} (\text{insert\_interval\_begin } x \ c \ s) = \text{true} \wedge$   
 $(\text{InZ } y (\text{insert\_interval\_begin } x \ c \ s) \leftrightarrow$   
 $(\text{List.In } y (\text{elementsZ\_single } x \ c) \vee \text{InZ } y \ s)))$  ).

Proof.

```

intros y.
induction s as [| [y' c'] s' IH]. {
  intros x c - H_c_neq H_z_lt.
  rewrite /insert_interval_begin lnZ_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 !lnZ_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 !lnZ_cons !interval_list_invariant_cons
      interval_list_elements_greater_cons.
    repeat split  $\Rightarrow$  //.
  } {
    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')  $\Rightarrow$  //.
      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''  $\neq$  0%N. {
      by apply merge_interval_size_neq_0.
    }
    move : (IH x c'' H_inv_s' H_pre H_pre2)  $\Rightarrow$  {IH} {H_pre} {H_pre2} [->]  $\rightarrow$ .
    split; first reflexivity.
    unfold c''; clear c''.
    rewrite ln_merge_interval. {
      rewrite lnZ_cons.
      tauto.
    } {
      by apply Z.lt_le_incl.
    } {
      by apply Z.lt_le_incl.
    }
  }
}
Qed.

```

## add specification

Lemma addZ\_lnZ :

$$\forall (s : \mathbf{t}) (x \ y : \mathbf{Z}),$$

$$\text{interval\_list\_invariant } s = \text{true} \rightarrow$$

$$(\text{lnZ } y (\text{addZ } x \ s) \leftrightarrow x = y \vee \text{lnZ } y \ s).$$

Proof.

$$\text{move } \Rightarrow s \ x \ y.$$

$$\text{induction } s \text{ as } [| [z \ c] \ s' \ IH]. \{$$

```

move ⇒ -.
rewrite /lnZ addZ_alt_def
      elementsZ_cons elementsZ_nil app_nil_l -in_rev
      ln_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 ?lnZ_cons ?ln_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 ln_elementsZ_single.
  } {
    rewrite IH.
    tauto.
  } {
    move ⇒ H_x_eq.
    have → : (lnZ y (insert_interval_begin z (N.succ c) s') ↔
      List.In y (elementsZ_single z (N.succ c)) ∨ lnZ 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 :  $\forall s x,$   
 interval\_list\_invariant  $s = \text{true} \rightarrow$   
 interval\_list\_invariant (addZ  $x s$ ) = true.

Proof.

```

move  $\Rightarrow s x.$ 
induction s as [| [z c] s' IH]. {
  move  $\Rightarrow \_.$ 
  by simpl.
} {
  move  $\Rightarrow$  /interval_list_invariant_cons [H_greater] [H_c_neq_0]
    H_inv_c'.
  move : (IH H_inv_c')  $\Rightarrow \{IH\}$  IH.
  rewrite addZ_alt_def.
  have H_succ :  $\forall z, z + \mathbb{Z}.\text{of\_N } 1 = \mathbb{Z}.\text{succ } z$  by done.
  move : (interval_1_compare_elim x z c).
  case_eq (interval_1_compare x (z, c))  $\Rightarrow H\_comp;$ 
  rewrite ?lnZ_cons ?ln_elementsZ_single1 ?H_succ ?Z.lt_succ_r //. {
    move  $\Rightarrow H\_z\_gt.$ 
    rewrite interval_list_invariant_cons /= !andb_true_iff !H_succ.
    repeat split  $\Rightarrow$  //. {
      by apply Z.ltb_lt.
    } {
      apply negb_true_iff, N.eqb_neq  $\Rightarrow$  //.
    }
  } {
    move  $\Rightarrow ?;$  subst.
    rewrite /= !andb_true_iff.
    repeat split  $\Rightarrow$  //. {
      move : H_greater.
      rewrite Z.add_succ_l -Z.add_succ_r N2Z.inj_succ //.
    } {
      apply negb_true_iff, N.eqb_neq  $\Rightarrow$  //.
      apply N.neq_succ_0.
    }
  } {
    move  $\Rightarrow$  [] // -.
    rewrite interval_list_invariant_cons /=.
    tauto.
  } {
    rewrite interval_list_invariant_cons.
    move  $\Rightarrow H\_lt\_x.$ 

```

```

    repeat split  $\Rightarrow$  //.
    apply interval_list_elements_greater_intro  $\Rightarrow$  //.
    move  $\Rightarrow$  xx.
    rewrite addZ_lnZ  $\Rightarrow$  //.
    move  $\Rightarrow$  [ $\leftarrow$  //].
    apply interval_list_elements_greater_alt2_def  $\Rightarrow$  //.
  } {
    move  $\Rightarrow$  H_x_eq.
    apply insert_interval_begin_spec  $\Rightarrow$  //. {
      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* :  $\forall$  '(**Ok** *s*), **Ok** (add *x s*).

Proof.

```

    move  $\Rightarrow$  H_ok_s.
    move : (H_ok_s).
    rewrite /Ok /!sOk /is_encoded_elems_list /add.
    move  $\Rightarrow$  [H_isok_s] H_pre.
    split. {
      apply addZ_invariant  $\Rightarrow$  //.
    } {
      intros y.
      move : (addZ_lnZ s (Enc.encode x) y H_isok_s).
      rewrite /lnZ  $\Rightarrow$   $\rightarrow$ .
      move  $\Rightarrow$  []. {
        move  $\Rightarrow$   $\leftarrow$ .
        by  $\exists$  x.
      } {
        move  $\Rightarrow$  /H_pre //.
      }
    }
}

```

Qed.

Lemma add\_spec :

$\forall$  (*s* : **t**) (*x y* : **elt**) (*Hs* : **Ok** *s*),  
 ln *y* (add *x s*)  $\leftrightarrow$  *Enc.E.eq* *y x*  $\vee$  ln *y s*.

Proof.

```

  intros s x y Hs.

```

```

have Hs' := (add_ok s x Hs).
rewrite !ln_lnZ.
rewrite /add addZ_lnZ. {
  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, (ln x empty ↔ False).
  rewrite /Empty /empty /ln elements_nil.
  intros a.
  rewrite lnA_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 /ln lnA_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.
     $\exists x$ .
    split  $\Rightarrow$  //.
    apply ln_elementsZ_single_hd  $\Rightarrow$  //.
  }
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 :  $\forall x y : \text{elt}, \text{In } y (\text{singleton } x) \leftrightarrow \text{Enc.E.eq } y x$ .
Proof.
  intros x y.
  rewrite singleton_alt_def.
  rewrite (add_spec empty x y) /empty /ln_elements_nil lnA_nil.
  split. {
    move  $\Rightarrow$  [] //.
  } {
    by left.
  }
Qed.

```

### add\_list specification

```

Lemma add_list_ok :  $\forall l s, \text{Ok } s \rightarrow \text{Ok } (\text{add\_list } l s)$ .
Proof.
  induction l as [| x l' IH]. {
    done.
  } {
    move  $\Rightarrow$  s H_s_ok /=.
    apply IH.
    by apply add_ok.
  }
Qed.

Lemma add_list_spec :  $\forall x l s, \text{Ok } s \rightarrow$ 
   $(\text{In } x (\text{add\_list } l s) \leftrightarrow (\text{SetoidList.InA } \text{Enc.E.eq } x l) \vee \text{In } x s)$ .

```

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  $\Rightarrow$  union  $l1$   $s2$ 
      | ICR_subsume_2  $\Rightarrow$  union  $s1$   $l2$ 
    end
  end.
end.
Proof.
  intros  $s1$   $s2$ .
  rewrite /union union_aux_flatten_alt_def.
  case  $s1$  as [| [ $y1$   $c1$ ]  $l1$ ]  $\Rightarrow$  //.
  case  $s2$  as [| [ $y2$   $c2$ ]  $l2$ ]  $\Rightarrow$  //.
  case (interval_compare ( $y1$ ,  $c1$ ) ( $y2$ ,  $c2$ ));
    rewrite union_aux_alt_def //.
Qed.
Lemma union_lnZ :
   $\forall$  ( $s1$   $s2$  : t),
    interval_list_invariant  $s1$  = true  $\rightarrow$ 
    interval_list_invariant  $s2$  = true  $\rightarrow$ 
     $\forall$   $y$ , (lnZ  $y$  (union  $s1$   $s2$ )  $\leftrightarrow$  lnZ  $y$   $s1$   $\vee$  lnZ  $y$   $s2$ ).
Proof.
  intro  $s1$ .
  induction  $s1$  as [| [ $y1$   $c1$ ]  $l1$   $IH1$ ]. {
    intros  $s2$  - -  $y$ .
    rewrite union_alt_def /lnZ /=.
    tauto.
  } {
    induction  $s2$  as [| [ $y2$   $c2$ ]  $l2$   $IH2$ ]. {
      intros - -  $y$ .
      rewrite union_alt_def /lnZ /=.
      tauto.
    } {
      move  $\Rightarrow$   $H\_inv\_s1$   $H\_inv\_s2$ .
      move : ( $H\_inv\_s1$ ) ( $H\_inv\_s2$ ).
      rewrite !interval_list_invariant_cons.
      move  $\Rightarrow$  [ $H\_gr\_l1$ ] [ $H\_c1\_neq\_0$ ]  $H\_inv\_l1$ .
      move  $\Rightarrow$  [ $H\_gr\_l2$ ] [ $H\_c2\_neq\_0$ ]  $H\_inv\_l2$ .
      move : ( $IH2$   $H\_inv\_s1$   $H\_inv\_l2$ )  $\Rightarrow$  { $IH2$ }  $IH2$ .
      have :  $\forall$   $s2$  : t,
        interval_list_invariant  $s2$  = true  $\rightarrow$ 
         $\forall$   $y$  : Z, lnZ  $y$  (union  $l1$   $s2$ )  $\leftrightarrow$  lnZ  $y$   $l1$   $\vee$  lnZ  $y$   $s2$ . {
          intros. by apply  $IH1$ .
        }
      move  $\Rightarrow$  { $IH1$ }  $IH1$   $y$ .
      rewrite union_alt_def.
    }
  }

```

```

move : (interval_compare_elim y1 c1 y2 c2).
case (interval_compare (y1, c1) (y2, c2)). {
  rewrite !lnZ_cons IH1 // lnZ_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 ∧
    (lnZ y (insert_interval_begin y1 c'' l2) ↔
     List.In y (elementsZ_single y1 c'') ∨ lnZ 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 !lnZ_cons /c''.
rewrite -H_y2_eq ln_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 ∧
    (lnZ y (insert_interval_begin y1 c'' l2) ↔
     List.In y (elementsZ_single y1 c'') ∨ lnZ 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 !lnZ_cons /c''.
rewrite ln_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 ∧
    (lnZ y (insert_interval_begin y2 c'' l2) ↔
     List.In y (elementsZ_single y2 c'') ∨ lnZ 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 !lnZ_cons /c''.
rewrite ln_merge_interval. {
  tauto.
} {
  by apply Z.lt_le_incl.
} {
  by apply Z.lt_le_incl.
}
} {
  move ⇒ [? ?]; subst.
  rewrite IH1 ⇒ //.
  rewrite !lnZ_cons.

```

```

tauto.
} {
  move  $\Rightarrow [H\_y2\_le] [H\_y1\_c1\_le] \dots$ .
  rewrite IH1  $\Rightarrow //$ .
  rewrite !InZ_cons.
  suff : (List.In y (elementsZ_single y1 c1)  $\rightarrow$ 
    List.In y (elementsZ_single y2 c2)). {
    tauto.
  }
  rewrite !In_elementsZ_single.
  move  $\Rightarrow [H\_y1\_le H\_y\_lt]$ .
  split; omega.
} {
  move  $\Rightarrow [H\_y1\_le] [H\_y2\_c2\_le] \dots$ .
  rewrite IH2.
  rewrite !InZ_cons.
  suff : (List.In y (elementsZ_single y2 c2)  $\rightarrow$ 
    List.In y (elementsZ_single y1 c1)). {
    tauto.
  }
  rewrite !In_elementsZ_single.
  move  $\Rightarrow [H\_y2\_le H\_y\_lt]$ .
  split; omega.
} {
  rewrite !InZ_cons IH2 !InZ_cons.
  tauto.
} {
  move  $\Rightarrow H\_y1\_eq$ .
  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  $\wedge$ 
    (InZ y (insert_interval_begin y2 c'' l2)  $\leftrightarrow$ 
    List.In y (elementsZ_single y2 c'')  $\vee$  InZ y l2). {
    apply insert_interval_begin_spec  $\Rightarrow //$ . {
      eapply interval_list_elements_greater_impl; eauto.
      apply Z.le_add_r.
    } {
      by apply merge_interval_size_neq_0.
    }
  }

```



```

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.lt_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_lnZ ⇒ //.
    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 :  $\forall (Ok\ s1, Ok\ s2), Ok\ (union\ s1\ s2)$ .

Proof.

```

move  $\Rightarrow$   $H\_ok\_s1$   $H\_ok\_s2$ .
move : ( $H\_ok\_s1$ ) ( $H\_ok\_s2$ ).
rewrite /Ok /IsOk /is_encoded_elems_list /add.
move  $\Rightarrow$  [ $H\_inv\_s1$ ]  $H\_pre1$ .
move  $\Rightarrow$  [ $H\_inv\_s2$ ]  $H\_pre2$ .
split. {
  apply union_invariant  $\Rightarrow$  //.
} {
  intros y.
  move : (union_InZ  $s1$   $s2$   $H\_inv\_s1$   $H\_inv\_s2$ ).
  rewrite /InZ  $\Rightarrow$   $\rightarrow$ .
  move  $\Rightarrow$  []. {
    apply  $H\_pre1$ .
  } {
    apply  $H\_pre2$ .
  }
}

```

Qed.

Lemma union\_spec :

$$\forall (s \ s' : t) (x : \text{elt}) (Hs : \mathbf{Ok} \ s) (Hs' : \mathbf{Ok} \ s'),$$

$$\text{In } x \ (\text{union } s \ s') \leftrightarrow \text{In } x \ s \vee \text{In } x \ s'.$$

Proof.

```

intros s s' x  $H\_ok$   $H\_ok'$ .
rewrite !In_InZ.
rewrite union_InZ  $\Rightarrow$  //. {
  apply  $H\_ok$ .
} {
  apply  $H\_ok'$ .
}

```

Qed.

## inter specification

Lemma inter\_aux\_alt\_def :

$$\forall (y2 : \mathbf{Z}) (c2 : \mathbf{N}) (s : t) \ acc,$$

$$\text{inter\_aux } y2 \ c2 \ acc \ s = \text{match inter\_aux } y2 \ c2 \ \text{nil } s \text{ with}$$

$$(\text{acc}', s') \Rightarrow (\text{acc}' ++ \text{acc}, s')$$

$$\text{end.}$$

Proof.

```

intros y2 c2.
induction s as [|  $y1$   $c1$ ] s' IH  $\Rightarrow$  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, lnZ x1 acc → lnZ 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, (lnZ y acc' ↔
        (lnZ y acc ∨ (lnZ y s ∧ (List.In y (elementsZ_single y2 c2)))))) ∧
      (∀ y, lnZ y s' → lnZ y s) ∧
      (∀ y, lnZ y s → y2 + Z.of_N c2 < y → lnZ 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 lnZ_nil.
    tauto.
  } {
    rewrite interval_list_invariant_cons.

```

```

move  $\Rightarrow$   $H\_inv\_acc$  [ $H\_gr\_s1'$ ] [ $H\_c1\_neq\_0$ ]  $H\_inv\_s1'$ .
move  $\Rightarrow$   $H\_in\_acc\_lt$   $H\_c2\_neq\_0$ .

rewrite inter_aux_alt_def.
case_eq (inter_aux y2 c2 nil ((y1, c1) :: s1')).
move  $\Rightarrow$  acc' s'  $H\_inter\_aux\_eq$ .

set P1 :=  $\forall y : \mathbf{Z}$ ,
  (lnZ y acc'  $\leftrightarrow$ 
    ((lnZ y ((y1, c1) :: s1')  $\wedge$  List.In y (elementsZ_single y2 c2)))).
set P2 := ( $\forall y$ ,
  (lnZ y s'  $\rightarrow$  lnZ y ((y1, c1) :: s1'))  $\wedge$ 
  (lnZ y ((y1, c1) :: s1')  $\rightarrow$ 
    y2 +  $\mathbf{Z.of\_N}$  c2 < y  $\rightarrow$  lnZ y s')).
set P3 := interval_list_invariant (rev acc') = true.
set P4 := interval_list_invariant s' = true.

suff : (P1  $\wedge$  P2  $\wedge$  P3  $\wedge$  P4). {
  move  $\Rightarrow$  [ $H\_P1$ ] [ $H\_P2$ ] [ $H\_P3$ ]  $H\_P4$ .
  split; last split; last split; last split. {
    move  $\Rightarrow$  y.
    move : ( $H\_P1$  y).
    rewrite !lnZ_app lnZ_cons !ln_elementsZ_single.
    move  $\Rightarrow$   $\leftarrow$ .
    tauto.
  } {
    move  $\Rightarrow$  y  $H\_y\_in$ .
    by apply  $H\_P2$ .
  } {
    move  $\Rightarrow$  y  $H\_y\_in$ .
    by apply  $H\_P2$ .
  } {
    rewrite rev_app_distr.
    apply interval_list_invariant_app_intro  $\Rightarrow$  //.
    move  $\Rightarrow$  x1 x2.
    rewrite !lnZ_rev.
    move  $\Rightarrow$   $H\_x1\_in$  /  $H\_P1$  [ $H\_x2\_in1$ ]  $H\_x2\_in2$ .
    apply  $H\_in\_acc\_lt$   $\Rightarrow$  //.
  } {
    apply  $H\_P4$ .
  }
}

move : ( $H\_gr\_s1'$ ).
rewrite interval_list_elements_greater_alt2_def  $\Rightarrow$  //  $\Rightarrow$   $H\_gr\_s1'\_alt$ .

```

```

have : ∀ (acc : list (Z × N)),
  interval_list_invariant (rev acc) = true →
  (∀ y, lnZ 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,
    lnZ x1 acc0 →
    lnZ 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 lnZ_cons !ln_elementsZ_single
      H_in_acc0.
    tauto.
  } {
    move ⇒ y.
    split. {
      move ⇒ /H2.
      rewrite lnZ_cons.
      by right.
    } {

```

```

rewrite lnZ_cons ln_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 lnZ_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 lnZ_nil.

```

```

    split  $\Rightarrow$  //.
    omega.
  } {
    rewrite  $H_{eq\_y2}$ .
    apply Z.le_add_r.
  }
} {
  move  $\Rightarrow [H_{y1\_lt\_y2}] [H_{y2\_lt\_yc1}] H_{yc1\_lt\_yc2}$ .
  apply IH. {
    rewrite interval_list_invariant_sing.
    by apply Z.to_N_minus_neq_0.
  } {
    move  $\Rightarrow x$ .
    rewrite lnZ_cons lnZ_nil ln_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  $\Rightarrow [H_{y2\_lt}] [H_{y1\_lt}] H_{yc1\_lt}$ .
  move  $\Rightarrow [] H_{acc'} H_{s'}$ .
  clear IH P1 P2 P3 P4 H_comp.
  subst s' acc'.
  split; last split; last split. {
    move  $\Rightarrow y$ .
    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 !lnZ_cons !ln_elementsZ_single lnZ_nil H_yc2_intro.
  split. {
    move  $\Rightarrow []$  //.
    move  $\Rightarrow [H_{y1\_le}] H_{y\_lt}$ .
    split; last by omega.
    left. omega.
  } {
    move  $\Rightarrow [H_{in\_s}] [H_{y2\_le}] H_{y\_lt}$ .
    left.
    split; last assumption.
  }
}

```

```

    move :  $H_{in\_s} \Rightarrow []$ . {
      tauto.
    } {
      move  $\Rightarrow$   $/H_{gr\_s1'}_{alt} H_{lt\_y}$ .
      apply Z.le_trans with ( $m := y1 + \text{Z.of\_N } c1$ ). {
        by apply Z.le_add_r.
      } {
        by apply Z.lt_le_incl.
      }
    }
  } {
    move  $\Rightarrow y$ .
    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  $\Rightarrow [H_{y12\_eq}] H_{c12\_eq} [] H_{acc'} H_{s'}$ .
  clear  $IH P1 P2 P3 P4 H_{comp}$ .
  subst.
  split; last split; last split. {
    move  $\Rightarrow y$ .
    rewrite !InZ_cons InZ_nil In_elementsZ_single.
    split; last by tauto. {
      move  $\Rightarrow [] //$ .
      tauto.
    }
  }
} {
  move  $\Rightarrow y$ .
  rewrite InZ_cons In_elementsZ_single.
  split; first by right.
  move  $\Rightarrow [] //$ .
  move  $\Rightarrow [-] H_{y\_lt} H_{lt\_y}$ .
  exfalso.
  suff : ( $y2 + \text{Z.of\_N } c2 < y2 + \text{Z.of\_N } c2$ ) by
    apply Z.lt_irrefl.
  apply Z.lt_trans with ( $m := y$ )  $\Rightarrow //$ .

```

```

} {
  rewrite interval_list_invariant_sing //.
} {
  assumption.
}
} {
  move  $\Rightarrow$   $[H\_y2\_le\_y1] [H\_yc1\_le\_yc2] \_.$ 
  apply IH. {
    by rewrite interval_list_invariant_sing.
  } {
    move  $\Rightarrow$   $y.$ 
    rewrite lnZ_cons lnZ_nil ln_elementsZ_single.
    split. {
      move  $\Rightarrow$   $[]$  //.
      move  $\Rightarrow$   $[H\_y1\_le] H\_y\_lt.$ 
      split; first done.
      split; omega.
    } {
      move  $\Rightarrow$   $[?] \_.$ 
      by left.
    }
  } {
    assumption.
  }
} {
  rewrite  $/P1 /P2 /P3 /P4.$ 
  move  $\Rightarrow$   $[H\_y1\_le] [H\_yc2\_le] \_.$ 
  move  $\Rightarrow$   $[] H\_acc' H\_s'.$ 
  clear IH P1 P2 P3 P4 H_comp.
  subst.
  split; last split; last split. {
    move  $\Rightarrow$   $y.$ 
    rewrite !lnZ_cons !ln_elementsZ_single lnZ_nil.
    split. {
      move  $\Rightarrow$   $[]$  //.
      move  $\Rightarrow$   $[H\_y2\_le] H\_y\_lt.$ 
      split; last by omega.
      left. omega.
    } {
      move  $\Rightarrow$   $[H\_in\_s] [H\_y2\_le] H\_y\_lt.$ 
      by left.
    }
  }
}

```

```

} {
  tauto.
} {
  by rewrite interval_list_invariant_sing.
} {
  by rewrite interval_list_invariant_cons.
}
} {
  rewrite /P1 /P2 /P3 /P4.
  move  $\Rightarrow$   $H_{yc2\_lt} [] H_{acc'} H_{s'}$ .
  clear IH P1 P2 P3 P4 H_comp.
  subst.
  split; last split; last split. {
    move  $\Rightarrow$  y.
    rewrite lnZ_cons !ln_elementsZ_single lnZ_nil.
    split; first done.
    move  $\Rightarrow$  [] []. {
      move  $\Rightarrow$   $[H_{y1\_le\_y}] H_{y\_lt\_yc1}$ .
      move  $\Rightarrow$   $[H_{y2\_le\_y}] H_{y\_lt\_yc2}$ .
      omega.
    } {
      move  $\Rightarrow$   $/H_{gr\_s1'}\_alt H_{lt\_y} [-] H_{y\_lt}$ .
      suff :  $(y1 + \text{Z.of\_N } c1 < y1)$ . {
        apply Z.nlt_ge.
        apply Z.le_add_r.
      }
      omega.
    }
  }
} {
  tauto.
} {
  done.
} {
  by rewrite interval_list_invariant_cons.
}
} {
  rewrite /P1 /P2 /P3 /P4.
  move  $\Rightarrow$   $H_{y1\_eq} [] H_{acc'} H_{s'}$ .
  clear IH P1 P2 P3 P4 H_comp.
  subst acc' s'.
  split; last split; last split. {
    move  $\Rightarrow$  y.

```





```

rewrite /rev' rev_append_rev app_nil_r.
split; last done.
move ⇒ y.
rewrite lnZ_nil lnZ_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 lnZ_nil
              lnZ_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,
    lnZ x1 acc →
    lnZ 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 lnZ_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,
  lnZ x1 acc' → lnZ x2 s1' → lnZ 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 lnZ_cons; by right.
  }
} {
  rewrite ln_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' lnZ_cons
  ln_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. {
  apply inter_invariant ⇒ //.
} {
  intros y.
  move : (inter_lnZ s1 s2 H_inv_s1 H_inv_s2).
  rewrite /lnZ ⇒ →.
  move ⇒ [].
  move ⇒ /H_pre1 //.
}
Qed.

```

Lemma inter\_spec :

$$\forall (s \ s' : t) (x : \text{elt}) (Hs : \mathbf{Ok} \ s) (Hs' : \mathbf{Ok} \ s'),$$

$$\text{ln } x \ (\text{inter } s \ s') \leftrightarrow \text{ln } x \ s \wedge \text{ln } x \ s'.$$

Proof.

```

  intros s s' x H_ok H_ok'.
  rewrite !ln_lnZ.
  rewrite inter_lnZ ⇒ //. {
    apply H_ok.
  } {
    apply H_ok'.
  }
Qed.

```

## diff specification

Lemma diff\_aux\_alt\_def :

$$\forall (y2 : \mathbf{Z}) (c2 : \mathbf{N}) (s : t) \text{ acc},$$

$$\text{diff\_aux } y2 \ c2 \ \text{acc } s = \text{match diff\_aux } y2 \ c2 \ \text{nil } s \text{ with}$$

$$\quad (acc', s') \Rightarrow (acc' ++ acc, s')$$

$$\text{end.}$$

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 :

$\forall (y2 : \mathbb{Z}) (c2 : \mathbb{N}) (s : t) \text{ acc},$   
 $\text{interval\_list\_invariant } (\text{List.rev } \text{acc}) = \text{true} \rightarrow$   
 $\text{interval\_list\_invariant } s = \text{true} \rightarrow$   
 $(\forall x1 \ x2, \text{InZ } x1 \ \text{acc} \rightarrow \text{InZ } x2 \ s \rightarrow \mathbb{Z}.\text{succ } x1 < x2) \rightarrow$   
 $(\forall x, \text{InZ } x \ \text{acc} \rightarrow x < y2) \rightarrow$   
 $(c2 \neq 0\%N) \rightarrow$   
 $\text{match } (\text{diff\_aux } y2 \ c2 \ \text{acc } s) \text{ with}$   
 $(\text{acc}', s') \Rightarrow (\forall y, \text{InZ } y \ (\text{List.rev\_append } \text{acc}' \ s') \leftrightarrow$   
 $\text{InZ } y \ (\text{List.rev\_append } \text{acc } s) \wedge \sim (\text{List.In } y \ (\text{elementsZ\_single}$   
 $y2 \ c2))) \wedge$   
 $(\text{interval\_list\_invariant } (\text{List.rev\_append } \text{acc}' \ s') = \text{true}) \wedge$   
 $(\forall x, \text{InZ } x \ \text{acc}' \rightarrow x < y2 + \mathbb{Z}.\text{of\_N } c2)$

end.

Proof.

$\text{intros } y2 \ c2.$   
 $\text{induction } s \text{ as } [[y1 \ c1] \ s1' \ IH] \Rightarrow \text{acc}. \{$   
 $\text{rewrite } / \text{diff\_aux -rev\_alt}.$   
 $\text{move } \Rightarrow H\_inv\_acc \_ \_ H\_in\_acc \ H\_c2\_neq.$   
 $\text{split; last split. } \{$   
 $\text{move } \Rightarrow y; \text{split; last by move } \Rightarrow [] \ / \ /.$   
 $\text{rewrite InZ\_rev}.$   
 $\text{move } \Rightarrow H\_in. \text{split } \Rightarrow \ / \ /.$   
 $\text{move } \Rightarrow / \text{In\_elementsZ\_single} \Rightarrow [] \ [] \ / \ \mathbb{Z}.\text{nlt\_ge } H\_neq.$   
 $\text{contradict } H\_neq.$   
 $\text{by apply } H\_in\_acc.$   
 $\} \{$   
 $\text{assumption}.$   
 $\} \{$   
 $\text{intros } x \ H\_in\_acc'.$   
 $\text{apply } \mathbb{Z}.\text{lt\_le\_trans} \text{ with } (m := y2). \{$   
 $\text{by apply } H\_in\_acc.$   
 $\} \{$   
 $\text{by apply } \mathbb{Z}.\text{le\_add\_r}.$   
 $\}$   
 $\}$   
 $\} \{$   
 $\text{rewrite interval\_list\_invariant\_cons}.$   
 $\text{move } \Rightarrow H\_inv\_acc \ [H\_gr\_s1'] \ [H\_c1\_neq\_0] \ H\_inv\_s1'.$   
 $\text{move } \Rightarrow H\_in\_s1 \ H\_in\_acc \ H\_c2\_neq\_0.$

```

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,
  (lnZ y acc' ∨ lnZ y s') ↔
  lnZ y ((y1, c1) :: s1') ∧ ¬ List.In y (elementsZ_single y2 c2).
set P2 := interval_list_invariant (rev acc' ++ s') = true.
set P3 := ∀ x : Z, lnZ 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 !lnZ_app
      !lnZ_rev ln_elementsZ_single.
    suff : (lnZ 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 Z.lt_not_le.
  } {
    rewrite rev_append_rev rev_app_distr -app_assoc.
    apply interval_list_invariant_app_intro => //.
    move => x1 x2.
    rewrite lnZ_app !lnZ_rev.
    move => H_in_acc' H_x2_in_s'.
    suff : (lnZ x2 ((y1, c1) :: s1')). {
      by apply H_in_s1.
    }
    move : (H_P1 x2).
    tauto.
  } {
    move => x.
    rewrite lnZ_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,
    lnZ 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,
    lnZ x1 acc0 → lnZ 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, lnZ 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 !lnZ_app !lnZ_rev ln_elementsZ_single.
    move ⇒ →.
    rewrite lnZ_cons ln_elementsZ_single.
    split. {
      rewrite H_in_acc0 -(Z.nle_gt y2 y).
      tauto.
    } {

```



```

    rewrite  $H\_in\_acc0$  -(Z.nle_gt  $y2$   $y$ ).
    move  $\Rightarrow []$   $H\_in$   $H\_nin\_i2$ .
    split; last by assumption.
    move :  $H\_in \Rightarrow []$   $H\_in$ ; last by right.
    left.
    omega.
  }
}
}
move  $\Rightarrow \{IH\}$   $IH$ .

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 ( $y1$ ,  $c1$ ) ( $y2$ ,  $c2$ ))  $\Rightarrow H\_comp$ ;
                                                                    fold diff_aux. {
  move  $\Rightarrow H\_lt\_y2$ .
  apply  $IH$ . {
    by rewrite interval_list_invariant_sing.
  } {
    move  $\Rightarrow x$ .
    rewrite lnZ_cons ln_elementsZ_single.
    split; last by tauto.
    move  $\Rightarrow []$ ; last done.
    move  $\Rightarrow [H\_y1\_le\ H\_x\_lt]$ .
    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  $\Rightarrow H\_eq\_y2$ .
  apply  $IH$ . {
    by rewrite interval_list_invariant_sing.
  } {
    move  $\Rightarrow x$ .
    rewrite lnZ_cons ln_elementsZ_single - $H\_eq\_y2$ .
    split; last by tauto.
    move  $\Rightarrow []$ ; last done.
    move  $\Rightarrow []$ . done.
  } {

```

```

    rewrite  $H_{eq\_y2}$ .
    by apply  $Z\_le\_add\_r$ .
  }
} {
  move  $\Rightarrow [H_{y1\_lt\_y2}] [H_{y2\_lt\_yc1}] H_{yc1\_lt\_yc2}$ .
  apply  $IH$ . {
    rewrite interval_list_invariant_sing.
    by apply  $Z\_to\_N\_minus\_neq\_0$ .
  } {
    move  $\Rightarrow x$ .
    rewrite  $\ln Z\_cons \ln\_elementsZ\_single$   $Z2N.id$ ; last omega.
    replace  $(y1 + (y2 - y1))$  with  $y2$  by omega.
    split; last tauto.
    move  $\Rightarrow []$  //.
    move  $\Rightarrow [H_{y1\_le}] H_{x\_lt}$ .
    repeat split  $\Rightarrow$  //.
    apply  $Z.lt\_trans$  with  $(m := y2) \Rightarrow$  //.
  } {
    by apply  $Z.lt\_le\_incl$ .
  }
} {
  rewrite  $/P1 /P2 /P3$ .
  move  $\Rightarrow [H_{y2\_lt}] [H_{y1\_lt}] H_{yc1\_lt} [H_{acc'}] H_{s'}$ .
  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  $H_{nin\_yc2} : \forall y,$ 
     $\ln Z y s1' \rightarrow \neg y2 \leq y < y2 + Z.of\_N c2$ . {
    move  $\Rightarrow y / H_{gr\_s1'\_alt} H_{lt\_y}$ .
    move  $\Rightarrow [H_{y2\_le\_y}]$ .
    apply  $Z.le\_ngt, Z.lt\_le\_incl$ .
    by apply  $Z.lt\_trans$  with  $(m := y1 + Z.of\_N c1)$ .
  }
  split; last split. {
    move  $\Rightarrow y$ .
    rewrite  $!\ln Z\_cons !\ln\_elementsZ\_single H_{yc1\_intro}$ .
    split. {
      move  $\Rightarrow []$  //.
      move  $\Rightarrow []$ . {

```

```

    move  $\Rightarrow$   $[H\_le\_y] \ H\_y\_lt$ .
    split. {
      left; omega.
    } {
      move  $\Rightarrow$   $[-]$ .
      by apply Z.nlt_ge.
    }
  } {
    move :  $(H\_nin\_yc2 \ y)$ . tauto.
  }
} {
  move  $\Rightarrow$   $[] \ []$ ; last by right; right.
  move  $\Rightarrow$   $[H\_y\_ge] \ H\_y\_lt\_yc1 \ H\_nin\_yc2'$ .
  right; left. omega.
}
} {
  rewrite interval_list_invariant_cons  $H\_yc1\_intro$ .
  split  $\Rightarrow$  //.
  split  $\Rightarrow$  //.
  by apply Z_to_N_minus_neq_0.
} {
  move  $\Rightarrow$   $[] \ //$ .
}
} {
  rewrite  $/P1 \ /P2 \ /P3$ .
  move  $\Rightarrow$   $[H\_y12\_eq] \ H\_c12\_eq \ [] \ H\_acc' \ H\_s'$ .
  clear  $IH \ P1 \ P2 \ P3 \ H\_comp$ .
  subst.
  split; last split. {
    move  $\Rightarrow$   $y$ .
    rewrite lnZ_cons ln_elementsZ_single.
    split; last by tauto. {
      move  $\Rightarrow$   $[] \ //$ .
      move  $\Rightarrow$   $H\_y\_in$ .
      split; first by right.
      move  $\Rightarrow$   $[] \ -$ .
      by apply Z.nlt_ge, Z.lt_le_incl,  $H\_gr\_s1'\_alt$ .
    }
  }
} {
  apply  $H\_inv\_s1'$ .
} {
  move  $\Rightarrow$   $x \ [] \ //$ .
}

```

```

}
} {
  move  $\Rightarrow [H\_y2\_le\_y1] [H\_yc1\_le\_yc2] \_.$ 
  apply IH. {
    done.
  } {
    move  $\Rightarrow x.$ 
    split; first done.
    omega.
  } {
    assumption.
  }
} {
  rewrite /P1 /P2 /P3.
  move  $\Rightarrow [H\_y1\_le] [H\_yc2\_le\_yc1] \_ [] H\_acc' H\_s'.$ 
  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  $\Rightarrow y.$ 
    rewrite !lnZ_insert_intervalZ_guarded
      !lnZ_cons !ln_elementsZ_single
      H_yc1_intro H_y1_intro lnZ_nil.
    split. {
      rewrite -!or_assoc.
      move  $\Rightarrow [[] [] [] []] //.$  {
        move  $\Rightarrow [H\_y1\_le\_y] H\_y\_lt.$ 
        split. {
          left.
          split  $\Rightarrow //.$ 
          apply Z.lt_le_trans with ( $m := y2 + Z.of\_N\ c2$ )  $\Rightarrow //.$ 
          apply Z.lt_trans with ( $m := y2$ )  $\Rightarrow //.$ 
          by apply Z.lt_add_r.
        } {
          move  $\Rightarrow [] /Z.le\_ngt //.$ 
        }
      }
    }
  }

```

```

} {
  move  $\Rightarrow [H\_y2c\_le\_y] H\_y\_lt\_yc1$ .
  split. {
    left.
    split  $\Rightarrow //$ .
    apply Z.le_trans with ( $m := y2 + \text{Z.of\_N } c2$ )  $\Rightarrow //$ .
    apply Z.le_trans with ( $m := y2$ )  $\Rightarrow //$ .
    apply Z.le_add_r.
  } {
    move  $\Rightarrow [] - / \text{Z.lt\_nge} //$ .
  }
} {
  move  $\Rightarrow H\_y\_in\_s1'$ .
  split; first by right.
  suff  $H\_suff : y2 + \text{Z.of\_N } c2 \leq y$ . {
    move  $\Rightarrow [] - / \text{Z.lt\_nge} //$ .
  }
  apply Z.le_trans with ( $m := y1 + \text{Z.of\_N } c1$ )  $\Rightarrow //$ .
  apply Z.lt\_le\_incl.
  by apply  $H\_gr\_s1'\_alt$ .
}
} {
  move  $\Rightarrow [] []$ ; last by tauto.
  move  $\Rightarrow [H\_y1\_le\_y] H\_y\_lt H\_neq\_y2$ .
  apply not_and in  $H\_neq\_y2$ ; last by apply Z.le_decidable.
  case  $H\_neq\_y2$ . {
    move  $\Rightarrow / \text{Z.nle\_gt } H\_y\_lt'$ .
    left; left; done.
  } {
    move  $\Rightarrow / \text{Z.nlt\_ge } H\_le\_y$ .
    right; left; done.
  }
}
} {
  rewrite insert_intervalZ_guarded_rev_nil_app.
  rewrite !interval_list_invariant_insert_intervalZ_guarded  $\Rightarrow //$ . {
    rewrite  $H\_yc1\_intro \Rightarrow //$ .
  } {
    rewrite /insert_intervalZ_guarded.
    case_eq ((Z.to_N ( $y1 + \text{Z.of\_N } c1 - (y2 + \text{Z.of\_N } c2)$ ) =? 0)%N). {
      rewrite  $H\_y1\_intro$ .
      move  $\Rightarrow / \text{N.eqb\_eq} / \text{N2Z.inj\_iff}$ .
    }
  }
}

```

```

      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 lnZ_insert_intervalZ_guarded lnZ_cons ln_elementsZ_single H_y1_intro lnZ_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 lnZ_cons ln_elementsZ_single.
    split; last by tauto. {
      move ⇒ [] //.
      move ⇒ H_y_in.
      split; first assumption.
      rewrite ln_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 lnZ_cons ln_elementsZ_single.
    split; last by tauto. {
      move ⇒ [] //.
      move ⇒ H_y_in.
      split; first assumption.
      rewrite ln_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 :

$\forall (s2\ s1\ acc : t),$   
 $interval\_list\_invariant\ (rev\_append\ acc\ s1) = true \rightarrow$

```

interval_list_invariant s2 = true →
(∀ x1 x2, lnZ x1 acc → lnZ x2 s2 → Z.succ x1 < x2) →
((∀ y, (lnZ y (diff_aux2 acc s1 s2) ↔
      ((lnZ y acc) ∨ (lnZ y s1)) ∧ ¬lnZ 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 lnZ_app lnZ_rev lnZ_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 lnZ_app lnZ_nil lnZ_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, lnZ x acc → x < y2). {
  move ⇒ x H_x_in.
  have H_y2_in : (lnZ y2 ((y2, c2) :: s2')). {
    rewrite lnZ_cons.
    left.
    by apply ln_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,
    lnZ x1 acc → lnZ 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 lnZ_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,
  lnZ x1 acc' → lnZ 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  $\Rightarrow H\_gr\_s2'$ .
      by apply  $H\_gr\_s2'$ .
    }
  }

  move : ( $IH\ s1'\ acc'\ H\_inv\_res\ H\_inv\_s2'\ H\_acc'\_s2'$ ).
  move  $\Rightarrow []\ H\_inZ\_diff\_res \rightarrow$ .
  split; last done.
  move  $\Rightarrow y$ .
  rewrite  $H\_inZ\_diff\_res$ .
  move : ( $H\_inZ\_res\ y$ ).
  rewrite ! $rev\_append\_rev$  ! $lnZ\_app$  ! $lnZ\_rev$   $lnZ\_cons$ .
  move  $\Rightarrow \rightarrow$ .
  tauto.
}
}
}
Qed.

```

Lemma diff\_lnZ :

```

 $\forall (s1\ s2 : t),$ 
  interval_list_invariant  $s1 = true \rightarrow$ 
  interval_list_invariant  $s2 = true \rightarrow$ 
   $\forall y, (lnZ\ y\ (diff\ s1\ s2) \leftrightarrow lnZ\ y\ s1 \wedge \neg lnZ\ y\ s2)$ .

```

Proof.

```

  intros  $s1\ s2\ H\_inv\_s1\ H\_inv\_s2\ y$ .
  rewrite /diff.

  move : ( $diff\_aux2\_props\ s2\ s1\ nil$ ).
  move  $\Rightarrow []\ /\$ .
  move  $\Rightarrow H\_in\_diff\ \_$ .
  rewrite  $H\_in\_diff\ lnZ\_nil$ .
  tauto.

```

Qed.

Lemma diff\_invariant :

```

 $\forall (s1\ s2 : t),$ 
  interval_list_invariant  $s1 = true \rightarrow$ 
  interval_list_invariant  $s2 = true \rightarrow$ 
  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_lnZ s1 s2 H_inv_s1 H_inv_s2).
    rewrite /lnZ ⇒ →.
    move ⇒ [].
    move ⇒ /H_pre1 //.
  }
Qed.

```

Lemma diff\_spec :

$$\forall (s \ s' : t) (x : \text{elt}) (Hs : \text{Ok } s) (Hs' : \text{Ok } s'),$$

$$\ln x (\text{diff } s \ s') \leftrightarrow \ln x s \wedge \neg \ln x s'.$$

Proof.

```

  intros s s' x H_ok H_ok'.
  rewrite !ln_lnZ.
  rewrite diff_lnZ ⇒ //. {
    apply H_ok.
  } {
    apply H_ok'.
  }
Qed.

```

## remove specification

Lemma removeZ\_alt\_def :  $\forall x \ s \ acc,$   
 interval\_list\_invariant  $s = \text{true} \rightarrow$   
 removeZ\_aux  $acc \ x \ s = \text{match } \text{diff\_aux } x \ (1\%N) \ acc \ s \text{ with}$   
      $(acc', s') \Rightarrow \text{rev\_append } acc' \ s'$   
 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  $\Rightarrow [H\_y1\_eq] H\_c1\_eq$ .
  move  $\Rightarrow [] //$ .
  move  $\Rightarrow [H\_lt\_y2] H\_y2\_lt$ .
  have  $H\_y2\_nlt : \sim (y2 < y1)$ . {
    apply  $Z.nlt\_ge \Rightarrow //$ .
  }
  move :  $(H\_y2\_nlt) (H\_y2\_lt) \Rightarrow /Z.ltb\_nlt \rightarrow /Z.ltb\_lt \rightarrow$ .
  rewrite  $/insert\_intervalZ\_guarded$ .
  have  $\rightarrow : (Z.to\_N (y1 + Z.of\_N c1 - Z.succ y2) =? 0 = true)\%N$ . {
    rewrite  $H\_y1\_eq H\_c1\_eq Z.add\_1\_r Z.sub\_diag //$ .
  }

  have  $\rightarrow : (Z.to\_N (y2 - y1) =? 0 = true)\%N$ . {
    rewrite  $H\_y1\_eq Z.sub\_diag //$ .
  }
  done.
} {
  move  $\Rightarrow [H\_y2\_le] [H\_yc1\_le] -$ .
  move  $\Rightarrow [] //$ .
  move  $\Rightarrow [H\_y1\_le] H\_y2\_lt$ .
  have  $H\_y2\_nlt : \sim (y2 < y1)$ . {
    apply  $Z.nlt\_ge \Rightarrow //$ .
  }
  move :  $(H\_y2\_nlt) (H\_y2\_lt) \Rightarrow /Z.ltb\_nlt \rightarrow /Z.ltb\_lt \rightarrow$ .
  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  $\rightarrow : (Z.to\_N (y1 + Z.of\_N c1 - Z.succ y2) =? 0 = true)\%N$ . {
  rewrite  $H\_yc1\_eq Z.sub\_diag //$ .
}

have  $\rightarrow : (Z.to\_N (y2 - y1) =? 0 = true)\%N$ . {
  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.lt_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.lt_lt → //.
  }
}
Qed.

```

Lemma removeZ\_interval\_list\_invariant :  $\forall s x, \text{interval\_list\_invariant } s = \text{true} \rightarrow \text{interval\_list\_invariant } (\text{removeZ } x \ s) = \text{true}.$

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 :

$\forall (s : t) (x \ y : \mathbf{Z}) (Hs : \text{interval\_list\_invariant } s = \text{true}),$   
 $\text{InZ } y \ (\text{removeZ } x \ s) \leftrightarrow \text{InZ } y \ s \wedge \neg \text{Z.eq } y \ x.$

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 :  $\forall '(\mathbf{Ok} \ s), \mathbf{Ok} \ (\text{remove } x \ s).$

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  $\Rightarrow$   $y$ .
    move : (removeZ_spec  $s$  ( $Enc.encode\ x$ )  $y$   $H\_is\_ok\_s$ ).
    rewrite /lnZ  $\Rightarrow$   $\rightarrow$   $\llbracket$   $H\_y\_in\ \_$ .
    apply  $H\_enc\_s \Rightarrow //$ .
  }

```

Qed.

Lemma remove\_spec :

$\forall (s : t) (x\ y : elt) (Hs : \mathbf{Ok}\ s),$   
 $\ln\ y\ (\text{remove}\ x\ s) \leftrightarrow \ln\ y\ s \wedge \neg Enc.E.eq\ y\ x.$

Proof.

```

  intros  $s\ x\ y\ Hs$ .
  have  $H\_rs := (\text{remove\_ok}\ s\ x\ Hs)$ .
  rewrite /remove !ln_lnZ.
  rewrite removeZ_spec. {
    rewrite  $Enc.encode\_eq //$ .
  } {
    apply  $Hs$ .
  }

```

Qed.

## remove\_list specification

Lemma remove\_list\_ok :  $\forall\ l\ s, \mathbf{Ok}\ s \rightarrow \mathbf{Ok}\ (\text{remove\_list}\ l\ s).$

Proof.

```

  induction  $l$  as  $\llbracket x\ l'\ IH \rrbracket. \{$ 
    done.
  } {
    move  $\Rightarrow$   $s\ H\_s\_ok /=$ .
    apply  $IH$ .
    by apply remove_ok.
  }

```

Qed.

Lemma remove\_list\_spec :  $\forall\ x\ l\ s, \mathbf{Ok}\ s \rightarrow$

$(\ln\ x\ (\text{remove\_list}\ l\ s) \leftrightarrow \sim(\ln A\ Enc.E.eq\ x\ l) \wedge \ln\ x\ s).$

Proof.

```

  move  $\Rightarrow$   $x$ .
  induction  $l$  as  $\llbracket y\ l'\ IH \rrbracket. \{$ 
    intros  $s\ H$ .
    rewrite /=  $\ln A\_nil$ .
    tauto.
  } {
    move  $\Rightarrow$   $s\ H\_ok /=$ .
  }

```



```

    rewrite IH remove_spec lnA_cons.
    tauto.
  }
Qed.

```

## subset specification

```

Lemma subset_flatten_alt_def :  $\forall (s1\ s2 : t)$ ,
  subset s1 s2 =
  match (s1, s2) with
  | (nil, _)  $\Rightarrow$  true
  | (_ :: _, nil)  $\Rightarrow$  false
  | ((y1, c1) :: l1, (y2, c2) :: l2)  $\Rightarrow$ 
    match (interval_compare (y1, c1) (y2, c2)) with
    | ICR_before  $\Rightarrow$  false
    | ICR_before_touch  $\Rightarrow$  false
    | ICR_after  $\Rightarrow$  subset s1 l2
    | ICR_after_touch  $\Rightarrow$  false
    | ICR_overlap_before  $\Rightarrow$  false
    | ICR_overlap_after  $\Rightarrow$  false
    | ICR_equal  $\Rightarrow$  subset l1 l2
    | ICR_subsume_1  $\Rightarrow$  subset l1 s2
    | ICR_subsume_2  $\Rightarrow$  false
    end
  end.

```

Proof.

```

  intros s1 s2.
  case s1, s2  $\Rightarrow$  /.

```

Qed.

```

Lemma subset_props_aux :  $\forall y1\ c1\ l1\ y2\ c2\ l2$ ,
  ( $\exists y$ ,  $\text{lnZ } y ((y1, c1) :: l1) \wedge \neg \text{lnZ } y ((y2, c2) :: l2)$ )  $\rightarrow$ 
  (false = true  $\leftrightarrow$ 
  ( $\forall y : \mathbf{Z}$ ,
     $\text{lnZ } y ((y1, c1) :: l1) \rightarrow \text{lnZ } y ((y2, c2) :: l2)$ )).

```

Proof.

```

  intros y1 c1 l1 y2 c2 l2.
  move  $\Rightarrow$  [y] [H_y_in H_y_nin].
  split; first done.
  move  $\Rightarrow$  H.
  contradict H_y_nin.
  by apply H.

```

Qed.

Lemma subset\_props\_aux\_before :  $\forall y1\ c1\ l1\ y2\ c2\ l2,$   
 $(c1 \neq 0\%N) \rightarrow$   
 $\text{interval\_list\_invariant } ((y2, c2) :: l2) = \text{true} \rightarrow$   
 $(y1 < y2) \rightarrow$   
 $(\text{false} = \text{true} \leftrightarrow$   
 $(\forall y : \mathbb{Z},$   
 $\text{InZ } y ((y1, c1) :: l1) \rightarrow \text{InZ } y ((y2, c2) :: l2)))$ .

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.
exists y1.
split. {
  rewrite InZ_cons.
  left.
  by apply In_elementsZ_single_hd.
} {
  rewrite InZ_cons.
  suff :  $\neg (\text{List.In } y1 (\text{elementsZ\_single } y2\ c2)) \wedge \neg \text{InZ } y1\ l2$  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 :  $\forall s1\ s2 : t,$   
 $\text{interval\_list\_invariant } s1 = \text{true} \rightarrow$   
 $\text{interval\_list\_invariant } s2 = \text{true} \rightarrow$   
 $(\text{subset } s1\ s2 = \text{true} \leftrightarrow$   
 $(\forall y, \text{InZ } y\ s1 \rightarrow \text{InZ } y\ s2))$ .

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 lnZ_nil => {H} H.
  contradict H.
  rewrite lnZ_cons; left.
  by apply ln_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, lnZ y l1 → lnZ y s2)). {
    intros. by apply IH1.
  }
  move => {IH1} IH1.
  have H_yc2_nin : ¬ lnZ (y2 + Z.of_N c2) ((y2, c2) :: l2). {
    rewrite !lnZ_cons !ln_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  $\Rightarrow$   $H_{y2\_eq}$ .
    apply subset_props_aux_before  $\Rightarrow$  //.
    rewrite  $-H_{y2\_eq}$ .
    by apply Z.lt_add_r.
  } {
    move  $\Rightarrow$   $[H_{y1\_lt}] \dots$ .
    apply subset_props_aux_before  $\Rightarrow$  //.
  } {
    move  $\Rightarrow$   $[H_{y2\_lt}] [H_{y1\_lt}] H_{yc2\_lt}$ .
    apply subset_props_aux.
     $\exists (y2 + \text{Z.of\_N } c2)$ .
    split  $\Rightarrow$  //.
    rewrite !InZ_cons !In_elementsZ_single.
    left.
    split  $\Rightarrow$  //.
    by apply Z.lt_le_incl.
  } {
    move  $\Rightarrow$   $[H_{y1\_eq}] H_{c1\_eq}$ ; subst.
    rewrite IH1  $\Rightarrow$  //.
    split; move  $\Rightarrow$   $H_{pre} \ y$ ; move :  $(H_{pre} \ y) \Rightarrow \{H_{pre}\}$ ;
      rewrite !InZ_cons. {
        tauto.
      } {
        move  $\Rightarrow$   $H_{pre} \ H_{y\_in\_l1}$ .
        suff :  $\sim (\text{List.In } y \ (\text{elementsZ\_single } y2 \ c2))$ . {
          tauto.
        }
        move :  $H_{gr\_l1}$ .
        rewrite interval_list_elements_greater_alt2_def
          // In_elementsZ_single.
        move  $\Rightarrow$   $H$ ; move :  $(H \ y \ H_{y\_in\_l1}) \Rightarrow \{H\}$ .
        move  $\Rightarrow$   $/\text{Z.lt\_ngt } H_{neq} \ [-]$  //.
      }
  } {
    move  $\Rightarrow$   $[H_{y2\_lt\_y1}] [H_{yc1\_le}] \dots$ .
    rewrite IH1.
    split; move  $\Rightarrow$   $H_{pre} \ y$ ; move :  $(H_{pre} \ y) \Rightarrow \{H_{pre}\}$ ;
      rewrite !InZ_cons. {
        move  $\Rightarrow$   $H \ []$ ; last apply  $H$ . move  $\Rightarrow$   $\{H\}$ .
        rewrite !In_elementsZ_single.
      }
  }

```

```

    move  $\Rightarrow$   $[H\_y1\_le] \ H\_y\_lt$ .
    left.
    omega.
  } {
    move  $\Rightarrow H\_pre \ H\_y\_in\_l1$ .
    apply  $H\_pre$ .
    by right.
  } {
    assumption.
  }
} {
  move  $\Rightarrow [H\_y1\_le] \ [-] \ []$ . {
    apply subset_props_aux_before  $\Rightarrow //$ .
  } {
    move  $\Rightarrow H\_yc2\_lt$ .
    apply subset_props_aux.
     $\exists (y2 + Z.of\_N \ c2)$ .
    split  $\Rightarrow //$ .
    rewrite !lnZ_cons !ln_elementsZ_single.
    left.
    split  $\Rightarrow //$ .
    apply  $Z.le\_trans$  with  $(m := y2) \Rightarrow //$ .
    apply Z_le_add_r.
  }
} {
  move  $\Rightarrow H\_yc2\_lt\_y1$ .
  rewrite IH2.
  split; move  $\Rightarrow H\_pre \ y$ ; move :  $(H\_pre \ y) \Rightarrow \{H\_pre\}$ ;
                                                                    rewrite !lnZ_cons. {
    tauto.
  } {
    rewrite !ln_elementsZ_single.
    move  $\Rightarrow H\_pre \ H\_y\_in$ .
    suff :  $\sim (y2 \leq y < y2 + Z.of\_N \ c2)$ . {
      tauto.
    }
    move  $\Rightarrow [H\_y2\_le \ H\_y\_lt]$ .
    move :  $H\_y\_in \Rightarrow []$ . {
      move  $\Rightarrow [H\_y1\_le] \ H\_y\_lt'$ .
      omega.
    }
  } {
    eapply Nin_elements_greater; eauto.
  }
}

```



```

    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 !ln_lnZ //.
  }
Qed.

```

## elements and elementsZ specification

Lemma elements\_spec1 :  $\forall (s : t) (x : \text{elt}) (Hs : \mathbf{Ok} \ s), \text{List.In } x \ (\text{elements } s) \leftrightarrow \text{In } x \ s$ .

Proof.

```
intros s x Hs.
```

```
by rewrite ln_alt_def.
```

Qed.

Lemma NoDupA\_elementsZ\_single:  $\forall c \ x,$   
 $\mathbf{NoDupA} \ Z.\text{eq} \ (\text{elementsZ\_single } x \ c).$

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 !lnA_alt.
    move => [-] [<-] H_y_in.
    move => [-] [<-] H_y_in'.
    move : H_y_in H_y_in'.
    rewrite ln_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 :  $\forall (s : \mathbf{t}) (Hs : \mathbf{Ok} \ s), \mathbf{NoDupA} \ \mathbf{Z.eq} \ (\mathbf{elementsZ} \ s).$ 
Proof.
  induction s as [| [x c] s' IH]. {
    move  $\Rightarrow$  -.
    rewrite elementsZ_nil.
    apply NoDupA_nil.
  } {
    move  $\Rightarrow$  H_ok_s.
    move : (H_ok_s)  $\Rightarrow$  /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  $\Rightarrow$  y.
      rewrite !InA_alt.
      move  $\Rightarrow$  [-] [<-] H_y_in.
      move  $\Rightarrow$  [-] [<-] H_y_in'.
      move : H_y_in'.
      rewrite -in_rev In_elementsZ_single /=.
      move  $\Rightarrow$  [H_x_le] H_y_lt.
      eapply (Nin_elements_greater s' (x + Z.of_N c))  $\Rightarrow$  //. {
        apply H_s'.
      } {
        apply Z.lt_le_incl, H_y_lt.
      } {
        apply H_y_in.
      }
    }
  }
}
Qed.

```

Lemma elements\_spec2w :  $\forall (s : \mathbf{t}) (Hs : \mathbf{Ok} \ s), \mathbf{NoDupA} \ \mathbf{Enc.E.eq} \ (\mathbf{elements} \ s).$



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 :  $\forall s1\ s2,$   
 $\text{equal } s1\ s2 = \text{true} \leftrightarrow (s1 = s2).$

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 :

$\forall (s\ s' : t) \{Hs : \mathbf{Ok}\ s\} \{Hs' : \mathbf{Ok}\ s'\},$   
 $(\forall x, (\text{InZ } x\ s \leftrightarrow \text{InZ } x\ s')) \rightarrow (s = s').$

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 !ln_lnZ 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 !ln_lnZ.

    suff H_ex : (∀ s'', Ok s'' → lnZ 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  $\Rightarrow H3$ .
      move :  $HH (H3)$ .
      move :  $(H\_ex\ s'\ Hs'\ H3) \Rightarrow [z] \leftarrow$ .
      rewrite  $Enc.decode\_encode\_ok \Rightarrow \leftarrow //$ .
    }
  }
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 :  $\forall\ s1\ s2$ ,  
 (compare  $s1\ s2 = Eq \leftrightarrow equal\ s1\ s2 = true$ ).

Proof.

```

induction s1 as [| [y1 c1] s1' IH];
  case s2 as [| [y2 c2] s2']  $\Rightarrow //$ .
  rewrite /= !andb_true_iff -IH Z.eqb_eq N.eqb_eq.
  move : (Z.compare_eq_iff y1 y2).
  case (Z.compare y1 y2). {
    move  $\Rightarrow H$ .
    have  $\rightarrow : y1 = y2$ . by apply H.
    clear H.

    move : (N.compare_eq_iff c1 c2).
    case (N.compare c1 c2). {
      move  $\Rightarrow H$ .
      have  $\rightarrow : c1 = c2$ . by apply H.
      tauto.
    } {
      move  $\Rightarrow H$ .
      have H_neq :  $\sim(c1 = c2)$ . by rewrite -H  $\Rightarrow \{H\}$ .
      tauto.
    } {
      move  $\Rightarrow H$ .

```

```

      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 :  $\forall s$ ,  
 compare nil s = Lt  $\leftrightarrow$  s  $\neq$  nil.

Proof.

```

  intros s.
  case s ⇒ //=.
  split ⇒ //.

```

Qed.

Lemma compare\_eq\_Lt\_nil\_r :  $\forall s$ ,  
 ~(compare s nil = Lt).

Proof.

```

  intros s.
  case s as [| [y1 c1] s'] ⇒ //=.

```

Qed.

Lemma compare\_eq\_Lt\_cons :  $\forall y1\ y2\ c1\ c2\ s1\ s2$ ,  
 compare ((y1, c1)::s1) ((y2, c2)::s2) = Lt  $\leftrightarrow$   
 (y1 < y2)  $\vee$  ((y1 = y2)  $\wedge$  (c1 < c2)%N)  $\vee$   
 ((y1 = y2)  $\wedge$  (c1 = c2)  $\wedge$  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:  $\forall (s1\ s2 : t)$ ,  
(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 :  $\forall s1\ s2$ ,  
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  $\Rightarrow$  [ $H\_y1\_lt$  | [ $[->]$   $H\_c1\_lt$  |  $[->]$   $[->]$   $H\_comp$ ]]
[ $H\_y2\_lt$  | [ $[-<-]$   $H\_c2\_lt$  |  $[-<-]$   $[-<-]$   $H\_comp'$ ]]. {
  left.
  by apply Z.lt_trans with ( $m := y2$ ).
} {
  by left.
} {
  by left.
} {
  by left.
} {
  right; left.
  split  $\Rightarrow$  //.
  by apply N.lt_trans with ( $m := c2$ ).
} {
  by right; left.
} {
  by left.
} {
  by right; left.
} {
  right; right.
  split  $\Rightarrow$  //.
  split  $\Rightarrow$  //.
  by apply ( $IH\ s2'$ ).
}
Qed.

```

## elements is sorted

Lemma elementsZ\_single\_sorted :  $\forall\ c\ x$ ,  
**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 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 :  $\forall s,$ 
  Ok  $s \rightarrow$ 
  sort  $Enc.E.lt$  (elements  $s$ ).
Proof.
  move  $\Rightarrow s [H\_inv] H\_enc.$ 
  rewrite /elements rev_map_alt_def -map_rev.
  have : ( $\forall x : \mathbf{Z}, List.In\ x\ (\text{rev}\ (\text{elementsZ}\ s)) \rightarrow$ 
     $\exists e : Enc.E.t, Enc.encode\ e = x$ ). {
    move  $\Rightarrow x.$ 
    move : ( $H\_enc\ x$ ).
    rewrite in_rev //.
  }
  move : (elementsZ_sorted  $s\ H\_inv$ )  $\Rightarrow \{H\_enc\}$ .
  generalize (rev (elementsZ  $s$ )).
  induction  $l$  as [|  $x\ xs\ IH$ ]. {
    rewrite /=  $\Rightarrow$  _ ..
    apply Sorted_nil.
  } {
    move  $\Rightarrow H\_sort\ H\_enc.$ 
    apply Sorted_inv in  $H\_sort$  as [ $H\_sort\ H\_hd\_rel$ ].
    simpl.
    apply Sorted_cons. {
      apply  $IH \Rightarrow$  //.
      move  $\Rightarrow xx\ H\_xx\_in.$ 
      apply  $H\_enc.$ 
      by apply in_cons.
    } {
      move :  $H\_hd\_rel\ H\_enc.$ 
      case  $xs \Rightarrow$  //=.
      move  $\Rightarrow x'\ xs'\ H\_hd\_rel\ H\_enc.$ 
      apply HdRel_inv in  $H\_hd\_rel$ .
      apply HdRel_cons.
      rewrite -Enc.encode_lt.
      have [ $y\ H\_y$ ] : ( $\exists y, Enc.encode\ y = x$ ). {
        apply  $H\_enc.$  by left.
      }
      have [ $y'\ H\_y'$ ] : ( $\exists y', Enc.encode\ y' = x'$ ). {
        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 :

$\forall (s : t) (x : \mathbf{Z}),$   
 interval\_list\_invariant  $s = \text{true} \rightarrow$   
 min\_eltZ  $s = \text{Some } x \rightarrow \text{InZ } x \ s.$

Proof.

intros  $s \ x.$   
 case  $s$  as  $[[x' \ c] \ s']$ . {  
 rewrite /min\_eltZ //.  
 } {  
 rewrite /min\_eltZ InZ\_cons interval\_list\_invariant\_cons.  
 move  $\Rightarrow [-] [H\_c\_neq] - [->].$   
 left.  
 by apply In\_elementsZ\_single\_hd.  
 }  
 }

Qed.

Lemma min\_eltZ\_spec2 :

$\forall (s : t) (x \ y : \mathbf{Z}) (Hs : \mathbf{Ok} \ s),$   
 min\_eltZ  $s = \text{Some } x \rightarrow \text{InZ } y \ s \rightarrow \neg \mathbf{Z.lt} \ y \ x.$

Proof.

intros  $s \ x \ y \ H\_ok \ H\_min \ H\_in \ H\_y\_lt\_x.$   
 eapply (Nin\_elements\_greater  $s \ (\mathbf{Z.pred} \ x)) \Rightarrow //$ ; last apply  $H\_in$ . {  
 move :  $H\_ok \ H\_min$ .  
 case  $s \Rightarrow //$ .  
 move  $\Rightarrow [z \ c] \ s' - [<-].$   
 rewrite interval\_list\_elements\_greater\_cons.  
 apply  $\mathbf{Z.lt\_pred\_l}$ .  
 } {  
 apply  $H\_ok$ .  
 } {  
 by apply  $\mathbf{Z.lt\_le\_pred}$ .  
 }  
 }

Qed.

Definition min\_eltZ\_spec3 :

$\forall (s : t),$   
 min\_eltZ  $s = \text{None} \rightarrow \forall x, \neg \text{InZ } x \ s.$

Proof.

intros  $s$ .

```

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

Definition min_elt_spec1 :
   $\forall (s : t) (x : \text{elt}) (Hs : \mathbf{Ok} \ s), \text{min\_elt } s = \text{Some } x \rightarrow \text{In } x \ s.$ 
Proof.
  rewrite /min_elt.
  move => s x H_ok.
  case_eq (min_eltZ s) => //.
  move => z H_min_elt [<-].
  apply lnZ_ln => //.
  apply min_eltZ_spec1 => //.
  apply H_ok.
Qed.

Definition min_elt_spec2 :
   $\forall (s : t) (x \ y : \text{elt}) (Hs : \mathbf{Ok} \ s), \text{min\_elt } s = \text{Some } x \rightarrow \text{In } y \ s \rightarrow \sim(\text{Enc.E.lt } y \ x).$ 
Proof.
  rewrite /min_elt.
  move => s x y H_ok.
  case_eq (min_eltZ s) => //.
  move => z H_min_elt [<-].
  rewrite ln_lnZ => 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 :
   $\forall s : t, \text{min\_elt } s = \text{None} \rightarrow \text{Empty } s.$ 
Proof.
  rewrite /min_elt /min_eltZ /Empty /ln.
  case s as [| [x' c] s'] => //.
  move => _ e.
  rewrite elements_nil lnA_nil //.

```

Qed.

Definition choose\_spec1 :

$\forall (s : t) (x : \text{elt}) (Hs : \text{Ok } s), \text{choose } s = \text{Some } x \rightarrow \text{In } x \ s.$

Proof.

rewrite /choose.

apply min\_elt\_spec1.

Qed.

Definition choose\_spec2 :

$\forall s : t, \text{choose } s = \text{None} \rightarrow \text{Empty } s.$

Proof.

rewrite /choose.

apply min\_elt\_spec3.

Qed.

Lemma choose\_spec3:  $\forall s \ s' \ x \ x', \text{Ok } s \rightarrow \text{Ok } s' \rightarrow$

$\text{choose } s = \text{Some } x \rightarrow \text{choose } s' = \text{Some } x' \rightarrow \text{Equal } s \ s' \rightarrow x = x'.$

Proof.

intros  $s \ s' \ x \ x' \ Hs \ Hs' \ Hx \ Hx'.$

rewrite -equal\_spec equal\_alt\_def  $\Rightarrow H\_s\_eq.$

move :  $Hx \ Hx'.$

rewrite  $H\_s\_eq \Rightarrow \rightarrow [] //$ .

Qed.

Definition max\_eltZ\_spec1 :

$\forall (s : t) (x : \mathbb{Z}),$

$\text{interval\_list\_invariant } s = \text{true} \rightarrow$

$\text{max\_eltZ } s = \text{Some } x \rightarrow \text{InZ } x \ s.$

Proof.

intros  $s \ x.$

induction  $s$  as  $[[x' \ c] \ s' \ IH]. \{$

rewrite /max\_eltZ //.

$\} \{$

rewrite InZ\_cons interval\_list\_invariant\_cons /.

move  $\Rightarrow [-] [H\_c\_neg].$

case  $s'$  as  $[[y' \ c'] \ s'']. \{$

move  $\Rightarrow - [<-].$

left.  $\{$

rewrite In\_elementsZ\_single.

split.  $\{$

rewrite -Z.lt\_le\_pred.

by apply Z\_lt\_add\_r.

$\} \{$

apply Z.lt\_pred\_l.

```

    }
  }
} {
  move  $\Rightarrow H\_inv\ H\_max\_eq$ .
  right.
  by apply IH.
}
}
}
Qed.

Lemma max_eltZ_spec2 :
   $\forall (s : t) (x\ y : \mathbb{Z})$ ,
  interval_list_invariant  $s = \text{true} \rightarrow$ 
  max_eltZ  $s = \text{Some } x \rightarrow \text{InZ } y\ s \rightarrow \neg \mathbb{Z}.lt\ x\ y$ .
Proof.
  induction s as [| [y c] s' IH]. {
    done.
  } {
    move  $\Rightarrow x\ x'$ .
    rewrite interval_list_invariant_cons.
    move  $\Rightarrow [H\_gr]\ [H\_c\_neq\_0]\ H\_inv\_s'$ .
    have  $H\_gr' : (\forall xx : \mathbb{Z}, \text{InZ } xx\ (s') \rightarrow y + \mathbb{Z}.of\_N\ c < xx)$ . {
      apply interval_list_elements_greater_alt2_def  $\Rightarrow //$ .
    }

    case s' as [| [y' c'] s'']. {
      move  $\Rightarrow [ < - ]$ .
      rewrite InZ_cons InZ_nil In_elementsZ_single.
      omega.
    } {
      move  $\Rightarrow H\_max\_eq$ .
      rewrite InZ_cons.
      move  $\Rightarrow []$ ; last by apply IH.
      rewrite In_elementsZ_single.
      move  $\Rightarrow [ - ]\ H\_x'_lt\ H\_lt\_x'$ .

      have  $H\_x\_in : \text{InZ } x\ ((y',\ c') :: s'')$ . {
        by apply max_eltZ_spec1.
      }

      move : ( $H\_gr' - H\_x\_in$ ).
      apply  $\mathbb{Z}.nlt\_ge, \mathbb{Z}.lt\_le\_incl$ .
      by apply  $\mathbb{Z}.lt\_trans$  with ( $m := x'$ ).
    }
  }

```

}

Qed.

Lemma max\_eltZ\_eq\_None :

$\forall (s : t),$   
 $\text{max\_eltZ } s = \text{None} \rightarrow s = \text{nil}.$

Proof.

induction s as [| [x' c] s' IH]  $\Rightarrow$  //.

move : IH.

case s' as [| [y' c'] s'']  $\Rightarrow$  // =.

move  $\Rightarrow$  H H\_pre.

move : (H H\_pre)  $\Rightarrow$  //.

Qed.

Definition max\_eltZ\_spec3 :

$\forall (s : t),$   
 $\text{max\_eltZ } s = \text{None} \rightarrow \forall x, \neg \text{lnZ } x s.$

Proof.

move  $\Rightarrow$  s /max\_eltZ\_eq\_None  $\rightarrow$  x /lnZ\_nil //.

Qed.

Definition max\_elt\_spec1 :

$\forall (s : t) (x : \text{elt}) (Hs : \text{Ok } s), \text{max\_elt } s = \text{Some } x \rightarrow \text{ln } x s.$

Proof.

rewrite /max\_elt.

move  $\Rightarrow$  s x H\_ok.

case\_eq (max\_eltZ s)  $\Rightarrow$  //.

move  $\Rightarrow$  z H\_max\_elt [<-].

apply lnZ\_ln  $\Rightarrow$  //.

apply max\_eltZ\_spec1  $\Rightarrow$  //.

apply H\_ok.

Qed.

Definition max\_elt\_spec2 :

$\forall (s : t) (x y : \text{elt}) (Hs : \text{Ok } s), \text{max\_elt } s = \text{Some } x \rightarrow \text{ln } y s \rightarrow \sim(\text{Enc.E.lt } x y).$

Proof.

rewrite /max\_elt.

move  $\Rightarrow$  s x y H\_ok.

move : (H\_ok)  $\Rightarrow$  [H\_inv] \_.

case\_eq (max\_eltZ s)  $\Rightarrow$  //.

move  $\Rightarrow$  z H\_max\_elt [<-].

rewrite ln\_lnZ  $\Rightarrow$  H\_inZ.

rewrite -Enc.encode\_lt.

apply (max\_eltZ\_spec2 \_ \_ H\_inv)  $\Rightarrow$  //.

suff  $\rightarrow : \text{Enc.encode } (\text{Enc.decode } z) = z \Rightarrow$  //.

```

    apply encode_decode_eq with (s := s) => //.
    apply max_eltZ_spec1 => //.
Qed.

```

Definition max\_elt\_spec3 :

$$\forall s : t, \text{max\_elt } s = \text{None} \rightarrow \text{Empty } s.$$

Proof.

```

    intro s.
    rewrite /max_elt /Empty.
    case_eq (max_eltZ s) => //.
    move => /max_eltZ_eq_None → _ x.
    rewrite /!n elements_nil !nA_nil //.
Qed.

```

## fold specification

Lemma fold\_spec :

$$\forall (s : t) (A : \text{Type}) (i : A) (f : \text{elt} \rightarrow A \rightarrow A),$$

$$\text{fold } f \ s \ i = \text{fold\_left } (\text{flip } f) \ (\text{elements } s) \ i.$$

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 :  $\forall (s : t) (c : \mathbf{N}),$

$$\text{cardinalN } c \ s = (c + \mathbf{N.of\_nat} (\text{length } (\text{elements } s))) \% N.$$

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 :

$$\forall (s : t) (f : \text{elt} \rightarrow \text{bool}) (Hs : \mathbf{Ok} \ s),$$

**Proper** (Enc.E.eq==>eq) f →

$$(\text{exists\_} f \ s = \text{true} \leftrightarrow \text{Exists} (\text{fun } x \Rightarrow f \ x = \text{true}) \ s).$$

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 ( $x : \mathbf{Z}$ ) acc  $c :=$   
 interval\_list\_invariant ( $\text{List.rev (partitionZ\_fold\_skip acc } c)$ ) = true  $\wedge$   
 match  $c$  with  
   None  $\Rightarrow (\forall y', \text{InZ } y' \text{ acc} \rightarrow \mathbf{Z.succ } y' < x)$   
   | Some ( $y, c'$ )  $\Rightarrow (x = y + \mathbf{Z.of\_N } c')$   
 end.

Lemma partitionZ\_aux\_invariant\_insert :  $\forall x \text{ acc } c,$   
 partitionZ\_aux\_invariant  $x \text{ acc } c \rightarrow$   
 partitionZ\_aux\_invariant ( $\mathbf{Z.succ } x$ ) 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 lnZ_rev lnZ_cons lnZ_nil ln_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 lnZ_cons lnZ_nil ln_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 lnZ_cons ln_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  $\Rightarrow [H\_inv] H\_in$ .
  split; first done.
  move  $\Rightarrow y' H\_y\_in$ .
  apply Z.lt.trans with (m := x). {
    by apply H_in.
  } {
    apply Z.lt.succ_diag_r.
  }
} {
  move  $\Rightarrow [y\ c'] [H\_inv] \rightarrow$ .
  split  $\Rightarrow //$ .
  move  $\Rightarrow y'$ .
  rewrite lnZ_cons ln_elementsZ_single.
  move  $\Rightarrow []$ . {
    move  $\Rightarrow [-]$ .
    rewrite -Z.succ.lt.mono //.
  } {
    move : H_inv.
    rewrite /= !interval_list_invariant_app_iff interval_list_invariant_cons.
    move  $\Rightarrow [-] [] [-] [H\_c'\_neq] - H\_pre H\_y\_in$ .
    apply Z.lt.trans with (m := y). {
      apply H_pre. {
        by rewrite lnZ_rev.
      } {
        rewrite lnZ_cons.
        left.
        by apply ln_elementsZ_single_hd.
      }
    }
  }
  apply Z.lt.succ_r, Z_le_add_r.
}
}
}
Qed.

```

```

Definition partitionZ_fold_current (c : option (Z  $\times$  N)) :=
  match c with
  | None  $\Rightarrow$  nil
  | Some yc  $\Rightarrow$  yc : nil
  end.

```

Lemma lnZ\_partitionZ\_fold\_current\_Some :  $\forall\ yc\ y,$

$\text{InZ } y \text{ (partitionZ\_fold\_current (Some } yc)) \leftrightarrow$   
 $\text{InZ } y \text{ (} yc :: \text{nil})$ .

Proof. *done.* Qed.

Lemma  $\text{InZ\_partitionZ\_fold\_insert} : \forall c x y l,$   
 $\text{match } c \text{ with}$   
 $| \text{Some } (y, c') \Rightarrow x = y + \text{Z.of\_N } c'$   
 $| \text{None} \Rightarrow \text{True}$   
 $\text{end} \rightarrow ($   
 $\text{InZ } y \text{ (partitionZ\_fold\_insert } c x :: l) \leftrightarrow$   
 $((x = y) \vee \text{InZ } y \text{ (partitionZ\_fold\_current } c) \vee$   
 $\text{InZ } y l))$ .

Proof.

$\text{intros } c x y l.$   
 $\text{rewrite /partitionZ\_fold\_insert /partitionZ\_fold\_current}$   
 $\text{ /partitionZ\_fold\_skip.}$   
 $\text{case } c. \{$   
 $\text{move } \Rightarrow [y' c'] \rightarrow.$   
 $\text{rewrite !InZ\_cons elementsZ\_single\_succ in\_app\_iff}$   
 $\text{InZ\_nil /}=.$   
 $\text{tauto.}$   
 $\} \{$   
 $\text{rewrite InZ\_cons InZ\_nil In\_elementsZ\_single1.}$   
 $\text{tauto.}$   
 $\}$

Qed.

Lemma  $\text{InZ\_partitionZ\_fold\_skip} : \forall c acc y,$   
 $\text{InZ } y \text{ (partitionZ\_fold\_skip } acc c) \leftrightarrow$   
 $(\text{InZ } y \text{ (partitionZ\_fold\_current } c) \vee \text{InZ } y acc).$

Proof.

$\text{intros } c acc y.$   
 $\text{rewrite /partitionZ\_fold\_skip /partitionZ\_fold\_current}$   
 $\text{ /partitionZ\_fold\_skip.}$   
 $\text{case } c. \{$   
 $\text{move } \Rightarrow [y' c'].$   
 $\text{rewrite !InZ\_cons InZ\_nil /}=.$   
 $\text{tauto.}$   
 $\} \{$   
 $\text{rewrite InZ\_nil.}$   
 $\text{tauto.}$   
 $\}$

Qed.

Lemma  $\text{filterZ\_single\_aux\_props} :$

```

 $\forall f\ c\ x\ acc\ cur,$ 
  partitionZ_aux_invariant  $x\ acc\ cur \rightarrow$ 
  match (filterZ_single_aux  $f\ (acc,\ cur)\ x\ c$ ) with
    ( $acc',\ c'$ )  $\Rightarrow$ 
    let  $r :=$  partitionZ_fold_skip  $acc'\ c'$  in
    interval_list_invariant ( $List.rev\ r$ ) = true  $\wedge$ 
    ( $\forall y',\ InZ\ y'\ r \leftrightarrow (InZ\ y'\ (partitionZ_fold_skip\ acc\ cur) \vee$ 
      ( $f\ y' = true \wedge List.In\ y'\ (elementsZ\_single\ x\ c))))$ 

```

end.

Proof.

```

intro f.
induction c as [| c' IH] using N.peano_ind. {
  intros x acc cur.
  rewrite /partitionZ_aux_invariant.
  move  $\Rightarrow$  [H_inv] -.
  rewrite /filterZ_single_aux fold_elementsZ_single_zero /=.
  tauto.
}
intros x acc cur H_inv.
have  $\rightarrow :$  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  $\Rightarrow acc'\ cur'\ H\_fold\_eq$ .
case_eq (filterZ_single_aux  $f\ (acc',\ cur')\ (Z.succ\ x)\ c'$ ).
move  $\Rightarrow acc'' cur'' H\_succ\_eq$ .
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  $\Rightarrow$  [ $\leftarrow$ ]  $\leftarrow$ . {
    apply partitionZ_aux_invariant_insert.
  } {
    apply partitionZ_aux_invariant_skip.
  }
}
}

move : ( $IH\ (Z.succ\ x)\ acc'\ cur'\ H\_inv'$ )  $\Rightarrow \{IH\}$ .
rewrite  $H\_succ\_eq$  /=.
set  $r :=$  partitionZ_fold_skip  $acc'' cur''$ .
move  $\Rightarrow$  [ $H\_inv\_r$ ]  $H\_in\_r$ .

```

```

split; first assumption.
move ⇒ y'.
move : H_fold_eq.
rewrite H_in_r /filterZ_fold_fun.
case_eq (f x) ⇒ H_fx [←] ←. {
  rewrite lnZ_partitionZ_fold_skip lnZ_partitionZ_fold_current_Some lnZ_partitionZ_fold_skip
elementsZ_single_succ_front.
  rewrite lnZ_partitionZ_fold_insert; last first. {
    move : H_inv.
    rewrite /partitionZ_aux_invariant ⇒ [[]].
    case cur ⇒ //.
  }
  rewrite lnZ_nil /=.
  split; last by tauto.
  move ⇒ []; last by tauto.
  move ⇒ []; last by tauto.
  move ⇒ []. {
    move ⇒ ←.
    tauto.
  } {
    tauto.
  }
} {
  rewrite lnZ_partitionZ_fold_skip /partitionZ_fold_current lnZ_partitionZ_fold_skip ele-
mentsZ_single_succ_front !lnZ_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, lnZ y' acc → Z.succ y' < x) →
  match (filterZ_single f acc x c) with
  r ⇒

```

```

interval_list_invariant (List.rev r) = true ∧
  (∀ y', lnZ y' r ↔ (lnZ 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, lnZ x1 acc → lnZ x2 s → Z.succ x1 < x2) →
    match (filterZ_aux acc f s) with
    r ⇒
      interval_list_invariant r = true ∧
      (∀ y', lnZ y' r ↔ (lnZ y' acc ∨
        (f y' = true ∧ lnZ 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 lnZ_rev lnZ_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, lnZ y' acc → Z.succ y' < y). {
  move ⇒ x1 H_x1_in.
  apply H_in_acc ⇒ //.
  rewrite lnZ_cons.
  by left; apply ln_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,
  lnZ x1 acc' → lnZ x2 s' → Z.succ x1 < x2). {
  move ⇒ x1 x2.
  rewrite H_in_acc' ln_elementsZ_single.
  move ⇒ []. {
    move ⇒ H_x1_in H_x2_in.
    apply H_in_acc ⇒ //.
    rewrite lnZ_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' lnZ_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', lnZ y' r ↔ (f y' = true ∧ lnZ 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, lnZ x1 nil → lnZ 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 lnZ_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 /lsOk /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 →
(ln x (filter f s) ↔ ln x s ∧ f x = true).

```

Proof.

```

move ⇒ s x f H_ok.
suff H_suff :
  (∀ x, (lnZ x (filter f s)) ↔
    lnZ x s ∧ f (Enc.decode x) = true). {
  rewrite !ln_alt_def /elements !rev_map_alt_def
    -!in_rev !in_map_iff.
  setoid_rewrite H_suff.
  rewrite /lnZ.
  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 :  $\forall f \ c \ y \ acc\_t \ c\_t \ acc\_f \ c\_f$ ,  
 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 :  $\forall f \ s \ acc\_t \ acc\_f$ ,  
 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 :

$\forall (s : t) (f : \text{elt} \rightarrow \text{bool}),$

Equal (fst (partition f s)) (filter f s).

Proof.

intros s f.

rewrite partition\_alt\_def /fst /Equal //.

Qed.

Lemma partition\_spec2 :

$\forall (s : t) (f : \text{elt} \rightarrow \text{bool}),$

Ok s  $\rightarrow$

Equal (snd (partition f s)) (filter (fun x  $\Rightarrow$  negb (f x)) s).

Proof.

intros s f.

rewrite partition\_alt\_def /snd /Equal //.

Qed.

End RAW.

### 3.1.5 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 ln (*x* : elt)(*s* : t) := Raw.ln *x* s.(this).

Definition Equal (*s s'* : t) :=  $\forall a : \text{elt}, \text{ln } a s \leftrightarrow \text{ln } a s'$ .

Definition Subset (*s s'* : t) :=  $\forall a : \text{elt}, \text{ln } a s \rightarrow \text{ln } a s'$ .

Definition Empty (*s* : t) :=  $\forall a : \text{elt}, \neg \text{ln } a s$ .

Definition For\_all (*P* : elt  $\rightarrow$  Prop)(*s* : t) :=  $\forall x, \text{ln } x s \rightarrow P x$ .

Definition Exists (*P* : elt  $\rightarrow$  Prop)(*s* : t) :=  $\exists x, \text{ln } x s \wedge P x$ .

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  $(x : \text{elt})(s : t) : t := \text{Mkt} (\text{Raw.remove } x \ s.(\text{this})).$   
 Definition singleton  $(x : \text{elt}) : t := \text{Mkt} (\text{Raw.singleton } x).$   
 Definition union  $(s \ s' : t) : t := \text{Mkt} (\text{Raw.union } s \ s').$   
 Definition inter  $(s \ s' : t) : t := \text{Mkt} (\text{Raw.inter } s \ s').$   
 Definition diff  $(s \ s' : t) : t := \text{Mkt} (\text{Raw.diff } s \ s').$   
 Definition equal  $(s \ s' : t) := \text{Raw.equal } s \ s'.$   
 Definition subset  $(s \ s' : t) := \text{Raw.subset } s \ s'.(\text{this}).$   
 Definition empty  $: t := \text{Mkt } \text{Raw.empty}.$   
 Definition is\_empty  $(s : t) := \text{Raw.is\_empty } s.$   
 Definition elements  $(s : t) : \text{list } \text{elt} := \text{Raw.elements } s.$   
 Definition min\_elt  $(s : t) : \text{option } \text{elt} := \text{Raw.min\_elt } s.$   
 Definition max\_elt  $(s : t) : \text{option } \text{elt} := \text{Raw.max\_elt } s.$   
 Definition choose  $(s : t) : \text{option } \text{elt} := \text{Raw.choose } s.$   
 Definition compare  $(s1 \ s2 : t) : \text{comparison} := \text{Raw.compare } s1 \ s2.$   
 Definition fold  $\{A : \text{Type}\}(f : \text{elt} \rightarrow A \rightarrow A)(s : t) : A \rightarrow A := \text{Raw.fold } f \ s.$   
 Definition cardinal  $(s : t) := \text{Raw.cardinal } s.$   
 Definition filter  $(f : \text{elt} \rightarrow \text{bool})(s : t) : t := \text{Mkt} (\text{Raw.filter } f \ s).$   
 Definition for\_all  $(f : \text{elt} \rightarrow \text{bool})(s : t) := \text{Raw.for\_all } f \ s.$   
 Definition exists\_  $(f : \text{elt} \rightarrow \text{bool})(s : t) := \text{Raw.exists\_} f \ s.$   
 Definition partition  $(f : \text{elt} \rightarrow \text{bool})(s : t) : t \times t :=$   
     let  $p := \text{Raw.partition } f \ s$  in  $(\text{Mkt} (\text{fst } p), \text{Mkt} (\text{snd } p)).$   
 Instance In\_compat : **Proper**  $(E.\text{eq} ==> \text{eq} ==> \text{iff})$  In.  
 Proof.  
     repeat red.  
     move  $\Rightarrow x \ y \ H.\text{eq\_xy} \ x' \ y' \rightarrow.$   
     rewrite /In /Raw.In.  
     setoid\_rewrite  $H.\text{eq\_xy}.$   
     done.  
 Qed.  
 Definition eq  $: t \rightarrow t \rightarrow \text{Prop} := \text{Equal}.$   
 Instance eq\_equiv : **Equivalence** eq.  
 Proof. firstorder. Qed.  
 Definition eq\_dec  $: \forall (s \ s' : t), \{ \text{eq } s \ s' \} + \{ \neg \text{eq } s \ s' \}.$   
 Proof.  
     intros  $(s, Hs) (s', Hs').$   
     change  $(\{ \text{Raw.Equal } s \ s' \} + \{ \neg \text{Raw.Equal } s \ s' \}).$   
     destruct  $(\text{Raw.equal } s \ s') \text{ eqn} : H; [left|right];$   
     rewrite  $\leftarrow \text{Raw.equal\_spec};$  congruence.  
 Defined.  
 Definition lt  $: t \rightarrow t \rightarrow \text{Prop} := \text{Raw.lt}.$   
 Instance lt\_strorder : **StrictOrder** lt.

Proof.

```
unfold lt.
constructor. {
  move : Raw.lt_irreflexive.
  rewrite /Irreflexive /complement /Reflexive.
  move  $\Rightarrow$   $H$   $x$ .
  apply  $H$ .
} {
  move : Raw.lt_transitive.
  rewrite /Transitive.
  move  $\Rightarrow$   $H$   $x$   $y$   $z$ .
  apply  $H$ .
}
```

Qed.

Instance lt\_compat : Proper (eq $\Rightarrow$ eq $\Rightarrow$ iff) lt.

Proof.

```
repeat red.
move  $\Rightarrow$  [ $x1$   $H_{x1\_ok}$ ] [ $y1$   $H_{y1\_ok}$ ]  $H_{eq}$ .
move  $\Rightarrow$  [ $x2$   $H_{x2\_ok}$ ] [ $y2$   $H_{y2\_ok}$ ].
move :  $H_{eq}$ .
rewrite /eq /lt /Equal /ln /=.
replace ( $\forall a : \text{elt}, \text{Raw.ln } a \ x1 \leftrightarrow \text{Raw.ln } a \ y1$ ) with
  ( $\text{Raw.Equal } x1 \ y1$ ) by done.
replace ( $\forall a : \text{elt}, \text{Raw.ln } a \ x2 \leftrightarrow \text{Raw.ln } a \ y2$ ) with
  ( $\text{Raw.Equal } x2 \ y2$ ) by done.
rewrite -!Raw.equal_spec !Raw.equal_alt_def.
move  $\Rightarrow \rightarrow \rightarrow //$ .
```

Qed.

Section Spec.

Variable  $s \ s' : t$ .

Variable  $x \ y : \text{elt}$ .

Variable  $f : \text{elt} \rightarrow \text{bool}$ .

Notation compatb := (Proper (E.eq $\Rightarrow$ Logic.eq)) (*only parsing*).

Lemma mem\_spec : mem  $x$   $s$  = true  $\leftrightarrow$  ln  $x$   $s$ .

Proof. exact (Raw.mem\_spec \_ \_ \_). Qed.

Lemma equal\_spec : equal  $s \ s'$  = true  $\leftrightarrow$  Equal  $s \ s'$ .

Proof. rewrite Raw.equal\_spec //. Qed.

Lemma subset\_spec : subset  $s \ s'$  = true  $\leftrightarrow$  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  $\leftrightarrow$  Empty  $s$ .

Proof. rewrite Raw.is\_empty\_spec //. Qed.  
 Lemma add\_spec :  $\text{In } y \text{ (add } x \text{ s)} \leftrightarrow E.eq \ y \ x \vee \text{In } y \text{ s}$ .  
 Proof. exact (Raw.add\_spec \_ \_ \_). Qed.  
 Lemma remove\_spec :  $\text{In } y \text{ (remove } x \text{ s)} \leftrightarrow \text{In } y \text{ s} \wedge \neg E.eq \ y \ x$ .  
 Proof. exact (Raw.remove\_spec \_ \_ \_). Qed.  
 Lemma singleton\_spec :  $\text{In } y \text{ (singleton } x) \leftrightarrow E.eq \ y \ x$ .  
 Proof. exact (Raw.singleton\_spec \_). Qed.  
 Lemma union\_spec :  $\text{In } x \text{ (union } s \text{ s')} \leftrightarrow \text{In } x \text{ s} \vee \text{In } x \text{ s'}$ .  
 Proof. exact (Raw.union\_spec \_ \_ \_). Qed.  
 Lemma inter\_spec :  $\text{In } x \text{ (inter } s \text{ s')} \leftrightarrow \text{In } x \text{ s} \wedge \text{In } x \text{ s'}$ .  
 Proof. exact (Raw.inter\_spec \_ \_ \_). Qed.  
 Lemma diff\_spec :  $\text{In } x \text{ (diff } s \text{ s')} \leftrightarrow \text{In } x \text{ s} \wedge \neg \text{In } x \text{ s'}$ .  
 Proof. exact (Raw.diff\_spec \_ \_ \_). Qed.  
 Lemma fold\_spec :  $\forall (A : \text{Type}) (i : A) (f : \text{elt} \rightarrow A \rightarrow A),$   
      $\text{fold } f \text{ s } i = \text{fold\_left } (\text{fun } a \text{ e} \Rightarrow f \text{ e } a) \text{ (elements } s) \text{ } i$ .  
 Proof. exact (@Raw.fold\_spec \_). Qed.  
 Lemma cardinal\_spec :  $\text{cardinal } s = \text{length } (\text{elements } s)$ .  
 Proof. exact (@Raw.cardinal\_spec s). Qed.  
 Lemma filter\_spec :  $\text{compatb } f \rightarrow$   
      $(\text{In } x \text{ (filter } f \text{ s)} \leftrightarrow \text{In } x \text{ s} \wedge f \text{ x} = \text{true})$ .  
 Proof. move  $\Rightarrow$  \_; exact (@Raw.filter\_spec \_ \_ \_). Qed.  
 Lemma for\_all\_spec :  $\text{compatb } f \rightarrow$   
      $(\text{for\_all } f \text{ s} = \text{true} \leftrightarrow \text{For\_all } (\text{fun } x \Rightarrow f \text{ x} = \text{true}) \text{ s})$ .  
 Proof. exact (@Raw.for\_all\_spec \_ \_). Qed.  
 Lemma exists\_spec :  $\text{compatb } f \rightarrow$   
      $(\text{exists\_ } f \text{ s} = \text{true} \leftrightarrow \text{Exists } (\text{fun } x \Rightarrow f \text{ x} = \text{true}) \text{ s})$ .  
 Proof. exact (@Raw.exists\_spec \_ \_). Qed.  
 Lemma partition\_spec1 :  $\text{compatb } f \rightarrow \text{Equal } (\text{fst } (\text{partition } f \text{ s})) \text{ (filter } f \text{ s})$ .  
 Proof. move  $\Rightarrow$  \_; exact (@Raw.partition\_spec1 \_ \_). Qed.  
 Lemma partition\_spec2 :  $\text{compatb } f \rightarrow$   
      $\text{Equal } (\text{snd } (\text{partition } f \text{ s})) \text{ (filter } (\text{fun } x \Rightarrow \text{negb } (f \text{ x})) \text{ s})$ .  
 Proof. move  $\Rightarrow$  \_; exact (@Raw.partition\_spec2 \_ \_). Qed.  
 Lemma elements\_spec1 :  $\text{InA } E.eq \ x \text{ (elements } s) \leftrightarrow \text{In } x \text{ s}$ .  
 Proof. rewrite /In /Raw.In /elements //. Qed.  
 Lemma elements\_spec2w :  $\text{NoDupA } E.eq \text{ (elements } s)$ .  
 Proof. exact (Raw.elements\_spec2w \_). Qed.  
 Lemma elements\_spec2 :  $\text{sort } E.lt \text{ (elements } s)$ .  
 Proof. exact (Raw.elements\_sorted \_). Qed.  
 Lemma choose\_spec1 :  $\text{choose } s = \text{Some } x \rightarrow \text{In } x \text{ s}$ .  
 Proof. exact (Raw.choose\_spec1 \_ \_). Qed.  
 Lemma choose\_spec2 :  $\text{choose } s = \text{None} \rightarrow \text{Empty } s$ .  
 Proof. exact (Raw.choose\_spec2 \_). Qed.



Lemma choose\_spec3 : choose  $s = \text{Some } x \rightarrow \text{choose } s' = \text{Some } y \rightarrow$   
 Equal  $s s' \rightarrow E.\text{eq } x y$ .

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  $s = \text{Some } x \rightarrow \text{In } x s$ .

Proof. exact (Raw.min\_elt\_spec1 \_ \_ \_). Qed.

Lemma min\_elt\_spec2 : min\_elt  $s = \text{Some } x \rightarrow \text{In } y s \rightarrow \neg E.\text{lt } y x$ .

Proof. exact (Raw.min\_elt\_spec2 \_ \_ \_ \_). Qed.

Lemma min\_elt\_spec3 : choose  $s = \text{None} \rightarrow \text{Empty } s$ .

Proof. exact (Raw.min\_elt\_spec3 \_). Qed.

Lemma max\_elt\_spec1 : max\_elt  $s = \text{Some } x \rightarrow \text{In } x s$ .

Proof. exact (Raw.max\_elt\_spec1 \_ \_ \_). Qed.

Lemma max\_elt\_spec2 : max\_elt  $s = \text{Some } x \rightarrow \text{In } y s \rightarrow \neg E.\text{lt } x y$ .

Proof. exact (Raw.max\_elt\_spec2 \_ \_ \_ \_). Qed.

Lemma max\_elt\_spec3 : max\_elt  $s = \text{None} \rightarrow \text{Empty } s$ .

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.

### 3.1.6 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:  $\forall (e : E.t),$ 
    decode (encode e) = e.
  Proof. by []. Qed.
  Lemma encode_eq :  $\forall (e1\ e2 : E.t),$ 
    (Z.eq (encode e1) (encode e2))  $\leftrightarrow$  E.eq e1 e2.
  Proof. by []. Qed.
  Lemma encode_lt :  $\forall (e1\ e2 : E.t),$ 
    (Z.lt (encode e1) (encode e2))  $\leftrightarrow$  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:  $\forall (e : E.t),$ 
    decode (encode e) = e.
  Proof.
    intros e.
    rewrite /encode /decode N2Z.id //.
  Qed.
  Lemma encode_eq :  $\forall (e1\ e2 : E.t),$ 
    (Z.eq (encode e1) (encode e2))  $\leftrightarrow$  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 := 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 NAT := MSetIntervals ELEMENTENCODENAT.

```

# Chapter 4

## Library

## MSetsExtra.MSetFoldWithAbort

### 4.1 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 mathcomp.ssreflect.ssreflect.
Require Import MSetWithDups.
Require Import Int.
Require Import MSetGenTree MSetAVL MSetRBT.
Require Import MSetList MSetWeakList.
```

#### 4.1.1 Fold With Abort Operations

We want to provide an efficient folding operation. Efficiency 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 :=  
 $(elt \rightarrow A \rightarrow A) \rightarrow f$        $(elt \rightarrow A \rightarrow \text{bool}) \rightarrow f\_abort$        $t \rightarrow$  input set       $A$   
 $\rightarrow$  base value       $A$ .

**Definition** `foldWithAbortSpecPred`  $\{elt\ t : \text{Type}\}$

$(In : elt \rightarrow t \rightarrow \text{Prop})$   
 $(fold : \forall \{A : \text{Type}\}, (elt \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$   
 $(foldWithAbort : \forall \{A : \text{Type}\}, foldWithAbortType\ elt\ t\ A) : \text{Prop} :=$

∀

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

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

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

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

$f\_abort$  is OK, i.e. all other elements can be skipped without leaving the equivalence relation.       $(\forall e1 \text{ st},$

$\text{In } e1 \text{ st} \rightarrow f\_abort \ e1 \text{ st} = \text{true} \rightarrow$   
 $(\forall \text{ st' } e2, P \text{ st st'} \rightarrow$   
 $\text{In } e2 \text{ st} \rightarrow e2 \neq e1 \rightarrow$   
 $P \text{ st} (f \ e2 \text{ st'}))) \rightarrow$

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

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

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

$\text{elt}$  element type of set       $t$  type of set       $A$  return type :=  
 $(\text{elt} \rightarrow A \rightarrow A) \rightarrow f$        $(\text{elt} \rightarrow A \rightarrow \text{bool}) \rightarrow f\_gt$        $t \rightarrow$  input set       $A \rightarrow$   
 base value       $A$ .

**Definition** foldWithAbortGtSpecPred {elt t : Type}

$(lt : \text{elt} \rightarrow \text{elt} \rightarrow \text{Prop})$   
 $(In : \text{elt} \rightarrow t \rightarrow \text{Prop})$   
 $(\text{fold} : \forall \{A : \text{Type}\}, (\text{elt} \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$

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

$\forall$   
 $(A : Type)$   
 result type  
 $(i i' : A)$   
 base values for foldWithAbort and fold  
 $(f : elt \rightarrow A \rightarrow A) (f' : elt \rightarrow A \rightarrow A)$   
 fold functions for foldWithAbort and fold  
 $(f\_gt : elt \rightarrow A \rightarrow \text{bool})$   
 abort function  
 $(s : t)$  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  $P$        $(\forall st st' e,$   
 $In e s \rightarrow P st st' \rightarrow P (f e st) (f e st')) \rightarrow$

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

$f\_abort$  is OK, i.e. all other 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       $P (foldWithAbortGt f f\_gt s i) (fold f' s i')$ .

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 :=  
 $(elt \rightarrow A \rightarrow \text{bool}) \rightarrow f\_lt$        $(elt \rightarrow A \rightarrow A) \rightarrow f$        $(elt \rightarrow A \rightarrow \text{bool}) \rightarrow f\_gt$   
 $t \rightarrow$  input set       $A \rightarrow$  base value       $A$ .

Definition  $\text{foldWithAbortGtLtSpecPred } \{elt\ t : \text{Type}\}$

$(lt : elt \rightarrow elt \rightarrow \text{Prop})$   
 $(In : elt \rightarrow t \rightarrow \text{Prop})$   
 $(fold : \forall \{A : \text{Type}\}, (elt \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$   
 $(foldWithAbortGtLt : \forall \{A : \text{Type}\}, \text{foldWithAbortGtLtType } elt\ t\ A) : \text{Prop} :=$

$\forall$

$(A : \text{Type})$   
 result type  
 $(i\ i' : A)$   
 base values for  $\text{foldWithAbort}$  and  $\text{fold}$   
 $(f : elt \rightarrow A \rightarrow A)\ (f' : elt \rightarrow A \rightarrow A)$   
 fold functions for  $\text{foldWithAbort}$  and  $\text{fold}$   
 $(f\_lt\ f\_gt : elt \rightarrow A \rightarrow \text{bool})$   
 abort functions  
 $(s : t)$  sets to fold over  
 $(P : A \rightarrow A \rightarrow \text{Prop})$  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 :=

$(elt \rightarrow B) \rightarrow f\_precompute$        $(elt \rightarrow B \rightarrow A \rightarrow \text{bool}) \rightarrow f\_lt$        $(elt \rightarrow B \rightarrow A \rightarrow A) \rightarrow f$        $(elt \rightarrow B \rightarrow A \rightarrow \text{bool}) \rightarrow f\_gt$        $t \rightarrow$  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`  $\{elt \ t : \text{Type}\}$

$(lt : elt \rightarrow elt \rightarrow \text{Prop})$   
 $(In : elt \rightarrow t \rightarrow \text{Prop})$   
 $(fold : \forall \{A : \text{Type}\}, (elt \rightarrow A \rightarrow A) \rightarrow t \rightarrow A \rightarrow A)$   
 $(foldWithAbortPrecompute : \forall \{A \ B : \text{Type}\}, foldWithAbortPrecomputeType \ elt \ t \ A \ B)$   
 $: \text{Prop} :=$

$\forall$

$(A \ B : \text{Type})$   
 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`  
 $(f\_lt \ f\_gt : elt \rightarrow B \rightarrow A \rightarrow \text{bool})$   
 abort functions  
 $(s : t)$  sets to fold over  
 $(P : A \rightarrow A \rightarrow \text{Prop})$  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.  
 $(\forall e1\ st,$

$In\ e1\ s \rightarrow f\_lt\ e1\ (f\_precompute\ e1)\ st = \text{true} \rightarrow$   
 $(\forall st'\ e2, P\ st\ st' \rightarrow$   
 $In\ e2\ s \rightarrow lt\ e2\ e1 \rightarrow$   
 $P\ st\ (f\ e2\ (f\_precompute\ e2)\ st')) \rightarrow$

$f\_gt$  is OK, i.e. greater elements can be skipped without leaving the equivalence relation.  
 $(\forall e1\ st,$

$In\ e1\ s \rightarrow f\_gt\ e1\ (f\_precompute\ e1)\ st = \text{true} \rightarrow$   
 $(\forall st'\ e2, P\ st\ st' \rightarrow$   
 $In\ e2\ s \rightarrow lt\ e1\ e2 \rightarrow$   
 $P\ st\ (f\ e2\ (f\_precompute\ e2)\ st')) \rightarrow$

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` ( $E : \text{ORDEREDTYPE}$ ) (`Import C : WSETSONWITHDUPS`  $E$ ).

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

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

**End** `HASFOLDWITHABORT`.

### 4.1.2 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 :  $\forall (A : \text{Type}) (i : A)$   
 $(f : \text{elt} \rightarrow A \rightarrow A) (f' : \text{elt} \rightarrow A \rightarrow A) (f\_lt f\_gt : \text{elt} \rightarrow A \rightarrow \text{bool}) (s : t),$

$(\forall e \text{ st}, \text{In } e \text{ s} \rightarrow (f \text{ e st} = f' \text{ e st})) \rightarrow$

$(\forall e1 \text{ st},$   
 $\text{In } e1 \text{ s} \rightarrow f\_lt \text{ e1 st} = \text{true} \rightarrow$   
 $(\forall e2, \text{In } e2 \text{ s} \rightarrow E.lt \text{ e2 e1} \rightarrow$   
 $(f \text{ e2 st} = \text{st}))) \rightarrow$

$(\forall e1 \text{ st},$   
 $\text{In } e1 \text{ s} \rightarrow f\_gt \text{ e1 st} = \text{true} \rightarrow$   
 $(\forall e2, \text{In } e2 \text{ s} \rightarrow E.lt \text{ e1 e2} \rightarrow$   
 $(f \text{ e2 st} = \text{st}))) \rightarrow$

$(\text{foldWithAbortGtLt } f\_lt \text{ f } f\_gt \text{ s } i) = (\text{fold } f' \text{ s } i).$

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, ln e s → (f e st = f' e st)) →

  (∀ e1 st,
    ln e1 s → f_gt e1 st = true →
    (∀ e2, ln 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 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, ln e s → (f e st = f' e st)) →

  (∀ e1 st,
    ln e1 s → f_abort e1 st = true →
    (∀ e2, ln 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 or 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*,  $\exists$ ,  $\forall$  and more.

Record **FoldWithAbortSpecArg**  $\{B\} := \{$   
 $\text{fwasa\_f\_pre} : (\text{elt} \rightarrow B);$  The precompute function  $\text{fwasa\_f\_lt} : (\text{elt} \rightarrow B \rightarrow$   
 $\text{bool});$   $\text{f\_lt}$  without state argument  $\text{fwasa\_f\_gt} : (\text{elt} \rightarrow B \rightarrow \text{bool});$   $\text{f\_gt}$  without  
state argument  $\text{fwasa\_P}' : (\text{elt} \rightarrow B \rightarrow \text{bool})$  the predicate  $P \quad \}$ .

*foldWithAbortSpecForPred*  $s P \text{fwasa}$  holds, if the argument *fwasa* fits the predicate  $P$  for set  $s$ .

Definition **foldWithAbortSpecArgsForPred**  $\{A : \text{Type}\}$   
 $(s : t) (P : \text{elt} \rightarrow \text{bool}) (\text{fwasa} : @\text{FoldWithAbortSpecArg } A) :=$

the predicate  $P'$  coincides for  $s$  and the given precomputation with  $P$   $(\forall e, \text{In } e s \rightarrow (\text{fwasa\_P}' \text{fwasa } e (\text{fwasa\_f\_pre } \text{fwasa } e) = P e)) \wedge$

If  $\text{fwasa\_f\_lt}$  holds, all elements smaller than the current one don't satisfy predicate  $P$ .  $(\forall e1,$

$\text{In } e1 s \rightarrow \text{fwasa\_f\_lt } \text{fwasa } e1 (\text{fwasa\_f\_pre } \text{fwasa } e1) = \text{true} \rightarrow$   
 $(\forall e2, \text{In } e2 s \rightarrow E.\text{lt } e2 e1 \rightarrow (P e2 = \text{false}))) \wedge$

If  $\text{fwasa\_f\_gt}$  holds, all elements greater than the current one don't satisfy predicate  $P$ .  $(\forall e1,$

$\text{In } e1 s \rightarrow \text{fwasa\_f\_gt } \text{fwasa } e1 (\text{fwasa\_f\_pre } \text{fwasa } e1) = \text{true} \rightarrow$   
 $(\forall e2, \text{In } e2 s \rightarrow E.\text{lt } e1 e2 \rightarrow (P e2 = \text{false}))).$

## Filter with abort

Definition **filter\_with\_abort**  $\{B\} (\text{fwasa} : @\text{FoldWithAbortSpecArg } B) s :=$

*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} :  $\forall$  fwasa P s,  
 @foldWithAbortSpecArgsForPred B s P fwasa  $\rightarrow$   
 Proper (E.eq ==> Logic.eq) P  $\rightarrow$   
 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 -> :  $\forall$  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)  $\wedge$  P e = true)  $\vee$  (In e acc). {
  rewrite elements_spec1.
```

```

    suff : (¬ln 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} (fwasas : @FoldWithAbortSpecArg B) s :=  
 foldWithAbortPrecompute (fwasas\_f\_pre fwasas)  
 (fun e p st ⇒ match st with None ⇒ (fwasas\_f\_lt fwasas e p) | \_ ⇒ true end)  
 (fun e e\_pre st ⇒ match st with None ⇒  
 if (fwasas\_P' fwasas e e\_pre) then Some e else None | \_ ⇒ st end)  
 (fun e p st ⇒ match st with None ⇒ (fwasas\_f\_gt fwasas e p) | \_ ⇒ true end)  
 s None.

Lemma choose\_with\_abort\_spec {B} : ∀ fwasas P s,  
 @foldWithAbortSpecArgsForPred B s P fwasas →  
 Proper (E.eq ==> Logic.eq) P →  
 (match (choose\_with\_abort fwasas s) with  
 | None ⇒ (∀ e, ln e s → P e = false)  
 | Some e ⇒ ln 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 fwasas := {
  fwasas_f_pre := f_pre;
  fwasas_f_lt := f_lt;
  fwasas_f_gt := f_gt;
  fwasas_P' := P' }.
suff : (match (choose_with_abort fwasas 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 fwasas s). {
    move ⇒ e.
    rewrite elements_spec1 //.
  } {
    move ⇒ H e H_in.
    apply H.
    rewrite elements_spec1 //.
  }
}

have → : (choose_with_abort fwasas 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} (fwasas : @FoldWithAbortSpecArg B) s :=  
 match choose\_with\_abort fwasas s with  
 | None => false  
 | Some \_ => true  
end.

Lemma exists\_with\_abort\_spec {B} : ∀ fwasas P s,  
 @foldWithAbortSpecArgsForPred B s P fwasas →  
 Proper (E.eq ==> Logic.eq) P →  
 (exists\_with\_abort fwasas s =  
 exists\_ P s).

Proof.

```
intros fwasas P s H_fwasas H_proper.  
apply Logic.eq_sym.  
rewrite /exists_with_abort.  
move : (choose_with_abort_spec _ _ _ H_fwasas H_proper).  
case (choose_with_abort fwasas 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} fwasas s :=  
 negb (@exists\_with\_abort B fwasas s).

Lemma forall\_with\_abort\_spec {B} : ∀ fwasas s P,  
 @foldWithAbortSpecArgsForPred B s P fwasas →  
 Proper (E.eq ==> Logic.eq) P →  
 (forall\_with\_abort fwasas s =  
 for\_all (fun e => negb (P e)) s).

Proof.

```
intros fwasas 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  $\Rightarrow$   $H\_pre\ x\ H\_x\_in$ .
  rewrite negb_true_iff -not_true_iff_false  $\Rightarrow$   $H\_Px$ .
  apply  $H\_pre$ .
  by  $\exists\ x$ .
} {
  move  $\Rightarrow$   $H\_pre\ [x]\ [H\_x\_in]\ H\_P\_x$ .
  move : ( $H\_pre\ x\ H\_x\_in$ ).
  rewrite  $H\_P\_x$ .
  done.
}
Qed.

```

End HASFOLDWITHABORTOPS.

### 4.1.3 Modules Types For Sets with Fold with Abort

```

Module Type WSETSWITHDUPSFLDA.
  Declare Module E : ORDEREDTYPE.
  Include WSETSONWITHDUPS E.
  Include HASFOLDWITHABORT E.
  Include HASFOLDWITHABORTOPS E.
End WSETSWITHDUPSFLDA.

Module Type WSETSWITHFLDA <: WSETS.
  Declare Module E : ORDEREDTYPE.
  Include WSETSON E.
  Include HASFOLDWITHABORT E.
  Include HASFOLDWITHABORTOPS E.
End WSETSWITHFLDA.

Module Type SETSWITHFLDA <: SETS.
  Declare Module E : ORDEREDTYPE.
  Include SETSON E.
  Include HASFOLDWITHABORT E.
  Include HASFOLDWITHABORTOPS E.
End SETSWITHFLDA.

```

## 4.1.4 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.ln e s → P st st' → P (f e (f_pre e) st) (f e (f_pre e) st')) →`

`(∀ e st, M.ln e s → P (f e (f_pre e) st) (f' e st)) →`

`(∀ e1 st,`

`M.ln e1 s → f_lt e1 (f_pre e1) st = true →`

`(∀ st' e2, P st st' →`

`M.ln e2 s → E.lt e2 e1 →`

`P st (f e2 (f_pre e2) st')))) →`

`(∀ e1 st,`

`M.ln e1 s → f_gt e1 (f_pre e1) st = true →`

`(∀ st' e2, P st st' →`

`M.ln e2 s → E.lt e1 e2 →`

$$P \text{ st } (f \text{ e2 } (f\_pre \text{ e2}) \text{ st}') \rightarrow$$

$$P \text{ i } i' \rightarrow$$

$$P \text{ (foldWithAbort\_Raw } f\_pre \text{ f\_lt f f\_gt s i) (fold f' s i')}.$$

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.ln e base → M.ln e s). {
  rewrite /ln /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.ln e0 l → M.ln e0 s ∧ E.lt e0 e). {
    intros e0 H_in_l.
    split; last by apply H_lt_tree_l.
    eapply H_sub.
    rewrite /M.ln M.ln_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.ln e0 r → M.ln e0 s ∧ E.lt e e0). {
    intros e0 H_in_r.
    split; last by apply H_gt_tree_r.
    eapply H_sub.
    rewrite /M.ln M.ln_node_iff.
    tauto.
  }
}

```

```

}
move : (IHr H_bst_r) ⇒ {IHr} IHr {H_bst_r} {H_gt_tree_r}.
have H_in_e : M.ln e s. {
  eapply H_sub.
  rewrite /M.ln M.ln_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.ln M.ln_node_iff. tauto.
      }
      have [] : (M.ln e' s ∧ E.lt e' e). {
        apply H_sub_l.
        rewrite /M.ln M.ln_node_iff.
        right; left.
        apply Equivalence_Reflexive.
      }
    }
  }

```

```

    move  $\Rightarrow$   $H\_e\_in\ H\_lt\_in$ .
    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  $H\_RL \Rightarrow //$ .
    apply  $IHl$ ; last first. {
      move  $\Rightarrow y\ H\_y\_in$ .
      apply  $H\_sub\_l$ .
      rewrite  $/M.ln\ M.ln\_node\_iff$ . tauto.
    }
    assumption.
  }
} {
  move  $\Rightarrow$  -.
  apply  $IHl \Rightarrow //$ .
  eapply  $H\_sub\_l$ .
}
}
have  $H\_P\_st1 : P\ st1\ st1'$ . {
  rewrite  $/st1\ /st1'$ .
  rewrite  $-H\_f'$  //.
  apply  $H\_P\_f \Rightarrow //$ .
}
have  $H\_P\_st2 : P\ st2\ st2'$ . {
  rewrite  $/st2\ /st2'$ .
  clearbody  $st1\ st1'$ .
  case_eq  $(f\_gt\ e\ (f\_pre\ e)\ st1)$ . {
    move  $\Rightarrow H\_gt\_eq$ .
    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  $\Rightarrow H\_equiv\_P\ H\_RG\ H\_f'$ .
    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  $\Rightarrow y\ H\_y\_in$ .
      apply  $H\_sub\_r$ .
    }
  }
}

```



```

      rewrite /M.ln M.ln_node_iff. tauto.
    }
    have [] : (M.ln e' s  $\wedge$  E.lt e e'). {
      apply H_sub_r.
      rewrite /M.ln M.ln_node_iff.
      right; left.
      apply Equivalence_Reflexive.
    }
    move  $\Rightarrow$  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  $\Rightarrow$  //.
    apply IHl; last first. {
      move  $\Rightarrow$  y H_y_in.
      apply H_sub_r.
      rewrite /M.ln M.ln_node_iff. tauto.
    }
    assumption.
  }
} {
  move  $\Rightarrow$  -.
  apply IHR  $\Rightarrow$  //.
  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}  $f\_pre$   $f\_lt$  ( $f$ :  $\text{elt} \rightarrow B \rightarrow A \rightarrow A$ )  $f\_gt$   
 $t$  ( $base$ :  $A$ ):  $A :=$   
foldWithAbort\_Raw  $f\_pre$   $f\_lt$   $f$   $f\_gt$  ( $t$ .(this))  $base$ .

Lemma foldWithAbortPrecomputeSpec : foldWithAbortPrecomputeSpecPred X.lt In fold (@fold-  
WithAbortPrecompute).

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 (@fold-  
WithAbortPrecompute).

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.

## 4.1.5 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 WRRAW2SETSON 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_xxs H_nodup_xxs {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_xxs : (∀ 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_xs_xs.
    rewrite InA_alt.
    by ∃ e.
  } {
    by left.
  } {
    by right.
  }
}
move : H_in_xs H_neq_xs.

clear IH H_in_xs 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_xs' : 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_xs_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_xs')  $\Rightarrow$  [] H_cmp. {
      apply H_RG_x  $\Rightarrow$  //.
    } {
      apply H_RL_x  $\Rightarrow$  //.
    }
  }
}
}
}
Qed.

```

Include HASFOLDWITHABORTOPS X.

End MAKEWEAKLISTSETSWITHFOLDA.