Contents

Chapter 1

Library 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` 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 In : elt → t → Prop.
Declare Instance In_compat : Proper (E.eq==>eq==>iff) In.

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) s := ∃ x , In x s ∧ P x.

Notation "s [=] t" := (Equal s t) (at level 70, no associativity).
Notation "s [<=] t" := (Subset s t) (at level 70, no associativity).

Definition eq : t → t → Prop := Equal.
Include IsEq.   eq is obviously an equivalence, for subtyping only    Include HasEqDec.

Section Spec.
Variable s s': t.
Variable x y : elt.
Variable f : elt → bool.
Notation compatb := (Proper (E.eq==>Logic.eq)) (only parsing).

Parameter mem_spec : mem x s = true ↔ In x s.
Parameter equal_spec : equal s s' = true ↔ s[=]s'.
Parameter subset_spec : subset s s' = true ↔ s[<=]s'.
Parameter empty_spec : Empty empty.
Parameter is_empty_spec : is_empty s = true ↔ Empty s.
Parameter add_spec : In y (add x s) ↔ E.eq y x ∨ In y s.
Parameter remove_spec : In y (remove x s) ↔ In y s ∧ ¬E.eq y x.
Parameter singleton_spec : In y (singleton x) ↔ E.eq y x.
Parameter union_spec : In x (union s s') ↔ In x s ∨ In x s'.
Parameter inter_spec : In x (inter s s') ↔ In x s ∧ In x s'.
Parameter diff_spec : In x (diff s s') ↔ In x s ∧ ¬In x s'.
Parameter fold_spec : ∀ (A : Type) (i : A) (f : elt → A → A),
  fold f s i = fold_left (flip f) (elements s) i.
Parameter filter_spec : compatb f →
  (In x (filter f s) ↔ In x s ∧ f x = true).
Parameter for_all_spec : compatb f →
  (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
Parameter exists_spec : compatb f →
  (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
Parameter partition_spec1 : compatb f →
  fst (partition f s) [=] filter f s.
Parameter partition_spec2 : compatb f →
  snd (partition f s) [=] filter (fun x ⇒ negb (f x)) s.
```

Parameter *elements_spec1* : **InA** *E.eq* x (*elements* s) $\leftrightarrow$ *In* x s.
Parameter *choose_spec1* : *choose* s = Some x $\rightarrow$ *In* x s.
Parameter *choose_spec2* : *choose* s = None $\rightarrow$ 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* $\rightarrow$ **list** elt.

Parameter *elements_dist_spec1* : $\forall$ x s, **InA** *E.eq* x (*elements_dist* s) $\leftrightarrow$ **InA** *E.eq* x (*elements* s).

Parameter *elements_dist_spec2w* : $\forall$ s, **NoDupA** *E.eq* (*elements_dist* s).

Cardinality can then be specified with respect to *elements_dist*. Parameter *cardinal_spec* : $\forall$ 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 : $\forall$ x s, **InA** *E.eq* x (elements_dist s) $\leftrightarrow$ **InA** *E.eq* x (*elements* s).

Lemma elements_dist_spec2w : $\forall$ s, **NoDupA** *E.eq* (elements_dist s).

End WSetsOn_TO_WSetsOnWithDupsExtra.

Chapter 2

Library MSetFoldWithAbort

2.1 Fold with abort for sets

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

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

2.1.1 Fold With Abort Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\forall$

$(A : \texttt{Type})$

result type

$(i\ i' : A)$

base values for foldWithAbort and fold

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

fold functions for foldWithAbort and fold

$(f\_gt : elt \to A \to \textbf{bool})$

abort function

$(s : t)$ sets to fold over

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

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

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

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

f_abort is OK, i.e. all other elements can be skipped without leaving the equivalence relation. $(\forall\ e1\ st,$

$In\ e1\ s \to f\_gt\ e1\ st = \textsf{true} \to$

$(\forall\ st'\ e2,\ P\ st\ st' \to$

$In\ e2\ s \to lt\ e1\ e2 \to$

$P\ st\ (f\ e2\ st'))) \to$

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

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

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

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

Definition foldWithAbortGtLtType

elt element type of set t type of set A return type :=

$(elt \to A \to \textbf{bool}) \to$ f_lt $(elt \to A \to A) \to$ f $(elt \to A \to \textbf{bool}) \to$ f_gt

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Module Types

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

```
Module Type HasFoldWithAbort (E : OrderedType) (Import C : WSetsOnWithDups E).

  Parameter foldWithAbortPrecompute : ∀ {A B : Type},
    foldWithAbortPrecomputeType elt t A B.

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

End HasFoldWithAbort.
```

2.1.2 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 $\rightarrow$ A $\rightarrow$ A)) f_gt :=
foldWithAbortPrecompute (`fun` _ $\Rightarrow$ tt) (`fun` e _ st $\Rightarrow$ f_lt e st)
(`fun` e _ st $\Rightarrow$ f e st) (`fun` e _ st $\Rightarrow$ f_gt e st).

`Lemma` foldWithAbortGtLtSpec :
foldWithAbortGtLtSpecPred *E.lt* *In* (@*fold*) (@foldWithAbortGtLt).

`Definition` foldWithAbortGt $\{A\}$ (f : (elt $\rightarrow$ A $\rightarrow$ A)) f_gt :=
foldWithAbortPrecompute (`fun` _ $\Rightarrow$ tt) (`fun` _ _ _ $\Rightarrow$ false)
(`fun` e _ st $\Rightarrow$ f e st) (`fun` e _ st $\Rightarrow$ f_gt e st).

`Lemma` foldWithAbortGtSpec :
foldWithAbortGtSpecPred *E.lt* *In* (@*fold*) (@foldWithAbortGt).

`Definition` foldWithAbort $\{A\}$ (f : (elt $\rightarrow$ A $\rightarrow$ A)) f_abort :=
foldWithAbortPrecompute (`fun` _ $\Rightarrow$ tt) (`fun` e _ st $\Rightarrow$ f_abort e st)
(`fun` e _ st $\Rightarrow$ f e st) (`fun` e _ st $\Rightarrow$ f_abort e st).

`Lemma` foldWithAbortSpec :
foldWithAbortSpecPred *In* (@*fold*) (@foldWithAbort).

Specialisations for equality

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

($\forall$ e st, *In* e s $\rightarrow$ (f e (f_pre e) st = f' e st)) $\rightarrow$

($\forall$ $e1$ st,
In $e1$ s $\rightarrow$ f_lt $e1$ (f_pre $e1$) st = true $\rightarrow$

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

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

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

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

(foldWithAbortGtLt *f_lt* *f* *f_gt* *s* *i*) = (*fold* *f'* *s* *i*).
Lemma foldWithAbortGtSpec_Equal : ∀ (*A* : Type) (*i* : *A*)
(*f* : elt → *A* → *A*) (*f'* : elt → *A* → *A*) (*f_gt* : elt → *A* → **bool**) (*s* : *t*),

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

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

(foldWithAbortGt *f* *f_gt* *s* *i*) = (*fold* *f'* *s* *i*).
Lemma foldWithAbortSpec_Equal : ∀ (*A* : Type) (*i* : *A*)
(*f* : elt → *A* → *A*) (*f'* : elt → *A* → *A*) (*f_abort* : elt → *A* → **bool**) (*s* : *t*),

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

$(\forall\ e1\ st,$
$\quad \mathsf{In}\ e1\ s \rightarrow f\_abort\ e1\ st = \mathsf{true} \rightarrow$
$\quad (\forall\ e2,\ \mathsf{In}\ e2\ s \rightarrow e1 \neq e2 \rightarrow$
$\qquad (f\ e2\ st = st))) \rightarrow$

$(\mathsf{foldWithAbort}\ f\ f\_abort\ s\ i) = (\mathit{fold}\ f'\ s\ i).$

FoldWithAbortSpecArgs

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

Given a *FoldWithAbortSpecArg* for a predicate P and a set s, many operations can be implemented efficiently. Below we will provide efficient versions of *filter*, *choose*, ∃, ∀ and more. `Record` **FoldWithAbortSpecArg** $\{B\}$:= {
fwasa_f_pre : (elt → B); The precompute function fwasa_f_lt : (elt → B → **bool**); f_lt without state argument fwasa_f_gt : (elt → B → **bool**); f_gt without state argument fwasa_P' : (elt → B → **bool**) the predicate P }.

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

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

If *fwasa_f_lt* holds, all elements smaller than the current one don't satisfy predicate P. $(\forall\ e1,$
$\quad \mathsf{In}\ e1\ s \rightarrow \mathsf{fwasa\_f\_lt}\ \mathit{fwasa}\ e1\ (\mathsf{fwasa\_f\_pre}\ \mathit{fwasa}\ e1) = \mathsf{true} \rightarrow$
$\quad (\forall\ e2,\ \mathsf{In}\ e2\ s \rightarrow \mathsf{E.lt}\ e2\ e1 \rightarrow (P\ e2 = \mathsf{false}))) \wedge$

If *fwasa_f_gt* holds, all elements greater than the current one don't satisfy predicate P. $(\forall\ e1,$
$\quad \mathsf{In}\ e1\ s \rightarrow \mathsf{fwasa\_f\_gt}\ \mathit{fwasa}\ e1\ (\mathsf{fwasa\_f\_pre}\ \mathit{fwasa}\ e1) = \mathsf{true} \rightarrow$
$\quad (\forall\ e2,\ \mathsf{In}\ e2\ s \rightarrow \mathsf{E.lt}\ e1\ e2 \rightarrow (P\ e2 = \mathsf{false}))).$

Filter with abort

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

@*foldWithAbortPrecompute* t B (fwasa_f_pre $fwasa$) (fun e p _ $\Rightarrow$ fwasa_f_lt $fwasa$ e p)
(fun e e_pre s $\Rightarrow$ if fwasa_P' $fwasa$ e e_pre then *add* e s else s)
(fun e p _ $\Rightarrow$ 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).

Choose with abort

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

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

Lemma choose_with_abort_spec $\{B\}$: $\forall$ $fwasa$ P s,
@foldWithAbortSpecArgsForPred B s P $fwasa$ $\rightarrow$
Proper (*E.eq* ==> **Logic.eq**) P $\rightarrow$
(match (choose_with_abort $fwasa$ s) with
| None $\Rightarrow$ ($\forall$ e, *In* e s $\rightarrow$ P e = false)
| Some e $\Rightarrow$ *In* e s $\wedge$ (P e = true)
end).

Exists and Forall with abort

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

Lemma exists_with_abort_spec $\{B\}$: $\forall$ $fwasa$ P s,
@foldWithAbortSpecArgsForPred B s P $fwasa$ $\rightarrow$
Proper (*E.eq* ==> **Logic.eq**) P $\rightarrow$
(exists_with_abort $fwasa$ s =
*exists_* P s).

Negation leads to forall. Definition forall_with_abort $\{B\}$ $fwasa$ s :=
negb (@exists_with_abort B $fwasa$ s).

```
  Lemma forall_with_abort_spec {B} : ∀ fwasa s P,
    @foldWithAbortSpecArgsForPred B s P fwasa →
    Proper (E.eq ==> Logic.eq) P →
    (forall_with_abort fwasa s =
     for_all (fun e ⇒ negb (P e)) s).
End HasFoldWithAbortOps.
```

2.1.3 Modules Types For Sets with Fold with Abort

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

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

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

2.1.4 Implementations

GenTree implementation

Finally, provide such a fold with abort operation for generic trees. `Module MakeGenTreeFoldA`
```
(Import E : OrderedType) (Import I:InfoTyp)
  (Import Raw:Ops E I)
  (M : MSetGenTree.Props E I Raw).

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

```
        let st0 := if f_lt x x_pre base then base else foldWithAbort_Raw f_pre f_lt f f_gt
l base in
        let st1 := f x x_pre st0 in
        let st2 := if f_gt x x_pre st1 then st1 else foldWithAbort_Raw f_pre f_lt f f_gt
r st1 in
        st2
    end.
  Lemma foldWithAbort_RawSpec : ∀ (A B : Type) (i i' : A) (f_pre : E.t → B)
      (f : E.t → B → A → A) (f' : E.t → A → A) (f_lt f_gt : E.t → B → A → bool) (s
: Raw.tree)
      (P : A → A → Prop),

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


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


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

      P i i' →
      P (foldWithAbort_Raw f_pre f_lt f f_gt s i) (fold f' s i').
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 $\{A\ B\colon$ `Type`$\}$ f_pre f_lt $(f\colon \mathsf{elt} \to B \to A \to A)$ f_gt t $(base\colon\ A)$: A :=
    foldWithAbort_Raw f_pre f_lt f f_gt (t.(this)) $base$.

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

`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 $\{A\ B\colon$ `Type`$\}$ f_pre f_lt $(f\colon \mathsf{elt} \to B \to A \to A)$ f_gt t $(base\colon\ A)$: A :=
    foldWithAbort_Raw f_pre f_lt f f_gt (t.(this)) $base$.

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

`Include` HasFoldWithAbortOps X.

`End` MakeRBTSetsWithFoldA.

2.1.5 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 (@foldWithAbortPrecompute).
  Include HasFoldWithAbortOps X.

End MakeListSetsWithFoldA.
```

Unsorted Lists without Dups Implementation

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

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

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

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

End MakeWeakListSetsWithFoldA.
```

Chapter 3

Library MSetIntervals

3.1 Weak sets implemented by interval lists

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

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

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

3.1.1 Auxiliary stuff

Simple auxiliary lemmata

```
Lemma Z_le_add_r : ∀ (z : Z) (n : N),
  z ≤ z + Z.of_N n.
```

```
Lemma add_add_sub_eq : ∀ (x y : Z), (x + (y - x) = y).
```

```
Lemma NoDupA_map {A B} : ∀ (eqA : A → A → Prop) (eqB : B → B → Prop) (f : A →
B) l,
  NoDupA eqA l →
  (∀ x1 x2, List.In x1 l → List.In x2 l →
                    eqB (f x1) (f x2) → eqA x1 x2) →
  NoDupA eqB (map f l).
```

rev_map

rev_map is used for efficiency. Fixpoint rev_map_aux $\{A\ B\}$ $(f : A \to B)$ $(acc$: list $B)$ $(l$: list $A)$:=
  match l with
    | nil ⇒ acc
    | x :: xs ⇒ rev_map_aux f ((f x) :: acc) xs
  end.

Definition rev_map $\{A\ B\}$ $(f : A \to B)$ $(l$: list $A)$: list B := rev_map_aux f nil l.

Lemmata about rev_map Lemma rev_map_aux_alt_def $\{A\ B\}$: ∀ $(f : A \to B)$ l acc,
  rev_map_aux f acc l = List.rev_append (List.map f l) acc.
Lemma rev_map_alt_def $\{A\ B\}$: ∀ $(f : A \to B)$ l,
  rev_map f l = List.rev (List.map f l).

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 *ElementEncode*. It provides encode and decode function.

Module Type ElementEncode.
  Declare Module E : DecidableType.

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

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

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

End ElementEncode.

3.1.3 Set Operations

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

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

In contrast

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

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

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

```
Module Ops (Enc : ElementEncode) <: WOps 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).

  Fixpoint elementsZ_aux'' (acc : list Z) (x : Z) (c : N) (H : Acc N.lt c) { struct H } :
list Z :=
    match c as c0 return c = c0 → list Z with
    | N0 ⇒ fun _ ⇒ acc
    | c ⇒ fun Heq ⇒ elementsZ_aux'' (x::acc) (Z.succ x) (N.pred c) (acc_pred _ _ Heq H)
    end (refl_equal _).

  Extraction Inline elementsZ_aux''.

  Definition elementsZ_aux' acc x c := elementsZ_aux'' acc x c (lt_wf_0 _).

  Fixpoint elementsZ_aux acc (s : t) : list Z :=
    match s with
    | nil ⇒ acc
    | (x, c) :: s' ⇒
        elementsZ_aux (elementsZ_aux' acc x c) s'
    end.

  Definition elementsZ (s : t) : list Z :=
    elementsZ_aux 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.
```

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

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

```
  match s with
  | nil ⇒ List.rev' ((x, (1%N))::acc)
  | (y, c) :: l ⇒
      match (Z.compare (Z.succ x) y) with
      | Lt ⇒ List.rev_append ((x, (1%N))::acc) s
      | Eq ⇒ List.rev_append ((x, N.succ c)::acc) l
      | Gt ⇒ match (Z.compare x (y+Z.of_N c)) with
                 | Lt ⇒ List.rev_append acc s
                 | Gt ⇒ addZ_aux ((y,c) :: acc) x l
                 | Eq ⇒ match l with
                            | nil ⇒ List.rev' ((y, N.succ c)::acc)
                            | (z, c') :: l' ⇒ if (Z.eqb z (Z.succ x)) then
                                 List.rev_append ((y,N.succ (c+c')) :: acc) l'
                              else
                                 List.rev_append ((y,N.succ c) :: acc) l
                        end
                 end
      end
  end.
Definition addZ x s := addZ_aux nil x s.
Definition add x s := addZ (Enc.encode x) s.
```

add_list simple extension to add many elements, which then allows to define from_elements.

```
Definition add_list (l : list elt) (s : t) : t :=
    List.fold_left (fun s x ⇒ add x s) l s.
Definition from_elements (l : list elt) : t := add_list l empty.
```

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

```
Lemma singleton_alt_def : ∀ x, singleton x = add x empty.
```

removing needs to be done with code extraction in mind again. `Definition removeZ_aux_insert_guarded (x : Z) (c : N) s :=`

```
    if (N.eqb c 0) then s else (x, c) :: s.
Fixpoint removeZ_aux (acc : list (Z × N)) (x : Z) (s : t) : t :=
  match s with
  | nil ⇒ List.rev' acc
  | (y, c) :: l ⇒
      if (Z.ltb x y) then List.rev_append acc s else
      if (Z.ltb x (y+Z.of_N c)) then (
         List.rev_append (removeZ_aux_insert_guarded (Z.succ x)
            (Z.to_N ((y+Z.of_N c)- (Z.succ x))))
           (removeZ_aux_insert_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.
```

all other operations are defined trivially (if not always efficiently) in terms of already defined ones. In the future it might be worth implementing some of them more efficiently.

```
Definition union (s1 s2 : t) :=
  add_list (elements s1) s2.
Definition filter (f : elt -> bool) (s : t) : t :=
  from_elements (List.filter f (elements s)).
Definition inter (s1 s2 : t) : t :=
  filter (fun x => mem x s2) s1.
Definition diff (s1 s2 : t) : t :=
  remove_list (elements s2) s1.
Definition subset s s' :=
  List.forallb (fun x => mem x s') (elements s).
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.
Definition fold {B : Type} (f : elt -> B -> B) (s : t) (i : B) : B :=
  List.fold_left (flip f) (elements s) i.
Definition for_all (f : elt -> bool) (s : t) : bool :=
  List.forallb f (elements s).
Definition exists_ (f : elt -> bool) (s : t) : bool :=
  List.existsb f (elements s).
Definition partition (f : elt -> bool) (s : t) : t * t :=
  (filter f s, filter (fun x => negb (f x)) 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 chooseZ (s : t) : option Z :=
  match List.rev' (elementsZ s) with
  | nil => None
```

```
    | x :: _ ⇒ Some x
    end.
  Definition choose (s : t) : option elt :=
    match elements s with
      | nil ⇒ None
      | e :: _ ⇒ Some e
    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 inf (x:Z) (l: t) :=
   match l with
   | nil ⇒ true
   | (y,_)::_ ⇒ Z.ltb x y
   end.
  Fixpoint isok (l : t) :=
   match l with
   | nil ⇒ true
   | (x, c) ::l ⇒ inf (x+(Z.of_N c)) l && negb (N.eqb c 0) && isok l
   end.
  Definition is_encoded_elems_list (l : list Z) : Prop :=
    (∀ x, List.In x l → ∃ e, Enc.encode e = x).
  Definition IsOk s := (isok s = true ∧ is_encoded_elems_list (elementsZ s)).
```

Defining notations

```
  Section ForNotations.
    Class Ok (s:t) : Prop := ok : IsOk s.
    Hint Resolve @ok.
    Hint Unfold Ok.
    Instance IsOk_Ok s `(Hs : IsOk s) : Ok s := { ok := Hs }.
    Definition In x s := (SetoidList.InA Enc.E.eq x (elements s)).
```

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

Lemma elements_nil : (elements (nil : t) = nil).

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

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

Lemma elementsZ_single_base : ∀ x,
  elementsZ_single x (0%N) = nil.

Lemma elementsZ_single_succ : ∀ x c,
  elementsZ_single x (N.succ c) =
  elementsZ_single x c ++ (x+Z.of_N c)::nil.

Lemma elementsZ_single_add : ∀ x c2 c1,
  elementsZ_single x (c1 + c2)%N =
  elementsZ_single x c1 ++ elementsZ_single (x+Z.of_N c1) c2.
Lemma elementsZ_single_succ_front : ∀ x c,
  elementsZ_single x (N.succ c) =
  x :: elementsZ_single (Z.succ x) c.

Lemma In_elementsZ_single : ∀ c y x,
  List.In y (elementsZ_single x c) ↔
  (x ≤ y) ∧ (y < (x+Z.of_N c)).
Lemma In_elementsZ_single1 : ∀ y x,
  List.In y (elementsZ_single x (1%N)) ↔
  (x = y).
Lemma length_elementsZ_single : ∀ cx x,
  length (elementsZ_single x cx) = N.to_nat cx.
Lemma elementsZ_aux''_irrel : ∀ c acc x H1 H2,
    elementsZ_aux'' acc x c H1 = elementsZ_aux'' acc x c H2.
```

Lemma elementsZ_aux'_pos : $\forall\ s\ x\ p$,
elementsZ_aux' $s\ x$ (N.pos p) = elementsZ_aux' $(x :: s)$ (Z.succ x) (Pos.pred_N p).

Lemma elementsZ_aux'_zero : $\forall\ s\ x$,
elementsZ_aux' $s\ x$ $(0\%N)$ = s.

Lemma elementsZ_aux'_succ : $\forall\ s\ x\ c$,
elementsZ_aux' $s\ x$ (N.succ c) = elementsZ_aux' $(x :: s)$ (Z.succ x) c.

Lemma elementsZ_single_intro : $\forall\ c\ s\ x$,
elementsZ_aux' $s\ x\ c$ =
(List.rev (elementsZ_single $x\ c$)) ++ s.

Lemma elementsZ_aux_alt_def : $\forall\ s\ acc$,
elementsZ_aux $acc\ s$ = elementsZ s ++ acc.

Lemma elementsZ_cons : $\forall\ x\ c\ s$, elementsZ $(((x, c) :: s) : \mathsf{t})$ =
((elementsZ s) ++ (List.rev (elementsZ_single $x\ c$))).

Lemma elements_cons : $\forall\ x\ c\ s$, elements $(((x, c) :: s) : \mathsf{t})$ =
((elements_single $x\ c$) ++ elements s).

Lemma In_elementsZ_single_hd : $\forall\ (c : \mathsf{N})\ x$, $(c \neq 0)\%N \rightarrow$ List.In x (elementsZ_single x c).

Alternative definition of addZ

addZ is defined with efficient execution in mind. We derive first an alternative definition that demonstrates the intention better and is better suited for proofs. Lemma addZ_ind :
$\forall\ (P : \mathsf{Z} \rightarrow \mathsf{list}\ (\mathsf{Z} \times \mathsf{N}) \rightarrow \mathtt{Prop})$,

$(\forall\ (x : \mathsf{Z}),\ P\ x\ \mathsf{nil}) \rightarrow$

$(\forall\ (x : \mathsf{Z})\ (l : \mathsf{list}\ (\mathsf{Z} \times \mathsf{N}))\ (c : \mathsf{N})$,
$P\ x\ ((x + 1, c) :: l)) \rightarrow$

$(\forall\ (x : \mathsf{Z})\ (l : \mathsf{list}\ (\mathsf{Z} \times \mathsf{N}))\ (y : \mathsf{Z})\ (c : \mathsf{N})$,
$(x + 1\ ?{=}\ y) = \mathsf{Lt} \rightarrow$
$P\ x\ ((y, c) :: l)) \rightarrow$

$(\forall\ (y : \mathsf{Z})\ (c : \mathsf{N})$,
$((y + \mathsf{Z.of\_N}\ c) + 1\ ?{=}\ y) = \mathsf{Gt} \rightarrow$
$P\ (y + \mathsf{Z.of\_N}\ c)\ ((y, c) :: \mathsf{nil})) \rightarrow$

```
(∀ (l : list (Z × N)) (y : Z) (c c' : N),
  ((y + Z.of_N c) + 1 ?= y) = Gt →
  (P (y+Z.of_N c) l) →
  P (y+Z.of_N c) ((y, c) :: (((y+Z.of_N c) + 1, c') :: l))) →


(∀ (l : list (Z × N)) (y : Z) (c : N) (z : Z) (c' : N),
  ((y + Z.of_N c) + 1 ?= y) = Gt →
  (z =? (y+Z.of_N c) + 1) = false →
  (P (y+Z.of_N c) ((y, c) :: (z, c') :: l))) →


(∀ (x : Z) (l : list (Z × N)) (y : Z) (c : N),
  (x + 1 ?= y) = Gt →
  (x ?= y + Z.of_N c) = Lt →
  P x ((y, c) :: l)) →


(∀ (x : Z)(l : list (Z × N)) (y : Z) (c : N),
  (x + 1 ?= y) = Gt →
  (x ?= y + (Z.of_N c)) = Gt →
  (P x l) →
  P x ((y, c) :: l)) →


∀ (x : Z) (s : list (Z × N)),
P x s.
Lemma addZ_aux_alt_def : ∀ x s acc,
  addZ_aux acc x s = (List.rev acc) ++ addZ x s.
Lemma addZ_alt_def : ∀ x s,
  addZ x s =
  match s with
  | nil ⇒ (x, 1%N)::nil
  | (y, c) :: l ⇒
      match (Z.compare (x+1) y) with
      | Lt ⇒ (x, 1%N)::s
      | Eq ⇒ (x, (c+1)%N)::l
      | Gt ⇒ match (Z.compare x (y+Z.of_N c)) with
               | Lt ⇒ s
               | Gt ⇒ (y,c) :: addZ x l
               | Eq ⇒ match l with
```

```
                  | nil => (y, (c+1)%N)::nil
                  | (z, c') :: l' => if (Z.eqb z (x + 1)) then
                      (y, (c + c' + 1)%N) :: l'
                    else
                      (y,(c+1)%N) :: (z, c') :: l'
                end
          end
    end
  end.
```

Alternative definition of removeZ

removeZ is defined with efficient execution in mind. We derive first an alternative definition that demonstrates the intention better and is better suited for proofs.

```
Lemma removeZ_aux_alt_def : ∀ s x acc,
  removeZ_aux acc x s = (List.rev acc) ++ removeZ x s.
Lemma removeZ_alt_def : ∀ x s,
  removeZ x s =
  match s with
  | nil => nil
  | (y, c) :: l =>
      if (Z.ltb x y) then s else
      if (Z.ltb x (y+Z.of_N c)) then (
        (removeZ_aux_insert_guarded y (Z.to_N (x-y))
          (removeZ_aux_insert_guarded (Z.succ x) (Z.to_N ((y+Z.of_N c)- (Z.succ x)))
l))
      ) else (y, c) :: removeZ x l
  end.
```

Auxiliary Lemmata about Invariant

```
Lemma inf_impl : ∀ x y s,
  (y ≤ x) → inf x s = true → inf y s = true.
Lemma Ok_nil : Ok nil ↔ True.
Lemma is_encoded_elems_list_app : ∀ l1 l2,
  is_encoded_elems_list (l1 ++ l2) ↔
  (is_encoded_elems_list l1 ∧ is_encoded_elems_list l2).
Lemma is_encoded_elems_list_rev : ∀ l,
  is_encoded_elems_list (List.rev l) ↔
  is_encoded_elems_list l.
Lemma isok_cons : ∀ y c s', isok ((y, c) :: s') = true ↔
  (inf (y+Z.of_N c) s' = true ∧ ((c ≠ 0)%N) ∧ isok s' = true).
```

```
Lemma Ok_cons : ∀ y c s', Ok ((y, c) :: s') ↔
   (inf (y+Z.of_N c) s' = true ∧ ((c ≠ 0)%N) ∧
    is_encoded_elems_list (elementsZ_single y c) ∧ Ok s').
Lemma Nin_elements_greater : ∀ s y,
    inf y s = true →
    isok s = true →
    ∀ x, x ≤ y →
    ~(InZ x s).
Lemma isok_inf_nin :
    ∀ x s,
      isok s = true →
      inf x s = true →
      ¬ (InZ x s).
```

Properties of In and InZ

```
Lemma In_alt_def : ∀ x s, Ok s →
   (In x s ↔ List.In x (elements s)).
Lemma In_InZ : ∀ x s, Ok s →
   (In x s ↔ InZ (Enc.encode x) s).
```

Membership specification

```
Lemma memZ_spec :
 ∀ (s : t) (x : Z) (Hs : Ok s), memZ x s = true ↔ InZ x s.
Lemma mem_spec :
 ∀ (s : t) (x : elt) (Hs : Ok s), mem x s = true ↔ In x s.
```

add specification

```
Lemma addZ_spec :
 ∀ (s : t) (x y : Z) (Hs : Ok s),
   InZ y (addZ x s) ↔ Z.eq y x ∨ InZ y s.
Lemma addZ_isok : ∀ s x, isok s = true → isok (addZ x s) = true.
Global Instance add_ok s x : ∀ `(Ok s), Ok (add x s).
Lemma add_spec :
 ∀ (s : t) (x y : elt) (Hs : Ok s),
    In y (add x s) ↔ Enc.E.eq y x ∨ In y s.
```

remove specification

```
Lemma isok_removeZ_aux_insert_guarded : ∀ x c s,
   isok s = true → inf (x + Z.of_N c) s = true →
```

isok (removeZ_aux_insert_guarded x c s) = true.

Lemma inf_removeZ_aux_insert_guarded : ∀ x c y s,
inf y (removeZ_aux_insert_guarded x c s) = true ↔
(if (c =? 0)%N then (inf y s = true) else (y < x)).

Lemma removeZ_counter_pos_aux : ∀ y c x,
x < y + Z.of_N c →
$0 \leq y$ + Z.of_N c - Z.succ x.

Lemma removeZ_isok : ∀ s x, isok s = true → isok (removeZ x s) = true.

Lemma elementsZ_removeZ_aux_insert_guarded : ∀ x c s,
elementsZ (removeZ_aux_insert_guarded x c s) = elementsZ ((x, c) :: s).

Lemma removeZ_spec :
∀ (s : t) (x y : **Z**) (Hs : isok s = true),
InZ y (removeZ x s) ↔ InZ y s ∧ ¬Z.eq y x.

Global Instance remove_ok s x : ∀ ‘(**Ok** s), **Ok** (remove x s).

Lemma remove_spec :
∀ (s : t) (x y : elt) (Hs : **Ok** s),
In y (remove x s) ↔ In y s ∧ ¬*Enc.E.eq* y x.

empty specification

Global Instance empty_ok : **Ok** empty.

Lemma empty_spec' : ∀ x, (In x empty ↔ **False**).

Lemma empty_spec : Empty empty.

is_empty specification

Lemma is_empty_spec : ∀ (s : t) (Hs : **Ok** s), is_empty s = true ↔ Empty s.

singleton specification

Global Instance singleton_ok x : **Ok** (singleton x).

Lemma singleton_spec : ∀ x y : elt, In y (singleton x) ↔ *Enc.E.eq* y x.

add_list specification

Lemma add_list_ok : ∀ l s, **Ok** s → **Ok** (add_list l s).

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

remove_list specification

Lemma remove_list_ok : ∀ l s, **Ok** s → **Ok** (remove_list l s).
Lemma remove_list_spec : ∀ x l s, **Ok** s →
  (In x (remove_list l s) ↔ ~(**InA** *Enc.E.eq* x l) ∧ In x s).

union specification

Global Instance union_ok s s' : ∀ '(**Ok** s, **Ok** s'), **Ok** (union s s').

Lemma union_spec :
 ∀ (s s' : t) (x : elt) (Hs : **Ok** s) (Hs' : **Ok** s'),
 In x (union s s') ↔ In x s ∨ In x s'.

filter specification

Global Instance filter_ok s f : ∀ '(**Ok** s), **Ok** (filter f s).

Lemma filter_spec :
 ∀ (s : t) (x : elt) (f : elt → **bool**),
 Proper (*Enc.E.eq*==>**eq**) f →
 (In x (filter f s) ↔ In x s ∧ f x = true).

inter specification

Global Instance inter_ok s s' : ∀ '(**Ok** s, **Ok** s'), **Ok** (inter s s').

Lemma inter_spec :
 ∀ (s s' : t) (x : elt) (Hs : **Ok** s) (Hs' : **Ok** s'),
 In x (inter s s') ↔ In x s ∧ In x s'.

diff specification

Global Instance diff_ok s s' : ∀ '(**Ok** s, **Ok** s'), **Ok** (diff s s').

Lemma diff_spec :
 ∀ (s s' : t) (x : elt) (Hs : **Ok** s) (Hs' : **Ok** s'),
 In x (diff s s') ↔ In x s ∧ ¬In x s'.

subset specification

Lemma subset_spec :
 ∀ (s s' : t) (Hs : **Ok** s) (Hs' : **Ok** s'),
 subset s s' = true ↔ Subset s s'.

elements and elementsZ specification

Lemma elements_spec1 : ∀ (s : t) (x : elt) (Hs : **Ok** s), List.In x (elements s) ↔ In x s.

Lemma NoDupA_elementsZ_single: ∀ c x,
  NoDupA Z.eq (elementsZ_single x c).
Lemma elementsZ_spec2w : ∀ (s : t) (Hs : **Ok** s), **NoDupA** Z.eq (elementsZ s).
Lemma elements_spec2w : ∀ (s : t) (Hs : **Ok** s), **NoDupA** *Enc.E.eq* (elements s).

equal specification

```
Lemma equal_alt_def : ∀ s1 s2,
  equal s1 s2 = true ↔ (s1 = s2).
Lemma elementsZ_cons_le_start : ∀ x cx xs y cy ys,
    isok ((x , cx) :: xs) = true →
    isok ((y , cy) :: ys) = true →
    (∀ z, List.In z (elementsZ ((y , cy) :: ys)) →
                List.In z (elementsZ ((x , cx) :: xs))) →
    (x ≤ y).
Lemma elementsZ_cons_le_end : ∀ x cx xs y cy ys,
    isok ((x , cx) :: xs) = true →
    isok ((y , cy) :: ys) = true →
    (x ≤ y + Z.of_N cy) →
    (∀ z, List.In z (elementsZ ((x , cx) :: xs)) →
                List.In z (elementsZ ((y , cy) :: ys))) →
    (x + Z.of_N cx ≤ y + Z.of_N cy).
Lemma elementsZ_cons_equiv_hd : ∀ x cx xs y cy ys,
    isok ((x , cx) :: xs) = true →
    isok ((y , cy) :: ys) = true →
    (∀ z, List.In z (elementsZ ((x , cx) :: xs)) ↔
                List.In z (elementsZ ((y , cy) :: ys))) →
    (x = y) ∧ (cx = cy).
Lemma elementsZ_single_equiv : ∀ x cx y cy,
    (cx ≠ 0)%N →
    (cy ≠ 0)%N →
    (∀ z, List.In z (elementsZ_single x cx) ↔
                List.In z (elementsZ_single y cy)) →
    (x = y) ∧ (cx = cy).
Lemma equal_elementsZ :
  ∀ (s s' : t) {Hs : Ok s} {Hs' : Ok s'},
  (∀ x, (InZ x s ↔ InZ x s')) → (s = s').
Lemma equal_spec :
  ∀ (s s' : t) {Hs : Ok s} {Hs' : Ok s'},
  equal s s' = true ↔ Equal s s'.
```

choose specification

```
Lemma choose_alt_def : ∀ s,
  choose s = match chooseZ s with
    | None ⇒ None
    | Some e ⇒ Some (Enc.decode e)
  end.

Definition choose_spec1 :
  ∀ (s : t) (x : elt), choose s = Some x → In x s.

Definition choose_spec2 :
  ∀ s : t, choose s = None → Empty s.

Lemma chooseZ_min :
  ∀ (s : t) (x y : Z) (Hs : Ok s),
  chooseZ s = Some x → InZ y s → ¬ Z.lt y x.
Lemma chooseZ_InZ :
  ∀ (s : t) (x : Z),
  chooseZ s = Some x → InZ x s.

Lemma chooseZ_spec3: ∀ s s' x x', Ok s → Ok s' →
 chooseZ s = Some x → chooseZ s' = Some x' → Equal s s' → x = x'.
```

fold specification

```
Lemma fold_spec :
 ∀ (s : t) (A : Type) (i : A) (f : elt → A → A),
 fold f s i = fold_left (flip f) (elements s) i.
```

cardinal specification

```
Lemma cardinalN_spec : ∀ (s : t) (c : N),
  cardinalN c s = (c + N.of_nat (length (elements s)))%N.
Lemma cardinal_spec :
 ∀ (s : t),
 cardinal s = length (elements s).
```

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).
```

exists specification

```
Lemma exists_spec :
  ∀ (s : t) (f : elt → bool) (Hs : Ok s),
  Proper (Enc.E.eq==>eq) f →
  (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
```

partition specification

```
Global Instance partition_ok1 s f : ∀ `(Ok s), Ok (fst (partition f s)).

Global Instance partition_ok2 s f : ∀ `(Ok s), Ok (snd (partition f s)).

Lemma partition_spec1 :
  ∀ (s : t) (f : elt → bool),
  Proper (Enc.E.eq==>eq) f → Equal (fst (partition f s)) (filter f s).

Lemma partition_spec2 :
  ∀ (s : t) (f : elt → bool),
  Proper (Enc.E.eq==>eq) f →
  Equal (snd (partition f s)) (filter (fun x ⇒ negb (f x)) s).

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) <: WSetsOn Enc.E.
  Module E := Enc.E.
  Module Raw := Raw Enc.

 Local Local

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

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

 Definition mem (x : elt)(s : t) := Raw.mem x s.(this).
 Definition add (x : elt)(s : t) : t := Mkt (Raw.add x s.(this)).
```

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

Instance In_compat : Proper (E.eq==>eq==>iff) In.

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

Instance eq_equiv : Equivalence eq.

Definition eq_dec : ∀ (s s':t), { eq s s' }+{ ¬eq s s' }.

Section Spec.
 Variable s s' : t.
 Variable x y : elt.
 Variable f : elt → bool.
 Notation compatb := (Proper (E.eq==>Logic.eq)) (only parsing).

 Lemma mem_spec : mem x s = true ↔ In x s.
 Lemma equal_spec : equal s s' = true ↔ Equal s s'.
 Lemma subset_spec : subset s s' = true ↔ Subset s s'.
 Lemma empty_spec : Empty empty.
 Lemma is_empty_spec : is_empty s = true ↔ Empty s.
 Lemma add_spec : In y (add x s) ↔ E.eq y x ∨ In y s.
 Lemma remove_spec : In y (remove x s) ↔ In y s ∧ ¬E.eq y x.
 Lemma singleton_spec : In y (singleton x) ↔ E.eq y x.
 Lemma union_spec : In x (union s s') ↔ In x s ∨ In x s'.
 Lemma inter_spec : In x (inter s s') ↔ In x s ∧ In x s'.
 Lemma diff_spec : In x (diff s s') ↔ In x s ∧ ¬In x s'.
 Lemma fold_spec : ∀ (A : Type) (i : A) (f : elt → A → A),
     fold f s i = fold_left (fun a e ⇒ f e a) (elements s) i.
 Lemma cardinal_spec : cardinal s = length (elements s).
```

```
  Lemma filter_spec : compatb f →
    (In x (filter f s) ↔ In x s ∧ f x = true).
  Lemma for_all_spec : compatb f →
    (for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).
  Lemma exists_spec : compatb f →
    (exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).
  Lemma partition_spec1 : compatb f → Equal (fst (partition f s)) (filter f s).
  Lemma partition_spec2 : compatb f →
      Equal (snd (partition f s)) (filter (fun x ⇒ negb (f x)) s).
  Lemma elements_spec1 : InA E.eq x (elements s) ↔ In x s.
  Lemma elements_spec2w : NoDupA E.eq (elements s).
  Lemma choose_spec1 : choose s = Some x → In x s.
  Lemma choose_spec2 : choose s = None → Empty s.
 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: ∀ (e : E.t),
    decode (encode e) = e.
  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
End ElementEncodeZ.
Module MSetIntervalsZ <: WSetsOn Z := MSetIntervals ElementEncodeZ.
```

N

```
Module ElementEncodeN <: ElementEncode.
  Module E := N.
  Definition encode (n : N) := Z.of_N n.
  Definition decode (z : Z) := Z.to_N z.
  Lemma decode_encode_ok: ∀ (e : E.t),
```

```
    decode (encode e) = e.
  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
End ElementEncodeN.
```

```
Module MSetIntervalsN <: WSetsOn N := MSetIntervals ElementEncodeN.
```

nat

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

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

  Lemma decode_encode_ok: ∀ (e : E.t),
    decode (encode e) = e.

  Lemma encode_eq : ∀ (e1 e2 : E.t),
    (Z.eq (encode e1) (encode e2)) ↔ E.eq e1 e2.
End ElementEncodeNat.
```

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

Chapter 4

Library MSetListWithDups

4.1 Weak sets implemented as lists with duplicates

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

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

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

4.1.1 Removing duplicates from sorted lists

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

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

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

```
  Module Import XTotalLeBool <: TotalLeBool.
    Definition t := X.t.
    Definition leb x y :=
      match X.compare x y with
```

```
      | Lt ⇒ true
      | Eq ⇒ true
      | Gt ⇒ false
     end.
   Infix "<=?" := leb (at level 35).
   Theorem leb_total : ∀ (a1 a2 : t), (a1 <=? a2 = true) ∨ (a2 <=? a1 = true).
   Definition le x y := (leb x y = true).
 End XTotalLeBool.
 Lemma eqb_eq_alt : ∀ x y, eqb x y = true ↔ eq x y.
```

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).
 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.
  Lemma remove_dups_from_sortedA_hd :
      ∀ x xs,
      ∃ (x':t) xs',
        remove_dups_from_sortedA (x :: xs) =
        (x' :: xs') ∧ (eqb x x' = true).
```

Finally we get our main result for removing duplicates from sorted lists `Lemma remove_dups_from_sorte`

```
:
    ∀ (l : list t),
      Sorted le l →
      let l' := remove_dups_from_sortedA l in (

      Sorted lt l' ∧
      NoDupA eq l' ∧
      (∀ x, InA eq x l ↔ InA eq x l')).
```

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')).
End RemoveDupsFromSorted.
```

4.1.2 Operations Module

With removing duplicates defined, we can implement the operations for our set implementation easily.

```
Module Ops (X:OrderedType) <: WOps X.

  Module RDFS := RemoveDupsFromSorted X.
  Module Import MX := OrderedTypeFacts X.

  Definition elt := X.t.
  Definition t := list elt.

  Definition empty : t := nil.

  Definition is_empty (l : t) := match l with nil ⇒ true | _ ⇒ false end.
  Fixpoint mem (x : elt) (s : t) : bool :=
    match s with
```

```
  | nil ⇒ false
  | y :: l ⇒
         match X.compare x y with
             Eq ⇒ true
           | _ ⇒ mem x l
         end
  end.

Definition add x (s : t) := x :: s.
Definition singleton (x : elt) := x :: nil.

Fixpoint rev_filter_aux acc (f : elt → bool) s :=
  match s with
     nil ⇒ acc
   | (x :: xs) ⇒ rev_filter_aux (if (f x) then (x :: acc) else acc) f xs
  end.
Definition rev_filter := rev_filter_aux nil.

Definition filter (f : elt → bool) (s : t) : t := rev_filter f s.

Definition remove x s :=
  rev_filter (fun y ⇒ match X.compare x y with Eq ⇒ false | _ ⇒ true end) s.

Definition union (s1 s2 : t) : t :=
  List.rev_append s2 s1.

Definition inter (s1 s2 : t) : t :=
  rev_filter (fun y ⇒ mem y s2) s1.

Definition elements (x : t) : list elt := x.

Definition elements_dist (x : t) : list elt :=
  RDFS.remove_dups_by_sortingA x.

Definition fold {B : Type} (f : elt → B → B) (s : t) (i : B) : B :=
  fold_left (flip f) (elements s) i.

Definition diff (s s' : t) : t := fold remove s' s.

Definition subset (s s' : t) : bool :=
  List.forallb (fun x ⇒ mem x s') s.

Definition equal (s s' : t) : bool := andb (subset s s') (subset s' s).

Fixpoint for_all (f : elt → bool) (s : t) : bool :=
  match s with
  | nil ⇒ true
  | x :: l ⇒ if f x then for_all f l else false
  end.

Fixpoint exists_ (f : elt → bool) (s : t) : bool :=
  match s with
```

```
  | nil ⇒ false
  | x :: l ⇒ if f x then true else exists_ f l
  end.

Fixpoint partition_aux (a1 a2 : t) (f : elt → bool) (s : t) : t × t :=
  match s with
  | nil ⇒ (a1 , a2)
  | x :: l ⇒
      if f x then partition_aux (x :: a1) a2 f l else
                  partition_aux a1 (x :: a2) f l
  end.

Definition partition := partition_aux nil nil.

Definition cardinal (s : t) : nat := length (elements_dist s).

Definition choose (s : t) : option elt :=
   match s with
    | nil ⇒ None
    | x::_ ⇒ Some x
   end.

End Ops.
```

4.1.3 Main Module

Using these operations, we can define the main functor. For this, we need to prove that the provided operations do indeed satisfy the weak set interface. This is mostly straightforward and unsurprising. The only interesting part is that removing duplicates from a sorted list behaves as expected. This has however already been proved in module *RemoveDupsFromSorted*.

```
Module Make (E:OrderedType) <: WSetsOnWithDupsExtra E.
  Include Ops E.
  Import MX.
```

4.1.4 Proofs of set operation specifications.

Logical predicates

```
Definition In x (s : t) := SetoidList.InA E.eq x s.

Instance In_compat : Proper (E.eq==>eq==>iff) In.

Definition Equal s s' := ∀ a : elt, In a s ↔ In a s'.
Definition Subset s s' := ∀ a : elt, In a s → In a s'.
Definition Empty s := ∀ a : elt, ¬ In a s.
Definition For_all (P : elt → Prop) s := ∀ x, In x s → P x.
Definition Exists (P : elt → Prop) s := ∃ x , In x s ∧ P x.

Notation "s [=] t" := (Equal s t) (at level 70, no associativity).
Notation "s [<=] t" := (Subset s t) (at level 70, no associativity).
```

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

Lemma eq_equiv : **Equivalence** eq.

Specifications of set operators Notation compatb := (**Proper** (*E.eq*==>**Logic.eq**)) (*only parsing*).

Lemma mem_spec : ∀ s x, mem x s = true ↔ In x s.

Lemma subset_spec : ∀ s s', subset s s' = true ↔ s[<=]s'.

Lemma equal_spec : ∀ s s', equal s s' = true ↔ s[=]s'.

Lemma eq_dec : ∀ x y : t, {eq x y} + {¬ eq x y}.

Lemma empty_spec : Empty empty.

Lemma is_empty_spec : ∀ s, is_empty s = true ↔ Empty s.

Lemma add_spec : ∀ s x y, In y (add x s) ↔ *E.eq* y x ∨ In y s.

Lemma singleton_spec : ∀ x y, In y (singleton x) ↔ *E.eq* y x.

Hint Resolve (@Equivalence_Reflexive _ _ *E.eq_equiv*).

Hint Immediate (@Equivalence_Symmetric _ _ *E.eq_equiv*).

Hint Resolve (@Equivalence_Transitive _ _ *E.eq_equiv*).

Lemma rev_filter_aux_spec : ∀ s acc x f, compatb f →
(In x (rev_filter_aux acc f s) ↔ (In x s ∧ f x = true) ∨ (In x acc)).

Lemma filter_spec : ∀ s x f, compatb f →
(In x (filter f s) ↔ In x s ∧ f x = true).

Lemma remove_spec : ∀ s x y, In y (remove x s) ↔ In y s ∧ ¬*E.eq* y x.

Lemma union_spec : ∀ s s' x, In x (union s s') ↔ In x s ∨ In x s'.

Lemma inter_spec : ∀ s s' x, In x (inter s s') ↔ In x s ∧ In x s'.

Lemma fold_spec : ∀ s (A : Type) (i : A) (f : elt → A → A),
fold f s i = fold_left (flip f) (elements s) i.

Lemma elements_spec1 : ∀ s x, **InA** *E.eq* x (elements s) ↔ In x s.

Lemma diff_spec : ∀ s s' x, In x (diff s s') ↔ In x s ∧ ¬In x s'.

Lemma cardinal_spec : ∀ s, cardinal s = length (elements_dist s).

Lemma for_all_spec : ∀ s f, compatb f →
(for_all f s = true ↔ For_all (fun x ⇒ f x = true) s).

Lemma exists_spec : ∀ s f, compatb f →
(exists_ f s = true ↔ Exists (fun x ⇒ f x = true) s).

Lemma partition_aux_spec : ∀ $a1$ $a2$ s f,
(partition_aux $a1$ $a2$ f s = (rev_filter_aux $a1$ f s, rev_filter_aux $a2$ (fun x ⇒ negb (f x)) s)).

Lemma partition_spec1 : ∀ s f, compatb f →
fst (partition f s) [=] filter f s.

Lemma partition_spec2 : ∀ s f, compatb f →
snd (partition f s) [=] filter (fun x ⇒ negb (f x)) s.

Lemma choose_spec1 : ∀ s x, choose s = Some x → In x s.

```
Lemma choose_spec2 : ∀ s, choose s = None → Empty s.

Lemma elements_dist_spec_full :
  ∀ s,
    Sorted E.lt (elements_dist s) ∧
    NoDupA E.eq (elements_dist s) ∧
    (∀ x, InA E.eq x (elements_dist s) ↔ InA E.eq x (elements s)).

Lemma elements_dist_spec1 : ∀ x s, InA E.eq x (elements_dist s) ↔
                                     InA E.eq x (elements s).

Lemma elements_dist_spec2w : ∀ s, NoDupA E.eq (elements_dist s).

End Make.
```