Isabelle Collections Framework

By Peter Lammich and Andreas Lochbihler

March 12, 2013

Abstract

This development provides an efficient, extensible, machine checked collections framework for use in Isabelle/HOL. The library adopts the concepts of interface, implementation and generic algorithm from object-oriented programming and implements them in Isabelle/HOL.

The framework features the use of data refinement techniques to refine an abstract specification (using high-level concepts like sets) to a more concrete implementation (using collection datastructures, like red-black-trees). The code-generator of Isabelle/HOL can be used to generate efficient code in all supported target languages, i.e. Haskell, SML, and OCaml.

Contents

Chapter 1

Introduction

This development provides an efficient, extensible, machine checked collections framework for use in Isabelle/HOL. The library adopts the concepts of interface, implementation and generic algorithm known from object oriented (collection) libraries like the C++ Standard Template Library[3] or the Java Collections Framework[1] and makes them available in the Isabelle/HOL environment.

The library uses data refinement techniques to refine an abstract specification (in terms of high-level concepts such as sets) to a more concrete implementation (based on collection datastructures like red-black-trees). This allows algorithms to be proven on the abstract level at which proofs are simpler because they are not cluttered with low-level details.

The code-generator of Isabelle/HOL can be used to generate efficient code in all supported target languages, i.e. Haskell, SML, and OCaml.

For more documentation and introductory material refer to the userguide (Section 5) and the ITP-2010 paper [2].

1.1 Document Structure

Chapter **??** contains the abstract specifiation of the collections, which are (ordered) maps, (ordered) sets, sequences, finger trees and (unique) priority queues. In chapter 3, generic algorithms implement operations on these collections and adapters between them. Chapter **??** contains the concrete implementations of the data structures and their integration with the Isabelle Collectons Framework. There are implemenations for maps and sets using (associative) lists, red-black trees, hashing and tries. Implementations for finger trees and priority queues can be found an AFP entry of its own. Chapter **??** also contains an overview of all interfaces and implementations. Chapter 5 provides a userguide to the Isabelle Collections Framework. Some examples can be found in chapter 6. Finally, a short conclusion and outlook

to further work is given in 7.

Chapter 2

Specifications

2.1 Specification of Maps

theory *MapSpec*
imports *Main ../iterator/SetIterator*
begin

This theory specifies map operations by means of mapping to HOL's map type, i.e. $'k \rightharpoonup 'v$.

locale *map* =
  fixes $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$ — Abstraction to map datatype
  fixes *invar* :: $'s \Rightarrow$ *bool* — Invariant

2.1.1 Basic Map Functions

Empty Map

locale *map-empty* = *map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *empty* :: *unit* $\Rightarrow$ $'s$
  assumes *empty-correct*:
    α (*empty* ()) = *Map.empty*
    invar (*empty* ())

Lookup

locale *map-lookup* = *map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *lookup* :: $'u \Rightarrow 's \Rightarrow 'v$ *option*
  assumes *lookup-correct*:
    invar $m \Longrightarrow$ *lookup* k $m = \alpha$ m k

Update

locale *map-update* = *map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$

fixes *update* :: $'u \Rightarrow 'v \Rightarrow 's \Rightarrow 's$
assumes *update-correct*:
$invar\ m \Longrightarrow \alpha\ (update\ k\ v\ m) = (\alpha\ m)(k \mapsto v)$
$invar\ m \Longrightarrow invar\ (update\ k\ v\ m)$

Disjoint Update

locale *map-update-dj* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *update-dj* :: $'u \Rightarrow 'v \Rightarrow 's \Rightarrow 's$
assumes *update-dj-correct*:
$[\![invar\ m;\ k \notin dom\ (\alpha\ m)]\!] \Longrightarrow \alpha\ (update\text{-}dj\ k\ v\ m) = (\alpha\ m)(k \mapsto v)$
$[\![invar\ m;\ k \notin dom\ (\alpha\ m)]\!] \Longrightarrow invar\ (update\text{-}dj\ k\ v\ m)$

Delete

locale *map-delete* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *delete* :: $'u \Rightarrow 's \Rightarrow 's$
assumes *delete-correct*:
$invar\ m \Longrightarrow \alpha\ (delete\ k\ m) = (\alpha\ m)\ |`\ (-\{k\})$
$invar\ m \Longrightarrow invar\ (delete\ k\ m)$

Add

locale *map-add* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *add* :: $'s \Rightarrow 's \Rightarrow 's$
assumes *add-correct*:
$invar\ m1 \Longrightarrow invar\ m2 \Longrightarrow \alpha\ (add\ m1\ m2) = \alpha\ m1\ ++\ \alpha\ m2$
$invar\ m1 \Longrightarrow invar\ m2 \Longrightarrow invar\ (add\ m1\ m2)$

locale *map-add-dj* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *add-dj* :: $'s \Rightarrow 's \Rightarrow 's$
assumes *add-dj-correct*:
$[\![invar\ m1;\ invar\ m2;\ dom\ (\alpha\ m1) \cap dom\ (\alpha\ m2) = \{\}]\!] \Longrightarrow \alpha\ (add\text{-}dj\ m1\ m2) = \alpha\ m1\ ++\ \alpha\ m2$
$[\![invar\ m1;\ invar\ m2;\ dom\ (\alpha\ m1) \cap dom\ (\alpha\ m2) = \{\}\ ]\!] \Longrightarrow invar\ (add\text{-}dj\ m1\ m2)$

Emptiness Check

locale *map-isEmpty* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *isEmpty* :: $'s \Rightarrow bool$
assumes *isEmpty-correct* : $invar\ m \Longrightarrow isEmpty\ m \longleftrightarrow \alpha\ m = Map.empty$

Singleton Maps

locale *map-sng* = *map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *sng* :: $'u \Rightarrow 'v \Rightarrow 's$
  assumes *sng-correct* :
    $\alpha\ (sng\ k\ v) = [k \mapsto v]$
    $invar\ (sng\ k\ v)$

locale *map-isSng* = *map* +
  constrains $\alpha :: 's \Rightarrow 'k \rightharpoonup 'v$
  fixes *isSng* :: $'s \Rightarrow bool$
  assumes *isSng-correct*:
    $invar\ s \Longrightarrow isSng\ s \longleftrightarrow (\exists\, k\ v.\ \alpha\ s = [k \mapsto v])$
begin
  lemma *isSng-correct-exists1* :
    $invar\ s \Longrightarrow (isSng\ s \longleftrightarrow (\exists!k.\ \exists\, v.\ (\alpha\ s\ k = Some\ v)))$
    ⟨*proof*⟩

  lemma *isSng-correct-card* :
    $invar\ s \Longrightarrow (isSng\ s \longleftrightarrow (card\ (dom\ (\alpha\ s)) = 1))$
    ⟨*proof*⟩

end

Finite Maps

locale *finite-map* = *map* +
  assumes *finite*[*simp*, *intro*!]: $invar\ m \Longrightarrow finite\ (dom\ (\alpha\ m))$

Size

locale *map-size* = *finite-map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *size* :: $'s \Rightarrow nat$
  assumes *size-correct*: $invar\ s \Longrightarrow size\ s = card\ (dom\ (\alpha\ s))$

locale *map-size-abort* = *finite-map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *size-abort* :: $nat \Rightarrow 's \Rightarrow nat$
  assumes *size-abort-correct*: $invar\ s \Longrightarrow size\text{-}abort\ m\ s = min\ m\ (card\ (dom\ (\alpha\ s)))$

Iterators

An iteration combinator over a map applies a function to a state for each map entry, in arbitrary order. Proving of properties is done by invariant reasoning. An iterator can also contain a continuation condition. Iteration is interrupted if the condition becomes false.

locale *map-iteratei* = *finite-map* +
  constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
  fixes *iteratei* :: $'s \Rightarrow ('u \times 'v, 'σ)$ *set-iterator*

  assumes *iteratei-rule*: *invar m* $\Longrightarrow$ *map-iterator* (*iteratei m*) (α *m*)
begin
  lemma *iteratei-rule-P*:
    assumes *invar m*
      and *I0*: *I* (*dom* (α *m*)) $\sigma 0$
      and *IP*: !!*k v it* σ. ⟦ *c* σ; *k* ∈ *it*; α *m k* = *Some v*; *it* ⊆ *dom* (α *m*); *I it* σ ⟧
          $\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) σ)
      and *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
      and *II*: !!σ *it*. ⟦ *it* ⊆ *dom* (α *m*); *it* ≠ {}; ¬ *c* σ; *I it* σ ⟧ $\Longrightarrow$ *P* σ
    shows *P* (*iteratei m c f* $\sigma 0$)
    ⟨*proof*⟩

  lemma *iteratei-rule-insert-P*:
    assumes
      invar m
      I {} $\sigma 0$
      !!*k v it* σ. ⟦ *c* σ; *k* ∈ (*dom* (α *m*) − *it*); α *m k* = *Some v*; *it* ⊆ *dom* (α *m*); *I* *it* σ ⟧
          $\Longrightarrow$ *I* (*insert k it*) (*f* (*k*, *v*) σ)
      !!σ. *I* (*dom* (α *m*)) σ $\Longrightarrow$ *P* σ
      !!σ *it*. ⟦ *it* ⊆ *dom* (α *m*); *it* ≠ *dom* (α *m*);
            ¬ (*c* σ);
            I it σ ⟧ $\Longrightarrow$ *P* σ
    shows *P* (*iteratei m c f* $\sigma 0$)
    ⟨*proof*⟩

  lemma *iterate-rule-P*:
    ⟦
      invar m;
      I (*dom* (α *m*)) $\sigma 0$;
      !!*k v it* σ. ⟦ *k* ∈ *it*; α *m k* = *Some v*; *it* ⊆ *dom* (α *m*); *I it* σ ⟧
          $\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) σ);
      !!σ. *I* {} σ $\Longrightarrow$ *P* σ
    ⟧ $\Longrightarrow$ *P* (*iteratei m* (λ-. *True*) *f* $\sigma 0$)
    ⟨*proof*⟩

  lemma *iterate-rule-insert-P*:
    ⟦
      invar m;
      I {} $\sigma 0$;
      !!*k v it* σ. ⟦ *k* ∈ (*dom* (α *m*) − *it*); α *m k* = *Some v*; *it* ⊆ *dom* (α *m*); *I it* σ ⟧
          $\Longrightarrow$ *I* (*insert k it*) (*f* (*k*, *v*) σ);
      !!σ. *I* (*dom* (α *m*)) σ $\Longrightarrow$ *P* σ
    ⟧ $\Longrightarrow$ *P* (*iteratei m* (λ-. *True*) *f* $\sigma 0$)

⟨*proof*⟩
end

lemma *map-iteratei-I* :
assumes $\bigwedge m.$ *invar* $m \Longrightarrow$ *map-iterator* (*iti* m) (α m)
shows *map-iteratei* α *invar iti*
⟨*proof*⟩

Bounded Quantification

locale *map-ball* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *ball* :: $'s \Rightarrow ('u \times 'v \Rightarrow bool) \Rightarrow bool$
assumes *ball-correct*: *invar* $m \Longrightarrow$ *ball* $m\ P \longleftrightarrow (\forall u\ v.\ \alpha\ m\ u = Some\ v \longrightarrow P\ (u, v))$

locale *map-bexists* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *bexists* :: $'s \Rightarrow ('u \times 'v \Rightarrow bool) \Rightarrow bool$
assumes *bexists-correct*: *invar* $m \Longrightarrow$ *bexists* $m\ P \longleftrightarrow (\exists u\ v.\ \alpha\ m\ u = Some\ v \wedge P\ (u, v))$

Selection of Entry

locale *map-sel* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *sel* :: $'s \Rightarrow ('u \times 'v \Rightarrow 'r\ option) \Rightarrow 'r\ option$
assumes *selE*:
⟦ *invar* m; $\alpha\ m\ u = Some\ v$; $f\ (u, v) = Some\ r$;
!!$u\ v\ r$. ⟦ *sel* $m\ f = Some\ r$; $\alpha\ m\ u = Some\ v$; $f\ (u, v) = Some\ r$ ⟧ $\Longrightarrow Q$
⟧ $\Longrightarrow Q$
assumes *selI*:
⟦ *invar* m; $\forall u\ v.\ \alpha\ m\ u = Some\ v \longrightarrow f\ (u, v) = None$ ⟧ $\Longrightarrow$ *sel* $m\ f = None$

begin
lemma *sel-someE*:
⟦ *invar* m; *sel* $m\ f = Some\ r$;
!!$u\ v$. ⟦ $\alpha\ m\ u = Some\ v$; $f\ (u, v) = Some\ r$ ⟧ $\Longrightarrow P$
⟧ $\Longrightarrow P$
⟨*proof*⟩

lemma *sel-noneD*: ⟦*invar* m; *sel* $m\ f = None$; $\alpha\ m\ u = Some\ v$⟧ $\Longrightarrow f\ (u, v) = None$
⟨*proof*⟩

end

— Equivalent description of sel-map properties
lemma *map-sel-altI*:
assumes *S1*:

!!$s\ f\ r\ P$. $[\![$ *invar* s; *sel* $s\ f$ = *Some* r;
!!$u\ v$. $[\![\alpha\ s\ u$ = *Some* v; $f\ (u,\ v)$ = *Some* $r]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
assumes *S2*:
!!$s\ f\ u\ v$. $[\![$*invar* s; *sel* $s\ f$ = *None*; $\alpha\ s\ u$ = *Some* $v]\!] \Longrightarrow f\ (u,\ v)$ = *None*
shows *map-sel* α *invar sel*
⟨*proof*⟩

Selection of Entry (without mapping)

locale *map-sel′* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *sel′* :: $'s \Rightarrow ('u \times 'v \Rightarrow bool) \Rightarrow ('u \times 'v)$ *option*
assumes *sel′E*:
$[\![$ *invar* m; $\alpha\ m\ u$ = *Some* v; $P\ (u,\ v)$;
!!$u\ v$. $[\![$ *sel′* $m\ P$ = *Some* (u,v); $\alpha\ m\ u$ = *Some* v; $P\ (u,\ v)]\!] \Longrightarrow Q$
$]\!] \Longrightarrow Q$
assumes *sel′I*:
$[\![$ *invar* m; $\forall u\ v.\ \alpha\ m\ u$ = *Some* $v \longrightarrow \neg\ P\ (u,\ v)\ ]\!] \Longrightarrow$ *sel′* $m\ P$ = *None*

begin
lemma *sel′-someE*:
$[\![$ *invar* m; *sel′* $m\ P$ = *Some* (u,v);
!!$u\ v$. $[\![\ \alpha\ m\ u$ = *Some* v; $P\ (u,\ v)\ ]\!] \Longrightarrow$ *thesis*
$]\!] \Longrightarrow$ *thesis*
⟨*proof*⟩

lemma *sel′-noneD*: $[\![$*invar* m; *sel′* $m\ P$ = *None*; $\alpha\ m\ u$ = *Some* $v]\!] \Longrightarrow \neg\ P\ (u,\ v)$
⟨*proof*⟩

lemma *sel′-SomeD*:
$[\![$ *sel′* $m\ P$ = *Some* $(u,\ v)$; *invar* $m\ ]\!] \Longrightarrow \alpha\ m\ u$ = *Some* $v \wedge P\ (u,\ v)$
⟨*proof*⟩
end

Map to List Conversion

locale *map-to-list* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *to-list* :: $'s \Rightarrow ('u \times 'v)$ *list*
assumes *to-list-correct*:
invar $m \Longrightarrow$ *map-of* (*to-list* m) = $\alpha\ m$
invar $m \Longrightarrow$ *distinct* (*map fst* (*to-list* m))

List to Map Conversion

locale *list-to-map* = *map* +
constrains $\alpha :: 's \Rightarrow 'u \rightharpoonup 'v$
fixes *to-map* :: $('u \times 'v)$ *list* $\Rightarrow 's$

assumes *to-map-correct*:
α (*to-map l*) = *map-of l*
invar (*to-map l*)

Image of a Map

This locale allows to apply a function to both the keys and the values of a map while at the same time filtering entries.

definition *transforms-to-unique-keys* ::
('*u1* ⇀ '*v1*) ⇒ ('*u1* × '*v1* ⇀ ('*u2* × '*v2*)) ⇒ *bool*
where
transforms-to-unique-keys m f ≡ (∀ *k1 k2 v1 v2 k' v1' v2'*. (
m k1 = *Some v1* ∧
m k2 = *Some v2* ∧
f (*k1*, *v1*) = *Some* (*k'*, *v1'*) ∧
f (*k2*, *v2*) = *Some* (*k'*, *v2'*)) −−>
(*k1* = *k2*))

locale *map-image-filter* = *map* α*1 invar1* + *map* α*2 invar2*
for α*1* :: '*m1* ⇒ '*u1* ⇀ '*v1* **and** *invar1*
and α*2* :: '*m2* ⇒ '*u2* ⇀ '*v2* **and** *invar2*
+
fixes *map-image-filter* :: ('*u1* × '*v1* ⇒ ('*u2* × '*v2*) *option*) ⇒ '*m1* ⇒ '*m2*
assumes *map-image-filter-correct-aux1*:
⋀*k' v'*.
⟦*invar1 m*; *transforms-to-unique-keys* (α*1 m*) *f*⟧ ⟹
(*invar2* (*map-image-filter f m*) ∧
((α*2* (*map-image-filter f m*) *k'* = *Some v'*) ⟷
(∃ *k v*. (α*1 m k* = *Some v*) ∧ *f* (*k*, *v*) = *Some* (*k'*, *v'*))))
begin

lemma *map-image-filter-correct-aux2* :
assumes *invar1 m*
and *transforms-to-unique-keys* (α*1 m*) *f*
shows (α*2* (*map-image-filter f m*) *k'* = *None*) ⟷
(∀ *k v v'*. α*1 m k* = *Some v* ⟶ *f* (*k*, *v*) ≠ *Some* (*k'*, *v'*))
⟨*proof*⟩

lemmas *map-image-filter-correct* =
conjunct1 [*OF map-image-filter-correct-aux1*]
conjunct2 [*OF map-image-filter-correct-aux1*]
map-image-filter-correct-aux2
end

Most of the time the mapping function is only applied to values. Then, the precondition disapears.

locale *map-value-image-filter* = *map* α*1 invar1* + *map* α*2 invar2*

```
  for α1 :: 'm1 ⇒ 'u ⇀ 'v1 and invar1
  and α2 :: 'm2 ⇒ 'u ⇀ 'v2 and invar2
  +
  fixes map-value-image-filter :: ('u ⇒ 'v1 ⇒ 'v2 option) ⇒ 'm1 ⇒ 'm2
  assumes map-value-image-filter-correct :
    invar1 m ⟹
     invar2 (map-value-image-filter f m) ∧
     (α2 (map-value-image-filter f m) =
      (λk. Option.bind (α1 m k) (f k)))
begin

  lemma map-value-image-filter-correct-alt :
    invar1 m ⟹
     invar2 (map-value-image-filter f m)
    invar1 m ⟹
     (α2 (map-value-image-filter f m) k = Some v') ⟷
     (∃ v. (α1 m k = Some v) ∧ f k v = Some v')
    invar1 m ⟹
     (α2 (map-value-image-filter f m) k = None) ⟷
     (∀ v. (α1 m k = Some v) −−> f k v = None)
  ⟨proof⟩
end


locale map-restrict = map α1 invar1 + map α2 invar2
  for α1 :: 'm1 ⇒ 'u ⇀ 'v and invar1
  and α2 :: 'm2 ⇒ 'u ⇀ 'v and invar2
  +
  fixes restrict :: ('u × 'v ⇒ bool) ⇒ 'm1 ⇒ 'm2
  assumes restrict-correct-aux1 :
    invar1 m ⟹ α2 (restrict P m) = α1 m |` {k. ∃ v. α1 m k = Some v ∧ P (k,
v)}
    invar1 m ⟹ invar2 (restrict P m)
begin
  lemma restrict-correct-aux2 :
    invar1 m ⟹ α2 (restrict (λ(k,-). P k) m) = α1 m |` {k. P k}
  ⟨proof⟩

  lemmas restrict-correct =
     restrict-correct-aux1
     restrict-correct-aux2
end
```

2.1.2 Ordered Maps

```
  locale ordered-map = map α invar
    for α :: 's ⇒ ('u::linorder) ⇀ 'v and invar

  locale ordered-finite-map = finite-map α invar + ordered-map α invar
```

for $\alpha :: 's \Rightarrow ('u::linorder) \rightharpoonup 'v$ **and** *invar*

Ordered Iteration

locale *map-iterateoi* = *ordered-finite-map* α *invar*
for $\alpha :: 's \Rightarrow ('u::linorder) \rightharpoonup 'v$ **and** *invar*
+
fixes *iterateoi* :: $'s \Rightarrow ('u \times 'v,'\sigma)$ *set-iterator*
assumes *iterateoi-rule*:
invar m $\Longrightarrow$ *map-iterator-linord* (*iterateoi m*) (α *m*)
begin
lemma *iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
assumes *MINV*: *invar m*
assumes *I0*: *I* (*dom* (α *m*)) $\sigma 0$
assumes *IP*: !!*k v it* σ. ⟦
c σ;
k $\in$ *it*;
$\forall j \in it.\ k \leq j$;
$\forall j \in dom\ (\alpha\ m) - it.\ j \leq k$;
$\alpha\ m\ k = Some\ v$;
it $\subseteq$ *dom* (α *m*);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) σ)
assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow$ *P* σ
assumes *II*: !!σ *it*. ⟦
it $\subseteq$ *dom* (α *m*);
it $\neq$ {};
¬ *c* σ;
I it σ;
$\forall k \in it.\ \forall j \in dom\ (\alpha\ m) - it.\ j \leq k$
⟧ $\Longrightarrow$ *P* σ
shows *P* (*iterateoi m c f* $\sigma 0$)
⟨*proof*⟩

lemma *iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar m*
assumes *I0*: *I* (*dom* (α *m*)) $\sigma 0$
assumes *IP*: !!*k v it* σ. ⟦ *k* $\in$ *it*; $\forall j \in it.\ k \leq j$; $\forall j \in dom\ (\alpha\ m) - it.\ j \leq k$; α *m k* = *Some v*; *it* $\subseteq$ *dom* (α *m*); *I it* σ ⟧
$\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) σ)
assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow$ *P* σ
shows *P* (*iterateoi m* (λ-. *True*) *f* $\sigma 0$)
⟨*proof*⟩
end

lemma *map-iterateoi-I* :
assumes $\bigwedge m.$ *invar m* $\Longrightarrow$ *map-iterator-linord* (*itoi m*) (α *m*)
shows *map-iterateoi* α *invar itoi*
⟨*proof*⟩

locale *map-reverse-iterateoi* = *ordered-finite-map* α *invar*
  for α :: $'s \Rightarrow ('u::linorder) \rightharpoonup 'v$ **and** *invar*
  +
  fixes *reverse-iterateoi* :: $'s \Rightarrow ('u \times 'v, 'σ)$ *set-iterator*
  assumes *reverse-iterateoi-rule*:
    invar m $\Longrightarrow$ *map-iterator-rev-linord* (*reverse-iterateoi m*) (α *m*)
begin
 lemma *reverse-iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
    assumes *MINV*: *invar m*
    assumes *I0*: *I* (*dom* (α *m*)) *σ0*
    assumes *IP*: !!*k v it σ*. ⟦
      c σ;
      k ∈ *it*;
      ∀*j*∈*it*. *k*≥*j*;
      ∀*j*∈*dom* (α *m*) − *it*. *j*≥*k*;
      α *m k* = *Some v*;
      it ⊆ *dom* (α *m*);
      I it σ
    ⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) *σ*)
    assumes *IF*: !!*σ*. *I* {} *σ* $\Longrightarrow$ *P σ*
    assumes *II*: !!*σ it*. ⟦
      it ⊆ *dom* (α *m*);
      it ≠ {};
      ¬ *c σ*;
      I it σ;
      ∀*k*∈*it*. ∀*j*∈*dom* (α *m*) − *it*. *j*≥*k*
    ⟧ $\Longrightarrow$ *P σ*
    shows *P* (*reverse-iterateoi m c f σ0*)
  ⟨*proof*⟩

  lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
    assumes *MINV*: *invar m*
    assumes *I0*: *I* (*dom* (α *m*)) *σ0*
    assumes *IP*: !!*k v it σ*. ⟦
      k ∈ *it*;
      ∀*j*∈*it*. *k*≥*j*;
      ∀*j*∈*dom* (α *m*) − *it*. *j*≥*k*;
      α *m k* = *Some v*;
      it ⊆ *dom* (α *m*);
      I it σ
    ⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f* (*k*, *v*) *σ*)
    assumes *IF*: !!*σ*. *I* {} *σ* $\Longrightarrow$ *P σ*
    shows *P* (*reverse-iterateoi m* (λ-. *True*) *f σ0*)
  ⟨*proof*⟩
end

lemma *map-reverse-iterateoi-I* :
assumes $\bigwedge$*m*. *invar m* $\Longrightarrow$ *map-iterator-rev-linord* (*ritoi m*) (α *m*)

shows *map-reverse-iterateoi* α *invar ritoi*
⟨*proof*⟩

Minimal and Maximal Elements

locale *map-min* = *ordered-map* +
constrains α :: $'s \Rightarrow 'u$::*linorder* $\rightharpoonup 'v$
fixes *min* :: $'s \Rightarrow ('u \times 'v \Rightarrow bool) \Rightarrow ('u \times 'v)$ *option*
assumes *min-correct*:
⟦ *invar s*; *rel-of* (α *s*) *P* ≠ {} ⟧ ⟹ *min s P* ∈ *Some* ‘ *rel-of* (α *s*) *P*
⟦ *invar s*; (*k*,*v*) ∈ *rel-of* (α *s*) *P* ⟧ ⟹ *fst* (*the* (*min s P*)) ≤ *k*
⟦ *invar s*; *rel-of* (α *s*) *P* = {} ⟧ ⟹ *min s P* = *None*
begin
lemma *minE*:
assumes *A*: *invar s* *rel-of* (α *s*) *P* ≠ {}
obtains *k v* **where**
min s P = *Some* (*k*,*v*) (*k*,*v*)∈*rel-of* (α *s*) *P* ∀ (*k′*,*v′*)∈*rel-of* (α *s*) *P*. *k* ≤ *k′*
⟨*proof*⟩

lemmas *minI* = *min-correct*(*3*)

lemma *min-Some*:
⟦ *invar s*; *min s P* = *Some* (*k*,*v*) ⟧ ⟹ (*k*,*v*)∈*rel-of* (α *s*) *P*
⟦ *invar s*; *min s P* = *Some* (*k*,*v*); (*k′*,*v′*)∈*rel-of* (α *s*) *P* ⟧ ⟹ *k*≤*k′*
⟨*proof*⟩

lemma *min-None*:
⟦ *invar s*; *min s P* = *None* ⟧ ⟹ *rel-of* (α *s*) *P* = {}
⟨*proof*⟩

end

locale *map-max* = *ordered-map* +
constrains α :: $'s \Rightarrow 'u$::*linorder* $\rightharpoonup 'v$
fixes *max* :: $'s \Rightarrow ('u \times 'v \Rightarrow bool) \Rightarrow ('u \times 'v)$ *option*
assumes *max-correct*:
⟦ *invar s*; *rel-of* (α *s*) *P* ≠ {} ⟧ ⟹ *max s P* ∈ *Some* ‘ *rel-of* (α *s*) *P*
⟦ *invar s*; (*k*,*v*) ∈ *rel-of* (α *s*) *P* ⟧ ⟹ *fst* (*the* (*max s P*)) ≥ *k*
⟦ *invar s*; *rel-of* (α *s*) *P* = {} ⟧ ⟹ *max s P* = *None*
begin
lemma *maxE*:
assumes *A*: *invar s* *rel-of* (α *s*) *P* ≠ {}
obtains *k v* **where**
max s P = *Some* (*k*,*v*) (*k*,*v*)∈*rel-of* (α *s*) *P* ∀ (*k′*,*v′*)∈*rel-of* (α *s*) *P*. *k* ≥ *k′*
⟨*proof*⟩

lemmas *maxI* = *max-correct*(*3*)

lemma *max-Some*:
⟦ *invar s*; *max s P* = *Some* (*k*,*v*) ⟧ ⟹ (*k*,*v*)∈*rel-of* (*α s*) *P*
⟦ *invar s*; *max s P* = *Some* (*k*,*v*); (*k'*,*v'*)∈*rel-of* (*α s*) *P* ⟧ ⟹ *k*≥*k'*
⟨*proof*⟩

lemma *max-None*:
⟦ *invar s*; *max s P* = *None* ⟧ ⟹ *rel-of* (*α s*) *P* = {}
⟨*proof*⟩

end

Conversion to List

locale *map-to-sorted-list* = *ordered-map α invar* + *map-to-list α invar to-list*
for *α* :: *'s* ⇒ *'u*::*linorder* ⇀ *'v* **and** *invar to-list* +
assumes *to-list-sorted*:
invar m ⟹ *sorted* (*map fst* (*to-list m*))

2.1.3 Record Based Interface

record (*'k*,*'v*,*'s*) *map-ops* =
map-op-α :: *'s* ⇒ *'k* ⇀ *'v*
map-op-invar :: *'s* ⇒ *bool*
map-op-empty :: *unit* ⇒ *'s*
map-op-sng :: *'k* ⇒ *'v* ⇒ *'s*
map-op-lookup :: *'k* ⇒ *'s* ⇒ *'v option*
map-op-update :: *'k* ⇒ *'v* ⇒ *'s* ⇒ *'s*
map-op-update-dj :: *'k* ⇒ *'v* ⇒ *'s* ⇒ *'s*
map-op-delete :: *'k* ⇒ *'s* ⇒ *'s*
map-op-restrict :: (*'k* × *'v* ⇒ *bool*) ⇒ *'s* ⇒ *'s*
map-op-add :: *'s* ⇒ *'s* ⇒ *'s*
map-op-add-dj :: *'s* ⇒ *'s* ⇒ *'s*
map-op-isEmpty :: *'s* ⇒ *bool*
map-op-isSng :: *'s* ⇒ *bool*
map-op-ball :: *'s* ⇒ (*'k* × *'v* ⇒ *bool*) ⇒ *bool*
map-op-bexists :: *'s* ⇒ (*'k* × *'v* ⇒ *bool*) ⇒ *bool*
map-op-size :: *'s* ⇒ *nat*
map-op-size-abort :: *nat* ⇒ *'s* ⇒ *nat*
map-op-sel :: *'s* ⇒ (*'k* × *'v* ⇒ *bool*) ⇒ (*'k*×*'v*) *option* — Version without mapping
map-op-to-list :: *'s* ⇒ (*'k*×*'v*) *list*
map-op-to-map :: (*'k*×*'v*) *list* ⇒ *'s*

locale *StdMapDefs* =
fixes *ops* :: (*'k*,*'v*,*'s*,*'more*) *map-ops-scheme*
begin
abbreviation *α* **where** *α* == *map-op-α ops*
abbreviation *invar* **where** *invar* == *map-op-invar ops*
abbreviation *empty* **where** *empty* == *map-op-empty ops*

abbreviation *sng* **where** *sng* == *map-op-sng ops*
abbreviation *lookup* **where** *lookup* == *map-op-lookup ops*
abbreviation *update* **where** *update* == *map-op-update ops*
abbreviation *update-dj* **where** *update-dj* == *map-op-update-dj ops*
abbreviation *delete* **where** *delete* == *map-op-delete ops*
abbreviation *restrict* **where** *restrict* == *map-op-restrict ops*
abbreviation *add* **where** *add* == *map-op-add ops*
abbreviation *add-dj* **where** *add-dj* == *map-op-add-dj ops*
abbreviation *isEmpty* **where** *isEmpty* == *map-op-isEmpty ops*
abbreviation *isSng* **where** *isSng* == *map-op-isSng ops*
abbreviation *ball* **where** *ball* == *map-op-ball ops*
abbreviation *bexists* **where** *bexists* == *map-op-bexists ops*
abbreviation *size* **where** *size* == *map-op-size ops*
abbreviation *size-abort* **where** *size-abort* == *map-op-size-abort ops*
abbreviation *sel* **where** *sel* == *map-op-sel ops*
abbreviation *to-list* **where** *to-list* == *map-op-to-list ops*
abbreviation *to-map* **where** *to-map* == *map-op-to-map ops*
end

locale *StdMap* = *StdMapDefs ops* +
map α *invar* +
map-empty α *invar empty* +
map-sng α *invar sng* +
map-lookup α *invar lookup* +
map-update α *invar update* +
map-update-dj α *invar update-dj* +
map-delete α *invar delete* +
map-restrict α *invar* α *invar restrict* +
map-add α *invar add* +
map-add-dj α *invar add-dj* +
map-isEmpty α *invar isEmpty* +
map-isSng α *invar isSng* +
map-ball α *invar ball* +
map-bexists α *invar bexists* +
map-size α *invar size* +
map-size-abort α *invar size-abort* +
map-sel' α *invar sel* +
map-to-list α *invar to-list* +
list-to-map α *invar to-map*
for *ops*
begin
lemmas *correct* =
empty-correct
sng-correct
lookup-correct
update-correct
update-dj-correct
delete-correct

restrict-correct
add-correct
add-dj-correct
isEmpty-correct
isSng-correct
ball-correct
bexists-correct
size-correct
size-abort-correct
to-list-correct
to-map-correct
end

record $('k,'v,'s)$ *omap-ops* = $('k,'v,'s)$ *map-ops* +
map-op-min :: $'s \Rightarrow ('k \times 'v \Rightarrow bool) \Rightarrow ('k \times 'v)$ *option*
map-op-max :: $'s \Rightarrow ('k \times 'v \Rightarrow bool) \Rightarrow ('k \times 'v)$ *option*

locale *StdOMapDefs* = *StdMapDefs ops*
for *ops* :: $('k{::}linorder,'v,'s,'more)$ *omap-ops-scheme*
begin
abbreviation *min* **where** *min* == *map-op-min ops*
abbreviation *max* **where** *max* == *map-op-max ops*
end

locale *StdOMap* =
StdOMapDefs ops +
StdMap ops +
map-min α *invar min* +
map-max α *invar max*
for *ops*
begin
end

end

2.2 Specification of Sets

theory *SetSpec*
imports
Main
../common/Misc
../iterator/SetIterator
begin

This theory specifies set operations by means of a mapping to HOL's standard sets.

notation *insert* (*set'-ins*)

locale *set* =
— Abstraction to set
fixes $\alpha :: 's \Rightarrow 'x\ set$
— Invariant
fixes *invar* :: $'s \Rightarrow bool$

2.2.1 Basic Set Functions

Empty set

locale *set-empty* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *empty* :: $unit \Rightarrow 's$
assumes *empty-correct*:
$\alpha\ (empty\ ()) = \{\}$
$invar\ (empty\ ())$

Membership Query

locale *set-memb* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *memb* :: $'x \Rightarrow 's \Rightarrow bool$
assumes *memb-correct*:
$invar\ s \Longrightarrow memb\ x\ s \longleftrightarrow x \in \alpha\ s$

Insertion of Element

locale *set-ins* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *ins* :: $'x \Rightarrow 's \Rightarrow 's$
assumes *ins-correct*:
$invar\ s \Longrightarrow \alpha\ (ins\ x\ s) = set\text{-}ins\ x\ (\alpha\ s)$
$invar\ s \Longrightarrow invar\ (ins\ x\ s)$

Disjoint Insert

locale *set-ins-dj* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *ins-dj* :: $'x \Rightarrow 's \Rightarrow 's$
assumes *ins-dj-correct*:
$[\![invar\ s;\ x \notin \alpha\ s]\!] \Longrightarrow \alpha\ (ins\text{-}dj\ x\ s) = set\text{-}ins\ x\ (\alpha\ s)$
$[\![invar\ s;\ x \notin \alpha\ s]\!] \Longrightarrow invar\ (ins\text{-}dj\ x\ s)$

Deletion of Single Element

locale *set-delete* = *set* +

constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *delete* $:: 'x \Rightarrow 's \Rightarrow 's$
assumes *delete-correct*:
$invar\ s \Longrightarrow \alpha\ (delete\ x\ s) = \alpha\ s - \{x\}$
$invar\ s \Longrightarrow invar\ (delete\ x\ s)$

Emptiness Check

locale *set-isEmpty* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *isEmpty* $:: 's \Rightarrow bool$
assumes *isEmpty-correct*:
$invar\ s \Longrightarrow isEmpty\ s \longleftrightarrow \alpha\ s = \{\}$

Bounded Quantifiers

locale *set-ball* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *ball* $:: 's \Rightarrow ('x \Rightarrow bool) \Rightarrow bool$
assumes *ball-correct*: $invar\ S \Longrightarrow ball\ S\ P \longleftrightarrow (\forall\, x{\in}\alpha\ S.\ P\ x)$

locale *set-bexists* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *bexists* $:: 's \Rightarrow ('x \Rightarrow bool) \Rightarrow bool$
assumes *bexists-correct*: $invar\ S \Longrightarrow bexists\ S\ P \longleftrightarrow (\exists\, x{\in}\alpha\ S.\ P\ x)$

Finite Set

locale *finite-set* = *set* +
assumes *finite*[*simp*, *intro*!]: $invar\ s \Longrightarrow finite\ (\alpha\ s)$

Size

locale *set-size* = *finite-set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *size* $:: 's \Rightarrow nat$
assumes *size-correct*:
$invar\ s \Longrightarrow size\ s = card\ (\alpha\ s)$

locale *set-size-abort* = *finite-set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *size-abort* $:: nat \Rightarrow 's \Rightarrow nat$
assumes *size-abort-correct*:
$invar\ s \Longrightarrow size\text{-}abort\ m\ s = min\ m\ (card\ (\alpha\ s))$

Singleton sets

locale *set-sng* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *sng* $:: 'x \Rightarrow 's$

assumes *sng-correct*:
$\alpha\ (sng\ x) = \{x\}$
$invar\ (sng\ x)$

locale *set-isSng* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes $isSng :: 's \Rightarrow bool$
assumes *isSng-correct*:
$invar\ s \Longrightarrow isSng\ s \longleftrightarrow (\exists\, e.\ \alpha\ s = \{e\})$
begin
lemma *isSng-correct-exists1* :
$invar\ s \Longrightarrow (isSng\ s \longleftrightarrow (\exists !e.\ (e \in \alpha\ s)))$
⟨*proof*⟩

lemma *isSng-correct-card* :
$invar\ s \Longrightarrow (isSng\ s \longleftrightarrow (card\ (\alpha\ s) = 1))$
⟨*proof*⟩
end

2.2.2 Iterators

An iterator applies a function to a state and all the elements of the set. The function is applied in any order. Proofs over the iteration are done by establishing invariants over the iteration. Iterators may have a break-condition, that interrupts the iteration before the last element has been visited.

locale *set-iteratei* = *finite-set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes $iteratei :: 's \Rightarrow ('x, 'σ)\ set\text{-}iterator$

assumes *iteratei-rule*: $invar\ S \Longrightarrow set\text{-}iterator\ (iteratei\ S)\ (\alpha\ S)$
begin
lemma *iteratei-rule-P*:
$[\![$
$invar\ S;$
$I\ (\alpha\ S)\ σ0;$
$!!x\ it\ σ.\ [\![\ c\ σ;\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ σ\ ]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ σ);$
$!!σ.\ I\ \{\}\ σ \Longrightarrow P\ σ;$
$!!σ\ it.\ [\![\ it \subseteq \alpha\ S;\ it \neq \{\};\ \neg\ c\ σ;\ I\ it\ σ\ ]\!] \Longrightarrow P\ σ$
$]\!] \Longrightarrow P\ (iteratei\ S\ c\ f\ σ0)$
⟨*proof*⟩

lemma *iteratei-rule-insert-P*:
$[\![$
$invar\ S;$
$I\ \{\}\ σ0;$
$!!x\ it\ σ.\ [\![\ c\ σ;\ x \in \alpha\ S - it;\ it \subseteq \alpha\ S;\ I\ it\ σ\ ]\!] \Longrightarrow I\ (insert\ x\ it)\ (f\ x\ σ);$
$!!σ.\ I\ (\alpha\ S)\ σ \Longrightarrow P\ σ;$

!!σ *it*. ⟦ *it* ⊆ α *S*; *it* ≠ α *S*; ¬ *c* σ; *I it* σ ⟧ ⟹ *P* σ
⟧ ⟹ *P* (*iteratei S c f* σ*0*)
⟨*proof*⟩

Versions without break condition.

lemma *iterate-rule-P*:
⟦
invar S;
I (α *S*) σ*0*;
!!*x it* σ. ⟦ *x* ∈ *it*; *it* ⊆ α *S*; *I it* σ ⟧ ⟹ *I* (*it* − {*x*}) (*f x* σ);
!!σ. *I* {} σ ⟹ *P* σ
⟧ ⟹ *P* (*iteratei S* (λ-. *True*) *f* σ*0*)
⟨*proof*⟩

lemma *iterate-rule-insert-P*:
⟦
invar S;
I {} σ*0*;
!!*x it* σ. ⟦ *x* ∈ α *S* − *it*; *it* ⊆ α *S*; *I it* σ ⟧ ⟹ *I* (*insert x it*) (*f x* σ);
!!σ. *I* (α *S*) σ ⟹ *P* σ
⟧ ⟹ *P* (*iteratei S* (λ-. *True*) *f* σ*0*)
⟨*proof*⟩
end

lemma *set-iteratei-I* :
assumes ⋀*s*. *invar s* ⟹ *set-iterator* (*iti s*) (α *s*)
shows *set-iteratei* α *invar iti*
⟨*proof*⟩

2.2.3 More Set Operations

Copy

locale *set-copy* = *set* α*1 invar1* + *set* α*2 invar2*
for α*1* :: *'s1* ⇒ *'a set* **and** *invar1*
and α*2* :: *'s2* ⇒ *'a set* **and** *invar2*
+
fixes *copy* :: *'s1* ⇒ *'s2*
assumes *copy-correct*:
invar1 s1 ⟹ α*2* (*copy s1*) = α*1 s1*
invar1 s1 ⟹ *invar2* (*copy s1*)

Union

locale *set-union* = *set* α*1 invar1* + *set* α*2 invar2* + *set* α*3 invar3*
for α*1* :: *'s1* ⇒ *'a set* **and** *invar1*
and α*2* :: *'s2* ⇒ *'a set* **and** *invar2*
and α*3* :: *'s3* ⇒ *'a set* **and** *invar3*
+
fixes *union* :: *'s1* ⇒ *'s2* ⇒ *'s3*

assumes *union-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ $\alpha 3$ (*union s1 s2*) = $\alpha 1$ *s1* $\cup$ $\alpha 2$ *s2*
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *invar3* (*union s1 s2*)

locale *set-union-dj* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2* + *set* $\alpha 3$ *invar3*
for $\alpha 1$ :: *'s1* $\Rightarrow$ *'a set* **and** *invar1*
and $\alpha 2$ :: *'s2* $\Rightarrow$ *'a set* **and** *invar2*
and $\alpha 3$ :: *'s3* $\Rightarrow$ *'a set* **and** *invar3*
+
fixes *union-dj* :: *'s1* $\Rightarrow$ *'s2* $\Rightarrow$ *'s3*
assumes *union-dj-correct*:
⟦*invar1 s1*; *invar2 s2*; $\alpha 1$ *s1* $\cap$ $\alpha 2$ *s2* = {}⟧ $\Longrightarrow$ $\alpha 3$ (*union-dj s1 s2*) = $\alpha 1$ *s1* $\cup$ $\alpha 2$ *s2*
⟦*invar1 s1*; *invar2 s2*; $\alpha 1$ *s1* $\cap$ $\alpha 2$ *s2* = {}⟧ $\Longrightarrow$ *invar3* (*union-dj s1 s2*)

locale *set-union-list* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1$ :: *'s1* $\Rightarrow$ *'a set* **and** *invar1*
and $\alpha 2$ :: *'s2* $\Rightarrow$ *'a set* **and** *invar2*
+
fixes *union-list* :: *'s1 list* $\Rightarrow$ *'s2*
assumes *union-list-correct*:
$\forall$ *s1*$\in$*set l*. *invar1 s1* $\Longrightarrow$ $\alpha 2$ (*union-list l*) = $\bigcup$ {$\alpha 1$ *s1* |*s1*. *s1* $\in$ *set l*}
$\forall$ *s1*$\in$*set l*. *invar1 s1* $\Longrightarrow$ *invar2* (*union-list l*)

Difference

locale *set-diff* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1$ :: *'s1* $\Rightarrow$ *'a set* **and** *invar1*
and $\alpha 2$ :: *'s2* $\Rightarrow$ *'a set* **and** *invar2*
+
fixes *diff* :: *'s1* $\Rightarrow$ *'s2* $\Rightarrow$ *'s1*
assumes *diff-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ $\alpha 1$ (*diff s1 s2*) = $\alpha 1$ *s1* − $\alpha 2$ *s2*
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *invar1* (*diff s1 s2*)

Intersection

locale *set-inter* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2* + *set* $\alpha 3$ *invar3*
for $\alpha 1$ :: *'s1* $\Rightarrow$ *'a set* **and** *invar1*
and $\alpha 2$ :: *'s2* $\Rightarrow$ *'a set* **and** *invar2*
and $\alpha 3$ :: *'s3* $\Rightarrow$ *'a set* **and** *invar3*
+
fixes *inter* :: *'s1* $\Rightarrow$ *'s2* $\Rightarrow$ *'s3*
assumes *inter-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ $\alpha 3$ (*inter s1 s2*) = $\alpha 1$ *s1* $\cap$ $\alpha 2$ *s2*
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *invar3* (*inter s1 s2*)

Subset

locale *set-subset* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a$ *set* **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a$ *set* **and** *invar2*
+
fixes *subset* :: $'s1 \Rightarrow 's2 \Rightarrow$ *bool*
assumes *subset-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *subset s1 s2* $\longleftrightarrow$ $\alpha 1$ *s1* $\subseteq$ $\alpha 2$ *s2*

Equal

locale *set-equal* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a$ *set* **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a$ *set* **and** *invar2*
+
fixes *equal* :: $'s1 \Rightarrow 's2 \Rightarrow$ *bool*
assumes *equal-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *equal s1 s2* $\longleftrightarrow$ $\alpha 1$ *s1* = $\alpha 2$ *s2*

Image and Filter

locale *set-image-filter* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a$ *set* **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'b$ *set* **and** *invar2*
+
fixes *image-filter* :: $('a \Rightarrow 'b$ *option*$) \Rightarrow 's1 \Rightarrow 's2$
assumes *image-filter-correct-aux*:
invar1 s $\Longrightarrow$ $\alpha 2$ (*image-filter f s*) = { *b* . $\exists\, a{\in}\alpha 1$ *s*. *f a* = *Some b* }
invar1 s $\Longrightarrow$ *invar2* (*image-filter f s*)
begin
— This special form will be checked first by the simplifier:
lemma *image-filter-correct-aux2*:
invar1 s $\Longrightarrow$
$\alpha 2$ (*image-filter* (λx. *if P x then* (*Some* (*f x*)) *else None*) *s*)
= *f* ‘ {$x{\in}\alpha 1$ *s*. *P x*}
⟨*proof*⟩

lemmas *image-filter-correct* =
image-filter-correct-aux2 image-filter-correct-aux

end

locale *set-inj-image-filter* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a$ *set* **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'b$ *set* **and** *invar2*
+
fixes *inj-image-filter* :: $('a \Rightarrow 'b$ *option*$) \Rightarrow 's1 \Rightarrow 's2$
assumes *inj-image-filter-correct*:
⟦*invar1 s*; *inj-on f* ($\alpha 1$ *s* $\cap$ *dom f*)⟧ $\Longrightarrow$ $\alpha 2$ (*inj-image-filter f s*) = { *b* . $\exists\, a{\in}\alpha 1$

$s.\ f\ a = Some\ b\ \}$
$[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s \cap dom\ f)]\!] \Longrightarrow invar2\ (inj\text{-}image\text{-}filter\ f\ s)$

Image

locale *set-image* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'b\ set$ **and** *invar2*
+
fixes *image* :: $('a \Rightarrow 'b) \Rightarrow 's1 \Rightarrow 's2$
assumes *image-correct*:
$invar1\ s \Longrightarrow \alpha 2\ (image\ f\ s) = f`\alpha 1\ s$
$invar1\ s \Longrightarrow invar2\ (image\ f\ s)$

locale *set-inj-image* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'b\ set$ **and** *invar2*
+
fixes *inj-image* :: $('a \Rightarrow 'b) \Rightarrow 's1 \Rightarrow 's2$
assumes *inj-image-correct*:
$[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s)]\!] \Longrightarrow \alpha 2\ (inj\text{-}image\ f\ s) = f`\alpha 1\ s$
$[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s)]\!] \Longrightarrow invar2\ (inj\text{-}image\ f\ s)$

Filter

locale *set-filter* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a\ set$ **and** *invar2*
+
fixes *filter* :: $('a \Rightarrow bool) \Rightarrow 's1 \Rightarrow 's2$
assumes *filter-correct*:
$invar1\ s \Longrightarrow \alpha 2\ (filter\ P\ s) = \{e.\ e \in \alpha 1\ s \wedge P\ e\}$
$invar1\ s \Longrightarrow invar2\ (filter\ P\ s)$

Union of Set of Sets

locale *set-Union-image* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2* + *set* $\alpha 3$ *invar3*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'b\ set$ **and** *invar2*
and $\alpha 3 :: 's3 \Rightarrow 'b\ set$ **and** *invar3*
+
fixes *Union-image* :: $('a \Rightarrow 's2) \Rightarrow 's1 \Rightarrow 's3$
assumes *Union-image-correct*:
$[\![\ invar1\ s;\ !!x.\ x{\in}\alpha 1\ s \Longrightarrow invar2\ (f\ x)\ ]\!] \Longrightarrow \alpha 3\ (Union\text{-}image\ f\ s) = \bigcup \alpha 2`f`\alpha 1\ s$
$[\![\ invar1\ s;\ !!x.\ x{\in}\alpha 1\ s \Longrightarrow invar2\ (f\ x)\ ]\!] \Longrightarrow invar3\ (Union\text{-}image\ f\ s)$

Disjointness Check

locale *set-disjoint* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a\ set$ **and** *invar2*
+
fixes *disjoint* :: $'s1 \Rightarrow 's2 \Rightarrow bool$
assumes *disjoint-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow disjoint\ s1\ s2 \longleftrightarrow \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}$

Disjointness Check With Witness

locale *set-disjoint-witness* = *set* $\alpha 1$ *invar1* + *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a\ set$ **and** *invar2*
+
fixes *disjoint-witness* :: $'s1 \Rightarrow 's2 \Rightarrow 'a\ option$
assumes *disjoint-witness-correct*:
$[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \Longrightarrow \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}$
$[\![invar1\ s1;\ invar2\ s2;\ disjoint\text{-}witness\ s1\ s2 = Some\ a]\!]$
$\Longrightarrow a \in \alpha 1\ s1 \cap \alpha 2\ s2$
begin
lemma *disjoint-witness-None*: $[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \longleftrightarrow \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}$
⟨*proof*⟩

lemma *disjoint-witnessI*: $[\![$
$invar1\ s1;$
$invar2\ s2;$
$\alpha 1\ s1 \cap \alpha 2\ s2 \neq \{\};$
$!!a.\ [\![disjoint\text{-}witness\ s1\ s2 = Some\ a]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

end

Selection of Element

locale *set-sel* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *sel* :: $'s \Rightarrow ('x \Rightarrow 'r\ option) \Rightarrow 'r\ option$
assumes *selE*:
$[\![\ invar\ s;\ x \in \alpha\ s;\ f\ x = Some\ r;$
$!!x\ r.\ [\![sel\ s\ f = Some\ r;\ x \in \alpha\ s;\ f\ x = Some\ r\ ]\!] \Longrightarrow Q$
$]\!] \Longrightarrow Q$
assumes *selI*: $[\![invar\ s;\ \forall x \in \alpha\ s.\ f\ x = None\ ]\!] \Longrightarrow sel\ s\ f = None$
begin

lemma *sel-someD*:

⟦ *invar s*; *sel s f* = *Some r*; !!*x*. ⟦*x*∈*α s*; *f x* = *Some r*⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *sel-noneD*:
⟦ *invar s*; *sel s f* = *None*; *x*∈*α s* ⟧ ⟹ *f x* = *None*
⟨*proof*⟩
end

— Selection of element (without mapping)
locale *set-sel′* = *set* +
constrains *α* :: *′s* ⇒ *′x set*
fixes *sel′* :: *′s* ⇒ (*′x* ⇒ *bool*) ⇒ *′x option*
assumes *sel′E*:
⟦ *invar s*; *x*∈*α s*; *P x*;
!!*x*. ⟦*sel′ s P* = *Some x*; *x*∈*α s*; *P x* ⟧ ⟹ *Q*
⟧ ⟹ *Q*
assumes *sel′I*: ⟦*invar s*; ∀ *x*∈*α s*. ¬ *P x* ⟧ ⟹ *sel′ s P* = *None*
begin

lemma *sel′-someD*:
⟦ *invar s*; *sel′ s P* = *Some x* ⟧ ⟹ *x*∈*α s* ∧ *P x*
⟨*proof*⟩

lemma *sel′-noneD*:
⟦ *invar s*; *sel′ s P* = *None*; *x*∈*α s* ⟧ ⟹ ¬*P x*
⟨*proof*⟩
end

Conversion of Set to List

locale *set-to-list* = *set* +
constrains *α* :: *′s* ⇒ *′x set*
fixes *to-list* :: *′s* ⇒ *′x list*
assumes *to-list-correct*:
invar s ⟹ *set* (*to-list s*) = *α s*
invar s ⟹ *distinct* (*to-list s*)

Conversion of List to Set

locale *list-to-set* = *set* +
constrains *α* :: *′s* ⇒ *′x set*
fixes *to-set* :: *′x list* ⇒ *′s*
assumes *to-set-correct*:
α (*to-set l*) = *set l*
invar (*to-set l*)

2.2.4 Ordered Sets

locale *ordered-set* = *set α invar*
for *α* :: *′s* ⇒ (*′u*::*linorder*) *set* **and** *invar*

locale *ordered-finite-set* = *finite-set* α *invar* + *ordered-set* α *invar*
for $\alpha :: 's \Rightarrow ('u::linorder)$ *set* **and** *invar*

Ordered Iteration

locale *set-iterateoi* = *ordered-finite-set* α *invar*
for $\alpha :: 's \Rightarrow ('u::linorder)$ *set* **and** *invar*
+
fixes *iterateoi* :: $'s \Rightarrow ('u,'\sigma)$ *set-iterator*
assumes *iterateoi-rule*: *invar* $m \Longrightarrow$ *set-iterator-linord* (*iterateoi* m) (α m)
begin
lemma *iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
assumes *MINV*: *invar* m
assumes *I0*: I (α m) $\sigma 0$
assumes *IP*: $!!k\ it\ \sigma.\ [\![$
$c\ \sigma$;
$k \in it$;
$\forall j{\in}it.\ k{\leq}j$;
$\forall j{\in}\alpha\ m - it.\ j{\leq}k$;
$it \subseteq \alpha\ m$;
$I\ it\ \sigma$
$]\!] \Longrightarrow I\ (it - \{k\})\ (f\ k\ \sigma)$
assumes *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
assumes *II*: $!!\sigma\ it.\ [\![$
$it \subseteq \alpha\ m$;
$it \neq \{\}$;
$\neg\ c\ \sigma$;
$I\ it\ \sigma$;
$\forall k{\in}it.\ \forall j{\in}\alpha\ m - it.\ j{\leq}k$
$]\!] \Longrightarrow P\ \sigma$
shows P (*iterateoi* m c f $\sigma 0$)
⟨*proof*⟩

lemma *iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar* m
assumes *I0*: I ((α m)) $\sigma 0$
assumes *IP*: $!!k\ it\ \sigma.\ [\![\ k \in it;\ \forall j{\in}it.\ k{\leq}j;\ \forall j{\in}(\alpha\ m) - it.\ j{\leq}k;\ it \subseteq (\alpha\ m);$
$I\ it\ \sigma\ ]\!]$
$\Longrightarrow I\ (it - \{k\})\ (f\ k\ \sigma)$
assumes *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
shows P (*iterateoi* m (λ-. *True*) f $\sigma 0$)
⟨*proof*⟩
end

lemma *set-iterateoi-I* :
assumes $\bigwedge s.$ *invar* $s \Longrightarrow$ *set-iterator-linord* (*itoi* s) (α s)
shows *set-iterateoi* α *invar itoi*
⟨*proof*⟩

locale *set-reverse-iterateoi* = *ordered-finite-set* α *invar*
for $\alpha :: 's \Rightarrow ('u::linorder)$ *set* **and** *invar*
+
fixes *reverse-iterateoi* :: $'s \Rightarrow ('u,'\sigma)$ *set-iterator*
assumes *reverse-iterateoi-rule*:
invar m $\Longrightarrow$ *set-iterator-rev-linord* (*reverse-iterateoi m*) (α *m*)
begin
lemma *reverse-iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
assumes *MINV*: *invar m*
assumes *I0*: *I* ((α *m*)) $\sigma 0$
assumes *IP*: !!*k it* σ. ⟦
c σ;
k $\in$ *it*;
$\forall j \in it.\ k \geq j$;
$\forall j \in (\alpha\ m) - it.\ j \geq k$;
it $\subseteq$ (α *m*);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
assumes *II*: !!σ *it*. ⟦
it $\subseteq$ (α *m*);
it $\neq$ {};
¬ *c* σ;
I it σ;
$\forall k \in it.\ \forall j \in (\alpha\ m) - it.\ j \geq k$
⟧ $\Longrightarrow$ *P* σ
shows *P* (*reverse-iterateoi m c f* $\sigma 0$)
⟨*proof*⟩

lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar m*
assumes *I0*: *I* ((α *m*)) $\sigma 0$
assumes *IP*: !!*k it* σ. ⟦
k $\in$ *it*;
$\forall j \in it.\ k \geq j$;
$\forall j \in (\alpha\ m) - it.\ j \geq k$;
it $\subseteq$ (α *m*);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
shows *P* (*reverse-iterateoi m* (λ-. *True*) *f* $\sigma 0$)
⟨*proof*⟩
end

lemma *set-reverse-iterateoi-I* :
assumes $\bigwedge s$. *invar s* $\Longrightarrow$ *set-iterator-rev-linord* (*itoi s*) (α *s*)
shows *set-reverse-iterateoi* α *invar itoi*
⟨*proof*⟩

Minimal and Maximal Element

locale *set-min* = *ordered-set* +
  constrains $\alpha :: 's \Rightarrow 'u$::*linorder set*
  fixes *min* :: $'s \Rightarrow ('u \Rightarrow bool) \Rightarrow 'u$ *option*
  assumes *min-correct*:
    $[\![invar\ s;\ x{\in}\alpha\ s;\ P\ x\ ]\!] \Longrightarrow min\ s\ P \in Some\ `\ \{x{\in}\alpha\ s.\ P\ x\}$
    $[\![invar\ s;\ x{\in}\alpha\ s;\ P\ x\ ]\!] \Longrightarrow (the\ (min\ s\ P)) \leq x$
    $[\![invar\ s;\ \{x{\in}\alpha\ s.\ P\ x\} = \{\}\ ]\!] \Longrightarrow min\ s\ P = None$
begin
 lemma *minE*:
   assumes *A*: *invar s* $x{\in}\alpha\ s$ $P\ x$
   obtains x' **where**
   $min\ s\ P = Some\ x'$ $x'{\in}\alpha\ s$ $P\ x'$ $\forall x{\in}\alpha\ s.\ P\ x \longrightarrow x' \leq x$
 ⟨*proof*⟩

 lemmas *minI* = *min-correct*(*3*)

 lemma *min-Some*:
   $[\![invar\ s;\ min\ s\ P = Some\ x\ ]\!] \Longrightarrow x{\in}\alpha\ s$
   $[\![invar\ s;\ min\ s\ P = Some\ x\ ]\!] \Longrightarrow P\ x$
   $[\![invar\ s;\ min\ s\ P = Some\ x;\ x'{\in}\alpha\ s;\ P\ x']\!] \Longrightarrow x{\leq}x'$
   ⟨*proof*⟩

 lemma *min-None*:
   $[\![invar\ s;\ min\ s\ P = None\ ]\!] \Longrightarrow \{x{\in}\alpha\ s.\ P\ x\} = \{\}$
   ⟨*proof*⟩

end

locale *set-max* = *ordered-set* +
  constrains $\alpha :: 's \Rightarrow 'u$::*linorder set*
  fixes *max* :: $'s \Rightarrow ('u \Rightarrow bool) \Rightarrow 'u$ *option*
  assumes *max-correct*:
    $[\![invar\ s;\ x{\in}\alpha\ s;\ P\ x\ ]\!] \Longrightarrow max\ s\ P \in Some\ `\ \{x{\in}\alpha\ s.\ P\ x\}$
    $[\![invar\ s;\ x{\in}\alpha\ s;\ P\ x\ ]\!] \Longrightarrow the\ (max\ s\ P) \geq x$
    $[\![invar\ s;\ \{x{\in}\alpha\ s.\ P\ x\} = \{\}\ ]\!] \Longrightarrow max\ s\ P = None$
begin
 lemma *maxE*:
   assumes *A*: *invar s* $x{\in}\alpha\ s$ $P\ x$
   obtains x' **where**
   $max\ s\ P = Some\ x'$ $x'{\in}\alpha\ s$ $P\ x'$ $\forall x{\in}\alpha\ s.\ P\ x \longrightarrow x' \geq x$
 ⟨*proof*⟩

 lemmas *maxI* = *max-correct*(*3*)

 lemma *max-Some*:
   $[\![invar\ s;\ max\ s\ P = Some\ x\ ]\!] \Longrightarrow x{\in}\alpha\ s$
   $[\![invar\ s;\ max\ s\ P = Some\ x\ ]\!] \Longrightarrow P\ x$
   $[\![invar\ s;\ max\ s\ P = Some\ x;\ x'{\in}\alpha\ s;\ P\ x']\!] \Longrightarrow x{\geq}x'$

⟨*proof*⟩

lemma *max-None*:
⟦ *invar s*; *max s P* = *None* ⟧ ⟹ {*x*∈α *s*. *P x*} = {}
⟨*proof*⟩

end

2.2.5 Conversion to List

locale *set-to-sorted-list* = *ordered-set* α *invar* + *set-to-list* α *invar to-list*
for α :: *'s* ⇒ *'u*::*linorder set* **and** *invar to-list* +
assumes *to-list-sorted*: *invar m* ⟹ *sorted* (*to-list m*)

2.2.6 Record Based Interface

record (*'x*,*'s*) *set-ops* =
set-op-α :: *'s* ⇒ *'x set*
set-op-invar :: *'s* ⇒ *bool*
set-op-empty :: *unit* ⇒ *'s*
set-op-sng :: *'x* ⇒ *'s*
set-op-memb :: *'x* ⇒ *'s* ⇒ *bool*
set-op-ins :: *'x* ⇒ *'s* ⇒ *'s*
set-op-ins-dj :: *'x* ⇒ *'s* ⇒ *'s*
set-op-delete :: *'x* ⇒ *'s* ⇒ *'s*
set-op-isEmpty :: *'s* ⇒ *bool*
set-op-isSng :: *'s* ⇒ *bool*
set-op-ball :: *'s* ⇒ (*'x* ⇒ *bool*) ⇒ *bool*
set-op-bexists :: *'s* ⇒ (*'x* ⇒ *bool*) ⇒ *bool*
set-op-size :: *'s* ⇒ *nat*
set-op-size-abort :: *nat* ⇒ *'s* ⇒ *nat*
set-op-union :: *'s* ⇒ *'s* ⇒ *'s*
set-op-union-dj :: *'s* ⇒ *'s* ⇒ *'s*
set-op-diff :: *'s* ⇒ *'s* ⇒ *'s*
set-op-filter :: (*'x* ⇒ *bool*) ⇒ *'s* ⇒ *'s*
set-op-inter :: *'s* ⇒ *'s* ⇒ *'s*
set-op-subset :: *'s* ⇒ *'s* ⇒ *bool*
set-op-equal :: *'s* ⇒ *'s* ⇒ *bool*
set-op-disjoint :: *'s* ⇒ *'s* ⇒ *bool*
set-op-disjoint-witness :: *'s* ⇒ *'s* ⇒ *'x option*
set-op-sel :: *'s* ⇒ (*'x* ⇒ *bool*) ⇒ *'x option* — Version without mapping
set-op-to-list :: *'s* ⇒ *'x list*
set-op-from-list :: *'x list* ⇒ *'s*

locale *StdSetDefs* =
fixes *ops* :: (*'x*,*'s*,*'more*) *set-ops-scheme*
begin
abbreviation α **where** α == *set-op-α ops*
abbreviation *invar* **where** *invar* == *set-op-invar ops*

abbreviation *empty* **where** *empty == set-op-empty ops*
abbreviation *sng* **where** *sng == set-op-sng ops*
abbreviation *memb* **where** *memb == set-op-memb ops*
abbreviation *ins* **where** *ins == set-op-ins ops*
abbreviation *ins-dj* **where** *ins-dj == set-op-ins-dj ops*
abbreviation *delete* **where** *delete == set-op-delete ops*
abbreviation *isEmpty* **where** *isEmpty == set-op-isEmpty ops*
abbreviation *isSng* **where** *isSng == set-op-isSng ops*
abbreviation *ball* **where** *ball == set-op-ball ops*
abbreviation *bexists* **where** *bexists == set-op-bexists ops*
abbreviation *size* **where** *size == set-op-size ops*
abbreviation *size-abort* **where** *size-abort == set-op-size-abort ops*
abbreviation *union* **where** *union == set-op-union ops*
abbreviation *union-dj* **where** *union-dj == set-op-union-dj ops*
abbreviation *diff* **where** *diff == set-op-diff ops*
abbreviation *filter* **where** *filter == set-op-filter ops*
abbreviation *inter* **where** *inter == set-op-inter ops*
abbreviation *subset* **where** *subset == set-op-subset ops*
abbreviation *equal* **where** *equal == set-op-equal ops*
abbreviation *disjoint* **where** *disjoint == set-op-disjoint ops*
abbreviation *disjoint-witness* **where** *disjoint-witness == set-op-disjoint-witness ops*
abbreviation *sel* **where** *sel == set-op-sel ops*
abbreviation *to-list* **where** *to-list == set-op-to-list ops*
abbreviation *from-list* **where** *from-list == set-op-from-list ops*
end

locale *StdSet = StdSetDefs ops +*
set α invar +
set-empty α invar empty +
set-sng α invar sng +
set-memb α invar memb +
set-ins α invar ins +
set-ins-dj α invar ins-dj +
set-delete α invar delete +
set-isEmpty α invar isEmpty +
set-isSng α invar isSng +
set-ball α invar ball +
set-bexists α invar bexists +
set-size α invar size +
set-size-abort α invar size-abort +
set-union α invar α invar α invar union +
set-union-dj α invar α invar α invar union-dj +
set-diff α invar α invar diff +
set-filter α invar α invar filter +
set-inter α invar α invar α invar inter +
set-subset α invar α invar subset +
set-equal α invar α invar equal +

set-disjoint α *invar* α *invar disjoint* +
set-disjoint-witness α *invar* α *invar disjoint-witness* +
set-sel' α *invar sel* +
set-to-list α *invar to-list* +
list-to-set α *invar from-list*
for *ops*
begin

lemmas *correct* =
empty-correct
sng-correct
memb-correct
ins-correct
ins-dj-correct
delete-correct
isEmpty-correct
isSng-correct
ball-correct
bexists-correct
size-correct
size-abort-correct
union-correct
union-dj-correct
diff-correct
filter-correct
inter-correct
subset-correct
equal-correct
disjoint-correct
disjoint-witness-correct
to-list-correct
to-set-correct

end

record ('*x*,'*s*) *oset-ops* = ('*x*::*linorder*,'*s*) *set-ops* +
set-op-min :: '*s* ⇒ ('*x* ⇒ *bool*) ⇒ '*x option*
set-op-max :: '*s* ⇒ ('*x* ⇒ *bool*) ⇒ '*x option*

locale *StdOSetDefs* = *StdSetDefs ops*
for *ops* :: ('*x*::*linorder*,'*s*,'*more*) *oset-ops-scheme*
begin
abbreviation *min* **where** *min* == *set-op-min ops*
abbreviation *max* **where** *max* == *set-op-max ops*
end

locale *StdOSet* =

StdOSetDefs ops +
StdSet ops +
set-min α *invar min* +
set-max α *invar max*
for *ops*
begin
end

no-notation *insert* (*set'-ins*)

end

General Algorithms for Iterators over Finite Sets **theory** *SetIteratorGA*
imports *Main SetIterator SetIteratorOperations*
begin

2.2.7 Quantification

definition *iterate-ball* **where**
iterate-ball (*it*::('*x*,*bool*) *set-iterator*) *P* = *it id* ($\lambda x\ \sigma.\ P\ x$) *True*

lemma *iterate-ball-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-ball it P* = ($\forall\, x{\in}S0.\ P\ x$)
⟨*proof*⟩

definition *iterate-bex* **where**
iterate-bex (*it*::('*x*,*bool*) *set-iterator*) *P* = *it* ($\lambda\sigma.\ \neg\sigma$) ($\lambda x\ \sigma.\ P\ x$) *False*

lemma *iterate-bex-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-bex it P* = ($\exists\, x{\in}S0.\ P\ x$)
⟨*proof*⟩

2.2.8 Iterator to List

definition *iterate-to-list* **where**
iterate-to-list (*it*::('*x*,'*x list*) *set-iterator*) = *it* (λ-. *True*) ($\lambda x\ \sigma.\ x\ \#\ \sigma$) []

lemma *iterate-to-list-foldli* [*simp*] :
iterate-to-list (*foldli xs*) = *rev xs*
⟨*proof*⟩

lemma *iterate-to-list-genord-correct* :
assumes *it*: *set-iterator-genord it S0 R*
shows *set* (*iterate-to-list it*) = *S0* ∧ *distinct* (*iterate-to-list it*) ∧
sorted-by-rel R (*rev* (*iterate-to-list it*))
⟨*proof*⟩

lemma *iterate-to-list-correct* :
assumes *it*: *set-iterator it S0*
shows *set* (*iterate-to-list it*) = *S0* ∧ *distinct* (*iterate-to-list it*)
⟨*proof*⟩

lemma *iterate-to-list-linord-correct* :
fixes *S0* :: ('*a*::{*linorder*}) *set*
assumes *it-OK*: *set-iterator-linord it S0*
shows *set* (*iterate-to-list it*) = *S0* ∧ *distinct* (*iterate-to-list it*) ∧
sorted (*rev* (*iterate-to-list it*))
⟨*proof*⟩

lemma *iterate-to-list-rev-linord-correct* :
fixes *S0* :: ('*a*::{*linorder*}) *set*
assumes *it-OK*: *set-iterator-rev-linord it S0*
shows *set* (*iterate-to-list it*) = *S0* ∧ *distinct* (*iterate-to-list it*) ∧
sorted (*iterate-to-list it*)
⟨*proof*⟩

lemma *iterate-to-list-map-linord-correct* :
assumes *it-OK*: *map-iterator-linord it m*
shows *map-of* (*iterate-to-list it*) = *m* ∧ *distinct* (*map fst* (*iterate-to-list it*)) ∧
sorted (*map fst* (*rev* (*iterate-to-list it*)))
⟨*proof*⟩

lemma *iterate-to-list-map-rev-linord-correct* :
assumes *it-OK*: *map-iterator-rev-linord it m*
shows *map-of* (*iterate-to-list it*) = *m* ∧ *distinct* (*map fst* (*iterate-to-list it*)) ∧
sorted (*map fst* (*iterate-to-list it*))
⟨*proof*⟩

2.2.9 Size

lemma *set-iterator-finite* :
assumes *it*: *set-iterator it S0*
shows *finite S0*
⟨*proof*⟩

lemma *map-iterator-finite* :
assumes *it*: *map-iterator it m*
shows *finite* (*dom m*)
⟨*proof*⟩

definition *iterate-size* **where**
iterate-size (*it*::('*x*,*nat*) *set-iterator*) = *it* (λ-. *True*) (λ*x* σ. *Suc* σ) *0*

lemma *iterate-size-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-size it* = *card S0* ∧ *finite S0*

⟨*proof*⟩

definition *iterate-size-abort* **where**
iterate-size-abort (*it*::('*x*,*nat*) *set-iterator*) *n* = *it* (λσ. σ < *n*) (λ*x* σ. *Suc* σ) *0*

lemma *iterate-size-abort-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-size-abort it n* = (*min n* (*card S0*)) ∧ *finite S0*
⟨*proof*⟩

2.2.10 Emptyness Check

definition *iterate-is-empty-by-size* **where**
iterate-is-empty-by-size it = (*iterate-size-abort it 1* = *0*)

lemma *iterate-is-empty-by-size-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-is-empty-by-size it* = (*S0* = {})
⟨*proof*⟩

definition *iterate-is-empty* **where**
iterate-is-empty (*it*::('*x*,*bool*) *set-iterator*) = (*it* (λ*b*. *b*) (λ- -. *False*) *True*)

lemma *iterate-is-empty-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-is-empty it* = (*S0* = {})
⟨*proof*⟩

2.2.11 Check for singleton Sets

definition *iterate-is-sng* **where**
iterate-is-sng it = (*iterate-size-abort it 2* = *1*)

lemma *iterate-is-sng-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-is-sng it* = (*card S0* = *1*)
⟨*proof*⟩

2.2.12 Selection

definition *iterate-sel* **where**
iterate-sel (*it*::('*x*,'*y option*) *set-iterator*) *f* = *it* (λσ. σ = *None*) (λ*x* σ. *f x*) *None*

lemma *iterate-sel-genord-correct* :
assumes *it-OK*: *set-iterator-genord it S0 R*
shows *iterate-sel it f* = *None* ⟷ (∀ *x*∈*S0*. (*f x* = *None*))
iterate-sel it f = *Some y* ⟹ (∃ *x* ∈ *S0*. *f x* = *Some y* ∧ (∀ *x'* ∈ *S0*−{*x*}. ∀ *y*. *f x'* = *Some y'* ⟶ *R x x'*))
⟨*proof*⟩

definition *iterate-sel-no-map* **where**
iterate-sel-no-map it P = iterate-sel it (λx. *if P x then Some x else None*)
lemmas *iterate-sel-no-map-alt-def = iterate-sel-no-map-def*[*unfolded iterate-sel-def*, *code*]

lemma *iterate-sel-no-map-genord-correct* :
assumes *it-OK*: *set-iterator-genord it S0 R*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall x{\in}S0.\ \neg(P\ x)$)
iterate-sel-no-map it P = Some x $\Longrightarrow$ ($x \in S0 \wedge P\ x \wedge (\forall x' \in S0{-}\{x\}.\ P\ x' \longrightarrow R\ x\ x')$)
⟨*proof*⟩

lemma *iterate-sel-no-map-correct* :
assumes *it-OK*: *set-iterator it S0*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall x{\in}S0.\ \neg(P\ x)$)
iterate-sel-no-map it P = Some x $\Longrightarrow$ $x \in S0 \wedge P\ x$
⟨*proof*⟩

lemma *iterate-sel-no-map-linord-correct* :
assumes *it-OK*: *set-iterator-linord it S0*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall x{\in}S0.\ \neg(P\ x)$)
iterate-sel-no-map it P = Some x $\Longrightarrow$ ($x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x \leq x')$)
⟨*proof*⟩

lemma *iterate-sel-no-map-rev-linord-correct* :
assumes *it-OK*: *set-iterator-rev-linord it S0*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall x{\in}S0.\ \neg(P\ x)$)
iterate-sel-no-map it P = Some x $\Longrightarrow$ ($x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x' \leq x)$)
⟨*proof*⟩

lemma *iterate-sel-no-map-map-correct* :
assumes *it-OK*: *map-iterator it m*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall k\ v.\ m\ k = Some\ v \longrightarrow \neg(P\ (k, v))$)
iterate-sel-no-map it P = Some $(k, v) \Longrightarrow (m\ k = Some\ v \wedge P\ (k, v))$
⟨*proof*⟩

lemma *iterate-sel-no-map-map-linord-correct* :
assumes *it-OK*: *map-iterator-linord it m*
shows *iterate-sel-no-map it P = None* $\longleftrightarrow$ ($\forall k\ v.\ m\ k = Some\ v \longrightarrow \neg(P\ (k, v))$)
iterate-sel-no-map it P = Some $(k, v) \Longrightarrow (m\ k = Some\ v \wedge P\ (k, v) \wedge (\forall k'\ v'.\ m\ k' = Some\ v' \wedge$
$P\ (k', v') \longrightarrow k \leq k')$)

⟨*proof*⟩

lemma *iterate-sel-no-map-map-rev-linord-correct* :
assumes *it-OK*: *map-iterator-rev-linord it m*
shows *iterate-sel-no-map it P = None* ⟷ (∀ *k v*. *m k = Some v* ⟶ ¬(*P* (*k*, *v*)))
iterate-sel-no-map it P = Some (*k*, *v*) ⟹ (*m k = Some v* ∧ *P* (*k*, *v*) ∧ (∀ *k′ v′* . *m k′ = Some v′* ∧
P (*k′*, *v′*) ⟶ *k′* ≤ *k*))
⟨*proof*⟩

2.2.13 Creating ordered iterators

One can transform an iterator into an ordered one by converting it to list, sorting this list and then converting back to an iterator. In general, this brute-force method is inefficient, though.

definition *iterator-to-ordered-iterator* **where**
iterator-to-ordered-iterator sort-fun it =
foldli (*sort-fun* (*iterate-to-list it*))

lemma *iterator-to-ordered-iterator-correct* :
assumes *sort-fun-OK*: ⋀*l*. *sorted-by-rel R* (*sort-fun l*) ∧ *multiset-of* (*sort-fun l*) = *multiset-of l*
and *it-OK*: *set-iterator it S0*
shows *set-iterator-genord* (*iterator-to-ordered-iterator sort-fun it*) *S0 R*
⟨*proof*⟩

definition *iterator-to-ordered-iterator-quicksort* **where**
iterator-to-ordered-iterator-quicksort R it =
iterator-to-ordered-iterator (*quicksort-by-rel R* []) *it*

lemmas *iterator-to-ordered-iterator-quicksort-code*[*code*] =
iterator-to-ordered-iterator-quicksort-def[*unfolded iterator-to-ordered-iterator-def*]

lemma *iterator-to-ordered-iterator-quicksort-correct* :
assumes *lin* : ⋀*x y*. (*R x y*) ∨ (*R y x*)
and *trans-R*: ⋀*x y z*. *R x y* ⟹ *R y z* ⟹ *R x z*
and *it-OK*: *set-iterator it S0*
shows *set-iterator-genord* (*iterator-to-ordered-iterator-quicksort R it*) *S0 R*
⟨*proof*⟩

definition *iterator-to-ordered-iterator-mergesort* **where**
iterator-to-ordered-iterator-mergesort R it =
iterator-to-ordered-iterator (*mergesort-by-rel R*) *it*

lemmas *iterator-to-ordered-iterator-mergesort-code*[*code*] =
iterator-to-ordered-iterator-mergesort-def[*unfolded iterator-to-ordered-iterator-def*]

lemma *iterator-to-ordered-iterator-mergesort-correct* :
assumes *lin* : $\bigwedge x\ y.\ (R\ x\ y) \vee (R\ y\ x)$
and *trans-R*: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
and *it-OK*: *set-iterator it S0*
shows *set-iterator-genord* (*iterator-to-ordered-iterator-mergesort R it*) *S0 R*
⟨*proof*⟩

end

2.3 Specification of Sequences

theory *ListSpec*
imports *Main*
../common/Misc
../iterator/SetIteratorOperations
../iterator/SetIteratorGA
begin

2.3.1 Definition

locale *list* =
— Abstraction to HOL-lists
fixes α :: $'s \Rightarrow 'x$ *list*
— Invariant
fixes *invar* :: $'s \Rightarrow$ *bool*

2.3.2 Functions

locale *list-empty* = *list* +
constrains α :: $'s \Rightarrow 'x$ *list*
fixes *empty* :: *unit* $\Rightarrow 's$
assumes *empty-correct*:
α (*empty* ()) = []
invar (*empty* ())

locale *list-isEmpty* = *list* +
constrains α :: $'s \Rightarrow 'x$ *list*
fixes *isEmpty* :: $'s \Rightarrow$ *bool*
assumes *isEmpty-correct*:
invar s $\Longrightarrow$ *isEmpty s* $\longleftrightarrow$ α *s* = []

locale *list-iteratei* = *list* +
constrains α :: $'s \Rightarrow 'x$ *list*
fixes *iteratei* :: $'s \Rightarrow ('x,'\sigma)$ *set-iterator*
assumes *iteratei-correct*:

invar s ⟹ iteratei s = foldli (α s)
begin
lemma *iteratei-no-sel-rule*:
invar s ⟹ distinct (α s) ⟹ set-iterator (iteratei s) (set (α s))
⟨*proof*⟩
end

lemma *list-iteratei-iteratei-linord-rule*:
list-iteratei α invar iteratei ⟹ invar s ⟹ distinct (α s) ⟹ sorted (α s) ⟹
set-iterator-linord (iteratei s) (set (α s))
⟨*proof*⟩

lemma *list-iteratei-iteratei-rev-linord-rule*:
list-iteratei α invar iteratei ⟹ invar s ⟹ distinct (α s) ⟹ sorted (rev (α s))
⟹
set-iterator-rev-linord (iteratei s) (set (α s))
⟨*proof*⟩

locale *list-reverse-iteratei = list +*
constrains *α :: 's ⇒ 'x list*
fixes *reverse-iteratei :: 's ⇒ ('x,'σ) set-iterator*
assumes *reverse-iteratei-correct*:
invar s ⟹ reverse-iteratei s = foldri (α s)
begin
lemma *reverse-iteratei-no-sel-rule*:
invar s ⟹ distinct (α s) ⟹ set-iterator (reverse-iteratei s) (set (α s))
⟨*proof*⟩
end

lemma *list-reverse-iteratei-iteratei-linord-rule*:
list-reverse-iteratei α invar iteratei ⟹ invar s ⟹ distinct (α s) ⟹ sorted (rev (α s)) ⟹
set-iterator-linord (iteratei s) (set (α s))
⟨*proof*⟩

lemma *list-reverse-iteratei-iteratei-rev-linord-rule*:
list-reverse-iteratei α invar iteratei ⟹ invar s ⟹ distinct (α s) ⟹ sorted (α s) ⟹
set-iterator-rev-linord (iteratei s) (set (α s))
⟨*proof*⟩

locale *list-size = list +*
constrains *α :: 's ⇒ 'x list*
fixes *size :: 's ⇒ nat*
assumes *size-correct*:
invar s ⟹ size s = length (α s)

Stack

locale *list-push* = *list* +
  constrains $\alpha :: 's \Rightarrow 'x\ list$
  fixes *push* :: $'x \Rightarrow 's \Rightarrow 's$
  assumes *push-correct*:
    $invar\ s \Longrightarrow \alpha\ (push\ x\ s) = x\#\alpha\ s$
    $invar\ s \Longrightarrow invar\ (push\ x\ s)$

locale *list-pop* = *list* +
  constrains $\alpha :: 's \Rightarrow 'x\ list$
  fixes *pop* :: $'s \Rightarrow ('x \times 's)$
  assumes *pop-correct*:
    $[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow fst\ (pop\ s) = hd\ (\alpha\ s)$
    $[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow \alpha\ (snd\ (pop\ s)) = tl\ (\alpha\ s)$
    $[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow invar\ (snd\ (pop\ s))$
begin
  lemma *popE*:
    assumes *I*: $invar\ s \quad \alpha\ s \neq []$
    obtains s' **where** $pop\ s = (hd\ (\alpha\ s),\ s') \quad invar\ s' \quad \alpha\ s' = tl\ (\alpha\ s)$
  ⟨*proof*⟩

The following shortcut notations are not meant for generating efficient code, but solely to simplify reasoning

  definition $head\ s == fst\ (pop\ s)$
  definition $tail\ s == snd\ (pop\ s)$

  lemma *tail-correct*: $[\![invar\ F;\ \alpha\ F \neq []]\!] \Longrightarrow \alpha\ (tail\ F) = tl\ (\alpha\ F)$
    ⟨*proof*⟩

  lemma *head-correct*: $[\![invar\ F;\ \alpha\ F \neq []]\!] \Longrightarrow (head\ F) = hd\ (\alpha\ F)$
    ⟨*proof*⟩

  lemma *pop-split*: $pop\ F = (head\ F,\ tail\ F)$
    ⟨*proof*⟩

end

locale *list-top* = *list* +
  constrains $\alpha :: 's \Rightarrow 'x\ list$
  fixes *top* :: $'s \Rightarrow 'x$
  assumes *top-correct*:
    $[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow top\ s = hd\ (\alpha\ s)$

Queue

locale *list-enqueue* = *list* +
  constrains $\alpha :: 's \Rightarrow 'x\ list$
  fixes *enqueue* :: $'x \Rightarrow 's \Rightarrow 's$
  assumes *enqueue-correct*:

```
    invar s ⟹ α (enqueue x s) = α s @ [x]
    invar s ⟹ invar (enqueue x s)

  — Same specification as pop
locale list-dequeue = list-pop
begin
  lemmas dequeue-correct = pop-correct
  lemmas dequeueE = popE
  lemmas dequeue-split=pop-split
end

  — Same specification as top
locale list-next = list-top
begin
  lemmas next-correct = top-correct
end
```

Indexing

```
locale list-get = list +
  constrains α :: 's ⇒ 'x list
  fixes get :: 's ⇒ nat ⇒ 'x
  assumes get-correct:
    ⟦invar s; i<length (α s)⟧ ⟹ get s i = α s ! i

locale list-set = list +
  constrains α :: 's ⇒ 'x list
  fixes set :: 's ⇒ nat ⇒ 'x ⇒ 's
  assumes set-correct:
    ⟦invar s; i<length (α s)⟧ ⟹ α (set s i x) = α s [i := x]
    ⟦invar s; i<length (α s)⟧ ⟹ invar (set s i x)

  — Same specification as enqueue
locale list-add = list-enqueue
begin
  lemmas add-correct = enqueue-correct
end


end
```

2.4 Specification of Annotated Lists

```
theory AnnotatedListSpec
imports Main
begin
```

2.4.1 Introduction

We define lists with annotated elements. The annotations form a monoid.

We provide standard list operations and the split-operation, that splits the list according to its annotations.

locale *al* =
— Annotated lists are abstracted to lists of pairs of elements and annotations.
fixes $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $invar :: 's \Rightarrow bool$

2.4.2 Basic Annotated List Operations

Empty Annotated List

locale *al-empty* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $empty :: unit \Rightarrow 's$
assumes *empty-correct*:
$invar\ (empty\ ())$
$\alpha\ (empty\ ()) = Nil$

Emptiness Check

locale *al-isEmpty* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $isEmpty :: 's \Rightarrow bool$
assumes *isEmpty-correct*:
$invar\ s \Longrightarrow isEmpty\ s \longleftrightarrow \alpha\ s = Nil$

Counting Elements

locale *al-count* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $count :: 's \Rightarrow nat$
assumes *count-correct*:
$invar\ s \Longrightarrow count\ s = length(\alpha\ s)$

Appending an Element from the Left

locale *al-consl* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $consl :: 'e \Rightarrow 'a \Rightarrow 's \Rightarrow 's$
assumes *consl-correct*:
$invar\ s \Longrightarrow invar\ (consl\ e\ a\ s)$
$invar\ s \Longrightarrow (\alpha\ (consl\ e\ a\ s)) = (e,a)\ \#\ (\alpha\ s)$

Appending an Element from the Right

locale *al-consr* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$

fixes *consr* :: $'s \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
assumes *consr-correct*:
$invar\ s \Longrightarrow invar\ (consr\ s\ e\ a)$
$invar\ s \Longrightarrow (\alpha\ (consr\ s\ e\ a)) = (\alpha\ s)\ @\ [(e,a)]$

Take the First Element

locale *al-head* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes *head* :: $'s \Rightarrow ('e \times 'a)$
assumes *head-correct*:
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow head\ s = hd\ (\alpha\ s)$

Drop the First Element

locale *al-tail* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes *tail* :: $'s \Rightarrow 's$
assumes *tail-correct*:
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow \alpha\ (tail\ s) = tl\ (\alpha\ s)$
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow invar\ (tail\ s)$

Take the Last Element

locale *al-headR* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes *headR* :: $'s \Rightarrow ('e \times 'a)$
assumes *headR-correct*:
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow headR\ s = last\ (\alpha\ s)$

Drop the Last Element

locale *al-tailR* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes *tailR* :: $'s \Rightarrow 's$
assumes *tailR-correct*:
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow \alpha\ (tailR\ s) = butlast\ (\alpha\ s)$
$[\![invar\ s;\ \alpha\ s \neq Nil]\!] \Longrightarrow invar\ (tailR\ s)$

Fold a Function over the Elements from the Left

locale *al-foldl* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes *foldl* :: $('z \Rightarrow 'e \times 'a \Rightarrow 'z) \Rightarrow 'z \Rightarrow 's \Rightarrow 'z$
assumes *foldl-correct*:
$invar\ s \Longrightarrow foldl\ f\ \sigma\ s = List.foldl\ f\ \sigma\ (\alpha\ s)$

Fold a Function over the Elements from the Right

locale *al-foldr* = *al* +

constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $foldr :: ('e \times 'a \Rightarrow 'z \Rightarrow 'z) \Rightarrow 's \Rightarrow 'z \Rightarrow 'z$
assumes *foldr-correct*:
$invar\ s \Longrightarrow foldr\ f\ s\ \sigma = List.foldr\ f\ (\alpha\ s)\ \sigma$

Concatenation of Two Annotated Lists

locale *al-app* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $app :: 's \Rightarrow 's \Rightarrow 's$
assumes *app-correct*:
$[\![invar\ s; invar\ s']\!] \Longrightarrow \alpha\ (app\ s\ s') = (\alpha\ s)\ @\ (\alpha\ s')$
$[\![invar\ s; invar\ s']\!] \Longrightarrow invar\ (app\ s\ s')$

Readout the Summed up Annotations

locale *al-annot* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $annot :: 's \Rightarrow 'a$
assumes *annot-correct*:
$invar\ s \Longrightarrow (annot\ s) = (listsum\ (map\ snd\ (\alpha\ s)))$

Split by Monotone Predicate

locale *al-splits* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
fixes $splits :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 's \Rightarrow ('s \times ('e \times 'a) \times 's)$
assumes *splits-correct*:
$[\![invar\ s;$
$\forall a\ b.\ p\ a \longrightarrow p\ (a + b);$
$\neg\ p\ i;$
$p\ (i + listsum\ (map\ snd\ (\alpha\ s)));$
$(splits\ p\ i\ s) = (l, (e,a), r)]\!]$
$\Longrightarrow$
$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)\ \wedge$
$\neg\ p\ (i + listsum\ (map\ snd\ (\alpha\ l)))\ \wedge$
$p\ (i + listsum\ (map\ snd\ (\alpha\ l)) + a)\ \wedge$
$invar\ l\ \wedge$
$invar\ r$

begin
lemma *splitsE*:
assumes
invar: $invar\ s$ **and**
mono: $\forall a\ b.\ p\ a \longrightarrow p\ (a + b)$ **and**
init-ff: $\neg\ p\ i$ **and**
sum-tt: $p\ (i + listsum\ (map\ snd\ (\alpha\ s)))$
obtains $l\ e\ a\ r$ **where**
$(splits\ p\ i\ s) = (l, (e,a), r)$

$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)$
$\neg\ p\ (i + listsum\ (map\ snd\ (\alpha\ l)))$
$p\ (i + listsum\ (map\ snd\ (\alpha\ l)) + a)$
invar l
invar r
⟨*proof*⟩
end

2.4.3 Record Based Interface

record $('e,'a,'s)$ *alist-ops* =
alist-op-α $::'s \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
alist-op-invar $::'s \Rightarrow bool$
alist-op-empty $::unit \Rightarrow 's$
alist-op-isEmpty $::'s \Rightarrow bool$
alist-op-count $::'s \Rightarrow nat$
alist-op-consl $::'e \Rightarrow 'a \Rightarrow 's \Rightarrow 's$
alist-op-consr $::'s \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
alist-op-head $::'s \Rightarrow ('e \times 'a)$
alist-op-tail $::'s \Rightarrow 's$
alist-op-headR $::'s \Rightarrow ('e \times 'a)$
alist-op-tailR $::'s \Rightarrow 's$
alist-op-app $::'s \Rightarrow 's \Rightarrow 's$
alist-op-annot $::'s \Rightarrow 'a$
alist-op-splits $::('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 's \Rightarrow ('s \times ('e \times 'a) \times 's)$

locale *StdALDefs* =
fixes *ops* $::('e,'a::monoid\text{-}add,'s,'more)$ *alist-ops-scheme*
begin
abbreviation *α* **where** $\alpha == alist\text{-}op\text{-}\alpha\ ops$
abbreviation *invar* **where** $invar == alist\text{-}op\text{-}invar\ ops$
abbreviation *empty* **where** $empty == alist\text{-}op\text{-}empty\ ops$
abbreviation *isEmpty* **where** $isEmpty == alist\text{-}op\text{-}isEmpty\ ops$
abbreviation *count* **where** $count == alist\text{-}op\text{-}count\ ops$
abbreviation *consl* **where** $consl == alist\text{-}op\text{-}consl\ ops$
abbreviation *consr* **where** $consr == alist\text{-}op\text{-}consr\ ops$
abbreviation *head* **where** $head == alist\text{-}op\text{-}head\ ops$
abbreviation *tail* **where** $tail == alist\text{-}op\text{-}tail\ ops$
abbreviation *headR* **where** $headR == alist\text{-}op\text{-}headR\ ops$
abbreviation *tailR* **where** $tailR == alist\text{-}op\text{-}tailR\ ops$
abbreviation *app* **where** $app == alist\text{-}op\text{-}app\ ops$
abbreviation *annot* **where** $annot == alist\text{-}op\text{-}annot\ ops$
abbreviation *splits* **where** $splits == alist\text{-}op\text{-}splits\ ops$
end

locale *StdAL* = *StdALDefs ops* +
al α invar +
al-empty α invar empty +
al-isEmpty α invar isEmpty +

al-count α *invar count* +
al-consl α *invar consl* +
al-consr α *invar consr* +
al-head α *invar head* +
al-tail α *invar tail* +
al-headR α *invar headR* +
al-tailR α *invar tailR* +
al-app α *invar app* +
al-annot α *invar annot* +
al-splits α *invar splits*
for *ops*
begin
lemmas *correct* =
empty-correct
isEmpty-correct
count-correct
consl-correct
consr-correct
head-correct
tail-correct
headR-correct
tailR-correct
app-correct
annot-correct
end

end

2.5 Specification of Priority Queues

theory *PrioSpec*
imports *Main* ~~/*src*/*HOL*/*Library*/*Multiset*
begin

We specify priority queues, that are abstracted to multisets of pairs of elements and priorities.

locale *prio* =
fixes α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset* — Abstraction to multiset
fixes *invar* :: $'p \Rightarrow$ *bool* — Invariant

2.5.1 Basic Priority Queue Functions

Empty Queue

locale *prio-empty* = *prio* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *empty* :: *unit* $\Rightarrow$ $'p$
assumes *empty-correct*:

invar (*empty* ())
α (*empty* ()) = {#}

Emptiness Predicate

locale *prio-isEmpty* = *prio* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *isEmpty* :: $'p \Rightarrow$ *bool*
assumes *isEmpty-correct*:
invar p $\Longrightarrow$ (*isEmpty p*) = (α *p* = {#})

Find Minimal Element

locale *prio-find* = *prio* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *find* :: $'p \Rightarrow ('e \times 'a::linorder)$
assumes *find-correct*: $[\![$*invar p*; α *p* $\neq$ {#}$]\!] \Longrightarrow$
(*find p*) $\in\#$ (α *p*) $\wedge$ ($\forall$ *y* $\in$ *set-of* (α *p*). *snd* (*find p*) $\leq$ *snd y*)

Insert

locale *prio-insert* = *prio* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *insert* :: $'e \Rightarrow 'a \Rightarrow 'p \Rightarrow 'p$
assumes *insert-correct*:
invar p $\Longrightarrow$ *invar* (*insert e a p*)
invar p $\Longrightarrow$ α (*insert e a p*) = (α *p*) + {#(*e*,*a*)#}

Meld Two Queues

locale *prio-meld* = *prio* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *meld* :: $'p \Rightarrow 'p \Rightarrow 'p$
assumes *meld-correct*:
$[\![$*invar p*; *invar p'*$]\!] \Longrightarrow$ *invar* (*meld p p'*)
$[\![$*invar p*; *invar p'*$]\!] \Longrightarrow \alpha$ (*meld p p'*) = (α *p*) + (α *p'*)

Delete Minimal Element

Delete the same element that find will return

locale *prio-delete* = *prio-find* +
constrains α :: $'p \Rightarrow ('e \times 'a::linorder)$ *multiset*
fixes *delete* :: $'p \Rightarrow 'p$
assumes *delete-correct*:
$[\![$*invar p*; α *p* $\neq$ {#}$]\!] \Longrightarrow$ *invar* (*delete p*)
$[\![$*invar p*; α *p* $\neq$ {#}$]\!] \Longrightarrow \alpha$ (*delete p*) = (α *p*) $-$ {# (*find p*) #}

2.5.2 Record based interface

record ($'e$, $'a$, $'p$) *prio-ops* =

```
  prio-op-α :: 'p ⇒ ('e × 'a) multiset
  prio-op-invar :: 'p ⇒ bool
  prio-op-empty :: unit ⇒ 'p
  prio-op-isEmpty :: 'p ⇒ bool
  prio-op-insert :: 'e ⇒ 'a ⇒ 'p ⇒ 'p
  prio-op-find :: 'p ⇒ 'e × 'a
  prio-op-delete :: 'p ⇒ 'p
  prio-op-meld :: 'p ⇒ 'p ⇒ 'p

locale StdPrioDefs =
  fixes ops :: ('e,'a::linorder,'p) prio-ops
begin
  abbreviation α where α == prio-op-α ops
  abbreviation invar where invar == prio-op-invar ops
  abbreviation empty where empty == prio-op-empty ops
  abbreviation isEmpty where isEmpty == prio-op-isEmpty ops
  abbreviation insert where insert == prio-op-insert ops
  abbreviation find where find == prio-op-find ops
  abbreviation delete where delete == prio-op-delete ops
  abbreviation meld where meld == prio-op-meld ops
end

locale StdPrio = StdPrioDefs ops +
  prio α invar +
  prio-empty α invar empty +
  prio-isEmpty α invar isEmpty +
  prio-find α invar find +
  prio-insert α invar insert +
  prio-meld α invar meld +
  prio-delete α invar find delete
  for ops
begin
  lemmas correct =
    empty-correct
    isEmpty-correct
    find-correct
    insert-correct
    meld-correct
    delete-correct
end

end
```

2.6 Specification of Unique Priority Queues

```
theory PrioUniqueSpec
imports Main
begin
```

We define unique priority queues, where each element may occur at most once. We provide operations to get and remove the element with the minimum priority, as well as to access and change an elements priority (decrease-key operation).

Unique priority queues are abstracted to maps from elements to priorities.

locale *uprio* =
fixes $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes *invar* :: $'s \Rightarrow bool$

locale *uprio-finite* = *uprio* +
assumes *finite-correct*:
$invar\ s \Longrightarrow finite\ (dom\ (\alpha\ s))$

2.6.1 Basic Upriority Queue Functions

Empty Queue

locale *uprio-empty* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes *empty* :: *unit* $\Rightarrow 's$
assumes *empty-correct*:
$invar\ (empty\ ())$
$\alpha\ (empty\ ()) = Map.empty$

Emptiness Predicate

locale *uprio-isEmpty* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes *isEmpty* :: $'s \Rightarrow bool$
assumes *isEmpty-correct*:
$invar\ s \Longrightarrow (isEmpty\ s) = (\alpha\ s = Map.empty)$

Find and Remove Minimal Element

locale *uprio-pop* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes *pop* :: $'s \Rightarrow ('e \times 'a \times 's)$
assumes *pop-correct*:
$[\![invar\ s;\ \alpha\ s \neq Map.empty;\ pop\ s = (e,a,s')]\!] \Longrightarrow$
$invar\ s' \wedge$
$\alpha\ s' = (\alpha\ s)(e := None) \wedge$
$(\alpha\ s)\ e = Some\ a \wedge$
$(\forall y \in ran\ (\alpha\ s).\ a \leq y)$
begin

lemma *popE*:
assumes
invar s
$\alpha\ s \neq Map.empty$

obtains $e\ a\ s'$ **where**
$pop\ s = (e,\ a,\ s')$
$invar\ s'$
$\alpha\ s' = (\alpha\ s)(e := None)$
$(\alpha\ s)\ e = Some\ a$
$(\forall\, y \in ran\ (\alpha\ s).\ a \leq y)$
$\langle proof \rangle$

end

Insert

If an existing element is inserted, its priority will be overwritten. This can be used to implement a decrease-key operation.

locale *uprio-insert* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes $insert :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
assumes *insert-correct*:
$invar\ s \Longrightarrow invar\ (insert\ s\ e\ a)$
$invar\ s \Longrightarrow \alpha\ (insert\ s\ e\ a) = (\alpha\ s)(e \mapsto a)$

Distinct Insert

This operation only allows insertion of elements that are not yet in the queue.

locale *uprio-distinct-insert* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes $insert :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
assumes *distinct-insert-correct*:
$[\![invar\ s;\ e \notin dom\ (\alpha\ s)]\!] \Longrightarrow invar\ (insert\ s\ e\ a)$
$[\![invar\ s;\ e \notin dom\ (\alpha\ s)]\!] \Longrightarrow \alpha\ (insert\ s\ e\ a) = (\alpha\ s)(e \mapsto a)$

Looking up Priorities

locale *uprio-prio* = *uprio* +
constrains $\alpha :: 's \Rightarrow ('e \rightharpoonup 'a::linorder)$
fixes $prio :: 's \Rightarrow 'e \Rightarrow 'a\ option$
assumes *prio-correct*:
$invar\ s \Longrightarrow prio\ s\ e = (\alpha\ s)\ e$

2.6.2 Record Based Interface

record $('e,\ 'a,\ 's)$ *uprio-ops* =
$upr\text{-}\alpha :: 's \Rightarrow ('e \rightharpoonup 'a)$
upr-invar $:: 's \Rightarrow bool$
upr-empty $:: unit \Rightarrow 's$
upr-isEmpty $:: 's \Rightarrow bool$
upr-insert $:: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$

```
  upr-pop :: 's ⇒ ('e × 'a × 's)
  upr-prio :: 's ⇒ 'e ⇒ 'a option


locale StdUprioDefs =
  fixes ops :: ('e,'a::linorder,'s, 'more) uprio-ops-scheme
begin
  abbreviation α where α == upr-α ops
  abbreviation invar where invar == upr-invar ops
  abbreviation empty where empty == upr-empty ops
  abbreviation isEmpty where isEmpty == upr-isEmpty ops
  abbreviation insert where insert == upr-insert ops
  abbreviation pop where pop == upr-pop ops
  abbreviation prio where prio == upr-prio ops
end

locale StdUprio =  StdUprioDefs ops +
  uprio-finite α invar +
  uprio-empty α invar empty +
  uprio-isEmpty α invar isEmpty +
  uprio-insert α invar insert +
  uprio-pop α invar pop +
  uprio-prio α invar prio
  for ops
begin
  lemmas correct =
    finite-correct
    empty-correct
    isEmpty-correct
    insert-correct
    prio-correct
end

end
```

Chapter 3

Generic algorithms

General Algorithms for Iterators over Finite Sets **theory** *SetIteratorGACollections*
imports
Main
SetIterator
SetIteratorOperations
../spec/SetSpec
../spec/MapSpec
../common/Misc
begin

3.0.3 Iterate add to Set

definition *iterate-add-to-set* **where**
iterate-add-to-set s ins (*it*::('*x*,'*x-set*) *set-iterator*) =
it (λ-. *True*) (λ*x* σ. *ins x* σ) *s*

lemma *iterate-add-to-set-correct* :
assumes *ins-OK*: *set-ins* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
shows α (*iterate-add-to-set s ins it*) = *S0* ∪ α *s* ∧ *invar* (*iterate-add-to-set s ins it*)
⟨*proof*⟩

lemma *iterate-add-to-set-dj-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins-dj*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: *S0* ∩ α *s* = {}
shows α (*iterate-add-to-set s ins-dj it*) = *S0* ∪ α *s* ∧ *invar* (*iterate-add-to-set s ins-dj it*)
⟨*proof*⟩

3.0.4 Iterator to Set

definition *iterate-to-set* **where**
iterate-to-set emp ins-dj (*it*::('*x*,'*x-set*) *set-iterator*) =
iterate-add-to-set (*emp* ()) *ins-dj it*

lemma *iterate-to-set-alt-def*[*code*] :
iterate-to-set emp ins-dj (*it*::('*x*,'*x-set*) *set-iterator*) =
it (λ-. *True*) (λ*x* σ. *ins-dj x* σ) (*emp* ())
⟨*proof*⟩

lemma *iterate-to-set-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins-dj*
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *it*: *set-iterator it S0*
shows α (*iterate-to-set emp ins-dj it*) = *S0* ∧ *invar* (*iterate-to-set emp ins-dj it*)
⟨*proof*⟩

3.0.5 Iterate image/filter add to Set

Iterators only visit element once. Therefore the image operations makes sense for filters only if an injective function is used. However, when adding to a set using non-injective functions is fine.

lemma *iterate-image-filter-add-to-set-correct* :
assumes *ins-OK*: *set-ins* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
shows α (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*)) =
{*b* . ∃ *a*. *a* ∈ *S0* ∧ *f a* = *Some b*} ∪ α *s* ∧
invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
⟨*proof*⟩

lemma *iterate-image-filter-to-set-correct* :
assumes *ins-OK*: *set-ins* α *invar ins*
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *it*: *set-iterator it S0*
shows α (*iterate-to-set emp ins* (*set-iterator-image-filter f it*)) =
{*b* . ∃ *a*. *a* ∈ *S0* ∧ *f a* = *Some b*} ∧
invar (*iterate-to-set emp ins* (*set-iterator-image-filter f it*))
⟨*proof*⟩

For completeness lets also consider injective versions.

lemma *iterate-inj-image-filter-add-to-set-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: {*y*. ∃ *x*. *x* ∈ *S0* ∧ *f x* = *Some y*} ∩ α *s* = {}
assumes *f-inj-on*: *inj-on f* (*S0* ∩ *dom f*)

shows α (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*)) =
$\{b . \exists a. a \in S0 \wedge f\ a = Some\ b\} \cup \alpha\ s \wedge$
invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
⟨*proof*⟩

lemma *iterate-inj-image-filter-to-set-correct* :
assumes *ins-OK*: *set-ins-dj* α *invar ins*
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *it*: *set-iterator it S0*
assumes *f-inj-on*: *inj-on f* (*S0* ∩ *dom f*)
shows α (*iterate-to-set emp ins* (*set-iterator-image-filter f it*)) =
$\{b . \exists a. a \in S0 \wedge f\ a = Some\ b\} \wedge$
invar (*iterate-to-set emp ins* (*set-iterator-image-filter f it*))
⟨*proof*⟩

3.0.6 Iterate diff Set

definition *iterate-diff-set* **where**
iterate-diff-set s del (*it*::('*x*,'*x-set*) *set-iterator*) =
it (λ-. *True*) (λ*x* σ. *del x* σ) *s*

lemma *iterate-diff-correct* :
assumes *del-OK*: *set-delete* α *invar del*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
shows α (*iterate-diff-set s del it*) = $\alpha\ s - S0 \wedge$ *invar* (*iterate-diff-set s del it*)
⟨*proof*⟩

3.0.7 Iterate add to Map

definition *iterate-add-to-map* **where**
iterate-add-to-map m update (*it*::('*k* × '*v*,'*kv-map*) *set-iterator*) =
it (λ-. *True*) (λ(*k*,*v*) σ. *update k v* σ) *m*

lemma *iterate-add-to-map-correct* :
assumes *upd-OK*: *map-update* α *invar upd*
assumes *m-OK*: *invar m*
assumes *it*: *map-iterator it M*
shows α (*iterate-add-to-map m upd it*) = $\alpha\ m +\!\!+ M \wedge$ *invar* (*iterate-add-to-map m upd it*)
⟨*proof*⟩

lemma *iterate-add-to-map-dj-correct* :
assumes *upd-OK*: *map-update-dj* α *invar upd*
assumes *m-OK*: *invar m*
assumes *it*: *map-iterator it M*
assumes *dj*: *dom M* ∩ *dom* (α *m*) = {}
shows α (*iterate-add-to-map m upd it*) = $\alpha\ m +\!\!+ M \wedge$ *invar* (*iterate-add-to-map m upd it*)

⟨*proof*⟩

3.0.8 Iterator to Map

definition *iterate-to-map* **where**
iterate-to-map emp upd-dj (*it*::('*k* × '*v*,'*kv-map*) *set-iterator*) =
iterate-add-to-map (*emp* ()) *upd-dj it*

lemma *iterate-to-map-alt-def*[*code*] :
iterate-to-map emp upd-dj it =
it (λ-. *True*) (λ(*k*, *v*) *σ*. *upd-dj k v σ*) (*emp* ())
⟨*proof*⟩

lemma *iterate-to-map-correct* :
assumes *upd-dj-OK*: *map-update-dj α invar upd-dj*
assumes *emp-OK*: *map-empty α invar emp*
assumes *it*: *map-iterator it M*
shows *α* (*iterate-to-map emp upd-dj it*) = *M* ∧ *invar* (*iterate-to-map emp upd-dj it*)
⟨*proof*⟩

end

3.1 Generic Algorithms for Maps

theory *MapGA*
imports
../spec/MapSpec
../common/Misc
../iterator/SetIteratorGA
../iterator/SetIteratorGACollections
begin

3.1.1 Disjoint Update (by update)

lemma (**in** *map-update*) *update-dj-by-update*:
map-update-dj α invar update
⟨*proof*⟩

3.1.2 Disjoint Add (by add)

lemma (**in** *map-add*) *add-dj-by-add*:
map-add-dj α invar add
⟨*proof*⟩

3.1.3 Add (by iterate)

definition *it-add* **where**
it-add update iti = (λ*m1 m2*. *iterate-add-to-map m1 update* (*iti m2*))

lemma *it-add-code*[*code*]:
it-add update iti m1 m2 = *iti m2* (λ-. *True*) (λ(*x*, *y*). *update x y*) *m1*
⟨*proof*⟩

lemma *it-add-correct*:
fixes α :: '*s* ⇒ '*u* ⇀ '*v*
assumes *iti*: *map-iteratei* α *invar iti*
assumes *upd-OK*: *map-update* α *invar update*
shows *map-add* α *invar* (*it-add update iti*)
⟨*proof*⟩

3.1.4 Disjoint Add (by iterate)

lemma *it-add-dj-correct*:
fixes α :: '*s* ⇒ '*u* ⇀ '*v*
assumes *iti*: *map-iteratei* α *invar iti*
assumes *upd-dj-OK*: *map-update-dj* α *invar update*
shows *map-add-dj* α *invar* (*it-add update iti*)
⟨*proof*⟩

3.1.5 Emptiness check (by iteratei)

definition *iti-isEmpty* **where**
iti-isEmpty iti = (λ*m*. *iterate-is-empty* (*iti m*))

lemma *iti-isEmpty*[*code*] :
iti-isEmpty iti m = *iti m* (λ*c*. *c*) (λ- -. *False*) *True*
⟨*proof*⟩

lemma *iti-isEmpty-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-isEmpty* α *invar* (*iti-isEmpty iti*)
⟨*proof*⟩

3.1.6 Iterators

Iteratei (by iterateoi)

lemma *iti-by-itoi*:
assumes *map-iterateoi* α *invar it*
shows *map-iteratei* α *invar it*
⟨*proof*⟩

Iteratei (by reverse_iterateoi)

lemma *iti-by-ritoi*:

assumes *map-reverse-iterateoi* α *invar it*
shows *map-iteratei* α *invar it*
⟨*proof*⟩

3.1.7 Selection (by iteratei)

definition *iti-sel* **where**
iti-sel iti = ($\lambda m\ f.$ *iterate-sel* (*iti m*) *f*)

lemma *iti-sel-code*[*code*] :
iti-sel iti m f = *iti m* ($\lambda\sigma.\ \sigma$ = *None*) ($\lambda x\ \sigma.\ f\ x$) *None*
⟨*proof*⟩

lemma *iti-sel-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-sel* α *invar* (*iti-sel iti*)
⟨*proof*⟩

3.1.8 Map-free selection by selection

definition *iti-sel-no-map* **where**
iti-sel-no-map iti = ($\lambda m\ P.$ *iterate-sel-no-map* (*iti m*) *P*)

lemma *iti-sel-no-map-code*[*code*] :
iti-sel-no-map iti m P = *iti m* ($\lambda\sigma.\ \sigma$ = *None*) ($\lambda x\ \sigma.$ *if P x then Some x else None*) *None*
⟨*proof*⟩

lemma *iti-sel′-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-sel′* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.1.9 Map-free selection by selection

definition *sel-sel′*
:: ('*s* ⇒ ('*k* × '*v* ⇒ - *option*) ⇒ - *option*) ⇒ '*s* ⇒ ('*k* × '*v* ⇒ *bool*) ⇒ ('*k*×'*v*) *option*
where *sel-sel′ sel s P* = *sel s* ($\lambda kv.$ *if P kv then Some kv else None*)

lemma *sel-sel′-correct*:
assumes *map-sel* α *invar sel*
shows *map-sel′* α *invar* (*sel-sel′ sel*)
⟨*proof*⟩

3.1.10 Bounded Quantification (by sel)

definition *sel-ball*
:: ('*s* ⇒ ('*k* × '*v* ⇒ *unit option*) ⇀ *unit*) ⇒ '*s* ⇒ ('*k* × '*v* ⇒ *bool*) ⇒ *bool*
where *sel-ball sel s P* == *sel s* ($\lambda kv.$ *if* ¬*P kv then Some* () *else None*) = *None*

lemma *sel-ball-correct*:
assumes *map-sel* α *invar sel*
shows *map-ball* α *invar* (*sel-ball sel*)
⟨*proof*⟩

definition *sel-bexists*
:: ($'s \Rightarrow$ ($'k \times 'v \Rightarrow$ *unit option*) $\rightharpoonup$ *unit*) $\Rightarrow$ $'s \Rightarrow$ ($'k \times 'v \Rightarrow$ *bool*) $\Rightarrow$ *bool*
where *sel-bexists sel s P* == *sel s* (λkv. *if P kv then Some* () *else None*) = *Some* ()

lemma *sel-bexists-correct*:
assumes *map-sel* α *invar sel*
shows *map-bexists* α *invar* (*sel-bexists sel*)
⟨*proof*⟩

definition *neg-ball-bexists*
:: ($'s \Rightarrow$ ($'k \times 'v \Rightarrow$ *bool*) $\Rightarrow$ *bool*) $\Rightarrow$ $'s \Rightarrow$ ($'k \times 'v \Rightarrow$ *bool*) $\Rightarrow$ *bool*
where *neg-ball-bexists ball s P* == $\neg$ (*ball s* (λkv. $\neg$(*P kv*)))

lemma *neg-ball-bexists-correct*:
fixes *ball*
assumes *map-ball* α *invar ball*
shows *map-bexists* α *invar* (*neg-ball-bexists ball*)
⟨*proof*⟩

3.1.11 Bounded Quantification (by iterate)

definition *iti-ball* **where**
iti-ball iti = (λm. *iterate-ball* (*iti m*))

lemma *iti-ball-code*[*code*] :
iti-ball iti m P = *iti m* (λc. *c*) ($\lambda x\ \sigma$. *P x*) *True*
⟨*proof*⟩

lemma *iti-ball-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-ball* α *invar* (*iti-ball iti*)
⟨*proof*⟩

definition *iti-bexists* **where**
iti-bexists iti = (λm. *iterate-bex* (*iti m*))

lemma *iti-bexists-code*[*code*] :
iti-bexists iti m P = *iti m* (λc. $\neg c$) ($\lambda x\ \sigma$. *P x*) *False*
⟨*proof*⟩

lemma *iti-bexists-correct*:
assumes *iti*: *map-iteratei* α *invar iti*

shows *map-bexists* α *invar* (*iti-bexists iti*)
⟨*proof*⟩

3.1.12 Size (by iterate)

definition *it-size* **where**
it-size iti = (λm. *iterate-size* (*iti m*))

lemma *it-size-code*[*code*] :
it-size iti m = *iti m* (λ-. *True*) (λ*x n* . *Suc n*) *0*
⟨*proof*⟩

lemma *it-size-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-size* α *invar* (*it-size iti*)
⟨*proof*⟩

3.1.13 Size with abort (by iterate)

definition *iti-size-abort* **where**
iti-size-abort iti = ($\lambda n\ m$. *iterate-size-abort* (*iti m*) *n*)

lemma *iti-size-abort-code*[*code*] :
iti-size-abort iti n m = *iti m* ($\lambda\sigma$. $\sigma < n$) (λ*x*. *Suc*) *0*
⟨*proof*⟩

lemma *iti-size-abort-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-size-abort* α *invar* (*iti-size-abort iti*)
⟨*proof*⟩

3.1.14 Singleton check (by size-abort)

definition *sza-isSng* **where**
sza-isSng iti = (λm. *iterate-is-sng* (*iti m*))

lemma *sza-isSng-code*[*code*] :
sza-isSng iti m = (*iti m* ($\lambda\sigma$. $\sigma < 2$) (λ*x*. *Suc*) *0* = *1*)
⟨*proof*⟩

lemma *sza-isSng-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-isSng* α *invar* (*sza-isSng iti*)
⟨*proof*⟩

3.1.15 Map to List (by iterate)

definition *it-map-to-list* **where**
it-map-to-list iti = (λm. *iterate-to-list* (*iti m*))

lemma *it-map-to-list-code*[*code*] :
it-map-to-list iti m = *iti m* (λ-. *True*) *op* # []
⟨*proof*⟩

lemma *it-map-to-list-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-to-list* α *invar* (*it-map-to-list iti*)
⟨*proof*⟩

3.1.16 List to Map

fun *gen-list-to-map-aux*
:: ('*k* ⇒ '*v* ⇒ '*m* ⇒ '*m*) ⇒ '*m* ⇒ ('*k*×'*v*) *list* ⇒ '*m*
where
gen-list-to-map-aux upd accs [] = *accs* |
gen-list-to-map-aux upd accs ((*k*,*v*)#*l*) = *gen-list-to-map-aux upd* (*upd k v accs*)
l

definition *gen-list-to-map empt upd l* == *gen-list-to-map-aux upd* (*empt* ()) (*rev l*)

lemma *gen-list-to-map-correct*:
assumes *map-empty* α *invar empt*
assumes *map-update* α *invar upd*
shows *list-to-map* α *invar* (*gen-list-to-map empt upd*)
⟨*proof*⟩

3.1.17 Singleton (by empty, update)

definition *eu-sng*
:: (*unit* ⇒ '*m*) ⇒ ('*k* ⇒ '*v* ⇒ '*m* ⇒ '*m*) ⇒ '*k* ⇒ '*v* ⇒ '*m*
where *eu-sng empt update k v* == *update k v* (*empt* ())

lemma *eu-sng-correct*:
fixes *empty*
assumes *map-empty* α *invar empty*
assumes *map-update* α *invar update*
shows *map-sng* α *invar* (*eu-sng empty update*)
⟨*proof*⟩

3.1.18 Min (by iterateoi)

lemma *itoi-min-correct*:
assumes *iti*: *map-iterateoi* α *invar iti*
shows *map-min* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.1.19 Max (by reverse_iterateoi)

lemma *ritoi-max-correct*:

assumes *iti*: *map-reverse-iterateoi* α *invar iti*
shows *map-max* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.1.20 Conversion to sorted list (by reverse_iterateo)

lemma *rito-map-to-sorted-list-correct*:
assumes *iti*: *map-reverse-iterateoi* α *invar iti*
shows *map-to-sorted-list* α *invar* (*it-map-to-list iti*)
⟨*proof*⟩

3.1.21 image restrict

definition *it-map-image-filter* ::
($'s1 \Rightarrow ('u1 \times 'v1, 's2)$ *set-iterator*) $\Rightarrow$ (*unit* $\Rightarrow$ $'s2$) $\Rightarrow$ ($'u2 \Rightarrow 'v2 \Rightarrow 's2 \Rightarrow 's2$) $\Rightarrow$ ($'u1 \times 'v1 \Rightarrow ('u2 \times 'v2)$ *option*) $\Rightarrow$ $'s1 \Rightarrow 's2$
where *it-map-image-filter map-it map-emp map-up-dj f m* $\equiv$
iterate-to-map map-emp map-up-dj (*set-iterator-image-filter f* (*map-it m*))

lemma *it-map-image-filter-alt-def*[*code*]:
it-map-image-filter map-it map-emp map-up-dj f m $\equiv$
map-it m (λ-. *True*) ($\lambda kv\ \sigma$. *case* (*f kv*) *of None* $\Rightarrow \sigma$ | *Some* (k', v') $\Rightarrow$ (*map-up-dj* k' v' σ)) (*map-emp* ())
⟨*proof*⟩

lemma *it-map-image-filter-correct*:
fixes $\alpha 1 :: 's1 \Rightarrow 'u1 \rightharpoonup 'v1$
fixes $\alpha 2 :: 's2 \Rightarrow 'u2 \rightharpoonup 'v2$
assumes *it*: *map-iteratei* $\alpha 1$ *invar1 map-it*
assumes *emp*: *map-empty* $\alpha 2$ *invar2 map-emp*
assumes *up*: *map-update-dj* $\alpha 2$ *invar2 map-up-dj*
shows *map-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*it-map-image-filter map-it map-emp map-up-dj*)
⟨*proof*⟩

definition *mif-map-value-image-filter* ::
(($'u \times 'v1 \Rightarrow ('u \times 'v2)$ *option*) $\Rightarrow$ $'s1 \Rightarrow 's2$) $\Rightarrow$
($'u \Rightarrow 'v1 \Rightarrow 'v2$ *option*) $\Rightarrow$ $'s1 \Rightarrow 's2$
where
mif-map-value-image-filter mif f $\equiv$
mif ($\lambda(k, v)$. *case* (*f k v*) *of Some* $v' \Rightarrow$ *Some* (k, v') | *None* $\Rightarrow$ *None*)

lemma *mif-map-value-image-filter-correct* :
fixes $\alpha 1 :: 's1 \Rightarrow 'u \rightharpoonup 'v1$
fixes $\alpha 2 :: 's2 \Rightarrow 'u \rightharpoonup 'v2$
shows *map-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 mif* $\Longrightarrow$
map-value-image-filter $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*mif-map-value-image-filter mif*)
⟨*proof*⟩

definition *it-map-value-image-filter* ::
(*'s1* ⇒ (*'u* × *'v1*,*'s2*) *set-iterator*) ⇒ (*unit* ⇒ *'s2*) ⇒ (*'u* ⇒ *'v2* ⇒ *'s2* ⇒ *'s2*)
⇒
(*'u* ⇒ *'v1* ⇒ *'v2 option*) ⇒ *'s1* ⇒ *'s2* **where**
it-map-value-image-filter map-it map-emp map-up-dj f ≡
it-map-image-filter map-it map-emp map-up-dj
(λ(*k*, *v*). *case* (*f k v*) *of Some v'* ⇒ *Some* (*k*, *v'*) | *None* ⇒ *None*)

lemma *it-map-value-image-filter-alt-def* :
it-map-value-image-filter map-it map-emp map-up-dj =
mif-map-value-image-filter (*it-map-image-filter map-it map-emp map-up-dj*)
⟨*proof*⟩

lemma *it-map-value-image-filter-correct*:
fixes *α1* :: *'s1* ⇒ *'u* ⇀ *'v1*
fixes *α2* :: *'s2* ⇒ *'u* ⇀ *'v2*
assumes *it*: *map-iteratei α1 invar1 map-it*
assumes *emp*: *map-empty α2 invar2 map-emp*
assumes *up*: *map-update-dj α2 invar2 map-up-dj*
shows *map-value-image-filter α1 invar1 α2 invar2*
(*it-map-value-image-filter map-it map-emp map-up-dj*)
⟨*proof*⟩

definition *mif-map-restrict* ::
((*'u* × *'v* ⇒ (*'u* × *'v*) *option*) ⇒ *'s1* ⇒ *'s2*) ⇒
(*'u* × *'v* ⇒ *bool*) ⇒ *'s1* ⇒ *'s2*
where
mif-map-restrict mif P ≡
mif (λ(*k*, *v*). *if* (*P* (*k*, *v*)) *then Some* (*k*, *v*) *else None*)

lemma *mif-map-restrict-alt-def* :
mif-map-restrict mif P =
mif-map-value-image-filter mif (λ*k v*. *if* (*P* (*k*, *v*)) *then Some v else None*)
⟨*proof*⟩

lemma *mif-map-restrict-correct* :
fixes *α1* :: *'s1* ⇒ *'u* ⇀ *'v*
fixes *α2* :: *'s2* ⇒ *'u* ⇀ *'v*
shows *map-image-filter α1 invar1 α2 invar2 mif* ⟹
map-restrict α1 invar1 α2 invar2 (*mif-map-restrict mif*)
⟨*proof*⟩

definition *it-map-restrict* ::
(*'s1* ⇒ (*'u* × *'v*,*'s2*) *set-iterator*) ⇒ (*unit* ⇒ *'s2*) ⇒ (*'u* ⇒ *'v* ⇒ *'s2* ⇒ *'s2*) ⇒
(*'u* × *'v* ⇒ *bool*) ⇒ *'s1* ⇒ *'s2* **where**
it-map-restrict map-it map-emp map-up-dj P ≡
it-map-image-filter map-it map-emp map-up-dj
(λ(*k*, *v*). *if* (*P* (*k*, *v*)) *then Some* (*k*, *v*) *else None*)

lemma *it-map-restrict-alt-def* :
it-map-restrict map-it map-emp map-up-dj =
mif-map-restrict (*it-map-image-filter map-it map-emp map-up-dj*)
⟨*proof*⟩

lemma *it-map-restrict-correct*:
fixes *α1* :: *'s1* ⇒ *'u* ⇀ *'v*
fixes *α2* :: *'s2* ⇒ *'u* ⇀ *'v*
assumes *it*: *map-iteratei α1 invar1 map-it*
assumes *emp*: *map-empty α2 invar2 map-emp*
assumes *up*: *map-update-dj α2 invar2 map-up-dj*
shows *map-restrict α1 invar1 α2 invar2*
(*it-map-restrict map-it map-emp map-up-dj*)
⟨*proof*⟩

end

3.2 Generic Algorithms for Sets

theory *SetGA*
imports
../spec/SetSpec
../iterator/SetIteratorGA
../iterator/SetIteratorGACollections
begin

3.2.1 Singleton Set (by empty,insert)

definition *ei-sng* :: (*unit* ⇒ *'s*) ⇒ (*'x*⇒*'s*⇒*'s*) ⇒ *'x* ⇒ *'s* **where** *ei-sng e i x* = *i x* (*e* ())
lemma *ei-sng-correct*:
assumes *set-empty α invar e*
assumes *set-ins α invar i*
shows *set-sng α invar* (*ei-sng e i*)
⟨*proof*⟩

3.2.2 Disjoint Insert (by insert)

lemma (**in** *set-ins*) *ins-dj-by-ins*:
shows *set-ins-dj α invar ins*
⟨*proof*⟩

3.2.3 Disjoint Union (by union)

lemma (**in** *set-union*) *union-dj-by-union*:
shows *set-union-dj α1 invar1 α2 invar2 α3 invar3 union*
⟨*proof*⟩

3.2.4 Iterators

Iteratei (by iterateoi)

lemma *iti-by-itoi*:
assumes *set-iterateoi* α *invar it*
shows *set-iteratei* α *invar it*
⟨*proof*⟩

Iteratei (by reverse_iterateoi)

lemma *iti-by-ritoi*:
assumes *set-reverse-iterateoi* α *invar it*
shows *set-iteratei* α *invar it*
⟨*proof*⟩

3.2.5 Emptiness check (by iteratei)

definition *iti-isEmpty* **where**
iti-isEmpty iti = ($\lambda s.$ (*iterate-is-empty* (*iti s*)))

lemma *iti-isEmpty-code*[*code*] :
iti-isEmpty iti s = *iti s* ($\lambda c.$ *c*) (λ- -. *False*) *True*
⟨*proof*⟩

lemma *iti-isEmpty-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-isEmpty* α *invar* ($\lambda s.$ (*iterate-is-empty* (*iti s*)))
⟨*proof*⟩

3.2.6 Bounded Quantification (by iteratei)

definition *iti-ball* **where**
iti-ball iti = ($\lambda s.$ *iterate-ball* (*iti s*))

lemma *iti-ball-code*[*code*] :
iti-ball iti s P = *iti s* ($\lambda c.$ *c*) ($\lambda x\ \sigma.$ *P x*) *True*
⟨*proof*⟩

lemma *iti-ball-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-ball* α *invar* (*iti-ball iti*)
⟨*proof*⟩

definition *iti-bexists* **where**
iti-bexists iti = ($\lambda s.$ *iterate-bex* (*iti s*))

lemma *iti-bexists-code*[*code*] :
iti-bexists iti s P = *iti s* ($\lambda c.$ $\neg c$) ($\lambda x\ \sigma.$ *P x*) *False*
⟨*proof*⟩

lemma *iti-bexists-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-bexists* α *invar* (*iti-bexists iti*)
⟨*proof*⟩

definition *neg-ball-bexists*
:: ($'s \Rightarrow$ ($'a \Rightarrow$ *bool*) $\Rightarrow$ *bool*) $\Rightarrow$ $'s \Rightarrow$ ($'a \Rightarrow$ *bool*) $\Rightarrow$ *bool*
where *neg-ball-bexists ball s P* == $\neg$ (*ball s* ($\lambda x.\ \neg(P\ x)$))

lemma *neg-ball-bexists-correct*:
fixes *ball*
assumes *set-ball* α *invar ball*
shows *set-bexists* α *invar* (*neg-ball-bexists ball*)
⟨*proof*⟩

3.2.7 Size (by iterate)

definition *it-size* **where**
it-size iti = ($\lambda s.$ *iterate-size* (*iti s*))

lemma *it-size-code*[*code*] :
it-size iti s = *iti s* (λ-. *True*) ($\lambda x\ n$. *Suc n*) *0*
⟨*proof*⟩

lemma *it-size-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-size* α *invar* (*it-size iti*)
⟨*proof*⟩

3.2.8 Size with abort (by iterate)

definition *iti-size-abort* **where**
iti-size-abort iti = ($\lambda m\ s.$ *iterate-size-abort* (*iti s*) *m*)

lemma *iti-size-abort-code*[*code*] :
iti-size-abort iti m s = *iti s* ($\lambda\sigma.\ \sigma < m$) ($\lambda x.$ *Suc*) *0*
⟨*proof*⟩

lemma *iti-size-abort-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-size-abort* α *invar* (*iti-size-abort iti*)
⟨*proof*⟩

3.2.9 Singleton check (by size-abort)

definition *sza-isSng* **where**
sza-isSng iti = ($\lambda s.$ *iterate-is-sng* (*iti s*))

lemma *sza-isSng-code*[*code*] :
sza-isSng iti s = (*iti s* ($\lambda\sigma.\ \sigma < 2$) ($\lambda x.$ *Suc*) *0* = *1*)

⟨*proof*⟩

lemma *sza-isSng-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-isSng* α *invar* (*sza-isSng iti*)
⟨*proof*⟩

3.2.10 Copy (by iterate)

definition *it-copy* **where**
it-copy iti emp ins s = *iterate-to-set emp ins* (*iti s*)

lemma *it-copy-code*[*code*] :
it-copy iti emp ins s = *iti s* (λ-. *True*) *ins* (*emp* ())
⟨*proof*⟩

lemma *it-copy-correct*:
fixes α*1* :: *'s1* ⇒ *'x set*
fixes α*2* :: *'s2* ⇒ *'x set*
fixes *iteratei1* :: *'s1* ⇒ (*'x*,*'s2*) *set-iterator*
assumes *iti*: *set-iteratei* α*1 invar1 iteratei1*
assumes *emp-OK*: *set-empty* α*2 invar2 empty2*
assumes *ins-dj-OK*: *set-ins-dj* α*2 invar2 ins2*
shows *set-copy* α*1 invar1* α*2 invar2* (*it-copy iteratei1 empty2 ins2*)
⟨*proof*⟩

3.2.11 Union (by iterate)

definition *it-union* **where**
it-union it ins ≡ (λ*s1 s2*. *iterate-add-to-set s2 ins* (*it s1*))
lemma *it-union-code*[*code*] :
it-union it ins s1 s2 = *it s1* (λ-. *True*) *ins s2*
⟨*proof*⟩

lemma *it-union-correct*:
fixes α*1* :: *'s1* ⇒ *'x set*
fixes α*2* :: *'s2* ⇒ *'x set*
assumes *S1*: *set-iteratei* α*1 invar1 iteratei1*
assumes *S2*: *set-ins* α*2 invar2 ins2*
shows *set-union* α*1 invar1* α*2 invar2* α*2 invar2* (*it-union iteratei1 ins2*)
⟨*proof*⟩

3.2.12 Disjoint Union

definition [*code-unfold*]: *it-union-dj* ≡ *it-union*

lemma *it-union-dj-correct*:
fixes α*1* :: *'s1* ⇒ *'x set*
fixes α*2* :: *'s2* ⇒ *'x set*
assumes *S1*: *set-iteratei* α*1 invar1 iteratei1*

assumes *S2*: *set-ins-dj α2 invar2 ins2*
shows *set-union-dj α1 invar1 α2 invar2 α2 invar2* (*it-union-dj iteratei1 ins2*)
⟨*proof*⟩

3.2.13 Diff (by iterator)

definition *it-diff* **where**
it-diff iteratei1 del2 ≡ (λ*s2 s1*. *iterate-diff-set s2 del2* (*iteratei1 s1*))

lemma *it-diff-code*[*code*]:
it-diff it del s1 s2 = *it s2* (λ-. *True*) *del s1*
⟨*proof*⟩

lemma *it-diff-correct*:
fixes *α1* :: *'s1* ⇒ *'x set*
fixes *α2* :: *'s2* ⇒ *'x set*
assumes *S1*: *set-iteratei α1 invar1 iteratei1*
assumes *S2*: *set-delete α2 invar2 del2*
shows *set-diff α2 invar2 α1 invar1* (*it-diff iteratei1 del2*)
⟨*proof*⟩

3.2.14 Intersection (by iterator)

definition *it-inter*
:: (*'s1* ⇒ (*'x*,*'s3*) *set-iterator*) ⇒
(*'x* ⇒ *'s2* ⇒ *bool*) ⇒
(*unit* ⇒ *'s3*) ⇒
(*'x* ⇒ *'s3* ⇒ *'s3*) ⇒
's1 ⇒ *'s2* ⇒ *'s3*
where
it-inter iteratei1 memb2 empt3 insrt3 s1 s2 ≡
iterate-to-set empt3 insrt3 (*set-iterator-filter* (λ*x*. *memb2 x s2*) (*iteratei1 s1*))

lemma *it-inter-code*[*code*] :
it-inter iteratei1 memb2 empt3 insrt3 s1 s2 ==
iteratei1 s1 (λ-. *True*) (λ*x s*. *if memb2 x s2 then insrt3 x s else s*)
(*empt3* ())
⟨*proof*⟩

lemma *it-inter-correct*:
fixes *α1* :: *'s1* ⇒ *'x set*
fixes *α2* :: *'s2* ⇒ *'x set*
fixes *α3* :: *'s3* ⇒ *'x set*
assumes *it1*: *set-iteratei α1 invar1 iteratei1*
assumes *memb2*: *set-memb α2 invar2 memb2*
assumes *emp3*: *set-empty α3 invar3 empty3*
assumes *ins3*: *set-ins-dj α3 invar3 ins3*
shows *set-inter α1 invar1 α2 invar2 α3 invar3* (*it-inter iteratei1 memb2 empty3 ins3*)
⟨*proof*⟩

3.2.15 Subset (by ball)

definition *ball-subset* ::
  $('s1 \Rightarrow ('a \Rightarrow bool) \Rightarrow bool) \Rightarrow ('a \Rightarrow 's2 \Rightarrow bool) \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$
  where *ball-subset ball1 mem2 s1 s2 == ball1 s1* $(\lambda x.$ *mem2 x s2*)

lemma *ball-subset-correct*:
  assumes *set-ball* $\alpha 1$ *invar1 ball1*
  assumes *set-memb* $\alpha 2$ *invar2 mem2*
  shows *set-subset* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*ball-subset ball1 mem2*)
⟨*proof*⟩

3.2.16 Equality Test (by subset)

definition *subset-equal* ::
  $('s1 \Rightarrow 's2 \Rightarrow bool) \Rightarrow ('s2 \Rightarrow 's1 \Rightarrow bool) \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$
  where *subset-equal ss1 ss2 s1 s2 == ss1 s1 s2* $\wedge$ *ss2 s2 s1*

lemma *subset-equal-correct*:
  assumes *set-subset* $\alpha 1$ *invar1* $\alpha 2$ *invar2 ss1*
  assumes *set-subset* $\alpha 2$ *invar2* $\alpha 1$ *invar1 ss2*
  shows *set-equal* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*subset-equal ss1 ss2*)
⟨*proof*⟩

3.2.17 Image-Filter (by iterate)

definition *it-image-filter*
  :: $('s1 \Rightarrow ('a,\text{-})$ *set-iterator*$) \Rightarrow$ (*unit* $\Rightarrow$ *'s2*) $\Rightarrow$ $('b \Rightarrow 's2 \Rightarrow 's2)$
    $\Rightarrow ('a \rightharpoonup 'b) \Rightarrow 's1 \Rightarrow 's2$
  where *it-image-filter iteratei1 empty2 ins2 f s ==*
      iterate-to-set empty2 ins2 (*set-iterator-image-filter f* (*iteratei1 s*))

lemma *it-image-filter-code*[*code*] :
  it-image-filter iteratei1 empty2 ins2 f s ==
  iteratei1 s (λ-. *True*) ($\lambda x\ res.$ *case f x of Some v* $\Rightarrow$ *ins2 v res* | - $\Rightarrow$ *res*) (*empty2* ())
⟨*proof*⟩

lemma *it-image-filter-correct*:
  fixes $\alpha 1$ :: $'s1 \Rightarrow 'x1$ *set*
  fixes $\alpha 2$ :: $'s2 \Rightarrow 'x2$ *set*
  assumes *iti1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
  assumes *emp2*: *set-empty* $\alpha 2$ *invar2 empty2*
  assumes *ins*: *set-ins* $\alpha 2$ *invar2 ins2*
  shows *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*it-image-filter iteratei1 empty2 ins2*)
⟨*proof*⟩

3.2.18 Injective Image-Filter (by iterate)

definition [*code-unfold*] : *it-inj-image-filter* = *it-image-filter*

lemma *it-inj-image-filter-correct*:
fixes $\alpha 1$ :: *'s1* ⇒ *'x1 set*
fixes $\alpha 2$ :: *'s2* ⇒ *'x2 set*
assumes *iti1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
assumes *emp2*: *set-empty* $\alpha 2$ *invar2 empty2*
assumes *ins-dj2*: *set-ins-dj* $\alpha 2$ *invar2 ins2*
shows *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2*
(*it-inj-image-filter iteratei1 empty2 ins2*)
⟨*proof*⟩

3.2.19 Image (by image-filter)

definition *iflt-image iflt f s* == *iflt* (λ*x*. *Some* (*f x*)) *s*

lemma *iflt-image-correct*:
assumes *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt*
shows *set-image* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*iflt-image iflt*)
⟨*proof*⟩

3.2.20 Injective Image-Filter (by image-filter)

definition [*code-unfold*]: *iflt-inj-image* = *iflt-image*

lemma *iflt-inj-image-correct*:
assumes *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt*
shows *set-inj-image* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*iflt-inj-image iflt*)
⟨*proof*⟩

3.2.21 Filter (by image-filter)

definition *iflt-filter iflt P s* == *iflt* (λ*x*. *if P x then Some x else None*) *s*

lemma *iflt-filter-correct*:
fixes $\alpha 1$ :: *'s1* ⇒ *'a set*
fixes $\alpha 2$ :: *'s2* ⇒ *'a set*
assumes *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt*
shows *set-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*iflt-filter iflt*)
⟨*proof*⟩

3.2.22 union-list

definition *fold-union-list* **where**
fold-union-list emp un l =
foldl (λ*s s'*. *un s s'*) (*emp* ()) *l*

lemma *fold-union-list-correct* :
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *un-OK*: *set-union* α *invar* α *invar* α *invar un*
shows *set-union-list* α *invar* α *invar* (*fold-union-list emp un*)

⟨*proof*⟩

function *paired-union-list* :: - ⇒ - ⇒ *'a set list* ⇒ *'a set list* ⇒ *'a set* **where**
  paired-union-list emp un [] [] = *emp* ()
| *paired-union-list emp un* [] [*s*] = *s*
| *paired-union-list emp un* (*s1* # *l1*) [] = *paired-union-list emp un* [] (*s1* # *l1*)
| *paired-union-list emp un* (*s1* # *l1*) [*s2*] = *paired-union-list emp un* [] (*s2* # *s1* # *l1*)
| *paired-union-list emp un l1* (*s1* # *s2* # *l2*) =
   paired-union-list emp un ((*un s1 s2*) # *l1*) *l2*
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *paired-union-list-correct* :
 assumes *emp-OK*: *set-empty* α *invar emp*
 assumes *un-OK*: *set-union* α *invar* α *invar* α *invar un*
 shows *set-union-list* α *invar* α *invar* (*paired-union-list emp un* [])
⟨*proof*⟩

3.2.23 Union of image of Set (by iterate)

definition *it-Union-image*
 :: (*'s1* ⇒ (*'x*,-) *set-iterator*) ⇒ (*unit* ⇒ *'s3*) ⇒ (*'s2* ⇒ *'s3* ⇒ *'s3*)
    ⇒ (*'x* ⇒ *'s2*) ⇒ *'s1* ⇒ *'s3*
where *it-Union-image iti1 em3 un233 f S* ==
 iti1 S (λ-. *True*) (λ*x res*. *un233* (*f x*) *res*) (*em3* ())

lemma *it-Union-image-correct*:
 assumes *set-iteratei* α*1 invar1 iti1*
 assumes *set-empty* α*3 invar3 em3*
 assumes *set-union* α*2 invar2* α*3 invar3* α*3 invar3 un233*
 shows *set-Union-image* α*1 invar1* α*2 invar2* α*3 invar3* (*it-Union-image iti1 em3 un233*)
⟨*proof*⟩

3.2.24 Disjointness Check with Witness (by sel)

definition *sel-disjoint-witness sel1 mem2 s1 s2* ==
 sel1 s1 (λ*x*. *if mem2 x s2 then Some x else None*)

lemma *sel-disjoint-witness-correct*:
 assumes *set-sel* α*1 invar1 sel1*
 assumes *set-memb* α*2 invar2 mem2*
 shows *set-disjoint-witness* α*1 invar1* α*2 invar2* (*sel-disjoint-witness sel1 mem2*)
⟨*proof*⟩

3.2.25 Disjointness Check (by ball)

definition *ball-disjoint ball1 memb2 s1 s2* == *ball1 s1* (λ*x*. ¬ *memb2 x s2*)

lemma *ball-disjoint-correct*:
assumes *set-ball* $\alpha 1$ *invar1 ball1*
assumes *set-memb* $\alpha 2$ *invar2 memb2*
shows *set-disjoint* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*ball-disjoint ball1 memb2*)
⟨*proof*⟩

3.2.26 Selection (by iteratei)

definition *iti-sel* **where**
iti-sel iti = ($\lambda s\ f.$ *iterate-sel* (*iti s*) *f*)

lemma *iti-sel-code*[*code*]:
iti-sel iti s f = *iti s* ($\lambda\sigma.\ \sigma$ = *None*) ($\lambda x\ \sigma.\ f\ x$) *None*
⟨*proof*⟩

lemma *iti-sel-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-sel* α *invar* (*iti-sel iti*)
⟨*proof*⟩

3.2.27 Map-free selection by selection

definition *sel-sel′*
:: ('*s* ⇒ ('*x* ⇒ - *option*) ⇒ - *option*) ⇒ '*s* ⇒ ('*x* ⇒ *bool*) ⇒ '*x option*
where *sel-sel′ sel s P* = *sel s* ($\lambda x.$ *if P x then Some x else None*)

lemma *sel-sel′-correct*:
assumes *set-sel* α *invar sel*
shows *set-sel′* α *invar* (*sel-sel′ sel*)
⟨*proof*⟩

3.2.28 Map-free selection by iterate

definition *iti-sel-no-map* **where**
iti-sel-no-map iti = ($\lambda s\ P.$ *iterate-sel-no-map* (*iti s*) *P*)

lemma *iti-sel-no-map-code*[*code*]:
iti-sel-no-map iti s f = *iti s* ($\lambda\sigma.\ \sigma$ = *None*) ($\lambda x\ \sigma.$ *if f x then Some x else None*)
None
⟨*proof*⟩

lemma *iti-sel′-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-sel′* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.2.29 Set to List (by iterate)

definition *it-set-to-list* **where**

it-set-to-list iti = (λ*s*. *iterate-to-list* (*iti s*))

lemma *it-set-to-list-code*[*code*] :
it-set-to-list iti s = *iti s* (λ-. *True*) *op#* []
⟨*proof*⟩

lemma *it-set-to-list-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-to-list* α *invar* (*it-set-to-list iti*)
⟨*proof*⟩

3.2.30 List to Set

— Tail recursive version
fun *gen-list-to-set-aux*
:: ('*x* ⇒ '*s* ⇒ '*s*) ⇒ '*s* ⇒ '*x list* ⇒ '*s*
where
gen-list-to-set-aux ins accs [] = *accs* |
gen-list-to-set-aux ins accs (*x*#*l*) = *gen-list-to-set-aux ins* (*ins x accs*) *l*

definition *gen-list-to-set empt ins* == *gen-list-to-set-aux ins* (*empt* ())

lemma *gen-list-to-set-correct*:
assumes *set-empty* α *invar empt*
assumes *set-ins* α *invar ins*
shows *list-to-set* α *invar* (*gen-list-to-set empt ins*)
⟨*proof*⟩

3.2.31 More Generic Set Algorithms

These algorithms do not have a function specification in a locale, but their specification is done ad-hoc in the correctness lemma.

Image and Filter of Cartesian Product

definition *image-filter-cartesian-product*
:: ('*s1* ⇒ ('*x*,'*s3*) *set-iterator*) ⇒
('*s2* ⇒ ('*y*,'*s3*) *set-iterator*) ⇒
(*unit* ⇒ '*s3*) ⇒ ('*z* ⇒ '*s3* ⇒ '*s3*) ⇒
('*x* × '*y* ⇒ '*z option*) ⇒ '*s1* ⇒ '*s2* ⇒ '*s3*
where
image-filter-cartesian-product iterate1 iterate2 empty3 insert3 f s1 s2 ==
iterate-to-set empty3 insert3 (*set-iterator-image-filter f* (
set-iterator-product (*iterate1 s1*) (λ-. *iterate2 s2*)))

lemma *image-filter-cartesian-product-code*[*code*]:
image-filter-cartesian-product iterate1 iterate2 empty3 insert3 f s1 s2 ==
iterate1 s1 (λ-. *True*) (λ*x res*.
iterate2 s2 (λ-. *True*) (λ*y res*.

```
      case (f (x, y)) of
        None ⇒ res
      | Some z ⇒ (insert3 z res)
    ) res
  ) (empty3 ())
⟨proof⟩

lemma image-filter-cartesian-product-correct:
  assumes S: set-iteratei α1 invar1 iteratei1
             set-iteratei α2 invar2 iteratei2
             set-empty α3 invar3 empty3
             set-ins α3 invar3 ins3
  assumes I[simp, intro!]: invar1 s1    invar2 s2
  shows α3 (image-filter-cartesian-product iteratei1 iteratei2 empty3 ins3 f s1 s2)
   = { z | x y z. f (x,y) = Some z ∧ x∈α1 s1 ∧ y∈α2 s2 } (is ?T1)
  invar3 (image-filter-cartesian-product iteratei1 iteratei2 empty3 ins3 f s1 s2) (is
?T2)
⟨proof⟩


definition image-filter-cp where
  image-filter-cp iterate1 iterate2 empty3 insert3 f P s1 s2 ≡
   image-filter-cartesian-product iterate1 iterate2 empty3 insert3
     (λxy. if P xy then Some (f xy) else None) s1 s2

lemma image-filter-cp-correct:
  assumes S: set-iteratei α1 invar1 iterate1
             set-iteratei α2 invar2 iterate2
             set-empty α3 invar3 empty3
             set-ins α3 invar3 ins3
  assumes I: invar1 s1    invar2 s2
  shows
  α3 (image-filter-cp iterate1 iterate2 empty3 ins3 f P s1 s2)
   = { f (x, y) | x y. P (x, y) ∧ x∈α1 s1 ∧ y∈α2 s2 } (is ?T1)
  invar3 (image-filter-cp iterate1 iterate2 empty3 ins3 f P s1 s2) (is ?T2)
⟨proof⟩
```

Injective Image and Filter of Cartesian Product

```
definition[code-unfold]:
  inj-image-filter-cartesian-product = image-filter-cartesian-product

lemma inj-image-filter-cartesian-product-correct:
  assumes S: set-iteratei α1 invar1 iteratei1
             set-iteratei α2 invar2 iteratei2
             set-empty α3 invar3 empty3
             set-ins-dj α3 invar3 ins-dj3
  assumes I[simp, intro!]: invar1 s1    invar2 s2
  assumes INJ: !!x y x' y' z. ⟦ f (x, y) = Some z; f (x', y') = Some z ⟧ ⟹ x=x'
```

$\wedge$ $y{=}y'$
shows $\alpha 3$ (*inj-image-filter-cartesian-product iteratei1 iteratei2 empty3 ins-dj3 f s1 s2*)
$= \{\ z \mid x\ y\ z.\ f\ (x,\ y) = \mathit{Some}\ z \wedge x{\in}\alpha 1\ s1 \wedge y{\in}\alpha 2\ s2\ \}$ (**is** *?T1*)
invar3 (*inj-image-filter-cartesian-product iteratei1 iteratei2 empty3 ins-dj3 f s1 s2*) (**is** *?T2*)
⟨*proof*⟩

Injective Image and Filter of Cartesian Product

definition [*code-unfold*]: *inj-image-filter-cp = image-filter-cp*

lemma *inj-image-filter-cp-correct*:
assumes *S*: *set-iteratei* $\alpha 1$ *invar1 iterate1*
set-iteratei $\alpha 2$ *invar2 iterate2*
set-empty $\alpha 3$ *invar3 empty3*
set-ins-dj $\alpha 3$ *invar3 ins-dj3*
assumes *I*[*simp*, *intro*!]: *invar1 s1* *invar2 s2*
assumes *INJ*: !!$x\ y\ x'\ y'\ z.\ [\![\ P\ (x,\ y);\ P\ (x',\ y');\ f\ (x,\ y) = f\ (x',\ y')\ ]\!] \Longrightarrow x{=}x' \wedge y{=}y'$
shows $\alpha 3$ (*inj-image-filter-cp iterate1 iterate2 empty3 ins-dj3 f P s1 s2*)
$= \{\ f\ (x,\ y) \mid x\ y.\ P\ (x,\ y) \wedge x{\in}\alpha 1\ s1 \wedge y{\in}\alpha 2\ s2\ \}$ (**is** *?T1*)
invar3 (*inj-image-filter-cp iterate1 iterate2 empty3 ins-dj3 f P s1 s2*) (**is** *?T2*)
⟨*proof*⟩

Cartesian Product

definition *cart it1 it2 empty3 ins3 s1 s2 == image-filter-cartesian-product it1 it2 empty3 ins3* ($\lambda xy.$ *Some xy*) *s1 s2*

lemma *cart-code*[*code*] :
cart it1 it2 empty3 ins3 s1 s2 ≡
it1 s1 (λ-. *True*) ($\lambda x.$ *it2 s2* (λ-. *True*) ($\lambda y\ res.$ *ins3* (*x*,*y*) *res*)) (*empty3* ())
⟨*proof*⟩

lemma *cart-correct*:
assumes *S*: *set-iteratei* $\alpha 1$ *invar1 iterate1*
set-iteratei $\alpha 2$ *invar2 iterate2*
set-empty $\alpha 3$ *invar3 empty3*
set-ins-dj $\alpha 3$ *invar3 ins-dj3*
assumes *I*[*simp*, *intro*!]: *invar1 s1* *invar2 s2*
shows $\alpha 3$ (*cart iterate1 iterate2 empty3 ins-dj3 s1 s2*)
$= \alpha 1\ s1 \times \alpha 2\ s2$ (**is** *?T1*)
invar3 (*cart iterate1 iterate2 empty3 ins-dj3 s1 s2*) (**is** *?T2*)
⟨*proof*⟩

3.2.32 Min (by iterateoi)

lemma *itoi-min-correct*:
assumes *iti*: *set-iterateoi* α *invar iti*

shows *set-min* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.2.33 Max (by reverse_iterateoi)

lemma *ritoi-max-correct*:
assumes *iti*: *set-reverse-iterateoi* α *invar iti*
shows *set-max* α *invar* (*iti-sel-no-map iti*)
⟨*proof*⟩

3.2.34 Conversion to sorted list (by reverse_iterateo)

lemma *rito-set-to-sorted-list-correct*:
assumes *iti*: *set-reverse-iterateoi* α *invar iti*
shows *set-to-sorted-list* α *invar* (*it-set-to-list iti*)
⟨*proof*⟩

end

3.3 Implementing Sets by Maps

theory *SetByMap*
imports
../spec/SetSpec
../spec/MapSpec
../common/Misc
../iterator/SetIteratorOperations
../iterator/SetIteratorGA
begin

In this theory, we show how to implement sets by maps.

3.3.1 Definitions

definition *s-*α :: $('s \Rightarrow 'u \rightharpoonup unit) \Rightarrow 's \Rightarrow 'u\ set$
where *s-*α α *m* == *dom* (α *m*)
definition[*code-unfold*]: *s-empty empt* = *empt*
definition[*code-unfold*]: *s-sng sng x* = *sng x* ()
definition *s-memb lookup x s* == *lookup x s* ≠ *None*
definition[*code-unfold*]: *s-ins update x s* == *update x* () *s*
definition[*code-unfold*]: *s-ins-dj update-dj x s* == *update-dj x* () *s*
definition[*code-unfold*]: *s-delete delete* == *delete*
definition *s-sel*
:: $('s \Rightarrow ('u \times unit \Rightarrow 'r\ option) \Rightarrow 'r\ option) \Rightarrow$
$'s \Rightarrow ('u \Rightarrow 'r\ option) \Rightarrow 'r\ option$
where *s-sel sel s f* == *sel s* (λ(*u*, -). *f u*)
declare *s-sel-def*[*code-unfold*]
definition[*code-unfold*]: *s-isEmpty isEmpty* == *isEmpty*

definition[*code-unfold*]: *s-isSng isSng == isSng*

definition *s-iteratei* :: (*'s* ⇒ ((*'k* × *unit*),*'σ*) *set-iterator*) ⇒ (*'s* ⇒ (*'k*, *'σ*) *set-iterator*)
  where *s-iteratei it s == map-iterator-dom* (*it s*)
lemma *s-iteratei-code*[*code*] :
  s-iteratei it s c f = *it s c* (λ*x*. *f* (*fst x*))
⟨*proof*⟩

definition *s-ball*
  :: (*'s* ⇒ (*'u* × *unit* ⇒ *bool*) ⇒ *bool*) ⇒ *'s* ⇒ (*'u* ⇒ *bool*) ⇒ *bool*
  where *s-ball ball s P == ball s* (λ(*u*, -). *P u*)
declare *s-ball-def*[*code-unfold*]

definition *s-bexists*
  :: (*'s* ⇒ (*'u* × *unit* ⇒ *bool*) ⇒ *bool*) ⇒ *'s* ⇒ (*'u* ⇒ *bool*) ⇒ *bool*
  where *s-bexists bexists s P == bexists s* (λ(*u*,-). *P u*)
declare *s-bexists-def*[*code-unfold*]

definition[*code-unfold*]: *s-size sz* ≡ *sz*
definition[*code-unfold*]: *s-size-abort size-abort* ≡ *size-abort*

definition[*code-unfold*]: *s-union add == add*
definition[*code-unfold*]: *s-union-dj add-dj* = *add-dj*

lemmas *s-defs* =
  s-α-def
  s-empty-def
  s-sng-def
  s-memb-def
  s-ins-def
  s-ins-dj-def
  s-delete-def
  s-sel-def
  s-isEmpty-def
  s-isSng-def
  s-iteratei-def
  s-ball-def
  s-bexists-def
  s-size-def
  s-size-abort-def
  s-union-def
  s-union-dj-def

3.3.2 Correctness

lemma *s-empty-correct*:
  fixes *empty*
  assumes *map-empty α invar empty*

shows *set-empty* (*s-α α*) *invar* (*s-empty empty*)
⟨*proof*⟩

lemma *s-sng-correct*:
fixes *sng*
assumes *map-sng α invar sng*
shows *set-sng* (*s-α α*) *invar* (*s-sng sng*)
⟨*proof*⟩

lemma *s-memb-correct*:
fixes *lookup*
assumes *map-lookup α invar lookup*
shows *set-memb* (*s-α α*) *invar* (*s-memb lookup*)
⟨*proof*⟩

lemma *s-ins-correct*:
fixes *ins*
assumes *map-update α invar update*
shows *set-ins* (*s-α α*) *invar* (*s-ins update*)
⟨*proof*⟩

lemma *s-ins-dj-correct*:
fixes *ins-dj*
assumes *map-update-dj α invar update-dj*
shows *set-ins-dj* (*s-α α*) *invar* (*s-ins-dj update-dj*)
⟨*proof*⟩

lemma *s-delete-correct*:
fixes *delete*
assumes *map-delete α invar delete*
shows *set-delete* (*s-α α*) *invar* (*s-delete delete*)
⟨*proof*⟩

lemma *s-sel-correct*:
fixes *sel*
assumes *sel-OK*: *map-sel α invar sel*
shows *set-sel* (*s-α α*) *invar* (*s-sel sel*)
⟨*proof*⟩

lemma *s-isEmpty-correct*:
fixes *isEmpty*
assumes *map-isEmpty α invar isEmpty*
shows *set-isEmpty* (*s-α α*) *invar* (*s-isEmpty isEmpty*)
⟨*proof*⟩

lemma *s-isSng-correct*:
fixes *isSng*
assumes *map-isSng α invar isSng*

shows *set-isSng* (*s-α α*) *invar* (*s-isSng isSng*)
⟨*proof*⟩

lemma *s-iteratei-correct*:
fixes *iteratei*
assumes *map-it-OK*: *map-iteratei α invar iteratei*
shows *set-iteratei* (*s-α α*) *invar* (*s-iteratei iteratei*)
⟨*proof*⟩

lemma *s-iterateoi-correct*:
fixes *iteratei*
assumes *map-it-OK*: *map-iterateoi α invar iteratei*
shows *set-iterateoi* (*s-α α*) *invar* (*s-iteratei iteratei*)
⟨*proof*⟩

lemma *s-reverse-iterateoi-correct*:
fixes *iteratei*
assumes *map-it-OK*: *map-reverse-iterateoi α invar iteratei*
shows *set-reverse-iterateoi* (*s-α α*) *invar* (*s-iteratei iteratei*)
⟨*proof*⟩

lemma *s-ball-correct*:
fixes *ball*
assumes *map-ball α invar ball*
shows *set-ball* (*s-α α*) *invar* (*s-ball ball*)
⟨*proof*⟩

lemma *s-bexists-correct*:
fixes *bexists*
assumes *map-bexists α invar bexists*
shows *set-bexists* (*s-α α*) *invar* (*s-bexists bexists*)
⟨*proof*⟩

lemma *s-size-correct*:
fixes *size*
assumes *map-size α invar size*
shows *set-size* (*s-α α*) *invar* (*s-size size*)
⟨*proof*⟩

lemma *s-size-abort-correct*:
fixes *size-abort*
assumes *map-size-abort α invar size-abort*
shows *set-size-abort* (*s-α α*) *invar* (*s-size-abort size-abort*)
⟨*proof*⟩

lemma *s-union-correct*:
fixes *add*

assumes *map-add* α *invar add*
shows *set-union* (*s*-α α) *invar* (*s*-α α) *invar* (*s*-α α) *invar* (*s-union add*)
⟨*proof*⟩

lemma *s-union-dj-correct*:
fixes *add-dj*
assumes *map-add-dj* α *invar add-dj*
shows *set-union-dj* (*s*-α α) *invar* (*s*-α α) *invar* (*s*-α α) *invar* (*s-union-dj add-dj*)
⟨*proof*⟩

lemma *finite-set-by-map*:
assumes *BM*: *finite-map* α *invar*
shows *finite-set* (*s*-α α) *invar*
⟨*proof*⟩

end

3.4 Generic Algorithms for Sequences

theory *ListGA*
imports
../spec/ListSpec
../common/Misc
begin

3.4.1 Iterators

iteratei (by get, size)

fun *idx-iteratei-aux*
:: ($'s \Rightarrow nat \Rightarrow 'a$) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'s \Rightarrow$ ($'\sigma \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow '\sigma \Rightarrow '\sigma$) $\Rightarrow$ $'\sigma \Rightarrow$ $'\sigma$
where
idx-iteratei-aux get sz i l c f σ = (
if i=0 $\vee$ $\neg$ *c* σ *then* σ
else idx-iteratei-aux get sz (*i* − *1*) *l c f* (*f* (*get l* (*sz*−*i*)) σ)
)

declare *idx-iteratei-aux.simps*[*simp del*]

lemma *idx-iteratei-aux-simps*[*simp*]:
i=0 $\Longrightarrow$ *idx-iteratei-aux get sz i l c f* σ = σ
$\neg$*c* σ $\Longrightarrow$ *idx-iteratei-aux get sz i l c f* σ = σ
⟦*i*≠*0*; *c* σ⟧ $\Longrightarrow$ *idx-iteratei-aux get sz i l c f* σ = *idx-iteratei-aux get sz* (*i* − *1*) *l c f* (*f* (*get l* (*sz*−*i*)) σ)
⟨*proof*⟩

definition *idx-iteratei get sz l c f* σ == *idx-iteratei-aux get* (*sz l*) (*sz l*) *l c f* σ

lemma *idx-iteratei-eq-foldli*:
assumes *list-size* α *invar sz*
assumes *list-get* α *invar get*
assumes *invar s*
shows *idx-iteratei get sz s* = *foldli* (α *s*)
⟨*proof*⟩

lemma *idx-iteratei-correct*:
assumes *list-size* α *invar sz*
assumes *list-get* α *invar get*
shows *list-iteratei* α *invar* (*idx-iteratei get sz*)
⟨*proof*⟩

reverse_iteratei (by get, size)

fun *idx-reverse-iteratei-aux*
:: ('*s* ⇒ *nat* ⇒ '*a*) ⇒ *nat* ⇒ *nat* ⇒ '*s* ⇒ ('σ⇒*bool*) ⇒ ('*a* ⇒'σ ⇒ 'σ) ⇒ 'σ ⇒ 'σ
where
idx-reverse-iteratei-aux get sz i l c f σ = (
if i=0 ∨ ¬ *c* σ *then* σ
else idx-reverse-iteratei-aux get sz (*i* − *1*) *l c f* (*f* (*get l* (*i* − *1*)) σ)
)

declare *idx-reverse-iteratei-aux.simps*[*simp del*]

lemma *idx-reverse-iteratei-aux-simps*[*simp*]:
i=0 ⟹ *idx-reverse-iteratei-aux get sz i l c f* σ = σ
¬*c* σ ⟹ *idx-reverse-iteratei-aux get sz i l c f* σ = σ
⟦*i≠0*; *c* σ⟧ ⟹ *idx-reverse-iteratei-aux get sz i l c f* σ = *idx-reverse-iteratei-aux get sz* (*i* − *1*) *l c f* (*f* (*get l* (*i* − *1*)) σ)
⟨*proof*⟩

definition *idx-reverse-iteratei get sz l c f* σ == *idx-reverse-iteratei-aux get* (*sz l*) (*sz l*) *l c f* σ

lemma *idx-reverse-iteratei-eq-foldri*:
assumes *list-size* α *invar sz*
assumes *list-get* α *invar get*
assumes *invar s*
shows *idx-reverse-iteratei get sz s* = *foldri* (α *s*)
⟨*proof*⟩

lemma *idx-reverse-iteratei-correct*:
assumes *list-size* α *invar sz*
assumes *list-get* α *invar get*
shows *list-reverse-iteratei* α *invar* (*idx-reverse-iteratei get sz*)
⟨*proof*⟩

3.4.2 Size (by iterator)

definition *it-size* :: (*'s* ⇒ (*'x*,*nat*) *set-iterator*) ⇒ *'s* ⇒ *nat*
where *it-size iti l* == *iti l* (λ-. *True*) (λ*x res*. *Suc res*) (*0*::*nat*)

lemma *it-size-correct*:
assumes *list-iteratei* α *invar it*
shows *list-size* α *invar* (*it-size it*)
⟨*proof*⟩

By reverse_iterator

lemma *it-size-correct-rev*:
assumes *list-reverse-iteratei* α *invar it*
shows *list-size* α *invar* (*it-size it*)
⟨*proof*⟩

3.4.3 Get (by iteratori)

definition *iti-get*:: (*'s* ⇒ (*'x*,*nat* × *'x option*) *set-iterator*) ⇒ *'s* ⇒ *nat* ⇒ *'x*
where *iti-get iti s i* =
the (*snd* (*iti s*
(λ(*i*,*x*). *x*=*None*)
(λ*x* (*i*,-). *if i*=*0 then* (*0*,*Some x*) *else* (*i* − *1*,*None*))
(*i*,*None*)))

lemma *iti-get-correct*:
assumes *iti-OK*: *list-iteratei* α *invar iti*
shows *list-get* α *invar* (*iti-get iti*)
⟨*proof*⟩

end

3.5 Indices of Sets

theory *SetIndex*
imports
../common/Misc
../spec/MapSpec
../spec/SetSpec
begin

This theory defines an indexing operation that builds an index from a set and an indexing function.

Here, index is a map from indices to all values of the set with that index.

3.5.1 Indexing by Function

definition *index* :: $('a \Rightarrow 'i) \Rightarrow 'a\ set \Rightarrow 'i \Rightarrow 'a\ set$
where $index\ f\ s\ i == \{\ x{\in}s\ .\ f\ x = i\ \}$

lemma *indexI*: $[\![\ x{\in}s;\ f\ x = i\ ]\!] \Longrightarrow x{\in}index\ f\ s\ i$ ⟨*proof*⟩
lemma *indexD*:
$x{\in}index\ f\ s\ i \Longrightarrow x{\in}s$
$x{\in}index\ f\ s\ i \Longrightarrow f\ x = i$
⟨*proof*⟩

lemma *index-iff*[*simp*]: $x{\in}index\ f\ s\ i \longleftrightarrow x{\in}s \wedge f\ x = i$ ⟨*proof*⟩

3.5.2 Indexing by Map

definition *index-map* :: $('a \Rightarrow 'i) \Rightarrow 'a\ set \Rightarrow 'i \rightharpoonup 'a\ set$
where *index-map f s i* == *let s*=*index f s i in if s*={} *then None else Some s*

definition *im-α* **where** *im-α im i* == *case im i of None* ⇒ {} | *Some s* ⇒ *s*

lemma *index-map-correct*: *im-α* (*index-map f s*) = *index f s*
⟨*proof*⟩

3.5.3 Indexing by Maps and Sets from the Isabelle Collections Framework

In this theory, we define the generic algorithm as constants outside any locale, but prove the correctness lemmas inside a locale that assumes correctness of all prerequisite functions. Finally, we export the correctness lemmas from the locale.

— Lookup
definition *idx-lookup* :: - ⇒ - ⇒ *'i* ⇒ *'m* ⇒ *'t* **where**
idx-lookup mlookup tempty i m == *case mlookup i m of None* ⇒ (*tempty* ()) | *Some s* ⇒ *s*

locale *index-env* =
m: *map-lookup mα minvar mlookup* +
t: *set-empty tα tinvar tempty*
for *mα* :: *'m* ⇒ *'i* ⇀ *'t* **and** *minvar mlookup*
and *tα* :: *'t* ⇒ *'x set* **and** *tinvar tempty*
begin
— Mapping indices to abstract indices
definition *ci-α′* **where**
ci-α′ ci i == *case mα ci i of None* ⇒ *None* | *Some s* ⇒ *Some* (*tα s*)

definition *ci-α* == *im-α* ∘ *ci-α′*

definition *ci-invar* **where**
ci-invar ci == *minvar ci* ∧ (∀ *i s*. *mα ci i* = *Some s* ⟶ *tinvar s*)

```
  lemma ci-impl-minvar: ci-invar m ⟹ minvar m ⟨proof⟩

  definition is-index :: ('x ⇒ 'i) ⇒ 'x set ⇒ 'm ⇒ bool
  where
    is-index f s idx == ci-invar idx ∧ ci-α' idx = index-map f s

  lemma is-index-invar: is-index f s idx ⟹ ci-invar idx
    ⟨proof⟩

  lemma is-index-correct: is-index f s idx ⟹ ci-α idx = index f s
    ⟨proof⟩

  abbreviation lookup == idx-lookup mlookup tempty

  lemma lookup-invar': ci-invar m ⟹ tinvar (lookup i m)
    ⟨proof⟩

  lemma lookup-correct:
    assumes I[simp, intro!]: is-index f s idx
    shows
      tα (lookup i idx) = index f s i
      tinvar (lookup i idx)
  ⟨proof⟩
end


  — Building indices
definition idx-build-stepfun ::
  ('i ⇒ 'm ⇀ 't) ⇒
  ('i ⇒ 't ⇒ 'm ⇒ 'm) ⇒
  (unit ⇒ 't) ⇒
  ('x ⇒ 't ⇒ 't) ⇒
  ('x ⇒ 'i) ⇒ 'x ⇒ 'm ⇒ 'm where
  idx-build-stepfun mlookup mupdate tempty tinsert f x m ==
    let i=f x in
      (case mlookup i m of
        None ⇒ mupdate i (tinsert x (tempty ())) m |
        Some s ⇒ mupdate i (tinsert x s) m
    )

definition idx-build ::
  (unit ⇒ 'm) ⇒
  ('i ⇒ 'm ⇀ 't) ⇒
  ('i ⇒ 't ⇒ 'm ⇒ 'm) ⇒
  (unit ⇒ 't) ⇒
  ('x ⇒ 't ⇒ 't) ⇒
  ('s ⇒ ('x,'m) set-iterator) ⇒
  ('x ⇒ 'i) ⇒ 's ⇒ 'm where
```

idx-build mempty mlookup mupdate tempty tinsert siterate f s
== siterate s (λ-. True)
(idx-build-stepfun mlookup mupdate tempty tinsert f)
(mempty ())

locale *idx-build-env* =
index-env mα minvar mlookup tα tinvar tempty +
m: *map-empty mα minvar mempty* +
m: *map-update mα minvar mupdate* +
t: *set-ins tα tinvar tinsert* +
s: *set-iteratei sα sinvar siterate*
for *mα* :: *'m* ⇒ *'i* ⇀ *'t* **and** *minvar mempty mlookup mupdate*
and *tα* :: *'t* ⇒ *'x set* **and** *tinvar tempty tinsert*
and *sα* :: *'s* ⇒ *'x set* **and** *sinvar*
and *siterate* :: *'s* ⇒ (*'x*,*'m*) *set-iterator*
begin

abbreviation *idx-build-stepfun'* == *idx-build-stepfun mlookup mupdate tempty tinsert*
abbreviation *idx-build'* ==
idx-build mempty mlookup mupdate tempty tinsert siterate

lemma *idx-build-correct'*:
assumes *I*: *sinvar s*
shows *ci-α' (idx-build' f s)* = *index-map f (sα s)* (**is** *?T1*) **and**
[*simp*]: *ci-invar (idx-build' f s)* (**is** *?T2*)
⟨*proof*⟩

lemma *idx-build-correct*:
sinvar s ⟹ *is-index f (sα s) (idx-build' f s)*
⟨*proof*⟩

end

Exported Correctness Lemmas and Definitions

In order to allow simpler use, we make the correctness lemmas visible outside the locale. We also export the *index-env.is-index* predicate.

definition *idx-invar*
:: (*'m* ⇒ *'k* ⇀ *'t*) ⇒ (*'m* ⇒ *bool*)
⇒ (*'t* ⇒ *'v set*) ⇒ (*'t* ⇒ *bool*)
⇒ (*'v* ⇒ *'k*) ⇒ (*'v set*) ⇒ *'m* ⇒ *bool*
where
idx-invar mα minvar tα tinvar == *index-env.is-index mα minvar tα tinvar*

lemma *idx-build-correct*:
assumes *map-empty mα minvar mempty*
assumes *map-lookup mα minvar mlookup*

assumes *map-update mα minvar mupdate*
assumes *set-empty tα tinvar tempty*
assumes *set-ins tα tinvar tinsert*
assumes *set-iteratei sα sinvar siterate*

constrains $m\alpha :: 'm \Rightarrow 'i \rightharpoonup 't$
constrains $t\alpha :: 't \Rightarrow 'x\ set$
constrains $s\alpha :: 's \Rightarrow 'x\ set$
constrains *siterate* :: $'s \Rightarrow ('x,'m)$ *set-iterator*

assumes *INV*: *sinvar S*
shows *idx-invar mα minvar tα tinvar f* (*sα S*) (*idx-build mempty mlookup mupdate tempty tinsert siterate f S*)
⟨*proof*⟩

lemma *idx-lookup-correct*:
assumes *map-lookup mα minvar mlookup*
assumes *set-empty tα tinvar tempty*
assumes *INV*: *idx-invar mα minvar tα tinvar f S idx*
shows *tα* (*idx-lookup mlookup tempty k idx*) = *index f S k* (**is** *?T1*)
tinvar (*idx-lookup mlookup tempty k idx*) (**is** *?T2*)
⟨*proof*⟩

end

3.6 More Generic Algorithms

theory *Algos*
imports
../common/Misc
../spec/SetSpec
../spec/MapSpec
../spec/ListSpec
begin

3.6.1 Injective Map to Naturals

— Compute an injective map from objects into an initial segment of the natural numbers
definition *map-to-nat*
:: $('s \Rightarrow ('x,nat \times 'm)$ *set-iterator*$) \Rightarrow$
$(unit \Rightarrow 'm) \Rightarrow ('x \Rightarrow nat \Rightarrow 'm \Rightarrow 'm) \Rightarrow$
$'s \Rightarrow 'm$ **where**
map-to-nat iterate1 empty2 update2 s ==
snd (*iterate1 s* (λ-. *True*) (λ*x* (*c*,*m*). (*c*+*1*,*update2 x c m*)) (*0*,*empty2* ()))

— Whether a set is an initial segment of the natural numbers
definition *inatseg* :: *nat set* ⇒ *bool*

where *inatseg s == ∃k. s = {i::nat. i<k}*

lemma *inatseg-simps[simp]*:
inatseg {}
inatseg {0}
⟨*proof*⟩

lemma *map-to-nat-correct*:
fixes *α1 :: 's ⇒ 'x set*
fixes *α2 :: 'm ⇒ 'x ⇀ nat*
assumes *set-iteratei α1 invar1 iterate1*
assumes *map-empty α2 invar2 empty2*
assumes *map-update α2 invar2 update2*

assumes *INV[simp]*: *invar1 s*

defines *nm == map-to-nat iterate1 empty2 update2 s*

shows
— All elements have got a number
dom (α2 nm) = α1 s (**is** *?T1*) **and**
— No two elements got the same number
[*rule-format*]: *inj-on (α2 nm) (α1 s)* (**is** *?T2*) **and**
— Numbering is inatseg
[*rule-format*]: *inatseg (ran (α2 nm))* (**is** *?T3*) **and**
— The result satisfies the map invariant
invar2 nm (**is** *?T4*)
⟨*proof*⟩

3.6.2 Set to List(-interface)

Converting Set to List by Enqueue

definition *it-set-to-List-enq :: ('s ⇒ ('a,'f) set-iterator) ⇒*
(unit ⇒ 'f) ⇒ ('a⇒'f⇒'f) ⇒ 's ⇒ 'f
where *it-set-to-List-enq iterate emp enq S == iterate S (λ-. True) (λx F. enq x F) (emp ())*

lemma *it-set-to-List-enq-correct*:
assumes *set-iteratei α invar iterate*
assumes *list-empty αl invarl emp*
assumes *list-enqueue αl invarl enq*
assumes [*simp*]: *invar S*
shows
set (αl (it-set-to-List-enq iterate emp enq S)) = α S (**is** *?T1*)
invarl (it-set-to-List-enq iterate emp enq S) (**is** *?T2*)
distinct (αl (it-set-to-List-enq iterate emp enq S)) (**is** *?T3*)
⟨*proof*⟩

Converting Set to List by Push

definition *it-set-to-List-push* :: (*'s* ⇒ (*'a*,*'f*) *set-iterator*) ⇒
(*unit* ⇒ *'f*) ⇒ (*'a*⇒*'f*⇒*'f*) ⇒ *'s* ⇒ *'f*
where *it-set-to-List-push iterate emp push S* == *iterate S* (λ-. *True*) (λ*x F*. *push x F*) (*emp* ())

lemma *it-set-to-List-push-correct*:
assumes *set-iteratei* α *invar iterate*
assumes *list-empty* α*l invarl emp*
assumes *list-push* α*l invarl push*
assumes [*simp*]: *invar S*
shows
set (α*l* (*it-set-to-List-push iterate emp push S*)) = α *S* (**is** *?T1*)
invarl (*it-set-to-List-push iterate emp push S*) (**is** *?T2*)
distinct (α*l* (*it-set-to-List-push iterate emp push S*)) (**is** *?T3*)
⟨*proof*⟩

3.6.3 Map from Set

— Build a map using a set of keys and a function to compute the values.

definition *it-dom-fun-to-map* ::
(*'s* ⇒ (*'k*,*'m*) *set-iterator*) ⇒
(*'k* ⇒ *'v* ⇒ *'m* ⇒ *'m*) ⇒ (*unit* ⇒ *'m*) ⇒ *'s* ⇒ (*'k* ⇒ *'v*) ⇒ *'m*
where *it-dom-fun-to-map s-it up-dj emp s f* ==
s-it s (λ-. *True*) (λ*k m*. *up-dj k* (*f k*) *m*) (*emp* ())

lemma *it-dom-fun-to-map-correct*:
fixes α*1* :: *'m* ⇒ *'k* ⇒ *'v option*
fixes α*2* :: *'s* ⇒ *'k set*
assumes *s-it*: *set-iteratei* α*2 invar2 s-it*
assumes *m-up*: *map-update-dj* α*1 invar1 up-dj*
assumes *m-emp*: *map-empty* α*1 invar1 emp*
assumes *INV*: *invar2 s*
shows α*1* (*it-dom-fun-to-map s-it up-dj emp s f*) *k* = (*if k* ∈ α*2 s then Some* (*f k*) *else None*)
and *invar1* (*it-dom-fun-to-map s-it up-dj emp s f*)
⟨*proof*⟩

end

3.7 Implementing Priority Queues by Annotated Lists

theory *PrioByAnnotatedList*
imports
../spec/AnnotatedListSpec

../spec/PrioSpec
begin

In this theory, we implement priority queues by annotated lists.

The implementation is realized as a generic adapter from the AnnotatedList to the priority queue interface.

Priority queues are realized as a sequence of pairs of elements and associated priority. The monoids operation takes the element with minimum priority.

The element with minimum priority is extracted from the sum over all elements. Deleting the element with minimum priority is done by splitting the sequence at the point where the minimum priority of the elements read so far becomes equal to the minimum priority of all elements.

3.7.1 Definitions

Monoid

datatype $('e, 'a)$ *Prio* = *Infty* | *Prio* $'e$ $'a$

fun *p-unwrap* :: $('e,'a)$ *Prio* ⇒ $('e \times 'a)$ **where**
p-unwrap (*Prio e a*) = (*e* , *a*)

fun *p-min* :: $('e, 'a::linorder)$ *Prio* ⇒ $('e, 'a)$ *Prio* ⇒ $('e, 'a)$ *Prio* **where**
p-min Infty Infty = *Infty*|
p-min Infty (*Prio e a*) = *Prio e a*|
p-min (*Prio e a*) *Infty* = *Prio e a*|
p-min (*Prio e1 a*) (*Prio e2 b*) = (*if a* ≤ *b then Prio e1 a else Prio e2 b*)

lemma *p-min-re-neut*[*simp*]: *p-min a Infty* = *a* ⟨*proof*⟩
lemma *p-min-le-neut*[*simp*]: *p-min Infty a* = *a* ⟨*proof*⟩
lemma *p-min-asso*: *p-min* (*p-min a b*) *c* = *p-min a* (*p-min b c*)
⟨*proof*⟩
lemma *lp-mono*: *class.monoid-add p-min Infty*
⟨*proof*⟩

instantiation *Prio* :: (*type*,*linorder*) *monoid-add*
begin
definition *zero-def*: *0* == *Infty*
definition *plus-def*: *a*+*b* == *p-min a b*

instance ⟨*proof*⟩
end

fun *p-less-eq* :: $('e, 'a::linorder)$ *Prio* ⇒ $('e, 'a)$ *Prio* ⇒ *bool* **where**
p-less-eq (*Prio e a*) (*Prio f b*) = (*a* ≤ *b*)|
p-less-eq - *Infty* = *True*|
p-less-eq Infty (*Prio e a*) = *False*

fun *p-less* :: (*'e*, *'a*::*linorder*) *Prio* ⇒ (*'e*, *'a*) *Prio* ⇒ *bool* **where**
p-less (*Prio e a*) (*Prio f b*) = (*a* < *b*)|
p-less (*Prio e a*) *Infty* = *True*|
p-less Infty - = *False*

lemma *p-less-le-not-le* : *p-less x y* ⟷ *p-less-eq x y* ∧ ¬ (*p-less-eq y x*)
⟨*proof*⟩

lemma *p-order-refl* : *p-less-eq x x*
⟨*proof*⟩

lemma *p-le-inf* : *p-less-eq Infty x* ⟹ *x* = *Infty*
⟨*proof*⟩

lemma *p-order-trans* : ⟦*p-less-eq x y*; *p-less-eq y z*⟧ ⟹ *p-less-eq x z*
⟨*proof*⟩

lemma *p-linear2* : *p-less-eq x y* ∨ *p-less-eq y x*
⟨*proof*⟩

instantiation *Prio* :: (*type*, *linorder*) *preorder*
begin
definition *plesseq-def*: *less-eq* = *p-less-eq*
definition *pless-def*: *less* = *p-less*

instance
⟨*proof*⟩

end

Operations

definition *alprio-α* :: (*'s* ⇒ (*unit* × (*'e*,*'a*::*linorder*) *Prio*) *list*)
⇒ *'s* ⇒ (*'e* × *'a*::*linorder*) *multiset*
where
alprio-α α al == (*multiset-of* (*map p-unwrap* (*map snd* (*α al*))))

definition *alprio-invar* :: (*'s* ⇒ (*unit* × (*'c*, *'d*::*linorder*) *Prio*) *list*)
⇒ (*'s* ⇒ *bool*) ⇒ *'s* ⇒ *bool*
where
alprio-invar α invar al == *invar al* ∧ (∀ *x*∈*set* (*α al*). *snd x*≠*Infty*)

definition *alprio-empty* **where**
alprio-empty empt = *empt*

definition *alprio-isEmpty* **where**
alprio-isEmpty isEmpty = *isEmpty*

definition *alprio-insert* :: (*unit* ⇒ ('*e*,'*a*) *Prio* ⇒ '*s* ⇒ '*s*)
⇒ '*e* ⇒ '*a*::*linorder* ⇒ '*s* ⇒ '*s*
where
alprio-insert consl e a s = *consl* () (*Prio e a*) *s*

definition *alprio-find* :: ('*s* ⇒ ('*e*,'*a*::*linorder*) *Prio*) ⇒ '*s* ⇒ ('*e* × '*a*)
where
alprio-find annot s = *p-unwrap* (*annot s*)

definition *alprio-delete* :: ((('*e*,'*a*::*linorder*) *Prio* ⇒ *bool*)
⇒ ('*e*,'*a*) *Prio* ⇒ '*s* ⇒ ('*s* × (*unit* × ('*e*,'*a*) *Prio*) × '*s*))
⇒ ('*s* ⇒ ('*e*,'*a*) *Prio*) ⇒ ('*s* ⇒ '*s* ⇒ '*s*) ⇒ '*s* ⇒ '*s*
where
alprio-delete splits annot app s = (*let* (*l*, - , *r*)
= *splits* (λ *x*. *x*≤(*annot s*)) *Infty s in app l r*)

definition *alprio-meld* **where**
alprio-meld app = *app*

lemmas *alprio-defs* =
alprio-invar-def
alprio-α-def
alprio-empty-def
alprio-isEmpty-def
alprio-insert-def
alprio-find-def
alprio-delete-def
alprio-meld-def

3.7.2 Correctness

Auxiliary Lemmas

lemma *listsum-split*: *listsum* (*l* @ (*a*::'*a*::*monoid-add*) # *r*) = (*listsum l*) + *a* + (*listsum r*)
⟨*proof*⟩

lemma *p-linear*: (*x*::('*e*, '*a*::*linorder*) *Prio*) ≤ *y* ∨ *y* ≤ *x*
⟨*proof*⟩

lemma *p-min-mon*: (*x*::(('*e*,'*a*::*linorder*) *Prio*)) ≤ *y* ⟹ (*z* + *x*) ≤ *y*
⟨*proof*⟩

lemma *p-min-mon2*: *p-less-eq x y* ⟹ *p-less-eq* (*p-min z x*) *y*
⟨*proof*⟩

lemma *ls-min*: ∀ *x* ∈ *set* (*xs*:: ('*e*,'*a*::*linorder*) *Prio list*) . *listsum xs* ≤ *x*
⟨*proof*⟩

lemma *infadd*: $x \neq Infty \Longrightarrow x + y \neq Infty$
⟨*proof*⟩

lemma *prio-selects-one*: $a{+}b = a \vee a{+}b{=}(b{::}('e,'a{::}linorder)\ Prio)$
⟨*proof*⟩

lemma *listsum-in-set*: $(l{::}('x \times ('e,'a{::}linorder)\ Prio)\ list){\neq}[] \Longrightarrow$
listsum (*map snd l*) ∈ *set* (*map snd l*)
⟨*proof*⟩

lemma *p-unwrap-less-sum*: *snd* (*p-unwrap* ((*Prio e aa*) + *b*)) ≤ *aa*
⟨*proof*⟩

lemma *prio-add-alb*: ¬ $b \leq (a{::}('e,'a{::}linorder)Prio) \Longrightarrow b + a = a$
⟨*proof*⟩

lemma *prio-add-alb2*: $(a{::}('e,'a{::}linorder)Prio) \leq a + b \Longrightarrow a + b = a$
⟨*proof*⟩

lemma *prio-add-abc*:
assumes $(l{::}('e,'a{::}linorder)Prio) + a \leq c$
and ¬ $l \leq c$
shows ¬ $l \leq a$
⟨*proof*⟩

lemma *prio-add-abc2*:
assumes $(a{::}('e,'a{::}linorder)Prio) \leq a + b$
shows $a \leq b$
⟨*proof*⟩

Empty

lemma *alprio-empty-correct*:
assumes *al-empty* α *invar empt*
shows *prio-empty* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-empty empt*)
⟨*proof*⟩

Is Empty

lemma *alprio-isEmpty-correct*:
assumes *al-isEmpty* α *invar isEmpty*
shows *prio-isEmpty* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-isEmpty isEmpty*)
⟨*proof*⟩

Insert

lemma *alprio-insert-correct*:

assumes *al-consl* α *invar consl*
shows *prio-insert* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-insert consl*)
⟨*proof*⟩

Meld

lemma *alprio-meld-correct*:
assumes *al-app* α *invar app*
shows *prio-meld* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-meld app*)
⟨*proof*⟩

Find

lemma *annot-not-inf* :
assumes (*alprio-invar* α *invar*) s
and (*alprio-*α α) $s \neq \{\#\}$
and *al-annot* α *invar annot*
shows *annot* $s \neq$ *Infty*
⟨*proof*⟩

lemma *annot-in-set*:
assumes (*alprio-invar* α *invar*) s
and (*alprio-*α α) $s \neq \{\#\}$
and *al-annot* α *invar annot*
shows *p-unwrap* (*annot* s) $\in\#$ ((*alprio-*α α) s)
⟨*proof*⟩

lemma *listsum-less-elems*: $\forall\, x{\in}$*set xs*. *snd* $x \neq$ *Infty* $\Longrightarrow$
$\forall\, y{\in}$*set-of* (*multiset-of* (*map p-unwrap* (*map snd xs*))).
snd (*p-unwrap* (*listsum* (*map snd xs*))) $\leq$ *snd* y
⟨*proof*⟩

lemma *alprio-find-correct*:
assumes *al-annot* α *invar annot*
shows *prio-find* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-find annot*)
⟨*proof*⟩

Delete

lemma *delpred-mon*:
$\forall\,(a{::}\ ('e,\ 'a{::}linorder)\ Prio)\ b.\ ((\lambda\ x.\ x \leq y)\ a$
$\longrightarrow (\lambda\ x.\ x \leq y)\ (a + b))$
⟨*proof*⟩

lemma *alpriodel-invar*:
assumes *alprio-invar* α *invar* s
and *al-annot* α *invar annot*
and *alprio-*α α $s \neq \{\#\}$
and *al-splits* α *invar splits*

and *al-app* α *invar app*
shows *alprio-invar* α *invar* (*alprio-delete splits annot app s*)
⟨*proof*⟩

lemma *listsum-elem*:
assumes $ins = l$ @ $(a::('e,'a::linorder)Prio)$ # r
and ¬ *listsum l* ≤ *listsum ins*
and *listsum l* + *a* ≤ *listsum ins*
shows a = *listsum ins*
⟨*proof*⟩

lemma *alpriodel-right*:
assumes *alprio-invar* α *invar s*
and *al-annot* α *invar annot*
and *alprio-*α α *s* ≠ {#}
and *al-splits* α *invar splits*
and *al-app* α *invar app*
shows *alprio-*α α (*alprio-delete splits annot app s*) =
alprio-α α *s* − {#*p-unwrap* (*annot s*)#}
⟨*proof*⟩

lemma *alprio-delete-correct*:
assumes *al-annot* α *invar annot*
and *al-splits* α *invar splits*
and *al-app* α *invar app*
shows *prio-delete* (*alprio-*α α) (*alprio-invar* α *invar*)
(*alprio-find annot*) (*alprio-delete splits annot app*)
⟨*proof*⟩

lemmas *alprio-correct* =
alprio-empty-correct
alprio-isEmpty-correct
alprio-insert-correct
alprio-delete-correct
alprio-find-correct
alprio-meld-correct

end

3.8 Implementing Unique Priority Queues by Annotated Lists

theory *PrioUniqueByAnnotatedList*
imports
../spec/AnnotatedListSpec
../spec/PrioUniqueSpec

begin

In this theory we use annotated lists to implement unique priority queues with totally ordered elements.

This theory is written as a generic adapter from the AnnotatedList interface to the unique priority queue interface.

The annotated list stores a sequence of elements annotated with priorities[1]

The monoids operations forms the maximum over the elements and the minimum over the priorities. The sequence of pairs is ordered by ascending elements' order. The insertion point for a new element, or the priority of an existing element can be found by splitting the sequence at the point where the maximum of the elements read so far gets bigger than the element to be inserted.

The minimum priority can be read out as the sum over the whole sequence. Finding the element with minimum priority is done by splitting the sequence at the point where the minimum priority of the elements read so far becomes equal to the minimum priority of the whole sequence.

3.8.1 Definitions

Monoid

datatype $('e, 'a)\ LP = Infty \mid LP\ 'e\ 'a$

fun *p-unwrap* :: $('e,'a)\ LP \Rightarrow ('e \times 'a)$ **where**
  p-unwrap $(LP\ e\ a) = (e\ ,\ a)$

fun *p-min* :: $('e::linorder, 'a::linorder)\ LP \Rightarrow ('e, 'a)\ LP \Rightarrow ('e, 'a)\ LP$ **where**
  p-min Infty Infty $= Infty|$
  p-min Infty $(LP\ e\ a) = LP\ e\ a|$
  p-min $(LP\ e\ a)$ *Infty* $= LP\ e\ a|$
  p-min $(LP\ e1\ a)\ (LP\ e2\ b) = (LP\ (max\ e1\ e2)\ (min\ a\ b))$

fun *e-less-eq* :: $'e \Rightarrow ('e::linorder, 'a::linorder)\ LP \Rightarrow bool$ **where**
  e-less-eq e Infty $= False|$
  e-less-eq e $(LP\ e'\ \text{-}) = (e \leq e')$

Instantiation of classes

lemma *p-min-re-neut*[*simp*]: *p-min a Infty* $= a$ ⟨*proof*⟩
lemma *p-min-le-neut*[*simp*]: *p-min Infty a* $= a$ ⟨*proof*⟩
lemma *p-min-asso*: *p-min* $(p\text{-}min\ a\ b)\ c =$ *p-min a* $(p\text{-}min\ b\ c)$
  ⟨*proof*⟩

[1] Technically, the annotated list elements are of unit-type, and the annotations hold both, the priority queue elements and the priorities. This is required as we defined annotated lists to only sum up the elements annotations.

lemma *lp-mono*: *class.monoid-add p-min Infty* ⟨*proof*⟩

instantiation *LP* :: (*linorder*,*linorder*) *monoid-add*
begin
definition *zero-def*: *0* == *Infty*
definition *plus-def*: *a*+*b* == *p-min a b*

instance ⟨*proof*⟩
end

fun *p-less-eq* :: ('*e*, '*a*::*linorder*) *LP* ⇒ ('*e*, '*a*) *LP* ⇒ *bool* **where**
p-less-eq (*LP e a*) (*LP f b*) = (*a* ≤ *b*)|
p-less-eq - *Infty* = *True*|
p-less-eq Infty (*LP e a*) = *False*

fun *p-less* :: ('*e*, '*a*::*linorder*) *LP* ⇒ ('*e*, '*a*) *LP* ⇒ *bool* **where**
p-less (*LP e a*) (*LP f b*) = (*a* < *b*)|
p-less (*LP e a*) *Infty* = *True*|
p-less Infty - = *False*

lemma *p-less-le-not-le* : *p-less x y* ⟷ *p-less-eq x y* ∧ ¬ (*p-less-eq y x*)
⟨*proof*⟩

lemma *p-order-refl* : *p-less-eq x x*
⟨*proof*⟩

lemma *p-le-inf* : *p-less-eq Infty x* ⟹ *x* = *Infty*
⟨*proof*⟩

lemma *p-order-trans* : ⟦*p-less-eq x y*; *p-less-eq y z*⟧ ⟹ *p-less-eq x z*
⟨*proof*⟩

lemma *p-linear2* : *p-less-eq x y* ∨ *p-less-eq y x*
⟨*proof*⟩

instantiation *LP* :: (*type*, *linorder*) *preorder*
begin
definition *plesseq-def*: *less-eq* = *p-less-eq*
definition *pless-def*: *less* = *p-less*

instance
⟨*proof*⟩

end

Operations

definition *aluprio-α* :: ('*s* ⇒ (*unit* × ('*e*::*linorder*,'*a*::*linorder*) *LP*) *list*)
⇒ '*s* ⇒ ('*e*::*linorder* ⇀ '*a*::*linorder*)

where
aluprio-α α ft == (*map-of* (*map p-unwrap* (*map snd* (*α ft*))))

definition *aluprio-invar* :: ('*s* ⇒ (*unit* × ('*c*::*linorder*, '*d*::*linorder*) *LP*) *list*)
⇒ ('*s* ⇒ *bool*) ⇒ '*s* ⇒ *bool*
where
aluprio-invar α invar ft ==
invar ft
∧ (∀ *x*∈*set* (*α ft*). *snd x*≠*Infty*)
∧ *sorted* (*map fst* (*map p-unwrap* (*map snd* (*α ft*))))
∧ *distinct* (*map fst* (*map p-unwrap* (*map snd* (*α ft*))))

definition *aluprio-empty* **where**
aluprio-empty empt = *empt*

definition *aluprio-isEmpty* **where**
aluprio-isEmpty isEmpty = *isEmpty*

definition *aluprio-insert* ::
(('*e*::*linorder*,'*a*::*linorder*) *LP* ⇒ *bool*)
⇒ ('*e*,'*a*) *LP* ⇒ '*s* ⇒ ('*s* × (*unit* × ('*e*,'*a*) *LP*) × '*s*))
⇒ ('*s* ⇒ ('*e*,'*a*) *LP*)
⇒ ('*s* ⇒ *bool*)
⇒ ('*s* ⇒ '*s* ⇒ '*s*)
⇒ ('*s* ⇒ *unit* ⇒ ('*e*,'*a*) *LP* ⇒ '*s*)
⇒ '*s* ⇒ '*e* ⇒ '*a* ⇒ '*s*
where

aluprio-insert splits annot isEmpty app consr s e a =
(*if e-less-eq e* (*annot s*) ∧ ¬ *isEmpty s*
then
(*let* (*l*, (-,*lp*) , *r*) = *splits* (*e-less-eq e*) *Infty s in*
(*if e* < *fst* (*p-unwrap lp*)
then
app (*consr* (*consr l* () (*LP e a*)) () *lp*) *r*
else
app (*consr l* () (*LP e a*)) *r*))
else
consr s () (*LP e a*))

definition *aluprio-pop* :: ((('*e*::*linorder*,'*a*::*linorder*) *LP* ⇒ *bool*) ⇒ ('*e*,'*a*) *LP*
⇒ '*s* ⇒ ('*s* × (*unit* × ('*e*,'*a*) *LP*) × '*s*))
⇒ ('*s* ⇒ ('*e*,'*a*) *LP*)
⇒ ('*s* ⇒ '*s* ⇒ '*s*)
⇒ '*s*
⇒ '*e* ×'*a* ×'*s*
where
aluprio-pop splits annot app s =

```
  (let (l, (-,lp) , r) = splits (λ x. x ≤ (annot s)) Infty s
  in
    (case lp of
      (LP e a) ⇒
        (e, a, app l r) ))

definition aluprio-prio ::
 ((('e::linorder,'a::linorder) LP ⇒ bool) ⇒ ('e,'a) LP ⇒ 's
 ⇒ ('s × (unit × ('e,'a) LP) × 's))
  ⇒ ('s ⇒ ('e,'a) LP)
    ⇒ ('s ⇒ bool)
      ⇒ 's ⇒ 'e ⇒ 'a option
 where

 aluprio-prio splits annot isEmpty s e =
  (if e-less-eq e (annot s) ∧ ¬ isEmpty s
  then
    (let (l, (-,lp) , r) = splits (e-less-eq e) Infty s in
      (if e = fst (p-unwrap lp)
      then
        Some (snd (p-unwrap lp))
      else
        None))
  else
    None)


lemmas aluprio-defs =
aluprio-invar-def
aluprio-α-def
aluprio-empty-def
aluprio-isEmpty-def
aluprio-insert-def
aluprio-pop-def
aluprio-prio-def
```

3.8.2 Correctness

Auxiliary Lemmas

```
lemma p-linear: (x::('e, 'a::linorder) LP) ≤ y ∨ y ≤ x
  ⟨proof⟩


lemma e-less-eq-mon1: e-less-eq e x ⟹ e-less-eq e (x + y)
  ⟨proof⟩
lemma e-less-eq-mon2: e-less-eq e y ⟹ e-less-eq e (x + y)
  ⟨proof⟩
lemmas e-less-eq-mon =
  e-less-eq-mon1
```

e-less-eq-mon2

lemma *p-less-eq-mon*:
$(x::('e::linorder,'a::linorder)\ LP) \leq z \Longrightarrow (x + y) \leq z$
⟨*proof*⟩

lemma *p-less-eq-lem1*:
$[\![\neg\ (x::('e::linorder,'a::linorder)\ LP) \leq z;$
$(x + y) \leq z]\!]$
$\Longrightarrow y \leq z$
⟨*proof*⟩

lemma *infadd*: $x \neq Infty \Longrightarrow x + y \neq Infty$
⟨*proof*⟩

lemma *e-less-eq-listsum*:
$[\![\neg\ e\text{-}less\text{-}eq\ e\ (listsum\ xs)]\!] \Longrightarrow \forall x \in set\ xs.\ \neg\ e\text{-}less\text{-}eq\ e\ x$
⟨*proof*⟩

lemma *e-less-eq-p-unwrap*:
$[\![x \neq Infty;\neg\ e\text{-}less\text{-}eq\ e\ x]\!] \Longrightarrow fst\ (p\text{-}unwrap\ x) < e$
⟨*proof*⟩

lemma *e-less-eq-refl* :
$b \neq Infty \Longrightarrow e\text{-}less\text{-}eq\ (fst\ (p\text{-}unwrap\ b))\ b$
⟨*proof*⟩

lemma *e-less-eq-listsum2*:
assumes
$\forall x{\in}set\ (\alpha s).\ snd\ x \neq Infty$
$((),\ b) \in set\ (\alpha s)$
shows $e\text{-}less\text{-}eq\ (fst\ (p\text{-}unwrap\ b))\ (listsum\ (map\ snd\ (\alpha s)))$
⟨*proof*⟩

lemma *e-less-eq-lem1*:
$[\![\neg\ e\text{-}less\text{-}eq\ e\ a;e\text{-}less\text{-}eq\ e\ (a + b)]\!] \Longrightarrow e\text{-}less\text{-}eq\ e\ b$
⟨*proof*⟩

lemma *p-unwrap-less-sum*: $snd\ (p\text{-}unwrap\ ((LP\ e\ aa) + b)) \leq aa$
⟨*proof*⟩

lemma *listsum-less-elems*: $\forall x{\in}set\ xs.\ snd\ x \neq Infty \Longrightarrow$
$\forall y{\in}set\ (map\ snd\ (map\ p\text{-}unwrap\ (map\ snd\ xs))).$
$snd\ (p\text{-}unwrap\ (listsum\ (map\ snd\ xs))) \leq y$
⟨*proof*⟩

lemma *distinct-sortet-list-app*:
$[\![sorted\ xs;\ distinct\ xs;\ xs = as\ @\ b\ \#\ cs]\!]$

$\Longrightarrow \forall\ x{\in}\ set\ cs.\ b < x$
⟨*proof*⟩

lemma *distinct-sorted-list-lem1*:
assumes
sorted xs
sorted ys
distinct xs
distinct ys
$\forall\ x \in set\ xs.\ x < e$
$\forall\ y \in set\ ys.\ e < y$
shows
sorted $(xs\ @\ e\ \#\ ys)$
distinct $(xs\ @\ e\ \#\ ys)$
⟨*proof*⟩

lemma *distinct-sorted-list-lem2*:
assumes
sorted xs
sorted ys
distinct xs
distinct ys
$e < e'$
$\forall\ x \in set\ xs.\ x < e$
$\forall\ y \in set\ ys.\ e' < y$
shows
sorted $(xs\ @\ e\ \#\ e'\ \#\ ys)$
distinct $(xs\ @\ e\ \#\ e'\ \#\ ys)$
⟨*proof*⟩

lemma *map-of-distinct-upd*:
$x \notin set\ (map\ fst\ xs) \Longrightarrow [x \mapsto y] +\!\!+ map\text{-}of\ xs = (map\text{-}of\ xs)\ (x \mapsto y)$
⟨*proof*⟩

lemma *map-of-distinct-upd2*:
assumes $x \notin set(map\ fst\ xs)$
$x \notin set\ (map\ fst\ ys)$
shows $map\text{-}of\ (xs\ @\ (x,y)\ \#\ ys) = (map\text{-}of\ (xs\ @\ ys))(x \mapsto y)$
⟨*proof*⟩

lemma *map-of-distinct-upd3*:
assumes $x \notin set(map\ fst\ xs)$
$x \notin set\ (map\ fst\ ys)$
shows $map\text{-}of\ (xs\ @\ (x,y)\ \#\ ys) = (map\text{-}of\ (xs\ @\ (x,y')\ \#\ ys))(x \mapsto y)$
⟨*proof*⟩

lemma *map-of-distinct-upd4*:
assumes $x \notin set(map\ fst\ xs)$
$x \notin set\ (map\ fst\ ys)$

shows *map-of* (*xs* @ *ys*) = (*map-of* (*xs* @ (*x*,*y*) # *ys*))(*x* := *None*)
⟨*proof*⟩

lemma *map-of-distinct-lookup*:
assumes *x* ∉ *set*(*map fst xs*)
x ∉ *set* (*map fst ys*)
shows *map-of* (*xs* @ (*x*,*y*) # *ys*) *x* = *Some y*
⟨*proof*⟩

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ' *set al*
⟨*proof*⟩

Finite

lemma *aluprio-finite-correct*: *uprio-finite* (*aluprio-α α*) (*aluprio-invar α invar*)
⟨*proof*⟩

Empty

lemma *aluprio-empty-correct*:
assumes *al-empty α invar empt*
shows *uprio-empty* (*aluprio-α α*) (*aluprio-invar α invar*) (*aluprio-empty empt*)
⟨*proof*⟩

Is Empty

lemma *aluprio-isEmpty-correct*:
assumes *al-isEmpty α invar isEmpty*
shows *uprio-isEmpty* (*aluprio-α α*) (*aluprio-invar α invar*) (*aluprio-isEmpty isEmpty*)
⟨*proof*⟩

Insert

lemma *annot-inf*:
assumes *A*: *invar s* ∀ *x*∈*set* (*α s*). *snd x* ≠ *Infty* *al-annot α invar annot*
shows *annot s* = *Infty* ⟷ *α s* = []
⟨*proof*⟩

lemma *e-less-eq-annot*:

assumes *al-annot α invar annot*
invar s ∀ *x*∈*set* (*α s*). *snd x* ≠ *Infty* ¬ *e-less-eq e* (*annot s*)
shows ∀ *x* ∈ *set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (*α s*)). *x* < *e*
⟨*proof*⟩

lemma *aluprio-insert-correct*:
assumes

al-splits α *invar splits*
al-annot α *invar annot*
al-isEmpty α *invar isEmpty*
al-app α *invar app*
al-consr α *invar consr*
shows
uprio-insert (*aluprio-*α α) (*aluprio-invar* α *invar*)
(*aluprio-insert splits annot isEmpty app consr*)
⟨*proof*⟩

Prio

lemma *aluprio-prio-correct*:
assumes
al-splits α *invar splits*
al-annot α *invar annot*
al-isEmpty α *invar isEmpty*
shows
uprio-prio (*aluprio-*α α) (*aluprio-invar* α *invar*) (*aluprio-prio splits annot isEmpty*)
⟨*proof*⟩

Pop

lemma *aluprio-pop-correct*:
assumes *al-splits* α *invar splits*
al-annot α *invar annot*
al-app α *invar app*
shows
uprio-pop (*aluprio-*α α) (*aluprio-invar* α *invar*) (*aluprio-pop splits annot app*)
⟨*proof*⟩

lemmas *aluprio-correct* =
aluprio-finite-correct
aluprio-empty-correct
aluprio-isEmpty-correct
aluprio-insert-correct
aluprio-pop-correct
aluprio-prio-correct

end

Chapter 4

Implementations

4.1 Overview of Interfaces and Implementations

theory *Impl-Overview*
imports *Main*
begin

Interface *AnnotatedList* (theory *AnnotatedListSpec*)
Abstract type: $('e \times 'a{::}monoid\text{-}add)\ list$
Lists with annotated elements. The annotations form a monoid, and there is a split operation to split the list according to its annotations. This is the abstract concept implemented by finger trees.

Implementations:

FTAnnotatedListImpl (Type: $('a,'b{::}monoid\text{-}add)\ ft$) (Abbrv: *ft*) Annotated lists implemented by 2-3 finger trees.

Interface *List* (theory *ListSpec*)
Abstract type: $'a\ list$
This interface specifies sequences.

Implementations:

Fifo (Type: $'a\ fifo$) (Abbrv: *fifo*) Fifo-Queues implemented by two stacks.

Interface *Map* (theory *MapSpec*)
Abstract type: $'k \rightharpoonup 'v$
This interface specifies maps from keys to values.

Implementations:

ArrayHashMap (Type: $('k,'v)\ ahm$) (Abbrv: *ahm*,*a*) Array based hash maps without explicit invariant.

ArrayMapImpl (Type: $'v\ iam$) (Abbrv: *iam*,*im*) Maps of natural numbers implemented by arrays.

HashMap (Type: $'a$::*hashable hm*) (Abbrv: *hm*,*h*) Hash maps based on red-black trees.

ListMapImpl (Type: $'a$ *lm*) (Abbrv: *lm*,*l*) Maps implemented by associative lists. If you need efficient *insert-dj* operation, you should use list sets with explicit invariants (lmi).

ListMapImpl-Invar (Type: $'a$ *lmi*) (Abbrv: *lmi*,*l*) Maps implemented by associative lists. Version with explicit invariants that allows for efficient xxx-dj operations.

RBTMapImpl (Type: ($'k$::*linorder*,$'v$) *rm*) (Abbrv: *rm*,*r*) Maps over linearly ordered keys implemented by red-black trees.

TrieMapImpl (Type: ($'k$,$'v$) *tm*) (Abbrv: *tm*,*t*) Maps over keys of type $'k$ *list* implemented by tries.

Interface *Prio* (theory *PrioSpec*)
Abstract type: ($'e \times 'a$::*linorder*) *multiset*
Priority queues that may contain duplicate elements.
Implementations:

BinoPrioImpl (Type: ($'a$,$'p$::*linorder*) *bino*) (Abbrv: *bino*) Binomial heaps.

FTPrioImpl (Type: ($'a$,$'p$::*linorder*) *alprioi*) (Abbrv: *alprioi*) Priority queues based on 2-3 finger trees.

SkewPrioImpl (Type: ($'a$,$'p$::*linorder*) *skew*) (Abbrv: *skew*) Priority queues by skew binomial heaps.

Interface *PrioUnique* (theory *PrioUniqueSpec*)
Abstract type: ($'e \rightharpoonup 'a$::*linorder*)
Priority queues without duplicate elements. This interface defines a decrease-key operation.
Implementations:

FTPrioUniqueImpl (Type: ($'a$::*linorder*,$'p$::*linorder*) *aluprioi*) (Abbrv: *aluprioi*) Unique priority queues based on 2-3 finger trees.

Interface *Set* (theory *SetSpec*)
Abstract type: $'a$ *set*
Sets
Implementations:

ArrayHashSet (Type: ($'a$) *ahs*) (Abbrv: *ahs*,*a*) Array based hash sets without explicit invariant.

ArraySetImpl (Type: *ias*) (Abbrv: *ias*,*is*) Sets of natural numbers implemented by arrays.

HashSet (Type: *'a*::*hashable hs*) (Abbrv: *hs*,*h*) Hash sets based on red-black trees.

ListSetImpl (Type: *'a ls*) (Abbrv: *ls*,*l*) Sets implemented by distinct lists. For efficient *insert-dj*-operations, use the version with explicit invariant (lsi).

ListSetImpl-Invar (Type: *'a lsi*) (Abbrv: *lsi*,*l*) Sets by lists with distinct elements. Version with explicit invariant that supports efficient *insert-dj* operation.

ListSetImpl-NotDist (Type: *'a lsnd*) (Abbrv: *lsnd*) Sets implemented by lists that may contain duplicate elements. Insertion is quick, but other operations are less performant than on lists with distinct elements.

ListSetImpl-Sorted (Type: *'a*::*linorder lss*) (Abbrv: *lss*) Sets implemented by sorted lists.

RBTSetImpl (Type: (*'a*::*linorder*) *rs*) (Abbrv: *rs*,*r*) Sets over linearly ordered elements implemented by red-black trees.

TrieSetImpl (Type: (*'a*) *ts*) (Abbrv: *ts*,*t*) Sets of elements of type *'a list* implemented by tries.

end

4.2 Map Implementation by Associative Lists

theory *ListMapImpl*
imports
../spec/MapSpec
../common/Assoc-List
../gen-algo/MapGA
begin type-synonym (*'k*,*'v*) *lm* = (*'k*,*'v*) *assoc-list*

4.2.1 Functions

definition *lm-α* == *Assoc-List.lookup*
abbreviation (*input*) *lm-invar* **where** *lm-invar* == λ-. *True*
definition *lm-empty* = (λ-::*unit*. *Assoc-List.empty*)
definition *lm-lookup k m* == *Assoc-List.lookup m k*
definition *lm-update* == *Assoc-List.update*

Since we use the abstract type (*'k*, *'v*) *assoc-list* for associative lists, to preserve the invariant of distinct maps, we cannot just use Cons for disjoint

update, but must resort to ordinary update. The same applies to *lm-add-dj* below.

definition *lm-update-dj* == *lm-update*
definition *lm-delete k m* == *Assoc-List.delete k m*
definition *lm-iteratei* == *Assoc-List.iteratei*

definition *lm-isEmpty m* == *m=Assoc-List.empty*

definition *lm-add* == *it-add lm-update lm-iteratei*
definition *lm-add-dj* == *lm-add*

definition *lm-sel* == *iti-sel lm-iteratei*
definition *lm-sel'* == *iti-sel-no-map lm-iteratei*
definition *lm-to-list* :: ('*u*,'*v*) *lm* ⇒ ('*u* × '*v*) *list*
where *lm-to-list* = *Assoc-List.impl-of*
definition *list-to-lm* == *gen-list-to-map lm-empty lm-update*

4.2.2 Correctness

lemmas *lm-defs* =
lm-α-def
lm-empty-def
lm-lookup-def
lm-update-def
lm-update-dj-def
lm-delete-def
lm-iteratei-def
lm-isEmpty-def
lm-add-def
lm-add-dj-def
lm-sel-def
lm-sel'-def
lm-to-list-def
list-to-lm-def

lemma *lm-empty-impl*:
map-empty lm-α lm-invar lm-empty
⟨*proof*⟩

interpretation *lm*: *map-empty lm-α lm-invar lm-empty* ⟨*proof*⟩

lemma *lm-lookup-impl*:
map-lookup lm-α lm-invar lm-lookup
⟨*proof*⟩

interpretation *lm*: *map-lookup lm-α lm-invar lm-lookup* ⟨*proof*⟩

lemma *lm-update-impl*:
map-update lm-α lm-invar lm-update

⟨*proof*⟩

interpretation *lm*: *map-update lm-α lm-invar lm-update* ⟨*proof*⟩

lemma *lm-update-dj-impl*:
map-update-dj lm-α lm-invar lm-update-dj
⟨*proof*⟩

interpretation *lm*: *map-update-dj lm-α lm-invar lm-update-dj* ⟨*proof*⟩

lemma *lm-delete-impl*:
map-delete lm-α lm-invar lm-delete
⟨*proof*⟩

interpretation *lm*: *map-delete lm-α lm-invar lm-delete* ⟨*proof*⟩

lemma *lm-isEmpty-impl*:
map-isEmpty lm-α lm-invar lm-isEmpty
⟨*proof*⟩

interpretation *lm*: *map-isEmpty lm-α lm-invar lm-isEmpty* ⟨*proof*⟩

lemma *lm-is-finite-map*: *finite-map lm-α lm-invar*
⟨*proof*⟩

interpretation *lm*: *finite-map lm-α lm-invar* ⟨*proof*⟩

lemma *lm-iteratei-impl*:
map-iteratei lm-α lm-invar lm-iteratei
⟨*proof*⟩

interpretation *lm*: *map-iteratei lm-α lm-invar lm-iteratei* ⟨*proof*⟩

declare *lm.finite*[*simp del*, *rule del*]

lemma *lm-finite*[*simp*, *intro*!]: *finite* (*dom* (*lm-α m*))
⟨*proof*⟩

lemmas *lm-add-impl* = *it-add-correct*[*OF lm-iteratei-impl lm-update-impl*, *folded lm-add-def*]
interpretation *lm*: *map-add lm-α lm-invar lm-add* ⟨*proof*⟩

lemmas *lm-add-dj-impl* =
map-add.add-dj-by-add[*OF lm-add-impl*, *folded lm-add-dj-def*]

interpretation *lm*: *map-add-dj lm-α lm-invar lm-add-dj* ⟨*proof*⟩

lemmas *lm-sel-impl* = *iti-sel-correct*[*OF lm-iteratei-impl*, *folded lm-sel-def*]
interpretation *lm*: *map-sel lm-α lm-invar lm-sel* ⟨*proof*⟩

lemmas *lm-sel′-impl* = *iti-sel′-correct* [*OF lm-iteratei-impl*, *folded lm-sel′-def*]
interpretation *lm*: *map-sel′ lm-α lm-invar lm-sel′* ⟨*proof*⟩

lemma *lm-to-list-impl*: *map-to-list lm-α lm-invar lm-to-list*
⟨*proof*⟩

interpretation *lm*: *map-to-list lm-α lm-invar lm-to-list* ⟨*proof*⟩

lemmas *list-to-lm-impl* =
gen-list-to-map-correct[*OF lm-empty-impl lm-update-impl*,
folded list-to-lm-def]

interpretation *lm*: *list-to-map lm-α lm-invar list-to-lm*
⟨*proof*⟩

4.2.3 Code Generation

lemmas *lm-correct* =
lm.empty-correct
lm.lookup-correct
lm.update-correct
lm.update-dj-correct
lm.delete-correct
lm.isEmpty-correct
lm.add-correct
lm.add-dj-correct
lm.to-list-correct
lm.to-map-correct

export-code
lm-empty
lm-lookup
lm-update
lm-update-dj
lm-delete
lm-isEmpty
lm-iteratei
lm-add
lm-add-dj
lm-sel
lm-sel′
lm-to-list
list-to-lm
in *SML*
module-name *ListMap*
file −

end

4.3 Map Implementation by Association Lists with explicit invariants

theory *ListMapImpl-Invar*
imports
../spec/MapSpec
../common/Assoc-List
../gen-algo/MapGA
begin type-synonym $('k,'v)$ *lmi* = $('k \times 'v)$ *list*

4.3.1 Functions

definition *lmi-α* == *Map.map-of*
definition *lmi-invar m* == *distinct* (*List.map fst m*)
definition *lmi-empty* == $(\lambda$-::*unit*. []$)$
definition *lmi-lookup k m* == *Map.map-of m k*
definition *lmi-update* == *AList.update*
definition *lmi-update-dj k v m* == (k, v) # *m*
definition *lmi-delete k m* == *Assoc-List.delete-aux k m*
definition *lmi-iteratei* :: $('k,'v)$ *lmi* ⇒ $('k \times 'v,'\sigma)$ *set-iterator*
where *lmi-iteratei* = *foldli*
definition *lmi-isEmpty m* == *m*=[]

definition *lmi-add* == *it-add lmi-update lmi-iteratei*
definition *lmi-add-dj* :: $('k,'v)$ *lmi* ⇒ $('k,'v)$ *lmi* ⇒ $('k,'v)$ *lmi*
where *lmi-add-dj* == *revg*

definition *lmi-sel* == *iti-sel lmi-iteratei*
definition *lmi-sel'* == *sel-sel' lmi-sel*
definition *lmi-to-list* :: $('u,'v)$ *lmi* ⇒ $('u \times 'v)$ *list*
where *lmi-to-list* = *id*
definition *list-to-lmi* == *gen-list-to-map lmi-empty lmi-update*
definition *list-to-lmi-dj* :: $('u \times 'v)$ *list* ⇒ $('u,'v)$ *lmi*
where *list-to-lmi-dj* == *id*

4.3.2 Correctness

lemmas *lmi-defs* =
lmi-α-def
lmi-invar-def
lmi-empty-def
lmi-lookup-def
lmi-update-def
lmi-update-dj-def
lmi-delete-def
lmi-iteratei-def
lmi-isEmpty-def

lmi-add-def
lmi-add-dj-def
lmi-sel-def
lmi-sel′-def
lmi-to-list-def
list-to-lmi-def
list-to-lmi-dj-def

lemma *lmi-empty-impl*:
map-empty lmi-α lmi-invar lmi-empty
⟨*proof*⟩

interpretation *lmi*: *map-empty lmi-α lmi-invar lmi-empty* ⟨*proof*⟩

lemma *lmi-lookup-impl*:
map-lookup lmi-α lmi-invar lmi-lookup
⟨*proof*⟩

interpretation *lmi*: *map-lookup lmi-α lmi-invar lmi-lookup* ⟨*proof*⟩

lemma *lmi-update-impl*:
map-update lmi-α lmi-invar lmi-update
⟨*proof*⟩

interpretation *lmi*: *map-update lmi-α lmi-invar lmi-update* ⟨*proof*⟩

lemma *lmi-update-dj-impl*:
map-update-dj lmi-α lmi-invar lmi-update-dj
⟨*proof*⟩

interpretation *lmi*: *map-update-dj lmi-α lmi-invar lmi-update-dj* ⟨*proof*⟩

lemma *lmi-delete-impl*:
map-delete lmi-α lmi-invar lmi-delete
⟨*proof*⟩

interpretation *lmi*: *map-delete lmi-α lmi-invar lmi-delete* ⟨*proof*⟩

lemma *lmi-isEmpty-impl*:
map-isEmpty lmi-α lmi-invar lmi-isEmpty
⟨*proof*⟩

interpretation *lmi*: *map-isEmpty lmi-α lmi-invar lmi-isEmpty* ⟨*proof*⟩

lemma *lmi-is-finite-map*: *finite-map lmi-α lmi-invar*
⟨*proof*⟩

interpretation *lmi*: *finite-map lmi-α lmi-invar* ⟨*proof*⟩

lemma *lmi-iteratei-impl*:
map-iteratei lmi-α lmi-invar lmi-iteratei
⟨*proof*⟩

interpretation *lmi*: *map-iteratei lmi-α lmi-invar lmi-iteratei* ⟨*proof*⟩

declare *lmi.finite*[*simp del*, *rule del*]

lemma *lmi-finite*[*simp*, *intro*!]: *finite* (*dom* (*lmi-α m*))
⟨*proof*⟩

lemmas *lmi-add-impl* =
it-add-correct[*OF lmi-iteratei-impl lmi-update-impl*, *folded lmi-add-def*]
interpretation *lmi*: *map-add lmi-α lmi-invar lmi-add* ⟨*proof*⟩

lemma *lmi-add-dj-impl*:
shows *map-add-dj lmi-α lmi-invar lmi-add-dj*
⟨*proof*⟩

interpretation *lmi*: *map-add-dj lmi-α lmi-invar lmi-add-dj* ⟨*proof*⟩

lemmas *lmi-sel-impl* = *iti-sel-correct*[*OF lmi-iteratei-impl*, *folded lmi-sel-def*]
interpretation *lmi*: *map-sel lmi-α lmi-invar lmi-sel* ⟨*proof*⟩

lemmas *lmi-sel′-impl* = *sel-sel′-correct*[*OF lmi-sel-impl*, *folded lmi-sel′-def*]
interpretation *lmi*: *map-sel′ lmi-α lmi-invar lmi-sel′* ⟨*proof*⟩

lemma *lmi-to-list-impl*: *map-to-list lmi-α lmi-invar lmi-to-list*
⟨*proof*⟩
interpretation *lmi*: *map-to-list lmi-α lmi-invar lmi-to-list* ⟨*proof*⟩

lemmas *list-to-lmi-impl* =
gen-list-to-map-correct[*OF lmi-empty-impl lmi-update-impl*,
folded list-to-lmi-def]

interpretation *lmi*: *list-to-map lmi-α lmi-invar list-to-lmi*
⟨*proof*⟩

lemma *list-to-lmi-dj-correct*:
assumes [*simp*]: *distinct* (*map fst l*)
shows *lmi-α* (*list-to-lmi-dj l*) = *map-of l*
lmi-invar (*list-to-lmi-dj l*)
⟨*proof*⟩

lemma *lmi-to-list-to-lm*[*simp*]:
lmi-invar m ⟹ *lmi-α* (*list-to-lmi-dj* (*lmi-to-list m*)) = *lmi-α m*
⟨*proof*⟩

4.3.3 Code Generation

lemmas *lmi-correct* =
lmi.empty-correct
lmi.lookup-correct
lmi.update-correct
lmi.update-dj-correct
lmi.delete-correct
lmi.isEmpty-correct
lmi.add-correct
lmi.add-dj-correct
lmi.to-list-correct
lmi.to-map-correct
list-to-lmi-dj-correct

export-code
lmi-empty
lmi-lookup
lmi-update
lmi-update-dj
lmi-delete
lmi-isEmpty
lmi-iteratei
lmi-add
lmi-add-dj
lmi-sel
lmi-sel′
lmi-to-list
list-to-lmi
list-to-lmi-dj
in *SML*
module-name *ListMap*
file −

end

4.4 Map Implementation by Red-Black-Trees

theory *RBTMapImpl*
imports
../spec/MapSpec
../common/RBT-add
../gen-algo/MapGA
begin type-synonym $('k,'v)$ *rm* = $('k,'v)$ *RBT.rbt*

4.4.1 Definitions

definition *rm-α* == *RBT.lookup*

abbreviation (*input*) *rm-invar* **where** *rm-invar* == λ-. *True*
definition *rm-empty* == (λ-::*unit*. *RBT.empty*)
definition *rm-lookup k m* == *RBT.lookup m k*
definition *rm-update* == *RBT.insert*
definition *rm-update-dj* == *rm-update*
definition *rm-delete* == *RBT.delete*
definition *rm-iterateoi* **where** *rm-iterateoi r* == *RBT-add.rm-iterateoi* (*RBT.impl-of r*)
definition *rm-reverse-iterateoi* **where** *rm-reverse-iterateoi r* == *RBT-add.rm-reverse-iterateoi* (*RBT.impl-of r*)
definition *rm-iteratei* == *rm-iterateoi*

definition *rm-add* == *it-add rm-update rm-iteratei*
definition *rm-add-dj* == *rm-add*
definition *rm-isEmpty m* == *m*=*RBT.empty*
definition *rm-sel* == *iti-sel rm-iteratei*
definition *rm-sel′* = *iti-sel-no-map rm-iteratei*

definition *rm-to-list* == *it-map-to-list rm-reverse-iterateoi*
definition *list-to-rm* == *gen-list-to-map rm-empty rm-update*

definition *rm-min* == *iti-sel-no-map rm-iterateoi*
definition *rm-max* == *iti-sel-no-map rm-reverse-iterateoi*

4.4.2 Correctness

lemmas *rm-defs* =
rm-α-def
rm-empty-def
rm-lookup-def
rm-update-def
rm-update-dj-def
rm-delete-def
rm-iteratei-def
rm-iterateoi-def
rm-reverse-iterateoi-def
rm-add-def
rm-add-dj-def
rm-isEmpty-def
rm-sel-def
rm-sel′-def
rm-to-list-def
list-to-rm-def
rm-min-def
rm-max-def

lemma *rm-empty-impl*: *map-empty rm-α rm-invar rm-empty*
⟨*proof*⟩

lemma *rm-lookup-impl*: *map-lookup rm-α rm-invar rm-lookup*
⟨*proof*⟩

lemma *rm-update-impl*: *map-update rm-α rm-invar rm-update*
⟨*proof*⟩

lemma *rm-update-dj-impl*: *map-update-dj rm-α rm-invar rm-update-dj*
⟨*proof*⟩

lemma *rm-delete-impl*: *map-delete rm-α rm-invar rm-delete*
⟨*proof*⟩

lemma *rm-α-alist*: *rm-invar m* ⟹ *rm-α m* = *Map.map-of* (*RBT.entries m*)
⟨*proof*⟩

lemma *rm-α-finite*[*simp*, *intro*!]: *finite* (*dom* (*rm-α m*))
⟨*proof*⟩

lemma *rm-is-finite-map*: *finite-map rm-α rm-invar* ⟨*proof*⟩

lemma *map-to-set-lookup-entries*:
rbt-sorted t ⟹ *map-to-set* (*rbt-lookup t*) = *set* (*RBT-Impl.entries t*)
⟨*proof*⟩

lemma *rm-iterateoi-correct*:
fixes *t*::('*k*::*linorder*, '*v*) *RBT-Impl.rbt*
assumes *is-sort*: *rbt-sorted t*
defines *it* ≡ *RBT-add.rm-iterateoi*::(('*k*, '*v*) *RBT-Impl.rbt* ⇒ ('*k* × '*v*, '*σ*) *set-iterator*)
shows *map-iterator-linord* (*it t*) (*rbt-lookup t*)
⟨*proof*⟩

lemma *rm-iterateoi-impl*: *map-iterateoi rm-α rm-invar rm-iterateoi*
⟨*proof*⟩

lemma *rm-reverse-iterateoi-correct*:
fixes *t*::('*k*::*linorder*, '*v*) *RBT-Impl.rbt*
assumes *is-sort*: *rbt-sorted t*
defines *it* ≡ *RBT-add.rm-reverse-iterateoi*::(('*k*, '*v*) *RBT-Impl.rbt* ⇒ ('*k* × '*v*, '*σ*) *set-iterator*)
shows *map-iterator-rev-linord* (*it t*) (*rbt-lookup t*)
⟨*proof*⟩

lemma *rm-reverse-iterateoi-impl*: *map-reverse-iterateoi rm-α rm-invar rm-reverse-iterateoi*
⟨*proof*⟩

lemmas *rm-iteratei-impl* = *MapGA.iti-by-itoi*[*OF rm-iterateoi-impl*, *folded rm-iteratei-def*]

lemmas *rm-add-impl* =
it-add-correct[*OF rm-iteratei-impl rm-update-impl*, *folded rm-add-def*]

lemmas *rm-add-dj-impl* =
map-add.add-dj-by-add[*OF rm-add-impl*, *folded rm-add-dj-def*]

lemma *rm-isEmpty-impl*: *map-isEmpty rm-α rm-invar rm-isEmpty*
⟨*proof*⟩

lemmas *rm-sel-impl* = *iti-sel-correct*[*OF rm-iteratei-impl*, *folded rm-sel-def*]
lemmas *rm-sel′-impl* = *iti-sel′-correct*[*OF rm-iteratei-impl*, *folded rm-sel′-def*]

lemmas *rm-to-sorted-list-impl*
= *rito-map-to-sorted-list-correct*[*OF rm-reverse-iterateoi-impl*, *folded rm-to-list-def*]

lemmas *list-to-rm-impl*
= *gen-list-to-map-correct*[*OF rm-empty-impl rm-update-impl*,
folded list-to-rm-def]

lemmas *rm-min-impl* = *MapGA.itoi-min-correct*[*OF rm-iterateoi-impl*, *folded rm-min-def*]
lemmas *rm-max-impl* = *MapGA.ritoi-max-correct*[*OF rm-reverse-iterateoi-impl*,
folded rm-max-def]

interpretation *rm*: *map-empty rm-α rm-invar rm-empty*
⟨*proof*⟩
interpretation *rm*: *map-lookup rm-α rm-invar rm-lookup*
⟨*proof*⟩
interpretation *rm*: *map-update rm-α rm-invar rm-update*
⟨*proof*⟩
interpretation *rm*: *map-update-dj rm-α rm-invar rm-update-dj*
⟨*proof*⟩
interpretation *rm*: *map-delete rm-α rm-invar rm-delete*
⟨*proof*⟩
interpretation *rm*: *finite-map rm-α rm-invar*
⟨*proof*⟩
interpretation *rm*: *map-iteratei rm-α rm-invar rm-iteratei*
⟨*proof*⟩
interpretation *rm*: *map-iterateoi rm-α rm-invar rm-iterateoi*
⟨*proof*⟩
interpretation *rm*: *map-reverse-iterateoi rm-α rm-invar rm-reverse-iterateoi*
⟨*proof*⟩
interpretation *rm*: *map-add rm-α rm-invar rm-add*
⟨*proof*⟩
interpretation *rm*: *map-add-dj rm-α rm-invar rm-add-dj*
⟨*proof*⟩
interpretation *rm*: *map-isEmpty rm-α rm-invar rm-isEmpty*
⟨*proof*⟩
interpretation *rm*: *map-sel rm-α rm-invar rm-sel*
⟨*proof*⟩
interpretation *rm*: *map-sel′ rm-α rm-invar rm-sel′*

⟨*proof*⟩
interpretation *rm*: *map-to-sorted-list rm-α rm-invar rm-to-list*
⟨*proof*⟩
interpretation *rm*: *list-to-map rm-α rm-invar list-to-rm*
⟨*proof*⟩

interpretation *rm*: *map-min rm-α rm-invar rm-min*
⟨*proof*⟩
interpretation *rm*: *map-max rm-α rm-invar rm-max*
⟨*proof*⟩

declare *rm.finite*[*simp del*, *rule del*]

lemmas *rm-correct* =
rm.empty-correct
rm.lookup-correct
rm.update-correct
rm.update-dj-correct
rm.delete-correct
rm.add-correct
rm.add-dj-correct
rm.isEmpty-correct
rm.to-list-correct
rm.to-map-correct

4.4.3 Code Generation

export-code
rm-empty
rm-lookup
rm-update
rm-update-dj
rm-delete
rm-iteratei
rm-iterateoi
rm-reverse-iterateoi
rm-add
rm-add-dj
rm-isEmpty
rm-sel
rm-sel′
rm-to-list
list-to-rm
rm-min
rm-max
in *SML*
module-name *RBTMap*
file −

end

4.5 The hashable Typeclass

theory *HashCode*
imports *Main*
begin

In this theory a typeclass of hashable types is established. For hashable types, there is a function *hashcode*, that maps any entity of this type to an integer value.

This theory defines the hashable typeclass and provides instantiations for a couple of standard types.

type-synonym
hashcode = *nat*

class *hashable* =
fixes *hashcode* :: *'a* ⇒ *hashcode*
and *bounded-hashcode* :: *nat* ⇒ *'a* ⇒ *hashcode*
and *def-hashmap-size* :: *'a itself* ⇒ *nat*
assumes *bounded-hashcode-bounds*: *1 < n* ⟹ *bounded-hashcode n a < n*
and *def-hashmap-size*: *1 < def-hashmap-size TYPE('a)*

instantiation *unit* :: *hashable*
begin
definition [*simp*]: *hashcode* (*u* :: *unit*) = *0*
definition [*simp*]: *bounded-hashcode n* (*u* :: *unit*) = *0*
definition *def-hashmap-size* = (λ- :: *unit itself*. *2*)
instance ⟨*proof*⟩
end

instantiation *bool* :: *hashable*
begin
definition [*simp*]: *hashcode* (*b* :: *bool*) = (*if b then 1 else 0*)
definition [*simp*]: *bounded-hashcode n* (*b* :: *bool*) = (*if b then 1 else 0*)
definition *def-hashmap-size* = (λ- :: *bool itself*. *2*)
instance ⟨*proof*⟩
end

instantiation *int* :: *hashable*
begin
definition [*simp*]: *hashcode* (*i* :: *int*) = *nat* (*abs i*)
definition [*simp*]: *bounded-hashcode n* (*i* :: *int*) = *nat* (*abs i*) *mod n*
definition *def-hashmap-size* = (λ- :: *int itself*. *16*)
instance ⟨*proof*⟩
end

```
instantiation nat :: hashable
begin
  definition [simp]: hashcode (n :: nat) = n
  definition [simp]: bounded-hashcode n' (n :: nat) == n mod n'
  definition def-hashmap-size = (λ- :: nat itself. 16)
  instance ⟨proof⟩
end

fun num-of-nibble :: nibble ⇒ nat
  where
  num-of-nibble Nibble0 = 0 |
  num-of-nibble Nibble1 = 1 |
  num-of-nibble Nibble2 = 2 |
  num-of-nibble Nibble3 = 3 |
  num-of-nibble Nibble4 = 4 |
  num-of-nibble Nibble5 = 5 |
  num-of-nibble Nibble6 = 6 |
  num-of-nibble Nibble7 = 7 |
  num-of-nibble Nibble8 = 8 |
  num-of-nibble Nibble9 = 9 |
  num-of-nibble NibbleA = 10 |
  num-of-nibble NibbleB = 11 |
  num-of-nibble NibbleC = 12 |
  num-of-nibble NibbleD = 13 |
  num-of-nibble NibbleE = 14 |
  num-of-nibble NibbleF = 15

instantiation nibble :: hashable
begin
  definition [simp]: hashcode (c :: nibble) = num-of-nibble c
  definition [simp]: bounded-hashcode n c == num-of-nibble c mod n
  definition def-hashmap-size = (λ- :: nibble itself. 16)
  instance ⟨proof⟩
end

instantiation char :: hashable
begin
  fun hashcode-of-char ::  char ⇒ hashcode where
    hashcode-of-char (Char a b) = num-of-nibble a * 16 + num-of-nibble b

  definition [simp]: hashcode c == hashcode-of-char c
  definition [simp]: bounded-hashcode n c == hashcode-of-char c mod n
  definition def-hashmap-size = (λ- :: char itself. 32)
  instance ⟨proof⟩
end

instantiation prod :: (hashable, hashable) hashable
begin
```

definition *hashcode x == (hashcode (fst x) * 33 + hashcode (snd x))*
definition *bounded-hashcode n x == (bounded-hashcode n (fst x) * 33 + bounded-hashcode n (snd x)) mod n*
definition *def-hashmap-size = (λ- :: ('a × 'b) itself. def-hashmap-size TYPE('a) + def-hashmap-size TYPE('b))*
instance ⟨*proof*⟩
end

instantiation *sum* :: (*hashable*, *hashable*) *hashable*
begin
definition *hashcode x == (case x of Inl a ⇒ 2 * hashcode a | Inr b ⇒ 2 * hashcode b + 1)*
definition *bounded-hashcode n x == (case x of Inl a ⇒ bounded-hashcode n a | Inr b ⇒ (n − 1 − bounded-hashcode n b))*
definition *def-hashmap-size = (λ- :: ('a + 'b) itself. def-hashmap-size TYPE('a) + def-hashmap-size TYPE('b))*
instance ⟨*proof*⟩
end

instantiation *list* :: (*hashable*) *hashable*
begin
definition *hashcode = foldl (λh x. h * 33 + hashcode x) 5381*
definition *bounded-hashcode n = foldl (λh x. (h * 33 + bounded-hashcode n x) mod n) (5381 mod n)*
definition *def-hashmap-size = (λ- :: 'a list itself. 2 * def-hashmap-size TYPE('a))*
instance
⟨*proof*⟩
end

instantiation *option* :: (*hashable*) *hashable*
begin
definition *hashcode opt = (case opt of None ⇒ 0 | Some a ⇒ hashcode a + 1)*
definition *bounded-hashcode n opt = (case opt of None ⇒ 0 | Some a ⇒ (bounded-hashcode n a + 1) mod n)*
definition *def-hashmap-size = (λ- :: 'a option itself. def-hashmap-size TYPE('a) + 1)*
instance ⟨*proof*⟩
end

lemma *hashcode-option-simps* [*simp*]:
hashcode None = 0
hashcode (Some a) = 1 + hashcode a
⟨*proof*⟩

lemma *bounded-hashcode-option-simps* [*simp*]:
bounded-hashcode n None = 0
bounded-hashcode n (Some a) = (bounded-hashcode n a + 1) mod n
⟨*proof*⟩

end

4.6 Hash maps implementation

theory *HashMap-Impl*
imports
../common/Misc
ListMapImpl
RBTMapImpl
../common/HashCode
begin

We use a red-black tree instead of an indexed array. This has the disadvantage of being more complex, however we need not bother about a fixed-size array and rehashing if the array becomes too full.

The entries of the red-black tree are lists of (key,value) pairs.

4.6.1 Abstract Hashmap

We first specify the behavior of our hashmap on the level of maps. We will then show that our implementation based on hashcode-map and bucket-map is a correct implementation of this specification.

type-synonym
$('k,'v)$ *abs-hashmap* = *hashcode* $\rightharpoonup$ $('k \rightharpoonup 'v)$

— Map entry of map by function
abbreviation *map-entry k f m* == $m(k := f\ (m\ k))$

— Invariant: Buckets only contain entries with the right hashcode and there are no empty buckets
definition *ahm-invar*:: $('k$::*hashable*,$'v)$ *abs-hashmap* ⇒ *bool*
where *ahm-invar m* ==
$(\forall\, hc\ cm\ k.\ m\ hc = Some\ cm \wedge k \in dom\ cm \longrightarrow hashcode\ k = hc) \wedge$
$(\forall\, hc\ cm.\ m\ hc = Some\ cm \longrightarrow cm \neq empty)$

— Abstract a hashmap to the corresponding map
definition *ahm-α* **where**
ahm-α m k == *case m* (*hashcode k*) *of*
None ⇒ *None* |
Some cm ⇒ *cm k*

— Lookup an entry
definition *ahm-lookup* :: $'k$::*hashable* ⇒ $('k,'v)$ *abs-hashmap* ⇒ $'v$ *option*

where *ahm-lookup k m == (ahm-α m) k*

— The empty hashmap
definition *ahm-empty* :: (*'k*::*hashable*,*'v*) *abs-hashmap*
where *ahm-empty = empty*

— Update/insert an entry
definition *ahm-update* **where**
ahm-update k v m ==
case m (hashcode k) of
None ⇒ m (hashcode k ↦ [k ↦ v]) |
Some cm ⇒ m (hashcode k ↦ cm (k ↦ v))

— Delete an entry
definition *ahm-delete* **where**
ahm-delete k m == map-entry (hashcode k)
(λv. case v of
None ⇒ None |
Some bm ⇒ (
if bm |‘ (− {k}) = empty then
None
else
Some (bm |‘ (− {k}))
)
) m

definition *ahm-isEmpty* **where**
ahm-isEmpty m == m=Map.empty

Now follow correctness lemmas, that relate the hashmap operations to operations on the corresponding map. Those lemmas are named op_correct, where op is the operation.

lemma *ahm-invarI*: ⟦
!!*hc cm k*. ⟦*m hc = Some cm*; *k∈dom cm*⟧ ⟹ *hashcode k = hc*;
!!*hc cm*. ⟦ *m hc = Some cm* ⟧ ⟹ *cm ≠ empty*
⟧ ⟹ *ahm-invar m*
⟨*proof*⟩

lemma *ahm-invarD*: ⟦ *ahm-invar m*; *m hc = Some cm*; *k∈dom cm* ⟧ ⟹ *hashcode k = hc*
⟨*proof*⟩

lemma *ahm-invarDne*: ⟦ *ahm-invar m*; *m hc = Some cm* ⟧ ⟹ *cm ≠ empty*
⟨*proof*⟩

lemma *ahm-invar-bucket-not-empty*[*simp*]:
ahm-invar m ⟹ *m hc ≠ Some Map.empty*

⟨*proof*⟩

lemmas *ahm-lookup-correct* = *ahm-lookup-def*

lemma *ahm-empty-correct*:
ahm-α ahm-empty = *empty*
ahm-invar ahm-empty
⟨*proof*⟩

lemma *ahm-update-correct*:
ahm-α (*ahm-update k v m*) = *ahm-α m* (*k* ↦ *v*)
ahm-invar m ⟹ *ahm-invar* (*ahm-update k v m*)
⟨*proof*⟩

lemma *fun-upd-apply-ne*: *x*≠*y* ⟹ (*f*(*x*:=*v*)) *y* = *f y*
⟨*proof*⟩

lemma *cancel-one-empty-simp*: *m* |‘ (−{*k*}) = *Map.empty* ⟷ *dom m* ⊆ {*k*}
⟨*proof*⟩

lemma *ahm-delete-correct*:
ahm-α (*ahm-delete k m*) = (*ahm-α m*) |‘ (−{*k*})
ahm-invar m ⟹ *ahm-invar* (*ahm-delete k m*)
⟨*proof*⟩

lemma *ahm-isEmpty-correct*: *ahm-invar m* ⟹ *ahm-isEmpty m* ⟷ *ahm-α m* = *Map.empty*
⟨*proof*⟩

lemmas *ahm-correct* = *ahm-empty-correct ahm-lookup-correct ahm-update-correct*

ahm-delete-correct ahm-isEmpty-correct

— Bucket entries correspond to map entries
lemma *ahm-be-is-e*:
assumes *I*: *ahm-invar m*
assumes *A*: *m hc* = *Some bm* *bm k* = *Some v*
shows *ahm-α m k* = *Some v*
⟨*proof*⟩
lemma *ahm-e-is-be*: ⟦
ahm-α m k = *Some v*;
!!*bm*. ⟦*m* (*hashcode k*) = *Some bm*; *bm k* = *Some v* ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

4.6.2 Concrete Hashmap

In this section, we define the concrete hashmap that is made from the hash-code map and the bucket map.

We then show the correctness of the operations w.r.t. the abstract hashmap, and thus, indirectly, w.r.t. the corresponding map.

type-synonym
('k,'v) hm-impl = (hashcode, ('k,'v) lm) rm

Operations

```
  — Auxiliary function: Apply function to value of an entry
definition rm-map-entry
  :: hashcode ⇒ ('v option ⇒ 'v option) ⇒ (hashcode, 'v) rm ⇒ (hashcode,'v) rm
  where
  rm-map-entry k f m ==
      case rm-lookup k m of
        None ⇒ (
          case f None of
            None ⇒ m |
            Some v ⇒ rm-update k v m
        ) |
        Some v ⇒ (
          case f (Some v) of
            None ⇒ rm-delete k m |
            Some v' ⇒ rm-update k v' m
        )


  — Empty hashmap
definition empty :: unit ⇒ ('k :: hashable, 'v) hm-impl where empty == rm-empty

  — Update/insert entry
definition update :: 'k::hashable ⇒ 'v ⇒ ('k,'v) hm-impl ⇒ ('k,'v) hm-impl
  where
  update k v m ==
   let hc = hashcode k in
     case rm-lookup hc m of
       None ⇒ rm-update hc (lm-update k v (lm-empty ())) m |
       Some bm ⇒ rm-update hc (lm-update k v bm) m

  — Lookup value by key
definition lookup :: 'k::hashable ⇒ ('k,'v) hm-impl ⇒ 'v option where
  lookup k m ==
   case rm-lookup (hashcode k) m of
     None ⇒ None |
     Some lm ⇒ lm-lookup k lm

  — Delete entry by key
```

```
definition delete :: 'k::hashable ⇒ ('k,'v) hm-impl ⇒ ('k,'v) hm-impl where
  delete k m ==
   rm-map-entry (hashcode k)
     (λv. case v of
       None ⇒ None |
       Some lm ⇒ (
         let lm' = lm-delete k lm
         in  if lm-isEmpty lm' then None else Some lm'
       )
     ) m

  — Select arbitrary entry
definition sel :: ('k::hashable,'v) hm-impl ⇒ ('k × 'v ⇒ 'r option) ⇒ 'r option
  where sel m f == rm-sel m (λ(hc, lm). lm-sel lm f)

  — Emptiness check
definition isEmpty == rm-isEmpty

  — Interruptible iterator
definition iteratei m c f σ0 ==
  rm-iteratei m c (λ(hc, lm) σ.
    lm-iteratei lm c f σ
  ) σ0

lemma iteratei-alt-def :
  iteratei m = set-iterator-image snd (
     set-iterator-product (rm-iteratei m) (λhclm. lm-iteratei (snd hclm)))
⟨proof⟩
```

Correctness w.r.t. Abstract HashMap

The following lemmas establish the correctness of the operations w.r.t. the abstract hashmap.

They have the naming scheme op_correct', where op is the name of the operation.

```
  — Abstract concrete hashmap to abstract hashmap
definition hm-α' where hm-α' m == λhc. case rm-α m hc of
  None ⇒ None |
  Some lm ⇒ Some (lm-α lm)

  — Invariant for concrete hashmap: The hashcode-map and bucket-maps satisfy
     their invariants and the invariant of the corresponding abstract hashmap is
     satisfied.

definition invar m == ahm-invar (hm-α' m)

lemma rm-map-entry-correct:
  rm-α (rm-map-entry k f m) = (rm-α m)(k := f (rm-α m k))
```

⟨*proof*⟩

lemma *empty-correct'*:
hm-α' (*empty* ()) = *ahm-empty*
invar (*empty* ())
⟨*proof*⟩

lemma *lookup-correct'*:
invar m ⟹ *lookup k m* = *ahm-lookup k* (*hm-α' m*)
⟨*proof*⟩

lemma *update-correct'*:
invar m ⟹ *hm-α'* (*update k v m*) = *ahm-update k v* (*hm-α' m*)
invar m ⟹ *invar* (*update k v m*)
⟨*proof*⟩

lemma *delete-correct'*:
invar m ⟹ *hm-α'* (*delete k m*) = *ahm-delete k* (*hm-α' m*)
invar m ⟹ *invar* (*delete k m*)
⟨*proof*⟩

lemma *isEmpty-correct'*:
invar hm ⟹ *isEmpty hm* ⟷ *ahm-α* (*hm-α' hm*) = *Map.empty*
⟨*proof*⟩

lemma *sel-correct'*:
assumes *invar hm*
shows ⟦ *sel hm f* = *Some r*; ⋀*u v*. ⟦ *ahm-α* (*hm-α' hm*) *u* = *Some v*; *f* (*u*, *v*) = *Some r* ⟧ ⟹ *P* ⟧ ⟹ *P*
and ⟦ *sel hm f* = *None*; *ahm-α* (*hm-α' hm*) *u* = *Some v* ⟧ ⟹ *f* (*u*, *v*) = *None*
⟨*proof*⟩

lemma *iteratei-correct'*:
assumes *invar*: *invar hm*
shows *map-iterator* (*iteratei hm*) (*ahm-α* (*hm-α' hm*))
⟨*proof*⟩

lemmas *hm-correct'* = *empty-correct' lookup-correct' update-correct'*
delete-correct' isEmpty-correct'
sel-correct' iteratei-correct'
lemmas *hm-invars* = *empty-correct'*(*2*) *update-correct'*(*2*)
delete-correct'(*2*)

hide-const (**open**) *empty invar lookup update delete sel isEmpty iteratei*

end

4.7 Hash Maps

theory *HashMap*
imports *HashMap-Impl*
begin

4.7.1 Type definition

typedef $('k, 'v)$ *hashmap* = {*hm* :: $('k ::$ *hashable*, $'v)$ *hm-impl*. *HashMap-Impl.invar hm*}
morphisms *impl-of-RBT-HM RBT-HM*
⟨*proof*⟩

lemma *impl-of-RBT-HM-invar* [*simp*, *intro*!]: *HashMap-Impl.invar* (*impl-of-RBT-HM hm*)
⟨*proof*⟩

lemma *RBT-HM-imp-of-RBT-HM* [*code abstype*]:
RBT-HM (*impl-of-RBT-HM hm*) = *hm*
⟨*proof*⟩

definition *hm-empty-const* :: $('k ::$ *hashable*, $'v)$ *hashmap*
where *hm-empty-const* = *RBT-HM* (*HashMap-Impl.empty* ())

definition *hm-empty* :: *unit* ⇒ $('k ::$ *hashable*, $'v)$ *hashmap*
where *hm-empty* = (λ-. *hm-empty-const*)

definition *hm-lookup k hm* == *HashMap-Impl.lookup k* (*impl-of-RBT-HM hm*)

definition *hm-update* :: $('k ::$ *hashable*$)$ ⇒ $'v$ ⇒ $('k, 'v)$ *hashmap* ⇒ $('k, 'v)$ *hashmap*
where *hm-update k v hm* = *RBT-HM* (*HashMap-Impl.update k v* (*impl-of-RBT-HM hm*))

definition *hm-update-dj* :: $('k ::$ *hashable*$)$ ⇒ $'v$ ⇒ $('k, 'v)$ *hashmap* ⇒ $('k, 'v)$ *hashmap*
where *hm-update-dj* = *hm-update*

definition *hm-delete* :: $('k ::$ *hashable*$)$ ⇒ $('k, 'v)$ *hashmap* ⇒ $('k, 'v)$ *hashmap*
where *hm-delete k hm* = *RBT-HM* (*HashMap-Impl.delete k* (*impl-of-RBT-HM hm*))

definition *hm-isEmpty* :: $('k ::$ *hashable*, $'v)$ *hashmap* ⇒ *bool*
where *hm-isEmpty hm* = *HashMap-Impl.isEmpty* (*impl-of-RBT-HM hm*)

definition *hm-sel* :: $('k ::$ *hashable*, $'v)$ *hashmap* ⇒ $('k \times 'v \rightharpoonup 'a) \rightharpoonup 'a$
where *hm-sel hm* = *HashMap-Impl.sel* (*impl-of-RBT-HM hm*)

definition *hm-sel'* = *MapGA.sel-sel' hm-sel*

definition *hm-iteratei hm == HashMap-Impl.iteratei (impl-of-RBT-HM hm)*

lemma *impl-of-hm-empty* [*simp*, *code abstract*]:
impl-of-RBT-HM (hm-empty-const) = HashMap-Impl.empty ()
⟨*proof*⟩

lemma *impl-of-hm-update* [*simp*, *code abstract*]:
impl-of-RBT-HM (hm-update k v hm) = HashMap-Impl.update k v (impl-of-RBT-HM hm)
⟨*proof*⟩

lemma *impl-of-hm-delete* [*simp*, *code abstract*]:
impl-of-RBT-HM (hm-delete k hm) = HashMap-Impl.delete k (impl-of-RBT-HM hm)
⟨*proof*⟩

4.7.2 Correctness w.r.t. Map

The next lemmas establish the correctness of the hashmap operations w.r.t. the associated map. This is achieved by chaining the correctness lemmas of the concrete hashmap w.r.t. the abstract hashmap and the correctness lemmas of the abstract hashmap w.r.t. maps.

type-synonym *('k, 'v) hm = ('k, 'v) hashmap*

— Abstract concrete hashmap to map
definition *hm-α == ahm-α ∘ hm-α' ∘ impl-of-RBT-HM*

abbreviation (*input*) *hm-invar :: ('k :: hashable, 'v) hashmap ⇒ bool*
where *hm-invar == λ-. True*

lemma *hm-aux-correct*:
hm-α (hm-empty ()) = empty
hm-lookup k m = hm-α m k
hm-α (hm-update k v m) = (hm-α m)(k↦v)
hm-α (hm-delete k m) = (hm-α m) |` (−{k})
⟨*proof*⟩

4.7.3 Integration in Isabelle Collections Framework

In this section, we integrate hashmaps into the Isabelle Collections Framework.

lemma *hm-empty-impl*: *map-empty hm-α hm-invar hm-empty*
⟨*proof*⟩

lemma *hm-lookup-impl*: *map-lookup hm-α hm-invar hm-lookup*
⟨*proof*⟩

lemma *hm-update-impl*: *map-update hm-α hm-invar hm-update*
⟨*proof*⟩

lemmas *hm-update-dj-impl*
= *map-update.update-dj-by-update*[*OF hm-update-impl*,
folded hm-update-dj-def]

lemma *hm-delete-impl*: *map-delete hm-α hm-invar hm-delete*
⟨*proof*⟩

lemma *hm-finite*[*simp*, *intro*!]:
finite (*dom* (*hm-α m*))
⟨*proof*⟩

lemma *hm-is-finite-map*: *finite-map hm-α hm-invar*
⟨*proof*⟩

lemma *hm-isEmpty-impl*: *map-isEmpty hm-α hm-invar hm-isEmpty*
⟨*proof*⟩

lemma *hm-sel-impl*: *map-sel hm-α hm-invar hm-sel*
⟨*proof*⟩

lemma *hm-sel′-impl*: *map-sel′ hm-α hm-invar hm-sel′*
⟨*proof*⟩

lemma *hm-iteratei-impl*:
map-iteratei hm-α hm-invar hm-iteratei
⟨*proof*⟩

definition *hm-add* == *it-add hm-update hm-iteratei*
lemmas *hm-add-impl* = *it-add-correct*[*OF hm-iteratei-impl hm-update-impl*, *folded hm-add-def*]

definition *hm-add-dj* == *it-add hm-update-dj hm-iteratei*
lemmas *hm-add-dj-impl* =
it-add-dj-correct[*OF hm-iteratei-impl hm-update-dj-impl*, *folded hm-add-dj-def*]

definition *hm-to-list* == *it-map-to-list hm-iteratei*
lemmas *hm-to-list-impl* = *it-map-to-list-correct*[*OF hm-iteratei-impl*, *folded hm-to-list-def*]

definition *list-to-hm* == *gen-list-to-map hm-empty hm-update*
lemmas *list-to-hm-impl* =
gen-list-to-map-correct[*OF hm-empty-impl hm-update-impl*, *folded list-to-hm-def*]

interpretation *hm*: *map-empty hm-α hm-invar hm-empty* ⟨*proof*⟩
interpretation *hm*: *map-lookup hm-α hm-invar hm-lookup* ⟨*proof*⟩

interpretation *hm*: *map-update hm-α hm-invar hm-update* ⟨*proof*⟩
interpretation *hm*: *map-update-dj hm-α hm-invar hm-update-dj* ⟨*proof*⟩
interpretation *hm*: *map-delete hm-α hm-invar hm-delete* ⟨*proof*⟩
interpretation *hm*: *finite-map hm-α hm-invar* ⟨*proof*⟩
interpretation *hm*: *map-sel hm-α hm-invar hm-sel* ⟨*proof*⟩
interpretation *hm*: *map-sel′ hm-α hm-invar hm-sel′* ⟨*proof*⟩
interpretation *hm*: *map-isEmpty hm-α hm-invar hm-isEmpty* ⟨*proof*⟩
interpretation *hm*: *map-iteratei hm-α hm-invar hm-iteratei* ⟨*proof*⟩
interpretation *hm*: *map-add hm-α hm-invar hm-add* ⟨*proof*⟩
interpretation *hm*: *map-add-dj hm-α hm-invar hm-add-dj* ⟨*proof*⟩
interpretation *hm*: *map-to-list hm-α hm-invar hm-to-list* ⟨*proof*⟩
interpretation *hm*: *list-to-map hm-α hm-invar list-to-hm* ⟨*proof*⟩

declare *hm.finite*[*simp del*, *rule del*]

lemmas *hm-correct* =
hm.empty-correct
hm.lookup-correct
hm.update-correct
hm.update-dj-correct
hm.delete-correct
hm.add-correct
hm.add-dj-correct
hm.isEmpty-correct
hm.to-list-correct
hm.to-map-correct

4.7.4 Code Generation

export-code
hm-empty
hm-lookup
hm-update
hm-update-dj
hm-delete
hm-sel
hm-sel′
hm-isEmpty
hm-iteratei
hm-add
hm-add-dj
hm-to-list
list-to-hm
in *SML*
module-name *HashMap*
file −

end

4.8 Implementation of a trie with explicit invariants

theory *Trie-Impl* **imports**
../common/Assoc-List
begin

4.8.1 Type definition and primitive operations

datatype $('key, 'val)$ *trie* = *Trie* $'val$ *option* $('key \times ('key, 'val)\ trie)\ list$

lemma *trie-induct* [*case-names Trie*, *induct type*]:
$(\bigwedge vo\ kvs.\ (\bigwedge k\ t.\ (k, t) \in set\ kvs \Longrightarrow P\ t) \Longrightarrow P\ (Trie\ vo\ kvs)) \Longrightarrow P\ t$
⟨*proof*⟩

definition *empty* :: $('key, 'val)$ *trie*
where *empty* = *Trie None* []

fun *lookup* :: $('key, 'val)$ *trie* ⇒ $'key$ *list* ⇒ $'val$ *option*
where
lookup (*Trie vo* -) [] = *vo*
| *lookup* (*Trie* - *ts*) (*k*#*ks*) = (*case map-of ts k of None* ⇒ *None* | *Some t* ⇒ *lookup t ks*)

fun *update* :: $('key, 'val)$ *trie* ⇒ $'key$ *list* ⇒ $'val$ ⇒ $('key, 'val)$ *trie*
where
update (*Trie vo ts*) [] *v* = *Trie* (*Some v*) *ts*
| *update* (*Trie vo ts*) (*k*#*ks*) *v* = *Trie vo* (*Assoc-List.update-with-aux* (*Trie None* []) *k* (λ*t*. *update t ks v*) *ts*)

primrec *isEmpty* :: $('key, 'val)$ *trie* ⇒ *bool*
where *isEmpty* (*Trie vo ts*) ⟷ *vo* = *None* ∧ *ts* = []

fun *delete* :: $('key, 'val)$ *trie* ⇒ $'key$ *list* ⇒ $('key, 'val)$ *trie*
where
delete (*Trie vo ts*) [] = *Trie None ts*
| *delete* (*Trie vo ts*) (*k*#*ks*) =
(*case map-of ts k of None* ⇒ *Trie vo ts*
| *Some t* ⇒ *let t′* = *delete t ks*
in if isEmpty t′
then Trie vo (*Assoc-List.delete-aux k ts*)
else Trie vo (*AList.update k t′ ts*))

fun *trie-invar* :: $('key, 'val)$ *trie* ⇒ *bool*
where *trie-invar* (*Trie vo kts*) = (*distinct* (*map fst kts*) ∧ (∀(*k*, *t*) ∈ *set kts*. ¬ *isEmpty t* ∧ *trie-invar t*))

fun *iteratei-postfixed* :: *'key list* ⇒ (*'key*, *'val*) *trie* ⇒
  (*'key list* × *'val*, *'σ*) *set-iterator*
where
 iteratei-postfixed ks (*Trie vo ts*) *c f σ* =
  (*if c σ*
   then foldli ts c (λ(*k*, *t*) *σ*. *iteratei-postfixed* (*k* # *ks*) *t c f σ*)
        (*case vo of None* ⇒ *σ* | *Some v* ⇒ *f* (*ks*, *v*) *σ*)
   else σ)

definition *iteratei* :: (*'key*, *'val*) *trie* ⇒ (*'key list* × *'val*, *'σ*) *set-iterator*
where *iteratei t c f σ* = *iteratei-postfixed* [] *t c f σ*

lemma *iteratei-postfixed-interrupt*:
 ¬ *c σ* ⟹ *iteratei-postfixed ks t c f σ* = *σ*
⟨*proof*⟩

lemma *iteratei-interrupt*:
 ¬ *c σ* ⟹ *iteratei t c f σ* = *σ*
⟨*proof*⟩

lemma *iteratei-postfixed-alt-def* :
 iteratei-postfixed ks ((*Trie vo ts*)::(*'key*, *'val*) *trie*) =
  (*set-iterator-union*
   (*option-case set-iterator-emp* (λ*v*. *set-iterator-sng* (*ks*, *v*)) *vo*)
   (*set-iterator-image snd*
   (*set-iterator-product* (*foldli ts*)
     (λ(*k*, *t'*). *iteratei-postfixed* (*k* # *ks*) *t'*))
   ))
⟨*proof*⟩

4.8.2 Lookup simps

lemma *lookup-eq-Some-iff* :
assumes *invar*: *trie-invar* ((*Trie vo kvs*) :: (*'key*, *'val*) *trie*)
shows *lookup* (*Trie vo kvs*) *ks* = *Some v* ⟷
  (*ks* = [] ∧ *vo* = *Some v*) ∨
  (∃ *k t ks'*. *ks* = *k* # *ks'* ∧
    (*k*, *t*) ∈ *set kvs* ∧ *lookup t ks'* = *Some v*)
⟨*proof*⟩

lemma *lookup-eq-None-iff* :
assumes *invar*: *trie-invar* ((*Trie vo kvs*) :: (*'key*, *'val*) *trie*)
shows *lookup* (*Trie vo kvs*) *ks* = *None* ⟷
  (*ks* = [] ∧ *vo* = *None*) ∨
  (∃ *k ks'*. *ks* = *k* # *ks'* ∧ (∀ *t*. (*k*, *t*) ∈ *set kvs* ⟶ *lookup t ks'* = *None*))
⟨*proof*⟩

4.8.3 The empty trie

lemma *trie-invar-empty* [*simp*, *intro*!]: *trie-invar empty*
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
⟨*proof*⟩

lemma *lookup-empty′* [*simp*]:
lookup (*Trie None* []) *ks* = *None*
⟨*proof*⟩

4.8.4 Emptyness check

lemma *isEmpty-conv*:
isEmpty ts ⟷ *ts* = *Trie None* []
⟨*proof*⟩

lemma *update-not-empty*: ¬ *isEmpty* (*update t ks v*)
⟨*proof*⟩

lemma *isEmpty-lookup-empty*:
trie-invar t ⟹ *isEmpty t* ⟷ *lookup t* = *Map.empty*
⟨*proof*⟩

4.8.5 Trie update

lemma *lookup-update*:
lookup (*update t ks v*) *ks′* = (*if ks* = *ks′ then Some v else lookup t ks′*)
⟨*proof*⟩

lemma *lookup-update′*:
lookup (*update t ks v*) = (*lookup t*)(*ks* ↦ *v*)
⟨*proof*⟩

lemma *trie-invar-update*: *trie-invar t* ⟹ *trie-invar* (*update t ks v*)
⟨*proof*⟩

4.8.6 Trie removal

lemma *delete-eq-empty-lookup-other-fail*:
⟦ *delete t ks* = *Trie None* []; *ks′* ≠ *ks* ⟧ ⟹ *lookup t ks′* = *None*
⟨*proof*⟩

lemma *lookup-delete*:
trie-invar t ⟹ *lookup* (*delete t ks*) *ks′* = (*if ks* = *ks′ then None else lookup t ks′*)
⟨*proof*⟩

lemma *lookup-delete'*:
trie-invar t $\Longrightarrow$ *lookup* (*delete t ks*) = (*lookup t*)(*ks* := *None*)
⟨*proof*⟩

lemma *trie-invar-delete*:
trie-invar t $\Longrightarrow$ *trie-invar* (*delete t ks*)
⟨*proof*⟩

4.8.7 Domain of a trie

lemma *dom-lookup*:
dom (*lookup* (*Trie vo kts*)) =
($\bigcup$ *k*∈*dom* (*map-of kts*). *Cons k* ‘ *dom* (*lookup* (*the* (*map-of kts k*)))) ∪
(*if vo* = *None then* {} *else* {[]})
⟨*proof*⟩

lemma *finite-dom-trie-lookup*:
finite (*dom* (*lookup t*))
⟨*proof*⟩

lemma *dom-lookup-empty-conv*: *trie-invar t* $\Longrightarrow$ *dom* (*lookup t*) = {} $\longleftrightarrow$ *isEmpty t*
⟨*proof*⟩

4.8.8 Interuptible iterator

lemma *iteratei-postfixed-correct* :
assumes *invar*: *trie-invar* (*t* :: (*'key*, *'val*) *trie*)
shows *set-iterator* ((*iteratei-postfixed ks0 t*)::(*'key list* × *'val*, *'σ*) *set-iterator*)
((λ*ksv*. (*rev* (*fst ksv*) @ *ks0*, (*snd ksv*))) ‘ (*map-to-set* (*lookup t*)))
⟨*proof*⟩

definition *trie-reverse-key* **where**
trie-reverse-key ksv = (*rev* (*fst ksv*), (*snd ksv*))

lemma *trie-reverse-key-alt-def*[*code*] :
trie-reverse-key (*ks*, *v*) = (*rev ks*, *v*)
⟨*proof*⟩

lemma *trie-reverse-key-reverse*[*simp*] :
trie-reverse-key (*trie-reverse-key ksv*) = *ksv*
⟨*proof*⟩

lemma *trie-iteratei-correct*:
assumes *invar*: *trie-invar* (*t* :: (*'key*, *'val*) *trie*)
shows *set-iterator* ((*iteratei t*)::(*'key list* × *'val*, *'σ*) *set-iterator*)
(*trie-reverse-key* ‘ (*map-to-set* (*lookup t*)))
⟨*proof*⟩

hide-const (**open**) *empty isEmpty iteratei lookup update delete*
hide-type (**open**) *trie*

end

4.9 Tries without invariants

theory *Trie* **imports**
Trie-Impl
begin
⟨*proof*⟩

4.9.1 Abstract type definition

typedef (*'key*, *'val*) *trie* =
{*t* :: (*'key*, *'val*) *Trie-Impl.trie*. *trie-invar t*}
morphisms *impl-of Trie*
⟨*proof*⟩

lemma *trie-invar-impl-of* [*simp*, *intro*]: *trie-invar* (*impl-of t*)
⟨*proof*⟩

lemma *Trie-impl-of* [*code abstype*]: *Trie* (*impl-of t*) = *t*
⟨*proof*⟩

4.9.2 Primitive operations

definition *empty* :: (*'key*, *'val*) *trie*
where *empty* = *Trie* (*Trie-Impl.empty*)

definition *update* :: *'key list* ⇒ *'val* ⇒ (*'key*, *'val*) *trie* ⇒ (*'key*, *'val*) *trie*
where *update ks v t* = *Trie* (*Trie-Impl.update* (*impl-of t*) *ks v*)

definition *delete* :: *'key list* ⇒ (*'key*, *'val*) *trie* ⇒ (*'key*, *'val*) *trie*
where *delete ks t* = *Trie* (*Trie-Impl.delete* (*impl-of t*) *ks*)

definition *lookup* :: (*'key*, *'val*) *trie* ⇒ *'key list* ⇒ *'val option*
where *lookup t* = *Trie-Impl.lookup* (*impl-of t*)

definition *isEmpty* :: (*'key*, *'val*) *trie* ⇒ *bool*
where *isEmpty t* = *Trie-Impl.isEmpty* (*impl-of t*)

definition *iteratei* :: (*'key*, *'val*) *trie* ⇒ (*'key list* × *'val*, *'σ*) *set-iterator*
where *iteratei t* = *set-iterator-image trie-reverse-key* (*Trie-Impl.iteratei* (*impl-of t*))

lemma *iteratei-code*[*code*] :

iteratei t c f = Trie-Impl.iteratei (impl-of t) c (λ(ks, v). f (rev ks, v))
⟨*proof*⟩

lemma *impl-of-empty* [*code abstract*]: *impl-of empty = Trie-Impl.empty*
⟨*proof*⟩

lemma *impl-of-update* [*code abstract*]: *impl-of (update ks v t) = Trie-Impl.update (impl-of t) ks v*
⟨*proof*⟩

lemma *impl-of-delete* [*code abstract*]: *impl-of (delete ks t) = Trie-Impl.delete (impl-of t) ks*
⟨*proof*⟩

4.9.3 Correctness of primitive operations

lemma *lookup-empty* [*simp*]: *lookup empty = Map.empty*
⟨*proof*⟩

lemma *lookup-update* [*simp*]: *lookup (update ks v t) = (lookup t)(ks ↦ v)*
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *lookup (delete ks t) = (lookup t)(ks := None)*
⟨*proof*⟩

lemma *isEmpty-lookup*: *isEmpty t ⟷ lookup t = Map.empty*
⟨*proof*⟩

lemma *finite-dom-lookup*: *finite (dom (lookup t))*
⟨*proof*⟩

lemma *iteratei-correct*:
map-iterator (iteratei m) (lookup m)
⟨*proof*⟩

4.9.4 Type classes

instantiation *trie* :: (*equal*, *equal*) *equal* **begin**

definition *equal-class.equal* (*t* :: ('*a*, '*b*) *trie*) *t'* = (*impl-of t = impl-of t'*)

instance
⟨*proof*⟩
end

hide-const (**open**) *empty lookup update delete iteratei isEmpty*

end

4.10 Map implementation via tries

theory *TrieMapImpl* **imports**
Trie
../gen-algo/MapGA
begin

4.10.1 Operations

type-synonym $('k, 'v)$ *tm* $=$ $('k, 'v)$ *trie*

definition *tm-α* :: $('k, 'v)$ *tm* $\Rightarrow$ *'k list* $\rightharpoonup$ *'v*
where *tm-α* $=$ *Trie.lookup*

abbreviation (*input*) *tm-invar* :: $('k, 'v)$ *tm* $\Rightarrow$ *bool*
where *tm-invar* $\equiv$ λ*-*. *True*

definition *tm-empty* :: *unit* $\Rightarrow$ $('k, 'v)$ *tm*
where *tm-empty* $==$ (λ*-*::*unit*. *Trie.empty*)

definition *tm-lookup* :: *'k list* $\Rightarrow$ $('k, 'v)$ *tm* $\Rightarrow$ *'v option*
where *tm-lookup k t* $=$ *Trie.lookup t k*

definition *tm-update* :: *'k list* $\Rightarrow$ *'v* $\Rightarrow$ $('k, 'v)$ *tm* $\Rightarrow$ $('k, 'v)$ *tm*
where *tm-update* $=$ *Trie.update*

definition *tm-update-dj* $==$ *tm-update*

definition *tm-delete* :: *'k list* $\Rightarrow$ $('k, 'v)$ *tm* $\Rightarrow$ $('k, 'v)$ *tm*
where *tm-delete* $=$ *Trie.delete*

definition *tm-iteratei* :: $('k, 'v)$ *tm* $\Rightarrow$ $('k$ *list* $\times$ *'v*, *'σ*$)$ *set-iterator*
where
tm-iteratei $=$ *Trie.iteratei*

definition *tm-add* $==$ *it-add tm-update tm-iteratei*
definition *tm-add-dj* $==$ *tm-add*

definition *tm-isEmpty* :: $('k, 'v)$ *tm* $\Rightarrow$ *bool*
where *tm-isEmpty* $=$ *Trie.isEmpty*

definition *tm-sel* $==$ *iti-sel tm-iteratei*
definition *tm-sel′* $==$ *iti-sel-no-map tm-iteratei*

definition *tm-ball* $==$ *sel-ball tm-sel*
definition *tm-to-list* $==$ *it-map-to-list tm-iteratei*
definition *list-to-tm* $==$ *gen-list-to-map tm-empty tm-update*

lemmas *tm-defs* $=$
tm-α-def

tm-empty-def
tm-lookup-def
tm-update-def
tm-update-dj-def
tm-delete-def
tm-iteratei-def
tm-add-def
tm-add-dj-def
tm-isEmpty-def
tm-sel-def
tm-sel'-def
tm-ball-def
tm-to-list-def
list-to-tm-def

4.10.2 Correctness

lemma *tm-empty-impl*: *map-empty tm-α tm-invar tm-empty*
⟨*proof*⟩

lemma *tm-lookup-impl*: *map-lookup tm-α tm-invar tm-lookup*
⟨*proof*⟩

lemma *tm-update-impl*: *map-update tm-α tm-invar tm-update*
⟨*proof*⟩

lemma *tm-update-dj-impl*: *map-update-dj tm-α tm-invar tm-update-dj*
⟨*proof*⟩

lemma *tm-delete-impl*: *map-delete tm-α tm-invar tm-delete*
⟨*proof*⟩

lemma *tm-α-finite* [*simp*, *intro*!]:
finite (*dom* (*tm-α m*))
⟨*proof*⟩

lemma *tm-is-finite-map*: *finite-map tm-α tm-invar*
⟨*proof*⟩

lemma *tm-iteratei-impl*: *map-iteratei tm-α tm-invar tm-iteratei*
⟨*proof*⟩

lemma *tm-add-impl*: *map-add tm-α tm-invar tm-add*
⟨*proof*⟩

lemma *tm-add-dj-impl*: *map-add-dj tm-α tm-invar tm-add-dj*
⟨*proof*⟩

lemma *tm-isEmpty-impl*: *map-isEmpty tm-α tm-invar tm-isEmpty*

⟨*proof*⟩

lemma *tm-sel-impl*: *map-sel tm-α tm-invar tm-sel*
⟨*proof*⟩

lemma *tm-sel′-impl*: *map-sel′ tm-α tm-invar tm-sel′*
⟨*proof*⟩

lemma *tm-ball-impl*: *map-ball tm-α tm-invar tm-ball*
⟨*proof*⟩

lemma *tm-to-list-impl*: *map-to-list tm-α tm-invar tm-to-list*
⟨*proof*⟩

lemma *list-to-tm-impl*: *list-to-map tm-α tm-invar list-to-tm*
⟨*proof*⟩

interpretation *tm*: *map-empty tm-α tm-invar tm-empty*
⟨*proof*⟩
interpretation *tm*: *map-lookup tm-α tm-invar tm-lookup*
⟨*proof*⟩
interpretation *tm*: *map-update tm-α tm-invar tm-update*
⟨*proof*⟩
interpretation *tm*: *map-update-dj tm-α tm-invar tm-update-dj*
⟨*proof*⟩
interpretation *tm*: *map-delete tm-α tm-invar tm-delete*
⟨*proof*⟩
interpretation *tm*: *finite-map tm-α tm-invar*
⟨*proof*⟩
interpretation *tm*: *map-iteratei tm-α tm-invar tm-iteratei*
⟨*proof*⟩
interpretation *tm*: *map-add tm-α tm-invar tm-add*
⟨*proof*⟩
interpretation *tm*: *map-add-dj tm-α tm-invar tm-add-dj*
⟨*proof*⟩
interpretation *tm*: *map-isEmpty tm-α tm-invar tm-isEmpty*
⟨*proof*⟩
interpretation *tm*: *map-sel tm-α tm-invar tm-sel*
⟨*proof*⟩
interpretation *tm*: *map-sel′ tm-α tm-invar tm-sel′*
⟨*proof*⟩
interpretation *tm*: *map-ball tm-α tm-invar tm-ball*
⟨*proof*⟩
interpretation *tm*: *map-to-list tm-α tm-invar tm-to-list*
⟨*proof*⟩
interpretation *tm*: *list-to-map tm-α tm-invar list-to-tm*
⟨*proof*⟩

declare *tm.finite*[*simp del*, *rule del*]

lemmas *tm-correct* =
tm.empty-correct
tm.lookup-correct
tm.update-correct
tm.update-dj-correct
tm.delete-correct
tm.add-correct
tm.add-dj-correct
tm.isEmpty-correct
tm.ball-correct
tm.to-list-correct
tm.to-map-correct

4.10.3 Code Generation

export-code
tm-empty
tm-lookup
tm-update
tm-update-dj
tm-delete
tm-iteratei
tm-add
tm-add-dj
tm-isEmpty
tm-sel
tm-sel′
tm-ball
tm-to-list
list-to-tm
in *SML*
module-name *TrieMap*
file −

end

4.11 Array-based hash map implementation

theory *ArrayHashMap-Impl* **imports**
../common/HashCode
../common/Array
../gen-algo/ListGA
ListMapImpl
../iterator/SetIterator
../iterator/SetIteratorOperations
begin

Misc.

lemma *idx-iteratei-aux-array-get-Array-conv-nth*:
idx-iteratei-aux array-get sz i (*Array xs*) *c f σ* = *idx-iteratei-aux op* ! *sz i xs c f σ*
⟨*proof*⟩

lemma *idx-iteratei-array-get-Array-conv-nth*:
idx-iteratei array-get array-length (*Array xs*) = *idx-iteratei nth length xs*
⟨*proof*⟩

lemma *idx-iteratei-aux-nth-conv-foldli-drop*:
fixes *xs* :: *'b list*
assumes *i* ≤ *length xs*
shows *idx-iteratei-aux op* ! (*length xs*) *i xs c f σ* = *foldli* (*drop* (*length xs* − *i*) *xs*) *c f σ*
⟨*proof*⟩

lemma *idx-iteratei-nth-length-conv-foldli*: *idx-iteratei nth length* = *foldli*
⟨*proof*⟩

4.11.1 Type definition and primitive operations

definition *load-factor* :: *nat* — in percent
where *load-factor* = *75*

We do not use (*'k*, *'v*) *assoc-list* for the buckets but plain lists of key-value pairs. This speeds up rehashing because we then do not have to go through the abstract operations.

datatype (*'key*, *'val*) *hashmap* =
HashMap (*'key* × *'val*) *list array* *nat*

4.11.2 Operations

definition *new-hashmap-with* :: *nat* ⇒ (*'key* :: *hashable*, *'val*) *hashmap*
where ⋀*size*. *new-hashmap-with size* = *HashMap* (*new-array* [] *size*) *0*

definition *ahm-empty* :: *unit* ⇒ (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty* ≡ λ-. *new-hashmap-with* (*def-hashmap-size TYPE*(*'key*))

definition *bucket-ok* :: *nat* ⇒ *nat* ⇒ ((*'key* :: *hashable*) × *'val*) *list* ⇒ *bool*
where *bucket-ok len h kvs* = (∀ *k* ∈ *fst* ‘ *set kvs*. *bounded-hashcode len k* = *h*)

definition *ahm-invar-aux* :: *nat* ⇒ ((*'key* :: *hashable*) × *'val*) *list array* ⇒ *bool*
where
ahm-invar-aux n a ⟷
(∀ *h*. *h* < *array-length a* ⟶ *bucket-ok* (*array-length a*) *h* (*array-get a h*) ∧ *distinct* (*map fst* (*array-get a h*))) ∧
array-foldl (λ- *n kvs*. *n* + *size kvs*) *0 a* = *n* ∧
array-length a > *1*

primrec *ahm-invar* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *bool*
where *ahm-invar* (*HashMap a n*) = *ahm-invar-aux n a*

definition *ahm-α-aux* :: ((*'key* :: *hashable*) × *'val*) *list array* ⇒ *'key* ⇒ *'val option*
where [*simp*]: *ahm-α-aux a k* = *map-of* (*array-get a* (*bounded-hashcode* (*array-length a*) *k*)) *k*

primrec *ahm-α* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'key* ⇒ *'val option*
where
ahm-α (*HashMap a* -) = *ahm-α-aux a*

definition *ahm-lookup* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'val option*
where *ahm-lookup k hm* = *ahm-α hm k*

primrec *ahm-iteratei-aux* :: ((((*'key* :: *hashable*) × *'val*) *list array*) ⇒ (*'key* × *'val*, *'σ*) *set-iterator*
where *ahm-iteratei-aux* (*Array xs*) *c f* = *foldli* (*concat xs*) *c f*

primrec *ahm-iteratei* :: ((*'key* :: *hashable*, *'val*) *hashmap*) ⇒ ((*'key* × *'val*), *'σ*) *set-iterator*
where
ahm-iteratei (*HashMap a n*) = *ahm-iteratei-aux a*

definition *ahm-rehash-aux'* :: *nat* ⇒ *'key* × *'val* ⇒ ((*'key* :: *hashable*) × *'val*) *list array* ⇒ (*'key* × *'val*) *list array*
where
ahm-rehash-aux' n kv a =
(*let h* = *bounded-hashcode n* (*fst kv*)
in array-set a h (*kv* # *array-get a h*))

definition *ahm-rehash-aux* :: ((*'key* :: *hashable*) × *'val*) *list array* ⇒ *nat* ⇒ (*'key* × *'val*) *list array*
where
ahm-rehash-aux a sz = *ahm-iteratei-aux a* (λ*x*. *True*) (*ahm-rehash-aux' sz*) (*new-array* [] *sz*)

primrec *ahm-rehash* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *nat* ⇒ (*'key*, *'val*) *hashmap*
where *ahm-rehash* (*HashMap a n*) *sz* = *HashMap* (*ahm-rehash-aux a sz*) *n*

primrec *hm-grow* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *nat*
where *hm-grow* (*HashMap a n*) = *2* ∗ *array-length a* + *3*

primrec *ahm-filled* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *bool*
where *ahm-filled* (*HashMap a n*) = (*array-length a* ∗ *load-factor* ≤ *n* ∗ *100*)

primrec *ahm-update-aux* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'key* ⇒ *'val* ⇒ (*'key*, *'val*) *hashmap*

where
ahm-update-aux (*HashMap a n*) *k v* =
(*let h* = *bounded-hashcode* (*array-length a*) *k*;
m = *array-get a h*;
insert = *map-of m k* = *None*
in HashMap (*array-set a h* (*AList.update k v m*)) (*if insert then n* + *1 else n*))

definition *ahm-update* :: *'key* ⇒ *'val* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-update k v hm =
(*let hm'* = *ahm-update-aux hm k v*
in (*if ahm-filled hm' then ahm-rehash hm'* (*hm-grow hm'*) *else hm'*))

primrec *ahm-delete* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-delete k (*HashMap a n*) =
(*let h* = *bounded-hashcode* (*array-length a*) *k*;
m = *array-get a h*;
deleted = (*map-of m k* ≠ *None*)
in HashMap (*array-set a h* (*AList.delete k m*)) (*if deleted then n* − *1 else n*))

lemma *hm-grow-gt-1* [*iff*]:
Suc 0 < *hm-grow hm*
⟨*proof*⟩

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok len h* [] = *True*
⟨*proof*⟩

lemma *bucket-okD*:
⟦ *bucket-ok len h xs*; (*k*, *v*) ∈ *set xs* ⟧
⟹ *bounded-hashcode len k* = *h*
⟨*proof*⟩

lemma *bucket-okI*:
(⋀*k*. *k* ∈ *fst* ‘ *set kvs* ⟹ *bounded-hashcode len k* = *h*) ⟹ *bucket-ok len h kvs*
⟨*proof*⟩

4.11.3 *ahm-invar*

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux n a*
obtains ∀ *h*. *h* < *array-length a* ⟶ *bucket-ok* (*array-length a*) *h* (*array-get a h*)
∧ *distinct* (*map fst* (*array-get a h*))
and *n* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and** *array-length a* > *1*
⟨*proof*⟩

lemma *ahm-invar-auxI*:
⟦ ⋀*h*. *h* < *array-length a* ⟹ *bucket-ok* (*array-length a*) *h* (*array-get a h*);
⋀*h*. *h* < *array-length a* ⟹ *distinct* (*map fst* (*array-get a h*));
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*; *array-length a* > *1* ⟧
⟹ *ahm-invar-aux n a*
⟨*proof*⟩

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
⟨*proof*⟩

4.11.4 *ahm-α*

lemma *finite-dom-ahm-α-aux*:
assumes *ahm-invar-aux n a*
shows *finite* (*dom* (*ahm-α-aux a*))
⟨*proof*⟩

lemma *ahm-α-aux-conv-map-of-concat*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *ahm-α-aux* (*Array xs*) = *map-of* (*concat xs*)
⟨*proof*⟩

lemma *ahm-invar-aux-card-dom-ahm-α-auxD*:
assumes *inv*: *ahm-invar-aux n a*
shows *card* (*dom* (*ahm-α-aux a*)) = *n*
⟨*proof*⟩

lemma *finite-dom-ahm-α*:
ahm-invar hm ⟹ *finite* (*dom* (*ahm-α hm*))
⟨*proof*⟩

lemma *finite-map-ahm-α-aux*:
finite-map ahm-α-aux (*ahm-invar-aux n*)
⟨*proof*⟩

lemma *finite-map-ahm-α*:
finite-map ahm-α ahm-invar
⟨*proof*⟩

4.11.5 *ahm-empty*

lemma *ahm-invar-aux-new-array*:
assumes *n* > *1*
shows *ahm-invar-aux 0* (*new-array* [] *n*)
⟨*proof*⟩

lemma *ahm-invar-new-hashmap-with*:
n > *1* ⟹ *ahm-invar* (*new-hashmap-with n*)

⟨*proof*⟩

lemma *ahm-α-new-hashmap-with*:
  $n > 1 \Longrightarrow$ *ahm-α* (*new-hashmap-with n*) = *empty*
⟨*proof*⟩

lemma *ahm-invar-ahm-empty* [*simp*]: *ahm-invar* (*ahm-empty* ())
⟨*proof*⟩

lemma *ahm-empty-correct* [*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
⟨*proof*⟩

lemma *ahm-empty-impl*: *map-empty ahm-α ahm-invar ahm-empty*
⟨*proof*⟩

4.11.6 *ahm-lookup*

lemma *ahm-lookup-impl*: *map-lookup ahm-α ahm-invar ahm-lookup*
⟨*proof*⟩

4.11.7 *ahm-iteratei*

lemma *ahm-iteratei-aux-impl*:
  map-iteratei ahm-α-aux (*ahm-invar-aux n*) *ahm-iteratei-aux*
⟨*proof*⟩

lemma *ahm-iteratei-correct*:
  map-iteratei ahm-α ahm-invar ahm-iteratei
⟨*proof*⟩

lemma *ahm-iteratei-aux-code* [*code*]:
  ahm-iteratei-aux a c f σ = *idx-iteratei array-get array-length a c* (λ*x*. *foldli x c f*) *σ*
⟨*proof*⟩

4.11.8 *ahm-rehash*

lemma *array-length-ahm-rehash-aux′*:
  array-length (*ahm-rehash-aux′ n kv a*) = *array-length a*
⟨*proof*⟩

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
  assumes *inv*: *ahm-invar-aux n a*
  and *fresh*: *k* ∉ *fst* ‘ *set* (*array-get a* (*bounded-hashcode* (*array-length a*) *k*))
  shows *ahm-invar-aux* (*Suc n*) (*ahm-rehash-aux′* (*array-length a*) (*k*, *v*) *a*)
  (**is** *ahm-invar-aux - ?a*)
⟨*proof*⟩

lemma *ahm-rehash-aux-correct*:
  fixes *a* :: ((′*key* :: *hashable*) × ′*val*) *list array*

assumes *inv*: *ahm-invar-aux n a*
and *sz* > *1*
shows *ahm-invar-aux n* (*ahm-rehash-aux a sz*) (**is** *?thesis1*)
and *ahm-α-aux* (*ahm-rehash-aux a sz*) = *ahm-α-aux a* (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-rehash-correct*:
fixes *hm* :: (*'key* :: *hashable*, *'val*) *hashmap*
assumes *inv*: *ahm-invar hm*
and *sz* > *1*
shows *ahm-invar* (*ahm-rehash hm sz*) *ahm-α* (*ahm-rehash hm sz*) = *ahm-α hm*
⟨*proof*⟩

4.11.9 *ahm-update*

lemma *ahm-update-aux-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-update-aux hm k v*) (**is** *?thesis1*)
and *ahm-α* (*ahm-update-aux hm k v*) = (*ahm-α hm*)(*k* ↦ *v*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-update-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-update k v hm*)
and *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *ahm-update-impl*:
map-update ahm-α ahm-invar ahm-update
⟨*proof*⟩

4.11.10 *ahm-delete*

lemma *ahm-delete-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-delete k hm*) (**is** *?thesis1*)
and *ahm-α* (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*}) (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-delete-impl*:
map-delete ahm-α ahm-invar ahm-delete
⟨*proof*⟩

hide-const (**open**) *HashMap ahm-empty bucket-ok ahm-invar ahm-α ahm-lookup ahm-iteratei ahm-rehash hm-grow ahm-filled ahm-update ahm-delete*
hide-type (**open**) *hashmap*

end

4.12 Array-based hash maps without explicit invariants

theory *ArrayHashMap*
imports *ArrayHashMap-Impl*
begin

4.12.1 Abstract type definition

typedef (*'key* :: *hashable*, *'val*) *hashmap* =
{*hm* :: (*'key*, *'val*) *ArrayHashMap-Impl.hashmap*. *ArrayHashMap-Impl.ahm-invar* *hm*}
morphisms *impl-of HashMap*
⟨*proof*⟩

type-synonym (*'k*,*'v*) *ahm* = (*'k*,*'v*) *hashmap*

lemma *ahm-invar-impl-of* [*simp*, *intro*]: *ArrayHashMap-Impl.ahm-invar* (*impl-of* *hm*)
⟨*proof*⟩

lemma *HashMap-impl-of* [*code abstype*]: *HashMap* (*impl-of t*) = *t*
⟨*proof*⟩

4.12.2 Primitive operations

definition *ahm-empty-const* :: (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty-const* ≡ (*HashMap* (*ArrayHashMap-Impl.ahm-empty* ()))

definition *ahm-empty* :: *unit* ⇒ (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty* ≡ λ-. *ahm-empty-const*

definition *ahm-α* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'key* ⇒ *'val option*
where *ahm-α* *hm* = *ArrayHashMap-Impl.ahm-α* (*impl-of hm*)

definition *ahm-lookup* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'val option*
where *ahm-lookup k hm* = *ArrayHashMap-Impl.ahm-lookup k* (*impl-of hm*)

definition *ahm-iteratei* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key* × *'val*, *'σ*) *set-iterator*
where *ahm-iteratei hm* = *ArrayHashMap-Impl.ahm-iteratei* (*impl-of hm*)

definition *ahm-update* :: *'key* ⇒ *'val* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-update k v hm = *HashMap* (*ArrayHashMap-Impl.ahm-update k v* (*impl-of* *hm*))

definition *ahm-delete* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*)

hashmap
where
ahm-delete k hm = HashMap (ArrayHashMap-Impl.ahm-delete k (impl-of hm))

lemma *impl-of-ahm-empty* [*code abstract*]:
impl-of ahm-empty-const = ArrayHashMap-Impl.ahm-empty ()
⟨*proof*⟩

lemma *impl-of-ahm-update* [*code abstract*]:
impl-of (ahm-update k v hm) = ArrayHashMap-Impl.ahm-update k v (impl-of hm)
⟨*proof*⟩

lemma *impl-of-ahm-delete* [*code abstract*]:
impl-of (ahm-delete k hm) = ArrayHashMap-Impl.ahm-delete k (impl-of hm)
⟨*proof*⟩

4.12.3 Derived operations.

abbreviation (*input*) *ahm-invar* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *bool*
where *ahm-invar* ≡ λ-. *True*

Implementation for *ahm-update-dj* and *ahm-add-dj* could be more efficient: Adjusting the size field of the hashmap need not check whether the key was in the domain before and for we could just use Cons for updating the bucket.

definition *ahm-update-dj* == *ahm-update*

definition *ahm-add* == *it-add ahm-update ahm-iteratei*
definition *ahm-add-dj* == *ahm-add*
definition *ahm-isEmpty* == *iti-isEmpty ahm-iteratei*
definition *ahm-sel* == *iti-sel ahm-iteratei*
definition *ahm-sel'* == *iti-sel-no-map ahm-iteratei*

definition *ahm-to-list* == *it-map-to-list ahm-iteratei*
definition *list-to-ahm* == *gen-list-to-map ahm-empty ahm-update*

lemmas *ahm-defs* =
ahm-α-def
ahm-empty-def
ahm-lookup-def
ahm-update-def
ahm-update-dj-def
ahm-delete-def
ahm-iteratei-def
ahm-add-def
ahm-add-dj-def
ahm-isEmpty-def
ahm-sel-def

ahm-sel′-def
ahm-to-list-def
list-to-ahm-def

4.12.4 Correctness

lemma *finite-dom-ahm-α*:
finite (*dom* (*ahm-α hm*))
⟨*proof*⟩

lemma *finite-map-ahm-α*:
finite-map ahm-α ahm-invar
⟨*proof*⟩

interpretation *ahm*!: *finite-map ahm-α ahm-invar*
⟨*proof*⟩

lemma *ahm-empty-correct* [*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
⟨*proof*⟩

lemma *ahm-empty-impl*: *map-empty ahm-α ahm-invar ahm-empty*
⟨*proof*⟩

interpretation *ahm*!: *map-empty ahm-α ahm-invar ahm-empty* ⟨*proof*⟩

lemma *ahm-lookup-impl*: *map-lookup ahm-α ahm-invar ahm-lookup*
⟨*proof*⟩

interpretation *ahm*!: *map-lookup ahm-α ahm-invar ahm-lookup* ⟨*proof*⟩

lemma *ahm-iteratei-impl*:
map-iteratei ahm-α ahm-invar ahm-iteratei
⟨*proof*⟩

interpretation *ahm*!: *map-iteratei ahm-α ahm-invar ahm-iteratei*
⟨*proof*⟩

lemma *ahm-update-correct*: *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *ahm-update-impl*:
map-update ahm-α ahm-invar ahm-update
⟨*proof*⟩

interpretation *ahm*!: *map-update ahm-α ahm-invar ahm-update* ⟨*proof*⟩

lemma *ahm-delete-correct*:
ahm-α (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*})
⟨*proof*⟩

lemma *ahm-delete-impl*:
map-delete ahm-α ahm-invar ahm-delete
⟨*proof*⟩

interpretation *ahm*!: *map-delete ahm-α ahm-invar ahm-delete* ⟨*proof*⟩

lemma *ahm-update-dj-impl*: *map-update-dj ahm-α ahm-invar ahm-update-dj*
⟨*proof*⟩

lemma *ahm-add-impl*: *map-add ahm-α ahm-invar ahm-add*
⟨*proof*⟩

lemma *ahm-add-dj-impl*: *map-add-dj ahm-α ahm-invar ahm-add-dj*
⟨*proof*⟩

lemma *ahm-isEmpty-impl*: *map-isEmpty ahm-α ahm-invar ahm-isEmpty*
⟨*proof*⟩

lemma *ahm-sel-impl*: *map-sel ahm-α ahm-invar ahm-sel*
⟨*proof*⟩

lemma *ahm-sel′-impl*: *map-sel′ ahm-α ahm-invar ahm-sel′*
⟨*proof*⟩

lemma *ahm-to-list-impl*: *map-to-list ahm-α ahm-invar ahm-to-list*
⟨*proof*⟩

lemma *list-to-ahm-impl*: *list-to-map ahm-α ahm-invar list-to-ahm*
⟨*proof*⟩

interpretation *ahm*!: *map-update-dj ahm-α ahm-invar ahm-update-dj*
⟨*proof*⟩
interpretation *ahm*!: *map-add ahm-α ahm-invar ahm-add*
⟨*proof*⟩
interpretation *ahm*!: *map-add-dj ahm-α ahm-invar ahm-add-dj*
⟨*proof*⟩
interpretation *ahm*!: *map-isEmpty ahm-α ahm-invar ahm-isEmpty*
⟨*proof*⟩
interpretation *ahm*!: *map-sel ahm-α ahm-invar ahm-sel*
⟨*proof*⟩
interpretation *ahm*!: *map-sel′ ahm-α ahm-invar ahm-sel′*
⟨*proof*⟩
interpretation *ahm*!: *map-to-list ahm-α ahm-invar ahm-to-list*
⟨*proof*⟩
interpretation *ahm*!: *list-to-map ahm-α ahm-invar list-to-ahm*
⟨*proof*⟩

declare *ahm.finite*[*simp del*, *rule del*]

lemmas *ahm-correct* =
ahm.empty-correct
ahm.lookup-correct
ahm.update-correct
ahm.update-dj-correct
ahm.delete-correct
ahm.add-correct
ahm.add-dj-correct
ahm.isEmpty-correct
ahm.to-list-correct
ahm.to-map-correct

4.12.5 Code Generation

export-code
ahm-empty
ahm-lookup
ahm-update
ahm-update-dj
ahm-delete
ahm-iteratei
ahm-add
ahm-add-dj
ahm-isEmpty
ahm-sel
ahm-sel′
ahm-to-list
list-to-ahm
in *SML*
module-name *ArrayHashMap*
file −

end

4.13 Maps from Naturals by Arrays

theory *ArrayMapImpl*
imports
../spec/MapSpec
../gen-algo/MapGA
../common/Array
begin **type-synonym** *'v iam* = *'v option array*

4.13.1 Definitions

definition *iam-α* :: *'v iam* ⇒ *nat* ⇀ *'v* **where**

```
  iam-α a i ≡ if i < array-length a then array-get a i else None

abbreviation iam-invar :: 'v iam ⇒ bool where iam-invar ≡ λ-. True

definition iam-empty :: unit ⇒ 'v iam
  where iam-empty ≡ λ-::unit. array-of-list []

definition iam-lookup :: nat ⇒ 'v iam ⇀ 'v
  where iam-lookup k a ≡ iam-α a k

definition iam-increment (l::nat) idx ≡
  max (idx + 1 − l) (2 ∗ l + 3)

lemma incr-correct: ¬ idx < l ⟹ idx < l + iam-increment l idx
  ⟨proof⟩

definition iam-update :: nat ⇒ 'v ⇒ 'v iam ⇒ 'v iam
  where iam-update k v a ≡ let
  l = array-length a;
  a = if k < l then a else array-grow a (iam-increment l k) None
in
  array-set a k (Some v)


definition iam-update-dj ≡ iam-update

definition iam-delete :: nat ⇒ 'v iam ⇒ 'v iam
  where iam-delete k a ≡
  if k<array-length a then array-set a k None else a

fun iam-iteratei-aux
  :: nat ⇒ ('v iam) ⇒ ('σ⇒bool) ⇒ (nat × 'v⇒'σ⇒'σ) ⇒ 'σ ⇒ 'σ
  where
    iam-iteratei-aux 0 a c f σ = σ
  | iam-iteratei-aux i a c f σ = (
      if c σ then
        iam-iteratei-aux (i − 1) a c f (
          case array-get a (i − 1) of None ⇒ σ | Some x ⇒ f (i − 1, x) σ
        )
      else σ)

definition iam-iteratei :: 'v iam ⇒ (nat × 'v,'σ) set-iterator where
  iam-iteratei a = iam-iteratei-aux (array-length a) a

definition iam-add == it-add iam-update iam-iteratei
definition iam-add-dj == iam-add
definition iam-isEmpty == iti-isEmpty iam-iteratei
definition iam-sel == iti-sel iam-iteratei
definition iam-sel' = iti-sel-no-map iam-iteratei
```

definition *iam-to-list* == *it-map-to-list iam-iteratei*
definition *list-to-iam* == *gen-list-to-map iam-empty iam-update*

4.13.2 Correctness

lemmas *iam-defs* =
iam-α-def
iam-empty-def
iam-lookup-def
iam-update-def
iam-update-dj-def
iam-delete-def
iam-iteratei-def
iam-add-def
iam-add-dj-def
iam-isEmpty-def
iam-sel-def
iam-sel′-def
iam-to-list-def
list-to-iam-def

lemma *iam-empty-impl*: *map-empty iam-α iam-invar iam-empty*
⟨*proof*⟩

interpretation *iam*!: *map-empty iam-α iam-invar iam-empty*
⟨*proof*⟩

lemma *iam-lookup-impl*: *map-lookup iam-α iam-invar iam-lookup*
⟨*proof*⟩
interpretation *iam*!: *map-lookup iam-α iam-invar iam-lookup*
⟨*proof*⟩

lemma *array-get-set-iff*: *i*<*array-length a* ⟹
array-get (*array-set a i x*) *j* = (*if i*=*j then x else array-get a j*)
⟨*proof*⟩

lemma *iam-update-impl*: *map-update iam-α iam-invar iam-update*
⟨*proof*⟩
interpretation *iam*!: *map-update iam-α iam-invar iam-update*
⟨*proof*⟩

lemma *iam-update-dj-impl*: *map-update-dj iam-α iam-invar iam-update-dj*
⟨*proof*⟩
interpretation *iam*!: *map-update-dj iam-α iam-invar iam-update-dj*
⟨*proof*⟩

lemma *iam-delete-impl*: *map-delete iam-α iam-invar iam-delete*

⟨*proof*⟩
interpretation *iam*!: *map-delete iam-α iam-invar iam-delete*
⟨*proof*⟩

lemma *iam-iteratei-aux-foldli-conv* :
iam-iteratei-aux n a =
foldli (*List.map-filter* (λ*n*. *Option.map* (λ*v*. (*n*, *v*)) (*array-get a n*)) (*rev* [*0*..<*n*]))
⟨*proof*⟩

lemma *iam-iteratei-foldli-conv* :
iam-iteratei a =
foldli (*List.map-filter* (λ*n*. *Option.map* (λ*v*. (*n*, *v*)) (*array-get a n*)) (*rev* [*0*..<(*array-length a*)]))
⟨*proof*⟩

lemma *iam-iteratei-correct* :
fixes *m*::′*a option array*
defines *kvs* ≡ *List.map-filter* (λ*n*. *Option.map* (λ*v*. (*n*, *v*)) (*array-get m n*)) (*rev* [*0*..<(*array-length m*)])
shows *map-iterator-rev-linord* (*iam-iteratei m*) (*iam-α m*)
⟨*proof*⟩

lemma *iam-iteratei-rev-linord-impl*: *map-reverse-iterateoi iam-α iam-invar iam-iteratei*
⟨*proof*⟩

interpretation *iam*!: *map-reverse-iterateoi iam-α iam-invar iam-iteratei*
⟨*proof*⟩

lemma *iam-iteratei-impl*: *map-iteratei iam-α iam-invar iam-iteratei*
⟨*proof*⟩
interpretation *iam*!: *map-iteratei iam-α iam-invar iam-iteratei*
⟨*proof*⟩

lemma *iam-add-impl*: *map-add iam-α iam-invar iam-add*
⟨*proof*⟩
interpretation *iam*!: *map-add iam-α iam-invar iam-add* ⟨*proof*⟩

lemma *iam-add-dj-impl*: *map-add-dj iam-α iam-invar iam-add-dj*
⟨*proof*⟩
interpretation *iam*!: *map-add-dj iam-α iam-invar iam-add-dj*
⟨*proof*⟩

lemma *iam-isEmpty-impl*: *map-isEmpty iam-α iam-invar iam-isEmpty*
⟨*proof*⟩
interpretation *iam*!: *map-isEmpty iam-α iam-invar iam-isEmpty*
⟨*proof*⟩

lemma *iam-sel-impl*: *map-sel iam-α iam-invar iam-sel*

⟨*proof*⟩
interpretation *iam*!: *map-sel iam-α iam-invar iam-sel*
⟨*proof*⟩

lemma *iam-sel′-impl*: *map-sel′ iam-α iam-invar iam-sel′*
⟨*proof*⟩
interpretation *iam*!: *map-sel′ iam-α iam-invar iam-sel′*
⟨*proof*⟩

lemma *iam-to-list-impl*: *map-to-list iam-α iam-invar iam-to-list*
⟨*proof*⟩
interpretation *iam*!: *map-to-list iam-α iam-invar iam-to-list*
⟨*proof*⟩

lemma *list-to-iam-impl*: *list-to-map iam-α iam-invar list-to-iam*
⟨*proof*⟩
interpretation *iam*!: *list-to-map iam-α iam-invar list-to-iam*
⟨*proof*⟩

declare *iam.finite*[*simp del*, *rule del*]

lemmas *iam-correct* =
iam.empty-correct
iam.lookup-correct
iam.update-correct
iam.update-dj-correct
iam.delete-correct
iam.add-correct
iam.add-dj-correct
iam.isEmpty-correct
iam.to-list-correct
iam.to-map-correct

4.13.3 Code Generation

export-code
iam-empty
iam-lookup
iam-update
iam-update-dj
iam-delete
iam-iteratei
iam-add
iam-add-dj
iam-isEmpty
iam-sel
iam-sel′
iam-to-list
list-to-iam

in *SML*
module-name *ArrayMap*
file –

end

Standard Implementations of Maps **theory** *MapStdImpl*
imports
ListMapImpl ListMapImpl-Invar RBTMapImpl HashMap TrieMapImpl ArrayHashMap ArrayMapImpl
begin

This theory summarizes various standard implementation of maps, namely list-maps, RB-tree-maps, trie-maps, hashmaps, indexed array maps.

end

4.14 Set Implementation by List

theory *ListSetImpl*
imports *../spec/SetSpec ../gen-algo/SetGA ../common/Dlist-add*
begin type-synonym
'a ls = 'a dlist

4.14.1 Definitions

definition *ls-α :: 'a ls ⇒ 'a set* **where** *ls-α l == set (list-of-dlist l)*
abbreviation *(input) ls-invar :: 'a ls ⇒ bool* **where** *ls-invar == λ-. True*
definition *ls-empty :: unit ⇒ 'a ls* **where** *ls-empty == (λ-. Dlist.empty)*
definition *ls-memb :: 'a ⇒ 'a ls ⇒ bool* **where** *ls-memb x l == Dlist.member l x*
definition *ls-ins :: 'a ⇒ 'a ls ⇒ 'a ls* **where** *ls-ins == Dlist.insert*

Since we use the abstract type *'a dlist* for distinct lists, to preserve the invariant of distinct lists, we cannot just use Cons for disjoint insert, but must resort to ordinary insert. The same applies to *ls-union-dj* below. *list-to-lm-dj* below.

definition *ls-ins-dj :: 'a ⇒ 'a ls ⇒ 'a ls* **where** *ls-ins-dj == Dlist.insert*
definition *ls-delete :: 'a ⇒ 'a ls ⇒ 'a ls*
where *ls-delete == dlist-remove'*
definition *ls-iteratei :: 'a ls ⇒ ('a,'σ) set-iterator*
where *ls-iteratei == dlist-iteratei*

definition *ls-isEmpty :: 'a ls ⇒ bool* **where** *ls-isEmpty == Dlist.null*

definition *ls-union :: 'a ls ⇒ 'a ls ⇒ 'a ls*
where *ls-union == it-union ls-iteratei ls-ins*
definition *ls-inter :: 'a ls ⇒ 'a ls ⇒ 'a ls*
where *ls-inter == it-inter ls-iteratei ls-memb ls-empty ls-ins-dj*

definition *ls-union-dj* :: *'a ls* ⇒ *'a ls* ⇒ *'a ls*
where *ls-union-dj* == *ls-union*

definition *ls-image-filter*
where *ls-image-filter* == *it-image-filter ls-iteratei ls-empty ls-ins*

definition *ls-inj-image-filter*
where *ls-inj-image-filter* == *it-inj-image-filter ls-iteratei ls-empty ls-ins-dj*

definition *ls-image* == *iflt-image ls-image-filter*
definition *ls-inj-image* == *iflt-inj-image ls-inj-image-filter*

definition *ls-sel* :: *'a ls* ⇒ (*'a* ⇒ *'r option*) ⇒ *'r option*
where *ls-sel* == *iti-sel ls-iteratei*
definition *ls-sel'* == *iti-sel-no-map ls-iteratei*

definition *ls-to-list* :: *'a ls* ⇒ *'a list* **where** *ls-to-list* == *list-of-dlist*
definition *list-to-ls* :: *'a list* ⇒ *'a ls* **where** *list-to-ls* == *Dlist*

4.14.2 Correctness

lemmas *ls-defs* =
ls-α-def
ls-empty-def
ls-memb-def
ls-ins-def
ls-ins-dj-def
ls-delete-def
ls-iteratei-def
ls-isEmpty-def
ls-union-def
ls-inter-def
ls-union-dj-def
ls-image-filter-def
ls-inj-image-filter-def
ls-image-def
ls-inj-image-def
ls-sel-def
ls-sel'-def
ls-to-list-def
list-to-ls-def

lemma *ls-empty-impl*: *set-empty ls-α ls-invar ls-empty*
⟨*proof*⟩

lemma *ls-memb-impl*: *set-memb ls-α ls-invar ls-memb*
⟨*proof*⟩

lemma *ls-ins-impl*: *set-ins ls-α ls-invar ls-ins*
⟨*proof*⟩

lemma *ls-ins-dj-impl*: *set-ins-dj ls-α ls-invar ls-ins-dj*
⟨*proof*⟩

lemma *ls-delete-impl*: *set-delete ls-α ls-invar ls-delete*
⟨*proof*⟩

lemma *ls-α-finite*[*simp*, *intro*!]: *finite* (*ls-α l*)
⟨*proof*⟩

lemma *ls-is-finite-set*: *finite-set ls-α ls-invar*
⟨*proof*⟩

lemma *ls-iteratei-impl*: *set-iteratei ls-α ls-invar ls-iteratei*
⟨*proof*⟩

lemma *ls-isEmpty-impl*: *set-isEmpty ls-α ls-invar ls-isEmpty*
⟨*proof*⟩

lemmas *ls-union-impl* = *it-union-correct*[*OF ls-iteratei-impl ls-ins-impl*, *folded ls-union-def*]

lemmas *ls-inter-impl* = *it-inter-correct*[*OF ls-iteratei-impl ls-memb-impl ls-empty-impl ls-ins-dj-impl*, *folded ls-inter-def*]

lemmas *ls-union-dj-impl* = *set-union.union-dj-by-union*[*OF ls-union-impl*, *folded ls-union-dj-def*]

lemmas *ls-image-filter-impl* = *it-image-filter-correct*[*OF ls-iteratei-impl ls-empty-impl ls-ins-impl*, *folded ls-image-filter-def*]
lemmas *ls-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF ls-iteratei-impl ls-empty-impl ls-ins-dj-impl*, *folded ls-inj-image-filter-def*]

lemmas *ls-image-impl* = *iflt-image-correct*[*OF ls-image-filter-impl*, *folded ls-image-def*]
lemmas *ls-inj-image-impl* = *iflt-inj-image-correct*[*OF ls-inj-image-filter-impl*, *folded ls-inj-image-def*]

lemmas *ls-sel-impl* = *iti-sel-correct*[*OF ls-iteratei-impl*, *folded ls-sel-def*]
lemmas *ls-sel′-impl* = *iti-sel′-correct*[*OF ls-iteratei-impl*, *folded ls-sel′-def*]

lemma *ls-to-list-impl*: *set-to-list ls-α ls-invar ls-to-list*
⟨*proof*⟩

lemma *list-to-ls-impl*: *list-to-set ls-α ls-invar list-to-ls*
⟨*proof*⟩

interpretation *ls*: *set-empty ls-α ls-invar ls-empty* ⟨*proof*⟩
interpretation *ls*: *set-memb ls-α ls-invar ls-memb* ⟨*proof*⟩
interpretation *ls*: *set-ins ls-α ls-invar ls-ins* ⟨*proof*⟩
interpretation *ls*: *set-ins-dj ls-α ls-invar ls-ins-dj* ⟨*proof*⟩
interpretation *ls*: *set-delete ls-α ls-invar ls-delete* ⟨*proof*⟩
interpretation *ls*: *finite-set ls-α ls-invar* ⟨*proof*⟩
interpretation *ls*: *set-iteratei ls-α ls-invar ls-iteratei* ⟨*proof*⟩
interpretation *ls*: *set-isEmpty ls-α ls-invar ls-isEmpty* ⟨*proof*⟩
interpretation *ls*: *set-union ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-union* ⟨*proof*⟩
interpretation *ls*: *set-inter ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-inter* ⟨*proof*⟩
interpretation *ls*: *set-union-dj ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-union-dj* ⟨*proof*⟩
interpretation *ls*: *set-image-filter ls-α ls-invar ls-α ls-invar ls-image-filter* ⟨*proof*⟩
interpretation *ls*: *set-inj-image-filter ls-α ls-invar ls-α ls-invar ls-inj-image-filter* ⟨*proof*⟩
interpretation *ls*: *set-image ls-α ls-invar ls-α ls-invar ls-image* ⟨*proof*⟩
interpretation *ls*: *set-inj-image ls-α ls-invar ls-α ls-invar ls-inj-image* ⟨*proof*⟩
interpretation *ls*: *set-sel ls-α ls-invar ls-sel* ⟨*proof*⟩
interpretation *ls*: *set-sel′ ls-α ls-invar ls-sel′* ⟨*proof*⟩
interpretation *ls*: *set-to-list ls-α ls-invar ls-to-list* ⟨*proof*⟩
interpretation *ls*: *list-to-set ls-α ls-invar list-to-ls* ⟨*proof*⟩

declare *ls.finite*[*simp del*, *rule del*]

lemmas *ls-correct* =
ls.empty-correct
ls.memb-correct
ls.ins-correct
ls.ins-dj-correct
ls.delete-correct
ls.isEmpty-correct
ls.union-correct
ls.inter-correct
ls.union-dj-correct
ls.image-filter-correct
ls.inj-image-filter-correct
ls.image-correct
ls.inj-image-correct
ls.to-list-correct
ls.to-set-correct

4.14.3 Code Generation

lemma *list-of-dlist-list-to-ls* [*code abstract*]:
list-of-dlist (*list-to-ls xs*) = *remdups xs*
⟨*proof*⟩

export-code
ls-empty

ls-memb
ls-ins
ls-ins-dj
ls-delete
ls-iteratei
ls-isEmpty
ls-union
ls-inter
ls-union-dj
ls-image-filter
ls-inj-image-filter
ls-image
ls-inj-image
ls-sel
ls-sel'
ls-to-list
list-to-ls
in *SML*
module-name *ListSet*
file –

end

4.15 Set Implementation by List with explicit invariants

theory *ListSetImpl-Invar*
imports
../spec/SetSpec
../gen-algo/SetGA
../common/Misc
../common/Dlist-add
begin type-synonym
'a lsi = 'a list

4.15.1 Definitions

definition *lsi-α :: 'a lsi ⇒ 'a set* **where** *lsi-α == set*
definition *lsi-invar :: 'a lsi ⇒ bool* **where** *lsi-invar == distinct*
definition *lsi-empty :: unit ⇒ 'a lsi* **where** *lsi-empty == (λ-::unit. [])*
definition *lsi-memb :: 'a ⇒ 'a lsi ⇒ bool* **where** *lsi-memb x l == List.member l x*
definition *lsi-ins :: 'a ⇒ 'a lsi ⇒ 'a lsi* **where** *lsi-ins x l == if List.member l x then l else x#l*
definition *lsi-ins-dj :: 'a ⇒ 'a lsi ⇒ 'a lsi* **where** *lsi-ins-dj x l == x#l*

definition *lsi-delete :: 'a ⇒ 'a lsi ⇒ 'a lsi* **where** *lsi-delete x l == Dlist-add.dlist-remove1' x l*

x [] l

definition *lsi-iteratei* :: *'a lsi* ⇒ (*'a*,*'σ*) *set-iterator*
where *lsi-iteratei* = *foldli*

definition *lsi-isEmpty* :: *'a lsi* ⇒ *bool* **where** *lsi-isEmpty s* == *s*=[]

definition *lsi-union* :: *'a lsi* ⇒ *'a lsi* ⇒ *'a lsi*
where *lsi-union* == *it-union lsi-iteratei lsi-ins*
definition *lsi-inter* :: *'a lsi* ⇒ *'a lsi* ⇒ *'a lsi*
where *lsi-inter* == *it-inter lsi-iteratei lsi-memb lsi-empty lsi-ins-dj*
definition *lsi-union-dj* :: *'a lsi* ⇒ *'a lsi* ⇒ *'a lsi*
where *lsi-union-dj s1 s2* == *revg s1 s2* — Union of disjoint sets

definition *lsi-image-filter*
where *lsi-image-filter* == *it-image-filter lsi-iteratei lsi-empty lsi-ins*

definition *lsi-inj-image-filter*
where *lsi-inj-image-filter* == *it-inj-image-filter lsi-iteratei lsi-empty lsi-ins-dj*

definition *lsi-image* == *iflt-image lsi-image-filter*
definition *lsi-inj-image* == *iflt-inj-image lsi-inj-image-filter*

definition *lsi-sel* :: *'a lsi* ⇒ (*'a* ⇒ *'r option*) ⇒ *'r option*
where *lsi-sel* == *iti-sel lsi-iteratei*
definition *lsi-sel'* == *iti-sel-no-map lsi-iteratei*

definition *lsi-to-list* :: *'a lsi* ⇒ *'a list* **where** *lsi-to-list* == *id*
definition *list-to-lsi* :: *'a list* ⇒ *'a lsi* **where** *list-to-lsi* == *remdups*

4.15.2 Correctness

lemmas *lsi-defs* =
lsi-α-def
lsi-invar-def
lsi-empty-def
lsi-memb-def
lsi-ins-def
lsi-ins-dj-def
lsi-delete-def
lsi-iteratei-def
lsi-isEmpty-def
lsi-union-def
lsi-inter-def
lsi-union-dj-def
lsi-image-filter-def
lsi-inj-image-filter-def
lsi-image-def
lsi-inj-image-def

lsi-sel-def
lsi-sel'-def
lsi-to-list-def
list-to-lsi-def

lemma *lsi-empty-impl*: *set-empty lsi-α lsi-invar lsi-empty*
⟨*proof*⟩

lemma *lsi-memb-impl*: *set-memb lsi-α lsi-invar lsi-memb*
⟨*proof*⟩

lemma *lsi-ins-impl*: *set-ins lsi-α lsi-invar lsi-ins*
⟨*proof*⟩

lemma *lsi-ins-dj-impl*: *set-ins-dj lsi-α lsi-invar lsi-ins-dj*
⟨*proof*⟩

lemma *lsi-delete-impl*: *set-delete lsi-α lsi-invar lsi-delete*
⟨*proof*⟩

lemma *lsi-α-finite*[*simp*, *intro*!]: *finite* (*lsi-α l*)
⟨*proof*⟩

lemma *lsi-is-finite-set*: *finite-set lsi-α lsi-invar*
⟨*proof*⟩

lemma *lsi-iteratei-impl*: *set-iteratei lsi-α lsi-invar lsi-iteratei*
⟨*proof*⟩

lemma *lsi-isEmpty-impl*: *set-isEmpty lsi-α lsi-invar lsi-isEmpty*
⟨*proof*⟩

lemmas *lsi-union-impl* = *it-union-correct*[*OF lsi-iteratei-impl lsi-ins-impl*, *folded lsi-union-def*]

lemmas *lsi-inter-impl* = *it-inter-correct*[*OF lsi-iteratei-impl lsi-memb-impl lsi-empty-impl lsi-ins-dj-impl*, *folded lsi-inter-def*]

lemma *lsi-union-dj-impl*: *set-union-dj lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-union-dj*
⟨*proof*⟩

lemmas *lsi-image-filter-impl* = *it-image-filter-correct*[*OF lsi-iteratei-impl lsi-empty-impl lsi-ins-impl*, *folded lsi-image-filter-def*]
lemmas *lsi-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF lsi-iteratei-impl lsi-empty-impl lsi-ins-dj-impl*, *folded lsi-inj-image-filter-def*]

lemmas *lsi-image-impl* = *iflt-image-correct*[*OF lsi-image-filter-impl*, *folded lsi-image-def*]
lemmas *lsi-inj-image-impl* = *iflt-inj-image-correct*[*OF lsi-inj-image-filter-impl*, *folded*

lsi-inj-image-def]

lemmas *lsi-sel-impl* = *iti-sel-correct*[*OF lsi-iteratei-impl*, *folded lsi-sel-def*]
lemmas *lsi-sel'-impl* = *iti-sel'-correct*[*OF lsi-iteratei-impl*, *folded lsi-sel'-def*]

lemma *lsi-to-list-impl*: *set-to-list lsi-α lsi-invar lsi-to-list*
⟨*proof*⟩

lemma *list-to-lsi-impl*: *list-to-set lsi-α lsi-invar list-to-lsi*
⟨*proof*⟩

interpretation *lsi*: *set-empty lsi-α lsi-invar lsi-empty* ⟨*proof*⟩
interpretation *lsi*: *set-memb lsi-α lsi-invar lsi-memb* ⟨*proof*⟩
interpretation *lsi*: *set-ins lsi-α lsi-invar lsi-ins* ⟨*proof*⟩
interpretation *lsi*: *set-ins-dj lsi-α lsi-invar lsi-ins-dj* ⟨*proof*⟩
interpretation *lsi*: *set-delete lsi-α lsi-invar lsi-delete* ⟨*proof*⟩
interpretation *lsi*: *finite-set lsi-α lsi-invar* ⟨*proof*⟩
interpretation *lsi*: *set-iteratei lsi-α lsi-invar lsi-iteratei* ⟨*proof*⟩
interpretation *lsi*: *set-isEmpty lsi-α lsi-invar lsi-isEmpty* ⟨*proof*⟩
interpretation *lsi*: *set-union lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-union*
⟨*proof*⟩
interpretation *lsi*: *set-inter lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-inter*
⟨*proof*⟩
interpretation *lsi*: *set-union-dj lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-union-dj*
⟨*proof*⟩
interpretation *lsi*: *set-image-filter lsi-α lsi-invar lsi-α lsi-invar lsi-image-filter*
⟨*proof*⟩
interpretation *lsi*: *set-inj-image-filter lsi-α lsi-invar lsi-α lsi-invar lsi-inj-image-filter*
⟨*proof*⟩
interpretation *lsi*: *set-image lsi-α lsi-invar lsi-α lsi-invar lsi-image* ⟨*proof*⟩
interpretation *lsi*: *set-inj-image lsi-α lsi-invar lsi-α lsi-invar lsi-inj-image* ⟨*proof*⟩
interpretation *lsi*: *set-sel lsi-α lsi-invar lsi-sel* ⟨*proof*⟩
interpretation *lsi*: *set-sel' lsi-α lsi-invar lsi-sel'* ⟨*proof*⟩
interpretation *lsi*: *set-to-list lsi-α lsi-invar lsi-to-list* ⟨*proof*⟩
interpretation *lsi*: *list-to-set lsi-α lsi-invar list-to-lsi* ⟨*proof*⟩

declare *lsi.finite*[*simp del*, *rule del*]

lemmas *lsi-correct* =
lsi.empty-correct
lsi.memb-correct
lsi.ins-correct
lsi.ins-dj-correct
lsi.delete-correct
lsi.isEmpty-correct
lsi.union-correct
lsi.inter-correct
lsi.union-dj-correct
lsi.image-filter-correct

lsi.inj-image-filter-correct
lsi.image-correct
lsi.inj-image-correct
lsi.to-list-correct
lsi.to-set-correct

4.15.3 Code Generation

export-code
lsi-empty
lsi-memb
lsi-ins
lsi-ins-dj
lsi-delete
lsi-iteratei
lsi-isEmpty
lsi-union
lsi-inter
lsi-union-dj
lsi-image-filter
lsi-inj-image-filter
lsi-image
lsi-inj-image
lsi-sel
lsi-sel'
lsi-to-list
list-to-lsi
in *SML*
module-name *ListSet*
file −

end

4.16 Set Implementation by Red-Black-Tree

theory *RBTSetImpl*
imports
../spec/SetSpec
RBTMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

4.16.1 Definitions

type-synonym
$'a\ rs = ('a::linorder, unit)\ rm$

definition *rs-α* :: *'a::linorder rs* ⇒ *'a set* **where** *rs-α* == *s-α rm-α*
abbreviation (*input*) *rs-invar* :: *'a::linorder rs* ⇒ *bool* **where** *rs-invar* == *rm-invar*
definition *rs-empty* :: *unit* ⇒ *'a::linorder rs* **where** *rs-empty* == *s-empty rm-empty*
definition *rs-memb* :: *'a::linorder* ⇒ *'a rs* ⇒ *bool*
  where *rs-memb* == *s-memb rm-lookup*
definition *rs-ins* :: *'a::linorder* ⇒ *'a rs* ⇒ *'a rs*
  where *rs-ins* == *s-ins rm-update*
definition *rs-ins-dj* :: *'a::linorder* ⇒ *'a rs* ⇒ *'a rs* **where**
  rs-ins-dj == *s-ins-dj rm-update-dj*
definition *rs-delete* :: *'a::linorder* ⇒ *'a rs* ⇒ *'a rs*
  where *rs-delete* == *s-delete rm-delete*
definition *rs-sel* :: *'a::linorder rs* ⇒ (*'a* ⇒ *'r option*) ⇒ *'r option*
  where *rs-sel* == *s-sel rm-sel*
definition *rs-sel'* = *sel-sel' rs-sel*
definition *rs-isEmpty* :: *'a::linorder rs* ⇒ *bool*
  where *rs-isEmpty* == *s-isEmpty rm-isEmpty*

definition *rs-iteratei* :: *'a::linorder rs* ⇒ (*'a*,*'σ*) *set-iterator*
  where *rs-iteratei* == *s-iteratei rm-iteratei*

definition *rs-iterateoi* :: *'a::linorder rs* ⇒ (*'a*,*'σ*) *set-iterator*
  where *rs-iterateoi* == *s-iteratei rm-iterateoi*

definition *rs-reverse-iterateoi* :: *'a::linorder rs* ⇒ (*'a*,*'σ*) *set-iterator*
  where *rs-reverse-iterateoi* == *s-iteratei rm-reverse-iterateoi*

definition *rs-union* :: *'a::linorder rs* ⇒ *'a rs* ⇒ *'a rs*
  where *rs-union* == *s-union rm-add*
definition *rs-union-dj* :: *'a::linorder rs* ⇒ *'a rs* ⇒ *'a rs*
  where *rs-union-dj* == *s-union-dj rm-add-dj*
definition *rs-inter* :: *'a::linorder rs* ⇒ *'a rs* ⇒ *'a rs*
  where *rs-inter* == *it-inter rs-iteratei rs-memb rs-empty rs-ins-dj*

definition *rs-image-filter*
  where *rs-image-filter* == *it-image-filter rs-iteratei rs-empty rs-ins*
definition *rs-inj-image-filter*
  where *rs-inj-image-filter* == *it-inj-image-filter rs-iteratei rs-empty rs-ins-dj*

definition *rs-image* **where** *rs-image* == *iflt-image rs-image-filter*
definition *rs-inj-image* **where** *rs-inj-image* == *iflt-inj-image rs-inj-image-filter*

definition *rs-to-list* == *it-set-to-list rs-reverse-iterateoi*
definition *list-to-rs* == *gen-list-to-set rs-empty rs-ins*

definition *rs-min* == *iti-sel-no-map rs-iterateoi*
definition *rs-max* == *iti-sel-no-map rs-reverse-iterateoi*

4.16.2 Correctness

lemmas *rs-defs* =
list-to-rs-def
rs-α-def
rs-delete-def
rs-empty-def
rs-image-def
rs-image-filter-def
rs-inj-image-def
rs-inj-image-filter-def
rs-ins-def
rs-ins-dj-def
rs-inter-def
rs-isEmpty-def
rs-iteratei-def
rs-iterateoi-def
rs-reverse-iterateoi-def
rs-memb-def
rs-sel-def
rs-sel′-def
rs-to-list-def
rs-union-def
rs-union-dj-def
rs-min-def
rs-max-def

lemmas *rs-empty-impl* = *s-empty-correct*[*OF rm-empty-impl*, *folded rs-defs*]
lemmas *rs-memb-impl* = *s-memb-correct*[*OF rm-lookup-impl*, *folded rs-defs*]
lemmas *rs-ins-impl* = *s-ins-correct*[*OF rm-update-impl*, *folded rs-defs*]
lemmas *rs-ins-dj-impl* = *s-ins-dj-correct*[*OF rm-update-dj-impl*, *folded rs-defs*]
lemmas *rs-delete-impl* = *s-delete-correct*[*OF rm-delete-impl*, *folded rs-defs*]
lemmas *rs-iteratei-impl* = *s-iteratei-correct*[*OF rm-iteratei-impl*, *folded rs-defs*]
lemmas *rs-iterateoi-impl* = *s-iterateoi-correct*[*OF rm-iterateoi-impl*, *folded rs-defs*]
lemmas *rs-reverse-iterateoi-impl* = *s-reverse-iterateoi-correct*[*OF rm-reverse-iterateoi-impl*, *folded rs-defs*]
lemmas *rs-isEmpty-impl* = *s-isEmpty-correct*[*OF rm-isEmpty-impl*, *folded rs-defs*]
lemmas *rs-union-impl* = *s-union-correct*[*OF rm-add-impl*, *folded rs-defs*]
lemmas *rs-union-dj-impl* = *s-union-dj-correct*[*OF rm-add-dj-impl*, *folded rs-defs*]
lemmas *rs-sel-impl* = *s-sel-correct*[*OF rm-sel-impl*, *folded rs-defs*]
lemmas *rs-sel′-impl* = *sel-sel′-correct*[*OF rs-sel-impl*, *folded rs-sel′-def*]
lemmas *rs-inter-impl* = *it-inter-correct*[*OF rs-iteratei-impl rs-memb-impl rs-empty-impl rs-ins-dj-impl*, *folded rs-inter-def*]
lemmas *rs-image-filter-impl* = *it-image-filter-correct*[*OF rs-iteratei-impl rs-empty-impl rs-ins-impl*, *folded rs-image-filter-def*]
lemmas *rs-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF rs-iteratei-impl rs-empty-impl rs-ins-dj-impl*, *folded rs-inj-image-filter-def*]
lemmas *rs-image-impl* = *iflt-image-correct*[*OF rs-image-filter-impl*, *folded rs-image-def*]
lemmas *rs-inj-image-impl* = *iflt-inj-image-correct*[*OF rs-inj-image-filter-impl*, *folded*

rs-inj-image-def]
lemmas *rs-to-list-impl* = *rito-set-to-sorted-list-correct*[*OF rs-reverse-iterateoi-impl*, *folded rs-to-list-def*]
lemmas *list-to-rs-impl* = *gen-list-to-set-correct*[*OF rs-empty-impl rs-ins-impl*, *folded list-to-rs-def*]

lemmas *rs-min-impl* = *itoi-min-correct*[*OF rs-iterateoi-impl*, *folded rs-defs*]
lemmas *rs-max-impl* = *ritoi-max-correct*[*OF rs-reverse-iterateoi-impl*, *folded rs-defs*]

interpretation *rs*: *set-iteratei rs-α rs-invar rs-iteratei* ⟨*proof*⟩
interpretation *rs*: *set-iterateoi rs-α rs-invar rs-iterateoi* ⟨*proof*⟩
interpretation *rs*: *set-reverse-iterateoi rs-α rs-invar rs-reverse-iterateoi* ⟨*proof*⟩

interpretation *rs*: *set-inj-image-filter rs-α rs-invar rs-α rs-invar rs-inj-image-filter* ⟨*proof*⟩
interpretation *rs*: *set-image-filter rs-α rs-invar rs-α rs-invar rs-image-filter* ⟨*proof*⟩
interpretation *rs*: *set-inj-image rs-α rs-invar rs-α rs-invar rs-inj-image* ⟨*proof*⟩
interpretation *rs*: *set-union-dj rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-union-dj* ⟨*proof*⟩
interpretation *rs*: *set-union rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-union* ⟨*proof*⟩
interpretation *rs*: *set-inter rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-inter* ⟨*proof*⟩
interpretation *rs*: *set-image rs-α rs-invar rs-α rs-invar rs-image* ⟨*proof*⟩
interpretation *rs*: *set-sel rs-α rs-invar rs-sel* ⟨*proof*⟩
interpretation *rs*: *set-sel′ rs-α rs-invar rs-sel′* ⟨*proof*⟩
interpretation *rs*: *set-ins-dj rs-α rs-invar rs-ins-dj* ⟨*proof*⟩
interpretation *rs*: *set-delete rs-α rs-invar rs-delete* ⟨*proof*⟩
interpretation *rs*: *set-ins rs-α rs-invar rs-ins* ⟨*proof*⟩
interpretation *rs*: *set-memb rs-α rs-invar rs-memb* ⟨*proof*⟩
interpretation *rs*: *set-to-sorted-list rs-α rs-invar rs-to-list* ⟨*proof*⟩
interpretation *rs*: *list-to-set rs-α rs-invar list-to-rs* ⟨*proof*⟩
interpretation *rs*: *set-isEmpty rs-α rs-invar rs-isEmpty* ⟨*proof*⟩
interpretation *rs*: *set-empty rs-α rs-invar rs-empty* ⟨*proof*⟩
interpretation *rs*: *set-min rs-α rs-invar rs-min* ⟨*proof*⟩
interpretation *rs*: *set-max rs-α rs-invar rs-max* ⟨*proof*⟩

lemmas *rs-correct* =
rs.inj-image-filter-correct
rs.image-filter-correct
rs.inj-image-correct
rs.union-dj-correct
rs.union-correct
rs.inter-correct
rs.image-correct
rs.ins-dj-correct
rs.delete-correct
rs.ins-correct

rs.memb-correct
rs.to-list-correct
rs.to-set-correct
rs.isEmpty-correct
rs.empty-correct

4.16.3 Code Generation

export-code
rs-iteratei
rs-iterateoi
rs-reverse-iterateoi
rs-inj-image-filter
rs-image-filter
rs-inj-image
rs-union-dj
rs-union
rs-inter
rs-image
rs-sel
rs-sel′
rs-ins-dj
rs-delete
rs-ins
rs-memb
rs-to-list
list-to-rs
rs-isEmpty
rs-empty
rs-min
rs-max
in *SML*
module-name *RBTSet*
file −

end

4.17 Hash Set

theory *HashSet*
imports
../spec/SetSpec
HashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

4.17.1 Definitions

type-synonym
'a hs = (*'a*::*hashable*,*unit*) *hm*

definition *hs-α* :: *'a*::*hashable hs* ⇒ *'a set* **where** *hs-α* == *s-α hm-α*
abbreviation (*input*) *hs-invar* :: *'a*::*hashable hs* ⇒ *bool* **where** *hs-invar* == *λ-. True*
definition *hs-empty* :: *unit* ⇒ *'a*::*hashable hs* **where** *hs-empty* == *s-empty hm-empty*
definition *hs-memb* :: *'a*::*hashable* ⇒ *'a hs* ⇒ *bool*
where *hs-memb* == *s-memb hm-lookup*
definition *hs-ins* :: *'a*::*hashable* ⇒ *'a hs* ⇒ *'a hs*
where *hs-ins* == *s-ins hm-update*
definition *hs-ins-dj* :: *'a*::*hashable* ⇒ *'a hs* ⇒ *'a hs* **where**
hs-ins-dj == *s-ins-dj hm-update-dj*
definition *hs-delete* :: *'a*::*hashable* ⇒ *'a hs* ⇒ *'a hs*
where *hs-delete* == *s-delete hm-delete*
definition *hs-sel* :: *'a*::*hashable hs* ⇒ (*'a* ⇒ *'r option*) ⇒ *'r option*
where *hs-sel* == *s-sel hm-sel*
definition *hs-sel'* == *sel-sel' hs-sel*
definition *hs-isEmpty* :: *'a*::*hashable hs* ⇒ *bool*
where *hs-isEmpty* == *s-isEmpty hm-isEmpty*

definition *hs-iteratei* :: *'a*::*hashable hs* ⇒ (*'a*,*'σ*) *set-iterator*
where *hs-iteratei* == *s-iteratei hm-iteratei*
definition *hs-union* :: *'a*::*hashable hs* ⇒ *'a hs* ⇒ *'a hs*
where *hs-union* == *s-union hm-add*
definition *hs-union-dj* :: *'a*::*hashable hs* ⇒ *'a hs* ⇒ *'a hs*
where *hs-union-dj* == *s-union-dj hm-add-dj*
definition *hs-inter* :: *'a*::*hashable hs* ⇒ *'a hs* ⇒ *'a hs*
where *hs-inter* == *it-inter hs-iteratei hs-memb hs-empty hs-ins-dj*
definition *hs-image-filter*
where *hs-image-filter* == *it-image-filter hs-iteratei hs-empty hs-ins*
definition *hs-inj-image-filter*
where *hs-inj-image-filter* == *it-inj-image-filter hs-iteratei hs-empty hs-ins-dj*

definition *hs-image* **where** *hs-image* == *iflt-image hs-image-filter*
definition *hs-inj-image* **where** *hs-inj-image* == *iflt-inj-image hs-inj-image-filter*

definition *hs-to-list* == *it-set-to-list hs-iteratei*
definition *list-to-hs* == *gen-list-to-set hs-empty hs-ins*

4.17.2 Correctness

lemmas *hs-defs* =
list-to-hs-def
hs-α-def
hs-delete-def
hs-empty-def
hs-image-def

hs-image-filter-def
hs-inj-image-def
hs-inj-image-filter-def
hs-ins-def
hs-ins-dj-def
hs-inter-def
hs-isEmpty-def
hs-iteratei-def
hs-memb-def
hs-sel-def
hs-sel′-def
hs-to-list-def
hs-union-def
hs-union-dj-def

lemmas *hs-empty-impl* = *s-empty-correct*[*OF hm-empty-impl*, *folded hs-defs*]
lemmas *hs-memb-impl* = *s-memb-correct*[*OF hm-lookup-impl*, *folded hs-defs*]
lemmas *hs-ins-impl* = *s-ins-correct*[*OF hm-update-impl*, *folded hs-defs*]
lemmas *hs-ins-dj-impl* = *s-ins-dj-correct*[*OF hm-update-dj-impl*, *folded hs-defs*]
lemmas *hs-delete-impl* = *s-delete-correct*[*OF hm-delete-impl*, *folded hs-defs*]
lemmas *hs-iteratei-impl* = *s-iteratei-correct*[*OF hm-iteratei-impl*, *folded hs-defs*]
lemmas *hs-isEmpty-impl* = *s-isEmpty-correct*[*OF hm-isEmpty-impl*, *folded hs-defs*]
lemmas *hs-union-impl* = *s-union-correct*[*OF hm-add-impl*, *folded hs-defs*]
lemmas *hs-union-dj-impl* = *s-union-dj-correct*[*OF hm-add-dj-impl*, *folded hs-defs*]
lemmas *hs-sel-impl* = *s-sel-correct*[*OF hm-sel-impl*, *folded hs-defs*]
lemmas *hs-sel′-impl* = *sel-sel′-correct*[*OF hs-sel-impl*, *folded hs-sel′-def*]
lemmas *hs-inter-impl* = *it-inter-correct*[*OF hs-iteratei-impl hs-memb-impl hs-empty-impl hs-ins-dj-impl*, *folded hs-inter-def*]
lemmas *hs-image-filter-impl* = *it-image-filter-correct*[*OF hs-iteratei-impl hs-empty-impl hs-ins-impl*, *folded hs-image-filter-def*]
lemmas *hs-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF hs-iteratei-impl hs-empty-impl hs-ins-dj-impl*, *folded hs-inj-image-filter-def*]
lemmas *hs-image-impl* = *iflt-image-correct*[*OF hs-image-filter-impl*, *folded hs-image-def*]
lemmas *hs-inj-image-impl* = *iflt-inj-image-correct*[*OF hs-inj-image-filter-impl*, *folded hs-inj-image-def*]
lemmas *hs-to-list-impl* = *it-set-to-list-correct*[*OF hs-iteratei-impl*, *folded hs-to-list-def*]
lemmas *list-to-hs-impl* = *gen-list-to-set-correct*[*OF hs-empty-impl hs-ins-impl*, *folded list-to-hs-def*]

interpretation *hs*: *set-iteratei hs-α hs-invar hs-iteratei* ⟨*proof*⟩
interpretation *hs*: *set-inj-image-filter hs-α hs-invar hs-α hs-invar hs-inj-image-filter* ⟨*proof*⟩
interpretation *hs*: *set-image-filter hs-α hs-invar hs-α hs-invar hs-image-filter* ⟨*proof*⟩
interpretation *hs*: *set-inj-image hs-α hs-invar hs-α hs-invar hs-inj-image* ⟨*proof*⟩
interpretation *hs*: *set-union-dj hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-union-dj* ⟨*proof*⟩
interpretation *hs*: *set-union hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-union* ⟨*proof*⟩

interpretation *hs*: *set-inter hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-inter* ⟨*proof*⟩
interpretation *hs*: *set-image hs-α hs-invar hs-α hs-invar hs-image* ⟨*proof*⟩
interpretation *hs*: *set-sel hs-α hs-invar hs-sel* ⟨*proof*⟩
interpretation *hs*: *set-sel′ hs-α hs-invar hs-sel′* ⟨*proof*⟩
interpretation *hs*: *set-ins-dj hs-α hs-invar hs-ins-dj* ⟨*proof*⟩
interpretation *hs*: *set-delete hs-α hs-invar hs-delete* ⟨*proof*⟩
interpretation *hs*: *set-ins hs-α hs-invar hs-ins* ⟨*proof*⟩
interpretation *hs*: *set-memb hs-α hs-invar hs-memb* ⟨*proof*⟩
interpretation *hs*: *set-to-list hs-α hs-invar hs-to-list* ⟨*proof*⟩
interpretation *hs*: *list-to-set hs-α hs-invar list-to-hs* ⟨*proof*⟩
interpretation *hs*: *set-isEmpty hs-α hs-invar hs-isEmpty* ⟨*proof*⟩
interpretation *hs*: *set-empty hs-α hs-invar hs-empty* ⟨*proof*⟩

lemmas *hs-correct* =
hs.inj-image-filter-correct
hs.image-filter-correct
hs.inj-image-correct
hs.union-dj-correct
hs.union-correct
hs.inter-correct
hs.image-correct
hs.ins-dj-correct
hs.delete-correct
hs.ins-correct
hs.memb-correct
hs.to-list-correct
hs.to-set-correct
hs.isEmpty-correct
hs.empty-correct

4.17.3 Code Generation

export-code
hs-iteratei
hs-inj-image-filter
hs-image-filter
hs-inj-image
hs-union-dj
hs-union
hs-inter
hs-image
hs-sel
hs-sel′
hs-ins-dj
hs-delete
hs-ins
hs-memb

hs-to-list
list-to-hs
hs-isEmpty
hs-empty
in *SML*
module-name *HashSet*
file −

end

4.18 Set implementation via tries

theory *TrieSetImpl* **imports**
TrieMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

4.18.1 Definitions

type-synonym
$'a\ ts = ('a, unit)\ trie$

definition $ts\text{-}\alpha :: 'a\ ts \Rightarrow 'a\ list\ set$
where $ts\text{-}\alpha == s\text{-}\alpha\ tm\text{-}\alpha$

abbreviation (*input*) $ts\text{-}invar :: 'a\ ts \Rightarrow bool$
where $ts\text{-}invar == \lambda\text{-}.\ True$

definition $ts\text{-}empty :: unit \Rightarrow 'a\ ts$
where $ts\text{-}empty == s\text{-}empty\ tm\text{-}empty$

definition $ts\text{-}memb :: 'a\ list \Rightarrow 'a\ ts \Rightarrow bool$
where $ts\text{-}memb == s\text{-}memb\ tm\text{-}lookup$

definition $ts\text{-}ins :: 'a\ list \Rightarrow 'a\ ts \Rightarrow 'a\ ts$
where $ts\text{-}ins == s\text{-}ins\ tm\text{-}update$

definition $ts\text{-}ins\text{-}dj :: 'a\ list \Rightarrow 'a\ ts \Rightarrow 'a\ ts$ **where**
$ts\text{-}ins\text{-}dj == s\text{-}ins\text{-}dj\ tm\text{-}update\text{-}dj$

definition $ts\text{-}delete :: 'a\ list \Rightarrow 'a\ ts \Rightarrow 'a\ ts$
where $ts\text{-}delete == s\text{-}delete\ tm\text{-}delete$

definition $ts\text{-}sel :: 'a\ ts \Rightarrow ('a\ list \Rightarrow 'r\ option) \Rightarrow 'r\ option$
where $ts\text{-}sel == s\text{-}sel\ tm\text{-}sel$

definition $ts\text{-}sel' == sel\text{-}sel'\ ts\text{-}sel$

definition *ts-isEmpty* :: *'a ts* ⇒ *bool*
where *ts-isEmpty* == *s-isEmpty tm-isEmpty*

definition *ts-iteratei* :: *'a ts* ⇒ (*'a list*,*'σ*) *set-iterator*
where *ts-iteratei* == *s-iteratei tm-iteratei*

definition *ts-ball* :: *'a ts* ⇒ (*'a list* ⇒ *bool*) ⇒ *bool*
where *ts-ball* == *s-ball tm-ball*

definition *ts-union* :: *'a ts* ⇒ *'a ts* ⇒ *'a ts*
where *ts-union* == *s-union tm-add*

definition *ts-union-dj* :: *'a ts* ⇒ *'a ts* ⇒ *'a ts*
where *ts-union-dj* == *s-union-dj tm-add-dj*

definition *ts-inter* :: *'a ts* ⇒ *'a ts* ⇒ *'a ts*
where *ts-inter* == *it-inter ts-iteratei ts-memb ts-empty ts-ins-dj*

definition *ts-size* :: *'a ts* ⇒ *nat*
where *ts-size* == *it-size ts-iteratei*

definition *ts-image-filter*
where *ts-image-filter* == *it-image-filter ts-iteratei ts-empty ts-ins*

definition *ts-inj-image-filter*
where *ts-inj-image-filter* == *it-inj-image-filter ts-iteratei ts-empty ts-ins-dj*

definition *ts-image* **where** *ts-image* == *iflt-image ts-image-filter*
definition *ts-inj-image* **where** *ts-inj-image* == *iflt-inj-image ts-inj-image-filter*

definition *ts-to-list* == *it-set-to-list ts-iteratei*
definition *list-to-ts* == *gen-list-to-set ts-empty ts-ins*

4.18.2 Correctness

lemmas *ts-defs* =
list-to-ts-def
ts-α-def
ts-ball-def
ts-delete-def
ts-empty-def
ts-image-def
ts-image-filter-def
ts-inj-image-def
ts-inj-image-filter-def
ts-ins-def
ts-ins-dj-def
ts-inter-def

ts-isEmpty-def
ts-iteratei-def
ts-memb-def
ts-sel-def
ts-sel′-def
ts-size-def
ts-to-list-def
ts-union-def
ts-union-dj-def

lemmas *ts-empty-impl* = *s-empty-correct*[*OF tm-empty-impl*, *folded ts-defs*]
lemmas *ts-memb-impl* = *s-memb-correct*[*OF tm-lookup-impl*, *folded ts-defs*]
lemmas *ts-ins-impl* = *s-ins-correct*[*OF tm-update-impl*, *folded ts-defs*]
lemmas *ts-ins-dj-impl* = *s-ins-dj-correct*[*OF tm-update-dj-impl*, *folded ts-defs*]
lemmas *ts-delete-impl* = *s-delete-correct*[*OF tm-delete-impl*, *folded ts-defs*]
lemmas *ts-iteratei-impl* = *s-iteratei-correct*[*OF tm-iteratei-impl*, *folded ts-defs*]
lemmas *ts-isEmpty-impl* = *s-isEmpty-correct*[*OF tm-isEmpty-impl*, *folded ts-defs*]
lemmas *ts-union-impl* = *s-union-correct*[*OF tm-add-impl*, *folded ts-defs*]
lemmas *ts-union-dj-impl* = *s-union-dj-correct*[*OF tm-add-dj-impl*, *folded ts-defs*]
lemmas *ts-ball-impl* = *s-ball-correct*[*OF tm-ball-impl*, *folded ts-defs*]
lemmas *ts-sel-impl* = *s-sel-correct*[*OF tm-sel-impl*, *folded ts-defs*]
lemmas *ts-sel′-impl* = *sel-sel′-correct*[*OF ts-sel-impl*, *folded ts-defs*]
lemmas *ts-inter-impl* = *it-inter-correct*[*OF ts-iteratei-impl ts-memb-impl ts-empty-impl ts-ins-dj-impl*, *folded ts-inter-def*]
lemmas *ts-size-impl* = *it-size-correct*[*OF ts-iteratei-impl*, *folded ts-size-def*]
lemmas *ts-image-filter-impl* = *it-image-filter-correct*[*OF ts-iteratei-impl ts-empty-impl ts-ins-impl*, *folded ts-image-filter-def*]
lemmas *ts-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF ts-iteratei-impl ts-empty-impl ts-ins-dj-impl*, *folded ts-inj-image-filter-def*]
lemmas *ts-image-impl* = *iflt-image-correct*[*OF ts-image-filter-impl*, *folded ts-image-def*]
lemmas *ts-inj-image-impl* = *iflt-inj-image-correct*[*OF ts-inj-image-filter-impl*, *folded ts-inj-image-def*]
lemmas *ts-to-list-impl* = *it-set-to-list-correct*[*OF ts-iteratei-impl*, *folded ts-to-list-def*]
lemmas *list-to-ts-impl* = *gen-list-to-set-correct*[*OF ts-empty-impl ts-ins-impl*, *folded list-to-ts-def*]

interpretation *ts*: *set-iteratei ts-α ts-invar ts-iteratei* ⟨*proof*⟩
interpretation *ts*: *set-inj-image-filter ts-α ts-invar ts-α ts-invar ts-inj-image-filter* ⟨*proof*⟩
interpretation *ts*: *set-image-filter ts-α ts-invar ts-α ts-invar ts-image-filter* ⟨*proof*⟩
interpretation *ts*: *set-inj-image ts-α ts-invar ts-α ts-invar ts-inj-image* ⟨*proof*⟩
interpretation *ts*: *set-union-dj ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-union-dj* ⟨*proof*⟩
interpretation *ts*: *set-union ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-union* ⟨*proof*⟩
interpretation *ts*: *set-inter ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-inter* ⟨*proof*⟩
interpretation *ts*: *set-image ts-α ts-invar ts-α ts-invar ts-image* ⟨*proof*⟩
interpretation *ts*: *set-sel ts-α ts-invar ts-sel* ⟨*proof*⟩
interpretation *ts*: *set-sel′ ts-α ts-invar ts-sel′* ⟨*proof*⟩

interpretation *ts*: *set-ins-dj ts-α ts-invar ts-ins-dj* ⟨*proof*⟩
interpretation *ts*: *set-delete ts-α ts-invar ts-delete* ⟨*proof*⟩
interpretation *ts*: *set-ball ts-α ts-invar ts-ball* ⟨*proof*⟩
interpretation *ts*: *set-ins ts-α ts-invar ts-ins* ⟨*proof*⟩
interpretation *ts*: *set-memb ts-α ts-invar ts-memb* ⟨*proof*⟩
interpretation *ts*: *set-to-list ts-α ts-invar ts-to-list* ⟨*proof*⟩
interpretation *ts*: *list-to-set ts-α ts-invar list-to-ts* ⟨*proof*⟩
interpretation *ts*: *set-isEmpty ts-α ts-invar ts-isEmpty* ⟨*proof*⟩
interpretation *ts*: *set-size ts-α ts-invar ts-size* ⟨*proof*⟩
interpretation *ts*: *set-empty ts-α ts-invar ts-empty* ⟨*proof*⟩

lemmas *ts-correct* =
ts.inj-image-filter-correct
ts.image-filter-correct
ts.inj-image-correct
ts.union-dj-correct
ts.union-correct
ts.inter-correct
ts.image-correct
ts.ins-dj-correct
ts.delete-correct
ts.ball-correct
ts.ins-correct
ts.memb-correct
ts.to-list-correct
ts.to-set-correct
ts.isEmpty-correct
ts.size-correct
ts.empty-correct

4.18.3 Code Generation

export-code
ts-iteratei
ts-inj-image-filter
ts-image-filter
ts-inj-image
ts-union-dj
ts-union
ts-inter
ts-image
ts-sel
ts-sel′
ts-ins-dj
ts-delete
ts-ball
ts-ins
ts-memb
ts-to-list

list-to-ts
ts-isEmpty
ts-size
ts-empty
in *SML*
module-name *TrieSet*
file −

end

theory *ArrayHashSet*
imports
ArrayHashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

4.18.4 Definitions

type-synonym
$'a$ *ahs* = ($'a$::*hashable*,*unit*) *ahm*
definition *ahs-*α :: $'a$::*hashable ahs* ⇒ $'a$ *set* **where** *ahs-*α == *s-*α *ahm-*α
abbreviation (*input*) *ahs-invar* :: $'a$::*hashable ahs* ⇒ *bool* **where** *ahs-invar* == *ahm-invar*
definition *ahs-empty* :: *unit* ⇒ $'a$::*hashable ahs* **where** *ahs-empty* == *s-empty ahm-empty*
definition *ahs-memb* :: $'a$::*hashable* ⇒ $'a$ *ahs* ⇒ *bool*
where *ahs-memb* == *s-memb ahm-lookup*
definition *ahs-ins* :: $'a$::*hashable* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs*
where *ahs-ins* == *s-ins ahm-update*
definition *ahs-ins-dj* :: $'a$::*hashable* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs* **where**
ahs-ins-dj == *s-ins-dj ahm-update-dj*
definition *ahs-delete* :: $'a$::*hashable* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs*
where *ahs-delete* == *s-delete ahm-delete*
definition *ahs-sel* :: $'a$::*hashable ahs* ⇒ ($'a$ ⇒ $'r$ *option*) ⇒ $'r$ *option*
where *ahs-sel* == *s-sel ahm-sel*
definition *ahs-sel'* == *SetGA.sel-sel' ahs-sel*
definition *ahs-isEmpty* :: $'a$::*hashable ahs* ⇒ *bool*
where *ahs-isEmpty* == *s-isEmpty ahm-isEmpty*

definition *ahs-iteratei* :: $'a$::*hashable ahs* ⇒ ($'a$,$'\sigma$) *set-iterator*
where *ahs-iteratei* == *s-iteratei ahm-iteratei*

definition *ahs-union* :: $'a$::*hashable ahs* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs*
where *ahs-union* == *s-union ahm-add*
definition *ahs-union-dj* :: $'a$::*hashable ahs* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs*
where *ahs-union-dj* == *s-union-dj ahm-add-dj*
definition *ahs-inter* :: $'a$::*hashable ahs* ⇒ $'a$ *ahs* ⇒ $'a$ *ahs*

where *ahs-inter == it-inter ahs-iteratei ahs-memb ahs-empty ahs-ins-dj*

definition *ahs-image-filter*
where *ahs-image-filter == it-image-filter ahs-iteratei ahs-empty ahs-ins*
definition *ahs-inj-image-filter*
where *ahs-inj-image-filter == it-inj-image-filter ahs-iteratei ahs-empty ahs-ins-dj*

definition *ahs-image* **where** *ahs-image == iflt-image ahs-image-filter*
definition *ahs-inj-image* **where** *ahs-inj-image == iflt-inj-image ahs-inj-image-filter*

definition *ahs-to-list == it-set-to-list ahs-iteratei*
definition *list-to-ahs == gen-list-to-set ahs-empty ahs-ins*

4.18.5 Correctness

lemmas *ahs-defs =*
list-to-ahs-def
ahs-α-def
ahs-delete-def
ahs-empty-def
ahs-image-def
ahs-image-filter-def
ahs-inj-image-def
ahs-inj-image-filter-def
ahs-ins-def
ahs-ins-dj-def
ahs-inter-def
ahs-isEmpty-def
ahs-iteratei-def
ahs-memb-def
ahs-sel-def
ahs-sel′-def
ahs-to-list-def
ahs-union-def
ahs-union-dj-def

lemmas *ahs-empty-impl = s-empty-correct[OF ahm-empty-impl, folded ahs-defs]*
lemmas *ahs-memb-impl = s-memb-correct[OF ahm-lookup-impl, folded ahs-defs]*
lemmas *ahs-ins-impl = s-ins-correct[OF ahm-update-impl, folded ahs-defs]*
lemmas *ahs-ins-dj-impl = s-ins-dj-correct[OF ahm-update-dj-impl, folded ahs-defs]*
lemmas *ahs-delete-impl = s-delete-correct[OF ahm-delete-impl, folded ahs-defs]*
lemmas *ahs-iteratei-impl = s-iteratei-correct[OF ahm-iteratei-impl, folded ahs-defs]*
lemmas *ahs-isEmpty-impl = s-isEmpty-correct[OF ahm-isEmpty-impl, folded ahs-defs]*
lemmas *ahs-union-impl = s-union-correct[OF ahm-add-impl, folded ahs-defs]*
lemmas *ahs-union-dj-impl = s-union-dj-correct[OF ahm-add-dj-impl, folded ahs-defs]*
lemmas *ahs-sel-impl = s-sel-correct[OF ahm-sel-impl, folded ahs-defs]*
lemmas *ahs-sel′-impl = sel-sel′-correct[OF ahs-sel-impl, folded ahs-sel′-def]*
lemmas *ahs-inter-impl = it-inter-correct[OF ahs-iteratei-impl ahs-memb-impl ahs-empty-impl*

ahs-ins-dj-impl, *folded ahs-inter-def*]
lemmas *ahs-image-filter-impl* = *it-image-filter-correct*[*OF ahs-iteratei-impl ahs-empty-impl ahs-ins-impl*, *folded ahs-image-filter-def*]
lemmas *ahs-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF ahs-iteratei-impl ahs-empty-impl ahs-ins-dj-impl*, *folded ahs-inj-image-filter-def*]
lemmas *ahs-image-impl* = *iflt-image-correct*[*OF ahs-image-filter-impl*, *folded ahs-image-def*]
lemmas *ahs-inj-image-impl* = *iflt-inj-image-correct*[*OF ahs-inj-image-filter-impl*, *folded ahs-inj-image-def*]
lemmas *ahs-to-list-impl* = *it-set-to-list-correct*[*OF ahs-iteratei-impl*, *folded ahs-to-list-def*]
lemmas *list-to-ahs-impl* = *gen-list-to-set-correct*[*OF ahs-empty-impl ahs-ins-impl*, *folded list-to-ahs-def*]

interpretation *ahs*: *set-iteratei ahs-α ahs-invar ahs-iteratei* ⟨*proof*⟩

interpretation *ahs*: *set-inj-image-filter ahs-α ahs-invar ahs-α ahs-invar ahs-inj-image-filter* ⟨*proof*⟩
interpretation *ahs*: *set-image-filter ahs-α ahs-invar ahs-α ahs-invar ahs-image-filter* ⟨*proof*⟩
interpretation *ahs*: *set-inj-image ahs-α ahs-invar ahs-α ahs-invar ahs-inj-image* ⟨*proof*⟩
interpretation *ahs*: *set-union-dj ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-union-dj* ⟨*proof*⟩
interpretation *ahs*: *set-union ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-union* ⟨*proof*⟩
interpretation *ahs*: *set-inter ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-inter* ⟨*proof*⟩
interpretation *ahs*: *set-image ahs-α ahs-invar ahs-α ahs-invar ahs-image* ⟨*proof*⟩
interpretation *ahs*: *set-sel ahs-α ahs-invar ahs-sel* ⟨*proof*⟩
interpretation *ahs*: *set-sel′ ahs-α ahs-invar ahs-sel′* ⟨*proof*⟩
interpretation *ahs*: *set-ins-dj ahs-α ahs-invar ahs-ins-dj* ⟨*proof*⟩
interpretation *ahs*: *set-delete ahs-α ahs-invar ahs-delete* ⟨*proof*⟩
interpretation *ahs*: *set-ins ahs-α ahs-invar ahs-ins* ⟨*proof*⟩
interpretation *ahs*: *set-memb ahs-α ahs-invar ahs-memb* ⟨*proof*⟩
interpretation *ahs*: *set-to-list ahs-α ahs-invar ahs-to-list* ⟨*proof*⟩
interpretation *ahs*: *list-to-set ahs-α ahs-invar list-to-ahs* ⟨*proof*⟩
interpretation *ahs*: *set-isEmpty ahs-α ahs-invar ahs-isEmpty* ⟨*proof*⟩
interpretation *ahs*: *set-empty ahs-α ahs-invar ahs-empty* ⟨*proof*⟩

lemmas *ahs-correct* =
ahs.inj-image-filter-correct
ahs.image-filter-correct
ahs.inj-image-correct
ahs.union-dj-correct
ahs.union-correct
ahs.inter-correct
ahs.image-correct
ahs.ins-dj-correct

ahs.delete-correct
ahs.ins-correct
ahs.memb-correct
ahs.to-list-correct
ahs.to-set-correct
ahs.isEmpty-correct
ahs.empty-correct

4.18.6 Code Generation

export-code
ahs-iteratei
ahs-inj-image-filter
ahs-image-filter
ahs-inj-image
ahs-union-dj
ahs-union
ahs-inter
ahs-image
ahs-sel
ahs-sel′
ahs-ins-dj
ahs-delete
ahs-ins
ahs-memb
ahs-to-list
list-to-ahs
ahs-isEmpty
ahs-empty
in *SML*
module-name *ArrayHashSet*
file −

end

4.19 Set Implementation by Arrays

theory *ArraySetImpl*
imports
../spec/SetSpec
ArrayMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

4.19.1 Definitions

type-synonym *ias* = (*unit*) *iam*

definition *ias-α* :: *ias* ⇒ *nat set* **where** *ias-α* == *s-α iam-α*
abbreviation (*input*) *ias-invar* :: *ias* ⇒ *bool* **where** *ias-invar* == *iam-invar*
definition *ias-empty* :: *unit* ⇒ *ias* **where** *ias-empty* == *s-empty iam-empty*
definition *ias-memb* :: *nat* ⇒ *ias* ⇒ *bool*
where *ias-memb* == *s-memb iam-lookup*
definition *ias-ins* :: *nat* ⇒ *ias* ⇒ *ias*
where *ias-ins* == *s-ins iam-update*
definition *ias-ins-dj* :: *nat* ⇒ *ias* ⇒ *ias* **where**
ias-ins-dj == *s-ins-dj iam-update-dj*
definition *ias-delete* :: *nat* ⇒ *ias* ⇒ *ias*
where *ias-delete* == *s-delete iam-delete*
definition *ias-sel* :: *ias* ⇒ (*nat* ⇒ *'r option*) ⇒ *'r option*
where *ias-sel* == *s-sel iam-sel*
definition *ias-sel'* = *sel-sel' ias-sel*
definition *ias-isEmpty* :: *ias* ⇒ *bool*
where *ias-isEmpty* == *s-isEmpty iam-isEmpty*

definition *ias-iteratei* :: *ias* ⇒ (*nat*,*'σ*) *set-iterator*
where *ias-iteratei* == *s-iteratei iam-iteratei*

definition *ias-union* :: *ias* ⇒ *ias* ⇒ *ias*
where *ias-union* == *s-union iam-add*
definition *ias-union-dj* :: *ias* ⇒ *ias* ⇒ *ias*
where *ias-union-dj* == *s-union-dj iam-add-dj*
definition *ias-inter* :: *ias* ⇒ *ias* ⇒ *ias*
where *ias-inter* == *it-inter ias-iteratei ias-memb ias-empty ias-ins-dj*

definition *ias-image-filter*
where *ias-image-filter* == *it-image-filter ias-iteratei ias-empty ias-ins*
definition *ias-inj-image-filter*
where *ias-inj-image-filter* == *it-inj-image-filter ias-iteratei ias-empty ias-ins-dj*

definition *ias-image* **where** *ias-image* == *iflt-image ias-image-filter*
definition *ias-inj-image* **where** *ias-inj-image* == *iflt-inj-image ias-inj-image-filter*

definition *ias-to-list* == *it-set-to-list ias-iteratei*
definition *list-to-ias* == *gen-list-to-set ias-empty ias-ins*

4.19.2 Correctness

lemmas *ias-defs* =
list-to-ias-def
ias-α-def
ias-delete-def
ias-empty-def
ias-image-def
ias-image-filter-def
ias-inj-image-def

ias-inj-image-filter-def
ias-ins-def
ias-ins-dj-def
ias-inter-def
ias-isEmpty-def
ias-iteratei-def
ias-memb-def
ias-sel-def
ias-sel′-def
ias-to-list-def
ias-union-def
ias-union-dj-def

lemmas *ias-empty-impl* = *s-empty-correct*[*OF iam-empty-impl*, *folded ias-defs*]
lemmas *ias-memb-impl* = *s-memb-correct*[*OF iam-lookup-impl*, *folded ias-defs*]
lemmas *ias-ins-impl* = *s-ins-correct*[*OF iam-update-impl*, *folded ias-defs*]
lemmas *ias-ins-dj-impl* = *s-ins-dj-correct*[*OF iam-update-dj-impl*, *folded ias-defs*]
lemmas *ias-delete-impl* = *s-delete-correct*[*OF iam-delete-impl*, *folded ias-defs*]
lemmas *ias-iteratei-impl* = *s-iteratei-correct*[*OF iam-iteratei-impl*, *folded ias-defs*]
lemmas *ias-isEmpty-impl* = *s-isEmpty-correct*[*OF iam-isEmpty-impl*, *folded ias-defs*]
lemmas *ias-union-impl* = *s-union-correct*[*OF iam-add-impl*, *folded ias-defs*]
lemmas *ias-union-dj-impl* = *s-union-dj-correct*[*OF iam-add-dj-impl*, *folded ias-defs*]
lemmas *ias-sel-impl* = *s-sel-correct*[*OF iam-sel-impl*, *folded ias-defs*]
lemmas *ias-sel′-impl* = *sel-sel′-correct*[*OF ias-sel-impl*, *folded ias-sel′-def*]
lemmas *ias-inter-impl* = *it-inter-correct*[*OF ias-iteratei-impl ias-memb-impl ias-empty-impl ias-ins-dj-impl*, *folded ias-inter-def*]
lemmas *ias-image-filter-impl* = *it-image-filter-correct*[*OF ias-iteratei-impl ias-empty-impl ias-ins-impl*, *folded ias-image-filter-def*]
lemmas *ias-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF ias-iteratei-impl ias-empty-impl ias-ins-dj-impl*, *folded ias-inj-image-filter-def*]
lemmas *ias-image-impl* = *iflt-image-correct*[*OF ias-image-filter-impl*, *folded ias-image-def*]
lemmas *ias-inj-image-impl* = *iflt-inj-image-correct*[*OF ias-inj-image-filter-impl*, *folded ias-inj-image-def*]
lemmas *ias-to-list-impl* = *it-set-to-list-correct*[*OF ias-iteratei-impl*, *folded ias-to-list-def*]
lemmas *list-to-ias-impl* = *gen-list-to-set-correct*[*OF ias-empty-impl ias-ins-impl*, *folded list-to-ias-def*]

interpretation *ias*: *set-iteratei ias-α ias-invar ias-iteratei* ⟨*proof*⟩

interpretation *ias*: *set-inj-image-filter ias-α ias-invar ias-α ias-invar ias-inj-image-filter* ⟨*proof*⟩
interpretation *ias*: *set-image-filter ias-α ias-invar ias-α ias-invar ias-image-filter* ⟨*proof*⟩
interpretation *ias*: *set-inj-image ias-α ias-invar ias-α ias-invar ias-inj-image* ⟨*proof*⟩
interpretation *ias*: *set-union-dj ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-union-dj* ⟨*proof*⟩
interpretation *ias*: *set-union ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-union* ⟨*proof*⟩

interpretation *ias*: *set-inter ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-inter* ⟨*proof*⟩
interpretation *ias*: *set-image ias-α ias-invar ias-α ias-invar ias-image* ⟨*proof*⟩
interpretation *ias*: *set-sel ias-α ias-invar ias-sel* ⟨*proof*⟩
interpretation *ias*: *set-sel′ ias-α ias-invar ias-sel′* ⟨*proof*⟩
interpretation *ias*: *set-ins-dj ias-α ias-invar ias-ins-dj* ⟨*proof*⟩
interpretation *ias*: *set-delete ias-α ias-invar ias-delete* ⟨*proof*⟩
interpretation *ias*: *set-ins ias-α ias-invar ias-ins* ⟨*proof*⟩
interpretation *ias*: *set-memb ias-α ias-invar ias-memb* ⟨*proof*⟩
interpretation *ias*: *set-to-list ias-α ias-invar ias-to-list* ⟨*proof*⟩
interpretation *ias*: *list-to-set ias-α ias-invar list-to-ias* ⟨*proof*⟩
interpretation *ias*: *set-isEmpty ias-α ias-invar ias-isEmpty* ⟨*proof*⟩
interpretation *ias*: *set-empty ias-α ias-invar ias-empty* ⟨*proof*⟩

lemmas *ias-correct* =
ias.inj-image-filter-correct
ias.image-filter-correct
ias.inj-image-correct
ias.union-dj-correct
ias.union-correct
ias.inter-correct
ias.image-correct
ias.ins-dj-correct
ias.delete-correct
ias.ins-correct
ias.memb-correct
ias.to-list-correct
ias.to-set-correct
ias.isEmpty-correct
ias.empty-correct

4.19.3 Code Generation

export-code
ias-iteratei
ias-inj-image-filter
ias-image-filter
ias-inj-image
ias-union-dj
ias-union
ias-inter
ias-image
ias-sel
ias-sel′
ias-ins-dj
ias-delete
ias-ins
ias-memb

ias-to-list
list-to-ias
ias-isEmpty
ias-empty
in *SML*
module-name *ArraySet*
file –

end

Standard Set Implementations **theory** *SetStdImpl*
imports
ListSetImpl
ListSetImpl-Invar
RBTSetImpl HashSet
TrieSetImpl
ArrayHashSet
ArraySetImpl
begin

This theory summarizes standard set implementations, namely list-sets RB-tree-sets, trie-sets and hashsets.

end

4.20 Fifo Queue by Pair of Lists

theory *Fifo*
imports *../gen-algo/ListGA*
begin

A fifo-queue is implemented by a pair of two lists (stacks). New elements are pushed on the first stack, and elements are popped from the second stack. If the second stack is empty, the first stack is reversed and replaces the second stack.

If list reversal is implemented efficiently (what is the case in Isabelle/HOL, cf *List.rev-conv-fold*) the amortized time per buffer operation is constant.

Moreover, this fifo implementation also supports efficient push and pop operations.

4.20.1 Definitions

type-synonym *'a fifo* = *'a list* × *'a list*

— The empty fifo
definition *fifo-empty* :: *unit* ⇒ *'a fifo*
where *fifo-empty* == λ-::*unit*. ([],[])

— True, iff the fifo is empty
definition *fifo-isEmpty* :: *'a fifo* ⇒ *bool* **where** *fifo-isEmpty F* == *F*=([],[])

— Add an element to the fifo
definition *fifo-enqueue* :: *'a* ⇒ *'a fifo* ⇒ *'a fifo*
where *fifo-enqueue a F* == (*a*#*fst F*, *snd F*)

— Get an element from the fifo
definition *fifo-dequeue* :: *'a fifo* ⇒ (*'a* × *'a fifo*) **where**
fifo-dequeue F ==
case snd F of
(*a*#*l*) ⇒ (*a*, (*fst F*, *l*)) |
[] ⇒ *let rp*=*rev* (*fst F*) *in*
(*hd rp*, ([], *tl rp*))

— Stack view on fifo: Pop
abbreviation *fifo-pop* == *fifo-dequeue*
— Stack view on fifo: Push
definition *fifo-push* :: *'a* ⇒ *'a fifo* ⇒ *'a fifo*
where *fifo-push x F* == *case F of* (*e*,*d*) ⇒ (*e*,*x*#*d*)

— Abstraction of the fifo to a list. The next element to be got is at the head of the list, and new elements are appended at the end of the list
definition *fifo-α* :: *'a fifo* ⇒ *'a list* **where** *fifo-α F* == *snd F* @ *rev* (*fst F*)

— This fifo implementation has no invariants, any pair of lists is a valid fifo
definition [*simp*, *intro*!]: *fifo-invar x* = *True*

4.20.2 Correctness

lemma *fifo-empty-impl*: *list-empty fifo-α fifo-invar fifo-empty*
⟨*proof*⟩

lemma *fifo-isEmpty-impl*: *list-isEmpty fifo-α fifo-invar fifo-isEmpty*
⟨*proof*⟩

lemma *fifo-enqueue-impl*: *list-enqueue fifo-α fifo-invar fifo-enqueue*
⟨*proof*⟩

lemma *fifo-dequeue-impl*: *list-dequeue fifo-α fifo-invar fifo-dequeue*
⟨*proof*⟩

interpretation *fifo*: *list-empty fifo-α fifo-invar fifo-empty* ⟨*proof*⟩
interpretation *fifo*: *list-isEmpty fifo-α fifo-invar fifo-isEmpty* ⟨*proof*⟩
interpretation *fifo*: *list-enqueue fifo-α fifo-invar fifo-enqueue* ⟨*proof*⟩
interpretation *fifo*: *list-dequeue fifo-α fifo-invar fifo-dequeue* ⟨*proof*⟩

lemma *fifo-pop-impl*: *list-pop fifo-α fifo-invar fifo-pop*
⟨*proof*⟩

lemma *fifo-push-impl*: *list-push fifo-α fifo-invar fifo-push*
⟨*proof*⟩

interpretation *fifo*: *list-pop fifo-α fifo-invar fifo-pop* ⟨*proof*⟩
interpretation *fifo*: *list-push fifo-α fifo-invar fifo-push* ⟨*proof*⟩

lemmas *fifo-correct* =
fifo.empty-correct(*1*)
fifo.isEmpty-correct[*simplified*]
fifo.enqueue-correct(*1*)[*simplified*]
fifo.dequeue-correct(*1*,*2*)[*simplified*]
fifo.push-correct(*1*)[*simplified*]
fifo.pop-correct(*1*,*2*)[*simplified*]

4.20.3 Code Generation

export-code
fifo-empty fifo-isEmpty fifo-enqueue fifo-dequeue fifo-push fifo-pop
in *SML*
module-name *Fifo*
file −

end

4.21 Implementation of Priority Queues by Binomial Heap

theory *BinoPrioImpl*
imports
../../Binomial−Heaps/BinomialHeap
../spec/PrioSpec
begin

type-synonym ('*a*,'*b*) *bino* = ('*a*,'*b*) *BinomialHeap*

4.21.1 Definitions

definition *bino-α* **where** *bino-α q* ≡ *BinomialHeap.to-mset q*
definition *bino-insert* **where** *bino-insert* ≡ *BinomialHeap.insert*
abbreviation (*input*) *bino-invar* **where** *bino-invar* ≡ λ-. *True*
definition *bino-find* **where** *bino-find* ≡ *BinomialHeap.findMin*
definition *bino-delete* **where** *bino-delete* ≡ *BinomialHeap.deleteMin*
definition *bino-meld* **where** *bino-meld* ≡ *BinomialHeap.meld*

definition *bino-empty* **where** *bino-empty* ≡ λ-::*unit*. *BinomialHeap.empty*
definition *bino-isEmpty* **where** *bino-isEmpty* = *BinomialHeap.isEmpty*

definition *bino-ops* == (|
  prio-op-α = *bino-α*,
  prio-op-invar = *bino-invar*,
  prio-op-empty = *bino-empty*,
  prio-op-isEmpty = *bino-isEmpty*,
  prio-op-insert = *bino-insert*,
  prio-op-find = *bino-find*,
  prio-op-delete = *bino-delete*,
  prio-op-meld = *bino-meld*
|)

lemmas *bino-defs* =
  bino-α-def
  bino-insert-def
  bino-find-def
  bino-delete-def
  bino-meld-def
  bino-empty-def
  bino-isEmpty-def

4.21.2 Correctness

theorem *bino-empty-impl*: *prio-empty bino-α bino-invar bino-empty*
  ⟨*proof*⟩

theorem *bino-isEmpty-impl*: *prio-isEmpty bino-α bino-invar bino-isEmpty*
  ⟨*proof*⟩

theorem *bino-find-impl*: *prio-find bino-α bino-invar bino-find*
  ⟨*proof*⟩

lemma *bino-insert-impl*: *prio-insert bino-α bino-invar bino-insert*
  ⟨*proof*⟩

lemma *bino-meld-impl*: *prio-meld bino-α bino-invar bino-meld*
  ⟨*proof*⟩

lemma *bino-delete-impl*:
  prio-delete bino-α bino-invar bino-find bino-delete
  ⟨*proof*⟩

interpretation *bino*: *prio-empty bino-α bino-invar bino-empty*
⟨*proof*⟩
interpretation *bino*: *prio-isEmpty bino-α bino-invar bino-isEmpty*
⟨*proof*⟩
interpretation *bino*: *prio-insert bino-α bino-invar bino-insert*

⟨*proof*⟩
interpretation *bino*: *prio-delete bino-α bino-invar bino-find bino-delete*
⟨*proof*⟩
interpretation *bino*: *prio-meld bino-α bino-invar bino-meld*
⟨*proof*⟩

interpretation *binos*: *StdPrio bino-ops*
⟨*proof*⟩

lemmas *bino-correct* =
bino.empty-correct
bino.isEmpty-correct
bino.insert-correct
bino.meld-correct
bino.find-correct
bino.delete-correct

4.21.3 Code Generation

Unfold record definition for code generation

lemma [*code-unfold*]:
prio-op-empty bino-ops = *bino-empty*
prio-op-isEmpty bino-ops = *bino-isEmpty*
prio-op-insert bino-ops = *bino-insert*
prio-op-find bino-ops = *bino-find*
prio-op-delete bino-ops = *bino-delete*
prio-op-meld bino-ops = *bino-meld*
⟨*proof*⟩

export-code
bino-empty
bino-isEmpty
bino-insert
bino-find
bino-delete
bino-meld
in *SML*
module-name *Bino*
file −

end

4.22 Implementation of Priority Queues by Skew Binomial Heaps

theory *SkewPrioImpl*

imports
../../Binomial−Heaps/SkewBinomialHeap
../spec/PrioSpec
begin

4.22.1 Definitions

type-synonym $('a,'b)$ *skew* $= ('a, 'b)$ *SkewBinomialHeap*

definition *skew-α* **where** *skew-α q* ≡ *SkewBinomialHeap.to-mset q*
definition *skew-insert* **where** *skew-insert* ≡ *SkewBinomialHeap.insert*
abbreviation (*input*) *skew-invar* **where** *skew-invar* ≡ λ-. *True*
definition *skew-find* **where** *skew-find* ≡ *SkewBinomialHeap.findMin*
definition *skew-delete* **where** *skew-delete* ≡ *SkewBinomialHeap.deleteMin*
definition *skew-meld* **where** *skew-meld* ≡ *SkewBinomialHeap.meld*
definition *skew-empty* **where** *skew-empty* ≡ λ-::*unit*. *SkewBinomialHeap.empty*
definition *skew-isEmpty* **where** *skew-isEmpty* = *SkewBinomialHeap.isEmpty*

definition *skew-ops* == (|
prio-op-α = *skew-α*,
prio-op-invar = *skew-invar*,
prio-op-empty = *skew-empty*,
prio-op-isEmpty = *skew-isEmpty*,
prio-op-insert = *skew-insert*,
prio-op-find = *skew-find*,
prio-op-delete = *skew-delete*,
prio-op-meld = *skew-meld*
|)

lemmas *skew-defs* =
skew-α-def
skew-insert-def
skew-find-def
skew-delete-def
skew-meld-def
skew-empty-def
skew-isEmpty-def

4.22.2 Correctness

theorem *skew-empty-impl*: *prio-empty skew-α skew-invar skew-empty*
⟨*proof*⟩

theorem *skew-isEmpty-impl*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
⟨*proof*⟩

theorem *skew-find-impl*: *prio-find skew-α skew-invar skew-find*
⟨*proof*⟩

lemma *skew-insert-impl*: *prio-insert skew-α skew-invar skew-insert*

⟨*proof*⟩

lemma *skew-meld-impl*: *prio-meld skew-α skew-invar skew-meld*
⟨*proof*⟩

lemma *skew-delete-impl*:
prio-delete skew-α skew-invar skew-find skew-delete
⟨*proof*⟩

interpretation *skew*: *prio-empty skew-α skew-invar skew-empty*
⟨*proof*⟩
interpretation *skew*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
⟨*proof*⟩
interpretation *skew*: *prio-insert skew-α skew-invar skew-insert*
⟨*proof*⟩
interpretation *skew*: *prio-delete skew-α skew-invar skew-find skew-delete*
⟨*proof*⟩
interpretation *skew*: *prio-meld skew-α skew-invar skew-meld*
⟨*proof*⟩

interpretation *skews*: *StdPrio skew-ops*
⟨*proof*⟩

lemmas *skew-correct* =
skew.empty-correct
skew.isEmpty-correct
skew.insert-correct
skew.meld-correct
skew.find-correct
skew.delete-correct

4.22.3 Code Generation

Unfold record definition for code generation

lemma [*code-unfold*]:
prio-op-empty skew-ops = *skew-empty*
prio-op-isEmpty skew-ops = *skew-isEmpty*
prio-op-insert skew-ops = *skew-insert*
prio-op-find skew-ops = *skew-find*
prio-op-delete skew-ops = *skew-delete*
prio-op-meld skew-ops = *skew-meld*
⟨*proof*⟩

export-code
skew-empty
skew-isEmpty
skew-insert
skew-find

skew-delete
skew-meld
in *SML*
module-name *Skew*
file −

end

4.23 Implementation of Annotated Lists by 2-3 Finger Trees

theory *FTAnnotatedListImpl*
imports
../../Finger−Trees/FingerTree
../spec/AnnotatedListSpec
begin

type-synonym $('a,'b)$ *ft* = $('a,'b)$ *FingerTree*

4.23.1 Definitions

definition *ft-α* **where**
ft-α ≡ *FingerTree.toList*
abbreviation (*input*) *ft-invar* **where**
ft-invar ≡ λ-. *True*
definition *ft-empty* **where**
ft-empty ≡ λ-::*unit*. *FingerTree.empty*
definition *ft-isEmpty* **where**
ft-isEmpty ≡ *FingerTree.isEmpty*
definition *ft-count* **where**
ft-count ≡ *FingerTree.count*
definition *ft-consl* **where**
ft-consl e a s = *FingerTree.lcons* (*e*,*a*) *s*
definition *ft-consr* **where**
ft-consr s e a = *FingerTree.rcons s* (*e*,*a*)
definition *ft-head* **where**
ft-head ≡ *FingerTree.head*
definition *ft-tail* **where**
ft-tail ≡ *FingerTree.tail*
definition *ft-headR* **where**
ft-headR ≡ *FingerTree.headR*
definition *ft-tailR* **where**
ft-tailR ≡ *FingerTree.tailR*
definition *ft-foldl* **where**
ft-foldl ≡ *FingerTree.foldl*

definition *ft-foldr* **where**
ft-foldr ≡ *FingerTree.foldr*
definition *ft-app* **where**
ft-app ≡ *FingerTree.app*
definition *ft-annot* **where**
ft-annot ≡ *FingerTree.annot*
definition *ft-splits* **where**
ft-splits ≡ *FingerTree.splitTree*

lemmas *ft-defs* =
ft-α-def
ft-empty-def
ft-isEmpty-def
ft-count-def
ft-consl-def
ft-consr-def
ft-head-def
ft-tail-def
ft-headR-def
ft-tailR-def
ft-foldl-def
ft-foldr-def
ft-app-def
ft-annot-def
ft-splits-def

lemma *ft-empty-impl*: *al-empty ft-α ft-invar ft-empty*
⟨*proof*⟩

lemma *ft-consl-impl*: *al-consl ft-α ft-invar ft-consl*
⟨*proof*⟩

lemma *ft-consr-impl*: *al-consr ft-α ft-invar ft-consr*
⟨*proof*⟩

lemma *ft-isEmpty-impl*: *al-isEmpty ft-α ft-invar ft-isEmpty*
⟨*proof*⟩

lemma *ft-count-impl*: *al-count ft-α ft-invar ft-count*
⟨*proof*⟩

lemma *ft-head-impl*: *al-head ft-α ft-invar ft-head*
⟨*proof*⟩

lemma *ft-tail-impl*: *al-tail ft-α ft-invar ft-tail*
⟨*proof*⟩

lemma *ft-headR-impl*: *al-headR ft-α ft-invar ft-headR*
⟨*proof*⟩

lemma *ft-tailR-impl*: *al-tailR ft-α ft-invar ft-tailR*
⟨*proof*⟩

lemma *ft-foldl-impl*: *al-foldl ft-α ft-invar ft-foldl*
⟨*proof*⟩

lemma *ft-foldr-impl*: *al-foldr ft-α ft-invar ft-foldr*
⟨*proof*⟩

lemma *ft-app-impl*: *al-app ft-α ft-invar ft-app*
⟨*proof*⟩

lemma *ft-annot-impl*: *al-annot ft-α ft-invar ft-annot*
⟨*proof*⟩

lemma *ft-splits-impl*: *al-splits ft-α ft-invar ft-splits*
⟨*proof*⟩

4.23.2 Interpretations

interpretation *ft*: *al-empty ft-α ft-invar ft-empty* ⟨*proof*⟩
interpretation *ft*: *al-isEmpty ft-α ft-invar ft-isEmpty* ⟨*proof*⟩
interpretation *ft*: *al-count ft-α ft-invar ft-count* ⟨*proof*⟩
interpretation *ft*: *al-consl ft-α ft-invar ft-consl* ⟨*proof*⟩
interpretation *ft*: *al-consr ft-α ft-invar ft-consr* ⟨*proof*⟩
interpretation *ft*: *al-head ft-α ft-invar ft-head* ⟨*proof*⟩
interpretation *ft*: *al-tail ft-α ft-invar ft-tail* ⟨*proof*⟩
interpretation *ft*: *al-headR ft-α ft-invar ft-headR* ⟨*proof*⟩
interpretation *ft*: *al-tailR ft-α ft-invar ft-tailR* ⟨*proof*⟩
interpretation *ft*: *al-foldl ft-α ft-invar ft-foldl* ⟨*proof*⟩
interpretation *ft*: *al-foldr ft-α ft-invar ft-foldr* ⟨*proof*⟩
interpretation *ft*: *al-app ft-α ft-invar ft-app* ⟨*proof*⟩
interpretation *ft*: *al-annot ft-α ft-invar ft-annot* ⟨*proof*⟩
interpretation *ft*: *al-splits ft-α ft-invar ft-splits* ⟨*proof*⟩

lemmas *ft-correct* =
ft.empty-correct
ft.isEmpty-correct
ft.count-correct
ft.consl-correct
ft.consr-correct
ft.head-correct
ft.tail-correct
ft.headR-correct
ft.tailR-correct
ft.foldl-correct
ft.foldr-correct
ft.app-correct

ft.annot-correct
ft.splits-correct

Record Based Implementation

definition *ft-ops* = (|
alist-op-α = *ft-α*,
alist-op-invar = *ft-invar*,
alist-op-empty = *ft-empty*,
alist-op-isEmpty = *ft-isEmpty*,
alist-op-count = *ft-count*,
alist-op-consl = *ft-consl*,
alist-op-consr = *ft-consr*,
alist-op-head = *ft-head*,
alist-op-tail = *ft-tail*,
alist-op-headR = *ft-headR*,
alist-op-tailR = *ft-tailR*,
alist-op-app = *ft-app*,
alist-op-annot = *ft-annot*,
alist-op-splits = *ft-splits*
|)

interpretation *ftr*!: *StdAL ft-ops*
⟨*proof*⟩

lemma *ft-ops-unfold*[*code-unfold*]:
alist-op-α ft-ops = *ft-α*
alist-op-invar ft-ops = *ft-invar*
alist-op-empty ft-ops = *ft-empty*
alist-op-isEmpty ft-ops = *ft-isEmpty*
alist-op-count ft-ops = *ft-count*
alist-op-consl ft-ops = *ft-consl*
alist-op-consr ft-ops = *ft-consr*
alist-op-head ft-ops = *ft-head*
alist-op-tail ft-ops = *ft-tail*
alist-op-headR ft-ops = *ft-headR*
alist-op-tailR ft-ops = *ft-tailR*
alist-op-app ft-ops = *ft-app*
alist-op-annot ft-ops = *ft-annot*
alist-op-splits ft-ops = *ft-splits*
⟨*proof*⟩

interpretation *ftr*!: *al-foldl* (*alist-op-α ft-ops*) (*alist-op-invar ft-ops*) *ft-foldl*
⟨*proof*⟩
interpretation *ftr*!: *al-foldr* (*alist-op-α ft-ops*) (*alist-op-invar ft-ops*) *ft-foldr*
⟨*proof*⟩

lemmas *ft-impl* =
ft-empty-impl

ft-isEmpty-impl
ft-count-impl
ft-consl-impl
ft-consr-impl
ft-head-impl
ft-tail-impl
ft-headR-impl
ft-tailR-impl
ft-foldl-impl
ft-foldr-impl
ft-app-impl
ft-annot-impl
ft-splits-impl

4.23.3 Code Generation

export-code
ft-empty
ft-isEmpty
ft-count
ft-consl
ft-head
ft-tail
ft-headR
ft-tailR
ft-foldl
ft-foldr
ft-app
ft-splits
in *SML*
module-name *FT*

end

4.24 Implementation of Priority Queues by Finger Trees

theory *FTPrioImpl*
imports *FTAnnotatedListImpl*
../gen-algo/PrioByAnnotatedList
begin

type-synonym $('a,'p)$ *alprioi* = $(unit, ('a, 'p)\ Prio)\ FingerTree$

4.24.1 Definitions

definition *alprioi-α* :: $('a,'p::linorder)$ *alprioi* ⇒ $('a\times'p)$ *multiset* **where**

alprioi-α = *alprio-α ft-α*
definition *alprioi-invar* **where**
alprioi-invar = *alprio-invar ft-α ft-invar*
definition *alprioi-empty*
:: *unit* ⇒ (*unit*,('*a*,'*b*::*linorder*) *Prio*) *FingerTree* **where**
alprioi-empty = *alprio-empty ft-empty*
definition *alprioi-isEmpty* **where**
alprioi-isEmpty ≡ *alprio-isEmpty ft-isEmpty*
definition *alprioi-insert* :: - ⇒ - ⇒ - ⇒ (*unit*,('*a*,'*b*::*linorder*) *Prio*) *FingerTree*
where
alprioi-insert = *alprio-insert ft-consl*
definition *alprioi-find* **where**
alprioi-find = *alprio-find ft-annot*
definition *alprioi-delete* **where**
alprioi-delete = *alprio-delete ft-splits ft-annot ft-app*
definition *alprioi-meld* **where**
alprioi-meld = *alprio-meld ft-app*

definition *alprioi-ops* == (|
prio-op-α = *alprioi-α*,
prio-op-invar = *alprioi-invar*,
prio-op-empty = *alprioi-empty*,
prio-op-isEmpty = *alprioi-isEmpty*,
prio-op-insert = *alprioi-insert*,
prio-op-find = *alprioi-find*,
prio-op-delete = *alprioi-delete*,
prio-op-meld = *alprioi-meld*
|)

4.24.2 Correctness

lemmas *alprioi-defs* =
alprioi-invar-def
alprioi-α-def
alprioi-empty-def
alprioi-isEmpty-def
alprioi-insert-def
alprioi-find-def
alprioi-delete-def
alprioi-meld-def

lemmas *alprioi-empty-impl* = *alprio-empty-correct*[*OF ft-empty-impl*, *folded alprioi-defs*]
lemmas *alprioi-isEmpty-impl* = *alprio-isEmpty-correct*[*OF ft-isEmpty-impl*, *folded alprioi-defs*]
lemmas *alprioi-insert-impl* = *alprio-insert-correct*[*OF ft-consl-impl*, *folded alprioi-defs*]
lemmas *alprioi-find-impl* = *alprio-find-correct*[*OF ft-annot-impl*, *folded alprioi-defs*]
lemmas *alprioi-delete-impl* = *alprio-delete-correct*[*OF ft-annot-impl ft-splits-impl ft-app-impl*, *folded alprioi-defs*]
lemmas *alprioi-meld-impl* = *alprio-meld-correct*[*OF ft-app-impl*, *folded alprioi-defs*]

lemmas *alprioi-impl* =
alprioi-empty-impl
alprioi-isEmpty-impl
alprioi-insert-impl
alprioi-find-impl
alprioi-delete-impl
alprioi-meld-impl

interpretation *alprioi*: *prio-empty alprioi-α alprioi-invar alprioi-empty*
⟨*proof*⟩
interpretation *alprioi*: *prio-isEmpty alprioi-α alprioi-invar alprioi-isEmpty*
⟨*proof*⟩
interpretation *alprioi*: *prio-insert alprioi-α alprioi-invar alprioi-insert*
⟨*proof*⟩
interpretation *alprioi*: *prio-find alprioi-α alprioi-invar alprioi-find*
⟨*proof*⟩
interpretation *alprioi*: *prio-delete alprioi-α alprioi-invar alprioi-find alprioi-delete*
⟨*proof*⟩
interpretation *alprioi*: *prio-meld alprioi-α alprioi-invar alprioi-meld*
⟨*proof*⟩

lemmas *alprioi-correct* =
alprioi.empty-correct
alprioi.isEmpty-correct
alprioi.insert-correct
alprioi.meld-correct
alprioi.find-correct
alprioi.delete-correct

Record Based Interface

interpretation *alprioir*: *StdPrio alprioi-ops*
⟨*proof*⟩

4.24.3 Code Generation

— Unfold record definition for code generation
lemma *alprioi-ops-unfold* [*code-unfold*]:
prio-op-α alprioi-ops = *alprioi-α*
prio-op-invar alprioi-ops = *alprioi-invar*
prio-op-empty alprioi-ops = *alprioi-empty*
prio-op-isEmpty alprioi-ops = *alprioi-isEmpty*
prio-op-insert alprioi-ops = *alprioi-insert*
prio-op-find alprioi-ops = *alprioi-find*
prio-op-delete alprioi-ops = *alprioi-delete*
prio-op-meld alprioi-ops = *alprioi-meld*
⟨*proof*⟩

export-code

alprioi-empty
alprioi-isEmpty
alprioi-insert
alprioi-find
alprioi-delete
alprioi-meld
in *SML*
module-name *ALPRIOI*

end

4.25 Implementation of Unique Priority Queues by Finger Trees

theory *FTPrioUniqueImpl*
imports
FTAnnotatedListImpl
../gen-algo/PrioUniqueByAnnotatedList
begin

4.25.1 Definitions

type-synonym ('*a*,'*b*) *aluprioi* = (*unit*, ('*a*, '*b*) *LP*) *FingerTree*

definition *aluprioi-α* :: ('*a*::*linorder*,'*b*::*linorder*) *aluprioi* ⇒ '*a* ⇀ '*b* **where**
aluprioi-α = *aluprio-α ft-α*
definition *aluprioi-invar* **where**
aluprioi-invar = *aluprio-invar ft-α ft-invar*
definition *aluprioi-empty*
:: *unit* ⇒ (*unit*,('*a*::*linorder*,'*b*::*linorder*) *LP*) *FingerTree* **where**
aluprioi-empty = *aluprio-empty ft-empty*
definition *aluprioi-isEmpty* **where**
aluprioi-isEmpty ≡ *aluprio-isEmpty ft-isEmpty*
definition *aluprioi-insert* :: - ⇒ - ⇒ - ⇒ (*unit*,('*a*::*linorder*,'*b*::*linorder*) *LP*) *FingerTree* **where**
aluprioi-insert = *aluprio-insert ft-splits ft-annot ft-isEmpty ft-app ft-consr*
definition *aluprioi-pop* **where**
aluprioi-pop = *aluprio-pop ft-splits ft-annot ft-app*
definition *aluprioi-prio* **where**
aluprioi-prio = *aluprio-prio ft-splits ft-annot ft-isEmpty*

definition *aluprioi-ops* == (|
upr-α = *aluprioi-α*,
upr-invar = *aluprioi-invar*,
upr-empty = *aluprioi-empty*,
upr-isEmpty = *aluprioi-isEmpty*,

upr-insert = *aluprioi-insert*,
upr-pop = *aluprioi-pop*,
upr-prio = *aluprioi-prio*
⟫

lemmas *aluprioi-defs* =
aluprioi-invar-def
aluprioi-α-def
aluprioi-empty-def
aluprioi-isEmpty-def
aluprioi-insert-def
aluprioi-pop-def
aluprioi-prio-def

4.25.2 Correctness

lemmas *aluprioi-finite-impl* = *aluprio-finite-correct*[*of ft-α ft-invar*, *folded aluprioi-defs*]
lemmas *aluprioi-empty-impl* = *aluprio-empty-correct*[*OF ft-empty-impl*, *folded aluprioi-defs*]
lemmas *aluprioi-isEmpty-impl* = *aluprio-isEmpty-correct*[*OF ft-isEmpty-impl*, *folded aluprioi-defs*]
lemmas *aluprioi-insert-impl* = *aluprio-insert-correct*[
OF
ft-splits-impl ft-annot-impl ft-isEmpty-impl
ft-app-impl ft-consr-impl,
folded aluprioi-defs]
lemmas *aluprioi-pop-impl* = *aluprio-pop-correct*[*OF ft-splits-impl ft-annot-impl ft-app-impl*, *folded aluprioi-defs*]
lemmas *aluprioi-prio-impl* = *aluprio-prio-correct*[*OF ft-splits-impl ft-annot-impl ft-isEmpty-impl*, *folded aluprioi-defs*]

lemmas *aluprioi-impl* =
aluprioi-finite-impl
aluprioi-empty-impl
aluprioi-isEmpty-impl
aluprioi-insert-impl
aluprioi-pop-impl
aluprioi-prio-impl

interpretation *aluprioi*: *uprio-finite aluprioi-α aluprioi-invar*
⟨*proof*⟩
interpretation *aluprioi*: *uprio-empty aluprioi-α aluprioi-invar aluprioi-empty*
⟨*proof*⟩
interpretation *aluprioi*: *uprio-isEmpty aluprioi-α aluprioi-invar aluprioi-isEmpty*

⟨*proof*⟩
interpretation *aluprioi*: *uprio-insert aluprioi-α aluprioi-invar aluprioi-insert*
⟨*proof*⟩
interpretation *aluprioi*: *uprio-pop aluprioi-α aluprioi-invar aluprioi-pop*
⟨*proof*⟩

interpretation *aluprioi*: *uprio-prio aluprioi-α aluprioi-invar aluprioi-prio*
⟨*proof*⟩

lemmas *aluprioi-correct* =
aluprioi.finite-correct
aluprioi.empty-correct
aluprioi.isEmpty-correct
aluprioi.insert-correct
aluprioi.pop-correct
aluprioi.prio-correct

Record Based Interface

interpretation *aluprioir*: *StdUprio aluprioi-ops*
⟨*proof*⟩

lemma *aluprioi-ops-unfold* [*code-unfold*]:
upr-empty aluprioi-ops = *aluprioi-empty*
upr-isEmpty aluprioi-ops = *aluprioi-isEmpty*
upr-insert aluprioi-ops = *aluprioi-insert*
upr-pop aluprioi-ops = *aluprioi-pop*
upr-prio aluprioi-ops = *aluprioi-prio*
⟨*proof*⟩

4.25.3 Code Generation

export-code
aluprioi-empty
aluprioi-isEmpty
aluprioi-insert
aluprioi-pop
aluprioi-prio
in *SML*
module-name *ALUPRIOI*

end

theory *ListAdd*
imports *Main*
begin

lemma (**in** *linorder*) *sorted-hd-min*: $[\![xs \neq []; \mathit{sorted}\ xs]\!] \Longrightarrow \forall x \in \mathit{set}\ xs.\ \mathit{hd}\ xs \leq x$
⟨*proof*⟩

lemma *map-hd*: $[\![xs \neq []]\!] \Longrightarrow f\ (\mathit{hd}\ xs) = \mathit{hd}\ (\mathit{map}\ f\ xs)$

⟨*proof*⟩

lemma *map-tl*: *map f (tl xs) = tl (map f xs)*
⟨*proof*⟩

lemma *drop-1-tl*: *drop 1 xs = tl xs*
⟨*proof*⟩

lemma *remove1-tl*: *xs ≠ [] ⟹ remove1 (hd xs) xs = tl xs*
⟨*proof*⟩

lemma *sorted-append-bigger*:
⟦*sorted xs*; ∀ *x* ∈ *set xs*. *x* ≤ *y*⟧ ⟹ *sorted (xs @ [y])*
⟨*proof*⟩

fun *remove-rev* **where**
remove-rev a x [] = a
| *remove-rev a x (y # xs) =*
remove-rev (if x = y then a else (y # a)) x xs

lemma *remove-rev-alt-def* :
remove-rev a x xs = (filter (λy. y ≠ x) (rev xs)) @ a
⟨*proof*⟩

4.25.4 Implementation of Mergesort

fun *merge* :: *'a*::{*linorder*} *list* ⇒ *'a list* ⇒ *'a list* **where**
merge [] l2 = l2
| *merge l1 [] = l1*
| *merge (x1 # l1) (x2 # l2) =*
(if (x1 < x2) then x1 # (merge l1 (x2 # l2)) else
(if (x1 = x2) then x1 # (merge l1 l2) else x2 # (merge (x1 # l1) l2)))

lemma *merge-correct* :
assumes *l1-OK*: *distinct l1 ∧ sorted l1*
assumes *l2-OK*: *distinct l2 ∧ sorted l2*
shows *distinct (merge l1 l2) ∧ sorted (merge l1 l2) ∧ set (merge l1 l2) = set l1*
∪ *set l2*
⟨*proof*⟩

function *merge-list* :: *'a*::{*linorder*} *list list* ⇒ *'a list list* ⇒ *'a list* **where**

```
  merge-list [] [] = []
| merge-list [] [l] = l
| merge-list (la # acc2) [] = merge-list [] (la # acc2)
| merge-list (la # acc2) [l] = merge-list [] (l # la # acc2)
| merge-list acc2 (l1 # l2 # ls) =
   merge-list ((merge l1 l2) # acc2) ls
⟨proof⟩
termination
⟨proof⟩

lemma merge-list-correct :
assumes ls-OK: ⋀l. l ∈ set ls ⟹ distinct l ∧ sorted l
assumes as-OK: ⋀l. l ∈ set as ⟹ distinct l ∧ sorted l
shows distinct (merge-list as ls) ∧ sorted (merge-list as ls) ∧
      set (merge-list as ls) = set (concat (as @ ls))
⟨proof⟩


definition mergesort where
  mergesort xs = merge-list [] (map (λx. [x]) xs)

lemma mergesort-correct :
  distinct (mergesort l) ∧ sorted (mergesort l) ∧ (set (mergesort l) = set l)
⟨proof⟩
```

4.25.5 Operations on sorted Lists

```
fun inter-sorted :: 'a::{linorder} list ⇒ 'a list ⇒ 'a list where
  inter-sorted [] l2 = []
| inter-sorted l1 [] = []
| inter-sorted (x1 # l1) (x2 # l2) =
   (if (x1 < x2) then (inter-sorted l1 (x2 # l2)) else
    (if (x1 = x2) then x1 # (inter-sorted l1 l2) else inter-sorted (x1 # l1) l2))

lemma inter-sorted-correct :
assumes l1-OK: distinct l1 ∧ sorted l1
assumes l2-OK: distinct l2 ∧ sorted l2
shows distinct (inter-sorted l1 l2) ∧ sorted (inter-sorted l1 l2) ∧
      set (inter-sorted l1 l2) = set l1 ∩ set l2
⟨proof⟩

fun diff-sorted :: 'a::{linorder} list ⇒ 'a list ⇒ 'a list where
  diff-sorted [] l2 = []
| diff-sorted l1 [] = l1
| diff-sorted (x1 # l1) (x2 # l2) =
   (if (x1 < x2) then x1 # (diff-sorted l1 (x2 # l2)) else
    (if (x1 = x2) then (diff-sorted l1 l2) else diff-sorted (x1 # l1) l2))

lemma diff-sorted-correct :
```

assumes *l1-OK*: *distinct l1* ∧ *sorted l1*
assumes *l2-OK*: *distinct l2* ∧ *sorted l2*
shows *distinct* (*diff-sorted l1 l2*) ∧ *sorted* (*diff-sorted l1 l2*) ∧
set (*diff-sorted l1 l2*) = *set l1* − *set l2*
⟨*proof*⟩

fun *subset-sorted* :: *'a*::{*linorder*} *list* ⇒ *'a list* ⇒ *bool* **where**
subset-sorted [] *l2* = *True*
| *subset-sorted* (*x1* # *l1*) [] = *False*
| *subset-sorted* (*x1* # *l1*) (*x2* # *l2*) =
(*if* (*x1* < *x2*) *then False else*
(*if* (*x1* = *x2*) *then* (*subset-sorted l1 l2*) *else subset-sorted* (*x1* # *l1*) *l2*))

lemma *subset-sorted-correct* :
assumes *l1-OK*: *distinct l1* ∧ *sorted l1*
assumes *l2-OK*: *distinct l2* ∧ *sorted l2*
shows *subset-sorted l1 l2* ⟷ *set l1* ⊆ *set l2*
⟨*proof*⟩

lemma *set-eq-sorted-correct* :
assumes *l1-OK*: *distinct l1* ∧ *sorted l1*
assumes *l2-OK*: *distinct l2* ∧ *sorted l2*
shows *l1* = *l2* ⟷ *set l1* = *set l2*
⟨*proof*⟩

fun *memb-sorted* **where**
memb-sorted [] *x* = *False*
| *memb-sorted* (*y* # *xs*) *x* =
(*if* (*y* < *x*) *then memb-sorted xs x else* (*x* = *y*))

lemma *memb-sorted-correct* :
sorted xs ⟹ *memb-sorted xs x* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

fun *insertion-sort* **where**
insertion-sort x [] = [*x*]
| *insertion-sort x* (*y* # *xs*) =
(*if* (*y* < *x*) *then y* # *insertion-sort x xs else*
(*if* (*x* = *y*) *then y* # *xs else x* # *y* # *xs*))

lemma *insertion-sort-correct* :
sorted xs ⟹ *distinct xs* ⟹
distinct (*insertion-sort x xs*) ∧
sorted (*insertion-sort x xs*) ∧
set (*insertion-sort x xs*) = *set* (*x* # *xs*)
⟨*proof*⟩

fun *delete-sorted* **where**

```
  delete-sorted x [] = []
| delete-sorted x (y # xs) =
   (if (y < x) then y # delete-sorted x xs else
    (if (x = y) then xs else y # xs))

lemma delete-sorted-correct :
  sorted xs ⟹ distinct xs ⟹
   distinct (delete-sorted x xs) ∧
   sorted (delete-sorted x xs) ∧
   set (delete-sorted x xs) = set xs − {x}
⟨proof⟩


lemma sorted-filter :
  sorted l ⟹ sorted (filter P l)
⟨proof⟩

end
```

4.26 Set Implementation by sorted Lists

```
theory ListSetImpl-Sorted
imports
  ../spec/SetSpec
  ../gen-algo/SetGA
  ../common/Misc
  ../common/ListAdd
  ../common/Dlist-add
begin type-synonym
  'a lss = 'a list
```

4.26.1 Definitions

```
definition lss-α :: 'a lss ⇒ 'a set where lss-α == set
definition lss-invar :: 'a::{linorder} lss ⇒ bool where lss-invar l == distinct l ∧
sorted l
definition lss-empty :: unit ⇒ 'a lss where lss-empty == (λ-::unit. [])
definition lss-memb :: 'a::{linorder} ⇒ 'a lss ⇒ bool where lss-memb x l ==
ListAdd.memb-sorted l x
definition lss-ins :: 'a::{linorder} ⇒ 'a lss ⇒ 'a lss where lss-ins x l == Lis-
tAdd.insertion-sort x l
definition lss-ins-dj :: 'a::{linorder} ⇒ 'a lss ⇒ 'a lss where lss-ins-dj == lss-ins

definition lss-delete :: 'a::{linorder} ⇒ 'a lss ⇒ 'a lss where lss-delete x l ==
delete-sorted x l

definition lss-iterateoi :: 'a lss ⇒ ('a,'σ) set-iterator
where lss-iterateoi l = foldli l
```

definition *lss-reverse-iterateoi* :: *'a lss* ⇒ (*'a*,*'σ*) *set-iterator*
where *lss-reverse-iterateoi l* = *foldli* (*rev l*)

definition *lss-iteratei* :: *'a lss* ⇒ (*'a*,*'σ*) *set-iterator*
where *lss-iteratei l* = *foldli l*

definition *lss-isEmpty* :: *'a lss* ⇒ *bool* **where** *lss-isEmpty s* == *s*=[]

definition *lss-union* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
where *lss-union s1 s2* == *merge s1 s2*
definition *lss-union-list* :: *'a*::{*linorder*} *lss list* ⇒ *'a lss*
where *lss-union-list l* == *merge-list* [] *l*
definition *lss-inter* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
where *lss-inter* == *inter-sorted*
definition *lss-union-dj* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
where *lss-union-dj* == *lss-union* — Union of disjoint sets

definition *lss-image-filter*
where *lss-image-filter f l* =
mergesort (*List.map-filter f l*)

definition *lss-filter* **where** [*code-unfold*]: *lss-filter* = *List.filter*

definition *lss-inj-image-filter*
where *lss-inj-image-filter* == *lss-image-filter*

definition *lss-image* == *iflt-image lss-image-filter*
definition *lss-inj-image* == *iflt-inj-image lss-inj-image-filter*

definition *lss-sel* :: *'a lss* ⇒ (*'a* ⇒ *'r option*) ⇒ *'r option*
where *lss-sel* == *iti-sel lss-iteratei*
definition *lss-sel'* == *iti-sel-no-map lss-iteratei*

definition *lss-to-list* :: *'a lss* ⇒ *'a list* **where** *lss-to-list* == *id*
definition *list-to-lss* :: *'a*::{*linorder*} *list* ⇒ *'a lss* **where** *list-to-lss* == *mergesort*

4.26.2 Correctness

lemmas *lss-defs* =
lss-α-def
lss-invar-def
lss-empty-def
lss-memb-def
lss-ins-def
lss-ins-dj-def
lss-delete-def
lss-iteratei-def
lss-isEmpty-def

lss-union-def
lss-union-list-def
lss-inter-def
lss-union-dj-def
lss-image-filter-def
lss-inj-image-filter-def
lss-image-def
lss-inj-image-def
lss-sel-def
lss-sel'-def
lss-to-list-def
list-to-lss-def

lemma *lss-empty-impl*: *set-empty lss-α lss-invar lss-empty*
⟨*proof*⟩

lemma *lss-memb-impl*: *set-memb lss-α lss-invar lss-memb*
⟨*proof*⟩

lemma *lss-ins-impl*: *set-ins lss-α lss-invar lss-ins*
⟨*proof*⟩

lemma *lss-ins-dj-impl*: *set-ins-dj lss-α lss-invar lss-ins-dj*
⟨*proof*⟩

lemma *lss-delete-impl*: *set-delete lss-α lss-invar lss-delete*
⟨*proof*⟩

lemma *lss-α-finite*[*simp*, *intro*!]: *finite* (*lss-α l*)
⟨*proof*⟩

lemma *lss-is-finite-set*: *finite-set lss-α lss-invar*
⟨*proof*⟩

lemma *lss-iterateoi-impl*: *set-iterateoi lss-α lss-invar lss-iterateoi*
⟨*proof*⟩

lemma *lss-reverse-iterateoi-impl*: *set-reverse-iterateoi lss-α lss-invar lss-reverse-iterateoi*
⟨*proof*⟩

lemma *lss-iteratei-impl*: *set-iteratei lss-α lss-invar lss-iteratei*
⟨*proof*⟩

lemma *lss-isEmpty-impl*: *set-isEmpty lss-α lss-invar lss-isEmpty*
⟨*proof*⟩

lemma *lss-inter-impl*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
⟨*proof*⟩

lemma *lss-union-impl*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*
⟨*proof*⟩

lemma *lss-union-list-impl*: *set-union-list lss-α lss-invar lss-α lss-invar lss-union-list*
⟨*proof*⟩

lemma *lss-union-dj-impl*: *set-union-dj lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union-dj*
⟨*proof*⟩

lemma *lss-image-filter-impl* : *set-image-filter lss-α lss-invar lss-α lss-invar lss-image-filter*
⟨*proof*⟩

lemma *lss-inj-image-filter-impl* : *set-inj-image-filter lss-α lss-invar lss-α lss-invar lss-inj-image-filter*
⟨*proof*⟩

lemma *lss-filter-impl* : *set-filter lss-α lss-invar lss-α lss-invar lss-filter*
⟨*proof*⟩

lemmas *lss-image-impl* = *iflt-image-correct*[*OF lss-image-filter-impl*, *folded lss-image-def*]
lemmas *lss-inj-image-impl* = *iflt-inj-image-correct*[*OF lss-inj-image-filter-impl*, *folded lss-inj-image-def*]

lemmas *lss-sel-impl* = *iti-sel-correct*[*OF lss-iteratei-impl*, *folded lss-sel-def*]
lemmas *lss-sel′-impl* = *iti-sel′-correct*[*OF lss-iteratei-impl*, *folded lss-sel′-def*]

lemma *lss-to-list-impl*: *set-to-list lss-α lss-invar lss-to-list*
⟨*proof*⟩

lemma *list-to-lss-impl*: *list-to-set lss-α lss-invar list-to-lss*
⟨*proof*⟩

interpretation *lss*: *set-empty lss-α lss-invar lss-empty* ⟨*proof*⟩
interpretation *lss*: *set-memb lss-α lss-invar lss-memb* ⟨*proof*⟩
interpretation *lss*: *set-ins lss-α lss-invar lss-ins* ⟨*proof*⟩
interpretation *lss*: *set-ins-dj lss-α lss-invar lss-ins-dj* ⟨*proof*⟩
interpretation *lss*: *set-delete lss-α lss-invar lss-delete* ⟨*proof*⟩
interpretation *lss*: *finite-set lss-α lss-invar* ⟨*proof*⟩
interpretation *lss*: *set-iteratei lss-α lss-invar lss-iteratei* ⟨*proof*⟩
interpretation *lss*: *set-isEmpty lss-α lss-invar lss-isEmpty* ⟨*proof*⟩
interpretation *lss*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*
⟨*proof*⟩
interpretation *lss*: *set-union-list lss-α lss-invar lss-α lss-invar lss-union-list* ⟨*proof*⟩
interpretation *lss*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
⟨*proof*⟩
interpretation *lss*: *set-union-dj lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union-dj*
⟨*proof*⟩
interpretation *lss*: *set-image-filter lss-α lss-invar lss-α lss-invar lss-image-filter*

⟨*proof*⟩
interpretation *lss*: *set-inj-image-filter lss-α lss-invar lss-α lss-invar lss-inj-image-filter* ⟨*proof*⟩
interpretation *lss*: *set-filter lss-α lss-invar lss-α lss-invar lss-filter* ⟨*proof*⟩
interpretation *lss*: *set-image lss-α lss-invar lss-α lss-invar lss-image* ⟨*proof*⟩
interpretation *lss*: *set-inj-image lss-α lss-invar lss-α lss-invar lss-inj-image* ⟨*proof*⟩
interpretation *lss*: *set-sel lss-α lss-invar lss-sel* ⟨*proof*⟩
interpretation *lss*: *set-sel′ lss-α lss-invar lss-sel′* ⟨*proof*⟩
interpretation *lss*: *set-to-list lss-α lss-invar lss-to-list* ⟨*proof*⟩
interpretation *lss*: *list-to-set lss-α lss-invar list-to-lss* ⟨*proof*⟩

declare *lss.finite*[*simp del*, *rule del*]

lemmas *lss-correct* =
lss.empty-correct
lss.memb-correct
lss.ins-correct
lss.ins-dj-correct
lss.delete-correct
lss.isEmpty-correct
lss.union-correct
lss.union-list-correct
lss.inter-correct
lss.union-dj-correct
lss.image-filter-correct
lss.inj-image-filter-correct
lss.image-correct
lss.inj-image-correct
lss.to-list-correct
lss.to-set-correct

4.26.3 Code Generation

export-code
lss-empty
lss-memb
lss-ins
lss-ins-dj
lss-delete
lss-iteratei
lss-isEmpty
lss-union
lss-inter
lss-union
lss-union-list
lss-image-filter
lss-inj-image-filter
lss-image
lss-inj-image

lss-sel
lss-sel′
lss-to-list
list-to-lss
in *SML*
module-name *ListSet*
file –

4.26.4 Things often defined in StdImpl

definition *lss-min* **where** *lss-min = iti-sel-no-map lss-iterateoi*
lemmas *lss-min-impl = itoi-min-correct*[*OF lss-iterateoi-impl*, *folded lss-min-def*]
interpretation *lss*: *set-min lss-α lss-invar lss-min* ⟨*proof*⟩

definition *lss-max* **where** *lss-max = iti-sel-no-map lss-reverse-iterateoi*
lemmas *lss-max-impl = ritoi-max-correct*[*OF lss-reverse-iterateoi-impl*, *folded lss-max-def*]
interpretation *lss*: *set-max lss-α lss-invar lss-max* ⟨*proof*⟩

definition *lss-disjoint-witness == SetGA.sel-disjoint-witness lss-sel lss-memb*
lemmas *lss-disjoint-witness-impl = SetGA.sel-disjoint-witness-correct*[*OF lss-sel-impl lss-memb-impl*, *folded lss-disjoint-witness-def*]
interpretation *lss*: *set-disjoint-witness lss-α lss-invar lss-α lss-invar lss-disjoint-witness* ⟨*proof*⟩

definition *lss-ball == SetGA.iti-ball lss-iteratei*
lemmas *lss-ball-impl = SetGA.iti-ball-correct*[*OF lss-iteratei-impl*, *folded lss-ball-def*]
interpretation *lss*: *set-ball lss-α lss-invar lss-ball* ⟨*proof*⟩

definition *lss-bexists == SetGA.iti-bexists lss-iteratei*
lemmas *lss-bexists-impl = SetGA.iti-bexists-correct*[*OF lss-iteratei-impl*, *folded lss-bexists-def*]
interpretation *lss*: *set-bexists lss-α lss-invar lss-bexists* ⟨*proof*⟩

definition *lss-disjoint == SetGA.ball-disjoint lss-ball lss-memb*
lemmas *lss-disjoint-impl = SetGA.ball-disjoint-correct*[*OF lss-ball-impl lss-memb-impl*, *folded lss-disjoint-def*]
interpretation *lss*: *set-disjoint lss-α lss-invar lss-α lss-invar lss-disjoint* ⟨*proof*⟩

definition *lss-size == SetGA.it-size lss-iteratei*
lemmas *lss-size-impl = SetGA.it-size-correct*[*OF lss-iteratei-impl*, *folded lss-size-def*]
interpretation *lss*: *set-size lss-α lss-invar lss-size* ⟨*proof*⟩

definition *lss-size-abort == SetGA.iti-size-abort lss-iteratei*
lemmas *lss-size-abort-impl = SetGA.iti-size-abort-correct*[*OF lss-iteratei-impl*, *folded lss-size-abort-def*]
interpretation *lss*: *set-size-abort lss-α lss-invar lss-size-abort* ⟨*proof*⟩

definition *lss-isSng == SetGA.sza-isSng lss-iteratei*
lemmas *lss-isSng-impl = SetGA.sza-isSng-correct*[*OF lss-iteratei-impl*, *folded lss-isSng-def*]
interpretation *lss*: *set-isSng lss-α lss-invar lss-isSng* ⟨*proof*⟩

definition *lss-sng x* = [*x*]
lemma *lss-sng-impl* : *set-sng lss-α lss-invar lss-sng*
⟨*proof*⟩
interpretation *lss*: *set-sng lss-α lss-invar lss-sng* ⟨*proof*⟩

definition *lss-equal l1 l2* = (*l1* = *l2*)
lemma *lss-equal-impl* : *set-equal lss-α lss-invar lss-α lss-invar lss-equal*
⟨*proof*⟩
interpretation *lss*: *set-equal lss-α lss-invar lss-α lss-invar lss-equal* ⟨*proof*⟩

definition [*code-unfold*]: *lss-subset* = *subset-sorted*
lemma *lss-subset-impl* : *set-subset lss-α lss-invar lss-α lss-invar lss-subset*
⟨*proof*⟩
interpretation *lss*: *set-subset lss-α lss-invar lss-α lss-invar lss-subset* ⟨*proof*⟩

definition [*code-unfold*]: *lss-diff* = *diff-sorted*
lemma *lss-diff-impl* : *set-diff lss-α lss-invar lss-α lss-invar lss-diff*
⟨*proof*⟩
interpretation *lss*: *set-diff lss-α lss-invar lss-α lss-invar lss-diff* ⟨*proof*⟩

end

4.27 Set Implementation by non-distinct Lists

theory *ListSetImpl-NotDist*
imports
../spec/SetSpec
../gen-algo/SetGA
../common/Misc
../common/ListAdd
begin type-synonym
'a lsnd = *'a list*

4.27.1 Definitions

definition *lsnd-α* :: *'a lsnd* ⇒ *'a set* **where** *lsnd-α* == *set*
definition *lsnd-invar* :: *'a lsnd* ⇒ *bool* **where** *lsnd-invar* == (λ-. *True*)
definition *lsnd-empty* :: *unit* ⇒ *'a lsnd* **where** *lsnd-empty* == (λ-::*unit*. [])
definition *lsnd-memb* :: *'a* ⇒ *'a lsnd* ⇒ *bool* **where** *lsnd-memb x l* == *List.member l x*
definition *lsnd-ins* :: *'a* ⇒ *'a lsnd* ⇒ *'a lsnd* **where** *lsnd-ins x l* == *x*#*l*
definition *lsnd-ins-dj* :: *'a* ⇒ *'a lsnd* ⇒ *'a lsnd* **where** *lsnd-ins-dj x l* == *x*#*l*

definition *lsnd-delete* :: *'a* ⇒ *'a lsnd* ⇒ *'a lsnd* **where** *lsnd-delete x l* == *remove-rev* [] *x l*

definition *lsnd-iteratei* :: *'a lsnd* ⇒ (*'a*,*'σ*) *set-iterator*

where *lsnd-iteratei l = foldli (remdups l)*

definition *lsnd-isEmpty :: 'a lsnd ⇒ bool* **where** *lsnd-isEmpty s == s=[]*

definition *lsnd-union :: 'a lsnd ⇒ 'a lsnd ⇒ 'a lsnd*
where *lsnd-union s1 s2 == revg s1 s2*
definition *lsnd-inter :: 'a lsnd ⇒ 'a lsnd ⇒ 'a lsnd*
where *lsnd-inter == it-inter lsnd-iteratei lsnd-memb lsnd-empty lsnd-ins-dj*
definition *lsnd-union-dj :: 'a lsnd ⇒ 'a lsnd ⇒ 'a lsnd*
where *lsnd-union-dj s1 s2 == revg s1 s2* — Union of disjoint sets

definition *lsnd-image-filter*
where *lsnd-image-filter == it-image-filter lsnd-iteratei lsnd-empty lsnd-ins*

definition *lsnd-inj-image-filter*
where *lsnd-inj-image-filter == it-inj-image-filter lsnd-iteratei lsnd-empty lsnd-ins-dj*

definition *lsnd-image == iflt-image lsnd-image-filter*
definition *lsnd-inj-image == iflt-inj-image lsnd-inj-image-filter*

definition *lsnd-sel :: 'a lsnd ⇒ ('a ⇒ 'r option) ⇒ 'r option*
where *lsnd-sel == iti-sel lsnd-iteratei*
definition *lsnd-sel' == iti-sel-no-map lsnd-iteratei*

definition *lsnd-to-list :: 'a lsnd ⇒ 'a list* **where** *lsnd-to-list == remdups*
definition *list-to-lsnd :: 'a list ⇒ 'a lsnd* **where** *list-to-lsnd == id*

4.27.2 Correctness

lemmas *lsnd-defs =*
lsnd-α-def
lsnd-invar-def
lsnd-empty-def
lsnd-memb-def
lsnd-ins-def
lsnd-ins-dj-def
lsnd-delete-def
lsnd-iteratei-def
lsnd-isEmpty-def
lsnd-union-def
lsnd-inter-def
lsnd-union-dj-def
lsnd-image-filter-def
lsnd-inj-image-filter-def
lsnd-image-def
lsnd-inj-image-def
lsnd-sel-def
lsnd-sel'-def
lsnd-to-list-def

list-to-lsnd-def

lemma *lsnd-empty-impl*: *set-empty lsnd-α lsnd-invar lsnd-empty*
⟨*proof*⟩

lemma *lsnd-memb-impl*: *set-memb lsnd-α lsnd-invar lsnd-memb*
⟨*proof*⟩

lemma *lsnd-ins-impl*: *set-ins lsnd-α lsnd-invar lsnd-ins*
⟨*proof*⟩

lemma *lsnd-ins-dj-impl*: *set-ins-dj lsnd-α lsnd-invar lsnd-ins-dj*
⟨*proof*⟩

lemma *lsnd-delete-impl*: *set-delete lsnd-α lsnd-invar lsnd-delete*
⟨*proof*⟩

lemma *lsnd-α-finite*[*simp*, *intro*!]: *finite* (*lsnd-α l*)
⟨*proof*⟩

lemma *lsnd-is-finite-set*: *finite-set lsnd-α lsnd-invar*
⟨*proof*⟩

lemma *lsnd-iteratei-impl*: *set-iteratei lsnd-α lsnd-invar lsnd-iteratei*
⟨*proof*⟩

lemma *lsnd-isEmpty-impl*: *set-isEmpty lsnd-α lsnd-invar lsnd-isEmpty*
⟨*proof*⟩

lemmas *lsnd-inter-impl* = *it-inter-correct*[*OF lsnd-iteratei-impl lsnd-memb-impl lsnd-empty-impl lsnd-ins-dj-impl*, *folded lsnd-inter-def*]

lemma *lsnd-union-impl*: *set-union lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union*
⟨*proof*⟩

lemma *lsnd-union-dj-impl*: *set-union-dj lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union-dj*
⟨*proof*⟩

lemmas *lsnd-image-filter-impl* = *it-image-filter-correct*[*OF lsnd-iteratei-impl lsnd-empty-impl lsnd-ins-impl*, *folded lsnd-image-filter-def*]
lemmas *lsnd-inj-image-filter-impl* = *it-inj-image-filter-correct*[*OF lsnd-iteratei-impl lsnd-empty-impl lsnd-ins-dj-impl*, *folded lsnd-inj-image-filter-def*]

lemmas *lsnd-image-impl* = *iflt-image-correct*[*OF lsnd-image-filter-impl*, *folded lsnd-image-def*]
lemmas *lsnd-inj-image-impl* = *iflt-inj-image-correct*[*OF lsnd-inj-image-filter-impl*, *folded lsnd-inj-image-def*]

lemmas *lsnd-sel-impl* = *iti-sel-correct*[*OF lsnd-iteratei-impl*, *folded lsnd-sel-def*]
lemmas *lsnd-sel'-impl* = *iti-sel'-correct*[*OF lsnd-iteratei-impl*, *folded lsnd-sel'-def*]

lemma *lsnd-to-list-impl*: *set-to-list lsnd-α lsnd-invar lsnd-to-list*
⟨*proof*⟩

lemma *list-to-lsnd-impl*: *list-to-set lsnd-α lsnd-invar list-to-lsnd*
⟨*proof*⟩

interpretation *lsnd*: *set-empty lsnd-α lsnd-invar lsnd-empty* ⟨*proof*⟩
interpretation *lsnd*: *set-memb lsnd-α lsnd-invar lsnd-memb* ⟨*proof*⟩
interpretation *lsnd*: *set-ins lsnd-α lsnd-invar lsnd-ins* ⟨*proof*⟩
interpretation *lsnd*: *set-ins-dj lsnd-α lsnd-invar lsnd-ins-dj* ⟨*proof*⟩
interpretation *lsnd*: *set-delete lsnd-α lsnd-invar lsnd-delete* ⟨*proof*⟩
interpretation *lsnd*: *finite-set lsnd-α lsnd-invar* ⟨*proof*⟩
interpretation *lsnd*: *set-iteratei lsnd-α lsnd-invar lsnd-iteratei* ⟨*proof*⟩
interpretation *lsnd*: *set-isEmpty lsnd-α lsnd-invar lsnd-isEmpty* ⟨*proof*⟩
interpretation *lsnd*: *set-union lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar*
lsnd-union ⟨*proof*⟩
interpretation *lsnd*: *set-inter lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar*
lsnd-inter ⟨*proof*⟩
interpretation *lsnd*: *set-union-dj lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar*
lsnd-union-dj ⟨*proof*⟩
interpretation *lsnd*: *set-image-filter lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-image-filter*
⟨*proof*⟩
interpretation *lsnd*: *set-inj-image-filter lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-inj-image-filter*
⟨*proof*⟩
interpretation *lsnd*: *set-image lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-image* ⟨*proof*⟩
interpretation *lsnd*: *set-inj-image lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-inj-image*
⟨*proof*⟩
interpretation *lsnd*: *set-sel lsnd-α lsnd-invar lsnd-sel* ⟨*proof*⟩
interpretation *lsnd*: *set-sel' lsnd-α lsnd-invar lsnd-sel'* ⟨*proof*⟩
interpretation *lsnd*: *set-to-list lsnd-α lsnd-invar lsnd-to-list* ⟨*proof*⟩
interpretation *lsnd*: *list-to-set lsnd-α lsnd-invar list-to-lsnd* ⟨*proof*⟩

declare *lsnd.finite*[*simp del*, *rule del*]

lemmas *lsnd-correct* =
lsnd.empty-correct
lsnd.memb-correct
lsnd.ins-correct
lsnd.ins-dj-correct
lsnd.delete-correct
lsnd.isEmpty-correct
lsnd.union-correct
lsnd.inter-correct
lsnd.union-dj-correct
lsnd.image-filter-correct
lsnd.inj-image-filter-correct

lsnd.image-correct
lsnd.inj-image-correct
lsnd.to-list-correct
lsnd.to-set-correct

4.27.3 Code Generation

export-code
lsnd-empty
lsnd-memb
lsnd-ins
lsnd-ins-dj
lsnd-delete
lsnd-iteratei
lsnd-isEmpty
lsnd-union
lsnd-inter
lsnd-union-dj
lsnd-image-filter
lsnd-inj-image-filter
lsnd-image
lsnd-inj-image
lsnd-sel
lsnd-sel′
lsnd-to-list
list-to-lsnd
in *SML*
module-name *ListSet*
file –

4.27.4 Things often defined in StdImpl

definition *lsnd-disjoint-witness* == *SetGA.sel-disjoint-witness lsnd-sel lsnd-memb*
lemmas *lsnd-disjoint-witness-impl* = *SetGA.sel-disjoint-witness-correct*[*OF lsnd-sel-impl lsnd-memb-impl*, *folded lsnd-disjoint-witness-def*]
interpretation *lsnd*: *set-disjoint-witness lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-disjoint-witness* ⟨*proof*⟩

definition *lsnd-ball* == *SetGA.iti-ball lsnd-iteratei*
lemmas *lsnd-ball-impl* = *SetGA.iti-ball-correct*[*OF lsnd-iteratei-impl*, *folded lsnd-ball-def*]
interpretation *lsnd*: *set-ball lsnd-α lsnd-invar lsnd-ball* ⟨*proof*⟩

definition *lsnd-bexists* == *SetGA.iti-bexists lsnd-iteratei*
lemmas *lsnd-bexists-impl* = *SetGA.iti-bexists-correct*[*OF lsnd-iteratei-impl*, *folded lsnd-bexists-def*]
interpretation *lsnd*: *set-bexists lsnd-α lsnd-invar lsnd-bexists* ⟨*proof*⟩

definition *lsnd-disjoint* == *SetGA.ball-disjoint lsnd-ball lsnd-memb*
lemmas *lsnd-disjoint-impl* = *SetGA.ball-disjoint-correct*[*OF lsnd-ball-impl lsnd-memb-impl*, *folded lsnd-disjoint-def*]

interpretation *lsnd*: *set-disjoint lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-disjoint*
⟨*proof*⟩

definition *lsnd-size* == *SetGA.it-size lsnd-iteratei*
lemmas *lsnd-size-impl* = *SetGA.it-size-correct*[*OF lsnd-iteratei-impl*, *folded lsnd-size-def*]
interpretation *lsnd*: *set-size lsnd-α lsnd-invar lsnd-size* ⟨*proof*⟩

definition *lsnd-size-abort* == *SetGA.iti-size-abort lsnd-iteratei*
lemmas *lsnd-size-abort-impl* = *SetGA.iti-size-abort-correct*[*OF lsnd-iteratei-impl*,
folded lsnd-size-abort-def]
interpretation *lsnd*: *set-size-abort lsnd-α lsnd-invar lsnd-size-abort* ⟨*proof*⟩

definition *lsnd-isSng* == *SetGA.sza-isSng lsnd-iteratei*
lemmas *lsnd-isSng-impl* = *SetGA.sza-isSng-correct*[*OF lsnd-iteratei-impl*, *folded*
lsnd-isSng-def]
interpretation *lsnd*: *set-isSng lsnd-α lsnd-invar lsnd-isSng* ⟨*proof*⟩

definition *lsnd-sng x* = [*x*]
lemma *lsnd-sng-impl* : *set-sng lsnd-α lsnd-invar lsnd-sng*
⟨*proof*⟩
interpretation *lsnd*: *set-sng lsnd-α lsnd-invar lsnd-sng* ⟨*proof*⟩

definition *lsnd-diff* == *SetGA.it-diff lsnd-iteratei lsnd-delete*
lemmas *lsnd-diff-impl* = *SetGA.it-diff-correct*[*OF lsnd-iteratei-impl lsnd-delete-impl*,
folded lsnd-diff-def]
interpretation *lsnd*: *set-diff lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-diff*
⟨*proof*⟩

definition *lsnd-subset* == *SetGA.ball-subset lsnd-ball lsnd-memb*
lemmas *lsnd-subset-impl* = *SetGA.ball-subset-correct*[*OF lsnd-ball-impl lsnd-memb-impl*,
folded lsnd-subset-def]
interpretation *lsnd*: *set-subset lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-subset*
⟨*proof*⟩

definition *lsnd-equal* == *SetGA.subset-equal lsnd-subset lsnd-subset*
lemmas *lsnd-equal-impl* = *SetGA.subset-equal-correct*[*OF lsnd-subset-impl lsnd-subset-impl*,
folded lsnd-equal-def]
interpretation *lsnd*: *set-equal lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-equal*
⟨*proof*⟩

definition *lsnd-filter* == *SetGA.iflt-filter lsnd-inj-image-filter*
lemmas *lsnd-filter-impl* = *SetGA.iflt-filter-correct*[*OF lsnd-inj-image-filter-impl*,
folded lsnd-filter-def]
interpretation *lsnd*: *set-filter lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-filter*
⟨*proof*⟩

end

4.28 Record-Based Set Interface: Implementation setup

theory *RecordSetImpl*
imports
../gen-algo/StdInst
ListSetImpl-Sorted
ListSetImpl-NotDist
begin

4.28.1 List Set

definition *ls-ops* = (|
set-op-α = *ls-α*,
set-op-invar = *ls-invar*,
set-op-empty = *ls-empty*,
set-op-sng = *ls-sng*,
set-op-memb = *ls-memb*,
set-op-ins = *ls-ins*,
set-op-ins-dj = *ls-ins-dj*,
set-op-delete = *ls-delete*,
set-op-isEmpty = *ls-isEmpty*,
set-op-isSng = *ls-isSng*,
set-op-ball = *ls-ball*,
set-op-bexists = *ls-bexists*,
set-op-size = *ls-size*,
set-op-size-abort = *ls-size-abort*,
set-op-union = *ls-union*,
set-op-union-dj = *ls-union-dj*,
set-op-diff = *ll-diff*,
set-op-filter = *ll-filter*,
set-op-inter = *ls-inter*,
set-op-subset = *ll-subset*,
set-op-equal = *ll-equal*,
set-op-disjoint = *ll-disjoint*,
set-op-disjoint-witness = *ll-disjoint-witness*,
set-op-sel = *ls-sel′*,
set-op-to-list = *ls-to-list*,
set-op-from-list = *list-to-ls*
|)

interpretation *lsr*!: *StdSet ls-ops*
⟨*proof*⟩

lemma *ls-ops-unfold*[*code-unfold*]:
set-op-α ls-ops = *ls-α*
set-op-invar ls-ops = *ls-invar*
set-op-empty ls-ops = *ls-empty*
set-op-sng ls-ops = *ls-sng*

set-op-memb ls-ops = ls-memb
set-op-ins ls-ops = ls-ins
set-op-ins-dj ls-ops = ls-ins-dj
set-op-delete ls-ops = ls-delete
set-op-isEmpty ls-ops = ls-isEmpty
set-op-isSng ls-ops = ls-isSng
set-op-ball ls-ops = ls-ball
set-op-bexists ls-ops = ls-bexists
set-op-size ls-ops = ls-size
set-op-size-abort ls-ops = ls-size-abort
set-op-union ls-ops = ls-union
set-op-union-dj ls-ops = ls-union-dj
set-op-diff ls-ops = ll-diff
set-op-filter ls-ops = ll-filter
set-op-inter ls-ops = ls-inter
set-op-subset ls-ops = ll-subset
set-op-equal ls-ops = ll-equal
set-op-disjoint ls-ops = ll-disjoint
set-op-disjoint-witness ls-ops = ll-disjoint-witness
set-op-sel ls-ops = ls-sel′
set-op-to-list ls-ops = ls-to-list
set-op-from-list ls-ops = list-to-ls
⟨*proof*⟩
interpretation *lsr*!: *set-iteratei* (*set-op-α ls-ops*) (*set-op-invar ls-ops*) *ls-iteratei*
⟨*proof*⟩

4.28.2 List Set with Invar

definition *lsi-ops* = (|
set-op-α = *lsi-α*,
set-op-invar = *lsi-invar*,
set-op-empty = *lsi-empty*,
set-op-sng = *lsi-sng*,
set-op-memb = *lsi-memb*,
set-op-ins = *lsi-ins*,
set-op-ins-dj = *lsi-ins-dj*,
set-op-delete = *lsi-delete*,
set-op-isEmpty = *lsi-isEmpty*,
set-op-isSng = *lsi-isSng*,
set-op-ball = *lsi-ball*,
set-op-bexists = *lsi-bexists*,
set-op-size = *lsi-size*,
set-op-size-abort = *lsi-size-abort*,
set-op-union = *lsi-union*,
set-op-union-dj = *lsi-union-dj*,
set-op-diff = *lili-diff*,
set-op-filter = *lili-filter*,
set-op-inter = *lsi-inter*,
set-op-subset = *lili-subset*,

set-op-equal = *lili-equal*,
set-op-disjoint = *lili-disjoint*,
set-op-disjoint-witness = *lili-disjoint-witness*,
set-op-sel = *lsi-sel′*,
set-op-to-list = *lsi-to-list*,
set-op-from-list = *list-to-lsi*
⦈

interpretation *lsir*!: *StdSet lsi-ops*
⟨*proof*⟩

lemma *lsi-ops-unfold*[*code-unfold*]:
set-op-α lsi-ops = *lsi-α*
set-op-invar lsi-ops = *lsi-invar*
set-op-empty lsi-ops = *lsi-empty*
set-op-sng lsi-ops = *lsi-sng*
set-op-memb lsi-ops = *lsi-memb*
set-op-ins lsi-ops = *lsi-ins*
set-op-ins-dj lsi-ops = *lsi-ins-dj*
set-op-delete lsi-ops = *lsi-delete*
set-op-isEmpty lsi-ops = *lsi-isEmpty*
set-op-isSng lsi-ops = *lsi-isSng*
set-op-ball lsi-ops = *lsi-ball*
set-op-bexists lsi-ops = *lsi-bexists*
set-op-size lsi-ops = *lsi-size*
set-op-size-abort lsi-ops = *lsi-size-abort*
set-op-union lsi-ops = *lsi-union*
set-op-union-dj lsi-ops = *lsi-union-dj*
set-op-diff lsi-ops = *lili-diff*
set-op-filter lsi-ops = *lili-filter*
set-op-inter lsi-ops = *lsi-inter*
set-op-subset lsi-ops = *lili-subset*
set-op-equal lsi-ops = *lili-equal*
set-op-disjoint lsi-ops = *lili-disjoint*
set-op-disjoint-witness lsi-ops = *lili-disjoint-witness*
set-op-sel lsi-ops = *lsi-sel′*
set-op-to-list lsi-ops = *lsi-to-list*
set-op-from-list lsi-ops = *list-to-lsi*
⟨*proof*⟩
interpretation *lsir*!: *set-iteratei* (*set-op-α lsi-ops*) (*set-op-invar lsi-ops*) *lsi-iteratei*
⟨*proof*⟩

4.28.3 List Set with Invar and non-distinct lists

definition *lsnd-ops* = (⦇
set-op-α = *lsnd-α*,
set-op-invar = *lsnd-invar*,
set-op-empty = *lsnd-empty*,
set-op-sng = *lsnd-sng*,

set-op-memb = *lsnd-memb*,
set-op-ins = *lsnd-ins*,
set-op-ins-dj = *lsnd-ins-dj*,
set-op-delete = *lsnd-delete*,
set-op-isEmpty = *lsnd-isEmpty*,
set-op-isSng = *lsnd-isSng*,
set-op-ball = *lsnd-ball*,
set-op-bexists = *lsnd-bexists*,
set-op-size = *lsnd-size*,
set-op-size-abort = *lsnd-size-abort*,
set-op-union = *lsnd-union*,
set-op-union-dj = *lsnd-union-dj*,
set-op-diff = *lsnd-diff*,
set-op-filter = *lsnd-filter*,
set-op-inter = *lsnd-inter*,
set-op-subset = *lsnd-subset*,
set-op-equal = *lsnd-equal*,
set-op-disjoint = *lsnd-disjoint*,
set-op-disjoint-witness = *lsnd-disjoint-witness*,
set-op-sel = *lsnd-sel′*,
set-op-to-list = *lsnd-to-list*,
set-op-from-list = *list-to-lsnd*
⦈

interpretation *lsndr*!: *StdSet lsnd-ops*
⟨*proof*⟩

lemma *lsnd-ops-unfold*[*code-unfold*]:
set-op-α lsnd-ops = *lsnd-α*
set-op-invar lsnd-ops = *lsnd-invar*
set-op-empty lsnd-ops = *lsnd-empty*
set-op-sng lsnd-ops = *lsnd-sng*
set-op-memb lsnd-ops = *lsnd-memb*
set-op-ins lsnd-ops = *lsnd-ins*
set-op-ins-dj lsnd-ops = *lsnd-ins-dj*
set-op-delete lsnd-ops = *lsnd-delete*
set-op-isEmpty lsnd-ops = *lsnd-isEmpty*
set-op-isSng lsnd-ops = *lsnd-isSng*
set-op-ball lsnd-ops = *lsnd-ball*
set-op-bexists lsnd-ops = *lsnd-bexists*
set-op-size lsnd-ops = *lsnd-size*
set-op-size-abort lsnd-ops = *lsnd-size-abort*
set-op-union lsnd-ops = *lsnd-union*
set-op-union-dj lsnd-ops = *lsnd-union-dj*
set-op-diff lsnd-ops = *lsnd-diff*
set-op-filter lsnd-ops = *lsnd-filter*
set-op-inter lsnd-ops = *lsnd-inter*
set-op-subset lsnd-ops = *lsnd-subset*
set-op-equal lsnd-ops = *lsnd-equal*

set-op-disjoint lsnd-ops = lsnd-disjoint
set-op-disjoint-witness lsnd-ops = lsnd-disjoint-witness
set-op-sel lsnd-ops = lsnd-sel′
set-op-to-list lsnd-ops = lsnd-to-list
set-op-from-list lsnd-ops = list-to-lsnd
⟨*proof*⟩
interpretation *lsndr*!: *set-iteratei* (*set-op-α lsnd-ops*) (*set-op-invar lsnd-ops*) *lsnd-iteratei* ⟨*proof*⟩

4.28.4 List Set by sorted lists

definition *lss-ops* :: (′*x* :: *linorder*, ′*x lss*) *oset-ops*
where *lss-ops* = (|
set-op-α = *lss-α*,
set-op-invar = *lss-invar*,
set-op-empty = *lss-empty*,
set-op-sng = *lss-sng*,
set-op-memb = *lss-memb*,
set-op-ins = *lss-ins*,
set-op-ins-dj = *lss-ins-dj*,
set-op-delete = *lss-delete*,
set-op-isEmpty = *lss-isEmpty*,
set-op-isSng = *lss-isSng*,
set-op-ball = *lss-ball*,
set-op-bexists = *lss-bexists*,
set-op-size = *lss-size*,
set-op-size-abort = *lss-size-abort*,
set-op-union = *lss-union*,
set-op-union-dj = *lss-union-dj*,
set-op-diff = *lss-diff*,
set-op-filter = *lss-filter*,
set-op-inter = *lss-inter*,
set-op-subset = *lss-subset*,
set-op-equal = *lss-equal*,
set-op-disjoint = *lss-disjoint*,
set-op-disjoint-witness = *lss-disjoint-witness*,
set-op-sel = *lss-sel′*,
set-op-to-list = *lss-to-list*,
set-op-from-list = *list-to-lss*,
set-op-min = *lss-min*,
set-op-max = *lss-max*
|)

interpretation *lssr*!: *StdSet lss-ops*
⟨*proof*⟩

interpretation *lssr*!: *StdOSet lss-ops*
⟨*proof*⟩

lemma *lss-ops-unfold*[*code-unfold*]:
set-op-α lss-ops = lss-α
set-op-invar lss-ops = lss-invar
set-op-empty lss-ops = lss-empty
set-op-sng lss-ops = lss-sng
set-op-memb lss-ops = lss-memb
set-op-ins lss-ops = lss-ins
set-op-ins-dj lss-ops = lss-ins-dj
set-op-delete lss-ops = lss-delete
set-op-isEmpty lss-ops = lss-isEmpty
set-op-isSng lss-ops = lss-isSng
set-op-ball lss-ops = lss-ball
set-op-size lss-ops = lss-size
set-op-size-abort lss-ops = lss-size-abort
set-op-union lss-ops = lss-union
set-op-union-dj lss-ops = lss-union-dj
set-op-diff lss-ops = lss-diff
set-op-filter lss-ops = lss-filter
set-op-inter lss-ops = lss-inter
set-op-subset lss-ops = lss-subset
set-op-equal lss-ops = lss-equal
set-op-disjoint lss-ops = lss-disjoint
set-op-disjoint-witness lss-ops = lss-disjoint-witness
set-op-sel lss-ops = lss-sel′
set-op-to-list lss-ops = lss-to-list
set-op-from-list lss-ops = list-to-lss
set-op-min lss-ops = lss-min
set-op-max lss-ops = lss-max
⟨*proof*⟩
interpretation *lssr*!: *set-iteratei* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-iteratei*
⟨*proof*⟩

interpretation *lssr*!: *set-iterateoi* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-iterateoi*
⟨*proof*⟩

interpretation *lssr*!: *set-reverse-iterateoi* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-reverse-iterateoi*
⟨*proof*⟩

4.28.5 RBT Set

definition *rs-ops* :: (*'x* :: *linorder*, *'x rs*) *oset-ops*
where *rs-ops* = (|
set-op-α = *rs-α*,
set-op-invar = *rs-invar*,
set-op-empty = *rs-empty*,
set-op-sng = *rs-sng*,

set-op-memb = *rs-memb*,
set-op-ins = *rs-ins*,
set-op-ins-dj = *rs-ins-dj*,
set-op-delete = *rs-delete*,
set-op-isEmpty = *rs-isEmpty*,
set-op-isSng = *rs-isSng*,
set-op-ball = *rs-ball*,
set-op-bexists = *rs-bexists*,
set-op-size = *rs-size*,
set-op-size-abort = *rs-size-abort*,
set-op-union = *rs-union*,
set-op-union-dj = *rs-union-dj*,
set-op-diff = *rr-diff*,
set-op-filter = *rr-filter*,
set-op-inter = *rs-inter*,
set-op-subset = *rr-subset*,
set-op-equal = *rr-equal*,
set-op-disjoint = *rr-disjoint*,
set-op-disjoint-witness = *rr-disjoint-witness*,
set-op-sel = *rs-sel′*,
set-op-to-list = *rs-to-list*,
set-op-from-list = *list-to-rs*,
set-op-min = *rs-min*,
set-op-max = *rs-max*
⦇

interpretation *rsr*!: *StdSet rs-ops*
⟨*proof*⟩

interpretation *rsr*!: *StdOSet rs-ops*
⟨*proof*⟩

lemma *rs-ops-unfold*[*code-unfold*]:
set-op-α rs-ops = *rs-α*
set-op-invar rs-ops = *rs-invar*
set-op-empty rs-ops = *rs-empty*
set-op-sng rs-ops = *rs-sng*
set-op-memb rs-ops = *rs-memb*
set-op-ins rs-ops = *rs-ins*
set-op-ins-dj rs-ops = *rs-ins-dj*
set-op-delete rs-ops = *rs-delete*
set-op-isEmpty rs-ops = *rs-isEmpty*
set-op-isSng rs-ops = *rs-isSng*
set-op-ball rs-ops = *rs-ball*
set-op-bexists rs-ops = *rs-bexists*
set-op-size rs-ops = *rs-size*
set-op-size-abort rs-ops = *rs-size-abort*
set-op-union rs-ops = *rs-union*
set-op-union-dj rs-ops = *rs-union-dj*

set-op-diff rs-ops = rr-diff
set-op-filter rs-ops = rr-filter
set-op-inter rs-ops = rs-inter
set-op-subset rs-ops = rr-subset
set-op-equal rs-ops = rr-equal
set-op-disjoint rs-ops = rr-disjoint
set-op-disjoint-witness rs-ops = rr-disjoint-witness
set-op-sel rs-ops = rs-sel′
set-op-to-list rs-ops = rs-to-list
set-op-from-list rs-ops = list-to-rs
set-op-min rs-ops = rs-min
set-op-max rs-ops = rs-max
⟨*proof*⟩
interpretation *rsr*!: *set-iteratei* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-iteratei*
⟨*proof*⟩

interpretation *rsr*!: *set-iterateoi* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-iterateoi*
⟨*proof*⟩

interpretation *rsr*!: *set-reverse-iterateoi* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-reverse-iterateoi*
⟨*proof*⟩

4.28.6 HashSet

definition *hs-ops* = (|
set-op-α = *hs-α*,
set-op-invar = *hs-invar*,
set-op-empty = *hs-empty*,
set-op-sng = *hs-sng*,
set-op-memb = *hs-memb*,
set-op-ins = *hs-ins*,
set-op-ins-dj = *hs-ins-dj*,
set-op-delete = *hs-delete*,
set-op-isEmpty = *hs-isEmpty*,
set-op-isSng = *hs-isSng*,
set-op-ball = *hs-ball*,
set-op-bexists = *hs-bexists*,
set-op-size = *hs-size*,
set-op-size-abort = *hs-size-abort*,
set-op-union = *hs-union*,
set-op-union-dj = *hs-union-dj*,
set-op-diff = *hh-diff*,
set-op-filter = *hh-filter*,
set-op-inter = *hs-inter*,
set-op-subset = *hh-subset*,
set-op-equal = *hh-equal*,
set-op-disjoint = *hh-disjoint*,
set-op-disjoint-witness = *hh-disjoint-witness*,

set-op-sel = *hs-sel′*,
set-op-to-list = *hs-to-list*,
set-op-from-list = *list-to-hs*
(|)

interpretation *hsr*!: *StdSet hs-ops*
⟨*proof*⟩

lemma *hs-ops-unfold*[*code-unfold*]:
set-op-α hs-ops = *hs-α*
set-op-invar hs-ops = *hs-invar*
set-op-empty hs-ops = *hs-empty*
set-op-sng hs-ops = *hs-sng*
set-op-memb hs-ops = *hs-memb*
set-op-ins hs-ops = *hs-ins*
set-op-ins-dj hs-ops = *hs-ins-dj*
set-op-delete hs-ops = *hs-delete*
set-op-isEmpty hs-ops = *hs-isEmpty*
set-op-isSng hs-ops = *hs-isSng*
set-op-ball hs-ops = *hs-ball*
set-op-bexists hs-ops = *hs-bexists*
set-op-size hs-ops = *hs-size*
set-op-size-abort hs-ops = *hs-size-abort*
set-op-union hs-ops = *hs-union*
set-op-union-dj hs-ops = *hs-union-dj*
set-op-diff hs-ops = *hh-diff*
set-op-filter hs-ops = *hh-filter*
set-op-inter hs-ops = *hs-inter*
set-op-subset hs-ops = *hh-subset*
set-op-equal hs-ops = *hh-equal*
set-op-disjoint hs-ops = *hh-disjoint*
set-op-disjoint-witness hs-ops = *hh-disjoint-witness*
set-op-sel hs-ops = *hs-sel′*
set-op-to-list hs-ops = *hs-to-list*
set-op-from-list hs-ops = *list-to-hs*
⟨*proof*⟩
interpretation *hsr*!: *set-iteratei* (*set-op-α hs-ops*) (*set-op-invar hs-ops*) *hs-iteratei*
⟨*proof*⟩

4.28.7 Array Hash Set

definition *ahs-ops* = (|
set-op-α = *ahs-α*,
set-op-invar = *ahs-invar*,
set-op-empty = *ahs-empty*,
set-op-sng = *ahs-sng*,
set-op-memb = *ahs-memb*,
set-op-ins = *ahs-ins*,
set-op-ins-dj = *ahs-ins-dj*,

set-op-delete = *ahs-delete*,
set-op-isEmpty = *ahs-isEmpty*,
set-op-isSng = *ahs-isSng*,
set-op-ball = *ahs-ball*,
set-op-bexists = *ahs-bexists*,
set-op-size = *ahs-size*,
set-op-size-abort = *ahs-size-abort*,
set-op-union = *ahs-union*,
set-op-union-dj = *ahs-union-dj*,
set-op-diff = *aa-diff*,
set-op-filter = *aa-filter*,
set-op-inter = *ahs-inter*,
set-op-subset = *aa-subset*,
set-op-equal = *aa-equal*,
set-op-disjoint = *aa-disjoint*,
set-op-disjoint-witness = *aa-disjoint-witness*,
set-op-sel = *ahs-sel′*,
set-op-to-list = *ahs-to-list*,
set-op-from-list = *list-to-ahs*
⦇

interpretation *ahsr*!: *StdSet ahs-ops*
⟨*proof*⟩

lemma *ahs-ops-unfold*[*code-unfold*]:
set-op-α ahs-ops = *ahs-α*
set-op-invar ahs-ops = *ahs-invar*
set-op-empty ahs-ops = *ahs-empty*
set-op-sng ahs-ops = *ahs-sng*
set-op-memb ahs-ops = *ahs-memb*
set-op-ins ahs-ops = *ahs-ins*
set-op-ins-dj ahs-ops = *ahs-ins-dj*
set-op-delete ahs-ops = *ahs-delete*
set-op-isEmpty ahs-ops = *ahs-isEmpty*
set-op-isSng ahs-ops = *ahs-isSng*
set-op-ball ahs-ops = *ahs-ball*
set-op-bexists ahs-ops = *ahs-bexists*
set-op-size ahs-ops = *ahs-size*
set-op-size-abort ahs-ops = *ahs-size-abort*
set-op-union ahs-ops = *ahs-union*
set-op-union-dj ahs-ops = *ahs-union-dj*
set-op-diff ahs-ops = *aa-diff*
set-op-filter ahs-ops = *aa-filter*
set-op-inter ahs-ops = *ahs-inter*
set-op-subset ahs-ops = *aa-subset*
set-op-equal ahs-ops = *aa-equal*
set-op-disjoint ahs-ops = *aa-disjoint*
set-op-disjoint-witness ahs-ops = *aa-disjoint-witness*
set-op-sel ahs-ops = *ahs-sel′*

set-op-to-list ahs-ops = ahs-to-list
set-op-from-list ahs-ops = list-to-ahs
⟨*proof*⟩
interpretation *ahsr*!: *set-iteratei* (*set-op-α ahs-ops*) (*set-op-invar ahs-ops*) *ahs-iteratei* ⟨*proof*⟩

4.28.8 Array Set

definition *ias-ops* = (|
set-op-α = ias-α,
set-op-invar = ias-invar,
set-op-empty = ias-empty,
set-op-sng = ias-sng,
set-op-memb = ias-memb,
set-op-ins = ias-ins,
set-op-ins-dj = ias-ins-dj,
set-op-delete = ias-delete,
set-op-isEmpty = ias-isEmpty,
set-op-isSng = ias-isSng,
set-op-ball = ias-ball,
set-op-bexists = ias-bexists,
set-op-size = ias-size,
set-op-size-abort = ias-size-abort,
set-op-union = ias-union,
set-op-union-dj = ias-union-dj,
set-op-diff = isis-diff,
set-op-filter = isis-filter,
set-op-inter = ias-inter,
set-op-subset = isis-subset,
set-op-equal = isis-equal,
set-op-disjoint = isis-disjoint,
set-op-disjoint-witness = isis-disjoint-witness,
set-op-sel = ias-sel′,
set-op-to-list = ias-to-list,
set-op-from-list = list-to-ias
|)

interpretation *iasr*!: *StdSet ias-ops*
⟨*proof*⟩

lemma *ias-ops-unfold*[*code-unfold*]:
set-op-α ias-ops = ias-α
set-op-invar ias-ops = ias-invar
set-op-empty ias-ops = ias-empty
set-op-sng ias-ops = ias-sng
set-op-memb ias-ops = ias-memb
set-op-ins ias-ops = ias-ins
set-op-ins-dj ias-ops = ias-ins-dj
set-op-delete ias-ops = ias-delete

set-op-isEmpty ias-ops = ias-isEmpty
set-op-isSng ias-ops = ias-isSng
set-op-ball ias-ops = ias-ball
set-op-bexists ias-ops = ias-bexists
set-op-size ias-ops = ias-size
set-op-size-abort ias-ops = ias-size-abort
set-op-union ias-ops = ias-union
set-op-union-dj ias-ops = ias-union-dj
set-op-diff ias-ops = isis-diff
set-op-filter ias-ops = isis-filter
set-op-inter ias-ops = ias-inter
set-op-subset ias-ops = isis-subset
set-op-equal ias-ops = isis-equal
set-op-disjoint ias-ops = isis-disjoint
set-op-disjoint-witness ias-ops = isis-disjoint-witness
set-op-sel ias-ops = ias-sel′
set-op-to-list ias-ops = ias-to-list
set-op-from-list ias-ops = list-to-ias
⟨*proof*⟩
interpretation *iasr*!: *set-iteratei* (*set-op-α ias-ops*) (*set-op-invar ias-ops*) *ias-iteratei* ⟨*proof*⟩

end

4.29 Record-based Map Interface: Implementation setup

theory *RecordMapImpl*
imports
../gen-algo/StdInst
begin

4.29.1 Hash Maps

definition *hm-ops* = (|
map-op-α = hm-α,
map-op-invar = hm-invar,
map-op-empty = hm-empty,
map-op-sng = hm-sng,
map-op-lookup = hm-lookup,
map-op-update = hm-update,
map-op-update-dj = hm-update-dj,
map-op-delete = hm-delete,
map-op-restrict = hh-restrict,
map-op-add = hm-add,
map-op-add-dj = hm-add-dj,
map-op-isEmpty = hm-isEmpty,

map-op-isSng = hm-isSng,
map-op-ball = hm-ball,
map-op-bexists = hm-bexists,
map-op-size = hm-size,
map-op-size-abort = hm-size-abort,
map-op-sel = hm-sel′,
map-op-to-list = hm-to-list,
map-op-to-map = list-to-hm
⦈

interpretation *hmr*!: *StdMap hm-ops*
⟨*proof*⟩

lemma *hm-ops-unfold*[*code-unfold*]:
map-op-α hm-ops = hm-α
map-op-invar hm-ops = hm-invar
map-op-empty hm-ops = hm-empty
map-op-sng hm-ops = hm-sng
map-op-lookup hm-ops = hm-lookup
map-op-update hm-ops = hm-update
map-op-update-dj hm-ops = hm-update-dj
map-op-delete hm-ops = hm-delete
map-op-restrict hm-ops = hh-restrict
map-op-add hm-ops = hm-add
map-op-add-dj hm-ops = hm-add-dj
map-op-isEmpty hm-ops = hm-isEmpty
map-op-isSng hm-ops = hm-isSng
map-op-ball hm-ops = hm-ball
map-op-bexists hm-ops = hm-bexists
map-op-size hm-ops = hm-size
map-op-size-abort hm-ops = hm-size-abort
map-op-sel hm-ops = hm-sel′
map-op-to-list hm-ops = hm-to-list
map-op-to-map hm-ops = list-to-hm
⟨*proof*⟩
interpretation *hmr*!: *map-iteratei* (*map-op-α hm-ops*) (*map-op-invar hm-ops*) *hm-iteratei*
⟨*proof*⟩

4.29.2 RBT Maps

definition *rm-ops* :: (′*k*::*linorder*,′*v*,(′*k*,′*v*) *rm*) *omap-ops*
where *rm-ops* = ⦇
map-op-α = rm-α,
map-op-invar = rm-invar,
map-op-empty = rm-empty,
map-op-sng = rm-sng,
map-op-lookup = rm-lookup,
map-op-update = rm-update,

map-op-update-dj = *rm-update-dj*,
map-op-delete = *rm-delete*,
map-op-restrict = *rr-restrict*,
map-op-add = *rm-add*,
map-op-add-dj = *rm-add-dj*,
map-op-isEmpty = *rm-isEmpty*,
map-op-isSng = *rm-isSng*,
map-op-ball = *rm-ball*,
map-op-bexists = *rm-bexists*,
map-op-size = *rm-size*,
map-op-size-abort = *rm-size-abort*,
map-op-sel = *rm-sel′*,
map-op-to-list = *rm-to-list*,
map-op-to-map = *list-to-rm*,
map-op-min = *rm-min*,
map-op-max = *rm-max*
⦇

interpretation *rmr*!: *StdMap rm-ops*
⟨*proof*⟩

interpretation *rmr*!: *StdOMap rm-ops*
⟨*proof*⟩

lemma *rm-ops-unfold*[*code-unfold*]:
map-op-α rm-ops = *rm-α*
map-op-invar rm-ops = *rm-invar*
map-op-empty rm-ops = *rm-empty*
map-op-sng rm-ops = *rm-sng*
map-op-lookup rm-ops = *rm-lookup*
map-op-update rm-ops = *rm-update*
map-op-update-dj rm-ops = *rm-update-dj*
map-op-delete rm-ops = *rm-delete*
map-op-restrict rm-ops = *rr-restrict*
map-op-add rm-ops = *rm-add*
map-op-add-dj rm-ops = *rm-add-dj*
map-op-isEmpty rm-ops = *rm-isEmpty*
map-op-isSng rm-ops = *rm-isSng*
map-op-ball rm-ops = *rm-ball*
map-op-bexists rm-ops = *rm-bexists*
map-op-size rm-ops = *rm-size*
map-op-size-abort rm-ops = *rm-size-abort*
map-op-sel rm-ops = *rm-sel′*
map-op-to-list rm-ops = *rm-to-list*
map-op-to-map rm-ops = *list-to-rm*
map-op-min rm-ops = *rm-min*
map-op-max rm-ops = *rm-max*
⟨*proof*⟩

interpretation *rmr*!: *map-iteratei* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-iteratei*
⟨*proof*⟩

interpretation *rmr*!: *map-iterateoi* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-iterateoi*
⟨*proof*⟩

interpretation *rmr*!: *map-reverse-iterateoi* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-reverse-iterateoi*
⟨*proof*⟩

4.29.3 List Maps

definition *lm-ops* = (|
map-op-α = *lm-α*,
map-op-invar = *lm-invar*,
map-op-empty = *lm-empty*,
map-op-sng = *lm-sng*,
map-op-lookup = *lm-lookup*,
map-op-update = *lm-update*,
map-op-update-dj = *lm-update-dj*,
map-op-delete = *lm-delete*,
map-op-restrict = *ll-restrict*,
map-op-add = *lm-add*,
map-op-add-dj = *lm-add-dj*,
map-op-isEmpty = *lm-isEmpty*,
map-op-isSng = *lm-isSng*,
map-op-ball = *lm-ball*,
map-op-bexists = *lm-bexists*,
map-op-size = *lm-size*,
map-op-size-abort = *lm-size-abort*,
map-op-sel = *lm-sel′*,
map-op-to-list = *lm-to-list*,
map-op-to-map = *list-to-lm*
|)

interpretation *lmr*!: *StdMap lm-ops*
⟨*proof*⟩

lemma *lm-ops-unfold*[*code-unfold*]:
map-op-α lm-ops = *lm-α*
map-op-invar lm-ops = *lm-invar*
map-op-empty lm-ops = *lm-empty*
map-op-sng lm-ops = *lm-sng*
map-op-lookup lm-ops = *lm-lookup*
map-op-update lm-ops = *lm-update*
map-op-update-dj lm-ops = *lm-update-dj*
map-op-delete lm-ops = *lm-delete*

map-op-restrict lm-ops = ll-restrict
map-op-add lm-ops = lm-add
map-op-add-dj lm-ops = lm-add-dj
map-op-isEmpty lm-ops = lm-isEmpty
map-op-isSng lm-ops = lm-isSng
map-op-ball lm-ops = lm-ball
map-op-bexists lm-ops = lm-bexists
map-op-size lm-ops = lm-size
map-op-size-abort lm-ops = lm-size-abort
map-op-sel lm-ops = lm-sel′
map-op-to-list lm-ops = lm-to-list
map-op-to-map lm-ops = list-to-lm
⟨*proof*⟩
interpretation *lmr*!: *map-iteratei* (*map-op-α lm-ops*) (*map-op-invar lm-ops*) *lm-iteratei*
⟨*proof*⟩

4.29.4 Array Hash Maps

definition *ahm-ops* = (‖
map-op-α = ahm-α,
map-op-invar = ahm-invar,
map-op-empty = ahm-empty,
map-op-sng = ahm-sng,
map-op-lookup = ahm-lookup,
map-op-update = ahm-update,
map-op-update-dj = ahm-update-dj,
map-op-delete = ahm-delete,
map-op-restrict = aa-restrict,
map-op-add = ahm-add,
map-op-add-dj = ahm-add-dj,
map-op-isEmpty = ahm-isEmpty,
map-op-isSng = ahm-isSng,
map-op-ball = ahm-ball,
map-op-bexists = ahm-bexists,
map-op-size = ahm-size,
map-op-size-abort = ahm-size-abort,
map-op-sel = ahm-sel′,
map-op-to-list = ahm-to-list,
map-op-to-map = list-to-ahm
‖)

interpretation *ahmr*!: *StdMap ahm-ops*
⟨*proof*⟩

lemma *ahm-ops-unfold*[*code-unfold*]:
map-op-α ahm-ops = ahm-α
map-op-invar ahm-ops = ahm-invar
map-op-empty ahm-ops = ahm-empty

map-op-sng ahm-ops = ahm-sng
map-op-lookup ahm-ops = ahm-lookup
map-op-update ahm-ops = ahm-update
map-op-update-dj ahm-ops = ahm-update-dj
map-op-delete ahm-ops = ahm-delete
map-op-restrict ahm-ops = aa-restrict
map-op-add ahm-ops = ahm-add
map-op-add-dj ahm-ops = ahm-add-dj
map-op-isEmpty ahm-ops = ahm-isEmpty
map-op-isSng ahm-ops = ahm-isSng
map-op-ball ahm-ops = ahm-ball
map-op-bexists ahm-ops = ahm-bexists
map-op-size ahm-ops = ahm-size
map-op-size-abort ahm-ops = ahm-size-abort
map-op-sel ahm-ops = ahm-sel′
map-op-to-list ahm-ops = ahm-to-list
map-op-to-map ahm-ops = list-to-ahm
⟨*proof*⟩

interpretation *ahmr*!: *map-iteratei* (*map-op-α ahm-ops*) (*map-op-invar ahm-ops*) *ahm-iteratei*
⟨*proof*⟩

4.29.5 Indexed Array Maps

definition *iam-ops* = (|
map-op-α = iam-α,
map-op-invar = iam-invar,
map-op-empty = iam-empty,
map-op-sng = iam-sng,
map-op-lookup = iam-lookup,
map-op-update = iam-update,
map-op-update-dj = iam-update-dj,
map-op-delete = iam-delete,
map-op-restrict = imim-restrict,
map-op-add = iam-add,
map-op-add-dj = iam-add-dj,
map-op-isEmpty = iam-isEmpty,
map-op-isSng = iam-isSng,
map-op-ball = iam-ball,
map-op-bexists = iam-bexists,
map-op-size = iam-size,
map-op-size-abort = iam-size-abort,
map-op-sel = iam-sel′,
map-op-to-list = iam-to-list,
map-op-to-map = list-to-iam
|)

interpretation *iamr*!: *StdMap iam-ops*
⟨*proof*⟩

lemma *iam-ops-unfold*[*code-unfold*]:
map-op-α iam-ops = *iam-α*
map-op-invar iam-ops = *iam-invar*
map-op-empty iam-ops = *iam-empty*
map-op-sng iam-ops = *iam-sng*
map-op-lookup iam-ops = *iam-lookup*
map-op-update iam-ops = *iam-update*
map-op-update-dj iam-ops = *iam-update-dj*
map-op-delete iam-ops = *iam-delete*
map-op-restrict iam-ops = *imim-restrict*
map-op-add iam-ops = *iam-add*
map-op-add-dj iam-ops = *iam-add-dj*
map-op-isEmpty iam-ops = *iam-isEmpty*
map-op-isSng iam-ops = *iam-isSng*
map-op-ball iam-ops = *iam-ball*
map-op-bexists iam-ops = *iam-bexists*
map-op-size iam-ops = *iam-size*
map-op-size-abort iam-ops = *iam-size-abort*
map-op-sel iam-ops = *iam-sel'*
map-op-to-list iam-ops = *iam-to-list*
map-op-to-map iam-ops = *list-to-iam*
⟨*proof*⟩
interpretation *iamr*!: *map-iteratei* (*map-op-α iam-ops*) (*map-op-invar iam-ops*) *iam-iteratei*
⟨*proof*⟩

end

4.30 Deprecated: Data Refinement for the While-Combinator

theory *DatRef*
imports
Main
common/Misc
~~/src/HOL/Library/While-Combinator
begin

Note that this theory is deprecated. For new developments, the refinement framework (Refine-Monadic entry of the AFP) should be used.

In this theory, a data refinement framework for non-deterministic while-loops is developed. The refinement is based on showing simulation w.r.t.

an abstraction function. The case of deterministic while-loops is explicitely handled, to support proper code-generation using the While-Combinator.

Note that this theory is deprecated. For new developments, the refinement framework (Refine-Monadic entry of the AFP) should be used.

A nondeterministic while-algorithm is described by a set of states, a continuation condition, a step relation, a set of possible initial states and an invariant.

— Encapsulates a while-algorithm and its invariant
record *'S while-algo* =
— Termination condition
wa-cond :: *'S set*
— Step relation (nondeterministic)
wa-step :: (*'S* × *'S*) *set*
— Initial state (nondeterministic)
wa-initial :: *'S set*
— Invariant
wa-invar :: *'S set*

A while-algorithm is called *well-defined* iff the invariant holds for all reachable states and the accessible part of the step-relation is well-founded.

— Conditions that must hold for a well-defined while-algorithm
locale *while-algo* =
fixes *WA* :: *'S while-algo*

— A step must preserve the invariant
assumes *step-invar*:
⟦ *s*∈*wa-invar WA*; *s*∈*wa-cond WA*; (*s*,*s'*)∈*wa-step WA* ⟧ ⟹ *s'*∈*wa-invar WA*
— Initial states must satisfy the invariant
assumes *initial-invar*: *wa-initial WA* ⊆ *wa-invar WA*
— The accessible part of the step relation must be well-founded
assumes *step-wf*:
wf { (*s'*,*s*). *s*∈*wa-invar WA* ∧ *s*∈*wa-cond WA* ∧ (*s*,*s'*)∈*wa-step WA* }

Next, a refinement relation for while-algorithms is defined. Note that the involved while-algorithms are not required to be well-defined. Later, some lemmas to transfer well-definedness along refinement relations are shown.

Refinement involves a concrete algorithm, an abstract algorithm and an abstraction function. In essence, a refinement establishes a simulation of the concrete algorithm by the abstract algorithm w.r.t. the abstraction function.

locale *wa-refine* =
— Concrete algorithm
fixes *WAC* :: *'C while-algo*
— Abstract algorithm
fixes *WAA* :: *'A while-algo*

— Abstraction function
fixes $\alpha :: 'C \Rightarrow 'A$

— Condition implemented correctly: The concrete condition must be stronger than the abstract one. Intuitively, this ensures that the concrete loop will not run longer than the abstract one that it is simulated by.
assumes *cond-abs*: $[\![s{\in}wa\text{-}invar\ WAC;\ s{\in}wa\text{-}cond\ WAC\ ]\!] \Longrightarrow \alpha\ s \in wa\text{-}cond\ WAA$

— Step implemented correctly: The abstract step relation must simulate the concrete step relation
assumes *step-abs*: $[\![s{\in}wa\text{-}invar\ WAC;\ s{\in}wa\text{-}cond\ WAC;\ (s,s'){\in}wa\text{-}step\ WAC\ ]\!]$
$\Longrightarrow (\alpha\ s, \alpha\ s'){\in}wa\text{-}step\ WAA$
— Initial states implemented correctly: The abstractions of the concrete initial states must be abstract initial states.
assumes *initial-abs*: $\alpha\ `\ wa\text{-}initial\ WAC \subseteq wa\text{-}initial\ WAA$
— Invariant implemented correctly: The concrete invariant must be stronger then the abstract invariant. Note that, usually, the concrete invariant will be of the form $I\text{-}add \cap \{s.\ \alpha\ s \in wa\text{-}invar\ WAA\}$, where *I-add* are the additional invariants added by the concrete algorithm.
assumes *invar-abs*: $\alpha\ `\ wa\text{-}invar\ WAC \subseteq wa\text{-}invar\ WAA$
begin

lemma *initial-abs'*: $s{\in}wa\text{-}initial\ WAC \Longrightarrow \alpha\ s \in wa\text{-}initial\ WAA$
⟨*proof*⟩

lemma *invar-abs'*: $s{\in}wa\text{-}invar\ WAC \Longrightarrow \alpha\ s \in wa\text{-}invar\ WAA$
⟨*proof*⟩

end

— Given a concrete while-algorithm and a well-defined abstract while-algorithm, this lemma shows refinement and well-definedness of the concrete while-algorithm.
Assuming well-definedness of the abstract algorithm and refinement, some proof-obligations for well-definedness of the concrete algorithm can be discharged automatically.
For this purpose, the invariant is split into a concrete and an abstract part. The abstract part claims that the abstraction of a state satisfies the abstract invariant. The concrete part makes some additional claims about a valid concrete state. Then, after having shown refinement, the assumptions that the abstract part of the invariant is preserved, can be discharged automatically.
lemma *wa-refine-intro*:
fixes *condc* :: $'C\ set$ **and**
stepc :: $('C{\times}'C)\ set$ **and**
initialc :: $'C\ set$ **and**
invar-addc :: $'C\ set$
fixes *WAA* :: $'A\ while\text{-}algo$
fixes $\alpha :: 'C \Rightarrow 'A$

assumes *while-algo WAA*

— The concrete step preserves the concrete part of the invariant
assumes *step-invarc*:
!!$s\ s'.\ [\![\ s{\in}invar\text{-}addc;\ s{\in}condc;\ \alpha\ s\ \in\ wa\text{-}invar\ WAA;\ (s,s'){\in}stepc\ ]\!]$
$\Longrightarrow s'{\in}invar\text{-}addc$
— The concrete initial states satisfy the concrete part of the invariant
assumes *initial-invarc*: $initialc \subseteq invar\text{-}addc$

— Condition implemented correctly
assumes *cond-abs*:
!!$s.\ [\![\ s{\in}invar\text{-}addc;\ \alpha\ s\ \in\ wa\text{-}invar\ WAA;\ s{\in}condc\ ]\!] \Longrightarrow \alpha\ s\ \in\ wa\text{-}cond\ WAA$
— Step implemented correctly
assumes *step-abs*:
!!$s\ s'.\ [\![\ s{\in}invar\text{-}addc;\ s{\in}condc;\ \alpha\ s\ \in\ wa\text{-}invar\ WAA;\ (s,s'){\in}stepc\ ]\!]$
$\Longrightarrow (\alpha\ s,\ \alpha\ s'){\in}wa\text{-}step\ WAA$
— Initial states implemented correctly
assumes *initial-abs*: $\alpha\ `\ initialc \subseteq wa\text{-}initial\ WAA$

— Concrete while-algorithm: The invariant is separated into a concrete and an abstract part
defines $WAC == (\!|$
wa-cond=*condc*,
wa-step=*stepc*,
wa-initial=*initialc*,
wa-invar=$(invar\text{-}addc \cap \{s.\ \alpha\ s{\in}\ wa\text{-}invar\ WAA\})$ $|\!)$

shows
while-algo WAC $\wedge$
wa-refine WAC WAA α (**is** *?T1* $\wedge$ *?T2*)
⟨*proof*⟩
lemma (**in** *wa-refine*) *wa-intro*:
— Concrete part of the invariant
fixes *addi* :: *'C set*
— The abstract algorithm is well-defined
assumes *while-algo WAA*
— The invariant can be split into concrete and abstract part
assumes *icf*: *wa-invar WAC* = $addi \cap \{s.\ \alpha\ s \in wa\text{-}invar\ WAA\}$

— The step-relation preserves the concrete part of the invariant
assumes *step-addi*:
!!$s\ s'.\ [\![\ s{\in}addi;\ s{\in}wa\text{-}cond\ WAC;\ \alpha\ s\ \in\ wa\text{-}invar\ WAA;$
$(s,s'){\in}wa\text{-}step\ WAC$
$]\!] \Longrightarrow s'{\in}addi$

— The initial states satisfy the concrete part of the invariant
assumes *initial-addi*: *wa-initial WAC* $\subseteq$ *addi*

shows

while-algo WAC
⟨*proof*⟩

A special case of refinement occurs, if the concrete condition implements the abstract condition precisely. In this case, the concrete algorithm will run as long as the abstract one that it is simulated by. This allows to transfer properties of the result from the abstract algorithm to the concrete one.

— Precise refinement
locale *wa-precise-refine* = *wa-refine* +
constrains $\alpha :: 'C \Rightarrow 'A$
assumes *cond-precise*:
$\forall s.\ s \in$*wa-invar WAC* $\wedge\ \alpha\ s \in$*wa-cond WAA* $\longrightarrow s \in$*wa-cond WAC*
begin
— Transfer correctness property
lemma *transfer-correctness*:
assumes *A*: $\forall s.\ s \in$*wa-invar WAA* $\wedge\ s \notin$*wa-cond WAA* $\longrightarrow$ *P s*
shows $\forall sc.\ sc \in$*wa-invar WAC* $\wedge\ sc \notin$*wa-cond WAC* $\longrightarrow$ *P* (α *sc*)
⟨*proof*⟩
end

Refinement as well as precise refinement is reflexive and transitive

lemma *wa-ref-refl*: *wa-refine WA WA id*
⟨*proof*⟩

lemma *wa-pref-refl*: *wa-precise-refine WA WA id*
⟨*proof*⟩

lemma *wa-ref-trans*:
assumes *wa-refine WC WB* $\alpha 1$
assumes *wa-refine WB WA* $\alpha 2$
shows *wa-refine WC WA* ($\alpha 2 \circ \alpha 1$)
⟨*proof*⟩

lemma *wa-pref-trans*:
assumes *wa-precise-refine WC WB* $\alpha 1$
assumes *wa-precise-refine WB WA* $\alpha 2$
shows *wa-precise-refine WC WA* ($\alpha 2 \circ \alpha 1$)
⟨*proof*⟩

A well-defined while-algorithm is *deterministic*, iff the step relation is a function and there is just one initial state. Such an algorithm is suitable for direct implementation by the while-combinator.

For deterministic while-algorithm, an own record is defined, as well as a function that maps it to the corresponding record for non-deterministic while algorithms. This makes sense as the step-relation may then be modeled as a function, and the initial state may be modeled as a single state rather than a (singleton) set of states.

record $'S$ *det-while-algo* =

— Termination condition
dwa-cond :: *'S* ⇒ *bool*
— Step function
dwa-step :: *'S* ⇒ *'S*
— Initial state
dwa-initial :: *'S*
— Invariant
dwa-invar :: *'S set*

— Maps the record for deterministic while-algo to the corresponding record for the non-deterministic one
definition *det-wa-wa DWA* == (|
wa-cond={*s*. *dwa-cond DWA s*},
wa-step={(*s*,*dwa-step DWA s*) | *s*. *True*},
wa-initial={*dwa-initial DWA*},
wa-invar = *dwa-invar DWA*|)

— Conditions for a deterministic while-algorithm
locale *det-while-algo* =
fixes *WA* :: *'S det-while-algo*
— The step preserves the invariant
assumes *step-invar*:
⟦ *s*∈*dwa-invar WA*; *dwa-cond WA s* ⟧ ⟹ *dwa-step WA s* ∈ *dwa-invar WA*
— The initial state satisfies the invariant
assumes *initial-invar*: *dwa-initial WA* ∈ *dwa-invar WA*
— The relation made up by the step-function is well-founded.
assumes *step-wf*:
wf { (*dwa-step WA s*,*s*) | *s*. *s*∈*dwa-invar WA* ∧ *dwa-cond WA s* }

begin
lemma *is-while-algo*: *while-algo* (*det-wa-wa WA*)
⟨*proof*⟩

end

lemma *det-while-algo-intro*:
assumes *while-algo* (*det-wa-wa DWA*)
shows *det-while-algo DWA*
⟨*proof*⟩
theorem *dwa-is-wa*:
while-algo (*det-wa-wa DWA*) ⟷ *det-while-algo DWA*
⟨*proof*⟩

definition (**in** *det-while-algo*)
loop == (*while* (*dwa-cond WA*) (*dwa-step WA*) (*dwa-initial WA*))

— Proof rule for deterministic while loops
lemma (**in** *det-while-algo*) *while-proof*:

assumes *inv-imp*: $\bigwedge s.\ [\![s \in dwa\text{-}invar\ WA;\ \neg\ dwa\text{-}cond\ WA\ s]\!] \Longrightarrow Q\ s$
shows *Q loop*
⟨*proof*⟩
lemma (**in** *det-while-algo*) *while-proof'*:
assumes *inv-imp*:
$\forall s.\ s \in wa\text{-}invar\ (det\text{-}wa\text{-}wa\ WA) \wedge s \notin wa\text{-}cond\ (det\text{-}wa\text{-}wa\ WA) \longrightarrow Q\ s$
shows *Q loop*
⟨*proof*⟩

lemma (**in** *det-while-algo*) *loop-invar*:
loop ∈ *dwa-invar WA*
⟨*proof*⟩

end

4.31 Standard Collections

theory *Collections*
imports
common/Misc

spec/SetSpec
spec/MapSpec
spec/ListSpec
spec/AnnotatedListSpec
spec/PrioSpec
spec/PrioUniqueSpec

impl/SetStdImpl
impl/MapStdImpl
gen-algo/StdInst
impl/RecordSetImpl
impl/RecordMapImpl
impl/Fifo
impl/BinoPrioImpl
impl/SkewPrioImpl
impl/FTAnnotatedListImpl
impl/FTPrioImpl
impl/FTPrioUniqueImpl

DatRef

begin

This theory summarizes the components of the Isabelle Collection Framework.

end

Chapter 5

Isabelle Collections Framework Userguide

theory *Userguide*
imports
Collections
~~/src/HOL/Library/Code-Target-Numeral
begin

5.1 Introduction

The Isabelle Collections Framework defines interfaces of various collection types and provides some standard implementations and generic algorithms.

The relation between the data structures of the collection framework and standard Isabelle types (e.g. for sets and maps) is established by abstraction functions.

Amongst others, the following interfaces and data-structures are provided by the Isabelle Collections Framework (For a complete list, see the overview section in the implementations chapter of the proof document):

- Set and map implementations based on (associative) lists, red-black trees, hashing and tries.
- An implementation of a FIFO-queue based on two stacks.
- Annotated lists implemented by finger trees.
- Priority queues implemented by binomial heaps, skew binomial heaps, and annotated lists (via finger trees).

The red-black trees are imported from the standard isabelle library. The binomial and skew binomial heaps are imported from the *Binomial-Heaps*

entry of the archive of formal proofs. The finger trees are imported from the *Finger-Trees* entry of the archive of formal proofs.

5.1.1 Getting Started

To get started with the Isabelle Collections Framework (assuming that you are already familiar with Isabelle/HOL and Isar), you should first read the introduction (this section), that provides many basic examples. Section 5.2 explains the concepts of the Isabelle Collections Framework in more detail. Section 5.3 provides information on extending the framework along with detailed examples, and Section 5.4 contains a discussion on the design of this framework. There is also a paper [2] on the design of the Isabelle Collections Framework available.

5.1.2 Introductory Example

We introduce the Isabelle Collections Framework by a simple example.

Given a set of elements represented by a red-black tree, and a list, we want to filter out all elements that are not contained in the set. This can be done by Isabelle/HOL's *filter*-function[1]:

definition *rbt-restrict-list* :: *'a::linorder rs* ⇒ *'a list* ⇒ *'a list*
where *rbt-restrict-list s l* == [*x←l. rs-memb x s*]

The type *'a rs* is the type of sets backed by red-black trees. Note that the element type of sets backed by red-black trees must be of sort *linorder*. The function *rs-memb* tests membership on such sets.

Next, we show correctness of our function:

lemma *rbt-restrict-list-correct*:
  assumes [*simp*]: *rs-invar s*
  shows *rbt-restrict-list s l* = [*x←l. x∈rs-α s*]
  ⟨*proof*⟩

The lemma *rs.memb-correct*:

True ⟹ *rs-memb x s* = (*x* ∈ *rs-α s*)

states correctness of the *rs-memb*-function. The function *rs-α* maps a red-black-tree to the set that it represents. Moreover, we have to explicitely keep track of the invariants of the used data structure, in this case red-black trees. The premise *True* represents the invariant assumption for the collection data structure. Red-black-trees are invariant-free, so this defaults

[1] Note that Isabelle/HOL uses the list comprehension syntax [*x←l. P x*] as syntactic sugar for filtering a list.

to *True*. For uniformity reasons, these (unnecessary) invariant assumptions are present in all correctness lemmata.

Many of the correctness lemmas for standard RBT-set-operations are summarized by the lemma *rs-correct*:

⟦*True*; *inj-on f* (*rs-α s* ∩ *dom f*)⟧
⟹ *rs-α* (*rs-inj-image-filter f s*) = {*b*. ∃ *a*∈*rs-α s*. *f a* = *Some b*}
⟦*True*; *inj-on f* (*rs-α s* ∩ *dom f*)⟧ ⟹ *True*
True ⟹
rs-α (*rs-image-filter* (λ*x*. *if P x then Some* (*f x*) *else None*) *s*) =
f ‘ {*x* ∈ *rs-α s*. *P x*}
True ⟹ *rs-α* (*rs-image-filter f s*) = {*b*. ∃ *a*∈*rs-α s*. *f a* = *Some b*}
True ⟹ *True*
⟦*True*; *inj-on f* (*rs-α s*)⟧ ⟹ *rs-α* (*rs-inj-image f s*) = *f* ‘ *rs-α s*
⟦*True*; *inj-on f* (*rs-α s*)⟧ ⟹ *True*
⟦*True*; *True*; *rs-α s1* ∩ *rs-α s2* = {}⟧
⟹ *rs-α* (*rs-union-dj s1 s2*) = *rs-α s1* ∪ *rs-α s2*
⟦*True*; *True*; *rs-α s1* ∩ *rs-α s2* = {}⟧ ⟹ *True*
⟦*True*; *True*⟧ ⟹ *rs-α* (*rs-union s1 s2*) = *rs-α s1* ∪ *rs-α s2*
⟦*True*; *True*⟧ ⟹ *True*
⟦*True*; *True*⟧ ⟹ *rs-α* (*rs-inter s1 s2*) = *rs-α s1* ∩ *rs-α s2*
⟦*True*; *True*⟧ ⟹ *True*
True ⟹ *rs-α* (*rs-image f s*) = *f* ‘ *rs-α s*
True ⟹ *True*
⟦*True*; *x* ∉ *rs-α s*⟧ ⟹ *rs-α* (*rs-ins-dj x s*) = *insert x* (*rs-α s*)
⟦*True*; *x* ∉ *rs-α s*⟧ ⟹ *True*
True ⟹ *rs-α* (*rs-delete x s*) = *rs-α s* − {*x*}
True ⟹ *True*
True ⟹ *rs-α* (*rs-ins x s*) = *insert x* (*rs-α s*)
True ⟹ *True*
True ⟹ *rs-memb x s* = (*x* ∈ *rs-α s*)
True ⟹ *set* (*rs-to-list s*) = *rs-α s*
True ⟹ *distinct* (*rs-to-list s*)
rs-α (*list-to-rs l*) = *set l*
True
True ⟹ *rs-isEmpty s* = (*rs-α s* = {})
rs-α (*rs-empty* ()) = {}
True

All implementations provided by this library are compatible with the Isabelle/HOL code-generator. Now follow some examples of using the code-generator and the related value-command:

There are conversion functions from lists to sets and, vice-versa, from sets to lists:

value *list-to-rs* [*1*::*int*..*10*]
value *rs-to-list* (*list-to-rs* [*1*::*int* .. *10*])
value *rs-to-list* (*list-to-rs* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*])

Note that sets make no guarantee about ordering, hence the only thing we can prove about conversion from sets to lists is: *rs.to-list-correct*:

True $\Longrightarrow$ *set* (*rs-to-list s*) = *rs-α s*
True $\Longrightarrow$ *distinct* (*rs-to-list s*)

value *rbt-restrict-list* (*list-to-rs* [*1*::*nat*,*2*,*3*,*4*,*5*]) [*1*::*nat*,*9*,*2*,*3*,*4*,*5*,*6*,*5*,*4*,*3*,*6*,*7*,*8*,*9*]

definition *test n* = *list-to-rs* [*1*..*int-of-integer n*]

⟨*ML*⟩

5.1.3 Theories

To make available the whole collections framework to your formalization, import the theory *Collections*.
Other theories in the Isabelle Colelction Framework include:

SetSpec Specification of sets and set functions

SetGA Generic algorithms for sets

SetStdImpl Standard set implementations (list, rb-tree, hashing, tries)

MapSpec Specification of maps

MapGA Generic algorithms for maps

MapStdImpl Standard map implementations (list,rb-tree, hashing, tries)

Algos Various generic algorithms

SetIndex Generic algorithm for building indices of sets

ListSpec Specification of lists

Fifo Amortized fifo queue

DatRef Data refinement for the while combinator

5.1.4 Iterators

An important concept when using collections are iterators. An iterator is a kind of generalized fold-functional. Like the fold-functional, it applies a function to all elements of a set and modifies a state. There are no guarantees about the iteration order. But, unlike the fold functional, you can prove useful properties of iterations even if the function is not left-commutative.

Proofs about iterations are done in invariant style, establishing an invariant over the iteration.

The iterator combinator for red-black tree sets is *rs-iteratei*, and the proof-rule that is usually used is: *rs.iterate-rule-P*:

$$\begin{array}{l}[\![\mathit{True};\ I\ (\mathit{rs}\text{-}\alpha\ S)\ \sigma 0;\\ \bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq \mathit{rs}\text{-}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);\\ \bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma]\!]\\ \Longrightarrow P\ (\mathit{rs\text{-}iteratei}\ S\ (\lambda\text{-}.\ \mathit{True})\ f\ \sigma 0)\end{array}$$

The invariant I is parameterized with the set of remaining elements that have not yet been iterated over and the current state. The invariant has to hold for all elements remaining and the initial state: $I\ (\mathit{rs}\text{-}\alpha\ S)\ \sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

$$\bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq \mathit{rs}\text{-}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$$

And the proposition to be shown for the final state must be a consequence of the invarant for no elements remaining: $\bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$.

A generalization of iterators are *interruptible iterators* where iteration is only continues while some condition on the state holds. Reasoning over interruptible iterators is also done by invariants: *rs.iteratei-rule-P*:

$$\begin{array}{l}[\![\mathit{True};\ I\ (\mathit{rs}\text{-}\alpha\ S)\ \sigma 0;\\ \bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq \mathit{rs}\text{-}\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);\\ \bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge \sigma\ it.\ [\![it \subseteq \mathit{rs}\text{-}\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!]\\ \Longrightarrow P\ (\mathit{rs\text{-}iteratei}\ S\ c\ f\ \sigma 0)\end{array}$$

Here, interruption of the iteration is handled by the premise

$$\bigwedge \sigma\ it.\ [\![it \subseteq \mathit{rs}\text{-}\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma$$

that shows the proposition from the invariant for any intermediate state of the iteration where the continuation condition does not hold (and thus the iteration is interrupted).

As an example of reasoning about results of iterators, we implement a function that converts a hashset to a list that contains precisely the elements of the set.

definition *hs-to-list′ s* == *hs-iteratei s* (λ-. *True*) (*op* #) []

The correctness proof works by establishing the invariant that the list contains all elements that have already been iterated over. Again *True* denotes the invariant for hashsets which defaults to *True*.

lemma *hs-to-list′-correct*:
assumes *INV*: *hs-invar s*
shows *set* (*hs-to-list′ s*) = *hs-α s*

⟨*proof*⟩

As an example for an interruptible iterator, we define a bounded existential-quantification over the list elements. As soon as the first element is found that fulfills the predicate, the iteration is interrupted. The state of the iteration is simply a boolean, indicating the (current) result of the quantification:

definition *hs-bex s P* == *hs-iteratei s* (λσ. ¬ σ) (λx σ. *P x*) *False*

lemma *hs-bex-correct*:
hs-invar s ⟹ *hs-bex s P* ⟷ (∃ *x*∈*hs-α s*. *P x*)
⟨*proof*⟩

5.2 Structure of the Framework

The concepts of the framework are roughly based on the object-oriented concepts of interfaces, implementations and generic algorithms.
The concepts used in the framework are the following:

Interfaces An interface describes some concept by providing an abstraction mapping α to a related Isabelle/HOL-concept. The definition is generic in the datatype used to implement the concept (i.e. the concrete data structure). An interface is specified by means of a locale that fixes the abstraction mapping and an invariant. For example, the set-interface contains an abstraction mapping to sets, and is specified by the locale *SetSpec.set*. An interface roughly matches the concept of a (collection) interface in Java, e.g. *java.util.Set*.

Functions A function specifies some functionality involving interfaces. A function is specified by means of a locale. For example, membership query for a set is specified by the locale *SetSpec.set-memb* and equality test between two sets is a function specified by *SetSpec.set-equal*. A function roughly matches a method declared in an interface, e.g. *java.util.Set#contains, java.util.Set#equals*.

Generic Algorithms A generic algorithm specifies, in a generic way, how to implement a function using other functions. For example, the equality test for sets may be implemented using a subset function. It is described by the constant *SetGA.subset-equal* and the corresponding lemma *SetGA.subset-equal-correct*. There is no direct match of generic algorithms in the Java Collections Framework. The most related concept are abstract collection interfaces, that provide some default algorithms, e.g. *java.util.AbstractSet*. The concept of *Algorithm* in the C++ Standard Template Library [3] matches the concept of Generic Algorithm quite well.

Implementation An implementation of an interface provides a data structure for that interface together with an abstraction mapping and an invariant. Moreover, it provides implementations for some (or all) functions of that interface. For example, red-black trees are an implementation of the set-interface, with the abstraction mapping *rs-α* and invariant *rs-invar*; and the constant *rs-ins* implements the insert-function, as stated by the lemma *rs-ins-impl*. An implementation matches a concrete collection interface in Java, e.g. *java.util.TreeSet*, and the methods implemented by such an interface, e.g. *java.util.TreeSet#add*.

Instantiation An instantiation of a generic algorithm provides actual implementations for the used functions. For example, the generic equality-test algorithm can be instantiated to use red-black-trees for both arguments (resulting in the function *rr-equal* and the lemma *rr-equal-impl*). While some of the functions of an implementation need to be implemented specifically, many functions may be obtained by instantiating generic algorithms. In Java, instantiation of a generic algorithm is matched most closely by inheriting from an abstract collection interface. In the C++ Standard Template Library instantiation of generic algorithms is done implicitely when using them.

5.2.1 Naming Conventions

The Isabelle Collections Framework follows these general naming conventions. Each implementation has a two-letter (or three-letter) and a one-letter (or two-letter) abbreviation, that are used as prefixes for the related constants, lemmas and instantiations.

The two-letter and three-letter abbreviations should be unique over all interfaces and instantiations, the one-letter abbreviations should be unique over all implementations of the same interface. Names that reference the implementation of only one interface are prefixed with that implementation's two-letter abbreviation (e.g. *hs-ins* for insertion into a HashSet (hs,h)), names that reference more than one implementation are prefixed with the one-letter (or two-letter) abbreviations (e.g. *lhh-union* for set union between a ListSet(ls,l) and a HashSet(hs,h), yielding a HashSet)

The most important abbreviations are:

lm,l List Map

lmi,li List Map with explicit invariant

rm,r RB-Tree Map

hm,h Hash Map

ahm,a Array-based hash map

tm,t Trie Map

ls,l List Set

lsi,li List Set with explicit invariant

rs,r RB-Tree Set

hs,h Hash Set

ahs,a Array-based hash map

ts,t Trie Set

Each function *name* of an interface *interface* is declared in a locale *interface-name*. This locale provides a fact *name-correct*. For example, there is the locale *set-ins* providing the fact *set-ins.ins-correct*. An implementation instantiates the locales of all implemented functions, using its two-letter abbreviation as instantiation prefix. For example, the HashSet-implementation instantiates the locale *set-ins* with the prefix *hs*, yielding the lemma *hs.ins-correct*. Moreover, an implementation with two-letter abbreviation *aa* provides a lemma *aa-correct* that summarizes the correctness facts for the basic operations. It should only contain those facts that are safe to be used with the simplifier. E.g., the correctness facts for basic operations on hash sets are available via the lemma *hs-correct*.

5.3 Extending the Framework

This section illustrates, by example, how to add new interfaces, functions, generic algorithms and implementations to the framework:

5.3.1 Interfaces

An interface provides a new concept, that is usually mapped to a related Isabelle/HOL-concept. An interface is defined by providing a locale that fixes an abstraction mapping and an invariant. For example, consider the definition of an interface for sets:

locale *set′* =
  — Abstraction mapping to Isabelle/HOL sets
  fixes α :: *′s* ⇒ *′a set*
  — Invariant
  fixes *invar* :: *′s* ⇒ *bool*

The invariant makes it possible for an implementation to restrict to certain subsets of the type's universal set. Usually, it is convenient to hide

this invariant in a typedef and to set up the code generator approriately. However, in some cases such invariants may enable more efficient implementations (e.g. disoint insert for distinct lists), so all specifications should be with respect to a implementation-provided invariant. Most implementations will just set this invariant to λ-. *True*.

5.3.2 Functions

A function describes some operation on instances of an interface. It is specified by providing a locale that includes the locale of the interface, fixes a parameter for the operation and makes a correctness assumption. For an interface *interface* and an operation *name*, the function's locale has the name *interface-name*, the fixed parameter has the name *name* and the correctness assumption has the name *name-correct*.

As an example, consider the specifications of the insert function for sets and the empty set:

locale *set′-ins* = *set′* +
— Give reasonable names to types:
constrains $\alpha :: 's \Rightarrow 'a\ set$
— Parameter for function:
fixes *ins* :: $'a \Rightarrow 's \Rightarrow 's$
— Correctness assumption. A correctness assumption usually consists of two parts:

- A description of the operation on the abstract level, assuming that the operands satisfy the invariants.
- The invariant preservation assumptions, i.e. assuming that the result satisfies its invariants if the operands do.

assumes *ins-correct*:
$invar\ s \Longrightarrow \alpha\ (ins\ x\ s) = insert\ x\ (\alpha\ s)$
$invar\ s \Longrightarrow invar\ (ins\ x\ s)$

locale *set′-empty* = *set′* +
constrains $\alpha :: 's \Rightarrow 'a\ set$
fixes *empty* :: $'s$
assumes *empty-correct*:
$\alpha\ empty = \{\}$
invar empty

In general, more than one interface or more than one instance of the same interface may be involved in a function. Consider, for example, the intersection of two sets. It involves three instances of set interfaces, two for the operands and one for the result:

locale *set′-inter* = *set′* $\alpha 1$ *invar1* + *set′* $\alpha 2$ *invar2* + *set′* $\alpha 3$ *invar3*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*

```
  and α2 :: 's2 ⇒ 'a set and invar2
  and α3 :: 's3 ⇒ 'a set and invar3
  +
  fixes inter :: 's1 ⇒ 's2 ⇒ 's3
  assumes inter-correct:
    ⟦invar1 s1; invar2 s2⟧ ⟹ α3 (inter s1 s2) = α1 s1 ∩ α2 s2
    ⟦invar1 s1; invar2 s2⟧ ⟹ invar3 (inter s1 s2)
```

For use in further examples, we also specify a function that converts a list to a set

```
locale set'-list-to-set = set' +
  constrains α :: 's ⇒ 'a set
  fixes list-to-set :: 'a list ⇒ 's
  assumes list-to-set-correct:
    α (list-to-set l) = set l
    invar (list-to-set l)
```

5.3.3 Generic Algorithm

A generic algorithm describes how to implement a function using implementations of other functions. Thereby, it is parametric in the actual implementations of the functions.

A generic algorithm comes with the definition of a function and a correctness lemma. The function takes the required functions as arguments. The convention for argument order is that the required functions come first, then the implemented function's arguments.

Consider, for example, the generic algorithm to convert a list to a set[2]. This function requires implementations of the *empty* and *ins* functions[3]:

```
fun list-to-set' :: 's ⇒ ('a ⇒ 's ⇒ 's)
  ⇒ 'a list ⇒ 's where
  list-to-set' empt ins' [] = empt |
  list-to-set' empt ins' (a#ls) = ins' a (list-to-set' empt ins' ls)

lemma list-to-set'-correct:
  fixes empty ins
  — Assumptions about the required function implementations:
  assumes set'-empty α invar empty
  assumes set'-ins α invar ins
  — Provided function:
  shows set'-list-to-set α invar (list-to-set' empty ins)
⟨proof⟩
```

[2] To keep the presentation simple, we use a non-tail-recursive version here

[3] Due to name-clashes with *Map.empty* we have to use slightly different parameter names here

Generic Algorithms with ad-hoc function specification The collection framework also contains a few generic algorithms that do not implement a function that is specified via a locale, but the function is specified ad-hoc within the correctness lemma. An example is the generic algorithm *Algos.map-to-nat* that computes an injective map from the elements of a given finite set to an initial segment of the natural numbers. There is no locale specifying such a function, but the function is implicitly specified by the correctness lemma *map-to-nat-correct*:

⟦*set-iteratei α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *dom* (*α2* (*map-to-nat iterate1 empty2 update2 s*)) = *α1 s*
⟦*set-iteratei α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *inj-on* (*α2* (*map-to-nat iterate1 empty2 update2 s*)) (*α1 s*)
⟦*set-iteratei α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *inatseg* (*ran* (*α2* (*map-to-nat iterate1 empty2 update2 s*)))
⟦*set-iteratei α1 invar1 iterate1*; *map-empty α2 invar2 empty2*;
map-update α2 invar2 update2; *invar1 s*⟧
⟹ *invar2* (*map-to-nat iterate1 empty2 update2 s*)

This kind of ad-hoc specification should only be used when it is unlikely that the same function may be implemented differently.

5.3.4 Implementation

An implementation of an interface defines an actual data structure, an invariant, and implementations of the functions. An implementation has a two-letter (or three-letter) abbreviation that should be unique and a one-letter (or two-letter) abbreviation that should be unique amongst all implementations of the same interface.

Consider, for example, a set implementation by distinct lists. It has the abbreviations (lsi,li). To avoid name clashes with the existing list-set implementation in the framework, we use ticks (') here and there to disambiguate the names.

— The type of the data structure should be available as the two-letter abbreviation:
type-synonym *'a lsi' = 'a list*
— The abstraction function:
definition *lsi'-α == set*
— The invariant: In our case we constrain the lists to be distinct:
definition *lsi'-invar == distinct*
— The locale of the interface is interpreted with the two-letter abbreviation as prefix:
interpretation *lsi'*: *set' lsi'-α lsi'-invar* ⟨*proof*⟩

Next, we implement some functions. The implementation of a function *name* is prefixed by the two-letter prefix:

definition *lsi′-empty* == []

Each function implementation has a corresponding lemma that shows the instantiation of the locale. It is named by the function's name suffixed with *-impl*:

lemma *lsi′-empty-impl*: *set′-empty lsi′-α lsi′-invar lsi′-empty*
⟨*proof*⟩

The corresponding function's locale is interpreted with the function implementation and the interface's two-letter abbreviation as prefix:

interpretation *lsi′*: *set′-empty lsi′-α lsi′-invar lsi′-empty*
⟨*proof*⟩

This generates the lemma *lsi′.empty-correct*:

lsi′-α lsi′-empty = {}
lsi′-invar lsi′-empty

definition *lsi′-ins x l* == *if x∈set l then l else x#l*

Correctness may optionally be established using separate lemmas. These should be suffixed with *_aux* to indicate that they should not be used by other proofs:

lemma *lsi′-ins-correct-aux*:
lsi′-invar l ⟹ *lsi′-α* (*lsi′-ins x l*) = *insert x* (*lsi′-α l*)
lsi′-invar l ⟹ *lsi′-invar* (*lsi′-ins x l*)
⟨*proof*⟩

lemma *lsi′-ins-impl*: *set′-ins lsi′-α lsi′-invar lsi′-ins*
⟨*proof*⟩

interpretation *lsi′*: *set′-ins lsi′-α lsi′-invar lsi′-ins*
⟨*proof*⟩

5.3.5 Instantiations (Generic Algorithm)

The instantiation of a generic algorithm substitutes actual implementations for the required functions. A generic algorithm is instantiated by providing a definition that fixes the function parameters accordingly. Moreover, an *impl*-lemma and an interpretation of the implemented function's locale is provided. These can usually be constructed canonically from the generic algorithm's correctness lemma:

For example, consider conversion from lists to list-sets by instantiating the *list-to-set′*-algorithm:

definition *lsi'-list-to-set* == *list-to-set' lsi'-empty lsi'-ins*
lemmas *lsi'-list-to-set-impl* = *list-to-set'-correct*[*OF lsi'-empty-impl lsi'-ins-impl*, *folded lsi'-list-to-set-def*]
interpretation *lsi'*: *set'-list-to-set lsi'-α lsi'-invar lsi'-list-to-set*
⟨*proof*⟩

Note that the actual framework slightly deviates from the naming convention here, the corresponding instantiation of *SetGA.gen-list-to-set* is called *list-to-ls*, the *impl*-lemma is called *list-to-ls-impl*.

Generating all possible instantiations of generic algorithms based on the available implementations can be done mechanically. Currently, we have not implemented such an approach on the Isabelle ML-level. However, we used an ad-hoc ruby-script (*scripts/inst.rb*) to generate the thy-file *StdInst.thy* from the file *StdInst.in.thy*.

5.4 Design Issues

In this section, we motivate some of the design decisions of the Isabelle Collections Framework and report our experience with alternatives. Many of the design decisions are justified by restrictions of Isabelle/HOL and the code generator, so that there may be better options if those restrictions should vanish from future releases of Isabelle/HOL.

The main design goals of this development are:

1. Make available various implementations of collections under a unified interface.
2. It should be easy to extend the framework by new interfaces, functions, algorithms, and implementations.
3. Allow simple and concise reasoning over functions using collections.
4. Allow generic algorithms, that are independent of the actual data structure that is used.
5. Support generation of executable code.
6. Let the user precisely control what data structures are used in the implementation.

5.4.1 Data Refinement

In order to allow simple reasoning over collections, we use a data refinement approach. Each collection interface has an abstraction function that maps

it on a related Isabelle/HOL concept (abstract level). The specification of functions are also relative to the abstraction. This allows most of the correctness reasoning to be done on the abstract level. On this level, the tool support is more elaborated and one is not yet fixed to a concrete implementation. In a next step, the abstract specification is refined to use an actual implementation (concrete level). The correctness properties proven on the abstract level usually transfer easily to the concrete level.

Moreover, the user has precise control how the refinement is done, i.e. what data structures are used. An alternative would be to do refinement completely automatic, as e.g. done in the code generator setup of the Theory *Executable-Set*. This has the advantage that it induces less writing overhead. The disadvantage is that the user loses a great amount of control over the refinement. For example, in *Executable-Set*, all sets have to be represented by lists, and there is no possibility to represent one set differently from another.

5.4.2 Record Based Interfaces

We have experimented with grouping functions of an interface together via a record. This has the advantage that parameterization of generic algorithms becomes simpler, as multiple function parameters are replaced by a single record parameter. For maps and sets, theories *RecordSetImpl* and *RecordMapImpl* provide these instantiations for all implementations (except for tries). Moreover, the priority queue implementations contain such records for all important operations. The records do not include operations that depend on extra type variables because these operations would become monomorphic due to Isabelle's type system restrictions.

5.4.3 Locales for Generic Algorithms

Another tempting possibility to define a generic algorithm is to define a locale that includes the locales of all required functions, and do the definition of the generic algorithm inside that locale. This has the advantage that the function parameters are made implicit, thus improving readability. On the other hand, the code generator has problems with generating code from definitions inside a locale. Currently, one has to manually set up the code generator for such definitions. Moreover, when fixing function parameters in the declaration of the locale, their types will be inferred independently of the definitions later done in the locale context. In order to get the correct types, one has to add explicit type constraints. These tend to become rather lengthy, especially for iterator states. The approach taken in this framework – passing the required functions as explicit parameters to a generic algorithm – usually needs less type constraints, as type inference usually does most of

the job, in particular it infers the correct types of iterator states.

5.4.4 Explicit Invariants vs Typedef

The interfaces of this framework use explicit invariants. This provides a more general specification which allows some operations to be sometimes implemented more efficiently, cf. *lsi-ins-dj* in *ListSetImpl-Invar*.

Most implementations, however, hide the invariant in a typedef and setup the code generator appropriately. In that case, the invariant is just λ-. *True*, which still shows up in some premises and conclusions due to uniformity reasons.

end

Chapter 6

Examples

6.1 Examples from ITP-2010 slides

theory *itp-2010*
imports *../Collections*
begin

Illustrates the various possibilities how to use the ICF in your own algorithms by simple examples. The examples all use the data refinement scheme, and either define a generic algorithm or fix the operations.

— Example for slides

6.1.1 List to Set

In this simple example we do conversion from a list to a set. We define an abstract algorithm. This is then refined by a generic algorithm using a locale and by a generic algorithm fixing its operations as parameters.

Straightforward version

— Abstract algorithm
fun *set-a* **where**
  set-a [] *s* = *s* |
  set-a (*a*#*l*) *s* = *set-a l* (*insert a s*)

— Correctness of aa
lemma *set-a-correct*: *set-a l s* = *set l* ∪ *s*
  ⟨*proof*⟩
fun (**in** *StdSetDefs*) *set-i* **where**
  set-i [] *s* = *s* |
  set-i (*a*#*l*) *s* = *set-i l* (*ins a s*)

— Correct implementation of ca

lemma (**in** *StdSet*) *set-i-impl*: *invar s* $\Longrightarrow$ *invar* (*set-i l s*) $\wedge$ α (*set-i l s*) = *set-a l* (α *s*)
⟨*proof*⟩

definition *hs-seti* == *hsr.set-i*
declare *hsr.set-i.simps*[*folded hs-seti-def*, *code*]

lemmas *hs-set-i-impl* = *hsr.set-i-impl*[*folded hs-seti-def*]

export-code *hs-seti* **in** *SML* **file** −

— Code generation
⟨*ML*⟩

Tail-Recursive version

— Abstract algorithm
fun *set-a2* **where**
set-a2 [] = {} |
set-a2 (*a*#*l*) = (*insert a* (*set-a2 l*))

— Correctness of aa
lemma *set-a2-correct*: *set-a2 l* = *set l*
⟨*proof*⟩
fun (**in** *StdSetDefs*) *set-i2* **where**
set-i2 [] = *empty* () |
set-i2 (*a*#*l*) = (*ins a* (*set-i2 l*))

— Correct implementation of ca
lemma (**in** *StdSet*) *set-i2-impl*: *invar s* $\Longrightarrow$ *invar* (*set-i2 l*) $\wedge$ α (*set-i2 l*) = *set-a2 l*
⟨*proof*⟩
definition *hs-seti2* == *hsr.set-i2*
declare *hsr.set-i2.simps*[*folded hs-seti2-def*, *code*]

lemmas *hs-set-i2-impl* = *hsr.set-i2-impl*[*folded hs-seti2-def*]

— Code generation
⟨*ML*⟩

With explicit operation parameters

— Alternative for few operation parameters
fun *set-i′* **where**
!!*ins*. *set-i′ ins* [] *s* = *s* |
!!*ins*. *set-i′ ins* (*a*#*l*) *s* = *set-i′ ins l* (*ins a s*)

lemma (**in** *StdSet*) *set-i′-impl*:
invar s $\Longrightarrow$ *invar* (*set-i′ ins l s*) $\wedge$ α (*set-i′ ins l s*) = *set-a l* (α *s*)
⟨*proof*⟩

definition *hs-seti'* == *set-i' hs-ins*
lemmas *hs-set-i'-impl* = *hsr.set-i'-impl*[*folded hs-seti'-def*, *unfolded hs-ops-unfold*]

— Code generation
⟨*ML*⟩

6.1.2 Complex Example: Filter Average

In this more complex example, we develop a function that filters from a set all numbers that are above the average of the set.

First, we formulate this as a generic algorithm using a locale. This solution illustrates some technical problems with larger generic algorithms:

- Operations that are used with different types within the generic algorithm need to be specified multiple times, once for every type. This is an inherent problem with Isabelle's type system, and there is no obvious solution. This effect occurs especially when one uses the same type of set/map for different element types, or uses operations that have some additional type parameters, like the iterators in our example.

- The code generator has problems generating code for instantiated algorithms. This is due to the resulting equations having the instantiated part fixed on their lhs. This problem could probably be solved within the code generator. The workaround we use here is to define one new constant per function and instantiation and alter the code equations using this definitions. This could probably be automated and/or integrated into the code generator.

The second possibility avoids the above problems by fixing the used implementaion beforehand. Changing the implementation is still easy by changing the used operations. In this example, all used operations are introduced by abbbreviations, localizing the required changes to a small part of the theory.

abbreviation *average S* == $\sum$ *S div card S*

locale *MyContextDefs* =
  StdSetDefs ops **for** *ops* :: (*nat*,'*s*,'*more*) *set-ops-scheme* +
  fixes *iterate* :: '*s* => (*nat*,*nat*×*nat*) *set-iterator*
  fixes *iterate'* :: '*s* ⇒ (*nat*,'*s*) *set-iterator*
begin
  definition *avg-aux* :: '*s* ⇒ *nat*×*nat*
    where
    avg-aux s == *iterate s* (λ-. *True*) (λ*x* (*c*,*s*). (*c*+*1*, *s*+*x*)) (*0*,*0*)

  definition *avg s* == *case avg-aux s of* (*c*,*s*) ⇒ *s div c*

```
  definition filter-le-avg s == let a=avg s in
    iterate' s (λ-. True) (λx s. if x≤a then ins x s else s) (empty ())
end

locale MyContext = MyContextDefs ops iterate iterate' +
  StdSet ops +
  it!: set-iteratei α invar iterate +
  it'!: set-iteratei α invar iterate'
  for ops iterate iterate'
begin
  lemma avg-aux-correct: invar s ⟹ avg-aux s = (card (α s), ∑(α s) )
    ⟨proof⟩

  lemma avg-correct: invar s ⟹ avg s = average (α s)
    ⟨proof⟩

  lemma filter-le-avg-correct:
    invar s ⟹
      invar (filter-le-avg s) ∧
      α (filter-le-avg s) = {x∈α s. x≤average (α s)}
    ⟨proof⟩
end

interpretation hs-ctx: MyContext hs-ops hs-iteratei hs-iteratei
  ⟨proof⟩

definition test-hs == hs-ins (1::nat) (hs-ins 10 (hs-ins 11 (hs-empty ())))
definition testfun-hs == hs-ctx.filter-le-avg

definition hs-avg-aux == hs-ctx.avg-aux
definition hs-avg == hs-ctx.avg
definition hs-filter-le-avg == hs-ctx.filter-le-avg
lemmas hs-ctx-defs = hs-avg-aux-def hs-avg-def hs-filter-le-avg-def

lemmas [code] = hs-ctx.avg-aux-def[folded hs-ctx-defs]
lemmas [code] = hs-ctx.avg-def[folded hs-ctx-defs]
lemmas [code] = hs-ctx.filter-le-avg-def[folded hs-ctx-defs]


interpretation rs-ctx: MyContext rs-ops rs-iteratei rs-iteratei
  ⟨proof⟩

definition test-rs == rs-ins (1::nat) (rs-ins 10 (rs-ins 11 (rs-empty ())))
definition testfun-rs == rs-ctx.filter-le-avg

definition rs-avg-aux == rs-ctx.avg-aux
definition rs-avg == rs-ctx.avg
definition rs-filter-le-avg == rs-ctx.filter-le-avg
```

lemmas *rs-ctx-defs* = *rs-avg-aux-def rs-avg-def rs-filter-le-avg-def*

lemmas [*code*] = *rs-ctx.avg-aux-def*[*folded rs-ctx-defs*]
lemmas [*code*] = *rs-ctx.avg-def*[*folded rs-ctx-defs*]
lemmas [*code*] = *rs-ctx.filter-le-avg-def*[*folded rs-ctx-defs*]

— Code generation
⟨*ML*⟩

type-synonym *'a my-set* = *'a hs*
abbreviation *my-α* == *hs-α*
abbreviation *my-invar* == *hs-invar*
abbreviation *my-empty* == *hs-empty*
abbreviation *my-ins* == *hs-ins*
abbreviation *my-iterate* == *hs-iteratei*
lemmas *my-correct* = *hs-correct*
lemmas *my-iterate-rule-P* = *hs.iterate-rule-P*

definition *avg-aux* :: *nat my-set* ⇒ *nat*×*nat*
where
avg-aux s == *my-iterate s* (λ-. *True*) (λ*x* (*c*,*s*). (*c*+*1*, *s*+*x*)) (*0*,*0*)

definition *avg s* == *case avg-aux s of* (*c*,*s*) ⇒ *s div c*

definition *filter-le-avg s* == *let a*=*avg s in*
my-iterate s (λ-. *True*) (λ*x s*. *if x*≤*a then my-ins x s else s*) (*my-empty* ())

lemma *avg-aux-correct*: *my-invar s* ⟹ *avg-aux s* = (*card* (*my-α s*), ∑ (*my-α s*))
⟨*proof*⟩

lemma *avg-correct*: *my-invar s* ⟹ *avg s* = *average* (*my-α s*)
⟨*proof*⟩

lemma *filter-le-avg-correct*:
my-invar s ⟹
my-invar (*filter-le-avg s*) ∧
my-α (*filter-le-avg s*) = {*x*∈*my-α s*. *x*≤*average* (*my-α s*)}
⟨*proof*⟩

definition *test-set* == *my-ins* (*1*::*nat*) (*my-ins 2* (*my-ins 3* (*my-empty* ())))

export-code *avg-aux avg filter-le-avg test-set* **in** *SML* **module-name** *Test* **file** –

end

6.2 State Space Exploration

theory *Exploration*
imports *Main ../common/Misc ../DatRef*
begin

In this theory, a workset algorithm for state-space exploration is defined. It explores the set of states that are reachable by a given relation, starting at a given set of initial states.

The specification makes use of the data refinement framework for while-algorithms (cf. Section 4.30), and is thus suited for being refined to an executable algorithm, using the Isabelle Collections Framework to provide the necessary data structures.

6.2.1 Generic Search Algorithm

— The algorithm contains a set of discovered states and a workset
type-synonym $'\Sigma$ *sse-state* = $'\Sigma$ *set* × $'\Sigma$ *set*

— Loop body
inductive-set
sse-step :: ($'\Sigma$×$'\Sigma$) *set* ⇒ ($'\Sigma$ *sse-state* × $'\Sigma$ *sse-state*) *set*
for R **where**
⟦ $\sigma \in W$;
$\Sigma' = \Sigma \cup (R``\{\sigma\})$;
$W' = (W - \{\sigma\}) \cup ((R``\{\sigma\}) - \Sigma)$
⟧ ⟹ $((\Sigma, W),(\Sigma', W')) \in$ *sse-step* R

— Loop condition
definition *sse-cond* :: $'\Sigma$ *sse-state set* **where**
sse-cond = $\{(\Sigma, W).\ W \neq \{\}\}$

— Initial state
definition *sse-initial* :: $'\Sigma$ *set* ⇒ $'\Sigma$ *sse-state* **where**
sse-initial Σi == (Σi,Σi)

— Invariants:

- The workset contains only states that are already discovered.
- All discovered states are target states
- If there is a target state that is not discovered yet, then there is an element in the workset from that this target state is reachable without using discovered states as intermediate states. This supports the intuition of the workset as a frontier between the sets of discovered and undiscovered states.

definition *sse-invar* :: $'\Sigma$ *set* $\Rightarrow$ $('\Sigma \times '\Sigma)$ *set* $\Rightarrow$ $'\Sigma$ *sse-state set* **where**
sse-invar $\Sigma i\ R = \{(\Sigma, W).$
$W \subseteq \Sigma \wedge$
$(\Sigma \subseteq R^* ``\Sigma i) \wedge$
$(\forall \sigma \in (R^* ``\Sigma i) - \Sigma.\ \exists \sigma h \in W.\ (\sigma h, \sigma) \in (R - (UNIV \times \Sigma))^*)$
$\}$

definition *sse-algo* $\Sigma i\ R ==$
$(\!|$ *wa-cond*=*sse-cond*,
wa-step=*sse-step* R,
wa-initial $= \{$*sse-initial* $\Sigma i\}$,
wa-invar $=$ *sse-invar* $\Sigma i\ R$ $|\!)$

definition *sse-term-rel* $\Sigma i\ R ==$
$\{\ (\sigma', \sigma).\ \sigma \in$ *sse-invar* $\Sigma i\ R \wedge (\sigma, \sigma') \in$ *sse-step* $R\ \}$

— Termination: Either a new state is discovered, or the workset shrinks
theorem *sse-term*:
assumes *finite*[*simp*, *intro*!]: *finite* $(R^* ``\Sigma i)$
shows *wf* (*sse-term-rel* $\Sigma i\ R$)
⟨*proof*⟩

lemma *sse-invar-initial*: (*sse-initial* Σi) $\in$ *sse-invar* $\Sigma i\ R$
⟨*proof*⟩
theorem *sse-invar-final*:
$\forall S.\ S \in$ *wa-invar* (*sse-algo* $\Sigma i\ R$) $\wedge\ S \notin$ *wa-cond* (*sse-algo* $\Sigma i\ R$)
$\longrightarrow$ *fst* $S = R^* ``\Sigma i$
⟨*proof*⟩

lemma *sse-invar-step*: $[\![S \in$ *sse-invar* $\Sigma i\ R$; $(S, S') \in$ *sse-step* $R]\!]$
$\Longrightarrow S' \in$ *sse-invar* $\Sigma i\ R$
— Split the goal by the invariant:
⟨*proof*⟩
theorem *sse-while-algo*: *finite* $(R^* ``\Sigma i) \Longrightarrow$ *while-algo* (*sse-algo* $\Sigma i\ R$)
⟨*proof*⟩

6.2.2 Depth First Search

In this section, the generic state space exploration algorithm is refined to a DFS-algorithm, that uses a stack to implement the workset.

type-synonym $'\Sigma$ *dfs-state* $=$ $'\Sigma$ *set* $\times$ $'\Sigma$ *list*

definition *dfs-*α :: $'\Sigma$ *dfs-state* $\Rightarrow$ $'\Sigma$ *sse-state*
where *dfs-*α $S ==$ *let* $(\Sigma, W) = S$ *in* $(\Sigma,$*set* $W)$

definition *dfs-invar-add* :: $'\Sigma$ *dfs-state set*
where *dfs-invar-add* $== \{(\Sigma, W).$ *distinct* $W\}$

definition *dfs-invar* $\Sigma i\ R$ == *dfs-invar-add* $\cap$ { s. *dfs-*α $s \in$ *sse-invar* $\Sigma i\ R$ }

inductive-set *dfs-initial* :: $'\Sigma$ *set* $\Rightarrow$ $'\Sigma$ *dfs-state set* **for** Σi
where $[\![$ *distinct* W; *set* $W = \Sigma i]\!] \Longrightarrow (\Sigma i, W) \in$ *dfs-initial* Σi

inductive-set *dfs-step* :: $('\Sigma \times '\Sigma)$ *set* $\Rightarrow$ ($'\Sigma$ *dfs-state* $\times '\Sigma$ *dfs-state*) *set*
for R **where**
$[\![$ $W = \sigma \# Wtl$;
distinct Wn;
set $Wn = R\,``\{\sigma\} - \Sigma$;
$W' = Wn @ Wtl$;
$\Sigma' = R\,``\{\sigma\} \cup \Sigma$
$]\!] \Longrightarrow ((\Sigma, W), (\Sigma', W')) \in$ *dfs-step* R

definition *dfs-cond* :: $'\Sigma$ *dfs-state set*
where *dfs-cond* == { (Σ, W). $W \neq []$}

definition *dfs-algo* $\Sigma i\ R$ == (|
wa-cond = *dfs-cond*,
wa-step = *dfs-step* R,
wa-initial = *dfs-initial* Σi,
wa-invar = *dfs-invar* $\Sigma i\ R$ |)

— The DFS-algorithm refines the state-space exploration algorithm
theorem *dfs-pref-sse*:
wa-precise-refine (*dfs-algo* $\Sigma i\ R$) (*sse-algo* $\Sigma i\ R$) *dfs-*α
⟨*proof*⟩
theorem *dfs-while-algo*:
assumes *finite*[*simp*, *intro*!]: *finite* ($R^*\,``\Sigma i$)
shows *while-algo* (*dfs-algo* $\Sigma i\ R$)
⟨*proof*⟩
theorems *dfs-invar-final* =
wa-precise-refine.transfer-correctness[*OF dfs-pref-sse sse-invar-final*]

end

6.3 DFS Implementation by Hashset

theory *Exploration-DFS*
imports *../Collections Exploration*
begin

This theory implements the DFS-algorithm by using a hashset to remember the explored states. It illustrates how to use data refinement with the Isabelle Collections Framework in a realistic, non-trivial application.

6.3.1 Definitions

```
— The concrete algorithm uses a hashset ('q hs) and a worklist.
type-synonym 'q hs-dfs-state = 'q hs × 'q list

— The loop terminates on empty worklist
definition hs-dfs-cond :: 'q hs-dfs-state ⇒ bool
  where hs-dfs-cond S == let (Q,W) = S in W≠[]

— Refinement of a DFS-step, using hashset operations
definition hs-dfs-step
  :: ('q::hashable ⇒ 'q ls) ⇒ 'q hs-dfs-state ⇒ 'q hs-dfs-state
  where hs-dfs-step post S == let
    (Q,W) = S;
    σ=hd W
  in
    ls-iteratei (post σ) (λ-. True) (λx (Q,W).
      if hs-memb x Q then
        (Q,W)
      else (hs-ins x Q,x#W)
      )
      (Q, tl W)


— Convert post-function to relation
definition hs-R :: ('q ⇒ 'q ls) ⇒ ('q×'q) set
  where hs-R post == {(q,q'). q'∈ls-α (post q)}

— Initial state: Set of initial states in discovered set and on worklist
definition hs-dfs-initial :: 'q::hashable hs ⇒ 'q hs-dfs-state
  where hs-dfs-initial Σi == (Σi,hs-to-list Σi)

— Abstraction mapping to abstract-DFS state
definition hs-dfs-α :: 'q::hashable hs-dfs-state ⇒ 'q dfs-state
  where hs-dfs-α S == let (Q,W)=S in (hs-α Q,W)



— Combined concrete and abstract level invariant
definition hs-dfs-invar
  :: 'q::hashable hs ⇒ ('q ⇒ 'q ls) ⇒ 'q hs-dfs-state set
  where hs-dfs-invar Σi post ==
    (*hs-dfs-invar-add ∩*) { s. (hs-dfs-α s) ∈ dfs-invar (hs-α Σi) (hs-R post) }

— The deterministic while-algorithm
definition hs-dfs-dwa Σi post == (|
  dwa-cond = hs-dfs-cond,
  dwa-step = hs-dfs-step post,
  dwa-initial = hs-dfs-initial Σi,
  dwa-invar = hs-dfs-invar Σi post
```

⟫

— Executable DFS-search. Given a set of initial states, and a successor function, this function performs a DFS search to return the set of reachable states.
definition *hs-dfs* Σ*i post*
== *fst* (*while hs-dfs-cond* (*hs-dfs-step post*) (*hs-dfs-initial* Σ*i*))

6.3.2 Refinement

We first show that a concrete step implements its abstract specification, and preserves the additional concrete invariant

lemma *hs-dfs-step-correct*:

assumes *ne*: *hs-dfs-cond* (*Q*, *W*)
shows (*hs-dfs-α* (*Q*, *W*),*hs-dfs-α* (*hs-dfs-step post* (*Q*, *W*)))
∈*dfs-step* (*hs-R post*) (**is** *?T1*)

⟨*proof*⟩
theorem *hs-dfs-pref-dfs*:

shows *wa-precise-refine*
(*det-wa-wa* (*hs-dfs-dwa* Σ*i post*))
(*dfs-algo* (*hs-α* Σ*i*) (*hs-R post*))
hs-dfs-α
⟨*proof*⟩
theorem *hs-dfs-while-algo*:

assumes *finite*[*simp*]: *finite* ((*hs-R post*)* `` *hs-α* Σ*i*)
shows *while-algo* (*det-wa-wa* (*hs-dfs-dwa* Σ*i post*))
⟨*proof*⟩
theorems *hs-dfs-det-while-algo* = *det-while-algo-intro*[*OF hs-dfs-while-algo*]

— Transferred correctness theorem
theorems *hs-dfs-invar-final* =
wa-precise-refine.transfer-correctness[*OF*
hs-dfs-pref-dfs dfs-invar-final]

— The executable implementation is correct
theorem *hs-dfs-correct*:

assumes *finite*[*simp*]: *finite* ((*hs-R post*)* `` *hs-α* Σ*i*)
shows *hs-α* (*hs-dfs* Σ*i post*) = (*hs-R post*)* ``*hs-α* Σ*i* (**is** *?T1*)

⟨*proof*⟩

6.3.3 Code Generation

export-code *hs-dfs* **in** *SML* **module-name** *DFS* **file** –

export-code *hs-dfs* **in** *OCaml* **module-name** *DFS* **file** –
export-code *hs-dfs* **in** *Haskell* **module-name** *DFS* **file** –

end

6.4 All Working Examples

theory *All*
imports *../Collections*
itp-2010
Exploration
Exploration-DFS
begin

end

Chapter 7

Conclusion

This work presented the Isabelle Collections Framework, an efficient and extensible collections framework for Isabelle/HOL. The framework features data-refinement techniques to refine algorithms to use concrete collection datastructures, and is compatible with the Isabelle/HOL code generator, such that efficient code can be generated for all supported target languages. Finally, we defined a data refinement framework for the while-combinator, and used it to specify a state-space exploration algorithm and stepwise refined the specification to an executable DFS-algorithm using a hashset to store the set of already known states.

Up to now, interfaces for sets and maps are specified and implemented using lists, red-black-trees, and hashing. Moreover, an amortized constant time fifo-queue (based on two stacks) has been implemented. However, the framwork is extensible, i.e. new interfaces, algorithms and implementations can easily be added and integrated with the existing ones.

7.1 Future Work

There are several starting points for future work:

- Currently, the generic algorithms are instantiated semi-automatically by an ad-hoc ruby script and a manual description of the generic algorithms to be instantiated. However, the process of instantiating generic algorithms could be fully mechanized, if one would add support for specifying interfaces, implementations and generic algorithms in Isabelle/HOL.
- Generic algorithms are declared as functions that take the required functions as arguments. While this is convenient for small generic algorithms, it can become tedious for larger algorithms, involving many interfaces. Luckily, the generic algorithms defined in this library are

rather small. For bigger developments, we currently recommend to fix the used implementations, i.e. to define specific algorithms rather than generic ones. This is, for example, done in the DFS-search example (Section 6.3), where we fix hashsets instead of defining a generic algorithm for all set implementations. Hence there is a need for exploring more convenient ways to specify generic algorithms.

7.2 Trusted Code Base

In this section we shortly characterize on what our formal proofs depend, i.e. how to interpret the information contained in this formal proof and the fact that it is accepted by the Isabelle/HOL system.

First of all, you have to trust the theorem prover and its axiomatization of HOL, the ML-platform, the operating system software and the hardware it runs on. All these components are, in theory, able to cause false theorems to be proven. However, the probability of a false theorem to get proven due to a hardware error or an error in the operating system software is reasonably low. There are errors in hardware and operating systems, but they will usually cause the system to crash or exhibit other unexpected behaviour, instead of causing Isabelle to quitely accept a false theorem and behave normal otherwise. The theorem prover itself is a bit more critical in this aspect. However, Isabelle/HOL is implemented in LCF-style, i.e. all the proofs are eventually checked by a small kernel of trusted code, containing rather simple operations. HOL is the logic that is most frequently used with Isabelle, and it is unlikely that it's axiomatization in Isabelle is inconsistent and no one has found and reported this inconsistency yet.

The next crucial point is the code generator of Isabelle. We derive executable code from our specifications. The code generator contains another (thin) layer of untrusted code. This layer has some known deficiencies[1] (as of Isabelle2009) in the sense that invalid code is generated. This code is then rejected by the target language's compiler or interpreter, but does not silently compute the wrong thing.

Moreover, assuming correctness of the code generator, the generated code is only guaranteed to be *partially* correct[2], i.e. there are no formal termination guarantees.

[1] For example, the Haskell code generator may generate variables starting with uppercase letters, while the Haskell-specification requires variables to start with lowercase letters. Moreover, the ML code generator does not know the ML value restriction, and may generate code that violates this restriction.

[2] A simple example is the always-diverging function $\mathsf{f}_{\mathsf{div}} :: \mathsf{bool} = \mathsf{while}\ (\lambda x.\ \mathsf{True})\ \mathsf{id}\ \mathsf{True}$ that is definable in HOL. The lemma $\forall x.\ x = \mathsf{if}\ \mathsf{f}_{\mathsf{div}}\ \mathsf{then}\ x\ \mathsf{else}\ x$ is provable in Isabelle and rewriting based on it could, theoretically, be inserted before the code generation process, resulting in code that always diverges

Furthermore, manual adaptations of the code generator setup are also part of the trusted code base. For array-based hash maps, the Isabelle Collections Framework provides an ML implementation for arrays with in-place updates that is unverified; for Haskell, we use the DiffArray implementation from the Haskell library. Other than this, the Isabelle Collections Framework does not add any adaptations other than those available in the Isabelle/HOL library, in particular Efficient_Nat.

7.3 Acknowledgement

We thank Tobias Nipkow for encouraging us to make the collections framework an independent development. Moreover, we thank Markus Müller-Olm for discussion about data-refinement. Finally, we thank the people on the Isabelle mailing list for quick and useful response to any Isabelle-related questions.

Bibliography

[1] Java: The collections framework. Available on: http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html.

[2] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[3] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, November 1995.