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*:
    $\alpha\ (empty\ ()) = Map.empty$
    $invar\ (empty\ ())$

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 $update\text{-}dj :: {'u} \Rightarrow {'v} \Rightarrow {'s} \Rightarrow {'s}$
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 $add\text{-}dj :: {'s} \Rightarrow {'s} \Rightarrow {'s}$
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 α :: $'s \Rightarrow 'u \rightharpoonup 'v$
  fixes *sng* :: $'u \Rightarrow 'v \Rightarrow 's$
  assumes *sng-correct* :
    α (*sng k v*) = $[k \mapsto v]$
    invar (*sng k v*)

locale *map-isSng* = *map* +
  constrains α :: $'s \Rightarrow 'k \rightharpoonup 'v$
  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* $\longleftrightarrow$ $(\exists!k.\ \exists\, v.\ (\alpha\ s\ k = Some\ v)))$
    apply (*auto simp add*: *isSng-correct split*: *split-if-asm*)
    apply (*rule-tac x=k* **in** *exI*)
    apply (*rule-tac x=v* **in** *exI*)
    apply (*rule ext*)
    apply (*case-tac* α *s x*)
    apply *auto*
    apply *force*
    done

  lemma *isSng-correct-card* :
    invar s $\Longrightarrow$ (*isSng s* $\longleftrightarrow$ (*card* (*dom* (α *s*)) = *1*))
    by (*auto simp add*: *isSng-correct card-Suc-eq dom-eq-singleton-conv*)

end

Finite Maps

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

Size

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

locale *map-size-abort* = *finite-map* +
  constrains α :: $'s \Rightarrow 'u \rightharpoonup 'v$
  fixes *size-abort* :: $nat \Rightarrow 's \Rightarrow nat$
  assumes *size-abort-correct*: *invar s* $\Longrightarrow$ *size-abort m s* = *min m* (*card* (*dom* (α
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 α :: $'s \Rightarrow 'u \rightharpoonup 'v$
  fixes *iteratei* :: $'s \Rightarrow ('u \times 'v,'\sigma)$ *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* $\in$ *it*; α *m k* = *Some v*; *it* $\subseteq$ *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* $\subseteq$ *dom* (α *m*); *it* $\neq$ {}; $\neg$ *c* σ; *I it* σ $]\!]$ $\Longrightarrow$ *P* σ
    shows *P* (*iteratei m c f* $\sigma 0$)
    using *map-iterator-rule-P* [*OF iteratei-rule*, *of m I* $\sigma 0$ *c f P*]
    by (*simp-all add*: *assms*)

  lemma *iteratei-rule-insert-P*:
    assumes
      invar m
      I {} $\sigma 0$
      !!*k v it* σ. $[\![$ *c* σ; *k* $\in$ (*dom* (α *m*) $-$ *it*); α *m k* = *Some v*; *it* $\subseteq$ *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* $\subseteq$ *dom* (α *m*); *it* $\neq$ *dom* (α *m*);
          $\neg$ (*c* σ);
          I it σ $]\!]$ $\Longrightarrow$ *P* σ
    shows *P* (*iteratei m c f* $\sigma 0$)
    using *map-iterator-rule-insert-P* [*OF iteratei-rule*, *of m I* $\sigma 0$ *c f P*]
    by (*simp-all add*: *assms*)

  lemma *iterate-rule-P*:
    $[\![$
      invar m;
      I (*dom* (α *m*)) $\sigma 0$;
      !!*k v it* σ. $[\![$ *k* $\in$ *it*; α *m k* = *Some v*; *it* $\subseteq$ *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* $\sigma 0$)
    using *iteratei-rule-P* [*of m I* $\sigma 0$ λ-. *True f P*]
    by *fast*

lemma *iterate-rule-insert-P*:
$\llbracket$
invar m;
$I\ \{\}\ \sigma 0$;
$!!k\ v\ it\ \sigma.\ \llbracket\ k \in (dom\ (\alpha\ m) - it);\ \alpha\ m\ k = Some\ v;\ it \subseteq dom\ (\alpha\ m);\ I\ it\ \sigma$
$\rrbracket$
$\Longrightarrow I\ (insert\ k\ it)\ (f\ (k,\ v)\ \sigma)$;
$!!\sigma.\ I\ (dom\ (\alpha\ m))\ \sigma \Longrightarrow P\ \sigma$
$\rrbracket \Longrightarrow P\ (iteratei\ m\ (\lambda\text{-}.\ True)\ f\ \sigma 0)$
using *iteratei-rule-insert-P* [*of m I* $\sigma 0$ λ-. *True f P*]
by *fast*
end

lemma *map-iteratei-I* :
assumes $\bigwedge m.\ invar\ m \Longrightarrow map\text{-}iterator\ (iti\ m)\ (\alpha\ m)$
shows *map-iteratei* α *invar iti*
proof
fix *m*
assume *invar-m*: *invar m*
from *assms(1)*[*OF invar-m*] **show** *it-OK*: *map-iterator* (*iti m*) (α *m*) **.**

from *set-iterator-genord.finite-S0* [*OF it-OK*[*unfolded set-iterator-def*]]
show *finite* (*dom* (α *m*)) **by** (*simp add*: *finite-map-to-set*)
qed

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*:
$\llbracket\ invar\ m;\ \alpha\ m\ u = Some\ v;\ f\ (u,\ v) = Some\ r;$
$!!u\ v\ r.\ \llbracket\ sel\ m\ f = Some\ r;\ \alpha\ m\ u = Some\ v;\ f\ (u,\ v) = Some\ r\ \rrbracket \Longrightarrow Q$
$\rrbracket \Longrightarrow Q$
assumes *selI*:
$\llbracket\ invar\ m;\ \forall u\ v.\ \alpha\ m\ u = Some\ v \longrightarrow f\ (u,\ v) = None\ \rrbracket \Longrightarrow sel\ m\ f = None$

```
begin
  lemma sel-someE:
    ⟦ invar m; sel m f = Some r;
      !!u v. ⟦ α m u = Some v; f (u, v) = Some r ⟧ ⟹ P
    ⟧ ⟹ P
    apply (cases ∃ u v r. α m u = Some v ∧ f (u, v) = Some r)
    apply safe
    apply (erule-tac u=u and v=v and r=ra in selE)
    apply assumption
    apply assumption
    apply simp
    apply (auto)
    apply (drule (1) selI)
    apply simp
    done

  lemma sel-noneD: ⟦invar m; sel m f = None; α m u = Some v⟧ ⟹ f (u, v)
= None
    apply (rule ccontr)
    apply simp
    apply (erule exE)
    apply (erule-tac f=f and u=u and v=v and r=y in selE)
    apply auto
    done

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. ⟦α s u = Some v; f (u, v) = Some r⟧ ⟹ P
               ⟧ ⟹ P
  assumes S2:
    !!s f u v. ⟦invar s; sel s f = None; α s u = Some v⟧ ⟹ f (u, v) = None
  shows map-sel α invar sel
proof –
  show ?thesis
    apply (unfold-locales)
    apply (case-tac sel m f)
    apply (force dest: S2)
    apply (force elim: S1)
    apply (case-tac sel m f)
    apply assumption
    apply (force elim: S1)
    done
qed
```

Selection of Entry (without mapping)

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

begin
  lemma *sel'-someE*:
    $\llbracket$ *invar m*; *sel' m P* = *Some* (*u*,*v*);
      !!*u v*. $\llbracket$ α *m u* = *Some v*; *P* (*u*, *v*) $\rrbracket$ $\Longrightarrow$ *thesis*
    $\rrbracket$ $\Longrightarrow$ *thesis*
    apply (*cases* $\exists$ *u v*. α *m u* = *Some v* $\wedge$ *P* (*u*, *v*))
    apply *safe*
    apply (*erule-tac u*=*ua* **and** *v*=*va* **in** *sel'E*)
    apply *assumption*
    apply *assumption*
    apply *simp*
    apply (*auto*)
    apply (*drule* (*1*) *sel'I*)
    apply *simp*
    done

  lemma *sel'-noneD*: $\llbracket$*invar m*; *sel' m P* = *None*; α *m u* = *Some v*$\rrbracket$ $\Longrightarrow$ $\neg$ *P* (*u*,
v)
    apply (*rule ccontr*)
    apply *simp*
    apply (*erule* (*2*) *sel'E*[**where** *P*=*P*])
    apply *auto*
    done

  lemma *sel'-SomeD*:
    $\llbracket$ *sel' m P* = *Some* (*u*, *v*); *invar m* $\rrbracket$ $\Longrightarrow$ α *m u* = *Some v* $\wedge$ *P* (*u*, *v*)
    apply(*cases* $\exists$ *u' v'*. α *m u'* = *Some v'* $\wedge$ *P* (*u'*, *v'*))
     apply *clarsimp*
     apply(*erule* (*2*) *sel'E*[**where** *P*=*P*])
     apply *simp*
    apply(*clarsimp*)
    apply(*drule* (*1*) *sel'I*)
    apply *simp*
    done
end

Map to List Conversion

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

List to Map Conversion

locale *list-to-map* = *map* +
  constrains α :: $'s \Rightarrow 'u \rightharpoonup 'v$
  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* ::
  $('u1 \rightharpoonup 'v1) \Rightarrow ('u1 \times 'v1 \rightharpoonup ('u2 \times 'v2)) \Rightarrow$ *bool*
  where
  transforms-to-unique-keys m f $\equiv$ ($\forall$ *k1 k2 v1 v2 k' v1' v2'*. (
        m k1 = *Some v1* $\wedge$
        m k2 = *Some v2* $\wedge$
        f (*k1*, *v1*) = *Some* (*k'*, *v1'*) $\wedge$
        f (*k2*, *v2*) = *Some* (*k'*, *v2'*)) -->
      (*k1* = *k2*))

locale *map-image-filter* = *map* α*1 invar1* + *map* α*2 invar2*
  for α*1* :: $'m1 \Rightarrow 'u1 \rightharpoonup 'v1$ **and** *invar1*
  and α*2* :: $'m2 \Rightarrow 'u2 \rightharpoonup 'v2$ **and** *invar2*
  +
  fixes *map-image-filter* :: $('u1 \times 'v1 \Rightarrow ('u2 \times 'v2)$ *option*$) \Rightarrow 'm1 \Rightarrow 'm2$
  assumes *map-image-filter-correct-aux1*:
    $\bigwedge$*k' v'*.
    ⟦*invar1 m*; *transforms-to-unique-keys* (α*1 m*) *f*⟧ $\Longrightarrow$
    (*invar2* (*map-image-filter f m*) $\wedge$
     ((α*2* (*map-image-filter f m*) *k'* = *Some v'*) $\longleftrightarrow$
      ($\exists$ *k v*. (α*1 m k* = *Some v*) $\wedge$ *f* (*k*, *v*) = *Some* (*k'*, *v'*))))
begin

  lemma *map-image-filter-correct-aux2* :
    assumes *invar1 m*

and *transforms-to-unique-keys* $(\alpha 1\ m)\ f$
shows $(\alpha 2\ (\textit{map-image-filter}\ f\ m)\ k' = \textit{None}) \longleftrightarrow$
$(\forall\, k\ v\ v'.\ \alpha 1\ m\ k = \textit{Some}\ v \longrightarrow f\ (k,\ v) \neq \textit{Some}\ (k',\ v'))$
proof −
note *map-image-filter-correct-aux1* [*OF assms*]
have *Some-eq*: $\bigwedge v'.\ (\alpha 2\ (\textit{map-image-filter}\ f\ m)\ k' = \textit{Some}\ v') =$
$(\exists\, k\ v.\ \alpha 1\ m\ k = \textit{Some}\ v \wedge f\ (k,\ v) = \textit{Some}\ (k',\ v'))$
by (*simp add*: *map-image-filter-correct-aux1* [*OF assms*])

have *intro-some*: $(\alpha 2\ (\textit{map-image-filter}\ f\ m)\ k' = \textit{None}) \longleftrightarrow$
$(\forall\, v'.\ \alpha 2\ (\textit{map-image-filter}\ f\ m)\ k' \neq \textit{Some}\ v')$ **by** *auto*

from *intro-some Some-eq* **show** *?thesis* **by** *auto*
qed

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* $\alpha 1$ *invar1* + *map* $\alpha 2$ *invar2*
for $\alpha 1 :: 'm1 \Rightarrow 'u \rightharpoonup 'v1$ **and** *invar1*
and $\alpha 2 :: 'm2 \Rightarrow 'u \rightharpoonup 'v2$ **and** *invar2*
+
fixes *map-value-image-filter* :: $('u \Rightarrow 'v1 \Rightarrow 'v2\ \textit{option}) \Rightarrow 'm1 \Rightarrow 'm2$
assumes *map-value-image-filter-correct* :
$\textit{invar1}\ m \Longrightarrow$
$\textit{invar2}\ (\textit{map-value-image-filter}\ f\ m) \wedge$
$(\alpha 2\ (\textit{map-value-image-filter}\ f\ m) =$
$(\lambda k.\ \textit{Option.bind}\ (\alpha 1\ m\ k)\ (f\ k)))$
begin

lemma *map-value-image-filter-correct-alt* :
$\textit{invar1}\ m \Longrightarrow$
$\textit{invar2}\ (\textit{map-value-image-filter}\ f\ m)$
$\textit{invar1}\ m \Longrightarrow$
$(\alpha 2\ (\textit{map-value-image-filter}\ f\ m)\ k = \textit{Some}\ v') \longleftrightarrow$
$(\exists\, v.\ (\alpha 1\ m\ k = \textit{Some}\ v) \wedge f\ k\ v = \textit{Some}\ v')$
$\textit{invar1}\ m \Longrightarrow$
$(\alpha 2\ (\textit{map-value-image-filter}\ f\ m)\ k = \textit{None}) \longleftrightarrow$
$(\forall\, v.\ (\alpha 1\ m\ k = \textit{Some}\ v) \mathrel{-\!-\!>} f\ k\ v = \textit{None})$
proof −
assume *invar-m* : *invar1 m*
note *aux* = *map-value-image-filter-correct* [*OF invar-m*]

from *aux* **show** *invar2* (*map-value-image-filter f m*) **by** *simp*

```
    from aux show (α2 (map-value-image-filter f m) k = Some v') ⟷
     (∃ v. (α1 m k = Some v) ∧ f k v = Some v')
      by (cases α1 m k, simp-all)
    from aux show (α2 (map-value-image-filter f m) k = None) ⟷
     (∀ v. (α1 m k = Some v) --> f k v = None)
      by (cases α1 m k, simp-all)
  qed
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 -
    assume invar-m : invar1 m
    have α1 m |` {k. (∃ v. α1 m k = Some v) ∧ P k} = α1 m |` {k. P k}
      (is α1 m |` ?A1 = α1 m |` ?A2)
    proof
      fix k
      show (α1 m |` ?A1) k = (α1 m |` ?A2) k
      proof (cases k ∈ ?A2)
        case False thus ?thesis by simp
      next
        case True
        hence P-k : P k by simp

        show ?thesis
          by (cases α1 m k, simp-all add: P-k)
      qed
    qed
    with invar-m show α2 (restrict (λ(k, -). P k) m) = α1 m |` {k. P k}
      by (simp add: restrict-correct-aux1)
  qed

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

2.1.2 Ordered Maps

locale *ordered-map* = *map* α *invar*
  for α :: $'s \Rightarrow ('u::linorder) \rightharpoonup 'v$ **and** *invar*

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

Ordered Iteration

locale *map-iterateoi* = *ordered-finite-map* α *invar*
  for α :: $'s \Rightarrow ('u::linorder) \rightharpoonup 'v$ **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;
      $k \in it$;
      $\forall j{\in}it.\ k{\le}j$;
      $\forall j{\in}dom\ (\alpha\ m) - it.\ j{\le}k$;
      $\alpha\ m\ k = Some\ v$;
      $it \subseteq dom\ (\alpha\ m)$;
      I it σ
    $]\!] \Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
    assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
    assumes *II*: !!σ *it*. $[\![$
      $it \subseteq dom\ (\alpha\ m)$;
      $it \neq \{\}$;
      $\neg\ c\ \sigma$;
      I it σ;
      $\forall k{\in}it.\ \forall j{\in}dom\ (\alpha\ m) - it.\ j{\le}k$
    $]\!] \Longrightarrow P\ \sigma$
    shows *P* (*iterateoi m c f* $\sigma 0$)
  using *map-iterator-linord-rule-P* [*OF iterateoi-rule*, *of m I* $\sigma 0$ *c f P*] *assms*
  by *simp*

  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. $[\![\ k \in it;\ \forall j{\in}it.\ k{\le}j;\ \forall j{\in}dom\ (\alpha\ m) - it.\ j{\le}k;\ \alpha\ m$
$k = Some\ v;\ it \subseteq dom\ (\alpha\ m);\ I\ it\ \sigma\ ]\!]$
        $\Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
    assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
    shows *P* (*iterateoi m* (λ-. *True*) *f* $\sigma 0$)
  using *map-iterator-linord-rule-P* [*OF iterateoi-rule*, *of m I* $\sigma 0$ λ-. *True f P*]

assms
  by *simp*
 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
  fix *m*
  assume *invar-m*: *invar m*
  from *assms*(*1*)[*OF invar-m*] **show** *it-OK*: *map-iterator-linord* (*itoi m*) (α *m*)
.

  from *set-iterator-genord.finite-S0* [*OF it-OK*[*unfolded set-iterator-map-linord-def*]]
  show *finite* (*dom* (α *m*)) **by** (*simp add*: *finite-map-to-set*)
 qed

 locale *map-reverse-iterateoi* = *ordered-finite-map* α *invar*
  for α :: $'s \Rightarrow ('u$::*linorder*) $\rightharpoonup$ $'v$ **and** *invar*
  +
  fixes *reverse-iterateoi* :: $'s \Rightarrow ('u \times 'v,'\sigma)$ *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*)) $\sigma 0$
   assumes *IP*: !!*k v it* σ. $[\![$
    c σ;
    $k \in it$;
    $\forall j{\in}it.\ k{\geq}j$;
    $\forall j{\in}dom\ (\alpha\ m) - it.\ j{\geq}k$;
    $\alpha\ m\ k = Some\ v$;
    $it \subseteq dom\ (\alpha\ m)$;
    I it σ
   $]\!] \Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
   assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
   assumes *II*: !!σ *it*. $[\![$
    $it \subseteq dom\ (\alpha\ m)$;
    $it \neq \{\}$;
    $\neg\ c\ \sigma$;
    I it σ;
    $\forall k{\in}it.\ \forall j{\in}dom\ (\alpha\ m) - it.\ j{\geq}k$
   $]\!] \Longrightarrow P\ \sigma$
   shows *P* (*reverse-iterateoi m c f* $\sigma 0$)
  using *map-iterator-rev-linord-rule-P* [*OF reverse-iterateoi-rule*, *of m I* $\sigma 0$ *c f*
P] *assms*
  by *simp*

lemma *reverse-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. ⟦
$k \in it$;
$\forall j \in it.\ k \geq j$;
$\forall j \in dom\ (\alpha\ m) - it.\ j \geq k$;
$\alpha\ m\ k = Some\ v$;
$it \subseteq dom\ (\alpha\ m)$;
$I\ it\ \sigma$
⟧ $\Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
shows P (*reverse-iterateoi m* (λ-. *True*) *f* $\sigma 0$)
using *map-iterator-rev-linord-rule-P*[*OF reverse-iterateoi-rule*, *of m I* $\sigma 0$ λ-. *True f P*] *assms*
by *simp*
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
fix *m*
assume *invar-m*: *invar m*
from *assms*(*1*)[*OF invar-m*] **show** *it-OK*: *map-iterator-rev-linord* (*ritoi m*) (α *m*) **.**

from *set-iterator-genord.finite-S0* [*OF it-OK*[*unfolded set-iterator-map-rev-linord-def*]]
show *finite* (*dom* (α *m*)) **by** (*simp add*: *finite-map-to-set*)
qed

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*) $P \neq \{\}$ ⟧ $\Longrightarrow$ *min s P* $\in$ *Some* ‘ *rel-of* (α *s*) *P*
⟦ *invar s*; $(k,v) \in$ *rel-of* (α *s*) *P* ⟧ $\Longrightarrow$ *fst* (*the* (*min s P*)) $\leq k$
⟦ *invar s*; *rel-of* (α *s*) $P = \{\}$ ⟧ $\Longrightarrow$ *min s P* = *None*
begin
lemma *minE*:
assumes *A*: *invar s*   *rel-of* (α *s*) $P \neq \{\}$
obtains *k v* **where**
min s P = *Some* (*k*,*v*) $(k,v) \in$ *rel-of* (α *s*) *P* $\forall (k',v') \in$ *rel-of* (α *s*) *P*. $k \leq k'$
proof −
from *min-correct*(*1*)[*OF A*] **have** *MIS*: *min s P* $\in$ *Some* ‘ *rel-of* (α *s*) *P* **.**
then obtain *k v* **where** *KV*: *min s P* = *Some* (*k*,*v*) $(k,v) \in$ *rel-of* (α *s*) *P*

```
      by auto
    show thesis
      apply (rule that[OF KV])
      apply (clarify)
      apply (drule min-correct(2)[OF ‹invar s›])
      apply (simp add: KV(1))
      done
  qed

  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'
    apply −
    apply (cases rel-of (α s) P = {})
    apply (drule (1) min-correct(3))
    apply simp
    apply (erule (1) minE)
    apply auto [1]
    apply (drule (1) min-correct(2))
    apply auto
    done

  lemma min-None:
    ⟦ invar s; min s P = None ⟧ ⟹ rel-of (α s) P = {}
    apply (cases rel-of (α s) P = {})
    apply simp
    apply (drule (1) min-correct(1))
    apply auto
    done

end

locale map-max = ordered-map +
  constrains α :: 's ⇒ 'u::linorder ⇀ 'v
  fixes max :: 's ⇒ ('u × 'v ⇒ bool) ⇒ ('u × '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 −
    from max-correct(1)[OF A] have MIS: max s P ∈ Some ‘ rel-of (α s) P .
```

```
    then obtain k v where KV: max s P = Some (k,v)    (k,v)∈rel-of (α s) P
      by auto
    show thesis
      apply (rule that[OF KV])
      apply (clarify)
      apply (drule max-correct(2)[OF ‹invar s›])
      apply (simp add: KV(1))
      done
  qed

  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'
    apply −
    apply (cases rel-of (α s) P = {})
    apply (drule (1) max-correct(3))
    apply simp
    apply (erule (1) maxE)
    apply auto [1]
    apply (drule (1) max-correct(2))
    apply auto
    done

  lemma max-None:
    ⟦ invar s; max s P = None ⟧ ⟹ rel-of (α s) P = {}
    apply (cases rel-of (α s) P = {})
    apply simp
    apply (drule (1) max-correct(1))
    apply auto
    done

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 :: $'k \Rightarrow 'v \Rightarrow 's \Rightarrow 's$
map-op-update-dj :: $'k \Rightarrow 'v \Rightarrow 's \Rightarrow 's$
map-op-delete :: $'k \Rightarrow 's \Rightarrow 's$
map-op-restrict :: $('k \times 'v \Rightarrow bool) \Rightarrow 's \Rightarrow 's$
map-op-add :: $'s \Rightarrow 's \Rightarrow 's$
map-op-add-dj :: $'s \Rightarrow 's \Rightarrow 's$
map-op-isEmpty :: $'s \Rightarrow bool$
map-op-isSng :: $'s \Rightarrow bool$
map-op-ball :: $'s \Rightarrow ('k \times 'v \Rightarrow bool) \Rightarrow bool$
map-op-bexists :: $'s \Rightarrow ('k \times 'v \Rightarrow bool) \Rightarrow bool$
map-op-size :: $'s \Rightarrow nat$
map-op-size-abort :: $nat \Rightarrow 's \Rightarrow nat$
map-op-sel :: $'s \Rightarrow ('k \times 'v \Rightarrow bool) \Rightarrow ('k\times'v)$ *option* — Version without mapping
map-op-to-list :: $'s \Rightarrow ('k\times'v)$ *list*
map-op-to-map :: $('k\times'v)$ *list* $\Rightarrow 's$

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 α :: $'s \Rightarrow 'x$ *set*
  — Invariant
  fixes *invar* :: $'s \Rightarrow$ *bool*

2.2.1 Basic Set Functions

Empty set

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

Membership Query

locale *set-memb* = *set* +
  constrains α :: $'s \Rightarrow 'x$ *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 $ins\text{-}dj :: 'x \Rightarrow 's \Rightarrow 's$
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* (α *s*)

Size

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

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

Singleton sets

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

locale *set-isSng* = *set* +
  constrains α :: $'s \Rightarrow 'x$ *set*
  fixes *isSng* :: $'s \Rightarrow$ *bool*
  assumes *isSng-correct*:
    invar s $\Longrightarrow$ *isSng s* $\longleftrightarrow$ ($\exists$ *e*. α *s* = $\{e\}$)
begin
  lemma *isSng-correct-exists1* :
    invar s $\Longrightarrow$ (*isSng s* $\longleftrightarrow$ ($\exists$!*e*. (*e* $\in$ α *s*)))
  by (*auto simp add*: *isSng-correct*)

  lemma *isSng-correct-card* :
    invar s $\Longrightarrow$ (*isSng s* $\longleftrightarrow$ (*card* (α *s*) = *1*))
  by (*auto simp add*: *isSng-correct card-Suc-eq*)
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 α :: $'s \Rightarrow 'x$ *set*
  fixes *iteratei* :: $'s \Rightarrow ('x,\ '\sigma)$ *set-iterator*

  assumes *iteratei-rule*: *invar* $S \Longrightarrow$ *set-iterator* (*iteratei* S) (α S)
begin
  lemma *iteratei-rule-P*:
    $\llbracket$
      invar S;
      $I\ (\alpha\ S)\ \sigma 0$;
      $!!x\ it\ \sigma.\ \llbracket\ c\ \sigma;\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$;
      $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$;
      $!!\sigma\ it.\ \llbracket\ it \subseteq \alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma\ \rrbracket \Longrightarrow P\ \sigma$
    $\rrbracket \Longrightarrow P$ (*iteratei* $S\ c\ f\ \sigma 0$)
   apply (*rule set-iterator-rule-P* [*OF iteratei-rule*, *of* $S\ I\ \sigma 0\ c\ f\ P$])
   apply *simp-all*
  done

  lemma *iteratei-rule-insert-P*:
    $\llbracket$
      invar S;
      $I\ \{\}\ \sigma 0$;
      $!!x\ it\ \sigma.\ \llbracket\ c\ \sigma;\ x \in \alpha\ S - it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow I\ (insert\ x\ it)\ (f\ x\ \sigma)$;
      $!!\sigma.\ I\ (\alpha\ S)\ \sigma \Longrightarrow P\ \sigma$;
      $!!\sigma\ it.\ \llbracket\ it \subseteq \alpha\ S;\ it \neq \alpha\ S;\ \neg\ c\ \sigma;\ I\ it\ \sigma\ \rrbracket \Longrightarrow P\ \sigma$
    $\rrbracket \Longrightarrow P$ (*iteratei* $S\ c\ f\ \sigma 0$)
    apply (*rule set-iterator-rule-insert-P* [*OF iteratei-rule*, *of* $S\ I\ \sigma 0\ c\ f\ P$])
    apply *simp-all*
  done

Versions without break condition.

  lemma *iterate-rule-P*:
    $\llbracket$
      invar S;
      $I\ (\alpha\ S)\ \sigma 0$;
      $!!x\ it\ \sigma.\ \llbracket\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$;
      $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
    $\rrbracket \Longrightarrow P$ (*iteratei* S (λ-. *True*) $f\ \sigma 0$)
   apply (*rule set-iterator-no-cond-rule-P* [*OF iteratei-rule*, *of* $S\ I\ \sigma 0\ f\ P$])
   apply *simp-all*
  done

  lemma *iterate-rule-insert-P*:
    $\llbracket$
      invar S;
      $I\ \{\}\ \sigma 0$;
      $!!x\ it\ \sigma.\ \llbracket\ x \in \alpha\ S - it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ \rrbracket \Longrightarrow I\ (insert\ x\ it)\ (f\ x\ \sigma)$;
      $!!\sigma.\ I\ (\alpha\ S)\ \sigma \Longrightarrow P\ \sigma$
    $\rrbracket \Longrightarrow P$ (*iteratei* S (λ-. *True*) $f\ \sigma 0$)

apply (*rule set-iterator-no-cond-rule-insert-P* [*OF iteratei-rule*, *of* $S\ I\ \sigma 0\ f\ P$])
apply *simp-all*
done
end

lemma *set-iteratei-I* :
assumes $\bigwedge s.$ *invar s* $\Longrightarrow$ *set-iterator* (*iti s*) (α *s*)
shows *set-iteratei* α *invar iti*
proof
fix *s*
assume *invar-s*: *invar s*
from *assms*(*1*)[*OF invar-s*] **show** *it-OK*: *set-iterator* (*iti s*) (α *s*) **.**

from *set-iterator-genord.finite-S0* [*OF it-OK*[*unfolded set-iterator-def*]]
show *finite* (α *s*) **.**
qed

2.2.3 More Set Operations

Copy

locale *set-copy* = *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 *copy* :: *'s1* $\Rightarrow$ *'s2*
assumes *copy-correct*:
invar1 s1 $\Longrightarrow$ $\alpha 2$ (*copy s1*) = $\alpha 1$ *s1*
invar1 s1 $\Longrightarrow$ *invar2* (*copy s1*)

Union

locale *set-union* = *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* :: *'s1* $\Rightarrow$ *'s2* $\Rightarrow$ *'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*:

$[\![invar1\ s1;\ invar2\ s2;\ \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}]\!] \Longrightarrow \alpha 3\ (union\text{-}dj\ s1\ s2) = \alpha 1\ s1 \cup \alpha 2\ s2$
$[\![invar1\ s1;\ invar2\ s2;\ \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}]\!] \Longrightarrow invar3\ (union\text{-}dj\ s1\ s2)$

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*:
$\forall s1 \in set\ l.\ invar1\ s1 \Longrightarrow \alpha 2\ (union\text{-}list\ l) = \bigcup \{\alpha 1\ s1\ |s1.\ s1 \in set\ l\}$
$\forall s1 \in set\ l.\ invar1\ s1 \Longrightarrow invar2\ (union\text{-}list\ l)$

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 $image\text{-}filter :: ('a \Rightarrow 'b\ option) \Rightarrow 's1 \Rightarrow 's2$
assumes *image-filter-correct-aux*:
$invar1\ s \Longrightarrow \alpha 2\ (image\text{-}filter\ f\ s) = \{\ b\ .\ \exists\, a{\in}\alpha 1\ s.\ f\ a = Some\ b\ \}$
$invar1\ s \Longrightarrow invar2\ (image\text{-}filter\ f\ s)$
begin
— This special form will be checked first by the simplifier:
lemma *image-filter-correct-aux2*:
$invar1\ s \Longrightarrow$
$\alpha 2\ (image\text{-}filter\ (\lambda x.\ if\ P\ x\ then\ (Some\ (f\ x))\ else\ None)\ s)$
$= f\ `\ \{x{\in}\alpha 1\ s.\ P\ x\}$
by (*auto simp add*: *image-filter-correct-aux*)

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 $inj\text{-}image\text{-}filter :: ('a \Rightarrow 'b\ option) \Rightarrow 's1 \Rightarrow 's2$
assumes *inj-image-filter-correct*:
$[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s \cap dom\ f)]\!] \Longrightarrow \alpha 2\ (inj\text{-}image\text{-}filter\ f\ s) = \{\ b\ .\ \exists\, a{\in}\alpha 1\ s.\ f\ a = Some\ b\ \}$
$[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s \cap dom\ f)]\!] \Longrightarrow invar2\ (inj\text{-}image\text{-}filter\ f\ s)$

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*:
    $[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s)]\!] \Longrightarrow \alpha 2\ (inj\text{-}image\ f\ s) = f\text{`}\alpha 1\ s$
    $[\![invar1\ s;\ inj\text{-}on\ f\ (\alpha 1\ s)]\!] \Longrightarrow invar2\ (inj\text{-}image\ f\ s)$

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*:
   $[\![\ invar1\ s;\ !!x.\ x{\in}\alpha 1\ s \Longrightarrow invar2\ (f\,x)\ ]\!] \Longrightarrow \alpha 3\ (Union\text{-}image\,f\,s) = \bigcup \alpha 2\text{`}f\text{`}\alpha 1$
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*:
$[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \Longrightarrow \alpha1\ s1 \cap \alpha2\ s2 = \{\}$
$[\![invar1\ s1;\ invar2\ s2;\ disjoint\text{-}witness\ s1\ s2 = Some\ a]\!]$
$\Longrightarrow a{\in}\alpha1\ s1 \cap \alpha2\ s2$
begin
lemma *disjoint-witness-None*: $[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \longleftrightarrow \alpha1\ s1 \cap \alpha2\ s2 = \{\}$
by (*case-tac disjoint-witness s1 s2*)
(*auto dest*: *disjoint-witness-correct*)

lemma *disjoint-witnessI*: $[\![$
$invar1\ s1;$
$invar2\ s2;$
$\alpha1\ s1 \cap \alpha2\ s2 \neq \{\};$
$!!a.\ [\![disjoint\text{-}witness\ s1\ s2 = Some\ a]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*case-tac disjoint-witness s1 s2*)
(*auto dest*: *disjoint-witness-correct*)

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*:
$[\![\ invar\ s;\ sel\ s\ f = Some\ r;\ !!x.\ [\![x{\in}\alpha\ s;\ f\ x = Some\ r]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
apply (*cases* $\exists\, x{\in}\alpha\ s.\ \exists\, r.\ f\ x = Some\ r$)
apply (*safe*)
apply (*erule-tac f=f* **and** *x=x* **in** *selE*)
apply *auto*
apply (*drule* (*1*) *selI*)
apply *simp*
done

lemma *sel-noneD*:
$[\![\ invar\ s;\ sel\ s\ f = None;\ x{\in}\alpha\ s\ ]\!] \Longrightarrow f\ x = None$
apply (*cases* $\exists\, x{\in}\alpha\ s.\ \exists\, r.\ f\ x = Some\ r$)
apply (*safe*)

apply ($erule\text{-}tac\ f{=}f$ **and** $x{=}xa$ **in** $selE$)
apply $auto$
done
end

— Selection of element (without mapping)
locale $set\text{-}sel' = set\ +$
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes $sel' :: 's \Rightarrow ('x \Rightarrow bool) \Rightarrow 'x\ option$
assumes $sel'E$:
$[\![\ invar\ s;\ x{\in}\alpha\ s;\ P\ x;$
$!!x.\ [\![sel'\ s\ P = Some\ x;\ x{\in}\alpha\ s;\ P\ x\ ]\!] \Longrightarrow Q$
$]\!] \Longrightarrow Q$
assumes $sel'I$: $[\![invar\ s;\ \forall\,x{\in}\alpha\ s.\ \neg\ P\ x\ ]\!] \Longrightarrow sel'\ s\ P = None$
begin

lemma $sel'\text{-}someD$:
$[\![\ invar\ s;\ sel'\ s\ P = Some\ x\ ]\!] \Longrightarrow x{\in}\alpha\ s \wedge P\ x$
apply ($cases\ \exists\,x{\in}\alpha\ s.\ P\ x$)
apply ($safe$)
apply ($erule\text{-}tac\ P{=}P$ **and** $x{=}xa$ **in** $sel'E$)
apply $auto$
apply ($erule\text{-}tac\ P{=}P$ **and** $x{=}xa$ **in** $sel'E$)
apply $auto$
apply ($drule\ (1)\ sel'I$)
apply $simp$
apply ($drule\ (1)\ sel'I$)
apply $simp$
done

lemma $sel'\text{-}noneD$:
$[\![\ invar\ s;\ sel'\ s\ P = None;\ x{\in}\alpha\ s\ ]\!] \Longrightarrow \neg P\ x$
apply ($cases\ \exists\,x{\in}\alpha\ s.\ P\ x$)
apply ($safe$)
apply ($erule\text{-}tac\ P{=}P$ **and** $x{=}xa$ **in** $sel'E$)
apply $auto$
done
end

Conversion of Set to List

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

Conversion of List to Set

locale *list-to-set* = *set* +
  constrains α :: $'s \Rightarrow 'x$ *set*
  fixes *to-set* :: $'x$ *list* $\Rightarrow 's$
  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 :: $'s \Rightarrow ('u$::*linorder*) *set* **and** *invar*

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

Ordered Iteration

  locale *set-iterateoi* = *ordered-finite-set* α *invar*
    for α :: $'s \Rightarrow ('u$::*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* σ. $[\![$
        c σ;
        $k \in it$;
        $\forall j{\in}it.\ k{\le}j$;
        $\forall j{\in}\alpha\ m - it.\ j{\le}k$;
        $it \subseteq \alpha\ m$;
        I it σ
      $]\!] \Longrightarrow I\ (it - \{k\})\ (f\ k\ \sigma)$
      assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
      assumes *II*: !!σ *it*. $[\![$
        $it \subseteq \alpha\ m$;
        $it \neq \{\}$;
        $\neg\ c\ \sigma$;
        I it σ;
        $\forall k{\in}it.\ \forall j{\in}\alpha\ m - it.\ j{\le}k$
      $]\!] \Longrightarrow P\ \sigma$
      shows *P* (*iterateoi m c f* $\sigma 0$)
    using *set-iterator-linord-rule-P* [*OF iterateoi-rule*, *OF MINV*, *of I* $\sigma 0$ *c f P*,
      OF I0 - IF] *IP II*
    by *simp*

    lemma *iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:

assumes *MINV*: *invar m*
assumes *I0*: *I* ((α *m*)) *σ0*
assumes *IP*: !!*k it σ*. ⟦ *k* ∈ *it*; ∀ *j*∈*it*. *k*≤*j*; ∀ *j*∈(α *m*) − *it*. *j*≤*k*; *it* ⊆ (α *m*); *I it σ* ⟧
⟹ *I* (*it* − {*k*}) (*f k σ*)
assumes *IF*: !!*σ*. *I* {} *σ* ⟹ *P σ*
shows *P* (*iterateoi m* (λ-. *True*) *f σ0*)
apply (*rule iterateoi-rule-P* [**where** *I* = *I*])
apply (*simp-all add*: *assms*)
done
end

lemma *set-iterateoi-I* :
assumes ⋀*s*. *invar s* ⟹ *set-iterator-linord* (*itoi s*) (α *s*)
shows *set-iterateoi* α *invar itoi*
proof
fix *s*
assume *invar-s*: *invar s*
from *assms*(*1*)[*OF invar-s*] **show** *it-OK*: *set-iterator-linord* (*itoi s*) (α *s*) **.**

from *set-iterator-genord.finite-S0* [*OF it-OK*[*unfolded set-iterator-linord-def*]]
show *finite* (α *s*) **by** *simp*
qed

locale *set-reverse-iterateoi* = *ordered-finite-set* α *invar*
for α :: *'s* ⇒ (*'u*::*linorder*) *set* **and** *invar*
\+
fixes *reverse-iterateoi* :: *'s* ⇒ (*'u*,*'σ*) *set-iterator*
assumes *reverse-iterateoi-rule*:
invar m ⟹ *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*)) *σ0*
assumes *IP*: !!*k it σ*. ⟦
c σ;
k ∈ *it*;
∀ *j*∈*it*. *k*≥*j*;
∀ *j*∈(α *m*) − *it*. *j*≥*k*;
it ⊆ (α *m*);
I it σ
⟧ ⟹ *I* (*it* − {*k*}) (*f k σ*)
assumes *IF*: !!*σ*. *I* {} *σ* ⟹ *P σ*
assumes *II*: !!*σ it*. ⟦
it ⊆ (α *m*);
it ≠ {};
¬ *c σ*;
I it σ;
∀ *k*∈*it*. ∀ *j*∈(α *m*) − *it*. *j*≥*k*

```
    ⟧ ⟹ P σ
  shows P (reverse-iterateoi m c f σ0)
  using set-iterator-rev-linord-rule-P [OF reverse-iterateoi-rule, OF MINV, of I
σ0 c f P,
     OF I0 - IF] IP II
  by simp

  lemma reverse-iterateo-rule-P[case-names minv inv0 inv-pres i-complete]:
    assumes MINV: invar m
    assumes I0: I ((α m)) σ0
    assumes IP: !!k it σ. ⟦
      k ∈ it;
      ∀ j∈it. k≥j;
      ∀ j∈ (α m) − it. j≥k;
      it ⊆ (α m);
      I it σ
    ⟧ ⟹ I (it − {k}) (f k σ)
    assumes IF: !!σ. I {} σ ⟹ P σ
  shows P (reverse-iterateoi m (λ-. True) f σ0)
    apply (rule reverse-iterateoi-rule-P [where I = I])
    apply (simp-all add: assms)
  done
end

lemma set-reverse-iterateoi-I :
assumes ⋀s. invar s ⟹ set-iterator-rev-linord (itoi s) (α s)
shows set-reverse-iterateoi α invar itoi
proof
  fix s
  assume invar-s: invar s
  from assms(1)[OF invar-s] show it-OK: set-iterator-rev-linord (itoi s) (α s) .

 from set-iterator-genord.finite-S0 [OF it-OK[unfolded set-iterator-rev-linord-def]]
  show finite (α s) by simp
qed
```

Minimal and Maximal Element

```
locale set-min = ordered-set +
  constrains α :: 's ⇒ 'u::linorder set
  fixes min :: 's ⇒ ('u ⇒ bool) ⇒ 'u option
  assumes min-correct:
    ⟦ invar s; x∈α s; P x ⟧ ⟹ min s P ∈ Some ‘ {x∈α s. P x}
    ⟦ invar s; x∈α s; P x ⟧ ⟹ (the (min s P)) ≤ x
    ⟦ invar s; {x∈α s. P x} = {} ⟧ ⟹ min s P = None
begin
 lemma minE:
   assumes A: invar s    x∈α s    P x
   obtains x' where
```

$min\ s\ P = Some\ x' \quad x'{\in}\alpha\ s \quad P\ x' \quad \forall x{\in}\alpha\ s.\ P\ x \longrightarrow x' \leq x$
proof −
from *min-correct(1)[of s x P, OF A]* **have** *MIS*: $min\ s\ P \in Some\ `\ \{x \in \alpha\ s.\ P\ x\}$ **.**
then obtain x' **where** *KV*: $min\ s\ P = Some\ x' \quad x'{\in}\alpha\ s \quad P\ x'$
by *auto*
show *thesis*
apply (*rule that[OF KV]*)
apply (*clarify*)
apply (*drule (1) min-correct(2)[OF ‹invar s›]*)
apply (*simp add*: *KV(1)*)
done
qed

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

lemma *min-Some*:
$[\![\ invar\ s;\ min\ s\ P = Some\ x\ ]\!] \Longrightarrow x{\in}\alpha\ s$
$[\![\ invar\ s;\ min\ s\ P = Some\ x\ ]\!] \Longrightarrow P\ x$
$[\![\ invar\ s;\ min\ s\ P = Some\ x;\ x'{\in}\alpha\ s;\ P\ x']\!] \Longrightarrow x{\leq}x'$
apply −
apply (*cases* $\{x \in \alpha\ s.\ P\ x\} = \{\}$)
apply (*drule (1) min-correct(3)*)
apply *simp*
apply *simp*
apply (*erule exE*)
apply *clarify*
apply (*drule (2) min-correct(1)[of s - P]*)
apply *auto [1]*

apply (*cases* $\{x \in \alpha\ s.\ P\ x\} = \{\}$)
apply (*drule (1) min-correct(3)*)
apply *simp*
apply *simp*
apply (*erule exE*)
apply *clarify*
apply (*drule (2) min-correct(1)[of s - P]*)
apply *auto [1]*

apply (*drule (2) min-correct(2)*[**where** *P=P*])
apply *auto*
done

lemma *min-None*:
$[\![\ invar\ s;\ min\ s\ P = None\ ]\!] \Longrightarrow \{x{\in}\alpha\ s.\ P\ x\} = \{\}$
apply (*cases* $\{x{\in}\alpha\ s.\ P\ x\} = \{\}$)
apply *simp*
apply *simp*
apply (*erule exE*)

```
    apply clarify
    apply (drule (2) min-correct(1)[where P=P])
    apply auto
    done

  end

  locale set-max = ordered-set +
    constrains α :: 's ⇒ 'u::linorder set
    fixes max :: 's ⇒ ('u ⇒ bool) ⇒ 'u option
    assumes max-correct:
      ⟦ invar s; x∈α s; P x ⟧ ⟹ max s P ∈ Some ‘ {x∈α s. P x}
      ⟦ invar s; x∈α s; P x ⟧ ⟹ the (max s P) ≥ x
      ⟦ invar s; {x∈α s. P x} = {} ⟧ ⟹ max s P = None
  begin
   lemma maxE:
    assumes A: invar s    x∈α s    P x
    obtains x' where
    max s P = Some x'    x'∈α s    P x'    ∀x∈α s. P x ⟶ x' ≥ x
   proof -
    from max-correct(1)[where P=P, OF A] have
      MIS: max s P ∈ Some ‘ {x∈α s. P x} .
    then obtain x' where KV: max s P = Some x'    x'∈ α s    P x'
      by auto
    show thesis
      apply (rule that[OF KV])
      apply (clarify)
      apply (drule (1) max-correct(2)[OF ‹invar s›])
      apply (simp add: KV(1))
      done
   qed

   lemmas maxI = max-correct(3)

   lemma max-Some:
    ⟦ invar s; max s P = Some x ⟧ ⟹ x∈α s
    ⟦ invar s; max s P = Some x ⟧ ⟹ P x
    ⟦ invar s; max s P = Some x; x'∈α s; P x'⟧ ⟹ x≥x'
    apply -
    apply (cases {x∈α s. P x} = {})
    apply (drule (1) max-correct(3))
    apply simp
    apply simp
    apply (erule exE)
    apply clarify
    apply (drule (2) max-correct(1)[of s - P])
    apply auto [1]

    apply (cases {x∈α s. P x} = {})
```

apply (*drule* (*1*) *max-correct*(*3*))
apply *simp*
apply *simp*
apply (*erule exE*)
apply *clarify*
apply (*drule* (*2*) *max-correct*(*1*)[*of s - P*])
apply *auto* [*1*]

apply (*drule* (*1*) *max-correct*(*2*)[**where** *P*=*P*])
apply *auto*
done

lemma *max-None*:
$[\![$ *invar s*; *max s P* = *None* $]\!] \Longrightarrow \{x{\in}\alpha\ s.\ P\ x\} = \{\}$
apply (*cases* $\{x{\in}\alpha\ s.\ P\ x\} = \{\}$)
apply *simp*
apply *simp*
apply (*erule exE*)
apply *clarify*
apply (*drule* (*1*) *max-correct*(*1*)[**where** *P*=*P*])
apply *auto*
done

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 :: $'s \Rightarrow 'u$::*linorder set* **and** *invar to-list* +
assumes *to-list-sorted*: *invar m* $\Longrightarrow$ *sorted* (*to-list m*)

2.2.6 Record Based Interface

record ($'x$,$'s$) *set-ops* =
set-op-α :: $'s \Rightarrow 'x$ *set*
set-op-invar :: $'s \Rightarrow$ *bool*
set-op-empty :: *unit* $\Rightarrow 's$
set-op-sng :: $'x \Rightarrow 's$
set-op-memb :: $'x \Rightarrow 's \Rightarrow$ *bool*
set-op-ins :: $'x \Rightarrow 's \Rightarrow 's$
set-op-ins-dj :: $'x \Rightarrow 's \Rightarrow 's$
set-op-delete :: $'x \Rightarrow 's \Rightarrow 's$
set-op-isEmpty :: $'s \Rightarrow$ *bool*
set-op-isSng :: $'s \Rightarrow$ *bool*
set-op-ball :: $'s \Rightarrow ('x \Rightarrow$ *bool*$) \Rightarrow$ *bool*
set-op-bexists :: $'s \Rightarrow ('x \Rightarrow$ *bool*$) \Rightarrow$ *bool*
set-op-size :: $'s \Rightarrow$ *nat*
set-op-size-abort :: *nat* $\Rightarrow 's \Rightarrow$ *nat*
set-op-union :: $'s \Rightarrow 's \Rightarrow 's$
set-op-union-dj :: $'s \Rightarrow 's \Rightarrow 's$

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 :: $'s \Rightarrow ('x \Rightarrow bool) \Rightarrow 'x$ *option*
  set-op-max :: $'s \Rightarrow ('x \Rightarrow bool) \Rightarrow 'x$ *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* = $(\forall x \in S0.\ P\ x)$
unfolding *iterate-ball-def*
apply (*rule set-iterator-rule-P* [*OF it*,
      where $I = \lambda S\ \sigma.\ \sigma = (\forall x \in S0 - S.\ P\ x)$])
apply *auto*
done

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* = ($\exists\, x{\in}S0.\ P\ x$)
unfolding *iterate-bex-def*
apply (*rule set-iterator-rule-P* [*OF it*, **where** $I = \lambda S\ \sigma.\ \sigma = (\exists\, x{\in}S0{-}S.\ P\ x)$])
apply *auto*
done

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*
unfolding *iterate-to-list-def*
by (*induct xs rule*: *rev-induct*, *simp-all add*: *foldli-snoc*)

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*))
using *it* **unfolding** *set-iterator-genord-foldli-conv* **by** *auto*

lemma *iterate-to-list-correct* :
assumes *it*: *set-iterator it S0*
shows *set* (*iterate-to-list it*) = *S0* ∧ *distinct* (*iterate-to-list it*)
using *iterate-to-list-genord-correct* [*OF it*[*unfolded set-iterator-def*]]
by *simp*

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*))
using *it-OK* **unfolding** *set-iterator-linord-foldli-conv* **by** *auto*

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*)
using *it-OK* **unfolding** *set-iterator-rev-linord-foldli-conv* **by** *auto*

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*)))
using *it-OK* **unfolding** *map-iterator-linord-foldli-conv*
by *clarify* (*simp add*: *rev-map*[*symmetric*])

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*))
using *it-OK* **unfolding** *map-iterator-rev-linord-foldli-conv*
by *clarify* (*simp add*: *rev-map*[*symmetric*])

2.2.9 Size

lemma *set-iterator-finite* :
assumes *it*: *set-iterator it S0*
shows *finite S0*
using *set-iterator-genord.finite-S0* [*OF it*[*unfolded set-iterator-def*]] **.**

lemma *map-iterator-finite* :
assumes *it*: *map-iterator it m*
shows *finite* (*dom m*)
using *set-iterator-genord.finite-S0* [*OF it*[*unfolded set-iterator-def*]]
by (*simp add*: *finite-map-to-set*)

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*
unfolding *iterate-size-def*
apply (*rule-tac set-iterator-rule-insert-P* [*OF it*,
where *I* = λ*S* σ. σ = *card S* ∧ *finite S*])
apply *auto*
done

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*
unfolding *iterate-size-abort-def*
proof (*rule set-iterator-rule-insert-P* [*OF it*,
where *I* = λ*S* σ. σ = (*min n* (*card S*)) ∧ *finite S*])
case (*goal4* σ *S*)
assume *S* ⊆ *S0*   *S* ≠ *S0*   ¬ σ < *n*   σ = *min n* (*card S*) ∧ *finite S*

from ‹σ = *min n* (*card S*) ∧ *finite S*› ‹¬ σ < *n*›

have $\sigma = n \quad n \leq$ *card S*
by (*auto simp add*: *min-less-iff-disj*)

note *fin-S0* = *set-iterator-genord.finite-S0* [*OF it*[*unfolded set-iterator-def*]]
from *card-mono* [*OF fin-S0* ‹$S \subseteq S0$›] **have** *card S* $\leq$ *card S0* **.**

with ‹$\sigma = n$› ‹$n \leq$ *card S*› *fin-S0*
show $\sigma =$ *min n* (*card S0*) $\wedge$ *finite S0* **by** *simp*
qed *simp-all*

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* = {})
using *iterate-size-abort-correct*[*OF it*, *of 1*]
unfolding *iterate-is-empty-by-size-def*
by (*cases card S0*) *auto*

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

lemma *iterate-is-empty-correct* :
assumes *it*: *set-iterator it S0*
shows *iterate-is-empty it* = (*S0* = {})
unfolding *iterate-is-empty-def*
apply (*rule set-iterator-rule-insert-P* [*OF it*,
where $I = \lambda S\ \sigma.\ \sigma \longleftrightarrow S = \{\}$])
apply *auto*
done

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*)
using *iterate-size-abort-correct*[*OF it*, *of 2*]
unfolding *iterate-is-sng-def*
apply (*cases card S0*, *simp*, *rename-tac n'*)
apply (*case-tac n'*)
apply *auto*
done

2.2.12 Selection

definition *iterate-sel* **where**
iterate-sel $(it::('x,'y\ option)\ set\text{-}iterator)\ f = it\ (\lambda\sigma.\ \sigma = None)\ (\lambda x\ \sigma.\ f\ x)\ None$

lemma *iterate-sel-genord-correct* :
assumes *it-OK*: *set-iterator-genord it S0 R*
shows $iterate\text{-}sel\ it\ f = None \longleftrightarrow (\forall x{\in}S0.\ (f\ x = None))$
$iterate\text{-}sel\ it\ f = Some\ y \Longrightarrow (\exists x \in S0.\ f\ x = Some\ y \wedge (\forall x' \in S0{-}\{x\}.\ \forall y.\ f\ x' = Some\ y' \longrightarrow R\ x\ x'))$
proof −
show $iterate\text{-}sel\ it\ f = None \longleftrightarrow (\forall x{\in}S0.\ (f\ x = None))$
unfolding *iterate-sel-def*
apply (*rule-tac set-iterator-genord.iteratei-rule-insert-P* [*OF it-OK*,
where $I = \lambda S\ \sigma.\ (\sigma = None) \longleftrightarrow (\forall x{\in}S.\ (f\ x = None))$])
apply *auto*
done
next
have $iterate\text{-}sel\ it\ f = Some\ y \longrightarrow (\exists x \in S0.\ f\ x = Some\ y \wedge (\forall x' \in S0{-}\{x\}.\ \forall y'.\ f\ x' = Some\ y' \longrightarrow R\ x\ x'))$
unfolding *iterate-sel-def*
apply (*rule-tac set-iterator-genord.iteratei-rule-insert-P* [*OF it-OK*,
where $I = \lambda S\ \sigma.\ (\forall y.\ \sigma = Some\ y \longrightarrow (\exists x \in S.\ f\ x = Some\ y \wedge (\forall x' \in S{-}\{x\}.\forall y'.\ f\ x' = Some\ y' \longrightarrow R\ x\ x'))) \wedge$
$((\sigma = None) \longleftrightarrow (\forall x{\in}S.\ f\ x = None))$])
apply *simp*
apply (*auto simp add*: *Bex-def subset-iff Ball-def*)
apply *metis*
done
moreover assume $iterate\text{-}sel\ it\ f = Some\ y$
finally show $(\exists x \in S0.\ f\ x = Some\ y \wedge (\forall x' \in S0{-}\{x\}.\ \forall y.\ f\ x' = Some\ y' \longrightarrow R\ x\ x'))$ **by** *blast*
qed

definition *iterate-sel-no-map* **where**
$iterate\text{-}sel\text{-}no\text{-}map\ it\ P = iterate\text{-}sel\ it\ (\lambda x.\ if\ P\ x\ then\ Some\ x\ else\ None)$
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 $iterate\text{-}sel\text{-}no\text{-}map\ it\ P = None \longleftrightarrow (\forall x{\in}S0.\ \neg(P\ x))$
$iterate\text{-}sel\text{-}no\text{-}map\ it\ P = Some\ x \Longrightarrow (x \in S0 \wedge P\ x \wedge (\forall x' \in S0{-}\{x\}.\ P\ x' \longrightarrow R\ x\ x'))$
unfolding *iterate-sel-no-map-def*
using *iterate-sel-genord-correct*[*OF it-OK*, *of* $\lambda x.\ if\ P\ x\ then\ Some\ x\ else\ None$]
apply (*simp-all add*: *Bex-def*)
apply (*metis option.inject option.simps*(*2*))

done

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 –
note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-def*], *of P*]
thus *iterate-sel-no-map it P = None* $\longleftrightarrow$ $(\forall x{\in}S0.\ \neg(P\ x))$
iterate-sel-no-map it P = Some x $\Longrightarrow x \in S0 \wedge P\ x$
by *simp-all*
qed

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 $\Longrightarrow (x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x \le x'))$
proof –
note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-linord-def*], *of P*]
thus *iterate-sel-no-map it P = None* $\longleftrightarrow$ $(\forall x{\in}S0.\ \neg(P\ x))$
iterate-sel-no-map it P = Some x $\Longrightarrow (x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x \le x'))$
by *auto*
qed

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 $\Longrightarrow (x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x' \le x))$
proof –
note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-rev-linord-def*], *of P*]
thus *iterate-sel-no-map it P = None* $\longleftrightarrow$ $(\forall x{\in}S0.\ \neg(P\ x))$
iterate-sel-no-map it P = Some x $\Longrightarrow (x \in S0 \wedge P\ x \wedge (\forall x'{\in}S0.\ P\ x' \longrightarrow x' \le x))$
by *auto*
qed

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

note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-def*], *of P*]
thus *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*))
by (*auto simp add*: *map-to-set-def*)
qed

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 –
note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-map-linord-def*], *of P*]
thus *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'*))
apply (*auto simp add*: *map-to-set-def Ball-def*)
apply (*metis order-refl*)
done
qed

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* $\longleftrightarrow$ ($\forall$ *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 –
note *iterate-sel-no-map-genord-correct* [*OF it-OK*[*unfolded set-iterator-map-rev-linord-def*], *of P*]
thus *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*))
apply (*auto simp add*: *map-to-set-def Ball-def*)
apply (*metis order-refl*)
done
qed

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*: $\bigwedge$*l. sorted-by-rel R* (*sort-fun l*) $\wedge$ *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 −
def *l* $\equiv$ *iterate-to-list it*
have *l-props*: *set l* = *S0* *distinct l*
using *iterate-to-list-correct* [*OF it-OK*, *folded l-def*] **by** *simp-all*

with *sort-fun-OK*[*of l*] **have** *sort-l-props*:
sorted-by-rel R (*sort-fun l*)
set (*sort-fun l*) = *S0* *distinct* (*sort-fun l*)
apply (*simp-all*)
apply (*metis set-of-multiset-of*)
apply (*metis distinct-count-atmost-1 set-of-multiset-of*)
done

show *?thesis*
apply (*rule set-iterator-genord-I*[*of sort-fun l*])
apply (*simp-all add*: *sort-l-props iterator-to-ordered-iterator-def l-def*[*symmetric*])
done
qed

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* : $\bigwedge$*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-quicksort R it*) *S0 R*
unfolding *iterator-to-ordered-iterator-quicksort-def*
apply (*rule iterator-to-ordered-iterator-correct* [*OF - it-OK*])
apply (*simp-all add*: *sorted-by-rel-quicksort-by-rel*[*OF lin trans-R*])

done

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*
unfolding *iterator-to-ordered-iterator-mergesort-def*
apply (*rule iterator-to-ordered-iterator-correct [OF - it-OK]*)
apply (*simp-all add: sorted-by-rel-mergesort-by-rel[OF lin trans-R]*)
done

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* ⇒ *'x list*
 fixes *isEmpty* :: *'s* ⇒ *bool*
 assumes *isEmpty-correct*:
  invar s ⟹ *isEmpty s* ⟷ α *s* = []

locale *list-iteratei* = *list* +
 constrains α :: *'s* ⇒ *'x list*
 fixes *iteratei* :: *'s* ⇒ (*'x*,*'*σ) *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*))
  by (*simp add*: *iteratei-correct set-iterator-foldli-correct*)
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*))
 by (*simp add*: *list-iteratei-def set-iterator-linord-foldli-correct*)

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*))
 by (*simp add*: *list-iteratei-def set-iterator-rev-linord-foldli-correct*)

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*))
  by (*simp add*: *reverse-iteratei-correct set-iterator-foldri-correct*)
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*))
 by (*simp add*: *list-reverse-iteratei-def set-iterator-linord-foldri-correct*)

lemma *list-reverse-iteratei-iteratei-rev-linord-rule*:

list-reverse-iteratei α *invar iteratei* $\Longrightarrow$ *invar s* $\Longrightarrow$ *distinct* $(\alpha\ s) \Longrightarrow$ *sorted* $(\alpha$ $s) \Longrightarrow$
set-iterator-rev-linord (*iteratei s*) (*set* $(\alpha\ s)$)
by (*simp add*: *list-reverse-iteratei-def set-iterator-rev-linord-foldri-correct*)

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

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 −
from *pop-correct*(*1*,*2*,*3*)[*OF I*] **have**
C: *fst* (*pop s*) = *hd* $(\alpha\ s)$
α (*snd* (*pop s*)) = *tl* $(\alpha\ s)$
invar (*snd* (*pop s*)) **.**
from *that*[*of snd* (*pop s*), *OF* - *C*(*3*,*2*), *folded C*(*1*)] **show** *thesis*
by *simp*
qed

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* $(\alpha\ F)$
by (*simp add*: *tail-def pop-correct*)

lemma *head-correct*: $[\![invar\ F;\ \alpha\ F \neq []]\!] \Longrightarrow (head\ F) = hd\ (\alpha\ F)$
by (*simp add*: *head-def pop-correct*)

lemma *pop-split*: $pop\ F = (head\ F,\ tail\ F)$
apply (*cases pop F*)
apply (*simp add*: *head-def tail-def*)
done

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*:
$invar\ s \Longrightarrow \alpha\ (enqueue\ x\ s) = \alpha\ s\ @\ [x]$
$invar\ s \Longrightarrow invar\ (enqueue\ x\ s)$

— 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 $\alpha :: 's \Rightarrow 'x\ list$
fixes $get :: 's \Rightarrow nat \Rightarrow 'x$
assumes *get-correct*:
$[\![invar\ s;\ i < length\ (\alpha\ s)]\!] \Longrightarrow get\ s\ i = \alpha\ s\ !\ i$

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

assumes *set-correct*:
$[\![invar\ s;\ i{<}length\ (\alpha\ s)]\!] \Longrightarrow \alpha\ (set\ s\ i\ x) = \alpha\ s\ [i := x]$
$[\![invar\ s;\ i{<}length\ (\alpha\ s)]\!] \Longrightarrow invar\ (set\ s\ i\ x)$

— 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 $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
  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 $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
  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 $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
  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 $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
  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 $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
  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*:
    $[\![invar\ s;$
      $\forall\, a\ b.\ p\ a \longrightarrow p\ (a + b);$

$\neg\ p\ i$;
$p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ s)))$;
$(\mathit{splits}\ p\ i\ s) = (l,\ (e,a),\ r)\rrbracket$
$\Longrightarrow$
$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)\ \wedge$
$\neg\ p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ l)))\ \wedge$
$p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ l)) + a)\ \wedge$
$\mathit{invar}\ l\ \wedge$
$\mathit{invar}\ r$

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: $p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ s)))$
obtains *l e a r* **where**
$(\mathit{splits}\ p\ i\ s) = (l,\ (e,a),\ r)$
$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)$
$\neg\ p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ l)))$
$p\ (i + \mathit{listsum}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ l)) + a)$
invar l
invar r
using *assms*
apply (*cases splits p i s*)
apply (*case-tac b*)
apply (*drule-tac i = i* **and** *p = p*
and *l = a* **and** *r = c* **and** *e = aa* **and** *a = ba* **in** *splits-correct*)
apply (*simp-all*)
apply *blast*
done
end

2.4.3 Record Based Interface

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

```
  alist-op-annot :: 's ⇒ 'a
  alist-op-splits :: ('a ⇒ bool) ⇒ 'a ⇒ 's ⇒ ('s × ('e × 'a) × 's)

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

Meld Two Queues

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

Delete Minimal Element

Delete the same element that find will return

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

2.5.2 Record based interface

record $('e,\ 'a,\ 'p)$ *prio-ops* =
prio-op-α :: $'p \Rightarrow ('e \times 'a)\ multiset$
prio-op-invar :: $'p \Rightarrow bool$
prio-op-empty :: $unit \Rightarrow 'p$
prio-op-isEmpty :: $'p \Rightarrow bool$
prio-op-insert :: $'e \Rightarrow 'a \Rightarrow 'p \Rightarrow 'p$
prio-op-find :: $'p \Rightarrow 'e \times 'a$
prio-op-delete :: $'p \Rightarrow 'p$
prio-op-meld :: $'p \Rightarrow 'p \Rightarrow 'p$

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 α :: $'s \Rightarrow ('e \rightharpoonup 'a::linorder)$
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 α :: $'s \Rightarrow ('e \rightharpoonup 'a::linorder)$
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**
$pop\ s = (e,\ a,\ s')$
$invar\ s'$
$\alpha\ s' = (\alpha\ s)(e := None)$
$(\alpha\ s)\ e = Some\ a$
$(\forall y \in ran\ (\alpha\ s).\ a \leq y)$
using *assms*
apply (*cases pop s*)
apply (*drule* (*2*) *pop-correct*)
apply *blast*
done

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-α :: $'s \Rightarrow ('e \rightharpoonup 'a)$
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 :: $'s \Rightarrow ('e \times 'a \times 's)$
upr-prio :: $'s \Rightarrow 'e \Rightarrow 'a\ option$

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**
  $iterate\text{-}add\text{-}to\text{-}set\ s\ ins\ (it{::}('x,'x\text{-}set)\ set\text{-}iterator) =$
   $it\ (\lambda\text{-}.\ True)\ (\lambda x\ \sigma.\ ins\ x\ \sigma)\ s$

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* $\cup$ α *s* $\wedge$ *invar* (*iterate-add-to-set s ins it*)
unfolding *iterate-add-to-set-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
  where *?I*=$\lambda S\ \sigma.\ \alpha\ \sigma = S \cup \alpha\ s \wedge invar\ \sigma$])
apply (*insert ins-OK s-OK*)
apply (*simp-all add*: *set-ins-def*)
done

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* $\cap$ α *s* = {}
shows α (*iterate-add-to-set s ins-dj it*) = *S0* $\cup$ α *s* $\wedge$ *invar* (*iterate-add-to-set s*

ins-dj it)
unfolding *iterate-add-to-set-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=$\lambda S\ \sigma.\ \alpha\ \sigma = S \cup \alpha\ s \wedge$ *invar* σ])
apply (*insert ins-dj-OK s-OK dj*)
apply (*simp-all add*: *set-ins-dj-def set-eq-iff*)
done

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*) ($\lambda x\ \sigma$. *ins-dj x* σ) (*emp* ())
unfolding *iterate-to-set-def iterate-add-to-set-def* **by** *simp*

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* $\wedge$ *invar* (*iterate-to-set emp ins-dj it*)
unfolding *iterate-to-set-def*
using *iterate-add-to-set-dj-correct* [*OF ins-dj-OK - it*, *of emp* ()] *emp-OK*
by (*simp add*: *set-empty-def*)

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*)) =
     $\{b\ .\ \exists\, a.\ a \in$ *S0* $\wedge f\ a =$ *Some* $b\} \cup \alpha\ s \wedge$
   invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
unfolding *iterate-add-to-set-def set-iterator-image-filter-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
   where *?I*=$\lambda S\ \sigma.\ \alpha\ \sigma = \{b\ .\ \exists\, a.\ a \in S \wedge f\ a =$ *Some* $b\} \cup \alpha\ s \wedge$ *invar* σ])
apply (*insert ins-OK s-OK*)
apply (*simp-all add*: *set-ins-def split*: *option.split*)
apply *auto*
done

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* . $\exists$ *a*. *a* $\in$ *S0* $\wedge$ *f a* = *Some b*} $\wedge$
invar (*iterate-to-set emp ins* (*set-iterator-image-filter f it*))
unfolding *iterate-to-set-def*
using *iterate-image-filter-add-to-set-correct* [*OF ins-OK - it*, *of emp* () *f*] *emp-OK*
by (*simp add*: *set-empty-def*)

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*. $\exists$ *x*. *x* $\in$ *S0* $\wedge$ *f x* = *Some y*} $\cap$ α *s* = {}
assumes *f-inj-on*: *inj-on f* (*S0* $\cap$ *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*} $\cup$ α *s* $\wedge$
invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
proof −
from *set-iterator-image-filter-correct* [*OF it f-inj-on*]
have *it-f*: *set-iterator* (*set-iterator-image-filter f it*)
{*y*. $\exists$ *x*. *x* $\in$ *S0* $\wedge$ *f x* = *Some y*} **by** *simp*

from *iterate-add-to-set-dj-correct* [*OF ins-dj-OK*, *OF s-OK it-f dj*]
show *?thesis* **by** *auto*
qed

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* $\cap$ *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*))
unfolding *iterate-to-set-def*
using *iterate-inj-image-filter-add-to-set-correct* [*OF ins-OK - it - f-inj-on*, *of emp* ()] *emp-OK*
by (*simp add*: *set-empty-def*)

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*) = α *s* − *S0* ∧ *invar* (*iterate-diff-set s del it*)
unfolding *iterate-diff-set-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=λ*S* σ. α σ = α *s* − *S* ∧ *invar* σ])
apply (*insert del-OK s-OK*)
apply (*auto simp add*: *set-delete-def set-eq-iff*)
done

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*) = α *m* ++ *M* ∧ *invar* (*iterate-add-to-map m upd it*)
using *assms*
unfolding *iterate-add-to-map-def*
apply (*rule-tac map-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=λ*d* σ. (α σ = α *m* ++ *M* |‘ *d*) ∧ *invar* σ])
apply (*simp-all add*: *map-update-def restrict-map-insert*)
done

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*) = α *m* ++ *M* ∧ *invar* (*iterate-add-to-map m upd it*)
using *assms*
unfolding *iterate-add-to-map-def*
apply (*rule-tac map-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=λ*d* σ. (α σ = α *m* ++ *M* |‘ *d*) ∧ *invar* σ])
apply (*simp-all add*: *map-update-dj-def restrict-map-insert set-eq-iff*)
done

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* ())
unfolding *iterate-to-map-def iterate-add-to-map-def* **by** *simp*

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*)
unfolding *iterate-to-map-def*
using *iterate-add-to-map-dj-correct* [*OF upd-dj-OK - it*, *of emp* ()] *emp-OK*
by (*simp add*: *map-empty-def*)

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*
apply (*unfold-locales*)
apply (*auto simp add*: *update-correct*)
done

3.1.2 Disjoint Add (by add)

lemma (**in** *map-add*) *add-dj-by-add*:
map-add-dj α *invar add*
apply (*unfold-locales*)
apply (*auto simp add*: *add-correct*)
done

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*
unfolding *it-add-def iterate-add-to-map-def* **by** *simp*

lemma *it-add-correct*:
fixes α :: $'s \Rightarrow 'u \rightharpoonup 'v$
assumes *iti*: *map-iteratei* α *invar iti*
assumes *upd-OK*: *map-update* α *invar update*
shows *map-add* α *invar* (*it-add update iti*)
unfolding *it-add-def*
proof
fix *m1 m2*
assume *invar m1* *invar m2*
with *map-iteratei.iteratei-rule*[*OF iti*, *of m2*]
have *iti-m2*: *map-iterator* (*iti m2*) (α *m2*) **by** *simp*

from *iterate-add-to-map-correct* [*OF upd-OK*, *OF* ‹*invar m1*› *iti-m2*]
show α (*iterate-add-to-map m1 update* (*iti m2*)) = α *m1* ++ α *m2*
invar (*iterate-add-to-map m1 update* (*iti m2*))
by *simp-all*
qed

3.1.4 Disjoint Add (by iterate)

lemma *it-add-dj-correct*:
fixes α :: $'s \Rightarrow 'u \rightharpoonup 'v$
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*)
unfolding *it-add-def*
proof
fix *m1 m2*
assume *invar m1* *invar m2* **and** *dom-dj*: *dom* (α *m1*) $\cap$ *dom* (α *m2*) = {}
with *map-iteratei.iteratei-rule*[*OF iti*, *OF* ‹*invar m2*›]
have *iti-m2*: *map-iterator* (*iti m2*) (α *m2*) **by** *simp*

from *iterate-add-to-map-dj-correct* [*OF upd-dj-OK*, *OF* ‹*invar m1*› *iti-m2*] *dom-dj*
show α (*iterate-add-to-map m1 update* (*iti m2*)) = α *m1* ++ α *m2*
invar (*iterate-add-to-map m1 update* (*iti m2*))
by *auto*
qed

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*
unfolding *iti-isEmpty-def iterate-is-empty-def* **by** *simp*

lemma *iti-isEmpty-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-isEmpty* α *invar* (*iti-isEmpty iti*)
unfolding *iti-isEmpty-def*
proof
fix *m*
assume *invar m*
with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

from *iterate-is-empty-correct*[*OF iti'*]
show *iterate-is-empty* (*iti m*) = (α *m* = *Map.empty*)
by (*simp add*: *map-to-set-empty-iff*)
qed

3.1.6 Iterators

Iteratei (by iterateoi)

lemma *iti-by-itoi*:
assumes *map-iterateoi* α *invar it*
shows *map-iteratei* α *invar it*
using *assms*
unfolding *map-iterateoi-def map-iteratei-def map-iteratei-axioms-def*
map-iterateoi-axioms-def set-iterator-map-linord-def
finite-map-def ordered-finite-map-def
by (*metis set-iterator-intro*)

Iteratei (by reverse_iterateoi)

lemma *iti-by-ritoi*:
assumes *map-reverse-iterateoi* α *invar it*
shows *map-iteratei* α *invar it*
using *assms*
unfolding *map-reverse-iterateoi-def map-iteratei-def map-iteratei-axioms-def*
map-reverse-iterateoi-axioms-def set-iterator-map-rev-linord-def
finite-map-def ordered-finite-map-def
by (*metis set-iterator-intro*)

3.1.7 Selection (by iteratei)

definition *iti-sel* **where**

iti-sel iti = (λ*m f*. *iterate-sel* (*iti m*) *f*)

lemma *iti-sel-code*[*code*] :
iti-sel iti m f = *iti m* ($\lambda\sigma$. σ = *None*) (λ*x* σ. *f x*) *None*
unfolding *iti-sel-def iterate-sel-def* **by** *simp*

lemma *iti-sel-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-sel* α *invar* (*iti-sel iti*)
proof –
{ **fix** *m f*
assume *invar m*
with *map-iteratei.iteratei-rule* [*OF iti*, *of m*]
have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*
note *iterate-sel-genord-correct*[*OF iti'*[*unfolded set-iterator-def*], *of f*]
}
thus *?thesis*
unfolding *map-sel-def iti-sel-def*
apply (*simp add*: *Bex-def Ball-def image-iff map-to-set-def*)
apply (*metis option.exhaust*)
done
qed

3.1.8 Map-free selection by selection

definition *iti-sel-no-map* **where**
iti-sel-no-map iti = (λ*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* ($\lambda\sigma$. σ = *None*) (λ*x* σ. *if P x then Some x else None*) *None*
unfolding *iti-sel-no-map-def iterate-sel-no-map-alt-def* **by** *simp*

lemma *iti-sel'-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-sel'* α *invar* (*iti-sel-no-map iti*)
proof –
{ **fix** *m P*
assume *invar m*
with *map-iteratei.iteratei-rule* [*OF iti*, *of m*]
have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*
note *iterate-sel-no-map-correct*[*OF iti'*, *of P*]
}
thus *?thesis*
unfolding *map-sel'-def iti-sel-no-map-def*
apply (*simp add*: *Bex-def Ball-def image-iff map-to-set-def*)
apply *clarify*
apply (*metis option.exhaust PairE*)
done

qed

3.1.9 Map-free selection by selection

definition *sel-sel′*
:: ($'s \Rightarrow ('k \times 'v \Rightarrow$ - *option*$) \Rightarrow$ - *option*$) \Rightarrow 's \Rightarrow ('k \times 'v \Rightarrow$ *bool*$) \Rightarrow ('k \times 'v)$ *option*
where *sel-sel′ sel s P* = *sel s* (λ*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 −
interpret *map-sel* α *invar sel* **by** *fact*

show *?thesis*
proof
case *goal1* **show** *?case*
apply (*rule selE*[*OF goal1*(*1*,*2*), **where** *f*=(λ*kv. if P kv then Some kv else None*)])
apply (*simp add*: *goal1*)
apply (*simp split*: *split-if-asm*)
apply (*fold sel-sel′-def*)
apply (*blast intro*: *goal1*(*4*))
done
next
case *goal2* **thus** *?case*
apply (*auto simp add*: *sel-sel′-def*)
apply (*drule selI*[**where** *f*=(λ*kv. if P kv then Some kv else None*)])
apply *auto*
done
qed
qed

3.1.10 Bounded Quantification (by sel)

definition *sel-ball*
:: ($'s \Rightarrow ('k \times 'v \Rightarrow$ *unit option*$) \rightharpoonup$ *unit*$) \Rightarrow 's \Rightarrow ('k \times 'v \Rightarrow$ *bool*$) \Rightarrow$ *bool*
where *sel-ball sel s P* == *sel s* (λ*kv. if* $\neg$*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 −
interpret *map-sel* α *invar sel* **by** *fact*

show *?thesis*
apply *unfold-locales*
apply (*unfold sel-ball-def*)
apply (*auto elim*: *sel-someE dest*: *sel-noneD split*: *split-if-asm*)

done
qed

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* ($\lambda kv.$ *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 −
interpret *map-sel* α *invar sel* **by** *fact*

show *?thesis*
apply *unfold-locales*
apply (*unfold sel-bexists-def*)
apply *auto*
apply (*force elim*!: *sel-someE split*: *split-if-asm*) []
apply (*erule-tac f*=($\lambda kv.$ *if P kv then Some* () *else None*) **in** *selE*)
apply *assumption*
apply *simp*
apply *simp*
done
qed

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* ($\lambda kv.$ $\neg$(*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 −
interpret *map-ball* α *invar ball* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
apply (*unfold neg-ball-bexists-def*)
apply (*simp add*: *ball-correct*)
done
qed

3.1.11 Bounded Quantification (by iterate)

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

lemma *iti-ball-code*[*code*] :

iti-ball iti m P = *iti m* (λ*c*. *c*) (λ*x* σ. *P x*) *True*
unfolding *iti-ball-def iterate-ball-def*[*abs-def*] *id-def* **by** *simp*

lemma *iti-ball-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-ball* α *invar* (*iti-ball iti*)
unfolding *iti-ball-def*
proof
fix *m P*
assume *invar m*
with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

from *iterate-ball-correct*[*OF iti'*, *of P*]
show *iterate-ball* (*iti m*) *P* = ($\forall$ *u v*. α *m u* = *Some v* $\longrightarrow$ *P* (*u*, *v*))
by (*simp add*: *map-to-set-def*)
qed

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

lemma *iti-bexists-code*[*code*] :
iti-bexists iti m P = *iti m* (λ*c*. $\neg$*c*) (λ*x* σ. *P x*) *False*
unfolding *iti-bexists-def iterate-bex-def*[*abs-def*] **by** *simp*

lemma *iti-bexists-correct*:
assumes *iti*: *map-iteratei* α *invar iti*
shows *map-bexists* α *invar* (*iti-bexists iti*)
unfolding *iti-bexists-def*
proof
fix *m P*
assume *invar m*
with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

from *iterate-bex-correct*[*OF iti'*, *of P*]
show *iterate-bex* (*iti m*) *P* = ($\exists$ *u v*. α *m u* = *Some v* $\wedge$ *P* (*u*, *v*))
by (*simp add*: *map-to-set-def*)
qed

3.1.12 Size (by iterate)

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

lemma *it-size-code*[*code*] :
it-size iti m = *iti m* (λ-. *True*) (λ*x n* . *Suc n*) *0*
unfolding *it-size-def iterate-size-def* **by** *simp*

lemma *it-size-correct*:
  assumes *iti*: *map-iteratei* α *invar iti*
  shows *map-size* α *invar* (*it-size iti*)
unfolding *it-size-def*
proof
  fix *m*
  assume *invar m*
  with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
  have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

  from *iterate-size-correct* [*OF iti'*]
  show *finite* (*dom* (α *m*))
      iterate-size (*iti m*) = *card* (*dom* (α *m*))
    by (*simp-all add*: *finite-map-to-set card-map-to-set*)
qed

3.1.13 Size with abort (by iterate)

definition *iti-size-abort* **where**
  iti-size-abort iti = (λn *m*. *iterate-size-abort* (*iti m*) *n*)

lemma *iti-size-abort-code*[*code*] :
  iti-size-abort iti n m = *iti m* ($\lambda\sigma$. $\sigma < n$) (λx. *Suc*) *0*
unfolding *iti-size-abort-def iterate-size-abort-def* **by** *simp*

lemma *iti-size-abort-correct*:
  assumes *iti*: *map-iteratei* α *invar iti*
  shows *map-size-abort* α *invar* (*iti-size-abort iti*)
unfolding *iti-size-abort-def*
proof
  fix *m n*
  assume *invar m*
  with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
  have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

  from *iterate-size-abort-correct* [*OF iti'*]
  show *finite* (*dom* (α *m*))
      iterate-size-abort (*iti m*) *n* = *min n* (*card* (*dom* (α *m*)))
    by (*simp-all add*: *finite-map-to-set card-map-to-set*)
qed

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* ($\lambda\sigma$. $\sigma < 2$) (λx. *Suc*) *0* = *1*)
unfolding *sza-isSng-def iterate-is-sng-def iterate-size-abort-def* **by** *simp*

lemma *sza-isSng-correct*:
  assumes *iti*: *map-iteratei* α *invar iti*
  shows *map-isSng* α *invar* (*sza-isSng iti*)
unfolding *sza-isSng-def*
proof
  fix *m*
  assume *invar m*
  with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
  have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*

  have *card* (*map-to-set* (α *m*)) = (*Suc 0*) $\longleftrightarrow$ ($\exists$ *kv*. *map-to-set* (α *m*) = {*kv*})
by (*simp add*: *card-Suc-eq*)
  also have ... $\longleftrightarrow$ ($\exists$ *kv*. α *m* = [*fst kv* $\mapsto$ *snd kv*]) **unfolding** *map-to-set-def*
    apply (*rule iff-exI*)
    apply (*simp add*: *set-eq-iff fun-eq-iff all-conj-distrib*)
    apply *auto*
    apply (*metis option.exhaust*)
    apply (*metis option.simps*(*1*,*2*))
  done
  finally have *card* (*map-to-set* (α *m*)) = (*Suc 0*) $\longleftrightarrow$ ($\exists$ *k v*. α *m* = [*k* $\mapsto$ *v*])
by *simp*

  with *iterate-is-sng-correct* [*OF iti'*]
  show *iterate-is-sng* (*iti m*) = ($\exists$ *k v*. α *m* = [*k* $\mapsto$ *v*])
   by *simp*
qed

3.1.15 Map to List (by iterate)

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

lemma *it-map-to-list-code*[*code*] :
  it-map-to-list iti m = *iti m* (λ-. *True*) *op* # []
unfolding *it-map-to-list-def iterate-to-list-def* **by** *simp*

lemma *it-map-to-list-correct*:
  assumes *iti*: *map-iteratei* α *invar iti*
  shows *map-to-list* α *invar* (*it-map-to-list iti*)
unfolding *it-map-to-list-def*
proof
  fix *m*
  assume *invar m*
  with *map-iteratei.iteratei-rule*[*OF iti*, *of m*]
  have *iti'*: *map-iterator* (*iti m*) (α *m*) **by** *simp*
  let *?l* = *iterate-to-list* (*iti m*)

  from *iterate-to-list-correct* [*OF iti'*]
  have *set-l-eq*: *set ?l* = *map-to-set* (α *m*) **and** *dist-l*: *distinct ?l* **by** *simp-all*

from *dist-l* **show** *dist-fst-l*: *distinct* (*map fst ?l*)
by (*simp add*: *distinct-map set-l-eq map-to-set-def inj-on-def*)

from *map-of-map-to-set*[*of ?l* α *m*, *OF dist-fst-l*] *set-l-eq*
show *map-of* (*iterate-to-list* (*iti m*)) = α *m* **by** *simp*
qed

3.1.16 List to Map

fun *gen-list-to-map-aux*
:: ($'k \Rightarrow 'v \Rightarrow 'm \Rightarrow 'm$) $\Rightarrow$ $'m \Rightarrow$ ($'k \times 'v$) *list* $\Rightarrow 'm$
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 −
interpret *map-empty* α *invar empt* **by** *fact*
interpret *map-update* α *invar upd* **by** *fact*

{ — Show a generalized lemma
fix *l accs*
have *invar accs* $\Longrightarrow$ α (*gen-list-to-map-aux upd accs l*) = α *accs* ++ *map-of* (*rev l*)
$\wedge$ *invar* (*gen-list-to-map-aux upd accs l*)
apply (*induct l arbitrary*: *accs*)
apply *simp*
apply (*case-tac a*)
apply (*simp add*: *update-correct*)
done
} **thus** *?thesis*
apply (*unfold-locales*)
apply (*unfold gen-list-to-map-def*)
apply (*auto simp add*: *empty-correct*)
done
qed

3.1.17 Singleton (by empty, update)

definition *eu-sng*
:: (*unit* $\Rightarrow 'm$) $\Rightarrow$ ($'k \Rightarrow 'v \Rightarrow 'm \Rightarrow 'm$) $\Rightarrow$ $'k \Rightarrow 'v \Rightarrow 'm$
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 −
  interpret *map-empty* α *invar empty* **by** *fact*
  interpret *map-update* α *invar update* **by** *fact*

  show *?thesis*
    apply (*unfold-locales*)
    apply (*simp-all add*: *update-correct empty-correct eu-sng-def*)
  done
qed

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*)
unfolding *iti-sel-no-map-def*
proof
  fix *m P*

  assume *invar m*
  with *map-iterateoi.iterateoi-rule* [*OF iti*, *of m*]
  have *iti'*: *map-iterator-linord* (*iti m*) (α *m*) **by** *simp*
  note *sel-correct* = *iterate-sel-no-map-map-linord-correct*[*OF iti'*, *of P*]

  { **assume** *rel-of* (α *m*) *P* $\neq$ {}
    with *sel-correct* **show** *iterate-sel-no-map* (*iti m*) *P* $\in$ *Some* ‘ *rel-of* (α *m*) *P*
      by (*auto simp add*: *image-iff rel-of-def*)
  }

  { **assume** *rel-of* (α *m*) *P* = {}
     with *sel-correct* **show** *iterate-sel-no-map* (*iti m*) *P* = *None*
      by (*auto simp add*: *image-iff rel-of-def*)
  }

  { **fix** *k v*
    assume (*k*, *v*) $\in$ *rel-of* (α *m*) *P*
    with *sel-correct* **show** *fst* (*the* (*iterate-sel-no-map* (*iti m*) *P*)) $\leq$ *k*
      by (*auto simp add*: *image-iff rel-of-def*)
  }
qed

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)
unfolding iti-sel-no-map-def
proof
  fix m P

  assume invar m
  with map-reverse-iterateoi.reverse-iterateoi-rule [OF iti, of m]
  have iti': map-iterator-rev-linord (iti m) (α m) by simp
  note sel-correct = iterate-sel-no-map-map-rev-linord-correct[OF iti', of P]

  { assume rel-of (α m) P ≠ {}
    with sel-correct show iterate-sel-no-map (iti m) P ∈ Some ‘ rel-of (α m) P
      by (auto simp add: image-iff rel-of-def)
  }

  { assume rel-of (α m) P = {}
     with sel-correct show iterate-sel-no-map (iti m) P = None
      by (auto simp add: image-iff rel-of-def)
  }

  { fix k v
    assume (k, v) ∈ rel-of (α m) P
    with sel-correct show k ≤ fst (the (iterate-sel-no-map (iti m) P))
      by (auto simp add: image-iff rel-of-def)
  }
qed
```

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)
unfolding it-map-to-list-def
proof
  fix m
  assume invar m
  with map-reverse-iterateoi.reverse-iterateoi-rule[OF iti, of m]
  have iti': map-iterator-rev-linord (iti m) (α m) by simp
  let ?l = iterate-to-list (iti m)

  from iterate-to-list-map-rev-linord-correct [OF iti']
  show sorted (map fst ?l)
       distinct (map fst ?l)
       map-of (iterate-to-list (iti m)) = α m by simp-all
qed
```

3.1.21 image restrict

```
definition it-map-image-filter ::
```

$('s1 \Rightarrow ('u1 \times 'v1,'s2)\ set\text{-}iterator) \Rightarrow (unit \Rightarrow 's2) \Rightarrow ('u2 \Rightarrow 'v2 \Rightarrow 's2 \Rightarrow 's2) \Rightarrow ('u1 \times 'v1 \Rightarrow ('u2 \times 'v2)\ option) \Rightarrow 's1 \Rightarrow 's2$
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* ())
unfolding *it-map-image-filter-def iterate-to-map-alt-def set-iterator-image-filter-def prod-case-beta*
by *simp*

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
fix *m* k' v' **and** *f* :: $('u1 \times 'v1) \Rightarrow ('u2 \times 'v2)\ option$
assume *invar-m*: *invar1 m* **and**
unique-f: *transforms-to-unique-keys* ($\alpha 1$ *m*) *f*

from *map-iteratei.iteratei-rule*[*OF it*, *OF invar-m*]
have *iti-m*: *map-iterator* (*map-it m*) ($\alpha 1$ *m*) **by** *simp*

from *unique-f* **have** *inj-on-f*: *inj-on f* (*map-to-set* ($\alpha 1$ *m*) $\cap$ *dom f*)
unfolding *transforms-to-unique-keys-def inj-on-def Ball-def map-to-set-def*
by *auto* (*metis option.inject*)

def $vP \equiv \lambda k\ v.\ \exists k'\ v'.\ \alpha 1\ m\ k' = Some\ v' \wedge f\ (k', v') = Some\ (k, v)$
have *vP-intro*: $\bigwedge k\ v.\ (\exists k'\ v'.\ \alpha 1\ m\ k' = Some\ v' \wedge f\ (k', v') = Some\ (k, v)) \longleftrightarrow vP\ k\ v$
unfolding *vP-def* **by** *simp*
{ fix *k v*
have *Eps-Opt* (*vP k*) = *Some v* $\longleftrightarrow$ *vP k v*
using *unique-f* **unfolding** *vP-def transforms-to-unique-keys-def*
apply (*rule-tac Eps-Opt-eq-Some*)
apply (*metis Pair-eq option.inject*)
done
} note *Eps-vP-elim*[*simp*] = *this*
have *map-intro*: $\{y.\ \exists x.\ x \in map\text{-}to\text{-}set\ (\alpha 1\ m) \wedge f\ x = Some\ y\} = map\text{-}to\text{-}set\ (\lambda k.\ Eps\text{-}Opt\ (vP\ k))$
by (*simp add*: *map-to-set-def vP-intro set-eq-iff split*: *prod.splits*)

from *set-iterator-image-filter-correct* [*OF iti-m*, *OF inj-on-f*, *unfolded map-intro*]

have *iti-filter*: *map-iterator* (*set-iterator-image-filter f* (*map-it m*))
(*λk. Eps-Opt* (*vP k*)) **by** *auto*

from *iterate-to-map-correct* [*OF up emp iti-filter*]
show *invar2* (*it-map-image-filter map-it map-emp map-up-dj f m*) ∧
(*α2* (*it-map-image-filter map-it map-emp map-up-dj f m*) *k′* = *Some v′*) =
(∃ *k v*. *α1 m k* = *Some v* ∧ *f* (*k*, *v*) = *Some* (*k′*, *v′*))
unfolding *it-map-image-filter-def vP-def*[*symmetric*]
by (*simp add*: *vP-intro*)
qed

definition *mif-map-value-image-filter* ::
((*′u* × *′v1* ⇒ (*′u* × *′v2*) *option*) ⇒ *′s1* ⇒ *′s2*) ⇒
(*′u* ⇒ *′v1* ⇒ *′v2 option*) ⇒ *′s1* ⇒ *′s2*
where
mif-map-value-image-filter mif f ≡
mif (λ(*k*, *v*). *case* (*f k v*) *of Some v′* ⇒ *Some* (*k*, *v′*) | *None* ⇒ *None*)

lemma *mif-map-value-image-filter-correct* :
fixes *α1* :: *′s1* ⇒ *′u* ⇀ *′v1*
fixes *α2* :: *′s2* ⇒ *′u* ⇀ *′v2*
shows *map-image-filter α1 invar1 α2 invar2 mif* ⟹
map-value-image-filter α1 invar1 α2 invar2 (*mif-map-value-image-filter mif*)
proof −
assume *map-image-filter α1 invar1 α2 invar2 mif*
then interpret *map-image-filter α1 invar1 α2 invar2 mif* **.**

show *map-value-image-filter α1 invar1 α2 invar2* (*mif-map-value-image-filter mif*)
proof (*intro map-value-image-filter.intro conjI*)
fix *m*
fix *f* :: *′u* ⇒ *′v1* ⇒ *′v2 option*
assume *invar*: *invar1 m*

let *?f* = λ(*k*, *v*). *case* (*f k v*) *of Some v′* ⇒ *Some* (*k*, *v′*) | *None* ⇒ *None*
have *unique-f*: *transforms-to-unique-keys* (*α1 m*) *?f*
unfolding *transforms-to-unique-keys-def*
by (*auto split*: *option.split*)

note *correct′* = *map-image-filter-correct* [*OF invar unique-f*, *folded mif-map-value-image-filter-def*]

from *correct′* **show** *invar2* (*mif-map-value-image-filter mif f m*)
by *simp*

have ⋀*k*. *α2* (*mif-map-value-image-filter mif f m*) *k* = *Option.bind* (*α1 m k*) (*f k*)
proof −

fix k
show $\alpha 2$ (*mif-map-value-image-filter mif f m*) k = *Option.bind* ($\alpha 1$ *m k*) (*f k*)
using *correct'*
apply (*cases Option.bind* ($\alpha 1$ *m k*) (*f k*))
apply (*auto split*: *option.split*)
apply (*case-tac* $\alpha 1$ *m k*)
apply (*auto split*: *option.split*)
apply (*metis option.inject*)
done
qed
thus $\alpha 2$ (*mif-map-value-image-filter mif f m*) = (λk. *Option.bind* ($\alpha 1$ *m k*) (*f k*))
by (*simp add*: *fun-eq-iff*)
qed
qed

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*)
apply (*simp add*: *fun-eq-iff*)
unfolding *it-map-image-filter-def mif-map-value-image-filter-def*
it-map-value-image-filter-def
by *fast*

lemma *it-map-value-image-filter-correct*:
fixes $\alpha 1$ :: '*s1* ⇒ '*u* ⇀ '*v1*
fixes $\alpha 2$ :: '*s2* ⇒ '*u* ⇀ '*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-value-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2*
(*it-map-value-image-filter map-it map-emp map-up-dj*)
proof −
note *mif-OK* = *it-map-image-filter-correct* [*OF it emp up*]
note *mif-map-value-image-filter-correct* [*OF mif-OK*]
thus *?thesis*
unfolding *it-map-value-image-filter-alt-def*
by *fast*
qed

definition *mif-map-restrict* ::
$(('u \times 'v \Rightarrow ('u \times 'v)\ option) \Rightarrow 's1 \Rightarrow 's2) \Rightarrow$
$('u \times 'v \Rightarrow bool) \Rightarrow 's1 \Rightarrow 's2$
where
mif-map-restrict mif P $\equiv$
mif $(\lambda(k, v).$ *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 $(\lambda k\ v.$ *if* $(P\ (k, v))$ *then Some v else None*)
proof –
have $(\lambda k\ v.$ *if* $P\ (k, v)$ *then Some* (k, v) *else None*) =
$(\lambda k\ v.$ *case if* $P\ (k, v)$ *then Some v else None of None* $\Rightarrow$ *None*
| *Some* $v' \Rightarrow$ *Some* (k, v'))
by (*simp add*: *fun-eq-iff*)
thus *?thesis*
unfolding *mif-map-restrict-def mif-map-value-image-filter-def*
by *simp*
qed

lemma *mif-map-restrict-correct* :
fixes $\alpha 1 :: 's1 \Rightarrow 'u \rightharpoonup 'v$
fixes $\alpha 2 :: 's2 \Rightarrow 'u \rightharpoonup 'v$
shows *map-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 mif* $\Longrightarrow$
map-restrict $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*mif-map-restrict mif*)
proof –
assume *map-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 mif*
then interpret *map-value-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2*
mif-map-value-image-filter mif
by (*rule mif-map-value-image-filter-correct*)

show *?thesis*
unfolding *mif-map-restrict-alt-def map-restrict-def*
apply (*auto simp add*: *map-value-image-filter-correct fun-eq-iff*
restrict-map-def)
apply (*case-tac* $\alpha 1$ *m x*)
apply *auto*
done
qed

definition *it-map-restrict* ::
$('s1 \Rightarrow ('u \times 'v,'s2)\ set\text{-}iterator) \Rightarrow (unit \Rightarrow 's2) \Rightarrow ('u \Rightarrow 'v \Rightarrow 's2 \Rightarrow 's2) \Rightarrow$
$('u \times 'v \Rightarrow bool) \Rightarrow 's1 \Rightarrow 's2$ **where**
it-map-restrict map-it map-emp map-up-dj P $\equiv$
it-map-image-filter map-it map-emp map-up-dj
$(\lambda(k, v).$ *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*)
apply (*simp add*: *fun-eq-iff*)
unfolding *it-map-image-filter-def mif-map-restrict-def*
it-map-restrict-def
by *fast*

lemma *it-map-restrict-correct*:
fixes $\alpha 1$ :: $'s1 \Rightarrow 'u \rightharpoonup 'v$
fixes $\alpha 2$ :: $'s2 \Rightarrow 'u \rightharpoonup 'v$
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-restrict* $\alpha 1$ *invar1* $\alpha 2$ *invar2*
(*it-map-restrict map-it map-emp map-up-dj*)
proof −
note *mif-OK* = *it-map-image-filter-correct* [*OF it emp up*]
note *mif-map-restrict-correct* [*OF mif-OK*]
thus *?thesis*
unfolding *it-map-restrict-alt-def*
by *fast*
qed

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* :: ($unit \Rightarrow 's$) $\Rightarrow$ ($'x \Rightarrow 's \Rightarrow 's$) $\Rightarrow$ $'x \Rightarrow 's$ **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 −
interpret *set-empty* α *invar e* + *set-ins* α *invar i* **by** *fact+*
show *?thesis*
apply (*unfold-locales*)
unfolding *ei-sng-def*
by (*auto simp*: *empty-correct ins-correct*)

qed

3.2.2 Disjoint Insert (by insert)

lemma (**in** *set-ins*) *ins-dj-by-ins*:
  shows *set-ins-dj* α *invar ins*
  by (*unfold-locales*)
    (*simp-all add*: *ins-correct*)

3.2.3 Disjoint Union (by union)

lemma (**in** *set-union*) *union-dj-by-union*:
  shows *set-union-dj* $\alpha 1$ *invar1* $\alpha 2$ *invar2* $\alpha 3$ *invar3 union*
  by (*unfold-locales*)
    (*simp-all add*: *union-correct*)

3.2.4 Iterators

Iteratei (by iterateoi)

lemma *iti-by-itoi*:
  assumes *set-iterateoi* α *invar it*
  shows *set-iteratei* α *invar it*
proof –
  interpret *set-iterateoi* α *invar it* **by** *fact*
  show *?thesis*
  proof (*unfold-locales*)
    fix *S*
    assume *invar S*
    with *iterateoi-rule*[*of S*] **have** *set-iterator-linord* (*it S*) (α *S*) **.**
    thus *set-iterator* (*it S*) (α *S*)
      by (*simp add*: *set-iterator-linord-def set-iterator-intro*)
  qed
qed

Iteratei (by reverse_iterateoi)

lemma *iti-by-ritoi*:
  assumes *set-reverse-iterateoi* α *invar it*
  shows *set-iteratei* α *invar it*
proof –
  interpret *set-reverse-iterateoi* α *invar it* **by** *fact*
  show *?thesis*
  proof (*unfold-locales*)
    fix *S*
    assume *invar S*
    with *reverse-iterateoi-rule*[*of S*] **have** *set-iterator-rev-linord* (*it S*) (α *S*) **.**
    thus *set-iterator* (*it S*) (α *S*)
      by (*simp add*: *set-iterator-rev-linord-def set-iterator-intro*)
  qed

qed

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*
unfolding *iti-isEmpty-def iterate-is-empty-def* **by** *simp*

lemma *iti-isEmpty-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-isEmpty* α *invar* ($\lambda s.$ (*iterate-is-empty* (*iti s*)))
proof
fix *s*
assume *invar s*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

from *iterate-is-empty-correct* [*OF iti-s*]
show *iterate-is-empty* (*iti s*) = (α *s* = {}) **.**
qed

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*
unfolding *iti-ball-def iterate-ball-def id-def*
by *simp*

lemma *iti-ball-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-ball* α *invar* (*iti-ball iti*)
unfolding *iti-ball-def*
proof
fix *s P*
assume *invar s*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

from *iterate-ball-correct* [*OF iti-s*, *of P*]
show *iterate-ball* (*iti s*) *P* = ($\forall x \in \alpha\ s.$ *P x*) **.**
qed

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* ($\lambda c.$ ¬*c*) (λx σ. *P x*) *False*
unfolding *iti-bexists-def iterate-bex-def*
by *simp*

lemma *iti-bexists-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-bexists* α *invar* (*iti-bexists iti*)
unfolding *iti-bexists-def*
proof
fix *s P*
assume *invar s*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

from *iterate-bex-correct* [*OF iti-s*, *of P*]
show *iterate-bex* (*iti s*) *P* = ($\exists x \in \alpha$ *s*. *P x*) **.**
qed

definition *neg-ball-bexists*
:: ('*s* ⇒ ('*a* ⇒ *bool*) ⇒ *bool*) ⇒ '*s* ⇒ ('*a* ⇒ *bool*) ⇒ *bool*
where *neg-ball-bexists ball s P* == ¬ (*ball s* ($\lambda x.$ ¬(*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 −
interpret *set-ball* α *invar ball* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
apply (*unfold neg-ball-bexists-def*)
apply (*simp add*: *ball-correct*)
done
qed

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*) (λx *n* . *Suc n*) *0*
unfolding *it-size-def iterate-size-def* **by** *simp*

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

unfolding *it-size-def*
proof
  fix *s*
  assume *invar s*
  with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
  have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

  from *iterate-size-correct* [*OF iti-s*]
  show *iterate-size* (*iti s*) = *card* (α *s*)   *finite* (α *s*) **by** *simp-all*
qed

3.2.8 Size with abort (by iterate)

definition *iti-size-abort* **where**
  iti-size-abort iti = (λ*m s*. *iterate-size-abort* (*iti s*) *m*)

lemma *iti-size-abort-code*[*code*] :
  iti-size-abort iti m s = *iti s* ($\lambda\sigma$. σ < *m*) (λ*x*. *Suc*) *0*
unfolding *iti-size-abort-def iterate-size-abort-def* **by** *simp*

lemma *iti-size-abort-correct*:
  assumes *iti*: *set-iteratei* α *invar iti*
  shows *set-size-abort* α *invar* (*iti-size-abort iti*)
unfolding *iti-size-abort-def*
proof
  fix *s n*
  assume *invar s*
  with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
  have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

  from *iterate-size-abort-correct* [*OF iti-s*]
  show *iterate-size-abort* (*iti s*) *n* = *min n* (*card* (α *s*))   *finite* (α *s*)
    by *simp-all*
qed

3.2.9 Singleton check (by size-abort)

definition *sza-isSng* **where**
  sza-isSng iti = (λ*s*. *iterate-is-sng* (*iti s*))

lemma *sza-isSng-code*[*code*] :
  sza-isSng iti s = (*iti s* ($\lambda\sigma$. σ < *2*) (λ*x*. *Suc*) *0* = *1*)
unfolding *sza-isSng-def iterate-is-sng-def iterate-size-abort-def*
by *simp*

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

fix *s*
assume *invar s*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

have *card* (α *s*) = (*Suc 0*) $\longleftrightarrow$ ($\exists$ *e*. α *s* = {*e*}) **by** (*simp add*: *card-Suc-eq*)
with *iterate-is-sng-correct* [*OF iti-s*]
show *iterate-is-sng* (*iti s*) = ($\exists$ *e*. α *s* = {*e*})
by *simp*
qed

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* ())
unfolding *it-copy-def iterate-to-set-alt-def* **by** *simp*

lemma *it-copy-correct*:
fixes α*1* :: *'s1* $\Rightarrow$ *'x set*
fixes α*2* :: *'s2* $\Rightarrow$ *'x set*
fixes *iteratei1* :: *'s1* $\Rightarrow$ (*'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*)
unfolding *it-copy-def*[*abs-def*]
proof
fix *s1*
assume *invar1 s1*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s1*]
have *iti-s*: *set-iterator* (*iteratei1 s1*) (α*1 s1*) **by** *simp*

from *iterate-to-set-correct*[*OF ins-dj-OK emp-OK iti-s*]
show α*2* (*iterate-to-set empty2 ins2* (*iteratei1 s1*)) = α*1 s1*
invar2 (*iterate-to-set empty2 ins2* (*iteratei1 s1*))
by *simp-all*
qed

3.2.11 Union (by iterate)

definition *it-union* **where**
it-union it ins $\equiv$ (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*
unfolding *it-union-def iterate-add-to-set-def* **by** *simp*

lemma *it-union-correct*:

fixes $\alpha 1$ :: $'s1 \Rightarrow 'x$ *set*
fixes $\alpha 2$ :: $'s2 \Rightarrow 'x$ *set*
assumes *S1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
assumes *S2*: *set-ins* $\alpha 2$ *invar2 ins2*
shows *set-union* $\alpha 1$ *invar1* $\alpha 2$ *invar2* $\alpha 2$ *invar2* (*it-union iteratei1 ins2*)
unfolding *it-union-def*
proof
fix *s1 s2*
assume *invar1 s1*
with *set-iteratei.iteratei-rule* [*OF S1*, *of s1*]
have *iti-s*: *set-iterator* (*iteratei1 s1*) ($\alpha 1$ *s1*) **by** *simp*

assume *invar2 s2*
with *iterate-add-to-set-correct*[*OF S2 - iti-s*, *of s2*]
show $\alpha 2$ (*iterate-add-to-set s2 ins2* (*iteratei1 s1*)) = $\alpha 1$ *s1* $\cup$ $\alpha 2$ *s2*
invar2 (*iterate-add-to-set s2 ins2* (*iteratei1 s1*))
by *simp-all*
qed

3.2.12 Disjoint Union

definition [*code-unfold*]: *it-union-dj* $\equiv$ *it-union*

lemma *it-union-dj-correct*:
fixes $\alpha 1$ :: $'s1 \Rightarrow 'x$ *set*
fixes $\alpha 2$ :: $'s2 \Rightarrow 'x$ *set*
assumes *S1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
assumes *S2*: *set-ins-dj* $\alpha 2$ *invar2 ins2*
shows *set-union-dj* $\alpha 1$ *invar1* $\alpha 2$ *invar2* $\alpha 2$ *invar2* (*it-union-dj iteratei1 ins2*)
unfolding *it-union-def it-union-dj-def*
proof
fix *s1 s2*
assume *invar1 s1*
with *set-iteratei.iteratei-rule* [*OF S1*, *of s1*]
have *iti-s*: *set-iterator* (*iteratei1 s1*) ($\alpha 1$ *s1*) **by** *simp*

assume *invar2 s2* $\alpha 1$ *s1* $\cap$ $\alpha 2$ *s2* = {}
with *iterate-add-to-set-dj-correct*[*OF S2 - iti-s*, *of s2*]
show $\alpha 2$ (*iterate-add-to-set s2 ins2* (*iteratei1 s1*)) = $\alpha 1$ *s1* $\cup$ $\alpha 2$ *s2*
invar2 (*iterate-add-to-set s2 ins2* (*iteratei1 s1*))
by *simp-all*
qed

3.2.13 Diff (by iterator)

definition *it-diff* **where**
it-diff iteratei1 del2 $\equiv$ (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*

unfolding *it-diff-def iterate-diff-set-def* **by** *simp*

lemma *it-diff-correct*:
  fixes $\alpha 1$ :: *'s1* $\Rightarrow$ *'x set*
  fixes $\alpha 2$ :: *'s2* $\Rightarrow$ *'x set*
  assumes *S1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
  assumes *S2*: *set-delete* $\alpha 2$ *invar2 del2*
  shows *set-diff* $\alpha 2$ *invar2* $\alpha 1$ *invar1* (*it-diff iteratei1 del2*)
unfolding *it-diff-def*
proof
  fix *s1 s2*
  assume *invar1 s1*
  with *set-iteratei.iteratei-rule* [*OF S1*, *of s1*]
  have *iti-s*: *set-iterator* (*iteratei1 s1*) ($\alpha 1$ *s1*) **by** *simp*

  assume *invar2 s2*
  with *iterate-diff-correct*[*OF S2 - iti-s*, *of s2*]
  show $\alpha 2$ (*iterate-diff-set s2 del2* (*iteratei1 s1*)) = $\alpha 2$ *s2* $-$ $\alpha 1$ *s1*
      invar2 (*iterate-diff-set s2 del2* (*iteratei1 s1*))
    by *simp-all*
qed

3.2.14 Intersection (by iterator)

definition *it-inter*
  :: (*'s1* $\Rightarrow$ (*'x*,*'s3*) *set-iterator*) $\Rightarrow$
     (*'x* $\Rightarrow$ *'s2* $\Rightarrow$ *bool*) $\Rightarrow$
     (*unit* $\Rightarrow$ *'s3*) $\Rightarrow$
     (*'x* $\Rightarrow$ *'s3* $\Rightarrow$ *'s3*) $\Rightarrow$
     's1 $\Rightarrow$ *'s2* $\Rightarrow$ *'s3*
  where
  it-inter iteratei1 memb2 empt3 insrt3 s1 s2 $\equiv$
   iterate-to-set empt3 insrt3 (*set-iterator-filter* ($\lambda x.$ *memb2 x s2*) (*iteratei1 s1*))

lemma *it-inter-code*[*code*] :
  it-inter iteratei1 memb2 empt3 insrt3 s1 s2 ==
        iteratei1 s1 (λ-. *True*) (λx *s*. *if memb2 x s2 then insrt3 x s else s*)
(*empt3* ())
   unfolding *it-inter-def iterate-to-set-alt-def set-iterator-filter-alt-def* **by** *simp*

lemma *it-inter-correct*:
  fixes $\alpha 1$ :: *'s1* $\Rightarrow$ *'x set*
  fixes $\alpha 2$ :: *'s2* $\Rightarrow$ *'x set*
  fixes $\alpha 3$ :: *'s3* $\Rightarrow$ *'x set*
  assumes *it1*: *set-iteratei* $\alpha 1$ *invar1 iteratei1*
  assumes *memb2*: *set-memb* $\alpha 2$ *invar2 memb2*
  assumes *emp3*: *set-empty* $\alpha 3$ *invar3 empty3*
  assumes *ins3*: *set-ins-dj* $\alpha 3$ *invar3 ins3*
  shows *set-inter* $\alpha 1$ *invar1* $\alpha 2$ *invar2* $\alpha 3$ *invar3* (*it-inter iteratei1 memb2 empty3*

```
ins3)
proof
  fix s1 s2
  assume invar1 s1    invar2 s2

  from set-iteratei.iteratei-rule [OF it1, OF ‹invar1 s1›]
  have iti-s1: set-iterator (iteratei1 s1) (α1 s1) by simp

  have {x ∈ α1 s1. memb2 x s2} = α1 s1 ∩ α2 s2 using ‹invar2 s2› memb2
    unfolding set-memb-def by auto
  with set-iterator-filter-correct [OF iti-s1, of λx. memb2 x s2]
  have iti-s12: set-iterator (set-iterator-filter (λx. memb2 x s2) (iteratei1 s1)) (α1
s1 ∩ α2 s2)
    by simp

  from iterate-to-set-correct[OF ins3 emp3 iti-s12]
  show α3 (it-inter iteratei1 memb2 empty3 ins3 s1 s2) = α1 s1 ∩ α2 s2
       invar3 (it-inter iteratei1 memb2 empty3 ins3 s1 s2)
    unfolding it-inter-def by simp-all
qed
```

3.2.15 Subset (by ball)

```
definition ball-subset ::
  ('s1 ⇒ ('a ⇒ bool) ⇒ bool) ⇒ ('a ⇒ 's2 ⇒ bool) ⇒ 's1 ⇒ 's2 ⇒ bool
  where ball-subset ball1 mem2 s1 s2 == ball1 s1 (λx. mem2 x s2)

lemma ball-subset-correct:
  assumes set-ball α1 invar1 ball1
  assumes set-memb α2 invar2 mem2
  shows set-subset α1 invar1 α2 invar2 (ball-subset ball1 mem2)
proof –
  interpret s1: set-ball α1 invar1 ball1 by fact
  interpret s2: set-memb α2 invar2 mem2 by fact

  show ?thesis
    apply (unfold-locales)
    apply (unfold ball-subset-def)
    apply (auto simp add: s1.ball-correct s2.memb-correct)
    done
qed
```

3.2.16 Equality Test (by subset)

```
definition subset-equal ::
  ('s1 ⇒ 's2 ⇒ bool) ⇒ ('s2 ⇒ 's1 ⇒ bool) ⇒ 's1 ⇒ 's2 ⇒ bool
  where subset-equal ss1 ss2 s1 s2 == ss1 s1 s2 ∧ ss2 s2 s1

lemma subset-equal-correct:
  assumes set-subset α1 invar1 α2 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 –
interpret *s1*: *set-subset* $\alpha 1$ *invar1* $\alpha 2$ *invar2 ss1* **by** *fact*
interpret *s2*: *set-subset* $\alpha 2$ *invar2* $\alpha 1$ *invar1 ss2* **by** *fact*

show *?thesis*
apply *unfold-locales*
apply (*unfold subset-equal-def*)
apply (*auto simp add*: *s1.subset-correct s2.subset-correct*)
done
qed

3.2.17 Image-Filter (by iterate)

definition *it-image-filter*
:: (*'s1* ⇒ (*'a*,-) *set-iterator*) ⇒ (*unit* ⇒ *'s2*) ⇒ (*'b* ⇒ *'s2* ⇒ *'s2*)
⇒ (*'a* ⇀ *'b*) ⇒ *'s1* ⇒ *'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*) (λ*x res*. *case f x of Some v* ⇒ *ins2 v res* | - ⇒ *res*) (*empty2* ())
unfolding *it-image-filter-def iterate-to-set-alt-def set-iterator-image-filter-def*
by *simp*

lemma *it-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*: *set-ins* $\alpha 2$ *invar2 ins2*
shows *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*it-image-filter iteratei1 empty2 ins2*)
proof
fix *s* **and** *f* :: *'x1* ⇒ *'x2 option*
assume *invar1 s*
from *set-iteratei.iteratei-rule* [*OF iti1*, *OF* ‹*invar1 s*›]
have *iti-s1*: *set-iterator* (*iteratei1 s*) ($\alpha 1$ *s*) **by** *simp*

from *iterate-image-filter-to-set-correct*[*OF ins emp2 iti-s1*]
show $\alpha 2$ (*it-image-filter iteratei1 empty2 ins2 f s*) =
$\{b.\ \exists a \in \alpha 1\ s.\ f\ a = Some\ b\}$
invar2 (*it-image-filter iteratei1 empty2 ins2 f s*)
unfolding *it-image-filter-def* **by** *auto*
qed

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 $\alpha 1$ :: $'s1 \Rightarrow 'x1$ *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-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
  fix *s* **and** *f* :: $'x1 \Rightarrow 'x2$ *option*
  assume *invar1 s*   *inj-on f* ($\alpha 1$ *s* $\cap$ *dom f*)
  from *set-iteratei.iteratei-rule* [*OF iti1*, *OF* ‹*invar1 s*›]
  have *iti-s1*: *set-iterator* (*iteratei1 s*) ($\alpha 1$ *s*) **by** *simp*

  from *set-iterator-image-filter-correct* [*OF iti-s1*, *OF* ‹*inj-on f* ($\alpha 1$ *s* $\cap$ *dom f*)›]
  have *iti-s1-filter*: *set-iterator* (*set-iterator-image-filter f* (*iteratei1 s*))
    $\{y.\ \exists x.\ x \in \alpha 1\ s \wedge f\ x = Some\ y\}$
    by *simp*

  from *iterate-to-set-correct*[*OF ins-dj2 emp2*, *OF iti-s1-filter*]
  show $\alpha 2$ (*it-inj-image-filter iteratei1 empty2 ins2 f s*) =
      $\{b.\ \exists a{\in}\alpha 1\ s.\ f\ a = Some\ b\}$
     invar2 (*it-inj-image-filter iteratei1 empty2 ins2 f s*)
    unfolding *it-inj-image-filter-def it-image-filter-def* **by** *auto*
qed

3.2.19 Image (by image-filter)

definition *iflt-image iflt f s* == *iflt* ($\lambda x.$ *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 –
  interpret *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*unfold iflt-image-def*)
    apply (*auto simp add*: *image-filter-correct*)
    done
qed

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 −
  interpret *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt* **by** *fact*

  show *?thesis*
    apply (*unfold-locales*)
    apply (*unfold iflt-image-def iflt-inj-image-def*)
    apply (*subst inj-image-filter-correct*)
    apply (*auto simp add*: *dom-const intro*: *inj-onI dest*: *inj-onD*)
    apply (*subst inj-image-filter-correct*)
    apply (*auto simp add*: *dom-const intro*: *inj-onI dest*: *inj-onD*)
    done
qed

3.2.21 Filter (by image-filter)

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

lemma *iflt-filter-correct*:
  fixes $\alpha 1$ :: $'s1 \Rightarrow 'a$ *set*
  fixes $\alpha 2$ :: $'s2 \Rightarrow 'a$ *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 (*rule set-filter.intro*)
  fix *s P*
  assume *invar-s*: *invar1 s*
  interpret *S*: *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt* **by** *fact*

  let $?f' = \lambda x{::}'a$. *if P x then Some x else None*
  have *inj-f'*: *inj-on* $?f'$ ($\alpha 1$ *s* $\cap$ *dom* $?f'$)
    by (*simp add*: *inj-on-def Ball-def domIff*)
  note *correct'* = *S.inj-image-filter-correct* [*OF invar-s inj-f'*,
    folded iflt-filter-def]

  show *invar2* (*iflt-filter iflt P s*)
      $\alpha 2$ (*iflt-filter iflt P s*) = $\{e \in \alpha 1\ s.\ P\ e\}$
    by (*auto simp add*: *correct'*)
qed

3.2.22 union-list

definition *fold-union-list* **where**
  fold-union-list emp un l =
  foldl ($\lambda s\ s'$. *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
fix *l*
note *correct* = *set-empty.empty-correct*[*OF emp-OK*]
set-union.union-correct[*OF un-OK*]

assume $\forall$ *s1*$\in$*set l*. *invar s1*
hence *aux*: $\bigwedge s.$ *invar s* $\Longrightarrow$
α (*foldl* ($\lambda s\ s'.$ *un s s'*) *s l*) = $\bigcup\{\alpha\ s1\ |s1.\ s1 \in set\ l\} \cup \alpha\ s \wedge$
invar (*foldl* ($\lambda s\ s'.$ *un s s'*) *s l*)
by (*induct l*) (*auto simp add*: *correct*)

from *aux* [*of emp*()]
show α (*fold-union-list emp un l*) = $\bigcup\{\alpha\ s1\ |s1.\ s1 \in set\ l\}$
invar (*fold-union-list emp un l*)
unfolding *fold-union-list-def*
by (*simp-all add*: *correct*)
qed

function *paired-union-list* :: - $\Rightarrow$ - $\Rightarrow$ *'a set list* $\Rightarrow$ *'a set list* $\Rightarrow$ *'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*
by *pat-completeness simp-all*
termination
by (*relation measure* (λ(-, -, *l1*, *l2*). *3* $*$ *length l1* + *2* $*$ *length l2*)) (*simp-all*)

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
fix *l*
note *correct* = *set-empty.empty-correct*[*OF emp-OK*]
set-union.union-correct[*OF un-OK*]

assume *l-OK*: $\forall$ *s1*$\in$*set l*. *invar s1*
{ **fix** *as*
from *correct l-OK*
have $\forall$ *s*$\in$*set as*. *invar s* $\Longrightarrow$
α (*paired-union-list emp un as l*) = $\bigcup\{\alpha\ s1\ |s1.\ s1 \in set\ (as\ @\ l)\} \wedge$
invar (*paired-union-list emp un as l*) (**is** - $\Longrightarrow$ *?P emp un as l*)
proof (*induct emp un as l rule*: *paired-union-list.induct*)
case *1* **thus** *?case* **by** *simp*
next

case *2* **thus** *?case* **by** *simp*
next
case *3* **thus** *?case* **by** *simp*
next
case *4* **thus** *?case* **by** *auto*
next
case (*5 emp un l1 s1 s2 l2*)
from *5*(*1−7*) **have** *ind-hyp*: *?P emp un* ((*un s1 s2*)#*l1*) *l2* **by** *simp*
note *l1-OK* = *5*(*2*)
note *emp-un-correct* = *5*(*3−6*)
note *s12-l2-OK* = *5*(*7*)

from *emp-un-correct s12-l2-OK* **have** *un-s12*: α (*un s1 s2*) = α *s1* $\cup$ α *s2* **by** *simp*

from *un-s12*
show *?case* **by** (*simp add*: *ind-hyp*) *auto*
qed
} **note** *aux* = *this*

from *aux*[*of* []]
show α (*paired-union-list emp un* [] *l*) = $\bigcup\{\alpha$ *s1* |*s1*. *s1* $\in$ *set l*$\}$
invar (*paired-union-list emp un* [] *l*) **by** *simp-all*
qed

3.2.23 Union of image of Set (by iterate)

definition *it-Union-image*
:: (*'s1* $\Rightarrow$ (*'x*,-) *set-iterator*) $\Rightarrow$ (*unit* $\Rightarrow$ *'s3*) $\Rightarrow$ (*'s2* $\Rightarrow$ *'s3* $\Rightarrow$ *'s3*)
$\Rightarrow$ (*'x* $\Rightarrow$ *'s2*) $\Rightarrow$ *'s1* $\Rightarrow$ *'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 −
interpret *set-iteratei* α*1 invar1 iti1* **by** *fact*
interpret *set-empty* α*3 invar3 em3* **by** *fact*
interpret *set-union* α*2 invar2* α*3 invar3* α*3 invar3 un233* **by** *fact*

{
fix *s f*
have $[\![$*invar1 s*; $\bigwedge$*x*. *x* $\in$ α*1 s* $\Longrightarrow$ *invar2* (*f x*)$]\!]$ $\Longrightarrow$
α*3* (*it-Union-image iti1 em3 un233 f s*) = $\bigcup$$\alpha$*2* ‘ *f* ‘ α*1 s*
$\wedge$ *invar3* (*it-Union-image iti1 em3 un233 f s*)

```
    apply (unfold it-Union-image-def)
    apply (rule-tac I=λit res. invar3 res ∧ α3 res = ⋃α2`f`(α1 s − it) in
iterate-rule-P)
    apply (fastforce simp add: empty-correct union-correct)+
    done
  }
  thus ?thesis
    apply unfold-locales
    apply auto
    done
qed
```

3.2.24 Disjointness Check with Witness (by sel)

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

lemma sel-disjoint-witness-correct:
  assumes set-sel α1 invar1 sel1
  assumes set-memb α2 invar2 mem2
  shows set-disjoint-witness α1 invar1 α2 invar2 (sel-disjoint-witness sel1 mem2)
proof −
  interpret s1: set-sel α1 invar1 sel1 by fact
  interpret s2: set-memb α2 invar2 mem2 by fact
  show ?thesis
    apply (unfold-locales)
    apply (unfold sel-disjoint-witness-def)
    apply (auto dest: s1.sel-noneD
            elim: s1.sel-someD
            simp add: s2.memb-correct
            split: split-if-asm)
    done
qed
```

3.2.25 Disjointness Check (by ball)

```
definition ball-disjoint ball1 memb2 s1 s2 == ball1 s1 (λx. ¬ memb2 x s2)

lemma ball-disjoint-correct:
  assumes set-ball α1 invar1 ball1
  assumes set-memb α2 invar2 memb2
  shows set-disjoint α1 invar1 α2 invar2 (ball-disjoint ball1 memb2)
proof −
  interpret s1: set-ball α1 invar1 ball1 by fact
  interpret s2: set-memb α2 invar2 memb2 by fact

  show ?thesis
    apply (unfold-locales)
    apply (unfold ball-disjoint-def)
    apply (auto simp add: s1.ball-correct s2.memb-correct)
```

done
qed

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*
unfolding *iti-sel-def iterate-sel-def* **by** *simp*

lemma *iti-sel-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-sel* α *invar* (*iti-sel iti*)
unfolding *set-sel-def iti-sel-def*
using *iterate-sel-genord-correct* [*OF set-iteratei.iteratei-rule* [*OF iti*, *unfolded set-iterator-def*]]
apply (*simp add*: *Bex-def*) **apply** *clarify*
apply (*metis option.exhaust*)
done

3.2.27 Map-free selection by selection

definition *sel-sel'*
:: ($'s \Rightarrow$ ($'x \Rightarrow$ - *option*) $\Rightarrow$ - *option*) $\Rightarrow$ $'s \Rightarrow$ ($'x \Rightarrow$ *bool*) $\Rightarrow$ $'x$ *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 −
interpret *set-sel* α *invar sel* **by** *fact*

show *?thesis*
proof
case *goal1* **show** *?case*
apply (*rule selE*[*OF goal1*(*1*,*2*), **where** *f*=($\lambda x.$ *if P x then Some x else None*)])
apply (*simp add*: *goal1*)
apply (*simp split*: *split-if-asm*)
apply (*fold sel-sel'-def*)
apply (*blast intro*: *goal1*(*4*))
done
next
case *goal2* **thus** *?case*
apply (*auto simp add*: *sel-sel'-def*)
apply (*drule selI*[**where** *f*=($\lambda x.$ *if P x then Some x else None*)])
apply *auto*
done
qed

qed

3.2.28 Map-free selection by iterate

definition *iti-sel-no-map* **where**
iti-sel-no-map iti = (λs *P*. *iterate-sel-no-map* (*iti s*) *P*)

lemma *iti-sel-no-map-code*[*code*]:
iti-sel-no-map iti s f = *iti s* ($\lambda\sigma$. σ = *None*) (λx σ. *if f x then Some x else None*) *None*
unfolding *iti-sel-no-map-def iterate-sel-no-map-alt-def* **by** *simp*

lemma *iti-sel′-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-sel′* α *invar* (*iti-sel-no-map iti*)
unfolding *set-sel′-def iti-sel-no-map-def*
using *iterate-sel-no-map-correct* [*OF set-iteratei.iteratei-rule* [*OF iti*]]
apply (*simp add*: *Bex-def Ball-def*)
apply (*metis option.exhaust*)
done

3.2.29 Set to List (by iterate)

definition *it-set-to-list* **where**
it-set-to-list iti = (λs. *iterate-to-list* (*iti s*))

lemma *it-set-to-list-code*[*code*] :
it-set-to-list iti s = *iti s* (λ-. *True*) *op#* []
unfolding *it-set-to-list-def iterate-to-list-def* **by** *simp*

lemma *it-set-to-list-correct*:
assumes *iti*: *set-iteratei* α *invar iti*
shows *set-to-list* α *invar* (*it-set-to-list iti*)
unfolding *it-set-to-list-def*
proof
fix *s*
assume *invar s*
with *set-iteratei.iteratei-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator* (*iti s*) (α *s*) **by** *simp*

from *iterate-to-list-correct* [*OF iti-s*]
show *set* (*iterate-to-list* (*iti s*)) = α *s*
distinct (*iterate-to-list* (*iti s*)) **by** *simp-all*
qed

3.2.30 List to Set

— Tail recursive version
fun *gen-list-to-set-aux*
:: ($'x \Rightarrow 's \Rightarrow 's) \Rightarrow 's \Rightarrow 'x$ *list* $\Rightarrow 's$

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 –
interpret *set-empty* α *invar empt* **by** *fact*
interpret *set-ins* α *invar ins* **by** *fact*

{ — Show a generalized lemma
fix *l accs*
have *invar accs* $\Longrightarrow$ α (*gen-list-to-set-aux ins accs l*) = *set l* $\cup$ α *accs*
$\wedge$ *invar* (*gen-list-to-set-aux ins accs l*)
by (*induct l arbitrary*: *accs*)
(*auto simp add*: *ins-correct*)
} **thus** *?thesis*
apply (*unfold-locales*)
apply (*unfold gen-list-to-set-def*)
apply (*auto simp add*: *empty-correct*)
done
qed

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*
:: ($'s1 \Rightarrow ('x,'s3)$ *set-iterator*) $\Rightarrow$
($'s2 \Rightarrow ('y,'s3)$ *set-iterator*) $\Rightarrow$
(*unit* $\Rightarrow 's3) \Rightarrow ('z \Rightarrow 's3 \Rightarrow 's3) \Rightarrow$
$('x \times 'y \Rightarrow 'z$ *option*$) \Rightarrow 's1 \Rightarrow 's2 \Rightarrow 's3$
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*) (λx *res*.
iterate2 s2 (λ-. *True*) (λy *res*.
case (*f* (*x*, *y*)) *of*

None ⇒ *res*
| *Some z* ⇒ (*insert3 z res*)
) *res*
) (*empty3* ())
unfolding *image-filter-cartesian-product-def iterate-to-set-alt-def*
set-iterator-image-filter-def set-iterator-product-def
by *simp*

lemma *image-filter-cartesian-product-correct*:
assumes *S*: *set-iteratei* α*1 invar1 iteratei1*
set-iteratei α*2 invar2 iteratei2*
set-empty α*3 invar3 empty3*
set-ins α*3 invar3 ins3*
assumes *I*[*simp*, *intro*!]: *invar1 s1* *invar2 s2*
shows α*3* (*image-filter-cartesian-product iteratei1 iteratei2 empty3 ins3 f s1 s2*)
= { *z* | *x y z*. *f* (*x*,*y*) = *Some z* ∧ *x*∈α*1 s1* ∧ *y*∈α*2 s2* } (**is** *?T1*)
invar3 (*image-filter-cartesian-product iteratei1 iteratei2 empty3 ins3 f s1 s2*) (**is** *?T2*)
proof −
from *set-iteratei.iteratei-rule* [*OF S*(*1*), *OF* ‹*invar1 s1*›]
have *it-s1*: *set-iterator* (*iteratei1 s1*) (α*1 s1*) **by** *simp*

from *set-iteratei.iteratei-rule* [*OF S*(*2*), *OF* ‹*invar2 s2*›]
have *it-s2*: *set-iterator* (*iteratei2 s2*) (α*2 s2*) **by** *simp*

from *set-iterator-product-correct* [*OF it-s1*, *OF it-s2*]
have *it-s12*: *set-iterator* (*set-iterator-product* (*iteratei1 s1*) (λ-. *iteratei2 s2*))
(α*1 s1* × α*2 s2*) **by** *simp*

from *iterate-image-filter-to-set-correct*[*OF S*(*4*) *S*(*3*) *it-s12*, *of f*]
show *?T1 ?T2*
unfolding *image-filter-cartesian-product-def* **by** *auto*
qed

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

lemma *image-filter-cp-correct*:
assumes *S*: *set-iteratei* α*1 invar1 iterate1*
set-iteratei α*2 invar2 iterate2*
set-empty α*3 invar3 empty3*
set-ins α*3 invar3 ins3*
assumes *I*: *invar1 s1* *invar2 s2*
shows
α*3* (*image-filter-cp iterate1 iterate2 empty3 ins3 f P s1 s2*)
= { *f* (*x*, *y*) | *x y*. *P* (*x*, *y*) ∧ *x*∈α*1 s1* ∧ *y*∈α*2 s2* } (**is** *?T1*)

invar3 (*image-filter-cp iterate1 iterate2 empty3 ins3 f P s1 s2*) (**is** *?T2*)
proof −
note *image-filter-cartesian-product-correct* [*OF S*, *OF I*]
thus *?T1* *?T2*
unfolding *image-filter-cp-def*
by *auto*
qed

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* α*1 invar1 iteratei1*
set-iteratei α*2 invar2 iteratei2*
set-empty α*3 invar3 empty3*
set-ins-dj α*3 invar3 ins-dj3*
assumes *I*[*simp*, *intro*!]: *invar1 s1* *invar2 s2*
assumes *INJ*: !!$x\ y\ x'\ y'\ z$. ⟦ $f\ (x, y) = Some\ z$; $f\ (x', y') = Some\ z$ ⟧ $\Longrightarrow x{=}x' \wedge y{=}y'$
shows α*3* (*inj-image-filter-cartesian-product iteratei1 iteratei2 empty3 ins-dj3 f s1 s2*)
= { z | $x\ y\ z$. $f\ (x, y) = Some\ z \wedge x{\in}\alpha 1\ s1 \wedge y{\in}\alpha 2\ s2$ } (**is** *?T1*)
invar3 (*inj-image-filter-cartesian-product iteratei1 iteratei2 empty3 ins-dj3 f s1 s2*) (**is** *?T2*)
proof −
from *set-iteratei.iteratei-rule* [*OF S*(*1*), *OF* ‹*invar1 s1*›]
have *it-s1*: *set-iterator* (*iteratei1 s1*) (α*1 s1*) **by** *simp*

from *set-iteratei.iteratei-rule* [*OF S*(*2*), *OF* ‹*invar2 s2*›]
have *it-s2*: *set-iterator* (*iteratei2 s2*) (α*2 s2*) **by** *simp*

from *set-iterator-product-correct* [*OF it-s1*, *OF it-s2*]
have *it-s12*: *set-iterator* (*set-iterator-product* (*iteratei1 s1*) (λ-. *iteratei2 s2*))
(α*1 s1* × α*2 s2*) **by** *simp*

from *INJ* **have** *f-inj-on*: *inj-on f* (α*1 s1* × α*2 s2* ∩ *dom f*)
unfolding *inj-on-def dom-def* **by** *auto*

from *iterate-inj-image-filter-to-set-correct*[*OF S*(*4*) *S*(*3*) *it-s12 f-inj-on*]
show *?T1 ?T2*
unfolding *image-filter-cartesian-product-def inj-image-filter-cartesian-product-def*

by *auto*
qed

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 α1 invar1 iterate1*
    set-iteratei α2 invar2 iterate2
    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*. ⟦ *P* (*x*, *y*); *P* (*x′*, *y′*); *f* (*x*, *y*) = *f* (*x′*, *y′*) ⟧ ⟹ *x*=*x′* ∧ *y*=*y′*
  shows *α3* (*inj-image-filter-cp iterate1 iterate2 empty3 ins-dj3 f P s1 s2*)
    = { *f* (*x*, *y*) | *x y*. *P* (*x*, *y*) ∧ *x*∈*α1 s1* ∧ *y*∈*α2 s2* } (**is** *?T1*)
  invar3 (*inj-image-filter-cp iterate1 iterate2 empty3 ins-dj3 f P s1 s2*) (**is** *?T2*)
proof −
  let *?f* = λ*xy*. *if P xy then Some* (*f xy*) *else None*
  from *INJ* **have** *INJ′*:
    !!*x y x′ y′ z*. ⟦ *?f* (*x*, *y*) = *Some z*; *?f* (*x′*, *y′*) = *Some z* ⟧ ⟹ *x*=*x′* ∧ *y*=*y′*
   by (*auto simp add*: *split-if-eq1*)

  note *inj-image-filter-cartesian-product-correct* [*OF S*, *OF I*,
    where *f* = *?f*]

  with *INJ′* **show** *?T1*   *?T2*
    unfolding *image-filter-cp-def inj-image-filter-cp-def*
      inj-image-filter-cartesian-product-def
    by *auto*
qed

Cartesian Product

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

lemma *cart-code*[*code*] :
cart it1 it2 empty3 ins3 s1 s2 ≡
 it1 s1 (λ-. *True*) (λ*x*. *it2 s2* (λ-. *True*) (λ*y res*. *ins3* (*x*,*y*) *res*)) (*empty3* ())
 unfolding *cart-def image-filter-cartesian-product-code*
by *simp*

lemma *cart-correct*:
  assumes *S*: *set-iteratei α1 invar1 iterate1*
    set-iteratei α2 invar2 iterate2
    set-empty α3 invar3 empty3
    set-ins-dj α3 invar3 ins-dj3
  assumes *I*[*simp*, *intro*!]: *invar1 s1*   *invar2 s2*
  shows *α3* (*cart iterate1 iterate2 empty3 ins-dj3 s1 s2*)
    = *α1 s1* × *α2 s2* (**is** *?T1*)
  invar3 (*cart iterate1 iterate2 empty3 ins-dj3 s1 s2*) (**is** *?T2*)
  apply −
  apply (*unfold cart-def inj-image-filter-cartesian-product-def*[*symmetric*])

apply (*subst inj-image-filter-cartesian-product-correct*[*OF S*, *OF I*])
apply *auto*
apply (*subst inj-image-filter-cartesian-product-correct*[*OF S*, *OF I*])
apply *auto*
done

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*)
unfolding *set-min-def iti-sel-no-map-def*
using *iterate-sel-no-map-linord-correct* [*OF set-iterateoi.iterateoi-rule* [*OF iti*]]
apply (*simp add*: *Bex-def Ball-def image-iff*)
apply (*metis option.exhaust the.simps*)
done

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*)
unfolding *set-max-def iti-sel-no-map-def*
using *iterate-sel-no-map-rev-linord-correct* [*OF set-reverse-iterateoi.reverse-iterateoi-rule* [*OF iti*]]
apply (*simp add*: *Bex-def Ball-def image-iff*)
apply (*metis option.exhaust the.simps*)
done

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*)
unfolding *it-set-to-list-def*
proof
fix *s*
assume *invar s*
with *set-reverse-iterateoi.reverse-iterateoi-rule* [*OF iti*, *of s*]
have *iti-s*: *set-iterator-rev-linord* (*iti s*) (α *s*) **by** *simp*

from *iterate-to-list-rev-linord-correct* [*OF iti-s*]
show *set* (*iterate-to-list* (*iti s*)) = α *s*
sorted (*iterate-to-list* (*iti s*))
distinct (*iterate-to-list* (*iti s*)) **by** *simp-all*
qed

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 $s\text{-}\alpha$ :: $('s \Rightarrow 'u \rightharpoonup unit) \Rightarrow 's \Rightarrow 'u\ set$
  where $s\text{-}\alpha\ \alpha\ m == dom\ (\alpha\ m)$
definition[*code-unfold*]: $s\text{-}empty\ empt = empt$
definition[*code-unfold*]: $s\text{-}sng\ sng\ x = sng\ x\ ()$
definition $s\text{-}memb\ lookup\ x\ s == lookup\ x\ s \neq None$
definition[*code-unfold*]: $s\text{-}ins\ update\ x\ s == update\ x\ ()\ s$
definition[*code-unfold*]: $s\text{-}ins\text{-}dj\ update\text{-}dj\ x\ s == update\text{-}dj\ x\ ()\ s$
definition[*code-unfold*]: $s\text{-}delete\ delete == delete$
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 $s\text{-}sel\ sel\ s\ f == sel\ s\ (\lambda(u,\ \text{-}).\ f\ u)$
declare *s-sel-def*[*code-unfold*]
definition[*code-unfold*]: $s\text{-}isEmpty\ isEmpty == isEmpty$
definition[*code-unfold*]: $s\text{-}isSng\ isSng == isSng$

definition *s-iteratei* :: $('s \Rightarrow (('k \times unit),'\sigma)\ set\text{-}iterator) \Rightarrow ('s \Rightarrow ('k,\ '\sigma)\ set\text{-}iterator)$
  where $s\text{-}iteratei\ it\ s == map\text{-}iterator\text{-}dom\ (it\ s)$
lemma *s-iteratei-code*[*code*] :
  $s\text{-}iteratei\ it\ s\ c\ f = it\ s\ c\ (\lambda x.\ f\ (fst\ x))$
unfolding *s-iteratei-def map-iterator-dom-def set-iterator-image-alt-def* **by** *simp*

definition *s-ball*
  :: $('s \Rightarrow ('u \times unit \Rightarrow bool) \Rightarrow bool) \Rightarrow 's \Rightarrow ('u \Rightarrow bool) \Rightarrow bool$
  where $s\text{-}ball\ ball\ s\ P == ball\ s\ (\lambda(u,\ \text{-}).\ P\ u)$
declare *s-ball-def*[*code-unfold*]

definition *s-bexists*
  :: $('s \Rightarrow ('u \times unit \Rightarrow bool) \Rightarrow bool) \Rightarrow 's \Rightarrow ('u \Rightarrow bool) \Rightarrow bool$
  where $s\text{-}bexists\ bexists\ s\ P == bexists\ s\ (\lambda(u,\text{-}).\ P\ u)$
declare *s-bexists-def*[*code-unfold*]

definition[*code-unfold*]: $s\text{-}size\ sz \equiv sz$
definition[*code-unfold*]: $s\text{-}size\text{-}abort\ size\text{-}abort \equiv size\text{-}abort$

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 −
interpret *map-empty* α *invar empty* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs empty-correct*)
qed

lemma *s-sng-correct*:
fixes *sng*
assumes *map-sng* α *invar sng*
shows *set-sng* (*s-*α α) *invar* (*s-sng sng*)
proof −
interpret *map-sng* α *invar sng* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs sng-correct*)
qed

lemma *s-memb-correct*:
fixes *lookup*

assumes *map-lookup* α *invar lookup*
shows *set-memb* (*s-*α α) *invar* (*s-memb lookup*)
proof −
interpret *map-lookup* α *invar lookup* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs lookup-correct*)
qed

lemma *s-ins-correct*:
fixes *ins*
assumes *map-update* α *invar update*
shows *set-ins* (*s-*α α) *invar* (*s-ins update*)
proof −
interpret *map-update* α *invar update* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs update-correct*)
qed

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 −
interpret *map-update-dj* α *invar update-dj* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs update-dj-correct*)
qed

lemma *s-delete-correct*:
fixes *delete*
assumes *map-delete* α *invar delete*
shows *set-delete* (*s-*α α) *invar* (*s-delete delete*)
proof −
interpret *map-delete* α *invar delete* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs delete-correct*)
qed

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
fix *s* **and** *f* :: *'b* $\Rightarrow$ *'c option*
assume *invar-s*: *invar s*

{
  assume $\forall x{\in}s\text{-}\alpha\ \alpha\ s.\ f\,x = None$
  with *map-sel.selI* [*OF sel-OK*, *OF invar-s*, *of* $\lambda(u, \text{-}).\ f\,u$]
  show *s-sel sel s f = None*
    by (*simp add*: *Ball-def s-α-def dom-def s-sel-def*)
}

{ **fix** *x r Q*
  assume $x \in s\text{-}\alpha\ \alpha\ s \quad f\,x = Some\ r$
    $\bigwedge x\ r.\ s\text{-}sel\ sel\ s\ f = Some\ r \Longrightarrow$
      $x \in s\text{-}\alpha\ \alpha\ s \Longrightarrow f\,x = Some\ r \Longrightarrow Q$
  with *map-sel.selE* [*OF sel-OK*, *OF invar-s*, *of x* () $\lambda(u, \text{-}).\ f\ u\ r\ Q$]
  show *Q* **by** (*simp add*: *s-α-def dom-def s-sel-def*)
}
qed

lemma *s-isEmpty-correct*:
  fixes *isEmpty*
  assumes *map-isEmpty α invar isEmpty*
  shows *set-isEmpty* (*s-α α*) *invar* (*s-isEmpty isEmpty*)
proof −
  interpret *map-isEmpty α invar isEmpty* **by** *fact*
  show *?thesis*
    by *unfold-locales*
      (*auto simp add*: *s-defs isEmpty-correct*)
qed

lemma *s-isSng-correct*:
  fixes *isSng*
  assumes *map-isSng α invar isSng*
  shows *set-isSng* (*s-α α*) *invar* (*s-isSng isSng*)
proof −
  interpret *map-isSng α invar isSng* **by** *fact*
  show *?thesis*
    by *unfold-locales*
      (*auto simp add*: *s-defs isSng-correct dom-eq-singleton-conv*)
qed

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
  fix *s*
  assume *invar-s*: *invar s*

from *map-iteratei.iteratei-rule*[*OF map-it-OK*,*OF invar-s*]
have *iti-s*: *map-iterator* (*iteratei s*) (α *s*) **.**

from *set-iterator-finite*[*OF iti-s*]
show *finite* (*s-*α α *s*)
unfolding *s-*α*-def finite-map-to-set* **.**

from *map-iterator-dom-correct* [*OF iti-s*]
show *set-iterator* (*s-iteratei iteratei s*) (*s-*α α *s*)
unfolding *s-*α*-def s-iteratei-def* **.**
qed

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
fix *s*
assume *invar-s*: *invar s*

from *map-iterateoi.iterateoi-rule*[*OF map-it-OK*,*OF invar-s*]
have *iti-s*: *map-iterator-linord* (*iteratei s*) (α *s*) **.**

from *set-iterator-genord.finite-S0*[*OF iti-s*[*unfolded set-iterator-map-linord-def*]]

show *finite* (*s-*α α *s*)
unfolding *s-*α*-def finite-map-to-set* **.**

from *map-iterator-linord-dom-correct* [*OF iti-s*]
show *set-iterator-linord* (*s-iteratei iteratei s*) (*s-*α α *s*)
unfolding *s-*α*-def s-iteratei-def* **.**
qed

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
fix *s*
assume *invar-s*: *invar s*

from *map-reverse-iterateoi.reverse-iterateoi-rule*[*OF map-it-OK*,*OF invar-s*]
have *iti-s*: *map-iterator-rev-linord* (*iteratei s*) (α *s*) **.**

from *set-iterator-genord.finite-S0*[*OF iti-s*[*unfolded set-iterator-map-rev-linord-def*]]

show *finite* (*s-*α α *s*)
unfolding *s-*α*-def finite-map-to-set* **.**

from *map-iterator-rev-linord-dom-correct* [*OF iti-s*]
show *set-iterator-rev-linord* (*s-iteratei iteratei s*) (*s-*α α *s*)
unfolding *s-*α*-def s-iteratei-def* **.**
qed

lemma *s-ball-correct*:
fixes *ball*
assumes *map-ball* α *invar ball*
shows *set-ball* (*s-*α α) *invar* (*s-ball ball*)
proof −
interpret *map-ball* α *invar ball* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs ball-correct*)
qed

lemma *s-bexists-correct*:
fixes *bexists*
assumes *map-bexists* α *invar bexists*
shows *set-bexists* (*s-*α α) *invar* (*s-bexists bexists*)
proof −
interpret *map-bexists* α *invar bexists* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs bexists-correct*)
qed

lemma *s-size-correct*:
fixes *size*
assumes *map-size* α *invar size*
shows *set-size* (*s-*α α) *invar* (*s-size size*)
proof −
interpret *map-size* α *invar size* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs size-correct*)
qed

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 −
interpret *map-size-abort* α *invar size-abort* **by** *fact*
show *?thesis*
by *unfold-locales*
(*auto simp add*: *s-defs size-abort-correct*)
qed

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 −
  interpret *map-add* α *invar add* **by** *fact*
  show *?thesis*
    by *unfold-locales*
      (*auto simp add*: *s-defs add-correct*)
qed

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 −
  interpret *map-add-dj* α *invar add-dj* **by** *fact*
  show *?thesis*
    by *unfold-locales*
      (*auto simp add*: *s-defs add-dj-correct*)
qed

lemma *finite-set-by-map*:
  assumes *BM*: *finite-map* α *invar*
  shows *finite-set* (*s-*α α) *invar*
proof −
  interpret *finite-map* α *invar* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*unfold s-*α*-def*)
    apply *simp*
    done
qed

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 $i=0 \lor \neg\ c\ \sigma$ *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* $\sigma = \sigma$
  $\neg c\ \sigma \Longrightarrow$ *idx-iteratei-aux get sz i l c f* $\sigma = \sigma$
  $[\![i\neq 0;\ c\ \sigma]\!] \Longrightarrow$ *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)
  apply –
  apply (*subst idx-iteratei-aux.simps*, *simp*)+
  done

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 –
  interpret *list-size* α *invar sz* **by** *fact*
  interpret *list-get* α *invar get* **by** *fact*

  {
    fix *n l*
    assume *A*: *Suc n* $\leq$ *length l*
    hence *B*: *length l* − *Suc n* < *length l* **by** *simp*
    from *A* **have** [*simp*]: *Suc* (*length l* − *Suc n*) = *length l* − *n* **by** *simp*
    from *nth-drop'*[*OF B*, *simplified*] **have**
      drop (*length l* − *Suc n*) *l* = *l*!(*length l* − *Suc n*)#*drop* (*length l* − *n*) *l*
      by *simp*
  } **note** *drop-aux*=*this*

  {
    fix *s c f* σ *i*
    assume *invar s* $i \leq sz\ s$
    hence *idx-iteratei-aux get* (*sz s*) *i s c f* σ = *foldli* (*drop* (*sz s* − *i*) (α *s*)) *c f* σ
    proof (*induct i arbitrary*: σ)

    case *0* **with** *size-correct*[*of s*] **show** *?case* **by** *simp*
  next
    case (*Suc n*)
    note [*simp*, *intro*!] = *Suc.prems*(*1*)
    show *?case* **proof** (*cases c* σ)
      case *False* **thus** *?thesis* **by** *simp*
    next
      case *True*[*simp*, *intro*!]
      show *?thesis* **using** *Suc* **by** (*simp add*: *get-correct size-correct drop-aux*)
    qed
  qed
} **note** *aux=this*

show *?thesis*
  unfolding *idx-iteratei-def*[*abs-def*]
  by (*auto simp add*: *fun-eq-iff aux*[*OF* ‹*invar s*›, *of sz s*, *simplified*])
qed

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*)
using *assms*
unfolding *list-iteratei-def*
by (*simp add*: *idx-iteratei-eq-foldli*)

reverse_iteratei (by get, size)

fun *idx-reverse-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-reverse-iteratei-aux get sz i l c f σ = (
    if $i=0 \vee \neg\ c\ \sigma$ *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*]:
  $i=0 \Longrightarrow$ *idx-reverse-iteratei-aux get sz i l c f* $\sigma = \sigma$
  $\neg c\ \sigma \Longrightarrow$ *idx-reverse-iteratei-aux get sz i l c f* $\sigma = \sigma$
  $[\![i\neq 0;\ c\ \sigma]\!] \Longrightarrow$ *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)
  by (*subst idx-reverse-iteratei-aux.simps*, *simp*)+

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 –
  interpret list-size α invar sz by fact
  interpret list-get α invar get by fact

  {
    fix s c f σ i
    assume invar s   i≤sz s
    hence idx-reverse-iteratei-aux get (sz s) i s c f σ = foldri (take i (α s)) c f σ
    proof (induct i arbitrary: σ)
      case 0 with size-correct[of s] show ?case by simp
    next
      case (Suc n)
      note [simp, intro!] = Suc.prems(1)
      show ?case proof (cases c σ)
        case False thus ?thesis by simp
      next
        case True[simp, intro!]
        show ?thesis using Suc
          by (simp add: get-correct size-correct take-Suc-conv-app-nth)
      qed
    qed
  } note aux=this

  show ?thesis
    unfolding idx-reverse-iteratei-def[abs-def]
    apply (simp add: fun-eq-iff aux[OF ‹invar s›, of sz s, simplified])
    apply (simp add: size-correct[OF ‹invar s›])
  done
qed

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)
using assms
unfolding list-reverse-iteratei-def
by (simp add: idx-reverse-iteratei-eq-foldri)
```

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) (λx res. Suc res) (0::nat)

lemma it-size-correct:
```

assumes *list-iteratei* α *invar it*
shows *list-size* α *invar* (*it-size it*)
proof −
interpret *list-iteratei* α *invar it* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
apply (*unfold it-size-def*)
apply (*simp add*: *iteratei-correct foldli-foldl*)
done
qed

By reverse_iterator

lemma *it-size-correct-rev*:
assumes *list-reverse-iteratei* α *invar it*
shows *list-size* α *invar* (*it-size it*)
proof −
interpret *list-reverse-iteratei* α *invar it* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
apply (*unfold it-size-def*)
apply (*simp add*: *reverse-iteratei-correct foldri-foldr*)
done
qed

3.4.3 Get (by iteratori)

definition *iti-get*:: $('s \Rightarrow ('x,nat \times 'x\ option)\ set\text{-}iterator) \Rightarrow 's \Rightarrow nat \Rightarrow 'x$
where *iti-get iti s i* =
the (*snd* (*iti s*
$(\lambda(i,x).\ x=None)$
$(\lambda x\ (i,\text{-}).\ if\ i=0\ then\ (0,Some\ x)\ else\ (i - 1,None))$
$(i,None)))$

lemma *iti-get-correct*:
assumes *iti-OK*: *list-iteratei* α *invar iti*
shows *list-get* α *invar* (*iti-get iti*)
proof
fix *s i*
assume *invar s* $i < length\ (\alpha\ s)$

from *list-iteratei.iteratei-correct*[*OF iti-OK*, *OF* ‹*invar s*›]
have *iti-s-eq*: *iti s* = *foldli* (α *s*) **by** *simp*

def $l \equiv \alpha\ s$
from ‹$i < length\ (\alpha\ s)$›
show *iti-get iti s i* = α *s* ! *i*
unfolding *iti-get-def iti-s-eq l-def*[*symmetric*]
proof (*induct i arbitrary*: *l*)
case *0*

```
    then obtain x xs where l-eq[simp]: l = x # xs by (cases l, auto)
    thus ?case by simp
  next
    case (Suc i)
    note ind-hyp = Suc(1)
    note Suc-i-le = Suc(2)
    from Suc-i-le obtain x xs where l-eq[simp]: l = x # xs by (cases l, auto)

    from ind-hyp [of xs] Suc-i-le
    show ?case by simp
  qed
qed


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 ⇒ 'i) ⇒ 'a set ⇒ 'i ⇒ 'a set
  where index f s i == { x∈s . f x = i }

lemma indexI: ⟦ x∈s; f x = i ⟧ ⟹ x∈index f s i by (unfold index-def) auto
lemma indexD:
  x∈index f s i ⟹ x∈s
  x∈index f s i ⟹ f x = i
  by (unfold index-def) auto

lemma index-iff[simp]: x∈index f s i ⟷ x∈s ∧ f x = i by (simp add: index-def)
```

3.5.2 Indexing by Map

```
definition index-map :: ('a ⇒ 'i) ⇒ 'a set ⇒ 'i ⇀ '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*
apply (*rule ext*)
apply (*unfold index-def index-map-def im-α-def*)
apply *auto*
done

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* **by** (*unfold ci-invar-def*) *auto*

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*
by (*simp add*: *is-index-def*)

lemma *is-index-correct*: *is-index f s idx* ⟹ *ci-α idx* = *index f s*
by (*unfold is-index-def index-map-def ci-α-def*)
(*simp add*: *index-map-correct*)

```
  abbreviation lookup == idx-lookup mlookup tempty

  lemma lookup-invar': ci-invar m ⟹ tinvar (lookup i m)
    apply (unfold ci-invar-def idx-lookup-def)
    apply (auto split: option.split simp add: m.lookup-correct t.empty-correct)
    done

  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 −
    case goal2 thus ?case using I by (simp add: is-index-def lookup-invar')
  next
    case goal1
    have [simp, intro!]: minvar idx
      using ci-impl-minvar[OF is-index-invar[OF I]]
      by simp
    thus ?case
    proof (cases mlookup i idx)
      case None
      hence [simp]: mα idx i = None by (simp add: m.lookup-correct)
      from is-index-correct[OF I] have index f s i = ci-α idx i by simp
      also have ... = {} by (simp add: ci-α-def ci-α'-def im-α-def)
      finally show ?thesis by (simp add: idx-lookup-def None t.empty-correct)
    next
      case (Some si)
      hence [simp]: mα idx i = Some si by (simp add: m.lookup-correct)
      from is-index-correct[OF I] have index f s i = ci-α idx i by simp
      also have ... = tα si by (simp add: ci-α-def ci-α'-def im-α-def)
      finally show ?thesis by (simp add: idx-lookup-def Some t.empty-correct)
    qed
  qed
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 $\Rightarrow$ *mupdate i* (*tinsert x s*) *m*
)

definition *idx-build* ::
(*unit* $\Rightarrow$ *'m*) $\Rightarrow$
(*'i* $\Rightarrow$ *'m* $\rightharpoonup$ *'t*) $\Rightarrow$
(*'i* $\Rightarrow$ *'t* $\Rightarrow$ *'m* $\Rightarrow$ *'m*) $\Rightarrow$
(*unit* $\Rightarrow$ *'t*) $\Rightarrow$
(*'x* $\Rightarrow$ *'t* $\Rightarrow$ *'t*) $\Rightarrow$
(*'s* $\Rightarrow$ (*'x*,*'m*) *set-iterator*) $\Rightarrow$
(*'x* $\Rightarrow$ *'i*) $\Rightarrow$ *'s* $\Rightarrow$ *'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 t*α *tinvar tempty* +
m: *map-empty m*α *minvar mempty* +
m: *map-update m*α *minvar mupdate* +
t: *set-ins t*α *tinvar tinsert* +
s: *set-iteratei s*α *sinvar siterate*
for *m*α :: *'m* $\Rightarrow$ *'i* $\rightharpoonup$ *'t* **and** *minvar mempty mlookup mupdate*
and *t*α :: *'t* $\Rightarrow$ *'x set* **and** *tinvar tempty tinsert*
and *s*α :: *'s* $\Rightarrow$ *'x set* **and** *sinvar*
and *siterate* :: *'s* $\Rightarrow$ (*'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* (*s*α *s*) (**is** *?T1*) **and**
[*simp*]: *ci-invar* (*idx-build' f s*) (**is** *?T2*)
proof –
have *sinvar s* $\Longrightarrow$
ci-α*'* (*idx-build' f s*) = *index-map f* (*s*α *s*) $\wedge$ *ci-invar* (*idx-build' f s*)
apply (*unfold idx-build-def*)
apply (*rule-tac*
I=λ*it m. ci-*α*' m* = *index-map f* (*s*α *s* − *it*) $\wedge$ *ci-invar m*
in *iterate-rule-P*)
apply *assumption*
apply (*simp add*: *ci-invar-def m.empty-correct*)
apply (*rule ext*)
apply (*unfold ci-*α*'-def index-map-def index-def*)[*1*]

```
    apply (simp add: m.empty-correct)
    defer
    apply simp
    apply (rule conjI)
    defer
    apply (unfold idx-build-stepfun-def)[1]
    apply (auto
      simp add: ci-invar-def m.update-correct m.lookup-correct
              t.empty-correct t.ins-correct Let-def
      split: option.split) [1]

    apply (rule ext)
  proof -
    case (goal1 x it m i)
    hence INV[simp, intro!]: minvar m by (simp add: ci-invar-def)
    from goal1 have
      INVS[simp, intro]: !!q s. mα m q = Some s ⟹ tinvar s
      by (simp add: ci-invar-def)

    show ?case proof (cases i=f x)
      case True[simp]
      show ?thesis proof (cases mα m (f x))
        case None[simp]
        hence idx-build-stepfun' f x m = mupdate i (tinsert x (tempty ())) m
          apply (unfold idx-build-stepfun-def)
          apply (simp add: m.update-correct m.lookup-correct t.empty-correct)
          done
        hence ci-α' (idx-build-stepfun' f x m) i = Some {x}
          by (simp add: m.update-correct
                    t.ins-correct t.empty-correct ci-α'-def)
        also {
          have None = ci-α' m (f x)
            by (simp add: ci-α'-def)
          also from goal1(4) have ... = index-map f (sα s − it) i by simp
          finally have E: {xa ∈ sα s − it. f xa = i} = {}
            by (simp add: index-map-def index-def split: split-if-asm)
          moreover have
            {xa ∈ sα s − (it − {x}). f xa = i}
             = {xa ∈ sα s − it. f xa = i} ∪ {x}
            using goal1(2,3) by auto
          ultimately have Some {x} = index-map f (sα s − (it − {x})) i
            by (unfold index-map-def index-def) auto
        } finally show ?thesis .
      next
        case (Some ss)[simp]
        hence [simp, intro!]: tinvar ss by (simp del: Some)
        hence idx-build-stepfun' f x m = mupdate (f x) (tinsert x ss) m
       by (unfold idx-build-stepfun-def) (simp add: m.update-correct m.lookup-correct)
        hence ci-α' (idx-build-stepfun' f x m) i = Some (insert x (tα ss))
```

by (*simp add*: *m.update-correct t.ins-correct ci-*α'*-def*)
also {
have *Some* ($t\alpha$ *ss*) = *ci-*α' *m* (*f x*)
by (*simp add*: *ci-*α'*-def*)
also from *goal1*(*4*) **have** ... = *index-map f* ($s\alpha$ *s* − *it*) *i* **by** *simp*
finally have *E*: $\{xa \in s\alpha\ s - it.\ f\ xa = i\} = t\alpha\ ss$
by (*simp add*: *index-map-def index-def split*: *split-if-asm*)
moreover have
$\{xa \in s\alpha\ s - (it - \{x\}).\ f\ xa = i\}$
$= \{xa \in s\alpha\ s - it.\ f\ xa = i\} \cup \{x\}$
using *goal1*(*2*,*3*) **by** *auto*
ultimately have
Some (*insert x* ($t\alpha$ *ss*)) = *index-map f* ($s\alpha$ *s* − (*it* − {*x*})) *i*
by (*unfold index-map-def index-def*) *auto*
}
finally show *?thesis* **.**
qed
next
case *False* **hence** *C*: $i \neq f\ x \quad f\ x \neq i$ **by** *simp-all*
have *ci-*α' (*idx-build-stepfun'* *f x m*) *i* = *ci-*α' *m i*
apply (*unfold ci-*α'*-def idx-build-stepfun-def*)
apply (*simp*
split: *option.split-asm option.split*
add: *Let-def m.lookup-correct m.update-correct*
t.ins-correct t.empty-correct C)
done
also from *goal1*(*4*) **have** *ci-*α' *m i* = *index-map f* ($s\alpha$ *s* − *it*) *i* **by** *simp*
also have
$\{xa \in s\alpha\ s - (it - \{x\}).\ f\ xa = i\} = \{xa \in s\alpha\ s - it.\ f\ xa = i\}$
using *goal1*(*2*,*3*) *C* **by** *auto*
hence *index-map f* ($s\alpha$ *s* − *it*) *i* = *index-map f* ($s\alpha$ *s* − (*it*−{*x*})) *i*
by (*unfold index-map-def index-def*) *simp*
finally show *?thesis* **.**
qed
qed
with *I* **show** *?T1 ?T2* **by** *auto*
qed

lemma *idx-build-correct*:
sinvar s $\Longrightarrow$ *is-index f* ($s\alpha$ *s*) (*idx-build'* *f s*)
by (*simp add*: *idx-build-correct'* *index-map-correct ci-*α*-def is-index-def*)

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 *mα* :: 'm ⇒ 'i ⇀ 't
constrains *tα* :: 't ⇒ 'x set
constrains *sα* :: 's ⇒ 'x set
constrains *siterate* :: 's ⇒ ('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 −
interpret *m*: *map-empty mα minvar mempty* **by** *fact*
interpret *m*: *map-lookup mα minvar mlookup* **by** *fact*
interpret *m*: *map-update mα minvar mupdate* **by** *fact*
interpret *s*: *set-empty tα tinvar tempty* **by** *fact*
interpret *s*: *set-ins tα tinvar tinsert* **by** *fact*
interpret *t*: *set-iteratei sα sinvar siterate* **by** *fact*

interpret *idx-build-env*
mα minvar mempty mlookup mupdate
tα tinvar tempty tinsert
sα sinvar siterate
by *unfold-locales*
from *idx-build-correct*[*OF INV*] **show** *?thesis*
by (*unfold idx-invar-def idx-build-def*)
qed

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 −
interpret *m*: *map-lookup mα minvar mlookup* **by** *fact*
interpret *s*: *set-empty tα tinvar tempty* **by** *fact*

interpret *index-env*
  mα minvar mlookup
  tα tinvar tempty
  by *unfold-locales*

from *lookup-correct*[*OF INV*[*unfolded idx-invar-def*], *of k*] **show** *?T1 ?T2*
  by (*simp-all add*: *idx-invar-def idx-lookup-def*)
qed

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* ⇒ (′*x*,*nat* × ′*m*) *set-iterator*) ⇒
    (*unit* ⇒ ′*m*) ⇒ (′*x* ⇒ *nat* ⇒ ′*m* ⇒ ′*m*) ⇒
    ′*s* ⇒ ′*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*}
  by (*unfold inatseg-def*)
    auto

lemma *map-to-nat-correct*:
  fixes α*1* :: ′*s* ⇒ ′*x set*
  fixes α*2* :: ′*m* ⇒ ′*x* ⇀ *nat*
  assumes *set-iteratei* α*1 invar1 iterate1*
  assumes *map-empty* α*2 invar2 empty2*
  assumes *map-update* α*2 invar2 update2*

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

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

shows
— All elements have got a number
dom (α*2 nm*) = α*1 s* (**is** *?T1*) **and**
— No two elements got the same number
[*rule-format*]: *inj-on* (α*2 nm*) (α*1 s*) (**is** *?T2*) **and**
— Numbering is inatseg
[*rule-format*]: *inatseg* (*ran* (α*2 nm*)) (**is** *?T3*) **and**
— The result satisfies the map invariant
invar2 nm (**is** *?T4*)
proof −
  interpret *s1*: *set-iteratei* α*1 invar1 iterate1* **by** *fact*
  interpret *m2*: *map-empty* α*2 invar2 empty2* **by** *fact*
  interpret *m2*: *map-update* α*2 invar2 update2* **by** *fact*

 have *i-aux*: !!*m S S′ k v*. ⟦*inj-on m S*; *S′* = *insert k S*; *v*∉*ran m*⟧
   ⟹ *inj-on* (*m*(*k*↦*v*)) *S′*
  apply (*rule inj-onI*)
  apply (*simp split*: *split-if-asm*)
  apply (*simp add*: *ran-def*)
  apply (*simp add*: *ran-def*)
  apply *blast*
  apply (*blast dest*: *inj-onD*)
  done

 have *?T1* ∧ *?T2* ∧ *?T3* ∧ *?T4*
  apply (*unfold nm-def map-to-nat-def*)
  apply (*rule-tac I*=λ*it* (*c*,*m*).
   invar2 m ∧
   dom (α*2 m*) = α*1 s* − *it* ∧
   inj-on (α*2 m*) (α*1 s* − *it*) ∧
   (*ran* (α*2 m*) = {*i*. *i*<*c*})
   in *s1*.*iterate-rule-P*)
  apply *simp*
  apply (*simp add*: *m2*.*empty-correct*)
  apply (*case-tac* σ)
  apply (*simp add*: *m2*.*empty-correct m2*.*update-correct*)
  apply (*intro conjI*)
  apply *blast*
  apply *clarify*
  apply (*rule-tac m*=α*2 ba* **and**
   k=*x* **and** *v*=*aa* **and**
   S′=(α*1 s* − (*it* − {*x*})) **and**
   S=(α*1 s* − *it*)
   in *i-aux*)

```
    apply auto [3]
    apply auto [1]
    apply (case-tac σ)
    apply (auto simp add: inatseg-def)
    done
  thus ?T1 ?T2 ?T3 ?T4 by auto
qed
```

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 −
  interpret set-iteratei α invar iterate by fact
  interpret list-empty αl invarl emp by fact
  interpret list-enqueue αl invarl enq by fact
  have ?T1 ∧ ?T2 ∧ ?T3
    apply (unfold it-set-to-List-enq-def)
    apply (rule-tac
      I=λit F. set (αl F) = α S − it ∧ invarl F ∧ distinct (αl F)
      in iterate-rule-P)
    apply (auto simp add: enqueue-correct empty-correct)
    done
  thus ?T1 ?T2 ?T3 by auto
qed
```

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) (λx F.
push x F) (emp ())

lemma it-set-to-List-push-correct:
  assumes set-iteratei α invar iterate
  assumes list-empty αl invarl emp
```

```
  assumes list-push αl invarl push
  assumes [simp]: invar S
  shows
    set (αl (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 (αl (it-set-to-List-push iterate emp push S)) (is ?T3)
proof −
  interpret set-iteratei α invar iterate by fact
  interpret list-empty αl invarl emp by fact
  interpret list-push αl invarl push by fact
  have ?T1 ∧ ?T2 ∧ ?T3
    apply (unfold it-set-to-List-push-def)
    apply (rule-tac
      I=λit F. set (αl F) = α S − it ∧ invarl F ∧ distinct (αl F)
      in iterate-rule-P)
    apply (auto simp add: push-correct empty-correct)
    done
  thus ?T1 ?T2 ?T3 by auto
qed
```

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) (λk m. up-dj k (f k) m) (emp ())

lemma it-dom-fun-to-map-correct:
  fixes α1 :: 'm ⇒ 'k ⇒ 'v option
  fixes α2 :: 's ⇒ 'k set
  assumes s-it: set-iteratei α2 invar2 s-it
  assumes m-up: map-update-dj α1 invar1 up-dj
  assumes m-emp: map-empty α1 invar1 emp
  assumes INV: invar2 s
  shows α1 (it-dom-fun-to-map s-it up-dj emp s f) k = (if k ∈ α2 s then Some (f
k) else None)
  and invar1 (it-dom-fun-to-map s-it up-dj emp s f)
proof −
  interpret s-it: set-iteratei α2 invar2 s-it using s-it .
  interpret m: map-update-dj α1 invar1 up-dj using m-up .
  interpret m: map-empty α1 invar1 emp using m-emp .


  have α1 (it-dom-fun-to-map s-it up-dj emp s f) k = (if k ∈ α2 s then Some (f
k) else None) ∧
    invar1 (it-dom-fun-to-map s-it up-dj emp s f)
```

unfolding *it-dom-fun-to-map-def*
apply (*rule s-it.iterate-rule-P*[**where**
$I = \lambda it\ res.\ invar1\ res \wedge (\forall\, k.\ \alpha 1\ res\ k = (if\ (k \in (\alpha 2\ s) - it)\ then\ Some\ (f\ k)\ else\ None))$
])
apply (*simp add*: *INV*)

apply (*simp add*: *m.empty-correct*)

apply (*subgoal-tac* $x \notin dom\ (\alpha 1\ \sigma)$)

apply (*auto simp*: *INV m.empty-correct m.update-dj-correct*) []

apply *auto*
done
thus $\alpha 1$ (*it-dom-fun-to-map s-it up-dj emp s f*) $k = (if\ k \in \alpha 2\ s\ then\ Some\ (f\ k)\ else\ None)$
and *invar1* (*it-dom-fun-to-map s-it up-dj emp s f*)
by *auto*
qed

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* × '*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* **by** (*induct a*) *auto*
lemma *p-min-le-neut*[*simp*]: *p-min Infty a* = *a* **by** (*induct a*) *auto*
lemma *p-min-asso*: *p-min* (*p-min a b*) *c* = *p-min a* (*p-min b c*)
apply(*induct a b rule*: *p-min.induct*)
apply *auto*
apply (*induct c*)
apply *auto*
apply (*induct c*)
apply *auto*
done
lemma *lp-mono*: *class.monoid-add p-min Infty*
by *unfold-locales* (*auto simp add*: *p-min-asso*)

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

instance by
intro-classes
(*auto simp add*: *p-min-asso zero-def plus-def*)
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*)

by (*induct x y rule*: *p-less.induct*) *auto*

lemma *p-order-refl* : *p-less-eq x x*
by (*induct x*) *auto*

lemma *p-le-inf* : *p-less-eq Infty x* $\Longrightarrow$ *x* = *Infty*
by (*induct x*) *auto*

lemma *p-order-trans* : $[\![$*p-less-eq x y*; *p-less-eq y z*$]\!]$ $\Longrightarrow$ *p-less-eq x z*
apply (*induct y z rule*: *p-less.induct*)
apply *auto*
apply (*induct x*)
apply *auto*
apply (*cases x*)
apply *auto*
apply(*induct x*)
apply (*auto simp add*: *p-le-inf*)
apply (*metis p-le-inf p-less-eq.simps*(*2*))
apply (*metis p-le-inf p-less-eq.simps*(*2*))
done

lemma *p-linear2* : *p-less-eq x y* $\vee$ *p-less-eq y x*
apply (*induct x y rule*: *p-less-eq.induct*)
apply *auto*
done

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

instance
apply (*intro-classes*)
apply (*simp only*: *p-less-le-not-le pless-def plesseq-def*)
apply (*simp only*: *p-order-refl plesseq-def pless-def*)
apply (*simp only*: *plesseq-def*)
apply (*metis p-order-trans*)
done

end

Operations

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

definition *alprio-invar* :: (*'s* $\Rightarrow$ (*unit* $\times$ (*'c*, *'d*::*linorder*) *Prio*) *list*)

$\Rightarrow$ (*'s* $\Rightarrow$ *bool*) $\Rightarrow$ *'s* $\Rightarrow$ *bool*
where
alprio-invar α *invar al* == *invar al* $\wedge$ ($\forall$ *x*$\in$*set* (α *al*). *snd x*$\neq$*Infty*)

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

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

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

definition *alprio-find* :: (*'s* $\Rightarrow$ (*'e*,*'a*::*linorder*) *Prio*) $\Rightarrow$ *'s* $\Rightarrow$ (*'e* $\times$ *'a*)
where
alprio-find annot s = *p-unwrap* (*annot s*)

definition *alprio-delete* :: (((*'e*,*'a*::*linorder*) *Prio* $\Rightarrow$ *bool*)
$\Rightarrow$ (*'e*,*'a*) *Prio* $\Rightarrow$ *'s* $\Rightarrow$ (*'s* $\times$ (*unit* $\times$ (*'e*,*'a*) *Prio*) $\times$ *'s*))
$\Rightarrow$ (*'s* $\Rightarrow$ (*'e*,*'a*) *Prio*) $\Rightarrow$ (*'s* $\Rightarrow$ *'s* $\Rightarrow$ *'s*) $\Rightarrow$ *'s* $\Rightarrow$ *'s*
where
alprio-delete splits annot app s = (*let* (*l*, - , *r*)
= *splits* (λ *x*. *x*$\leq$(*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*)
by (*induct l*) (*auto simp add*: *add-assoc*)

lemma *p-linear*: (*x*::(*'e*, *'a*::*linorder*) *Prio*) $\leq$ *y* $\vee$ *y* $\leq$ *x*
by (*unfold plesseq-def*) (*simp only*: *p-linear2*)

lemma *p-min-mon*: $(x::(('e,'a::linorder)\ Prio)) \le y \Longrightarrow (z + x) \le y$
apply (*unfold plus-def plesseq-def*)
apply (*induct x y rule*: *p-less-eq.induct*)
apply (*auto*)
apply (*induct z*)
apply (*auto*)
done

lemma *p-min-mon2*: *p-less-eq x y* $\Longrightarrow$ *p-less-eq* (*p-min z x*) *y*
apply (*induct x y rule*: *p-less-eq.induct*)
apply (*auto*)
apply (*induct z*)
apply (*auto*)
done

lemma *ls-min*: $\forall x \in set\ (xs::\ ('e,'a::linorder)\ Prio\ list)\ .\ listsum\ xs \le x$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons a ins*) **thus** *?case*
  apply (*auto simp add*: *plus-def plesseq-def*)
  apply (*cases a*)
  apply *auto*
  apply (*cases listsum ins*)
  apply *auto*
  apply (*case-tac x*)
  apply *auto*
  apply (*cases a*)
  apply *auto*
  apply (*cases listsum ins*)
  apply *auto*
  done
qed

lemma *infadd*: $x \neq Infty \Longrightarrow x + y \neq Infty$
apply (*unfold plus-def*)
apply (*induct x y rule*: *p-min.induct*)
apply *auto*
done

lemma *prio-selects-one*: $a+b = a \vee a+b=(b::('e,'a::linorder)\ Prio)$
  apply (*simp add*: *plus-def*)
  apply (*cases* (*a*,*b*) *rule*: *p-min.cases*)
  apply *simp-all*
  done

lemma *listsum-in-set*: $(l::('x \times ('e,'a::linorder)\ Prio)\ list) \neq [] \Longrightarrow$
listsum (*map snd l*) $\in$ *set* (*map snd l*)
apply (*induct l*)
apply *simp*
apply (*case-tac l*)
apply *simp*
using *prio-selects-one*
apply *auto*
apply *force*
apply *force*
done

lemma *p-unwrap-less-sum*: *snd* (*p-unwrap* ((*Prio e aa*) + *b*)) $\leq$ *aa*
apply (*cases b*)
apply (*auto simp add*: *plus-def*)
done

lemma *prio-add-alb*: $\neg$ *b* $\leq$ (*a*::('*e*,'*a*::*linorder*)*Prio*) $\Longrightarrow$ *b* + *a* = *a*
by (*auto simp add*: *plus-def*, *cases* (*a*,*b*) *rule*: *p-min.cases*) (*auto simp add*: *plesseq-def*)

lemma *prio-add-alb2*: (*a*::('*e*,'*a*::*linorder*)*Prio*) $\leq$ *a* + *b* $\Longrightarrow$ *a* + *b* = *a*
by (*auto simp add*: *plus-def*, *cases* (*a*,*b*) *rule*: *p-min.cases*) (*auto simp add*: *plesseq-def*)

lemma *prio-add-abc*:
assumes (*l*::('*e*,'*a*::*linorder*)*Prio*) + *a* $\leq$ *c*
and $\neg$ *l* $\leq$ *c*
shows $\neg$ *l* $\leq$ *a*
proof (*rule ccontr*)
assume $\neg$ $\neg$ *l* $\leq$ *a*
with *assms* **have** *l* + *a* = *l*
apply (*auto simp add*: *plus-def plesseq-def*)
apply (*cases* (*l*,*a*) *rule*: *p-less-eq.cases*)
apply *auto*
done
with *assms* **show** *False* **by** *simp*
qed

lemma *prio-add-abc2*:
assumes (*a*::('*e*,'*a*::*linorder*)*Prio*) $\leq$ *a* + *b*
shows *a* $\leq$ *b*
proof (*rule ccontr*)
assume *ann*: $\neg$ *a* $\leq$ *b*
hence *a* + *b* = *b*
apply (*auto simp add*: *plus-def plesseq-def*)
apply (*cases* (*a*,*b*) *rule*: *p-min.cases*)
apply *auto*

done
thus *False* **using** *assms ann* **by** *simp*
qed

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 −
interpret *al-empty* α *invar empt* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
apply (*unfold alprio-invar-def*)
apply *auto*
apply (*unfold alprio-empty-def*)
apply (*auto simp add*: *empty-correct*)
apply (*unfold alprio-*α*-def*)
apply *auto*
apply (*simp only*: *empty-correct*)
done
qed

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 −
interpret *al-isEmpty* α *invar isEmpty* **by** *fact*
show *?thesis* **by** (*unfold-locales*) (*auto simp add*: *alprio-defs isEmpty-correct*)
qed

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 −
interpret *al-consl* α *invar consl* **by** *fact*
show *?thesis* **by** *unfold-locales* (*auto simp add*: *alprio-defs consl-correct*)
qed

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 −
interpret *al-app* α *invar app* **by** *fact*

```
  show ?thesis by unfold-locales (auto simp add: alprio-defs app-correct)
qed
```

Find

```
lemma annot-not-inf :
  assumes (alprio-invar α invar) s
  and (alprio-α α) s ≠ {#}
  and al-annot α invar annot
  shows annot s ≠ Infty
proof −
  interpret al-annot α invar annot by fact
  show ?thesis
  proof −
    from assms(1) have invs: invar s by (simp add: alprio-defs)
    from assms(2) have sne: set (α s) ≠ {}
    proof (cases set (α s) = {})
      case True
      hence α s = [] by simp
      hence (alprio-α α) s = {#} by (simp add: alprio-defs)
      from this assms(2) show ?thesis by simp
    next
      case False thus ?thesis by simp
    qed
    hence (α s) ≠ [] by simp
    hence ∃ x xs. (α s) = x # xs by (cases α s) auto
    from this obtain x xs where [simp]: (α s) = x # xs by blast
    from this assms(1) have snd x ≠ Infty by (auto simp add: alprio-defs)
    hence listsum (map snd (α s)) ≠ Infty by (auto simp add: infadd)
    thus annot s ≠ Infty using annot-correct invs by simp
  qed
qed

lemma annot-in-set:
  assumes (alprio-invar α invar) s
  and (alprio-α α) s ≠ {#}
  and al-annot α invar annot
  shows p-unwrap (annot s) ∈# ((alprio-α α) s)
proof −
  interpret al-annot α invar annot by fact
  from assms(2) have snn: α s ≠ [] by (auto simp add: alprio-defs)
  from assms(1) have invs: invar s by (simp add: alprio-defs)
  hence ans: annot s = listsum (map snd (α s)) by (simp add: annot-correct)
  let ?P = map snd (α s)
  have annot s ∈ set ?P
    by (unfold ans) (rule listsum-in-set[OF snn])
  hence p-unwrap (annot s) ∈ set (map p-unwrap ?P)
    by (metis image-iff in-set-conv-decomp set-map split-list-last)
  thus ?thesis
```

```
  by (metis mem-set-multiset-eq alprio-α-def)
qed

lemma  listsum-less-elems: ∀ x∈set xs. snd x ≠ Infty ⟹
  ∀ y∈set-of (multiset-of (map p-unwrap (map snd xs))).
            snd (p-unwrap (listsum (map snd xs))) ≤ snd y
  proof (induct xs)
  case Nil thus ?case by simp
  next
  case (Cons a as) thus ?case
    apply auto
    apply (cases (snd a) rule: p-unwrap.cases)
    apply auto
    apply (cases listsum (map snd as))
    apply auto
    apply (metis linorder-linear p-min-re-neut
      p-unwrap.simps plus-def [abs-def] snd-eqD)
    apply (auto simp add: p-unwrap-less-sum)
    apply (unfold plus-def)
    apply (cases (snd a, listsum (map snd as)) rule: p-min.cases)
    apply auto
    apply (cases map snd as)
    apply (auto simp add: infadd)
    done
qed

lemma alprio-find-correct:
  assumes  al-annot α invar annot
  shows prio-find (alprio-α α) (alprio-invar α invar) (alprio-find annot)
proof –
  interpret al-annot α invar annot by fact
  show ?thesis
    apply unfold-locales
    apply (rule conjI)
    apply (insert assms)
    apply (unfold alprio-find-def)
    apply (simp add:annot-in-set)
    apply (unfold alprio-defs)
    apply (simp add: annot-correct)
    apply (auto simp add: listsum-less-elems)
    done
qed
```

Delete

```
lemma delpred-mon:
  ∀ (a:: ('e, 'a::linorder) Prio) b. ((λ x. x ≤ y) a
    ⟶ (λ x. x ≤ y) (a + b))
proof (intro impI allI)
```

fix *a b*
show $a \leq y \Longrightarrow a + b \leq y$
apply (*induct a b rule*: *p-less.induct*)
apply (*auto simp add*: *p-less-eq-def plus-def*)
apply (*metis linorder-linear order-trans*
p-linear p-min.simps(*4*) *p-min-mon plus-def prio-selects-one*)
apply (*metis order-trans p-linear p-min-mon p-min-re-neut plus-def*)
done
qed

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 −
interpret *al-splits* α *invar splits* **by** *fact*
let *?P* = $\lambda x.\ x \leq$ *annot s*
obtain *l p r* **where**
[*simp*]:*splits ?P Infty s* = (*l*, *p*, *r*)
by (*cases splits ?P Infty s*) *auto*
obtain *e a* **where**
p = (*e*, *a*)
by (*cases p*, *blast*)
hence
lear:*splits ?P Infty s* = (*l*, (*e*,*a*), *r*)
by *simp*
from *annot-not-inf*[*OF assms*(*1*) *assms*(*3*) *assms*(*2*)] **have**
annot s $\neq$ *Infty* **.**
hence
sv1: ¬ *Infty* $\leq$ *annot s*
by (*simp add*: *plesseq-def*, *cases annot s*, *auto*)
from *assms*(*1*) **have**
invs: *invar s*
unfolding *alprio-invar-def* **by** *simp*
interpret *al-annot* α *invar annot* **by** *fact*
from *invs* **have**
sv2: *Infty* + *listsum* (*map snd* (α *s*)) $\leq$ *annot s*
by (*auto simp add*: *annot-correct plus-def*
plesseq-def p-min-le-neut p-order-refl)
note *sp* = *splits-correct*[*of s ?P Infty l e a r*]
note *dp* = *delpred-mon*[*of annot s*]
from *sp*[*OF invs dp sv1 sv2 lear*] **have**
invlr: *invar l* ∧ *invar r* **and**
alr: α *s* = α *l* @ (*e*, *a*) # α *r*
by *auto*

interpret *al-app* α *invar app* **by** *fact*
from *invlr app-correct* **have**
invapplr: *invar* (*app l r*)
by *simp*
from *invlr app-correct* **have**
sr: α (*app l r*) = (α *l*) @ (α *r*)
by *simp*
from *alr* **have**
set (α *s*) $\supseteq$ (*set* (α *l*) *Un set* (α *r*))
by *auto*
with *app-correct*[*of l r*] *invlr* **have**
set (α *s*) $\supseteq$ *set* (α (*app l r*)) **by** *auto*
with *invapplr assms*(*1*)
show *?thesis*
unfolding *alprio-defs* **by** *auto*
qed

lemma *listsum-elem*:
assumes *ins* = *l* @ (*a*::('*e*,'*a*::*linorder*)*Prio*) # *r*
and $\neg$ *listsum l* $\leq$ *listsum ins*
and *listsum l* + *a* $\leq$ *listsum ins*
shows *a* = *listsum ins*
proof −
have $\neg$ *listsum l* $\leq$ *a* **using** *assms prio-add-abc* **by** *simp*
hence *lpa*: *listsum l* + *a* = *a* **using** *prio-add-alb* **by** *auto*
hence *als*: *a* $\leq$ *listsum ins* **using** *assms*(*3*) **by** *simp*
have *listsum ins* = *a* + *listsum r*
using *lpa listsum-split*[*of l a r*] *assms*(*1*) **by** *auto*
thus *?thesis* **using** *prio-add-alb2*[*of a listsum r*] *prio-add-abc2 als*
by *auto*
qed

lemma *alpriodel-right*:
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-*α α (*alprio-delete splits annot app s*) =
alprio-α α *s* − {#*p-unwrap* (*annot s*)#}
proof −
interpret *al-splits* α *invar splits* **by** *fact*
let *?P* = λ*x*. *x* $\leq$ *annot s*
obtain *l p r* **where**
[*simp*]:*splits ?P Infty s* = (*l*, *p*, *r*)
by (*cases splits ?P Infty s*) *auto*
obtain *e a* **where**
p = (*e*, *a*)

by (*cases p*, *blast*)
hence
lear:*splits ?P Infty s* = (*l*, (*e*,*a*), *r*)
by *simp*
from *annot-not-inf*[*OF assms*(*1*) *assms*(*3*) *assms*(*2*)] **have**
annot s ≠ *Infty* **.**
hence
sv1: ¬ *Infty* ≤ *annot s*
by (*simp add*: *plesseq-def*, *cases annot s*, *auto*)
from *assms*(*1*) **have**
invs: *invar s*
unfolding *alprio-invar-def* **by** *simp*
interpret *al-annot* α *invar annot* **by** *fact*
from *invs* **have**
sv2: *Infty* + *listsum* (*map snd* (α *s*)) ≤ *annot s*
by (*auto simp add*: *annot-correct plus-def*
plesseq-def p-min-le-neut p-order-refl)
note *sp* = *splits-correct*[*of s ?P Infty l e a r*]
note *dp* = *delpred-mon*[*of annot s*]

from *sp*[*OF invs dp sv1 sv2 lear*] **have**
invlr: *invar l* ∧ *invar r* **and**
alr: α *s* = α *l* @ (*e*, *a*) # α *r* **and**
anlel: ¬ *listsum* (*map snd* (α *l*)) ≤ *annot s* **and**
aneqa: (*listsum* (*map snd* (α *l*)) + *a*) ≤ *annot s*
by (*auto simp add*: *plus-def zero-def*)
have *mapalr*: *map snd* (α *s*) = (*map snd* (α *l*)) @ *a* # (*map snd* (α *r*))
using *alr* **by** *simp*
note *lsa* = *listsum-elem*[*of map snd* (α *s*) *map snd* (α *l*) *a* *map snd* (α *r*)]
note *lsa2* = *lsa*[*OF mapalr*]
hence *a-is-annot*: *a* = *annot s*
using *annot-correct*[*OF invs*] *anlel aneqa* **by** *auto*
have *map p-unwrap* (*map snd* (α *s*)) =
(*map p-unwrap* (*map snd* (α *l*))) @ (*p-unwrap a*)
(*map p-unwrap* (*map snd* (α *r*)))
using *alr* **by** *simp*
hence *alpriolst*: (*alprio-*α α *s*) = (*alprio-*α α *l*) +{# *p-unwrap a* #}+ (*alprio-*α
α *r*)
unfolding *alprio-defs*
by (*simp add*: *algebra-simps*)
interpret *al-app* α *invar app* **by** *fact*
from *alpriolst* **show** *?thesis* **using** *app-correct*[*of l r*] *invlr a-is-annot*
by (*auto simp add*: *alprio-defs algebra-simps*)
qed

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 −
interpret *al-annot* α *invar annot* **by** *fact*
interpret *al-splits* α *invar splits* **by** *fact*
interpret *al-app* α *invar app* **by** *fact*
show *?thesis*
apply *intro-locales*
apply (*rule alprio-find-correct*,*simp add*: *assms*)
apply *unfold-locales*
apply (*insert assms*)
apply (*simp add*: *alpriodel-invar*)
apply (*simp add*: *alpriodel-right alprio-find-def*)
done
qed

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

[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.

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 | LP 'e 'a*

fun *p-unwrap :: ('e,'a) LP ⇒ ('e × 'a)* **where**
p-unwrap (LP e a) = (e , a)

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

Instantiation of classes

lemma *p-min-re-neut[simp]: p-min a Infty = a* **by** *(induct a) auto*
lemma *p-min-le-neut[simp]: p-min Infty a = a* **by** *(induct a) auto*
lemma *p-min-asso: p-min (p-min a b) c = p-min a (p-min b c)*
apply(*induct a b rule: p-min.induct*)
apply (*auto*)
apply (*induct c*)
apply (*auto*)
apply (*metis min-max.sup-assoc*)
apply (*metis min-max.inf-assoc*)
done

lemma *lp-mono: class.monoid-add p-min Infty* **by** *unfold-locales (auto simp add: p-min-asso)*

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

instance by

```
  intro-classes
(auto simp add: p-min-asso zero-def plus-def)
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)
  by (induct x y rule: p-less.induct) auto

lemma p-order-refl : p-less-eq x x
  by (induct x) auto

lemma p-le-inf : p-less-eq Infty x ⟹ x = Infty
  by (induct x) auto

lemma p-order-trans : ⟦p-less-eq x y; p-less-eq y z⟧ ⟹ p-less-eq x z
  apply (induct y z rule: p-less.induct)
  apply auto
  apply (induct x)
  apply auto
  apply (cases x)
  apply auto
  apply(induct x)
  apply (auto simp add: p-le-inf)
  apply (metis p-le-inf p-less-eq.simps(2))
  apply (metis p-le-inf p-less-eq.simps(2))
  done

lemma p-linear2 : p-less-eq x y ∨ p-less-eq y x
  apply (induct x y rule: p-less-eq.induct)
  apply auto
  done

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

instance
  apply (intro-classes)
  apply (simp only: p-less-le-not-le pless-def plesseq-def)
```

```
  apply (simp only: p-order-refl plesseq-def pless-def)
  apply (simp only: plesseq-def)
  apply (metis p-order-trans)
  done

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* :: $((('e::linorder,'a::linorder)\ LP \Rightarrow bool) \Rightarrow ('e,'a)\ LP$
$\Rightarrow 's \Rightarrow ('s \times (unit \times ('e,'a)\ LP) \times 's))$
$\Rightarrow ('s \Rightarrow ('e,'a)\ LP)$
$\Rightarrow ('s \Rightarrow 's \Rightarrow 's)$
$\Rightarrow 's$
$\Rightarrow 'e \times 'a \times 's$
where
aluprio-pop splits annot app s =
(*let* (*l*, (-,*lp*) , *r*) = *splits* (λ *x*. $x \leq$ (*annot s*)) *Infty s*
in
(*case lp of*
(*LP e a*) $\Rightarrow$
(*e*, *a*, *app l r*)))

definition *aluprio-prio* ::
$((('e::linorder,'a::linorder)\ LP \Rightarrow bool) \Rightarrow ('e,'a)\ LP \Rightarrow 's$
$\Rightarrow ('s \times (unit \times ('e,'a)\ LP) \times 's))$
$\Rightarrow ('s \Rightarrow ('e,'a)\ LP)$
$\Rightarrow ('s \Rightarrow bool)$
$\Rightarrow 's \Rightarrow 'e \Rightarrow 'a\ option$
where

aluprio-prio splits annot isEmpty s e =
(*if e-less-eq e* (*annot s*) $\wedge \neg$ *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*: $(x::('e,\ 'a::linorder)\ LP) \leq y \vee y \leq x$
by (*unfold plesseq-def*) (*simp only*: *p-linear2*)

lemma *e-less-eq-mon1*: *e-less-eq e x* $\Longrightarrow$ *e-less-eq e* $(x + y)$
apply (*cases x*)
apply (*auto simp add*: *plus-def*)
apply (*cases y*)
apply (*auto simp add*: *min-max.le-supI1*)
done
lemma *e-less-eq-mon2*: *e-less-eq e y* $\Longrightarrow$ *e-less-eq e* $(x + y)$
apply (*cases x*)
apply (*auto simp add*: *plus-def*)
apply (*cases y*)
apply (*auto simp add*: *min-max.le-supI2*)
done
lemmas *e-less-eq-mon* =
e-less-eq-mon1
e-less-eq-mon2

lemma *p-less-eq-mon*:
$(x::('e::linorder,'a::linorder)\ LP) \leq z \Longrightarrow (x + y) \leq z$
apply(*cases y*)
apply(*auto simp add*: *plus-def*)
apply (*cases x*)
apply (*cases z*)
apply (*auto simp add*: *plesseq-def*)
apply (*cases z*)
apply (*auto simp add*: *min-max.le-infI1*)
done

lemma *p-less-eq-lem1*:
$[\![\neg\ (x::('e::linorder,'a::linorder)\ LP) \leq z;$
$(x + y) \leq z]\!]$
$\Longrightarrow y \leq z$
apply (*cases x,auto simp add*: *plus-def*)
apply (*cases y*, *auto*)
apply (*cases z*, *auto simp add*: *plesseq-def*)
apply (*metis min-le-iff-disj*)
done

lemma *infadd*: $x \neq Infty \Longrightarrow x + y \neq Infty$
apply (*unfold plus-def*)
apply (*induct x y rule*: *p-min.induct*)
apply *auto*
done

lemma *e-less-eq-listsum*:
  $[\![\neg$ *e-less-eq e* (*listsum xs*)$]\!] \Longrightarrow \forall\, x \in$ *set xs*. $\neg$ *e-less-eq e x*
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a xs*)
  hence $\neg$ *e-less-eq e* (*listsum xs*) **by** (*auto simp add*: *e-less-eq-mon*)
  hence *v1*: $\forall\, x \in$*set xs*. $\neg$ *e-less-eq e x* **using** *Cons.hyps* **by** *simp*
  from *Cons.prems* **have** $\neg$ *e-less-eq e a* **by** (*auto simp add*: *e-less-eq-mon*)
  with *v1* **show** $\forall\, x \in$*set* (*a#xs*). $\neg$ *e-less-eq e x* **by** *simp*
qed

lemma *e-less-eq-p-unwrap*:
  $[\![x \neq$ *Infty*;$\neg$ *e-less-eq e x*$]\!] \Longrightarrow$ *fst* (*p-unwrap x*) $<$ *e*
  by (*cases x*) *auto*

lemma *e-less-eq-refl* :
  b $\neq$ *Infty* $\Longrightarrow$ *e-less-eq* (*fst* (*p-unwrap b*)) *b*
  by (*cases b*) *auto*

lemma *e-less-eq-listsum2*:
  assumes
  $\forall\, x \in$*set* (αs). *snd x* $\neq$ *Infty*
  ((), *b*) $\in$ *set* (αs)
  shows *e-less-eq* (*fst* (*p-unwrap b*)) (*listsum* (*map snd* (αs)))
  apply(*insert assms*)
  apply (*induct* αs)
  apply (*auto simp add*: *zero-def e-less-eq-mon e-less-eq-refl*)
  done

lemma *e-less-eq-lem1*:
  $[\![\neg$ *e-less-eq e a*;*e-less-eq e* (*a* + *b*)$]\!] \Longrightarrow$ *e-less-eq e b*
  apply (*auto simp add*: *plus-def*)
  apply (*cases a*)
  apply *auto*
  apply (*cases b*)
  apply *auto*
  apply (*metis le-max-iff-disj*)
  done

lemma *p-unwrap-less-sum*: *snd* (*p-unwrap* ((*LP e aa*) + *b*)) $\leq$ *aa*
  apply (*cases b*)
  apply (*auto simp add*: *plus-def*)
done

lemma *listsum-less-elems*: $\forall\, x \in$*set xs*. *snd x* $\neq$ *Infty* $\Longrightarrow$
  $\forall\, y \in$*set* (*map snd* (*map p-unwrap* (*map snd xs*))).

```
          snd (p-unwrap (listsum (map snd xs))) ≤ y
  proof (induct xs)
  case Nil thus ?case by simp
  next
  case (Cons a as) thus ?case
    apply auto
    apply (cases (snd a) rule: p-unwrap.cases)
    apply auto
    apply (cases listsum (map snd as))
    apply auto
    apply (metis linorder-linear p-min-re-neut p-unwrap.simps
      plus-def[abs-def] snd-eqD)
    apply (auto simp add: p-unwrap-less-sum)
    apply (unfold plus-def)
    apply (cases (snd a, listsum (map snd as)) rule: p-min.cases)
    apply auto
    apply (cases map snd as)
    apply (auto simp add: infadd)
    apply (metis min-max.le-infI2 snd-conv)
    done
qed

lemma distinct-sortet-list-app:
  ⟦sorted xs; distinct xs; xs = as @ b # cs⟧
  ⟹ ∀ x∈ set cs. b < x
  by (metis distinct.simps(2) distinct-append
    linorder-antisym-conv2 sorted-Cons sorted-append)

lemma distinct-sorted-list-lem1:
  assumes
  sorted xs
  sorted ys
  distinct xs
  distinct ys
   ∀ x ∈ set xs. x < e
   ∀ y ∈ set ys. e < y
  shows
  sorted (xs @ e # ys)
  distinct (xs @ e # ys)
proof −
  from assms (5,6)
  have ∀x∈set xs. ∀y∈set ys. x ≤ y by force
  thus sorted (xs @ e # ys)
    using assms
    by (auto simp add: sorted-append sorted-Cons)
  have set xs ∩ set ys = {} using assms (5,6) by force
  thus distinct (xs @ e # ys)
    using assms
    by (auto simp add: distinct-append)
```

qed

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 −
  have *sorted* $(e'\ \#\ ys)$
    distinct $(e'\ \#\ ys)$
    $\forall\ y \in set\ (e'\ \#\ ys).\ e < y$
    using *assms*(*2*,*4*,*5*,*7*)
    by (*auto simp add*: *sorted-Cons*)
  thus *sorted* $(xs\ @\ e\ \#\ e'\ \#\ ys)$
  distinct $(xs\ @\ e\ \#\ e'\ \#\ ys)$
    using *assms*(*1*,*3*,*6*) *distinct-sorted-list-lem1*[*of xs* e' # *ys e*]
    by *auto*
qed

lemma *map-of-distinct-upd*:
  $x \notin set\ (map\ fst\ xs) \Longrightarrow [x \mapsto y]\ ++\ map\text{-}of\ xs = (map\text{-}of\ xs)\ (x \mapsto y)$
  by (*induct xs*) (*auto simp add*: *fun-upd-twist*)

lemma *map-of-distinct-upd2*:
  assumes $x \notin set(map\ fst\ xs)$
  $x \notin set\ (map\ fst\ ys)$
  shows $map\text{-}of\ (xs\ @\ (x,y)\ \#\ ys) = (map\text{-}of\ (xs\ @\ ys))(x \mapsto y)$
  apply(*insert assms*)
  apply(*induct xs*)
  apply (*auto intro*: *ext*)
  done

lemma *map-of-distinct-upd3*:
  assumes $x \notin set(map\ fst\ xs)$
  $x \notin set\ (map\ fst\ ys)$
  shows $map\text{-}of\ (xs\ @\ (x,y)\ \#\ ys) = (map\text{-}of\ (xs\ @\ (x,y')\ \#\ ys))(x \mapsto y)$
  apply(*insert assms*)
  apply(*induct xs*)
  apply (*auto intro*: *ext*)
  done

lemma *map-of-distinct-upd4*:

```
  assumes x ∉ set(map fst xs)
  x ∉ set (map fst ys)
  shows map-of (xs @ ys) = (map-of (xs @ (x,y) # ys))(x := None)
  apply(insert assms)
  apply(induct xs)
  apply (auto simp add: map-of-eq-None-iff intro: ext)
  done

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 –
  have map-of (xs @ (x,y) # ys) = (map-of (xs @ ys)) (x ↦ y)
    using assms map-of-distinct-upd2 by simp
  thus ?thesis
    by simp
qed

lemma ran-distinct:
  assumes dist: distinct (map fst al)
  shows ran (map-of al) = snd ‘ set al
using assms proof (induct al)
  case Nil then show ?case by simp
next
  case (Cons kv al)
  then have ran (map-of al) = snd ‘ set al by simp
  moreover from Cons.prems have map-of al (fst kv) = None
    by (simp add: map-of-eq-None-iff)
  ultimately show ?case by (simp only: map-of.simps ran-map-upd) simp
qed
```

Finite

```
lemma aluprio-finite-correct: uprio-finite (aluprio-α α) (aluprio-invar α invar)
  by(unfold-locales) (simp add: aluprio-defs finite-dom-map-of)
```

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 –
  interpret al-empty α invar empt by fact
  show ?thesis
    apply (unfold-locales)
    apply (auto simp add: empty-correct aluprio-defs)
    done
qed
```

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 −
  interpret *al-isEmpty* α *invar isEmpty* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*auto simp add*: *aluprio-defs isEmpty-correct*)
    apply (*metis Nil-is-map-conv hd-in-set length-map length-remove1*
      length-sort map-eq-imp-length-eq map-fst-zip map-map map-of.simps(*1*)
      map-of-eq-None-iff remove1.simps(*1*) *set-map*)
    done
qed

Insert

lemma *annot-inf*:
  assumes *A*: *invar s* $\forall x \in set\ (\alpha\ s).\ snd\ x \neq Infty$ *al-annot* α *invar annot*
  shows $annot\ s = Infty \longleftrightarrow \alpha\ s = []$
proof −
  from *A* **have** *invs*: *invar s* **by** (*simp add*: *aluprio-defs*)
  interpret *al-annot* α *invar annot* **by** *fact*
  show $annot\ s = Infty \longleftrightarrow \alpha\ s = []$
  proof (*cases* $\alpha\ s = []$)
    case *True*
    hence $map\ snd\ (\alpha\ s) = []$ **by** *simp*
    hence $listsum\ (map\ snd\ (\alpha\ s)) = Infty$
      by (*auto simp add*: *zero-def*)
    with *invs* **have** $annot\ s = Infty$ **by** (*auto simp add*: *annot-correct*)
    with *True* **show** *?thesis* **by** *simp*
  next
    case *False*
    hence $\exists x\ xs.\ (\alpha\ s) = x\ \#\ xs$ **by** (*cases* α *s*) *auto*
    from *this* **obtain** *x xs* **where** [*simp*]: $(\alpha\ s) = x\ \#\ xs$ **by** *blast*
    from *this assms*(*2*) **have** $snd\ x \neq Infty$ **by** (*auto simp add*: *aluprio-defs*)
    hence $listsum\ (map\ snd\ (\alpha\ s)) \neq Infty$ **by** (*auto simp add*: *infadd*)
    thus *?thesis* **using** *annot-correct invs False* **by** *simp*
  qed
qed

lemma *e-less-eq-annot*:

  assumes *al-annot* α *invar annot*
   invar s $\forall x \in set\ (\alpha\ s).\ snd\ x \neq Infty$ $\neg$ *e-less-eq e* (*annot s*)
  shows $\forall x \in set\ (map\ (fst \circ (p\text{-}unwrap \circ snd))\ (\alpha\ s)).\ x < e$
proof −
  interpret *al-annot* α *invar annot* **by** *fact*

```
  from assms(2) have annot s = listsum (map snd (α s))
    by (auto simp add: annot-correct)
  with assms(4) have
    ∀ x ∈ set (map snd (α s)). ¬ e-less-eq e x
    by (metis e-less-eq-listsum)
  with assms(3)
  show ?thesis
    by (auto simp add: e-less-eq-p-unwrap)
qed

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 −
  interpret al-splits α invar splits by fact
  interpret al-annot α invar annot by fact
  interpret al-isEmpty α invar isEmpty by fact
  interpret al-app α invar app by fact
  interpret al-consr α invar consr by fact
  show ?thesis
  proof (unfold-locales,unfold aluprio-defs)
    case goal1 note g1asms = this
    thus ?case proof (cases e-less-eq e (annot s) ∧ ¬ isEmpty s)
      case False with g1asms show ?thesis
        apply (auto simp add: consr-correct )
      proof −
        case goal1
        with assms(2) have
          ∀ x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α s)). x < e
          by (simp add: e-less-eq-annot)
        with goal1(3) show ?case
          by(auto simp add: sorted-append)
      next
        case goal2
        hence annot s = listsum (map snd (α s))
          by (simp add: annot-correct)
        with goal2
        show ?case
          by (auto simp add: e-less-eq-listsum2)
      next
        case goal3
        hence α s = [] by (auto simp add: isEmpty-correct)
```

```
      thus ?case by simp
    next
      case goal4
      hence α s = [] by (auto simp add: isEmpty-correct)
      with goal4(6) show ?case by simp
    qed
  next
    case True note T1 = this
    obtain l uu lp r where
      l-lp-r: (splits (e-less-eq e) Infty s) = (l, ((), lp), r)
      by (cases splits (e-less-eq e) Infty s, auto)
    note v2 = splits-correct[of s e-less-eq e Infty l   () lp r]
    have
      v3: invar s
      ¬ e-less-eq e Infty
      e-less-eq e (Infty + listsum (map snd (α s)))
      using T1 g1asms annot-correct
      by (auto simp add: plus-def)
    have
      v4: α s = α l @ ((), lp) # α r
      ¬ e-less-eq e (Infty + listsum (map snd (α l)))
      e-less-eq e (Infty + listsum (map snd (α l)) + lp)
      invar l
      invar r
      using v2[OF v3(1) - v3(2) v3(3) l-lp-r] e-less-eq-mon(1) by auto
    hence v5: e-less-eq e lp
      by (metis e-less-eq-lem1)
    hence v6: e ≤ (fst (p-unwrap lp))
      by (cases lp) auto
    have (Infty + listsum (map snd (α l))) = (annot l)
      by (metis add-0-left annot-correct v4(4) zero-def)
    hence v7:¬ e-less-eq e (annot l)
      using v4(2) by simp
    have ∀x∈set (α l). snd x ≠ Infty
      using g1asms v4(1) by simp
    hence v7: ∀x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α l)). x < e
      using v4(4) v7 assms(2)
      by(simp add: e-less-eq-annot)
    have v8:map fst (map p-unwrap (map snd (α s))) =
      map fst (map p-unwrap (map snd (α l))) @ fst(p-unwrap lp) #
      map fst (map p-unwrap (map snd (α r)))
      using v4(1)
      by simp
    note distinct-sortet-list-app[of map fst (map p-unwrap (map snd (α s)))
      map fst (map p-unwrap (map snd (α l)))   fst(p-unwrap lp)
      map fst (map p-unwrap (map snd (α r)))]
    hence v9:
      ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). fst(p-unwrap lp) < x
      using v4(1) g1asms v8
```

```
    by auto
  have v10:
    sorted (map fst (map p-unwrap (map snd (α l))))
    distinct (map fst (map p-unwrap (map snd (α l))))
    sorted (map fst (map p-unwrap (map snd (α r))))
    distinct (map fst (map p-unwrap (map snd (α l))))
    using g1asms v8
    by (auto simp add: sorted-append sorted-Cons)

  from l-lp-r T1 g1asms show ?thesis
  proof (fold aluprio-insert-def, cases e < fst (p-unwrap lp))
    case True
    hence v11:
      aluprio-insert splits annot isEmpty app consr s e a
        = app (consr (consr l () (LP e a)) () lp) r
      using l-lp-r T1
      by (auto simp add: aluprio-defs)
    have v12: invar (app (consr (consr l () (LP e a)) () lp) r)
      using v4(4,5)
      by (auto simp add: app-correct consr-correct)
    have v13:
      α (app (consr (consr l () (LP e a)) () lp) r)
        = α l @ ((),(LP e a)) # ((), lp) # α r
      using v4(4,5) by (auto simp add: app-correct consr-correct)
    hence v14:
      (∀x∈set (α (app (consr (consr l () (LP e a)) () lp) r)).
         snd x ≠ Infty)
      using g1asms v4(1)
      by auto
    have v15: e = fst(p-unwrap (LP e a)) by simp
    hence v16:
      sorted (map fst (map p-unwrap
         (map snd (α l @ ((),(LP e a)) # ((), lp) # α r))))
      distinct (map fst (map p-unwrap
         (map snd (α l @ ((),(LP e a)) # ((), lp) # α r))))
      using v10(1,3) v7 True v9 v4(1) g1asms distinct-sorted-list-lem2
      by (auto simp add: sorted-append sorted-Cons)
    thus invar (aluprio-insert splits annot isEmpty app consr s e a) ∧
      (∀x∈set (α (aluprio-insert splits annot isEmpty app consr s e a)).
         snd x ≠ Infty) ∧
      sorted (map fst (map p-unwrap (map snd (α
         (aluprio-insert splits annot isEmpty app consr s e a))))) ∧
      distinct (map fst (map p-unwrap (map snd (α
         (aluprio-insert splits annot isEmpty app consr s e a)))))
      using v11 v12 v13 v14
      by simp
  next
    case False
    hence v11:
```

```
        aluprio-insert splits annot isEmpty app consr s e a
          = app (consr l () (LP e a)) r
        using l-lp-r T1
        by (auto simp add: aluprio-defs)
      have v12: invar (app (consr l () (LP e a)) r) using v4(4,5)
        by (auto simp add: app-correct consr-correct)
      have v13: α (app (consr l () (LP e a)) r) = α l @ ((),(LP e a)) # α r
        using v4(4,5) by (auto simp add: app-correct consr-correct)
      hence v14: (∀ x∈set (α (app (consr l () (LP e a)) r)). snd x ≠ Infty)
        using g1asms v4(1)
        by auto
      have v15: e = fst(p-unwrap (LP e a)) by simp
      have v16: e = fst(p-unwrap lp)
        using False v5 by (cases lp) auto
      hence v17:
        sorted (map fst (map p-unwrap
          (map snd (α l @ ((),(LP e a)) # α r))))
        distinct (map fst (map p-unwrap
          (map snd (α l @ ((),(LP e a)) # α r))))
        using v16 v15 v10(1,3) v7 True v9 v4(1)
          g1asms distinct-sorted-list-lem1
        by (auto simp add: sorted-append sorted-Cons)
      thus invar (aluprio-insert splits annot isEmpty app consr s e a) ∧
        (∀ x∈set (α (aluprio-insert splits annot isEmpty app consr s e a)).
          snd x ≠ Infty) ∧
        sorted (map fst (map p-unwrap (map snd (α
          (aluprio-insert splits annot isEmpty app consr s e a))))) ∧
        distinct (map fst (map p-unwrap (map snd (α
          (aluprio-insert splits annot isEmpty app consr s e a)))))
        using v11 v12 v13 v14
        by simp
    qed
  qed
next
  case goal2 note g1asms = this
  thus ?case proof (cases e-less-eq e (annot s) ∧ ¬ isEmpty s)
    case False with g1asms show ?thesis
      apply (auto simp add: consr-correct)
    proof −
      case goal1
      with assms(2) have
        ∀ x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α s)). x < e
        by (simp add: e-less-eq-annot)
      hence e ∉ set (map fst ((map (p-unwrap ∘ snd)) (α s)))
        by auto
      thus ?case
        by (auto simp add: map-of-distinct-upd)
    next
      case goal2
```

```
    hence α s = [] by (auto simp add: isEmpty-correct)
    thus ?case
      by simp
  qed
next
  case True note T1 = this
  obtain l uu lp r where
    l-lp-r: (splits (e-less-eq e) Infty s) = (l, ((), lp), r)
    by (cases splits (e-less-eq e) Infty s, auto)
  note v2 = splits-correct[of s e-less-eq e Infty l   () lp r]
  have
    v3: invar s
    ¬ e-less-eq e Infty
    e-less-eq e (Infty + listsum (map snd (α s)))
    using T1 g1asms annot-correct
    by (auto simp add: plus-def)
  have
    v4: α s = α l @ ((), lp) # α r
    ¬ e-less-eq e (Infty + listsum (map snd (α l)))
    e-less-eq e (Infty + listsum (map snd (α l)) + lp)
    invar l
    invar r
    using v2[OF v3(1) - v3(2) v3(3) l-lp-r] e-less-eq-mon(1) by auto
  hence v5: e-less-eq e lp
    by (metis e-less-eq-lem1)
  hence v6: e ≤ (fst (p-unwrap lp))
    by (cases lp) auto
  have (Infty + listsum (map snd (α l))) = (annot l)
    by (metis add-0-left annot-correct v4(4) zero-def)
  hence v7:¬ e-less-eq e (annot l)
    using v4(2) by simp
  have ∀x∈set (α l). snd x ≠ Infty
    using g1asms v4(1) by simp
  hence v7: ∀x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α l)). x < e
    using v4(4) v7 assms(2)
    by(simp add: e-less-eq-annot)
  have v8:map fst (map p-unwrap (map snd (α s))) =
    map fst (map p-unwrap (map snd (α l))) @ fst(p-unwrap lp) #
    map fst (map p-unwrap (map snd (α r)))
    using v4(1)
    by simp
  note distinct-sortet-list-app[of map fst (map p-unwrap (map snd (α s)))
    map fst (map p-unwrap (map snd (α l)))   fst(p-unwrap lp)
    map fst (map p-unwrap (map snd (α r)))]
  hence v9:
   ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). fst(p-unwrap lp) < x
    using v4(1) g1asms v8
    by auto
  hence v10: ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). e < x
```

using *v6* **by** *auto*
have *v11*:
$e \notin set\ (map\ fst\ (map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ l))))$
$e \notin set\ (map\ fst\ (map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ r))))$
using *v7 v10 v8 g1asms*
by *auto*
from *l-lp-r T1 g1asms* **show** *?thesis*
proof (*fold aluprio-insert-def*, *cases* $e < fst\ (p\text{-}unwrap\ lp)$)
case *True*
hence *v12*:
aluprio-insert splits annot isEmpty app consr s e a
$= app\ (consr\ (consr\ l\ ()\ (LP\ e\ a))\ ()\ lp)\ r$
using *l-lp-r T1*
by (*auto simp add*: *aluprio-defs*)
have *v13*:
$\alpha\ (app\ (consr\ (consr\ l\ ()\ (LP\ e\ a))\ ()\ lp)\ r)$
$= \alpha\ l\ @\ ((),(LP\ e\ a))\ \#\ ((),\ lp)\ \#\ \alpha\ r$
using *v4*(*4*,*5*) **by** (*auto simp add*: *app-correct consr-correct*)
have *v14*: $e = fst(p\text{-}unwrap\ (LP\ e\ a))$ **by** *simp*
have *v15*: $e \notin set\ (map\ fst\ (map\ p\text{-}unwrap\ (map\ snd(((),lp)\#\alpha\ r))))$
using *v11*(*2*) *True* **by** *auto*
note *map-of-distinct-upd2*[*OF v11*(*1*) *v15*]
thus
$map\text{-}of\ (map\ p\text{-}unwrap\ (map\ snd\ (\alpha$
$(aluprio\text{-}insert\ splits\ annot\ isEmpty\ app\ consr\ s\ e\ a))))$
$= map\text{-}of\ (map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ s)))(e \mapsto a)$
using *v12 v13 v4*(*1*)
by *simp*
next
case *False*
hence *v12*:
aluprio-insert splits annot isEmpty app consr s e a
$= app\ (consr\ l\ ()\ (LP\ e\ a))\ r$
using *l-lp-r T1*
by (*auto simp add*: *aluprio-defs*)
have *v13*:
$\alpha\ (app\ (consr\ l\ ()\ (LP\ e\ a))\ r) = \alpha\ l\ @\ ((),(LP\ e\ a))\ \#\ \alpha\ r$
using *v4*(*4*,*5*) **by** (*auto simp add*: *app-correct consr-correct*)
have *v14*: $e = fst(p\text{-}unwrap\ lp)$
using *False v5* **by** (*cases lp*) *auto*
note *v15* = *map-of-distinct-upd3*[*OF v11*(*1*) *v11*(*2*)]
have *v16*:$(map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ s))) =$
$(map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ l)))\ @\ (e,snd(p\text{-}unwrap\ lp))\ \#$
$(map\ p\text{-}unwrap\ (map\ snd\ (\alpha\ r)))$
using *v4*(*1*) *v14*
by *simp*
note *v15*[*of a snd*(*p-unwrap lp*)]
thus
$map\text{-}of\ (map\ p\text{-}unwrap\ (map\ snd\ (\alpha$

(*aluprio-insert splits annot isEmpty app consr s e a*))))
= *map-of* (*map p-unwrap* (*map snd* (α *s*)))(*e* $\mapsto$ *a*)
using *v12 v13 v16*
by *simp*
qed
qed
qed
qed

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 −
interpret *al-splits* α *invar splits* **by** *fact*
interpret *al-annot* α *invar annot* **by** *fact*
interpret *al-isEmpty* α *invar isEmpty* **by** *fact*
show *?thesis*
proof (*unfold-locales*)
fix *s e*
assume *inv1*: *aluprio-invar* α *invar s*
hence *sinv*: *invar s*
($\forall$ *x*$\in$*set* (α *s*). *snd x*$\neq$*Infty*)
sorted (*map fst* (*map p-unwrap* (*map snd* (α *s*))))
distinct (*map fst* (*map p-unwrap* (*map snd* (α *s*))))
by (*auto simp add*: *aluprio-defs*)
show *aluprio-prio splits annot isEmpty s e* = *aluprio-*α α *s e*
proof(*cases e-less-eq e* (*annot s*) $\wedge$ $\neg$ *isEmpty s*)
case *False* **note** *F1* = *this*
thus *?thesis*
proof(*cases isEmpty s*)
case *True*
hence α *s* = []
using *sinv isEmpty-correct* **by** *simp*
hence *aluprio-*α α *s* = *empty* **by** (*simp add*:*aluprio-defs*)
hence *aluprio-*α α *s e* = *None* **by** *simp*
thus *aluprio-prio splits annot isEmpty s e* = *aluprio-*α α *s e*
using *F1*
by (*auto simp add*: *aluprio-defs*)
next
case *False*
hence *v3*:$\neg$ *e-less-eq e* (*annot s*) **using** *F1* **by** *simp*
note *v4*=*e-less-eq-annot*[*OF assms*(*2*)]
note *v4*[*OF sinv*(*1*) *sinv*(*2*) *v3*]

hence *v5*:*e*∉*set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (α *s*))
  by *auto*
hence *map-of* (*map* (*p-unwrap* ∘ *snd*) (α *s*)) *e* = *None*
  using *map-of-eq-None-iff*
  by (*metis map-map map-of-eq-None-iff set-map v5*)
thus *aluprio-prio splits annot isEmpty s e* = *aluprio-α α s e*
  using *F1*
  by (*auto simp add*: *aluprio-defs*)
qed
next
case *True* **note** *T1* = *this*
obtain *l uu lp r* **where**
l-lp-r: (*splits* (*e-less-eq e*) *Infty s*) = (*l*, ((), *lp*), *r*)
by (*cases splits* (*e-less-eq e*) *Infty s*, *auto*)
note *v2* = *splits-correct*[*of s e-less-eq e Infty l* () *lp r*]
have
v3: *invar s*
¬ *e-less-eq e Infty*
e-less-eq e (*Infty* + *listsum* (*map snd* (α *s*)))
using *T1 sinv annot-correct*
by (*auto simp add*: *plus-def*)
have
v4: α *s* = α *l* @ ((), *lp*) # α *r*
¬ *e-less-eq e* (*Infty* + *listsum* (*map snd* (α *l*)))
e-less-eq e (*Infty* + *listsum* (*map snd* (α *l*)) + *lp*)
invar l
invar r
using *v2*[*OF v3*(*1*) - *v3*(*2*) *v3*(*3*) *l-lp-r*] *e-less-eq-mon*(*1*) **by** *auto*
hence *v5*: *e-less-eq e lp*
by (*metis e-less-eq-lem1*)
hence *v6*: *e* ≤ (*fst* (*p-unwrap lp*))
by (*cases lp*) *auto*
have (*Infty* + *listsum* (*map snd* (α *l*))) = (*annot l*)
by (*metis add-0-left annot-correct v4*(*4*) *zero-def*)
hence *v7*:¬ *e-less-eq e* (*annot l*)
using *v4*(*2*) **by** *simp*
have ∀ *x*∈*set* (α *l*). *snd x* ≠ *Infty*
using *sinv v4*(*1*) **by** *simp*
hence *v7*: ∀ *x* ∈ *set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (α *l*)). *x* < *e*
using *v4*(*4*) *v7 assms*(*2*)
by(*simp add*: *e-less-eq-annot*)
have *v8*:*map fst* (*map p-unwrap* (*map snd* (α *s*))) =
map fst (*map p-unwrap* (*map snd* (α *l*))) @ *fst*(*p-unwrap lp*) #
map fst (*map p-unwrap* (*map snd* (α *r*)))
using *v4*(*1*)
by *simp*
note *distinct-sortet-list-app*[*of map fst* (*map p-unwrap* (*map snd* (α *s*)))
map fst (*map p-unwrap* (*map snd* (α *l*))) *fst*(*p-unwrap lp*)
map fst (*map p-unwrap* (*map snd* (α *r*)))]

```
    hence v9:
      ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). fst(p-unwrap lp) < x
      using v4(1) sinv v8
      by auto
    hence v10: ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). e < x
      using v6 by auto
    have v11:
      e ∉ set (map fst (map p-unwrap (map snd (α l))))
      e ∉ set (map fst (map p-unwrap (map snd (α r))))
      using v7 v10 v8 sinv
      by auto
    from l-lp-r T1 sinv show ?thesis
    proof (cases e = fst (p-unwrap lp))
      case False
      have v12: e ∉ set (map fst (map p-unwrap (map snd(α s))))
        using v11 False v4(1) by auto
      hence map-of (map (p-unwrap ∘ snd) (α s)) e = None
        using map-of-eq-None-iff
        by (metis map-map map-of-eq-None-iff set-map v12)
      thus ?thesis
        using T1 False l-lp-r
        by (auto simp add: aluprio-defs)
    next
      case True
      have v12: map (p-unwrap ∘ snd) (α s) =
        map p-unwrap (map snd (α l)) @ (e,snd (p-unwrap lp)) #
        map p-unwrap (map snd (α r))
        using v4(1) True by simp
      note map-of-distinct-lookup[OF v11]
      hence
        map-of (map (p-unwrap ∘ snd) (α s)) e = Some (snd (p-unwrap lp))
        using v12 by simp
      thus ?thesis
        using T1 True l-lp-r
        by (auto simp add: aluprio-defs)
    qed
   qed
  qed
qed
```

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 –
```

interpret *al-splits α invar splits* **by** *fact*
interpret *al-annot α invar annot* **by** *fact*
interpret *al-app α invar app* **by** *fact*
show *?thesis*
proof (*unfold-locales*)
  fix $s\ e\ a\ s'$
  assume *A*: *aluprio-invar α invar s*
    aluprio-α α s ≠ empty
    aluprio-pop splits annot app s = (*e*, *a*, *s'*)
  hence *v1*: *α s* ≠ []
    by (*auto simp add*: *aluprio-defs*)
  obtain *l lp r* **where**
    l-lp-r: *splits* (*λ x. x≤annot s*) *Infty s* = (*l*,((),*lp*),*r*)
    by (*cases splits* (*λ x. x≤annot s*) *Infty s*, *auto*)
  have *invs*:
    invar s
    (∀ *x*∈*set* (*α s*). *snd x* ≠ *Infty*)
    sorted (*map fst* (*map p-unwrap* (*map snd* (*α s*))))
    distinct (*map fst* (*map p-unwrap* (*map snd* (*α s*))))
    using *A* **by** (*auto simp add*:*aluprio-defs*)
  note *a1* = *annot-inf*[*of invar s α annot*]
  note *a1*[*OF invs*(*1*) *invs*(*2*) *assms*(*2*)]
  hence *v2*: *annot s* ≠ *Infty*
    using *v1* **by** *simp*
  hence *v3*:
    ¬ *Infty* ≤ *annot s*
    by(*cases annot s*) (*auto simp add*: *plesseq-def*)
  have *v4*: *annot s* = *listsum* (*map snd* (*α s*))
    by (*auto simp add*: *annot-correct invs*(*1*))
  hence
    v5:
    (*Infty* + *listsum* (*map snd* (*α s*))) ≤ *annot s*
    by (*auto simp add*: *plus-def*)
  note *p-mon* = *p-less-eq-mon*[*of - annot s*]
  note *v6* = *splits-correct*[*OF invs*(*1*)]
  note *v7* = *v6*[*of λ x. x ≤ annot s*]
  note *v7*[*OF - v3 v5 l-lp-r*] *p-mon*
  hence *v8*:
     α s = *α l* @ ((), *lp*) # *α r*
    ¬ *Infty* + *listsum* (*map snd* (*α l*)) ≤ *annot s*
    Infty + *listsum* (*map snd* (*α l*)) + *lp* ≤ *annot s*
    invar l
    invar r
    by *auto*
  hence *v9*: *lp* ≠ *Infty*
    using *invs*(*2*) **by** *auto*
  hence *v10*:
    s' = *app l r*
    (*e*,*a*) = *p-unwrap lp*

```
    using l-lp-r A(3)
    apply (auto simp add: aluprio-defs)
    apply (cases lp)
    apply auto
    apply (cases lp)
    apply auto
    done
  have lp ≤ annot s
    using v8(2,3) p-less-eq-lem1
    by auto
  hence v11: a ≤ snd (p-unwrap (annot s))
    using v10(2) v2 v9
    apply (cases annot s)
    apply auto
    apply (cases lp)
    apply (auto simp add: plesseq-def)
    done
  note listsum-less-elems[OF invs(2)]
  hence v12: ∀ y∈set (map snd (map p-unwrap (map snd (α s)))). a ≤ y
    using v4 v11 by auto
  have ran (aluprio-α α s) = set (map snd (map p-unwrap (map snd (α s))))
    using ran-distinct[OF invs(4)]
    apply (unfold aluprio-defs)
    apply (simp only: set-map)
    done
  hence ziel1: ∀ y∈ran (aluprio-α α s). a ≤ y
    using v12 by simp
  have v13:
    map p-unwrap (map snd (α s))
      = map p-unwrap (map  snd (α l)) @ (e,a) # map p-unwrap (map snd (α
r))
    using v8(1) v10 by auto
   hence v14:
    map fst (map p-unwrap (map snd (α s)))
      = map fst (map p-unwrap (map snd (α l))) @ e
          # map fst (map p-unwrap (map snd (α r)))
     by auto
  hence v15:
    e ∉ set (map fst (map p-unwrap (map snd (α l))))
    e ∉ set (map fst (map p-unwrap (map snd (α r))))
    using invs(4) by auto
  note map-of-distinct-lookup[OF v15]
  note this[of a]
  hence ziel2: aluprio-α α s e = Some a
    using  v13
    by (unfold aluprio-defs, auto)
  have v16:
    α s' = α l @ α r
    invar s'
```

```
      using v8(4,5) app-correct v10 by auto
    note map-of-distinct-upd4[OF v15]
    note this[of a]
    hence
      ziel3: aluprio-α α s' = (aluprio-α α s)(e := None)
      unfolding aluprio-defs
      using v16(1) v13 by auto
    have ziel4: aluprio-invar α invar s'
      using v16 v8(1) invs(2,3,4)
      unfolding aluprio-defs
      by (auto simp add: sorted-Cons sorted-append)

    show aluprio-invar α invar s' ∧
          aluprio-α α s' = (aluprio-α α s)(e := None) ∧
          aluprio-α α s e = Some a ∧ (∀ y∈ran (aluprio-α α s). a ≤ y)
      using ziel1 ziel2 ziel3 ziel4 by simp
  qed
qed

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* × *'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* ⇀ *'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
by (*unfold-locales*) (*auto simp add*: *lm-defs fun-eq-iff*)

interpretation *lm*: *map-empty lm-α lm-invar lm-empty* **by**(*fact lm-empty-impl*)

lemma *lm-lookup-impl*:
map-lookup lm-α lm-invar lm-lookup
by (*unfold-locales*)(*simp add*: *lm-defs*)

interpretation *lm*: *map-lookup lm-α lm-invar lm-lookup* **using** *lm-lookup-impl* **.**

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

by (*unfold-locales*)(*simp-all add*: *lm-defs*)

interpretation *lm*: *map-update lm-α lm-invar lm-update* **using** *lm-update-impl* **.**

lemma *lm-update-dj-impl*:
map-update-dj lm-α lm-invar lm-update-dj
by (*unfold-locales*) (*simp-all add*: *lm-defs*)

interpretation *lm*: *map-update-dj lm-α lm-invar lm-update-dj* **using** *lm-update-dj-impl*
.

lemma *lm-delete-impl*:
map-delete lm-α lm-invar lm-delete
by (*unfold-locales*)(*simp-all add*: *lm-defs*)

interpretation *lm*: *map-delete lm-α lm-invar lm-delete* **using** *lm-delete-impl* **.**

lemma *lm-isEmpty-impl*:
map-isEmpty lm-α lm-invar lm-isEmpty
apply (*unfold-locales*)
apply (*unfold lm-defs*)
apply(*rule iffI*)
apply(*simp add*: *impl-of-inject lookup-empty′*)
apply(*case-tac m*)
apply(*simp add*: *Assoc-List.empty-def Assoc-List.lookup-def Assoc-List-inverse Assoc-List-inject*)
apply(*case-tac y*)
apply *simp-all*
done

interpretation *lm*: *map-isEmpty lm-α lm-invar lm-isEmpty* **using** *lm-isEmpty-impl*
.

lemma *lm-is-finite-map*: *finite-map lm-α lm-invar*
by *unfold-locales*(*simp add*: *lm-defs*)

interpretation *lm*: *finite-map lm-α lm-invar* **using** *lm-is-finite-map* **.**

lemma *lm-iteratei-impl*:
map-iteratei lm-α lm-invar lm-iteratei
unfolding *map-iteratei-def finite-map-def map-iteratei-axioms-def lm-defs*
by (*simp add*: *iteratei-correct*)

interpretation *lm*: *map-iteratei lm-α lm-invar lm-iteratei* **using** *lm-iteratei-impl*
.

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

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

by (*auto simp add*: *lm-α-def*)

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* **using** *lm-add-impl* **.**

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* **using** *lm-add-dj-impl* **.**

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* **using** *lm-sel-impl* **.**

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'* **using** *lm-sel'-impl* **.**

lemma *lm-to-list-impl*: *map-to-list lm-α lm-invar lm-to-list*
by(*unfold-locales*)(*auto simp add*: *lm-defs Assoc-List.lookup-def*)

interpretation *lm*: *map-to-list lm-α lm-invar lm-to-list* **using** *lm-to-list-impl* **.**

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*
using *list-to-lm-impl* **.**

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* == (λ-::*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
apply (*unfold-locales*)
apply (*auto*
simp add: *lmi-defs AList.update-conv′ AList.distinct-update*)
done

interpretation *lmi*: *map-empty lmi-α lmi-invar lmi-empty* **using** *lmi-empty-impl*
.

lemma *lmi-lookup-impl*:
map-lookup lmi-α lmi-invar lmi-lookup
apply (*unfold-locales*)
apply (*auto*
simp add: *lmi-defs AList.update-conv′ AList.distinct-update*)
done

interpretation *lmi*: *map-lookup lmi-α lmi-invar lmi-lookup* **using** *lmi-lookup-impl*
.

lemma *lmi-update-impl*:
map-update lmi-α lmi-invar lmi-update
apply (*unfold-locales*)
apply (*auto*
simp add: *lmi-defs AList.update-conv′ AList.distinct-update*)
done

```
interpretation lmi: map-update lmi-α lmi-invar lmi-update using lmi-update-impl
.

lemma lmi-update-dj-impl:
  map-update-dj lmi-α lmi-invar lmi-update-dj
  apply (unfold-locales)
  apply (auto simp add: lmi-defs)
  done

interpretation lmi: map-update-dj lmi-α lmi-invar lmi-update-dj using lmi-update-dj-impl
.

lemma lmi-delete-impl:
  map-delete lmi-α lmi-invar lmi-delete
  apply (unfold-locales)
  apply (auto simp add: lmi-defs AList.update-conv' AList.distinct-update
                 Assoc-List.map-of-delete-aux')
  done

interpretation lmi: map-delete lmi-α lmi-invar lmi-delete using lmi-delete-impl
.

lemma lmi-isEmpty-impl:
  map-isEmpty lmi-α lmi-invar lmi-isEmpty
  apply (unfold-locales)
  apply (unfold lmi-defs)
  apply (case-tac m)
  apply auto
  done

interpretation lmi: map-isEmpty lmi-α lmi-invar lmi-isEmpty using lmi-isEmpty-impl
.

lemma lmi-is-finite-map: finite-map lmi-α lmi-invar
by unfold-locales(auto simp add: lmi-defs)

interpretation lmi: finite-map lmi-α lmi-invar using lmi-is-finite-map .

lemma lmi-iteratei-impl:
  map-iteratei lmi-α lmi-invar lmi-iteratei
proof
  fix m :: ('a × 'b) list
  assume invar: lmi-invar m

  from set-iterator-foldli-correct [of m] invar
  have set-iterator (foldli m) (set m)
    by (simp add: lmi-invar-def distinct-map)

  with invar have map-iterator (foldli m) (map-of m)
```

```
    by (simp add: map-to-set-map-of lmi-invar-def)
  thus map-iterator (lmi-iteratei m) (lmi-α m)
    unfolding lmi-α-def lmi-iteratei-def .
qed

interpretation lmi: map-iteratei lmi-α lmi-invar lmi-iteratei using lmi-iteratei-impl
.

declare lmi.finite[simp del, rule del]

lemma lmi-finite[simp, intro!]: finite (dom (lmi-α m))
  by (auto simp add: lmi-α-def)

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 using lmi-add-impl .


lemma lmi-add-dj-impl:
  shows map-add-dj lmi-α lmi-invar lmi-add-dj
  apply (unfold-locales)
  apply (auto simp add: lmi-defs)
  apply (blast intro: map-add-comm)
  apply (simp add: rev-map[symmetric])
  apply fastforce
  done

interpretation lmi: map-add-dj lmi-α lmi-invar lmi-add-dj using lmi-add-dj-impl
.

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 using lmi-sel-impl .

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′ using lmi-sel′-impl .

lemma lmi-to-list-impl: map-to-list lmi-α lmi-invar lmi-to-list
  by (unfold-locales)
     (auto simp add: lmi-defs)
interpretation lmi: map-to-list lmi-α lmi-invar lmi-to-list using lmi-to-list-impl
.

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
  using list-to-lmi-impl .
```

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*)
  by (*auto simp add*: *lmi-defs*)

lemma *lmi-to-list-to-lm*[*simp*]:
  lmi-invar m $\Longrightarrow$ *lmi-*α (*list-to-lmi-dj* (*lmi-to-list m*)) = *lmi-*α *m*
  by (*simp add*: *lmi.to-list-correct list-to-lmi-dj-correct*)

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*
by (*unfold-locales*, *unfold rm-defs*)
(*simp-all*)

lemma *rm-lookup-impl*: *map-lookup rm-α rm-invar rm-lookup*
by (*unfold-locales*, *unfold rm-defs*)
(*simp-all*)

lemma *rm-update-impl*: *map-update rm-α rm-invar rm-update*
by (*unfold-locales*, *unfold rm-defs*)
(*simp-all*)

lemma *rm-update-dj-impl*: *map-update-dj rm-α rm-invar rm-update-dj*
by (*unfold-locales*, *unfold rm-defs*)
(*simp-all*)

lemma *rm-delete-impl*: *map-delete rm-α rm-invar rm-delete*
by (*unfold-locales*, *unfold rm-defs*)
(*simp-all*)

lemma *rm-α-alist*: *rm-invar m* $\Longrightarrow$ *rm-α m* = *Map.map-of* (*RBT.entries m*)
by (*simp add*: *rm-α-def*)

lemma *rm-α-finite*[*simp*, *intro*!]: *finite* (*dom* (*rm-α m*))
by (*simp add*: *rm-α-def*)

lemma *rm-is-finite-map*: *finite-map rm-α rm-invar* **by** (*unfold-locales*) *auto*

lemma *map-to-set-lookup-entries*:
rbt-sorted t $\Longrightarrow$ *map-to-set* (*rbt-lookup t*) = *set* (*RBT-Impl.entries t*)
using *RBT-Impl.map-of-entries*[*symmetric*, *of t*]
by (*simp add*: *RBT-Impl.distinct-entries map-to-set-map-of*)

lemma *rm-iterateoi-correct*:
fixes *t*::(*'k*::*linorder*, *'v*) *RBT-Impl.rbt*
assumes *is-sort*: *rbt-sorted t*
defines *it* $\equiv$ *RBT-add.rm-iterateoi*::((*'k*, *'v*) *RBT-Impl.rbt* $\Rightarrow$ (*'k* $\times$ *'v*, *'σ*) *set-iterator*)
shows *map-iterator-linord* (*it t*) (*rbt-lookup t*)
using *is-sort*
proof (*induct t*)
case *Empty*

show *?case* **unfolding** *it-def*
by (*simp add*: *rm-iterateoi-alt-def map-iterator-linord-emp-correct rbt-lookup-Empty*)
next
case (*Branch c l k v r*)
note *is-sort-t* = *Branch*(*3*)

from *Branch*(*1*) *is-sort-t* **have** *l-it*: *map-iterator-linord* (*it l*) (*rbt-lookup l*) **by** *simp*
from *Branch*(*2*) *is-sort-t* **have** *r-it*: *map-iterator-linord* (*it r*) (*rbt-lookup r*) **by** *simp*
note *kv-it* = *map-iterator-linord-sng-correct*[*of k v*]

have *kv-r-it* : *set-iterator-map-linord*
(*set-iterator-union* (*set-iterator-sng* (*k*, *v*)) (*it r*))
(*map-to-set* [*k* ↦ *v*] ∪ *map-to-set* (*rbt-lookup r*))
proof (*rule map-iterator-linord-union-correct* [*OF kv-it r-it*])
fix *kv kv′*
assume *pre*: *kv* ∈ *map-to-set* [*k* ↦ *v*]   *kv′* ∈ *map-to-set* (*rbt-lookup r*)
obtain *k′ v′* **where** *kv′-eq*[*simp*]: *kv′* = (*k′*, *v′*) **by** (*rule PairE*)

from *pre is-sort-t* **show** *fst kv* < *fst kv′*
apply (*simp add*: *map-to-set-lookup-entries split*: *prod.splits*)
apply (*metis entry-in-tree-keys rbt-greater-prop*)
done
qed

have *l-kv-r-it* : *set-iterator-map-linord* (*it* (*Branch c l k v r*))
(*map-to-set* (*rbt-lookup l*) ∪ (*map-to-set* [*k* ↦ *v*] ∪ *map-to-set* (*rbt-lookup r*)))
unfolding *it-def rm-iterateoi-alt-def*
unfolding *it-def*[*symmetric*]
proof (*rule map-iterator-linord-union-correct* [*OF l-it kv-r-it*])
fix *kv1 kv2*
assume *pre*: *kv1* ∈ *map-to-set* (*rbt-lookup l*)
kv2 ∈ *map-to-set* [*k* ↦ *v*] ∪ *map-to-set* (*rbt-lookup r*)

obtain *k1 v1* **where** *kv1-eq*[*simp*]: *kv1* = (*k1*, *v1*) **by** (*rule PairE*)
obtain *k2 v2* **where** *kv2-eq*[*simp*]: *kv2* = (*k2*, *v2*) **by** (*rule PairE*)

from *pre is-sort-t* **show** *fst kv1* < *fst kv2*
apply (*simp add*: *map-to-set-lookup-entries split*: *prod.splits*)
apply (*metis entry-in-tree-keys rbt-greater-prop rbt-less-prop trt-trans′*)
done
qed

from *is-sort-t*
have *map-eq*: *map-to-set* (*rbt-lookup l*) ∪ (*map-to-set* [*k* ↦ *v*] ∪ *map-to-set* (*rbt-lookup r*)) =
map-to-set (*rbt-lookup* (*Branch c l k v r*))
by (*simp add*: *set-eq-iff map-to-set-lookup-entries*)

from *l-kv-r-it*[*unfolded map-eq*]
show *?case* **.**
qed

lemma *rm-iterateoi-impl*: *map-iterateoi rm-α rm-invar rm-iterateoi*
proof
fix *m*:: (*'a*, *'b*) *rm*
show *finite* (*dom* (*rm-α m*)) **by** *simp*
next
fix *m*:: (*'a*, *'b*) *rm*
show *map-iterator-linord* (*RBTMapImpl.rm-iterateoi m*) (*rm-α m*)
unfolding *rm-α-def RBTMapImpl.rm-iterateoi-def*
by *transfer* (*simp add*: *rm-iterateoi-correct*)
qed

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*)
using *is-sort*
proof (*induct t*)
case *Empty*
show *?case* **unfolding** *it-def*
by (*simp add*: *rm-reverse-iterateoi-alt-def map-iterator-rev-linord-emp-correct rbt-lookup-Empty*)
next
case (*Branch c l k v r*)
note *is-sort-t* = *Branch*(*3*)

from *Branch*(*1*) *is-sort-t* **have** *l-it*: *map-iterator-rev-linord* (*it l*) (*rbt-lookup l*) **by** *simp*
from *Branch*(*2*) *is-sort-t* **have** *r-it*: *map-iterator-rev-linord* (*it r*) (*rbt-lookup r*) **by** *simp*
note *kv-it* = *map-iterator-rev-linord-sng-correct*[*of k v*]

have *kv-l-it* : *set-iterator-map-rev-linord*
(*set-iterator-union* (*set-iterator-sng* (*k*, *v*)) (*it l*))
(*map-to-set* [*k* ↦ *v*] ∪ *map-to-set* (*rbt-lookup l*))
proof (*rule map-iterator-rev-linord-union-correct* [*OF kv-it l-it*])
fix *kv kv'*
assume *pre*: *kv* ∈ *map-to-set* [*k* ↦ *v*] *kv'* ∈ *map-to-set* (*rbt-lookup l*)
obtain *k' v'* **where** *kv'-eq*[*simp*]: *kv'* = (*k'*, *v'*) **by** (*rule PairE*)

from *pre is-sort-t* **show** *fst kv* > *fst kv'*
apply (*simp add*: *map-to-set-lookup-entries split*: *prod.splits*)
apply (*metis entry-in-tree-keys rbt-less-prop*)

done
qed

have *r-kv-l-it* : *set-iterator-map-rev-linord* (*it* (*Branch c l k v r*))
(*map-to-set* (*rbt-lookup r*) $\cup$ (*map-to-set* [$k \mapsto v$] $\cup$ *map-to-set* (*rbt-lookup l*)))
unfolding *it-def rm-reverse-iterateoi-alt-def*
unfolding *it-def*[*symmetric*]
proof (*rule map-iterator-rev-linord-union-correct* [*OF r-it kv-l-it*])
fix *kv1 kv2*
assume *pre*: *kv1* $\in$ *map-to-set* (*rbt-lookup r*)
kv2 $\in$ *map-to-set* [$k \mapsto v$] $\cup$ *map-to-set* (*rbt-lookup l*)

obtain *k1 v1* **where** *kv1-eq*[*simp*]: *kv1* = (*k1*, *v1*) **by** (*rule PairE*)
obtain *k2 v2* **where** *kv2-eq*[*simp*]: *kv2* = (*k2*, *v2*) **by** (*rule PairE*)

from *pre is-sort-t* **show** *fst kv1* > *fst kv2*
apply (*simp add*: *map-to-set-lookup-entries split*: *prod.splits*)
apply (*metis entry-in-tree-keys rbt-greater-prop rbt-less-prop trt-trans'*)
done
qed

from *is-sort-t*
have *map-eq*: *map-to-set* (*rbt-lookup r*) $\cup$ (*map-to-set* [$k \mapsto v$] $\cup$ *map-to-set* (*rbt-lookup l*)) =
map-to-set (*rbt-lookup* (*Branch c l k v r*))
by (*auto simp add*: *set-eq-iff map-to-set-lookup-entries*)

from *r-kv-l-it*[*unfolded map-eq*]
show *?case* **.**
qed

lemma *rm-reverse-iterateoi-impl*: *map-reverse-iterateoi rm-α rm-invar rm-reverse-iterateoi*
proof
fix *m*:: ('*a*, '*b*) *rm*
show *finite* (*dom* (*rm-α m*)) **by** *simp*
next
fix *m*:: ('*a*, '*b*) *rm*
show *map-iterator-rev-linord* (*RBTMapImpl.rm-reverse-iterateoi m*) (*rm-α m*)
unfolding *rm-α-def RBTMapImpl.rm-reverse-iterateoi-def*
by (*transfer*) (*simp add*: *rm-reverse-iterateoi-correct*)
qed

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*
  apply (*unfold-locales*)
  apply (*unfold rm-defs*)
  apply (*case-tac m*)
  apply *simp*
  done

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*
  using *rm-empty-impl* .
interpretation *rm*: *map-lookup rm-α rm-invar rm-lookup*
  using *rm-lookup-impl* .
interpretation *rm*: *map-update rm-α rm-invar rm-update*
  using *rm-update-impl* .
interpretation *rm*: *map-update-dj rm-α rm-invar rm-update-dj*
  using *rm-update-dj-impl* .
interpretation *rm*: *map-delete rm-α rm-invar rm-delete*
  using *rm-delete-impl* .
interpretation *rm*: *finite-map rm-α rm-invar*
  using *rm-is-finite-map* .
interpretation *rm*: *map-iteratei rm-α rm-invar rm-iteratei*
  using *rm-iteratei-impl* .
interpretation *rm*: *map-iterateoi rm-α rm-invar rm-iterateoi*
  using *rm-iterateoi-impl* .
interpretation *rm*: *map-reverse-iterateoi rm-α rm-invar rm-reverse-iterateoi*
  using *rm-reverse-iterateoi-impl* .
interpretation *rm*: *map-add rm-α rm-invar rm-add*
  using *rm-add-impl* .
interpretation *rm*: *map-add-dj rm-α rm-invar rm-add-dj*
  using *rm-add-dj-impl* .
interpretation *rm*: *map-isEmpty rm-α rm-invar rm-isEmpty*
  using *rm-isEmpty-impl* .
interpretation *rm*: *map-sel rm-α rm-invar rm-sel*
  using *rm-sel-impl* .

interpretation *rm*: *map-sel′ rm-α rm-invar rm-sel′*
using *rm-sel′-impl* **.**
interpretation *rm*: *map-to-sorted-list rm-α rm-invar rm-to-list*
using *rm-to-sorted-list-impl* **.**
interpretation *rm*: *list-to-map rm-α rm-invar list-to-rm*
using *list-to-rm-impl* **.**

interpretation *rm*: *map-min rm-α rm-invar rm-min*
using *rm-min-impl* **.**
interpretation *rm*: *map-max rm-α rm-invar rm-max*
using *rm-max-impl* **.**

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* :: $'a \Rightarrow$ *hashcode*
  and *bounded-hashcode* :: *nat* $\Rightarrow$ $'a \Rightarrow$ *hashcode*
  and *def-hashmap-size* :: $'a$ *itself* $\Rightarrow$ *nat*
  assumes *bounded-hashcode-bounds*: $1 < n \Longrightarrow$ *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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-unit-def*)
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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-bool-def*)
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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-int-def*)

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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-nat-def*)
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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-nibble-def*)
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 by(*intro-classes*)(*simp-all add*: *def-hashmap-size-char-def*)
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 using *def-hashmap-size*[**where** *?*′*a*=′*a*] *def-hashmap-size*[**where** *?*′*a*=′*b*]
by(*intro-classes*)(*simp-all add*: *bounded-hashcode-prod-def def-hashmap-size-prod-def*)
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 using *def-hashmap-size*[**where** *?*′*a*=′*a*] *def-hashmap-size*[**where** *?*′*a*=′*b*]
by(*intro-classes*)(*simp-all add*: *bounded-hashcode-sum-def bounded-hashcode-bounds def-hashmap-size-sum-def split*: *sum.split*)
end

instantiation *list* :: (*hashable*) *hashable*
begin
definition *hashcode* = *foldl* (λ*h x*. *h* ∗ *33* + *hashcode x*) *5381*
definition *bounded-hashcode n* = *foldl* (λ*h 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
fix *n* :: *nat* **and** *xs* :: ′*a list*
assume *1* < *n*
thus *bounded-hashcode n xs* < *n* **unfolding** *bounded-hashcode-list-def*
by(*cases xs rule*: *rev-cases*) *simp-all*
next
from *def-hashmap-size*[**where** *?*′*a* = ′*a*]
show *1* < *def-hashmap-size TYPE*(′*a list*)
by(*simp add*: *def-hashmap-size-list-def*)
qed
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 using *def-hashmap-size*[**where** *?'a = 'a*]
by(*intro-classes*)(*simp-all add*: *bounded-hashcode-option-def def-hashmap-size-option-def split*: *option.split*)
end

lemma *hashcode-option-simps* [*simp*]:
hashcode None = 0
hashcode (*Some a*) = *1* + *hashcode a*
by(*simp-all add*: *hashcode-option-def*)

lemma *bounded-hashcode-option-simps* [*simp*]:
bounded-hashcode n None = 0
bounded-hashcode n (*Some a*) = (*bounded-hashcode n a* + *1*) *mod n*
by(*simp-all add*: *bounded-hashcode-option-def*)

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* $\Rightarrow$ *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*
by (*unfold ahm-invar-def*) *blast*

lemma *ahm-invarD*: ⟦ *ahm-invar m*; *m hc* = *Some cm*; *k*∈*dom cm* ⟧ ⟹ *hashcode k* = *hc*
by (*unfold ahm-invar-def*) *blast*

lemma *ahm-invarDne*: ⟦ *ahm-invar m*; *m hc* = *Some cm* ⟧ ⟹ *cm* ≠ *empty*
by (*unfold ahm-invar-def*) *blast*

lemma *ahm-invar-bucket-not-empty*[*simp*]:
ahm-invar m ⟹ *m hc* ≠ *Some Map.empty*
by (*auto dest*: *ahm-invarDne*)

lemmas *ahm-lookup-correct* = *ahm-lookup-def*

lemma *ahm-empty-correct*:
ahm-α ahm-empty = *empty*
ahm-invar ahm-empty
apply (*rule ext*)
apply (*unfold ahm-empty-def*)
apply (*auto simp add*: *ahm-α-def intro*: *ahm-invarI split*: *option.split*)
done

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*)
apply (*rule ext*)
apply (*unfold ahm-update-def*)
apply (*auto simp add*: *ahm-α-def split*: *option.split*)
apply (*rule ahm-invarI*)
apply (*auto dest*: *ahm-invarD ahm-invarDne split*: *split-if-asm*)
apply (*rule ahm-invarI*)
apply (*auto dest*: *ahm-invarD split*: *split-if-asm*)
apply (*drule* (*1*) *ahm-invarD*)
apply *auto*
done

lemma *fun-upd-apply-ne*: *x*≠*y* ⟹ (*f*(*x*:=*v*)) *y* = *f y*
by *simp*

lemma *cancel-one-empty-simp*: *m* |‘ (−{*k*}) = *Map.empty* ⟷ *dom m* ⊆ {*k*}
proof
assume *m* |‘ (− {*k*}) = *Map.empty*
hence {} = *dom* (*m* |‘ (−{*k*})) **by** *auto*

```
  also have ... = dom m − {k} by auto
  finally show dom m ⊆ {k} by blast
next
  assume dom m ⊆ {k}
  hence dom m − {k} = {} by auto
  hence dom (m |‘ (−{k})) = {} by auto
  thus m |‘ (−{k}) = Map.empty by (simp only: dom-empty-simp)
qed

lemma ahm-delete-correct:
  ahm-α (ahm-delete k m) = (ahm-α m) |‘ (−{k})
  ahm-invar m ⟹ ahm-invar (ahm-delete k m)
  apply (rule ext)
  apply (unfold ahm-delete-def)
  apply (auto simp add: ahm-α-def Let-def Map.restrict-map-def
            split: option.split)[1]
  apply (drule-tac x=x in fun-cong)
  apply (auto)[1]
  apply (rule ahm-invarI)
  apply (auto split: split-if-asm option.split-asm dest: ahm-invarD)
  apply (drule (1) ahm-invarD)
  apply (auto simp add: restrict-map-def split: split-if-asm option.split-asm)
  done

lemma ahm-isEmpty-correct: ahm-invar m ⟹ ahm-isEmpty m ⟷ ahm-α m =
Map.empty
proof
  assume ahm-invar m    ahm-isEmpty m
  thus ahm-α m = Map.empty
    by (auto simp add: ahm-isEmpty-def ahm-α-def intro: ext)
next
  assume I: ahm-invar m
    and E: ahm-α m = Map.empty

  show ahm-isEmpty m
  proof (rule ccontr)
    assume ¬ahm-isEmpty m
    then obtain hc bm where MHC: m hc = Some bm
      by (unfold ahm-isEmpty-def)
        (blast elim: nempty-dom dest: domD)
    from ahm-invarDne[OF I, OF MHC] obtain k v where
      BMK: bm k = Some v
      by (blast elim: nempty-dom dest: domD)
    from ahm-invarD[OF I, OF MHC] BMK have [simp]: hashcode k = hc
      by auto
    hence ahm-α m k = Some v
      by (simp add: ahm-α-def MHC BMK)
    with E show False by simp
  qed
```

qed

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*
using *A*
apply (*auto simp add*: *ahm-*α*-def split*: *option.split dest*: *ahm-invarD*[*OF I*])
apply (*frule ahm-invarD*[*OF I*, **where** *k*=*k*])
apply *auto*
done

— Map entries correspond to bucket entries
lemma *ahm-e-is-be*: ⟦
ahm-α *m k* = *Some v*;
!!*bm*. ⟦*m* (*hashcode k*) = *Some bm*; *bm k* = *Some v* ⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
by (*unfold ahm-*α*-def*)
(*auto split*: *option.split-asm*)

4.6.2 Concrete Hashmap

In this section, we define the concrete hashmap that is made from the hashcode 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*)
(λ*v*. *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*) (λ*hclm*. *lm-iteratei* (*snd hclm*)))
proof −
have *aux*: $\bigwedge c\ f.\ (\lambda(hc, lm).\ lm\text{-}iteratei\ lm\ c\ f) = (\lambda m.\ lm\text{-}iteratei\ (snd\ m)\ c\ f)$
by *auto*
show *?thesis*
unfolding *set-iterator-product-def set-iterator-image-alt-def*
iteratei-def[*abs-def*] *aux*
by (*simp add*: *split-beta*)
qed

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* == λ*hc*. *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*))
apply (*auto*
simp add: *rm-map-entry-def rm.delete-correct rm.lookup-correct rm.update-correct*
split: *option.split*)
apply (*auto simp add*: *Map.restrict-map-def fun-upd-def intro*: *ext*)
done

lemma *empty-correct'*:
hm-α' (*empty* ()) = *ahm-empty*
invar (*empty* ())
by(*simp-all add*: *hm-*α'*-def empty-def ahm-empty-def rm-correct invar-def ahm-invar-def*)

lemma *lookup-correct'*:
invar m $\Longrightarrow$ *lookup k m* = *ahm-lookup k* (*hm-*α' *m*)
apply (*unfold lookup-def invar-def*)
apply (*auto split*: *option.split*
simp add: *ahm-lookup-def ahm-*α*-def hm-*α'*-def*

```
                  rm-correct lm-correct)
  done

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 -
  assume invar m
  thus hm-α' (update k v m) = ahm-update k v (hm-α' m)
    apply (unfold invar-def)
    apply (rule ext)
    apply (auto simp add: update-def ahm-update-def hm-α'-def Let-def
                          rm-correct lm-correct
                split: option.split)
    done
  thus invar m ⟹ invar (update k v m)
    by (simp add: invar-def ahm-update-correct)
qed

lemma delete-correct':
  invar m ⟹ hm-α' (delete k m) = ahm-delete k (hm-α' m)
  invar m ⟹ invar (delete k m)
proof -
  assume invar m
  thus hm-α' (delete k m) = ahm-delete k (hm-α' m)
    apply (unfold invar-def)
    apply (rule ext)
   apply (auto simp add: delete-def ahm-delete-def hm-α'-def rm-map-entry-correct
                          rm-correct lm-correct Let-def
                split: option.split option.split-asm)
    done
  thus invar (delete k m) using ‹invar m›
    by (simp add: ahm-delete-correct invar-def)
qed

lemma isEmpty-correct':
  invar hm ⟹ isEmpty hm ⟷ ahm-α (hm-α' hm) = Map.empty
apply (simp add: isEmpty-def rm.isEmpty-correct invar-def
                 ahm-isEmpty-correct[unfolded ahm-isEmpty-def, symmetric])
by (auto simp add: hm-α'-def ahm-α-def fun-eq-iff split: option.split-asm)

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 -
  assume sel: sel hm f = Some r
    and P: ⋀u v. ⟦ahm-α (hm-α' hm) u = Some v; f (u, v) = Some r⟧ ⟹ P
```

from ‹*invar hm*› **have** *IA*: *ahm-invar* (*hm-*α' *hm*) **by**(*simp add*: *invar-def*)
from *TrueI sel* **obtain** *hc lm* **where** *rm-*α *hm hc* = *Some lm*
and *lm-sel lm f* = *Some r*
unfolding *sel-def* **by** (*rule rm.sel-someE*) *simp*
from *TrueI* ‹*lm-sel lm f* = *Some r*›
obtain *k v* **where** *lm-*α *lm k* = *Some v* *f* (*k*, *v*) = *Some r*
by(*rule lm.sel-someE*)
from ‹*rm-*α *hm hc* = *Some lm*› **have** *hm-*α' *hm hc* = *Some* (*lm-*α *lm*)
by(*simp add*: *hm-*α'*-def*)
with *IA* **have** *ahm-*α (*hm-*α' *hm*) *k* = *Some v* **using** ‹*lm-*α *lm k* = *Some v*›
by(*rule ahm-be-is-e*)
thus *P* **using** ‹*f* (*k*, *v*) = *Some r*› **by**(*rule P*)
next
assume *sel*: *sel hm f* = *None*
and α: *ahm-*α (*hm-*α' *hm*) *u* = *Some v*
from ‹*invar hm*› **have** *IA*: *ahm-invar* (*hm-*α' *hm*) **by**(*simp add*: *invar-def*)
from α **obtain** *lm* **where** α*-u*: *hm-*α' *hm* (*hashcode u*) = *Some lm*
and *lm u* = *Some v* **by** (*rule ahm-e-is-be*)
from α*-u* **obtain** *lmc* **where** *rm-*α *hm* (*hashcode u*) = *Some lmc* *lm* = *lm-*α *lmc*
by(*auto simp add*: *hm-*α'*-def split*: *option.split-asm*)
with ‹*lm u* = *Some v*› **have** *lm-*α *lmc u* = *Some v* **by** *simp*
from *sel rm.sel-noneD* [*OF TrueI* - ‹*rm-*α *hm* (*hashcode u*) = *Some lmc*›,
of (λ(*hc*, *lm*). *lm-sel lm f*)]
have *lm-sel lmc f* = *None* **unfolding** *sel-def* **by** *simp*
with *TrueI* **show** *f* (*u*, *v*) = *None* **using** ‹*lm-*α *lmc u* = *Some v*› **by**(*rule lm.sel-noneD*)
qed

lemma *iteratei-correct*$'$:
assumes *invar*: *invar hm*
shows *map-iterator* (*iteratei hm*) (*ahm-*α (*hm-*α' *hm*))
proof –
from *map-iteratei.iteratei-rule*[*OF rm-iteratei-impl*]
have *it-rm*: *map-iterator* (*rm-iteratei hm*) (*rm-*α *hm*) **by** *simp*

from *map-iteratei.iteratei-rule*[*OF lm-iteratei-impl*]
have *it-lm*: $\bigwedge$*lm*. *map-iterator* (*lm-iteratei lm*) (*lm-*α *lm*) **by** *simp*

from *set-iterator-product-correct* [*OF it-rm*, *of* λ*hclm*. *lm-iteratei* (*snd hclm*)
λ*hclm*. *map-to-set* (*lm-*α (*snd hclm*)), *OF it-lm*]
have *it-prod*: *set-iterator*
(*set-iterator-product* (*rm-iteratei hm*) (λ*hclm*. *lm-iteratei* (*snd hclm*)))
(*SIGMA hclm*:*map-to-set* (*rm-*α *hm*). *map-to-set* (*lm-*α (*snd hclm*))) **by** *simp*

show *?thesis* **unfolding** *iteratei-alt-def*
proof (*rule set-iterator-image-correct*[*OF it-prod*])
from *invar*

```
    show inj-on snd
      (SIGMA hclm:map-to-set (rm-α hm). map-to-set (lm-α (snd hclm)))
      apply (simp add: inj-on-def invar-def ahm-invar-def hm-α'-def Ball-def
                  map-to-set-def split: option.splits)
      apply (metis domI option.inject)
    done
  next
    from invar
    show map-to-set (ahm-α (hm-α' hm)) =
        snd ‘ (SIGMA hclm:map-to-set (rm-α hm). map-to-set (lm-α (snd hclm)))

      apply (simp add: inj-on-def invar-def ahm-invar-def hm-α'-def Ball-def
                  map-to-set-def set-eq-iff image-iff split: option.splits)
      apply (auto simp add: dom-def ahm-α-def split: option.splits)
    done
  qed
qed


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
  show HashMap-Impl.empty () ∈ {hm. HashMap-Impl.invar hm}
    by(simp add: HashMap-Impl.empty-correct')
qed

lemma impl-of-RBT-HM-invar [simp, intro!]: HashMap-Impl.invar (impl-of-RBT-HM
hm)
using impl-of-RBT-HM[of hm] by simp
```

lemma *RBT-HM-imp-of-RBT-HM* [*code abstype*]:
RBT-HM (*impl-of-RBT-HM hm*) = *hm*
by(*fact impl-of-RBT-HM-inverse*)

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* × '*v* ⇀ '*a*) ⇀ '*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* ()
by(*simp add*: *hm-empty-const-def empty-correct' RBT-HM-inverse*)

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*)
by(*simp add*: *hm-update-def update-correct' RBT-HM-inverse*)

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*)
by(*simp add*: *hm-delete-def delete-correct' RBT-HM-inverse*)

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*})
by(*auto simp add*: *hm-α-def hm-correct′ hm-empty-def ahm-correct hm-lookup-def*)

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*
by (*unfold-locales*)
(*simp-all add*: *hm-aux-correct*)

lemma *hm-lookup-impl*: *map-lookup hm-α hm-invar hm-lookup*
by (*unfold-locales*)
(*simp-all add*: *hm-aux-correct*)

lemma *hm-update-impl*: *map-update hm-α hm-invar hm-update*
by (*unfold-locales*)
(*simp-all add*: *hm-aux-correct*)

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*
by (*unfold-locales*)
(*simp-all add*: *hm-aux-correct*)

lemma *hm-finite*[*simp*, *intro*!]:
finite (*dom* (*hm-α m*))

proof(*cases m*)
case (*RBT-HM m′*)
hence *SS*: *dom* (*hm-α m*) ⊆ ⋃{ *dom* (*lm-α lm*) | *lm hc. rm-α m′ hc* = *Some lm* }
apply(*clarsimp simp add*: *RBT-HM-inverse hm-α-def hm-α′-def* [*abs-def*] *ahm-α-def* [*abs-def*])
apply(*auto split*: *option.split-asm option.split*)
done
moreover have *finite* ... (**is** *finite* (⋃ *?A*))
proof
have { *dom* (*lm-α lm*) | *lm hc. rm-α m′ hc* = *Some lm* }
⊆ (*λhc. dom* (*lm-α* (*the* (*rm-α m′ hc*)))) ‘ (*dom* (*rm-α m′*))
(**is** *?S* ⊆ -)
by *force*
thus *finite ?A* **by**(*rule finite-subset*) *auto*
next
fix *M*
assume *M* ∈ *?A*
thus *finite M* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *RBT-HM* **by**(*rule finite-subset*)
qed

lemma *hm-is-finite-map*: *finite-map hm-α hm-invar*
by(*unfold-locales*) *auto*

lemma *hm-isEmpty-impl*: *map-isEmpty hm-α hm-invar hm-isEmpty*
by(*unfold-locales*)(*simp add*: *hm-isEmpty-def hm-α-def isEmpty-correct′*)

lemma *hm-sel-impl*: *map-sel hm-α hm-invar hm-sel*
unfolding *hm-sel-def* [*abs-def*] *hm-α-def o-def*
by(*rule map-sel-altI*)(*blast intro*: *sel-correct′*[*OF impl-of-RBT-HM-invar*])+

lemma *hm-sel′-impl*: *map-sel′ hm-α hm-invar hm-sel′*
unfolding *hm-sel′-def*
by(*rule sel-sel′-correct hm-sel-impl*)+

lemma *hm-iteratei-impl*:
map-iteratei hm-α hm-invar hm-iteratei
apply (*unfold-locales*)
apply *simp*
apply (*unfold hm-α-def hm-iteratei-def o-def*)
apply(*rule iteratei-correct′*[*OF impl-of-RBT-HM-invar*]) **.**

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* **using** *hm-empty-impl*
.
interpretation *hm*: *map-lookup hm-α hm-invar hm-lookup* **using** *hm-lookup-impl*
.
interpretation *hm*: *map-update hm-α hm-invar hm-update* **using** *hm-update-impl*
.
interpretation *hm*: *map-update-dj hm-α hm-invar hm-update-dj* **using** *hm-update-dj-impl*
.
interpretation *hm*: *map-delete hm-α hm-invar hm-delete* **using** *hm-delete-impl* **.**
interpretation *hm*: *finite-map hm-α hm-invar* **using** *hm-is-finite-map* **.**
interpretation *hm*: *map-sel hm-α hm-invar hm-sel* **using** *hm-sel-impl* **.**
interpretation *hm*: *map-sel′ hm-α hm-invar hm-sel′* **using** *hm-sel′-impl* **.**
interpretation *hm*: *map-isEmpty hm-α hm-invar hm-isEmpty* **using** *hm-isEmpty-impl*
.
interpretation *hm*: *map-iteratei hm-α hm-invar hm-iteratei* **using** *hm-iteratei-impl*
.
interpretation *hm*: *map-add hm-α hm-invar hm-add* **using** *hm-add-impl* **.**
interpretation *hm*: *map-add-dj hm-α hm-invar hm-add-dj* **using** *hm-add-dj-impl*
.
interpretation *hm*: *map-to-list hm-α hm-invar hm-to-list* **using** *hm-to-list-impl* **.**
interpretation *hm*: *list-to-map hm-α hm-invar list-to-hm* **using** *list-to-hm-impl* **.**

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* × ('*key*, '*val*) *trie*) *list*

lemma *trie-induct* [*case-names Trie*, *induct type*]:
  $(\bigwedge vo\ kvs.\ (\bigwedge k\ t.\ (k,\ t) \in set\ kvs \Longrightarrow P\ t) \Longrightarrow P\ (Trie\ vo\ kvs)) \Longrightarrow P\ t$
apply(*induction-schema*)
  apply *pat-completeness*
 apply(*lexicographic-order*)
done

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
by(*cases t*) *simp*

lemma *iteratei-interrupt*:
  ¬ *c* σ ⟹ *iteratei t c f* σ = σ
unfolding *iteratei-def* **by** (*simp add*: *iteratei-postfixed-interrupt*)

lemma *iteratei-postfixed-alt-def* :
  iteratei-postfixed ks ((*Trie vo ts*)::('*key*, '*val*) *trie*) =
   (*set-iterator-union*

```
    (option-case set-iterator-emp (λv. set-iterator-sng (ks, v)) vo)
    (set-iterator-image snd
    (set-iterator-product (foldli ts)
      (λ(k, t'). iteratei-postfixed (k # ks) t'))
    ))
proof -
  have aux: ⋀c f. (λ(k, t). iteratei-postfixed (k # ks) t c f) =
            (λa. iteratei-postfixed (fst a # ks) (snd a) c f)
    by auto

  show ?thesis
    apply (rule ext)+ apply (rename-tac c f σ)
    apply (simp add: set-iterator-product-def set-iterator-image-filter-def
                set-iterator-union-def set-iterator-sng-def set-iterator-image-alt-def
                 prod-case-beta set-iterator-emp-def
          split: option.splits)
    apply (simp add: aux)
  done
qed
```

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 (cases ks)
  case Nil thus ?thesis by simp
next
  case (Cons k ks')
  note ks-eq[simp] = Cons

  show ?thesis
  proof (cases map-of kvs k)
    case None thus ?thesis
      apply (simp)
      apply (auto simp add: map-of-eq-None-iff image-iff Ball-def)
    done
  next
    case (Some t') note map-eq = this
    from invar have dist-kvs: distinct (map fst kvs) by simp

    from map-of-eq-Some-iff[OF dist-kvs, of k] map-eq
    show ?thesis by simp metis
  qed
qed
```

```
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))
using lookup-eq-Some-iff[of vo kvs ks, OF invar]
  apply (cases ks)
    apply auto[]
  apply (auto split: option.split)
    apply (metis option.simps(3) weak-map-of-SomeI)
    apply (metis option.exhaust)
    apply (metis option.exhaust)
done
```

4.8.3 The empty trie

```
lemma trie-invar-empty [simp, intro!]: trie-invar empty
by(simp add: empty-def)

lemma lookup-empty [simp]:
  lookup empty = Map.empty
proof
  fix ks show lookup empty ks = Map.empty ks
    by(cases ks)(auto simp add: empty-def)
qed

lemma lookup-empty' [simp]:
  lookup (Trie None []) ks = None
by(simp add: lookup-empty[unfolded empty-def])
```

4.8.4 Emptyness check

```
lemma isEmpty-conv:
  isEmpty ts ⟷ ts = Trie None []
by(cases ts)(simp)

lemma update-not-empty: ¬ isEmpty (update t ks v)
apply(cases t)
apply(rename-tac kvs)
apply(cases ks)
apply(case-tac [2] kvs)
apply auto
done

lemma isEmpty-lookup-empty:
  trie-invar t ⟹ isEmpty t ⟷ lookup t = Map.empty
proof(induct t)
  case (Trie vo kvs)
  thus ?case
    apply(cases kvs)
```

apply(*auto simp add*: *fun-eq-iff elim*: *allE*[**where** x=[]])
apply(*erule meta-allE*)+
apply(*erule meta-impE*)
apply(*rule disjI1*)
apply(*fastforce intro*: *exI*[**where** $x=a \# b$, *standard*])+
done
qed

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(*induct t ks v arbitrary*: *ks' rule*: *update.induct*)
case (*1 vo ts v*)
show *?case* **by**(*fastforce simp add*: *neq-Nil-conv dest*: *not-sym*)
next
case (*2 vo ts k ks v*)
note *IH* = ‹$\bigwedge$*t ks'*. *lookup* (*update t ks v*) *ks'* = (*if ks* = *ks' then Some v else lookup t ks'*)›
show *?case* **by**(*cases ks'*)(*auto simp add*: *map-of-update-with-aux IH split*: *option.split*)
qed

lemma *lookup-update'*:
lookup (*update t ks v*) = (*lookup t*)(*ks* $\mapsto$ *v*)
by(*rule ext*)(*simp add*: *lookup-update*)

lemma *trie-invar-update*: *trie-invar t* $\Longrightarrow$ *trie-invar* (*update t ks v*)
by(*induct t ks v rule*: *update.induct*)(*auto simp add*: *set-update-with-aux update-not-empty split*: *option.splits*)

4.8.6 Trie removal

lemma *delete-eq-empty-lookup-other-fail*:
⟦ *delete t ks* = *Trie None* []; *ks'* $\neq$ *ks* ⟧ $\Longrightarrow$ *lookup t ks'* = *None*
proof(*induct t ks arbitrary*: *ks' rule*: *delete.induct*)
case (*1 vo ts*)
thus *?case* **by**(*auto simp add*: *neq-Nil-conv*)
next
case (*2 vo ts k ks*)
note *IH* = ‹$\bigwedge$*t ks'*. ⟦*map-of ts k* = *Some t*; *delete t ks* = *Trie None* []; *ks'* $\neq$ *ks*⟧ $\Longrightarrow$ *lookup t ks'* = *None*›
note *ks'* = ‹*ks'* $\neq$ *k* # *ks*›
note *empty* = ‹*delete* (*Trie vo ts*) (*k* # *ks*) = *Trie None* []›
show *?case*
proof(*cases map-of ts k*)
case (*Some t*)
from *Some empty* **show** *?thesis*
proof(*cases ks'*)

```
      case (Cons k′ ks″)
      with Some empty ks′ show ?thesis
      proof(cases k′ = k)
        case False
        from Some Cons empty have delete-aux k ts = []
          by(clarsimp simp add: Let-def split: split-if-asm)
        with False have map-of ts k′ = None
       by(cases map-of ts k′)(auto dest: map-of-is-SomeD simp add: delete-aux-eq-Nil-conv)
        thus ?thesis using False Some Cons empty by simp
      next
        case True
        with Some empty ks′ Cons show ?thesis
          by(clarsimp simp add: IH Let-def isEmpty-conv split: split-if-asm)
      qed
    qed(simp add: Let-def split: split-if-asm)
  next
    case None thus ?thesis using empty by simp
  qed
qed

lemma lookup-delete:
  trie-invar t ⟹ lookup (delete t ks) ks′ = (if ks = ks′ then None else lookup t
ks′)
proof(induct t ks arbitrary: ks′ rule: delete.induct)
  case (1 vo ts)
  show ?case by(fastforce dest: not-sym simp add: neq-Nil-conv)
next
  case (2 vo ts k ks)
  note IH = ‹⋀t ks′. ⟦ map-of ts k = Some t; trie-invar t ⟧
          ⟹ lookup (delete t ks) ks′ = (if ks = ks′ then None else lookup t ks′)›
  note invar = ‹trie-invar (Trie vo ts)›
  show ?case
  proof(cases ks′)
    case Nil thus ?thesis
      by(simp split: option.split add: Let-def)
  next
    case (Cons k′ ks″)[simp]
    show ?thesis
    proof(cases k′ = k)
      case False thus ?thesis using invar
       by(auto simp add: Let-def update-conv′ map-of-delete-aux split: option.split)
    next
      case True[simp]
      show ?thesis
      proof(cases map-of ts k)
        case None thus ?thesis by simp
      next
        case (Some t)
        thus ?thesis
```

```
        proof(cases isEmpty (delete t ks))
          case True
          with Some invar show ?thesis
         by(auto simp add: map-of-delete-aux isEmpty-conv dest: delete-eq-empty-lookup-other-fail)
        next
          case False
        thus ?thesis using Some IH[of t ks''] invar by(auto simp add: update-conv')
        qed
      qed
    qed
  qed
qed

lemma lookup-delete':
  trie-invar t ⟹ lookup (delete t ks) = (lookup t)(ks := None)
by(rule ext)(simp add: lookup-delete)

lemma trie-invar-delete:
  trie-invar t ⟹ trie-invar (delete t ks)
proof(induct t ks rule: delete.induct)
  case (1 vo ts)
  thus ?case by simp
next
  case (2 vo ts k ks)
  note invar = ‹trie-invar (Trie vo ts)›
  show ?case
  proof(cases map-of ts k)
    case None
    thus ?thesis using invar by simp
  next
    case (Some t)
    with invar have trie-invar t by auto
    with Some have trie-invar (delete t ks) by(rule 2)
    from invar Some have distinct: distinct (map fst ts)    ¬ isEmpty t by auto
    show ?thesis
    proof(cases isEmpty (delete t ks))
      case True
      { fix k' t'
        assume k't': (k', t') ∈ set (delete-aux k ts)
        with distinct have map-of (delete-aux k ts) k' = Some t' by simp
        hence map-of ts k' = Some t' using distinct
          by (auto
            simp del: map-of-eq-Some-iff map-upd-eq-restrict
            simp add: map-of-delete-aux
            split: split-if-asm)
        with invar  have ¬ isEmpty t' ∧ trie-invar t' by auto }
      with invar have trie-invar (Trie vo (delete-aux k ts)) by auto
      thus ?thesis using True Some by(simp)
    next
```

```
    case False
    { fix k' t'
      assume k't':(k', t') ∈ set (AList.update k (delete t ks) ts)
      hence map-of (AList.update k (delete t ks) ts) k' = Some t'
        using invar by(auto simp add: distinct-update)
     hence eq: ((map-of ts)(k ↦ delete t ks)) k' = Some t' unfolding update-conv
.
      have ¬ isEmpty t' ∧ trie-invar t'
      proof(cases k' = k)
        case True
        with eq have t' = delete t ks by simp
        with ‹trie-invar (delete t ks)› False
        show ?thesis by simp
      next
        case False
        with eq distinct have (k', t') ∈ set ts by simp
        with invar show ?thesis by auto
      qed }
    thus ?thesis using Some invar False by(auto simp add: distinct-update)
  qed
 qed
qed
```

4.8.7 Domain of a trie

```
lemma dom-lookup:
  dom (lookup (Trie vo kts)) =
  (⋃k∈dom (map-of kts). Cons k ‘ dom (lookup (the (map-of kts k)))) ∪
  (if vo = None then {} else {[]})
unfolding dom-def
apply(rule sym)
apply(safe)
  apply simp
 apply(clarsimp simp add: split-if-asm)
apply(case-tac x)
apply(auto split: option.split-asm)
done

lemma finite-dom-trie-lookup:
  finite (dom (lookup t))
proof(induct t)
  case (Trie vo kts)
  have finite (⋃k∈dom (map-of kts). Cons k ‘ dom (lookup (the (map-of kts k))))
  proof(rule finite-UN-I)
    show finite (dom (map-of kts)) by(rule finite-dom-map-of)
  next
    fix k
    assume k ∈ dom (map-of kts)
    then obtain v where (k, v) ∈ set kts    map-of kts k = Some v by(auto dest:
```

map-of-SomeD)
  hence *finite* (*dom* (*lookup* (*the* (*map-of kts k*)))) **by** *simp*(*rule Trie*)
  thus *finite* (*Cons k* ‘ *dom* (*lookup* (*the* (*map-of kts k*)))) **by**(*rule finite-imageI*)
 qed
 thus *?case* **by**(*simp add*: *dom-lookup*)
qed

lemma *dom-lookup-empty-conv*: *trie-invar t* $\Longrightarrow$ *dom* (*lookup t*) = {} $\longleftrightarrow$ *isEmpty t*
proof(*induct t*)
 case (*Trie vo kvs*)
 show *?case*
 proof
  assume *dom*: *dom* (*lookup* (*Trie vo kvs*)) = {}
  have *vo* = *None*
  proof(*cases vo*)
   case (*Some v*)
   hence [] $\in$ *dom* (*lookup* (*Trie vo kvs*)) **by** *auto*
   with *dom* **have** *False* **by** *simp*
   thus *?thesis* **..**
  qed
  moreover have *kvs* = []
  proof(*cases kvs*)
   case (*Cons kt kvs'*)
   with ‹*trie-invar* (*Trie vo kvs*)›
   have ¬ *isEmpty* (*snd kt*)   *trie-invar* (*snd kt*) **by** *auto*
   from *Cons* **have** (*fst kt*, *snd kt*) $\in$ *set kvs* **by** *simp*
   hence *dom* (*lookup* (*snd kt*)) = {} $\longleftrightarrow$ *isEmpty* (*snd kt*)
    using ‹*trie-invar* (*snd kt*)› **by**(*rule Trie*)
   with ‹¬ *isEmpty* (*snd kt*)› **have** *dom* (*lookup* (*snd kt*)) $\neq$ {} **by** *simp*
   with *dom Cons* **have** *False* **by**(*auto simp add*: *dom-lookup*)
   thus *?thesis* **..**
  qed
  ultimately show *isEmpty* (*Trie vo kvs*) **by** *simp*
 next
  assume *isEmpty* (*Trie vo kvs*)
  thus *dom* (*lookup* (*Trie vo kvs*)) = {}
   by(*simp add*: *lookup-empty*[*unfolded empty-def*])
 qed
qed

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*)))
using *invar*
proof (*induct t arbitrary*: *ks0*)

```
case (Trie vo kvs)
note ind-hyp = Trie(1)
note invar = Trie(2)

from invar
have dist-fst-kvs : distinct (map fst kvs)
 and dist-kvs: distinct kvs
 and invar-child: ⋀k t. (k, t) ∈ set kvs ⟹ trie-invar t
by (simp-all add: Ball-def distinct-map)

— root iterator
def it-vo ≡ (case vo of None ⇒ set-iterator-emp
                 | Some v ⇒ set-iterator-sng (ks0, v)) ::
            ('key list × 'val, 'σ) set-iterator
def vo-S ≡ case vo of None ⇒ {} | Some v ⇒ {(ks0, v)}
have it-vo-OK: set-iterator it-vo vo-S
  unfolding it-vo-def vo-S-def
  by (simp split: option.split
        add: set-iterator-emp-correct set-iterator-sng-correct)

— children iterator
def it-prod ≡ (set-iterator-product (foldli kvs)
    (λ(k, y). iteratei-postfixed (k # ks0) y))::
      (('key × ('key, 'val) trie) × 'key list × 'val, 'σ) set-iterator

  def it-prod-S ≡ SIGMA kt:set kvs.
     (λksv. (rev (fst ksv) @ ((fst kt) # ks0), snd ksv)) `
     map-to-set (lookup (snd kt))

have it-prod-OK: set-iterator it-prod it-prod-S
proof -
  from set-iterator-foldli-correct[OF dist-kvs]
  have it-foldli: set-iterator (foldli kvs) (set kvs) .

  { fix kt
    assume kt-in: kt ∈ set kvs
    hence k-t-in: (fst kt, snd kt) ∈ set kvs by simp

    note ind-hyp [OF k-t-in, OF invar-child[OF k-t-in], of fst kt # ks0]
  } note it-child = this

  show ?thesis
    unfolding it-prod-def it-prod-S-def
    apply (rule set-iterator-product-correct [OF it-foldli])
    apply (insert it-child)
    apply (simp add: prod-case-beta)
  done
qed
```

```
  have it-image-OK : set-iterator (set-iterator-image snd it-prod) (snd ` it-prod-S)
  proof (rule set-iterator-image-correct[OF it-prod-OK])
    from dist-fst-kvs
    have ⋀k v1 v2. (k, v1) ∈ set kvs ⟹ (k, v2) ∈ set kvs ⟹ v1 = v2
      by (induct kvs) (auto simp add: image-iff)
    thus inj-on snd it-prod-S
      unfolding inj-on-def it-prod-S-def
      apply (simp add: image-iff Ball-def map-to-set-def)
      apply auto
    done
  qed auto

  — overall iterator
  have it-all-OK: set-iterator
      ((iteratei-postfixed ks0 (Trie vo kvs)):: ('key list × 'val, 'σ) set-iterator)
    (vo-S ∪ snd ` it-prod-S)
    unfolding iteratei-postfixed-alt-def
      it-vo-def[symmetric]
      it-prod-def[symmetric]
  proof (rule set-iterator-union-correct [OF it-vo-OK it-image-OK])
    show vo-S ∩ snd ` it-prod-S = {}
      unfolding vo-S-def it-prod-S-def
      by (simp split: option.split add: set-eq-iff image-iff)
  qed

  — rewrite result set
  have it-set-rewr: ((λksv. (rev (fst ksv) @ ks0, snd ksv)) `
      map-to-set (lookup (Trie vo kvs))) = (vo-S ∪ snd ` it-prod-S)
    (is ?ls = ?rs)
    apply (simp add: map-to-set-def lookup-eq-Some-iff[OF invar]
                set-eq-iff image-iff vo-S-def it-prod-S-def Ball-def Bex-def)
    apply (simp split: option.split del: ex-simps add: ex-simps[symmetric])
    apply (intro allI impI iffI)
    apply auto[]
    apply (metis append-Cons append-Nil append-assoc rev.simps)
  done

  — done
  show ?case
    unfolding it-set-rewr using it-all-OK by fast
qed

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)
unfolding trie-reverse-key-def by auto
```

lemma *trie-reverse-key-reverse*[*simp*] :
  trie-reverse-key (*trie-reverse-key ksv*) = *ksv*
by (*simp add*: *trie-reverse-key-def*)

lemma *trie-iteratei-correct*:
  assumes *invar*: *trie-invar* (t :: ($'key$, $'val$) *trie*)
  shows *set-iterator* ((*iteratei t*)::($'key$ *list* $\times$ $'val$, $'\sigma$) *set-iterator*)
    (*trie-reverse-key* ‘ (*map-to-set* (*lookup t*)))
unfolding *trie-reverse-key-def*[*abs-def*] *iteratei-def*[*abs-def*]
using *iteratei-postfixed-correct* [*OF invar*, *of* []]
by *simp*

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

4.9.1 Abstract type definition

typedef ($'key$, $'val$) *trie* =
  {t :: ($'key$, $'val$) *Trie-Impl.trie*. *trie-invar t*}
  morphisms *impl-of Trie*
proof
  show *Trie-Impl.empty* $\in$ *?trie* **by**(*simp*)
qed

lemma *trie-invar-impl-of* [*simp*, *intro*]: *trie-invar* (*impl-of t*)
using *impl-of*[*of t*] **by** *simp*

lemma *Trie-impl-of* [*code abstype*]: *Trie* (*impl-of t*) = *t*
by(*rule impl-of-inverse*)

4.9.2 Primitive operations

definition *empty* :: ($'key$, $'val$) *trie*
where *empty* = *Trie* (*Trie-Impl.empty*)

definition *update* :: $'key$ *list* $\Rightarrow$ $'val$ $\Rightarrow$ ($'key$, $'val$) *trie* $\Rightarrow$ ($'key$, $'val$) *trie*
where *update ks v t* = *Trie* (*Trie-Impl.update* (*impl-of t*) *ks v*)

definition *delete* :: $'key$ *list* $\Rightarrow$ ($'key$, $'val$) *trie* $\Rightarrow$ ($'key$, $'val$) *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))
unfolding *iteratei-def set-iterator-image-alt-def*
apply *(subgoal-tac (λx. f (trie-reverse-key x)) = (λ(ks, v). f (rev ks, v)))*
apply *(auto simp add: trie-reverse-key-def)*
done

lemma *impl-of-empty [code abstract]: impl-of empty = Trie-Impl.empty*
by(*simp add: empty-def Trie-inverse*)

lemma *impl-of-update [code abstract]: impl-of (update ks v t) = Trie-Impl.update (impl-of t) ks v*
by(*simp add: update-def Trie-inverse trie-invar-update*)

lemma *impl-of-delete [code abstract]: impl-of (delete ks t) = Trie-Impl.delete (impl-of t) ks*
by(*simp add: delete-def Trie-inverse trie-invar-delete*)

4.9.3 Correctness of primitive operations

lemma *lookup-empty [simp]: lookup empty = Map.empty*
by(*simp add: lookup-def empty-def Trie-inverse*)

lemma *lookup-update [simp]: lookup (update ks v t) = (lookup t)(ks ↦ v)*
by(*simp add: lookup-def update-def Trie-inverse trie-invar-update lookup-update'*)

lemma *lookup-delete [simp]: lookup (delete ks t) = (lookup t)(ks := None)*
by(*simp add: lookup-def delete-def Trie-inverse trie-invar-delete lookup-delete'*)

lemma *isEmpty-lookup: isEmpty t ⟷ lookup t = Map.empty*
by(*simp add: isEmpty-def lookup-def isEmpty-lookup-empty*)

lemma *finite-dom-lookup: finite (dom (lookup t))*
by(*simp add: lookup-def finite-dom-trie-lookup*)

lemma *iteratei-correct*:

map-iterator (*iteratei m*) (*lookup m*)
proof −
note *it-base* = *Trie-Impl.trie-iteratei-correct* [*of impl-of m*]
show *?thesis*
unfolding *iteratei-def lookup-def*
apply (*rule set-iterator-image-correct* [*OF it-base*])
apply (*simp-all add*: *set-eq-iff image-iff inj-on-def*)
done
qed

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
qed(*simp add*: *equal-trie-def impl-of-inject*)
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* $\times\ 'v,\ '\sigma)$ *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*
by(*unfold-locales*)(*simp-all add*: *tm-defs fun-eq-iff*)

lemma *tm-lookup-impl*: *map-lookup tm-*α *tm-invar tm-lookup*
by(*unfold-locales*)(*auto simp add*: *tm-defs*)

lemma *tm-update-impl*: *map-update tm-α tm-invar tm-update*
by(*unfold-locales*)(*simp-all add*: *tm-defs*)

lemma *tm-update-dj-impl*: *map-update-dj tm-α tm-invar tm-update-dj*
unfolding *tm-update-dj-def*
by(*rule map-update.update-dj-by-update*)(*rule tm-update-impl*)

lemma *tm-delete-impl*: *map-delete tm-α tm-invar tm-delete*
by(*unfold-locales*)(*simp-all add*: *tm-defs*)

lemma *tm-α-finite* [*simp*, *intro*!]:
finite (*dom* (*tm-α m*))
by(*simp add*: *tm-defs finite-dom-lookup*)

lemma *tm-is-finite-map*: *finite-map tm-α tm-invar*
by *unfold-locales simp*

lemma *tm-iteratei-impl*: *map-iteratei tm-α tm-invar tm-iteratei*
by(*unfold-locales*)(*simp*, *unfold tm-defs*, *rule iteratei-correct*)

lemma *tm-add-impl*: *map-add tm-α tm-invar tm-add*
unfolding *tm-add-def*
by(*rule it-add-correct tm-iteratei-impl tm-update-impl*)+

lemma *tm-add-dj-impl*: *map-add-dj tm-α tm-invar tm-add-dj*
unfolding *tm-add-dj-def*
by(*rule map-add.add-dj-by-add tm-add-impl*)+

lemma *tm-isEmpty-impl*: *map-isEmpty tm-α tm-invar tm-isEmpty*
by *unfold-locales*(*simp add*: *tm-defs isEmpty-lookup*)

lemma *tm-sel-impl*: *map-sel tm-α tm-invar tm-sel*
unfolding *tm-sel-def*
by(*rule iti-sel-correct tm-iteratei-impl*)+

lemma *tm-sel'-impl*: *map-sel' tm-α tm-invar tm-sel'*
unfolding *tm-sel'-def*
by(*rule iti-sel'-correct tm-iteratei-impl*)+

lemma *tm-ball-impl*: *map-ball tm-α tm-invar tm-ball*
unfolding *tm-ball-def*
by(*rule sel-ball-correct tm-sel-impl*)+

lemma *tm-to-list-impl*: *map-to-list tm-α tm-invar tm-to-list*
unfolding *tm-to-list-def*
by(*rule it-map-to-list-correct tm-iteratei-impl*)+

lemma *list-to-tm-impl*: *list-to-map tm-α tm-invar list-to-tm*
unfolding *list-to-tm-def*

by(*rule gen-list-to-map-correct tm-empty-impl tm-update-impl*)+

interpretation *tm*: *map-empty tm-*α *tm-invar tm-empty*
using *tm-empty-impl* **.**
interpretation *tm*: *map-lookup tm-*α *tm-invar tm-lookup*
using *tm-lookup-impl* **.**
interpretation *tm*: *map-update tm-*α *tm-invar tm-update*
using *tm-update-impl* **.**
interpretation *tm*: *map-update-dj tm-*α *tm-invar tm-update-dj*
using *tm-update-dj-impl* **.**
interpretation *tm*: *map-delete tm-*α *tm-invar tm-delete*
using *tm-delete-impl* **.**
interpretation *tm*: *finite-map tm-*α *tm-invar*
using *tm-is-finite-map* **.**
interpretation *tm*: *map-iteratei tm-*α *tm-invar tm-iteratei*
using *tm-iteratei-impl* **.**
interpretation *tm*: *map-add tm-*α *tm-invar tm-add*
using *tm-add-impl* **.**
interpretation *tm*: *map-add-dj tm-*α *tm-invar tm-add-dj*
using *tm-add-dj-impl* **.**
interpretation *tm*: *map-isEmpty tm-*α *tm-invar tm-isEmpty*
using *tm-isEmpty-impl* **.**
interpretation *tm*: *map-sel tm-*α *tm-invar tm-sel*
using *tm-sel-impl* **.**
interpretation *tm*: *map-sel′ tm-*α *tm-invar tm-sel′*
using *tm-sel′-impl* **.**
interpretation *tm*: *map-ball tm-*α *tm-invar tm-ball*
using *tm-ball-impl* **.**
interpretation *tm*: *map-to-list tm-*α *tm-invar tm-to-list*
using *tm-to-list-impl* **.**
interpretation *tm*: *list-to-map tm-*α *tm-invar list-to-tm*
using *list-to-tm-impl* **.**

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
apply(*induct get*≡*op* ! :: *'b list* ⇒ *nat* ⇒ *'b sz i xs c f* σ *rule*: *idx-iteratei-aux.induct*)
apply(*subst* (*1 2*) *idx-iteratei-aux.simps*)
apply *simp*
done

lemma *idx-iteratei-array-get-Array-conv-nth*:
idx-iteratei array-get array-length (*Array xs*) = *idx-iteratei nth length xs*
by(*simp add*: *idx-iteratei-def fun-eq-iff idx-iteratei-aux-array-get-Array-conv-nth*)

lemma *idx-iteratei-aux-nth-conv-foldli-drop*:

fixes *xs* :: *'b list*
assumes $i \leq$ *length xs*
shows *idx-iteratei-aux op* ! (*length xs*) *i xs c f* σ = *foldli* (*drop* (*length xs* − *i*) *xs*) *c f* σ
using *assms*
proof(*induct get*≡*op* ! :: *'b list* ⇒ *nat* ⇒ *'b sz*≡ *length xs i xs c f* σ *rule*: *idx-iteratei-aux.induct*)
case (*1 i l c f* σ)
show *?case*
proof(*cases i* = *0* ∨ ¬ *c* σ)
case *True* **thus** *?thesis*
by(*subst idx-iteratei-aux.simps*)(*auto*)
next
case *False*
hence *i*: *i* > *0* **and** *c*: *c* σ **by** *auto*
hence *idx-iteratei-aux op* ! (*length l*) *i l c f* σ = *idx-iteratei-aux op* ! (*length l*) (*i* − *1*) *l c f* (*f* (*l* ! (*length l* − *i*)) σ)
by(*subst idx-iteratei-aux.simps*) *simp*
also have ... = *foldli* (*drop* (*length l* − (*i* − *1*)) *l*) *c f* (*f* (*l* ! (*length l* − *i*)) σ)
using ‹*i* ≤ *length l*› *i c* **by** −(*rule 1*, *auto*)
also from ‹*i* ≤ *length l*› *i*
have *drop* (*length l* − *i*) *l* = (*l* ! (*length l* − *i*)) # *drop* (*length l* − (*i* − *1*)) *l*
by(*subst nth-drop'*[*symmetric*])(*simp-all*, *metis Suc-eq-plus1-left add-diff-assoc*)
hence *foldli* (*drop* (*length l* − (*i* − *1*)) *l*) *c f* (*f* (*l* ! (*length l* − *i*)) σ) = *foldli* (*drop* (*length l* − *i*) *l*) *c f* σ
using *c* **by** *simp*
finally show *?thesis* **.**
qed
qed

lemma *idx-iteratei-nth-length-conv-foldli*: *idx-iteratei nth length* = *foldli*
by(*rule ext*)+(*simp add*: *idx-iteratei-def idx-iteratei-aux-nth-conv-foldli-drop*)

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*
by(*cases hm*)(*simp*)

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok len h* [] = *True*
by(*simp add*: *bucket-ok-def*)

lemma *bucket-okD*:
  ⟦ *bucket-ok len h xs*; (*k*, *v*) ∈ *set xs* ⟧
  ⟹ *bounded-hashcode len k* = *h*
by(*auto simp add*: *bucket-ok-def*)

lemma *bucket-okI*:
($\bigwedge k$. $k \in$ *fst* ‘ *set kvs* $\Longrightarrow$ *bounded-hashcode len k* $= h$) $\Longrightarrow$ *bucket-ok len h kvs*
by(*simp add*: *bucket-ok-def*)

4.11.3 *ahm-invar*

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux n a*
obtains $\forall$ *h*. *h* $<$ *array-length a* $\longrightarrow$ *bucket-ok* (*array-length a*) *h* (*array-get a h*) $\wedge$ *distinct* (*map fst* (*array-get a h*))
and *n* $=$ *array-foldl* (λ- *n kvs*. *n* $+$ *length kvs*) *0 a* **and** *array-length a* $>$ *1*
using *assms* **unfolding** *ahm-invar-aux-def* **by** *blast*

lemma *ahm-invar-auxI*:
⟦ $\bigwedge h$. *h* $<$ *array-length a* $\Longrightarrow$ *bucket-ok* (*array-length a*) *h* (*array-get a h*);
$\bigwedge h$. *h* $<$ *array-length a* $\Longrightarrow$ *distinct* (*map fst* (*array-get a h*));
n $=$ *array-foldl* (λ- *n kvs*. *n* $+$ *length kvs*) *0 a*; *array-length a* $>$ *1* ⟧
$\Longrightarrow$ *ahm-invar-aux n a*
unfolding *ahm-invar-aux-def* **by** *blast*

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
proof −
{ fix *h*
assume *h* $<$ *length xs*
with *inv* **have** *bucket-ok* (*length xs*) *h* (*xs* ! *h*)   *distinct* (*map fst* (*xs* ! *h*))
by(*simp-all add*: *ahm-invar-aux-def*) **}**
note *no-junk* $=$ *this*

show *?thesis* **unfolding** *map-concat*
proof(*rule distinct-concat′*)
have *distinct* [*x*←*xs* . *x* $\neq$ []] **unfolding** *distinct-conv-nth*
proof(*intro allI ballI impI*)
fix *i j*
assume *i* $<$ *length* [*x*←*xs* . *x* $\neq$ []]   *j* $<$ *length* [*x*←*xs* . *x* $\neq$ []]   *i* $\neq$ *j*
from *filter-nth-ex-nth*[*OF* ‹*i* $<$ *length* [*x*←*xs* . *x* $\neq$ []]›]
obtain *i′* **where** *i′* $\geq$ *i*   *i′* $<$ *length xs* **and** *ith*: [*x*←*xs* . *x* $\neq$ []] ! *i* $=$ *xs* ! *i′*
and *eqi*: [*x*←*take i′ xs* . *x* $\neq$ []] $=$ *take i* [*x*←*xs* . *x* $\neq$ []] **by** *blast*
from *filter-nth-ex-nth*[*OF* ‹*j* $<$ *length* [*x*←*xs* . *x* $\neq$ []]›]
obtain *j′* **where** *j′* $\geq$ *j*   *j′* $<$ *length xs* **and** *jth*: [*x*←*xs* . *x* $\neq$ []] ! *j* $=$ *xs* ! *j′*
and *eqj*: [*x*←*take j′ xs* . *x* $\neq$ []] $=$ *take j* [*x*←*xs* . *x* $\neq$ []] **by** *blast*
show [*x*←*xs* . *x* $\neq$ []] ! *i* $\neq$ [*x*←*xs* . *x* $\neq$ []] ! *j*
proof
assume [*x*←*xs* . *x* $\neq$ []] ! *i* $=$ [*x*←*xs* . *x* $\neq$ []] ! *j*
hence *eq*: *xs* ! *i′* $=$ *xs* ! *j′* **using** *ith jth* **by** *simp*
from ‹*i* $<$ *length* [*x*←*xs* . *x* $\neq$ []]›
have [*x*←*xs* . *x* $\neq$ []] ! *i* $\in$ *set* [*x*←*xs* . *x* $\neq$ []] **by**(*rule nth-mem*)

with *ith* **have** $xs\ !\ i' \neq []$ **by** *simp*
then obtain *kv* **where** $kv \in set\ (xs\ !\ i')$ **by**(*fastforce simp add*: *neq-Nil-conv*)
with *no-junk*[*OF* ‹$i' < length\ xs$›] **have** *bounded-hashcode* (*length xs*) (*fst kv*) $= i'$
by(*simp add*: *bucket-ok-def*)
moreover from *eq* ‹$kv \in set\ (xs\ !\ i')$› **have** $kv \in set\ (xs\ !\ j')$ **by** *simp*
with *no-junk*[*OF* ‹$j' < length\ xs$›] **have** *bounded-hashcode* (*length xs*) (*fst kv*) $= j'$
by(*simp add*: *bucket-ok-def*)
ultimately have [*simp*]: $i' = j'$ **by** *simp*
from ‹$i < length\ [x \leftarrow xs\ .\ x \neq []]$› **have** $i = length\ (take\ i\ [x \leftarrow xs\ .\ x \neq []])$ **by** *simp*
also from *eqi eqj* **have** $take\ i\ [x \leftarrow xs\ .\ x \neq []] = take\ j\ [x \leftarrow xs\ .\ x \neq []]$ **by** *simp*
finally show *False* **using** ‹$i \neq j$› ‹$j < length\ [x \leftarrow xs\ .\ x \neq []]$› **by** *simp*
qed
qed
moreover have *inj-on* (*map fst*) $\{x \in set\ xs.\ x \neq []\}$
proof(*rule inj-onI*)
fix *x y*
assume $x \in \{x \in set\ xs.\ x \neq []\}$ $\quad y \in \{x \in set\ xs.\ x \neq []\}$ $\quad map\ fst\ x = map\ fst\ y$
hence $x \in set\ xs$ $\quad y \in set\ xs$ $\quad x \neq []$ $\quad y \neq []$ **by** *auto*
from ‹$x \in set\ xs$› **obtain** *i* **where** $xs\ !\ i = x$ $\quad i < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$y \in set\ xs$› **obtain** *j* **where** $xs\ !\ j = y$ $\quad j < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$x \neq []$› **obtain** $k\ v\ x'$ **where** $x = (k,\ v)\ \#\ x'$ **by**(*cases x*) *auto*
with *no-junk*[*OF* ‹$i < length\ xs$›] ‹$xs\ !\ i = x$›
have *bounded-hashcode* (*length xs*) $k = i$ **by**(*auto simp add*: *bucket-ok-def*)
moreover from ‹$map\ fst\ x = map\ fst\ y$› ‹$x = (k,\ v)\ \#\ x'$› **obtain** v' **where** $(k,\ v') \in set\ y$ **by** *fastforce*
with *no-junk*[*OF* ‹$j < length\ xs$›] ‹$xs\ !\ j = y$›
have *bounded-hashcode* (*length xs*) $k = j$ **by**(*auto simp add*: *bucket-ok-def*)
ultimately have $i = j$ **by** *simp*
with ‹$xs\ !\ i = x$› ‹$xs\ !\ j = y$› **show** $x = y$ **by** *simp*
qed
ultimately show *distinct* $[ys \leftarrow map\ (map\ fst)\ xs\ .\ ys \neq []]$
by(*simp add*: *filter-map o-def distinct-map*)
next
fix *ys*
assume $ys \in set\ (map\ (map\ fst)\ xs)$
thus *distinct ys* **by**(*clarsimp simp add*: *set-conv-nth*)(*rule no-junk*)
next
fix *ys zs*
assume $ys \in set\ (map\ (map\ fst)\ xs)$ $\quad zs \in set\ (map\ (map\ fst)\ xs)$ $\quad ys \neq zs$
then obtain $ys'\ zs'$ **where** [*simp*]: $ys = map\ fst\ ys'$ $\quad zs = map\ fst\ zs'$
and $ys' \in set\ xs$ $\quad zs' \in set\ xs$ **by** *auto*
have $fst\ `\ set\ ys' \cap fst\ `\ set\ zs' = \{\}$

proof(*rule equals0I*)
fix *k*
assume $k \in fst$ ‘ *set ys'* $\cap$ *fst* ‘ *set zs'*
then obtain *v v'* **where** $(k, v) \in$ *set ys'* $(k, v') \in$ *set zs'* **by**(*auto*)
from ‹*ys'* $\in$ *set xs*› **obtain** *i* **where** *xs* ! *i* = *ys'* *i* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v) \in$ *set ys'*› **have** *bounded-hashcode* (*length xs*) *k* = *i* **by**(*auto dest*: *no-junk bucket-okD*)
moreover
from ‹*zs'* $\in$ *set xs*› **obtain** *j* **where** *xs* ! *j* = *zs'* *j* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v') \in$ *set zs'*› **have** *bounded-hashcode* (*length xs*) *k* = *j* **by**(*auto dest*: *no-junk bucket-okD*)
ultimately have *i* = *j* **by** *simp*
with ‹*xs* ! *i* = *ys'*› ‹*xs* ! *j* = *zs'*› **have** *ys'* = *zs'* **by** *simp*
with ‹*ys* $\neq$ *zs*› **show** *False* **by** *simp*
qed
thus *set ys* $\cap$ *set zs* = {} **by** *simp*
qed
qed

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 –
have *dom* (*ahm-α-aux a*) $\subseteq$ ($\bigcup$ *h* $\in$ *range* (*bounded-hashcode* (*array-length a*) :: *'a* $\Rightarrow$ *nat*). *dom* (*map-of* (*array-get a h*)))
by(*force simp add*: *dom-map-of-conv-image-fst ahm-α-aux-def dest*: *map-of-SomeD*)
moreover have *finite* ...
proof(*rule finite-UN-I*)
from ‹*ahm-invar-aux n a*› **have** *array-length a* > *1* **by**(*simp add*: *ahm-invar-aux-def*)
hence *range* (*bounded-hashcode* (*array-length a*) :: *'a* $\Rightarrow$ *nat*) $\subseteq$ {*0*..<*array-length a*}
by(*auto simp add*: *bounded-hashcode-bounds*)
thus *finite* (*range* (*bounded-hashcode* (*array-length a*) :: *'a* $\Rightarrow$ *nat*))
by(*rule finite-subset*) *simp*
qed(*rule finite-dom-map-of*)
ultimately show *?thesis* **by**(*rule finite-subset*)
qed

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
fix *k*
show *ahm-α-aux* (*Array xs*) *k* = *map-of* (*concat xs*) *k*
proof(*cases map-of* (*concat xs*) *k*)

case *None*
hence $k \notin$ *fst* ‘ *set* (*concat xs*) **by**(*simp add*: *map-of-eq-None-iff*)
hence $k \notin$ *fst* ‘ *set* (*xs* ! *bounded-hashcode* (*length xs*) *k*)
proof(*rule contrapos-nn*)
assume $k \in$ *fst* ‘ *set* (*xs* ! *bounded-hashcode* (*length xs*) *k*)
then obtain *v* **where** $(k, v) \in$ *set* (*xs* ! *bounded-hashcode* (*length xs*) *k*) **by** *auto*
moreover from *inv* **have** *bounded-hashcode* (*length xs*) $k <$ *length xs*
by(*simp add*: *bounded-hashcode-bounds ahm-invar-aux-def*)
ultimately show $k \in$ *fst* ‘ *set* (*concat xs*)
by(*force intro*: *rev-image-eqI*)
qed
thus *?thesis* **unfolding** *None* **by**(*simp add*: *map-of-eq-None-iff*)
next
case (*Some v*)
hence $(k, v) \in$ *set* (*concat xs*) **by**(*rule map-of-SomeD*)
then obtain *ys* **where** *ys* $\in$ *set xs* $(k, v) \in$ *set ys*
unfolding *set-concat* **by** *blast*
from ‹*ys* $\in$ *set xs*› **obtain** *i j* **where** $i <$ *length xs* *xs* ! $i =$ *ys*
unfolding *set-conv-nth* **by** *auto*
with *inv* ‹$(k, v) \in$ *set ys*›
show *?thesis* **unfolding** *Some*
by(*force dest*: *bucket-okD simp add*: *ahm-invar-aux-def*)
qed
qed

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(*cases a*)
case (*Array xs*)[*simp*]
from *inv* **have** *card* (*dom* (*ahm-α-aux* (*Array xs*))) = *card* (*dom* (*map-of* (*concat xs*)))
by(*simp add*: *ahm-α-aux-conv-map-of-concat*)
also from *inv* **have** *distinct* (*map fst* (*concat xs*))
by(*simp add*: *ahm-invar-distinct-fst-concatD*)
hence *card* (*dom* (*map-of* (*concat xs*))) = *length* (*concat xs*)
by(*rule card-dom-map-of*)
also have *length* (*concat xs*) = *foldl op + 0* (*map length xs*)
by (*simp add*: *length-concat foldl-conv-fold add-commute fold-plus-listsum-rev*)
also from *inv*
have ... $= n$ **unfolding** *foldl-map* **by**(*simp add*: *ahm-invar-aux-def array-foldl-foldl*)
finally show *?thesis* **by**(*simp*)
qed

lemma *finite-dom-ahm-α*:
ahm-invar hm $\Longrightarrow$ *finite* (*dom* (*ahm-α hm*))
by(*cases hm*)(*auto intro*: *finite-dom-ahm-α-aux*)

lemma *finite-map-ahm-α-aux*:
  finite-map ahm-α-aux (*ahm-invar-aux n*)
by(*unfold-locales*)(*rule finite-dom-ahm-α-aux*)

lemma *finite-map-ahm-α*:
  finite-map ahm-α ahm-invar
by(*unfold-locales*)(*rule finite-dom-ahm-α*)

4.11.5 *ahm-empty*

lemma *ahm-invar-aux-new-array*:
  assumes $n > 1$
  shows *ahm-invar-aux 0* (*new-array* [] *n*)
proof −
  have *foldl* (λb (k, v). b + *length* v) *0* (*zip* [*0*..<*n*] (*replicate n* [])) = *0*
    by(*induct n*)(*simp-all add*: *replicate-Suc-conv-snoc del*: *replicate-Suc*)
  with *assms* **show** *?thesis* **by**(*simp add*: *ahm-invar-aux-def array-foldl-new-array*)
qed

lemma *ahm-invar-new-hashmap-with*:
  $n > 1 \Longrightarrow$ *ahm-invar* (*new-hashmap-with n*)
by(*auto simp add*: *ahm-invar-def new-hashmap-with-def intro*: *ahm-invar-aux-new-array*)

lemma *ahm-α-new-hashmap-with*:
  $n > 1 \Longrightarrow$ *ahm-α* (*new-hashmap-with n*) = *empty*
by(*simp add*: *new-hashmap-with-def bounded-hashcode-bounds fun-eq-iff*)

lemma *ahm-invar-ahm-empty* [*simp*]: *ahm-invar* (*ahm-empty* ())
using *def-hashmap-size*[**where** *?'a* = *'a*]
by(*auto intro*: *ahm-invar-new-hashmap-with simp add*: *ahm-empty-def*)

lemma *ahm-empty-correct* [*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
using *def-hashmap-size*[**where** *?'a* = *'a*]
by(*auto intro*: *ahm-α-new-hashmap-with simp add*: *ahm-empty-def*)

lemma *ahm-empty-impl*: *map-empty ahm-α ahm-invar ahm-empty*
by(*unfold-locales*)(*auto*)

4.11.6 *ahm-lookup*

lemma *ahm-lookup-impl*: *map-lookup ahm-α ahm-invar ahm-lookup*
by(*unfold-locales*)(*simp add*: *ahm-lookup-def*)

4.11.7 *ahm-iteratei*

lemma *ahm-iteratei-aux-impl*:
  map-iteratei ahm-α-aux (*ahm-invar-aux n*) *ahm-iteratei-aux*
proof
  fix *m* :: (*'a* × *'b*) *list array*
  assume *invar-m*: *ahm-invar-aux n m*

obtain *ms* **where** *m-eq*[*simp*]: *m* = *Array ms* **by** (*cases m*)

show *finite* (*dom* (*ahm-α-aux m*)) **by** (*metis finite-dom-ahm-α-aux invar-m*)

from *ahm-invar-distinct-fst-concatD*[*of n ms*] *invar-m*
have *dist*: *distinct* (*map fst* (*concat ms*)) **by** *simp*

show *map-iterator* (*ahm-iteratei-aux m*) (*ahm-α-aux m*)
using *set-iterator-foldli-correct*[*of concat ms*] *dist*
by (*simp add*: *ahm-α-aux-conv-map-of-concat*[*OF invar-m*[*unfolded m-eq*]]
ahm-iteratei-aux-def map-to-set-map-of[*OF dist*] *distinct-map*)
qed

lemma *ahm-iteratei-correct*:
map-iteratei ahm-α ahm-invar ahm-iteratei
proof
fix *hm* :: (*'a*, *'b*) *hashmap*
assume *invar-hm*: *ahm-invar hm*
obtain *A n* **where** *hm-eq* [*simp*]: *hm* = *HashMap A n* **by**(*cases hm*)

from *map-iteratei.iteratei-rule* [*OF ahm-iteratei-aux-impl*, *of n A*] *invar-hm*
show *map-it*: *map-iterator* (*ahm-iteratei hm*) (*ahm-α hm*) **by** *simp*

from *map-iterator-finite* [*OF map-it*] **show** *finite* (*dom* (*ahm-α hm*)) **.**
qed

lemma *ahm-iteratei-aux-code* [*code*]:
ahm-iteratei-aux a c f σ = *idx-iteratei array-get array-length a c* (*λx. foldli x c f*) *σ*
proof(*cases a*)
case (*Array xs*)[*simp*]

have *ahm-iteratei-aux a c f σ* = *foldli* (*concat xs*) *c f σ* **by** *simp*
also have ... = *foldli xs c* (*λx. foldli x c f*) *σ* **by** (*simp add*: *foldli-concat*)
also have ... = *idx-iteratei op* ! *length xs c* (*λx. foldli x c f*) *σ* **by** (*simp add*: *idx-iteratei-nth-length-conv-foldli*)
also have ... = *idx-iteratei array-get array-length a c* (*λx. foldli x c f*) *σ*
by(*simp add*: *idx-iteratei-array-get-Array-conv-nth*)
finally show *?thesis* **.**
qed

4.11.8 *ahm-rehash*

lemma *array-length-ahm-rehash-aux'*:
array-length (*ahm-rehash-aux' n kv a*) = *array-length a*
by(*simp add*: *ahm-rehash-aux'-def Let-def*)

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(rule ahm-invar-auxI)
  fix h
  assume h < array-length ?a
  hence hlen: h < array-length a by(simp add: array-length-ahm-rehash-aux')
  with inv have bucket: bucket-ok (array-length a) h (array-get a h)
    and dist: distinct (map fst (array-get a h))
    by(auto elim: ahm-invar-auxE)
  let ?h = bounded-hashcode (array-length a) k
  from hlen bucket show bucket-ok (array-length ?a) h (array-get ?a h)
   by(cases h = ?h)(auto simp add: ahm-rehash-aux'-def Let-def array-length-ahm-rehash-aux'
array-get-array-set-other dest: bucket-okD intro!: bucket-okI)
  from dist hlen fresh
  show distinct (map fst (array-get ?a h))
   by(cases h = ?h)(auto simp add: ahm-rehash-aux'-def Let-def array-get-array-set-other)
next
  let ?f = λn kvs. n + length kvs
  { fix n :: nat and xs :: ('a × 'b) list list
    have foldl ?f n xs = n + foldl ?f 0 xs
      by(induct xs arbitrary:  rule: rev-induct) simp-all }
  note fold = this
  let ?h = bounded-hashcode (array-length a) k

  obtain xs where a [simp]: a = Array xs by(cases a)
  from inv have [simp]: bounded-hashcode (length xs) k < length xs
    by(simp add: ahm-invar-aux-def bounded-hashcode-bounds)
 have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs by(simp add: nth-drop')
  from inv have n = array-foldl (λ- n kvs. n + length kvs) 0 a
    by(auto elim: ahm-invar-auxE)
  hence n = foldl ?f 0 (take ?h xs) + length (xs ! ?h) + foldl ?f 0 (drop (Suc ?h)
xs)
    by(simp add: array-foldl-foldl)(subst xs, simp, subst (1 2 3 4) fold, simp)
  thus Suc n = array-foldl (λ- n kvs. n + length kvs) 0 ?a
   by(simp add: ahm-rehash-aux'-def Let-def array-foldl-foldl foldl-list-update)(subst
(1 2 3 4) fold, simp)
next
  from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
  thus 1 < array-length ?a by(simp add: array-length-ahm-rehash-aux')
qed

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 −
```

interpret *ahm!*: *map-iteratei ahm-α-aux ahm-invar-aux n ahm-iteratei-aux*
by(*rule ahm-iteratei-aux-impl*)
let *?a* = *ahm-rehash-aux a sz*
let *?I* = λ*it a'*. *ahm-invar-aux* (*n* − *card it*) *a'* $\wedge$ *array-length a'* = *sz* $\wedge$ ($\forall$ *k*. *if k* $\in$ *it then ahm-α-aux a' k* = *None else ahm-α-aux a' k* = *ahm-α-aux a k*)
from *inv* **have** *?I* {} *?a* $\vee$ ($\exists$ *it*$\subseteq$*dom* (*ahm-α-aux a*). *it* $\neq$ {} $\wedge$ $\neg$ *True* $\wedge$ *?I it ?a*)
unfolding *ahm-rehash-aux-def*
proof(*rule ahm.iteratei-rule-P*)
from *inv* **have** *card* (*dom* (*ahm-α-aux a*)) = *n* **by**(*rule ahm-invar-aux-card-dom-ahm-α-auxD*)
moreover from ‹*1* < *sz*› **have** *ahm-invar-aux 0* (*new-array* ([] :: (*'key* × *'val*) *list*) *sz*)
by(*rule ahm-invar-aux-new-array*)
moreover {
fix *k*
assume *k* $\notin$ *dom* (*ahm-α-aux a*)
hence *ahm-α-aux a k* = *None* **by** *auto*
moreover have *bounded-hashcode sz k* < *sz* **using** ‹*1* < *sz*›
by(*simp add*: *bounded-hashcode-bounds*)
ultimately have *ahm-α-aux* (*new-array* [] *sz*) *k* = *ahm-α-aux a k* **by** *simp*
}
ultimately show *?I* (*dom* (*ahm-α-aux a*)) (*new-array* [] *sz*)
by(*auto simp add*: *bounded-hashcode-bounds*[*OF* ‹*1* < *sz*›])
next
fix *k* :: *'key*
and *v* :: *'val*
and *it a'*
assume *k* $\in$ *it* *ahm-α-aux a k* = *Some v*
and *it-sub*: *it* $\subseteq$ *dom* (*ahm-α-aux a*)
and *I*: *?I it a'*
from *I* **have** *inv'*: *ahm-invar-aux* (*n* − *card it*) *a'*
and *a'-eq-a*: $\bigwedge$*k*. *k* $\notin$ *it* $\Longrightarrow$ *ahm-α-aux a' k* = *ahm-α-aux a k*
and *a'-None*: $\bigwedge$*k*. *k* $\in$ *it* $\Longrightarrow$ *ahm-α-aux a' k* = *None*
and [*simp*]: *sz* = *array-length a'* **by**(*auto split*: *split-if-asm*)
from *it-sub finite-dom-ahm-α-aux*[*OF inv*] **have** *finite it* **by**(*rule finite-subset*)
moreover with ‹*k* $\in$ *it*› **have** *card it* > *0* **by**(*auto simp add*: *card-gt-0-iff*)
moreover from *finite-dom-ahm-α-aux*[*OF inv*] *it-sub*
have *card it* $\leq$ *card* (*dom* (*ahm-α-aux a*)) **by**(*rule card-mono*)
moreover have ... = *n* **using** *inv*
by(*simp add*: *ahm-invar-aux-card-dom-ahm-α-auxD*)
ultimately have *n* − *card* (*it* − {*k*}) = (*n* − *card it*) + *1* **using** ‹*k* $\in$ *it*› **by** *auto*
moreover from ‹*k* $\in$ *it*› **have** *ahm-α-aux a' k* = *None* **by**(*rule a'-None*)
hence *k* $\notin$ *fst* ‘ *set* (*array-get a'* (*bounded-hashcode* (*array-length a'*) *k*))
by(*simp add*: *map-of-eq-None-iff*)
ultimately have *ahm-invar-aux* (*n* − *card* (*it* − {*k*})) (*ahm-rehash-aux' sz* (*k*, *v*) *a'*)
using *inv'* **by**(*auto intro*: *ahm-rehash-aux'-preserves-ahm-invar-aux*)
moreover have *array-length* (*ahm-rehash-aux' sz* (*k*, *v*) *a'*) = *sz*

by(*simp add*: *array-length-ahm-rehash-aux′*)
moreover {
fix *k′*
assume $k' \in it - \{k\}$
with *bounded-hashcode-bounds*[*OF* ‹*1* < *sz*›, *of k′*] *a′-None*[*of k′*]
have *ahm-α-aux* (*ahm-rehash-aux′ sz* (*k*, *v*) *a′*) *k′* = *None*
by(*cases bounded-hashcode sz k* = *bounded-hashcode sz k′*)(*auto simp add*: *array-get-array-set-other ahm-rehash-aux′-def Let-def*)
} moreover {
fix *k′*
assume $k' \notin it - \{k\}$
with *bounded-hashcode-bounds*[*OF* ‹*1* < *sz*›, *of k′*] *bounded-hashcode-bounds*[*OF* ‹*1* < *sz*›, *of k*] *a′-eq-a*[*of k′*] ‹$k \in it$›
have *ahm-α-aux* (*ahm-rehash-aux′ sz* (*k*, *v*) *a′*) *k′* = *ahm-α-aux a k′*
unfolding *ahm-rehash-aux′-def Let-def* **using** ‹*ahm-α-aux a k* = *Some v*›
by(*cases bounded-hashcode sz k* = *bounded-hashcode sz k′*)(*case-tac* [!] *k′* = *k*, *simp-all add*: *array-get-array-set-other*) **}**
ultimately show *?I* (*it* − {*k*}) (*ahm-rehash-aux′ sz* (*k*, *v*) *a′*) **by** *simp*
qed *auto*
thus *?thesis1 ?thesis2* **unfolding** *ahm-rehash-aux-def*
by(*auto intro*: *ext*)
qed

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*
using *assms*
by −(*case-tac* [!] *hm*, *auto intro*: *ahm-rehash-aux-correct*)

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*)($k \mapsto v$) (**is** *?thesis2*)
proof −
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by**(*cases hm*)

let *?h* = *bounded-hashcode* (*array-length a*) *k*
let *?a′* = *array-set a ?h* (*AList.update k v* (*array-get a ?h*))
let *?n′* = *if map-of* (*array-get a ?h*) *k* = *None then n* + *1 else n*

have *ahm-invar* (*HashMap ?a′ ?n′*) **unfolding** *ahm-invar.simps*
proof(*rule ahm-invar-auxI*)
fix *h*
assume *h* < *array-length ?a′*

```
    hence h < array-length a by simp
    with inv have bucket-ok (array-length a) h (array-get a h)
      by(auto elim: ahm-invar-auxE)
    thus bucket-ok (array-length ?a') h (array-get ?a' h)
      using ‹h < array-length a›
      apply(cases h = bounded-hashcode (array-length a) k)
     apply(fastforce intro!: bucket-okI simp add: dom-update array-get-array-set-other
dest: bucket-okD del: imageE elim: imageE)+
      done
    from ‹h < array-length a› inv have distinct (map fst (array-get a h))
      by(auto elim: ahm-invar-auxE)
    with ‹h < array-length a›
    show distinct (map fst (array-get ?a' h))
     by(cases h = bounded-hashcode (array-length a) k)(auto simp add: array-get-array-set-other
intro: distinct-update)
  next
    obtain xs where a [simp]: a = Array xs by(cases a)

    let ?f = λn kvs. n + length kvs
    { fix n :: nat and xs :: ('a × 'b) list list
      have foldl ?f n xs = n + foldl ?f 0 xs
        by(induct xs arbitrary:  rule: rev-induct) simp-all }
    note fold = this

    from inv have [simp]: bounded-hashcode (length xs) k < length xs
      by(simp add: ahm-invar-aux-def bounded-hashcode-bounds)
   have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs by(simp add: nth-drop')
    from inv have n = array-foldl (λ- n kvs. n + length kvs) 0 a
      by(auto elim: ahm-invar-auxE)
     hence n = foldl ?f 0 (take ?h xs) + length (xs ! ?h) + foldl ?f 0 (drop (Suc
?h) xs)
      by(simp add: array-foldl-foldl)(subst xs, simp, subst (1 2 3 4) fold, simp)
    thus ?n' = array-foldl (λ- n kvs. n + length kvs) 0 ?a'
     apply(simp add: ahm-rehash-aux'-def Let-def array-foldl-foldl foldl-list-update
map-of-eq-None-iff)
      apply(subst (1 2 3 4 5 6 7 8) fold)
      apply(simp add: length-update)
      done
  next
    from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
    thus 1 < array-length ?a' by simp
  qed
  moreover have ahm-α (ahm-update-aux hm k v) = ahm-α hm(k ↦ v)
  proof
    fix k'
    from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
    with bounded-hashcode-bounds[OF this, of k]
    show ahm-α (ahm-update-aux hm k v) k' = (ahm-α hm(k ↦ v)) k'
     by(cases bounded-hashcode (array-length a) k = bounded-hashcode (array-length
```

a) *k'*)(*auto simp add*: *Let-def update-conv array-get-array-set-other*)
  qed
  ultimately show *?thesis1 ?thesis2* **by**(*simp-all add*: *Let-def*)
qed

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*)
using *assms*
by(*simp-all add*: *ahm-update-def Let-def ahm-rehash-correct ahm-update-aux-correct*)

lemma *ahm-update-impl*:
  map-update ahm-α ahm-invar ahm-update
by(*unfold-locales*)(*simp-all add*: *ahm-update-correct*)

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 −
  obtain *a n* **where** *hm* [*simp*]: *hm* = *HashMap a n* **by**(*cases hm*)

  let *?h* = *bounded-hashcode* (*array-length a*) *k*
  let *?a'* = *array-set a ?h* (*AList.delete k* (*array-get a ?h*))
  let *?n'* = *if map-of* (*array-get a* (*bounded-hashcode* (*array-length a*) *k*)) *k* = *None then n else n* − *1*

  have *ahm-invar-aux ?n' ?a'*
  proof(*rule ahm-invar-auxI*)
    fix *h*
    assume *h* < *array-length ?a'*
    hence *h* < *array-length a* **by** *simp*
    with *inv* **have** *bucket-ok* (*array-length a*) *h* (*array-get a h*)
      and *1* < *array-length a*
      and *distinct* (*map fst* (*array-get a h*)) **by**(*auto elim*: *ahm-invar-auxE*)
    thus *bucket-ok* (*array-length ?a'*) *h* (*array-get ?a' h*)
      and *distinct* (*map fst* (*array-get ?a' h*))
      using *bounded-hashcode-bounds*[*of array-length a k*]
       by−(*case-tac* [!] *h* = *bounded-hashcode* (*array-length a*) *k*, *auto simp add*: *array-get-array-set-other set-delete-conv intro*!: *bucket-okI dest*: *bucket-okD intro*: *distinct-delete*)
  next
    obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)

    let *?f* = *λn kvs. n* + *length kvs*
    { **fix** *n* :: *nat* **and** *xs* :: (*'a* × *'b*) *list list*

```
      have foldl ?f n xs = n + foldl ?f 0 xs
        by(induct xs arbitrary: rule: rev-induct) simp-all }
    note fold = this

    from inv have [simp]: bounded-hashcode (length xs) k < length xs
      by(simp add: ahm-invar-aux-def bounded-hashcode-bounds)
   from inv have distinct (map fst (array-get a ?h)) by(auto elim: ahm-invar-auxE)
    moreover
   have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs by(simp add: nth-drop')
    from inv have n = array-foldl (λ- n kvs. n + length kvs) 0 a
      by(auto elim: ahm-invar-auxE)
    hence n = foldl ?f 0 (take ?h xs) + length (xs ! ?h) + foldl ?f 0 (drop (Suc
?h) xs)
      by(simp add: array-foldl-foldl)(subst xs, simp, subst (1 2 3 4) fold, simp)
    ultimately show ?n' = array-foldl (λ- n kvs. n + length kvs) 0 ?a'
      apply(simp add: array-foldl-foldl foldl-list-update map-of-eq-None-iff)
      apply(subst (1 2 3 4 5 6 7 8) fold)
      apply(auto simp add: length-distinct in-set-conv-nth)
      done
  next
    from inv show 1 < array-length ?a' by(auto elim: ahm-invar-auxE)
  qed
  thus ?thesis1 by(auto simp add: Let-def)

  have ahm-α-aux ?a' = ahm-α-aux a |` (− {k})
  proof
    fix k' :: 'a
    from inv have bounded-hashcode (array-length a) k < array-length a
      by(auto elim: ahm-invar-auxE simp add: bounded-hashcode-bounds)
    thus ahm-α-aux ?a' k' = (ahm-α-aux a |` (− {k})) k'
     by(cases ?h = bounded-hashcode (array-length a) k')(auto simp add: restrict-map-def
array-get-array-set-other delete-conv)
  qed
  thus ?thesis2 by(simp add: Let-def)
qed

lemma ahm-delete-impl:
  map-delete ahm-α ahm-invar ahm-delete
by(unfold-locales)(blast intro: ahm-delete-correct)+

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
 interpret *map-empty ArrayHashMap-Impl.ahm-*α *ArrayHashMap-Impl.ahm-invar* *ArrayHashMap-Impl.ahm-empty*
  by(*rule ahm-empty-impl*)
 show *ArrayHashMap-Impl.ahm-empty* () $\in$ *?hashmap*
  by(*simp add*: *empty-correct*)
qed

type-synonym ($'k$,$'v$) *ahm* = ($'k$,$'v$) *hashmap*

lemma *ahm-invar-impl-of* [*simp*, *intro*]: *ArrayHashMap-Impl.ahm-invar* (*impl-of hm*)
using *impl-of*[*of hm*] **by** *simp*

lemma *HashMap-impl-of* [*code abstype*]: *HashMap* (*impl-of t*) = *t*
by(*rule impl-of-inverse*)

4.12.2 Primitive operations

definition *ahm-empty-const* :: ($'key$ :: *hashable*, $'val$) *hashmap*
where *ahm-empty-const* $\equiv$ (*HashMap* (*ArrayHashMap-Impl.ahm-empty* ()))

definition *ahm-empty* :: *unit* $\Rightarrow$ ($'key$ :: *hashable*, $'val$) *hashmap*
where *ahm-empty* $\equiv$ λ-. *ahm-empty-const*

definition *ahm-*α :: ($'key$ :: *hashable*, $'val$) *hashmap* $\Rightarrow$ $'key$ $\Rightarrow$ $'val$ *option*
where *ahm-*α *hm* = *ArrayHashMap-Impl.ahm-*α (*impl-of hm*)

definition *ahm-lookup* :: $'key$ $\Rightarrow$ ($'key$ :: *hashable*, $'val$) *hashmap* $\Rightarrow$ $'val$ *option*
where *ahm-lookup k hm* = *ArrayHashMap-Impl.ahm-lookup k* (*impl-of hm*)

definition *ahm-iteratei* :: ($'key$ :: *hashable*, $'val$) *hashmap* $\Rightarrow$ ($'key$ $\times$ $'val$, $'\sigma$) *set-iterator*
where *ahm-iteratei hm* = *ArrayHashMap-Impl.ahm-iteratei* (*impl-of hm*)

definition *ahm-update* :: $'key$ $\Rightarrow$ $'val$ $\Rightarrow$ ($'key$ :: *hashable*, $'val$) *hashmap* $\Rightarrow$ ($'key$,

'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 ()
by(*simp add: ahm-empty-const-def HashMap-inverse*)

lemma *impl-of-ahm-update [code abstract]:*
impl-of (ahm-update k v hm) = ArrayHashMap-Impl.ahm-update k v (impl-of hm)
by(*simp add: ahm-update-def HashMap-inverse ahm-update-correct*)

lemma *impl-of-ahm-delete [code abstract]:*
impl-of (ahm-delete k hm) = ArrayHashMap-Impl.ahm-delete k (impl-of hm)
by(*simp add: ahm-delete-def HashMap-inverse ahm-delete-correct*)

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*))
by(*simp add*: *ahm-*α*-def finite-dom-ahm-*α)

lemma *finite-map-ahm-*α:
finite-map ahm-α *ahm-invar*
by(*unfold-locales*)(*rule finite-dom-ahm-*α)

interpretation *ahm*!: *finite-map ahm-*α *ahm-invar*
by(*rule finite-map-ahm-*α)

lemma *ahm-empty-correct* [*simp*]: *ahm-*α (*ahm-empty* ()) = *Map.empty*
by(*simp add*: *ahm-*α*-def ahm-empty-def ahm-empty-const-def HashMap-inverse*)

lemma *ahm-empty-impl*: *map-empty ahm-*α *ahm-invar ahm-empty*
by *unfold-locales simp-all*

interpretation *ahm*!: *map-empty ahm-*α *ahm-invar ahm-empty* **by**(*rule ahm-empty-impl*)

lemma *ahm-lookup-impl*: *map-lookup ahm-*α *ahm-invar ahm-lookup*
by(*unfold-locales*)(*simp add*: *ahm-lookup-def ArrayHashMap-Impl.ahm-lookup-def ahm-*α*-def*)

interpretation *ahm*!: *map-lookup ahm-*α *ahm-invar ahm-lookup* **by**(*rule ahm-lookup-impl*)

lemma *ahm-iteratei-impl*:
map-iteratei ahm-α *ahm-invar ahm-iteratei*
proof
fix *m* :: (′*a*, ′*b*) *hashmap*

from *map-iteratei.iteratei-rule* [*OF ahm-iteratei-correct*, *of impl-of m*]
show *map-iterator* (*ahm-iteratei m*) (*ahm-*α *m*)
unfolding *ahm-iteratei-def ahm-*α*-def*
by *simp*
qed

interpretation *ahm*!: *map-iteratei ahm-*α *ahm-invar ahm-iteratei*

by(*rule ahm-iteratei-impl*)

lemma *ahm-update-correct*: *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)($k \mapsto v$)
by(*simp add*: *ahm-α-def ahm-update-def ahm-update-correct HashMap-inverse*)

lemma *ahm-update-impl*:
map-update ahm-α ahm-invar ahm-update
by(*unfold-locales*)(*simp-all add*: *ahm-update-correct*)

interpretation *ahm*!: *map-update ahm-α ahm-invar ahm-update* **by**(*rule ahm-update-impl*)

lemma *ahm-delete-correct*:
ahm-α (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*})
by(*simp add*: *ahm-α-def ahm-delete-def HashMap-inverse ahm-delete-correct*)

lemma *ahm-delete-impl*:
map-delete ahm-α ahm-invar ahm-delete
by(*unfold-locales*)(*blast intro*: *ahm-delete-correct*)+

interpretation *ahm*!: *map-delete ahm-α ahm-invar ahm-delete* **by**(*rule ahm-delete-impl*)

lemma *ahm-update-dj-impl*: *map-update-dj ahm-α ahm-invar ahm-update-dj*
unfolding *ahm-update-dj-def*
by(*rule map-update.update-dj-by-update*)(*rule ahm-update-impl*)

lemma *ahm-add-impl*: *map-add ahm-α ahm-invar ahm-add*
unfolding *ahm-add-def* **by**(*rule it-add-correct*)(*rule ahm-iteratei-impl ahm-update-impl*)+

lemma *ahm-add-dj-impl*: *map-add-dj ahm-α ahm-invar ahm-add-dj*
unfolding *ahm-add-dj-def* **by**(*rule map-add.add-dj-by-add*)(*rule ahm-add-impl*)

lemma *ahm-isEmpty-impl*: *map-isEmpty ahm-α ahm-invar ahm-isEmpty*
unfolding *ahm-isEmpty-def* **by**(*rule iti-isEmpty-correct*)(*rule ahm-iteratei-impl*)

lemma *ahm-sel-impl*: *map-sel ahm-α ahm-invar ahm-sel*
unfolding *ahm-sel-def* **by**(*rule iti-sel-correct*)(*rule ahm-iteratei-impl*)

lemma *ahm-sel′-impl*: *map-sel′ ahm-α ahm-invar ahm-sel′*
unfolding *ahm-sel′-def* **by** (*rule iti-sel′-correct*) (*rule ahm-iteratei-impl*)

lemma *ahm-to-list-impl*: *map-to-list ahm-α ahm-invar ahm-to-list*
unfolding *ahm-to-list-def* **by**(*rule it-map-to-list-correct*)(*rule ahm-iteratei-impl*)

lemma *list-to-ahm-impl*: *list-to-map ahm-α ahm-invar list-to-ahm*
unfolding *list-to-ahm-def* **by**(*rule gen-list-to-map-correct*)(*rule ahm-empty-impl ahm-update-impl*)+

interpretation *ahm*!: *map-update-dj ahm-α ahm-invar ahm-update-dj*
using *ahm-update-dj-impl* **.**

interpretation *ahm*!: *map-add ahm-*α *ahm-invar ahm-add*
  using *ahm-add-impl* **.**
interpretation *ahm*!: *map-add-dj ahm-*α *ahm-invar ahm-add-dj*
  using *ahm-add-dj-impl* **.**
interpretation *ahm*!: *map-isEmpty ahm-*α *ahm-invar ahm-isEmpty*
  using *ahm-isEmpty-impl* **.**
interpretation *ahm*!: *map-sel ahm-*α *ahm-invar ahm-sel*
  using *ahm-sel-impl* **.**
interpretation *ahm*!: *map-sel′ ahm-*α *ahm-invar ahm-sel′*
  using *ahm-sel′-impl* **.**
interpretation *ahm*!: *map-to-list ahm-*α *ahm-invar ahm-to-list*
  using *ahm-to-list-impl* **.**
interpretation *ahm*!: *list-to-map ahm-*α *ahm-invar list-to-ahm*
  using *list-to-ahm-impl* **.**

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 $iam\text{-}\alpha :: 'v\ iam \Rightarrow nat \rightharpoonup 'v$ **where**
$iam\text{-}\alpha\ a\ i \equiv if\ i < array\text{-}length\ a\ then\ array\text{-}get\ a\ i\ else\ None$

abbreviation $iam\text{-}invar :: 'v\ iam \Rightarrow bool$ **where** $iam\text{-}invar \equiv \lambda\text{-}.\ True$

definition $iam\text{-}empty :: unit \Rightarrow 'v\ iam$
where $iam\text{-}empty \equiv \lambda\text{-}::unit.\ array\text{-}of\text{-}list\ []$

definition $iam\text{-}lookup :: nat \Rightarrow 'v\ iam \rightharpoonup 'v$
where $iam\text{-}lookup\ k\ a \equiv iam\text{-}\alpha\ a\ k$

definition $iam\text{-}increment\ (l::nat)\ idx \equiv$
$max\ (idx + 1 - l)\ (2 * l + 3)$

lemma *incr-correct*: $\neg\ idx < l \Longrightarrow idx < l + iam\text{-}increment\ l\ idx$
unfolding *iam-increment-def* **by** *auto*

definition $iam\text{-}update :: nat \Rightarrow 'v \Rightarrow 'v\ iam \Rightarrow 'v\ iam$
where $iam\text{-}update\ k\ v\ a \equiv let$
$l = array\text{-}length\ a;$
$a = if\ k < l\ then\ a\ else\ array\text{-}grow\ a\ (iam\text{-}increment\ l\ k)\ None$
in
$array\text{-}set\ a\ k\ (Some\ v)$

definition $iam\text{-}update\text{-}dj \equiv iam\text{-}update$

definition $iam\text{-}delete :: nat \Rightarrow 'v\ iam \Rightarrow 'v\ iam$
where $iam\text{-}delete\ k\ a \equiv$
$if\ k<array\text{-}length\ a\ then\ array\text{-}set\ a\ k\ None\ else\ a$

fun *iam-iteratei-aux*
$:: nat \Rightarrow ('v\ iam) \Rightarrow ('\sigma \Rightarrow bool) \Rightarrow (nat \times 'v \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow '\sigma \Rightarrow '\sigma$
where

iam-iteratei-aux 0 a c f $\sigma = \sigma$
| *iam-iteratei-aux i a c f* $\sigma =$ (
if c σ *then*
iam-iteratei-aux $(i - 1)$ *a c f* (
case array-get a $(i - 1)$ *of None* $\Rightarrow \sigma$ | *Some x* $\Rightarrow$ *f* $(i - 1, x)$ σ
)
else σ)

definition *iam-iteratei* :: *'v iam* $\Rightarrow$ (*nat* $\times$ *'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*
apply *unfold-locales*
unfolding *iam-*α*-def*[*abs-def*] *iam-empty-def*
by *auto*

interpretation *iam*!: *map-empty iam-*α *iam-invar iam-empty*
using *iam-empty-impl* **.**

lemma *iam-lookup-impl*: *map-lookup iam-*α *iam-invar iam-lookup*
apply *unfold-locales*

unfolding *iam-α-def*[*abs-def*] *iam-lookup-def*
by *auto*
interpretation *iam*!: *map-lookup iam-α iam-invar iam-lookup*
using *iam-lookup-impl* **.**

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*)
by (*auto simp*: *array-get-array-set-other*)

lemma *iam-update-impl*: *map-update iam-α iam-invar iam-update*
apply *unfold-locales*
unfolding *iam-α-def*[*abs-def*] *iam-update-def*
apply (*rule ext*)
apply (*auto simp*: *Let-def array-get-set-iff incr-correct*)
done
interpretation *iam*!: *map-update iam-α iam-invar iam-update*
using *iam-update-impl* **.**

lemma *iam-update-dj-impl*: *map-update-dj iam-α iam-invar iam-update-dj*
apply (*unfold iam-update-dj-def*) **by** (*rule iam.update-dj-by-update*)
interpretation *iam*!: *map-update-dj iam-α iam-invar iam-update-dj*
using *iam-update-dj-impl* **.**

lemma *iam-delete-impl*: *map-delete iam-α iam-invar iam-delete*
apply *unfold-locales*
unfolding *iam-α-def*[*abs-def*] *iam-delete-def*
apply (*rule ext*)
apply (*auto simp*: *Let-def array-get-set-iff*)
done
interpretation *iam*!: *map-delete iam-α iam-invar iam-delete*
using *iam-delete-impl* **.**

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*]))
by (*induct n*) (*auto simp add*: *List.map-filter-def fun-eq-iff*)

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*)]))
unfolding *iam-iteratei-def iam-iteratei-aux-foldli-conv* **by** *simp*

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 (*rule map-iterator-rev-linord-I* [*of kvs*])
  show *iam-iteratei m* = *foldli kvs*
    unfolding *iam-iteratei-foldli-conv kvs-def* **by** *simp*
next
  def *al* ≡ *array-length m*
  show *dist-kvs*: *distinct* (*map fst kvs*) **and** *sorted* (*rev* (*map fst kvs*))
    unfolding *kvs-def al-def*[*symmetric*]
    apply (*induct al*)
  apply (*simp-all add*: *List.map-filter-simps set-map-filter image-iff sorted-append*
                 split: *option.split*)
  done

  from *dist-kvs*
  have $\bigwedge i$. *map-of kvs i* = *iam-α m i*
    unfolding *kvs-def*
    apply (*case-tac array-get m i*)
    apply (*simp-all add*: *iam-α-def map-of-eq-None-iff set-map-filter image-iff*)
  done
  thus *iam-α m* = *map-of kvs* **by** *auto*
qed

lemma *iam-iteratei-rev-linord-impl*: *map-reverse-iterateoi iam-α iam-invar iam-iteratei*
  apply *unfold-locales*
  apply (*simp add*: *iam-α-def*[*abs-def*] *dom-def*)
  apply (*simp add*: *iam-iteratei-correct*)
done

interpretation *iam*!: *map-reverse-iterateoi iam-α iam-invar iam-iteratei*
  using *iam-iteratei-rev-linord-impl* **.**

lemma *iam-iteratei-impl*: *map-iteratei iam-α iam-invar iam-iteratei*
  using *MapGA.iti-by-ritoi* [*OF iam-iteratei-rev-linord-impl*] **.**
interpretation *iam*!: *map-iteratei iam-α iam-invar iam-iteratei*
  using *iam-iteratei-impl* **.**

lemma *iam-add-impl*: *map-add iam-α iam-invar iam-add*
  unfolding *iam-add-def*
  using *it-add-correct*[*OF iam-iteratei-impl iam-update-impl*] **.**
interpretation *iam*!: *map-add iam-α iam-invar iam-add* **using** *iam-add-impl* **.**

lemma *iam-add-dj-impl*: *map-add-dj iam-α iam-invar iam-add-dj*
  unfolding *iam-add-dj-def* **by** (*rule iam.add-dj-by-add*)
interpretation *iam*!: *map-add-dj iam-α iam-invar iam-add-dj*
  using *iam-add-dj-impl* **.**

lemma *iam-isEmpty-impl*: *map-isEmpty iam-α iam-invar iam-isEmpty*
  unfolding *iam-isEmpty-def*
  using *iti-isEmpty-correct*[*OF iam-iteratei-impl*] **.**
interpretation *iam*!: *map-isEmpty iam-α iam-invar iam-isEmpty*

using *iam-isEmpty-impl* .

lemma *iam-sel-impl*: *map-sel iam-α iam-invar iam-sel*
unfolding *iam-sel-def*
using *iti-sel-correct*[*OF iam-iteratei-impl*] .
interpretation *iam*!: *map-sel iam-α iam-invar iam-sel*
using *iam-sel-impl* .

lemma *iam-sel′-impl*: *map-sel′ iam-α iam-invar iam-sel′*
unfolding *iam-sel′-def*
using *iti-sel′-correct*[*OF iam-iteratei-impl*] .
interpretation *iam*!: *map-sel′ iam-α iam-invar iam-sel′*
using *iam-sel′-impl* .

lemma *iam-to-list-impl*: *map-to-list iam-α iam-invar iam-to-list*
unfolding *iam-to-list-def*
using *it-map-to-list-correct*[*OF iam-iteratei-impl*] .
interpretation *iam*!: *map-to-list iam-α iam-invar iam-to-list*
using *iam-to-list-impl* .

lemma *list-to-iam-impl*: *list-to-map iam-α iam-invar list-to-iam*
unfolding *list-to-iam-def*
using *gen-list-to-map-correct*[*OF iam-empty-impl iam-update-impl*] .
interpretation *iam*!: *list-to-map iam-α iam-invar list-to-iam*
using *list-to-iam-impl* .

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 :: $'a\ ls \Rightarrow 'a\ set$ **where** *ls-*α l == *set* (*list-of-dlist* l)
abbreviation (*input*) *ls-invar* :: $'a\ ls \Rightarrow bool$ **where** *ls-invar* == λ-. *True*
definition *ls-empty* :: $unit \Rightarrow 'a\ ls$ **where** *ls-empty* == (λ-. *Dlist.empty*)
definition *ls-memb* :: $'a \Rightarrow 'a\ ls \Rightarrow bool$ **where** *ls-memb* $x\ l$ == *Dlist.member* $l\ x$
definition *ls-ins* :: $'a \Rightarrow 'a\ ls \Rightarrow 'a\ ls$ **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* :: $'a \Rightarrow 'a\ ls \Rightarrow 'a\ ls$ **where** *ls-ins-dj* == *Dlist.insert*
definition *ls-delete* :: $'a \Rightarrow 'a\ ls \Rightarrow 'a\ ls$
where *ls-delete* == *dlist-remove'*
definition *ls-iteratei* :: $'a\ ls \Rightarrow ('a,'\sigma)$ *set-iterator*
where *ls-iteratei* == *dlist-iteratei*

definition *ls-isEmpty* :: $'a$ *ls* $\Rightarrow$ *bool* **where** *ls-isEmpty* == *Dlist.null*

definition *ls-union* :: $'a$ *ls* $\Rightarrow$ $'a$ *ls* $\Rightarrow$ $'a$ *ls*
where *ls-union* == *it-union ls-iteratei ls-ins*
definition *ls-inter* :: $'a$ *ls* $\Rightarrow$ $'a$ *ls* $\Rightarrow$ $'a$ *ls*
where *ls-inter* == *it-inter ls-iteratei ls-memb ls-empty ls-ins-dj*
definition *ls-union-dj* :: $'a$ *ls* $\Rightarrow$ $'a$ *ls* $\Rightarrow$ $'a$ *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* $\Rightarrow$ $('a \Rightarrow 'r$ *option*$)$ $\Rightarrow$ $'r$ *option*
where *ls-sel* == *iti-sel ls-iteratei*
definition *ls-sel'* == *iti-sel-no-map ls-iteratei*

definition *ls-to-list* :: $'a$ *ls* $\Rightarrow$ $'a$ *list* **where** *ls-to-list* == *list-of-dlist*
definition *list-to-ls* :: $'a$ *list* $\Rightarrow$ $'a$ *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*
by(*unfold-locales*)(*auto simp add*: *ls-defs dlist-member-empty*)

lemma *ls-memb-impl*: *set-memb ls-α ls-invar ls-memb*
apply (*unfold-locales*)
apply (*auto simp add*: *ls-defs Dlist.member-def List.member-def*)
done

lemma *ls-ins-impl*: *set-ins ls-α ls-invar ls-ins*
by(*unfold-locales*)(*auto simp add*: *ls-defs*)

lemma *ls-ins-dj-impl*: *set-ins-dj ls-α ls-invar ls-ins-dj*
by(*unfold-locales*)(*auto simp add*: *ls-defs*)

lemma *ls-delete-impl*: *set-delete ls-α ls-invar ls-delete*
apply (*unfold-locales*)
apply (*auto simp add*: *ls-defs dlist-remove′-correct set-dlist-remove1′*
split: *split-if-asm*)
done

lemma *ls-α-finite*[*simp*, *intro*!]: *finite* (*ls-α l*)
by (*auto simp add*: *ls-defs*)

lemma *ls-is-finite-set*: *finite-set ls-α ls-invar*
by(*unfold-locales*)(*auto simp add*: *ls-defs*)

lemma *ls-iteratei-impl*: *set-iteratei ls-α ls-invar ls-iteratei*
by(*unfold-locales*)(*unfold ls-defs*, *simp*, *rule dlist-iteratei-correct*)

lemma *ls-isEmpty-impl*: *set-isEmpty ls-α ls-invar ls-isEmpty*
by (*unfold-locales*) (*auto simp add*: *ls-defs null-def member-def List.null-def List.member-def*)

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*
by(*unfold-locales*)(*auto simp add*: *ls-defs Dlist.member-def List.member-def*)

lemma *list-to-ls-impl*: *list-to-set ls-α ls-invar list-to-ls*
by(*unfold-locales*)(*auto simp add*: *ls-defs Dlist.member-def List.member-def*)

interpretation *ls*: *set-empty ls-α ls-invar ls-empty* **using** *ls-empty-impl* .
interpretation *ls*: *set-memb ls-α ls-invar ls-memb* **using** *ls-memb-impl* .
interpretation *ls*: *set-ins ls-α ls-invar ls-ins* **using** *ls-ins-impl* .
interpretation *ls*: *set-ins-dj ls-α ls-invar ls-ins-dj* **using** *ls-ins-dj-impl* .
interpretation *ls*: *set-delete ls-α ls-invar ls-delete* **using** *ls-delete-impl* .
interpretation *ls*: *finite-set ls-α ls-invar* **using** *ls-is-finite-set* .
interpretation *ls*: *set-iteratei ls-α ls-invar ls-iteratei* **using** *ls-iteratei-impl* .
interpretation *ls*: *set-isEmpty ls-α ls-invar ls-isEmpty* **using** *ls-isEmpty-impl* .
interpretation *ls*: *set-union ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-union* **using** *ls-union-impl* .
interpretation *ls*: *set-inter ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-inter* **using** *ls-inter-impl* .
interpretation *ls*: *set-union-dj ls-α ls-invar ls-α ls-invar ls-α ls-invar ls-union-dj* **using** *ls-union-dj-impl* .
interpretation *ls*: *set-image-filter ls-α ls-invar ls-α ls-invar ls-image-filter* **using** *ls-image-filter-impl* .
interpretation *ls*: *set-inj-image-filter ls-α ls-invar ls-α ls-invar ls-inj-image-filter* **using** *ls-inj-image-filter-impl* .
interpretation *ls*: *set-image ls-α ls-invar ls-α ls-invar ls-image* **using** *ls-image-impl* .
interpretation *ls*: *set-inj-image ls-α ls-invar ls-α ls-invar ls-inj-image* **using** *ls-inj-image-impl* .
interpretation *ls*: *set-sel ls-α ls-invar ls-sel* **using** *ls-sel-impl* .
interpretation *ls*: *set-sel′ ls-α ls-invar ls-sel′* **using** *ls-sel′-impl* .
interpretation *ls*: *set-to-list ls-α ls-invar ls-to-list* **using** *ls-to-list-impl* .
interpretation *ls*: *list-to-set ls-α ls-invar list-to-ls* **using** *list-to-ls-impl* .

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*
by(*simp add*: *list-to-ls-def*)

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*

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*
by (*unfold-locales*) (*auto simp add*: *lsi-defs*)

lemma *lsi-memb-impl*: *set-memb lsi-α lsi-invar lsi-memb*
by (*unfold-locales*)(*auto simp add*: *lsi-defs in-set-member*)

lemma *lsi-ins-impl*: *set-ins lsi-α lsi-invar lsi-ins*
by (*unfold-locales*) (*auto simp add*: *lsi-defs in-set-member*)

lemma *lsi-ins-dj-impl*: *set-ins-dj lsi-α lsi-invar lsi-ins-dj*
by (*unfold-locales*) (*auto simp add*: *lsi-defs*)

lemma *lsi-delete-impl*: *set-delete lsi-α lsi-invar lsi-delete*
by(*unfold-locales*)(*simp-all add*: *lsi-defs set-dlist-remove1'*
distinct-remove1')

lemma *lsi-α-finite*[*simp*, *intro*!]: *finite* (*lsi-α l*)
by (*auto simp add*: *lsi-defs*)

lemma *lsi-is-finite-set*: *finite-set lsi-α lsi-invar*
by (*unfold-locales*) (*auto simp add*: *lsi-defs*)

lemma *lsi-iteratei-impl*: *set-iteratei lsi-α lsi-invar lsi-iteratei*
by(*unfold-locales*)(*simp*, *unfold lsi-invar-def lsi-α-def lsi-iteratei-def*, *rule set-iterator-foldli-correct*)

lemma *lsi-isEmpty-impl*: *set-isEmpty lsi-α lsi-invar lsi-isEmpty*
by(*unfold-locales*)(*auto simp add*: *lsi-defs*)

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*
by(*unfold-locales*)(*auto simp add*: *lsi-defs*)

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*
by(*unfold-locales*)(*auto simp add*: *lsi-defs*)

lemma *list-to-lsi-impl*: *list-to-set lsi-α lsi-invar list-to-lsi*
by(*unfold-locales*)(*auto simp add*: *lsi-defs*)

interpretation *lsi*: *set-empty lsi-α lsi-invar lsi-empty* **using** *lsi-empty-impl* .
interpretation *lsi*: *set-memb lsi-α lsi-invar lsi-memb* **using** *lsi-memb-impl* .
interpretation *lsi*: *set-ins lsi-α lsi-invar lsi-ins* **using** *lsi-ins-impl* .
interpretation *lsi*: *set-ins-dj lsi-α lsi-invar lsi-ins-dj* **using** *lsi-ins-dj-impl* .
interpretation *lsi*: *set-delete lsi-α lsi-invar lsi-delete* **using** *lsi-delete-impl* .
interpretation *lsi*: *finite-set lsi-α lsi-invar* **using** *lsi-is-finite-set* .
interpretation *lsi*: *set-iteratei lsi-α lsi-invar lsi-iteratei* **using** *lsi-iteratei-impl* .
interpretation *lsi*: *set-isEmpty lsi-α lsi-invar lsi-isEmpty* **using** *lsi-isEmpty-impl* .
interpretation *lsi*: *set-union lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-union* **using** *lsi-union-impl* .
interpretation *lsi*: *set-inter lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-inter* **using** *lsi-inter-impl* .
interpretation *lsi*: *set-union-dj lsi-α lsi-invar lsi-α lsi-invar lsi-α lsi-invar lsi-union-dj* **using** *lsi-union-dj-impl* .
interpretation *lsi*: *set-image-filter lsi-α lsi-invar lsi-α lsi-invar lsi-image-filter* **using** *lsi-image-filter-impl* .
interpretation *lsi*: *set-inj-image-filter lsi-α lsi-invar lsi-α lsi-invar lsi-inj-image-filter* **using** *lsi-inj-image-filter-impl* .
interpretation *lsi*: *set-image lsi-α lsi-invar lsi-α lsi-invar lsi-image* **using** *lsi-image-impl* .

interpretation *lsi*: *set-inj-image lsi-α lsi-invar lsi-α lsi-invar lsi-inj-image* **using** *lsi-inj-image-impl* **.**
interpretation *lsi*: *set-sel lsi-α lsi-invar lsi-sel* **using** *lsi-sel-impl* **.**
interpretation *lsi*: *set-sel′ lsi-α lsi-invar lsi-sel′* **using** *lsi-sel′-impl* **.**
interpretation *lsi*: *set-to-list lsi-α lsi-invar lsi-to-list* **using** *lsi-to-list-impl* **.**
interpretation *lsi*: *list-to-set lsi-α lsi-invar list-to-lsi* **using** *list-to-lsi-impl* **.**

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* :: $'a{::}linorder\ rs \Rightarrow 'a\ set$ **where** *rs-α* == *s-α rm-α*
abbreviation (*input*) *rs-invar* :: $'a{::}linorder\ rs \Rightarrow bool$ **where** *rs-invar* == *rm-invar*
definition *rs-empty* :: $unit \Rightarrow 'a{::}linorder\ rs$ **where** *rs-empty* == *s-empty rm-empty*
definition *rs-memb* :: $'a{::}linorder \Rightarrow 'a\ rs \Rightarrow bool$
where *rs-memb* == *s-memb rm-lookup*
definition *rs-ins* :: $'a{::}linorder \Rightarrow 'a\ rs \Rightarrow 'a\ rs$
where *rs-ins* == *s-ins rm-update*
definition *rs-ins-dj* :: $'a{::}linorder \Rightarrow 'a\ rs \Rightarrow 'a\ rs$ **where**
rs-ins-dj == *s-ins-dj rm-update-dj*
definition *rs-delete* :: $'a{::}linorder \Rightarrow 'a\ rs \Rightarrow 'a\ rs$
where *rs-delete* == *s-delete rm-delete*
definition *rs-sel* :: $'a{::}linorder\ rs \Rightarrow ('a \Rightarrow 'r\ option) \Rightarrow 'r\ option$
where *rs-sel* == *s-sel rm-sel*
definition *rs-sel'* = *sel-sel' rs-sel*
definition *rs-isEmpty* :: $'a{::}linorder\ rs \Rightarrow bool$
where *rs-isEmpty* == *s-isEmpty rm-isEmpty*

definition *rs-iteratei* :: $'a{::}linorder\ rs \Rightarrow ('a,'\sigma)\ set\text{-}iterator$
where *rs-iteratei* == *s-iteratei rm-iteratei*

definition *rs-iterateoi* :: $'a{::}linorder\ rs \Rightarrow ('a,'\sigma)\ set\text{-}iterator$
where *rs-iterateoi* == *s-iteratei rm-iterateoi*

definition *rs-reverse-iterateoi* :: $'a{::}linorder\ rs \Rightarrow ('a,'\sigma)\ set\text{-}iterator$
where *rs-reverse-iterateoi* == *s-iteratei rm-reverse-iterateoi*

definition *rs-union* :: $'a{::}linorder\ rs \Rightarrow 'a\ rs \Rightarrow 'a\ rs$
where *rs-union* == *s-union rm-add*
definition *rs-union-dj* :: $'a{::}linorder\ rs \Rightarrow 'a\ rs \Rightarrow 'a\ rs$

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* **using** *rs-iteratei-impl* **.**

interpretation *rs*: *set-iterateoi rs-α rs-invar rs-iterateoi* **using** *rs-iterateoi-impl* **.**

interpretation *rs*: *set-reverse-iterateoi rs-α rs-invar rs-reverse-iterateoi* **using** *rs-reverse-iterateoi-impl* **.**

interpretation *rs*: *set-inj-image-filter rs-α rs-invar rs-α rs-invar rs-inj-image-filter* **using** *rs-inj-image-filter-impl* **.**
interpretation *rs*: *set-image-filter rs-α rs-invar rs-α rs-invar rs-image-filter* **using** *rs-image-filter-impl* **.**
interpretation *rs*: *set-inj-image rs-α rs-invar rs-α rs-invar rs-inj-image* **using** *rs-inj-image-impl* **.**
interpretation *rs*: *set-union-dj rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-union-dj* **using** *rs-union-dj-impl* **.**
interpretation *rs*: *set-union rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-union* **using** *rs-union-impl* **.**
interpretation *rs*: *set-inter rs-α rs-invar rs-α rs-invar rs-α rs-invar rs-inter* **using** *rs-inter-impl* **.**
interpretation *rs*: *set-image rs-α rs-invar rs-α rs-invar rs-image* **using** *rs-image-impl* **.**

interpretation *rs*: *set-sel rs-*α *rs-invar rs-sel* **using** *rs-sel-impl* **.**
interpretation *rs*: *set-sel'* *rs-*α *rs-invar rs-sel'* **using** *rs-sel'-impl* **.**
interpretation *rs*: *set-ins-dj rs-*α *rs-invar rs-ins-dj* **using** *rs-ins-dj-impl* **.**
interpretation *rs*: *set-delete rs-*α *rs-invar rs-delete* **using** *rs-delete-impl* **.**
interpretation *rs*: *set-ins rs-*α *rs-invar rs-ins* **using** *rs-ins-impl* **.**
interpretation *rs*: *set-memb rs-*α *rs-invar rs-memb* **using** *rs-memb-impl* **.**
interpretation *rs*: *set-to-sorted-list rs-*α *rs-invar rs-to-list* **using** *rs-to-list-impl* **.**
interpretation *rs*: *list-to-set rs-*α *rs-invar list-to-rs* **using** *list-to-rs-impl* **.**
interpretation *rs*: *set-isEmpty rs-*α *rs-invar rs-isEmpty* **using** *rs-isEmpty-impl* **.**
interpretation *rs*: *set-empty rs-*α *rs-invar rs-empty* **using** *rs-empty-impl* **.**
interpretation *rs*: *set-min rs-*α *rs-invar rs-min* **using** *rs-min-impl* **.**
interpretation *rs*: *set-max rs-*α *rs-invar rs-max* **using** *rs-max-impl* **.**

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 $hs\text{-}\alpha :: 'a::hashable\ hs \Rightarrow 'a\ set$ **where** $hs\text{-}\alpha == s\text{-}\alpha\ hm\text{-}\alpha$
abbreviation $(input)\ hs\text{-}invar :: 'a::hashable\ hs \Rightarrow bool$ **where** $hs\text{-}invar == \lambda\text{-}.$ *True*
definition $hs\text{-}empty :: unit \Rightarrow 'a::hashable\ hs$ **where** $hs\text{-}empty == s\text{-}empty\ hm\text{-}empty$
definition $hs\text{-}memb :: 'a::hashable \Rightarrow 'a\ hs \Rightarrow bool$
where $hs\text{-}memb == s\text{-}memb\ hm\text{-}lookup$
definition $hs\text{-}ins :: 'a::hashable \Rightarrow 'a\ hs \Rightarrow 'a\ hs$
where $hs\text{-}ins == s\text{-}ins\ hm\text{-}update$
definition $hs\text{-}ins\text{-}dj :: 'a::hashable \Rightarrow 'a\ hs \Rightarrow 'a\ hs$ **where**
$hs\text{-}ins\text{-}dj == s\text{-}ins\text{-}dj\ hm\text{-}update\text{-}dj$
definition $hs\text{-}delete :: 'a::hashable \Rightarrow 'a\ hs \Rightarrow 'a\ hs$
where $hs\text{-}delete == s\text{-}delete\ hm\text{-}delete$
definition $hs\text{-}sel :: 'a::hashable\ hs \Rightarrow ('a \Rightarrow 'r\ option) \Rightarrow 'r\ option$
where $hs\text{-}sel == s\text{-}sel\ hm\text{-}sel$
definition $hs\text{-}sel' == sel\text{-}sel'\ hs\text{-}sel$
definition $hs\text{-}isEmpty :: 'a::hashable\ hs \Rightarrow bool$
where $hs\text{-}isEmpty == s\text{-}isEmpty\ hm\text{-}isEmpty$

definition $hs\text{-}iteratei :: 'a::hashable\ hs \Rightarrow ('a,'\sigma)\ set\text{-}iterator$

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* **using** *hs-iteratei-impl* **.**

interpretation *hs*: *set-inj-image-filter hs-α hs-invar hs-α hs-invar hs-inj-image-filter* **using** *hs-inj-image-filter-impl* **.**
interpretation *hs*: *set-image-filter hs-α hs-invar hs-α hs-invar hs-image-filter* **using** *hs-image-filter-impl* **.**
interpretation *hs*: *set-inj-image hs-α hs-invar hs-α hs-invar hs-inj-image* **using** *hs-inj-image-impl* **.**
interpretation *hs*: *set-union-dj hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-union-dj* **using** *hs-union-dj-impl* **.**
interpretation *hs*: *set-union hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-union* **using** *hs-union-impl* **.**
interpretation *hs*: *set-inter hs-α hs-invar hs-α hs-invar hs-α hs-invar hs-inter* **using** *hs-inter-impl* **.**
interpretation *hs*: *set-image hs-α hs-invar hs-α hs-invar hs-image* **using** *hs-image-impl* **.**
interpretation *hs*: *set-sel hs-α hs-invar hs-sel* **using** *hs-sel-impl* **.**
interpretation *hs*: *set-sel′ hs-α hs-invar hs-sel′* **using** *hs-sel′-impl* **.**
interpretation *hs*: *set-ins-dj hs-α hs-invar hs-ins-dj* **using** *hs-ins-dj-impl* **.**
interpretation *hs*: *set-delete hs-α hs-invar hs-delete* **using** *hs-delete-impl* **.**
interpretation *hs*: *set-ins hs-α hs-invar hs-ins* **using** *hs-ins-impl* **.**
interpretation *hs*: *set-memb hs-α hs-invar hs-memb* **using** *hs-memb-impl* **.**
interpretation *hs*: *set-to-list hs-α hs-invar hs-to-list* **using** *hs-to-list-impl* **.**
interpretation *hs*: *list-to-set hs-α hs-invar list-to-hs* **using** *list-to-hs-impl* **.**
interpretation *hs*: *set-isEmpty hs-α hs-invar hs-isEmpty* **using** *hs-isEmpty-impl* **.**
interpretation *hs*: *set-empty hs-α hs-invar hs-empty* **using** *hs-empty-impl* **.**

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 $ts\text{-}isEmpty :: 'a\ ts \Rightarrow bool$
where $ts\text{-}isEmpty == s\text{-}isEmpty\ tm\text{-}isEmpty$

definition $ts\text{-}iteratei :: 'a\ ts \Rightarrow ('a\ list,'\sigma)\ set\text{-}iterator$
where $ts\text{-}iteratei == s\text{-}iteratei\ tm\text{-}iteratei$

definition $ts\text{-}ball :: 'a\ ts \Rightarrow ('a\ list \Rightarrow bool) \Rightarrow bool$
where $ts\text{-}ball == s\text{-}ball\ tm\text{-}ball$

definition $ts\text{-}union :: 'a\ ts \Rightarrow 'a\ ts \Rightarrow 'a\ ts$
where $ts\text{-}union == s\text{-}union\ tm\text{-}add$

definition $ts\text{-}union\text{-}dj :: 'a\ ts \Rightarrow 'a\ ts \Rightarrow 'a\ ts$
where $ts\text{-}union\text{-}dj == s\text{-}union\text{-}dj\ tm\text{-}add\text{-}dj$

definition $ts\text{-}inter :: 'a\ ts \Rightarrow 'a\ ts \Rightarrow 'a\ ts$
where $ts\text{-}inter == it\text{-}inter\ ts\text{-}iteratei\ ts\text{-}memb\ ts\text{-}empty\ ts\text{-}ins\text{-}dj$

definition $ts\text{-}size :: 'a\ ts \Rightarrow nat$

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* **using** *ts-iteratei-impl* **.**

interpretation *ts*: *set-inj-image-filter ts-α ts-invar ts-α ts-invar ts-inj-image-filter* **using** *ts-inj-image-filter-impl* **.**
interpretation *ts*: *set-image-filter ts-α ts-invar ts-α ts-invar ts-image-filter* **using** *ts-image-filter-impl* **.**
interpretation *ts*: *set-inj-image ts-α ts-invar ts-α ts-invar ts-inj-image* **using** *ts-inj-image-impl* **.**
interpretation *ts*: *set-union-dj ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-union-dj* **using** *ts-union-dj-impl* **.**
interpretation *ts*: *set-union ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-union* **using** *ts-union-impl* **.**
interpretation *ts*: *set-inter ts-α ts-invar ts-α ts-invar ts-α ts-invar ts-inter* **using** *ts-inter-impl* **.**
interpretation *ts*: *set-image ts-α ts-invar ts-α ts-invar ts-image* **using** *ts-image-impl* **.**
interpretation *ts*: *set-sel ts-α ts-invar ts-sel* **using** *ts-sel-impl* **.**
interpretation *ts*: *set-sel′ ts-α ts-invar ts-sel′* **using** *ts-sel′-impl* **.**
interpretation *ts*: *set-ins-dj ts-α ts-invar ts-ins-dj* **using** *ts-ins-dj-impl* **.**
interpretation *ts*: *set-delete ts-α ts-invar ts-delete* **using** *ts-delete-impl* **.**
interpretation *ts*: *set-ball ts-α ts-invar ts-ball* **using** *ts-ball-impl* **.**
interpretation *ts*: *set-ins ts-α ts-invar ts-ins* **using** *ts-ins-impl* **.**
interpretation *ts*: *set-memb ts-α ts-invar ts-memb* **using** *ts-memb-impl* **.**
interpretation *ts*: *set-to-list ts-α ts-invar ts-to-list* **using** *ts-to-list-impl* **.**
interpretation *ts*: *list-to-set ts-α ts-invar list-to-ts* **using** *list-to-ts-impl* **.**
interpretation *ts*: *set-isEmpty ts-α ts-invar ts-isEmpty* **using** *ts-isEmpty-impl* **.**
interpretation *ts*: *set-size ts-α ts-invar ts-size* **using** *ts-size-impl* **.**
interpretation *ts*: *set-empty ts-α ts-invar ts-empty* **using** *ts-empty-impl* **.**

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* ⇒ $('a,'σ)$ *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* **using** *ahs-iteratei-impl*

.

interpretation *ahs*: *set-inj-image-filter ahs-α ahs-invar ahs-α ahs-invar ahs-inj-image-filter* **using** *ahs-inj-image-filter-impl* .
interpretation *ahs*: *set-image-filter ahs-α ahs-invar ahs-α ahs-invar ahs-image-filter* **using** *ahs-image-filter-impl* .
interpretation *ahs*: *set-inj-image ahs-α ahs-invar ahs-α ahs-invar ahs-inj-image* **using** *ahs-inj-image-impl* .
interpretation *ahs*: *set-union-dj ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-union-dj* **using** *ahs-union-dj-impl* .
interpretation *ahs*: *set-union ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-union* **using** *ahs-union-impl* .
interpretation *ahs*: *set-inter ahs-α ahs-invar ahs-α ahs-invar ahs-α ahs-invar ahs-inter* **using** *ahs-inter-impl* .
interpretation *ahs*: *set-image ahs-α ahs-invar ahs-α ahs-invar ahs-image* **using** *ahs-image-impl* .
interpretation *ahs*: *set-sel ahs-α ahs-invar ahs-sel* **using** *ahs-sel-impl* .
interpretation *ahs*: *set-sel′ ahs-α ahs-invar ahs-sel′* **using** *ahs-sel′-impl* .
interpretation *ahs*: *set-ins-dj ahs-α ahs-invar ahs-ins-dj* **using** *ahs-ins-dj-impl* .
interpretation *ahs*: *set-delete ahs-α ahs-invar ahs-delete* **using** *ahs-delete-impl* .
interpretation *ahs*: *set-ins ahs-α ahs-invar ahs-ins* **using** *ahs-ins-impl* .
interpretation *ahs*: *set-memb ahs-α ahs-invar ahs-memb* **using** *ahs-memb-impl* .
interpretation *ahs*: *set-to-list ahs-α ahs-invar ahs-to-list* **using** *ahs-to-list-impl* .
interpretation *ahs*: *list-to-set ahs-α ahs-invar list-to-ahs* **using** *list-to-ahs-impl* .
interpretation *ahs*: *set-isEmpty ahs-α ahs-invar ahs-isEmpty* **using** *ahs-isEmpty-impl* .
interpretation *ahs*: *set-empty ahs-α ahs-invar ahs-empty* **using** *ahs-empty-impl* .

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* $\Rightarrow$ *nat set* **where** *ias-*α == *s-*α *iam-*α
abbreviation (*input*) *ias-invar* :: *ias* $\Rightarrow$ *bool* **where** *ias-invar* == *iam-invar*
definition *ias-empty* :: *unit* $\Rightarrow$ *ias* **where** *ias-empty* == *s-empty iam-empty*
definition *ias-memb* :: *nat* $\Rightarrow$ *ias* $\Rightarrow$ *bool*
where *ias-memb* == *s-memb iam-lookup*
definition *ias-ins* :: *nat* $\Rightarrow$ *ias* $\Rightarrow$ *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* **using** *ias-iteratei-impl*
.

interpretation *ias*: *set-inj-image-filter ias-α ias-invar ias-α ias-invar ias-inj-image-filter* **using** *ias-inj-image-filter-impl* **.**
interpretation *ias*: *set-image-filter ias-α ias-invar ias-α ias-invar ias-image-filter* **using** *ias-image-filter-impl* **.**
interpretation *ias*: *set-inj-image ias-α ias-invar ias-α ias-invar ias-inj-image* **using** *ias-inj-image-impl* **.**
interpretation *ias*: *set-union-dj ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-union-dj* **using** *ias-union-dj-impl* **.**
interpretation *ias*: *set-union ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-union* **using** *ias-union-impl* **.**
interpretation *ias*: *set-inter ias-α ias-invar ias-α ias-invar ias-α ias-invar ias-inter* **using** *ias-inter-impl* **.**
interpretation *ias*: *set-image ias-α ias-invar ias-α ias-invar ias-image* **using** *ias-image-impl* **.**
interpretation *ias*: *set-sel ias-α ias-invar ias-sel* **using** *ias-sel-impl* **.**
interpretation *ias*: *set-sel′ ias-α ias-invar ias-sel′* **using** *ias-sel′-impl* **.**

interpretation *ias*: *set-ins-dj ias-*α *ias-invar ias-ins-dj* **using** *ias-ins-dj-impl* **.**
interpretation *ias*: *set-delete ias-*α *ias-invar ias-delete* **using** *ias-delete-impl* **.**
interpretation *ias*: *set-ins ias-*α *ias-invar ias-ins* **using** *ias-ins-impl* **.**
interpretation *ias*: *set-memb ias-*α *ias-invar ias-memb* **using** *ias-memb-impl* **.**
interpretation *ias*: *set-to-list ias-*α *ias-invar ias-to-list* **using** *ias-to-list-impl* **.**
interpretation *ias*: *list-to-set ias-*α *ias-invar list-to-ias* **using** *list-to-ias-impl* **.**
interpretation *ias*: *set-isEmpty ias-*α *ias-invar ias-isEmpty* **using** *ias-isEmpty-impl*
.
interpretation *ias*: *set-empty ias-*α *ias-invar ias-empty* **using** *ias-empty-impl* **.**

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 $'a\ fifo = 'a\ list \times 'a\ list$

— 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* :: $'a \Rightarrow 'a$ *fifo* $\Rightarrow 'a$ *fifo*
where *fifo-enqueue a F* == $(a\#fst\ F,\ snd\ F)$

— Get an element from the fifo
definition *fifo-dequeue* :: $'a$ *fifo* $\Rightarrow ('a \times 'a$ *fifo*$)$ **where**
fifo-dequeue F ==
case snd F of
$(a\#l) \Rightarrow (a,\ (fst\ F,\ l))$ |
$[] \Rightarrow$ *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* :: $'a \Rightarrow 'a$ *fifo* $\Rightarrow 'a$ *fifo*
where *fifo-push x F* == *case F of* $(e,d) \Rightarrow (e,x\#d)$

— 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* $\Rightarrow 'a$ *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*
apply (*unfold-locales*)
by (*auto simp add*: *fifo-α-def fifo-empty-def*)

lemma *fifo-isEmpty-impl*: *list-isEmpty fifo-α fifo-invar fifo-isEmpty*
apply (*unfold-locales*)
by (*case-tac s*) (*auto simp add*: *fifo-isEmpty-def fifo-α-def*)

lemma *fifo-enqueue-impl*: *list-enqueue fifo-α fifo-invar fifo-enqueue*
apply (*unfold-locales*)
by (*auto simp add*: *fifo-enqueue-def fifo-α-def*)

lemma *fifo-dequeue-impl*: *list-dequeue fifo-α fifo-invar fifo-dequeue*
apply (*unfold-locales*)
apply (*case-tac s*)
apply (*case-tac b*)
apply (*auto simp add*: *fifo-dequeue-def fifo-α-def Let-def*) [*2*]
apply (*case-tac s*)
apply (*case-tac b*)
apply (*auto simp add*: *fifo-dequeue-def fifo-α-def Let-def*)
done

interpretation *fifo*: *list-empty fifo-α fifo-invar fifo-empty* **using** *fifo-empty-impl* **.**
interpretation *fifo*: *list-isEmpty fifo-α fifo-invar fifo-isEmpty* **using** *fifo-isEmpty-impl*
.
interpretation *fifo*: *list-enqueue fifo-α fifo-invar fifo-enqueue* **using** *fifo-enqueue-impl*
.
interpretation *fifo*: *list-dequeue fifo-α fifo-invar fifo-dequeue* **using** *fifo-dequeue-impl*
.

lemma *fifo-pop-impl*: *list-pop fifo-α fifo-invar fifo-pop*
  by *unfold-locales*

lemma *fifo-push-impl*: *list-push fifo-α fifo-invar fifo-push*
  apply *unfold-locales*
  by (*auto simp add*: *fifo-push-def fifo-α-def*)

interpretation *fifo*: *list-pop fifo-α fifo-invar fifo-pop* **using** *fifo-pop-impl* **.**
interpretation *fifo*: *list-push fifo-α fifo-invar fifo-push* **using** *fifo-push-impl* **.**

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*
by (*unfold-locales*, *auto simp add*: *bino-defs*)

theorem *bino-isEmpty-impl*: *prio-isEmpty bino-α bino-invar bino-isEmpty*
by *unfold-locales*
(*simp add*: *bino-defs BinomialHeap.isEmpty-correct BinomialHeap.empty-correct*)

theorem *bino-find-impl*: *prio-find bino-α bino-invar bino-find*
apply *unfold-locales*
apply (*simp add*: *bino-defs BinomialHeap.empty-correct BinomialHeap.findMin-correct*)
done

lemma *bino-insert-impl*: *prio-insert bino-α bino-invar bino-insert*
apply(*unfold-locales*)

```
  apply(unfold bino-defs)
  apply (simp-all add: BinomialHeap.insert-correct)
  done

lemma bino-meld-impl: prio-meld bino-α bino-invar bino-meld
  apply(unfold-locales)
  apply(unfold bino-defs)
  apply(simp-all add: BinomialHeap.meld-correct)
  done

lemma bino-delete-impl:
  prio-delete bino-α bino-invar bino-find bino-delete
  apply intro-locales
  apply (rule bino-find-impl)
  apply(unfold-locales)
 apply(simp-all add: bino-defs BinomialHeap.empty-correct BinomialHeap.deleteMin-correct)
done

interpretation bino: prio-empty bino-α bino-invar bino-empty
using bino-empty-impl .
interpretation bino: prio-isEmpty bino-α bino-invar bino-isEmpty
using bino-isEmpty-impl .
interpretation bino: prio-insert bino-α bino-invar bino-insert
using bino-insert-impl .
interpretation bino: prio-delete bino-α bino-invar bino-find bino-delete
using bino-delete-impl .
interpretation bino: prio-meld bino-α bino-invar bino-meld
using bino-meld-impl .

interpretation binos: StdPrio bino-ops
  apply (unfold bino-ops-def)
  apply intro-locales
  apply simp-all
  apply unfold-locales
  using prio-delete.delete-correct[OF bino-delete-impl]
  apply auto
  done

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*
by (*auto simp add*: *bino-ops-def*)

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*
by (*unfold-locales*, *auto simp add*: *skew-defs*)

theorem *skew-isEmpty-impl*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
by *unfold-locales*
(*simp add*: *skew-defs SkewBinomialHeap.isEmpty-correct SkewBinomialHeap.empty-correct*)

theorem *skew-find-impl*: *prio-find skew-α skew-invar skew-find*
apply *unfold-locales*
apply (*simp add*: *skew-defs SkewBinomialHeap.empty-correct SkewBinomial-Heap.findMin-correct*)
done

lemma *skew-insert-impl*: *prio-insert skew-α skew-invar skew-insert*
apply(*unfold-locales*)
apply(*unfold skew-defs*)
apply (*simp-all add*: *SkewBinomialHeap.insert-correct*)
done

lemma *skew-meld-impl*: *prio-meld skew-α skew-invar skew-meld*
apply(*unfold-locales*)
apply(*unfold skew-defs*)
apply(*simp-all add*: *SkewBinomialHeap.meld-correct*)
done

lemma *skew-delete-impl*:
prio-delete skew-α skew-invar skew-find skew-delete
apply *intro-locales*
apply (*rule skew-find-impl*)

apply(*unfold-locales*)
apply(*simp-all add*: *skew-defs SkewBinomialHeap.empty-correct SkewBinomialHeap.deleteMin-correct*)
done

interpretation *skew*: *prio-empty skew-α skew-invar skew-empty*
using *skew-empty-impl* **.**
interpretation *skew*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
using *skew-isEmpty-impl* **.**
interpretation *skew*: *prio-insert skew-α skew-invar skew-insert*
using *skew-insert-impl* **.**
interpretation *skew*: *prio-delete skew-α skew-invar skew-find skew-delete*
using *skew-delete-impl* **.**
interpretation *skew*: *prio-meld skew-α skew-invar skew-meld*
using *skew-meld-impl* **.**

interpretation *skews*: *StdPrio skew-ops*
apply (*unfold skew-ops-def*)
apply *intro-locales*
apply *simp-all*
apply *unfold-locales*
using *prio-delete.delete-correct*[*OF skew-delete-impl*]
apply *auto*
done

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*
by (*auto simp add*: *skew-ops-def*)

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*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.empty-correct*)
done

lemma *ft-consl-impl*: *al-consl ft-α ft-invar ft-consl*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.lcons-correct*)
done

lemma *ft-consr-impl*: *al-consr ft-α ft-invar ft-consr*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.rcons-correct*)
done

lemma *ft-isEmpty-impl*: *al-isEmpty ft-α ft-invar ft-isEmpty*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.isEmpty-correct FingerTree.empty-correct*)
done

lemma *ft-count-impl*: *al-count ft-α ft-invar ft-count*

apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.count-correct*)
done

lemma *ft-head-impl*: *al-head ft-α ft-invar ft-head*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.head-correct FingerTree.empty-correct*)
done

lemma *ft-tail-impl*: *al-tail ft-α ft-invar ft-tail*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.tail-correct FingerTree.empty-correct*)
done

lemma *ft-headR-impl*: *al-headR ft-α ft-invar ft-headR*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.headR-correct FingerTree.empty-correct*)
done

lemma *ft-tailR-impl*: *al-tailR ft-α ft-invar ft-tailR*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.tailR-correct FingerTree.empty-correct*)
done

lemma *ft-foldl-impl*: *al-foldl ft-α ft-invar ft-foldl*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.foldl-correct*)
done

lemma *ft-foldr-impl*: *al-foldr ft-α ft-invar ft-foldr*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.foldr-correct*)
done

lemma *ft-app-impl*: *al-app ft-α ft-invar ft-app*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.app-correct*)
done

lemma *ft-annot-impl*: *al-annot ft-α ft-invar ft-annot*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.annot-correct*)
done

lemma *ft-splits-impl*: *al-splits ft-α ft-invar ft-splits*
apply *unfold-locales*
apply (*unfold ft-defs*)
apply (*simp only*: *FingerTree.annot-correct*[*symmetric*])
apply (*frule* (*3*) *FingerTree.splitTree-correct*(*1*))

apply (*frule* (*3*) *FingerTree.splitTree-correct*(*2*))
apply (*frule* (*3*) *FingerTree.splitTree-correct*(*3*))
apply (*simp only*: *FingerTree.annot-correct*[*symmetric*])
apply *simp*
done

4.23.2 Interpretations

interpretation *ft*: *al-empty ft-α ft-invar ft-empty* **using** *ft-empty-impl* **.**
interpretation *ft*: *al-isEmpty ft-α ft-invar ft-isEmpty* **using** *ft-isEmpty-impl* **.**
interpretation *ft*: *al-count ft-α ft-invar ft-count* **using** *ft-count-impl* **.**
interpretation *ft*: *al-consl ft-α ft-invar ft-consl* **using** *ft-consl-impl* **.**
interpretation *ft*: *al-consr ft-α ft-invar ft-consr* **using** *ft-consr-impl* **.**
interpretation *ft*: *al-head ft-α ft-invar ft-head* **using** *ft-head-impl* **.**
interpretation *ft*: *al-tail ft-α ft-invar ft-tail* **using** *ft-tail-impl* **.**
interpretation *ft*: *al-headR ft-α ft-invar ft-headR* **using** *ft-headR-impl* **.**
interpretation *ft*: *al-tailR ft-α ft-invar ft-tailR* **using** *ft-tailR-impl* **.**
interpretation *ft*: *al-foldl ft-α ft-invar ft-foldl* **using** *ft-foldl-impl* **.**
interpretation *ft*: *al-foldr ft-α ft-invar ft-foldr* **using** *ft-foldr-impl* **.**
interpretation *ft*: *al-app ft-α ft-invar ft-app* **using** *ft-app-impl* **.**
interpretation *ft*: *al-annot ft-α ft-invar ft-annot* **using** *ft-annot-impl* **.**
interpretation *ft*: *al-splits ft-α ft-invar ft-splits* **using** *ft-splits-impl* **.**

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*
apply (*rule StdAL.intro*)
apply (*simp-all add*: *ft-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *ft-ops-def*)

interpretation *ftr*!: *al-foldl* (*alist-op-α ft-ops*) (*alist-op-invar ft-ops*) *ft-foldl*
using *ft-foldl-impl*[*folded ft-ops-unfold*] **.**
interpretation *ftr*!: *al-foldr* (*alist-op-α ft-ops*) (*alist-op-invar ft-ops*) *ft-foldr*
using *ft-foldr-impl*[*folded ft-ops-unfold*] **.**

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* $\Rightarrow$ $('a\times'p)$ *multiset* **where**
alprioi-α = *alprio-α ft-α*
definition *alprioi-invar* **where**
alprioi-invar = *alprio-invar ft-α ft-invar*
definition *alprioi-empty*
:: *unit* $\Rightarrow$ $($*unit*$,('a,'b::$*linorder*$)$ *Prio*$)$ *FingerTree* **where**
alprioi-empty = *alprio-empty ft-empty*
definition *alprioi-isEmpty* **where**
alprioi-isEmpty $\equiv$ *alprio-isEmpty ft-isEmpty*
definition *alprioi-insert* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ $($*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*
by (*rule alprioi-impl*)

interpretation *alprioi*: *prio-isEmpty alprioi-α alprioi-invar alprioi-isEmpty*
by (*rule alprioi-impl*)
interpretation *alprioi*: *prio-insert alprioi-α alprioi-invar alprioi-insert*
by (*rule alprioi-impl*)
interpretation *alprioi*: *prio-find alprioi-α alprioi-invar alprioi-find*
by (*rule alprioi-impl*)
interpretation *alprioi*: *prio-delete alprioi-α alprioi-invar alprioi-find alprioi-delete*
by (*rule alprioi-impl*)
interpretation *alprioi*: *prio-meld alprioi-α alprioi-invar alprioi-meld*
by (*rule alprioi-impl*)

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*
apply (*unfold alprioi-ops-def*)
apply *intro-locales*
apply (*simp-all add*: *alprioi-impl alprioi-delete-impl*[*unfolded prio-delete-def*])
done

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*
by (*auto simp add*: *alprioi-ops-def*)

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 $('a,'b)$ *alupriooi* = (*unit*, $('a, 'b)$ *LP*) *FingerTree*

definition *aluprioi-α* :: $('a::linorder,'b::linorder)$ *aluprioi* $\Rightarrow$ $'a \rightharpoonup 'b$ **where**
  aluprioi-α = *aluprio-α ft-α*
definition *aluprioi-invar* **where**
  aluprioi-invar = *aluprio-invar ft-α ft-invar*
definition *aluprioi-empty*
  :: *unit* $\Rightarrow$ (*unit*,$('a::linorder,'b::linorder)$ *LP*) *FingerTree* **where**
  aluprioi-empty = *aluprio-empty ft-empty*
definition *aluprioi-isEmpty* **where**
  aluprioi-isEmpty $\equiv$ *aluprio-isEmpty ft-isEmpty*
definition *aluprioi-insert* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ (*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

alupriori-empty-def
alupriori-isEmpty-def
alupriori-insert-def
alupriori-pop-def
alupriori-prio-def

4.25.2 Correctness

lemmas *alupriori-finite-impl* = *aluprio-finite-correct*[*of ft-α ft-invar*, *folded alupriori-defs*]
lemmas *alupriori-empty-impl* = *aluprio-empty-correct*[*OF ft-empty-impl*, *folded alupriori-defs*]
lemmas *alupriori-isEmpty-impl* = *aluprio-isEmpty-correct*[*OF ft-isEmpty-impl*, *folded alupriori-defs*]
lemmas *alupriori-insert-impl* = *aluprio-insert-correct*[
  OF
    ft-splits-impl ft-annot-impl ft-isEmpty-impl
    ft-app-impl ft-consr-impl,
  folded alupriori-defs]
lemmas *alupriori-pop-impl* = *aluprio-pop-correct*[*OF ft-splits-impl ft-annot-impl ft-app-impl*, *folded alupriori-defs*]
lemmas *alupriori-prio-impl* = *aluprio-prio-correct*[*OF ft-splits-impl ft-annot-impl ft-isEmpty-impl*, *folded alupriori-defs*]

lemmas *alupriori-impl* =
  alupriori-finite-impl
  alupriori-empty-impl
  alupriori-isEmpty-impl
  alupriori-insert-impl
  alupriori-pop-impl
  alupriori-prio-impl

interpretation *alupriori*: *uprio-finite alupriori-α alupriori-invar*
  by (*rule alupriori-impl*)
interpretation *alupriori*: *uprio-empty alupriori-α alupriori-invar alupriori-empty*
  by (*rule alupriori-impl*)
interpretation *alupriori*: *uprio-isEmpty alupriori-α alupriori-invar alupriori-isEmpty*

  by(*auto simp add*:*alupriori-impl*)
interpretation *alupriori*: *uprio-insert alupriori-α alupriori-invar alupriori-insert*
  by (*rule alupriori-impl*)
interpretation *alupriori*: *uprio-pop alupriori-α alupriori-invar alupriori-pop*
  by (*rule alupriori-impl*)
interpretation *alupriori*: *uprio-prio alupriori-α alupriori-invar alupriori-prio*
  by (*rule alupriori-impl*)

lemmas *alupriori-correct* =
  alupriori.finite-correct
  alupriori.empty-correct
  alupriori.isEmpty-correct
  alupriori.insert-correct

aluprioi.pop-correct
aluprioi.prio-correct

Record Based Interface

interpretation *aluprioir*: *StdUprio aluprioi-ops*
apply *intro-locales*
apply (*unfold aluprioi-ops-def*)
apply (*simp-all add*: *aluprioi-impl*)
done

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*
by (*auto simp add*: *aluprioi-ops-def*)

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*: $[\![xs \neq []; \mathit{sorted}\ xs]\!] \Longrightarrow \forall x \in \mathit{set}\ xs.\ \mathit{hd}\ xs \leq x$
by(*induct xs*, *auto simp add*: *sorted-Cons*)

lemma *map-hd*: $[\![xs \neq []]\!] \Longrightarrow f\ (\mathit{hd}\ xs) = \mathit{hd}\ (\mathit{map}\ f\ xs)$
apply(*induct xs*)
apply(*auto*)
done

lemma *map-tl*: $\mathit{map}\ f\ (\mathit{tl}\ xs) = \mathit{tl}\ (\mathit{map}\ f\ xs)$

apply(*induct xs*)
apply(*auto*)
done

lemma *drop-1-tl*: *drop 1 xs* = *tl xs*
apply(*induct xs*)
apply(*auto*)
done

lemma *remove1-tl*: $xs \neq [] \Longrightarrow$ *remove1* (*hd xs*) *xs* = *tl xs*
apply(*induct xs*, *simp*)
apply(*auto*)
done

lemma *sorted-append-bigger*:
$[\![$*sorted xs*; $\forall x \in$ *set xs*. $x \leq y]\!] \Longrightarrow$ *sorted* (*xs* @ [*y*])
apply (*induct xs*)
apply *simp*
proof −
case *goal1*
from *goal1* **have** *s*: *sorted xs* **by** (*cases xs*) *simp-all*
from *goal1* **have** *a*: $\forall x \in$*set xs*. $x \leq y$ **by** *simp*
from *goal1*(*1*)[*OF s a*] *goal1*(*2*−) **show** *?case* **by** (*cases xs*) *simp-all*
qed

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. $y \neq x$) (*rev xs*)) @ *a*
by (*induct xs arbitrary*: *a*) *auto*

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
using assms
proof (induct l1 arbitrary: l2)
  case Nil thus ?case by simp
next
  case (Cons x1 l1 l2)
  note x1-l1-props = Cons(2)
  note l2-props = Cons(3)

  from x1-l1-props have l1-props: distinct l1 ∧ sorted l1
                    and x1-nin-l1: x1 ∉ set l1
                    and x1-le: ⋀x. x ∈ set l1 ⟹ x1 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

  note ind-hyp-l1 = Cons(1)[OF l1-props]

  show ?case
  using l2-props
  proof (induct l2)
    case Nil with x1-l1-props show ?case by simp
  next
    case (Cons x2 l2)
    note x2-l2-props = Cons(2)
    from x2-l2-props have l2-props: distinct l2 ∧ sorted l2
                    and x2-nin-l2: x2 ∉ set l2
                    and x2-le: ⋀x. x ∈ set l2 ⟹ x2 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

    note ind-hyp-l2 = Cons(1)[OF l2-props]
    show ?case
    proof (cases x1 < x2)
      case True note x1-less-x2 = this

      from ind-hyp-l1[OF x2-l2-props] x1-less-x2 x1-nin-l1 x1-le x2-le
      show ?thesis
        apply (auto simp add: sorted-Cons Ball-def)
        apply (metis linorder-not-le)
        apply (metis linorder-not-less xt1(6) xt1(9))
      done
    next
      case False note x2-le-x1 = this

      show ?thesis
```

```
      proof (cases x1 = x2)
        case True note x1-eq-x2 = this

        from ind-hyp-l1[OF l2-props] x1-le x2-le x2-nin-l2 x1-eq-x2 x1-nin-l1
        show ?thesis by (simp add: x1-eq-x2 sorted-Cons Ball-def)
      next
        case False note x1-neq-x2 = this
        with x2-le-x1 have x2-less-x1 : x2 < x1 by auto

        from ind-hyp-l2 x2-le-x1 x1-neq-x2 x2-le x2-nin-l2 x1-le
        show ?thesis
          apply (simp add: x2-less-x1 sorted-Cons Ball-def)
          apply (metis linorder-not-le x2-less-x1 xt1(7))
        done
      qed
    qed
  qed
qed

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
by pat-completeness simp-all
termination
by (relation measure (λ(acc, ls). 3 ∗ length acc + 2 ∗ length ls)) (simp-all)

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))
using assms
proof (induct as ls rule: merge-list.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by simp
next
  case 3 thus ?case by simp
next
  case 4 thus ?case by auto
next
  case (5 acc l1 l2 ls)
  note ind-hyp = 5(1)
  note l12-l-OK = 5(2)
  note acc-OK = 5(3)
```

from *l12-l-OK acc-OK merge-correct*[*of l1 l2*]
  have *set-merge-eq*: *set* (*merge l1 l2*) = *set l1* ∪ *set l2* **by** *auto*

from *l12-l-OK acc-OK merge-correct*[*of l1 l2*]
have *distinct* (*merge-list* (*merge l1 l2* # *acc*) *ls*) ∧
    sorted (*merge-list* (*merge l1 l2* # *acc*) *ls*) ∧
    set (*merge-list* (*merge l1 l2* # *acc*) *ls*) =
    set (*concat* ((*merge l1 l2* # *acc*) @ *ls*))
  by (*rule-tac ind-hyp*) *auto*
with *set-merge-eq* **show** *?case* **by** *auto*
qed

definition *mergesort* **where**
  mergesort xs = *merge-list* [] (*map* (λ*x*. [*x*]) *xs*)

lemma *mergesort-correct* :
  distinct (*mergesort l*) ∧ *sorted* (*mergesort l*) ∧ (*set* (*mergesort l*) = *set l*)
proof −
  let *?l′* = *map* (λ*x*. [*x*]) *l*

  { **fix** *xs*
    assume *xs* ∈ *set ?l′*
    then obtain *x* **where** *xs-eq*: *xs* = [*x*] **by** *auto*
    hence *distinct xs* ∧ *sorted xs* **by** *simp*
  } **note** *l′-OK* = *this*

  from *merge-list-correct*[*of ?l′* [], *OF l′-OK*]
  show *?thesis* **unfolding** *mergesort-def* **by** *simp*
qed

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*
using *assms*
proof (*induct l1 arbitrary*: *l2*)
  case *Nil* **thus** *?case* **by** *simp*

```
next
  case (Cons x1 l1 l2)
  note x1-l1-props = Cons(2)
  note l2-props = Cons(3)

  from x1-l1-props have l1-props: distinct l1 ∧ sorted l1
                  and x1-nin-l1: x1 ∉ set l1
                  and x1-le: ⋀x. x ∈ set l1 ⟹ x1 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

  note ind-hyp-l1 = Cons(1)[OF l1-props]

  show ?case
  using l2-props
  proof (induct l2)
    case Nil with x1-l1-props show ?case by simp
  next
    case (Cons x2 l2)
    note x2-l2-props = Cons(2)
    from x2-l2-props have l2-props: distinct l2 ∧ sorted l2
                  and x2-nin-l2: x2 ∉ set l2
                  and x2-le: ⋀x. x ∈ set l2 ⟹ x2 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

    note ind-hyp-l2 = Cons(1)[OF l2-props]
    show ?case
    proof (cases x1 < x2)
      case True note x1-less-x2 = this

      from ind-hyp-l1[OF x2-l2-props] x1-less-x2 x1-nin-l1 x1-le x2-le
      show ?thesis
        apply (auto simp add: sorted-Cons Ball-def)
        apply (metis linorder-not-le)
      done
    next
      case False note x2-le-x1 = this

      show ?thesis
      proof (cases x1 = x2)
        case True note x1-eq-x2 = this

        from ind-hyp-l1[OF l2-props] x1-le x2-le x2-nin-l2 x1-eq-x2 x1-nin-l1
        show ?thesis by (simp add: x1-eq-x2 sorted-Cons Ball-def)
      next
        case False note x1-neq-x2 = this
        with x2-le-x1 have x2-less-x1 : x2 < x1 by auto

        from ind-hyp-l2 x2-le-x1 x1-neq-x2 x2-le x2-nin-l2 x1-le
        show ?thesis
```

```
          apply (auto simp add: x2-less-x1 sorted-Cons Ball-def)
          apply (metis linorder-not-le x2-less-x1)
        done
      qed
    qed
  qed
qed

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
using assms
proof (induct l1 arbitrary: l2)
  case Nil thus ?case by simp
next
  case (Cons x1 l1 l2)
  note x1-l1-props = Cons(2)
  note l2-props = Cons(3)

  from x1-l1-props have l1-props: distinct l1 ∧ sorted l1
                    and x1-nin-l1: x1 ∉ set l1
                    and x1-le: ⋀x. x ∈ set l1 ⟹ x1 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

  note ind-hyp-l1 = Cons(1)[OF l1-props]

  show ?case
  using l2-props
  proof (induct l2)
    case Nil with x1-l1-props show ?case by simp
  next
    case (Cons x2 l2)
    note x2-l2-props = Cons(2)
    from x2-l2-props have l2-props: distinct l2 ∧ sorted l2
                    and x2-nin-l2: x2 ∉ set l2
                    and x2-le: ⋀x. x ∈ set l2 ⟹ x2 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

    note ind-hyp-l2 = Cons(1)[OF l2-props]
    show ?case
```

```
    proof (cases x1 < x2)
      case True note x1-less-x2 = this

      from ind-hyp-l1[OF x2-l2-props] x1-less-x2 x1-nin-l1 x1-le x2-le
      show ?thesis
        apply simp
        apply (simp add: sorted-Cons Ball-def set-eq-iff)
        apply (metis linorder-not-le order-less-imp-not-eq2)
      done
    next
      case False note x2-le-x1 = this

      show ?thesis
      proof (cases x1 = x2)
        case True note x1-eq-x2 = this

        from ind-hyp-l1[OF l2-props] x1-le x2-le x2-nin-l2 x1-eq-x2 x1-nin-l1
        show ?thesis by (simp add: x1-eq-x2 sorted-Cons Ball-def)
      next
        case False note x1-neq-x2 = this
        with x2-le-x1 have x2-less-x1 : x2 < x1 by auto

        from x2-less-x1 x1-le have x2-nin-l1: x2 ∉ set l1
           by (metis linorder-not-less)

        from ind-hyp-l2 x1-le x2-nin-l1
        show ?thesis
             apply (simp add: x2-less-x1 x1-neq-x2 x2-le-x1 x1-nin-l1 sorted-Cons
Ball-def set-eq-iff)
          apply (metis x1-neq-x2)
        done
      qed
    qed
  qed
qed

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
using assms
proof (induct l1 arbitrary: l2)
```

```
  case Nil thus ?case by simp
next
  case (Cons x1 l1 l2)
  note x1-l1-props = Cons(2)
  note l2-props = Cons(3)

  from x1-l1-props have l1-props: distinct l1 ∧ sorted l1
                  and x1-nin-l1: x1 ∉ set l1
                  and x1-le: ⋀x. x ∈ set l1 ⟹ x1 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

  note ind-hyp-l1 = Cons(1)[OF l1-props]

  show ?case
  using l2-props
  proof (induct l2)
    case Nil with x1-l1-props show ?case by simp
  next
    case (Cons x2 l2)
    note x2-l2-props = Cons(2)
    from x2-l2-props have l2-props: distinct l2 ∧ sorted l2
                  and x2-nin-l2: x2 ∉ set l2
                  and x2-le: ⋀x. x ∈ set l2 ⟹ x2 ≤ x
    by (simp-all add: sorted-Cons Ball-def)

    note ind-hyp-l2 = Cons(1)[OF l2-props]
    show ?case
    proof (cases x1 < x2)
      case True note x1-less-x2 = this

      from ind-hyp-l1[OF x2-l2-props] x1-less-x2 x1-nin-l1 x1-le x2-le
      show ?thesis
        apply (auto simp add: sorted-Cons Ball-def)
        apply (metis linorder-not-le)
      done
    next
      case False note x2-le-x1 = this

      show ?thesis
      proof (cases x1 = x2)
        case True note x1-eq-x2 = this

        from ind-hyp-l1[OF l2-props] x1-le x2-le x2-nin-l2 x1-eq-x2 x1-nin-l1
        show ?thesis
          apply (simp add: subset-iff x1-eq-x2 sorted-Cons Ball-def)
          apply metis
        done
      next
        case False note x1-neq-x2 = this
```

```
      with x2-le-x1 have x2-less-x1 : x2 < x1 by auto

      from ind-hyp-l2 x2-le-x1 x1-neq-x2 x2-le x2-nin-l2 x1-le
      show ?thesis
        apply (simp add: subset-iff x2-less-x1 sorted-Cons Ball-def)
        apply (metis linorder-not-le x2-less-x1)
      done
    qed
  qed
 qed
qed

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
 using assms
proof -
 have l12-eq: l1 = l2 ⟷ subset-sorted l1 l2 ∧ subset-sorted l2 l1
 proof (induct l1 arbitrary: l2)
  case Nil thus ?case by (cases l2) auto
 next
  case (Cons x1 l1')
  note ind-hyp = Cons(1)

  show ?case
  proof (cases l2)
   case Nil thus ?thesis by simp
  next
   case (Cons x2 l2')
   thus ?thesis by (simp add: ind-hyp)
  qed
 qed
 also have ... ⟷ ((set l1 ⊆ set l2) ∧ (set l2 ⊆ set l1))
   using subset-sorted-correct[OF l1-OK l2-OK] subset-sorted-correct[OF l2-OK
l1-OK]
  by simp
 also have ... ⟷ set l1 = set l2 by auto
 finally show ?thesis .
qed

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
by (induct xs) (auto simp add: sorted-Cons Ball-def)
```

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*)
by (*induct xs*) (*auto simp add*: *sorted-Cons Ball-def*)

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*}
apply (*induct xs*)
apply *simp*
apply (*simp add*: *sorted-Cons Ball-def set-eq-iff*)
apply (*metis order-less-le*)
done

lemma *sorted-filter* :
  sorted l ⟹ *sorted* (*filter P l*)
by (*induct l*) (*simp-all add*: *sorted-Cons*)

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 == ListAdd.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*
by (*unfold-locales*) (*auto simp add*: *lss-defs*)

lemma *lss-memb-impl*: *set-memb lss-α lss-invar lss-memb*
by (*unfold-locales*) (*auto simp add*: *lss-defs memb-sorted-correct*)

lemma *lss-ins-impl*: *set-ins lss-α lss-invar lss-ins*
by (*unfold-locales*) (*auto simp add*: *lss-defs insertion-sort-correct*)

lemma *lss-ins-dj-impl*: *set-ins-dj lss-α lss-invar lss-ins-dj*
by (*unfold-locales*) (*auto simp add*: *lss-defs insertion-sort-correct*)

lemma *lss-delete-impl*: *set-delete lss-α lss-invar lss-delete*
by(*unfold-locales*)(*auto simp add*: *lss-delete-def lss-α-def lss-invar-def delete-sorted-correct*)

lemma *lss-α-finite*[*simp*, *intro*!]: *finite* (*lss-α l*)

by (*auto simp add*: *lss-defs*)

lemma *lss-is-finite-set*: *finite-set lss-α lss-invar*
by (*unfold-locales*) (*auto simp add*: *lss-defs*)

lemma *lss-iterateoi-impl*: *set-iterateoi lss-α lss-invar lss-iterateoi*
proof
  fix *l* :: *'a*::{*linorder*} *list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator-linord* (*lss-iterateoi l*) (*lss-α l*)
    apply (*rule-tac set-iterator-linord-I* [*of l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-iterateoi-def*)
  done
qed

lemma *lss-reverse-iterateoi-impl*: *set-reverse-iterateoi lss-α lss-invar lss-reverse-iterateoi*
proof
  fix *l* :: *'a list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator-rev-linord* (*lss-reverse-iterateoi l*) (*lss-α l*)
    apply (*rule-tac set-iterator-rev-linord-I* [*of rev l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-reverse-iterateoi-def*)
  done
qed

lemma *lss-iteratei-impl*: *set-iteratei lss-α lss-invar lss-iteratei*
proof
  fix *l* :: *'a list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator* (*lss-iteratei l*) (*lss-α l*)
    apply (*rule-tac set-iterator-I* [*of l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-iteratei-def*)
  done
qed

lemma *lss-isEmpty-impl*: *set-isEmpty lss-α lss-invar lss-isEmpty*
by(*unfold-locales*)(*auto simp add*: *lss-defs*)

lemma *lss-inter-impl*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
by (*unfold-locales*) (*auto simp add*: *lss-defs inter-sorted-correct*)

lemma *lss-union-impl*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*
by (*unfold-locales*) (*auto simp add*: *lss-defs merge-correct*)

lemma *lss-union-list-impl*: *set-union-list lss-α lss-invar lss-α lss-invar lss-union-list*
proof
  fix *l* :: *'a*::{*linorder*} *lss list*
  assume $\forall$ *s1*$\in$*set l*. *lss-invar s1*

  with *merge-list-correct* [*of l* []]
  show *lss-α* (*lss-union-list l*) = $\bigcup$\{*lss-α s1* |*s1*. *s1* $\in$ *set l*\}
      lss-invar (*lss-union-list l*)
    by (*auto simp add*: *lss-defs*)
qed

lemma *lss-union-dj-impl*: *set-union-dj lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union-dj*
by (*unfold-locales*) (*auto simp add*: *lss-defs merge-correct*)

lemma *lss-image-filter-impl* : *set-image-filter lss-α lss-invar lss-α lss-invar lss-image-filter*
apply (*unfold-locales*)
apply (*simp-all add*: *lss-invar-def lss-image-filter-def mergesort-correct lss-α-def*
          set-map-filter Bex-def)
done

lemma *lss-inj-image-filter-impl* : *set-inj-image-filter lss-α lss-invar lss-α lss-invar lss-inj-image-filter*
apply (*unfold-locales*)
apply (*simp-all add*: *lss-invar-def lss-inj-image-filter-def lss-image-filter-def*
          mergesort-correct lss-α-def
          set-map-filter Bex-def)
done

lemma *lss-filter-impl* : *set-filter lss-α lss-invar lss-α lss-invar lss-filter*
apply (*unfold-locales*)
apply (*simp-all add*: *lss-invar-def lss-filter-def sorted-filter lss-α-def*
          set-map-filter Bex-def)
done

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*
by(*unfold-locales*)(*auto simp add*: *lss-defs*)

lemma *list-to-lss-impl*: *list-to-set lss-α lss-invar list-to-lss*
by(*unfold-locales*)(*auto simp add*: *lss-defs mergesort-correct*)

interpretation *lss*: *set-empty lss-α lss-invar lss-empty* **using** *lss-empty-impl* .
interpretation *lss*: *set-memb lss-α lss-invar lss-memb* **using** *lss-memb-impl* .
interpretation *lss*: *set-ins lss-α lss-invar lss-ins* **using** *lss-ins-impl* .
interpretation *lss*: *set-ins-dj lss-α lss-invar lss-ins-dj* **using** *lss-ins-dj-impl* .
interpretation *lss*: *set-delete lss-α lss-invar lss-delete* **using** *lss-delete-impl* .
interpretation *lss*: *finite-set lss-α lss-invar* **using** *lss-is-finite-set* .
interpretation *lss*: *set-iteratei lss-α lss-invar lss-iteratei* **using** *lss-iteratei-impl* .
interpretation *lss*: *set-isEmpty lss-α lss-invar lss-isEmpty* **using** *lss-isEmpty-impl*
.
interpretation *lss*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*
using *lss-union-impl* .
interpretation *lss*: *set-union-list lss-α lss-invar lss-α lss-invar lss-union-list* **using** *lss-union-list-impl* .
interpretation *lss*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
using *lss-inter-impl* .
interpretation *lss*: *set-union-dj lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union-dj*
using *lss-union-dj-impl* .
interpretation *lss*: *set-image-filter lss-α lss-invar lss-α lss-invar lss-image-filter*
using *lss-image-filter-impl* .
interpretation *lss*: *set-inj-image-filter lss-α lss-invar lss-α lss-invar lss-inj-image-filter*
using *lss-inj-image-filter-impl* .
interpretation *lss*: *set-filter lss-α lss-invar lss-α lss-invar lss-filter* **using** *lss-filter-impl*
.
interpretation *lss*: *set-image lss-α lss-invar lss-α lss-invar lss-image* **using** *lss-image-impl*
.
interpretation *lss*: *set-inj-image lss-α lss-invar lss-α lss-invar lss-inj-image* **using**
lss-inj-image-impl .
interpretation *lss*: *set-sel lss-α lss-invar lss-sel* **using** *lss-sel-impl* .
interpretation *lss*: *set-sel′ lss-α lss-invar lss-sel′* **using** *lss-sel′-impl* .
interpretation *lss*: *set-to-list lss-α lss-invar lss-to-list* **using** *lss-to-list-impl* .
interpretation *lss*: *list-to-set lss-α lss-invar list-to-lss* **using** *list-to-lss-impl* .

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* **using** *lss-min-impl* **.**

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* **using** *lss-max-impl* **.**

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* **using** *lss-disjoint-witness-impl* **.**

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* **using** *lss-ball-impl* **.**

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* **using** *lss-bexists-impl* **.**

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* **using** *lss-disjoint-impl* **.**

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* **using** *lss-size-impl* **.**

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* **using** *lss-size-abort-impl* **.**

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* **using** *lss-isSng-impl* **.**

definition *lss-sng x* = [*x*]
lemma *lss-sng-impl* : *set-sng lss-α lss-invar lss-sng*
  unfolding *set-sng-def lss-sng-def lss-invar-def lss-α-def*
  by *simp*
interpretation *lss*: *set-sng lss-α lss-invar lss-sng* **using** *lss-sng-impl* **.**

definition *lss-equal l1 l2* = (*l1* = *l2*)
lemma *lss-equal-impl* : *set-equal lss-α lss-invar lss-α lss-invar lss-equal*
  unfolding *set-equal-def lss-equal-def lss-α-def lss-invar-def*
  by (*simp add*: *set-eq-sorted-correct*)
interpretation *lss*: *set-equal lss-α lss-invar lss-α lss-invar lss-equal* **using** *lss-equal-impl* **.**

definition [*code-unfold*]: *lss-subset* = *subset-sorted*
lemma *lss-subset-impl* : *set-subset lss-α lss-invar lss-α lss-invar lss-subset*
  unfolding *set-subset-def lss-subset-def lss-α-def lss-invar-def*
  by (*simp add*: *subset-sorted-correct*)
interpretation *lss*: *set-subset lss-α lss-invar lss-α lss-invar lss-subset* **using** *lss-subset-impl* **.**

definition [*code-unfold*]: *lss-diff* = *diff-sorted*
lemma *lss-diff-impl* : *set-diff lss-*α *lss-invar lss-*α *lss-invar lss-diff*
unfolding *set-diff-def lss-diff-def lss-*α*-def lss-invar-def*
by (*simp add*: *diff-sorted-correct*)
interpretation *lss*: *set-diff lss-*α *lss-invar lss-*α *lss-invar lss-diff* **using** *lss-diff-impl*
.

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* $\Rightarrow$ *'a set* **where** *lsnd-*α == *set*
definition *lsnd-invar* :: *'a lsnd* $\Rightarrow$ *bool* **where** *lsnd-invar* == (λ-. *True*)
definition *lsnd-empty* :: *unit* $\Rightarrow$ *'a lsnd* **where** *lsnd-empty* == (λ-::*unit*. [])
definition *lsnd-memb* :: *'a* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *bool* **where** *lsnd-memb x l* == *List.member l x*
definition *lsnd-ins* :: *'a* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'a lsnd* **where** *lsnd-ins x l* == *x#l*
definition *lsnd-ins-dj* :: *'a* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'a lsnd* **where** *lsnd-ins-dj x l* == *x#l*

definition *lsnd-delete* :: *'a* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'a lsnd* **where** *lsnd-delete x l* == *remove-rev* [] *x l*

definition *lsnd-iteratei* :: *'a lsnd* $\Rightarrow$ (*'a*,*'σ*) *set-iterator*
where *lsnd-iteratei l* = *foldli* (*remdups l*)

definition *lsnd-isEmpty* :: *'a lsnd* $\Rightarrow$ *bool* **where** *lsnd-isEmpty s* == *s*=[]

definition *lsnd-union* :: *'a lsnd* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'a lsnd*
where *lsnd-union s1 s2* == *revg s1 s2*
definition *lsnd-inter* :: *'a lsnd* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'a lsnd*
where *lsnd-inter* == *it-inter lsnd-iteratei lsnd-memb lsnd-empty lsnd-ins-dj*
definition *lsnd-union-dj* :: *'a lsnd* $\Rightarrow$ *'a lsnd* $\Rightarrow$ *'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*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-memb-impl*: *set-memb lsnd-α lsnd-invar lsnd-memb*
by (*unfold-locales*)(*auto simp add*: *lsnd-defs in-set-member*)

lemma *lsnd-ins-impl*: *set-ins lsnd-α lsnd-invar lsnd-ins*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs in-set-member*)

lemma *lsnd-ins-dj-impl*: *set-ins-dj lsnd-α lsnd-invar lsnd-ins-dj*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-delete-impl*: *set-delete lsnd-α lsnd-invar lsnd-delete*
by (*unfold-locales*) (*auto simp add*: *lsnd-delete-def lsnd-α-def lsnd-invar-def remove-rev-alt-def*)

lemma *lsnd-α-finite*[*simp*, *intro*!]: *finite* (*lsnd-α l*)
by (*auto simp add*: *lsnd-defs*)

lemma *lsnd-is-finite-set*: *finite-set lsnd-α lsnd-invar*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-iteratei-impl*: *set-iteratei lsnd-α lsnd-invar lsnd-iteratei*
proof
fix *l* :: *'a list*
show *finite* (*lsnd-α l*)
unfolding *lsnd-α-def* **by** *simp*

show *set-iterator* (*lsnd-iteratei l*) (*lsnd-α l*)
apply (*rule set-iterator-I* [*of remdups l*])
apply (*simp-all add*: *lsnd-α-def lsnd-iteratei-def*)
done
qed

lemma *lsnd-isEmpty-impl*: *set-isEmpty lsnd-α lsnd-invar lsnd-isEmpty*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

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*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *lsnd-union-dj-impl*: *set-union-dj lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union-dj*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

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*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *list-to-lsnd-impl*: *list-to-set lsnd-*α *lsnd-invar list-to-lsnd*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

interpretation *lsnd*: *set-empty lsnd-*α *lsnd-invar lsnd-empty* **using** *lsnd-empty-impl*
.
interpretation *lsnd*: *set-memb lsnd-*α *lsnd-invar lsnd-memb* **using** *lsnd-memb-impl*
.
interpretation *lsnd*: *set-ins lsnd-*α *lsnd-invar lsnd-ins* **using** *lsnd-ins-impl* .
interpretation *lsnd*: *set-ins-dj lsnd-*α *lsnd-invar lsnd-ins-dj* **using** *lsnd-ins-dj-impl*
.
interpretation *lsnd*: *set-delete lsnd-*α *lsnd-invar lsnd-delete* **using** *lsnd-delete-impl*
.
interpretation *lsnd*: *finite-set lsnd-*α *lsnd-invar* **using** *lsnd-is-finite-set* .
interpretation *lsnd*: *set-iteratei lsnd-*α *lsnd-invar lsnd-iteratei* **using** *lsnd-iteratei-impl*
.
interpretation *lsnd*: *set-isEmpty lsnd-*α *lsnd-invar lsnd-isEmpty* **using** *lsnd-isEmpty-impl*
.
interpretation *lsnd*: *set-union lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-union* **using** *lsnd-union-impl* .
interpretation *lsnd*: *set-inter lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-inter* **using** *lsnd-inter-impl* .
interpretation *lsnd*: *set-union-dj lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-union-dj* **using** *lsnd-union-dj-impl* .
interpretation *lsnd*: *set-image-filter lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-image-filter* **using** *lsnd-image-filter-impl* .
interpretation *lsnd*: *set-inj-image-filter lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-inj-image-filter* **using** *lsnd-inj-image-filter-impl* .
interpretation *lsnd*: *set-image lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-image* **using** *lsnd-image-impl* .
interpretation *lsnd*: *set-inj-image lsnd-*α *lsnd-invar lsnd-*α *lsnd-invar lsnd-inj-image* **using** *lsnd-inj-image-impl* .
interpretation *lsnd*: *set-sel lsnd-*α *lsnd-invar lsnd-sel* **using** *lsnd-sel-impl* .
interpretation *lsnd*: *set-sel'* *lsnd-*α *lsnd-invar lsnd-sel'* **using** *lsnd-sel'-impl* .
interpretation *lsnd*: *set-to-list lsnd-*α *lsnd-invar lsnd-to-list* **using** *lsnd-to-list-impl*
.
interpretation *lsnd*: *list-to-set lsnd-*α *lsnd-invar list-to-lsnd* **using** *list-to-lsnd-impl*
.

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* **using** *lsnd-disjoint-witness-impl* **.**

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* **using** *lsnd-ball-impl* **.**

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* **using** *lsnd-bexists-impl*
.

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*
using *lsnd-disjoint-impl* .

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* **using** *lsnd-size-impl* .

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* **using** *lsnd-size-abort-impl*
.

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* **using** *lsnd-isSng-impl*
.

definition *lsnd-sng x* = [*x*]
lemma *lsnd-sng-impl* : *set-sng lsnd-α lsnd-invar lsnd-sng*
  unfolding *set-sng-def lsnd-sng-def lsnd-invar-def lsnd-α-def*
  by *simp*
interpretation *lsnd*: *set-sng lsnd-α lsnd-invar lsnd-sng* **using** *lsnd-sng-impl* .

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*
  using *lsnd-diff-impl* .

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*
  using *lsnd-subset-impl* .

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*
  using *lsnd-equal-impl* .

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*
  using *lsnd-filter-impl* **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *ls-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *ls-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *lsr*!: *set-iteratei* (*set-op-α ls-ops*) (*set-op-invar ls-ops*) *ls-iteratei*
using *ls-iteratei-impl*[*folded ls-ops-unfold*] **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *lsi-ops-def*)
apply *unfold-locales*
done

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*
by (*simp-all add*: *lsi-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *lsir*!: *set-iteratei* (*set-op-α lsi-ops*) (*set-op-invar lsi-ops*) *lsi-iteratei*
using *lsi-iteratei-impl*[*folded lsi-ops-unfold*] **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *lsnd-ops-def*)
apply *unfold-locales*

done

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*
by (*simp-all add*: *lsnd-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *lsndr*!: *set-iteratei* (*set-op-α lsnd-ops*) (*set-op-invar lsnd-ops*) *lsnd-iteratei* **using** *lsnd-iteratei-impl*[*folded lsnd-ops-unfold*] **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *lss-ops-def*)
apply *unfold-locales*
done

interpretation *lssr*!: *StdOSet lss-ops*
apply (*rule StdOSet.intro*)
apply (*rule StdSet.intro*)
apply (*simp-all add*: *lss-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *lss-ops-def*)

— Required to set up unfold_locales in contexts with additional iteratolss
interpretation *lssr*!: *set-iteratei* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-iteratei*
unfolding *lss-ops-unfold*
using *lss-iteratei-impl* .

interpretation *lssr*!: *set-iterateoi* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-iterateoi*
unfolding *lss-ops-unfold*
using *lss-iterateoi-impl* .

interpretation *lssr*!: *set-reverse-iterateoi* (*set-op-α lss-ops*) (*set-op-invar lss-ops*) *lss-reverse-iterateoi*
unfolding *lss-ops-unfold*
using *lss-reverse-iterateoi-impl* .

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *rs-ops-def*)
apply *unfold-locales*
done

interpretation *rsr*!: *StdOSet rs-ops*
apply (*rule StdOSet.intro*)
apply (*rule StdSet.intro*)
apply (*simp-all add*: *rs-ops-def*)
apply *unfold-locales*
done

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
by (*auto simp add*: *rs-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *rsr*!: *set-iteratei* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-iteratei*
unfolding *rs-ops-unfold*
using *rs-iteratei-impl* **.**

interpretation *rsr*!: *set-iterateoi* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-iterateoi*
unfolding *rs-ops-unfold*
using *rs-iterateoi-impl* **.**

interpretation *rsr*!: *set-reverse-iterateoi* (*set-op-α rs-ops*) (*set-op-invar rs-ops*) *rs-reverse-iterateoi*
unfolding *rs-ops-unfold*
using *rs-reverse-iterateoi-impl* **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *hs-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *hs-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *hsr*!: *set-iteratei* (*set-op-*α *hs-ops*) (*set-op-invar hs-ops*) *hs-iteratei*
using *hs-iteratei-impl*[*folded hs-ops-unfold*] **.**

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*
apply (*rule StdSet.intro*)
apply (*simp-all add*: *ahs-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *ahs-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *ahsr*!: *set-iteratei* (*set-op-α ahs-ops*) (*set-op-invar ahs-ops*) *ahs-iteratei* **using** *ahs-iteratei-impl*[*folded ahs-ops-unfold*] **.**

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*

apply (*rule StdSet.intro*)
apply (*simp-all add*: *ias-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *ias-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *iasr*!: *set-iteratei* (*set-op-*α *ias-ops*) (*set-op-invar ias-ops*) *ias-iteratei* **using** *ias-iteratei-impl*[*folded ias-ops-unfold*] **.**

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*
apply (*rule StdMap.intro*)
apply (*simp-all add*: *hm-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *hm-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *hmr*!: *map-iteratei* (*map-op-α hm-ops*) (*map-op-invar hm-ops*) *hm-iteratei*
using *hm-iteratei-impl*[*folded hm-ops-unfold*] **.**

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*
apply (*rule StdMap.intro*)
apply (*simp-all add*: *rm-ops-def*)
apply *unfold-locales*
done

interpretation *rmr*!: *StdOMap rm-ops*
apply (*rule StdOMap.intro*)
apply (*rule StdMap.intro*)
apply (*simp-all add*: *rm-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *rm-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators

interpretation *rmr*!: *map-iteratei* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-iteratei*
unfolding *rm-ops-unfold*
using *rm-iteratei-impl* **.**

interpretation *rmr*!: *map-iterateoi* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-iterateoi*
unfolding *rm-ops-unfold*
using *rm-iterateoi-impl* **.**

interpretation *rmr*!: *map-reverse-iterateoi* (*map-op-α rm-ops*) (*map-op-invar rm-ops*) *rm-reverse-iterateoi*
unfolding *rm-ops-unfold*
using *rm-reverse-iterateoi-impl* **.**

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*
apply (*rule StdMap.intro*)
apply (*simp-all add*: *lm-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *lm-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators

interpretation *lmr*!: *map-iteratei* (*map-op-α lm-ops*) (*map-op-invar lm-ops*) *lm-iteratei*
using *lm-iteratei-impl*[*folded lm-ops-unfold*] **.**

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*
apply (*rule StdMap.intro*)
apply (*simp-all add*: *ahm-ops-def*)
apply *unfold-locales*
done

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*
by (*auto simp add*: *ahm-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *ahmr*!: *map-iteratei* (*map-op-α ahm-ops*) (*map-op-invar ahm-ops*) *ahm-iteratei*
using *ahm-iteratei-impl*[*folded ahm-ops-unfold*] **.**

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*
apply (*rule StdMap.intro*)
apply (*simp-all add*: *iam-ops-def*)
apply *unfold-locales*
done

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
by (*auto simp add*: *iam-ops-def*)

— Required to set up unfold_locales in contexts with additional iterators
interpretation *iamr*!: *map-iteratei* (*map-op-α iam-ops*) (*map-op-invar iam-ops*) *iam-iteratei*
using *iam-iteratei-impl*[*folded iam-ops-unfold*] **.**

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*:
$\llbracket$ *s*∈*wa-invar WA*; *s*∈*wa-cond WA*; (*s*,*s'*)∈*wa-step WA* $\rrbracket \Longrightarrow$ *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 α :: *'C* ⇒ *'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*: α ‘ $wa\text{-}initial\ WAC \subseteq wa\text{-}initial\ WAA$

— 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*: α ‘ $wa\text{-}invar\ WAC \subseteq wa\text{-}invar\ WAA$
begin

lemma *initial-abs'*: $s{\in}wa\text{-}initial\ WAC \Longrightarrow \alpha\ s \in wa\text{-}initial\ WAA$
using *initial-abs* **by** *auto*

lemma *invar-abs'*: $s{\in}wa\text{-}invar\ WAC \Longrightarrow \alpha\ s \in wa\text{-}invar\ WAA$
using *invar-abs* **by** *auto*

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 α :: $'C \Rightarrow 'A$
assumes *while-algo WAA*

— The concrete step preserves the concrete part of the invariant
assumes *step-invarc*:
  !!s s'. ⟦ s∈*invar-addc*; s∈*condc*; α s ∈ *wa-invar WAA*; (s,s')∈*stepc* ⟧
      ⟹ s'∈*invar-addc*
— The concrete initial states satisfy the concrete part of the invariant
assumes *initial-invarc*: *initialc* ⊆ *invar-addc*

— Condition implemented correctly
assumes *cond-abs*:
  !!s. ⟦ s∈*invar-addc*; α s ∈ *wa-invar WAA*; s∈*condc* ⟧ ⟹ α s ∈ *wa-cond WAA*
— Step implemented correctly
assumes *step-abs*:
  !!s s'. ⟦ s∈*invar-addc*; s∈*condc*; α s ∈ *wa-invar WAA*; (s,s')∈*stepc* ⟧
      ⟹ (α s, α s')∈*wa-step WAA*
— Initial states implemented correctly
assumes *initial-abs*: α ‘ *initialc* ⊆ *wa-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-addc* ∩ {s. α s∈ *wa-invar WAA*}) |)

shows
  while-algo WAC ∧
   wa-refine WAC WAA α (**is** *?T1* ∧ *?T2*)
proof
 interpret *waa*: *while-algo WAA* **by** *fact*
 show *G1*: *?T1*
  apply (*unfold-locales*)
  apply (*simp-all add*: *WAC-def*)
  apply *safe*
  apply (*blast intro*!: *step-invarc*)

  apply (*frule* (*3*) *step-abs*)
  apply (*frule* (*2*) *cond-abs*)
  apply (*erule* (*2*) *waa.step-invar*)

  apply (*erule set-rev-mp*[*OF* - *initial-invarc*])

  apply (*insert initial-abs waa.initial-invar*) [*1*]
  apply *blast*

  apply (*rule-tac*
   r=*inv-image* { (s',s). s∈*wa-invar WAA*
         ∧ s∈*wa-cond WAA*
         ∧ (s,s')∈*wa-step WAA* } α

    in *wf-subset*)
  apply (*simp add*: *waa.step-wf*)
  apply (*auto simp add*: *cond-abs step-abs*) [*1*]
  done
show *?T2*
  apply (*unfold-locales*)
  apply (*auto simp add*: *cond-abs step-abs initial-abs WAC-def*)
  done
qed

— After refinement has been shown, this lemma transfers the well-definedness property up the refinement chain. Like in *wa-refine-intro*, some proof-obligations can be discharged by assuming refinement and well-definedness of the abstract algorithm.

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*: $wa\text{-}invar\ WAC = addi \cap \{s.\ \alpha\ s \in wa\text{-}invar\ WAA\}$

— 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*: $wa\text{-}initial\ WAC \subseteq addi$

shows
  while-algo WAC
proof −
interpret *waa*: *while-algo WAA* **by** *fact*
show *?thesis*
  apply (*unfold-locales*)
  apply (*subst icf*)
  apply *safe*
  apply (*simp only*: *icf*)
  apply *safe*
  apply (*blast intro*!: *step-addi*)

  apply (*frule* (*2*) *step-abs*)
  apply (*frule* (*1*) *cond-abs*)
  apply (*simp only*: *icf*)
  apply *clarify*
  apply (*erule* (*2*) *waa.step-invar*)

```
  apply (simp add: icf)
  apply (rule conjI)
  apply (erule set-rev-mp[OF - initial-addi])
  apply (insert initial-abs waa.initial-invar) [1]
  apply blast

  apply (rule-tac
    r=inv-image { (s',s). s∈wa-invar WAA
                 ∧ s∈wa-cond WAA
                 ∧ (s,s')∈wa-step WAA } α
    in wf-subset)
  apply (simp add: waa.step-wf)
  apply (auto simp add: cond-abs step-abs icf) [1]
  done
qed
```

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 α :: 'C ⇒ 'A
  assumes cond-precise:
    ∀ s. s∈wa-invar WAC ∧ α s∈wa-cond WAA ⟶ s∈wa-cond WAC
begin
  — Transfer correctness property
  lemma transfer-correctness:
    assumes A: ∀ s. s∈wa-invar WAA ∧ s∉wa-cond WAA ⟶ P s
    shows ∀ sc. sc∈wa-invar WAC ∧ sc∉wa-cond WAC ⟶ P (α sc)
    using A cond-abs invar-abs cond-precise by blast
end
```

Refinement as well as precise refinement is reflexive and transitive

```
lemma wa-ref-refl: wa-refine WA WA id
  by (unfold-locales) auto

lemma wa-pref-refl: wa-precise-refine WA WA id
  by (unfold-locales) auto

lemma wa-ref-trans:
  assumes wa-refine WC WB α1
  assumes wa-refine WB WA α2
  shows wa-refine WC WA (α2∘α1)
proof -
  interpret r1: wa-refine WC WB α1 by fact
  interpret r2: wa-refine WB WA α2 by fact

  show ?thesis
```

```
    apply unfold-locales
    apply (auto simp add:
      r1.invar-abs' r2.invar-abs'
      r1.cond-abs r2.cond-abs
      r1.step-abs r2.step-abs
      r1.initial-abs' r2.initial-abs')
    done
qed

lemma wa-pref-trans:
  assumes wa-precise-refine WC WB α1
  assumes wa-precise-refine WB WA α2
  shows wa-precise-refine WC WA (α2∘α1)
proof -
  interpret r1: wa-precise-refine WC WB α1 by fact
  interpret r2: wa-precise-refine WB WA α2 by fact

  show ?thesis
    apply intro-locales
    apply (rule wa-ref-trans)
    apply (unfold-locales)
    apply (auto simp add: r1.invar-abs' r2.invar-abs'
                     r1.cond-precise r2.cond-precise)
    done
qed
```

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*)
apply (*unfold-locales*)
apply (*auto simp add*: *det-wa-wa-def step-invar initial-invar*)
apply (*insert step-wf*)
apply (*erule-tac P*=*wf* **in** *back-subst*)
apply *auto*
done

end

lemma *det-while-algo-intro*:
assumes *while-algo* (*det-wa-wa DWA*)
shows *det-while-algo DWA*
proof −
interpret *while-algo* (*det-wa-wa DWA*) **by** *fact*

show *?thesis* **using** *step-invar initial-invar step-wf*
apply (*unfold-locales*)
apply (*unfold det-wa-wa-def*)
apply *auto*
apply (*erule-tac P*=*wf* **in** *back-subst*)
apply *auto*
done

qed

— A deterministic while-algorithm is well-defined, if and only if the corresponding non-deterministic while-algorithm is well-defined
theorem *dwa-is-wa*:
while-algo (*det-wa-wa DWA*) ⟷ *det-while-algo DWA*
using *det-while-algo-intro det-while-algo*.*is-while-algo* **by** *auto*

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*
apply (*unfold loop-def*)
apply (*rule-tac* $P{=}\lambda x.\ x{\in}dwa\text{-}invar\ WA$ **and**
$r{=}\{\ (dwa\text{-}step\ WA\ s{,}s)\ |\ s.\ s{\in}dwa\text{-}invar\ WA \wedge dwa\text{-}cond\ WA\ s\ \}$
in *while-rule*)
apply (*simp-all add*: *step-invar initial-invar step-wf inv-imp*)
done

— This version is useful when using transferred correctness lemmas
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*
using *inv-imp*
apply (*simp add*: *det-wa-wa-def*)
apply (*blast intro*: *while-proof*)
done

lemma (**in** *det-while-algo*) *loop-invar*:
$loop \in dwa\text{-}invar\ WA$
by (*rule while-proof*) *simp*

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*]
  by (*simp add*: *rbt-restrict-list-def rs.memb-correct*)

The lemma *rs.memb-correct*:

$True \Longrightarrow rs\text{-}memb\ x\ s = (x \in rs\text{-}\alpha\ s)$

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*:

$[\![\mathit{True};\ \mathit{inj\text{-}on}\ f\ (\mathit{rs\text{-}}\alpha\ s \cap \mathit{dom}\ f)]\!]$
$\Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}inj\text{-}image\text{-}filter}\ f\ s) = \{b.\ \exists\, a{\in}\mathit{rs\text{-}}\alpha\ s.\ f\ a = \mathit{Some}\ b\}$
$[\![\mathit{True};\ \mathit{inj\text{-}on}\ f\ (\mathit{rs\text{-}}\alpha\ s \cap \mathit{dom}\ f)]\!] \Longrightarrow \mathit{True}$
$\mathit{True} \Longrightarrow$
$\mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}image\text{-}filter}\ (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ \mathit{Some}\ (f\ x)\ \mathit{else}\ \mathit{None})\ s) =$
$f\ \text{`}\ \{x \in \mathit{rs\text{-}}\alpha\ s.\ P\ x\}$
$\mathit{True} \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}image\text{-}filter}\ f\ s) = \{b.\ \exists\, a{\in}\mathit{rs\text{-}}\alpha\ s.\ f\ a = \mathit{Some}\ b\}$
$\mathit{True} \Longrightarrow \mathit{True}$
$[\![\mathit{True};\ \mathit{inj\text{-}on}\ f\ (\mathit{rs\text{-}}\alpha\ s)]\!] \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}inj\text{-}image}\ f\ s) = f\ \text{`}\ \mathit{rs\text{-}}\alpha\ s$
$[\![\mathit{True};\ \mathit{inj\text{-}on}\ f\ (\mathit{rs\text{-}}\alpha\ s)]\!] \Longrightarrow \mathit{True}$
$[\![\mathit{True};\ \mathit{True};\ \mathit{rs\text{-}}\alpha\ s1 \cap \mathit{rs\text{-}}\alpha\ s2 = \{\}]\!]$
$\Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}union\text{-}dj}\ s1\ s2) = \mathit{rs\text{-}}\alpha\ s1 \cup \mathit{rs\text{-}}\alpha\ s2$
$[\![\mathit{True};\ \mathit{True};\ \mathit{rs\text{-}}\alpha\ s1 \cap \mathit{rs\text{-}}\alpha\ s2 = \{\}]\!] \Longrightarrow \mathit{True}$
$[\![\mathit{True};\ \mathit{True}]\!] \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}union}\ s1\ s2) = \mathit{rs\text{-}}\alpha\ s1 \cup \mathit{rs\text{-}}\alpha\ s2$
$[\![\mathit{True};\ \mathit{True}]\!] \Longrightarrow \mathit{True}$
$[\![\mathit{True};\ \mathit{True}]\!] \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}inter}\ s1\ s2) = \mathit{rs\text{-}}\alpha\ s1 \cap \mathit{rs\text{-}}\alpha\ s2$
$[\![\mathit{True};\ \mathit{True}]\!] \Longrightarrow \mathit{True}$
$\mathit{True} \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}image}\ f\ s) = f\ \text{`}\ \mathit{rs\text{-}}\alpha\ s$
$\mathit{True} \Longrightarrow \mathit{True}$
$[\![\mathit{True};\ x \notin \mathit{rs\text{-}}\alpha\ s]\!] \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}ins\text{-}dj}\ x\ s) = \mathit{insert}\ x\ (\mathit{rs\text{-}}\alpha\ s)$
$[\![\mathit{True};\ x \notin \mathit{rs\text{-}}\alpha\ s]\!] \Longrightarrow \mathit{True}$
$\mathit{True} \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}delete}\ x\ s) = \mathit{rs\text{-}}\alpha\ s - \{x\}$
$\mathit{True} \Longrightarrow \mathit{True}$
$\mathit{True} \Longrightarrow \mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}ins}\ x\ s) = \mathit{insert}\ x\ (\mathit{rs\text{-}}\alpha\ s)$
$\mathit{True} \Longrightarrow \mathit{True}$
$\mathit{True} \Longrightarrow \mathit{rs\text{-}memb}\ x\ s = (x \in \mathit{rs\text{-}}\alpha\ s)$
$\mathit{True} \Longrightarrow \mathit{set}\ (\mathit{rs\text{-}to\text{-}list}\ s) = \mathit{rs\text{-}}\alpha\ s$
$\mathit{True} \Longrightarrow \mathit{distinct}\ (\mathit{rs\text{-}to\text{-}list}\ s)$
$\mathit{rs\text{-}}\alpha\ (\mathit{list\text{-}to\text{-}rs}\ l) = \mathit{set}\ l$
True
$\mathit{True} \Longrightarrow \mathit{rs\text{-}isEmpty}\ s = (\mathit{rs\text{-}}\alpha\ s = \{\})$
$\mathit{rs\text{-}}\alpha\ (\mathit{rs\text{-}empty}\ ()) = \{\}$
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* $[1{::}\mathit{int}..10]$
value *rs-to-list* $(\mathit{list\text{-}to\text{-}rs}\ [1{::}\mathit{int}\ ..\ 10])$
value *rs-to-list* $(\mathit{list\text{-}to\text{-}rs}\ [1{::}\mathit{int},5,6,7,3,4,9,8,2,7,6])$

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-val ⟪ @{*code test*} *9000* ⟫

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*:

$[\![$*True*; I (*rs*-α S) $\sigma 0$;
$\bigwedge x$ *it* σ. $[\![x \in it$; $it \subseteq$ *rs*-α S; I *it* $\sigma]\!] \Longrightarrow I$ (*it* $-$ $\{x\}$) (f x σ);
$\bigwedge \sigma$. I $\{\}$ $\sigma \Longrightarrow P$ $\sigma]\!]$
$\Longrightarrow P$ (*rs-iteratei* S (λ-. *True*) f $\sigma 0$)

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 (*rs*-α S) $\sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

$\bigwedge x$ *it* σ. $[\![x \in it$; $it \subseteq$ *rs*-α S; I *it* $\sigma]\!] \Longrightarrow I$ (*it* $-$ $\{x\}$) (f x σ)

And the proposition to be shown for the final state must be a consequence of the invarant for no elements remaining: $\bigwedge \sigma$. I $\{\}$ $\sigma \Longrightarrow P$ σ.

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*:

$[\![$*True*; I (*rs*-α S) $\sigma 0$;
$\bigwedge x$ *it* σ. $[\![c$ σ; $x \in it$; $it \subseteq$ *rs*-α S; I *it* $\sigma]\!] \Longrightarrow I$ (*it* $-$ $\{x\}$) (f x σ);
$\bigwedge \sigma$. I $\{\}$ $\sigma \Longrightarrow P$ σ; $\bigwedge \sigma$ *it*. $[\![it \subseteq$ *rs*-α S; $it \neq \{\}$; $\neg$ c σ; I *it* $\sigma]\!] \Longrightarrow P$ $\sigma]\!]$
$\Longrightarrow P$ (*rs-iteratei* S c f $\sigma 0$)

Here, interruption of the iteration is handled by the premise

$\bigwedge \sigma$ *it*. $[\![it \subseteq$ *rs*-α S; $it \neq \{\}$; $\neg$ c σ; I *it* $\sigma]\!] \Longrightarrow P$ σ

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

apply (*unfold hs-to-list′-def*)
apply (*rule-tac*
  $I{=}\lambda it\ \sigma.\ set\ \sigma = hs\text{-}\alpha\ s - it$
  in *hs.iterate-rule-P*[*OF INV*])

The resulting proof obligations are easily discharged using auto:

apply *auto*
done

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 $hs\text{-}bex\ s\ P == hs\text{-}iteratei\ s\ (\lambda\sigma.\ \neg\ \sigma)\ (\lambda x\ \sigma.\ P\ x)\ False$

lemma *hs-bex-correct*:
  $hs\text{-}invar\ s \Longrightarrow hs\text{-}bex\ s\ P \longleftrightarrow (\exists\, x{\in}hs\text{-}\alpha\ s.\ P\ x)$
  apply (*unfold hs-bex-def*)

The invariant states that the current result matches the result of the quantification over the elements already iterated over:

apply (*rule-tac*
  $I{=}\lambda it\ \sigma.\ \sigma \longleftrightarrow (\exists\, x{\in}hs\text{-}\alpha\ s - it.\ P\ x)$
  in *hs.iteratei-rule-P*)

The resulting proof obligations are easily discharged by auto:

apply *auto*
done

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* $\Rightarrow$ *'a set*
— Invariant
fixes *invar* :: *'s* $\Rightarrow$ *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 α :: *'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 α :: $'s \Rightarrow 'a\ set$
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 $\alpha 2$ :: $'s2 \Rightarrow 'a\ set$ **and** *invar2*
and $\alpha 3$ :: $'s3 \Rightarrow 'a\ set$ **and** *invar3*
+
fixes *inter* :: $'s1 \Rightarrow 's2 \Rightarrow 's3$
assumes *inter-correct*:
$[\![invar1\ s1;\ invar2\ s2]\!] \Longrightarrow \alpha 3\ (inter\ s1\ s2) = \alpha 1\ s1 \cap \alpha 2\ s2$
$[\![invar1\ s1;\ invar2\ s2]\!] \Longrightarrow invar3\ (inter\ s1\ s2)$

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 :: $'s \Rightarrow 'a\ set$
fixes *list-to-set* :: $'a\ list \Rightarrow 's$
assumes *list-to-set-correct*:
$\alpha\ (list\text{-}to\text{-}set\ l) = set\ l$
$invar\ (list\text{-}to\text{-}set\ l)$

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′* :: $'s \Rightarrow ('a \Rightarrow 's \Rightarrow 's)$
$\Rightarrow$ *'a list* $\Rightarrow$ *'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 –
interpret *set′-empty* α *invar empty* **by** *fact*
interpret *set′-ins* α *invar ins* **by** *fact*

{
fix *l*
have α (*list-to-set′ empty ins l*) = *set l*
$\wedge$ *invar* (*list-to-set′ empty ins l*)
by (*induct l*)
(*simp-all add*: *empty-correct ins-correct*)
}
thus *?thesis*
by *unfold-locales auto*
qed

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* $\alpha 1$ *invar1 iterate1*; *map-empty* $\alpha 2$ *invar2 empty2*;
map-update $\alpha 2$ *invar2 update2*; *invar1 s*$]\!]$
$\Longrightarrow$ *dom* ($\alpha 2$ (*map-to-nat iterate1 empty2 update2 s*)) = $\alpha 1$ *s*
$[\![$*set-iteratei* $\alpha 1$ *invar1 iterate1*; *map-empty* $\alpha 2$ *invar2 empty2*;
map-update $\alpha 2$ *invar2 update2*; *invar1 s*$]\!]$
$\Longrightarrow$ *inj-on* ($\alpha 2$ (*map-to-nat iterate1 empty2 update2 s*)) ($\alpha 1$ *s*)

[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

$[\![$*set-iteratei* $\alpha 1$ *invar1 iterate1*; *map-empty* $\alpha 2$ *invar2 empty2*;
map-update $\alpha 2$ *invar2 update2*; *invar1 s*$]\!]$
$\Longrightarrow$ *inatseg* (*ran* ($\alpha 2$ (*map-to-nat iterate1 empty2 update2 s*)))
$[\![$*set-iteratei* $\alpha 1$ *invar1 iterate1*; *map-empty* $\alpha 2$ *invar2 empty2*;
map-update $\alpha 2$ *invar2 update2*; *invar1 s*$]\!]$
$\Longrightarrow$ *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* **.**

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*
by (*unfold-locales*) (*auto simp add*: *lsi'-empty-def lsi'-invar-def lsi'*-α-*def*)

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*
using *lsi'-empty-impl* **.**

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*)
by (*auto simp add*: *lsi'-ins-def lsi'-invar-def lsi'-α-def*)

lemma *lsi'-ins-impl*: *set'-ins lsi'-α lsi'-invar lsi'-ins*
by *unfold-locales*
(*simp-all add*: *lsi'-ins-correct-aux*)

interpretation *lsi'*: *set'-ins lsi'-α lsi'-invar lsi'-ins*
using *lsi'-ins-impl* **.**

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*
using *lsi'-list-to-set-impl* **.**

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*
  by (*induct l arbitrary*: *s*) *auto*

— Generic algorithm
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*)
by (*induct l arbitrary*: *s*) (*auto simp add*: *correct*)

— Instantiation

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-val ⟨⟨ @{*code hs-seti*} ⟩⟩

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*
by (*induct l*) *auto*

— Generic algorithm
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*
by (*induct l*) (*auto simp add*: *correct*)

— Instantiation
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-val ⟨⟨ @{*code hs-seti2*} ⟩⟩

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*)
by (*induct l arbitrary*: *s*) (*auto simp add*: *correct*)

— Instantiation
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-val ⟪ @{*code hs-seti′*} ⟫

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) (λx (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) )
    apply (unfold avg-aux-def)
    apply (rule-tac
      I=λit (c,sum). c=card (α s − it) ∧ sum=∑(α s − it)
      in it.iterate-rule-P)
    apply auto
    apply (subgoal-tac α s − (it − {x}) = insert x (α s − it))
    apply auto
    apply (subgoal-tac α s − (it − {x}) = insert x (α s − it))
    apply auto
    done

  lemma avg-correct: invar s ⟹ avg s = average (α s)
    unfolding avg-def
    using avg-aux-correct
    by auto

  lemma filter-le-avg-correct:
    invar s ⟹
      invar (filter-le-avg s) ∧
      α (filter-le-avg s) = {x∈α s. x≤average (α s)}
    unfolding filter-le-avg-def Let-def
    apply (rule-tac
      I=λit r. invar r ∧ α r = {x∈α s − it. x≤average (α s)}
      in it'.iterate-rule-P)
    apply (auto simp add: correct avg-correct)
    done
end

interpretation hs-ctx: MyContext hs-ops hs-iteratei hs-iteratei
```

by *unfold-locales*

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*
by *unfold-locales*

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-val ⟪ @{*code hs-filter-le-avg*} @{*code test-hs*} ⟫

— Using abbreviations

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*) (λx (*c*,*s*). (*c*+*1*, *s*+*x*)) (*0*,*0*)

definition *avg s* == *case avg-aux s of* (*c*,*s*) $\Rightarrow$ *s div c*

definition *filter-le-avg s* == *let a*=*avg s in*
my-iterate s (λ-. *True*) (λx *s*. *if* $x \leq a$ *then my-ins x s else s*) (*my-empty* ())

lemma *avg-aux-correct*: *my-invar s* $\Longrightarrow$ *avg-aux s* = (*card* (*my-*α *s*), $\sum$(*my-*α *s*))
apply (*unfold avg-aux-def*)
apply (*rule-tac*
I=λ*it* (*c*,*sum*). *c*=*card* (*my-*α *s* − *it*) $\wedge$ *sum*=$\sum$(*my-*α *s* − *it*)
in *my-iterate-rule-P*)
apply *auto*
apply (*subgoal-tac my-*α *s* − (*it* − {*x*}) = *insert x* (*my-*α *s* − *it*))
apply *auto*
apply (*subgoal-tac my-*α *s* − (*it* − {*x*}) = *insert x* (*my-*α *s* − *it*))
apply *auto*
done

lemma *avg-correct*: *my-invar s* $\Longrightarrow$ *avg s* = *average* (*my-*α *s*)
unfolding *avg-def*
using *avg-aux-correct*
by *auto*

lemma *filter-le-avg-correct*:
my-invar s $\Longrightarrow$
my-invar (*filter-le-avg s*) $\wedge$
my-α (*filter-le-avg s*) = {*x*$\in$*my-*α *s*. *x*$\leq$*average* (*my-*α *s*)}
unfolding *filter-le-avg-def Let-def*
apply (*rule-tac*
I=λ*it r*. *my-invar r* $\wedge$ *my-*α *r* = {*x*$\in$*my-*α *s* − *it*. *x*$\leq$*average* (*my-*α *s*)}
in *my-iterate-rule-P*)
apply (*auto simp add*: *my-correct avg-correct*)
done

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* $\times$ $'\Sigma$ *set*

— Loop body
inductive-set
sse-step :: $('\Sigma\times'\Sigma)$ *set* $\Rightarrow$ $('\Sigma$ *sse-state* $\times$ $'\Sigma$ *sse-state*$)$ *set*
for R **where**
$[\![\ \sigma\in W;$
$\Sigma' = \Sigma \cup (R``\{\sigma\});$
$W' = (W-\{\sigma\}) \cup ((R``\{\sigma\}) - \Sigma)$
$]\!] \Longrightarrow ((\Sigma,W),(\Sigma',W'))\in$*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* $\Rightarrow$ $'\Sigma$ *sse-state* **where**
sse-initial $\Sigma i == (\Sigma i,\Sigma i)$

— 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 $\Sigma i\ R = \{(\Sigma,W).$
$W\subseteq\Sigma\ \wedge$
$(\Sigma \subseteq R^*``\Sigma i)\ \wedge$

$(\forall\,\sigma{\in}(R^*\,``\Sigma i){-}\Sigma.\ \exists\,\sigma h{\in}W.\ (\sigma h,\sigma){\in}(R - (\mathit{UNIV} \times \Sigma))^*)$
}

definition *sse-algo* $\Sigma i\ R$ ==
(| *wa-cond=sse-cond*,
wa-step=sse-step R,
wa-initial = {*sse-initial* Σi},
wa-invar = *sse-invar* $\Sigma i\ R$ |)

definition *sse-term-rel* $\Sigma i\ R$ ==
{ (σ',σ). $\sigma{\in}$*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 −
have *wf* $((\{(\Sigma',\Sigma).\ \Sigma \subset \Sigma' \wedge \Sigma' \subseteq (R^*\,``\Sigma i)\})$ <*lex*> *finite-psubset*)
by (*auto intro*: *wf-bounded-supset*)
moreover have *sse-term-rel* $\Sigma i\ R \subseteq \ldots$ (**is** - $\subseteq$ *?R*)
proof
fix S
assume A: $S{\in}$*sse-term-rel* $\Sigma i\ R$
obtain $\Sigma\ W\ \Sigma'\ W'\ \sigma$ **where**
[*simp*]: $S{=}((\Sigma',W'),(\Sigma,W))$ **and**
S: $(\Sigma,W) \in$ *sse-invar* $\Sigma i\ R$
$\sigma{\in}W$
$\Sigma' = \Sigma \cup R``\{\sigma\}$
$W' = (W{-}\{\sigma\}) \cup (R``\{\sigma\} - \Sigma)$
proof −
obtain $\Sigma\ W\ \Sigma'\ W'$ **where** *SF*[*simp*]: $S{=}((\Sigma',W'),(\Sigma,W))$ **by** (*cases* S) *force*
from A **have** R: $(\Sigma,W) \in$ *sse-invar* $\Sigma i\ R$ $\quad ((\Sigma,W),(\Sigma',W')){\in}$*sse-step* R
by (*auto simp add*: *sse-term-rel-def*)
from *sse-step.cases*[*OF* $R(2)$] **obtain** σ **where** S:
$\sigma{\in}W$
$\Sigma' = \Sigma \cup R``\{\sigma\}$
$W' = (W{-}\{\sigma\}) \cup (R``\{\sigma\} - \Sigma)$
by *metis*
thus *?thesis*
by (*rule-tac that*[*OF SF* $R(1)$ S])
qed

from $S(1)$ **have**
[*simp*, *intro*!]: *finite* Σ $\quad$ *finite* W **and**
WSS: $W{\subseteq}\Sigma$ **and**
SSS: $\Sigma{\subseteq}R^*\,``\Sigma i$
by (*auto simp add*: *sse-invar-def intro*: *finite-subset*)

show $S{\in}$*?R* **proof** (*cases* $R``\{\sigma\} \subseteq \Sigma$)

case *True* **with** S **have** $\Sigma'{=}\Sigma \quad W' \subset W$ **by** *auto*
thus *?thesis* **by** (*simp*)
next
case *False*
with S **have** $\Sigma' \supset \Sigma$ **by** *auto*
moreover from $S(2)$ *WSS SSS* **have** $\sigma{\in}R^*$ ``Σi **by** *auto*
hence R``$\{\sigma\} \subseteq R^*$``Σi
by (*auto intro*: *rtrancl-into-rtrancl*)
with $S(3)$ *SSS* **have** $\Sigma' \subseteq R^*$``Σi **by** *auto*
ultimately show *?thesis* **by** *simp*
qed
qed
ultimately show *?thesis* **by** (*auto intro*: *wf-subset*)
qed

lemma *sse-invar-initial*: (*sse-initial* Σi) $\in$ *sse-invar* Σi R
by (*unfold sse-invar-def sse-initial-def*)
(*auto elim*: *rtrancl-last-touch*)

— Correctness theorem: If the loop terminates, the discovered states are exactly the reachable states
theorem *sse-invar-final*:
$\forall S.$ $S{\in}$*wa-invar* (*sse-algo* Σi R) $\wedge$ $S{\notin}$*wa-cond* (*sse-algo* Σi R)
$\longrightarrow$ *fst* $S = R^*$``Σi
by (*intro allI*, *case-tac* S)
(*auto simp add*: *sse-invar-def sse-cond-def sse-algo-def*)

lemma *sse-invar-step*: $[\![S{\in}$*sse-invar* Σi R; $(S,S'){\in}$*sse-step* $R]\!]$
$\Longrightarrow S'{\in}$*sse-invar* Σi R
— Split the goal by the invariant:
apply (*cases* S, *cases* S')
apply *clarsimp*
apply (*erule sse-step.cases*)
apply *clarsimp*
apply (*subst sse-invar-def*)
apply (*simp add*: *Let-def split-conv*)
apply (*intro conjI*)
— Solve the easy parts automatically
apply (*auto simp add*: *sse-invar-def*) [*3*]
apply (*force simp add*: *sse-invar-def*
dest: *rtrancl-into-rtrancl*) [*1*]
— Tackle the complex part (last part of the invariant) in Isar
proof (*intro ballI*)
fix σ W Σ σ'
assume A:
$(\Sigma, W){\in}$*sse-invar* Σi R
$\sigma{\in}W$
$\sigma'{\in}R^*$ `` $\Sigma i - (\Sigma \cup R$ `` $\{\sigma\})$
— Using the invariant of the original state, we obtain a state in the original

workset and a path not touching the originally discovered states
from $A(3)$ **have** $\sigma' \in R^*$ `` $\Sigma i - \Sigma$ **by** *auto*
with $A(1)$ **obtain** σh **where** *IP*:
$\sigma h \in W$
$(\sigma h, \sigma') \in (R - (UNIV \times \Sigma))^*$
and *SS*:
$W \subseteq \Sigma$
$\Sigma \subseteq R^*$ `` Σi
by (*unfold sse-invar-def*) *force*

— We now make a case distinction, whether the obtained path contains states from *post* σ or not:
from $IP(2)$ **show** $\exists \sigma h \in W - \{\sigma\} \cup (R$ `` $\{\sigma\} - \Sigma)$.
$(\sigma h, \sigma') \in (R - UNIV \times (\Sigma \cup R$ `` $\{\sigma\}))^*$
proof (*cases rule*: *rtrancl-last-visit*[**where** $S{=}R$ `` $\{\sigma\}$])
case *no-visit*
— In the case that the obtained path contains no states from *post* σ, we can take it.
hence *G1*: $(\sigma h, \sigma') \in (R - (UNIV \times (\Sigma \cup R$ `` $\{\sigma\})))^*$
by (*simp add*: *set-diff-diff-left Sigma-Un-distrib2*)
moreover have $\sigma h \neq \sigma$
— We may exclude the case that our obtained path started at σ, as all successors of σ are in R `` $\{\sigma\}$
proof
assume [*simp*]: $\sigma h = \sigma$
from *A SS* **have** $\sigma \neq \sigma'$ **by** *auto*
with *G1* **show** *False*
by (*auto simp add*: *elim*: *converse-rtranclE*)
qed
ultimately show *?thesis* **using** $IP(1)$ **by** *auto*
next
case (*last-visit-point* σt)
— If the obtained path contains a state from R `` $\{\sigma\}$, we simply pick the last one:
hence $(\sigma t, \sigma') \in (R - (UNIV \times (\Sigma \cup R$ `` $\{\sigma\})))^*$
by (*simp add*: *set-diff-diff-left Sigma-Un-distrib2*)
moreover from *last-visit-point*(2) **have** $\sigma t \notin \Sigma$
by (*auto elim*: *trancl.cases*)
ultimately show *?thesis* **using** *last-visit-point*(1) **by** *auto*
qed
qed

— The sse-algorithm is a well-defined while-algorithm
theorem *sse-while-algo*: *finite* $(R^*$ ``$\Sigma i) \Longrightarrow$ *while-algo* (*sse-algo* Σi R)
apply *unfold-locales*
apply (*auto simp add*: *sse-algo-def intro*: *sse-invar-step sse-invar-initial*)
apply (*drule sse-term*)
apply (*erule-tac wf-subset*)
apply (*unfold sse-term-rel-def*)

apply *auto*
done

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* (Σ, 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* Σi R == *dfs-invar-add* $\cap$ $\{$ $s.$ *dfs-α* $s \in$ *sse-invar* Σi 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* == $\{$ $(\Sigma, W).$ $W \neq []\}$

definition *dfs-algo* Σi R == (|
  wa-cond = *dfs-cond*,
  wa-step = *dfs-step* R,
  wa-initial = *dfs-initial* Σi,
  wa-invar = *dfs-invar* Σi R |)

— The DFS-algorithm refines the state-space exploration algorithm
theorem *dfs-pref-sse*:
  wa-precise-refine (*dfs-algo* Σi R) (*sse-algo* Σi R) *dfs-α*
  apply (*unfold-locales*)
  apply (*auto simp add*: *dfs-algo-def sse-algo-def dfs-cond-def sse-cond-def*
          dfs-α-def)
  apply (*erule dfs-step.cases*)
  apply (*rule-tac* σ=σ **in** *sse-step.intros*)
  apply (*auto simp add*: *dfs-invar-def sse-invar-def dfs-invar-add-def*
          sse-initial-def dfs-α-def
      elim: *dfs-initial.cases*)

done

— The DFS-algorithm is a well-defined while-algorithm
theorem *dfs-while-algo*:
assumes *finite*[*simp*, *intro*!]: *finite* (R^* ``Σi)
shows *while-algo* (*dfs-algo* Σi R)
proof −
interpret *wa-precise-refine* (*dfs-algo* Σi R) (*sse-algo* Σi R) *dfs*-α
using *dfs-pref-sse* **.**

have [*simp*]: *wa-invar* (*sse-algo* Σi R) = *sse-invar* Σi R
by (*simp add*: *sse-algo-def*)

show *?thesis*
apply (*rule wa-intro*)
apply (*simp add*: *sse-while-algo*)

apply (*simp add*: *dfs-invar-def dfs-algo-def*)

apply (*auto simp add*: *dfs-invar-add-def dfs-algo-def dfs*-α-*def*
dfs-cond-def sse-invar-def
elim!: *dfs-step.cases*) [*1*]

apply (*auto simp add*: *dfs-invar-add-def dfs-algo-def*
elim: *dfs-initial.cases*) [*1*]
done
qed

— The result of the DFS-algorithm is correct
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* $\times$ $'q$ *list*

— The loop terminates on empty worklist
definition *hs-dfs-cond* :: $'q$ *hs-dfs-state* $\Rightarrow$ *bool*
  where *hs-dfs-cond* S $==$ *let* $(Q,W) = S$ *in* $W \neq []$

— Refinement of a DFS-step, using hashset operations
definition *hs-dfs-step*
  :: $('q$::*hashable* $\Rightarrow$ $'q$ *ls*$)$ $\Rightarrow$ $'q$ *hs-dfs-state* $\Rightarrow$ $'q$ *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* :: $('q \Rightarrow 'q$ *ls*$)$ $\Rightarrow$ $('q\times'q)$ *set*
  where *hs-R post* $==$ $\{(q,q').\ q'\in$*ls-*α (*post* q)$\}$

— Initial state: Set of initial states in discovered set and on worklist
definition *hs-dfs-initial* :: $'q$::*hashable hs* $\Rightarrow$ $'q$ *hs-dfs-state*
  where *hs-dfs-initial* Σi $==$ $(\Sigma i$,*hs-to-list* $\Sigma i)$

— Abstraction mapping to abstract-DFS state
definition *hs-dfs-*α :: $'q$::*hashable hs-dfs-state* $\Rightarrow$ $'q$ *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* $\Rightarrow$ $('q \Rightarrow 'q$ *ls*$)$ $\Rightarrow$ $'q$ *hs-dfs-state set*
  where *hs-dfs-invar* Σi *post* $==$
    ($*$*hs-dfs-invar-add* $\cap*$) $\{$ *s*. (*hs-dfs-*α *s*) $\in$ *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* Σi *post*
 == *fst* (*while hs-dfs-cond* (*hs-dfs-step post*) (*hs-dfs-initial* Σi))

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)))
  $\in$*dfs-step* (*hs-R post*) (**is** *?T1*)

proof −

 from *ne* **obtain** σ *Wtl* **where**
  [*simp*]: W=σ#*Wtl*
  by (*cases W*) (*auto simp add*: *hs-dfs-cond-def*)

 have *G*: *let* (Q',W') = *hs-dfs-step post* (Q,W) *in*
  hs-invar Q' $\wedge$ ($\exists$ *Wn*.
   distinct Wn
   $\wedge$ *set Wn* = *ls-*α (*post* σ) − *hs-*α *Q*
   $\wedge$ W'=*Wn*@*Wtl*
   $\wedge$ *hs-*α Q'=*ls-*α (*post* σ) $\cup$ *hs-*α *Q*)
  apply (*simp add*: *hs-dfs-step-def*)
  apply (*rule-tac*
   I=λ*it* (Q',W'). *hs-invar* Q' $\wedge$ ($\exists$ *Wn*.
     distinct Wn
     $\wedge$ *set Wn* = (*ls-*α (*post* σ) − *it*) − *hs-*α *Q*
     $\wedge$ W'=*Wn*@*Wtl*
     $\wedge$ *hs-*α Q'=(*ls-*α (*post* σ) − *it*) $\cup$ *hs-*α *Q*)
   in *ls.iterate-rule-P*)
  apply *simp*
  apply *simp*
  apply (*force split*: *split-if-asm simp add*: *hs-correct*)
  apply *force*
  done
 from *G* **show** *?T1*
  apply (*cases hs-dfs-step post* (*Q*, *W*))
  apply (*auto simp add*: *hs-R-def hs-dfs-*α*-def intro*!: *dfs-step.intros*)
  done

qed

— Prove refinement

theorem *hs-dfs-pref-dfs*:

```
  shows wa-precise-refine
    (det-wa-wa (hs-dfs-dwa Σi post))
    (dfs-algo (hs-α Σi) (hs-R post))
    hs-dfs-α
  apply (unfold-locales)
  apply (auto simp add: det-wa-wa-def hs-dfs-dwa-def hs-dfs-α-def
                        hs-dfs-cond-def dfs-algo-def dfs-cond-def) [1]
  apply (auto simp add: det-wa-wa-def hs-dfs-dwa-def dfs-algo-def
                        hs-dfs-invar-def
              intro!: hs-dfs-step-correct(1)) [1]
  apply (auto simp add: det-wa-wa-def hs-dfs-dwa-def hs-dfs-α-def
                        hs-dfs-cond-def dfs-algo-def dfs-cond-def
                        hs-dfs-invar-def hs-dfs-step-def hs-dfs-initial-def
                        hs.to-list-correct
              intro: dfs-initial.intros
  ) [3]
  done
```

— Show that concrete algorithm is a while-algo

theorem *hs-dfs-while-algo*:

```
  assumes finite[simp]: finite ((hs-R post)* `` hs-α Σi)
  shows while-algo (det-wa-wa (hs-dfs-dwa Σi post))
proof -
  interpret ref: wa-precise-refine
    (det-wa-wa (hs-dfs-dwa Σi post))
    (dfs-algo (hs-α Σi) (hs-R post))
    hs-dfs-α
    using hs-dfs-pref-dfs .
  show ?thesis
    apply (rule ref.wa-intro[where addi=UNIV])
    apply (simp add: dfs-while-algo)
    apply (simp add: det-wa-wa-def hs-dfs-dwa-def hs-dfs-invar-def dfs-algo-def)

    apply (auto simp add: det-wa-wa-def hs-dfs-dwa-def) [1]
    apply simp

    done
qed
```

— Show that concrete algo is a deterministic while-algo

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* $((hs\text{-}R\ post)^*$ ‘‘ $hs\text{-}\alpha\ \Sigma i)$
shows $hs\text{-}\alpha\ (hs\text{-}dfs\ \Sigma i\ post) = (hs\text{-}R\ post)^*$‘‘$hs\text{-}\alpha\ \Sigma i$ (**is** *?T1*)

proof −
from *hs-dfs-det-while-algo*[*OF finite*] **interpret**
dwa: *det-while-algo* (*hs-dfs-dwa* Σi *post*) **.**

have
LC: (*while hs-dfs-cond* (*hs-dfs-step post*) (*hs-dfs-initial* Σi)) = *dwa.loop*
by (*unfold dwa.loop-def*) (*simp add*: *hs-dfs-dwa-def*)

have *?T1* (∗∧ *?T2*∗)
apply (*unfold hs-dfs-def*)
apply (*simp only*: *LC*)
apply (*rule dwa.while-proof*)
apply (*case-tac s*)
using *hs-dfs-invar-final*[*of* Σi *post*]
apply (*auto simp add*: *det-wa-wa-def hs-dfs-dwa-def*
dfs-α-def hs-dfs-α-def) [*1*]

done
thus *?T1* **by** *auto*
qed

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 $\mathsf{f_{div}} :: \mathsf{bool} = \mathsf{while}\ (\lambda x.\ \mathsf{True})\ \mathsf{id}\ \mathsf{True}$ that is definable in HOL. The lemma $\forall x.\ x = \mathsf{if}\ \mathsf{f_{div}}\ \mathsf{then}\ x\ \mathsf{else}\ x$ 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.