Isabelle Collections Framework

By Peter Lammich and Andreas Lochbihler

May 23, 2025

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[5] 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 5.2) and the ITP-2010 paper [3].

Chapter 2

The Generic Collection Framework

The Generic Collection Framework is build on top of the Automatic Refinement Framework. It contains set and map datastructures that are fully nestable, and a library of generic algorithms that are automatically instantiated on demand.

2.1 Interfaces

2.1.1 Map Interface

theory *Intf-Map*
imports *Refine-Monadic.Refine-Monadic*
begin

consts *i-map* :: *interface* ⇒ *interface* ⇒ *interface*

definition [*simp*]: *op-map-empty* ≡ *Map.empty*
definition *op-map-lookup* :: *'k* ⇒ (*'k*⇀*'v*) ⇀ *'v*
  where [*simp*]: *op-map-lookup k m* ≡ *m k*
definition [*simp*]: *op-map-update k v m* ≡ *m*(*k*↦*v*)
definition [*simp*]: *op-map-delete k m* ≡ *m* |` (−{*k*})
definition [*simp*]: *op-map-restrict P m* ≡ *m* |` {*k*∈*dom m*. *P* (*k*, *the* (*m k*))}
definition [*simp*]: *op-map-isEmpty x* ≡ *x*=*Map.empty*
definition [*simp*]: *op-map-isSng x* ≡ ∃ *k v*. *x*=[*k*↦*v*]
definition [*simp*]: *op-map-ball m P* ≡ *Ball* (*map-to-set m*) *P*
definition [*simp*]: *op-map-bex m P* ≡ *Bex* (*map-to-set m*) *P*
definition [*simp*]: *op-map-size m* ≡ *card* (*dom m*)
definition [*simp*]: *op-map-size-abort n m* ≡ *min n* (*card* (*dom m*))
definition [*simp*]: *op-map-sel m P* ≡ *SPEC* (λ(*k*,*v*). *m k* = *Some v* ∧ *P k v*)
definition [*simp*]: *op-map-pick m* ≡ *SPEC* (λ(*k*,*v*). *m k* = *Some v*)

definition [*simp*]: *op-map-pick-remove m* ≡

SPEC ($\lambda((k,v),m'). m\ k = Some\ v \wedge m' = m \mid` (-\{k\})$)

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma [*autoref-op-pat*]:
$Map.empty \equiv$ *op-map-empty*
$(m::'k \rightharpoonup' v)\ k \equiv$ *op-map-lookup*\$*k*\$*m*
$m(k \mapsto v) \equiv$ *op-map-update*\$*k*\$*v*\$*m*
$m \mid` (-\{k\}) \equiv$ *op-map-delete*\$*k*\$*m*
$m \mid` \{k \in dom\ m.\ P\ (k,\ the\ (m\ k))\} \equiv$ *op-map-restrict*\$*P*\$*m*

$m = Map.empty \equiv$ *op-map-isEmpty*\$*m*
$Map.empty = m \equiv$ *op-map-isEmpty*\$*m*
$dom\ m = \{\} \equiv$ *op-map-isEmpty*\$*m*
$\{\} = dom\ m \equiv$ *op-map-isEmpty*\$*m*

$\exists k\ v.\ m = [k \mapsto v] \equiv$ *op-map-isSng*\$*m*
$\exists k\ v.\ [k \mapsto v] = m \equiv$ *op-map-isSng*\$*m*
$\exists k.\ dom\ m = \{k\} \equiv$ *op-map-isSng*\$*m*
$\exists k.\ \{k\} = dom\ m \equiv$ *op-map-isSng*\$*m*
$1 = card\ (dom\ m) \equiv$ *op-map-isSng*\$*m*

$\bigwedge P.\ Ball\ (map\text{-}to\text{-}set\ m)\ P \equiv$ *op-map-ball*\$*m*\$*P*
$\bigwedge P.\ Bex\ (map\text{-}to\text{-}set\ m)\ P \equiv$ *op-map-bex*\$*m*\$*P*

$card\ (dom\ m) \equiv$ *op-map-size*\$*m*

$min\ n\ (card\ (dom\ m)) \equiv$ *op-map-size-abort*\$*n*\$*m*
$min\ (card\ (dom\ m))\ n \equiv$ *op-map-size-abort*\$*n*\$*m*

$\bigwedge P.\ SPEC\ (\lambda(k,v).\ m\ k = Some\ v \wedge P\ k\ v) \equiv$ *op-map-sel*\$*m*\$*P*
$\bigwedge P.\ SPEC\ (\lambda(k,v).\ P\ k\ v \wedge m\ k = Some\ v) \equiv$ *op-map-sel*\$*m*\$*P*

$\bigwedge P.\ SPEC\ (\lambda(k,v).\ m\ k = Some\ v) \equiv$ *op-map-pick*\$*m*
$\bigwedge P.\ SPEC\ (\lambda(k,v).\ (k,v) \in map\text{-}to\text{-}set\ m) \equiv$ *op-map-pick*\$*m*
⟨*proof*⟩

lemma [*autoref-op-pat*]:
SPEC ($\lambda((k,v),m'). m\ k = Some\ v \wedge m' = m \mid` (-\{k\})$)
$\equiv$ *op-map-pick-remove*\$*m*
⟨*proof*⟩

lemma *op-map-pick-remove-alt*:
do $\{((k,v),m) \leftarrow$ *op-map-pick-remove* m; $f\ k\ v\ m\}$
$=$ (
do {
$(k,v) \leftarrow SPEC\ (\lambda(k,v).\ m\ k = Some\ v)$;
let $m = m \mid` (-\{k\})$;

```
    f k v m
  })
  ⟨proof⟩

 lemma [autoref-op-pat]:
  do {
   (k,v)←SPEC (λ(k,v). m k = Some v);
    let m=m |` (−{k});
    f k v m
  } ≡ do {((k,v),m) ← op-map-pick-remove m; f k v m}
  ⟨proof⟩


end

lemma [autoref-itype]:
 op-map-empty ::ᵢ ⟨Ik,Iv⟩ᵢi-map
 op-map-lookup ::ᵢ Ik →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Iv⟩ᵢi-option
 op-map-update ::ᵢ Ik →ᵢ Iv →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Ik,Iv⟩ᵢi-map
 op-map-delete ::ᵢ Ik →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Ik,Iv⟩ᵢi-map
 op-map-restrict
  ::ᵢ (⟨Ik,Iv⟩ᵢi-prod →ᵢ i-bool) →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Ik,Iv⟩ᵢi-map
 op-map-isEmpty ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ i-bool
 op-map-isSng ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ i-bool
 op-map-ball ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ (⟨Ik,Iv⟩ᵢi-prod →ᵢ i-bool) →ᵢ i-bool
 op-map-bex ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ (⟨Ik,Iv⟩ᵢi-prod →ᵢ i-bool) →ᵢ i-bool
 op-map-size ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ i-nat
 op-map-size-abort ::ᵢ i-nat →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ i-nat
 (++) ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨Ik,Iv⟩ᵢi-map
 map-of ::ᵢ ⟨⟨Ik,Iv⟩ᵢi-prod⟩ᵢi-list →ᵢ ⟨Ik,Iv⟩ᵢi-map

 op-map-sel ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ (Ik →ᵢ Iv →ᵢ i-bool)
  →ᵢ ⟨⟨Ik,Iv⟩ᵢi-prod⟩ᵢi-nres
 op-map-pick ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨⟨Ik,Iv⟩ᵢi-prod⟩ᵢi-nres
 op-map-pick-remove
  ::ᵢ ⟨Ik,Iv⟩ᵢi-map →ᵢ ⟨⟨⟨Ik,Iv⟩ᵢi-prod,⟨Ik,Iv⟩ᵢi-map⟩ᵢi-prod⟩ᵢi-nres
 ⟨proof⟩

lemma hom-map1[autoref-hom]:
 CONSTRAINT Map.empty (⟨Rk,Rv⟩Rm)
 CONSTRAINT map-of (⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨Rk,Rv⟩Rm)
 CONSTRAINT (++) (⟨Rk,Rv⟩Rm → ⟨Rk,Rv⟩Rm → ⟨Rk,Rv⟩Rm)
 ⟨proof⟩

term op-map-restrict
lemma hom-map2[autoref-hom]:
 CONSTRAINT op-map-lookup (Rk→⟨Rk,Rv⟩Rm→⟨Rv⟩option-rel)
 CONSTRAINT op-map-update (Rk→Rv→⟨Rk,Rv⟩Rm→⟨Rk,Rv⟩Rm)
 CONSTRAINT op-map-delete (Rk→⟨Rk,Rv⟩Rm→⟨Rk,Rv⟩Rm)
```

CONSTRAINT op-map-restrict ((⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → ⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT op-map-isEmpty (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-isSng (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-ball (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*)→*Id*)
CONSTRAINT op-map-bex (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*)→*Id*)
CONSTRAINT op-map-size (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-size-abort (*Id*→⟨*Rk*,*Rv*⟩*Rm*→*Id*)

CONSTRAINT op-map-sel (⟨*Rk*,*Rv*⟩*Rm*→(*Rk* → *Rv* → *bool-rel*)→⟨*Rk*×$_r$*Rv*⟩*nres-rel*)
CONSTRAINT op-map-pick (⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*×$_r$*Rv*⟩*nres-rel*)
CONSTRAINT op-map-pick-remove (⟨*Rk*,*Rv*⟩*Rm* → ⟨(*Rk*×$_r$*Rv*)×$_r$⟨*Rk*,*Rv*⟩*Rm*⟩*nres-rel*)
⟨*proof*⟩

definition *finite-map-rel R* ≡ *Range R* ⊆ *Collect* (*finite* ∘ *dom*)
lemma *finite-map-rel-trigger*: *finite-map-rel R* ⟹ *finite-map-rel R* ⟨*proof*⟩

⟨*ML*⟩

end

2.1.2 Set Interface

theory *Intf-Set*
imports *Refine-Monadic.Refine-Monadic*
begin
consts *i-set* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of set-rel i-set*]

definition [*simp*]: *op-set-delete x s* ≡ *s* − {*x*}
definition [*simp*]: *op-set-isEmpty s* ≡ *s* = {}
definition [*simp*]: *op-set-isSng s* ≡ *card s* = *1*
definition [*simp*]: *op-set-size-abort m s* ≡ *min m* (*card s*)
definition [*simp*]: *op-set-disjoint a b* ≡ *a*∩*b*={}
definition [*simp*]: *op-set-filter P s* ≡ {*x*∈*s*. *P x*}
definition [*simp*]: *op-set-sel P s* ≡ *SPEC* (λ*x*. *x*∈*s* ∧ *P x*)
definition [*simp*]: *op-set-pick s* ≡ *SPEC* (λ*x*. *x*∈*s*)
definition [*simp*]: *op-set-to-sorted-list ordR s*
≡ *SPEC* (λ*l*. *set l* = *s* ∧ *distinct l* ∧ *sorted-wrt ordR l*)
definition [*simp*]: *op-set-to-list s* ≡ *SPEC* (λ*l*. *set l* = *s* ∧ *distinct l*)
definition [*simp*]: *op-set-cart x y* ≡ *x* × *y*

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-op-pat*]:
fixes *s a b* :: *'a set* **and** *x*::*'a* **and** *P* :: *'a* ⇒ *bool*

shows
$s - \{x\} \equiv$ *op-set-delete*$x$$s$

$s = \{\} \equiv$ *op-set-isEmpty*s
$\{\}=s \equiv$ *op-set-isEmpty*s

card $s = 1 \equiv$ *op-set-isSng*s
$\exists x.\ s=\{x\} \equiv$ *op-set-isSng*s
$\exists x.\ \{x\}=s \equiv$ *op-set-isSng*s

min m (*card* s) $\equiv$ *op-set-size-abort*$m$$s$
min (*card* s) $m \equiv$ *op-set-size-abort*$m$$s$

$a \cap b=\{\} \equiv$ *op-set-disjoint*$a$$b$

$\{x \in s.\ P\ x\} \equiv$ *op-set-filter*$P$$s$

SPEC $(\lambda x.\ x \in s \wedge P\ x) \equiv$ *op-set-sel*$P$$s$
SPEC $(\lambda x.\ P\ x \wedge x \in s) \equiv$ *op-set-sel*$P$$s$

SPEC $(\lambda x.\ x \in s) \equiv$ *op-set-pick*s
⟨*proof*⟩

lemma [*autoref-op-pat*]:
$a \times b \equiv$ *op-set-cart* a b
⟨*proof*⟩

lemma [*autoref-op-pat*]:
SPEC $(\lambda(u,v).\ (u,v) \in s) \equiv$ *op-set-pick*s
SPEC $(\lambda(u,v).\ P\ u\ v \wedge (u,v) \in s) \equiv$ *op-set-sel*$(*case-prod* P)$$s$
SPEC $(\lambda(u,v).\ (u,v) \in s \wedge P\ u\ v) \equiv$ *op-set-sel*$(*case-prod* P)$$s$
⟨*proof*⟩

lemma [*autoref-op-pat*]:
SPEC $(\lambda l.\ set\ l = s \wedge distinct\ l \wedge sorted\text{-}wrt\ ordR\ l)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$
SPEC $(\lambda l.\ set\ l = s \wedge sorted\text{-}wrt\ ordR\ l \wedge distinct\ l)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$
SPEC $(\lambda l.\ distinct\ l \wedge set\ l = s \wedge sorted\text{-}wrt\ ordR\ l)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$
SPEC $(\lambda l.\ distinct\ l \wedge sorted\text{-}wrt\ ordR\ l \wedge set\ l = s)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$
SPEC $(\lambda l.\ sorted\text{-}wrt\ ordR\ l \wedge distinct\ l \wedge set\ l = s)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$
SPEC $(\lambda l.\ sorted\text{-}wrt\ ordR\ l \wedge set\ l = s \wedge distinct\ l)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$

SPEC $(\lambda l.\ s = set\ l \wedge distinct\ l \wedge sorted\text{-}wrt\ ordR\ l)$
$\equiv$ *OP* $(op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)$$s$

$SPEC\ (\lambda l.\ s = set\ l \wedge sorted\text{-}wrt\ ordR\ l \wedge distinct\ l)$
$\equiv OP\ (op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)\$s$
$SPEC\ (\lambda l.\ distinct\ l \wedge s = set\ l \wedge sorted\text{-}wrt\ ordR\ l)$
$\equiv OP\ (op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)\$s$
$SPEC\ (\lambda l.\ distinct\ l \wedge sorted\text{-}wrt\ ordR\ l \wedge s = set\ l)$
$\equiv OP\ (op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)\$s$
$SPEC\ (\lambda l.\ sorted\text{-}wrt\ ordR\ l \wedge distinct\ l \wedge s = set\ l)$
$\equiv OP\ (op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)\$s$
$SPEC\ (\lambda l.\ sorted\text{-}wrt\ ordR\ l \wedge s = set\ l \wedge distinct\ l)$
$\equiv OP\ (op\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR)\$s$

$SPEC\ (\lambda l.\ set\ l = s \wedge distinct\ l) \equiv op\text{-}set\text{-}to\text{-}list\$s$
$SPEC\ (\lambda l.\ distinct\ l \wedge set\ l = s) \equiv op\text{-}set\text{-}to\text{-}list\$s$

$SPEC\ (\lambda l.\ s = set\ l \wedge distinct\ l) \equiv op\text{-}set\text{-}to\text{-}list\$s$
$SPEC\ (\lambda l.\ distinct\ l \wedge s = set\ l) \equiv op\text{-}set\text{-}to\text{-}list\$s$
⟨*proof*⟩

end

lemma [*autoref-itype*]:
$\{\} ::_i \langle I\rangle_i i\text{-}set$
$insert ::_i I \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$op\text{-}set\text{-}delete ::_i I \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$(\in) ::_i I \to_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$op\text{-}set\text{-}isEmpty ::_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$op\text{-}set\text{-}isSng ::_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$(\cup) ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$(\cap) ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$((-) :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set) ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$((=) :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool) ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$(\subseteq) ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$op\text{-}set\text{-}disjoint ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set \to_i i\text{-}bool$
$Ball ::_i \langle I\rangle_i i\text{-}set \to_i (I \to_i i\text{-}bool) \to_i i\text{-}bool$
$Bex ::_i \langle I\rangle_i i\text{-}set \to_i (I \to_i i\text{-}bool) \to_i i\text{-}bool$
$op\text{-}set\text{-}filter ::_i (I \to_i i\text{-}bool) \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$card ::_i \langle I\rangle_i i\text{-}set \to_i i\text{-}nat$
$op\text{-}set\text{-}size\text{-}abort ::_i i\text{-}nat \to_i \langle I\rangle_i i\text{-}set \to_i i\text{-}nat$
$set ::_i \langle I\rangle_i i\text{-}list \to_i \langle I\rangle_i i\text{-}set$
$op\text{-}set\text{-}sel ::_i (I \to_i i\text{-}bool) \to_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}nres$
$op\text{-}set\text{-}pick ::_i \langle I\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}nres$
$Sigma ::_i \langle Ia\rangle_i i\text{-}set \to_i (Ia \to_i \langle Ib\rangle_i i\text{-}set) \to_i \langle \langle Ia, Ib\rangle_i i\text{-}prod\rangle_i i\text{-}set$
$(`) ::_i (Ia \to_i Ib) \to_i \langle Ia\rangle_i i\text{-}set \to_i \langle Ib\rangle_i i\text{-}set$
$op\text{-}set\text{-}cart ::_i \langle Ix\rangle_i Isx \to_i \langle Iy\rangle_i Isy \to_i \langle \langle Ix,\ Iy\rangle_i i\text{-}prod\rangle_i Isp$
$Union ::_i \langle \langle I\rangle_i i\text{-}set\rangle_i i\text{-}set \to_i \langle I\rangle_i i\text{-}set$
$atLeastLessThan ::_i i\text{-}nat \to_i i\text{-}nat \to_i \langle i\text{-}nat\rangle_i i\text{-}set$
⟨*proof*⟩

lemma *hom-set1*[*autoref-hom*]:

CONSTRAINT {} (⟨*R*⟩*Rs*)
CONSTRAINT insert (*R*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT (∈) (*R*→⟨*R*⟩*Rs*→*Id*)
CONSTRAINT (∪) (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT (∩) (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT (−) (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT (=) (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→*Id*)
CONSTRAINT (⊆) (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→*Id*)
CONSTRAINT Ball (⟨*R*⟩*Rs*→(*R*→*Id*)→*Id*)
CONSTRAINT Bex (⟨*R*⟩*Rs*→(*R*→*Id*)→*Id*)
CONSTRAINT card (⟨*R*⟩*Rs*→*Id*)
CONSTRAINT set (⟨*R*⟩*Rl*→⟨*R*⟩*Rs*)
CONSTRAINT (‘) ((*Ra*→*Rb*) → ⟨*Ra*⟩*Rs*→⟨*Rb*⟩*Rs*)
CONSTRAINT Union (⟨⟨*R*⟩*Ri*⟩*Ro* → ⟨*R*⟩*Ri*)
⟨*proof*⟩

lemma *hom-set2*[*autoref-hom*]:
CONSTRAINT op-set-delete (*R*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT op-set-isEmpty (⟨*R*⟩*Rs*→*Id*)
CONSTRAINT op-set-isSng (⟨*R*⟩*Rs*→*Id*)
CONSTRAINT op-set-size-abort (*Id*→⟨*R*⟩*Rs*→*Id*)
CONSTRAINT op-set-disjoint (⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→*Id*)
CONSTRAINT op-set-filter ((*R*→*Id*)→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*)
CONSTRAINT op-set-sel ((*R* → *Id*)→⟨*R*⟩*Rs*→⟨*R*⟩*Rn*)
CONSTRAINT op-set-pick (⟨*R*⟩*Rs*→⟨*R*⟩*Rn*)
⟨*proof*⟩

lemma *hom-set-Sigma*[*autoref-hom*]:
CONSTRAINT Sigma (⟨*Ra*⟩*Rs* → (*Ra* → ⟨*Rb*⟩*Rs*) → ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*Rs2*)
⟨*proof*⟩

definition *finite-set-rel R* ≡ *Range R* ⊆ *Collect* (*finite*)

lemma *finite-set-rel-trigger*: *finite-set-rel R* ⟹ *finite-set-rel R* ⟨*proof*⟩

⟨*ML*⟩

end

2.1.3 Hashable Interface

theory *Intf-Hash*
imports
Main
../../Lib/HashCode
../../Lib/Code-Target-ICF
Automatic-Refinement.Automatic-Refinement
begin

type-synonym *'a eq = 'a ⇒ 'a ⇒ bool*
type-synonym *'k bhc = nat ⇒ 'k ⇒ nat*

Abstract and concrete hash functions

definition *is-bounded-hashcode* :: *('c×'a) set ⇒ 'c eq ⇒ 'c bhc ⇒ bool*
where *is-bounded-hashcode R eq bhc ≡*
((eq,(=)) ∈ R → R → bool-rel) ∧
(∀ n. ∀ x ∈ Domain R. ∀ y ∈ Domain R. eq x y ⟶ bhc n x = bhc n y) ∧
(∀ n x. 1 < n ⟶ bhc n x < n)
definition *abstract-bounded-hashcode* :: *('c×'a) set ⇒ 'c bhc ⇒ 'a bhc*
where *abstract-bounded-hashcode Rk bhc n x' ≡*
if x' ∈ Range Rk
then THE c. ∃ x. (x,x') ∈ Rk ∧ bhc n x = c
else 0

lemma *is-bounded-hashcodeI*[*intro*]:
((eq,(=)) ∈ R → R → bool-rel) ⟹
(⋀x y n. x ∈ Domain R ⟹ y ∈ Domain R ⟹ eq x y ⟹ bhc n x = bhc n y) ⟹
(⋀x n. 1 < n ⟹ bhc n x < n) ⟹ is-bounded-hashcode R eq bhc
⟨*proof*⟩

lemma *is-bounded-hashcodeD*[*dest*]:
assumes *is-bounded-hashcode R eq bhc*
shows *(eq,(=)) ∈ R → R → bool-rel* **and**
⋀n x y. x ∈ Domain R ⟹ y ∈ Domain R ⟹ eq x y ⟹ bhc n x = bhc n y **and**
⋀n x. 1 < n ⟹ bhc n x < n
⟨*proof*⟩

lemma *bounded-hashcode-welldefined*:
assumes *BHC*: *is-bounded-hashcode Rk eq bhc* **and**
R1: *(x1,x') ∈ Rk* **and** *R2*: *(x2,x') ∈ Rk*
shows *bhc n x1 = bhc n x2*
⟨*proof*⟩

lemma *abstract-bhc-correct*[*intro*]:
assumes *is-bounded-hashcode Rk eq bhc*
shows *(bhc, abstract-bounded-hashcode Rk bhc) ∈*
nat-rel → Rk → nat-rel (**is** *(bhc, ?bhc') ∈ -*)
⟨*proof*⟩

lemma *abstract-bhc-is-bhc*[*intro*]:
fixes *Rk* :: *('c×'a) set*
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
shows *is-bounded-hashcode Id (=) (abstract-bounded-hashcode Rk bhc)*
(**is** *is-bounded-hashcode - (=) ?bhc'*)

⟨*proof*⟩

lemma *hashable-bhc-is-bhc*[*autoref-ga-rules*]:
⟦*STRUCT-EQ-tag eq* (=); *REL-FORCE-ID R*⟧ ⟹ *is-bounded-hashcode R eq bounded-hashcode-nat*
⟨*proof*⟩

Default hash map size

definition *is-valid-def-hm-size* :: *'k itself* ⇒ *nat* ⇒ *bool*
where *is-valid-def-hm-size type n* ≡ *n* > *1*

lemma *hashable-def-size-is-def-size*[*autoref-ga-rules*]:
shows *is-valid-def-hm-size TYPE*('*k*::*hashable*) (*def-hashmap-size TYPE*('*k*))
⟨*proof*⟩

end

2.1.4 Orderings By Comparison Operator

theory *Intf-Comp*
imports
Automatic-Refinement.Automatic-Refinement
begin

Basic Definitions

datatype *comp-res* = *LESS* | *EQUAL* | *GREATER*

consts *i-comp-res* :: *interface*
abbreviation *comp-res-rel* ≡ *Id* :: (*comp-res* × -) *set*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of comp-res-rel i-comp-res*]

definition *comp2le cmp a b* ≡
case cmp a b of LESS ⇒ *True* | *EQUAL* ⇒ *True* | *GREATER* ⇒ *False*

definition *comp2lt cmp a b* ≡
case cmp a b of LESS ⇒ *True* | *EQUAL* ⇒ *False* | *GREATER* ⇒ *False*

definition *comp2eq cmp a b* ≡
case cmp a b of LESS ⇒ *False* | *EQUAL* ⇒ *True* | *GREATER* ⇒ *False*

locale *linorder-on* =
fixes *D* :: *'a set*
fixes *cmp* :: *'a* ⇒ *'a* ⇒ *comp-res*
assumes *lt-eq*: ⟦*x*∈*D*; *y*∈*D*⟧ ⟹ *cmp x y* = *LESS* ⟷ (*cmp y x* = *GREATER*)
assumes *refl*[*simp*, *intro*!]: *x*∈*D* ⟹ *cmp x x* = *EQUAL*

assumes *trans*[*trans*]:
$[\![x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = LESS;\ cmp\ y\ z = LESS]\!] \Longrightarrow cmp\ x\ z = LESS$
$[\![x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = LESS;\ cmp\ y\ z = EQUAL]\!] \Longrightarrow cmp\ x\ z = LESS$
$[\![x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = EQUAL;\ cmp\ y\ z = LESS]\!] \Longrightarrow cmp\ x\ z = LESS$
$[\![x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = EQUAL;\ cmp\ y\ z = EQUAL]\!] \Longrightarrow cmp\ x\ z = EQUAL$
begin
abbreviation *le* ≡ *comp2le cmp*
abbreviation *lt* ≡ *comp2lt cmp*

lemma *eq-sym*: $[\![x{\in}D;\ y{\in}D]\!] \Longrightarrow cmp\ x\ y = EQUAL \Longrightarrow cmp\ y\ x = EQUAL$
⟨*proof*⟩
end

abbreviation *linorder* ≡ *linorder-on UNIV*

lemma *linorder-to-class*:
assumes *linorder cmp*
assumes [*simp*]: $\bigwedge x\ y.\ cmp\ x\ y = EQUAL \Longrightarrow x{=}y$
shows *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
⟨*proof*⟩

definition *dflt-cmp le lt a b* ≡
if lt a b then LESS
else if le a b then EQUAL
else GREATER

lemma (**in** *linorder*) *class-to-linorder*:
linorder (*dflt-cmp* (≤) (<))
⟨*proof*⟩

lemma *restrict-linorder*: $[\![$*linorder-on D cmp* ; $D'{\subseteq}D]\!] \Longrightarrow$ *linorder-on D′ cmp*
⟨*proof*⟩

Operations on Linear Orderings

Map with injective function

definition *cmp-img* **where** *cmp-img f cmp a b* ≡ *cmp* (*f a*) (*f b*)

lemma *img-linorder*[*intro?*]:
assumes *LO*: *linorder-on* (*f*‘*D*) *cmp*
shows *linorder-on D* (*cmp-img f cmp*)
⟨*proof*⟩

Combine

definition *cmp-combine D1 cmp1 D2 cmp2 a b* ≡
if a∈*D1* ∧ *b*∈*D1 then cmp1 a b*
else if a∈*D1* ∧ *b*∈*D2 then LESS*
else if a∈*D2* ∧ *b*∈*D1 then GREATER*

else cmp2 a b

lemma *UnE′*:
assumes $x \in A \cup B$
obtains $x \in A \mid x \notin A \quad x \in B$
⟨*proof*⟩

lemma *combine-linorder*[*intro?*]:
assumes *linorder-on D1 cmp1*
assumes *linorder-on D2 cmp2*
assumes $D = D1 \cup D2$
shows *linorder-on D* (*cmp-combine D1 cmp1 D2 cmp2*)
⟨*proof*⟩

Universal Linear Ordering

With Zorn's Lemma, we get a universal linear (even wf) ordering

definition *univ-order-rel* ≡ (*SOME r. well-order-on UNIV r*)
definition *univ-cmp x y* ≡
if x=y then EQUAL
else if $(x,y) \in$*univ-order-rel then LESS*
else GREATER

lemma *univ-wo*: *well-order-on UNIV univ-order-rel*
⟨*proof*⟩

lemma *univ-linorder*[*intro?*]: *linorder univ-cmp*
⟨*proof*⟩

Extend any linear order to a universal order

definition *cmp-extend D cmp* ≡
cmp-combine D cmp UNIV univ-cmp

lemma *extend-linorder*[*intro?*]:
linorder-on D cmp ⟹ *linorder* (*cmp-extend D cmp*)
⟨*proof*⟩

Lexicographic Order on Lists

fun *cmp-lex* **where**
cmp-lex cmp [] [] = *EQUAL*
| *cmp-lex cmp* [] - = *LESS*
| *cmp-lex cmp* - [] = *GREATER*
| *cmp-lex cmp* (*a*#*l*) (*b*#*m*) = (
case cmp a b of
LESS ⇒ *LESS*
| *EQUAL* ⇒ *cmp-lex cmp l m*
| *GREATER* ⇒ *GREATER*)

primrec *cmp-lex′* **where**
cmp-lex′ cmp [] *m* = (*case m of* [] ⇒ *EQUAL* | - ⇒ *LESS*)
| *cmp-lex′ cmp* (*a*#*l*) *m* = (*case m of* [] ⇒ *GREATER* | (*b*#*m*) ⇒
(*case cmp a b of*
LESS ⇒ *LESS*
| *EQUAL* ⇒ *cmp-lex′ cmp l m*
| *GREATER* ⇒ *GREATER*
))

lemma *cmp-lex-alt*: *cmp-lex cmp l m* = *cmp-lex′ cmp l m*
⟨*proof*⟩

lemma (**in** *linorder-on*) *lex-linorder*[*intro?*]:
linorder-on (*lists D*) (*cmp-lex cmp*)
⟨*proof*⟩

Lexicographic Order on Pairs **fun** *cmp-prod* **where**
cmp-prod cmp1 cmp2 (*a1*,*a2*) (*b1*,*b2*)
= (
case cmp1 a1 b1 of
LESS ⇒ *LESS*
| *EQUAL* ⇒ *cmp2 a2 b2*
| *GREATER* ⇒ *GREATER*)

lemma *cmp-prod-alt*: *cmp-prod* = (λ*cmp1 cmp2* (*a1*,*a2*) (*b1*,*b2*). (
case cmp1 a1 b1 of
LESS ⇒ *LESS*
| *EQUAL* ⇒ *cmp2 a2 b2*
| *GREATER* ⇒ *GREATER*))
⟨*proof*⟩

lemma *prod-linorder*[*intro?*]:
assumes *A*: *linorder-on A cmp1*
assumes *B*: *linorder-on B cmp2*
shows *linorder-on* (*A*×*B*) (*cmp-prod cmp1 cmp2*)
⟨*proof*⟩

Universal Ordering for Sets that is Effective for Finite Sets

Sorted Lists of Sets Some more results about sorted lists of finite sets

lemma *set-to-map-set-is-map-of*:
distinct (*map fst l*) ⟹ *set-to-map* (*set l*) = *map-of l*
⟨*proof*⟩

context *linorder* **begin**

lemma *sorted-list-of-set-eq-nil2*[*simp*]:
assumes *finite A*

shows [] = *sorted-list-of-set A* ⟷ *A*={}
⟨*proof*⟩

lemma *set-insort*[*simp*]: *set* (*insort x l*) = *insert x* (*set l*)
⟨*proof*⟩

lemma *sorted-list-of-set-inj-aux*:
fixes *A B* :: *'a set*
assumes *finite A*
assumes *finite B*
assumes *sorted-list-of-set A* = *sorted-list-of-set B*
shows *A*=*B*
⟨*proof*⟩

lemma *sorted-list-of-set-inj*: *inj-on sorted-list-of-set* (*Collect finite*)
⟨*proof*⟩

definition *sorted-list-of-map m* ≡
map (λ*k*. (*k*, *the* (*m k*))) (*sorted-list-of-set* (*dom m*))

lemma *the-sorted-list-of-map*:
assumes *distinct* (*map fst l*)
assumes *sorted* (*map fst l*)
shows *sorted-list-of-map* (*map-of l*) = *l*
⟨*proof*⟩

lemma *map-of-sorted-list-of-map*[*simp*]:
assumes *FIN*: *finite* (*dom m*)
shows *map-of* (*sorted-list-of-map m*) = *m*
⟨*proof*⟩

lemma *sorted-list-of-map-inj-aux*:
fixes *A B* :: *'a*⇀*'b*
assumes [*simp*]: *finite* (*dom A*)
assumes [*simp*]: *finite* (*dom B*)
assumes *E*: *sorted-list-of-map A* = *sorted-list-of-map B*
shows *A*=*B*
⟨*proof*⟩

lemma *sorted-list-of-map-inj*:
inj-on sorted-list-of-map (*Collect* (*finite o dom*))
⟨*proof*⟩
end

definition *cmp-set cmp* ≡
cmp-extend (*Collect finite*) (
cmp-img
(*linorder.sorted-list-of-set* (*comp2le cmp*))
(*cmp-lex cmp*)

)

thm *img-linorder*

lemma *set-ord-linear*[*intro?*]:
linorder cmp $\Longrightarrow$ *linorder* (*cmp-set cmp*)
⟨*proof*⟩

definition *cmp-map cmpk cmpv* ≡
cmp-extend (*Collect* (*finite o dom*)) (
cmp-img
(*linorder.sorted-list-of-map* (*comp2le cmpk*))
(*cmp-lex* (*cmp-prod cmpk cmpv*))
)

lemma *map-to-set-inj*[*intro*!]: *inj map-to-set*
⟨*proof*⟩

corollary *map-to-set-inj'*[*intro*!]: *inj-on map-to-set S*
⟨*proof*⟩

lemma *map-ord-linear*[*intro?*]:
assumes *A*: *linorder cmpk*
assumes *B*: *linorder cmpv*
shows *linorder* (*cmp-map cmpk cmpv*)
⟨*proof*⟩

locale *eq-linorder-on* = *linorder-on* +
assumes *cmp-imp-equal*: ⟦*x*∈*D*; *y*∈*D*⟧ $\Longrightarrow$ *cmp x y* = *EQUAL* $\Longrightarrow$ *x* = *y*
begin
lemma *cmp-eq*[*simp*]: ⟦*x*∈*D*; *y*∈*D*⟧ $\Longrightarrow$ *cmp x y* = *EQUAL* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩
end

abbreviation *eq-linorder* ≡ *eq-linorder-on UNIV*

lemma *dflt-cmp-2inv*[*simp*]:
dflt-cmp (*comp2le cmp*) (*comp2lt cmp*) = *cmp*
⟨*proof*⟩

lemma (**in** *linorder*) *dflt-cmp-inv2*[*simp*]:
shows
(*comp2le* (*dflt-cmp* (≤) (<)))= (≤)
(*comp2lt* (*dflt-cmp* (≤) (<)))= (<)
⟨*proof*⟩

lemma *eq-linorder-class-conv*:

eq-linorder cmp $\longleftrightarrow$ *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
⟨*proof*⟩

lemma (**in** *linorder*) *class-to-eq-linorder*:
eq-linorder (*dflt-cmp* ($\leq$) ($<$))
⟨*proof*⟩

lemma *eq-linorder-comp2eq-eq*:
assumes *eq-linorder cmp*
shows *comp2eq cmp* = (=)
⟨*proof*⟩

lemma *restrict-eq-linorder*:
assumes *eq-linorder-on D cmp*
assumes *S*: $D' \subseteq D$
shows *eq-linorder-on D' cmp*
⟨*proof*⟩

lemma *combine-eq-linorder*[*intro?*]:
assumes *A*: *eq-linorder-on D1 cmp1*
assumes *B*: *eq-linorder-on D2 cmp2*
assumes *EQ*: $D = D1 \cup D2$
shows *eq-linorder-on D* (*cmp-combine D1 cmp1 D2 cmp2*)
⟨*proof*⟩

lemma *img-eq-linorder*[*intro?*]:
assumes *A*: *eq-linorder-on* (*f*‘*D*) *cmp*
assumes *INJ*: *inj-on f D*
shows *eq-linorder-on D* (*cmp-img f cmp*)
⟨*proof*⟩

lemma *univ-eq-linorder*[*intro?*]:
shows *eq-linorder univ-cmp*
⟨*proof*⟩

lemma *extend-eq-linorder*[*intro?*]:
assumes *eq-linorder-on D cmp*
shows *eq-linorder* (*cmp-extend D cmp*)
⟨*proof*⟩

lemma *lex-eq-linorder*[*intro?*]:
assumes *eq-linorder-on D cmp*
shows *eq-linorder-on* (*lists D*) (*cmp-lex cmp*)
⟨*proof*⟩

lemma *prod-eq-linorder*[*intro?*]:
assumes *eq-linorder-on D1 cmp1*
assumes *eq-linorder-on D2 cmp2*
shows *eq-linorder-on* ($D1 \times D2$) (*cmp-prod cmp1 cmp2*)

⟨*proof*⟩

lemma *set-ord-eq-linorder*[*intro?*]:
eq-linorder cmp ⟹ *eq-linorder* (*cmp-set cmp*)
⟨*proof*⟩

lemma *map-ord-eq-linorder*[*intro?*]:
⟦*eq-linorder cmpk*; *eq-linorder cmpv*⟧ ⟹ *eq-linorder* (*cmp-map cmpk cmpv*)
⟨*proof*⟩

definition *cmp-unit* :: *unit* ⇒ *unit* ⇒ *comp-res*
where [*simp*]: *cmp-unit u v* ≡ *EQUAL*

lemma *cmp-unit-eq-linorder*:
eq-linorder cmp-unit
⟨*proof*⟩

Parametricity

lemma *param-cmp-extend*[*param*]:
assumes (*cmp*,*cmp'*)∈*R* → *R* → *Id*
assumes *Range R* ⊆ *D*
shows (*cmp*,*cmp-extend D cmp'*) ∈ *R* → *R* → *Id*
⟨*proof*⟩

lemma *param-cmp-img*[*param*]:
(*cmp-img*,*cmp-img*) ∈ (*Ra*→*Rb*) → (*Rb*→*Rb*→*Rc*) → *Ra* → *Ra* → *Rc*
⟨*proof*⟩

lemma *param-comp-res*[*param*]:
(*LESS*,*LESS*)∈*Id*
(*EQUAL*,*EQUAL*)∈*Id*
(*GREATER*,*GREATER*)∈*Id*
(*case-comp-res*,*case-comp-res*)∈*Ra*→*Ra*→*Ra*→*Id*→*Ra*
⟨*proof*⟩

term *cmp-lex*
lemma *param-cmp-lex*[*param*]:
(*cmp-lex*,*cmp-lex*)∈(*Ra*→*Rb*→*Id*)→⟨*Ra*⟩*list-rel*→⟨*Rb*⟩*list-rel*→*Id*
⟨*proof*⟩

term *cmp-prod*
lemma *param-cmp-prod*[*param*]:
(*cmp-prod*,*cmp-prod*)∈
(*Ra*→*Rb*→*Id*)→(*Rc*→*Rd*→*Id*)→⟨*Ra*,*Rc*⟩*prod-rel*→⟨*Rb*,*Rd*⟩*prod-rel*→*Id*
⟨*proof*⟩

lemma *param-cmp-unit*[*param*]:
(*cmp-unit*,*cmp-unit*)∈*Id*→*Id*→*Id*

⟨*proof*⟩

lemma *param-comp2eq*[*param*]: (*comp2eq*,*comp2eq*)∈(*R*→*R*→*Id*)→*R*→*R*→*Id*
⟨*proof*⟩

lemma *cmp-combine-paramD*:
assumes (*cmp*,*cmp-combine D1 cmp1 D2 cmp2*)∈*R*→*R*→*Id*
assumes *Range R* ⊆ *D1*
shows (*cmp*,*cmp1*)∈*R*→*R*→*Id*
⟨*proof*⟩

lemma *cmp-extend-paramD*:
assumes (*cmp*,*cmp-extend D cmp′*)∈*R*→*R*→*Id*
assumes *Range R* ⊆ *D*
shows (*cmp*,*cmp′*)∈*R*→*R*→*Id*
⟨*proof*⟩

Tuning of Generated Implementation

lemma [*autoref-post-simps*]: *comp2eq* (*dflt-cmp* (≤) ((<)::-::*linorder*⇒-)) = (=)
⟨*proof*⟩

end

2.2 Generic Algorithms

2.2.1 Generic Set Algorithms

theory *Gen-Set*
imports *../Intf/Intf-Set ../../Iterator/Iterator*
begin

lemma *foldli-union*: *det-fold-set X* (λ-. *True*) *insert a* ((∪) *a*)
⟨*proof*⟩

definition *gen-union*
:: - ⇒ (*'k* ⇒ *'s2* ⇒ *'s2*)
⇒ *'s1* ⇒ *'s2* ⇒ *'s2*
where
gen-union it ins A B ≡ *it A* (λ-. *True*) *ins B*

lemma *gen-union*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *INS*: *GEN-OP ins Set.insert* (*Rk*→⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs2*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)

shows (*gen-union* (λ*x*. *foldli* (*tsl x*)) *ins*,(∪))
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs2*)
⟨*proof*⟩

lemma *foldli-inter*: *det-fold-set X* (λ-. *True*)
(λ*x s*. *if x*∈*a then insert x s else s*) {} (λ*s*. *s*∩*a*)
(**is** *det-fold-set - - ?f - -*)
⟨*proof*⟩

definition *gen-inter* :: - ⇒
('*k* ⇒ '*s2* ⇒ *bool*) ⇒ -
where *gen-inter it1 memb2 ins3 empty3 s1 s2*
≡ *it1 s1* (λ-. *True*)
(λ*x s*. *if memb2 x s2 then ins3 x s else s*) *empty3*

lemma *gen-inter*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)
assumes *MEMB*:
GEN-OP memb2 (∈) (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
assumes *INS*:
GEN-OP ins3 Set.insert (*Rk*→⟨*Rk*⟩*Rs3*→⟨*Rk*⟩*Rs3*)
assumes *EMPTY*:
GEN-OP empty3 {} (⟨*Rk*⟩*Rs3*)
shows (*gen-inter* (λ*x*. *foldli* (*tsl x*)) *memb2 ins3 empty3*,(∩))
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs3*)
⟨*proof*⟩

lemma *foldli-diff*:
det-fold-set X (λ-. *True*) (λ*x s*. *op-set-delete x s*) *s* ((−) *s*)
⟨*proof*⟩

definition *gen-diff* :: ('*k*⇒'*s1*⇒'*s1*) ⇒ - ⇒ '*s2* ⇒ -
where *gen-diff del1 it2 s1 s2*
≡ *it2 s2* (λ-. *True*) (λ*x s*. *del1 x s*) *s1*

lemma *gen-diff*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *DEL*:
GEN-OP del1 op-set-delete (*Rk* → ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs1*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs2 it2*)
shows (*gen-diff del1* (λ*x*. *foldli* (*it2 x*)),(−))
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs1*)
⟨*proof*⟩

lemma *foldli-ball-aux*:
foldli l (λ*x*. *x*) (λ*x* -. *P x*) *b* ⟷ *b* ∧ *Ball* (*set l*) *P*
⟨*proof*⟩

lemma *foldli-ball*: *det-fold-set X* (λ*x*. *x*) (λ*x* -. *P x*) *True* (λ*s*. *Ball s P*)
⟨*proof*⟩

definition *gen-ball* :: - ⇒ '*s* ⇒ ('*k* ⇒ *bool*) ⇒ -
where *gen-ball it s P* ≡ *it s* (λ*x*. *x*) (λ*x* -. *P x*) *True*

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-ball* (λ*x*. *foldli* (*it x*)),*Ball*) ∈ ⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*
⟨*proof*⟩

lemma *foldli-bex-aux*: *foldli l* (λ*x*. ¬*x*) (λ*x* -. *P x*) *b* ⟷ *b* ∨ *Bex* (*set l*) *P*
⟨*proof*⟩

lemma *foldli-bex*: *det-fold-set X* (λ*x*. ¬*x*) (λ*x* -. *P x*) *False* (λ*s*. *Bex s P*)
⟨*proof*⟩

definition *gen-bex* :: - ⇒ '*s* ⇒ ('*k* ⇒ *bool*) ⇒ -
where *gen-bex it s P* ≡ *it s* (λ*x*. ¬*x*) (λ*x* -. *P x*) *False*

lemma *gen-bex*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-bex* (λ*x*. *foldli* (*it x*)),*Bex*) ∈ ⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*
⟨*proof*⟩

lemma *ball-subseteq*:
(*Ball s1* (λ*x*. *x*∈*s2*)) ⟷ *s1* ⊆ *s2*
⟨*proof*⟩

definition *gen-subseteq*
:: ('*s1* ⇒ ('*k* ⇒ *bool*) ⇒ *bool*) ⇒ ('*k* ⇒ '*s2* ⇒ *bool*) ⇒ -
where *gen-subseteq ball1 mem2 s1 s2* ≡ *ball1 s1* (λ*x*. *mem2 x s2*)

lemma *gen-subseteq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball1 Ball* (⟨*Rk*⟩*Rs1* → (*Rk*→*Id*) → *Id*)
assumes *GEN-OP mem2* (∈) (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
shows (*gen-subseteq ball1 mem2*,(⊆)) ∈ ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs2* → *Id*
⟨*proof*⟩

definition *gen-equal ss1 ss2 s1 s2* ≡ *ss1 s1 s2* ∧ *ss2 s2 s1*

lemma *gen-equal*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ss1* (⊆) (⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs2* → *Id*)
assumes *GEN-OP ss2* (⊆) (⟨*Rk*⟩*Rs2* → ⟨*Rk*⟩*Rs1* → *Id*)
shows (*gen-equal ss1 ss2*, (=)) ∈ ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs2* → *Id*

⟨*proof*⟩

lemma *foldli-card-aux*: *distinct l* ⟹ *foldli l* (λ-. *True*)
(λ- *n*. *Suc n*) *n* = *n* + *card* (*set l*)
⟨*proof*⟩

lemma *foldli-card*: *det-fold-set X* (λ-. *True*) (λ- *n*. *Suc n*) *0 card*
⟨*proof*⟩

definition *gen-card* **where**
gen-card it s ≡ *it s* (λ*x*. *True*) (λ- *n*. *Suc n*) *0*

lemma *gen-card*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-card* (λ*x*. *foldli* (*it x*)),*card*) ∈ ⟨*Rk*⟩*Rs* → *Id*
⟨*proof*⟩

lemma *fold-set*: *fold Set.insert l s* = *s* ∪ *set l*
⟨*proof*⟩

definition *gen-set* :: *'s* ⇒ (*'k* ⇒ *'s* ⇒ *'s*) ⇒ - **where**
gen-set emp ins l = *fold ins l emp*

lemma *gen-set*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *EMPTY*:
GEN-OP emp {} (⟨*Rk*⟩*Rs*)
assumes *INS*:
GEN-OP ins Set.insert (*Rk*→⟨*Rk*⟩*Rs*→⟨*Rk*⟩*Rs*)
shows (*gen-set emp ins*,*set*)∈⟨*Rk*⟩*list-rel*→⟨*Rk*⟩*Rs*
⟨*proof*⟩

lemma *ball-isEmpty*: *op-set-isEmpty s* = (∀ *x*∈*s*. *False*)
⟨*proof*⟩

definition *gen-isEmpty* :: (*'s* ⇒ (*'k* ⇒ *bool*) ⇒ *bool*) ⇒ *'s* ⇒ *bool* **where**
gen-isEmpty ball s ≡ *ball s* (λ-. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball Ball* (⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*)
shows (*gen-isEmpty ball*,*op-set-isEmpty*) ∈ ⟨*Rk*⟩*Rs* → *Id*
⟨*proof*⟩

lemma *foldli-size-abort-aux*:
⟦*n0*≤*m*; *distinct l*⟧ ⟹
foldli l (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *n0* = *min m* (*n0* + *card* (*set l*))
⟨*proof*⟩

lemma *foldli-size-abort*:
det-fold-set X (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0* (*op-set-size-abort m*)
⟨*proof*⟩

definition *gen-size-abort* **where**
gen-size-abort it m s ≡ *it s* (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0*

lemma *gen-size-abort*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-size-abort* (λ*x*. *foldli* (*it x*)),*op-set-size-abort*)
∈ *Id* → ⟨*Rk*⟩*Rs* → *Id*
⟨*proof*⟩

lemma *size-abort-isSng*: *op-set-isSng s* ⟷ *op-set-size-abort 2 s* = *1*
⟨*proof*⟩

definition *gen-isSng* :: (*nat* ⇒ *'s* ⇒ *nat*) ⇒ - **where**
gen-isSng sizea s ≡ *sizea 2 s* = *1*

lemma *gen-isSng*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP sizea op-set-size-abort* (*Id* → (⟨*Rk*⟩*Rs*) → *Id*)
shows (*gen-isSng sizea*,*op-set-isSng*) ∈ ⟨*Rk*⟩*Rs* → *Id*
⟨*proof*⟩

lemma *foldli-disjoint-aux*:
foldli l1 (λ*x*. *x*) (λ*x* -. ¬*x*∈*s2*) *b* ⟷ *b* ∧ *op-set-disjoint* (*set l1*) *s2*
⟨*proof*⟩

lemma *foldli-disjoint*:
det-fold-set X (λ*x*. *x*) (λ*x* -. ¬*x*∈*s2*) *True* (λ*s1*. *op-set-disjoint s1 s2*)
⟨*proof*⟩

definition *gen-disjoint*
:: - ⇒ (*'k*⇒*'s2*⇒*bool*) ⇒ -
where *gen-disjoint it1 mem2 s1 s2*
≡ *it1 s1* (λ*x*. *x*) (λ*x* -. ¬*mem2 x s2*) *True*

lemma *gen-disjoint*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *MEM*: *GEN-OP mem2* (∈) (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
shows (*gen-disjoint* (λ*x*. *foldli* (*it1 x*)) *mem2*,*op-set-disjoint*)
∈ ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs2* → *Id*
⟨*proof*⟩

lemma *foldli-filter-aux*:

foldli l (λ-. *True*) (λ*x s*. *if P x then insert x s else s*) *s0*
= *s0* ∪ *op-set-filter P* (*set l*)
⟨*proof*⟩

lemma *foldli-filter*:
det-fold-set X (λ-. *True*) (λ*x s*. *if P x then insert x s else s*) {}
(*op-set-filter P*)
⟨*proof*⟩

definition *gen-filter*
where *gen-filter it1 emp2 ins2 P s1* ≡
it1 s1 (λ-. *True*) (λ*x s*. *if P x then ins2 x s else s*) *emp2*

lemma *gen-filter*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *INS*:
GEN-OP ins2 Set.insert (*Rk*→⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (⟨*Rk*⟩*Rs2*)
shows (*gen-filter* (λ*x*. *foldli* (*it1 x*)) *empty2 ins2*,*op-set-filter*)
∈ (*Rk*→*Id*) → (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*)
⟨*proof*⟩

lemma *foldli-image-aux*:
foldli l (λ-. *True*) (λ*x s*. *insert* (*f x*) *s*) *s0*
= *s0* ∪ *f*‘(*set l*)
⟨*proof*⟩

lemma *foldli-image*:
det-fold-set X (λ-. *True*) (λ*x s*. *insert* (*f x*) *s*) {}
((‘) *f*)
⟨*proof*⟩

definition *gen-image*
where *gen-image it1 emp2 ins2 f s1* ≡
it1 s1 (λ-. *True*) (λ*x s*. *ins2* (*f x*) *s*) *emp2*

lemma *gen-image*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *INS*:
GEN-OP ins2 Set.insert (*Rk′*→⟨*Rk′*⟩*Rs2*→⟨*Rk′*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (⟨*Rk′*⟩*Rs2*)
shows (*gen-image* (λ*x*. *foldli* (*it1 x*)) *empty2 ins2*,(‘))
∈ (*Rk*→*Rk′*) → (⟨*Rk*⟩*Rs1*) → (⟨*Rk′*⟩*Rs2*)
⟨*proof*⟩

lemma *foldli-pick*:
assumes $l \neq []$
obtains x **where** $x \in set\ l$
and $(foldli\ l\ (case\text{-}option\ True\ (\lambda\text{-}.\ False))\ (\lambda x\ \text{-}.\ Some\ x)\ None) = Some\ x$
⟨*proof*⟩

definition *gen-pick* **where**
gen-pick it s ≡
(*the* (*it s* (*case-option True* (λ-. *False*)) (λ*x* -. *Some x*) *None*))

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *gen-pick*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
assumes *NE*: *SIDE-PRECOND* ($s' \neq \{\}$)
assumes *SREF*: $(s,s') \in \langle Rk \rangle Rs$
shows (*RETURN* (*gen-pick* (λ*x*. *foldli* (*it x*)) *s*),
(*OP op-set-pick* ::: $\langle Rk \rangle Rs \to \langle Rk \rangle nres\text{-}rel$)$s' \in \langle Rk \rangle nres\text{-}rel$
⟨*proof*⟩
end

definition *gen-Sigma*
where *gen-Sigma it1 it2 empX insX s1 f2* ≡
it1 s1 (λ-. *True*) (λ*x s*.
it2 (*f2 x*) (λ-. *True*) (λ*y s*. *insX* (*x*,*y*) *s*) *s*
) *empX*

lemma *foldli-Sigma-aux*:
fixes *s* :: *'s1-impl* **and** *s'*:: *'k set*
fixes *f* :: *'k-impl* ⇒ *'s2-impl* **and** *f'*:: *'k* ⇒ *'l set*
fixes *s0* :: *'kl-impl* **and** *s0'* :: (*'k*×*'l*) *set*
assumes *IT1*: *is-set-to-list Rk Rs1 it1*
assumes *IT2*: *is-set-to-list Rl Rs2 it2*
assumes *INS*:
(*insX*, *Set.insert*) ∈
(⟨*Rk*,*Rl*⟩*prod-rel* → ⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3* → ⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*)
assumes *S0R*: (*s0*, *s0'*) ∈ ⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*
assumes *SR*: (*s*, *s'*) ∈ ⟨*Rk*⟩*Rs1*
assumes *FR*: (*f*, *f'*) ∈ *Rk* → ⟨*Rl*⟩*Rs2*
shows (*foldli* (*it1 s*) (λ-. *True*) (λ*x s*.
foldli (*it2* (*f x*)) (λ-. *True*) (λ*y s*. *insX* (*x*,*y*) *s*) *s*
) *s0*,*s0'* ∪ *Sigma s' f'*)
∈ ⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*
⟨*proof*⟩

lemma *gen-Sigma*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT1*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *IT2*: *SIDE-GEN-ALGO* (*is-set-to-list Rl Rs2 it2*)
assumes *EMPTY*:
GEN-OP empX {} (⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*)
assumes *INS*:
GEN-OP insX Set.insert
(⟨*Rk*,*Rl*⟩*prod-rel*→⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*→⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*)
shows (*gen-Sigma* (λ*x*. *foldli* (*it1 x*)) (λ*x*. *foldli* (*it2 x*)) *empX insX*,*Sigma*)
∈ (⟨*Rk*⟩*Rs1*) → (*Rk* → ⟨*Rl*⟩*Rs2*) → ⟨⟨*Rk*,*Rl*⟩*prod-rel*⟩*Rs3*
⟨*proof*⟩

lemma *gen-cart*:
assumes *PRIO-TAG-GEN-ALGO*
assumes [*param*]: (*sigma*, *Sigma*) ∈ (⟨*Rx*⟩*Rsx* → (*Rx* → ⟨*Ry*⟩*Rsy*) → ⟨*Rx* ×$_r$ *Ry*⟩*Rsp*)
shows (λ*x y*. *sigma x* (λ-. *y*), *op-set-cart*) ∈ ⟨*Rx*⟩*Rsx* → ⟨*Ry*⟩*Rsy* → ⟨*Rx* ×$_r$ *Ry*⟩*Rsp*
⟨*proof*⟩
lemmas [*autoref-rules*] = *gen-cart*[*OF - GEN-OP-D*]

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *op-set-to-sorted-list-autoref*[*autoref-rules*]:
assumes *SIDE-GEN-ALGO* (*is-set-to-sorted-list ordR Rk Rs tsl*)
shows (λ*si*. *RETURN* (*tsl si*), *OP* (*op-set-to-sorted-list ordR*))
∈ ⟨*Rk*⟩*Rs* → ⟨⟨*Rk*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩

lemma *op-set-to-list-autoref*[*autoref-rules*]:
assumes *SIDE-GEN-ALGO* (*is-set-to-sorted-list ordR Rk Rs tsl*)
shows (λ*si*. *RETURN* (*tsl si*), *op-set-to-list*)
∈ ⟨*Rk*⟩*Rs* → ⟨⟨*Rk*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩

end

lemma *foldli-Union*: *det-fold-set X* (λ-. *True*) (∪) {} *Union*
⟨*proof*⟩

definition *gen-Union*
:: - ⇒ *'s3* ⇒ (*'s2* ⇒ *'s3* ⇒ *'s3*)
⇒ *'s1* ⇒ *'s3*
where
gen-Union it emp un X ≡ *it X* (λ-. *True*) *un emp*

lemma *gen-Union*[*autoref-rules-raw*]:

assumes *PRIO-TAG-GEN-ALGO*
assumes *EMP*: *GEN-OP emp* {} (⟨*Rk*⟩*Rs3*)
assumes *UN*: *GEN-OP un* (∪) (⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs3*→⟨*Rk*⟩*Rs3*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list* (⟨*Rk*⟩*Rs2*) *Rs1 tsl*)
shows (*gen-Union* (λ*x*. *foldli* (*tsl x*)) *emp un*, *Union*) ∈ ⟨⟨*Rk*⟩*Rs2*⟩*Rs1* → ⟨*Rk*⟩*Rs3*
⟨*proof*⟩

definition *atLeastLessThan-impl a b* ≡ *do* {
(-,*r*) ← *WHILET* (λ(*i*,*r*). *i*<*b*) (λ(*i*,*r*). *RETURN* (*i*+*1*, *insert i r*)) (*a*,{});
RETURN r
}

lemma *atLeastLessThan-impl-correct*:
atLeastLessThan-impl a b ≤ *SPEC* (λ*r*. *r* = {*a*..<*b*::*nat*})
⟨*proof*⟩

schematic-goal *atLeastLessThan-code-aux*:
notes [*autoref-rules*] = *IdI*[*of a*] *IdI*[*of b*]
assumes [*autoref-rules*]: (*emp*,{})∈*Rs*
assumes [*autoref-rules*]: (*ins*,*insert*)∈*nat-rel* → *Rs* → *Rs*
shows (*?c*, *atLeastLessThan-impl*)
∈ *nat-rel* → *nat-rel* → ⟨*Rs*⟩*nres-rel*
⟨*proof*⟩
concrete-definition *atLeastLessThan-code* **uses** *atLeastLessThan-code-aux*

schematic-goal *atLeastLessThan-tr-aux*:
RETURN ?c ≤ *atLeastLessThan-code emp ins a b*
⟨*proof*⟩
concrete-definition *atLeastLessThan-tr*
for *emp ins a b* **uses** *atLeastLessThan-tr-aux*

lemma *atLeastLessThan-gen*[*autoref-rules*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP emp* {} *Rs*
assumes *GEN-OP ins insert* (*nat-rel* → *Rs* → *Rs*)
shows (*atLeastLessThan-tr emp ins*, *atLeastLessThan*)
∈ *nat-rel* → *nat-rel* → *Rs*
⟨*proof*⟩

end

2.2.2 Generic Map Algorithms

theory *Gen-Map*
imports *../Intf/Intf-Map ../../Iterator/Iterator*
begin
lemma *map-to-set-distinct-conv*:
assumes *distinct tsl′* **and** *map-to-set m′* = *set tsl′*
shows *distinct* (*map fst tsl′*)

⟨*proof*⟩

lemma *foldli-add*: *det-fold-map X*
(λ-. *True*) (λ(*k*,*v*) *m*. *op-map-update k v m*) *m* ((++) *m*)
⟨*proof*⟩

definition *gen-add*
:: ('*s2* ⇒ -) ⇒ ('*k* ⇒ '*v* ⇒ '*s1* ⇒ '*s1*) ⇒ '*s1* ⇒ '*s2* ⇒ '*s1*
where
gen-add it upd A B ≡ *it B* (λ-. *True*) (λ(*k*,*v*) *m*. *upd k v m*) *A*

lemma *gen-add*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *UPD*: *GEN-OP ins op-map-update* (*Rk*→*Rv*→⟨*Rk*,*Rv*⟩*Rs1*→⟨*Rk*,*Rv*⟩*Rs1*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rs2 tsl*)
shows (*gen-add* (*foldli o tsl*) *ins*,(++))
∈ (⟨*Rk*,*Rv*⟩*Rs1*) → (⟨*Rk*,*Rv*⟩*Rs2*) → (⟨*Rk*,*Rv*⟩*Rs1*)
⟨*proof*⟩

lemma *foldli-restrict*: *det-fold-map X* (λ-. *True*)
(λ(*k*,*v*) *m*. *if P* (*k*,*v*) *then op-map-update k v m else m*) *Map.empty*
(*op-map-restrict P*) (**is** *det-fold-map - - ?f - -*)
⟨*proof*⟩

definition *gen-restrict* :: ('*s1* ⇒ -) ⇒ -
where *gen-restrict it upd emp P m*
≡ *it m* (λ-. *True*) (λ(*k*,*v*) *m*. *if P* (*k*,*v*) *then upd k v m else m*) *emp*

lemma *gen-restrict*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rs1 tsl*)
assumes *INS*:
GEN-OP upd op-map-update (*Rk*→*Rv*→⟨*Rk*,*Rv*⟩*Rs2*→⟨*Rk*,*Rv*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP emp Map.empty (⟨*Rk*,*Rv*⟩*Rs2*)
shows (*gen-restrict* (*foldli o tsl*) *upd emp*,*op-map-restrict*)
∈ (⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → (⟨*Rk*,*Rv*⟩*Rs1*) → (⟨*Rk*,*Rv*⟩*Rs2*)
⟨*proof*⟩

lemma *fold-map-of*:
fold (λ(*k*,*v*) *s*. *op-map-update k v s*) (*rev l*) *Map.empty* = *map-of l*
⟨*proof*⟩

definition *gen-map-of* :: '*m* ⇒ ('*k*⇒'*v*⇒'*m*⇒'*m*) ⇒ - **where**
gen-map-of emp upd l ≡ *fold* (λ(*k*,*v*) *s*. *upd k v s*) (*rev l*) *emp*

lemma *gen-map-of*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *UPD*: *GEN-OP upd op-map-update* (*Rk*→*Rv*→⟨*Rk*,*Rv*⟩*Rm*→⟨*Rk*,*Rv*⟩*Rm*)
assumes *EMPTY*: *GEN-OP emp Map.empty* (⟨*Rk*,*Rv*⟩*Rm*)
shows (*gen-map-of emp upd*,*map-of*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨*Rk*,*Rv*⟩*Rm*
⟨*proof*⟩

lemma *foldli-ball-aux*:
distinct (*map fst l*) ⟹ *foldli l* (λ*x*. *x*) (λ*x* -. *P x*) *b*
⟷ *b* ∧ *op-map-ball* (*map-of l*) *P*
⟨*proof*⟩

lemma *foldli-ball*:
det-fold-map X (λ*x*. *x*) (λ*x* -. *P x*) *True* (λ*m*. *op-map-ball m P*)
⟨*proof*⟩

definition *gen-ball* :: (′*m* ⇒ -) ⇒ - **where**
gen-ball it m P ≡ *it m* (λ*x*. *x*) (λ*x* -. *P x*) *True*

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-ball* (*foldli o tsl*),*op-map-ball*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → (⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → *Id*
⟨*proof*⟩

lemma *foldli-bex-aux*:
distinct (*map fst l*) ⟹ *foldli l* (λ*x*. ¬*x*) (λ*x* -. *P x*) *b*
⟷ *b* ∨ *op-map-bex* (*map-of l*) *P*
⟨*proof*⟩

lemma *foldli-bex*:
det-fold-map X (λ*x*. ¬*x*) (λ*x* -. *P x*) *False* (λ*m*. *op-map-bex m P*)
⟨*proof*⟩

definition *gen-bex* :: (′*m* ⇒ -) ⇒ - **where**
gen-bex it m P ≡ *it m* (λ*x*. ¬*x*) (λ*x* -. *P x*) *False*

lemma *gen-bex*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-bex* (*foldli o tsl*),*op-map-bex*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → (⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → *Id*
⟨*proof*⟩

lemma *ball-isEmpty*: *op-map-isEmpty m* = *op-map-ball m* (λ-. *False*)
⟨*proof*⟩

definition *gen-isEmpty ball m* ≡ *ball m* (λ-. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *BALL*:
GEN-OP ball op-map-ball (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*) → *Id*)
shows (*gen-isEmpty ball*,*op-map-isEmpty*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → *Id*
⟨*proof*⟩

lemma *foldli-size-aux*: *distinct* (*map fst l*)
⟹ *foldli l* (λ-. *True*) (λ- *n*. *Suc n*) *n* = *n* + *op-map-size* (*map-of l*)
⟨*proof*⟩

lemma *foldli-size*: *det-fold-map X* (λ-. *True*) (λ- *n*. *Suc n*) *0 op-map-size*
⟨*proof*⟩

definition *gen-size* :: (′*m* ⇒ -) ⇒ -
where *gen-size it m* ≡ *it m* (λ-. *True*) (λ- *n*. *Suc n*) *0*

lemma *gen-size*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-size* (*foldli o tsl*),*op-map-size*) ∈ ⟨*Rk*,*Rv*⟩*Rm* → *Id*
⟨*proof*⟩

lemma *foldli-size-abort-aux*:
⟦*n0*≤*m*; *distinct* (*map fst l*)⟧ ⟹
foldli l (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *n0* = *min m* (*n0* + *card* (*dom* (*map-of l*)))
⟨*proof*⟩

lemma *foldli-size-abort*:
det-fold-map X (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0* (*op-map-size-abort m*)
⟨*proof*⟩

definition *gen-size-abort* :: (′*s* ⇒ -) ⇒ - **where**
gen-size-abort it m s ≡ *it s* (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0*

lemma *gen-size-abort*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-size-abort* (*foldli o tsl*),*op-map-size-abort*)
∈ *Id* → ⟨*Rk*,*Rv*⟩*Rm* → *Id*
⟨*proof*⟩

lemma *size-abort-isSng*: *op-map-isSng s* ⟷ *op-map-size-abort 2 s* = *1*
⟨*proof*⟩

definition *gen-isSng* :: (*nat* ⇒ ′*s* ⇒ *nat*) ⇒ - **where**
gen-isSng sizea s ≡ *sizea 2 s* = *1*

lemma *gen-isSng*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP sizea op-map-size-abort* (*Id* → (⟨*Rk*,*Rv*⟩*Rm*) → *Id*)
shows (*gen-isSng sizea*,*op-map-isSng*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → *Id*
⟨*proof*⟩

lemma *foldli-pick*:
assumes *l*≠[]
obtains *k v* **where** (*k*,*v*)∈*set l*
and (*foldli l* (*case-option True* (λ-. *False*)) (λ*x* -. *Some x*) *None*)
= *Some* (*k*,*v*)
⟨*proof*⟩

definition *gen-pick* **where**
gen-pick it s ≡
(*the* (*it s* (*case-option True* (λ-. *False*)) (λ*x* -. *Some x*) *None*))

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *gen-pick*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm it*)
assumes *NE*: *SIDE-PRECOND* (*m*′≠*Map.empty*)
assumes *SREF*: (*m*,*m*′)∈⟨*Rk*,*Rv*⟩*Rm*
shows (*RETURN* (*gen-pick* (λ*x*. *foldli* (*it x*)) *m*),
(*OP op-map-pick* ::: ⟨*Rk*,*Rv*⟩*Rm*→⟨*Rk*×$_r$*Rv*⟩*nres-rel*)$*m*′)∈⟨*Rk*×$_r$*Rv*⟩*nres-rel*
⟨*proof*⟩
end

definition *gen-map-pick-remove pick del m* ≡ *do* {
(*k*,*v*)←*pick m*;
let m = *del k m*;
RETURN ((*k*,*v*),*m*)
}

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *gen-map-pick-remove*
[*unfolded gen-map-pick-remove-def*, *autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *PICK*: *SIDE-GEN-OP* (
(*pick m*,

(*OP op-map-pick* ::: ⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*×$_r$*Rv*⟩*nres-rel*)$*m*′) ∈
⟨*Rk*×$_r$*Rv*⟩*nres-rel*)
assumes *DEL*: *GEN-OP del op-map-delete* (*Rk* → ⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*,*Rv*⟩*Rm*)
assumes [*param*]: (*m*,*m*′)∈⟨*Rk*,*Rv*⟩*Rm*
shows (*gen-map-pick-remove pick del m*,
(*OP op-map-pick-remove*
::: ⟨*Rk*,*Rv*⟩*Rm* → ⟨(*Rk*×$_r$*Rv*) ×$_r$ ⟨*Rk*,*Rv*⟩*Rm*⟩*nres-rel*)$*m*′)
∈ ⟨(*Rk*×$_r$*Rv*) ×$_r$ ⟨*Rk*,*Rv*⟩*Rm*⟩*nres-rel*
⟨*proof*⟩
end

end

2.2.3 Generic Map To Set Converter

theory *Gen-Map2Set*
imports
../Intf/Intf-Map
../Intf/Intf-Set
../Intf/Intf-Comp
../../Iterator/Iterator
begin

lemma *map-fst-unit-distinct-eq*[*simp*]:
fixes *l* :: (′*k*×*unit*) *list*
shows *distinct* (*map fst l*) ⟷ *distinct l*
⟨*proof*⟩

definition
map2set-rel ::
((′*ki*×′*k*) *set* ⇒ (*unit*×*unit*) *set* ⇒ (′*mi*×(′*k*⇀*unit*))*set*) ⇒
(′*ki*×′*k*) *set* ⇒
(′*mi*×(′*k set*)) *set*
where
map2set-rel-def-internal:
map2set-rel R Rk ≡ ⟨*Rk*,*Id*::(*unit*×-) *set*⟩*R O* {(*m*,*dom m*)| *m*. *True*}

lemma *map2set-rel-def*: ⟨*Rk*⟩(*map2set-rel R*)
= ⟨*Rk*,*Id*::(*unit*×-) *set*⟩*R O* {(*m*,*dom m*)| *m*. *True*}
⟨*proof*⟩

lemma *map2set-relI*:
assumes (*s*,*m*′)∈⟨*Rk*,*Id*⟩*R* **and** *s*′=*dom m*′
shows (*s*,*s*′)∈⟨*Rk*⟩*map2set-rel R*
⟨*proof*⟩

lemma *map2set-relE*:
assumes (*s*,*s*′)∈⟨*Rk*⟩*map2set-rel R*

obtains m' **where** $(s,m')\in\langle Rk,Id\rangle R$ **and** $s'=dom\ m'$
⟨*proof*⟩

lemma *map2set-rel-sv*[*relator-props*]:
single-valued $(\langle Rk,Id\rangle Rm) \Longrightarrow$ *single-valued* $(\langle Rk\rangle map2set\text{-}rel\ Rm)$
⟨*proof*⟩

lemma *map2set-empty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP e op-map-empty* $(\langle Rk,Id\rangle R)$
shows $(e,\{\})\in\langle Rk\rangle map2set\text{-}rel\ R$
⟨*proof*⟩

lemmas [*autoref-rel-intf*] =
REL-INTFI[*of map2set-rel R i-set*] **for** *R*

definition *map2set-insert i k s* ≡ *i k* () *s*
lemma *map2set-insert*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP i op-map-update* $(Rk \to Id \to \langle Rk,Id\rangle R \to \langle Rk,Id\rangle R)$
shows
$(map2set\text{-}insert\ i,Set.insert)\in Rk\to\langle Rk\rangle map2set\text{-}rel\ R \to \langle Rk\rangle map2set\text{-}rel\ R$
⟨*proof*⟩

definition *map2set-memb l k s* ≡ *case l k s of None* ⇒ *False* | *Some* - ⇒ *True*
lemma *map2set-memb*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP l op-map-lookup* $(Rk \to \langle Rk,Id\rangle R \to \langle Id\rangle option\text{-}rel)$
shows $(map2set\text{-}memb\ l\ ,(\in))$
$\in Rk\to\langle Rk\rangle map2set\text{-}rel\ R\to Id$
⟨*proof*⟩

lemma *map2set-delete*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP d op-map-delete* $(Rk\to\langle Rk,Id\rangle R\to\langle Rk,Id\rangle R)$
shows $(d,op\text{-}set\text{-}delete)\in Rk\to\langle Rk\rangle map2set\text{-}rel\ R\to\langle Rk\rangle map2set\text{-}rel\ R$
⟨*proof*⟩

lemma *map2set-to-sorted-list*[*autoref-ga-rules*]:
fixes *it* :: $'m \Rightarrow ('k\times unit)\ list$
assumes *A*: *GEN-ALGO-tag* (*is-map-to-sorted-list ordR Rk Id R it*)
shows *is-set-to-sorted-list ordR Rk* (*map2set-rel R*)
(*it-to-list* (*map-iterator-dom o* (*foldli o it*)))
⟨*proof*⟩

lemma *map2set-to-list*[*autoref-ga-rules*]:
fixes *it* :: $'m \Rightarrow ('k\times unit)\ list$
assumes *A*: *GEN-ALGO-tag* (*is-map-to-list Rk Id R it*)

shows *is-set-to-list Rk* (*map2set-rel R*)
(*it-to-list* (*map-iterator-dom o* (*foldli o it*)))
⟨*proof*⟩

Transfering also non-basic operations results in specializations of map-algorithms to also be used for sets

lemma *map2set-union*[*autoref-rules-raw*]:
assumes *MINOR-PRIO-TAG* (− *9*)
assumes *GEN-OP u* (++) (⟨*Rk*,*Id*⟩*R*→⟨*Rk*,*Id*⟩*R*→⟨*Rk*,*Id*⟩*R*)
shows (*u*,(∪))∈⟨*Rk*⟩*map2set-rel R*→⟨*Rk*⟩*map2set-rel R*→⟨*Rk*⟩*map2set-rel R*
⟨*proof*⟩

lemmas [*autoref-ga-rules*] = *cmp-unit-eq-linorder*
lemmas [*autoref-rules-raw*] = *param-cmp-unit*

lemma *cmp-lex-zip-unit*[*simp*]:
cmp-lex (*cmp-prod cmp cmp-unit*) (*map* (λ*k*. (*k*, ())) *l*)
(*map* (λ*k*. (*k*, ())) *m*) =
cmp-lex cmp l m
⟨*proof*⟩

lemma *cmp-img-zip-unit*[*simp*]:
cmp-img (λ*m*. *map* (λ*k*. (*k*,())) (*f m*)) (*cmp-lex* (*cmp-prod cmp1 cmp-unit*))
= *cmp-img f* (*cmp-lex cmp1*)
⟨*proof*⟩

lemma *map2set-finite*[*relator-props*]:
assumes *finite-map-rel* (⟨*Rk*,*Id*⟩*R*)
shows *finite-set-rel* (⟨*Rk*⟩*map2set-rel R*)
⟨*proof*⟩

lemma *map2set-cmp*[*autoref-rules-raw*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmpk*)
assumes *MPAR*:
GEN-OP cmp (*cmp-map cmpk cmp-unit*) (⟨*Rk*,*Id*⟩*R* → ⟨*Rk*,*Id*⟩*R* → *Id*)
assumes *FIN*: *PREFER finite-map-rel* (⟨*Rk*, *Id*⟩*R*)
shows (*cmp*,*cmp-set cmpk*)∈⟨*Rk*⟩*map2set-rel R* → ⟨*Rk*⟩*map2set-rel R* → *Id*
⟨*proof*⟩

end

2.2.4 Generic Compare Algorithms

theory *Gen-Comp*
imports
../Intf/Intf-Comp
Automatic-Refinement.Automatic-Refinement

HOL−Library.Product-Lexorder
begin

Order for Product

lemma *autoref-prod-cmp-dflt-id*[*autoref-rules-raw*]:
(*dflt-cmp* (≤) (<), *dflt-cmp* (≤) (<)) ∈
⟨*Id*,*Id*⟩*prod-rel* → ⟨*Id*,*Id*⟩*prod-rel* → *Id*
⟨*proof*⟩

lemma *gen-prod-cmp-dflt*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP cmp1* (*dflt-cmp* (≤) (<)) (*R1* → *R1* → *Id*)
assumes *GEN-OP cmp2* (*dflt-cmp* (≤) (<)) (*R2* → *R2* → *Id*)
shows (*cmp-prod cmp1 cmp2*, *dflt-cmp* (≤) (<)) ∈
⟨*R1*,*R2*⟩*prod-rel* → ⟨*R1*,*R2*⟩*prod-rel* → *Id*
⟨*proof*⟩

end

2.3 Implementations

2.3.1 Stack by Array

theory *Impl-Array-Stack*
imports
Automatic-Refinement.Automatic-Refinement
../../Lib/Diff-Array
begin

type-synonym *'a array-stack* = *'a array* × *nat*

term *Diff-Array.array-length*

definition *as-raw-α s* ≡ *take* (*snd s*) (*list-of-array* (*fst s*))
definition *as-raw-invar s* ≡ *snd s* ≤ *array-length* (*fst s*)

definition *as-rel-def-internal*: *as-rel R* ≡ *br as-raw-α as-raw-invar O* ⟨*R*⟩*list-rel*
lemma *as-rel-def*: ⟨*R*⟩*as-rel* ≡ *br as-raw-α as-raw-invar O* ⟨*R*⟩*list-rel*
⟨*proof*⟩

lemma [*relator-props*]: *single-valued R* ⟹ *single-valued* (⟨*R*⟩*as-rel*)
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of as-rel i-list*]

definition *as-empty* (-::*unit*) ≡ (*array-of-list* [],*0*)

lemma *as-empty-refine*[*autoref-rules*]: (*as-empty* (),[]) ∈ ⟨*R*⟩*as-rel*
⟨*proof*⟩

definition *as-push s x* ≡ *let*
(*a*,*n*)=*s*;
a = *if n* = *array-length a then*
array-grow a (*max 4* (*2*∗*n*)) *x*
else a;
a = *array-set a n x*
in
(*a*,*n*+*1*)

lemma *as-push-refine*[*autoref-rules*]:
(*as-push*,*op-list-append-elem*) ∈ ⟨*R*⟩*as-rel* → *R* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

term *array-shrink*

definition *as-shrink s* ≡ *let*
(*a*,*n*) = *s*;
a = *if 128*∗*n* ≤ *array-length a* ∧ *n*>*4 then*
array-shrink a n
else a
in
(*a*,*n*)

lemma *as-shrink-id-refine*: (*as-shrink*,*id*) ∈ ⟨*R*⟩*as-rel* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

lemma *as-shrinkI*:
assumes [*param*]: (*s*,*a*)∈⟨*R*⟩*as-rel*
shows (*as-shrink s*,*a*)∈⟨*R*⟩*as-rel*
⟨*proof*⟩

definition *as-pop s* ≡ *let* (*a*,*n*)=*s in as-shrink* (*a*,*n* − *1*)

lemma *as-pop-refine*[*autoref-rules*]: (*as-pop*,*butlast*) ∈ ⟨*R*⟩*as-rel* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

definition *as-get s i* ≡ *let* (*a*,-::*nat*)=*s in array-get a i*

lemma *as-get-refine*:
assumes *1*: *i'*<*length l*
assumes *2*: (*a*,*l*)∈⟨*R*⟩*as-rel*
assumes *3*[*param*]: (*i*,*i'*)∈*nat-rel*
shows (*as-get a i*,*l*!*i'*)∈*R*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *as-get-autoref*[*autoref-rules*]:
assumes (*l*,*l'*)∈⟨*R*⟩*as-rel*
assumes (*i*,*i'*)∈*Id*
assumes *SIDE-PRECOND* (*i'* < *length l'*)
shows (*as-get l i*,(*OP nth* ::: ⟨*R*⟩*as-rel* → *nat-rel* → *R*)$*l'*$*i'*)∈*R*
⟨*proof*⟩

definition *as-set s i x* ≡ *let* (*a*,*n*::*nat*)=*s in* (*array-set a i x*,*n*)

lemma *as-set-refine*[*autoref-rules*]:
(*as-set*,*list-update*)∈⟨*R*⟩*as-rel* → *nat-rel* → *R* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

definition *as-length* :: *'a array-stack* ⇒ *nat* **where**
as-length = *snd*

lemma *as-length-refine*[*autoref-rules*]:
(*as-length*,*length*) ∈ ⟨*R*⟩*as-rel* → *nat-rel*
⟨*proof*⟩

definition *as-top s* ≡ *as-get s* (*as-length s* − *1*)

lemma *as-top-code*[*code*]: *as-top s* = (*let* (*a*,*n*)=*s in array-get a* (*n* − *1*))
⟨*proof*⟩

lemma *as-top-refine*: ⟦*l*≠[]; (*s*,*l*)∈⟨*R*⟩*as-rel*⟧ ⟹ (*as-top s*,*last l*)∈*R*
⟨*proof*⟩

lemma *as-top-autoref*[*autoref-rules*]:
assumes (*l*,*l'*)∈⟨*R*⟩*as-rel*
assumes *SIDE-PRECOND* (*l'* ≠ [])
shows (*as-top l*,(*OP last* ::: ⟨*R*⟩*as-rel* → *R*)$*l'*)∈*R*
⟨*proof*⟩

definition *as-is-empty s* ≡ *as-length s* = *0*
lemma *as-is-empty-code*[*code*]: *as-is-empty s* = (*snd s* = *0*)
⟨*proof*⟩

lemma *as-is-empty-refine*[*autoref-rules*]:
(*as-is-empty*,*is-Nil*) ∈ ⟨*R*⟩*as-rel* → *bool-rel*
⟨*proof*⟩

definition *as-take m s* ≡ *let* (*a*,*n*) = *s in*
if m<*n then*
as-shrink (*a*,*m*)
else (*a*,*n*)

lemma *as-take-refine*[*autoref-rules*]:
(*as-take*,*take*)∈*nat-rel* → ⟨*R*⟩*as-rel* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

definition *as-singleton x* ≡ (*array-of-list* [*x*],*1*)
lemma *as-singleton-refine*[*autoref-rules*]:
(*as-singleton*,*op-list-singleton*)∈*R* → ⟨*R*⟩*as-rel*
⟨*proof*⟩

end

end

2.3.2 List Based Sets

theory *Impl-List-Set*
imports
../../Iterator/Iterator
../Intf/Intf-Set
begin

lemma *list-all2-refl-conv*:
list-all2 P xs xs ⟷ (∀ *x*∈*set xs*. *P x x*)
⟨*proof*⟩

primrec *glist-member* :: ('*a*⇒'*a*⇒*bool*) ⇒ '*a* ⇒ '*a list* ⇒ *bool* **where**
glist-member eq x [] ⟷ *False*
| *glist-member eq x* (*y*#*ys*) ⟷ *eq x y* ∨ *glist-member eq x ys*

lemma *param-glist-member*[*param*]:
(*glist-member*,*glist-member*)∈(*Ra*→*Ra*→*Id*) → *Ra* → ⟨*Ra*⟩*list-rel* → *Id*
⟨*proof*⟩

lemma *list-member-alt*: *List.member* = (λ*l x*. *glist-member* (=) *x l*)
⟨*proof*⟩

thm *List.insert-def*
definition
glist-insert eq x xs = (*if glist-member eq x xs then xs else x*#*xs*)

lemma *param-glist-insert*[*param*]:
(*glist-insert*, *glist-insert*) ∈ (*R*→*R*→*Id*) → *R* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

primrec *rev-append* **where**
rev-append [] *ac* = *ac*
| *rev-append* (*x*#*xs*) *ac* = *rev-append xs* (*x*#*ac*)

lemma *rev-append-eq*: *rev-append l ac = rev l @ ac*
⟨*proof*⟩

primrec *glist-delete-aux* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ - **where**
glist-delete-aux eq x [] *as = as*
| *glist-delete-aux eq x* (*y*#*ys*) *as* = (
if eq x y then rev-append as ys
else glist-delete-aux eq x ys (*y*#*as*)
)

definition *glist-delete* **where**
glist-delete eq x l ≡ *glist-delete-aux eq x l* []

lemma *param-glist-delete*[*param*]:
(*glist-delete*, *glist-delete*) ∈ (*R*→*R*→*Id*) → *R* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

lemma *list-rel-Range*:
∀ *x'*∈*set l'*. *x'* ∈ *Range R* ⟹ *l'* ∈ *Range* (⟨*R*⟩*list-rel*)
⟨*proof*⟩

All finite sets can be represented

lemma *list-set-rel-range*:
Range (⟨*R*⟩*list-set-rel*) = { *S*. *finite S* ∧ *S*⊆*Range R* }
(**is** *?A* = *?B*)
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of list-set-rel i-set*]

lemma *list-set-rel-finite*[*autoref-ga-rules*]:
finite-set-rel (⟨*R*⟩*list-set-rel*)
⟨*proof*⟩

lemma *list-set-rel-sv*[*relator-props*]:
single-valued R ⟹ *single-valued* (⟨*R*⟩*list-set-rel*)
⟨*proof*⟩

lemma *Id-comp-Id*: *Id O Id = Id* ⟨*proof*⟩

lemma *glist-member-id-impl*:
(*glist-member* (=), (∈)) ∈ *Id* → *br set distinct* → *Id*
⟨*proof*⟩

lemma *glist-insert-id-impl*:

(*glist-insert* (=), *Set.insert*) ∈ *Id* → *br set distinct* → *br set distinct*
⟨*proof*⟩

lemma *glist-delete-id-impl*:
(*glist-delete* (=), λ*x s*. *s*−{*x*})
∈ *Id*→*br set distinct* → *br set distinct*
⟨*proof*⟩

lemma *list-set-autoref-empty*[*autoref-rules*]:
([],{})∈⟨*R*⟩*list-set-rel*
⟨*proof*⟩

lemma *list-set-autoref-member*[*autoref-rules*]:
assumes *GEN-OP eq* (=) (*R*→*R*→*Id*)
shows (*glist-member eq*,(∈)) ∈ *R* → ⟨*R*⟩*list-set-rel* → *Id*
⟨*proof*⟩

lemma *list-set-autoref-insert*[*autoref-rules*]:
assumes *GEN-OP eq* (=) (*R*→*R*→*Id*)
shows (*glist-insert eq*,*Set.insert*)
∈ *R* → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*
⟨*proof*⟩

lemma *list-set-autoref-delete*[*autoref-rules*]:
assumes *GEN-OP eq* (=) (*R*→*R*→*Id*)
shows (*glist-delete eq*,*op-set-delete*)
∈ *R* → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*
⟨*proof*⟩

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* (λ- -. *True*) *R list-set-rel id*
⟨*proof*⟩

lemma *list-set-it-simp*[*refine-transfer-post-simp*]:
foldli (*id l*) = *foldli l* ⟨*proof*⟩

lemma *glist-insert-dj-id-impl*:
⟦ *x*∉*s*; (*l*,*s*)∈*br set distinct* ⟧ ⟹ (*x*#*l*,*insert x s*)∈*br set distinct*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *list-set-autoref-insert-dj*[*autoref-rules*]:
assumes *PRIO-TAG-OPTIMIZATION*
assumes *SIDE-PRECOND-OPT* (*x*′∉*s*′)
assumes (*x*,*x*′)∈*R*
assumes (*s*,*s*′)∈⟨*R*⟩*list-set-rel*
shows (*x*#*s*,
(*OP Set.insert* ::: *R* → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*) $ *x*′ $ *s*′)
∈ ⟨*R*⟩*list-set-rel*

⟨*proof*⟩
end

More Operations

lemma *list-set-autoref-isEmpty*[*autoref-rules*]:
(*is-Nil*,*op-set-isEmpty*) ∈ ⟨*R*⟩*list-set-rel* → *bool-rel*
⟨*proof*⟩

lemma *list-set-autoref-filter*[*autoref-rules*]:
(*filter*,*op-set-filter*)
∈ (*R* → *bool-rel*) → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *list-set-autoref-inj-image*[*autoref-rules*]:
assumes *PRIO-TAG-OPTIMIZATION*
assumes *INJ*: *SIDE-PRECOND-OPT* (*inj-on f s*)
assumes [*param*]: (*fi*,*f*)∈*Ra*→*Rb*
assumes *LP*: (*l*,*s*)∈⟨*Ra*⟩*list-set-rel*
shows (*map fi l*,
(*OP* (‘) ::: (*Ra*→*Rb*) → ⟨*Ra*⟩*list-set-rel* → ⟨*Rb*⟩*list-set-rel*)$*f*$*s*)
∈ ⟨*Rb*⟩*list-set-rel*
⟨*proof*⟩

end

lemma *list-set-cart-autoref*[*autoref-rules*]:
fixes *Rx* :: (′*xi* × ′*x*) *set*
fixes *Ry* :: (′*yi* × ′*y*) *set*
shows (λ*xl yl*. [(*x*,*y*). *x*←*xl*, *y*←*yl*], *op-set-cart*)
∈ ⟨*Rx*⟩*list-set-rel* → ⟨*Ry*⟩*list-set-rel* → ⟨*Rx* ×*r* *Ry*⟩*list-set-rel*
⟨*proof*⟩

Optimizations

lemma *glist-delete-hd*: *eq x y* ⟹ *glist-delete eq x* (*y*#*s*) = *s*
⟨*proof*⟩

Hack to ensure specific ordering. Note that ordering has no meaning abstractly

definition [*simp*]: *LIST-SET-REV-TAG* ≡ λ*x*. *x*

lemma *LIST-SET-REV-TAG-autoref*[*autoref-rules*]:
(*rev*,*LIST-SET-REV-TAG*) ∈ ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*
⟨*proof*⟩

end
theory *Array-Iterator*
imports *Iterator ../Lib/Diff-Array*
begin

lemma *idx-iteratei-aux-array-get-Array-conv-nth*:
idx-iteratei-aux array-get sz i (Array xs) c f σ =
idx-iteratei-aux (!) sz i xs c f σ
⟨*proof*⟩

lemma *idx-iteratei-array-get-Array-conv-nth*:
idx-iteratei array-get array-length (Array xs) = idx-iteratei nth length xs
⟨*proof*⟩

end

2.3.3 List Based Maps

theory *Impl-List-Map*
imports
../../Iterator/Iterator
../Gen/Gen-Map
../Intf/Intf-Comp
../Intf/Intf-Map
begin

type-synonym *('k,'v) list-map = ('k×'v) list*

definition *list-map-invar = distinct o map fst*

definition *list-map-rel-internal-def*:
list-map-rel Rk Rv ≡ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel O br map-of list-map-invar

lemma *list-map-rel-def*:
⟨Rk,Rv⟩list-map-rel = ⟨⟨Rk,Rv⟩prod-rel⟩list-rel O br map-of list-map-invar
⟨*proof*⟩

lemma *list-rel-Range*:
∀ x'∈set l'. x' ∈ Range R ⟹ l' ∈ Range (⟨R⟩list-rel)
⟨*proof*⟩

All finite maps can be represented

lemma *list-map-rel-range*:
Range (⟨Rk,Rv⟩list-map-rel) =
{m. finite (dom m) ∧ dom m ⊆ Range Rk ∧ ran m ⊆ Range Rv}

(**is** *?A* = *?B*)
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of list-map-rel i-map*]

lemma *list-map-rel-finite*[*autoref-ga-rules*]:
finite-map-rel (⟨*Rk*,*Rv*⟩*list-map-rel*)
⟨*proof*⟩

lemma *list-map-rel-sv*[*relator-props*]:
single-valued Rk ⟹ *single-valued Rv* ⟹
single-valued (⟨*Rk*,*Rv*⟩*list-map-rel*)
⟨*proof*⟩

Implementation

primrec *list-map-lookup* ::
('*k* ⇒ '*k* ⇒ *bool*) ⇒ '*k* ⇒ ('*k*,'*v*) *list-map* ⇒ '*v option* **where**
list-map-lookup eq - [] = *None* |
list-map-lookup eq k (*y*#*ys*) =
(*if eq* (*fst y*) *k then Some* (*snd y*) *else list-map-lookup eq k ys*)

primrec *list-map-update-aux* :: ('*k* ⇒ '*k* ⇒ *bool*) ⇒ '*k* ⇒ '*v* ⇒
('*k*,'*v*) *list-map* ⇒ ('*k*,'*v*) *list-map* ⇒ ('*k*,'*v*) *list-map***where**
list-map-update-aux eq k v [] *accu* = (*k*,*v*) # *accu* |
list-map-update-aux eq k v (*x*#*xs*) *accu* =
(*if eq* (*fst x*) *k*
then (*k*,*v*) # *xs* @ *accu*
else list-map-update-aux eq k v xs (*x*#*accu*))

definition *list-map-update eq k v m* ≡
list-map-update-aux eq k v m []

primrec *list-map-delete-aux* :: ('*k* ⇒ '*k* ⇒ *bool*) ⇒ '*k* ⇒
('*k*, '*v*) *list-map* ⇒ ('*k*, '*v*) *list-map* ⇒ ('*k*, '*v*) *list-map* **where**
list-map-delete-aux eq k [] *accu* = *accu* |
list-map-delete-aux eq k (*x*#*xs*) *accu* =
(*if eq* (*fst x*) *k*
then xs @ *accu*
else list-map-delete-aux eq k xs (*x*#*accu*))

definition *list-map-delete eq k m* ≡ *list-map-delete-aux eq k m* []

definition *list-map-isEmpty* :: ('*k*,'*v*) *list-map* ⇒ *bool*
where *list-map-isEmpty* ≡ *List.null*

definition *list-map-isSng* :: ('*k*,'*v*) *list-map* ⇒ *bool*

where *list-map-isSng m* = (*case m of* [*x*] ⇒ *True* | - ⇒ *False*)

definition *list-map-size* :: ('*k*,'*v*) *list-map* ⇒ *nat*
where *list-map-size* ≡ *length*

definition *list-map-iteratei* :: ('*k*,'*v*) *list-map* ⇒ ('*b* ⇒ *bool*) ⇒
(('*k*×'*v*) ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*b*
where *list-map-iteratei* ≡ *foldli*

definition *list-map-to-list* :: ('*k*,'*v*) *list-map* ⇒ ('*k*×'*v*) *list*
where *list-map-to-list* = *id*

Parametricity

lemma *list-map-autoref-empty*[*autoref-rules*]:
([], *op-map-empty*)∈⟨*Rk*,*Rv*⟩*list-map-rel*
⟨*proof*⟩

lemma *param-list-map-lookup*[*param*]:
(*list-map-lookup*,*list-map-lookup*) ∈ (*Rk* → *Rk* → *bool-rel*) →
Rk → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma *list-map-autoref-lookup-aux*:
assumes *eq*: *GEN-OP eq* (=) (*Rk*→*Rk*→*Id*)
assumes *K*: (*k*, *k'*) ∈ *Rk*
assumes *M*: (*m*, *m'*) ∈ ⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel*
shows (*list-map-lookup eq k m*, *op-map-lookup k'* (*map-of m'*))
∈ ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma *list-map-autoref-lookup*[*autoref-rules*]:
assumes *GEN-OP eq* (=) (*Rk*→*Rk*→*Id*)
shows (*list-map-lookup eq*, *op-map-lookup*) ∈
Rk → ⟨*Rk*,*Rv*⟩*list-map-rel* → ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma *param-list-map-update-aux*[*param*]:
(*list-map-update-aux*,*list-map-update-aux*) ∈ (*Rk* → *Rk* → *bool-rel*) →
Rk → *Rv* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
→ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *param-list-map-update*[*param*]:
(*list-map-update*,*list-map-update*) ∈ (*Rk* → *Rk* → *bool-rel*) →
Rk → *Rv* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *list-map-autoref-update-aux1*:
assumes *eq*: $(eq,(=)) \in Rk \to Rk \to Id$
assumes *K*: $(k,\ k') \in Rk$
assumes *V*: $(v,\ v') \in Rv$
assumes *A*: $(accu,\ accu') \in \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
assumes *M*: $(m,\ m') \in \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
shows (*list-map-update-aux eq k v m accu*,
list-map-update-aux $(=)$ k' v' m' $accu'$)
$\in \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
⟨*proof*⟩

lemma *list-map-autoref-update1*[*param*]:
assumes *eq*: $(eq,(=)) \in Rk \to Rk \to Id$
shows (*list-map-update eq*, *list-map-update* $(=)$) $\in Rk \to Rv \to$
$\langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel \to \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
⟨*proof*⟩

lemma *map-add-sng-right*: m ++ $[k \mapsto v] = m(k \mapsto v)$
⟨*proof*⟩
lemma *map-add-sng-right'*:
m ++ $(\lambda a.$ *if* $a = k$ *then Some* v *else None*$) = m(k \mapsto v)$
⟨*proof*⟩

lemma *list-map-autoref-update-aux2*:
assumes *K*: $(k,\ k') \in Id$
assumes *V*: $(v,\ v') \in Id$
assumes *A*: $(accu,\ accu') \in br\ map\text{-}of\ list\text{-}map\text{-}invar$
assumes *A1*: *distinct* (*map fst* (m @ $accu$))
assumes *A2*: $k \notin$ *set* (*map fst accu*)
assumes *M*: $(m,\ m') \in br\ map\text{-}of\ list\text{-}map\text{-}invar$
shows (*list-map-update-aux* $(=)$ k v m $accu$,
$accu'$ ++ *op-map-update* k' v' m')
$\in br\ map\text{-}of\ list\text{-}map\text{-}invar$ (**is** (*?f m accu*, -) $\in$ -)
⟨*proof*⟩

lemma *list-map-autoref-update2*[*param*]:
shows (*list-map-update* $(=)$, *op-map-update*) $\in Id \to Id \to$
$br\ map\text{-}of\ list\text{-}map\text{-}invar \to br\ map\text{-}of\ list\text{-}map\text{-}invar$
⟨*proof*⟩

lemma *list-map-autoref-update*[*autoref-rules*]:
assumes *eq*: *GEN-OP eq* $(=)$ $(Rk \to Rk \to Id)$
shows (*list-map-update eq*, *op-map-update*) $\in$
$Rk \to Rv \to \langle Rk,Rv\rangle list\text{-}map\text{-}rel \to \langle Rk,Rv\rangle list\text{-}map\text{-}rel$
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *list-map-autoref-update-dj*[*autoref-rules*]:
  assumes *PRIO-TAG-OPTIMIZATION*
  assumes *new*: *SIDE-PRECOND-OPT* ($k' \notin$ *dom* m')
  assumes *K*: $(k,k')\in Rk$ **and** *V*: $(v,v')\in Rv$
  assumes *M*: $(l,m')\in\langle Rk,\ Rv\rangle$*list-map-rel*
  defines *R-annot* $\equiv Rk \to Rv \to \langle Rk,Rv\rangle$*list-map-rel* $\to \langle Rk,Rv\rangle$*list-map-rel*
  shows
    ((*k*, *v*)#*l*,
      (*OP op-map-update*:::*R-annot*)$k'$$v'$$m'$)
    $\in \langle Rk,Rv\rangle$*list-map-rel*
⟨*proof*⟩
end

lemma *param-list-map-delete-aux*[*param*]:
  (*list-map-delete-aux*,*list-map-delete-aux*) $\in (Rk \to Rk \to$ *bool-rel*$) \to$
    $Rk \to \langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel* $\to \langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*
    $\to \langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*
⟨*proof*⟩

lemma *param-list-map-delete*[*param*]:
  (*list-map-delete*,*list-map-delete*) $\in (Rk \to Rk \to$ *bool-rel*$) \to$
    $Rk \to \langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel* $\to \langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*
⟨*proof*⟩

lemma *list-map-autoref-delete-aux1*:
  assumes *eq*: $(eq,(=)) \in Rk \to Rk \to Id$
  assumes *K*: $(k,\ k') \in Rk$
  assumes *A*: $(accu,\ accu') \in \langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel*
  assumes *M*: $(m,\ m') \in \langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel*
  shows (*list-map-delete-aux eq k m accu*,
      list-map-delete-aux $(=)$ k' m' $accu'$)
        $\in \langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel*
⟨*proof*⟩

lemma *list-map-autoref-delete1*[*param*]:
  assumes *eq*: $(eq,(=)) \in Rk \to Rk \to Id$
  shows (*list-map-delete eq*, *list-map-delete* $(=)) \in Rk \to$
      $\langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel* $\to \langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel*
⟨*proof*⟩

lemma *list-map-autoref-delete-aux2*:
  assumes *K*: $(k,\ k') \in Id$
  assumes *A*: $(accu,\ accu') \in$ *br map-of list-map-invar*
  assumes *A1*: *distinct* (*map fst* (m @ *accu*))
  assumes *A2*: $k \notin$ *set* (*map fst accu*)
  assumes *M*: $(m,\ m') \in$ *br map-of list-map-invar*

shows (*list-map-delete-aux* (=) *k m accu*,
accu' ++ *op-map-delete k' m'*)
∈ *br map-of list-map-invar* (**is** (*?f m accu*, -) ∈ -)
⟨*proof*⟩

lemma *list-map-autoref-delete2*[*param*]:
shows (*list-map-delete* (=), *op-map-delete*) ∈ *Id* →
br map-of list-map-invar → *br map-of list-map-invar*
⟨*proof*⟩

lemma *list-map-autoref-delete*[*autoref-rules*]:
assumes *eq*: *GEN-OP eq* (=) (*Rk*→*Rk*→*Id*)
shows (*list-map-delete eq*, *op-map-delete*) ∈
Rk → ⟨*Rk*,*Rv*⟩*list-map-rel* → ⟨*Rk*,*Rv*⟩*list-map-rel*
⟨*proof*⟩

lemma *list-map-autoref-isEmpty*[*autoref-rules*]:
shows (*list-map-isEmpty*, *op-map-isEmpty*) ∈
⟨*Rk*,*Rv*⟩*list-map-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-list-map-isSng*[*param*]:
assumes (*l*,*l'*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
shows (*list-map-isSng l*, *list-map-isSng l'*) ∈ *bool-rel*
⟨*proof*⟩

lemma *list-map-autoref-isSng-aux*:
assumes (*l'*,*m'*) ∈ *br map-of list-map-invar*
shows (*list-map-isSng l'*, *op-map-isSng m'*) ∈ *bool-rel*
⟨*proof*⟩

lemma *list-map-autoref-isSng*[*autoref-rules*]:
(*list-map-isSng*, *op-map-isSng*) ∈ ⟨*Rk*,*Rv*⟩*list-map-rel* → *bool-rel*
⟨*proof*⟩

lemma *list-map-autoref-size-aux*:
assumes *distinct* (*map fst x*)
shows *card* (*dom* (*map-of x*)) = *length x*
⟨*proof*⟩

lemma *param-list-map-size*[*param*]:
(*list-map-size*, *list-map-size*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → *nat-rel*
⟨*proof*⟩

lemma *list-map-autoref-size*[*autoref-rules*]:
shows (*list-map-size*, *op-map-size*) ∈
⟨*Rk*,*Rv*⟩*list-map-rel* → *nat-rel*
⟨*proof*⟩

lemma *autoref-list-map-is-iterator*[*autoref-ga-rules*]:
shows *is-map-to-list Rk Rv list-map-rel list-map-to-list*
⟨*proof*⟩

lemma *pi-list-map*[*icf-proper-iteratorI*]:
proper-it (*list-map-iteratei m*) (*list-map-iteratei m*)
⟨*proof*⟩

lemma *pi'-list-map*[*icf-proper-iteratorI*]:
proper-it' *list-map-iteratei list-map-iteratei*
⟨*proof*⟩

primrec *list-map-pick-remove* **where**
list-map-pick-remove [] = *undefined*
| *list-map-pick-remove* (*kv*#*l*) = (*kv*,*l*)

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *list-map-autoref-pick-remove*[*autoref-rules*]:
assumes *NE*: *SIDE-PRECOND* (*m*≠*Map.empty*)
assumes *R*: (*l*,*m*)∈⟨*Rk*,*Rv*⟩*list-map-rel*
defines *Rres* ≡ ⟨(*Rk*×$_r$*Rv*) ×$_r$ ⟨*Rk*,*Rv*⟩*list-map-rel*⟩*nres-rel*
shows (
RETURN (*list-map-pick-remove l*),
(*OP op-map-pick-remove* ::: ⟨*Rk*,*Rv*⟩*list-map-rel* → *Rres*)$*m*
) ∈ *Rres*
⟨*proof*⟩
end

end

2.3.4 Array Based Hash-Maps

theory *Impl-Array-Hash-Map* **imports**
Automatic-Refinement.Automatic-Refinement
../../Iterator/Array-Iterator
../Gen/Gen-Map
../Intf/Intf-Hash
../Intf/Intf-Map
../../Lib/HashCode
../../Lib/Code-Target-ICF
../../Lib/Diff-Array
Impl-List-Map
begin

Type definition and primitive operations

definition *load-factor* :: *nat* — in percent

where *load-factor = 75*

datatype (*'key*, *'val*) *hashmap* =
HashMap (*'key*,*'val*) *list-map array* *nat*

Operations

definition *new-hashmap-with* :: *nat* ⇒ (*'k*, *'v*) *hashmap*
where ⋀*size*. *new-hashmap-with size* =
HashMap (*new-array* [] *size*) *0*

definition *ahm-empty* :: *nat* ⇒ (*'k*, *'v*) *hashmap*
where *ahm-empty def-size* ≡ *new-hashmap-with def-size*

definition *bucket-ok* :: *'k bhc* ⇒ *nat* ⇒ *nat* ⇒ (*'k*×*'v*) *list* ⇒ *bool*
where *bucket-ok bhc len h kvs* = (∀ *k* ∈ *fst* ‘ *set kvs*. *bhc len k* = *h*)

definition *ahm-invar-aux* :: *'k bhc* ⇒ *nat* ⇒ (*'k*×*'v*) *list array* ⇒ *bool*
where
ahm-invar-aux bhc n a ⟷
(∀ *h*. *h* < *array-length a* ⟶ *bucket-ok bhc* (*array-length a*) *h*
(*array-get a h*) ∧ *list-map-invar* (*array-get a h*)) ∧
array-foldl (λ- *n kvs*. *n* + *size kvs*) *0 a* = *n* ∧
array-length a > *1*

primrec *ahm-invar* :: *'k bhc* ⇒ (*'k*, *'v*) *hashmap* ⇒ *bool*
where *ahm-invar bhc* (*HashMap a n*) = *ahm-invar-aux bhc n a*

definition *ahm-lookup-aux* :: *'k eq* ⇒ *'k bhc* ⇒
'k ⇒ (*'k*, *'v*) *list-map array* ⇒ *'v option*
where [*simp*]: *ahm-lookup-aux eq bhc k a* = *list-map-lookup eq k* (*array-get a* (*bhc* (*array-length a*) *k*))

primrec *ahm-lookup* **where**
ahm-lookup eq bhc k (*HashMap a* -) = *ahm-lookup-aux eq bhc k a*

definition *ahm-α bhc m* ≡ λ*k*. *ahm-lookup* (=) *bhc k m*

definition *ahm-map-rel-def-internal*:
ahm-map-rel Rk Rv ≡ {(*HashMap a n*, *HashMap a' n*)| *a a' n n'*.
(*a*,*a'*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* ∧ (*n*,*n'*) ∈ *Id*}

lemma *ahm-map-rel-def*: ⟨*Rk*,*Rv*⟩ *ahm-map-rel* ≡
{(*HashMap a n*, *HashMap a' n*)| *a a' n n'*.
(*a*,*a'*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* ∧ (*n*,*n'*) ∈ *Id*}
⟨*proof*⟩

definition *ahm-map-rel'-def*:
ahm-map-rel' bhc ≡ *br* (*ahm-α bhc*) (*ahm-invar bhc*)

definition *ahm-rel-def-internal*: *ahm-rel bhc Rk Rv* =
⟨*Rk*,*Rv*⟩ *ahm-map-rel O ahm-map-rel′* (*abstract-bounded-hashcode Rk bhc*)
lemma *ahm-rel-def*: ⟨*Rk*, *Rv*⟩ *ahm-rel bhc* ≡
⟨*Rk*,*Rv*⟩ *ahm-map-rel O ahm-map-rel′* (*abstract-bounded-hashcode Rk bhc*)
⟨*proof*⟩
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ahm-rel bhc i-map*] **for** *bhc*

abbreviation *dflt-ahm-rel* ≡ *ahm-rel bounded-hashcode-nat*

primrec *ahm-iteratei-aux* :: ((′*k*×′*v*) *list array*) ⇒ (′*k*×′*v*, ′*σ*) *set-iterator*
where *ahm-iteratei-aux* (*Array xs*) *c f* = *foldli* (*concat xs*) *c f*

primrec *ahm-iteratei* :: ((′*k*, ′*v*) *hashmap*) ⇒ ((′*k*×′*v*), ′*σ*) *set-iterator*
where
ahm-iteratei (*HashMap a n*) = *ahm-iteratei-aux a*

definition *ahm-rehash-aux′* :: ′*k bhc* ⇒ *nat* ⇒ ′*k*×′*v* ⇒
(′*k*×′*v*) *list array* ⇒ (′*k*×′*v*) *list array*
where
ahm-rehash-aux′ bhc n kv a =
(*let h* = *bhc n* (*fst kv*)
in array-set a h (*kv* # *array-get a h*))

definition *ahm-rehash-aux* :: ′*k bhc* ⇒ (′*k*×′*v*) *list array* ⇒ *nat* ⇒
(′*k*×′*v*) *list array*
where
ahm-rehash-aux bhc a sz = *ahm-iteratei-aux a* (λ*x*. *True*)
(*ahm-rehash-aux′ bhc sz*) (*new-array* [] *sz*)

primrec *ahm-rehash* :: ′*k bhc* ⇒ (′*k*,′*v*) *hashmap* ⇒ *nat* ⇒ (′*k*,′*v*) *hashmap*
where *ahm-rehash bhc* (*HashMap a n*) *sz* = *HashMap* (*ahm-rehash-aux bhc a sz*)
n

primrec *hm-grow* :: (′*k*,′*v*) *hashmap* ⇒ *nat*
where *hm-grow* (*HashMap a n*) = 2 ∗ *array-length a* + 3

primrec *ahm-filled* :: (′*k*,′*v*) *hashmap* ⇒ *bool*
where *ahm-filled* (*HashMap a n*) = (*array-length a* ∗ *load-factor* ≤ *n* ∗ 100)

primrec *ahm-update-aux* :: ′*k eq* ⇒ ′*k bhc* ⇒ (′*k*,′*v*) *hashmap* ⇒
′*k* ⇒ ′*v* ⇒ (′*k*, ′*v*) *hashmap*
where
ahm-update-aux eq bhc (*HashMap a n*) *k v* =
(*let h* = *bhc* (*array-length a*) *k*;
m = *array-get a h*;
insert = *list-map-lookup eq k m* = *None*
in HashMap (*array-set a h* (*list-map-update eq k v m*))

(*if insert then n + 1 else n*))

definition *ahm-update* :: *'k eq* ⇒ *'k bhc* ⇒ *'k* ⇒ *'v* ⇒
(*'k*, *'v*) *hashmap* ⇒ (*'k*, *'v*) *hashmap*
where
ahm-update eq bhc k v hm =
(*let hm'* = *ahm-update-aux eq bhc hm k v*
in (*if ahm-filled hm' then ahm-rehash bhc hm'* (*hm-grow hm'*) *else hm'*))

primrec *ahm-delete* :: *'k eq* ⇒ *'k bhc* ⇒ *'k* ⇒
(*'k*,*'v*) *hashmap* ⇒ (*'k*,*'v*) *hashmap*
where
ahm-delete eq bhc k (*HashMap a n*) =
(*let h* = *bhc* (*array-length a*) *k*;
m = *array-get a h*;
deleted = (*list-map-lookup eq k m* ≠ *None*)
in HashMap (*array-set a h* (*list-map-delete eq k m*)) (*if deleted then n* − *1 else n*))

primrec *ahm-isEmpty* :: (*'k*,*'v*) *hashmap* ⇒ *bool* **where**
ahm-isEmpty (*HashMap - n*) = (*n* = *0*)

primrec *ahm-isSng* :: (*'k*,*'v*) *hashmap* ⇒ *bool* **where**
ahm-isSng (*HashMap - n*) = (*n* = *1*)

primrec *ahm-size* :: (*'k*,*'v*) *hashmap* ⇒ *nat* **where**
ahm-size (*HashMap - n*) = *n*

lemma *hm-grow-gt-1* [*iff*]:
Suc 0 < *hm-grow hm*
⟨*proof*⟩

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok bhc len h* [] = *True*
⟨*proof*⟩

lemma *bucket-okD*:
⟦ *bucket-ok bhc len h xs*; (*k*, *v*) ∈ *set xs* ⟧
⟹ *bhc len k* = *h*
⟨*proof*⟩

lemma *bucket-okI*:
(⋀*k*. *k* ∈ *fst* ‘ *set kvs* ⟹ *bhc len k* = *h*) ⟹ *bucket-ok bhc len h kvs*
⟨*proof*⟩

Parametricity

lemma *param-HashMap*[*param*]: (*HashMap*, *HashMap*) ∈
⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* → *nat-rel* → ⟨*Rk*,*Rv*⟩*ahm-map-rel*

⟨*proof*⟩

lemma *param-case-hashmap*[*param*]: (*case-hashmap*, *case-hashmap*) ∈
(⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* → *nat-rel* → *R*) →
⟨*Rk*,*Rv*⟩*ahm-map-rel* → *R*
⟨*proof*⟩

lemma *rec-hashmap-is-case*[*simp*]: *rec-hashmap* = *case-hashmap*
⟨*proof*⟩

ahm-invar

lemma *ahm-invar-auxD*:
assumes *ahm-invar-aux bhc n a*
shows ⋀*h*. *h* < *array-length a* ⟹
bucket-ok bhc (*array-length a*) *h* (*array-get a h*) **and**
⋀*h*. *h* < *array-length a* ⟹
list-map-invar (*array-get a h*) **and**
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and**
array-length a > *1*
⟨*proof*⟩

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux bhc n a*
obtains ∀ *h*. *h* < *array-length a* ⟶
bucket-ok bhc (*array-length a*) *h* (*array-get a h*) ∧
list-map-invar (*array-get a h*) **and**
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and**
array-length a > *1*
⟨*proof*⟩

lemma *ahm-invar-auxI*:
⟦ ⋀*h*. *h* < *array-length a* ⟹
bucket-ok bhc (*array-length a*) *h* (*array-get a h*);
⋀*h*. *h* < *array-length a* ⟹ *list-map-invar* (*array-get a h*);
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*; *array-length a* > *1* ⟧
⟹ *ahm-invar-aux bhc n a*
⟨*proof*⟩

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux bhc n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
⟨*proof*⟩

ahm-α

lemma *list-map-lookup-is-map-of*:
list-map-lookup (=) *k l* = *map-of l k*
⟨*proof*⟩
definition *ahm-α-aux bhc a* ≡

(λk. ahm-lookup-aux (=) bhc k a)
lemma *ahm-α-aux-def2*: *ahm-α-aux bhc a* = (λk. *map-of* (*array-get a* (*bhc* (*array-length a*) *k*)) *k*)
⟨*proof*⟩
lemma *ahm-α-def2*: *ahm-α bhc* (*HashMap a n*) = *ahm-α-aux bhc a*
⟨*proof*⟩

lemma *finite-dom-ahm-α-aux*:
assumes *is-bounded-hashcode Id* (=) *bhc* *ahm-invar-aux bhc n a*
shows *finite* (*dom* (*ahm-α-aux bhc a*))
⟨*proof*⟩

lemma *ahm-α-aux-new-array*[*simp*]:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc* *1* < *sz*
shows *ahm-α-aux bhc* (*new-array* [] *sz*) *k* = *None*
⟨*proof*⟩

lemma *ahm-α-aux-conv-map-of-concat*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *inv*: *ahm-invar-aux bhc n* (*Array xs*)
shows *ahm-α-aux bhc* (*Array xs*) = *map-of* (*concat xs*)
⟨*proof*⟩

lemma *ahm-invar-aux-card-dom-ahm-α-auxD*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *inv*: *ahm-invar-aux bhc n a*
shows *card* (*dom* (*ahm-α-aux bhc a*)) = *n*
⟨*proof*⟩

lemma *finite-dom-ahm-α*:
assumes *is-bounded-hashcode Id* (=) *bhc* *ahm-invar bhc hm*
shows *finite* (*dom* (*ahm-α bhc hm*))
⟨*proof*⟩

ahm-empty

lemma *ahm-invar-aux-new-array*:
assumes *n* > *1*
shows *ahm-invar-aux bhc 0* (*new-array* [] *n*)
⟨*proof*⟩

lemma *ahm-invar-new-hashmap-with*:
n > *1* ⟹ *ahm-invar bhc* (*new-hashmap-with n*)
⟨*proof*⟩

lemma *ahm-α-new-hashmap-with*:
assumes *is-bounded-hashcode Id* (=) *bhc* **and** *n* > *1*
shows *Map.empty* = *ahm-α bhc* (*new-hashmap-with n*)
⟨*proof*⟩

lemma *ahm-empty-impl*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *def-size*: *def-size* > *1*
shows (*ahm-empty def-size*, *Map.empty*) ∈ *ahm-map-rel′ bhc*
⟨*proof*⟩

lemma *param-ahm-empty*[*param*]:
assumes *def-size*: (*def-size*, *def-size′*) ∈ *nat-rel*
shows (*ahm-empty def-size* ,*ahm-empty def-size′*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
⟨*proof*⟩

lemma *autoref-ahm-empty*[*autoref-rules*]:
fixes *Rk* :: (*′kc*×*′ka*) *set*
assumes *bhc*: *SIDE-GEN-ALGO* (*is-bounded-hashcode Rk eq bhc*)
assumes *def-size*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*′kc*) *def-size*)
shows (*ahm-empty def-size*, *op-map-empty*) ∈ ⟨*Rk*, *Rv*⟩*ahm-rel bhc*
⟨*proof*⟩

ahm-lookup

lemma *param-ahm-lookup*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
defines *bhc′-def*: *bhc′* ≡ *abstract-bounded-hashcode Rk bhc*
assumes *inv*: *ahm-invar bhc′ m′*
assumes *K*: (*k*,*k′*) ∈ *Rk*
assumes *M*: (*m*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
shows (*ahm-lookup eq bhc k m*, *ahm-lookup* (=) *bhc′ k′ m′*) ∈
⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma *ahm-lookup-impl*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-lookup* (=) *bhc*, *op-map-lookup*) ∈ *Id* → *ahm-map-rel′ bhc* → *Id*
⟨*proof*⟩

lemma *autoref-ahm-lookup*[*autoref-rules*]:
assumes
bhc[*unfolded autoref-tag-defs*]: *SIDE-GEN-ALGO* (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-lookup eq bhc*, *op-map-lookup*)
∈ *Rk* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

ahm-iteratei

abbreviation *ahm-to-list* ≡ *it-to-list ahm-iteratei*

lemma *param-ahm-iteratei-aux*[*param*]:
(*ahm-iteratei-aux*,*ahm-iteratei-aux*) ∈ ⟨⟨*Ra*⟩*list-rel*⟩*array-rel* →
(*Rb* → *bool-rel*) → (*Ra* → *Rb* → *Rb*) → *Rb* → *Rb*
⟨*proof*⟩

lemma *param-ahm-iteratei*[*param*]:
(*ahm-iteratei*,*ahm-iteratei*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* →
(*Rb* → *bool-rel*) → (⟨*Rk*,*Rv*⟩*prod-rel* → *Rb* → *Rb*) → *Rb* → *Rb*
⟨*proof*⟩

lemma *param-ahm-to-list*[*param*]:
(*ahm-to-list*,*ahm-to-list*) ∈
⟨*Rk*, *Rv*⟩*ahm-map-rel* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *ahm-to-list-distinct*[*simp*,*intro*]:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *inv*: *ahm-invar bhc m*
shows *distinct* (*ahm-to-list m*)
⟨*proof*⟩

lemma *set-ahm-to-list*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *ref*: (*m*,*m*′) ∈ *ahm-map-rel*′ *bhc*
shows *map-to-set m*′ = *set* (*ahm-to-list m*)
⟨*proof*⟩

lemma *ahm-iteratei-aux-impl*:
assumes *inv*: *ahm-invar-aux bhc n a*
and *bhc*: *is-bounded-hashcode Id* (=) *bhc*
shows *map-iterator* (*ahm-iteratei-aux a*) (*ahm-α-aux bhc a*)
⟨*proof*⟩

lemma *ahm-iteratei-impl*:
assumes *inv*: *ahm-invar bhc m*
and *bhc*: *is-bounded-hashcode Id* (=) *bhc*
shows *map-iterator* (*ahm-iteratei m*) (*ahm-α bhc m*)
⟨*proof*⟩

lemma *autoref-ahm-is-iterator*[*autoref-ga-rules*]:

assumes *bhc*: *GEN-ALGO-tag* (*is-bounded-hashcode Rk eq bhc*)
shows *is-map-to-list Rk Rv* (*ahm-rel bhc*) *ahm-to-list*
⟨*proof*⟩

lemma *ahm-iteratei-aux-code*[*code*]:
ahm-iteratei-aux a c f σ = *idx-iteratei array-get array-length a c*
(λ*x*. *foldli x c f*) *σ*
⟨*proof*⟩

ahm-rehash

lemma *array-length-ahm-rehash-aux′*:
array-length (*ahm-rehash-aux′ bhc n kv a*) = *array-length a*
⟨*proof*⟩

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
assumes *inv*: *ahm-invar-aux bhc n a*
and *bhc*: *is-bounded-hashcode Id* (=) *bhc*
and *fresh*: *k* ∉ *fst* ‘ *set* (*array-get a* (*bhc* (*array-length a*) *k*))
shows *ahm-invar-aux bhc* (*Suc n*) (*ahm-rehash-aux′ bhc* (*array-length a*) (*k*, *v*) *a*)
(**is** *ahm-invar-aux bhc - ?a*)
⟨*proof*⟩

lemma *ahm-rehash-aux-correct*:
fixes *a* :: (′*k*×′*v*) *list array*
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
and *inv*: *ahm-invar-aux bhc n a*
and *sz* > *1*
shows *ahm-invar-aux bhc n* (*ahm-rehash-aux bhc a sz*) (**is** *?thesis1*)
and *ahm-α-aux bhc* (*ahm-rehash-aux bhc a sz*) = *ahm-α-aux bhc a* (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-rehash-correct*:
fixes *hm* :: (′*k*, ′*v*) *hashmap*
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
and *inv*: *ahm-invar bhc hm*
and *sz* > *1*
shows *ahm-invar bhc* (*ahm-rehash bhc hm sz*)
ahm-α bhc (*ahm-rehash bhc hm sz*) = *ahm-α bhc hm*
⟨*proof*⟩

ahm-update

lemma *param-hm-grow*[*param*]:
(*hm-grow*, *hm-grow*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *nat-rel*
⟨*proof*⟩

lemma *param-ahm-rehash-aux′*[*param*]:
assumes *is-bounded-hashcode Rk eq bhc*

assumes $1 < n$
assumes $(bhc,bhc') \in nat\text{-}rel \to Rk \to nat\text{-}rel$
assumes $(n,n') \in nat\text{-}rel$ **and** $n = array\text{-}length\ a$
assumes $(kv,kv') \in \langle Rk,Rv\rangle prod\text{-}rel$
assumes $(a,a') \in \langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel$
shows $(ahm\text{-}rehash\text{-}aux'\ bhc\ n\ kv\ a,\ ahm\text{-}rehash\text{-}aux'\ bhc'\ n'\ kv'\ a') \in$
$\langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel$
⟨*proof*⟩

lemma *param-new-array*[*param*]:
$(new\text{-}array,\ new\text{-}array) \in R \to nat\text{-}rel \to \langle R\rangle array\text{-}rel$
⟨*proof*⟩

lemma *param-foldli-induct*:
assumes l: $(l,l') \in \langle Ra\rangle list\text{-}rel$
assumes c: $(c,c') \in Rb \to bool\text{-}rel$
assumes σ: $(\sigma,\sigma') \in Rb$
assumes $P\sigma$: $P\ \sigma\ \sigma'$
assumes f: $\bigwedge a\ a'\ b\ b'.\ (a,a')\in Ra \Longrightarrow (b,b')\in Rb \Longrightarrow c\ b \Longrightarrow c'\ b' \Longrightarrow$
$P\ b\ b' \Longrightarrow (f\ a\ b,\ f'\ a'\ b') \in Rb \wedge$
$P\ (f\ a\ b)\ (f'\ a'\ b')$
shows $(foldli\ l\ c\ f\ \sigma,\ foldli\ l'\ c'\ f'\ \sigma') \in Rb$
⟨*proof*⟩

lemma *param-foldli-induct-nocond*:
assumes l: $(l,l') \in \langle Ra\rangle list\text{-}rel$
assumes σ: $(\sigma,\sigma') \in Rb$
assumes $P\sigma$: $P\ \sigma\ \sigma'$
assumes f: $\bigwedge a\ a'\ b\ b'.\ (a,a')\in Ra \Longrightarrow (b,b')\in Rb \Longrightarrow P\ b\ b' \Longrightarrow$
$(f\ a\ b,\ f'\ a'\ b') \in Rb \wedge P\ (f\ a\ b)\ (f'\ a'\ b')$
shows $(foldli\ l\ (\lambda\text{-}.\ True)\ f\ \sigma,\ foldli\ l'\ (\lambda\text{-}.\ True)\ f'\ \sigma') \in Rb$
⟨*proof*⟩

lemma *param-ahm-rehash-aux*[*param*]:
assumes bhc: $is\text{-}bounded\text{-}hashcode\ Rk\ eq\ bhc$
assumes $bhc\text{-}rel$: $(bhc,bhc') \in nat\text{-}rel \to Rk \to nat\text{-}rel$
assumes A: $(a,a') \in \langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel$
assumes N: $(n,n') \in nat\text{-}rel \quad 1 < n$
shows $(ahm\text{-}rehash\text{-}aux\ bhc\ a\ n,\ ahm\text{-}rehash\text{-}aux\ bhc'\ a'\ n') \in$
$\langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel$
⟨*proof*⟩

lemma *param-ahm-rehash*[*param*]:
assumes bhc: $is\text{-}bounded\text{-}hashcode\ Rk\ eq\ bhc$
assumes $bhc\text{-}rel$: $(bhc,bhc') \in nat\text{-}rel \to Rk \to nat\text{-}rel$
assumes M: $(m,m') \in \langle Rk,Rv\rangle ahm\text{-}map\text{-}rel$

assumes N: $(n,n') \in$ *nat-rel* $\quad 1 < n$
shows (*ahm-rehash bhc m n*, *ahm-rehash bhc′ m′ n′*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
⟨*proof*⟩

lemma *param-load-factor*[*param*]:
(*load-factor*, *load-factor*) ∈ *nat-rel*
⟨*proof*⟩

lemma *param-ahm-filled*[*param*]:
(*ahm-filled*, *ahm-filled*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-ahm-update-aux*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc-rel*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel*
assumes *inv*: *ahm-invar bhc′ m′*
assumes K: $(k,k') \in Rk$
assumes V: $(v,v') \in Rv$
assumes M: $(m,m') \in \langle Rk,Rv\rangle$*ahm-map-rel*
shows (*ahm-update-aux eq bhc m k v*,
ahm-update-aux (=) *bhc′ m′ k′ v′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
⟨*proof*⟩

lemma *param-ahm-update*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc-rel*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel*
assumes *inv*: *ahm-invar bhc′ m′*
assumes K: $(k,k') \in Rk$
assumes V: $(v,v') \in Rv$
assumes M: $(m,m') \in \langle Rk,Rv\rangle$*ahm-map-rel*
shows (*ahm-update eq bhc k v m*, *ahm-update* (=) *bhc′ k′ v′ m′*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
⟨*proof*⟩

lemma *length-list-map-update*:
length (*list-map-update* (=) *k v xs*) =
(*if list-map-lookup* (=) *k xs* = *None then Suc* (*length xs*) *else length xs*)
(**is** *?l-new* = -)
⟨*proof*⟩

lemma *length-list-map-delete*:
length (*list-map-delete* (=) *k xs*) =
(*if list-map-lookup* (=) *k xs* = *None then length xs else length xs* − *1*)
(**is** *?l-new* = -)
⟨*proof*⟩

lemma *ahm-update-impl*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-update* (=) *bhc*, *op-map-update*) ∈ (*Id*::(*'k*×*'k*) *set*) →
(*Id*::(*'v*×*'v*) *set*) → *ahm-map-rel' bhc* → *ahm-map-rel' bhc*
⟨*proof*⟩

lemma *autoref-ahm-update*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-update eq bhc*, *op-map-update*) ∈
Rk → *Rv* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
⟨*proof*⟩

ahm-delete

lemma *param-ahm-delete*[*param*]:

assumes *isbhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc*: (*bhc*,*bhc'*) ∈ *nat-rel* → *Rk* → *nat-rel*
assumes *inv*: *ahm-invar bhc' m'*
assumes *K*: (*k*,*k'*) ∈ *Rk*
assumes *M*: (*m*,*m'*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
shows
(*ahm-delete eq bhc k m*, *ahm-delete* (=) *bhc' k' m'*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
⟨*proof*⟩

lemma *ahm-delete-impl*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-delete* (=) *bhc*, *op-map-delete*) ∈ (*Id*::(*'k*×*'k*) *set*) →
ahm-map-rel' bhc → *ahm-map-rel' bhc*
⟨*proof*⟩

lemma *autoref-ahm-delete*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-delete eq bhc*, *op-map-delete*) ∈
Rk → ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
⟨*proof*⟩

Various simple operations

lemma *param-ahm-isEmpty*[*param*]:
(*ahm-isEmpty*, *ahm-isEmpty*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-ahm-isSng*[*param*]:
(*ahm-isSng*, *ahm-isSng*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-ahm-size*[*param*]:
(*ahm-size*, *ahm-size*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *nat-rel*
⟨*proof*⟩

lemma *ahm-isEmpty-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-isEmpty*, *op-map-isEmpty*) ∈ *ahm-map-rel′ bhc* → *bool-rel*
⟨*proof*⟩

lemma *ahm-isSng-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-isSng*, *op-map-isSng*) ∈ *ahm-map-rel′ bhc* → *bool-rel*
⟨*proof*⟩

lemma *ahm-size-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-size*, *op-map-size*) ∈ *ahm-map-rel′ bhc* → *nat-rel*
⟨*proof*⟩

lemma *autoref-ahm-isEmpty*[*autoref-rules*]:

assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-isEmpty*, *op-map-isEmpty*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → *bool-rel*
⟨*proof*⟩

lemma *autoref-ahm-isSng*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-isSng*, *op-map-isSng*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → *bool-rel*
⟨*proof*⟩

lemma *autoref-ahm-size*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-size*, *op-map-size*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → *nat-rel*
⟨*proof*⟩

lemma *ahm-map-rel-sv*[*relator-props*]:
assumes *SK*: *single-valued Rk*
assumes *SV*: *single-valued Rv*
shows *single-valued* (⟨*Rk*, *Rv*⟩*ahm-map-rel*)
⟨*proof*⟩

lemma *ahm-rel-sv*[*relator-props*]:
⟦*single-valued Rk*; *single-valued Rv*⟧
⟹ *single-valued* (⟨*Rk*,*Rv*⟩*ahm-rel bhc*)
⟨*proof*⟩

lemma *rbt-map-rel-finite*[*relator-props*]:
assumes *A*[*simplified*]: *GEN-ALGO-tag* (*is-bounded-hashcode Rk eq bhc*)
shows *finite-map-rel* (⟨*Rk*,*Rv*⟩*ahm-rel bhc*)
⟨*proof*⟩

Proper iterator proofs

lemma *pi-ahm*[*icf-proper-iteratorI*]:
proper-it (*ahm-iteratei m*) (*ahm-iteratei m*)
⟨*proof*⟩

lemma *pi′-ahm*[*icf-proper-iteratorI*]:
proper-it′ ahm-iteratei ahm-iteratei
⟨*proof*⟩

lemmas *autoref-ahm-rules* =
autoref-ahm-empty
autoref-ahm-lookup
autoref-ahm-update
autoref-ahm-delete
autoref-ahm-isEmpty
autoref-ahm-isSng
autoref-ahm-size

lemmas *autoref-ahm-rules-hashable*[*autoref-rules-raw*]
= *autoref-ahm-rules*[**where** *Rk*=*Rk*] **for** *Rk* :: (-×-::*hashable*) *set*

end

2.3.5 Red-Black Tree based Maps

theory *Impl-RBT-Map*
imports
HOL−Library.RBT-Impl
../../Lib/RBT-add
Automatic-Refinement.Automatic-Refinement
../../Iterator/Iterator
../Intf/Intf-Comp
../Intf/Intf-Map

begin

Standard Setup

inductive-set *color-rel* **where**
(*color.R*,*color.R*) ∈ *color-rel*
| (*color.B*,*color.B*) ∈ *color-rel*

inductive-cases *color-rel-elims*:
(*x*,*color.R*) ∈ *color-rel*
(*x*,*color.B*) ∈ *color-rel*
(*color.R*,*y*) ∈ *color-rel*
(*color.B*,*y*) ∈ *color-rel*

thm *color-rel-elims*

lemma *param-color*[*param*]:
(*color.R*,*color.R*)∈*color-rel*
(*color.B*,*color.B*)∈*color-rel*
(*case-color*,*case-color*)∈*R* → *R* → *color-rel* → *R*
⟨*proof*⟩

inductive-set *rbt-rel-aux* **for** *Ra Rb* **where**
(*rbt.Empty*,*rbt.Empty*)∈*rbt-rel-aux Ra Rb*
| ⟦ (*c*,*c'*)∈*color-rel*;
(*l*,*l'*)∈*rbt-rel-aux Ra Rb*; (*a*,*a'*)∈*Ra*; (*b*,*b'*)∈*Rb*;
(*r*,*r'*)∈*rbt-rel-aux Ra Rb* ⟧
⟹ (*rbt.Branch c l a b r*, *rbt.Branch c' l' a' b' r'*)∈*rbt-rel-aux Ra Rb*

inductive-cases *rbt-rel-aux-elims*:
(*x*,*rbt.Empty*)∈*rbt-rel-aux Ra Rb*
(*rbt.Empty*,*x'*)∈*rbt-rel-aux Ra Rb*
(*rbt.Branch c l a b r*,*x'*)∈*rbt-rel-aux Ra Rb*
(*x*,*rbt.Branch c' l' a' b' r'*)∈*rbt-rel-aux Ra Rb*

definition *rbt-rel* ≡ *rbt-rel-aux*
lemma *rbt-rel-aux-fold*: *rbt-rel-aux Ra Rb* ≡ ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemmas *rbt-rel-intros* = *rbt-rel-aux.intros*[*unfolded rbt-rel-aux-fold*]
lemmas *rbt-rel-cases* = *rbt-rel-aux.cases*[*unfolded rbt-rel-aux-fold*]
lemmas *rbt-rel-induct*[*induct set*]
= *rbt-rel-aux.induct*[*unfolded rbt-rel-aux-fold*]
lemmas *rbt-rel-elims* = *rbt-rel-aux-elims*[*unfolded rbt-rel-aux-fold*]

lemma *param-rbt1*[*param*]:
(*rbt.Empty*,*rbt.Empty*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel*
(*rbt.Branch*,*rbt.Branch*) ∈
color-rel → ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*

⟨*proof*⟩

lemma *param-case-rbt*[*param*]:
(*case-rbt*,*case-rbt*) ∈
Ra → (*color-rel* → ⟨*Rb*,*Rc*⟩*rbt-rel* → *Rb* → *Rc* → ⟨*Rb*,*Rc*⟩*rbt-rel* → *Ra*)
→ ⟨*Rb*,*Rc*⟩*rbt-rel* → *Ra*
⟨*proof*⟩

lemma *param-rec-rbt*[*param*]: (*rec-rbt*, *rec-rbt*) ∈
Ra → (*color-rel* → ⟨*Rb*,*Rc*⟩*rbt-rel* → *Rb* → *Rc* → ⟨*Rb*,*Rc*⟩*rbt-rel*
→ *Ra* → *Ra* → *Ra*) → ⟨*Rb*,*Rc*⟩*rbt-rel* → *Ra*
⟨*proof*⟩

lemma *param-paint*[*param*]:
(*paint*,*paint*)∈*color-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-balance*[*param*]:
shows (*balance*,*balance*) ∈
⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-ins*[*param*]:
fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-ins less*,*ord.rbt-ins less'*) ∈
(*Ra*→*Rb*→*Rb*→*Rb*) → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

term *rbt-insert*
lemma *param-rbt-insert*[*param*]:
fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-insert less*,*ord.rbt-insert less'*) ∈
Ra → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-lookup*[*param*]:
fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-lookup less*,*ord.rbt-lookup less'*) ∈
⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → ⟨*Rb*⟩*option-rel*
⟨*proof*⟩

term *balance-left*
lemma *param-balance-left*[*param*]:
(*balance-left*, *balance-left*) ∈
⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*

⟨*proof*⟩

term *balance-right*
lemma *param-balance-right*[*param*]:
(*balance-right*, *balance-right*) ∈
⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-combine*[*param*]:
(*combine*,*combine*)∈⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *ih-aux1*: ⟦ (*a'*,*b*)∈*R*; *a'*=*a* ⟧ ⟹ (*a*,*b*)∈*R* ⟨*proof*⟩
lemma *is-eq*: *a*=*b* ⟹ *a*=*b* ⟨*proof*⟩

lemma *param-rbt-del-aux*:
fixes *br*
fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows
⟦ (*ak1*,*ak1'*)∈*Ra*; (*al*,*al'*)∈⟨*Ra*,*Rb*⟩*rbt-rel*; (*ak*,*ak'*)∈*Ra*;
(*av*,*av'*)∈*Rb*; (*ar*,*ar'*)∈⟨*Ra*,*Rb*⟩*rbt-rel*
⟧ ⟹ (*ord.rbt-del-from-left less ak1 al ak av ar*,
ord.rbt-del-from-left less' ak1' al' ak' av' ar')
∈ ⟨*Ra*,*Rb*⟩*rbt-rel*
⟦ (*bk1*,*bk1'*)∈*Ra*; (*bl*,*bl'*)∈⟨*Ra*,*Rb*⟩*rbt-rel*; (*bk*,*bk'*)∈*Ra*;
(*bv*,*bv'*)∈*Rb*; (*br*,*br'*)∈⟨*Ra*,*Rb*⟩*rbt-rel*
⟧ ⟹ (*ord.rbt-del-from-right less bk1 bl bk bv br*,
ord.rbt-del-from-right less' bk1' bl' bk' bv' br')
∈ ⟨*Ra*,*Rb*⟩*rbt-rel*
⟦ (*ck*,*ck'*)∈*Ra*; (*ct*,*ct'*)∈⟨*Ra*,*Rb*⟩*rbt-rel* ⟧
⟹ (*ord.rbt-del less ck ct*, *ord.rbt-del less' ck' ct'*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-del*[*param*]:
fixes *less*
assumes *param-less*: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows
(*ord.rbt-del-from-left less*, *ord.rbt-del-from-left less'*) ∈
Ra → ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
(*ord.rbt-del-from-right less*, *ord.rbt-del-from-right less'*) ∈
Ra → ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
(*ord.rbt-del less*,*ord.rbt-del less'*) ∈
Ra → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-delete*[*param*]:
fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*

shows (*ord.rbt-delete less*, *ord.rbt-delete less'*)
∈ *Ra* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

term *ord.rbt-insert-with-key*

abbreviation *compare-rel* :: (*RBT-Impl.compare* × -) *set*
where *compare-rel* ≡ *Id*

lemma *param-compare*[*param*]:
(*RBT-Impl.LT*,*RBT-Impl.LT*)∈*compare-rel*
(*RBT-Impl.GT*,*RBT-Impl.GT*)∈*compare-rel*
(*RBT-Impl.EQ*,*RBT-Impl.EQ*)∈*compare-rel*
(*RBT-Impl.case-compare*,*RBT-Impl.case-compare*)∈*R*→*R*→*R*→*compare-rel*→*R*
⟨*proof*⟩

lemma *param-rbtreeify-aux*[*param*]:
⟦*n*≤*length kvs*; (*n*,*n'*)∈*nat-rel*; (*kvs*,*kvs'*)∈⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*⟧
⟹ (*rbtreeify-f n kvs*,*rbtreeify-f n' kvs'*)
∈ ⟨⟨*Ra*,*Rb*⟩*rbt-rel*, ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*
⟦*n*≤*Suc* (*length kvs*); (*n*,*n'*)∈*nat-rel*; (*kvs*,*kvs'*)∈⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*⟧
⟹ (*rbtreeify-g n kvs*,*rbtreeify-g n' kvs'*)
∈ ⟨⟨*Ra*,*Rb*⟩*rbt-rel*, ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*
⟨*proof*⟩

lemma *param-rbtreeify*[*param*]:
(*rbtreeify*, *rbtreeify*) ∈ ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-sunion-with*[*param*]:
fixes *less*
shows ⟦ (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*;
(*f*,*f'*)∈(*Ra*→*Rb*→*Rb*→*Rb*); (*a*,*a'*)∈⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*;
(*b*,*b'*)∈⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel* ⟧
⟹ (*ord.sunion-with less f a b*, *ord.sunion-with less' f' a' b'*) ∈
⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *skip-red-alt*:
RBT-Impl.skip-red t = (*case t of*
(*Branch color.R l k v r*) ⇒ *l*
| - ⇒ *t*)
⟨*proof*⟩

function *compare-height* ::
('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ *RBT-Impl.compare*
where
compare-height sx s t tx =

(*case* (*RBT-Impl.skip-red sx*, *RBT-Impl.skip-red s*, *RBT-Impl.skip-red t*, *RBT-Impl.skip-red tx*) *of*
(*Branch - sx' - - -*, *Branch - s' - - -*, *Branch - t' - - -*, *Branch - tx' - - -*) ⇒
compare-height (*RBT-Impl.skip-black sx'*) *s' t'* (*RBT-Impl.skip-black tx'*)
| (-, *rbt.Empty*, -, *Branch - - - - -*) ⇒ *RBT-Impl.LT*
| (*Branch - - - - -*, -, *rbt.Empty*, -) ⇒ *RBT-Impl.GT*
| (*Branch - sx' - - -*, *Branch - s' - - -*, *Branch - t' - - -*, *rbt.Empty*) ⇒
compare-height (*RBT-Impl.skip-black sx'*) *s' t' rbt.Empty*
| (*rbt.Empty*, *Branch - s' - - -*, *Branch - t' - - -*, *Branch - tx' - - -*) ⇒
compare-height rbt.Empty s' t' (*RBT-Impl.skip-black tx'*)
| - ⇒ *RBT-Impl.EQ*)
⟨*proof*⟩

lemma *skip-red-size*: *size* (*RBT-Impl.skip-red b*) ≤ *size b*
⟨*proof*⟩

lemma *skip-black-size*: *size* (*RBT-Impl.skip-black b*) ≤ *size b*
⟨*proof*⟩

termination
⟨*proof*⟩

lemmas [*simp del*] = *compare-height.simps*

lemma *compare-height-alt*:
RBT-Impl.compare-height sx s t tx = *compare-height sx s t tx*
⟨*proof*⟩

term *RBT-Impl.skip-red*
lemma *param-skip-red*[*param*]: (*RBT-Impl.skip-red*,*RBT-Impl.skip-red*)
∈ ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-skip-black*[*param*]: (*RBT-Impl.skip-black*,*RBT-Impl.skip-black*)
∈ ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
⟨*proof*⟩

term *case-rbt*
lemma *param-case-rbt'*:
assumes (*t*,*t'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*
assumes ⟦*t*=*rbt.Empty*; *t'*=*rbt.Empty*⟧ ⟹ (*fl*,*fl'*)∈*R*
assumes ⋀*c l k v r c' l' k' v' r'*. ⟦
t = *Branch c l k v r*; *t'* = *Branch c' l' k' v' r'*;
(*c*,*c'*)∈*color-rel*;
(*l*,*l'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*k*,*k'*)∈*Rk*; (*v*,*v'*)∈*Rv*; (*r*,*r'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*
⟧ ⟹ (*fb c l k v r*, *fb' c' l' k' v' r'*) ∈ *R*
shows (*case-rbt fl fb t*, *case-rbt fl' fb' t'*) ∈ *R*
⟨*proof*⟩

lemma *compare-height-param-aux*[*param*]:
⟦ (*sx*,*sx'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*s*,*s'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*;
(*t*,*t'*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*tx*,*tx'*)∈⟨*Rk*,*Rv*⟩*rbt-rel* ⟧
⟹ (*compare-height sx s t tx*, *compare-height sx' s' t' tx'*) ∈ *compare-rel*
⟨*proof*⟩

lemma *compare-height-param*[*param*]:
(*RBT-Impl.compare-height*,*RBT-Impl.compare-height*) ∈
⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
→ *compare-rel*
⟨*proof*⟩

lemma *rbt-rel-bheight*: (*t*, *t'*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel* ⟹ *bheight t* = *bheight t'*
⟨*proof*⟩

lemma *param-rbt-baliL*[*param*]: (*rbt-baliL*,*rbt-baliL*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb*
→ ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-baliR*[*param*]: (*rbt-baliR*,*rbt-baliR*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb*
→ ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-joinL*[*param*]: (*rbt-joinL*,*rbt-joinL*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* →
Rb → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-joinR*[*param*]:
(*rbt-joinR*,*rbt-joinR*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-join*[*param*]: (*rbt-join*,*rbt-join*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* →
⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-split*[*param*]:
fixes *less*
assumes [*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-split less*, *ord.rbt-split less'*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → ⟨⟨*Ra*,*Rb*⟩*rbt-rel*,⟨⟨*Rb*⟩*option-rel*,⟨*Ra*,*Rb*⟩*rbt-re*
⟨*proof*⟩

lemma *param-rbt-union-swap-rec*[*param*]:
fixes *less*
assumes [*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-union-swap-rec less*, *ord.rbt-union-swap-rec less'*) ∈
(*Ra* → *Rb* → *Rb* → *Rb*) → *Id* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

lemma *param-rbt-union*[*param*]:

fixes *less*
assumes *param-less*[*param*]: (*less*,*less'*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-union less*, *ord.rbt-union less'*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
⟨*proof*⟩

term *rm-iterateoi*
lemma *param-rm-iterateoi*[*param*]: (*rm-iterateoi*,*rm-iterateoi*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → (*Rc*→*Id*) → (⟨*Ra*,*Rb*⟩*prod-rel* → *Rc* → *Rc*) → *Rc* → *Rc*
⟨*proof*⟩

lemma *param-rm-reverse-iterateoi*[*param*]:
(*rm-reverse-iterateoi*,*rm-reverse-iterateoi*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → (*Rc*→*Id*) → (⟨*Ra*,*Rb*⟩*prod-rel* → *Rc* → *Rc*) → *Rc* → *Rc*
⟨*proof*⟩

lemma *param-color-eq*[*param*]:
((=), (=))∈*color-rel*→*color-rel*→*Id*
⟨*proof*⟩

lemma *param-color-of*[*param*]:
(*color-of*, *color-of*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*color-rel*
⟨*proof*⟩

term *bheight*
lemma *param-bheight*[*param*]:
(*bheight*,*bheight*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
⟨*proof*⟩

lemma *inv1-param*[*param*]: (*inv1*,*inv1*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
⟨*proof*⟩

lemma *inv2-param*[*param*]: (*inv2*,*inv2*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
⟨*proof*⟩

term *ord.rbt-less*
lemma *rbt-less-param*[*param*]: (*ord.rbt-less*,*ord.rbt-less*) ∈
(*Rk*→*Rk*→*Id*) → *Rk* → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
⟨*proof*⟩

term *ord.rbt-greater*
lemma *rbt-greater-param*[*param*]: (*ord.rbt-greater*,*ord.rbt-greater*) ∈
(*Rk*→*Rk*→*Id*) → *Rk* → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
⟨*proof*⟩

lemma *rbt-sorted-param*[*param*]:
(*ord.rbt-sorted*,*ord.rbt-sorted*)∈(*Rk*→*Rk*→*Id*)→⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
⟨*proof*⟩

lemma *is-rbt-param*[*param*]: (*ord.is-rbt*,*ord.is-rbt*) ∈
(*Rk*→*Rk*→*Id*) → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
⟨*proof*⟩

definition *rbt-map-rel′ lt* = *br* (*ord.rbt-lookup lt*) (*ord.is-rbt lt*)

lemma (**in** *linorder*) *rbt-map-impl*:
(*rbt.Empty*,*Map.empty*) ∈ *rbt-map-rel′* (<)
(*rbt-insert*,λ*k v m*. *m*(*k*↦*v*))
∈ *Id* → *Id* → *rbt-map-rel′* (<) → *rbt-map-rel′* (<)
(*rbt-lookup*,λ*m k*. *m k*) ∈ *rbt-map-rel′* (<) → *Id* → ⟨*Id*⟩*option-rel*
(*rbt-delete*,λ*k m*. *m*|‘(−{*k*})) ∈ *Id* → *rbt-map-rel′* (<) → *rbt-map-rel′* (<)
(*rbt-union*,(++))
∈ *rbt-map-rel′* (<) → *rbt-map-rel′* (<) → *rbt-map-rel′* (<)
⟨*proof*⟩

lemma *sorted-wrt-keys-true*[*simp*]: *sorted-wrt* (λ(-,-) (-,-). *True*) *l*
⟨*proof*⟩

definition *rbt-map-rel-def-internal*:
rbt-map-rel lt Rk Rv ≡ ⟨*Rk*,*Rv*⟩*rbt-rel O rbt-map-rel′ lt*

lemma *rbt-map-rel-def*:
⟨*Rk*,*Rv*⟩*rbt-map-rel lt* ≡ ⟨*Rk*,*Rv*⟩*rbt-rel O rbt-map-rel′ lt*
⟨*proof*⟩

lemma (**in** *linorder*) *autoref-gen-rbt-empty*:
(*rbt.Empty*,*Map.empty*) ∈ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
⟨*proof*⟩

lemma (**in** *linorder*) *autoref-gen-rbt-insert*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,(<)) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-insert less-impl*,λ*k v m*. *m*(*k*↦*v*)) ∈
Rk → *Rv* → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
⟨*proof*⟩

lemma (**in** *linorder*) *autoref-gen-rbt-lookup*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,(<)) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-lookup less-impl*, λ*m k*. *m k*) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → *Rk* → ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma (**in** *linorder*) *autoref-gen-rbt-delete*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,(<)) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-delete less-impl*, λ*k m*. *m* |‘(−{*k*})) ∈
Rk → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
⟨*proof*⟩

lemma (**in** *linorder*) *autoref-gen-rbt-union*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,(<)) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-union less-impl*, (++)) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
⟨*proof*⟩

A linear ordering on red-black trees

abbreviation *rbt-to-list t* ≡ *it-to-list rm-iterateoi t*

lemma (**in** *linorder*) *rbt-to-list-correct*:
assumes *SORTED*: *rbt-sorted t*
shows *rbt-to-list t* = *sorted-list-of-map* (*rbt-lookup t*) (**is** *?tl* = -)
⟨*proof*⟩

definition
cmp-rbt cmpk cmpv ≡ *cmp-img rbt-to-list* (*cmp-lex* (*cmp-prod cmpk cmpv*))

lemma (**in** *linorder*) *param-rbt-sorted-list-of-map*[*param*]:
shows (*rbt-to-list*, *sorted-list-of-map*) ∈
⟨*Rk*, *Rv*⟩*rbt-map-rel* (<) → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *param-rbt-sorted-list-of-map′*[*param*]:
assumes *ELO*: *eq-linorder cmp′*
shows (*rbt-to-list*,*linorder.sorted-list-of-map* (*comp2le cmp′*)) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
⟨*proof*⟩

lemma *rbt-linorder-impl*:
assumes *ELO*: *eq-linorder cmp′*
assumes [*param*]: (*cmp*,*cmp′*)∈*Rk*→*Rk*→*Id*
shows
(*cmp-rbt cmp*, *cmp-map cmp′*) ∈
(*Rv*→*Rv*→*Id*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → *Id*
⟨*proof*⟩

lemma *color-rel-sv*[*relator-props*]: *single-valued color-rel*
⟨*proof*⟩

lemma *rbt-rel-sv-aux*:
assumes *SK*: *single-valued Rk*
assumes *SV*: *single-valued Rv*
assumes *I1*: (*a*,*b*)∈(⟨*Rk*, *Rv*⟩*rbt-rel*)
assumes *I2*: (*a*,*c*)∈(⟨*Rk*, *Rv*⟩*rbt-rel*)
shows *b*=*c*
⟨*proof*⟩

lemma *rbt-rel-sv*[*relator-props*]:
assumes *SK*: *single-valued Rk*
assumes *SV*: *single-valued Rv*
shows *single-valued* (⟨*Rk*, *Rv*⟩*rbt-rel*)
⟨*proof*⟩

lemma *rbt-map-rel-sv*[*relator-props*]:
⟦*single-valued Rk*; *single-valued Rv*⟧
⟹ *single-valued* (⟨*Rk*,*Rv*⟩*rbt-map-rel lt*)
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of rbt-map-rel x i-map*] **for** *x*

Second Part: Binding

lemma *autoref-rbt-empty*[*autoref-rules*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
shows (*rbt.Empty*,*op-map-empty*) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
⟨*proof*⟩

lemma *autoref-rbt-update*[*autoref-rules*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
shows (*ord.rbt-insert* (*comp2lt cmp*),*op-map-update*) ∈
Rk→*Rv*→⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
⟨*proof*⟩

lemma *autoref-rbt-lookup*[*autoref-rules*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
shows (λ*k t*. *ord.rbt-lookup* (*comp2lt cmp*) *t k*, *op-map-lookup*) ∈
Rk → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨*Rv*⟩*option-rel*
⟨*proof*⟩

lemma *autoref-rbt-delete*[*autoref-rules*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)

shows (*ord.rbt-delete* (*comp2lt cmp*),*op-map-delete*) ∈
Rk → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
⟨*proof*⟩

lemma *autoref-rbt-union*[*autoref-rules*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
shows (*ord.rbt-union* (*comp2lt cmp*),(++)) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
⟨*proof*⟩

lemma *autoref-rbt-is-iterator*[*autoref-ga-rules*]:
assumes *ELO*: *GEN-ALGO-tag* (*eq-linorder cmp′*)
shows *is-map-to-sorted-list* (*comp2le cmp′*) *Rk Rv* (*rbt-map-rel* (*comp2lt cmp′*))
rbt-to-list
⟨*proof*⟩

lemmas [*autoref-ga-rules*] = *class-to-eq-linorder*

lemma (**in** *linorder*) *dflt-cmp-id*:
(*dflt-cmp* (≤) (<), *dflt-cmp* (≤) (<))∈*Id*→*Id*→*Id*
⟨*proof*⟩

lemmas [*autoref-rules*] = *dflt-cmp-id*

lemma *rbt-linorder-autoref*[*autoref-rules*]:
assumes *SIDE-GEN-ALGO* (*eq-linorder cmpk′*)
assumes *SIDE-GEN-ALGO* (*eq-linorder cmpv′*)
assumes *GEN-OP cmpk cmpk′* (*Rk*→*Rk*→*Id*)
assumes *GEN-OP cmpv cmpv′* (*Rv*→*Rv*→*Id*)
shows
(*cmp-rbt cmpk cmpv*, *cmp-map cmpk′ cmpv′*) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk′*) → *Id*
⟨*proof*⟩

lemma *map-linorder-impl*[*autoref-ga-rules*]:
assumes *GEN-ALGO-tag* (*eq-linorder cmpk*)
assumes *GEN-ALGO-tag* (*eq-linorder cmpv*)
shows *eq-linorder* (*cmp-map cmpk cmpv*)
⟨*proof*⟩

lemma *set-linorder-impl*[*autoref-ga-rules*]:
assumes *GEN-ALGO-tag* (*eq-linorder cmpk*)

shows *eq-linorder* (*cmp-set cmpk*)
⟨*proof*⟩

lemma (**in** *linorder*) *rbt-map-rel-finite-aux*:
finite-map-rel (⟨*Rk*,*Rv*⟩*rbt-map-rel* (<))
⟨*proof*⟩

lemma *rbt-map-rel-finite*[*relator-props*]:
assumes *ELO*: *GEN-ALGO-tag* (*eq-linorder cmpk*)
shows *finite-map-rel* (⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk*))
⟨*proof*⟩

abbreviation
dflt-rm-rel ≡ *rbt-map-rel* (*comp2lt* (*dflt-cmp* (≤) (<)))

lemmas [*autoref-post-simps*] = *dflt-cmp-inv2 dflt-cmp-2inv*

lemma [*simp*,*autoref-post-simps*]: *ord.rbt-ins* (<) = *rbt-ins*
⟨*proof*⟩

lemma [*autoref-post-simps*]: *ord.rbt-lookup* ((<)::-::*linorder*⇒-) = *rbt-lookup*
⟨*proof*⟩

lemma [*simp*,*autoref-post-simps*]:
ord.rbt-insert-with-key (<) = *rbt-insert-with-key*
ord.rbt-insert (<) = *rbt-insert*
⟨*proof*⟩

lemma *autoref-comp2eq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *ELC*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*R*→*R*→*Id*)
shows (*comp2eq cmp*, (=)) ∈ *R*→*R*→*Id*
⟨*proof*⟩

lemma *pi′-rm*[*icf-proper-iteratorI*]:
proper-it′ rm-iterateoi rm-iterateoi
proper-it′ rm-reverse-iterateoi rm-reverse-iterateoi
⟨*proof*⟩

declare *pi′-rm*[*proper-it*]

lemmas *autoref-rbt-rules* =
autoref-rbt-empty
autoref-rbt-lookup
autoref-rbt-update
autoref-rbt-delete

autoref-rbt-union

lemmas *autoref-rbt-rules-linorder*[*autoref-rules-raw*] =
autoref-rbt-rules[**where** *Rk*=*Rk*] **for** *Rk* :: (-×-::*linorder*) *set*

end

2.3.6 Set by Characteristic Function

theory *Impl-Cfun-Set*
imports *../Intf/Intf-Set*
begin

definition *fun-set-rel* **where**
fun-set-rel-internal-def:
fun-set-rel R ≡ (*R*→*bool-rel*) *O br Collect* (λ-. *True*)

lemma *fun-set-rel-def*: ⟨*R*⟩*fun-set-rel* = (*R*→*bool-rel*) *O br Collect* (λ-. *True*)
⟨*proof*⟩

lemma *fun-set-rel-sv*[*relator-props*]:
⟦*single-valued R*; *Range R* = *UNIV*⟧ ⟹ *single-valued* (⟨*R*⟩*fun-set-rel*)
⟨*proof*⟩

lemma *fun-set-rel-RUNIV*[*relator-props*]:
assumes *SV*: *single-valued R*
shows *Range* (⟨*R*⟩*fun-set-rel*) = *UNIV*
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of fun-set-rel i-set*]

lemma *fs-mem-refine*[*autoref-rules*]: (λ*x f*. *f x*,(∈)) ∈ *R* → ⟨*R*⟩*fun-set-rel* → *bool-rel*
⟨*proof*⟩

lemma *fun-set-Collect-refine*[*autoref-rules*]:
(λ*x*. *x*, *Collect*)∈(*R*→*bool-rel*) → ⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

lemma *fun-set-empty-refine*[*autoref-rules*]:
(λ-. *False*,{})∈⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

lemma *fun-set-UNIV-refine*[*autoref-rules*]:
(λ-. *True*, *UNIV*)∈⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

lemma *fun-set-union-refine*[*autoref-rules*]:
(λ*a b x*. *a x* ∨ *b x*,(∪))∈⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

lemma *fun-set-inter-refine*[*autoref-rules*]:
(λ*a b x*. *a x* ∧ *b x*,(∩))∈⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

lemma *fun-set-diff-refine*[*autoref-rules*]:
(λ*a b x*. *a x* ∧ ¬*b x*,(−))∈⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel* → ⟨*R*⟩*fun-set-rel*
⟨*proof*⟩

end

2.3.7 Array-Based Maps with Natural Number Keys

theory *Impl-Array-Map*
imports
Automatic-Refinement.Automatic-Refinement
../../Lib/Diff-Array
../../Iterator/Iterator
../Gen/Gen-Map
../Intf/Intf-Comp
../Intf/Intf-Map
begin

type-synonym *'v iam* = *'v option array*

Definitions

definition *iam-α* :: *'v iam* ⇒ *nat* ⇀ *'v* **where**
iam-α a i ≡ *if i* < *array-length a then array-get a i else None*

lemma [*code*]: *iam-α a i* ≡ *array-get-oo None a i*
⟨*proof*⟩

abbreviation *iam-invar* :: *'v iam* ⇒ *bool* **where** *iam-invar* ≡ λ-. *True*

definition *iam-empty* :: *unit* ⇒ *'v iam*
where *iam-empty* ≡ λ-::*unit*. *array-of-list* []

definition *iam-lookup* :: *nat* ⇒ *'v iam* ⇀ *'v*
where [*code-unfold*]: *iam-lookup k a* ≡ *iam-α a k*

definition *iam-increment* (*l*::*nat*) *idx* ≡
max (*idx* + *1* − *l*) (*2* ∗ *l* + *3*)

lemma *incr-correct*: ¬ *idx* < *l* ⟹ *idx* < *l* + *iam-increment l idx*
⟨*proof*⟩

definition *iam-update* :: *nat* ⇒ *'v* ⇒ *'v iam* ⇒ *'v iam*
where *iam-update k v a* ≡ *let*
l = *array-length a*;
a = *if k* < *l then a else array-grow a* (*iam-increment l k*) *None*
in
array-set a k (*Some v*)

lemma [*code*]: *iam-update k v a* ≡ *array-set-oo*
(λ-. *let l*=*array-length a in*
array-set (*array-grow a* (*iam-increment l k*) *None*) *k* (*Some v*))
a k (*Some v*)
⟨*proof*⟩

definition *iam-delete* :: *nat* ⇒ *'v iam* ⇒ *'v iam*
where *iam-delete k a* ≡
if k<*array-length a then array-set a k None else a*

lemma [*code*]: *iam-delete k a* ≡ *array-set-oo* (λ-. *a*) *a k None*
⟨*proof*⟩

primrec *iam-iteratei-aux*
:: *nat* ⇒ (*'v iam*) ⇒ (*'σ*⇒*bool*) ⇒ (*nat* × *'v*⇒*'σ*⇒*'σ*) ⇒ *'σ* ⇒ *'σ*
where
iam-iteratei-aux 0 a c f σ = *σ*
| *iam-iteratei-aux* (*Suc i*) *a c f σ* = (
if c σ then
iam-iteratei-aux i a c f (
case array-get a i of None ⇒ *σ* | *Some x* ⇒ *f* (*i*, *x*) *σ*
)
else σ)

definition *iam-iteratei* :: *'v iam* ⇒ (*nat* × *'v*,*'σ*) *set-iterator* **where**
iam-iteratei a = *iam-iteratei-aux* (*array-length a*) *a*

Parametricity

definition *iam-rel-def-internal*:
iam-rel R ≡ ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
lemma *iam-rel-def*: ⟨*R*⟩ *iam-rel* = ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
⟨*proof*⟩

lemma *iam-rel-sv*[*relator-props*]:
single-valued Rv ⟹ *single-valued* (⟨*Rv*⟩*iam-rel*)
⟨*proof*⟩

lemma *param-iam-α*[*param*]:
(*iam-α*, *iam-α*) ∈ ⟨*R*⟩ *iam-rel* → *nat-rel* → ⟨*R*⟩ *option-rel*
⟨*proof*⟩

lemma *param-iam-invar*[*param*]:
(*iam-invar*, *iam-invar*) ∈ ⟨*R*⟩ *iam-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-iam-empty*[*param*]:
(*iam-empty*, *iam-empty*) ∈ *unit-rel* → ⟨*R*⟩*iam-rel*
⟨*proof*⟩

lemma *param-iam-lookup*[*param*]:
(*iam-lookup*, *iam-lookup*) ∈ *nat-rel* → ⟨*R*⟩*iam-rel* → ⟨*R*⟩*option-rel*
⟨*proof*⟩

lemma *param-iam-increment*[*param*]:
(*iam-increment*, *iam-increment*) ∈ *nat-rel* → *nat-rel* → *nat-rel*
⟨*proof*⟩

lemma *param-iam-update*[*param*]:
(*iam-update*, *iam-update*) ∈ *nat-rel* → *R* → ⟨*R*⟩*iam-rel* → ⟨*R*⟩*iam-rel*
⟨*proof*⟩

lemma *param-iam-delete*[*param*]:
(*iam-delete*, *iam-delete*) ∈ *nat-rel* → ⟨*R*⟩*iam-rel* → ⟨*R*⟩*iam-rel*
⟨*proof*⟩

lemma *param-iam-iteratei-aux*[*param*]:
assumes *I*: *i* ≤ *array-length a*
assumes *IR*: (*i*,*i'*) ∈ *nat-rel*
assumes *AR*: (*a*,*a'*) ∈ ⟨*Ra*⟩*iam-rel*
assumes *CR*: (*c*,*c'*) ∈ *Rb* → *bool-rel*
assumes *FR*: (*f*,*f'*) ∈ ⟨*nat-rel*,*Ra*⟩*prod-rel* → *Rb* → *Rb*
assumes *σR*: (*σ*,*σ'*) ∈ *Rb*
shows (*iam-iteratei-aux i a c f σ*, *iam-iteratei-aux i' a' c' f' σ'*) ∈ *Rb*
⟨*proof*⟩

lemma *param-iam-iteratei*[*param*]:
(*iam-iteratei*,*iam-iteratei*) ∈ ⟨*Ra*⟩*iam-rel* → (*Rb* → *bool-rel*) →
(⟨*nat-rel*,*Ra*⟩*prod-rel* → *Rb* → *Rb*) → *Rb* → *Rb*
⟨*proof*⟩

Correctness

definition *iam-rel'* ≡ *br iam-α iam-invar*

lemma *iam-empty-correct*:
(*iam-empty* (), *Map.empty*) ∈ *iam-rel'*
⟨*proof*⟩

lemma *iam-update-correct*:
(*iam-update*,*op-map-update*) ∈ *nat-rel* → *Id* → *iam-rel′* → *iam-rel′*
⟨*proof*⟩

lemma *iam-lookup-correct*:
(*iam-lookup*,*op-map-lookup*) ∈ *Id* → *iam-rel′* → ⟨*Id*⟩*option-rel*
⟨*proof*⟩

lemma *array-get-set-iff*: *i*<*array-length a* ⟹
array-get (*array-set a i x*) *j* = (*if i*=*j then x else array-get a j*)
⟨*proof*⟩

lemma *iam-delete-correct*:
(*iam-delete*,*op-map-delete*) ∈ *Id* → *iam-rel′* → *iam-rel′*
⟨*proof*⟩

definition *iam-map-rel-def-internal*:
iam-map-rel Rk Rv ≡
if Rk=*nat-rel then* ⟨*Rv*⟩*iam-rel O iam-rel′ else* {}
lemma *iam-map-rel-def*:
⟨*nat-rel*,*Rv*⟩*iam-map-rel* ≡ ⟨*Rv*⟩*iam-rel O iam-rel′*
⟨*proof*⟩

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of iam-map-rel i-map*]

lemma *iam-map-rel-sv*[*relator-props*]:
single-valued Rv ⟹ *single-valued* (⟨*nat-rel*,*Rv*⟩*iam-map-rel*)
⟨*proof*⟩

lemma *iam-empty-impl*:
(*iam-empty* (), *op-map-empty*) ∈ ⟨*nat-rel*,*R*⟩*iam-map-rel*
⟨*proof*⟩

lemma *iam-lookup-impl*:
(*iam-lookup*, *op-map-lookup*)
∈ *nat-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*R*⟩*option-rel*
⟨*proof*⟩

lemma *iam-update-impl*:
(*iam-update*, *op-map-update*) ∈
nat-rel → *R* → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel*
⟨*proof*⟩

lemma *iam-delete-impl*:
(*iam-delete*, *op-map-delete*) ∈

nat-rel → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel*
⟨*proof*⟩

lemmas *iam-map-impl* =
iam-empty-impl
iam-lookup-impl
iam-update-impl
iam-delete-impl

declare *iam-map-impl*[*autoref-rules*]

Iterator proofs

abbreviation *iam-to-list a* ≡ *it-to-list iam-iteratei a*

lemma *distinct-iam-to-list-aux*:
shows ⟦*distinct xs*; ∀(*i*,-)∈*set xs*. *i* ≥ *n*⟧ ⟹
distinct (*iam-iteratei-aux n a*
(λ-. *True*) (λ*x y*. *y* @ [*x*]) *xs*)
(**is** ⟦-;-⟧ ⟹ *distinct* (*?iam-to-list-aux n xs*))
⟨*proof*⟩

lemma *distinct-iam-to-list*:
distinct (*iam-to-list a*)
⟨*proof*⟩

lemma *iam-to-list-set-correct-aux*:
assumes (*a*, *m*) ∈ *iam-rel′*
shows ⟦*n* ≤ *array-length a*; *map-to-set m* − {(*k*,*v*). *k* < *n*} = *set xs*⟧
⟹ *map-to-set m* =
set (*iam-iteratei-aux n a* (λ-. *True*) (λ*x y*. *y* @ [*x*]) *xs*)
⟨*proof*⟩

lemma *iam-to-list-set-correct*:
assumes (*a*, *m*) ∈ *iam-rel′*
shows *map-to-set m* = *set* (*iam-to-list a*)
⟨*proof*⟩

lemma *iam-iteratei-aux-append*:
n ≤ *length xs* ⟹ *iam-iteratei-aux n* (*Array* (*xs* @ *ys*)) =
iam-iteratei-aux n (*Array xs*)
⟨*proof*⟩

lemma *iam-iteratei-append*:
iam-iteratei (*Array* (*xs* @ [*None*])) *c f* σ =
iam-iteratei (*Array xs*) *c f* σ
iam-iteratei (*Array* (*xs* @ [*Some x*])) *c f* σ =
iam-iteratei (*Array xs*) *c f*
(*if c* σ *then* (*f* (*length xs*, *x*) σ) *else* σ)

⟨*proof*⟩

lemma *iam-iteratei-aux-Cons*:
n < *array-length a* ⟹
iam-iteratei-aux n a (λ-. *True*) (λ*x l*. *l* @ [*x*]) (*x*#*xs*) =
x # *iam-iteratei-aux n a* (λ-. *True*) (λ*x l*. *l* @ [*x*]) *xs*
⟨*proof*⟩

lemma *iam-to-list-append*:
iam-to-list (*Array* (*xs* @ [*None*])) = *iam-to-list* (*Array xs*)
iam-to-list (*Array* (*xs* @ [*Some x*])) =
(*length xs*, *x*) # *iam-to-list* (*Array xs*)
⟨*proof*⟩

lemma *autoref-iam-is-iterator*[*autoref-ga-rules*]:
shows *is-map-to-list nat-rel Rv iam-map-rel iam-to-list*
⟨*proof*⟩

lemmas [*autoref-ga-rules*] =
autoref-iam-is-iterator[*unfolded is-map-to-list-def*]

lemma *iam-iteratei-altdef*:
iam-iteratei a = *foldli* (*iam-to-list a*)
(**is** *?f a* = *?g* (*iam-to-list a*))
⟨*proof*⟩

lemma *pi-iam*[*icf-proper-iteratorI*]:
proper-it (*iam-iteratei a*) (*iam-iteratei a*)
⟨*proof*⟩

lemma *pi′-iam*[*icf-proper-iteratorI*]:
proper-it′ iam-iteratei iam-iteratei
⟨*proof*⟩

end

Chapter 3

The Original Isabelle Collection Framework

This chapter contains the original Isabelle Collection Framework. It contains a vast amount of verified collection data structures, that are included either directly or by parameterization via locales.

Generic algorithms need to be instantiated manually, and nesting of collections (e.g. sets of sets) is not supported.

3.1 Specifications

3.1.1 Specification of Sets

theory *SetSpec*
imports *ICF-Spec-Base*
begin

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

notation *insert* (‹*set'-ins*›)

type-synonym ('*x*,'*s*) *set-α* = '*s* ⇒ '*x set*
type-synonym ('*x*,'*s*) *set-invar* = '*s* ⇒ *bool*
locale *set* =
  — Abstraction to set
  fixes *α* :: '*s* ⇒ '*x set*
  — Invariant
  fixes *invar* :: '*s* ⇒ *bool*

locale *set-no-invar* = *set* +
  assumes *invar*[*simp*, *intro*!]: ⋀*s*. *invar s*

Basic Set Functions

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

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

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

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

Deletion of Single Element **locale** *set-delete* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *delete* :: $'x \Rightarrow 's \Rightarrow 's$
assumes *delete-correct*:
$invar\ s \Longrightarrow \alpha\ (delete\ x\ s) = \alpha\ s - \{x\}$
$invar\ s \Longrightarrow invar\ (delete\ x\ s)$

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

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

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

fixes *bex* :: $'s \Rightarrow ('x \Rightarrow bool) \Rightarrow bool$
assumes *bex-correct*: $invar\ S \Longrightarrow bex\ S\ P \longleftrightarrow (\exists\, x{\in}\alpha\ S.\ P\ x)$

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

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

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

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

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

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

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.

type-synonym $('x,'s)$ *set-list-it*

$= 's \Rightarrow ('x,'x\ list)\ set\text{-}iterator$
locale *poly-set-iteratei-defs* =
fixes *list-it* :: $'s \Rightarrow ('x,'x\ list)\ set\text{-}iterator$
begin
definition *iteratei* :: $'s \Rightarrow ('x,'\sigma)\ set\text{-}iterator$
where $iteratei\ S \equiv it\text{-}to\text{-}it\ (list\text{-}it\ S)$

abbreviation $iterate\ s \equiv iteratei\ s\ (\lambda\text{-}.\ True)$
end

locale *poly-set-iteratei* =
finite-set + *poly-set-iteratei-defs list-it*
for *list-it* :: $'s \Rightarrow ('x,'x\ list)\ set\text{-}iterator$ +
constrains $\alpha :: 's \Rightarrow 'x\ set$
assumes *list-it-correct*: $invar\ s \Longrightarrow set\text{-}iterator\ (list\text{-}it\ s)\ (\alpha\ s)$
begin
lemma *iteratei-correct*: $invar\ S \Longrightarrow set\text{-}iterator\ (iteratei\ S)\ (\alpha\ S)$
⟨*proof*⟩

lemma *pi-iteratei*[*icf-proper-iteratorI*]:
proper-it (*iteratei S*) (*iteratei S*)
⟨*proof*⟩

lemma *iteratei-rule-P*:
$[\![$
$invar\ S;$
$I\ (\alpha\ S)\ \sigma 0;$
$!!x\ it\ \sigma.\ [\![\ c\ \sigma;\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ ]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);$
$!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;$
$!!\sigma\ it.\ [\![\ it \subseteq \alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
$]\!] \Longrightarrow P\ (iteratei\ S\ c\ f\ \sigma 0)$
⟨*proof*⟩

lemma *iteratei-rule-insert-P*:
$[\![$
$invar\ S;$
$I\ \{\}\ \sigma 0;$
$!!x\ it\ \sigma.\ [\![\ c\ \sigma;\ x \in \alpha\ S - it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ ]\!] \Longrightarrow I\ (insert\ x\ it)\ (f\ x\ \sigma);$
$!!\sigma.\ I\ (\alpha\ S)\ \sigma \Longrightarrow P\ \sigma;$
$!!\sigma\ it.\ [\![\ it \subseteq \alpha\ S;\ it \neq \alpha\ S;\ \neg\ c\ \sigma;\ I\ it\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
$]\!] \Longrightarrow P\ (iteratei\ S\ c\ f\ \sigma 0)$
⟨*proof*⟩

Versions without break condition.

lemma *iterate-rule-P*:
$[\![$
$invar\ S;$
$I\ (\alpha\ S)\ \sigma 0;$

!!$x\ it\ \sigma$. $[\![\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ ]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$;
!!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
$]\!] \Longrightarrow P\ (iteratei\ S\ (\lambda\text{-}.\ True)\ f\ \sigma 0)$
⟨*proof*⟩

lemma *iterate-rule-insert-P*:
$[\![$
invar S;
$I\ \{\}\ \sigma 0$;
!!$x\ it\ \sigma$. $[\![\ x \in \alpha\ S - it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ ]\!] \Longrightarrow I\ (insert\ x\ it)\ (f\ x\ \sigma)$;
!!σ. $I\ (\alpha\ S)\ \sigma \Longrightarrow P\ \sigma$
$]\!] \Longrightarrow P\ (iteratei\ S\ (\lambda\text{-}.\ True)\ f\ \sigma 0)$
⟨*proof*⟩

end

More Set Operations

Copy **locale** *set-copy = s1*: *set α1 invar1 + s2*: *set α2 invar2*
for *α1* :: *'s1 ⇒ 'a set* **and** *invar1*
and *α2* :: *'s2 ⇒ 'a set* **and** *invar2*
+
fixes *copy* :: *'s1 ⇒ 's2*
assumes *copy-correct*:
invar1 s1 $\Longrightarrow$ *α2 (copy s1) = α1 s1*
invar1 s1 $\Longrightarrow$ *invar2 (copy s1)*

Union **locale** *set-union = s1*: *set α1 invar1 + s2*: *set α2 invar2 + s3*: *set α3 invar3*
for *α1* :: *'s1 ⇒ 'a set* **and** *invar1*
and *α2* :: *'s2 ⇒ 'a set* **and** *invar2*
and *α3* :: *'s3 ⇒ 'a set* **and** *invar3*
+
fixes *union* :: *'s1 ⇒ 's2 ⇒ 's3*
assumes *union-correct*:
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *α3 (union s1 s2) = α1 s1 ∪ α2 s2*
invar1 s1 $\Longrightarrow$ *invar2 s2* $\Longrightarrow$ *invar3 (union s1 s2)*

locale *set-union-dj* =
s1: *set α1 invar1 + s2*: *set α2 invar2 + s3*: *set α3 invar3*
for *α1* :: *'s1 ⇒ 'a set* **and** *invar1*
and *α2* :: *'s2 ⇒ 'a set* **and** *invar2*
and *α3* :: *'s3 ⇒ 'a set* **and** *invar3*
+
fixes *union-dj* :: *'s1 ⇒ 's2 ⇒ 's3*
assumes *union-dj-correct*:
$[\![$*invar1 s1*; *invar2 s2*; *α1 s1 ∩ α2 s2 = {}*$]\!] \Longrightarrow$ *α3 (union-dj s1 s2) = α1 s1 ∪ α2 s2*

$[\![invar1\ s1;\ invar2\ s2;\ \alpha1\ s1 \cap \alpha2\ s2 = \{\}]\!] \Longrightarrow invar3\ (union\text{-}dj\ s1\ s2)$

locale *set-union-list* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2*
for $\alpha1$ :: *'s1* ⇒ *'a set* **and** *invar1*
and $\alpha2$ :: *'s2* ⇒ *'a set* **and** *invar2*
+
fixes *union-list* :: *'s1 list* ⇒ *'s2*
assumes *union-list-correct*:
$\forall s1 \in set\ l.\ invar1\ s1 \Longrightarrow \alpha2\ (union\text{-}list\ l) = \bigcup \{\alpha1\ s1\ |s1.\ s1 \in set\ l\}$
$\forall s1 \in set\ l.\ invar1\ s1 \Longrightarrow invar2\ (union\text{-}list\ l)$

Difference **locale** *set-diff* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2*
for $\alpha1$ :: *'s1* ⇒ *'a set* **and** *invar1*
and $\alpha2$ :: *'s2* ⇒ *'a set* **and** *invar2*
+
fixes *diff* :: *'s1* ⇒ *'s2* ⇒ *'s1*
assumes *diff-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow \alpha1\ (diff\ s1\ s2) = \alpha1\ s1 - \alpha2\ s2$
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow invar1\ (diff\ s1\ s2)$

Intersection **locale** *set-inter* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2* + *s3*: *set* $\alpha3$ *invar3*
for $\alpha1$ :: *'s1* ⇒ *'a set* **and** *invar1*
and $\alpha2$ :: *'s2* ⇒ *'a set* **and** *invar2*
and $\alpha3$ :: *'s3* ⇒ *'a set* **and** *invar3*
+
fixes *inter* :: *'s1* ⇒ *'s2* ⇒ *'s3*
assumes *inter-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow \alpha3\ (inter\ s1\ s2) = \alpha1\ s1 \cap \alpha2\ s2$
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow invar3\ (inter\ s1\ s2)$

Subset **locale** *set-subset* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2*
for $\alpha1$ :: *'s1* ⇒ *'a set* **and** *invar1*
and $\alpha2$ :: *'s2* ⇒ *'a set* **and** *invar2*
+
fixes *subset* :: *'s1* ⇒ *'s2* ⇒ *bool*
assumes *subset-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow subset\ s1\ s2 \longleftrightarrow \alpha1\ s1 \subseteq \alpha2\ s2$

Equal **locale** *set-equal* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2*
for $\alpha1$ :: *'s1* ⇒ *'a set* **and** *invar1*
and $\alpha2$ :: *'s2* ⇒ *'a set* **and** *invar2*
+
fixes *equal* :: *'s1* ⇒ *'s2* ⇒ *bool*
assumes *equal-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow equal\ s1\ s2 \longleftrightarrow \alpha1\ s1 = \alpha2\ s2$

Image and Filter **locale** *set-image-filter* = *s1*: *set* $\alpha1$ *invar1* + *s2*: *set* $\alpha2$ *invar2*

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

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

end

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

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

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

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

lemma *disjoint-witnessI*: $[\![$
invar1 s1;
invar2 s2;

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

end

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

lemma *sel-someD*:
$[\![invar\ s;\ sel\ s\ f = Some\ r;\ !!x.\ [\![x{\in}\alpha\ s;\ f\ x = Some\ r]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *sel-noneD*:
$[\![invar\ s;\ sel\ s\ f = None;\ x{\in}\alpha\ s\ ]\!] \Longrightarrow f\ x = None$
⟨*proof*⟩
end

— Selection of element (without mapping)
locale *set-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′-someD*:
$[\![invar\ s;\ sel'\ s\ P = Some\ x\ ]\!] \Longrightarrow x{\in}\alpha\ s \wedge P\ x$
⟨*proof*⟩

lemma *sel′-noneD*:
$[\![invar\ s;\ sel'\ s\ P = None;\ x{\in}\alpha\ s\ ]\!] \Longrightarrow \neg P\ x$
⟨*proof*⟩
end

Conversion of Set to List **locale** *set-to-list* = *set* +
constrains $\alpha :: 's \Rightarrow 'x\ set$
fixes *to-list* :: $'s \Rightarrow 'x\ list$

assumes *to-list-correct*:
invar s $\Longrightarrow$ *set* (*to-list s*) = α *s*
invar s $\Longrightarrow$ *distinct* (*to-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*)

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** *poly-set-iterateoi-defs* =
fixes *olist-it* :: $'s \Rightarrow$ ($'x$,$'x$ *list*) *set-iterator*
begin
definition *iterateoi* :: $'s \Rightarrow$ ($'x$,$'\sigma$) *set-iterator*
where *iterateoi S* $\equiv$ *it-to-it* (*olist-it S*)

abbreviation *iterateo s* $\equiv$ *iterateoi s* (λ-. *True*)
end

locale *poly-set-iterateoi* =
finite-set α *invar* + *poly-set-iterateoi-defs list-ordered-it*
for α :: $'s \Rightarrow$ $'x$::*linorder set*
and *invar*
and *list-ordered-it* :: $'s \Rightarrow$ ($'x$,$'x$ *list*) *set-iterator* +
assumes *list-ordered-it-correct*: *invar x*
$\Longrightarrow$ *set-iterator-linord* (*list-ordered-it x*) (α *x*)
begin
lemma *iterateoi-correct*:
invar S $\Longrightarrow$ *set-iterator-linord* (*iterateoi S*) (α *S*)
⟨*proof*⟩

lemma *pi-iterateoi*[*icf-proper-iteratorI*]:
proper-it (*iterateoi S*) (*iterateoi S*)
⟨*proof*⟩

lemma *iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
assumes *MINV*: *invar s*
assumes *I0*: *I* (α *s*) σ*0*

assumes *IP*: !!k it σ. ⟦
$c\ \sigma$;
$k \in it$;
$\forall j{\in}it.\ k{\leq}j$;
$\forall j{\in}\alpha\ s - it.\ j{\leq}k$;
$it \subseteq \alpha\ s$;
$I\ it\ \sigma$
⟧ ⟹ $I\ (it - \{k\})\ (f\ k\ \sigma)$
assumes *IF*: !!σ. $I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
assumes *II*: !!σ it. ⟦
$it \subseteq \alpha\ s$;
$it \neq \{\}$;
$\neg\ c\ \sigma$;
$I\ it\ \sigma$;
$\forall k{\in}it.\ \forall j{\in}\alpha\ s - it.\ j{\leq}k$
⟧ ⟹ $P\ \sigma$
shows $P\ (iterateoi\ s\ c\ f\ \sigma 0)$
⟨*proof*⟩

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

locale *poly-set-rev-iterateoi-defs* =
fixes *list-rev-it* :: $'s \Rightarrow ('x{::}linorder, 'x\ list)\ set\text{-}iterator$
begin
definition *rev-iterateoi* :: $'s \Rightarrow ('x, '\sigma)\ set\text{-}iterator$
where *rev-iterateoi S* ≡ *it-to-it* (*list-rev-it S*)

abbreviation *rev-iterateo m* ≡ *rev-iterateoi m* (λ-. *True*)
abbreviation *reverse-iterateoi* ≡ *rev-iterateoi*
abbreviation *reverse-iterateo* ≡ *rev-iterateo*
end

locale *poly-set-rev-iterateoi* =
finite-set α *invar* + *poly-set-rev-iterateoi-defs list-rev-it*
for α :: $'s \Rightarrow 'x{::}linorder\ set$
and *invar*
and *list-rev-it* :: $'s \Rightarrow ('x, 'x\ list)\ set\text{-}iterator$ +
assumes *list-rev-it-correct*:
invar s ⟹ *set-iterator-rev-linord* (*list-rev-it s*) (α *s*)

begin
lemma *rev-iterateoi-correct*:
invar S $\Longrightarrow$ *set-iterator-rev-linord* (*rev-iterateoi S*) (α *S*)
⟨*proof*⟩

lemma *pi-rev-iterateoi*[*icf-proper-iteratorI*]:
proper-it (*rev-iterateoi S*) (*rev-iterateoi S*)
⟨*proof*⟩

lemma *rev-iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
assumes *MINV*: *invar s*
assumes *I0*: *I* ((α *s*)) σ*0*
assumes *IP*: !!*k it* σ. ⟦
c σ;
k ∈ *it*;
∀ *j*∈*it*. *k*≥*j*;
∀ *j*∈(α *s*) − *it*. *j*≥*k*;
it ⊆ (α *s*);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
assumes *II*: !!σ *it*. ⟦
it ⊆ (α *s*);
it ≠ {};
¬ *c* σ;
I it σ;
∀ *k*∈*it*. ∀ *j*∈(α *s*) − *it*. *j*≥*k*
⟧ $\Longrightarrow$ *P* σ
shows *P* (*rev-iterateoi s c f* σ*0*)
⟨*proof*⟩

lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar s*
assumes *I0*: *I* ((α *s*)) σ*0*
assumes *IP*: !!*k it* σ. ⟦
k ∈ *it*;
∀ *j*∈*it*. *k*≥*j*;
∀ *j*∈ (α *s*) − *it*. *j*≥*k*;
it ⊆ (α *s*);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
shows *P* (*rev-iterateo s f* σ*0*)
⟨*proof*⟩

end

Minimal and Maximal Element

locale *set-min* = *ordered-set* +
constrains α :: $'s \Rightarrow 'u$::*linorder set*

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

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

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

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

end

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

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

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

lemma *max-None*:

$[\![\mathit{invar}\ s;\ \mathit{max}\ s\ P = \mathit{None}\ ]\!] \Longrightarrow \{x{\in}\alpha\ s.\ P\ x\} = \{\}$
⟨*proof*⟩

end

Conversion to List

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

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

Record Based Interface

record $('x,'s)$ *set-ops* =
set-op-α :: $'s \Rightarrow 'x\ \mathit{set}$
set-op-invar :: $'s \Rightarrow \mathit{bool}$
set-op-empty :: $\mathit{unit} \Rightarrow 's$
set-op-memb :: $'x \Rightarrow 's \Rightarrow \mathit{bool}$
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-list-it :: $('x,'s)\ \mathit{set\text{-}list\text{-}it}$
set-op-sng :: $'x \Rightarrow 's$
set-op-isEmpty :: $'s \Rightarrow \mathit{bool}$
set-op-isSng :: $'s \Rightarrow \mathit{bool}$
set-op-ball :: $'s \Rightarrow ('x \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
set-op-bex :: $'s \Rightarrow ('x \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
set-op-size :: $'s \Rightarrow \mathit{nat}$
set-op-size-abort :: $\mathit{nat} \Rightarrow 's \Rightarrow \mathit{nat}$
set-op-union :: $'s \Rightarrow 's \Rightarrow 's$
set-op-union-dj :: $'s \Rightarrow 's \Rightarrow 's$
set-op-diff :: $'s \Rightarrow 's \Rightarrow 's$
set-op-filter :: $('x \Rightarrow \mathit{bool}) \Rightarrow 's \Rightarrow 's$
set-op-inter :: $'s \Rightarrow 's \Rightarrow 's$
set-op-subset :: $'s \Rightarrow 's \Rightarrow \mathit{bool}$
set-op-equal :: $'s \Rightarrow 's \Rightarrow \mathit{bool}$
set-op-disjoint :: $'s \Rightarrow 's \Rightarrow \mathit{bool}$
set-op-disjoint-witness :: $'s \Rightarrow 's \Rightarrow 'x\ \mathit{option}$

set-op-sel :: $'s \Rightarrow ('x \Rightarrow bool) \Rightarrow 'x\ option$ — Version without mapping
set-op-to-list :: $'s \Rightarrow 'x\ list$
set-op-from-list :: $'x\ list \Rightarrow 's$

locale *StdSetDefs* =
poly-set-iteratei-defs set-op-list-it ops
for *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 *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 *list-it* **where** *list-it* ≡ *set-op-list-it ops*
abbreviation *sng* **where** *sng* == *set-op-sng 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 *bex* **where** *bex* == *set-op-bex 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-memb α *invar memb* +
set-ins α *invar ins* +
set-ins-dj α *invar ins-dj* +
set-delete α *invar delete* +
poly-set-iteratei α *invar list-it* +
set-sng α *invar sng* +
set-isEmpty α *invar isEmpty* +
set-isSng α *invar isSng* +

set-ball α *invar ball* +
set-bex α *invar bex* +
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* :: ('*x*,'*s*,'*more*) *set-ops-scheme*
begin

lemmas *correct* =
empty-correct
sng-correct
memb-correct
ins-correct
ins-dj-correct
delete-correct
isEmpty-correct
isSng-correct
ball-correct
bex-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

lemmas *StdSet-intro* = *StdSet.intro*[*rem-dup-prems*]

locale *StdSet-no-invar* = *StdSet* + *set-no-invar* α *invar*

```
record ('x,'s) oset-ops = ('x::linorder,'s) set-ops +
  set-op-ordered-list-it :: 's ⇒ ('x,'x list) set-iterator
  set-op-rev-list-it :: 's ⇒ ('x,'x list) set-iterator
  set-op-min :: 's ⇒ ('x ⇒ bool) ⇒ 'x option
  set-op-max :: 's ⇒ ('x ⇒ bool) ⇒ 'x option
  set-op-to-sorted-list :: 's ⇒ 'x list
  set-op-to-rev-list :: 's ⇒ 'x list

locale StdOSetDefs = StdSetDefs ops
  + poly-set-iterateoi-defs set-op-ordered-list-it ops
  + poly-set-rev-iterateoi-defs set-op-rev-list-it ops
  for ops :: ('x::linorder,'s,'more) oset-ops-scheme
begin
  abbreviation ordered-list-it ≡ set-op-ordered-list-it ops
  abbreviation rev-list-it ≡ set-op-rev-list-it ops
  abbreviation min where min == set-op-min ops
  abbreviation max where max == set-op-max ops
  abbreviation to-sorted-list where
    to-sorted-list ≡ set-op-to-sorted-list ops
  abbreviation to-rev-list where to-rev-list ≡ set-op-to-rev-list ops
end

locale StdOSet =
  StdOSetDefs ops +
  StdSet ops +
  poly-set-iterateoi α invar ordered-list-it +
  poly-set-rev-iterateoi α invar rev-list-it +
  set-min α invar min +
  set-max α invar max +
  set-to-sorted-list α invar to-sorted-list +
  set-to-rev-list α invar to-rev-list
  for ops :: ('x::linorder,'s,'more) oset-ops-scheme
begin
end

lemmas StdOSet-intro =
  StdOSet.intro[OF StdSet-intro, rem-dup-prems]

no-notation insert (‹set'-ins›)


end
```

3.1.2 Specification of Sequences

```
theory ListSpec
imports ICF-Spec-Base
begin
```

Definition

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

locale *list-no-invar* = *list* +
  assumes *invar*[*simp*, *intro*!]: $\bigwedge l.$ *invar* *l*

Functions

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

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

locale *poly-list-iteratei* = *list* +
  constrains α :: $'s \Rightarrow 'x$ *list*
begin
  definition *iteratei* **where**
    iteratei-correct[*code-unfold*]: *iteratei* *s* $\equiv$ *foldli* (α *s*)
  definition *iterate* **where**
    iterate-correct[*code-unfold*]: *iterate* *s* $\equiv$ *foldli* (α *s*) (λ-. *True*)
end

locale *poly-list-rev-iteratei* = *list* +
  constrains α :: $'s \Rightarrow 'x$ *list*
begin
  definition *rev-iteratei* **where**
    rev-iteratei-correct[*code-unfold*]: *rev-iteratei* *s* $\equiv$ *foldri* (α *s*)
  definition *rev-iterate* **where**
    rev-iterate-correct[*code-unfold*]: *rev-iterate* *s* $\equiv$ *foldri* (α *s*) (λ-. *True*)
end

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

```
  assumes size-correct:
    invar s ⟹ size s = length (α s)

locale list-appendl = list +
  constrains α :: 's ⇒ 'x list
  fixes appendl :: 'x ⇒ 's ⇒ 's
  assumes appendl-correct:
    invar s ⟹ α (appendl x s) = x#α s
    invar s ⟹ invar (appendl x s)
begin
  abbreviation (input) push ≡ appendl
  lemmas push-correct = appendl-correct
end

locale list-removel = list +
  constrains α :: 's ⇒ 'x list
  fixes removel :: 's ⇒ ('x × 's)
  assumes removel-correct:
    ⟦invar s; α s ≠ []⟧ ⟹ fst (removel s) = hd (α s)
    ⟦invar s; α s ≠ []⟧ ⟹ α (snd (removel s)) = tl (α s)
    ⟦invar s; α s ≠ []⟧ ⟹ invar (snd (removel s))
begin
  lemma removelE:
    assumes I: invar s    α s ≠ []
    obtains s' where removel s = (hd (α s), s')    invar s'    α s' = tl (α s)
  ⟨proof⟩
```

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

```
  abbreviation (input) pop ≡ removel
  lemmas pop-correct = removel-correct

  abbreviation (input) dequeue ≡ removel
  lemmas dequeue-correct = removel-correct
end

locale list-leftmost = list +
  constrains α :: 's ⇒ 'x list
  fixes leftmost :: 's ⇒ 'x
  assumes leftmost-correct:
    ⟦invar s; α s ≠ []⟧ ⟹ leftmost s = hd (α s)
begin
  abbreviation (input) top where top ≡ leftmost
  lemmas top-correct = leftmost-correct
end

locale list-appendr = list +
  constrains α :: 's ⇒ 'x list
  fixes appendr :: 'x ⇒ 's ⇒ 's
```

assumes *appendr-correct*:
$invar\ s \Longrightarrow \alpha\ (appendr\ x\ s) = \alpha\ s\ @\ [x]$
$invar\ s \Longrightarrow invar\ (appendr\ x\ s)$
begin
abbreviation (*input*) *enqueue* $\equiv$ *appendr*
lemmas *enqueue-correct* = *appendr-correct*
end

locale *list-remover* = *list* +
constrains $\alpha :: 's \Rightarrow 'x\ list$
fixes *remover* :: $'s \Rightarrow 's \times 'x$
assumes *remover-correct*:
$[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow \alpha\ (fst\ (remover\ s)) = butlast\ (\alpha\ s)$
$[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow snd\ (remover\ s) = last\ (\alpha\ s)$
$[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow invar\ (fst\ (remover\ s))$

locale *list-rightmost* = *list* +
constrains $\alpha :: 's \Rightarrow 'x\ list$
fixes *rightmost* :: $'s \Rightarrow 'x$
assumes *rightmost-correct*:
$[\![invar\ s;\ \alpha\ s \neq []]\!] \Longrightarrow rightmost\ s = List.last\ (\alpha\ s)$
begin
abbreviation (*input*) *bot* **where** *bot* $\equiv$ *rightmost*
lemmas *bot-correct* = *rightmost-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*:
$[\![invar\ s;\ i<length\ (\alpha\ s)]\!] \Longrightarrow \alpha\ (set\ s\ i\ x) = (\alpha\ s)\ [i := x]$
$[\![invar\ s;\ i<length\ (\alpha\ s)]\!] \Longrightarrow invar\ (set\ s\ i\ x)$

record $('a,'s)$ *list-ops* =
list-op-α :: $'s \Rightarrow 'a\ list$
list-op-invar :: $'s \Rightarrow bool$
list-op-empty :: $unit \Rightarrow 's$
list-op-isEmpty :: $'s \Rightarrow bool$
list-op-size :: $'s \Rightarrow nat$
list-op-appendl :: $'a \Rightarrow 's \Rightarrow 's$
list-op-removel :: $'s \Rightarrow 'a \times 's$
list-op-leftmost :: $'s \Rightarrow 'a$
list-op-appendr :: $'a \Rightarrow 's \Rightarrow 's$

```
  list-op-remover :: 's ⇒ 's × 'a
  list-op-rightmost :: 's ⇒ 'a
  list-op-get :: 's ⇒ nat ⇒ 'a
  list-op-set :: 's ⇒ nat ⇒ 'a ⇒ 's

locale StdListDefs =
  poly-list-iteratei list-op-α ops    list-op-invar ops
  + poly-list-rev-iteratei list-op-α ops    list-op-invar ops
  for ops :: ('a,'s,'more) list-ops-scheme
begin
  abbreviation α where α ≡ list-op-α ops
  abbreviation invar where invar ≡ list-op-invar ops
  abbreviation empty where empty ≡ list-op-empty ops
  abbreviation isEmpty where isEmpty ≡ list-op-isEmpty ops
  abbreviation size where size ≡ list-op-size ops
  abbreviation appendl where appendl ≡ list-op-appendl ops
  abbreviation removel where removel ≡ list-op-removel ops
  abbreviation leftmost where leftmost ≡ list-op-leftmost ops
  abbreviation appendr where appendr ≡ list-op-appendr ops
  abbreviation remover where remover ≡ list-op-remover ops
  abbreviation rightmost where rightmost ≡ list-op-rightmost ops
  abbreviation get where get ≡ list-op-get ops
  abbreviation set where set ≡ list-op-set ops
end

locale StdList = StdListDefs ops
  + list α invar
  + list-empty α invar empty
  + list-isEmpty α invar isEmpty
  + list-size α invar size
  + list-appendl α invar appendl
  + list-removel α invar removel
  + list-leftmost α invar leftmost
  + list-appendr α invar appendr
  + list-remover α invar remover
  + list-rightmost α invar rightmost
  + list-get α invar get
  + list-set α invar set
  for ops :: ('a,'s,'more) list-ops-scheme
begin
  lemmas correct =
    empty-correct
    isEmpty-correct
    size-correct
    appendl-correct
    removel-correct
    leftmost-correct
    appendr-correct
    remover-correct
```

rightmost-correct
get-correct
set-correct

end

locale *StdList-no-invar* = *StdList* + *list-no-invar* α *invar*

end

3.1.3 Specification of Annotated Lists

theory *AnnotatedListSpec*
imports *ICF-Spec-Base*
begin

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 α :: $'s \Rightarrow ('e \times 'a::monoid\text{-}add)$ *list*
fixes *invar* :: $'s \Rightarrow$ *bool*

locale *al-no-invar* = *al* +
assumes *invar*[*simp*, *intro*!]: $\bigwedge l.$ *invar l*

Basic Annotated List Operations

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

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

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

locale *poly-al-fold* = *al* +
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add)\ list$
begin
definition *foldl* **where**
foldl-correct[*code-unfold*]: $foldl\ f\ \sigma\ s = List.foldl\ f\ \sigma\ (\alpha\ s)$
definition *foldr* **where**
foldr-correct[*code-unfold*]: $foldr\ f\ s\ \sigma = List.foldr\ f\ (\alpha\ s)\ \sigma$
end

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*:
$invar\ s \Longrightarrow (annot\ s) = (sum\text{-}list\ (map\ snd\ (\alpha\ s)))$

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*:
$[\![invar\ s;$
$\forall\ a\ b.\ p\ a \longrightarrow p\ (a + b);$
$\neg\ p\ i;$
$p\ (i + sum\text{-}list\ (map\ snd\ (\alpha\ s)));$
$(splits\ p\ i\ s) = (l, (e,a), r)]\!]$
$\Longrightarrow$
$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)\ \wedge$
$\neg\ p\ (i + sum\text{-}list\ (map\ snd\ (\alpha\ l)))\ \wedge$
$p\ (i + sum\text{-}list\ (map\ snd\ (\alpha\ l)) + a)\ \wedge$
$invar\ l\ \wedge$
$invar\ r$

begin
lemma *splitsE*:
assumes

invar: *invar s* **and**
mono: $\forall\, a\ b.\ p\ a \longrightarrow p\ (a + b)$ **and**
init-ff: $\neg\ p\ i$ **and**
sum-tt: $p\ (i + \textit{sum-list}\ (\textit{map snd}\ (\alpha\ s)))$
obtains *l e a r* **where**
$(\textit{splits}\ p\ i\ s) = (l,\ (e,a),\ r)$
$(\alpha\ s) = (\alpha\ l)\ @\ (e,a)\ \#\ (\alpha\ r)$
$\neg\ p\ (i + \textit{sum-list}\ (\textit{map snd}\ (\alpha\ l)))$
$p\ (i + \textit{sum-list}\ (\textit{map snd}\ (\alpha\ l)) + a)$
invar l
invar r
⟨*proof*⟩
end

Record Based Interface

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

locale *StdALDefs* = *poly-al-fold alist-op-α ops* *alist-op-invar ops*
for *ops* $::$ $('e,'a::\textit{monoid-add},'s,'\textit{more})$ *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
    foldl-correct
    foldr-correct
end

locale *StdAL-no-invar* = *StdAL* + *al-no-invar* α *invar*

end

3.1.4 Specification of Priority Queues

theory *PrioSpec*
imports *ICF-Spec-Base 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

locale *prio-no-invar* = *prio* +
assumes *invar*[*simp*, *intro*!]: $\bigwedge s.\ invar\ s$

Basic Priority Queue Functions

Empty Queue **locale** *prio-empty* = *prio* +
constrains $\alpha :: 'p \Rightarrow ('e \times 'a::linorder)\ multiset$
fixes *empty* :: $unit \Rightarrow 'p$
assumes *empty-correct*:
$invar\ (empty\ ())$
$\alpha\ (empty\ ()) = \{\#\}$

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

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

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

Record based interface

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

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

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

locale StdPrio-no-invar = StdPrio + prio-no-invar α invar

end
```

3.1.5 Specification of Unique Priority Queues

theory *PrioUniqueSpec*
imports *ICF-Spec-Base*
begin

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

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

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

locale *uprio-no-invar* = *uprio* +
assumes *invar*[*simp*, *intro*!]: $\bigwedge s.$ *invar s*

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

Basic Upriority Queue Functions

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

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*:
$invar\ s \Longrightarrow prio\ s\ e = (\alpha\ s)\ e$

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

locale *StdUprio-no-invar* = *StdUprio* + *uprio-no-invar* α *invar*

end

3.2 Generic Algorithms

3.2.1 General Algorithms for Iterators over Finite Sets

theory *SetIteratorCollectionsGA*
imports
  ../spec/SetSpec
  ../spec/MapSpec
begin

Iterate add to Set

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

it (λ-. *True*) (λ*x* σ. *ins x* σ) *s*

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

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

Iterator to Set

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

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

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

Iterate image/filter add to Set

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

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

⟨*proof*⟩

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

For completeness lets also consider injective versions.

lemma *iterate-inj-image-filter-add-to-set-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: $\{y.\ \exists\, x.\ x \in S0 \wedge f\ x = Some\ y\} \cap \alpha\ s = \{\}$
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*)) =
$\{b\ .\ \exists\, a.\ a \in S0 \wedge f\ a = Some\ b\} \cup \alpha\ s \wedge$
invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
⟨*proof*⟩

lemma *iterate-inj-image-filter-to-set-correct* :
assumes *ins-OK*: *set-ins-dj* α *invar ins*
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *it*: *set-iterator it S0*
assumes *f-inj-on*: *inj-on f* (*S0* $\cap$ *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*))
⟨*proof*⟩

Iterate diff Set

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

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

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*)
⟨*proof*⟩

lemma *iterate-add-to-map-dj-correct* :
assumes *upd-OK*: *map-update-dj* α *invar upd*
assumes *m-OK*: *invar m*
assumes *it*: *map-iterator it M*
assumes *dj*: *dom M* ∩ *dom* (α *m*) = {}
shows α (*iterate-add-to-map m upd it*) = α *m* ++ *M* ∧ *invar* (*iterate-add-to-map m upd it*)
⟨*proof*⟩

Iterator to Map

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

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

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

end

3.2.2 Generic Algorithms for Maps

theory *MapGA*
imports *SetIteratorCollectionsGA*
begin record ('*k*,'*v*,'*s*) *map-basic-ops* =

```
  bmap-op-α :: ('k,'v,'s) map-α
  bmap-op-invar :: ('k,'v,'s) map-invar
  bmap-op-empty :: ('k,'v,'s) map-empty
  bmap-op-lookup :: ('k,'v,'s) map-lookup
  bmap-op-update :: ('k,'v,'s) map-update
  bmap-op-update-dj :: ('k,'v,'s) map-update-dj
  bmap-op-delete :: ('k,'v,'s) map-delete
  bmap-op-list-it :: ('k,'v,'s) map-list-it

record ('k,'v,'s) omap-basic-ops = ('k,'v,'s) map-basic-ops +
  bmap-op-ordered-list-it :: 's ⇒ ('k,'v,('k×'v) list) map-iterator
  bmap-op-rev-list-it :: 's ⇒ ('k,'v,('k×'v) list) map-iterator

locale StdBasicMapDefs =
  poly-map-iteratei-defs bmap-op-list-it ops
  for ops :: ('k,'v,'s,'more) map-basic-ops-scheme
begin
  abbreviation α where α == bmap-op-α ops
  abbreviation invar where invar == bmap-op-invar ops
  abbreviation empty where empty == bmap-op-empty ops
  abbreviation lookup where lookup == bmap-op-lookup ops
  abbreviation update where update == bmap-op-update ops
  abbreviation update-dj where update-dj == bmap-op-update-dj ops
  abbreviation delete where delete == bmap-op-delete ops
  abbreviation list-it where list-it == bmap-op-list-it ops
end

locale StdBasicOMapDefs = StdBasicMapDefs ops
  + poly-map-iterateoi-defs bmap-op-ordered-list-it ops
  + poly-map-rev-iterateoi-defs bmap-op-rev-list-it ops
  for ops :: ('k::linorder,'v,'s,'more) omap-basic-ops-scheme
begin
  abbreviation ordered-list-it where ordered-list-it
    ≡ bmap-op-ordered-list-it ops
  abbreviation rev-list-it where rev-list-it
    ≡ bmap-op-rev-list-it ops
end

locale StdBasicMap = StdBasicMapDefs ops +
  map α invar +
  map-empty α invar empty +
  map-lookup α invar lookup  +
  map-update α invar update  +
  map-update-dj α invar update-dj +
  map-delete α invar delete  +
  poly-map-iteratei α invar list-it
  for ops :: ('k,'v,'s,'more) map-basic-ops-scheme
begin
  lemmas correct[simp] = empty-correct lookup-correct update-correct
```

```
    update-dj-correct delete-correct
end


locale StdBasicOMap =
  StdBasicOMapDefs ops +
  StdBasicMap ops +
  poly-map-iterateoi α invar ordered-list-it +
  poly-map-rev-iterateoi α invar rev-list-it
  for ops :: ('k::linorder,'v,'s,'more) omap-basic-ops-scheme
begin
end

context StdBasicMapDefs begin
  definition g-sng k v ≡ update k v (empty ())
  definition g-add m1 m2 ≡ iterate m2 (λ(k,v) σ. update k v σ) m1

  definition
    g-sel m P ≡
      iteratei m (λσ. σ = None) (λx σ. if P x then Some x else None) None

  definition g-bex m P ≡ iteratei m (λx. ¬x) (λkv σ. P kv) False
  definition g-ball m P ≡ iteratei m id (λkv σ. P kv) True

  definition g-size m ≡ iterate m (λ-. Suc) (0::nat)
  definition g-size-abort b m ≡ iteratei m (λs. s<b) (λ-. Suc) (0::nat)

  definition g-isEmpty m ≡ g-size-abort 1 m = 0
  definition g-isSng m ≡ g-size-abort 2 m = 1

  definition g-to-list m ≡ iterate m (#) []

  definition g-list-to-map l ≡ foldl (λm (k,v). update k v m) (empty ())
    (rev l)

  definition g-add-dj m1 m2 ≡ iterate m2 (λ(k,v) σ. update-dj k v σ) m1

  definition g-restrict P m ≡ iterate m
    (λ(k,v) σ. if P (k,v) then update-dj k v σ else σ) (empty ())

  definition dflt-ops :: ('k,'v,'s) map-ops
    where [icf-rec-def]:
    dflt-ops ≡
      (|
        map-op-α = α,
        map-op-invar = invar,
        map-op-empty = empty,
        map-op-lookup = lookup,
        map-op-update = update,
```

map-op-update-dj = *update-dj*,
map-op-delete = *delete*,
map-op-list-it = *list-it*,
map-op-sng = *g-sng*,
map-op-restrict = *g-restrict*,
map-op-add = *g-add*,
map-op-add-dj = *g-add-dj*,
map-op-isEmpty = *g-isEmpty*,
map-op-isSng = *g-isSng*,
map-op-ball = *g-ball*,
map-op-bex = *g-bex*,
map-op-size = *g-size*,
map-op-size-abort = *g-size-abort*,
map-op-sel = *g-sel*,
map-op-to-list = *g-to-list*,
map-op-to-map = *g-list-to-map*
⦈

⟨*ML*⟩

end

lemma *update-dj-by-update*:
assumes *map-update* α *invar update*
shows *map-update-dj* α *invar update*
⟨*proof*⟩

lemma *map-iterator-linord-is-it*:
map-iterator-linord m it ⟹ *map-iterator m it*
⟨*proof*⟩

lemma *map-rev-iterator-linord-is-it*:
map-iterator-rev-linord m it ⟹ *map-iterator m it*
⟨*proof*⟩

context *StdBasicMap*
begin
lemma *g-sng-impl*: *map-sng* α *invar g-sng*
⟨*proof*⟩

lemma *g-add-impl*: *map-add* α *invar g-add*
⟨*proof*⟩

lemma *g-sel-impl*: *map-sel′* α *invar g-sel*
⟨*proof*⟩

lemma *g-bex-impl*: *map-bex* α *invar g-bex*
⟨*proof*⟩

lemma *g-ball-impl*: *map-ball* α *invar g-ball*
⟨*proof*⟩

lemma *g-size-impl*: *map-size* α *invar g-size*
⟨*proof*⟩

lemma *g-size-abort-impl*: *map-size-abort* α *invar g-size-abort*
⟨*proof*⟩

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

lemma *g-isSng-impl*: *map-isSng* α *invar g-isSng*
⟨*proof*⟩

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

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

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

lemma *g-restrict-impl*: *map-restrict* α *invar* α *invar g-restrict*
⟨*proof*⟩

lemma *dflt-ops-impl*: *StdMap dflt-ops*
⟨*proof*⟩
end

context *StdBasicOMapDefs*
begin
definition
g-min m P ≡
iterateoi m (λσ. σ = *None*) (λ*x* σ. *if P x then Some x else None*) *None*

definition
g-max m P ≡
rev-iterateoi m (λσ. σ = *None*) (λ*x* σ. *if P x then Some x else None*) *None*

definition *g-to-sorted-list m* ≡ *rev-iterateo m* (#) []
definition *g-to-rev-list m* ≡ *iterateo m* (#) []

definition *dflt-oops* :: ('*k*,'*v*,'*s*) *omap-ops*
where [*icf-rec-def*]:
dflt-oops ≡ *map-ops.extend dflt-ops*
(⦇

```
      map-op-ordered-list-it = ordered-list-it,
      map-op-rev-list-it = rev-list-it,
      map-op-min = g-min,
      map-op-max = g-max,
      map-op-to-sorted-list = g-to-sorted-list,
      map-op-to-rev-list = g-to-rev-list
    ⦈
  ⟨ML⟩

end

context StdBasicOMap
begin
  lemma g-min-impl: map-min α invar g-min
  ⟨proof⟩

  lemma g-max-impl: map-max α invar g-max
  ⟨proof⟩

  lemma g-to-sorted-list-impl: map-to-sorted-list α invar g-to-sorted-list
  ⟨proof⟩

  lemma g-to-rev-list-impl: map-to-rev-list α invar g-to-rev-list
  ⟨proof⟩

  lemma dflt-oops-impl: StdOMap dflt-oops
  ⟨proof⟩

end

locale g-image-filter-defs-loc =
  m1: StdMapDefs ops1 +
  m2: StdMapDefs ops2
  for ops1 :: ('k1,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k2,'v2,'s2,'m2) map-ops-scheme
begin
  definition g-image-filter f m1 ≡ m1.iterate m1 (λkv σ. case f kv of
      None => σ
    | Some (k',v') => m2.update-dj k' v' σ
    ) (m2.empty ())
end

locale g-image-filter-loc = g-image-filter-defs-loc ops1 ops2 +
  m1: StdMap ops1 +
  m2: StdMap ops2
  for ops1 :: ('k1,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k2,'v2,'s2,'m2) map-ops-scheme
begin
  lemma g-image-filter-impl:
```

```
    map-image-filter m1.α m1.invar m2.α m2.invar g-image-filter
  ⟨proof⟩
end

sublocale g-image-filter-loc
  < map-image-filter m1.α m1.invar m2.α m2.invar g-image-filter
  ⟨proof⟩


locale g-value-image-filter-defs-loc =
  m1: StdMapDefs ops1 +
  m2: StdMapDefs ops2
  for ops1 :: ('k,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k,'v2,'s2,'m2) map-ops-scheme
begin
  definition g-value-image-filter f m1 ≡ m1.iterate m1 (λ(k,v) σ.
    case f k v of
      None => σ
    | Some v' => m2.update-dj k v' σ
    ) (m2.empty ())

end


lemma restrict-map-dom-subset: ⟦ dom m ⊆ R⟧ ⟹ m|`R = m
  ⟨proof⟩


locale g-value-image-filter-loc = g-value-image-filter-defs-loc ops1 ops2 +
  m1: StdMap ops1 +
  m2: StdMap ops2
  for ops1 :: ('k,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k,'v2,'s2,'m2) map-ops-scheme
begin
  lemma g-value-image-filter-impl:
    map-value-image-filter m1.α m1.invar m2.α m2.invar g-value-image-filter
    ⟨proof⟩
end

sublocale g-value-image-filter-loc
  < map-value-image-filter m1.α m1.invar m2.α m2.invar g-value-image-filter
  ⟨proof⟩


end
```

3.2.3 Generic Algorithms for Sets

theory *SetGA*

imports *../spec/SetSpec SetIteratorCollectionsGA*
begin

Generic Set Algorithms

locale *g-set-xx-defs-loc* =
  s1: *StdSetDefs ops1* + *s2*: *StdSetDefs ops2*
  for *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
  and *ops2* :: (*'x*,*'s2*,*'more2*) *set-ops-scheme*
begin
  definition *g-copy s* ≡ *s1.iterate s s2.ins-dj* (*s2.empty* ())
  definition *g-filter P s1* ≡ *s1.iterate s1*
    (λ*x* σ. *if P x then s2.ins-dj x* σ *else* σ)
    (*s2.empty* ())

  definition *g-union s1 s2* ≡ *s1.iterate s1 s2.ins s2*
  definition *g-diff s1 s2* ≡ *s2.iterate s2 s1.delete s1*

  definition *g-union-list* **where**
    g-union-list l =
      foldl (λ*s s'*. *g-union s' s*) (*s2.empty* ()) *l*

  definition *g-union-dj s1 s2* ≡ *s1.iterate s1 s2.ins-dj s2*

  definition *g-disjoint-witness s1 s2* ≡
    s1.sel s1 (λ*x*. *s2.memb x s2*)

  definition *g-disjoint s1 s2* ≡
    s1.ball s1 (λ*x*. ¬*s2.memb x s2*)
end

locale *g-set-xx-loc* = *g-set-xx-defs-loc ops1 ops2* +
  s1: *StdSet ops1* + *s2*: *StdSet ops2*
  for *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
  and *ops2* :: (*'x*,*'s2*,*'more2*) *set-ops-scheme*
begin
  lemma *g-copy-alt*:
    g-copy s = *iterate-to-set s2.empty s2.ins-dj* (*s1.iteratei s*)
    ⟨*proof*⟩

  lemma *g-copy-impl*: *set-copy s1.*α *s1.invar s2.*α *s2.invar g-copy*
  ⟨*proof*⟩

  lemma *g-filter-impl*: *set-filter s1.*α *s1.invar s2.*α *s2.invar g-filter*
  ⟨*proof*⟩

  lemma *g-union-alt*:
    g-union s1 s2 = *iterate-add-to-set s2 s2.ins* (*s1.iteratei s1*)
    ⟨*proof*⟩

lemma *g-diff-alt*:
g-diff s1 s2 = iterate-diff-set s1 s1.delete (s2.iteratei s2)
⟨*proof*⟩

lemma *g-union-impl*:
set-union s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union
⟨*proof*⟩

lemma *g-diff-impl*:
set-diff s1.α s1.invar s2.α s2.invar g-diff
⟨*proof*⟩

lemma *g-union-list-impl*:
shows *set-union-list s1.α s1.invar s2.α s2.invar g-union-list*
⟨*proof*⟩

lemma *g-union-dj-impl*:
set-union-dj s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union-dj
⟨*proof*⟩

lemma *g-disjoint-witness-impl*:
set-disjoint-witness s1.α s1.invar s2.α s2.invar g-disjoint-witness
⟨*proof*⟩

lemma *g-disjoint-impl*:
set-disjoint s1.α s1.invar s2.α s2.invar g-disjoint
⟨*proof*⟩
end

sublocale *g-set-xx-loc* <
set-copy s1.α s1.invar s2.α s2.invar g-copy ⟨*proof*⟩

sublocale *g-set-xx-loc* <
set-filter s1.α s1.invar s2.α s2.invar g-filter ⟨*proof*⟩

sublocale *g-set-xx-loc* <
set-union s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union
⟨*proof*⟩

sublocale *g-set-xx-loc* <
set-union-dj s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union-dj
⟨*proof*⟩

sublocale *g-set-xx-loc* <
set-diff s1.α s1.invar s2.α s2.invar g-diff
⟨*proof*⟩

sublocale *g-set-xx-loc* <

set-disjoint-witness s1.α s1.invar s2.α s2.invar g-disjoint-witness
⟨*proof*⟩

sublocale *g-set-xx-loc* <
set-disjoint s1.α s1.invar s2.α s2.invar g-disjoint ⟨*proof*⟩

locale *g-set-xxx-defs-loc* =
s1: *StdSetDefs ops1* +
s2: *StdSetDefs ops2* +
s3: *StdSetDefs ops3*
for *ops1* :: ('*x*,'*s1*,'*more1*) *set-ops-scheme*
and *ops2* :: ('*x*,'*s2*,'*more2*) *set-ops-scheme*
and *ops3* :: ('*x*,'*s3*,'*more3*) *set-ops-scheme*
begin
definition *g-inter s1 s2* ≡
s1.iterate s1 (λ*x s*. *if s2.memb x s2 then s3.ins-dj x s else s*)
(*s3.empty* ())
end

locale *g-set-xxx-loc* = *g-set-xxx-defs-loc ops1 ops2 ops3* +
s1: *StdSet ops1* +
s2: *StdSet ops2* +
s3: *StdSet ops3*
for *ops1* :: ('*x*,'*s1*,'*more1*) *set-ops-scheme*
and *ops2* :: ('*x*,'*s2*,'*more2*) *set-ops-scheme*
and *ops3* :: ('*x*,'*s3*,'*more3*) *set-ops-scheme*
begin
lemma *g-inter-impl*: *set-inter s1.α s1.invar s2.α s2.invar s3.α s3.invar*
g-inter
⟨*proof*⟩
end

sublocale *g-set-xxx-loc*
< *set-inter s1.α s1.invar s2.α s2.invar s3.α s3.invar g-inter*
⟨*proof*⟩

locale *g-set-xy-defs-loc* =
s1: *StdSet ops1* + *s2*: *StdSet ops2*
for *ops1* :: ('*x1*,'*s1*,'*more1*) *set-ops-scheme*
and *ops2* :: ('*x2*,'*s2*,'*more2*) *set-ops-scheme*
begin

definition *g-image-filter f s* ≡
s1.iterate s
(λ*x res*. *case f x of Some v* ⇒ *s2.ins v res* | - ⇒ *res*)
(*s2.empty* ())

definition *g-image f s* ≡
s1.iterate s (λ*x res*. *s2.ins* (*f x*) *res*) (*s2.empty* ())

definition *g-inj-image-filter f s* ≡
s1.iterate s
(λ*x res*. *case f x of Some v* ⇒ *s2.ins-dj v res* | - ⇒ *res*)
(*s2.empty* ())

definition *g-inj-image f s* ≡
s1.iterate s (λ*x res*. *s2.ins-dj* (*f x*) *res*) (*s2.empty* ())

end

locale *g-set-xy-loc* = *g-set-xy-defs-loc ops1 ops2* +
s1: *StdSet ops1* + *s2*: *StdSet ops2*
for *ops1* :: (*'x1*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x2*,*'s2*,*'more2*) *set-ops-scheme*
begin
lemma *g-image-filter-impl*:
set-image-filter s1.α s1.invar s2.α s2.invar g-image-filter
⟨*proof*⟩

lemma *g-image-alt*: *g-image f s* = *g-image-filter* (*Some o f*) *s*
⟨*proof*⟩

lemma *g-image-impl*: *set-image s1.α s1.invar s2.α s2.invar g-image*
⟨*proof*⟩

lemma *g-inj-image-filter-impl*:
set-inj-image-filter s1.α s1.invar s2.α s2.invar g-inj-image-filter
⟨*proof*⟩

lemma *g-inj-image-alt*: *g-inj-image f s* = *g-inj-image-filter* (*Some o f*) *s*
⟨*proof*⟩

lemma *g-inj-image-impl*:
set-inj-image s1.α s1.invar s2.α s2.invar g-inj-image
⟨*proof*⟩

end

sublocale *g-set-xy-loc* < *set-image-filter s1.α s1.invar s2.α s2.invar*
g-image-filter ⟨*proof*⟩

sublocale *g-set-xy-loc* < *set-image s1.α s1.invar s2.α s2.invar*
g-image ⟨*proof*⟩

sublocale *g-set-xy-loc* < *set-inj-image s1.α s1.invar s2.α s2.invar*
g-inj-image ⟨*proof*⟩

locale *g-set-xyy-defs-loc* =
s0: *StdSetDefs ops0* +
g-set-xx-defs-loc ops1 ops2
for *ops0* :: (*'x0*,*'s0*,*'more0*) *set-ops-scheme*
and *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x*,*'s2*,*'more2*) *set-ops-scheme*
begin
definition *g-Union-image*
:: (*'x0* ⇒ *'s1*) ⇒ *'s0* ⇒ *'s2*
where *g-Union-image f S*
== *s0.iterate S* (λ*x res*. *g-union* (*f x*) *res*) (*s2.empty* ())
end

locale *g-set-xyy-loc* = *g-set-xyy-defs-loc ops0 ops1 ops2* +
s0: *StdSet ops0* +
g-set-xx-loc ops1 ops2
for *ops0* :: (*'x0*,*'s0*,*'more0*) *set-ops-scheme*
and *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x*,*'s2*,*'more2*) *set-ops-scheme*
begin

lemma *g-Union-image-impl*:
set-Union-image s0.α s0.invar s1.α s1.invar s2.α s2.invar g-Union-image
⟨*proof*⟩
end

sublocale *g-set-xyy-loc* <
set-Union-image s0.α s0.invar s1.α s1.invar s2.α s2.invar g-Union-image
⟨*proof*⟩

Default Set Operations

record (*'x*,*'s*) *set-basic-ops* =
bset-op-α :: *'s* ⇒ *'x set*
bset-op-invar :: *'s* ⇒ *bool*
bset-op-empty :: *unit* ⇒ *'s*
bset-op-memb :: *'x* ⇒ *'s* ⇒ *bool*
bset-op-ins :: *'x* ⇒ *'s* ⇒ *'s*
bset-op-ins-dj :: *'x* ⇒ *'s* ⇒ *'s*
bset-op-delete :: *'x* ⇒ *'s* ⇒ *'s*
bset-op-list-it :: (*'x*,*'s*) *set-list-it*

```
record ('x,'s) oset-basic-ops = ('x::linorder,'s) set-basic-ops +
  bset-op-ordered-list-it :: 's ⇒ ('x,'x list) set-iterator
  bset-op-rev-list-it :: 's ⇒ ('x,'x list) set-iterator

locale StdBasicSetDefs =
  poly-set-iteratei-defs bset-op-list-it ops
  for ops :: ('x,'s,'more) set-basic-ops-scheme
begin
  abbreviation α where α == bset-op-α ops
  abbreviation invar where invar == bset-op-invar ops
  abbreviation empty where empty == bset-op-empty ops
  abbreviation memb where memb == bset-op-memb ops
  abbreviation ins where ins == bset-op-ins ops
  abbreviation ins-dj where ins-dj == bset-op-ins-dj ops
  abbreviation delete where delete == bset-op-delete ops
  abbreviation list-it where list-it ≡ bset-op-list-it ops
end

locale StdBasicOSetDefs = StdBasicSetDefs ops
  + poly-set-iterateoi-defs bset-op-ordered-list-it ops
  + poly-set-rev-iterateoi-defs bset-op-rev-list-it ops
  for ops :: ('x::linorder,'s,'more) oset-basic-ops-scheme
begin
  abbreviation ordered-list-it ≡ bset-op-ordered-list-it ops
  abbreviation rev-list-it ≡ bset-op-rev-list-it ops
end

locale StdBasicSet = StdBasicSetDefs ops +
  set α invar +
  set-empty α invar empty +
  set-memb α invar memb +
  set-ins α invar ins +
  set-ins-dj α invar ins-dj +
  set-delete α invar delete +
  poly-set-iteratei α invar list-it
  for ops :: ('x,'s,'more) set-basic-ops-scheme
begin

  lemmas correct[simp] =
    empty-correct
    memb-correct
    ins-correct
    ins-dj-correct
    delete-correct

end

locale StdBasicOSet =
  StdBasicOSetDefs ops +
```

```
  StdBasicSet ops +
  poly-set-iterateoi α invar ordered-list-it +
  poly-set-rev-iterateoi α invar rev-list-it
  for ops :: ('x::linorder,'s,'more) oset-basic-ops-scheme
begin
end

context StdBasicSetDefs
begin
  definition g-sng x ≡ ins x (empty ())
  definition g-isEmpty s ≡ iteratei s (λc. c) (λ- -. False) True
  definition g-sel' s P ≡ iteratei s ((=) None)
    (λx -. if P x then Some x else None) None

  definition g-ball s P ≡ iteratei s (λc. c) (λx σ. P x) True
  definition g-bex s P ≡ iteratei s (λc. ¬c) (λx σ. P x) False
  definition g-size s ≡ iteratei s (λ-. True) (λx n . Suc n) 0
  definition g-size-abort m s ≡ iteratei s (λσ. σ < m) (λx. Suc) 0
  definition g-isSng s ≡ (iteratei s (λσ. σ < 2) (λx. Suc) 0 = 1)

  definition g-union s1 s2 ≡ iterate s1 ins s2
  definition g-diff s1 s2 ≡ iterate s2 delete s1

  definition g-subset s1 s2 ≡ g-ball s1 (λx. memb x s2)

  definition g-equal s1 s2 ≡ g-subset s1 s2 ∧ g-subset s2 s1

  definition g-to-list s ≡ iterate s (#) []

  fun g-from-list-aux where
    g-from-list-aux accs [] = accs |
    g-from-list-aux accs (x#l) = g-from-list-aux (ins x accs) l
    — Tail recursive version

  definition g-from-list l == g-from-list-aux (empty ()) l

  definition g-inter s1 s2 ≡
    iterate s1 (λx s. if memb x s2 then ins-dj x s else s)
      (empty ())

  definition g-union-dj s1 s2 ≡ iterate s1 ins-dj s2
  definition g-filter P s ≡ iterate s
    (λx σ. if P x then ins-dj x σ else σ)
    (empty ())

  definition g-disjoint-witness s1 s2 ≡ g-sel' s1 (λx. memb x s2)
  definition g-disjoint s1 s2 ≡ g-ball s1 (λx. ¬memb x s2)

  definition dflt-ops
```

where [*icf-rec-def*]: *dflt-ops* ≡ (|
set-op-α = *α*,
set-op-invar = *invar*,
set-op-empty = *empty*,
set-op-memb = *memb*,
set-op-ins = *ins*,
set-op-ins-dj = *ins-dj*,
set-op-delete = *delete*,
set-op-list-it = *list-it*,
set-op-sng = *g-sng* ,
set-op-isEmpty = *g-isEmpty* ,
set-op-isSng = *g-isSng* ,
set-op-ball = *g-ball* ,
set-op-bex = *g-bex* ,
set-op-size = *g-size* ,
set-op-size-abort = *g-size-abort* ,
set-op-union = *g-union* ,
set-op-union-dj = *g-union-dj* ,
set-op-diff = *g-diff* ,
set-op-filter = *g-filter* ,
set-op-inter = *g-inter* ,
set-op-subset = *g-subset* ,
set-op-equal = *g-equal* ,
set-op-disjoint = *g-disjoint* ,
set-op-disjoint-witness = *g-disjoint-witness* ,
set-op-sel = *g-sel′* ,
set-op-to-list = *g-to-list* ,
set-op-from-list = *g-from-list*
|)

⟨*ML*⟩
end

context *StdBasicSet*
begin
lemma *g-sng-impl*: *set-sng α invar g-sng*
⟨*proof*⟩

lemma *g-ins-dj-impl*: *set-ins-dj α invar ins*
⟨*proof*⟩

lemma *g-isEmpty-impl*: *set-isEmpty α invar g-isEmpty*
⟨*proof*⟩

lemma *g-sel′-impl*: *set-sel′ α invar g-sel′*
⟨*proof*⟩

lemma *g-ball-alt*: *g-ball s P* = *iterate-ball* (*iteratei s*) *P*
⟨*proof*⟩

lemma *g-bex-alt*: *g-bex s P* = *iterate-bex* (*iteratei s*) *P*
⟨*proof*⟩

lemma *g-ball-impl*: *set-ball α invar g-ball*
⟨*proof*⟩

lemma *g-bex-impl*: *set-bex α invar g-bex*
⟨*proof*⟩

lemma *g-size-alt*: *g-size s* = *iterate-size* (*iteratei s*)
⟨*proof*⟩
lemma *g-size-abort-alt*: *g-size-abort m s* = *iterate-size-abort* (*iteratei s*) *m*
⟨*proof*⟩

lemma *g-size-impl*: *set-size α invar g-size*
⟨*proof*⟩

lemma *g-size-abort-impl*: *set-size-abort α invar g-size-abort*
⟨*proof*⟩

lemma *g-isSng-alt*: *g-isSng s* = *iterate-is-sng* (*iteratei s*)
⟨*proof*⟩

lemma *g-isSng-impl*: *set-isSng α invar g-isSng*
⟨*proof*⟩

lemma *g-union-impl*: *set-union α invar α invar α invar g-union*
⟨*proof*⟩

lemma *g-diff-impl*: *set-diff α invar α invar g-diff*
⟨*proof*⟩

lemma *g-subset-impl*: *set-subset α invar α invar g-subset*
⟨*proof*⟩

lemma *g-equal-impl*: *set-equal α invar α invar g-equal*
⟨*proof*⟩

lemma *g-to-list-impl*: *set-to-list α invar g-to-list*
⟨*proof*⟩

lemma *g-from-list-impl*: *list-to-set α invar g-from-list*
⟨*proof*⟩

lemma *g-inter-impl*: *set-inter α invar α invar α invar g-inter*
⟨*proof*⟩

lemma *g-union-dj-impl*: *set-union-dj α invar α invar α invar g-union-dj*
⟨*proof*⟩

lemma *g-filter-impl*: *set-filter* α *invar* α *invar g-filter*
⟨*proof*⟩

lemma *g-disjoint-witness-impl*: *set-disjoint-witness*
α *invar* α *invar g-disjoint-witness*
⟨*proof*⟩

lemma *g-disjoint-impl*: *set-disjoint*
α *invar* α *invar g-disjoint*
⟨*proof*⟩

end

context *StdBasicSet*
begin
lemma *dflt-ops-impl*: *StdSet dflt-ops*
⟨*proof*⟩
end

context *StdBasicOSetDefs*
begin
definition *g-min s P* ≡ *iterateoi s* (λ*x*. *x* = *None*)
(λ*x* -. *if P x then Some x else None*) *None*
definition *g-max s P* ≡ *rev-iterateoi s* (λ*x*. *x* = *None*)
(λ*x* -. *if P x then Some x else None*) *None*

definition *g-to-sorted-list s* ≡ *rev-iterateo s* (#) []
definition *g-to-rev-list s* ≡ *iterateo s* (#) []

definition *dflt-oops* :: (*'x*::*linorder*,*'s*) *oset-ops*
where [*icf-rec-def*]:
dflt-oops ≡ *set-ops.extend*
dflt-ops
(|
set-op-ordered-list-it = *ordered-list-it*,
set-op-rev-list-it = *rev-list-it*,
set-op-min = *g-min*,
set-op-max = *g-max*,
set-op-to-sorted-list = *g-to-sorted-list*,
set-op-to-rev-list = *g-to-rev-list*
|)
⟨*ML*⟩

end

context *StdBasicOSet*
begin
lemma *g-min-impl*: *set-min* α *invar g-min*

⟨*proof*⟩

lemma *g-max-impl*: *set-max* α *invar g-max*
⟨*proof*⟩

lemma *g-to-sorted-list-impl*: *set-to-sorted-list* α *invar g-to-sorted-list*
⟨*proof*⟩

lemma *g-to-rev-list-impl*: *set-to-rev-list* α *invar g-to-rev-list*
⟨*proof*⟩

lemma *dflt-oops-impl*: *StdOSet dflt-oops*
⟨*proof*⟩

end

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 **locale** *image-filter-cp-defs-loc* =
s1: *StdSetDefs ops1* +
s2: *StdSetDefs ops2* +
s3: *StdSetDefs ops3*
for *ops1* :: ('*x*,'*s1*,'*more1*) *set-ops-scheme*
and *ops2* :: ('*y*,'*s2*,'*more2*) *set-ops-scheme*
and *ops3* :: ('*z*,'*s3*,'*more3*) *set-ops-scheme*
begin

definition *image-filter-cartesian-product f s1 s2* ==
s1.iterate s1 (λ*x res*.
s2.iterate s2 (λ*y res*.
case (*f* (*x*, *y*)) *of*
None ⇒ *res*
| *Some z* ⇒ (*s3.ins z res*)
) *res*
) (*s3.empty* ())

lemma *image-filter-cartesian-product-alt*:
image-filter-cartesian-product f s1 s2 ==
iterate-to-set s3.empty s3.ins (*set-iterator-image-filter f* (
set-iterator-product (*s1.iteratei s1*) (λ-. *s2.iteratei s2*)))
⟨*proof*⟩

definition *image-filter-cp* **where**
image-filter-cp f P s1 s2 ≡
image-filter-cartesian-product

```
    (λxy. if P xy then Some (f xy) else None) s1 s2

end

locale image-filter-cp-loc = image-filter-cp-defs-loc ops1 ops2 ops3 +
  s1: StdSet ops1 +
  s2: StdSet ops2 +
  s3: StdSet ops3
  for ops1 :: ('x,'s1,'more1) set-ops-scheme
  and ops2 :: ('y,'s2,'more2) set-ops-scheme
  and ops3 :: ('z,'s3,'more3) set-ops-scheme
begin

  lemma image-filter-cartesian-product-correct:
    fixes f :: 'x × 'y ⇀ 'z
    assumes I[simp, intro!]: s1.invar s1    s2.invar s2
    shows s3.α (image-filter-cartesian-product f s1 s2)
     = { z | x y z. f (x,y) = Some z ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (is ?T1)
    s3.invar (image-filter-cartesian-product f s1 s2) (is ?T2)
  ⟨proof⟩

  lemma image-filter-cp-correct:
    assumes I: s1.invar s1    s2.invar s2
    shows
    s3.α (image-filter-cp f P s1 s2)
     = { f (x, y) | x y. P (x, y) ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (is ?T1)
    s3.invar (image-filter-cp f P s1 s2) (is ?T2)
  ⟨proof⟩

end

locale inj-image-filter-cp-defs-loc =
  s1: StdSetDefs ops1 +
  s2: StdSetDefs ops2 +
  s3: StdSetDefs ops3
  for ops1 :: ('x,'s1,'more1) set-ops-scheme
  and ops2 :: ('y,'s2,'more2) set-ops-scheme
  and ops3 :: ('z,'s3,'more3) set-ops-scheme
begin

  definition inj-image-filter-cartesian-product f s1 s2 ==
    s1.iterate s1 (λx res.
      s2.iterate s2 (λy res.
        case (f (x, y)) of
          None ⇒ res
        | Some z ⇒ (s3.ins-dj z res)
      ) res
    ) (s3.empty ())
```

lemma *inj-image-filter-cartesian-product-alt*:
inj-image-filter-cartesian-product f s1 s2 ==
iterate-to-set s3.empty s3.ins-dj (set-iterator-image-filter f (
set-iterator-product (s1.iteratei s1) (λ-. s2.iteratei s2)))
⟨*proof*⟩

definition *inj-image-filter-cp* **where**
inj-image-filter-cp f P s1 s2 ≡
inj-image-filter-cartesian-product
(λxy. if P xy then Some (f xy) else None) s1 s2

end

locale *inj-image-filter-cp-loc = inj-image-filter-cp-defs-loc ops1 ops2 ops3 +*
s1: *StdSet ops1 +*
s2: *StdSet ops2 +*
s3: *StdSet ops3*
for *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'y*,*'s2*,*'more2*) *set-ops-scheme*
and *ops3* :: (*'z*,*'s3*,*'more3*) *set-ops-scheme*
begin

lemma *inj-image-filter-cartesian-product-correct*:
fixes *f* :: *'x × 'y ⇀ 'z*
assumes *I*[*simp*, *intro*!]: *s1.invar s1 s2.invar s2*
assumes *INJ*: *inj-on f (s1.α s1 × s2.α s2 ∩ dom f)*
shows *s3.α (inj-image-filter-cartesian-product f s1 s2)*
= { z | x y z. f (x,y) = Some z ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (**is** *?T1*)
s3.invar (inj-image-filter-cartesian-product f s1 s2) (**is** *?T2*)
⟨*proof*⟩

lemma *inj-image-filter-cp-correct*:
assumes *I*: *s1.invar s1 s2.invar s2*
assumes *INJ*: *inj-on f {x∈s1.α s1 × s2.α s2. P x}*
shows
s3.α (inj-image-filter-cp f P s1 s2)
= { f (x, y) | x y. P (x, y) ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (**is** *?T1*)
s3.invar (inj-image-filter-cp f P s1 s2) (**is** *?T2*)
⟨*proof*⟩

end

Cartesian Product **locale** *cart-defs-loc = inj-image-filter-cp-defs-loc ops1 ops2 ops3*
for *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'y*,*'s2*,*'more2*) *set-ops-scheme*
and *ops3* :: (*'x×'y*,*'s3*,*'more3*) *set-ops-scheme*
begin

definition *cart s1 s2* ≡
s1.iterate s1
(λ*x*. *s2.iterate s2* (λ*y res*. *s3.ins-dj* (*x*,*y*) *res*))
(*s3.empty* ())

lemma *cart-alt*: *cart s1 s2* ==
inj-image-filter-cartesian-product Some s1 s2
⟨*proof*⟩

end

locale *cart-loc* = *cart-defs-loc ops1 ops2 ops3*
+ *inj-image-filter-cp-loc ops1 ops2 ops3*
for *ops1* :: ('*x*,'*s1*,'*more1*) *set-ops-scheme*
and *ops2* :: ('*y*,'*s2*,'*more2*) *set-ops-scheme*
and *ops3* :: ('*x*×'*y*,'*s3*,'*more3*) *set-ops-scheme*
begin

lemma *cart-correct*:
assumes *I*[*simp*, *intro*!]: *s1.invar s1* *s2.invar s2*
shows *s3.α* (*cart s1 s2*)
= *s1.α s1* × *s2.α s2* (**is** *?T1*)
s3.invar (*cart s1 s2*) (**is** *?T2*)
⟨*proof*⟩

end

Generic Algorithms outside basic-set

In this section, we present some generic algorithms that are not formulated in terms of basic-set. They are useful for setting up some data structures.

Image (by image-filter)

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

lemma *iflt-image-correct*:
assumes *set-image-filter α1 invar1 α2 invar2 iflt*
shows *set-image α1 invar1 α2 invar2* (*iflt-image iflt*)
⟨*proof*⟩

Injective Image-Filter (by image-filter)

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

lemma *iflt-inj-image-correct*:
assumes *set-inj-image-filter α1 invar1 α2 invar2 iflt*
shows *set-inj-image α1 invar1 α2 invar2* (*iflt-inj-image iflt*)
⟨*proof*⟩

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*⟩

end

3.2.4 Implementing Sets by Maps

theory *SetByMap*
imports
../spec/SetSpec
../spec/MapSpec
SetGA
MapGA
begin

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

Auxiliary lemma

lemma *foldli-foldli-map-eq*:
foldli (*foldli l* (λx. *True*) ($\lambda x\ l$. *l*@[*f x*]) []) *c f'* $\sigma 0$
= *foldli l c* (*f' o f*) $\sigma 0$
⟨*proof*⟩

locale *SetByMapDefs* =
map: *StdBasicMapDefs ops*
for *ops* :: ($'x$,*unit*,$'s$,$'more$) *map-basic-ops-scheme*
begin
definition α *s* ≡ *dom* (*map.*α *s*)
definition *invar s* ≡ *map.invar s*
definition *empty* **where** *empty* ≡ *map.empty*
definition *memb x s* ≡ *map.lookup x s* ≠ *None*
definition *ins x s* ≡ *map.update x* () *s*
definition *ins-dj x s* ≡ *map.update-dj x* () *s*
definition *delete x s* ≡ *map.delete x s*
definition *list-it* :: $'s \Rightarrow$ ($'x$,$'x$ *list*) *set-iterator*
where *list-it s c f* $\sigma 0$ ≡ *it-to-it* (*map.list-it s*) *c* (*f o fst*) $\sigma 0$

⟨*ML*⟩

lemma *list-it-alt*: *list-it s* = *map-iterator-dom* (*map.iteratei s*)
⟨*proof*⟩

lemma *list-it-unfold*:
it-to-it (*list-it s*) *c f σ0* = *map.iteratei s c* (*f o fst*) *σ0*
⟨*proof*⟩

definition [*icf-rec-def*]: *dflt-basic-ops* ≡ (|
bset-op-α = *α*,
bset-op-invar = *invar*,
bset-op-empty = *empty*,
bset-op-memb = *memb*,
bset-op-ins = *ins*,
bset-op-ins-dj = *ins-dj*,
bset-op-delete = *delete*,
bset-op-list-it = *list-it*
|)
⟨*ML*⟩

end

⟨*ML*⟩

locale *SetByMap* = *SetByMapDefs ops* +
map: *StdBasicMap ops*
for *ops* :: ('*x*,*unit*,'*s*,'*more*) *map-basic-ops-scheme*
begin
lemma *empty-impl*: *set-empty α invar empty*
⟨*proof*⟩

lemma *memb-impl*: *set-memb α invar memb*
⟨*proof*⟩

lemma *ins-impl*: *set-ins α invar ins*
⟨*proof*⟩

lemma *ins-dj-impl*: *set-ins-dj α invar ins-dj*
⟨*proof*⟩

lemma *delete-impl*: *set-delete α invar delete*
⟨*proof*⟩

lemma *list-it-impl*: *poly-set-iteratei α invar list-it*
⟨*proof*⟩

lemma *dflt-basic-ops-impl*: *StdBasicSet dflt-basic-ops*
⟨*proof*⟩
end

locale *OSetByOMapDefs* = *SetByMapDefs ops* +
  map: *StdBasicOMapDefs ops*
  for *ops* :: ('*x*::*linorder*,*unit*,'*s*,'*more*) *omap-basic-ops-scheme*
begin
  definition *ordered-list-it* :: '*s* ⇒ ('*x*,'*x list*) *set-iterator*
    where *ordered-list-it s c f σ0*
    ≡ *it-to-it* (*map.ordered-list-it s*) *c* (*f o fst*) *σ0*

  ⟨*ML*⟩

  definition *rev-list-it* :: '*s* ⇒ ('*x*,'*x list*) *set-iterator*
    where *rev-list-it s c f σ0* ≡ *it-to-it* (*map.rev-list-it s*) *c* (*f o fst*) *σ0*

  ⟨*ML*⟩

  definition [*icf-rec-def*]: *dflt-basic-oops* ≡
    set-basic-ops.extend dflt-basic-ops (‖
      bset-op-ordered-list-it = *ordered-list-it*,
      bset-op-rev-list-it = *rev-list-it*
      ‖)
  ⟨*ML*⟩

end

⟨*ML*⟩

locale *OSetByOMap* = *OSetByOMapDefs ops* +
  SetByMap ops + *map*: *StdBasicOMap ops*
  for *ops* :: ('*x*::*linorder*,*unit*,'*s*,'*more*) *omap-basic-ops-scheme*
begin
  lemma *ordered-list-it-impl*: *poly-set-iterateoi α invar ordered-list-it*
  ⟨*proof*⟩

  lemma *rev-list-it-impl*: *poly-set-rev-iterateoi α invar rev-list-it*
  ⟨*proof*⟩

  lemma *dflt-basic-oops-impl*: *StdBasicOSet dflt-basic-oops*
  ⟨*proof*⟩
end

sublocale *SetByMap* < *basic*: *StdBasicSet dflt-basic-ops*
  ⟨*proof*⟩

sublocale *OSetByOMap* < *obasic*: *StdBasicOSet dflt-basic-oops*
  ⟨*proof*⟩

lemma *proper-it'-map2set*: *proper-it' it it'*
⟹ *proper-it'* (λ*s c f*. *it s c* (*f o fst*)) (λ*s c f*. *it' s c* (*f o fst*))
⟨*proof*⟩

end

3.2.5 Generic Algorithms for Sequences

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

Iterators

iteratei (by get, size) **locale** *idx-iteratei-loc* =
list-size + *list-get* +
constrains α :: *'s* ⇒ *'a list*
assumes [*simp*]: ⋀*s*. *invar s*
begin

fun *idx-iteratei-aux*
:: *nat* ⇒ *nat* ⇒ *'s* ⇒ (*'σ*⇒*bool*) ⇒ (*'a* ⇒*'σ* ⇒ *'σ*) ⇒ *'σ* ⇒ *'σ*
where
idx-iteratei-aux sz i l c f σ = (
if i=0 ∨ ¬ *c σ then σ*
else idx-iteratei-aux 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 ⟹ *idx-iteratei-aux sz i l c f σ* = *σ*
¬*c σ* ⟹ *idx-iteratei-aux sz i l c f σ* = *σ*
⟦*i≠0*; *c σ*⟧ ⟹ *idx-iteratei-aux sz i l c f σ* = *idx-iteratei-aux sz* (*i* − *1*) *l c f* (*f* (*get l* (*sz*−*i*)) *σ*)
⟨*proof*⟩

definition *idx-iteratei* **where**
idx-iteratei l c f σ ≡ *idx-iteratei-aux* (*size l*) (*size l*) *l c f σ*

lemma *idx-iteratei-correct*:
shows *idx-iteratei s* = *foldli* (α *s*)
⟨*proof*⟩

lemmas *idx-iteratei-unfold*[*code-unfold*] = *idx-iteratei-correct*[*symmetric*]

reverse_iteratei (by get, size) **fun** *idx-reverse-iteratei-aux*
:: *nat* ⇒ *nat* ⇒ *'s* ⇒ (*'σ*⇒*bool*) ⇒ (*'a* ⇒*'σ* ⇒ *'σ*) ⇒ *'σ* ⇒ *'σ*

where
idx-reverse-iteratei-aux sz i l c f σ = (
if i=0 ∨ ¬ c σ then σ
else idx-reverse-iteratei-aux sz (i − 1) l c f (f (get l (i − 1)) σ)
)

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

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

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

lemma *idx-reverse-iteratei-correct*:
shows *idx-reverse-iteratei s = foldri (α s)*
⟨*proof*⟩

lemmas *idx-reverse-iteratei-unfold*[*code-unfold*]
= idx-reverse-iteratei-correct[*symmetric*]

end

Size (by iterator)

locale *it-size-loc = poly-list-iteratei +*
constrains *α :: 's ⇒ 'a list*
begin

definition *it-size :: 's ⇒ nat*
where *it-size l == iterate l (λx res. Suc res) (0::nat)*

lemma *it-size-impl*: **shows** *list-size α invar it-size*
⟨*proof*⟩
end

Size (by reverse_iterator)

locale *rev-it-size-loc = poly-list-rev-iteratei +*
constrains *α :: 's ⇒ 'a list*
begin

definition *rev-it-size :: 's ⇒ nat*
where *rev-it-size l == rev-iterate l (λx res. Suc res) (0::nat)*

lemma *rev-it-size-impl*:
shows *list-size α invar rev-it-size*

⟨*proof*⟩

end

Get (by iteratori)

locale *it-get-loc* = *poly-list-iteratei* +
  constrains α :: $'s \Rightarrow 'a$ *list*
begin

  definition *it-get*:: $'s \Rightarrow$ *nat* $\Rightarrow 'a$
    where *it-get s i* ≡
      the (*snd* (*iteratei s*
            (λ(*i*,*x*). *x*=*None*)
            (λ*x* (*i*,-). *if i*=*0 then* (*0*,*Some x*) *else* (*i* − *1*,*None*))
            (*i*,*None*)))

  lemma *it-get-correct*:
    shows *list-get* α *invar it-get*
  ⟨*proof*⟩
end

end

3.2.6 Indices of Sets

theory *SetIndex*
imports
  ../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.

Indexing by Function

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

lemma *indexI*: ⟦ *x*∈*s*; *f x* = *i* ⟧ ⟹ *x*∈*index f s i* ⟨*proof*⟩
lemma *indexD*:
  x∈*index f s i* ⟹ *x*∈*s*
  x∈*index f s i* ⟹ *f x* = *i*
  ⟨*proof*⟩

lemma *index-iff*[*simp*]: *x*∈*index f s i* ⟷ *x*∈*s* ∧ *f x* = *i* ⟨*proof*⟩

Indexing by Map

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

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

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

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.

locale *index-loc* =
m: *StdMap m-ops* +
s: *StdSet s-ops*
for *m-ops* :: ('*i*,'*s*,'*m*,'*more1*) *map-ops-scheme*
and *s-ops* :: ('*x*,'*s*,'*more2*) *set-ops-scheme*
begin
— Mapping indices to abstract indices
definition *ci-α′* **where**
ci-α′ ci i == *case m.α ci i of None* ⇒ *None* | *Some s* ⇒ *Some* (*s.α s*)

definition *ci-α* == *im-α* ∘ *ci-α′*

definition *ci-invar* **where**
ci-invar ci == *m.invar ci* ∧ (∀ *i s*. *m.α ci i* = *Some s* ⟶ *s.invar s*)

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

definition *is-index* :: $('x \Rightarrow 'i) \Rightarrow 'x\ set \Rightarrow 'm \Rightarrow bool$
where
is-index f s idx == *ci-invar idx* ∧ *ci-α′ idx* = *index-map f s*

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

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

definition *lookup* :: $'i \Rightarrow 'm \Rightarrow 's$ **where**
lookup i m == *case m.lookup i m of None* ⇒ (*s.empty* ()) | *Some s* ⇒ *s*

lemma *lookup-invar′*: *ci-invar m* ⟹ *s.invar* (*lookup i m*)
⟨*proof*⟩

lemma *lookup-correct*:
assumes *I*[*simp*, *intro*!]: *is-index f s idx*
shows
s.α (*lookup i idx*) = *index f s i*
s.invar (*lookup i idx*)
⟨*proof*⟩

end

locale *build-index-loc* = *index-loc m-ops s-ops* +
t: *StdSet t-ops*
for *m-ops* :: ('*i*,'*s*,'*m*,'*more1*) *map-ops-scheme*
and *s-ops* :: ('*x*,'*s*,'*more3*) *set-ops-scheme*
and *t-ops* :: ('*x*,'*t*,'*more2*) *set-ops-scheme*
begin

Building indices

definition *idx-build-stepfun* :: ('*x* ⇒ '*i*) ⇒ '*x* ⇒ '*m* ⇒ '*m* **where**
idx-build-stepfun f x m ==
let i=*f x in*
(*case m.lookup i m of*
None ⇒ *m.update i* (*s.ins x* (*s.empty* ())) *m* |
Some s ⇒ *m.update i* (*s.ins x s*) *m*
)

definition *idx-build* :: ('*x* ⇒ '*i*) ⇒ '*t* ⇒ '*m* **where**
idx-build f t == *t.iterate t* (*idx-build-stepfun f*) (*m.empty* ())

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

lemma *idx-build-is-index*:
t.invar t ⟹ *is-index f* (*t.α t*) (*idx-build f t*)
⟨*proof*⟩

end

end

3.2.7 More Generic Algorithms

theory *Algos*
imports
../spec/SetSpec
../spec/MapSpec

../spec/ListSpec
begin

Injective Map to Naturals

Whether a set is an initial segment of the natural numbers

definition *inatseg* :: *nat set* ⇒ *bool*
where *inatseg s* == ∃ *k*. *s* = {*i*::*nat*. *i*<*k*}

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

Compute an injective map from objects into an initial segment of the natural numbers

locale *map-to-nat-loc* =
s: *StdSet s-ops* +
m: *StdMap m-ops*
for *s-ops* :: ('*x*,'*s*,'*more1*) *set-ops-scheme*
and *m-ops* :: ('*x*,*nat*,'*m*,'*more2*) *map-ops-scheme*
begin

definition *map-to-nat*
:: '*s* ⇒ '*m* **where**
map-to-nat s ==
snd (*s.iterate s* (λ*x* (*c*,*m*). (*c*+*1*,*m.update x c m*)) (*0*,*m.empty* ()))

lemma *map-to-nat-correct*:
assumes *INV*[*simp*]: *s.invar s*
shows
— All elements have got a number
dom (*m.α* (*map-to-nat s*)) = *s.α s* (**is** *?T1*) **and**
— No two elements got the same number
[*rule-format*]: *inj-on* (*m.α* (*map-to-nat s*)) (*s.α s*) (**is** *?T2*) **and**
— Numbering is inatseg
[*rule-format*]: *inatseg* (*ran* (*m.α* (*map-to-nat s*))) (**is** *?T3*) **and**
— The result satisfies the map invariant
m.invar (*map-to-nat s*) (**is** *?T4*)
⟨*proof*⟩

end

Map from Set

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

locale *it-dom-fun-to-map-loc* =
s: *StdSet s-ops*

```
+ m: StdMap m-ops
  for s-ops :: ('k,'s,'more1) set-ops-scheme
  and m-ops :: ('k,'v,'m,'more2) map-ops-scheme
begin

  definition it-dom-fun-to-map ::
    's ⇒ ('k ⇒ 'v) ⇒ 'm
    where it-dom-fun-to-map s f ==
          s.iterate s (λk m. m.update-dj k (f k) m) (m.empty ())

  lemma it-dom-fun-to-map-correct:
    assumes INV: s.invar s
    shows m.α (it-dom-fun-to-map s f) k
      = (if k ∈ s.α s then Some (f k) else None) (is ?G1)
    and m.invar (it-dom-fun-to-map s f) (is ?G2)
  ⟨proof⟩

end


locale set-to-list-defs-loc =
  s: StdSetDefs s-ops
+ l: StdListDefs l-ops
  for s-ops :: ('x,'s,'more1) set-ops-scheme
  and l-ops :: ('x,'l,'more2) list-ops-scheme
begin
  definition g-set-to-listl s ≡ s.iterate s l.appendl (l.empty ())
  definition g-set-to-listr s ≡ s.iterate s l.appendr (l.empty ())
end

locale set-to-list-loc = set-to-list-defs-loc s-ops l-ops
+ s: StdSet s-ops
+ l: StdList l-ops
  for s-ops :: ('x,'s,'more1) set-ops-scheme
  and l-ops :: ('x,'l,'more2) list-ops-scheme
begin
  lemma g-set-to-listl-correct:
    assumes I: s.invar s
    shows List.set (l.α (g-set-to-listl s)) = s.α s
    and l.invar (g-set-to-listl s)
    and distinct (l.α (g-set-to-listl s))
    ⟨proof⟩

  lemma g-set-to-listr-correct:
    assumes I: s.invar s
    shows List.set (l.α (g-set-to-listr s)) = s.α s
    and l.invar (g-set-to-listr s)
    and distinct (l.α (g-set-to-listr s))
    ⟨proof⟩
```

end

end

3.2.8 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.

Definitions

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

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

fun *p-min* :: $('e, 'a::linorder)$ *Prio* $\Rightarrow$ $('e, 'a)$ *Prio* $\Rightarrow$ $('e, 'a)$ *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* $a \leq b$ *then Prio e1 a else Prio e2 b*)

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

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

instance ⟨*proof*⟩
end

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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

Correctness

Auxiliary Lemmas **lemma** *sum-list-split*: *sum-list* (*l* @ (*a*::'*a*::*monoid-add*)
r) = (*sum-list l*) + *a* + (*sum-list r*)
⟨*proof*⟩

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

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

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

lemma *ls-min*: $\forall x \in$ *set* (*xs*:: (*'e*,*'a*::*linorder*) *Prio list*) . *sum-list xs* $\leq x$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

Insert **lemma** *alprio-insert-correct*:

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

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

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

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

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

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

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

lemma *alpriodel-invar*:
assumes *alprio-invar* α *invar s*
and *al-annot* α *invar annot*
and *alprio-*α α *s* $\neq \{\#\}$
and *al-splits* α *invar splits*
and *al-app* α *invar app*
shows *alprio-invar* α *invar* (*alprio-delete splits annot app s*)
⟨*proof*⟩

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

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*⟩

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

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

locale *alprio-defs* = *StdALDefs ops*
  for *ops* :: (*unit*,(*'e*,*'a*::*linorder*) *Prio*,*'s*) *alist-ops*
begin
  definition [*icf-rec-def*]: *alprio-ops* ≡ (|
    prio-op-α = *alprio-α* α,
    prio-op-invar = *alprio-invar* α *invar*,
    prio-op-empty = *alprio-empty empty*,
    prio-op-isEmpty = *alprio-isEmpty isEmpty*,
    prio-op-insert = *alprio-insert consl*,
    prio-op-find = *alprio-find annot*,
    prio-op-delete = *alprio-delete splits annot app*,
    prio-op-meld = *alprio-meld app*
    |)

end

locale *alprio* = *alprio-defs ops* + *StdAL ops*
  for *ops* :: (*unit*,('*e*,'*a*::*linorder*) *Prio*,'*s*) *alist-ops*
begin
  lemma *alprio-ops-impl*: *StdPrio alprio-ops*
    ⟨*proof*⟩
end

end

3.2.9 Implementing Unique Priority Queues by Annotated Lists

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

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

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

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

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

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

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*|

[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.

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* ⟨*proof*⟩
lemma *p-min-le-neut*[*simp*]: *p-min Infty a = a* ⟨*proof*⟩
lemma *p-min-asso*: *p-min* (*p-min a b*) *c* = *p-min a* (*p-min b c*)
⟨*proof*⟩

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

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

instance ⟨*proof*⟩
end

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

Operations **definition** *aluprio-α* :: (*'s* ⇒ (*unit* × (*'e*::*linorder*,*'a*::*linorder*) *LP*) *list*)
⇒ *'s* ⇒ (*'e*::*linorder* ⇀ *'a*::*linorder*)
where
aluprio-α α ft == (*map-of* (*map p-unwrap* (*map snd* (*α ft*))))

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

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

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

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

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

```
        app (consr l () (LP e a)) r  ))
    else
      consr s () (LP e a))


definition aluprio-pop :: ((('e::linorder,'a::linorder) LP ⇒ bool) ⇒ ('e,'a) LP
  ⇒ 's ⇒ ('s × (unit × ('e,'a) LP) × 's))
    ⇒ ('s ⇒ ('e,'a) LP)
      ⇒ ('s ⇒ 's ⇒ 's)
        ⇒ 's
          ⇒ 'e ×'a ×'s
  where
  aluprio-pop splits annot app s =
    (let (l, (-,lp) , r) = splits (λ x. x ≤ (annot s)) Infty s
    in
      (case lp of
        (LP e a) ⇒
          (e, a, app l r) ))

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

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


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

Correctness

Auxiliary Lemmas **lemma** *p-linear*: $(x::('e, 'a::linorder)\ LP) \leq y \vee y \leq x$
⟨*proof*⟩

lemma *e-less-eq-mon1*: *e-less-eq e x* $\Longrightarrow$ *e-less-eq e* $(x + y)$
⟨*proof*⟩
lemma *e-less-eq-mon2*: *e-less-eq e y* $\Longrightarrow$ *e-less-eq e* $(x + y)$
⟨*proof*⟩
lemmas *e-less-eq-mon* =
e-less-eq-mon1
e-less-eq-mon2

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

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

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

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

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

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

lemma *e-less-eq-sum-list2*:
assumes
$\forall x{\in}set\ (\alpha s).$ *snd x* $\neq Infty$
$((), b) \in set\ (\alpha s)$
shows *e-less-eq* (*fst* (*p-unwrap b*)) (*sum-list* (*map snd* (αs)))
⟨*proof*⟩

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

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

lemma *sum-list-less-elems*: ∀ *x*∈*set xs*. *snd x* ≠ *Infty* ⟹
∀ *y*∈*set* (*map snd* (*map p-unwrap* (*map snd xs*))).
snd (*p-unwrap* (*sum-list* (*map snd xs*))) ≤ *y*
⟨*proof*⟩

lemma *distinct-sortet-list-app*:
⟦*sorted xs*; *distinct xs*; *xs* = *as* @ *b* # *cs*⟧
⟹ ∀ *x*∈ *set cs*. *b* < *x*
⟨*proof*⟩

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*⟩

lemma *distinct-sorted-list-lem2*:
assumes
sorted xs
sorted ys
distinct xs
distinct ys
e < *e*′
∀ *x* ∈ *set xs*. *x* < *e*
∀ *y* ∈ *set ys*. *e*′ < *y*
shows
sorted (*xs* @ *e* # *e*′ # *ys*)
distinct (*xs* @ *e* # *e*′ # *ys*)
⟨*proof*⟩

lemma *map-of-distinct-upd*:
x ∉ *set* (*map fst xs*) ⟹ [*x* ↦ *y*] ++ *map-of xs* = (*map-of xs*) (*x* ↦ *y*)
⟨*proof*⟩

lemma *map-of-distinct-upd2*:
assumes *x* ∉ *set*(*map fst xs*)
x ∉ *set* (*map fst ys*)
shows *map-of* (*xs* @ (*x*,*y*) # *ys*) = (*map-of* (*xs* @ *ys*))(*x* ↦ *y*)

⟨*proof*⟩

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

lemma *map-of-distinct-upd4*:
assumes $x \notin set(map\ fst\ xs)$
$x \notin set\ (map\ fst\ ys)$
shows $map\text{-}of\ (xs\ @\ ys) = (map\text{-}of\ (xs\ @\ (x,y)\ \#\ ys))(x := None)$
⟨*proof*⟩

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

lemma *ran-distinct*:
assumes *dist*: $distinct\ (map\ fst\ al)$
shows $ran\ (map\text{-}of\ al) = snd\ `\ set\ al$
⟨*proof*⟩

Finite **lemma** *aluprio-finite-correct*: $uprio\text{-}finite\ (aluprio\text{-}\alpha\ \alpha)\ (aluprio\text{-}invar\ \alpha\ invar)$
⟨*proof*⟩

Empty **lemma** *aluprio-empty-correct*:
assumes $al\text{-}empty\ \alpha\ invar\ empt$
shows $uprio\text{-}empty\ (aluprio\text{-}\alpha\ \alpha)\ (aluprio\text{-}invar\ \alpha\ invar)\ (aluprio\text{-}empty\ empt)$
⟨*proof*⟩

Is Empty **lemma** *aluprio-isEmpty-correct*:
assumes $al\text{-}isEmpty\ \alpha\ invar\ isEmpty$
shows $uprio\text{-}isEmpty\ (aluprio\text{-}\alpha\ \alpha)\ (aluprio\text{-}invar\ \alpha\ invar)\ (aluprio\text{-}isEmpty\ isEmpty)$
⟨*proof*⟩

Insert **lemma** *annot-inf*:
assumes *A*: $invar\ s \quad \forall x{\in}set\ (\alpha\ s).\ snd\ x \neq Infty \quad al\text{-}annot\ \alpha\ invar\ annot$
shows $annot\ s = Infty \longleftrightarrow \alpha\ s = []$
⟨*proof*⟩

lemma *e-less-eq-annot*:

assumes $al\text{-}annot\ \alpha\ invar\ annot$
$invar\ s \quad \forall x{\in}set\ (\alpha\ s).\ snd\ x \neq Infty \quad \neg\ e\text{-}less\text{-}eq\ e\ (annot\ s)$

shows $\forall x \in set\ (map\ (fst \circ (p\text{-}unwrap \circ snd))\ (\alpha\ s)).\ x < e$
⟨*proof*⟩

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

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

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

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

locale *aluprio-defs* = *StdALDefs ops*
for *ops* :: (*unit*,('*e*::*linorder*,'*a*::*linorder*) *LP*,'*s*) *alist-ops*
begin
definition [*icf-rec-def*]: *aluprio-ops* ≡ (|
upr-α = *aluprio-*α α,
upr-invar = *aluprio-invar* α *invar*,
upr-empty = *aluprio-empty empty*,
upr-isEmpty = *aluprio-isEmpty isEmpty*,
upr-insert = *aluprio-insert splits annot isEmpty app consr*,

upr-pop = aluprio-pop splits annot app,
upr-prio = aluprio-prio splits annot isEmpty
(|)

end

locale *aluprio = aluprio-defs ops + StdAL ops*
for *ops* :: (*unit*,(′*e*::*linorder*,′*a*::*linorder*) *LP*,′*s*) *alist-ops*
begin
lemma *aluprio-ops-impl*: *StdUprio aluprio-ops*
⟨*proof*⟩
end

end

3.3 Implementations

3.3.1 Map Implementation by Associative Lists

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

definition [*icf-rec-def*]: *lm-basic-ops* ≡ (|
bmap-op-α = Assoc-List.lookup,
bmap-op-invar = λ-. *True*,
bmap-op-empty = (λ-::*unit*. *Assoc-List.empty*),
bmap-op-lookup = (λ*k m*. *Assoc-List.lookup m k*),
bmap-op-update = Assoc-List.update,
bmap-op-update-dj = Assoc-List.update,
bmap-op-delete = Assoc-List.delete,
bmap-op-list-it = Assoc-List.iteratei
|)

⟨*ML*⟩
interpretation *lm-basic*: *StdBasicMapDefs lm-basic-ops* ⟨*proof*⟩
interpretation *lm-basic*: *StdBasicMap lm-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *lm-ops* ≡ *lm-basic.dflt-ops*
⟨*ML*⟩
interpretation *lm*: *StdMapDefs lm-ops* ⟨*proof*⟩
interpretation *lm*: *StdMap lm-ops*
⟨*proof*⟩
interpretation *lm*: *StdMap-no-invar lm-ops*

⟨*proof*⟩
⟨*ML*⟩

lemma *pi-lm*[*proper-it*]:
proper-it′ Assoc-List.iteratei Assoc-List.iteratei
⟨*proof*⟩

interpretation *pi-lm*: *proper-it-loc Assoc-List.iteratei Assoc-List.iteratei*
⟨*proof*⟩

lemma *pi-lm′*[*proper-it*]:
proper-it′ lm.iteratei lm.iteratei
⟨*proof*⟩

interpretation *pi-lm′*: *proper-it-loc lm.iteratei lm.iteratei*
⟨*proof*⟩

Code generator test

definition *test-codegen* ≡ (
lm.add ,
lm.add-dj ,
lm.ball ,
lm.bex ,
lm.delete ,
lm.empty ,
lm.isEmpty ,
lm.isSng ,
lm.iterate ,
lm.iteratei ,
lm.list-it ,
lm.lookup ,
lm.restrict ,
lm.sel ,
lm.size ,
lm.size-abort ,
lm.sng ,
lm.to-list ,
lm.to-map ,
lm.update ,
lm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.2 Map Implementation by Association Lists with explicit invariants

theory *ListMapImpl-Invar*

imports
../spec/MapSpec
../../Lib/Assoc-List
../gen-algo/MapGA
begin type-synonym ('k,'v) *lmi* = ('k×'v) *list*

term *revg*

definition *lmi-α* ≡ *Map.map-of*
definition *lmi-invar* ≡ λm. *distinct* (*List.map fst m*)

definition *lmi-basic-ops* :: ('k,'v,('k,'v) *lmi*) *map-basic-ops*
where [*icf-rec-def*]: *lmi-basic-ops* ≡ ⦇
bmap-op-α = *lmi-α*,
bmap-op-invar = *lmi-invar*,
bmap-op-empty = (λ-::*unit*. []),
bmap-op-lookup = (λk m. *Map.map-of* m k),
bmap-op-update = *AList.update*,
bmap-op-update-dj = (λk v m. (k, v) # m),
bmap-op-delete = *AList.delete-aux*,
bmap-op-list-it = *foldli*
⦈

⟨*ML*⟩
interpretation *lmi-basic*: *StdBasicMapDefs lmi-basic-ops* ⟨*proof*⟩
interpretation *lmi-basic*: *StdBasicMap lmi-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *lmi-ops* ≡ *lmi-basic.dflt-ops* ⦇
map-op-add-dj := *revg*,
map-op-to-list := *id*,
map-op-size := *length*,
map-op-isEmpty := *case-list True* (λ- -. *False*),
map-op-isSng := (λl. *case l of* [-] ⇒ *True* | - ⇒ *False*)
⦈

⟨*ML*⟩
interpretation *lmi*: *StdMapDefs lmi-ops* ⟨*proof*⟩
interpretation *lmi*: *StdMap lmi-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-lmi*[*proper-it*]:
proper-it' foldli foldli
⟨*proof*⟩

interpretation *pi-lmi*: *proper-it-loc foldli foldli*
⟨*proof*⟩

definition *lmi-from-list-dj* :: ('*k*×'*v*) *list* ⇒ ('*k*,'*v*) *lmi* **where**
lmi-from-list-dj ≡ *id*

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

Code generator test

definition *test-codegen* ≡ (
lmi.add ,
lmi.add-dj ,
lmi.ball ,
lmi.bex ,
lmi.delete ,
lmi.empty ,
lmi.isEmpty ,
lmi.isSng ,
lmi.iterate ,
lmi.iteratei ,
lmi.list-it ,
lmi.lookup ,
lmi.restrict ,
lmi.sel ,
lmi.size ,
lmi.size-abort ,
lmi.sng ,
lmi.to-list ,
lmi.to-map ,
lmi.update ,
lmi.update-dj,
lmi-from-list-dj
)

export-code *test-codegen* **checking** *SML*

end

3.3.3 Map Implementation by Red-Black-Trees

theory *RBTMapImpl*
imports
../spec/MapSpec
../../Lib/RBT-add
HOL−Library.RBT

../gen-algo/MapGA
begin hide-const (**open**) *RBT.map RBT.fold RBT.foldi RBT.empty RBT.insert*

type-synonym ('k,'v) *rm* = ('k,'v) *RBT.rbt*

definition *rm-basic-ops* :: ('k::*linorder*,'v,('k,'v) *rm*) *omap-basic-ops*
  where [*icf-rec-def*]: *rm-basic-ops* ≡ (|
  bmap-op-α = *RBT.lookup*,
  bmap-op-invar = λ-. *True*,
  bmap-op-empty = (λ-::*unit*. *RBT.empty*),
  bmap-op-lookup = (λk m. *RBT.lookup* m k),
  bmap-op-update = *RBT.insert*,
  bmap-op-update-dj = *RBT.insert*,
  bmap-op-delete = *RBT.delete*,
  bmap-op-list-it = (λr. *RBT-add.rm-iterateoi* (*RBT.impl-of* r)),
  bmap-op-ordered-list-it = (λr. *RBT-add.rm-iterateoi* (*RBT.impl-of* r)),
  bmap-op-rev-list-it = (λr. *RBT-add.rm-reverse-iterateoi* (*RBT.impl-of* r))
|)

⟨*ML*⟩
interpretation *rm-basic*: *StdBasicOMap rm-basic-ops*
  ⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *rm-ops* ≡ *rm-basic.dflt-oops*(|*map-op-add* := *RBT.union*|)
⟨*ML*⟩
interpretation *rm*: *StdOMap rm-ops*
⟨*proof*⟩

interpretation *rm*: *StdMap-no-invar rm-ops*
  ⟨*proof*⟩
⟨*ML*⟩

lemma *pi-rm*[*proper-it*]:
  proper-it' *RBT-add.rm-iterateoi RBT-add.rm-iterateoi*
  ⟨*proof*⟩
lemma *pi-rm-rev*[*proper-it*]:
  proper-it' *RBT-add.rm-reverse-iterateoi RBT-add.rm-reverse-iterateoi*
  ⟨*proof*⟩

interpretation *pi-rm*: *proper-it-loc RBT-add.rm-iterateoi RBT-add.rm-iterateoi*
  ⟨*proof*⟩
interpretation *pi-rm-rev*: *proper-it-loc RBT-add.rm-reverse-iterateoi*
  RBT-add.rm-reverse-iterateoi
  ⟨*proof*⟩

Code generator test

definition *test-codegen* ≡ (*rm.add* ,
rm.add-dj ,
rm.ball ,
rm.bex ,
rm.delete ,
rm.empty ,
rm.isEmpty ,
rm.isSng ,
rm.iterate ,
rm.iteratei ,
rm.iterateo ,
rm.iterateoi ,
rm.list-it ,
rm.lookup ,
rm.max ,
rm.min ,
rm.restrict ,
rm.rev-iterateo ,
rm.rev-iterateoi ,
rm.rev-list-it ,
rm.reverse-iterateo ,
rm.reverse-iterateoi ,
rm.sel ,
rm.size ,
rm.size-abort ,
rm.sng ,
rm.to-list ,
rm.to-map ,
rm.to-rev-list ,
rm.to-sorted-list ,
rm.update ,
rm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.4 Hash maps implementation

theory *HashMap-Impl*
imports
RBTMapImpl
ListMapImpl
../../Lib/HashCode
../../Lib/Code-Target-ICF
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.

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* **where** *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* $==$
$(\forall\, hc\ cm\ k.\ m\ hc = Some\ cm \wedge k \in dom\ cm \longrightarrow hashcode\ k = hc) \wedge$
$(\forall\, hc\ cm.\ m\ hc = Some\ cm \longrightarrow cm \neq Map.empty)$

— Abstract a hashmap to the corresponding map
definition *ahm-*α **where**
ahm-α *m k* $==$ *case m* (*hashcode k*) *of*
None $\Rightarrow$ *None* |
Some cm $\Rightarrow$ *cm k*

— Lookup an entry
definition *ahm-lookup* :: $'k$::*hashable* $\Rightarrow$ $('k,'v)$ *abs-hashmap* $\Rightarrow$ $'v$ *option*
where *ahm-lookup k m* $==$ (*ahm-*α *m*) *k*

— The empty hashmap
definition *ahm-empty* :: $('k$::*hashable*,$'v)$ *abs-hashmap*
where *ahm-empty* $=$ *Map.empty*

— Update/insert an entry
definition *ahm-update* **where**
ahm-update k v m $==$
case m (*hashcode k*) *of*
None $\Rightarrow$ *m* (*hashcode k* $\mapsto$ $[k \mapsto v]$) |
Some cm $\Rightarrow$ *m* (*hashcode k* $\mapsto$ *cm* $(k \mapsto v)$)

— 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}) = Map.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 (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 ≠ Map.empty*
⟧ ⟹ *ahm-invar m*
⟨*proof*⟩

lemma *ahm-invarD*: ⟦ *ahm-invar m*; *m hc = Some cm*; *k∈dom cm* ⟧ ⟹ *hashcode k = hc*
⟨*proof*⟩

lemma *ahm-invarDne*: ⟦ *ahm-invar m*; *m hc = Some cm* ⟧ ⟹ *cm ≠ Map.empty*
⟨*proof*⟩

lemma *ahm-invar-bucket-not-empty*[*simp*]:
ahm-invar m ⟹ *m hc ≠ Some Map.empty*
⟨*proof*⟩

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

lemma *ahm-empty-correct*:
ahm-α ahm-empty = Map.empty
ahm-invar ahm-empty
⟨*proof*⟩

lemma *ahm-update-correct*:
ahm-α (*ahm-update k v m*) = (*ahm-α m*)(*k* ↦ *v*)
ahm-invar m ⟹ *ahm-invar* (*ahm-update k v m*)
⟨*proof*⟩

lemma *fun-upd-apply-ne*: *x≠y* ⟹ (*f*(*x*:=*v*)) *y* = *f y*
⟨*proof*⟩

lemma *cancel-one-empty-simp*: $m \mid` (-\{k\}) = Map.empty \longleftrightarrow dom\ m \subseteq \{k\}$
⟨*proof*⟩

lemma *ahm-delete-correct*:
ahm-α (*ahm-delete k m*) = (*ahm-α m*) |` (−{*k*})
ahm-invar m ⟹ *ahm-invar* (*ahm-delete k m*)
⟨*proof*⟩

lemma *ahm-isEmpty-correct*: *ahm-invar m* ⟹ *ahm-isEmpty m* ⟷ *ahm-α m* = *Map.empty*
⟨*proof*⟩

lemmas *ahm-correct* = *ahm-empty-correct ahm-lookup-correct ahm-update-correct*

ahm-delete-correct ahm-isEmpty-correct

— Bucket entries correspond to map entries
lemma *ahm-be-is-e*:
assumes *I*: *ahm-invar m*
assumes *A*: *m hc* = *Some bm* *bm k* = *Some v*
shows *ahm-α m k* = *Some v*
⟨*proof*⟩
lemma *ahm-e-is-be*: ⟦
ahm-α m k = *Some v*;
!!*bm*. ⟦*m* (*hashcode k*) = *Some bm*; *bm k* = *Some v* ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

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 **definition** *rm-map-entry*
:: *hashcode* ⇒ ('*v option* ⇒ '*v option*) ⇒ (*hashcode*, '*v*) *rm* ⇒ (*hashcode*,'*v*) *rm*
where
rm-map-entry k f m ==
case rm.lookup k m of
None ⇒ (
case f None of
None ⇒ *m* |
Some v ⇒ *rm.update k v m*
) |

```
      Some v ⇒ (
        case f (Some v) of
          None ⇒ rm.delete k m |
          Some v' ⇒ rm.update k v' m
      )


  — Empty hashmap
definition empty :: unit ⇒ ('k :: hashable, 'v) hm-impl where empty == rm.empty

  — Update/insert entry
definition update :: 'k::hashable ⇒ 'v ⇒ ('k,'v) hm-impl ⇒ ('k,'v) hm-impl
  where
  update k v m ==
   let hc = hashcode k in
     case rm.lookup hc m of
       None ⇒ rm.update hc (lm.update k v (lm.empty ())) m |
       Some bm ⇒ rm.update hc (lm.update k v bm) m

  — Lookup value by key
definition lookup :: 'k::hashable ⇒ ('k,'v) hm-impl ⇒ 'v option where
  lookup k m ==
   case rm.lookup (hashcode k) m of
     None ⇒ None |
     Some lm ⇒ lm.lookup k lm

  — Delete entry by key
definition delete :: 'k::hashable ⇒ ('k,'v) hm-impl ⇒ ('k,'v) hm-impl where
  delete k m ==
   rm-map-entry (hashcode k)
     (λv. case v of
       None ⇒ None |
       Some lm ⇒ (
         let lm' = lm.delete k lm
         in  if lm.isEmpty lm' then None else Some lm'
       )
     ) m

  — Emptiness check
definition isEmpty == rm.isEmpty

  — Interruptible iterator
definition iteratei m c f σ0 ==
  rm.iteratei m c (λ(hc, lm) σ.
    lm.iteratei lm c f σ
  ) σ0

lemma iteratei-alt-def :
  iteratei m = set-iterator-image snd (
```

set-iterator-product (*rm.iteratei m*) (*λhclm. lm.iteratei* (*snd hclm*)))
⟨*proof*⟩

Correctness w.r.t. Abstract HashMap The following lemmas establish the correctness of the operations w.r.t. the abstract hashmap.

They have the naming scheme op_correct', where (is) the name of the operation.

definition *hm-α′* **where** *hm-α′ m* == *λhc. case rm.α m hc of*
None ⇒ *None* |
Some lm ⇒ *Some* (*lm.α lm*)

— Invariant for concrete hashmap: The hashcode-map and bucket-maps satisfy their invariants and the invariant of the corresponding abstract hashmap is satisfied.

definition *invar m* == *ahm-invar* (*hm-α′ m*)

lemma *rm-map-entry-correct*:
rm.α (*rm-map-entry k f m*) = (*rm.α m*)(*k* := *f* (*rm.α m k*))
⟨*proof*⟩

lemma *empty-correct′*:
hm-α′ (*empty* ()) = *ahm-empty*
invar (*empty* ())
⟨*proof*⟩

lemma *lookup-correct′*:
invar m ⟹ *lookup k m* = *ahm-lookup k* (*hm-α′ m*)
⟨*proof*⟩

lemma *update-correct′*:
invar m ⟹ *hm-α′* (*update k v m*) = *ahm-update k v* (*hm-α′ m*)
invar m ⟹ *invar* (*update k v m*)
⟨*proof*⟩

lemma *delete-correct′*:
invar m ⟹ *hm-α′* (*delete k m*) = *ahm-delete k* (*hm-α′ m*)
invar m ⟹ *invar* (*delete k m*)
⟨*proof*⟩

lemma *isEmpty-correct′*:
invar hm ⟹ *isEmpty hm* ⟷ *ahm-α* (*hm-α′ hm*) = *Map.empty*
⟨*proof*⟩

lemma *iteratei-correct′*:
assumes *invar*: *invar hm*

shows *map-iterator* (*iteratei hm*) (*ahm-α* (*hm-α′ hm*))
⟨*proof*⟩

lemmas *hm-correct′* = *empty-correct′ lookup-correct′ update-correct′*
delete-correct′ isEmpty-correct′
iteratei-correct′
lemmas *hm-invars* = *empty-correct′*(*2*) *update-correct′*(*2*)
delete-correct′(*2*)

hide-const (**open**) *empty invar lookup update delete isEmpty iteratei*

end

3.3.5 Hash Maps

theory *HashMap*
imports *HashMap-Impl*
begin

Type definition

typedef (**overloaded**) (*′k*, *′v*) *hashmap* = {*hm* :: (*′k* :: *hashable*, *′v*) *hm-impl*.
HashMap-Impl.invar hm}
morphisms *impl-of-RBT-HM RBT-HM*
⟨*proof*⟩

lemma *impl-of-RBT-HM-invar* [*simp*, *intro*!]: *HashMap-Impl.invar* (*impl-of-RBT-HM*
hm)
⟨*proof*⟩

lemma *RBT-HM-imp-of-RBT-HM* [*code abstype*]:
RBT-HM (*impl-of-RBT-HM hm*) = *hm*
⟨*proof*⟩

definition *hm-empty-const* :: (*′k* :: *hashable*, *′v*) *hashmap*
where *hm-empty-const* = *RBT-HM* (*HashMap-Impl.empty* ())

definition *hm-empty* :: *unit* ⇒ (*′k* :: *hashable*, *′v*) *hashmap*
where *hm-empty* = (*λ-*. *hm-empty-const*)

definition *hm-lookup k hm* == *HashMap-Impl.lookup k* (*impl-of-RBT-HM hm*)

definition *hm-update* :: (*′k* :: *hashable*) ⇒ *′v* ⇒ (*′k*, *′v*) *hashmap* ⇒ (*′k*, *′v*)
hashmap
where *hm-update k v hm* = *RBT-HM* (*HashMap-Impl.update k v* (*impl-of-RBT-HM*
hm))

definition *hm-update-dj* :: (*′k* :: *hashable*) ⇒ *′v* ⇒ (*′k*, *′v*) *hashmap* ⇒ (*′k*, *′v*)
hashmap

where *hm-update-dj* = *hm-update*

definition *hm-delete* :: (*'k* :: *hashable*) ⇒ (*'k*, *'v*) *hashmap* ⇒ (*'k*, *'v*) *hashmap*
where *hm-delete k hm* = *RBT-HM* (*HashMap-Impl.delete k* (*impl-of-RBT-HM hm*))

definition *hm-isEmpty* :: (*'k* :: *hashable*, *'v*) *hashmap* ⇒ *bool*
where *hm-isEmpty hm* = *HashMap-Impl.isEmpty* (*impl-of-RBT-HM hm*)

definition *hm-iteratei hm* == *HashMap-Impl.iteratei* (*impl-of-RBT-HM hm*)

lemma *impl-of-hm-empty* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-empty-const*) = *HashMap-Impl.empty* ()
⟨*proof*⟩

lemma *impl-of-hm-update* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-update k v hm*) = *HashMap-Impl.update k v* (*impl-of-RBT-HM hm*)
⟨*proof*⟩

lemma *impl-of-hm-delete* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-delete k hm*) = *HashMap-Impl.delete k* (*impl-of-RBT-HM hm*)
⟨*proof*⟩

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* ()) = *Map.empty*
hm-lookup k m = *hm-α m k*
hm-α (*hm-update k v m*) = (*hm-α m*)(*k*↦*v*)

hm-α (hm-delete k m) = (hm-α m) |‘ (−{k})
⟨*proof*⟩

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

lemma *hm-iteratei-impl*:
map-iterator (hm-iteratei m) (hm-α m)
⟨*proof*⟩

Integration in Isabelle Collections Framework

In this section, we integrate hashmaps into the Isabelle Collections Framework.

definition [*icf-rec-def*]: *hm-basic-ops* ≡ (‖
bmap-op-α = hm-α,
bmap-op-invar = λ-. True,
bmap-op-empty = hm-empty,
bmap-op-lookup = hm-lookup,
bmap-op-update = hm-update,
bmap-op-update-dj = hm-update, — TODO: Optimize bucket-ins here
bmap-op-delete = hm-delete,
bmap-op-list-it = hm-iteratei
‖)

⟨*ML*⟩
interpretation *hm-basic*: *StdBasicMap hm-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *hm-ops* ≡ *hm-basic.dflt-ops*

⟨*ML*⟩
interpretation *hm*: *StdMapDefs hm-ops* ⟨*proof*⟩
interpretation *hm*: *StdMap hm-ops*
⟨*proof*⟩
interpretation *hm*: *StdMap-no-invar hm-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-hm*[*proper-it*]:
shows *proper-it′ hm-iteratei hm-iteratei*
⟨*proof*⟩

interpretation *pi-hm*: *proper-it-loc hm-iteratei hm-iteratei*
⟨*proof*⟩

Code generator test

definition *test-codegen* **where** *test-codegen* ≡ (
hm.add ,
hm.add-dj ,
hm.ball ,
hm.bex ,
hm.delete ,
hm.empty ,
hm.isEmpty ,
hm.isSng ,
hm.iterate ,
hm.iteratei ,
hm.list-it ,
hm.lookup ,
hm.restrict ,
hm.sel ,
hm.size ,
hm.size-abort ,
hm.sng ,
hm.to-list ,
hm.to-map ,
hm.update ,
hm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.6 Implementation of a trie with explicit invariants

theory *Trie-Impl* **imports**
../../Lib/Assoc-List
Trie.Trie
begin

Interuptible iterator

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*:
$\neg\ c\ \sigma \Longrightarrow$ *iteratei-postfixed ks t c f* $\sigma = \sigma$
⟨*proof*⟩

lemma *iteratei-interrupt*:
$\neg\ c\ \sigma \Longrightarrow$ *iteratei t c f* $\sigma = \sigma$
⟨*proof*⟩

lemma *iteratei-postfixed-alt-def* :
iteratei-postfixed ks ((*Trie vo ts*)::('*key*, '*val*) *trie*) =
(*set-iterator-union*
(*case-option set-iterator-emp* (λ*v*. *set-iterator-sng* (*ks*, *v*)) *vo*)
(*set-iterator-image snd*
(*set-iterator-product* (*foldli ts*)
(λ(*k*, *t'*). *iteratei-postfixed* (*k* # *ks*) *t'*))
))
⟨*proof*⟩

lemma *iteratei-postfixed-correct* :
assumes *invar*: *invar-trie* (*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-trie t*)))
⟨*proof*⟩

definition *trie-reverse-key* **where**
trie-reverse-key ksv = (*rev* (*fst ksv*), (*snd ksv*))

lemma *trie-reverse-key-alt-def*[*code*] :
trie-reverse-key (*ks*, *v*) = (*rev ks*, *v*)
⟨*proof*⟩

lemma *trie-reverse-key-reverse*[*simp*] :
trie-reverse-key (*trie-reverse-key ksv*) = *ksv*
⟨*proof*⟩

lemma *trie-iteratei-correct*:
assumes *invar*: *invar-trie* (*t* :: ('*key*, '*val*) *trie*)
shows *set-iterator* ((*iteratei t*)::('*key list* × '*val*, '*σ*) *set-iterator*)
(*trie-reverse-key* ‘ (*map-to-set* (*lookup-trie t*)))
⟨*proof*⟩

hide-const (**open**) *iteratei*
hide-type (**open**) *trie*

end

3.3.7 Tries without invariants

theory *Trie2* **imports**
Trie-Impl
begin
⟨*proof*⟩

Abstract type definition

typedef ('*key*, '*val*) *trie* =
{*t* :: ('*key*, '*val*) *Trie.trie*. *invar-trie t*}
morphisms *impl-of Trie*
⟨*proof*⟩

lemma *invar-trie-impl-of* [*simp*, *intro*]: *invar-trie* (*impl-of t*)
⟨*proof*⟩

lemma *Trie-impl-of* [*code abstype*]: *Trie* (*impl-of t*) = *t*
⟨*proof*⟩

Primitive operations

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

definition *update* :: '*key list* ⇒ '*val* ⇒ ('*key*, '*val*) *trie* ⇒ ('*key*, '*val*) *trie*
where *update ks v t* = *Trie* (*update-trie ks v* (*impl-of t*))

definition *delete* :: '*key list* ⇒ ('*key*, '*val*) *trie* ⇒ ('*key*, '*val*) *trie*
where *delete ks t* = *Trie* (*delete-trie ks* (*impl-of t*))

definition *lookup* :: ('*key*, '*val*) *trie* ⇒ '*key list* ⇒ '*val option*
where *lookup t* = *lookup-trie* (*impl-of t*)

definition *isEmpty* :: ('*key*, '*val*) *trie* ⇒ *bool*
where *isEmpty t* = *is-empty-trie* (*impl-of t*)

definition *iteratei* :: ('*key*, '*val*) *trie* ⇒ ('*key list* × '*val*, '*σ*) *set-iterator*
where *iteratei t* = *set-iterator-image trie-reverse-key* (*Trie-Impl.iteratei* (*impl-of t*))

lemma *iteratei-code*[*code*] :
iteratei t c f = *Trie-Impl.iteratei* (*impl-of t*) *c* (λ(*ks*, *v*). *f* (*rev ks*, *v*))
⟨*proof*⟩

lemma *impl-of-empty* [*code abstract*]: *impl-of empty* = *empty-trie*
⟨*proof*⟩

lemma *impl-of-update* [*code abstract*]: *impl-of* (*update ks v t*) = *update-trie ks v*

(*impl-of t*)
⟨*proof*⟩

lemma *impl-of-delete* [*code abstract*]: *impl-of* (*delete ks t*) = *delete-trie ks* (*impl-of t*)
⟨*proof*⟩

Correctness of primitive operations

lemma *lookup-empty* [*simp*]: *lookup empty* = *Map.empty*
⟨*proof*⟩

lemma *lookup-update* [*simp*]: *lookup* (*update ks v t*) = (*lookup t*)(*ks* ↦ *v*)
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *lookup* (*delete ks t*) = (*lookup t*)(*ks* := *None*)
⟨*proof*⟩

lemma *isEmpty-lookup*: *isEmpty t* ⟷ *lookup t* = *Map.empty*
⟨*proof*⟩

lemma *finite-dom-lookup*: *finite* (*dom* (*lookup t*))
⟨*proof*⟩

lemma *iteratei-correct*:
map-iterator (*iteratei m*) (*lookup m*)
⟨*proof*⟩

Type classes

instantiation *trie* :: (*equal*, *equal*) *equal* **begin**

definition *equal-class.equal* (*t* :: ('*a*, '*b*) *trie*) *t'* = (*impl-of t* = *impl-of t'*)

instance
⟨*proof*⟩
end

hide-const (**open**) *empty lookup update delete iteratei isEmpty*

end

3.3.8 Map implementation via tries

theory *TrieMapImpl* **imports**
Trie2
../gen-algo/MapGA
begin

Operations

type-synonym $('k, 'v)\ tm = ('k, 'v)\ trie$

definition [*icf-rec-def*]: *tm-basic-ops* ≡ (|
  bmap-op-α = *Trie2.lookup*,
  bmap-op-invar = λ-. *True*,
  bmap-op-empty = (λ-::*unit*. *Trie2.empty*),
  bmap-op-lookup = (λ*k m*. *Trie2.lookup m k*),
  bmap-op-update = *Trie2.update*,
  bmap-op-update-dj = *Trie2.update*,
  bmap-op-delete = *Trie2.delete*,
  bmap-op-list-it = *Trie2.iteratei*
|)

⟨*ML*⟩
interpretation *tm-basic*: *StdBasicMap tm-basic-ops*
  ⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *tm-ops* ≡ *tm-basic.dflt-ops*

⟨*ML*⟩
interpretation *tm*: *StdMap tm-ops*
  ⟨*proof*⟩
interpretation *tm*: *StdMap-no-invar tm-ops*
  ⟨*proof*⟩
⟨*ML*⟩

lemma *pi-trie-impl*[*proper-it*]:
  shows *proper-it'*
    ((*Trie-Impl.iteratei*) :: - ⇒ (-,'σ *a*) *set-iterator*)
    ((*Trie-Impl.iteratei*) :: - ⇒ (-,'σ *b*) *set-iterator*)
  ⟨*proof*⟩

lemma *pi-trie*[*proper-it*]:
  proper-it' Trie2.iteratei Trie2.iteratei
  ⟨*proof*⟩

interpretation *pi-trie*: *proper-it-loc Trie2.iteratei Trie2.iteratei*
  ⟨*proof*⟩

Code generator test

definition *test-codegen* ≡ (
  tm.add ,
  tm.add-dj ,
  tm.ball ,
  tm.bex ,
  tm.delete ,

tm.empty ,
tm.isEmpty ,
tm.isSng ,
tm.iterate ,
tm.iteratei ,
tm.list-it ,
tm.lookup ,
tm.restrict ,
tm.sel ,
tm.size ,
tm.size-abort ,
tm.sng ,
tm.to-list ,
tm.to-map ,
tm.update ,
tm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.9 Array-based hash map implementation

theory *ArrayHashMap-Impl* **imports**
../../Lib/HashCode
../../Lib/Code-Target-ICF
../../Lib/Diff-Array
../gen-algo/ListGA
ListMapImpl
../../Iterator/Array-Iterator
begin

Misc.

⟨*ML*⟩
interpretation *a-idx-it*:
idx-iteratei-loc list-of-array λ-. *True array-length array-get*
⟨*proof*⟩
⟨*ML*⟩

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*

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-nat 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-nat* (*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-nat 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-nat* (*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-nat* (*array-length a*) *k*;
m = *array-get a h*;
deleted = (*map-of m k* ≠ *None*)
in HashMap (*array-set a h* (*AList.delete k m*)) (*if deleted then n* − *1 else n*))

lemma *hm-grow-gt-1* [*iff*]:
Suc 0 < *hm-grow hm*
⟨*proof*⟩

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok len h* [] = *True*
⟨*proof*⟩

lemma *bucket-okD*:
⟦ *bucket-ok len h xs*; (*k*, *v*) ∈ *set xs* ⟧
⟹ *bounded-hashcode-nat len k* = *h*
⟨*proof*⟩

lemma *bucket-okI*:
(⋀*k*. *k* ∈ *fst* ‘ *set kvs* ⟹ *bounded-hashcode-nat len k* = *h*) ⟹ *bucket-ok len h kvs*
⟨*proof*⟩

ahm-invar

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux n a*
obtains ∀ *h*. *h* < *array-length a* ⟶ *bucket-ok* (*array-length a*) *h* (*array-get a h*) ∧ *distinct* (*map fst* (*array-get a h*))
and *n* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and** *array-length a* > *1*
⟨*proof*⟩

lemma *ahm-invar-auxI*:
⟦ ⋀*h*. *h* < *array-length a* ⟹ *bucket-ok* (*array-length a*) *h* (*array-get a h*);
⋀*h*. *h* < *array-length a* ⟹ *distinct* (*map fst* (*array-get a h*));
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*; *array-length a* > *1* ⟧
⟹ *ahm-invar-aux n a*
⟨*proof*⟩

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
⟨*proof*⟩

ahm-α

lemma *finite-dom-ahm-α-aux*:
assumes *ahm-invar-aux n a*
shows *finite* (*dom* (*ahm-α-aux a*))
⟨*proof*⟩

lemma *ahm-α-aux-conv-map-of-concat*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *ahm-α-aux* (*Array xs*) = *map-of* (*concat xs*)
⟨*proof*⟩

lemma *ahm-invar-aux-card-dom-ahm-α-auxD*:
assumes *inv*: *ahm-invar-aux n a*
shows *card* (*dom* (*ahm-α-aux a*)) = *n*
⟨*proof*⟩

lemma *finite-dom-ahm-α*:
ahm-invar hm ⟹ *finite* (*dom* (*ahm-α hm*))

⟨*proof*⟩

lemma *finite-map-ahm-α-aux*:
finite-map ahm-α-aux (*ahm-invar-aux n*)
⟨*proof*⟩

lemma *finite-map-ahm-α*:
finite-map ahm-α ahm-invar
⟨*proof*⟩

ahm-empty

lemma *ahm-invar-aux-new-array*:
assumes *n* > *1*
shows *ahm-invar-aux 0* (*new-array* [] *n*)
⟨*proof*⟩

lemma *ahm-invar-new-hashmap-with*:
n > *1* ⟹ *ahm-invar* (*new-hashmap-with n*)
⟨*proof*⟩

lemma *ahm-α-new-hashmap-with*:
n > *1* ⟹ *ahm-α* (*new-hashmap-with n*) = *Map.empty*
⟨*proof*⟩

lemma *ahm-invar-ahm-empty* [*simp*]: *ahm-invar* (*ahm-empty* ())
⟨*proof*⟩

lemma *ahm-empty-correct* [*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
⟨*proof*⟩

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

ahm-lookup

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

ahm-iteratei

lemma *ahm-iteratei-aux-impl*:
assumes *invar-m*: *ahm-invar-aux n m*
shows *map-iterator* (*ahm-iteratei-aux m*) (*ahm-α-aux m*)
⟨*proof*⟩

lemma *ahm-iteratei-correct*:
assumes *invar-hm*: *ahm-invar hm*
shows *map-iterator* (*ahm-iteratei hm*) (*ahm-α hm*)
⟨*proof*⟩

lemma *ahm-iteratei-aux-code* [*code*]:
ahm-iteratei-aux a c f σ = *a-idx-it.idx-iteratei a c* (λ*x*. *foldli x c f*) *σ*
⟨*proof*⟩

ahm-rehash

lemma *array-length-ahm-rehash-aux′*:
array-length (*ahm-rehash-aux′ n kv a*) = *array-length a*
⟨*proof*⟩

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
assumes *inv*: *ahm-invar-aux n a*
and *fresh*: *k* ∉ *fst* ‘ *set* (*array-get a* (*bounded-hashcode-nat* (*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*⟩

declare [[*coercion-enabled* = *false*]]

lemma *ahm-rehash-aux-correct*:
fixes *a* :: ((′*key* :: *hashable*) × ′*val*) *list array*
assumes *inv*: *ahm-invar-aux n a*
and *sz* > *1*
shows *ahm-invar-aux n* (*ahm-rehash-aux a sz*) (**is** *?thesis1*)
and *ahm-α-aux* (*ahm-rehash-aux a sz*) = *ahm-α-aux a* (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-rehash-correct*:
fixes *hm* :: (′*key* :: *hashable*, ′*val*) *hashmap*
assumes *inv*: *ahm-invar hm*
and *sz* > *1*
shows *ahm-invar* (*ahm-rehash hm sz*) *ahm-α* (*ahm-rehash hm sz*) = *ahm-α hm*
⟨*proof*⟩

ahm-update

lemma *ahm-update-aux-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-update-aux hm k v*) (**is** *?thesis1*)
and *ahm-α* (*ahm-update-aux hm k v*) = (*ahm-α hm*)(*k* ↦ *v*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *ahm-update-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-update k v hm*)
and *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)(*k* ↦ *v*)
⟨*proof*⟩

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

ahm-delete

lemma *ahm-delete-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-delete k hm*) (**is** *?thesis1*)
and *ahm-α* (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*}) (**is** *?thesis2*)
⟨*proof*⟩

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

hide-const (**open**) *HashMap ahm-empty bucket-ok ahm-invar ahm-α ahm-lookup ahm-iteratei ahm-rehash hm-grow ahm-filled ahm-update ahm-delete*
hide-type (**open**) *hashmap*

end

3.3.10 Array-based hash maps without explicit invariants

theory *ArrayHashMap*
imports *ArrayHashMap-Impl*
begin

Abstract type definition

typedef (**overloaded**) (*'key* :: *hashable*, *'val*) *hashmap* =
{*hm* :: (*'key*, *'val*) *ArrayHashMap-Impl.hashmap*. *ArrayHashMap-Impl.ahm-invar hm*}
morphisms *impl-of HashMap*
⟨*proof*⟩

type-synonym ('*k*,'*v*) *ahm* = ('*k*,'*v*) *hashmap*

lemma *ahm-invar-impl-of* [*simp*, *intro*]: *ArrayHashMap-Impl.ahm-invar* (*impl-of hm*)
⟨*proof*⟩

lemma *HashMap-impl-of* [*code abstype*]: *HashMap* (*impl-of t*) = *t*
⟨*proof*⟩

Primitive operations

definition *ahm-empty-const* :: (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty-const* ≡ (*HashMap* (*ArrayHashMap-Impl.ahm-empty* ()))

definition *ahm-empty* :: *unit* ⇒ (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty* ≡ λ-. *ahm-empty-const*

definition *ahm-α* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'key* ⇒ *'val option*
where *ahm-α hm* = *ArrayHashMap-Impl.ahm-α* (*impl-of hm*)

definition *ahm-lookup* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'val option*
where *ahm-lookup k hm* = *ArrayHashMap-Impl.ahm-lookup k* (*impl-of hm*)

definition *ahm-iteratei* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key* × *'val*, *'σ*) *set-iterator*
where *ahm-iteratei hm* = *ArrayHashMap-Impl.ahm-iteratei* (*impl-of hm*)

definition *ahm-update* :: *'key* ⇒ *'val* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-update k v hm = *HashMap* (*ArrayHashMap-Impl.ahm-update k v* (*impl-of hm*))

definition *ahm-delete* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-delete k hm = *HashMap* (*ArrayHashMap-Impl.ahm-delete k* (*impl-of hm*))

lemma *impl-of-ahm-empty* [*code abstract*]:
impl-of ahm-empty-const = *ArrayHashMap-Impl.ahm-empty* ()
⟨*proof*⟩

lemma *impl-of-ahm-update* [*code abstract*]:
impl-of (*ahm-update k v hm*) = *ArrayHashMap-Impl.ahm-update k v* (*impl-of hm*)
⟨*proof*⟩

lemma *impl-of-ahm-delete* [*code abstract*]:
impl-of (*ahm-delete k hm*) = *ArrayHashMap-Impl.ahm-delete k* (*impl-of hm*)
⟨*proof*⟩

lemma *finite-dom-ahm-α*[*simp*]: *finite* (*dom* (*ahm-α hm*))
⟨*proof*⟩

lemma *ahm-empty-correct*[*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
⟨*proof*⟩

lemma *ahm-lookup-correct*[*simp*]: *ahm-lookup k m* = *ahm-α m k*
⟨*proof*⟩

lemma *ahm-update-correct*[*simp*]: *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *ahm-delete-correct*[*simp*]:
ahm-α (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*})
⟨*proof*⟩

lemma *ahm-iteratei-impl*[*simp*]: *map-iterator* (*ahm-iteratei m*) (*ahm-α m*)
⟨*proof*⟩

ICF Integration

definition [*icf-rec-def*]: *ahm-basic-ops* ≡ (|
bmap-op-α = *ahm-α*,
bmap-op-invar = λ-. *True*,
bmap-op-empty = *ahm-empty*,
bmap-op-lookup = *ahm-lookup*,
bmap-op-update = *ahm-update*,
bmap-op-update-dj = *ahm-update*, — TODO: We could use a more efficient bucket update here
bmap-op-delete = *ahm-delete*,
bmap-op-list-it = *ahm-iteratei*
|)

⟨*ML*⟩
interpretation *ahm-basic*: *StdBasicMap ahm-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *ahm-ops* ≡ *ahm-basic.dflt-ops*

⟨*ML*⟩
interpretation *ahm*: *StdMap ahm-ops*
⟨*proof*⟩
interpretation *ahm*: *StdMap-no-invar ahm-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-ahm*[*proper-it*]:
proper-it′ ahm-iteratei ahm-iteratei
⟨*proof*⟩

interpretation *pi-ahm*: *proper-it-loc ahm-iteratei ahm-iteratei*
⟨*proof*⟩

Code generator test

definition *test-codegen* **where** *test-codegen* ≡ (
ahm.add ,
ahm.add-dj ,
ahm.ball ,
ahm.bex ,
ahm.delete ,

ahm.empty ,
ahm.isEmpty ,
ahm.isSng ,
ahm.iterate ,
ahm.iteratei ,
ahm.list-it ,
ahm.lookup ,
ahm.restrict ,
ahm.sel ,
ahm.size ,
ahm.size-abort ,
ahm.sng ,
ahm.to-list ,
ahm.to-map ,
ahm.update ,
ahm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.11 Maps from Naturals by Arrays

theory *ArrayMapImpl*
imports
../spec/MapSpec
../gen-algo/MapGA
../../Lib/Diff-Array
begin **type-synonym** *'v iam = 'v option array*

Definitions

definition *iam-α :: 'v iam ⇒ nat ⇀ 'v* **where**
iam-α a i ≡ if i < array-length a then array-get a i else None

lemma [*code*]: *iam-α a i ≡ array-get-oo None a i*
⟨*proof*⟩

abbreviation (*input*) *iam-invar :: 'v iam ⇒ bool*
where *iam-invar ≡ λ-. True*

definition *iam-empty :: unit ⇒ 'v iam*
where *iam-empty ≡ λ-::unit. array-of-list []*

definition *iam-lookup :: nat ⇒ 'v iam ⇀ 'v*
where [*code-unfold*]: *iam-lookup k a ≡ iam-α a k*

definition *iam-increment* (*l*::*nat*) *idx ≡*
max (*idx + 1 − l*) (*2 ∗ l + 3*)

lemma *incr-correct*: $\neg\ idx < l \Longrightarrow idx < l + iam\text{-}increment\ l\ idx$
⟨*proof*⟩

definition *iam-update* :: $nat \Rightarrow 'v \Rightarrow 'v\ iam \Rightarrow 'v\ iam$
where *iam-update k v a* ≡ *let*
l = *array-length a*;
a = *if k* < *l then a else array-grow a* (*iam-increment l k*) *None*
in
array-set a k (*Some v*)

lemma [*code*]: *iam-update k v a* ≡ *array-set-oo*
(λ-. *array-set*
(*array-grow a* (*iam-increment* (*array-length a*) *k*) *None*) *k* (*Some v*))
a k (*Some v*)

⟨*proof*⟩

definition *iam-update-dj* ≡ *iam-update*

definition *iam-delete* :: $nat \Rightarrow 'v\ iam \Rightarrow 'v\ iam$
where *iam-delete k a* ≡
if k<*array-length a then array-set a k None else a*

lemma [*code*]: *iam-delete k a* ≡ *array-set-oo* (λ-. *a*) *a k None*
⟨*proof*⟩

fun *iam-rev-iterateoi-aux*
:: $nat \Rightarrow ('v\ iam) \Rightarrow ('\sigma \Rightarrow bool) \Rightarrow (nat \times 'v \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow '\sigma \Rightarrow '\sigma$
where
iam-rev-iterateoi-aux 0 a c f σ = *σ*
| *iam-rev-iterateoi-aux i a c f σ* = (
if c σ then
iam-rev-iterateoi-aux (*i* − *1*) *a c f* (
case array-get a (*i* − *1*) *of None* ⇒ *σ* | *Some x* ⇒ *f* (*i* − *1*, *x*) *σ*
)
else σ)

definition *iam-rev-iterateoi* :: $'v\ iam \Rightarrow (nat \times 'v, '\sigma)\ set\text{-}iterator$ **where**
iam-rev-iterateoi a ≡ *iam-rev-iterateoi-aux* (*array-length a*) *a*

function *iam-iterateoi-aux*
:: $nat \Rightarrow nat \Rightarrow ('v\ iam) \Rightarrow ('\sigma \Rightarrow bool) \Rightarrow (nat \times 'v \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow '\sigma \Rightarrow '\sigma$
where
iam-iterateoi-aux i len a c f σ =
(*if i* ≥ *len* ∨ ¬ *c σ then σ else let*
σ′ = (*case array-get a i of*
None ⇒ *σ*
| *Some x* ⇒ *f* (*i*,*x*) *σ*)

in iam-iterateoi-aux (i + 1) len a c f σ′)
⟨*proof*⟩
termination
⟨*proof*⟩

declare *iam-iterateoi-aux.simps*[*simp del*]

lemma *iam-iterateoi-aux-csimps*:
i ≥ len ⟹ iam-iterateoi-aux i len a c f σ = σ
¬ c σ ⟹ iam-iterateoi-aux i len a c f σ = σ
⟦ i< len; c σ ⟧ ⟹ iam-iterateoi-aux i len a c f σ =
(case array-get a i of
None ⇒ iam-iterateoi-aux (i + 1) len a c f σ
| Some x ⇒ iam-iterateoi-aux (i + 1) len a c f (f (i,x) σ))
⟨*proof*⟩

definition *iam-iterateoi* :: *′v iam ⇒ (nat × ′v,′σ) set-iterator* **where**
iam-iterateoi a = iam-iterateoi-aux 0 (array-length a) a

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

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

lemma *array-get-set-iff*: *i<array-length a ⟹*
array-get (array-set a i x) j = (if i=j then x else array-get a j)
⟨*proof*⟩

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

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

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

lemma *iam-rev-iterateoi-aux-foldli-conv* :
iam-rev-iterateoi-aux n a =
foldli (List.map-filter (λn. map-option (λv. (n, v)) (array-get a n)) (rev [0..<n]))
⟨*proof*⟩

lemma *iam-rev-iterateoi-foldli-conv* :
iam-rev-iterateoi a =
foldli (List.map-filter
(λn. map-option (λv. (n, v)) (array-get a n))
(rev [0..<(array-length a)]))

⟨*proof*⟩

lemma *iam-rev-iterateoi-correct* :
fixes *m*::'*a option array*
defines *kvs* ≡ *List.map-filter*
(λ*n*. *map-option* (λ*v*. (*n*, *v*)) (*array-get m n*)) (*rev* [*0*..<(*array-length m*)])
shows *map-iterator-rev-linord* (*iam-rev-iterateoi m*) (*iam-α m*)
⟨*proof*⟩

lemma *iam-rev-iterateoi-impl*:
poly-map-rev-iterateoi iam-α iam-invar iam-rev-iterateoi
⟨*proof*⟩

lemma *iam-iteratei-impl*:
poly-map-iteratei iam-α iam-invar iam-rev-iterateoi
⟨*proof*⟩

lemma *iam-iterateoi-aux-foldli-conv* :
iam-iterateoi-aux n (*array-length a*) *a c f σ* =
foldli (*List.map-filter* (λ*n*. *map-option* (λ*v*. (*n*, *v*)) (*array-get a n*))
([*n*..<*array-length a*])) *c f σ*
thm *iam-iterateoi-aux.induct*
⟨*proof*⟩

lemma *iam-iterateoi-foldli-conv* :
iam-iterateoi a =
foldli (*List.map-filter*
(λ*n*. *map-option* (λ*v*. (*n*, *v*)) (*array-get a n*))
([*0*..<(*array-length a*)]))
⟨*proof*⟩

lemmas [*simp*] = *map-filter-simps*
lemma *map-filter-append*[*simp*]: *List.map-filter f* (*la*@*lb*)
= *List.map-filter f la* @ *List.map-filter f lb*
⟨*proof*⟩

lemma *iam-iterateoi-correct*:
fixes *m*::'*a option array*
defines *kvs* ≡ *List.map-filter*
(λ*n*. *map-option* (λ*v*. (*n*, *v*)) (*array-get m n*)) ([*0*..<(*array-length m*)])
shows *map-iterator-linord* (*iam-iterateoi m*) (*iam-α m*)
⟨*proof*⟩

lemma *iam-iterateoi-impl*:
poly-map-iterateoi iam-α iam-invar iam-iterateoi
⟨*proof*⟩

definition *iam-basic-ops* :: (*nat*,'*a*,'*a iam*) *omap-basic-ops*

where [*icf-rec-def*]: *iam-basic-ops* ≡ (|
bmap-op-α = *iam-α*,
bmap-op-invar = λ-. *True*,
bmap-op-empty = *iam-empty*,
bmap-op-lookup = *iam-lookup*,
bmap-op-update = *iam-update*,
bmap-op-update-dj = *iam-update-dj*,
bmap-op-delete = *iam-delete*,
bmap-op-list-it = *iam-rev-iterateoi*,
bmap-op-ordered-list-it = *iam-iterateoi*,
bmap-op-rev-list-it = *iam-rev-iterateoi*
|)

⟨*ML*⟩
interpretation *iam-basic*: *StdBasicOMap iam-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *iam-ops* ≡ *iam-basic.dflt-oops*

⟨*ML*⟩
interpretation *iam*: *StdOMap iam-ops*
⟨*proof*⟩
interpretation *iam*: *StdMap-no-invar iam-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-iam*[*proper-it*]:
proper-it′ iam-iterateoi iam-iterateoi
⟨*proof*⟩

lemma *pi-iam-rev*[*proper-it*]:
proper-it′ iam-rev-iterateoi iam-rev-iterateoi
⟨*proof*⟩

interpretation *pi-iam*: *proper-it-loc iam-iterateoi iam-iterateoi*
⟨*proof*⟩

interpretation *pi-iam-rev*: *proper-it-loc iam-rev-iterateoi iam-rev-iterateoi*
⟨*proof*⟩

Code generator test

definition *test-codegen* ≡ (
iam.add ,
iam.add-dj ,
iam.ball ,
iam.bex ,
iam.delete ,
iam.empty ,

iam.isEmpty ,
iam.isSng ,
iam.iterate ,
iam.iteratei ,
iam.iterateo ,
iam.iterateoi ,
iam.list-it ,
iam.lookup ,
iam.max ,
iam.min ,
iam.restrict ,
iam.rev-iterateo ,
iam.rev-iterateoi ,
iam.rev-list-it ,
iam.reverse-iterateo ,
iam.reverse-iterateoi ,
iam.sel ,
iam.size ,
iam.size-abort ,
iam.sng ,
iam.to-list ,
iam.to-map ,
iam.to-rev-list ,
iam.to-sorted-list ,
iam.update ,
iam.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.12 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

3.3.13 Set Implementation by List

theory *ListSetImpl*
imports *../spec/SetSpec* *../gen-algo/SetGA* *../../Lib/Dlist-add*
begin type-synonym
'a ls = 'a dlist

Definitions

definition *ls-α* :: *'a ls* ⇒ *'a set* **where** *ls-α l* == *set* (*list-of-dlist l*)

definition *ls-basic-ops* :: (*'a*,*'a ls*) *set-basic-ops* **where**
[*icf-rec-def*]: *ls-basic-ops* ≡ (|
bset-op-α = *ls-α*,
bset-op-invar = λ-. *True*,
bset-op-empty = λ-. *Dlist.empty*,
bset-op-memb = (λ*x s*. *Dlist.member s x*),
bset-op-ins = *Dlist.insert*,
bset-op-ins-dj = *Dlist.insert*,
bset-op-delete = *dlist-remove'*,
bset-op-list-it = *dlist-iteratei*
|)

⟨*ML*⟩
interpretation *ls-basic*: *StdBasicSet ls-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *ls-ops* ≡ *ls-basic.dflt-ops*(|
set-op-to-list := *list-of-dlist*
|)

⟨*ML*⟩
interpretation *ls*: *StdSetDefs ls-ops* ⟨*proof*⟩
interpretation *ls*: *StdSet ls-ops*
⟨*proof*⟩

interpretation *ls*: *StdSet-no-invar ls-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-ls*[*proper-it*]:
proper-it' dlist-iteratei dlist-iteratei
⟨*proof*⟩

lemma *pi-ls'*[*proper-it*]:
proper-it' ls.iteratei ls.iteratei
⟨*proof*⟩

interpretation *pi-ls*: *proper-it-loc dlist-iteratei dlist-iteratei*

⟨*proof*⟩

interpretation *pi-ls'*: *proper-it-loc ls.iteratei ls.iteratei*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
ls.empty,
ls.memb,
ls.ins,
ls.delete,
ls.list-it,
ls.sng,
ls.isEmpty,
ls.isSng,
ls.ball,
ls.bex,
ls.size,
ls.size-abort,
ls.union,
ls.union-dj,
ls.diff,
ls.filter,
ls.inter,
ls.subset,
ls.equal,
ls.disjoint,
ls.disjoint-witness,
ls.sel,
ls.to-list,
ls.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.14 Set Implementation by List with explicit invariants

theory *ListSetImpl-Invar*
imports
../spec/SetSpec
../gen-algo/SetGA
../../Lib/Dlist-add
begin type-synonym
'a lsi = *'a list*

Definitions

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-basic-ops* :: ('*a*,'*a lsi*) *set-basic-ops* **where**
[*icf-rec-def*]: *lsi-basic-ops* ≡ (|
bset-op-α = *set*,
bset-op-invar = *distinct*,
bset-op-empty = λ-. [],
bset-op-memb = (λ*x s*. *List.member s x*),
bset-op-ins = *lsi-ins*,
bset-op-ins-dj = (#),
bset-op-delete = λ*x l*. *Dlist-add.dlist-remove1'* *x* [] *l*,
bset-op-list-it = *foldli*
|)

⟨*ML*⟩
interpretation *lsi-basic*: *StdBasicSet lsi-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *lsi-ops* ≡ *lsi-basic.dflt-ops* (|
set-op-union-dj := (@),
set-op-to-list := *id*
|)

⟨*ML*⟩
interpretation *lsi*: *StdSet lsi-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-lsi*[*proper-it*]:
proper-it' *foldli foldli*
⟨*proof*⟩

interpretation *pi-lsi*: *proper-it-loc foldli foldli*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
lsi.empty,
lsi.memb,
lsi.ins,
lsi.delete,
lsi.list-it,
lsi.sng,
lsi.isEmpty,
lsi.isSng,
lsi.ball,
lsi.bex,
lsi.size,
lsi.size-abort,
lsi.union,

lsi.union-dj,
lsi.diff,
lsi.filter,
lsi.inter,
lsi.subset,
lsi.equal,
lsi.disjoint,
lsi.disjoint-witness,
lsi.sel,
lsi.to-list,
lsi.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.15 Set Implementation by non-distinct Lists

theory *ListSetImpl-NotDist*
imports
../spec/SetSpec
../gen-algo/SetGA

begin type-synonym
'a lsnd = 'a list

Definitions

definition *lsnd-α* :: *'a lsnd ⇒ 'a set* **where** *lsnd-α == set*
abbreviation (*input*) *lsnd-invar*
:: *'a lsnd ⇒ bool* **where** *lsnd-invar == (λ-. True)*
definition *lsnd-empty* :: *unit ⇒ 'a lsnd* **where** *lsnd-empty == (λ-::unit. [])*
definition *lsnd-memb* :: *'a ⇒ 'a lsnd ⇒ bool* **where** *lsnd-memb x l == List.member l x*
definition *lsnd-ins* :: *'a ⇒ 'a lsnd ⇒ 'a lsnd* **where** *lsnd-ins x l == x#l*
definition *lsnd-ins-dj* :: *'a ⇒ 'a lsnd ⇒ 'a lsnd* **where** *lsnd-ins-dj x l == x#l*

definition *lsnd-delete* :: *'a ⇒ 'a lsnd ⇒ 'a lsnd* **where** *lsnd-delete x l == remove-rev x l*

definition *lsnd-iteratei* :: *'a lsnd ⇒ ('a,'σ) set-iterator*
where *lsnd-iteratei l = foldli (remdups l)*

definition *lsnd-isEmpty* :: *'a lsnd ⇒ bool* **where** *lsnd-isEmpty s == s=[]*

definition *lsnd-union* :: *'a lsnd ⇒ 'a lsnd ⇒ 'a lsnd*
where *lsnd-union s1 s2 == revg s1 s2*
definition *lsnd-union-dj* :: *'a lsnd ⇒ 'a lsnd ⇒ 'a lsnd*
where *lsnd-union-dj s1 s2 == revg s1 s2* — Union of disjoint sets

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*

Correctness

lemmas *lsnd-defs* =
lsnd-α-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-union-dj-def
lsnd-to-list-def
list-to-lsnd-def

lemma *lsnd-empty-impl*: *set-empty lsnd-α lsnd-invar lsnd-empty*
⟨*proof*⟩

lemma *lsnd-memb-impl*: *set-memb lsnd-α lsnd-invar lsnd-memb*
⟨*proof*⟩

lemma *lsnd-ins-impl*: *set-ins lsnd-α lsnd-invar lsnd-ins*
⟨*proof*⟩

lemma *lsnd-ins-dj-impl*: *set-ins-dj lsnd-α lsnd-invar lsnd-ins-dj*
⟨*proof*⟩

lemma *lsnd-delete-impl*: *set-delete lsnd-α lsnd-invar lsnd-delete*
⟨*proof*⟩

lemma *lsnd-α-finite*[*simp*, *intro*!]: *finite* (*lsnd-α l*)
⟨*proof*⟩

lemma *lsnd-is-finite-set*: *finite-set lsnd-α lsnd-invar*
⟨*proof*⟩

lemma *lsnd-iteratei-impl*: *poly-set-iteratei lsnd-α lsnd-invar lsnd-iteratei*
⟨*proof*⟩

lemma *lsnd-isEmpty-impl*: *set-isEmpty lsnd-α lsnd-invar lsnd-isEmpty*
⟨*proof*⟩

lemma *lsnd-union-impl*: *set-union lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar*
lsnd-union

⟨*proof*⟩

lemma *lsnd-union-dj-impl*: *set-union-dj lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union-dj*
⟨*proof*⟩

lemma *lsnd-to-list-impl*: *set-to-list lsnd-α lsnd-invar lsnd-to-list*
⟨*proof*⟩

lemma *list-to-lsnd-impl*: *list-to-set lsnd-α lsnd-invar list-to-lsnd*
⟨*proof*⟩

definition *lsnd-basic-ops* :: (*'x*,*'x lsnd*) *set-basic-ops*
  where [*icf-rec-def*]: *lsnd-basic-ops* ≡ ⦇
    bset-op-α = *lsnd-α*,
    bset-op-invar = *lsnd-invar*,
    bset-op-empty = *lsnd-empty*,
    bset-op-memb = *lsnd-memb*,
    bset-op-ins = *lsnd-ins*,
    bset-op-ins-dj = *lsnd-ins-dj*,
    bset-op-delete = *lsnd-delete*,
    bset-op-list-it = *lsnd-iteratei*
  ⦈

⟨*ML*⟩
interpretation *lsnd-basic*: *StdBasicSet lsnd-basic-ops*
  ⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *lsnd-ops* ≡ *lsnd-basic.dflt-ops* ⦇
  set-op-isEmpty := *lsnd-isEmpty*,
  set-op-union := *lsnd-union*,
  set-op-union-dj := *lsnd-union-dj*,
  set-op-to-list := *lsnd-to-list*,
  set-op-from-list := *list-to-lsnd*
  ⦈

⟨*ML*⟩
interpretation *lsnd*: *StdSetDefs lsnd-ops* ⟨*proof*⟩
interpretation *lsnd*: *StdSet lsnd-ops*
⟨*proof*⟩
interpretation *lsnd*: *StdSet-no-invar lsnd-ops*
  ⟨*proof*⟩
⟨*ML*⟩

lemma *pi-lsnd*[*proper-it*]:
  proper-it' *lsnd-iteratei lsnd-iteratei*
  ⟨*proof*⟩

interpretation *pi-lsnd*: *proper-it-loc lsnd-iteratei lsnd-iteratei*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
lsnd.empty,
lsnd.memb,
lsnd.ins,
lsnd.delete,
lsnd.list-it,
lsnd.sng,
lsnd.isEmpty,
lsnd.isSng,
lsnd.ball,
lsnd.bex,
lsnd.size,
lsnd.size-abort,
lsnd.union,
lsnd.union-dj,
lsnd.diff,
lsnd.filter,
lsnd.inter,
lsnd.subset,
lsnd.equal,
lsnd.disjoint,
lsnd.disjoint-witness,
lsnd.sel,
lsnd.to-list,
lsnd.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.16 Set Implementation by sorted Lists

theory *ListSetImpl-Sorted*
imports
../spec/SetSpec
../gen-algo/SetGA
../../Lib/Sorted-List-Operations
begin type-synonym
'a lss = *'a list*

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* == *Sorted-List-Operations.memb-sorted l x*
definition *lss-ins* :: *'a*::{*linorder*} ⇒ *'a lss* ⇒ *'a lss* **where** *lss-ins x l* == *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* == *Misc.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-remdups (*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-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-remdups*

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-to-list-def
list-to-lss-def

lemma *lss-empty-impl*: *set-empty lss-α lss-invar lss-empty*
⟨*proof*⟩

lemma *lss-memb-impl*: *set-memb lss-α lss-invar lss-memb*
⟨*proof*⟩

lemma *lss-ins-impl*: *set-ins lss-α lss-invar lss-ins*
⟨*proof*⟩

lemma *lss-ins-dj-impl*: *set-ins-dj lss-α lss-invar lss-ins-dj*
⟨*proof*⟩

lemma *lss-delete-impl*: *set-delete lss-α lss-invar lss-delete*
⟨*proof*⟩

lemma *lss-α-finite*[*simp*, *intro*!]: *finite* (*lss-α l*)
⟨*proof*⟩

lemma *lss-is-finite-set*: *finite-set lss-α lss-invar*
⟨*proof*⟩

lemma *lss-iterateoi-impl*: *poly-set-iterateoi lss-α lss-invar lss-iterateoi*
⟨*proof*⟩

lemma *lss-reverse-iterateoi-impl*: *poly-set-rev-iterateoi lss-α lss-invar lss-reverse-iterateoi*
⟨*proof*⟩

lemma *lss-iteratei-impl*: *poly-set-iteratei lss-α lss-invar lss-iteratei*
⟨*proof*⟩

lemma *lss-isEmpty-impl*: *set-isEmpty lss-α lss-invar lss-isEmpty*
⟨*proof*⟩

lemma *lss-inter-impl*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
⟨*proof*⟩

lemma *lss-union-impl*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*
⟨*proof*⟩

lemma *lss-union-list-impl*: *set-union-list lss-α lss-invar lss-α lss-invar lss-union-list*
⟨*proof*⟩

lemma *lss-union-dj-impl*: *set-union-dj lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union-dj*
⟨*proof*⟩

lemma *lss-image-filter-impl* : *set-image-filter lss-α lss-invar lss-α lss-invar lss-image-filter*
⟨*proof*⟩

lemma *lss-inj-image-filter-impl* : *set-inj-image-filter lss-α lss-invar lss-α lss-invar lss-inj-image-filter*
⟨*proof*⟩

lemma *lss-filter-impl* : *set-filter lss-α lss-invar lss-α lss-invar lss-filter*
⟨*proof*⟩

lemmas *lss-image-impl* = *iflt-image-correct*[*OF lss-image-filter-impl*, *folded lss-image-def*]
lemmas *lss-inj-image-impl* = *iflt-inj-image-correct*[*OF lss-inj-image-filter-impl*, *folded lss-inj-image-def*]

lemma *lss-to-list-impl*: *set-to-list lss-α lss-invar lss-to-list*
⟨*proof*⟩

lemma *list-to-lss-impl*: *list-to-set lss-α lss-invar list-to-lss*
⟨*proof*⟩

definition *lss-basic-ops* :: (*'x*::*linorder*,*'x lss*) *oset-basic-ops*
where [*icf-rec-def*]: *lss-basic-ops* ≡ (‖
bset-op-α = *lss-α*,
bset-op-invar = *lss-invar*,
bset-op-empty = *lss-empty*,
bset-op-memb = *lss-memb*,
bset-op-ins = *lss-ins*,
bset-op-ins-dj = *lss-ins-dj*,
bset-op-delete = *lss-delete*,
bset-op-list-it = *lss-iteratei*,
bset-op-ordered-list-it = *lss-iterateoi*,

bset-op-rev-list-it = *lss-reverse-iterateoi*
⦈

⟨*ML*⟩
interpretation *lss-basic*: *StdBasicOSet lss-basic-ops*
⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *lss-ops* ≡ *lss-basic.dflt-oops* ⦇
set-op-isEmpty := *lss-isEmpty*,
set-op-union := *lss-union*,
set-op-union-dj := *lss-union-dj*,
set-op-filter := *lss-filter*,
set-op-to-list := *lss-to-list*,
set-op-from-list := *list-to-lss*
⦈

⟨*ML*⟩
interpretation *lss*: *StdOSetDefs lss-ops* ⟨*proof*⟩
interpretation *lss*: *StdOSet lss-ops*
⟨*proof*⟩
⟨*ML*⟩

lemma *pi-lss*[*proper-it*]:
proper-it′ lss-iteratei lss-iteratei
⟨*proof*⟩

interpretation *pi-lss*: *proper-it-loc lss-iteratei lss-iteratei*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
lss.empty,
lss.memb,
lss.ins,
lss.delete,
lss.list-it,
lss.sng,
lss.isEmpty,
lss.isSng,
lss.ball,
lss.bex,
lss.size,
lss.size-abort,
lss.union,
lss.union-dj,
lss.diff,
lss.filter,
lss.inter,
lss.subset,

lss.equal,
lss.disjoint,
lss.disjoint-witness,
lss.sel,
lss.to-list,
lss.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.17 Set Implementation by Red-Black-Tree

theory *RBTSetImpl*
imports
../spec/SetSpec
RBTMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
'a rs = (*'a*::*linorder*,*unit*) *rm*

⟨*ML*⟩
interpretation *rs-sbm*: *OSetByOMap rm-basic-ops* ⟨*proof*⟩
⟨*ML*⟩

definition *rs-ops* :: (*'x*::*linorder*,*'x rs*) *oset-ops*
where [*icf-rec-def*]: *rs-ops* ≡ *rs-sbm.obasic.dflt-oops*

⟨*ML*⟩
interpretation *rs*: *StdOSetDefs rs-ops* ⟨*proof*⟩
interpretation *rs*: *StdOSet rs-ops*
⟨*proof*⟩

interpretation *rs*: *StdSet-no-invar rs-ops*
⟨*proof*⟩
⟨*ML*⟩

lemmas *rbt-it-to-it-map-code-unfold*[*code-unfold*] =
it-to-it-map-fold'[*OF pi-rm*]
it-to-it-map-fold'[*OF pi-rm-rev*]

lemma *pi-rs*[*proper-it*]:
proper-it' rs.iteratei rs.iteratei
proper-it' rs.iterateoi rs.iterateoi

proper-it′ rs.rev-iterateoi rs.rev-iterateoi
⟨*proof*⟩

interpretation
pi-rs: *proper-it-loc rs.iteratei rs.iteratei* +
pi-rs-o: *proper-it-loc rs.iterateoi rs.iterateoi* +
pi-rs-ro: *proper-it-loc rs.rev-iterateoi rs.rev-iterateoi*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
rs.empty,
rs.memb,
rs.ins,
rs.delete,
rs.list-it,
rs.sng,
rs.isEmpty,
rs.isSng,
rs.ball,
rs.bex,
rs.size,
rs.size-abort,
rs.union,
rs.union-dj,
rs.diff,
rs.filter,
rs.inter,
rs.subset,
rs.equal,
rs.disjoint,
rs.disjoint-witness,
rs.sel,
rs.to-list,
rs.from-list,

rs.ordered-list-it,
rs.rev-list-it,
rs.min,
rs.max,
rs.to-sorted-list,
rs.to-rev-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.18 Hash Set

theory *HashSet*
imports
../spec/SetSpec
HashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
'a hs = (*'a*::*hashable*,*unit*) *hm*

⟨*ML*⟩
interpretation *hs-sbm*: *SetByMap hm-basic-ops* ⟨*proof*⟩
⟨*ML*⟩

definition *hs-ops* :: (*'a*::*hashable*,*'a hs*) *set-ops*
where [*icf-rec-def*]:
hs-ops ≡ *hs-sbm.basic.dflt-ops*

⟨*ML*⟩
interpretation *hs*: *StdSet hs-ops*
⟨*proof*⟩
interpretation *hs*: *StdSet-no-invar hs-ops*
⟨*proof*⟩
⟨*ML*⟩

lemmas *hs-it-to-it-map-code-unfold*[*code-unfold*] =
it-to-it-map-fold'[*OF pi-hm*]

lemma *pi-hs*[*proper-it*]: *proper-it' hs.iteratei hs.iteratei*
⟨*proof*⟩

interpretation *pi-hs*: *proper-it-loc hs.iteratei hs.iteratei*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
hs.empty,
hs.memb,
hs.ins,
hs.delete,
hs.list-it,
hs.sng,
hs.isEmpty,
hs.isSng,
hs.ball,
hs.bex,

hs.size,
hs.size-abort,
hs.union,
hs.union-dj,
hs.diff,
hs.filter,
hs.inter,
hs.subset,
hs.equal,
hs.disjoint,
hs.disjoint-witness,
hs.sel,
hs.to-list,
hs.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.19 Set implementation via tries

theory *TrieSetImpl* **imports**
TrieMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
$'a$ *ts* = ($'a$, *unit*) *trie*

⟨*ML*⟩
interpretation *ts-sbm*: *SetByMap tm-basic-ops* ⟨*proof*⟩
⟨*ML*⟩

definition *ts-ops* :: ($'a$ *list*,$'a$ *ts*) *set-ops*
where [*icf-rec-def*]:
ts-ops ≡ *ts-sbm.basic.dflt-ops*

⟨*ML*⟩
interpretation *ts*: *StdSet ts-ops*
⟨*proof*⟩
interpretation *ts*: *StdSet-no-invar ts-ops*
⟨*proof*⟩
⟨*ML*⟩

lemmas *ts-it-to-it-map-code-unfold*[*code-unfold*] =
it-to-it-map-fold'[*OF pi-trie*]

lemma *pi-ts*[*proper-it*]: *proper-it'* *ts.iteratei* *ts.iteratei*
⟨*proof*⟩

interpretation *pi-ts*: *proper-it-loc* *ts.iteratei* *ts.iteratei*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
ts.empty,
ts.memb,
ts.ins,
ts.delete,
ts.list-it,
ts.sng,
ts.isEmpty,
ts.isSng,
ts.ball,
ts.bex,
ts.size,
ts.size-abort,
ts.union,
ts.union-dj,
ts.diff,
ts.filter,
ts.inter,
ts.subset,
ts.equal,
ts.disjoint,
ts.disjoint-witness,
ts.sel,
ts.to-list,
ts.from-list
)

export-code *test-codegen* **checking** *SML*

end

theory *ArrayHashSet*
imports
ArrayHashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
'a ahs = (*'a*::*hashable*,*unit*) *ahm*

⟨*ML*⟩
interpretation *ahs-sbm*: *SetByMap ahm-basic-ops* ⟨*proof*⟩
⟨*ML*⟩

definition *ahs-ops* :: (*'a*::*hashable*,*'a ahs*) *set-ops*
  where [*icf-rec-def*]:
  ahs-ops ≡ *ahs-sbm.basic.dflt-ops*

⟨*ML*⟩
interpretation *ahs*: *StdSet ahs-ops*
  ⟨*proof*⟩
interpretation *ahs*: *StdSet-no-invar ahs-ops*
  ⟨*proof*⟩
⟨*ML*⟩

lemmas *ahs-it-to-it-map-code-unfold*[*code-unfold*] =
  it-to-it-map-fold'[*OF pi-ahm*]

lemma *pi-ahs*[*proper-it*]: *proper-it' ahs.iteratei ahs.iteratei*
  ⟨*proof*⟩

interpretation *pi-ahs*: *proper-it-loc ahs.iteratei ahs.iteratei*
  ⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
  ahs.empty,
  ahs.memb,
  ahs.ins,
  ahs.delete,
  ahs.list-it,
  ahs.sng,
  ahs.isEmpty,
  ahs.isSng,
  ahs.ball,
  ahs.bex,
  ahs.size,
  ahs.size-abort,
  ahs.union,
  ahs.union-dj,
  ahs.diff,
  ahs.filter,
  ahs.inter,
  ahs.subset,
  ahs.equal,
  ahs.disjoint,
  ahs.disjoint-witness,
  ahs.sel,
  ahs.to-list,

ahs.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.20 Set Implementation by Arrays

theory *ArraySetImpl*
imports
../spec/SetSpec
ArrayMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym *ias* = (*unit*) *iam*

⟨*ML*⟩
interpretation *ias-sbm*: *OSetByOMap iam-basic-ops* ⟨*proof*⟩
⟨*ML*⟩
definition *ias-ops* :: (*nat*,*ias*) *oset-ops*
where [*icf-rec-def*]:
ias-ops ≡ *ias-sbm.obasic.dflt-oops*

⟨*ML*⟩
interpretation *ias*: *StdOSet ias-ops*
⟨*proof*⟩
interpretation *ias*: *StdSet-no-invar ias-ops*
⟨*proof*⟩
⟨*ML*⟩

lemmas *ias-it-to-it-map-code-unfold*[*code-unfold*] =
it-to-it-map-fold′[*OF pi-iam*]
it-to-it-map-fold′[*OF pi-iam-rev*]

lemma *pi-ias*[*proper-it*]:
proper-it′ ias.iteratei ias.iteratei
proper-it′ ias.iterateoi ias.iterateoi
proper-it′ ias.rev-iterateoi ias.rev-iterateoi
⟨*proof*⟩

interpretation
pi-ias: *proper-it-loc ias.iteratei ias.iteratei* +
pi-ias-o: *proper-it-loc ias.iterateoi ias.iterateoi* +
pi-ias-ro: *proper-it-loc ias.rev-iterateoi ias.rev-iterateoi*
⟨*proof*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
  ias.empty,
  ias.memb,
  ias.ins,
  ias.delete,
  ias.list-it,
  ias.sng,
  ias.isEmpty,
  ias.isSng,
  ias.ball,
  ias.bex,
  ias.size,
  ias.size-abort,
  ias.union,
  ias.union-dj,
  ias.diff,
  ias.filter,
  ias.inter,
  ias.subset,
  ias.equal,
  ias.disjoint,
  ias.disjoint-witness,
  ias.sel,
  ias.to-list,
  ias.from-list,

  ias.ordered-list-it,
  ias.rev-list-it,
  ias.min,
  ias.max,
  ias.to-sorted-list,
  ias.to-rev-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.21 Standard Set Implementations

theory *SetStdImpl*
imports
  ListSetImpl
  ListSetImpl-Invar
  ListSetImpl-NotDist
  ListSetImpl-Sorted
  RBTSetImpl HashSet
  TrieSetImpl

ArrayHashSet
ArraySetImpl
begin

This theory summarizes standard set implementations, namely list-sets RB-tree-sets, trie-sets and hashsets.

end

3.3.22 Fifo Queue by Pair of Lists

theory *Fifo*
imports
../gen-algo/ListGA
../tools/Record-Intf
../tools/Locale-Code
begin lemma *rev-tl-rev*: *rev* (*tl* (*rev l*)) = *butlast l*
⟨*proof*⟩

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.

Definitions

type-synonym *'a fifo* = *'a list* × *'a list*

Abstraction of the fifo to a list. The next element to be got is at the head of the list, and new elements are appended at the end of the list

definition *fifo-α* :: *'a fifo* ⇒ *'a list*
where *fifo-α F* == *snd F* @ *rev* (*fst F*)

This fifo implementation has no invariants, any pair of lists is a valid fifo

definition [*simp*, *intro*!]: *fifo-invar x* = *True*

— 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*=([],[])

```
definition fifo-size :: 'a fifo ⇒ nat where
  fifo-size F ≡ length (fst F) + length (snd F)

  — Add an element to the fifo
definition fifo-appendr :: 'a ⇒ 'a fifo ⇒ 'a fifo
  where fifo-appendr a F == (a#fst F, snd F)

definition fifo-appendl :: 'a ⇒ 'a fifo ⇒ 'a fifo
  where fifo-appendl x F == case F of (e,d) ⇒ (e,x#d)

— Get an element from the fifo
definition fifo-remover :: 'a fifo ⇒ ('a fifo × 'a) where
  fifo-remover F ==
    case fst F of
      (a#l) ⇒ ((l,snd F),a) |
      [] ⇒ let rp=rev (snd F) in
        ((tl rp,[]),hd rp)

definition fifo-removel :: 'a fifo ⇒ ('a × 'a fifo) where
  fifo-removel F ==
    case snd F of
      (a#l) ⇒ (a, (fst F, l)) |
      [] ⇒ let rp=rev (fst F) in
        (hd rp, ([], tl rp))


definition fifo-leftmost :: 'a fifo ⇒ 'a where
  fifo-leftmost F ≡ case F of (-,x#-) ⇒ x | (l,[]) ⇒ last l

definition fifo-rightmost :: 'a fifo ⇒ 'a where
  fifo-rightmost F ≡ case F of (x#-,-) ⇒ x | ([],l) ⇒ last l

definition fifo-iteratei F ≡ foldli (fifo-α F)
definition fifo-rev-iteratei F ≡ foldri (fifo-α F)

definition fifo-get F i ≡
  let
    l2 = length (snd F)
  in
    if i < l2 then
      snd F!i
    else
      (fst F)!(length (fst F) − Suc (i − l2))


definition fifo-set F i a ≡ case F of (f1,f2) ⇒
  let
    l2 = length f2
  in
```

if $i < l2$ *then*
  $(f1, f2[i:=a])$
else
  $(f1[length\ (fst\ F) - Suc\ (i - l2) := a], f2)$

Correctness

lemma *fifo-empty-impl*: *list-empty fifo-*α *fifo-invar fifo-empty*
⟨*proof*⟩

lemma *fifo-isEmpty-impl*: *list-isEmpty fifo-*α *fifo-invar fifo-isEmpty*
⟨*proof*⟩

lemma *fifo-size-impl*: *list-size fifo-*α *fifo-invar fifo-size*
⟨*proof*⟩

lemma *fifo-appendr-impl*: *list-appendr fifo-*α *fifo-invar fifo-appendr*
⟨*proof*⟩

lemma *fifo-appendl-impl*: *list-appendl fifo-*α *fifo-invar fifo-appendl*
⟨*proof*⟩

lemma *fifo-removel-impl*: *list-removel fifo-*α *fifo-invar fifo-removel*
⟨*proof*⟩

lemma *fifo-remover-impl*: *list-remover fifo-*α *fifo-invar fifo-remover*
⟨*proof*⟩

lemma *fifo-leftmost-impl*: *list-leftmost fifo-*α *fifo-invar fifo-leftmost*
⟨*proof*⟩

lemma *fifo-rightmost-impl*: *list-rightmost fifo-*α *fifo-invar fifo-rightmost*
⟨*proof*⟩

lemma *fifo-get-impl*: *list-get fifo-*α *fifo-invar fifo-get*
⟨*proof*⟩

lemma *fifo-set-impl*: *list-set fifo-*α *fifo-invar fifo-set*
⟨*proof*⟩

definition [*icf-rec-def*]: *fifo-ops* ≡ (|
  list-op-α = *fifo-*α,
  list-op-invar = *fifo-invar*,
  list-op-empty = *fifo-empty*,
  list-op-isEmpty = *fifo-isEmpty*,
  list-op-size = *fifo-size*,
  list-op-appendl = *fifo-appendl*,
  list-op-removel = *fifo-removel*,
  list-op-leftmost = *fifo-leftmost*,

list-op-appendr = *fifo-appendr*,
list-op-remover = *fifo-remover*,
list-op-rightmost = *fifo-rightmost*,
list-op-get = *fifo-get*,
list-op-set = *fifo-set*
⟫

⟨*ML*⟩
interpretation *fifo*: *StdList fifo-ops*
⟨*proof*⟩
interpretation *fifo*: *StdList-no-invar fifo-ops*
⟨*proof*⟩
⟨*ML*⟩

definition *test-codegen* **where** *test-codegen* ≡
(
fifo.empty,
fifo.isEmpty,
fifo.size,
fifo.appendl,
fifo.removel,
fifo.leftmost,
fifo.appendr,
fifo.remover,
fifo.rightmost,
fifo.get,
fifo.set,
fifo.iteratei,
fifo.rev-iteratei
)

export-code *test-codegen* **checking** *SML*

end

3.3.23 Implementation of Priority Queues by Binomial Heap

theory *BinoPrioImpl*
imports
Binomial−Heaps.BinomialHeap
../spec/PrioSpec
../tools/Record-Intf
../tools/Locale-Code
begin

type-synonym ('*a*,'*b*) *bino* = ('*a*,'*b*) *BinomialHeap*

Definitions

definition *bino-α* **where** *bino-α q* ≡ *BinomialHeap.to-mset q*
definition *bino-insert* **where** *bino-insert* ≡ *BinomialHeap.insert*
abbreviation (*input*) *bino-invar* :: ('*a*,'*b*) *BinomialHeap* ⇒ *bool*
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 [*icf-rec-def*]: *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

Correctness

theorem *bino-empty-impl*: *prio-empty bino-α bino-invar bino-empty*
⟨*proof*⟩

theorem *bino-isEmpty-impl*: *prio-isEmpty bino-α bino-invar bino-isEmpty*
⟨*proof*⟩

theorem *bino-find-impl*: *prio-find bino-α bino-invar bino-find*
⟨*proof*⟩

lemma *bino-insert-impl*: *prio-insert bino-α bino-invar bino-insert*
⟨*proof*⟩

lemma *bino-meld-impl*: *prio-meld bino-α bino-invar bino-meld*
⟨*proof*⟩

lemma *bino-delete-impl*:

prio-delete bino-α bino-invar bino-find bino-delete
⟨*proof*⟩

⟨*ML*⟩
interpretation *bino*: *StdPrio bino-ops*
⟨*proof*⟩
interpretation *bino*: *StdPrio-no-invar bino-ops*
⟨*proof*⟩
⟨*ML*⟩

definition *test-codegen* **where** *test-codegen* = (
bino.empty,
bino.isEmpty,
bino.find,
bino.insert,
bino.meld,
bino.delete
)

export-code *test-codegen* **checking** *SML*

end

3.3.24 Implementation of Priority Queues by Skew Binomial Heaps

theory *SkewPrioImpl*
imports
Binomial−Heaps.SkewBinomialHeap
../spec/PrioSpec
../tools/Record-Intf
../tools/Locale-Code
begin

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* :: ('*a*, '*b*) *SkewBinomialHeap* ⇒ *bool*
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 [*icf-rec-def*]: *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

Correctness

theorem *skew-empty-impl*: *prio-empty skew-α skew-invar skew-empty*
⟨*proof*⟩

theorem *skew-isEmpty-impl*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
⟨*proof*⟩

theorem *skew-find-impl*: *prio-find skew-α skew-invar skew-find*
⟨*proof*⟩

lemma *skew-insert-impl*: *prio-insert skew-α skew-invar skew-insert*
⟨*proof*⟩

lemma *skew-meld-impl*: *prio-meld skew-α skew-invar skew-meld*
⟨*proof*⟩

lemma *skew-delete-impl*:
prio-delete skew-α skew-invar skew-find skew-delete
⟨*proof*⟩

⟨*ML*⟩
interpretation *skew*: *StdPrio skew-ops*
⟨*proof*⟩
interpretation *skew*: *StdPrio-no-invar skew-ops*
⟨*proof*⟩
⟨*ML*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
skew.empty,
skew.isEmpty,

skew.find,
skew.insert,
skew.meld,
skew.delete
)

export-code *test-codegen* **checking** *SML*

end

3.3.25 Implementation of Annotated Lists by 2-3 Finger Trees

theory *FTAnnotatedListImpl*
imports
Finger−Trees.FingerTree
../tools/Locale-Code
../tools/Record-Intf
../spec/AnnotatedListSpec
begin

type-synonym $('a,'b)$ *ft* = $('a,'b)$ *FingerTree*

Definitions

definition *ft-α* **where**
ft-α ≡ *FingerTree.toList*
abbreviation (*input*) *ft-invar* :: $('a,'b)$ *FingerTree* ⇒ *bool* **where**
ft-invar ≡ λ-. *True*
definition *ft-empty* **where**
ft-empty ≡ λ-::*unit*. *FingerTree.empty*
definition *ft-isEmpty* **where**
ft-isEmpty ≡ *FingerTree.isEmpty*
definition *ft-count* **where**
ft-count ≡ *FingerTree.count*
definition *ft-consl* **where**
ft-consl e a s = *FingerTree.lcons* (*e*,*a*) *s*
definition *ft-consr* **where**
ft-consr s e a = *FingerTree.rcons s* (*e*,*a*)
definition *ft-head* **where**
ft-head ≡ *FingerTree.head*
definition *ft-tail* **where**
ft-tail ≡ *FingerTree.tail*
definition *ft-headR* **where**
ft-headR ≡ *FingerTree.headR*
definition *ft-tailR* **where**
ft-tailR ≡ *FingerTree.tailR*
definition *ft-foldl* **where**
ft-foldl ≡ *FingerTree.foldl*

definition *ft-foldr* **where**
ft-foldr ≡ *FingerTree.foldr*
definition *ft-app* **where**
ft-app ≡ *FingerTree.app*
definition *ft-annot* **where**
ft-annot ≡ *FingerTree.annot*
definition *ft-splits* **where**
ft-splits ≡ *FingerTree.splitTree*

lemmas *ft-defs* =
ft-α-def
ft-empty-def
ft-isEmpty-def
ft-count-def
ft-consl-def
ft-consr-def
ft-head-def
ft-tail-def
ft-headR-def
ft-tailR-def
ft-foldl-def
ft-foldr-def
ft-app-def
ft-annot-def
ft-splits-def

lemma *ft-empty-impl*: *al-empty ft-α ft-invar ft-empty*
⟨*proof*⟩

lemma *ft-consl-impl*: *al-consl ft-α ft-invar ft-consl*
⟨*proof*⟩

lemma *ft-consr-impl*: *al-consr ft-α ft-invar ft-consr*
⟨*proof*⟩

lemma *ft-isEmpty-impl*: *al-isEmpty ft-α ft-invar ft-isEmpty*
⟨*proof*⟩

lemma *ft-count-impl*: *al-count ft-α ft-invar ft-count*
⟨*proof*⟩

lemma *ft-head-impl*: *al-head ft-α ft-invar ft-head*
⟨*proof*⟩

lemma *ft-tail-impl*: *al-tail ft-α ft-invar ft-tail*
⟨*proof*⟩

lemma *ft-headR-impl*: *al-headR ft-α ft-invar ft-headR*
⟨*proof*⟩

lemma *ft-tailR-impl*: *al-tailR ft-α ft-invar ft-tailR*
⟨*proof*⟩

lemma *ft-foldl-impl*: *al-foldl ft-α ft-invar ft-foldl*
⟨*proof*⟩

lemma *ft-foldr-impl*: *al-foldr ft-α ft-invar ft-foldr*
⟨*proof*⟩

lemma *ft-foldl-cunfold*[*code-unfold*]:
List.foldl f σ (*ft-α t*) = *ft-foldl f σ t*
⟨*proof*⟩

lemma *ft-foldr-cunfold*[*code-unfold*]:
List.foldr f (*ft-α t*) *σ* = *ft-foldr f t σ*
⟨*proof*⟩

lemma *ft-app-impl*: *al-app ft-α ft-invar ft-app*
⟨*proof*⟩

lemma *ft-annot-impl*: *al-annot ft-α ft-invar ft-annot*
⟨*proof*⟩

lemma *ft-splits-impl*: *al-splits ft-α ft-invar ft-splits*
⟨*proof*⟩

Record Based Implementation **definition** [*icf-rec-def*]: *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*
|)

⟨*ML*⟩
interpretation *ft*: *StdAL ft-ops*
⟨*proof*⟩
interpretation *ft*: *StdAL-no-invar ft-ops*
⟨*proof*⟩

⟨*ML*⟩

definition *test-codegen* ≡ (
  ft.empty,
  ft.isEmpty,
  ft.count,
  ft.consl,
  ft.consr,
  ft.head,
  ft.tail,
  ft.headR,
  ft.tailR,
  ft.app,
  ft.annot,
  ft.splits,
  ft.foldl,
  ft.foldr
)

export-code *test-codegen* **checking** *SML*

end

3.3.26 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*

⟨*ML*⟩
interpretation *alprio-ga*: *alprio ft-ops* ⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *alprioi-ops* ≡ *alprio-ga.alprio-ops*

⟨*ML*⟩
interpretation *alprioi*: *StdPrio alprioi-ops*
  ⟨*proof*⟩
⟨*ML*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
  alprioi.empty,
  alprioi.isEmpty,

alprioi.insert,
alprioi.find,
alprioi.delete,
alprioi.meld
)

export-code *test-codegen* **checking** *SML*

end

3.3.27 Implementation of Unique Priority Queues by Finger Trees

theory *FTPrioUniqueImpl*
imports
FTAnnotatedListImpl
../gen-algo/PrioUniqueByAnnotatedList
begin

Definitions

type-synonym ('*a*,'*b*) *aluprioi* = (*unit*, ('*a*, '*b*) *LP*) *FingerTree*

⟨*ML*⟩
interpretation *aluprio-ga*: *aluprio ft-ops* ⟨*proof*⟩
⟨*ML*⟩

definition [*icf-rec-def*]: *aluprioi-ops* ≡ *aluprio-ga.aluprio-ops*

⟨*ML*⟩
interpretation *aluprioi*: *StdUprio aluprioi-ops*
⟨*proof*⟩
⟨*ML*⟩

definition *test-codegen* **where** *test-codegen* ≡ (
aluprioi.empty,
aluprioi.isEmpty,
aluprioi.insert,
aluprioi.pop,
aluprioi.prio
)

export-code *test-codegen* **checking** *SML*

end

3.4 Entry Points

3.4.1 Standard Collections

theory *Collections*
imports
ICF-Impl
ICF-Refine-Monadic
ICF-Autoref

DatRef

begin

This theory summarizes the components of the Isabelle Collection Framework.

end

3.4.2 Backwards Compatibility for Version 1

theory *CollectionsV1*
imports *Collections*
begin

This theory defines some stuff to establish (partial) backwards compatibility with ICF Version 1.

⟨*ML*⟩

Iterators

We define all the monomorphic iterator locales

Set **locale** *set-iteratei* = *finite-set* α *invar* **for** α :: $'s \Rightarrow 'x$ *set* **and** *invar* +
fixes *iteratei* :: $'s \Rightarrow ('x, 'σ)$ *set-iterator*

assumes *iteratei-rule*: *invar* $S \Longrightarrow$ *set-iterator* (*iteratei* S) (α S)
begin
lemma *iteratei-rule-P*:
$[\![$
invar S;
I (α S) $\sigma 0$;
$!!x\ it\ \sigma.\ [\![\ c\ \sigma;\ x \in it;\ it \subseteq \alpha\ S;\ I\ it\ \sigma\ ]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$;
$!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$;
$!!\sigma\ it.\ [\![\ it \subseteq \alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
$]\!] \Longrightarrow P$ (*iteratei* S c f $\sigma 0$)
⟨*proof*⟩

lemma *iteratei-rule-insert-P*:

⟦
invar S;
I {} *σ0*;
!!*x it σ*. ⟦ *c σ*; *x* ∈ *α S* − *it*; *it* ⊆ *α S*; *I it σ* ⟧ ⟹ *I* (*insert x it*) (*f x σ*);
!!*σ*. *I* (*α S*) *σ* ⟹ *P σ*;
!!*σ it*. ⟦ *it* ⊆ *α S*; *it* ≠ *α S*; ¬ *c σ*; *I it σ* ⟧ ⟹ *P σ*
⟧ ⟹ *P* (*iteratei S c f σ0*)
⟨*proof*⟩

Versions without break condition.

lemma *iterate-rule-P*:
⟦
invar S;
I (*α S*) *σ0*;
!!*x it σ*. ⟦ *x* ∈ *it*; *it* ⊆ *α S*; *I it σ* ⟧ ⟹ *I* (*it* − {*x*}) (*f x σ*);
!!*σ*. *I* {} *σ* ⟹ *P σ*
⟧ ⟹ *P* (*iteratei S* (λ-. *True*) *f σ0*)
⟨*proof*⟩

lemma *iterate-rule-insert-P*:
⟦
invar S;
I {} *σ0*;
!!*x it σ*. ⟦ *x* ∈ *α S* − *it*; *it* ⊆ *α S*; *I it σ* ⟧ ⟹ *I* (*insert x it*) (*f x σ*);
!!*σ*. *I* (*α S*) *σ* ⟹ *P σ*
⟧ ⟹ *P* (*iteratei S* (λ-. *True*) *f σ0*)
⟨*proof*⟩
end

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

locale *set-iterateoi* = *ordered-finite-set α invar*
for *α* :: *'s* ⇒ (*'u*::*linorder*) *set* **and** *invar*
+
fixes *iterateoi* :: *'s* ⇒ (*'u*,*'σ*) *set-iterator*
assumes *iterateoi-rule*:
invar s ⟹ *set-iterator-linord* (*iterateoi s*) (*α s*)
begin
lemma *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 $\subseteq$ α *m*;
I it σ
$\rrbracket \Longrightarrow$ *I* (*it* $-$ {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow$ *P* σ
assumes *II*: !!σ *it*. $\llbracket$
it $\subseteq$ α *m*;
it $\neq$ {};
$\neg$ *c* σ;
I it σ;
$\forall k{\in}it.\ \forall j{\in}\alpha\ m - it.\ j{\leq}k$
$\rrbracket \Longrightarrow$ *P* σ
shows *P* (*iterateoi m c f* $\sigma 0$)
⟨*proof*⟩

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

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

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

assumes *IF*: !!σ. I {} $\sigma \Longrightarrow P\ \sigma$
assumes *II*: !!σ *it*. ⟦
$it \subseteq (\alpha\ m)$;
$it \neq \{\}$;
$\neg\ c\ \sigma$;
$I\ it\ \sigma$;
$\forall k{\in}it.\ \forall j{\in}(\alpha\ m) - it.\ j{\geq}k$
⟧ $\Longrightarrow P\ \sigma$
shows P (*reverse-iterateoi m c f* $\sigma 0$)
⟨*proof*⟩

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

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

lemma (**in** *poly-set-iteratei*) *v1-iteratei-impl*:
set-iteratei α *invar iteratei*
⟨*proof*⟩
lemma (**in** *poly-set-iterateoi*) *v1-iterateoi-impl*:
set-iterateoi α *invar iterateoi*
⟨*proof*⟩
lemma (**in** *poly-set-rev-iterateoi*) *v1-reverse-iterateoi-impl*:
set-reverse-iterateoi α *invar rev-iterateoi*
⟨*proof*⟩

declare (**in** *poly-set-iteratei*) *v1-iteratei-impl*[*locale-witness-add*]
declare (**in** *poly-set-iterateoi*) *v1-iterateoi-impl*[*locale-witness-add*]
declare (**in** *poly-set-rev-iterateoi*)
v1-reverse-iterateoi-impl[*locale-witness-add*]

Map **locale** *map-iteratei* = *finite-map* α *invar* **for** α :: $'s \Rightarrow 'u \rightharpoonup 'v$ **and** *invar*
+

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* {} $\sigma \Longrightarrow$ *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* $\sigma 0$)
⟨*proof*⟩

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*)) $\sigma \Longrightarrow$ *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* $\sigma 0$)
⟨*proof*⟩

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* {} $\sigma \Longrightarrow$ *P* σ
$]\!] \Longrightarrow$ *P* (*iteratei m* (λ-. *True*) *f* $\sigma 0$)
⟨*proof*⟩

lemma *iterate-rule-insert-P*:
$[\![$
invar m;
I {} $\sigma 0$;
!!*k v it* σ. $[\![$ *k* $\in$ (*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*)) $\sigma \Longrightarrow$ *P* σ
$]\!] \Longrightarrow$ *P* (*iteratei m* (λ-. *True*) *f* $\sigma 0$)
⟨*proof*⟩
end

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

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*. $k \leq j$;
$\forall j \in$*dom* (α m) − *it*. $j \leq k$;
α m k = *Some* v;
it ⊆ *dom* (α m);
I it σ
⟧ $\Longrightarrow$ *I* (*it* − {k}) (*f* (k, v) σ)
assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow$ *P* σ
assumes *II*: !!σ *it*. ⟦
it ⊆ *dom* (α m);
it ≠ {};
¬ c σ;
I it σ;
$\forall k \in$*it*. $\forall j \in$*dom* (α m) − *it*. $j \leq k$
⟧ $\Longrightarrow$ *P* σ
shows *P* (*iterateoi* m c f $\sigma 0$)
⟨*proof*⟩

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

lemma *map-iterateoi-I* :

assumes $\bigwedge m.\ invar\ m \Longrightarrow map\text{-}iterator\text{-}linord\ (itoi\ m)\ (\alpha\ m)$
shows *map-iterateoi* α *invar itoi*
⟨*proof*⟩

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

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

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

lemma (**in** *poly-map-iteratei*) *v1-iteratei-impl*:
map-iteratei α *invar iteratei*
⟨*proof*⟩
lemma (**in** *poly-map-iterateoi*) *v1-iterateoi-impl*:
map-iterateoi α *invar iterateoi*
⟨*proof*⟩
lemma (**in** *poly-map-rev-iterateoi*) *v1-reverse-iterateoi-impl*:
map-reverse-iterateoi α *invar rev-iterateoi*
⟨*proof*⟩

declare (**in** *poly-map-iteratei*) *v1-iteratei-impl*[*locale-witness-add*]
declare (**in** *poly-map-iterateoi*) *v1-iterateoi-impl*[*locale-witness-add*]
declare (**in** *poly-map-rev-iterateoi*)
v1-reverse-iterateoi-impl[*locale-witness-add*]

Concrete Operation Names

We define abbreviations to recover the *xx-op*-names

⟨*ML*⟩

lemmas *hs-correct* = *hs.correct*
lemmas *hm-correct* = *hm.correct*
lemmas *rs-correct* = *rs.correct*
lemmas *rm-correct* = *rm.correct*
lemmas *ls-correct* = *ls.correct*
lemmas *lm-correct* = *lm.correct*
lemmas *lsi-correct* = *lsi.correct*
lemmas *lmi-correct* = *lmi.correct*
lemmas *lsnd-correct* = *lsnd.correct*
lemmas *lss-correct* = *lss.correct*
lemmas *ts-correct* = *ts.correct*
lemmas *tm-correct* = *tm.correct*
lemmas *ias-correct* = *ias.correct*
lemmas *iam-correct* = *iam.correct*
lemmas *ahs-correct* = *ahs.correct*
lemmas *ahm-correct* = *ahm.correct*
lemmas *bino-correct* = *bino.correct*
lemmas *fifo-correct* = *fifo.correct*
lemmas *ft-correct* = *ft.correct*
lemmas *alprioi-correct* = *alprioi.correct*

lemmas *aluprioi-correct* = *aluprioi.correct*
lemmas *skew-correct* = *skew.correct*

locale *list-enqueue* = *list-appendr*
locale *list-dequeue* = *list-removel*

locale *list-push* = *list-appendl*
locale *list-pop* = *list-remover*
locale *list-top* = *list-leftmost*
locale *list-bot* = *list-rightmost*

instantiation *rbt* :: ({*equal*,*linorder*},*equal*) *equal*
begin

definition *equal-class.equal* (*r* :: ('*a*, '*b*) *rbt*) *r'*
== *RBT.impl-of r* = *RBT.impl-of r'*

instance
⟨*proof*⟩
end

end

Chapter 4

Entry Points

4.1 Entry Points

This chapter describes the overall entrypoints to the combination of Automatic Refinement Framework, Monadic Refinement Framework, and Collection Framework. These are the theories a typical algorithm development should be based on.

4.1.1 Default Setup

theory *Refine-Dflt*
imports
Refine-Monadic.Refine-Monadic
GenCF/GenCF
Lib/Code-Target-ICF
begin

Configurations

lemmas *tyrel-dflt-nat-set* =
ty-REL[**where** *'a=nat set* **and** *R=⟨Id⟩dflt-rs-rel*]

lemmas *tyrel-dflt-bool-set* =
ty-REL[**where** *'a=bool set* **and** *R=⟨Id⟩list-set-rel*]

lemmas *tyrel-dflt-nat-map* =
ty-REL[**where** *R=⟨nat-rel,Rv⟩dflt-rm-rel*] **for** *Rv*

lemmas *tyrel-dflt-old* = *tyrel-dflt-nat-set tyrel-dflt-bool-set tyrel-dflt-nat-map*

lemmas *tyrel-dflt-linorder-set* =
ty-REL[**where** *R=⟨Id::('a::linorder×'a) set⟩dflt-rs-rel*]

lemmas *tyrel-dflt-linorder-map* =
ty-REL[**where** *R=⟨Id::('a::linorder×'a) set,R⟩dflt-rm-rel*] **for** *R*

lemmas *tyrel-dflt-bool-map* =
ty-REL[**where** *R*=⟨*Id*::(*bool*×*bool*) *set*,*R*⟩*list-map-rel*] **for** *R*

lemmas *tyrel-dflt* = *tyrel-dflt-linorder-set tyrel-dflt-bool-set tyrel-dflt-linorder-map tyrel-dflt-bool-map*

declare *tyrel-dflt*[*autoref-tyrel*]

⟨*ML*⟩

Fallbacks

⟨*ML*⟩

Quick test of setup:

context begin
private schematic-goal *test-dflt-tyrel1*: (*?c*::*?'c*,{*1*,*2*,*3*::*int*})∈*?R*
⟨*proof*⟩ **lemmas** *asm-rl*[*of* -∈⟨*int-rel*⟩*dflt-rs-rel*, *OF test-dflt-tyrel1*]

private schematic-goal *test-dflt-tyrel2*: (*?c*::*?'c*,{*True*, *False*})∈*?R*
⟨*proof*⟩ **lemmas** *asm-rl*[*of* -∈⟨*bool-rel*⟩*list-set-rel*, *OF test-dflt-tyrel2*]

private schematic-goal *test-dflt-tyrel3*: (*?c*::*?'c*,[*1*::*int*↦*0*::*nat*])∈*?R*
⟨*proof*⟩ **lemmas** *asm-rl*[*of* -∈⟨*int-rel*,*nat-rel*⟩*dflt-rm-rel*, *OF test-dflt-tyrel3*]

private schematic-goal *test-dflt-tyrel4*: (*?c*::*?'c*,[*False*↦*0*::*nat*])∈*?R*
⟨*proof*⟩ **lemmas** *asm-rl*[*of* -∈⟨*bool-rel*,*nat-rel*⟩*list-map-rel*, *OF test-dflt-tyrel4*]

end

end

4.1.2 Entry Point with genCF and original ICF

theory *Refine-Dflt-ICF*
imports
Refine-Monadic.Refine-Monadic
GenCF/GenCF
ICF/Collections
Lib/Code-Target-ICF
begin

Contains the Monadic Refinement Framework, the generic collection framework and the original Isabelle Collection Framework

⟨*ML*⟩

Fallbacks

⟨*ML*⟩

class *id-refine*

instance *nat* :: *id-refine* ⟨*proof*⟩
instance *bool* :: *id-refine* ⟨*proof*⟩
instance *int* :: *id-refine* ⟨*proof*⟩

lemmas [*autoref-tyrel*] =
ty-REL[**where** *'a=nat* **and** *R=nat-rel*]
ty-REL[**where** *'a=int* **and** *R=int-rel*]
ty-REL[**where** *'a=bool* **and** *R=bool-rel*]
lemmas [*autoref-tyrel*] =
ty-REL[**where** *'a=nat set* **and** *R=⟨Id⟩rs.rel*]
ty-REL[**where** *'a=int set* **and** *R=⟨Id⟩rs.rel*]
ty-REL[**where** *'a=bool set* **and** *R=⟨Id⟩lsi.rel*]
lemmas [*autoref-tyrel*] =
ty-REL[**where** *'a=(nat ⇁ 'b)*, **where** *R=⟨nat-rel,Rv⟩dflt-rm-rel*]
ty-REL[**where** *'a=(int ⇁ 'b)*, **where** *R=⟨int-rel,Rv⟩dflt-rm-rel*]
ty-REL[**where** *'a=(bool ⇁ 'b)*, **where** *R=⟨bool-rel,Rv⟩dflt-rm-rel*]
for *Rv*

lemmas [*autoref-tyrel*] =
ty-REL[**where** *'a=(nat ⇁ 'b::id-refine)*, **where** *R=⟨nat-rel,Id⟩rm.rel*]
ty-REL[**where** *'a=(int ⇁ 'b::id-refine)*, **where** *R=⟨int-rel,Id⟩rm.rel*]
ty-REL[**where** *'a=(bool ⇁ 'b::id-refine)*, **where** *R=⟨bool-rel,Id⟩rm.rel*]

end

4.1.3 Entry Point with only the ICF

theory *Refine-Dflt-Only-ICF*
imports
Refine-Monadic.Refine-Monadic
ICF/Collections
Lib/Code-Target-ICF
begin

Contains the Monadic Refinement Framework and the original Isabelle Collection Framework. The generic collection framework is not included

⟨*ML*⟩

end

Chapter 5

Userguides

This chapter contains various userguides.

5.1 Old Monadic Refinement Framework Userguide

5.1.1 Introduction

This is the old userguide from Refine-Monadic. It contains the manual approach of using the mondaic refinement framework with the Isabelle Collection Framework. An alternative, more simple approach is provided by the Automatic Refinement Framework and the Generic Collection Framework.

The Isabelle/HOL refinement framework is a library that supports program and data refinement.

Programs are specified using a nondeterminism monad: An element of the monad type is either a set of results, or the special element *FAIL*, that indicates a failed assertion.

The bind-operation of the monad applies a function to all elements of the result-set, and joins all possible results.

On the monad type, an ordering $\leq$ is defined, that is lifted subset ordering, where *FAIL* is the greatest element. Intuitively, $S \leq S'$ means that program S refines program S', i.e., all results of S are also results of S', and S may only fail if S' also fails.

5.1.2 Guided Tour

In this section, we provide a small example program development in our framework. All steps of the development are heavily commented.

Defining Programs

A program is defined using the Haskell-like do-notation, that is provided by the Isabelle/HOL library. We start with a simple example, that iterates over a set of numbers, and computes the maximum value and the sum of all elements.

```
definition sum-max :: nat set ⇒ (nat×nat) nres where
  sum-max V ≡ do {
    (-,s,m) ← WHILE (λ(V,s,m). V≠{}) (λ(V,s,m). do {
      x←SPEC (λx. x∈V);
      let V=V−{x};
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }
```

The type of the nondeterminism monad is *'a nres*, where *'a* is the type of the results. Note that this program has only one possible result, however, the order in which we iterate over the elements of the set is unspecified.

This program uses the following statements provided by our framework: While-loops, bindings, return, and specification. We briefly explain the statements here. A complete reference can be found in Section 5.1.5.

A while-loop has the form $WHILE\ b\ f\ \sigma_0$, where b is the continuation condition, f is the loop body, and σ_0 is the initial state. In our case, the state used for the loop is a triple $(V,\ s,\ m)$, where V is the set of remaining elements, s is the sum of the elements seen so far, and m is the maximum of the elements seen so far. The $WHILE\ b\ f\ \sigma_0$ construct describes a partially correct loop, i.e., it describes only those results that can be reached by finitely many iterations, and ignores infinite paths of the loop. In order to prove total correctness, the construct $WHILE_T\ b\ f\ \sigma_0$ is used. It fails if there exists an infinite execution of the loop.

A binding $do\ \{x{\leftarrow}(S_1{::}'a\ nres);\ S_2\}$ nondeterministically chooses a result of S_1, binds it to variable x, and then continues with S_2. If S_1 is *FAIL*, the bind statement also fails.

The syntactic form $do\ \{\ let\ x{=}V;\ (S{::}'a \Rightarrow 'b\ nres)\}$ assigns the value V to variable x, and continues with S.

The return statement $RETURN\ x$ specifies precisely the result x.

The specification statement $SPEC\ \Phi$ describes all results that satisfy the predicate Φ. This is the source of nondeterminism in programs, as there may be more than one such result. In our case, we describe any element of set V.

Note that these statement are shallowly embedded into Isabelle/HOL, i.e.,

they are ordinary Isabelle/HOL constants. The main advantage is, that any other construct and datatype from Isabelle/HOL may be used inside programs. In our case, we use Isabelle/HOL's predefined operations on sets and natural numbers. Another advantage is that extending the framework with new commands becomes fairly easy.

Proving Programs Correct

The next step in the program development is to prove the program correct w.r.t. a specification. In refinement notion, we have to prove that the program S refines a specification Φ if the precondition Ψ holds, i.e., $\Psi \Longrightarrow S \leq SPEC\ \Phi$.
For our purposes, we prove that *sum-max* really computes the sum and the maximum.

As usual, we have to think of a loop invariant first. In our case, this is rather straightforward. The main complication is introduced by the partially defined *Max*-operator of the Isabelle/HOL standard library.

definition *sum-max-invar* $V_0 \equiv \lambda(V,s::nat,m).$
$V \subseteq V_0$
$\wedge\ s = \sum (V_0 - V)$
$\wedge\ m = (if\ (V_0 - V) = \{\}\ then\ 0\ else\ Max\ (V_0 - V))$
$\wedge\ finite\ (V_0 - V)$

We have extracted the most complex verification condition — that the invariant is preserved by the loop body — to an own lemma. For complex proofs, it is always a good idea to do that, as it makes the proof more readable.

lemma *sum-max-invar-step*:
assumes $x \in V$ *sum-max-invar* V_0 (V,s,m)
shows *sum-max-invar* V_0 $(V - \{x\}, s + x, max\ m\ x)$

In our case the proof is rather straightforward, it only requires the lemma *it-step-insert-iff*, that handles the $V_0 - (V - \{x\})$ terms that occur in the invariant.

⟨*proof*⟩

The correctness is now proved by first invoking the verification condition generator, and then discharging the verification conditions by *auto*. Note that we have to apply the *sum-max-invar-step* lemma, *before* we unfold the definition of the invariant to discharge the remaining verification conditions.

theorem *sum-max-correct*:
assumes *PRE*: $V \neq \{\}$
shows *sum-max* $V \leq SPEC\ (\lambda(s,m).\ s = \sum V \wedge m = Max\ V)$

The precondition $V \neq \{\}$ is necessary, as the *Max*-operator from Isabelle/HOL's standard library is not defined for empty sets.

⟨*proof*⟩

In this proof, we specified the invariant explicitely. Alternatively, we may annotate the invariant at the while loop, using the syntax $WHILE^I\ b\ f\ \sigma_0$. Then, the verification condition generator will use the annotated invariant automatically.

Total Correctness Now, we reformulate our program to use a total correct while loop, and annotate the invariant at the loop. The invariant is strengthened by stating that the set of elements is finite.

```
definition sum-max'-invar V₀ σ ≡
  sum-max-invar V₀ σ
  ∧ (let (V,-,-)=σ in finite (V₀−V))

definition sum-max' :: nat set ⇒ (nat×nat) nres where
  sum-max' V ≡ do {
    (-,s,m) ← WHILE_T^sum-max'-invar V (λ(V,s,m). V≠{}) (λ(V,s,m). do {
      x←SPEC (λx. x∈V);
      let V=V−{x};
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }


theorem sum-max'-correct:
  assumes NE: V≠{} and FIN: finite V
  shows sum-max' V ≤ SPEC (λ(s,m). s=∑ V ∧ m=Max V)
  ⟨proof⟩
```

Refinement

The next step in the program development is to refine the initial program towards an executable program. This usually involves both, program refinement and data refinement. Program refinement means changing the structure of the program. Usually, some specification statements are replaced by more concrete implementations. Data refinement means changing the used data types towards implementable data types.

In our example, we implement the set V with a distinct list, and replace the specification statement $SPEC\ (\lambda x.\ x \in V)$ by the head operation on distinct lists. For the lists, we use the list-set data structure provided by the Isabelle Collection Framework [2, 4].

For this example, we write the refined program ourselves. An automation of this task can be achieved with the automatic refinement tool, which is avail-

able as a prototype in Refine-Autoref. Usage examples are in ex/Automatic-Refinement.

```
definition sum-max-impl :: nat ls ⇒ (nat×nat) nres where
  sum-max-impl V ≡ do {
    (-,s,m) ← WHILE (λ(V,s,m). ¬ls.isEmpty V) (λ(V,s,m). do {
      x←RETURN (the (ls.sel V (λx. True)));
      let V=ls.delete x V;
      let s=s+x;
      let m=max m x;
      RETURN (V,s,m)
    }) (V,0,0);
    RETURN (s,m)
  }
```

Note that we replaced the operations on sets by the respective operations on lists (with the naming scheme *ls.xxx*). The specification statement was replaced by *the* (*ls.sel V* (λ*x*. *True*)), i.e., selection of an element that satisfies the predicate λ*x*. *True*. As *ls.sel* returns an option datatype, we extract the value with *the*. Moreover, we omitted the loop invariant, as we don't need it any more.

Next, we have to show that our concrete pogram actually refines the abstract one.

```
theorem sum-max-impl-refine:
  assumes (V,V')∈build-rel ls.α ls.invar
  shows sum-max-impl V ≤ ⇓Id (sum-max V')
```

Let R be a *refinement relation*[1], that relates concrete and abstract values.

Then, the function $\Downarrow R$ maps a result-set over abstract values to the greatest result-set over concrete values that is compatible w.r.t. R. The value *FAIL* is mapped to itself.

Thus, the proposition $S \leq \Downarrow R\ S'$ means, that S refines S' w.r.t. R, i.e., every value in the result of S can be abstracted to a value in the result of S'.

Usually, the refinement relation consists of an invariant I and an abstraction function α. In this case, we may use the *br* I α-function to define the refinement relation.

In our example, we assume that the input is in the refinement relation specified by list-sets, and show that the output is in the identity relation. We use the identity here, as we do not change the datatypes of the output.

The proof is done automatically by the refinement verification condition generator. Note that the theory *Collection-Bindings* sets up all the necessary lemmas to discharge refinement conditions for the collection framework.

⟨*proof*⟩

Refinement is transitive, so it is easy to show that the concrete program meets the specification.

[1] Also called coupling invariant.

theorem *sum-max-impl-correct*:
assumes $(V,V')\in$*build-rel ls.α ls.invar* **and** $V'\neq\{\}$
shows *sum-max-impl* $V \leq$ *SPEC* $(\lambda(s,m).\ s=\sum V' \wedge m=$*Max* $V')$
⟨*proof*⟩

Just for completeness, we also refine the total correct program in the same way.

definition *sum-max′-impl* :: *nat ls* ⇒ $(nat\times nat)$ *nres* **where**
sum-max′-impl $V \equiv$ *do* {
($-$,s,m) ← *WHILE*$_T$ $(\lambda(V,s,m).\ \neg$*ls.isEmpty* $V)$ $(\lambda(V,s,m).$ *do* {
x←*RETURN* (*the* (*ls.sel* V $(\lambda x.$ *True*)));
let V=*ls.delete* x V;
let s=s+x;
let m=*max* m x;
RETURN (V,s,m)
}) $(V,0,0)$;
RETURN (s,m)
}

theorem *sum-max′-impl-refine*:
$(V,V')\in$*build-rel ls.α ls.invar* ⟹ *sum-max′-impl* $V \leq \Downarrow$*Id* (*sum-max′* V')
⟨*proof*⟩

theorem *sum-max′-impl-correct*:
assumes $(V,V')\in$*build-rel ls.α ls.invar* **and** $V'\neq\{\}$
shows *sum-max′-impl* $V \leq$ *SPEC* $(\lambda(s,m).\ s=\sum V' \wedge m=$*Max* $V')$
⟨*proof*⟩

Code Generation

In order to generate code from the above definitions, we convert the function defined in our monad to an ordinary, deterministic function, for that the Isabelle/HOL code generator can generate code.

For partial correct algorithms, we can generate code inside a deterministic result monad. The domain of this monad is a flat complete lattice, where top means a failed assertion and bottom means nontermination. (Note that executing a function in this monad will never return bottom, but just diverge). The construct *nres-of* x embeds the deterministic into the nondeterministic monad.

Thus, we have to construct a function *?sum-max-code* such that:

schematic-goal *sum-max-code-aux*: *nres-of ?sum-max-code* ≤ *sum-max-impl* V

This is done automatically by the transfer procedure of our framework.

⟨*proof*⟩

In order to define the function from the above lemma, we can use the command *concrete-definition*, that is provided by our framework:

concrete-definition *sum-max-code* **for** V **uses** *sum-max-code-aux*

This defines a new constant *sum-max-code*:

thm *sum-max-code-def*

And proves the appropriate refinement lemma:

thm *sum-max-code.refine*

Note that the *concrete-definition* command is sensitive to patterns of the form *RETURN* - and *nres-of*, in which case the defined constant will not contain the *RETURN* or *nres-of*. In any other case, the defined constant will just be the left hand side of the refinement statement.

Finally, we can prove a correctness statement that is independent from our refinement framework:

theorem *sum-max-code-correct*:
assumes $ls.\alpha\ V \neq \{\}$
shows *sum-max-code* $V = dRETURN\ (s,m) \Longrightarrow s{=}\sum(ls.\alpha\ V) \wedge m{=}Max\ (ls.\alpha\ V)$
and *sum-max-code* $V \neq dFAIL$

The proof is done by transitivity, and unfolding some definitions:

⟨*proof*⟩

For total correctness, the approach is the same. The only difference is, that we use *RETURN* instead of *nres-of*:

schematic-goal *sum-max′-code-aux*:
RETURN ?sum-max′-code $\leq$ *sum-max′-impl* V
⟨*proof*⟩

concrete-definition *sum-max′-code* **for** V **uses** *sum-max′-code-aux*

theorem *sum-max′-code-correct*:
$[\![ls.\alpha\ V \neq \{\}]\!] \Longrightarrow$ *sum-max′-code* $V = (\sum(ls.\alpha\ V),\ Max\ (ls.\alpha\ V))$
⟨*proof*⟩

If we use recursion combinators, a plain function can only be generated, if the recursion combinators can be defined. Alternatively, for total correct programs, we may generate a (plain) function that internally uses the deterministic monad, and then extracts the result.

schematic-goal *sum-max″-code-aux*:
RETURN ?sum-max″-code $\leq$ *sum-max′-impl* V
⟨*proof*⟩

concrete-definition *sum-max″-code* **for** V **uses** *sum-max″-code-aux*

theorem *sum-max″-code-correct*:

$[\![ls.\alpha\ V \neq \{\}]\!] \Longrightarrow$ *sum-max''-code* $V = (\sum (ls.\alpha\ V),\ Max\ (ls.\alpha\ V))$
⟨*proof*⟩

Now, we can generate verified code with the Isabelle/HOL code generator:

export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *SML*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *OCaml?*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *Haskell?*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *Scala*

Foreach-Loops

In the *sum-max* example above, we used a while-loop to iterate over the elements of a set. As this pattern is used commonly, there is an abbreviation for it in the refinement framework. The construct *FOREACH S f* σ_0 iterates $f::'x \Rightarrow 's \Rightarrow 's$ for each element in $S::'x\ set$, starting with state $\sigma_0::'s$.

With foreach-loops, we could have written our example as follows:

definition *sum-max-it* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
sum-max-it V ≡ *FOREACH V* (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

theorem *sum-max-it-correct*:
assumes *PRE*: *V*≠{} **and** *FIN*: *finite V*
shows *sum-max-it V* ≤ *SPEC* (λ(*s*,*m*). *s*=∑ *V* ∧ *m*=*Max V*)
⟨*proof*⟩

definition *sum-max-it-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**
sum-max-it-impl V ≡ *FOREACH* (*ls.α V*) (λ*x* (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

Note: The nondeterminism for iterators is currently resolved at transfer phase, where they are replaced by iterators from the ICF.

lemma *sum-max-it-impl-refine*:
notes [*refine*] = *inj-on-id*
assumes (*V*, *V'*)∈*build-rel ls.α ls.invar*
shows *sum-max-it-impl V* ≤ ⇓*Id* (*sum-max-it V'*)
⟨*proof*⟩

schematic-goal *sum-max-it-code-aux*:
RETURN ?sum-max-it-code ≤ *sum-max-it-impl V*
⟨*proof*⟩

Note that the transfer method has replaced the iterator by an iterator from the Isabelle Collection Framework.

thm *sum-max-it-code-aux*
concrete-definition *sum-max-it-code* **for** *V* **uses** *sum-max-it-code-aux*

theorem *sum-max-it-code-correct*:
assumes *ls.α V* ≠ {}

shows *sum-max-it-code V* = (∑ (*ls.α V*), *Max* (*ls.α V*))
⟨*proof*⟩

export-code *sum-max-it-code* **checking** *SML*
export-code *sum-max-it-code* **checking** *OCaml?*
export-code *sum-max-it-code* **checking** *Haskell?*
export-code *sum-max-it-code* **checking** *Scala*

definition *sum-max-it-list* ≡ *sum-max-it-code o ls.from-list*
⟨*ML*⟩

5.1.3 Pointwise Reasoning

In this section, we describe how to use pointwise reasoning to prove refinement statements and other relations between element of the nondeterminism monad.

Pointwise reasoning is often a powerful tool to show refinement between structurally different program fragments.

The refinement framework defines the predicates *nofail* and *inres*. *nofail S* states that *S* does not fail, and *inres S x* states that one possible result of *S* is *x* (Note that this includes the case that *S* fails).

Equality and refinement can be stated using *nofail* and *inres*:

(*?S* = *?S'*) = (*nofail ?S* = *nofail ?S'* ∧ (∀ *x*. *inres ?S x* = *inres ?S' x*))

(*?S* ≤ *?S'*) = (*nofail ?S'* ⟶ *nofail ?S* ∧ (∀ *x*. *inres ?S x* ⟶ *inres ?S' x*))

Useful corollaries of this lemma are *pw-leI*, *pw-eqI*, and *pwD*.

Once a refinement has been expressed via nofail/inres, the simplifier can be used to propagate the nofail and inres predicates inwards over the structure of the program. The relevant lemmas are contained in the named theorem collection *refine-pw-simps*.

As an example, we show refinement of two structurally different programs here, both returning some value in a certain range:

lemma *do* { *ASSERT* (*fst p* > *2*); *SPEC* (λ*x*. *x*≤(*2*::*nat*)∗(*fst p* + *snd p*)) }
≤ *do* { *let* (*x*,*y*)=*p*; *z*←*SPEC* (λ*z*. *z*≤*x*+*y*);
a←*SPEC* (λ*a*. *a*≤*x*+*y*); *ASSERT* (*x*>*2*); *RETURN* (*a*+*z*)}
⟨*proof*⟩

5.1.4 Arbitrary Recursion (TBD)

While-loops are suited to express tail-recursion. In order to express arbitrary recursion, the refinement framework provides the nrec-mode for the *partial-function* command, as well as the fixed point combinators *REC* (partial correctness) and REC_T (total correctness).

Examples for *partial-function* can be found in *ex/Refine-Fold*. Examples for the recursion combinators can be found in *ex/Recursion* and *ex/Nested-DFS*.

5.1.5 Reference

Statements

SUCCEED The empty set of results. Least element of the refinement ordering.

FAIL Result that indicates a failing assertion. Greatest element of the refinement ordering.

RES X All results from set *X*.

RETURN x Return single result *x*. Defined in terms of *RES*: *RETURN x* = *RES* $\{x\}$.

EMBED r Embed partial-correctness option type, i.e., succeed if *r*=*None*, otherwise return value of *r*.

SPEC Φ Specification. All results that satisfy predicate Φ. Defined in terms of *RES*: *SPEC* Φ = *SPEC* Φ

bind M f Binding. Nondeterministically choose a result from *M* and apply *f* to it. Note that usually the *do*-notation is used, i.e., *do* $\{x \leftarrow M; f\, x\}$ or *do* $\{M; f\}$ if the result of *M* is not important. If *M* fails, *bind M f* also fails.

ASSERT Φ Assertion. Fails if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSERT\ \Phi; f\}$.

ASSUME Φ Assumption. Succeeds if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSUME\ \Phi; f\}$.

REC body Recursion for partial correctness. May be used to express arbitrary recursion. Returns *SUCCEED* on nontermination.

REC_T *body* Recursion for total correctness. Returns *FAIL* on nontermination.

WHILE b f σ_0 Partial correct while-loop. Start with state σ_0, and repeatedly apply *f* as long as *b* holds for the current state. Non-terminating paths are ignored, i.e., they do not contribute a result.

$WHILE_T$ *b f* σ_0 Total correct while-loop. If there is a non-terminating path, the result is *FAIL*.

WHILE^I b f σ_0, *WHILE*$_T{}^I$ b f σ_0 While-loop with annotated invariant. It is asserted that the invariant holds.

FOREACH S f σ_0 Foreach loop. Start with state σ_0, and transform the state with $f\,x$ for each element $x{\in}S$. Asserts that S is finite.

FOREACH^I S f σ_0 Foreach-loop with annotated invariant.

Alternative syntax: *FOREACH*^{I} S f σ_0.

The invariant is a predicate of type $I{::}'a\ set \Rightarrow 'b \Rightarrow bool$, where $I\ it\ \sigma$ means, that the invariant holds for the remaining set of elements it and current state σ.

FOREACH$_C$ S c f σ_0 Foreach-loop with explicit continuation condition.

Alternative syntax: *FOREACH*$_C$ S c f σ_0.

If $c{::}'\sigma{\Rightarrow}bool$ becomes false for the current state, the iteration immediately terminates.

FOREACH$_C{}^I$ S c f σ_0 Foreach-loop with explicit continuation condition and annotated invariant.

Alternative syntax: *FOREACH*$_C{}^I$ S c f σ_0.

partial-function (*nrec*) Mode of the partial function package for the nondeterminism monad.

Refinement

($\leq$) Refinement ordering. $S \leq S'$ means, that every result in S is also a result in S'. Moreover, S may only fail if S' fails. $\leq$ forms a complete lattice, with least element *SUCCEED* and greatest element *FAIL*.

$\Downarrow R$ Concretization. Takes a refinement relation $R{::}('c{\times}'a)\ set$ that relates concrete to abstract values, and returns a concretization function $\Downarrow R$.

$\Uparrow R$ Abstraction. Takes a refinement relation and returns an abstraction function. The functions $\Downarrow R$ and $\Uparrow R$ form a Galois-connection, i.e., we have: $S \leq \Downarrow R\ S' \longleftrightarrow \Uparrow R\ S \leq S'$.

br α I Builds a refinement relation from an abstraction function and an invariant. Those refinement relations are always single-valued.

nofail S Predicate that states that S does not fail.

inres S x Predicate that states that S includes result x. Note that a failing program includes all results.

Proof Tools

Verification Condition Generator:

Method: *intro refine-vcg*

Attributes: *refine-vcg*

Transforms a subgoal of the form $S \leq SPEC\ \Phi$ into verification conditions by decomposing the structure of S. Invariants for loops without annotation must be specified explicitely by instantiating the respective proof-rule for the loop construct, e.g., *intro WHILE-rule*[*where* $I{=}\ldots$] *refine-vcg*.

refine-vcg is a named theorems collection that contains the rules that are used by default.

Refinement Condition Generator:

Method: *refine-rcg* [thms].

Attributes: *refine0*, *refine*, *refine2*.

Flags: *refine-no-prod-split*.

Tries to prove a subgoal of the form $S \leq \Downarrow R\ S'$ by decomposing the structure of S and S'. The rules to be used are contained in the theorem collection *refine*. More rules may be passed as argument to the method. Rules contained in *refine0* are always tried first, and rules in *refine2* are tried last. Usually, rules that decompose both programs equally should be put into *refine*. Rules that may make big steps, without decomposing the program further, should be put into *refine0* (e.g., *Id-refine*). Rules that decompose the programs differently and shall be used as last resort before giving up should be put into *refine2*, e.g., *remove-Let-refine*.

By default, this procedure will invoke the splitter to split product types in the goals. This behaviour can be disabled by setting the flag *refine-no-prod-split*.

Refinement Relation Heuristics:

Method: *refine-dref-type* [(trace)].

Attributes: *refine-dref-RELATES*, *refine-dref-pattern*.

Flags: *refine-dref-tracing*.

Tries to instantiate schematic refinement relations based on their type. By default, this rule is applied to all subgoals. Internally, it uses the rules declared as *refine-dref-pattern* to introduce a goal of the form

RELATES ?R, that is then solved by exhaustively applying rules declared as *refine-dref-RELATES*.

The flag *refine-dref-tracing* controls tracing of resolving *RELATES*-goals. Tracing may also be enabled by passing (trace) as argument.

Pointwise Reasoning Simplification Rules:

Attributes: *refine-pw-simps*

A theorem collection that contains simplification lemmas to push inwards *nofail* and *inres* predicates into program constructs.

Refinement Simp Rules:

Attributes: *refine-hsimp*

A theorem collection that contains some simplification lemmas that are useful to prove membership in refinement relations.

Transfer:

Method: *refine-transfer* [thms]

Attribute: *refine-transfer*

Tries to prove a subgoal of the form $\alpha\ f \leq S$ by decomposing the structure of f and S. This is usually used in connection with a schematic lemma, to generate f from the structure of S.

The theorems declared as *refine-transfer* are used to do the transfer. More theorems may be passed as arguments to the method. Moreover, some simplification for nested abstraction over product types ($\lambda(a,b)\ (c,d). \ldots$) is done, and the monotonicity prover is used on monotonicity goals.

There is a standard setup for α=*RETURN* (transfer to plain function for total correct code generation), and α=*nres-of* (transfer to deterministic result monad, for partial correct code generation).

Automatic Refinement:

Method: *refine-autoref*

Attributes: ...

See automatic refinement package for documentation (TBD)

Concrete Definition:

Command: *concrete-definition name* [*attribs*] *for params uses thm* where *attribs* and the *for*-part are optional.

Declares a new constant from the left-hand side of a refinement lemma. Has special handling for left-hand sides of the forms *RETURN* - and *nres-of*, in which cases those topmost functions are not included in the defined constant.

The refinement lemma is folded with the new constant and registered as *name.refine*.

Command: *prepare-code-thms thms* takes a list of definitional theorems and sets up lemmas for the code generator for those definitions. This includes handling of recursion combinators.

Packages

The following parts of the refinement framework are not included by default, but can be imported if necessary:

Collection-Bindings: Sets up refinement rules for the Isabelle Collection Framework. With this theory loaded, the refinement condition generator will discharge most data refinements using the ICF automatically. Moreover, the transfer procedure will replace *FOREACH*-statements by the corresponding ICF-iterators.

end

5.2 Isabelle Collections Framework Userguide

5.2.1 Introduction

This is the Userguide for the (old) Isabelle Collection Framework. It does not cover the Generic Collection Framework, nor the Automatic Refinement Framework.

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.

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. More examples are in the examples/ subdirectory of the collection framework. Section 5.2.2 explains the concepts of the Isabelle Collections Framework in more detail. Section 5.2.3 provides information on extending the framework along with detailed examples, and Section 5.2.4 contains a discussion on the design of this framework. There is also a paper [3] on the design of the Isabelle Collections Framework available.

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[2]:

definition *rbt-restrict-list* :: *'a::linorder rs* ⇒ *'a list* ⇒ *'a list*
where *rbt-restrict-list s l* == [*x*←*l*. *rs.memb x s*]

The type *'a rs* is the type of sets backed by red-black trees. Note that the element type of sets backed by red-black trees must be of sort *linorder*. The function *rs.memb* tests membership on such sets.

Next, we show correctness of our function:

lemma *rbt-restrict-list-correct*:
assumes [*simp*]: *rs.invar s*
shows *rbt-restrict-list s l* = [*x*←*l*. *x*∈*rs.α s*]
⟨*proof*⟩

The lemma *rs.memb-correct*:

[2]Note that Isabelle/HOL uses the list comprehension syntax [*x*←*l*. *P x*] as syntactic sugar for filtering a list.

$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$

states correctness of the *rs.memb*-function. The function $rs.\alpha$ 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 *rs.invar ?s* represents the invariant assumption for the collection data structure. Red-black-trees are invariant-free, so this defaults 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*:

$rs.\alpha\ (rs.empty\ ()) = \{\}$
$rs.invar\ (rs.empty\ ())$
$rs.\alpha\ (rs.sng\ x) = \{x\}$
$rs.invar\ (rs.sng\ x)$
$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.ins\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.ins\ x\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.\alpha\ (rs.ins\text{-}dj\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.invar\ (rs.ins\text{-}dj\ x\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.delete\ x\ s) = rs.\alpha\ s - \{x\}$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.delete\ x\ s)$
$rs.invar\ s \Longrightarrow rs.isEmpty\ s = (rs.\alpha\ s = \{\})$
$rs.invar\ s \Longrightarrow rs.isSng\ s = (\exists\ e.\ rs.\alpha\ s = \{e\})$
$rs.invar\ S \Longrightarrow rs.ball\ S\ P = (\forall\ x{\in}rs.\alpha\ S.\ P\ x)$
$rs.invar\ S \Longrightarrow rs.bex\ S\ P = (\exists\ x{\in}rs.\alpha\ S.\ P\ x)$
$rs.invar\ s \Longrightarrow rs.size\ s = card\ (rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.size\text{-}abort\ m\ s = min\ m\ (card\ (rs.\alpha\ s))$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.union\ s1\ s2) = rs.\alpha\ s1 \cup rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.union\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow rs.\alpha\ (rs.union\text{-}dj\ s1\ s2) = rs.\alpha\ s1 \cup rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow rs.invar\ (rs.union\text{-}dj\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.diff\ s1\ s2) = rs.\alpha\ s1 - rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.diff\ s1\ s2)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.filter\ P\ s) = \{e \in rs.\alpha\ s.\ P\ e\}$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.filter\ P\ s)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.inter\ s1\ s2) = rs.\alpha\ s1 \cap rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.inter\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.subset\ s1\ s2 = (rs.\alpha\ s1 \subseteq rs.\alpha\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.equal\ s1\ s2 = (rs.\alpha\ s1 = rs.\alpha\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.disjoint\ s1\ s2 = (rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\})$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.disjoint\text{-}witness\ s1\ s2 = None]\!]$
$\Longrightarrow rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.disjoint\text{-}witness\ s1\ s2 = Some\ a]\!]$
$\Longrightarrow a \in rs.\alpha\ s1 \cap rs.\alpha\ s2$
$rs.invar\ s \Longrightarrow set\ (rs.to\text{-}list\ s) = rs.\alpha\ s$

rs.invar s $\Longrightarrow$ *distinct* (*rs.to-list s*)
rs.α (*rs.from-list l*) = *set l*
rs.invar (*rs.from-list l*)

All implementations provided by this library are compatible with the Isabelle/HOL code-generator. Now follow some examples of using the code-generator. Note that the code generator can only generate code for plain constants without arguments, while the operations like *rs.memb* have arguments, that are only hidden by an abbreviation.

There are conversion functions from lists to sets and, vice-versa, from sets to lists:

definition *conv-tests* ≡ (
  rs.from-list [*1*::*int* .. *10*],
  rs.to-list (*rs.from-list* [*1*::*int* .. *10*]),
  rs.to-sorted-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*]),
  rs.to-rev-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*])
)

⟨*ML*⟩

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*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-list s*)

Some sets, like red-black-trees, also support conversion to sorted lists, and we have: *rs.to-sorted-list-correct*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-sorted-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-sorted-list s*)
rs.invar s $\Longrightarrow$ *sorted* (*rs.to-sorted-list s*)

and *rs.to-rev-list-correct*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-rev-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-rev-list s*)
rs.invar s $\Longrightarrow$ *sorted* (*rev* (*rs.to-rev-list s*))

definition *restrict-list-test* ≡ *rbt-restrict-list* (*rs.from-list* [*1*::*nat*,*2*,*3*,*4*,*5*]) [*1*::*nat*,*9*,*2*,*3*,*4*,*5*,*6*,*5*,*4*,*3*,*6*,*7*,*8*,*9*]

⟨*ML*⟩

definition *big-test n* = (*rs.from-list* [(*1*::*int*)..*n*])

⟨*ML*⟩

Theories

To make available the whole collections framework to your formalization, import the theory *Collections.Collections* which includes everything. Here is a small selection:

Collections.SetSpec Specification of sets and set functions

Collections.SetGA Generic algorithms for sets

Collections.SetStdImpl Standard set implementations (list, rb-tree, hashing, tries)

Collections.MapSpec Specification of maps

Collections.MapGA Generic algorithms for maps

Collections.MapStdImpl Standard map implementations (list,rb-tree, hashing, tries)

Collections.ListSpec Specification of lists

Collections.Fifo Amortized fifo queue

Collections.DatRef Data refinement for the while combinator

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.iterate*, and the proof-rule that is usually used is: *rs.iteratei-rule-P*:

$$\begin{array}{l}[\![rs.invar\ S;\ I\ (rs.\alpha\ S)\ \sigma 0;\\ \bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);\\ \bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge \sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!]\\ \Longrightarrow P\ (rs.iteratei\ S\ c\ f\ \sigma 0)\end{array}$$

The invariant I is parameterized with the set of remaining elements that have not yet been iterated over and the current state. The invariant has to hold for all elements remaining and the initial state: $I\ (rs.\alpha\ S)\ \sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

$\bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$

And the proposition to be shown for the final state must be a consequence of the invarant for no elements remaining: $\bigwedge\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$.

A generalization of iterators are *interruptible iterators* where iteration is only continues while some condition on the state holds. Reasoning over interruptible iterators is also done by invariants: *rs.iteratei-rule-P*:

$[\![rs.invar\ S;\ I\ (rs.\alpha\ S)\ \sigma 0;$
$\bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);$
$\bigwedge\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge\sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!]$
$\Longrightarrow P\ (rs.iteratei\ S\ c\ f\ \sigma 0)$

Here, interruption of the iteration is handled by the premise

$\bigwedge\sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma$

that shows the proposition from the invariant for any intermediate state of the iteration where the continuation condition does not hold (and thus the iteration is interrupted).

As an example of reasoning about results of iterators, we implement a function that converts a hashset to a list that contains precisely the elements of the set.

definition *hs-to-list′ s* == *hs.iteratei s* (λ-. *True*) (#) []

The correctness proof works by establishing the invariant that the list contains all elements that have already been iterated over. Again *hs.invar s* denotes the invariant for hashsets which defaults to *True*.

lemma *hs-to-list′-correct*:
assumes *INV*: *hs.invar s*
shows *set* (*hs-to-list′ s*) = *hs.α s*
⟨*proof*⟩

As an example for an interruptible iterator, we define a bounded existential-quantification over the list elements. As soon as the first element is found that fulfills the predicate, the iteration is interrupted. The state of the iteration is simply a boolean, indicating the (current) result of the quantification:

definition *hs-bex s P* == *hs.iteratei s* (λσ. ¬ σ) (λx σ. *P x*) *False*

lemma *hs-bex-correct*:
hs.invar s ⟹ *hs-bex s P* ⟷ (∃ *x*∈*hs.α s*. *P x*)
⟨*proof*⟩

5.2.2 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*.

Operation Records In order to reference an interface with a standard set of operations, those operations are summarized in a record, and there is a locale that fixes this record, and makes available all operations. For example, the locale *SetSpec.StdSet* fixes a record of standard set operations and assumes their correctness. It also defines abbreviations to easily access the members of the record. Internally, all the standard operations, like *hs.memb*, are introduced by interpretation of such an operation locale.

Generic Algorithms A generic algorithm specifies, in a generic way, how to implement a function using other functions. Usually, a generic algorithm lives in a locale that imports the necessary operation locales. For example, the locale *cart-loc* defines a generic algorithm for the cartesian product between two sets.

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 [5] 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 $rs.\alpha$ and invariant *rs.invar*; and the constant *rs.ins* implements the insert-function, as can be verified by *set-ins rs.*α *rs.invar rs.ins*. 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 cartesian-product algorithm can be instantiated to use red-black-trees for both arguments, and output a list, as will be illustrated below in Section 5.2.2. 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 by the compiler.

Instantiation of Generic Algorithms

A generic algorithm is instantiated by interpreting its locale with the wanted implementations. For example, to obtain a cartesian product between two red-black trees, yielding a list, we can do the following:

⟨*ML*⟩
interpretation *rrl*: *cart-loc rs-ops rs-ops ls-ops* ⟨*proof*⟩
⟨*ML*⟩

It is then available under the expected name:

term *rrl.cart*

Note the three lines of boilerplate code, that work around some technical problems of Isabelle/HOL: The *Locale-Code.open-block* and *Locale-Code.close-block* commands set up code generation for any locale that is interpreted in between them. They also have to be specified if an existing locale that already has interpretations is extended by new definitions.

The *ICF-Tools.revert-abbrevs rrl* reverts all abbreviations introduced by the locale, such that the displayed information becomes nicer.

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. *rrl.cart* for the cartesian product between two RBT-Sets, yielding a list-set)
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*.

5.2.3 Extending the Framework

The best way to add new features, i.e., interfaces, functions, generic algorithms, or implementations to the collection framework is to use one of the existing items as example.

5.2.4 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.

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 looses 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.

For a more detailed discussion of the data refinement issue, we refer to the monadic refinement framework, that is available in the AFP (http://isa-afp.org/entries/Refine_Monadic.shtml)

Operation Records

In order to allow convenient access to the most frequently used functions of an interface, we have grouped them together in a record, and defined a locale that only fixes this record. This greatly reduces the boilerplate required to define a new (generic) algorithm, as only the operation locale (instead of every single function) has to be included in the locale for the generic algorithm.

Note however, that parameters of locales are monomorphic inside the locale. Thus, we have to import an own instance for the locale for every element type of a set, or key/value type of a map. For iterators, where this problem was most annoying, we have installed a workaround that allows polymorphic iterators even inside locales.

Locales for Generic Algorithms

A generic algorithm is defined within a locale, that includes the required functions (or operation locales). If many instances of the same interface are required, prefixes are used to distinguish between them. This makes the code for a generic algorithm quite consise and readable.

However, there are some technical issues that one has to consider:

- When fixing 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.
- The code generator has problems with generating code from definitions inside a locale. Currently, the *Locale-Code*-package provides a rather convenient workaround for that issue: It requires the user to enclose interpretations and definitions of new constants inside already interpreted locales within two special commands, that set up the code generator appropriately.

Explicit Invariants vs Typedef

The interfaces of this framework use explicit invariants. This provides a more general specification which allows some operations to be implemented more efficiently, cf. *lsi.ins-dj* in *Collections.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*, and removed automatically by the simplifier and classical reasoner. However, it still shows up in some premises and conclusions due to uniformity reasons.

Chapter 6

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.

6.1 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.

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.

6.2 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.

[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

Bibliography

[1] Java: The collections framework. Available on: http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html.

[2] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/collections.shtml, Dec. 2009. Formal proof development.

[3] 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.

[4] 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.

[5] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, November 1995.