

Isabelle Collections Framework

By Peter Lammich and Andreas Lochbihler

April 20, 2020

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

1	Introduction	7
2	The Generic Collection Framework	9
2.1	Interfaces	9
2.1.1	Map Interface	9
2.1.2	Set Interface	12
2.1.3	Hashable Interface	15
2.1.4	Orderings By Comparison Operator	17
2.2	Generic Algorithms	25
2.2.1	Generic Set Algorithms	25
2.2.2	Generic Map Algorithms	34
2.2.3	Generic Map To Set Converter	38
2.2.4	Generic Compare Algorithms	41
2.3	Implementations	41
2.3.1	Stack by Array	41
2.3.2	List Based Sets	44
2.3.3	List Based Maps	48
2.3.4	Array Based Hash-Maps	54
2.3.5	Red-Black Tree based Maps	68
2.3.6	Set by Characteristic Function	79
2.3.7	Array-Based Maps with Natural Number Keys	80
3	The Original Isabelle Collection Framework	87
3.1	Specifications	87
3.1.1	Specification of Sets	87
3.1.2	Specification of Sequences	103
3.1.3	Specification of Annotated Lists	108
3.1.4	Specification of Priority Queues	112
3.1.5	Specification of Unique Priority Queues	115
3.2	Generic Algorithms	117
3.2.1	General Algorithms for Iterators over Finite Sets	117
3.2.2	Generic Algorithms for Maps	120
3.2.3	Generic Algorithms for Sets	126

3.2.4	Implementing Sets by Maps	141
3.2.5	Generic Algorithms for Sequences	144
3.2.6	Indices of Sets	146
3.2.7	More Generic Algorithms	148
3.2.8	Implementing Priority Queues by Annotated Lists . .	151
3.2.9	Implementing Unique Priority Queues by Annotated Lists	157
3.3	Implementations	165
3.3.1	Map Implementation by Associative Lists	165
3.3.2	Map Implementation by Association Lists with ex- plicit invariants	166
3.3.3	Map Implementation by Red-Black-Trees	168
3.3.4	Hash maps implementation	170
3.3.5	Hash Maps	176
3.3.6	Implementation of a trie with explicit invariants . . .	179
3.3.7	Tries without invariants	181
3.3.8	Map implementation via tries	182
3.3.9	Array-based hash map implementation	184
3.3.10	Array-based hash maps without explicit invariants . .	190
3.3.11	Maps from Naturals by Arrays	193
3.3.12	Standard Implementations of Maps	198
3.3.13	Set Implementation by List	199
3.3.14	Set Implementation by List with explicit invariants . .	200
3.3.15	Set Implementation by non-distinct Lists	202
3.3.16	Set Implementation by sorted Lists	205
3.3.17	Set Implementation by Red-Black-Tree	210
3.3.18	Hash Set	211
3.3.19	Set implementation via tries	213
3.3.20	Set Implementation by Arrays	216
3.3.21	Standard Set Implementations	217
3.3.22	Fifo Queue by Pair of Lists	218
3.3.23	Implementation of Priority Queues by Binomial Heap	221
3.3.24	Implementation of Priority Queues by Skew Binomial Heaps	223
3.3.25	Implementation of Annotated Lists by 2-3 Finger Trees	225
3.3.26	Implementation of Priority Queues by Finger Trees . .	228
3.3.27	Implementation of Unique Priority Queues by Finger Trees	229
3.4	Entry Points	229
3.4.1	Standard Collections	229
3.4.2	Backwards Compatibility for Version 1	230

4	Entry Points	239
4.1	Entry Points	239
4.1.1	Default Setup	239
4.1.2	Entry Point with genCF and original ICF	240
4.1.3	Entry Point with only the ICF	241
5	Userguides	243
5.1	Old Monadic Refinement Framework Userguide	243
5.1.1	Introduction	243
5.1.2	Guided Tour	243
5.1.3	Pointwise Reasoning	251
5.1.4	Arbitrary Recursion (TBD)	251
5.1.5	Reference	252
5.2	Isabelle Collections Framework Userguide	256
5.2.1	Introduction	256
5.2.2	Structure of the Framework	261
5.2.3	Extending the Framework	264
5.2.4	Design Issues	265
6	Conclusion	269
6.1	Trusted Code Base	269
6.2	Acknowledgement	270

Chapter 1

Introduction

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

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

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

For more documentation and introductory material refer to the userguide (Section 5.2) and the ITP-2010 paper [2].

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* \Rightarrow *interface* \Rightarrow *interface*

definition [*simp*]: *op-map-empty* \equiv *Map.empty*

definition *op-map-lookup* :: *'k* \Rightarrow (*'k* \rightarrow *'v*) \rightarrow *'v*

where [*simp*]: *op-map-lookup* *k m* \equiv *m k*

definition [*simp*]: *op-map-update* *k v m* \equiv *m*(*k* \mapsto *v*)

definition [*simp*]: *op-map-delete* *k m* \equiv *m* |' ($-\{k\}$)

definition [*simp*]: *op-map-restrict* *P m* \equiv *m* |' {*k* \in *dom m*. *P* (*k*, *the* (*m k*))}

definition [*simp*]: *op-map-isEmpty* *x* \equiv *x*=*Map.empty*

definition [*simp*]: *op-map-isSng* *x* \equiv \exists *k v*. *x*=*[k* \mapsto *v]*

definition [*simp*]: *op-map-ball* *m P* \equiv *Ball* (*map-to-set* *m*) *P*

definition [*simp*]: *op-map-bex* *m P* \equiv *Bex* (*map-to-set* *m*) *P*

definition [*simp*]: *op-map-size* *m* \equiv *card* (*dom m*)

definition [*simp*]: *op-map-size-abort* *n m* \equiv *min* *n* (*card* (*dom m*))

definition [*simp*]: *op-map-sel* *m P* \equiv *SPEC* ($\lambda(k,v)$. *m k* = *Some v* \wedge *P k v*)

definition [*simp*]: *op-map-pick* *m* \equiv *SPEC* ($\lambda(k,v)$. *m k* = *Some v*)

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

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

context begin interpretation *autoref-syn* *(proof)*

lemma [*autoref-op-pat*]:

$Map.empty \equiv op-map-empty$
 $(m::'k \rightarrow 'v) k \equiv op-map-lookup\$k\$m$
 $m(k \mapsto v) \equiv op-map-update\$k\$v\m
 $m |' (-\{k\}) \equiv op-map-delete\$k\$m$
 $m |' \{k \in dom\ m.\ P\ (k,\ the\ (m\ k))\} \equiv op-map-restrict\$P\$m$

$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

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

$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-to-set\ m) \equiv op-map-pick\m
(proof)

lemma [*autoref-op-pat*]:

$SPEC (\lambda((k,v),m'). m k = Some v \wedge m' = m |' (-\{k\}))$
 $\equiv op-map-pick-remove\m
(proof)

lemma *op-map-pick-remove-alt*:

$do\ \{(k,v),m\} \leftarrow op-map-pick-remove\ m;\ f\ k\ v\ m\}$
 $=$ (
 $do\ \{$
 $(k,v) \leftarrow SPEC (\lambda(k,v).\ m\ k = Some\ v);$
 $let\ m = m |' (-\{k\});$

```

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

op-map-sel :: ⟨Ik,Iv⟩i i-map → (Ik → Iv → i-bool)
  → ⟨⟨Ik,Iv⟩i i-prod⟩i i-nres
op-map-pick :: ⟨Ik,Iv⟩i i-map → ⟨⟨Ik,Iv⟩i i-prod⟩i i-nres
op-map-pick-remove
  :: ⟨Ik,Iv⟩i i-map → ⟨⟨⟨Ik,Iv⟩i i-prod, ⟨Ik,Iv⟩i i-map⟩i i-prod⟩i i-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 ( $\langle\langle Rk, Rv \rangle prod-rel \rightarrow Id \rangle \rightarrow \langle Rk, Rv \rangle Rm \rightarrow \langle Rk, Rv \rangle Rm$ )
CONSTRAINT op-map-isEmpty ( $\langle Rk, Rv \rangle Rm \rightarrow Id$ )
CONSTRAINT op-map-isSng ( $\langle Rk, Rv \rangle Rm \rightarrow Id$ )
CONSTRAINT op-map-ball ( $\langle Rk, Rv \rangle Rm \rightarrow (\langle Rk, Rv \rangle prod-rel \rightarrow Id) \rightarrow Id$ )
CONSTRAINT op-map-bex ( $\langle Rk, Rv \rangle Rm \rightarrow (\langle Rk, Rv \rangle prod-rel \rightarrow Id) \rightarrow Id$ )
CONSTRAINT op-map-size ( $\langle Rk, Rv \rangle Rm \rightarrow Id$ )
CONSTRAINT op-map-size-abort ( $Id \rightarrow \langle Rk, Rv \rangle Rm \rightarrow Id$ )

CONSTRAINT op-map-sel ( $\langle Rk, Rv \rangle Rm \rightarrow (Rk \rightarrow Rv \rightarrow bool-rel) \rightarrow \langle Rk \times_r Rv \rangle nres-rel$ )
CONSTRAINT op-map-pick ( $\langle Rk, Rv \rangle Rm \rightarrow \langle Rk \times_r Rv \rangle nres-rel$ )
CONSTRAINT op-map-pick-remove ( $\langle Rk, Rv \rangle Rm \rightarrow \langle (Rk \times_r Rv) \times_r \langle Rk, Rv \rangle Rm \rangle nres-rel$ )
 $\langle proof \rangle$ 

```

definition *finite-map-rel* $R \equiv Range\ R \subseteq Collect\ (finite \circ dom)$

lemma *finite-map-rel-trigger*: $finite-map-rel\ R \implies finite-map-rel\ R\ \langle proof \rangle$

$\langle ML \rangle$

end

2.1.2 Set Interface

theory *Intf-Set*

imports *Refine-Monadic.Refine-Monadic*

begin

consts *i-set* :: *interface* \Rightarrow *interface*

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

definition [*simp*]: *op-set-delete* $x\ s \equiv s - \{x\}$

definition [*simp*]: *op-set-isEmpty* $s \equiv s = \{\}$

definition [*simp*]: *op-set-isSng* $s \equiv card\ s = 1$

definition [*simp*]: *op-set-size-abort* $m\ s \equiv min\ m\ (card\ s)$

definition [*simp*]: *op-set-disjoint* $a\ b \equiv a \cap b = \{\}$

definition [*simp*]: *op-set-filter* $P\ s \equiv \{x \in s. P\ x\}$

definition [*simp*]: *op-set-sel* $P\ s \equiv SPEC\ (\lambda x. x \in s \wedge P\ x)$

definition [*simp*]: *op-set-pick* $s \equiv SPEC\ (\lambda x. x \in s)$

definition [*simp*]: *op-set-to-sorted-list* $ordR\ s$

$\equiv SPEC\ (\lambda l. set\ l = s \wedge distinct\ l \wedge sorted-wrt\ ordR\ l)$

definition [*simp*]: *op-set-to-list* $s \equiv SPEC\ (\lambda l. set\ l = s \wedge distinct\ l)$

definition [*simp*]: *op-set-cart* $x\ y \equiv x \times y$

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma [*autoref-op-pat*]:

fixes $s\ a\ b :: 'a\ set$ and $x :: 'a$ and $P :: 'a \Rightarrow bool$

shows

$$s - \{x\} \equiv \text{op-set-delete } x \ \$s$$

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

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

$$\text{card } s = 1 \equiv \text{op-set-isSng } \$s$$

$$\exists x. s = \{x\} \equiv \text{op-set-isSng } \$s$$

$$\exists x. \{x\} = s \equiv \text{op-set-isSng } \$s$$

$$\min m (\text{card } s) \equiv \text{op-set-size-abort } \$m \ \$s$$

$$\min (\text{card } s) m \equiv \text{op-set-size-abort } \$m \ \$s$$

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

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

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

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

$$\text{SPEC } (\lambda x. x \in s) \equiv \text{op-set-pick } \$s$$

(proof)

lemma [*autoref-op-pat*]:

$$a \times b \equiv \text{op-set-cart } a \ b$$

(proof)

lemma [*autoref-op-pat*]:

$$\text{SPEC } (\lambda (u,v). (u,v) \in s) \equiv \text{op-set-pick } \$s$$

$$\text{SPEC } (\lambda (u,v). P \ u \ v \wedge (u,v) \in s) \equiv \text{op-set-sel } (\text{case-prod } P) \ \$s$$

$$\text{SPEC } (\lambda (u,v). (u,v) \in s \wedge P \ u \ v) \equiv \text{op-set-sel } (\text{case-prod } P) \ \$s$$

(proof)

lemma [*autoref-op-pat*]:

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

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

$$\equiv \text{OP } (\text{op-set-to-sorted-list } \text{ordR}) \ \$s$$

$SPEC (\lambda l. s = set\ l \wedge sorted\text{-}wrt\ ordR\ l \wedge distinct\ l)$
 $\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
 $\langle proof \rangle$

end

lemma $[autoref\text{-}itype]$:

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

lemma $hom\text{-}set1 [autoref\text{-}hom]$:

```

CONSTRAINT {} ( $\langle R \rangle Rs$ )
CONSTRAINT insert ( $R \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT ( $\in$ ) ( $R \rightarrow \langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT ( $\cup$ ) ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT ( $\cap$ ) ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT ( $-$ ) ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT ( $=$ ) ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT ( $\subseteq$ ) ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT Ball ( $\langle R \rangle Rs \rightarrow (R \rightarrow Id) \rightarrow Id$ )
CONSTRAINT Bex ( $\langle R \rangle Rs \rightarrow (R \rightarrow Id) \rightarrow Id$ )
CONSTRAINT card ( $\langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT set ( $\langle R \rangle Rl \rightarrow \langle R \rangle Rs$ )
CONSTRAINT ( $\cdot$ ) ( $(Ra \rightarrow Rb) \rightarrow \langle Ra \rangle Rs \rightarrow \langle Rb \rangle Rs$ )
CONSTRAINT Union ( $(\langle R \rangle Ri) Ro \rightarrow \langle R \rangle Ri$ )
<proof>

```

lemma *hom-set2*[*autoref-hom*]:

```

CONSTRAINT op-set-delete ( $R \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT op-set-isEmpty ( $\langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT op-set-isSng ( $\langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT op-set-size-abort ( $Id \rightarrow \langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT op-set-disjoint ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow Id$ )
CONSTRAINT op-set-filter ( $(R \rightarrow Id) \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$ )
CONSTRAINT op-set-sel ( $(R \rightarrow Id) \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rn$ )
CONSTRAINT op-set-pick ( $\langle R \rangle Rs \rightarrow \langle R \rangle Rn$ )
<proof>

```

lemma *hom-set-Sigma*[*autoref-hom*]:

```

CONSTRAINT Sigma ( $\langle Ra \rangle Rs \rightarrow (Ra \rightarrow \langle Rb \rangle Rs) \rightarrow \langle \langle Ra, Rb \rangle prod-rel \rangle Rs2$ )
<proof>

```

definition *finite-set-rel* $R \equiv Range\ R \subseteq Collect\ (finite)$

lemma *finite-set-rel-trigger*: *finite-set-rel* $R \implies finite-set-rel\ R$ <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*]:
 $\llbracket \text{STRUCT-EQ-tag } eq \text{ (=)}; \text{REL-FORCE-ID } R \rrbracket \implies \text{is-bounded-hashcode } R \text{ eq}$
bounded-hashcode-nat
<proof>

Default hash map size

definition *is-valid-def-hm-size* :: *'k* *itself* \Rightarrow *nat* \Rightarrow *bool*
where *is-valid-def-hm-size* *type n* $\equiv n > 1$

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* \equiv *Id* :: (*comp-res* \times *-*) *set*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of comp-res-rel i-comp-res*]

definition *comp2le* *cmp a b* \equiv

case cmp a b of LESS \Rightarrow *True* | *EQUAL* \Rightarrow *True* | *GREATER* \Rightarrow *False*

definition *comp2lt* *cmp a b* \equiv

case cmp a b of LESS \Rightarrow *True* | *EQUAL* \Rightarrow *False* | *GREATER* \Rightarrow *False*

definition *comp2eq* *cmp a b* \equiv

case cmp a b of LESS \Rightarrow *False* | *EQUAL* \Rightarrow *True* | *GREATER* \Rightarrow *False*

locale *linorder-on* =

fixes *D* :: *'a* *set*

fixes *cmp* :: *'a* \Rightarrow *'a* \Rightarrow *comp-res*

assumes *lt-eq*: $\llbracket x \in D; y \in D \rrbracket \implies \text{cmp } x \ y = \text{LESS} \longleftrightarrow (\text{cmp } y \ x = \text{GREATER})$

assumes *refl*[*simp, intro!*]: $x \in D \implies \text{cmp } x \ x = \text{EQUAL}$

assumes *trans*[*trans*]:
 $\llbracket x \in D; y \in D; z \in D; \text{cmp } x \ y = \text{LESS}; \text{cmp } y \ z = \text{LESS} \rrbracket \implies \text{cmp } x \ z = \text{LESS}$
 $\llbracket x \in D; y \in D; z \in D; \text{cmp } x \ y = \text{LESS}; \text{cmp } y \ z = \text{EQUAL} \rrbracket \implies \text{cmp } x \ z = \text{LESS}$
 $\llbracket x \in D; y \in D; z \in D; \text{cmp } x \ y = \text{EQUAL}; \text{cmp } y \ z = \text{LESS} \rrbracket \implies \text{cmp } x \ z = \text{LESS}$
 $\llbracket x \in D; y \in D; z \in D; \text{cmp } x \ y = \text{EQUAL}; \text{cmp } y \ z = \text{EQUAL} \rrbracket \implies \text{cmp } x \ z = \text{EQUAL}$
begin
abbreviation *le* $\equiv \text{comp2le } \text{cmp}$
abbreviation *lt* $\equiv \text{comp2lt } \text{cmp}$

lemma *eq-sym*: $\llbracket x \in D; y \in D \rrbracket \implies \text{cmp } x \ y = \text{EQUAL} \implies \text{cmp } y \ x = \text{EQUAL}$
 $\langle \text{proof} \rangle$
end

abbreviation *linorder* $\equiv \text{linorder-on } \text{UNIV}$

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

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

lemma (**in** *linorder*) *class-to-linorder*:
linorder (*dflt-cmp* (\leq) ($<$))
 $\langle \text{proof} \rangle$

lemma *restrict-linorder*: $\llbracket \text{linorder-on } D \ \text{cmp} ; D' \subseteq D \rrbracket \implies \text{linorder-on } D' \ \text{cmp}$
 $\langle \text{proof} \rangle$

Operations on Linear Orderings

Map with injective function

definition *cmp-img* **where** *cmp-img* *f* *cmp* *a* *b* $\equiv \text{cmp } (f \ a) \ (f \ b)$

lemma *img-linorder*[*intro?*]:
assumes *LO*: *linorder-on* (*f*'*D*) *cmp*
shows *linorder-on* *D* (*cmp-img* *f* *cmp*)
 $\langle \text{proof} \rangle$

Combine

definition *cmp-combine* *D1* *cmp1* *D2* *cmp2* *a* *b* \equiv
if *a* \in *D1* \wedge *b* \in *D1* *then* *cmp1* *a* *b*

else if $a \in D1 \wedge b \in D2$ *then* *LESS*
else if $a \in D2 \wedge b \in D1$ *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* \equiv (*SOME* r . *well-order-on* $UNIV$ r)

definition *univ-cmp* x y \equiv
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* \equiv
cmp-combine D *cmp* $UNIV$ *univ-cmp*

lemma *extend-linorder[intro?]*:
linorder-on D *cmp* \implies *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 \implies *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*: $cmp\text{-}lex\ cmp\ l\ m = cmp\text{-}lex'\ cmp\ l\ m$
 ⟨*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*: $cmp\text{-}prod = (\lambda cmp1\ cmp2\ (a1,a2)\ (b1,b2).\ ($

```
  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-nil*[simp]:
assumes *finite A*
shows *sorted-list-of-set A = [] \longleftrightarrow A={}*
<proof>

lemma *sorted-list-of-set-eq-nil2*[simp]:
assumes *finite A*
shows *[] = sorted-list-of-set A \longleftrightarrow 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>

lemma *the-sorted-list-of-set*:
assumes *distinct l*
assumes *sorted l*
shows *sorted-list-of-set (set l) = l*
<proof>

definition *sorted-list-of-map m* \equiv
map ($\lambda k. (k, the (m k))$) (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 \rightarrow 'b*
assumes [simp]: *finite (dom A)*
assumes [simp]: *finite (dom B)*

assumes E : *sorted-list-of-map* $A = \text{sorted-list-of-map } B$
shows $A=B$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-map-inj*:
inj-on sorted-list-of-map (*Collect* (*finite o dom*))
 $\langle \text{proof} \rangle$

end

definition *cmp-set cmp* \equiv
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 \implies *linorder* (*cmp-set cmp*)
 $\langle \text{proof} \rangle$

definition *cmp-map cmpk cmpv* \equiv
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*
 $\langle \text{proof} \rangle$

corollary *map-to-set-inj'*[*intro!*]: *inj-on map-to-set* S
 $\langle \text{proof} \rangle$

lemma *map-ord-linear*[*intro?*]:
assumes A : *linorder cmpk*
assumes B : *linorder cmpv*
shows *linorder* (*cmp-map cmpk cmpv*)
 $\langle \text{proof} \rangle$

locale *eq-linorder-on* = *linorder-on* +
assumes *cmp-imp-equal*: $\llbracket x \in D; y \in D \rrbracket \implies \text{cmp } x \ y = \text{EQUAL} \implies x = y$
begin
lemma *cmp-eq*[*simp*]: $\llbracket x \in D; y \in D \rrbracket \implies \text{cmp } x \ y = \text{EQUAL} \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$
end

abbreviation $eq\text{-}linorder \equiv eq\text{-}linorder\text{-}on\ UNIV$

lemma $dflt\text{-}cmp\text{-}2inv[simp]$:
 $dflt\text{-}cmp (comp2le\ cmp) (comp2lt\ cmp) = cmp$
 $\langle proof \rangle$

lemma (**in** $linorder$) $dflt\text{-}cmp\text{-}inv2[simp]$:
shows
 $(comp2le (dflt\text{-}cmp (\le) (<))) = (\le)$
 $(comp2lt (dflt\text{-}cmp (\le) (<))) = (<)$
 $\langle proof \rangle$

lemma $eq\text{-}linorder\text{-}class\text{-}conv$:
 $eq\text{-}linorder\ cmp \longleftrightarrow class.linorder (comp2le\ cmp) (comp2lt\ cmp)$
 $\langle proof \rangle$

lemma (**in** $linorder$) $class\text{-}to\text{-}eq\text{-}linorder$:
 $eq\text{-}linorder (dflt\text{-}cmp (\le) (<))$
 $\langle proof \rangle$

lemma $eq\text{-}linorder\text{-}comp2eq\text{-}eq$:
assumes $eq\text{-}linorder\ cmp$
shows $comp2eq\ cmp = (=)$
 $\langle proof \rangle$

lemma $restrict\text{-}eq\text{-}linorder$:
assumes $eq\text{-}linorder\text{-}on\ D\ cmp$
assumes $S: D' \subseteq D$
shows $eq\text{-}linorder\text{-}on\ D'\ cmp$
 $\langle proof \rangle$

lemma $combine\text{-}eq\text{-}linorder[intro?]$:
assumes $A: eq\text{-}linorder\text{-}on\ D1\ cmp1$
assumes $B: eq\text{-}linorder\text{-}on\ D2\ cmp2$
assumes $EQ: D = D1 \cup D2$
shows $eq\text{-}linorder\text{-}on\ D (cmp\text{-}combine\ D1\ cmp1\ D2\ cmp2)$
 $\langle proof \rangle$

lemma $img\text{-}eq\text{-}linorder[intro?]$:
assumes $A: eq\text{-}linorder\text{-}on\ (f'D)\ cmp$
assumes $INJ: inj\text{-}on\ f\ D$
shows $eq\text{-}linorder\text{-}on\ D (cmp\text{-}img\ f\ cmp)$
 $\langle proof \rangle$

lemma $univ\text{-}eq\text{-}linorder[intro?]$:
shows $eq\text{-}linorder\ univ\text{-}cmp$
 $\langle proof \rangle$

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 × D2) (cmp-prod cmp1 cmp2)*
 ⟨*proof*⟩

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

lemma *map-ord-eq-linorder*[*intro?*]:
 $\llbracket \text{eq-linorder } \text{cmpk}; \text{eq-linorder } \text{cmpv} \rrbracket \implies \text{eq-linorder } (\text{cmp-map } \text{cmpk } \text{cmpv})$
 ⟨*proof*⟩

definition *cmp-unit* :: *unit* \Rightarrow *unit* \Rightarrow *comp-res*
where [*simp*]: *cmp-unit u v* \equiv *EQUAL*

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

Parametricity

lemma *param-cmp-extend*[*param*]:
assumes $(\text{cmp}, \text{cmp}') \in R \rightarrow R \rightarrow \text{Id}$
assumes $\text{Range } R \subseteq D$
shows $(\text{cmp}, \text{cmp-extend } D \text{ cmp}') \in R \rightarrow R \rightarrow \text{Id}$
 ⟨*proof*⟩

lemma *param-cmp-img*[*param*]:
 $(\text{cmp-img}, \text{cmp-img}) \in (R_a \rightarrow R_b) \rightarrow (R_b \rightarrow R_b \rightarrow R_c) \rightarrow R_a \rightarrow R_a \rightarrow R_c$
 ⟨*proof*⟩

lemma *param-comp-res*[*param*]:
 $(\text{LESS}, \text{LESS}) \in \text{Id}$
 $(\text{EQUAL}, \text{EQUAL}) \in \text{Id}$
 $(\text{GREATER}, \text{GREATER}) \in \text{Id}$
 $(\text{case-comp-res}, \text{case-comp-res}) \in R_a \rightarrow R_a \rightarrow R_a \rightarrow \text{Id} \rightarrow R_a$
 ⟨*proof*⟩

term *cmp-lex*

lemma *param-cmp-lex*[*param*]:

$(cmp-lex, cmp-lex) \in (Ra \rightarrow Rb \rightarrow Id) \rightarrow \langle Ra \rangle list-rel \rightarrow \langle Rb \rangle list-rel \rightarrow Id$
 $\langle proof \rangle$

term *cmp-prod*

lemma *param-cmp-prod*[*param*]:

$(cmp-prod, cmp-prod) \in$
 $(Ra \rightarrow Rb \rightarrow Id) \rightarrow (Rc \rightarrow Rd \rightarrow Id) \rightarrow \langle Ra, Rc \rangle prod-rel \rightarrow \langle Rb, Rd \rangle prod-rel \rightarrow Id$
 $\langle proof \rangle$

lemma *param-cmp-unit*[*param*]:

$(cmp-unit, cmp-unit) \in Id \rightarrow Id \rightarrow Id$
 $\langle proof \rangle$

lemma *param-comp2eq*[*param*]: $(comp2eq, comp2eq) \in (R \rightarrow R \rightarrow Id) \rightarrow R \rightarrow R \rightarrow Id$

$\langle proof \rangle$

lemma *cmp-combine-paramD*:

assumes $(cmp, cmp-combine D1 cmp1 D2 cmp2) \in R \rightarrow R \rightarrow Id$

assumes $Range R \subseteq D1$

shows $(cmp, cmp1) \in R \rightarrow R \rightarrow Id$

$\langle proof \rangle$

lemma *cmp-extend-paramD*:

assumes $(cmp, cmp-extend D cmp') \in R \rightarrow R \rightarrow Id$

assumes $Range R \subseteq D$

shows $(cmp, cmp') \in R \rightarrow R \rightarrow Id$

$\langle proof \rangle$

Tuning of Generated Implementation

lemma [*autoref-post-simps*]: $comp2eq (dftt-cmp (\leq) ((<)::-::linorder \Rightarrow -)) = (=)$

$\langle proof \rangle$

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 ((\cup) a)
 \langle *proof* \rangle

definition *gen-union*
 $:: - \Rightarrow ('k \Rightarrow 's2 \Rightarrow 's2)$
 $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow 's2$

where

gen-union *it ins* $A B \equiv$ *it* A (λ -. *True*) *ins* B

lemma *gen-union*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *INS*: *GEN-OP* *ins* *Set.insert* ($Rk \rightarrow \langle Rk \rangle Rs2 \rightarrow \langle Rk \rangle Rs2$)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list* Rk $Rs1$ *tsl*)
shows (*gen-union* (λx . *foldli* (*tsl* x)) *ins*, (\cup))
 $\in ((\langle Rk \rangle Rs1) \rightarrow (\langle Rk \rangle Rs2) \rightarrow (\langle Rk \rangle Rs2))$
 \langle *proof* \rangle

lemma *foldli-inter*: *det-fold-set* X (λ -. *True*)
 $(\lambda x$ *s*. *if* $x \in a$ *then insert* x *s* *else s*) $\{\}$ (λs . $s \cap a$)
(is *det-fold-set* - - ?*f* - -)
 \langle *proof* \rangle

definition *gen-inter* $:: - \Rightarrow$
 $('k \Rightarrow 's2 \Rightarrow \text{bool}) \Rightarrow -$
where *gen-inter* *it1 memb2 ins3 empty3* $s1$ $s2$
 \equiv *it1* $s1$ (λ -. *True*)
 $(\lambda x$ *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* (\in) ($Rk \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$)
assumes *INS*:
 $GEN-OP$ *ins3* *Set.insert* ($Rk \rightarrow \langle Rk \rangle Rs3 \rightarrow \langle Rk \rangle Rs3$)
assumes *EMPTY*:
 $GEN-OP$ *empty3* $\{\}$ ($\langle Rk \rangle Rs3$)
shows (*gen-inter* (λx . *foldli* (*tsl* x)) *memb2* *ins3* *empty3*, (\cap))
 $\in ((\langle Rk \rangle Rs1) \rightarrow (\langle Rk \rangle Rs2) \rightarrow (\langle Rk \rangle Rs3))$
 \langle *proof* \rangle

lemma *foldli-diff*:
det-fold-set X (λ -. *True*) (λx *s*. *op-set-delete* x *s*) s ($(-)$ s)
 \langle *proof* \rangle

definition *gen-diff* $:: ('k \Rightarrow 's1 \Rightarrow 's1) \Rightarrow - \Rightarrow 's2 \Rightarrow -$
where *gen-diff* *del1 it2* $s1$ $s2$
 \equiv *it2* $s2$ (λ -. *True*) (λx *s*. *del1* x *s*) $s1$

lemma *gen-diff*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *DEL*:
 $GEN-OP\ del1\ op\ set\ delete\ (Rk \rightarrow \langle Rk \rangle Rs1 \rightarrow \langle Rk \rangle Rs1)$
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list* *Rk Rs2 it2*)
shows (*gen-diff del1* ($\lambda x. foldli\ (it2\ x), (-)$)
 $\in (\langle Rk \rangle Rs1) \rightarrow (\langle Rk \rangle Rs2) \rightarrow (\langle Rk \rangle Rs1)$)
 $\langle proof \rangle$

lemma *foldli-ball-aux*:
 $foldli\ l\ (\lambda x. x)\ (\lambda x. P\ x)\ b \longleftrightarrow b \wedge Ball\ (set\ l)\ P$
 $\langle proof \rangle$

lemma *foldli-ball*: *det-fold-set* *X* ($\lambda x. x$) ($\lambda x. P\ x$) *True* ($\lambda s. Ball\ s\ P$)
 $\langle proof \rangle$

definition *gen-ball* :: $- \Rightarrow 's \Rightarrow ('k \Rightarrow bool) \Rightarrow -$
where *gen-ball* *it s P* $\equiv it\ s\ (\lambda x. x)\ (\lambda x. P\ x)\ True$

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* ($\lambda x. foldli\ (it\ x), Ball$) $\in \langle Rk \rangle Rs \rightarrow (Rk \rightarrow Id) \rightarrow Id$)
 $\langle proof \rangle$

lemma *foldli-bex-aux*: $foldli\ l\ (\lambda x. \neg x)\ (\lambda x. P\ x)\ b \longleftrightarrow b \vee Bex\ (set\ l)\ P$
 $\langle proof \rangle$

lemma *foldli-bex*: *det-fold-set* *X* ($\lambda x. \neg x$) ($\lambda x. P\ x$) *False* ($\lambda s. Bex\ s\ P$)
 $\langle proof \rangle$

definition *gen-bex* :: $- \Rightarrow 's \Rightarrow ('k \Rightarrow bool) \Rightarrow -$
where *gen-bex* *it s P* $\equiv it\ s\ (\lambda x. \neg x)\ (\lambda x. P\ x)\ False$

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* ($\lambda x. foldli\ (it\ x), Bex$) $\in \langle Rk \rangle Rs \rightarrow (Rk \rightarrow Id) \rightarrow Id$)
 $\langle proof \rangle$

lemma *ball-subseteq*:
 $(Ball\ s1\ (\lambda x. x \in s2)) \longleftrightarrow s1 \subseteq s2$
 $\langle proof \rangle$

definition *gen-subseteq*
:: $('s1 \Rightarrow ('k \Rightarrow bool) \Rightarrow bool) \Rightarrow ('k \Rightarrow 's2 \Rightarrow bool) \Rightarrow -$
where *gen-subseteq* *ball1 mem2 s1 s2* $\equiv ball1\ s1\ (\lambda x. mem2\ x\ s2)$

lemma *gen-subseteq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball1 Ball* ($\langle Rk \rangle Rs1 \rightarrow (Rk \rightarrow Id) \rightarrow Id$)
assumes *GEN-OP mem2* (\in) ($Rk \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$)
shows (*gen-subseteq ball1 mem2, (\subseteq)*) $\in \langle Rk \rangle Rs1 \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$
 $\langle proof \rangle$

definition *gen-equal* $ss1\ ss2\ s1\ s2 \equiv ss1\ s1\ s2 \wedge ss2\ s2\ s1$

lemma *gen-equal*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ss1* (\subseteq) ($\langle Rk \rangle Rs1 \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$)
assumes *GEN-OP ss2* (\subseteq) ($\langle Rk \rangle Rs2 \rightarrow \langle Rk \rangle Rs1 \rightarrow Id$)
shows (*gen-equal ss1 ss2, (=)*) $\in \langle Rk \rangle Rs1 \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$
 $\langle proof \rangle$

lemma *foldli-card-aux*: *distinct l* \implies *foldli l* ($\lambda\cdot$. *True*)
 $(\lambda\cdot n.$ *Suc n*) $n = n + \text{card}(\text{set } l)$
 $\langle proof \rangle$

lemma *foldli-card*: *det-fold-set X* ($\lambda\cdot$. *True*) ($\lambda\cdot n.$ *Suc n*) $0\ \text{card}$
 $\langle proof \rangle$

definition *gen-card where*
gen-card it s $\equiv it\ s\ (\lambda x.$ *True*) ($\lambda\cdot n.$ *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* ($\lambda x.$ *foldli* (*it x*)), *card*) $\in \langle Rk \rangle Rs \rightarrow Id$
 $\langle proof \rangle$

lemma *fold-set*: *fold Set.insert l s* $= s \cup \text{set } l$
 $\langle proof \rangle$

definition *gen-set* $:: 's \Rightarrow ('k \Rightarrow 's \Rightarrow 's) \Rightarrow -$ **where**
gen-set emp ins l $= \text{fold ins } l\ \text{emp}$

lemma *gen-set*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *EMPTY*:
 $GEN-OP\ \text{emp}\ \{\}\ (\langle Rk \rangle Rs)$
assumes *INS*:
 $GEN-OP\ \text{ins}\ \text{Set.insert}\ (Rk \rightarrow \langle Rk \rangle Rs \rightarrow \langle Rk \rangle Rs)$
shows (*gen-set emp ins, set*) $\in \langle Rk \rangle \text{list-rel} \rightarrow \langle Rk \rangle Rs$
 $\langle proof \rangle$

lemma *ball-isEmpty*: *op-set-isEmpty s* $= (\forall x \in s.$ *False*)
 $\langle proof \rangle$

definition *gen-isEmpty* :: (*'s* ⇒ (*'k* ⇒ *bool*) ⇒ *bool*) ⇒ *'s* ⇒ *bool* **where**
gen-isEmpty ball s ≡ *ball s* ($\lambda\cdot$. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball Ball* ($\langle Rk \rangle Rs \rightarrow (Rk \rightarrow Id) \rightarrow Id$)
shows (*gen-isEmpty ball, op-set-isEmpty*) ∈ $\langle Rk \rangle Rs \rightarrow Id$
<proof>

lemma *foldli-size-abort-aux*:
 $\llbracket n0 \leq m; \text{distinct } l \rrbracket \implies$
foldli l ($\lambda n. n < m$) ($\lambda\cdot$. *Suc n*) *n0* = *min m (n0 + card (set l))*
<proof>

lemma *foldli-size-abort*:
det-fold-set X ($\lambda n. n < m$) ($\lambda\cdot$. *Suc n*) *0* (*op-set-size-abort m*)
<proof>

definition *gen-size-abort* **where**
gen-size-abort it m s ≡ *it s* ($\lambda n. n < m$) ($\lambda\cdot$. *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* ($\lambda x. \text{foldli } (it\ x)$), *op-set-size-abort*)
 ∈ *Id* → $\langle Rk \rangle Rs \rightarrow Id$
<proof>

lemma *size-abort-isSng*: *op-set-isSng s* \longleftrightarrow *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* → ($\langle Rk \rangle Rs$) → *Id*)
shows (*gen-isSng sizea, op-set-isSng*) ∈ $\langle Rk \rangle Rs \rightarrow Id$
<proof>

lemma *foldli-disjoint-aux*:
foldli l1 ($\lambda x. x$) ($\lambda x\cdot. \neg x \in s2$) *b* \longleftrightarrow *b* ∧ *op-set-disjoint (set l1) s2*
<proof>

lemma *foldli-disjoint*:
det-fold-set X ($\lambda x. x$) ($\lambda x\cdot. \neg x \in s2$) *True* ($\lambda s1. \text{op-set-disjoint } s1\ s2$)
<proof>

definition *gen-disjoint*
 $:: - \Rightarrow ('k \Rightarrow 's2 \Rightarrow bool) \Rightarrow -$
where *gen-disjoint* *it1 mem2 s1 s2*
 $\equiv it1\ s1\ (\lambda x. x)\ (\lambda x. \neg mem2\ x\ s2)\ True$

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* (\in) (*Rk* \rightarrow $\langle Rk \rangle Rs2 \rightarrow Id$)
shows (*gen-disjoint* ($\lambda x. foldli\ (it1\ x)$) *mem2,op-set-disjoint*)
 $\in \langle Rk \rangle Rs1 \rightarrow \langle Rk \rangle Rs2 \rightarrow Id$
 $\langle proof \rangle$

lemma *foldli-filter-aux*:
 $foldli\ l\ (\lambda-. True)\ (\lambda x\ s. if\ P\ x\ then\ insert\ x\ s\ else\ s)\ s0$
 $= s0 \cup op-set-filter\ P\ (set\ l)$
 $\langle proof \rangle$

lemma *foldli-filter*:
 $det-fold-set\ X\ (\lambda-. True)\ (\lambda x\ s. if\ P\ x\ then\ insert\ x\ s\ else\ s)\ \{\}$
 $(op-set-filter\ P)$
 $\langle proof \rangle$

definition *gen-filter*
where *gen-filter* *it1 emp2 ins2 P s1* \equiv
 $it1\ s1\ (\lambda-. True)\ (\lambda x\ s. if\ P\ x\ then\ ins2\ x\ s\ else\ s)\ emp2$

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:*
 $GEN-OP\ ins2\ Set.insert\ (Rk \rightarrow \langle Rk \rangle Rs2 \rightarrow \langle Rk \rangle Rs2)$
assumes *EMPTY:*
 $GEN-OP\ empty2\ \{\}\ (\langle Rk \rangle Rs2)$
shows (*gen-filter* ($\lambda x. foldli\ (it1\ x)$) *empty2 ins2,op-set-filter*)
 $\in (Rk \rightarrow Id) \rightarrow (\langle Rk \rangle Rs1) \rightarrow (\langle Rk \rangle Rs2)$
 $\langle proof \rangle$

lemma *foldli-image-aux*:
 $foldli\ l\ (\lambda-. True)\ (\lambda x\ s. insert\ (f\ x)\ s)\ s0$
 $= s0 \cup f'(set\ l)$
 $\langle proof \rangle$

lemma *foldli-image*:
 $det-fold-set\ X\ (\lambda-. True)\ (\lambda x\ s. insert\ (f\ x)\ s)\ \{\}$
 $((\cdot)\ f)$
 $\langle proof \rangle$

definition *gen-image*

where *gen-image* *it1 emp2 ins2 f s1* \equiv
it1 s1 (λ -. *True*) ($\lambda x s$. *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'* \rightarrow (*Rk* \wedge) *Rs2* \rightarrow (*Rk* \wedge) *Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (*(Rk* \wedge) *Rs2*)
shows (*gen-image* (λx . *foldli* (*it1 x*) *empty2 ins2*, ('))
 \in (*Rk* \rightarrow *Rk'*) \rightarrow (*(Rk)* *Rs1*) \rightarrow (*(Rk* \wedge) *Rs2*)
 \langle *proof* \rangle

lemma *foldli-pick*:
assumes *l* \neq []
obtains *x* **where** *x* \in *set l*
and (*foldli l* (*case-option True* (λ -. *False*)) (λx -. *Some x*) *None*) = *Some x*
 \langle *proof* \rangle

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

context begin interpretation *autoref-syn* \langle *proof* \rangle

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 (Rk) Rs*
shows (*RETURN* (*gen-pick* (λx . *foldli* (*it x*) *s*),
(*OP op-set-pick* $::$ (*(Rk) Rs* \rightarrow (*(Rk) nres-rel*) *s'*) \in (*(Rk) nres-rel*)
 \langle *proof* \rangle

end

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

lemma *foldli-Sigma-aux*:
fixes *s* $::$ '*s1-impl* **and** *s'* $::$ '*k* *set*
fixes *f* $::$ '*k-impl* \Rightarrow '*s2-impl* **and** *f'* $::$ '*k* \Rightarrow '*l* *set*
fixes *s0* $::$ '*kl-impl* **and** *s0'* $::$ ('*k* \times '*l*) *set*
assumes *IT1: is-set-to-list Rk Rs1 it1*

assumes *IT2*: *is-set-to-list* *Rl Rs2 it2*
assumes *INS*:
 $(insX, Set.insert) \in$
 $\langle \langle Rk, Rl \rangle prod-rel \rightarrow \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3 \rightarrow \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3 \rangle$
assumes *SOR*: $(s0, s0') \in \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3$
assumes *SR*: $(s, s') \in \langle Rk \rangle Rs1$
assumes *FR*: $(f, f') \in Rk \rightarrow \langle Rl \rangle Rs2$
shows $(foldli (it1 s) (\lambda-. True) (\lambda x s.$
 $foldli (it2 (f x)) (\lambda-. True) (\lambda y s. insX (x,y) s) s$
 $) s0, s0' \cup Sigma s' f')$
 $\in \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3$
 $\langle proof \rangle$

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*:
 $GEN-OP empX \{ \} (\langle \langle Rk, Rl \rangle prod-rel \rangle Rs3)$
assumes *INS*:
 $GEN-OP insX Set.insert$
 $\langle \langle Rk, Rl \rangle prod-rel \rightarrow \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3 \rightarrow \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3 \rangle$
shows $(gen-Sigma (\lambda x. foldli (it1 x)) (\lambda x. foldli (it2 x)) empX insX, Sigma)$
 $\in (\langle Rk \rangle Rs1) \rightarrow (Rk \rightarrow \langle Rl \rangle Rs2) \rightarrow \langle \langle Rk, Rl \rangle prod-rel \rangle Rs3$
 $\langle proof \rangle$

lemma *gen-cart*:
assumes *PRIO-TAG-GEN-ALGO*
assumes [*param*]: $(sigma, Sigma) \in (\langle Rk \rangle Rsk \rightarrow (Rk \rightarrow \langle Ry \rangle Rsy) \rightarrow \langle Rk \times_r$
 $Ry \rangle Rsp)$
shows $(\lambda x y. sigma x (\lambda-. y), op-set-cart) \in \langle Rk \rangle Rsk \rightarrow \langle Ry \rangle Rsy \rightarrow \langle Rk \times_r$
 $Ry \rangle Rsp$
 $\langle proof \rangle$
lemmas [*autoref-rules*] = *gen-cart*[*OF - GEN-OP-D*]

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *op-set-to-sorted-list-autoref*[*autoref-rules*]:
assumes *SIDE-GEN-ALGO* (*is-set-to-sorted-list* *ordR Rk Rs tsl*)
shows $(\lambda si. RETURN (tsl si), OP (op-set-to-sorted-list ordR))$
 $\in \langle Rk \rangle Rs \rightarrow \langle \langle Rk \rangle list-rel \rangle nres-rel$
 $\langle proof \rangle$

lemma *op-set-to-list-autoref*[*autoref-rules*]:
assumes *SIDE-GEN-ALGO* (*is-set-to-sorted-list* *ordR Rk Rs tsl*)
shows $(\lambda si. RETURN (tsl si), op-set-to-list)$
 $\in \langle Rk \rangle Rs \rightarrow \langle \langle Rk \rangle list-rel \rangle nres-rel$

$\langle proof \rangle$

end

lemma *foldli-Union*: *det-fold-set* X $(\lambda-. True)$ (\cup) $\{\}$ *Union*
 $\langle proof \rangle$

definition *gen-Union*

$:: - \Rightarrow 's3 \Rightarrow ('s2 \Rightarrow 's3 \Rightarrow 's3)$
 $\Rightarrow 's1 \Rightarrow 's3$

where

gen-Union *it emp un X* \equiv *it X* $(\lambda-. True)$ *un emp*

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

assumes *PRIO-TAG-GEN-ALGO*

assumes *EMP*: *GEN-OP emp* $\{\}$ $(\langle Rk \rangle Rs3)$

assumes *UN*: *GEN-OP un* (\cup) $(\langle Rk \rangle Rs2 \rightarrow \langle Rk \rangle Rs3 \rightarrow \langle Rk \rangle Rs3)$

assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list* $(\langle Rk \rangle Rs2)$ *Rs1* *tsl*)

shows (*gen-Union* $(\lambda x. foldli (tsl x)) emp un, Union$) $\in \langle \langle Rk \rangle Rs2 \rangle Rs1 \rightarrow \langle Rk \rangle Rs3$
 $\langle proof \rangle$

definition *atLeastLessThan-impl* $a b \equiv do \{$

$(-,r) \leftarrow WHILET (\lambda(i,r). i < b) (\lambda(i,r). RETURN (i+1, insert i r)) (a, \{\});$
 $RETURN r$

$\}$

lemma *atLeastLessThan-impl-correct*:

atLeastLessThan-impl $a b \leq SPEC (\lambda r. r = \{a..<b::nat\})$
 $\langle proof \rangle$

schematic-goal *atLeastLessThan-code-aux*:

notes [*autoref-rules*] = *IdI*[*of a*] *IdI*[*of b*]

assumes [*autoref-rules*]: (*emp, \{\}*) $\in Rs$

assumes [*autoref-rules*]: (*ins, insert*) $\in nat-rel \rightarrow Rs \rightarrow Rs$

shows (*?c, atLeastLessThan-impl*)

$\in nat-rel \rightarrow nat-rel \rightarrow \langle Rs \rangle nres-rel$

$\langle proof \rangle$

concrete-definition *atLeastLessThan-code* **uses** *atLeastLessThan-code-aux*

schematic-goal *atLeastLessThan-tr-aux*:

$RETURN ?c \leq atLeastLessThan-code emp ins a b$

$\langle proof \rangle$

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* $(nat-rel \rightarrow Rs \rightarrow Rs)$

```

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*) ($\lambda(k,v)$ *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*) ($\lambda(k,v)$ *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*)

($\lambda(k,v)$ *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*) ($\lambda(k,v)$ *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 ($\langle Rk, Rv \rangle Rs2$)

shows (*gen-restrict* (*foldli o tsl*) *upd emp, op-map-restrict*)

$\in (\langle Rk, Rv \rangle prod-rel \rightarrow Id) \rightarrow (\langle Rk, Rv \rangle Rs1) \rightarrow (\langle Rk, Rv \rangle Rs2)$

$\langle proof \rangle$

lemma *fold-map-of*:

fold ($\lambda(k,v) s. op-map-update k v s$) (*rev l*) *Map.empty* = *map-of l*

$\langle proof \rangle$

definition *gen-map-of* :: $'m \Rightarrow ('k \Rightarrow 'v \Rightarrow 'm \Rightarrow 'm) \Rightarrow -$ **where**

gen-map-of emp upd l $\equiv fold (\lambda(k,v) s. upd k v s) (rev l) emp$

lemma *gen-map-of*[*autoref-rules-raw*]:

assumes *PRIO-TAG-GEN-ALGO*

assumes *UPD*: *GEN-OP upd op-map-update* ($Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle Rm \rightarrow \langle Rk, Rv \rangle Rm$)

assumes *EMPTY*: *GEN-OP emp Map.empty* ($\langle Rk, Rv \rangle Rm$)

shows (*gen-map-of emp upd, map-of*) $\in \langle \langle Rk, Rv \rangle prod-rel \rangle list-rel \rightarrow \langle Rk, Rv \rangle Rm$

$\langle proof \rangle$

lemma *foldli-ball-aux*:

distinct (*map fst l*) $\implies foldli l (\lambda x. x) (\lambda x -. P x) b$

$\longleftrightarrow b \wedge op-map-ball (map-of l) P$

$\langle proof \rangle$

lemma *foldli-ball*:

det-fold-map X ($\lambda x. x$) ($\lambda x -. P x$) *True* ($\lambda m. op-map-ball m P$)

$\langle proof \rangle$

definition *gen-ball* :: $('m \Rightarrow -) \Rightarrow -$ **where**

gen-ball it m P $\equiv it m (\lambda x. x) (\lambda x -. P x) True$

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*)

$\in \langle Rk, Rv \rangle Rm \rightarrow (\langle Rk, Rv \rangle prod-rel \rightarrow Id) \rightarrow Id$

$\langle proof \rangle$

lemma *foldli-bex-aux*:

distinct (*map fst l*) $\implies foldli l (\lambda x. \neg x) (\lambda x -. P x) b$

$\longleftrightarrow b \vee op-map-bex (map-of l) P$

$\langle proof \rangle$

lemma *foldli-bex*:

det-fold-map X ($\lambda x. \neg x$) ($\lambda x -. P x$) *False* ($\lambda m. op-map-bex m P$)

$\langle proof \rangle$

definition *gen-bex* :: ('m ⇒ -) ⇒ - **where**
gen-bex it m P ≡ *it m (λx. ¬x) (λx -. 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 (λn. n < m) (λ- n. Suc n) n0 = min m (n0 + card (dom (map-of l)))
 ⟨*proof*⟩

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

definition *gen-size-abort* :: ('s ⇒ -) ⇒ - **where**

gen-size-abort $it\ m\ s \equiv it\ s\ (\lambda n. n < m)\ (\lambda n. Suc\ n)\ 0$

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*)
 $\in Id \rightarrow \langle Rk, Rv \rangle Rm \rightarrow Id$
 $\langle proof \rangle$

lemma *size-abort-isSng*: *op-map-isSng s* \longleftrightarrow *op-map-size-abort 2 s = 1*
 $\langle proof \rangle$

definition *gen-isSng* :: (*nat* \Rightarrow '*s* \Rightarrow *nat*) \Rightarrow - **where**
gen-isSng sizea s \equiv *sizea 2 s = 1*

lemma *gen-isSng*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP sizea op-map-size-abort* (*Id* \rightarrow ($\langle Rk, Rv \rangle Rm$) \rightarrow *Id*)
shows (*gen-isSng sizea*, *op-map-isSng*)
 $\in \langle Rk, Rv \rangle Rm \rightarrow Id$
 $\langle proof \rangle$

lemma *foldli-pick*:
assumes $l \neq []$
obtains *k v* **where** $(k, v) \in set\ l$
and (*foldli l* (*case-option True* ($\lambda -. False$)) ($\lambda x -. Some\ x$) *None*)
 $= Some\ (k, v)$
 $\langle proof \rangle$

definition *gen-pick* **where**
gen-pick it s \equiv
 $(the\ (it\ s\ (case-option\ True\ (\lambda -. False)))\ (\lambda x -. Some\ x)\ None))$

context **begin** *interpretation autoref-syn* $\langle proof \rangle$

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* ($m' \neq Map.empty$)
assumes *SREF*: $(m, m') \in \langle Rk, Rv \rangle Rm$
shows (*RETURN* (*gen-pick* ($\lambda x. foldli\ (it\ x)\ m$),
 $(OP\ op-map-pick\ ::\ \langle Rk, Rv \rangle Rm \rightarrow \langle Rk \times_r Rv \rangle nres-rel$) $\$m'$) $\in \langle Rk \times_r Rv \rangle nres-rel$)
 $\langle proof \rangle$
end

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

context begin interpretation *autoref-syn* \langle *proof* \rangle

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* $::: \langle Rk, Rv \rangle Rm \rightarrow \langle Rk \times_r Rv \rangle nres-rel$)\$*m'*) \in
 $\langle Rk \times_r Rv \rangle nres-rel$)

assumes *DEL: GEN-OP del op-map-delete* (*Rk* $\rightarrow \langle Rk, Rv \rangle Rm \rightarrow \langle Rk, Rv \rangle Rm$)

assumes [*param*]: (*m, m'*) $\in \langle Rk, Rv \rangle Rm$

shows (*gen-map-pick-remove pick del m*,

(*OP op-map-pick-remove*

$::: \langle Rk, Rv \rangle Rm \rightarrow \langle (Rk \times_r Rv) \times_r \langle Rk, Rv \rangle Rm \rangle nres-rel$)\$*m'*)

$\in \langle (Rk \times_r Rv) \times_r \langle Rk, Rv \rangle Rm \rangle nres-rel$

\langle *proof* \rangle

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* $:: ('k \times unit)$ *list*

shows *distinct (map fst l)* \longleftrightarrow *distinct l*

\langle *proof* \rangle

definition

map2set-rel $::$

$(('ki \times 'k)$ *set* \Rightarrow (*unit* \times *unit*) *set* \Rightarrow (*'mi* \times (*'k* \rightarrow *unit*)) *set*) \Rightarrow

$(('ki \times 'k)$ *set* \Rightarrow

$(('mi \times ('k$ *set*)) *set*

where

map2set-rel-def-internal:
 $\text{map2set-rel } R \text{ } Rk \equiv \langle Rk, Id :: (\text{unit} \times -) \text{ set} \rangle R \text{ } O \{ (m, \text{dom } m) \mid m. \text{ True} \}$

lemma *map2set-rel-def*: $\langle Rk \rangle (\text{map2set-rel } R)$
 $= \langle Rk, Id :: (\text{unit} \times -) \text{ set} \rangle R \text{ } O \{ (m, \text{dom } m) \mid m. \text{ True} \}$
 $\langle \text{proof} \rangle$

lemma *map2set-relI*:
assumes $(s, m') \in \langle Rk, Id \rangle R$ **and** $s' = \text{dom } m'$
shows $(s, s') \in \langle Rk \rangle \text{map2set-rel } R$
 $\langle \text{proof} \rangle$

lemma *map2set-relE*:
assumes $(s, s') \in \langle Rk \rangle \text{map2set-rel } R$
obtains m' **where** $(s, m') \in \langle Rk, Id \rangle R$ **and** $s' = \text{dom } m'$
 $\langle \text{proof} \rangle$

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

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

lemmas [*autoref-rel-intf*] =
 $\text{REL-INTFI}[\text{of map2set-rel } R \text{ } i\text{-set}] \text{ for } R$

definition *map2set-insert* $i \text{ } k \text{ } s \equiv i \text{ } k \text{ } () \text{ } s$

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

definition *map2set-memb* $l \text{ } k \text{ } s \equiv \text{case } l \text{ } k \text{ } s \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } - \Rightarrow \text{True}$

lemma *map2set-memb[autoref-rules-raw]*:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP l op-map-lookup* $(Rk \rightarrow \langle Rk, Id \rangle R \rightarrow \langle Id \rangle \text{option-rel})$
shows $(\text{map2set-memb } l, (\in))$
 $\in Rk \rightarrow \langle Rk \rangle \text{map2set-rel } R \rightarrow Id$
 $\langle \text{proof} \rangle$

lemma *map2set-delete[autoref-rules-raw]*:
assumes *PRIO-TAG-GEN-ALGO*

assumes *GEN-OP* *d op-map-delete* ($Rk \rightarrow \langle Rk, Id \rangle R \rightarrow \langle Rk, Id \rangle R$)
shows ($d, op\text{-}set\text{-}delete$) $\in Rk \rightarrow \langle Rk \rangle \text{map2set-rel } R \rightarrow \langle Rk \rangle \text{map2set-rel } R$
 $\langle proof \rangle$

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*)))
 $\langle proof \rangle$

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*)))
 $\langle proof \rangle$

Transferring 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* (++) ($\langle Rk, Id \rangle R \rightarrow \langle Rk, Id \rangle R \rightarrow \langle Rk, Id \rangle R$)
shows ($u, (\cup)$) $\in \langle Rk \rangle \text{map2set-rel } R \rightarrow \langle Rk \rangle \text{map2set-rel } R \rightarrow \langle Rk \rangle \text{map2set-rel } R$
 $\langle proof \rangle$

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-unit*) (*map* ($\lambda k. (k, ())$) *l*)
(*map* ($\lambda k. (k, ())$) *m*) =
cmp-lex *cmp* *l* *m*
 $\langle proof \rangle$

lemma *cmp-img-zip-unit*[*simp*]:
cmp-img ($\lambda m. \text{map } (\lambda k. (k, ())) (f\ m)$) (*cmp-lex* (*cmp-prod* *cmp1* *cmp-unit*))
= *cmp-img* *f* (*cmp-lex* *cmp1*)
 $\langle proof \rangle$

lemma *map2set-finite*[*relator-props*]:
assumes *finite-map-rel* ($\langle Rk, Id \rangle R$)
shows *finite-set-rel* ($\langle Rk \rangle \text{map2set-rel } R$)
 $\langle proof \rangle$

lemma *map2set-cmp*[*autoref-rules-raw*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder* *cmpk*)


```

assumes MPAR:
  GEN-OP cmp (cmp-map cmpk cmp-unit) ( $\langle Rk, Id \rangle R \rightarrow \langle Rk, Id \rangle R \rightarrow Id$ )
assumes FIN: PREFER finite-map-rel ( $\langle Rk, Id \rangle R$ )
shows (cmp, cmp-set cmpk)  $\in \langle Rk \rangle \text{map2set-rel } R \rightarrow \langle Rk \rangle \text{map2set-rel } R \rightarrow Id$ 
 $\langle \text{proof} \rangle$ 

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 ( $\leq$ ) ( $<$ ), dflt-cmp ( $\leq$ ) ( $<$ ))  $\in$ 
   $\langle Id, Id \rangle \text{prod-rel} \rightarrow \langle Id, Id \rangle \text{prod-rel} \rightarrow Id$ 
 $\langle \text{proof} \rangle$ 

```

```

lemma gen-prod-cmp-dflt[autoref-rules-raw]:
assumes PRIO-TAG-GEN-ALGO
assumes GEN-OP cmp1 (dflt-cmp ( $\leq$ ) ( $<$ )) ( $R1 \rightarrow R1 \rightarrow Id$ )
assumes GEN-OP cmp2 (dflt-cmp ( $\leq$ ) ( $<$ )) ( $R2 \rightarrow R2 \rightarrow Id$ )
shows (cmp-prod cmp1 cmp2, dflt-cmp ( $\leq$ ) ( $<$ ))  $\in$ 
   $\langle R1, R2 \rangle \text{prod-rel} \rightarrow \langle R1, R2 \rangle \text{prod-rel} \rightarrow Id$ 
 $\langle \text{proof} \rangle$ 

```

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  $\times$  nat

term Diff-Array.array-length

```

definition $as\text{-raw-}\alpha\ s \equiv take\ (snd\ s)\ (list\text{-of-array}\ (fst\ s))$

definition $as\text{-raw-invar}\ s \equiv snd\ s \leq array\text{-length}\ (fst\ s)$

definition $as\text{-rel-def-internal}$: $as\text{-rel}\ R \equiv br\ as\text{-raw-}\alpha\ as\text{-raw-invar}\ O\ \langle R \rangle list\text{-rel}$

lemma $as\text{-rel-def}$: $\langle R \rangle as\text{-rel} \equiv br\ as\text{-raw-}\alpha\ as\text{-raw-invar}\ O\ \langle R \rangle list\text{-rel}$
 $\langle proof \rangle$

lemma $[relator\text{-props}]$: $single\text{-valued}\ R \implies single\text{-valued}\ (\langle R \rangle as\text{-rel})$
 $\langle proof \rangle$

lemmas $[autoref\text{-rel-intf}] = REL\text{-INTFI}[of\ as\text{-rel}\ i\text{-list}]$

definition $as\text{-empty}\ (-::unit) \equiv (array\text{-of-list}\ [], 0)$

lemma $as\text{-empty-refine}[autoref\text{-rules}]$: $(as\text{-empty}\ (), []) \in \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

definition $as\text{-push}\ s\ x \equiv let$

$(a, n) = s;$
 $a = if\ n = array\text{-length}\ a\ then$
 $\quad array\text{-grow}\ a\ (max\ 4\ (2*n))\ x$
 $\quad else\ a;$
 $a = array\text{-set}\ a\ n\ x$
 in
 $(a, n+1)$

lemma $as\text{-push-refine}[autoref\text{-rules}]$:

$(as\text{-push}, op\text{-list-append-elem}) \in \langle R \rangle as\text{-rel} \rightarrow R \rightarrow \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

term $array\text{-shrink}$

definition $as\text{-shrink}\ s \equiv let$

$(a, n) = s;$
 $a = if\ 128*n \leq array\text{-length}\ a \wedge n > 4\ then$
 $\quad array\text{-shrink}\ a\ n$
 $\quad else\ a$
 in
 (a, n)

lemma $as\text{-shrink-id-refine}$: $(as\text{-shrink}, id) \in \langle R \rangle as\text{-rel} \rightarrow \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

lemma $as\text{-shrinkI}$:

assumes $[param]$: $(s, a) \in \langle R \rangle as\text{-rel}$
shows $(as\text{-shrink}\ s, a) \in \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

definition $as\text{-pop } s \equiv let (a,n)=s \text{ in } as\text{-shrink } (a,n - 1)$

lemma $as\text{-pop-refine}[autoref\text{-rules}]$: $(as\text{-pop}, butlast) \in \langle R \rangle as\text{-rel} \rightarrow \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

definition $as\text{-get } s \ i \equiv let (a,::nat)=s \text{ in } array\text{-get } a \ i$

lemma $as\text{-get-refine}$:
assumes 1: $i' < length \ l$
assumes 2: $(a,l) \in \langle R \rangle as\text{-rel}$
assumes 3[$param$]: $(i,i') \in nat\text{-rel}$
shows $(as\text{-get } a \ i, !!i') \in R$
 $\langle proof \rangle$

context begin interpretation $autoref\text{-syn} \langle proof \rangle$

lemma $as\text{-get-autoref}[autoref\text{-rules}]$:
assumes $(l,l') \in \langle R \rangle as\text{-rel}$
assumes $(i,i') \in Id$
assumes $SIDE\text{-PRECOND } (i' < length \ l')$
shows $(as\text{-get } l \ i, (OP \ nth \ :: \ \langle R \rangle as\text{-rel} \rightarrow nat\text{-rel} \rightarrow R) \$l' \$i') \in R$
 $\langle proof \rangle$

definition $as\text{-set } s \ i \ x \equiv let (a,n::nat)=s \text{ in } (array\text{-set } a \ i \ x, n)$

lemma $as\text{-set-refine}[autoref\text{-rules}]$:
 $(as\text{-set}, list\text{-update}) \in \langle R \rangle as\text{-rel} \rightarrow nat\text{-rel} \rightarrow R \rightarrow \langle R \rangle as\text{-rel}$
 $\langle proof \rangle$

definition $as\text{-length} :: 'a \ array\text{-stack} \Rightarrow nat$ **where**
 $as\text{-length} = snd$

lemma $as\text{-length-refine}[autoref\text{-rules}]$:
 $(as\text{-length}, length) \in \langle R \rangle as\text{-rel} \rightarrow nat\text{-rel}$
 $\langle proof \rangle$

definition $as\text{-top } s \equiv as\text{-get } s \ (as\text{-length } s - 1)$

lemma $as\text{-top-code}[code]$: $as\text{-top } s = (let (a,n)=s \text{ in } array\text{-get } a \ (n - 1))$
 $\langle proof \rangle$

lemma $as\text{-top-refine}$: $\llbracket l \neq [] \rrbracket; (s,l) \in \langle R \rangle as\text{-rel} \implies (as\text{-top } s, last \ l) \in R$
 $\langle proof \rangle$

lemma $as\text{-top-autoref}[autoref\text{-rules}]$:
assumes $(l,l') \in \langle R \rangle as\text{-rel}$
assumes $SIDE\text{-PRECOND } (l' \neq [])$
shows $(as\text{-top } l, (OP \ last \ :: \ \langle R \rangle as\text{-rel} \rightarrow R) \$l') \in R$
 $\langle proof \rangle$

definition $as-is-empty\ s \equiv as-length\ s = 0$

lemma $as-is-empty-code[code]: as-is-empty\ s = (snd\ s = 0)$
 $\langle proof \rangle$

lemma $as-is-empty-refine[autoref-rules]:$
 $(as-is-empty, is-Nil) \in \langle R \rangle as-rel \rightarrow bool-rel$
 $\langle proof \rangle$

definition $as-take\ m\ s \equiv let\ (a, n) = s\ in$
 if $m < n$ then
 $as-shrink\ (a, m)$
 else (a, n)

lemma $as-take-refine[autoref-rules]:$
 $(as-take, take) \in nat-rel \rightarrow \langle R \rangle as-rel \rightarrow \langle R \rangle as-rel$
 $\langle proof \rangle$

definition $as-singleton\ x \equiv (array-of-list\ [x], 1)$

lemma $as-singleton-refine[autoref-rules]:$
 $(as-singleton, op-list-singleton) \in R \rightarrow \langle R \rangle as-rel$
 $\langle proof \rangle$

end

end

2.3.2 List Based Sets

theory *Impl-List-Set*

imports

../Iterator/Iterator

../Intf/Intf-Set

begin

lemma $list-all2-refl-conv:$
 $list-all2\ P\ xs\ xs \longleftrightarrow (\forall x \in set\ xs. P\ x\ x)$
 $\langle proof \rangle$

primrec $glist-member :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow bool$ **where**
 $glist-member\ eq\ x\ [] \longleftrightarrow False$
 $| glist-member\ eq\ x\ (y\#\!ys) \longleftrightarrow eq\ x\ y \vee glist-member\ eq\ x\ ys$

lemma $param-glist-member[param]:$
 $(glist-member, glist-member) \in (Ra \rightarrow Ra \rightarrow Id) \rightarrow Ra \rightarrow \langle Ra \rangle list-rel \rightarrow Id$
 $\langle proof \rangle$

lemma $list-member-alt: List.member = (\lambda l\ x. glist-member\ (=)\ x\ l)$

<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:*

$\forall x' \in \text{set } l'. x' \in \text{Range } R \implies l' \in \text{Range } (\langle R \rangle \text{list-rel})$

<proof>

All finite sets can be represented

lemma *list-set-rel-range:*

$\text{Range } (\langle R \rangle \text{list-set-rel}) = \{ S. \text{finite } S \wedge S \subseteq \text{Range } R \}$
(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)

$\langle proof \rangle$

lemma *list-set-rel-sv*[*relator-props*]:
single-valued $R \implies \text{single-valued } (\langle R \rangle \text{list-set-rel})$
 $\langle proof \rangle$

lemma *Id-comp-Id*: $Id \circ Id = Id$ $\langle proof \rangle$

lemma *glist-member-id-impl*:
 $(\text{glist-member } (=), (\in)) \in Id \rightarrow \text{br set distinct} \rightarrow Id$
 $\langle proof \rangle$

lemma *glist-insert-id-impl*:
 $(\text{glist-insert } (=), \text{Set.insert}) \in Id \rightarrow \text{br set distinct} \rightarrow \text{br set distinct}$
 $\langle proof \rangle$

lemma *glist-delete-id-impl*:
 $(\text{glist-delete } (=), \lambda x s. s - \{x\})$
 $\in Id \rightarrow \text{br set distinct} \rightarrow \text{br set distinct}$
 $\langle proof \rangle$

lemma *list-set-autoref-empty*[*autoref-rules*]:
 $([], \{\}) \in \langle R \rangle \text{list-set-rel}$
 $\langle proof \rangle$

lemma *list-set-autoref-member*[*autoref-rules*]:
assumes *GEN-OP eq* $(=) (R \rightarrow R \rightarrow Id)$
shows $(\text{glist-member } \text{eq}, (\in)) \in R \rightarrow \langle R \rangle \text{list-set-rel} \rightarrow Id$
 $\langle proof \rangle$

lemma *list-set-autoref-insert*[*autoref-rules*]:
assumes *GEN-OP eq* $(=) (R \rightarrow R \rightarrow Id)$
shows $(\text{glist-insert } \text{eq}, \text{Set.insert})$
 $\in R \rightarrow \langle R \rangle \text{list-set-rel} \rightarrow \langle R \rangle \text{list-set-rel}$
 $\langle proof \rangle$

lemma *list-set-autoref-delete*[*autoref-rules*]:
assumes *GEN-OP eq* $(=) (R \rightarrow R \rightarrow Id)$
shows $(\text{glist-delete } \text{eq}, \text{op-set-delete})$
 $\in R \rightarrow \langle R \rangle \text{list-set-rel} \rightarrow \langle R \rangle \text{list-set-rel}$
 $\langle proof \rangle$

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* $(\lambda - . \text{True}) R \text{list-set-rel id}$
 $\langle proof \rangle$

lemma *list-set-it-simp*[*refine-transfer-post-simp*]:
 $\text{foldli } (id \ l) = \text{foldli } l$ $\langle proof \rangle$

lemma *glist-insert-dj-id-impl*:
 $\llbracket x \notin s; (l,s) \in br\ set\ distinct \rrbracket \implies (x \# l, insert\ x\ s) \in br\ set\ distinct$
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *list-set-autoref-insert-dj*[*autoref-rules*]:
assumes *PRIO-TAG-OPTIMIZATION*
assumes *SIDE-PRECOND-OPT* $(x' \notin s')$
assumes $(x, x') \in R$
assumes $(s, s') \in \langle R \rangle list\ set\ rel$
shows $(x \# s,$
 $(OP\ Set.insert\ ::\ R \rightarrow \langle R \rangle list\ set\ rel \rightarrow \langle R \rangle list\ set\ rel)\ \$\ x'\ \$\ s')$
 $\in \langle R \rangle list\ set\ rel$
 $\langle proof \rangle$

end

More Operations

lemma *list-set-autoref-isEmpty*[*autoref-rules*]:
 $(is\ Nil, op\ set\ isEmpty) \in \langle R \rangle list\ set\ rel \rightarrow bool\ rel$
 $\langle proof \rangle$

lemma *list-set-autoref-filter*[*autoref-rules*]:
 $(filter, op\ set\ filter)$
 $\in (R \rightarrow bool\ rel) \rightarrow \langle R \rangle list\ set\ rel \rightarrow \langle R \rangle list\ set\ rel$
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *list-set-autoref-inj-image*[*autoref-rules*]:
assumes *PRIO-TAG-OPTIMIZATION*
assumes *INJ: SIDE-PRECOND-OPT* $(inj\ on\ f\ s)$
assumes [*param*]: $(fi, f) \in Ra \rightarrow Rb$
assumes *LP*: $(l, s) \in \langle Ra \rangle list\ set\ rel$
shows $(map\ fi\ l,$
 $(OP\ (')\ ::\ (Ra \rightarrow Rb) \rightarrow \langle Ra \rangle list\ set\ rel \rightarrow \langle Rb \rangle list\ set\ rel)\ \$f\ \$s)$
 $\in \langle Rb \rangle list\ set\ rel$
 $\langle proof \rangle$

end

lemma *list-set-cart-autoref*[*autoref-rules*]:
fixes $Rx :: ('xi \times 'x)\ set$
fixes $Ry :: ('yi \times 'y)\ set$
shows $(\lambda xl\ yl. [(x, y). x \leftarrow xl, y \leftarrow yl], op\ set\ cart)$
 $\in \langle Rx \rangle list\ set\ rel \rightarrow \langle Ry \rangle list\ set\ rel \rightarrow \langle Rx \times_r Ry \rangle list\ set\ rel$
 $\langle proof \rangle$

Optimizations

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

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

definition [*simp*]: $LIST\ SET\ REV\ TAG \equiv \lambda x. x$

lemma *LIST-SET-REV-TAG-autoref*[*autoref-rules*]:
 $(rev, LIST\ SET\ REV\ TAG) \in \langle R \rangle list\ set\ rel \rightarrow \langle R \rangle list\ set\ rel$
 ⟨*proof*⟩

end

theory *Array-Iterator*

imports *Iterator ../Lib/Diff-Array*

begin

lemma *idx-iteratei-aux-array-get-Array-conv-nth*:
 $idx\ iteratei\ aux\ array\ get\ sz\ i\ (Array\ xs)\ c\ f\ \sigma =$
 $idx\ iteratei\ aux\ (!)\ sz\ i\ xs\ c\ f\ \sigma$
 ⟨*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 $(k, v)\ list\ map = (k \times v)\ list$

definition $list\ map\ invar = distinct\ o\ map\ fst$

definition *list-map-rel-internal-def*:

$list\ map\ rel\ Rk\ Rv \equiv \langle \langle Rk, Rv \rangle prod\ rel \rangle list\ rel\ O\ br\ map\ of\ list\ map\ invar$

lemma *list-map-rel-def*:

$\langle Rk, Rv \rangle \text{list-map-rel} = \langle \langle Rk, Rv \rangle \text{prod-rel} \rangle \text{list-rel } O \text{ br map-of list-map-invar}$
 $\langle \text{proof} \rangle$

lemma *list-rel-Range*:

$\forall x' \in \text{set } l'. x' \in \text{Range } R \implies l' \in \text{Range } (\langle R \rangle \text{list-rel})$
 $\langle \text{proof} \rangle$

All finite maps can be represented

lemma *list-map-rel-range*:

$\text{Range } (\langle Rk, Rv \rangle \text{list-map-rel}) =$
 $\{m. \text{finite } (\text{dom } m) \wedge \text{dom } m \subseteq \text{Range } Rk \wedge \text{ran } m \subseteq \text{Range } Rv\}$
 $(\text{is } ?A = ?B)$
 $\langle \text{proof} \rangle$

lemmas $[\text{autoref-rel-intf}] = \text{REL-INTFI}[\text{of list-map-rel i-map}]$

lemma *list-map-rel-finite* $[\text{autoref-ga-rules}]$:

$\text{finite-map-rel } (\langle Rk, Rv \rangle \text{list-map-rel})$
 $\langle \text{proof} \rangle$

lemma *list-map-rel-sv* $[\text{relator-props}]$:

$\text{single-valued } Rk \implies \text{single-valued } Rv \implies$
 $\text{single-valued } (\langle Rk, Rv \rangle \text{list-map-rel})$
 $\langle \text{proof} \rangle$

Implementation

primrec *list-map-lookup* ::

$('k \Rightarrow 'k \Rightarrow \text{bool}) \Rightarrow 'k \Rightarrow ('k, 'v) \text{list-map} \Rightarrow 'v \text{ option}$ **where**
 $\text{list-map-lookup eq } \square = \text{None} \mid$
 $\text{list-map-lookup eq } k (y \# ys) =$
 $(\text{if eq } (\text{fst } y) k \text{ then Some } (\text{snd } y) \text{ else list-map-lookup eq } k ys)$

primrec *list-map-update-aux* :: $('k \Rightarrow 'k \Rightarrow \text{bool}) \Rightarrow 'k \Rightarrow 'v \Rightarrow$

$('k, 'v) \text{list-map} \Rightarrow ('k, 'v) \text{list-map} \Rightarrow ('k, 'v) \text{list-map}$ **where**
 $\text{list-map-update-aux eq } k v \square \text{ accu} = (k, v) \# \text{accu} \mid$
 $\text{list-map-update-aux eq } k v (x \# xs) \text{ accu} =$
 $(\text{if eq } (\text{fst } x) k$
 $\text{then } (k, v) \# xs @ \text{accu}$
 $\text{else list-map-update-aux eq } k v xs (x \# \text{accu}))$

definition *list-map-update* eq $k v m \equiv$

$\text{list-map-update-aux eq } k v m \square$

primrec *list-map-delete-aux* :: $('k \Rightarrow 'k \Rightarrow \text{bool}) \Rightarrow 'k \Rightarrow$

$('k, 'v) \text{list-map} \Rightarrow ('k, 'v) \text{list-map} \Rightarrow ('k, 'v) \text{list-map}$ **where**

$list-map-delete-aux\ eq\ k\ []\ accu = accu \mid$
 $list-map-delete-aux\ eq\ k\ (x\#xs)\ accu =$
 $(if\ eq\ (fst\ x)\ k$
 $then\ xs\ @\ accu$
 $else\ list-map-delete-aux\ eq\ k\ xs\ (x\#accu))$

definition $list-map-delete\ eq\ k\ m \equiv list-map-delete-aux\ eq\ k\ m\ []$

definition $list-map-isEmpty :: ('k, 'v)\ list-map \Rightarrow bool$
where $list-map-isEmpty \equiv List.null$

definition $list-map-isSng :: ('k, 'v)\ list-map \Rightarrow bool$
where $list-map-isSng\ m = (case\ m\ of\ [x] \Rightarrow True \mid - \Rightarrow False)$

definition $list-map-size :: ('k, 'v)\ list-map \Rightarrow nat$
where $list-map-size \equiv length$

definition $list-map-iteratei :: ('k, 'v)\ list-map \Rightarrow ('b \Rightarrow bool) \Rightarrow$
 $(('k \times 'v) \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'b$
where $list-map-iteratei \equiv foldli$

definition $list-map-to-list :: ('k, 'v)\ list-map \Rightarrow ('k \times 'v)\ list$
where $list-map-to-list = id$

Parametricity

lemma $list-map-autoref-empty[autoref-rules]:$
 $([],\ op-map-empty) \in \langle Rk, Rv \rangle list-map-rel$
 $\langle proof \rangle$

lemma $param-list-map-lookup[param]:$
 $(list-map-lookup, list-map-lookup) \in (Rk \rightarrow Rk \rightarrow bool-rel) \rightarrow$
 $Rk \rightarrow \langle \langle Rk, Rv \rangle prod-rel \rangle list-rel \rightarrow \langle Rv \rangle option-rel$
 $\langle proof \rangle$

lemma $list-map-autoref-lookup-aux:$
assumes $eq: GEN-OP\ eq\ (=)\ (Rk \rightarrow Rk \rightarrow Id)$
assumes $K: (k, k') \in Rk$
assumes $M: (m, m') \in \langle \langle Rk, Rv \rangle prod-rel \rangle list-rel$
shows $(list-map-lookup\ eq\ k\ m,\ op-map-lookup\ k'\ (map-of\ m'))$
 $\in \langle Rv \rangle option-rel$
 $\langle proof \rangle$

lemma $list-map-autoref-lookup[autoref-rules]:$
assumes $GEN-OP\ eq\ (=)\ (Rk \rightarrow Rk \rightarrow Id)$
shows $(list-map-lookup\ eq,\ op-map-lookup) \in$
 $Rk \rightarrow \langle Rk, Rv \rangle list-map-rel \rightarrow \langle Rv \rangle option-rel$
 $\langle proof \rangle$

lemma *param-list-map-update-aux*[*param*]:
 $(list\text{-}map\text{-}update\text{-}aux, list\text{-}map\text{-}update\text{-}aux) \in (Rk \rightarrow Rk \rightarrow bool\text{-}rel) \rightarrow$
 $Rk \rightarrow Rv \rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel \rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

lemma *param-list-map-update*[*param*]:
 $(list\text{-}map\text{-}update, list\text{-}map\text{-}update) \in (Rk \rightarrow Rk \rightarrow bool\text{-}rel) \rightarrow$
 $Rk \rightarrow Rv \rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel \rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

lemma *list-map-autoref-update-aux1*:
assumes *eq*: $(eq, (=)) \in Rk \rightarrow Rk \rightarrow Id$
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 $(list\text{-}map\text{-}update\text{-}aux\ eq\ k\ v\ m\ accu,$
 $list\text{-}map\text{-}update\text{-}aux\ (=)\ k'\ v'\ m'\ accu')$
 $\in \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

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

lemma *map-add-sng-right*: $m ++ [k \mapsto v] = m(k \mapsto v)$
 $\langle proof \rangle$

lemma *map-add-sng-right'*:
 $m ++ (\lambda a. \text{if } a = k \text{ then } Some\ v \text{ else } None) = m(k \mapsto v)$
 $\langle proof \rangle$

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 $(list\text{-}map\text{-}update\text{-}aux\ (=)\ k\ v\ m\ accu,$
 $accu' ++ op\text{-}map\text{-}update\ k'\ v'\ m')$
 $\in br\ map\text{-}of\ list\text{-}map\text{-}invar\ (is\ (?f\ m\ accu, -) \in -)$

$\langle proof \rangle$

lemma *list-map-autoref-update2*[*param*]:

shows $(list\text{-map}\text{-update} (=), op\text{-map}\text{-update}) \in Id \rightarrow Id \rightarrow$
 $br\ map\text{-of}\ list\text{-map}\text{-invar} \rightarrow br\ map\text{-of}\ list\text{-map}\text{-invar}$

$\langle proof \rangle$

lemma *list-map-autoref-update*[*autoref-rules*]:

assumes *eq*: *GEN-OP eq (=)* $(Rk \rightarrow Rk \rightarrow Id)$
shows $(list\text{-map}\text{-update}\ eq, op\text{-map}\text{-update}) \in$
 $Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle list\text{-map}\text{-rel} \rightarrow \langle Rk, Rv \rangle list\text{-map}\text{-rel}$

$\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

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*: $(l, m') \in \langle Rk, Rv \rangle list\text{-map}\text{-rel}$
defines *R-annot* $\equiv Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle list\text{-map}\text{-rel} \rightarrow \langle Rk, Rv \rangle list\text{-map}\text{-rel}$
shows
 $((k, v) \# l,$
 $(OP\ op\text{-map}\text{-update} ::: R\text{-annot}) \$ k \$ v \$ m')$
 $\in \langle Rk, Rv \rangle list\text{-map}\text{-rel}$

$\langle proof \rangle$

end

lemma *param-list-map-delete-aux*[*param*]:

$(list\text{-map}\text{-delete}\text{-aux}, list\text{-map}\text{-delete}\text{-aux}) \in (Rk \rightarrow Rk \rightarrow bool\text{-rel}) \rightarrow$
 $Rk \rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel} \rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel}$
 $\rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel}$

$\langle proof \rangle$

lemma *param-list-map-delete*[*param*]:

$(list\text{-map}\text{-delete}, list\text{-map}\text{-delete}) \in (Rk \rightarrow Rk \rightarrow bool\text{-rel}) \rightarrow$
 $Rk \rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel} \rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel}$

$\langle proof \rangle$

lemma *list-map-autoref-delete-aux1*:

assumes *eq*: $(eq, (=)) \in Rk \rightarrow Rk \rightarrow Id$
assumes *K*: $(k, k') \in Rk$
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 $(list\text{-map}\text{-delete}\text{-aux}\ eq\ k\ m\ accu,$
 $list\text{-map}\text{-delete}\text{-aux}\ (=)\ k'\ m'\ accu')$
 $\in \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel}$

$\langle proof \rangle$

lemma *list-map-autoref-delete1*[*param*]:

assumes $eq: (eq, (=)) \in Rk \rightarrow Rk \rightarrow Id$
shows $(list\text{-}map\text{-}delete\ eq, list\text{-}map\text{-}delete\ (=)) \in Rk \rightarrow$
 $\langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel \rightarrow \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

lemma $list\text{-}map\text{-}autoref\text{-}delete\text{-}aux2$:
assumes $K: (k, k') \in Id$
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 $(list\text{-}map\text{-}delete\text{-}aux\ (=)\ k\ m\ accu,$
 $accu' ++ op\text{-}map\text{-}delete\ k'\ m')$
 $\in br\ map\text{-}of\ list\text{-}map\text{-}invar\ (is\ (?f\ m\ accu, -) \in -)$
 $\langle proof \rangle$

lemma $list\text{-}map\text{-}autoref\text{-}delete2[param]$:
shows $(list\text{-}map\text{-}delete\ (=), op\text{-}map\text{-}delete) \in Id \rightarrow$
 $br\ map\text{-}of\ list\text{-}map\text{-}invar \rightarrow br\ map\text{-}of\ list\text{-}map\text{-}invar$
 $\langle proof \rangle$

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

lemma $list\text{-}map\text{-}autoref\text{-}isEmpty[autoref\text{-}rules]$:
shows $(list\text{-}map\text{-}isEmpty, op\text{-}map\text{-}isEmpty) \in$
 $\langle Rk, Rv \rangle list\text{-}map\text{-}rel \rightarrow bool\text{-}rel$
 $\langle proof \rangle$

lemma $param\text{-}list\text{-}map\text{-}isSng[param]$:
assumes $(l, l') \in \langle\langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
shows $(list\text{-}map\text{-}isSng\ l, list\text{-}map\text{-}isSng\ l') \in bool\text{-}rel$
 $\langle proof \rangle$

lemma $list\text{-}map\text{-}autoref\text{-}isSng\text{-}aux$:
assumes $(l', m') \in br\ map\text{-}of\ list\text{-}map\text{-}invar$
shows $(list\text{-}map\text{-}isSng\ l', op\text{-}map\text{-}isSng\ m') \in bool\text{-}rel$
 $\langle proof \rangle$

lemma $list\text{-}map\text{-}autoref\text{-}isSng[autoref\text{-}rules]$:
 $(list\text{-}map\text{-}isSng, op\text{-}map\text{-}isSng) \in \langle Rk, Rv \rangle list\text{-}map\text{-}rel \rightarrow bool\text{-}rel$
 $\langle proof \rangle$

lemma $list\text{-}map\text{-}autoref\text{-}size\text{-}aux$:

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 \Rightarrow ('k, 'v) *hashmap*
where \bigwedge *size*. *new-hashmap-with size* =
HashMap (new-array [] size) 0

definition *ahm-empty* :: nat \Rightarrow ('k, 'v) *hashmap*
where *ahm-empty def-size* \equiv *new-hashmap-with def-size*

definition *bucket-ok* :: 'k *bhc* \Rightarrow nat \Rightarrow nat \Rightarrow ('k \times 'v) *list* \Rightarrow bool
where *bucket-ok bhc len h kvs* = $(\forall k \in \text{fst } \text{' set } kvs. \text{ bhc len } k = h)$

definition *ahm-invar-aux* :: 'k *bhc* \Rightarrow nat \Rightarrow ('k \times 'v) *list array* \Rightarrow bool
where
ahm-invar-aux bhc n a \longleftrightarrow
 $(\forall h. h < \text{array-length } a \longrightarrow \text{bucket-ok bhc (array-length } a) h$
 $(\text{array-get } a h) \wedge \text{list-map-invar (array-get } a h)) \wedge$
 $\text{array-foldl } (\lambda _ n kvs. n + \text{size } kvs) 0 a = n \wedge$
 $\text{array-length } a > 1$

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

definition *ahm-lookup-aux* :: 'k *eq* \Rightarrow 'k *bhc* \Rightarrow
'k \Rightarrow ('k, 'v) *list-map array* \Rightarrow '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* \equiv $\lambda k. \text{ahm-lookup } (=) \text{ bhc } k m$

definition *ahm-map-rel-def-internal*:

$$\begin{aligned} \text{ahm-map-rel } Rk \ Rv &\equiv \{(HashMap \ a \ n, \ HashMap \ a' \ n) \mid a \ a' \ n \ n'\} \\ (a, a') &\in \langle\langle\langle Rk, Rv \rangle\text{prod-rel}\rangle\text{list-rel}\rangle\text{array-rel} \wedge (n, n') \in Id \end{aligned}$$

lemma *ahm-map-rel-def*: $\langle Rk, Rv \rangle \text{ ahm-map-rel} \equiv$

$$\begin{aligned} &\{(HashMap \ a \ n, \ HashMap \ a' \ n) \mid a \ a' \ n \ n'\} \\ &(a, a') \in \langle\langle\langle Rk, Rv \rangle\text{prod-rel}\rangle\text{list-rel}\rangle\text{array-rel} \wedge (n, n') \in Id \} \\ &\langle\text{proof}\rangle \end{aligned}$$

definition *ahm-map-rel'-def*:

$$\text{ahm-map-rel}' \text{ bhc} \equiv \text{br} \ (\text{ahm-}\alpha \ \text{bhc}) \ (\text{ahm-invar} \ \text{bhc})$$

definition *ahm-rel-def-internal*: $\text{ahm-rel} \ \text{bhc} \ Rk \ Rv =$

$$\langle Rk, Rv \rangle \text{ ahm-map-rel} \ O \ \text{ahm-map-rel}' \ (\text{abstract-bounded-hashcode} \ Rk \ \text{bhc})$$

lemma *ahm-rel-def*: $\langle Rk, Rv \rangle \text{ ahm-rel} \ \text{bhc} \equiv$

$$\begin{aligned} &\langle Rk, Rv \rangle \text{ ahm-map-rel} \ O \ \text{ahm-map-rel}' \ (\text{abstract-bounded-hashcode} \ Rk \ \text{bhc}) \\ &\langle\text{proof}\rangle \end{aligned}$$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[of *ahm-rel bhc i-map*] **for** *bhc*

abbreviation *dflt-ahm-rel* $\equiv \text{ahm-rel} \ \text{bounded-hashcode-nat}$

primrec *ahm-iteratei-aux* :: $((k \times v) \ \text{list} \ \text{array}) \Rightarrow (k \times v, \sigma) \ \text{set-iterator}$

where *ahm-iteratei-aux* (*Array xs*) *c f* = *foldli (concat xs) c f*

primrec *ahm-iteratei* :: $((k, v) \ \text{hashmap}) \Rightarrow ((k \times v), \sigma) \ \text{set-iterator}$

where

$$\text{ahm-iteratei} \ (\text{HashMap} \ a \ n) = \text{ahm-iteratei-aux} \ a$$

definition *ahm-rehash-aux'* :: $k \ \text{bhc} \Rightarrow \text{nat} \Rightarrow k \times v \Rightarrow$

$$(k \times v) \ \text{list} \ \text{array} \Rightarrow (k \times v) \ \text{list} \ \text{array}$$

where

$$\text{ahm-rehash-aux}' \ \text{bhc} \ n \ kv \ a =$$

$$(\text{let } h = \text{bhc} \ n \ (\text{fst} \ kv))$$

$$\text{in } \text{array-set} \ a \ h \ (kv \ \# \ \text{array-get} \ a \ h)$$

definition *ahm-rehash-aux* :: $k \ \text{bhc} \Rightarrow (k \times v) \ \text{list} \ \text{array} \Rightarrow \text{nat} \Rightarrow$

$$(k \times v) \ \text{list} \ \text{array}$$

where

$$\text{ahm-rehash-aux} \ \text{bhc} \ a \ sz = \text{ahm-iteratei-aux} \ a \ (\lambda x. \ \text{True})$$

$$(\text{ahm-rehash-aux}' \ \text{bhc} \ sz) \ (\text{new-array} \ [] \ sz)$$

primrec *ahm-rehash* :: $k \ \text{bhc} \Rightarrow (k, v) \ \text{hashmap} \Rightarrow \text{nat} \Rightarrow (k, v) \ \text{hashmap}$

where *ahm-rehash bhc* (*HashMap a n*) *sz* = *HashMap (ahm-rehash-aux bhc a sz)*

n

primrec *hm-grow* :: $(k, v) \ \text{hashmap} \Rightarrow \text{nat}$

where *hm-grow* (*HashMap a n*) = $2 * \text{array-length} \ a + 3$

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*:

$\llbracket \text{bucket-ok } \text{bhc } \text{len } h \text{ } xs; (k, v) \in \text{set } xs \rrbracket$
 $\implies \text{bhc } \text{len } k = h$
 <proof>

lemma *bucket-okI*:

$(\bigwedge k. k \in \text{fst } \text{'set } kvs \implies \text{bhc } \text{len } k = h) \implies \text{bucket-ok } \text{bhc } \text{len } h \text{ } kvs$
 <proof>

Parametricity

lemma *param-HashMap[param]*: $(\text{HashMap}, \text{HashMap}) \in$

$\langle \langle \langle Rk, Rv \rangle \text{prod-rel} \rangle \text{list-rel} \rangle \text{array-rel} \rightarrow \text{nat-rel} \rightarrow \langle Rk, Rv \rangle \text{ahm-map-rel}$
 <proof>

lemma *param-case-hashmap[param]*: $(\text{case-hashmap}, \text{case-hashmap}) \in$

$\langle \langle \langle Rk, Rv \rangle \text{prod-rel} \rangle \text{list-rel} \rangle \text{array-rel} \rightarrow \text{nat-rel} \rightarrow R \rightarrow$
 $\langle Rk, Rv \rangle \text{ahm-map-rel} \rightarrow R$

<proof>

lemma *rec-hashmap-is-case[simp]*: $\text{rec-hashmap} = \text{case-hashmap}$

<proof>

ahm-invar

lemma *ahm-invar-auxD*:

assumes *ahm-invar-aux* $\text{bhc } n \ a$

shows $\bigwedge h. h < \text{array-length } a \implies$

$\text{bucket-ok } \text{bhc } (\text{array-length } a) \ h \ (\text{array-get } a \ h)$ **and**

$\bigwedge h. h < \text{array-length } a \implies$

$\text{list-map-invar } (\text{array-get } a \ h)$ **and**

$n = \text{array-foldl } (\lambda \cdot n \ kvs. n + \text{length } kvs) \ 0 \ a$ **and**

$\text{array-length } a > 1$

<proof>

lemma *ahm-invar-auxE*:

assumes *ahm-invar-aux* $\text{bhc } n \ a$

obtains $\forall h. h < \text{array-length } a \implies$

$\text{bucket-ok } \text{bhc } (\text{array-length } a) \ h \ (\text{array-get } a \ h) \wedge$

$\text{list-map-invar } (\text{array-get } a \ h)$ **and**

$n = \text{array-foldl } (\lambda \cdot n \ kvs. n + \text{length } kvs) \ 0 \ a$ **and**

$\text{array-length } a > 1$

<proof>

lemma *ahm-invar-auxI*:

$\llbracket \bigwedge h. h < \text{array-length } a \implies$

$\text{bucket-ok } \text{bhc } (\text{array-length } a) \ h \ (\text{array-get } a \ h);$

$\bigwedge h. h < \text{array-length } a \implies \text{list-map-invar } (\text{array-get } a \ h);$

$n = \text{array-foldl } (\lambda \cdot n \ kvs. n + \text{length } kvs) \ 0 \ a; \text{array-length } a > 1 \rrbracket$

$\implies \text{ahm-invar-aux } \text{bhc } n \ a$

<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* ≡
 (λ*k*. *ahm-lookup-aux* (=) *bhc k a*)

lemma *ahm-α-aux-def2*: *ahm-α-aux* *bhc a* = (λ*k*. *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 \implies *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*) \in *ahm-map-rel'* *bhc*
 ⟨*proof*⟩

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

lemma *autoref-ahm-empty[autoref-rules]*:
fixes *Rk* :: ('*kc* \times '*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*) \in ⟨*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'* \equiv *abstract-bounded-hashcode* *Rk* *bhc*
assumes *inv*: *ahm-invar* *bhc'* *m'*
assumes *K*: (*k*,*k'*) \in *Rk*
assumes *M*: (*m*,*m'*) \in ⟨*Rk*,*Rv*⟩*ahm-map-rel*
shows (*ahm-lookup* *eq* *bhc* *k* *m*, *ahm-lookup* (=) *bhc'* *k'* *m'*) \in
 ⟨*Rv*⟩*option-rel*
 ⟨*proof*⟩

lemma *ahm-lookup-impl*:
assumes *bhc*: *is-bounded-hashcode* *Id* (=) *bhc*
shows (*ahm-lookup* (=) *bhc*, *op-map-lookup*) \in *Id* \rightarrow *ahm-map-rel'* *bhc* \rightarrow *Id*
 ⟨*proof*⟩

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

assumes
 $bhc[\text{unfolded autoref-tag-defs}] : \text{SIDE-GEN-ALGO } (is\text{-bounded-hashcode } Rk \text{ eq } bhc)$
shows $(ahm\text{-lookup eq } bhc, op\text{-map-lookup})$
 $\in Rk \rightarrow \langle Rk, Rv \rangle ahm\text{-rel } bhc \rightarrow \langle Rv \rangle option\text{-rel}$
 $\langle proof \rangle$

ahm-iteratei

abbreviation *ahm-to-list* $\equiv it\text{-to-list } ahm\text{-iteratei}$

lemma *param-ahm-iteratei-aux*[*param*]:

$(ahm\text{-iteratei-aux}, ahm\text{-iteratei-aux}) \in \langle \langle Ra \rangle list\text{-rel} \rangle array\text{-rel} \rightarrow$
 $(Rb \rightarrow bool\text{-rel}) \rightarrow (Ra \rightarrow Rb \rightarrow Rb) \rightarrow Rb \rightarrow Rb$
 $\langle proof \rangle$

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

$(ahm\text{-iteratei}, ahm\text{-iteratei}) \in \langle Rk, Rv \rangle ahm\text{-map-rel} \rightarrow$
 $(Rb \rightarrow bool\text{-rel}) \rightarrow (\langle Rk, Rv \rangle prod\text{-rel} \rightarrow Rb \rightarrow Rb) \rightarrow Rb \rightarrow Rb$
 $\langle proof \rangle$

lemma *param-ahm-to-list*[*param*]:

$(ahm\text{-to-list}, ahm\text{-to-list}) \in$
 $\langle Rk, Rv \rangle ahm\text{-map-rel} \rightarrow \langle \langle Rk, Rv \rangle prod\text{-rel} \rangle list\text{-rel}$
 $\langle proof \rangle$

lemma *ahm-to-list-distinct*[*simp, intro*]:

assumes $bhc : is\text{-bounded-hashcode } Id (=) bhc$
assumes $inv : ahm\text{-invar } bhc m$
shows $distinct (ahm\text{-to-list } m)$
 $\langle proof \rangle$

lemma *set-ahm-to-list*:

assumes $bhc : is\text{-bounded-hashcode } Id (=) bhc$
assumes $ref : (m, m') \in ahm\text{-map-rel}' bhc$
shows $map\text{-to-set } m' = set (ahm\text{-to-list } m)$
 $\langle proof \rangle$

lemma *ahm-iteratei-aux-impl*:

assumes $inv : ahm\text{-invar-aux } bhc n a$
and $bhc : is\text{-bounded-hashcode } Id (=) bhc$
shows $map\text{-iterator } (ahm\text{-iteratei-aux } a) (ahm\text{-}\alpha\text{-aux } bhc a)$
 $\langle proof \rangle$

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*
 $(\lambda x. \text{foldli } x \text{ } c \text{ } f) \sigma$
<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*: $k \notin \text{fst } \text{'set } (\text{array-get } a \text{ } (\text{bhc } (\text{array-length } a) \text{ } k))$
shows *ahm-invar-aux* *bhc* (*Suc* *n*) (*ahm-rehash-aux'* *bhc* (*array-length* *a*) (*k*, *v*)
a)
 $(\text{is } \text{ahm-invar-aux } \text{bhc } - \text{ ?}a)$
<proof>

lemma *ahm-rehash-aux-correct*:

fixes *a* :: $('k \times 'v)$ *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 $ahm\text{-invar } bhc (ahm\text{-rehash } bhc hm sz)$
 $ahm\text{-}\alpha bhc (ahm\text{-rehash } bhc hm sz) = ahm\text{-}\alpha bhc hm$
 $\langle proof \rangle$

ahm-update

lemma *param-hm-grow*[*param*]:
 $(hm\text{-grow}, hm\text{-grow}) \in \langle Rk, Rv \rangle ahm\text{-map-rel} \rightarrow nat\text{-rel}$
 $\langle proof \rangle$

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

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

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

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

lemma *param-ahm-rehash-aux*[*param*]:

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

lemma *param-ahm-rehash*[$param$]:
assumes bhc : *is-bounded-hashcode* Rk eq bhc
assumes bhc -rel: $(bhc, bhc') \in nat\text{-rel} \rightarrow Rk \rightarrow nat\text{-rel}$
assumes M : $(m, m') \in \langle Rk, Rv \rangle ahm\text{-map-rel}$
assumes N : $(n, n') \in nat\text{-rel} \quad 1 < n$
shows $(ahm\text{-rehash } bhc \ m \ n, ahm\text{-rehash } bhc' \ m' \ n') \in$
 $\langle Rk, Rv \rangle ahm\text{-map-rel}$
 $\langle proof \rangle$

lemma *param-load-factor*[$param$]:
 $(load\text{-factor}, load\text{-factor}) \in nat\text{-rel}$
 $\langle proof \rangle$

lemma *param-ahm-filled*[$param$]:
 $(ahm\text{-filled}, ahm\text{-filled}) \in \langle Rk, Rv \rangle ahm\text{-map-rel} \rightarrow bool\text{-rel}$
 $\langle proof \rangle$

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

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

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 $(ahm-update (=) bhc, op-map-update) \in (Id::('k \times 'k) set) \rightarrow$
 $(Id::('v \times 'v) set) \rightarrow ahm-map-rel' bhc \rightarrow ahm-map-rel' bhc$
 ⟨proof⟩

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

assumes bhc [*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode* $Rk eq bhc$)
shows $(ahm-update eq bhc, op-map-update) \in$
 $Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle ahm-rel bhc \rightarrow \langle Rk, Rv \rangle ahm-rel bhc$
 ⟨proof⟩

ahm-delete

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

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

lemma *ahm-delete-impl*:

assumes bhc : *is-bounded-hashcode* $Id (=) bhc$
shows $(ahm-delete (=) bhc, op-map-delete) \in (Id::('k \times 'k) set) \rightarrow$
 $ahm-map-rel' bhc \rightarrow ahm-map-rel' bhc$
 ⟨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*) \in
 $Rk \rightarrow \langle Rk, Rv \rangle \text{ahm-rel } bhc \rightarrow \langle Rk, Rv \rangle \text{ahm-rel } bhc$
<proof>

Various simple operations

lemma *param-ahm-isEmpty*[*param*]:
(*ahm-isEmpty, ahm-isEmpty*) $\in \langle Rk, Rv \rangle \text{ahm-map-rel} \rightarrow \text{bool-rel}$
<proof>

lemma *param-ahm-isSng*[*param*]:
(*ahm-isSng, ahm-isSng*) $\in \langle Rk, Rv \rangle \text{ahm-map-rel} \rightarrow \text{bool-rel}$
<proof>

lemma *param-ahm-size*[*param*]:
(*ahm-size, ahm-size*) $\in \langle Rk, Rv \rangle \text{ahm-map-rel} \rightarrow \text{nat-rel}$
<proof>

lemma *ahm-isEmpty-impl*:
assumes *is-bounded-hashcode Id (=) bhc*
shows (*ahm-isEmpty, op-map-isEmpty*) $\in \text{ahm-map-rel}' bhc \rightarrow \text{bool-rel}$
<proof>

lemma *ahm-isSng-impl*:
assumes *is-bounded-hashcode Id (=) bhc*
shows (*ahm-isSng, op-map-isSng*) $\in \text{ahm-map-rel}' bhc \rightarrow \text{bool-rel}$
<proof>

lemma *ahm-size-impl*:
assumes *is-bounded-hashcode Id (=) bhc*
shows (*ahm-size, op-map-size*) $\in \text{ahm-map-rel}' bhc \rightarrow \text{nat-rel}$
<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*) $\in \langle Rk, Rv \rangle \text{ahm-rel } bhc \rightarrow \text{bool-rel}$
<proof>

lemma *autoref-ahm-isSng*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)

shows $(ahm-isSng, op-map-isSng) \in \langle Rk, Rv \rangle ahm-rel\ bhc \rightarrow bool-rel$
 $\langle proof \rangle$

lemma *autoref-ahm-size*[*autoref-rules*]:
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows $(ahm-size, op-map-size) \in \langle Rk, Rv \rangle ahm-rel\ bhc \rightarrow nat-rel$
 $\langle proof \rangle$

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

lemma *ahm-rel-sv*[*relator-props*]:
 $\llbracket single-valued\ Rk; single-valued\ Rv \rrbracket$
 $\implies single-valued\ (\langle Rk, Rv \rangle ahm-rel\ bhc)$
 $\langle proof \rangle$

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

Proper iterator proofs

lemma *pi-ahm*[*icf-proper-iteratorI*]:
proper-it (*ahm-iteratei m*) (*ahm-iteratei m*)
 $\langle proof \rangle$

lemma *pi'-ahm*[*icf-proper-iteratorI*]:
proper-it' *ahm-iteratei ahm-iteratei*
 $\langle proof \rangle$

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** $Rk :: (-\times-::hashable)$ *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**

$(color.R, color.R) \in color-rel$
 $| (color.B, color.B) \in color-rel$

inductive-cases *color-rel-elim*:

$(x, color.R) \in color-rel$
 $(x, color.B) \in color-rel$
 $(color.R, y) \in color-rel$
 $(color.B, y) \in color-rel$

thm *color-rel-elim*

lemma *param-color*[*param*]:

$(color.R, color.R) \in color-rel$
 $(color.B, color.B) \in color-rel$
 $(case-color, case-color) \in R \rightarrow R \rightarrow color-rel \rightarrow R$
 $\langle proof \rangle$

inductive-set *rbt-rel-aux* **for** $Ra\ Rb$ **where**

$(rbt.Empty, rbt.Empty) \in rbt-rel-aux\ Ra\ Rb$
 $| \llbracket (c, c') \in color-rel;$
 $(l, l') \in rbt-rel-aux\ Ra\ Rb; (a, a') \in Ra; (b, b') \in Rb;$
 $(r, r') \in rbt-rel-aux\ Ra\ Rb \rrbracket$
 $\implies (rbt.Branch\ c\ l\ a\ b\ r, rbt.Branch\ c'\ l'\ a'\ b'\ r') \in rbt-rel-aux\ Ra\ Rb$

inductive-cases *rbt-rel-aux-elim*:

$(x, rbt.Empty) \in rbt-rel-aux\ Ra\ Rb$
 $(rbt.Empty, x') \in rbt-rel-aux\ Ra\ Rb$
 $(rbt.Branch\ c\ l\ a\ b\ r, x') \in rbt-rel-aux\ Ra\ Rb$
 $(x, rbt.Branch\ c'\ l'\ a'\ b'\ r') \in rbt-rel-aux\ Ra\ Rb$

definition $rbt-rel \equiv rbt-rel-aux$

lemma $rbt-rel-aux-fold$: $rbt-rel-aux\ Ra\ Rb \equiv \langle Ra, Rb \rangle rbt-rel$
 $\langle proof \rangle$

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 $param-rbt1[param]$:

$(rbt.Empty, rbt.Empty) \in \langle Ra, Rb \rangle rbt-rel$

$(rbt.Branch, rbt.Branch) \in$

$color-rel \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

lemma $param-case-rbt[param]$:

$(case-rbt, case-rbt) \in$

$Ra \rightarrow (color-rel \rightarrow \langle Rb, Rc \rangle rbt-rel \rightarrow Rb \rightarrow Rc \rightarrow \langle Rb, Rc \rangle rbt-rel \rightarrow Ra)$

$\rightarrow \langle Rb, Rc \rangle rbt-rel \rightarrow Ra$

$\langle proof \rangle$

lemma $param-rec-rbt[param]$: $(rec-rbt, rec-rbt) \in$

$Ra \rightarrow (color-rel \rightarrow \langle Rb, Rc \rangle rbt-rel \rightarrow Rb \rightarrow Rc \rightarrow \langle Rb, Rc \rangle rbt-rel$

$\rightarrow Ra \rightarrow Ra \rightarrow Ra) \rightarrow \langle Rb, Rc \rangle rbt-rel \rightarrow Ra$

$\langle proof \rangle$

lemma $param-paint[param]$:

$(paint, paint) \in color-rel \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

lemma $param-balance[param]$:

shows $(balance, balance) \in$

$\langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

lemma $param-rbt-ins[param]$:

fixes $less$

assumes $param-less[param]$: $(less, less') \in Ra \rightarrow Ra \rightarrow Id$

shows $(ord.rbt-ins\ less, ord.rbt-ins\ less') \in$

$(Ra \rightarrow Rb \rightarrow Rb \rightarrow Rb) \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

term $rbt-insert$

lemma $param-rbt-insert[param]$:

fixes $less$

assumes $param-less[param]$: $(less, less') \in Ra \rightarrow Ra \rightarrow Id$

shows $(ord.rbt-insert\ less, ord.rbt-insert\ less') \in$

$Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $\langle proof \rangle$

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

fixes *less*

assumes *param-less*[*param*]: (*less, less'*) $\in Ra \rightarrow Ra \rightarrow Id$

shows (*ord.rbt-lookup less, ord.rbt-lookup less'*) \in

$\langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow \langle Rb \rangle option-rel$

$\langle proof \rangle$

term *balance-left*

lemma *param-balance-left*[*param*]:

(*balance-left, balance-left*) \in

$\langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

term *balance-right*

lemma *param-balance-right*[*param*]:

(*balance-right, balance-right*) \in

$\langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

lemma *param-combine*[*param*]:

(*combine, combine*) $\in \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

lemma *ih-aux1*: $\llbracket (a', b) \in R; a' = a \rrbracket \implies (a, b) \in R$ $\langle proof \rangle$

lemma *is-eq*: $a = b \implies a = b$ $\langle proof \rangle$

lemma *param-rbt-del-aux*:

fixes *br*

fixes *less*

assumes *param-less*[*param*]: (*less, less'*) $\in Ra \rightarrow Ra \rightarrow Id$

shows

$\llbracket (ak1, ak1') \in Ra; (al, al') \in \langle Ra, Rb \rangle rbt-rel; (ak, ak') \in Ra;$

$(av, av') \in Rb; (ar, ar') \in \langle Ra, Rb \rangle rbt-rel$

$\rrbracket \implies (ord.rbt-del-from-left less ak1 al ak av ar,$

$ord.rbt-del-from-left less' ak1' al' ak' av' ar')$

$\in \langle Ra, Rb \rangle rbt-rel$

$\llbracket (bk1, bk1') \in Ra; (bl, bl') \in \langle Ra, Rb \rangle rbt-rel; (bk, bk') \in Ra;$

$(bv, bv') \in Rb; (br, br') \in \langle Ra, Rb \rangle rbt-rel$

$\rrbracket \implies (ord.rbt-del-from-right less bk1 bl bk bv br,$

$ord.rbt-del-from-right less' bk1' bl' bk' bv' br')$

$\in \langle Ra, Rb \rangle rbt-rel$

$\llbracket (ck, ck') \in Ra; (ct, ct') \in \langle Ra, Rb \rangle rbt-rel \rrbracket$

$\implies (ord.rbt-del less ck ct, ord.rbt-del less' ck' ct') \in \langle Ra, Rb \rangle rbt-rel$

$\langle proof \rangle$

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

fixes *less*

assumes *param-less*: $(less, less') \in Ra \rightarrow Ra \rightarrow Id$

shows

$(ord.rbt-del-from-left\ less, ord.rbt-del-from-left\ less') \in$
 $Ra \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $(ord.rbt-del-from-right\ less, ord.rbt-del-from-right\ less') \in$
 $Ra \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $(ord.rbt-del\ less, ord.rbt-del\ less') \in$
 $Ra \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $\langle proof \rangle$

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

fixes *less*

assumes *param-less*[*param*]: $(less, less') \in Ra \rightarrow Ra \rightarrow Id$

shows $(ord.rbt-delete\ less, ord.rbt-delete\ less') \in$

$\in Ra \rightarrow \langle Ra, Rb \rangle rbt-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $\langle proof \rangle$

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

abbreviation *compare-rel* :: $(RBT-Impl.compare \times -)$ *set*

where *compare-rel* $\equiv Id$

lemma *param-compare*[*param*]:

$(RBT-Impl.LT, RBT-Impl.LT) \in compare-rel$
 $(RBT-Impl.GT, RBT-Impl.GT) \in compare-rel$
 $(RBT-Impl.EQ, RBT-Impl.EQ) \in compare-rel$
 $(RBT-Impl.case-compare, RBT-Impl.case-compare) \in R \rightarrow R \rightarrow R \rightarrow compare-rel \rightarrow R$
 $\langle proof \rangle$

lemma *param-rbtreeify-aux*[*param*]:

$\llbracket n \leq length\ kvs; (n, n') \in nat-rel; (kvs, kvs') \in \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rrbracket$
 $\implies (rbtreeify-f\ n\ kvs, rbtreeify-f\ n'\ kvs')$
 $\in \langle \langle Ra, Rb \rangle rbt-rel, \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rangle prod-rel$
 $\llbracket n \leq Suc\ (length\ kvs); (n, n') \in nat-rel; (kvs, kvs') \in \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rrbracket$
 $\implies (rbtreeify-g\ n\ kvs, rbtreeify-g\ n'\ kvs')$
 $\in \langle \langle Ra, Rb \rangle rbt-rel, \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rangle prod-rel$
 $\langle proof \rangle$

lemma *param-rbtreeify*[*param*]:

$(rbtreeify, rbtreeify) \in \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rightarrow \langle Ra, Rb \rangle rbt-rel$
 $\langle proof \rangle$

lemma *param-sunion-with*[*param*]:

fixes *less*

shows $\llbracket (less, less') \in Ra \rightarrow Ra \rightarrow Id;$

$(f, f') \in (Ra \rightarrow Rb \rightarrow Rb \rightarrow Rb); (a, a') \in \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel;$

$(b, b') \in \langle \langle Ra, Rb \rangle prod-rel \rangle list-rel \rrbracket$

$\implies (ord.sunion-with\ less\ f\ a\ b, ord.sunion-with\ less'\ f'\ a'\ b') \in$

$\langle\langle Ra, Rb \rangle\text{prod-rel}\rangle\text{list-rel}$
 $\langle\text{proof}\rangle$

lemma *skip-red-alt*:
 $RBT\text{-Impl.skip-red } t = (\text{case } t \text{ of}$
 $(\text{Branch color.R l k v r}) \Rightarrow l$
 $| - \Rightarrow t)$
 $\langle\text{proof}\rangle$

function *compare-height* ::
 $('a, 'b) RBT\text{-Impl.rbt} \Rightarrow ('a, 'b) RBT\text{-Impl.rbt} \Rightarrow ('a, 'b) RBT\text{-Impl.rbt} \Rightarrow$
 $('a, 'b) RBT\text{-Impl.rbt} \Rightarrow RBT\text{-Impl.compare}$
where
 $\text{compare-height } sx \ s \ t \ tx =$
 $(\text{case } (RBT\text{-Impl.skip-red } sx, RBT\text{-Impl.skip-red } s, RBT\text{-Impl.skip-red } t, RBT\text{-Impl.skip-red}$
 $tx) \text{ of}$
 $(\text{Branch - } sx' \ - \ - \ -, \text{Branch - } s' \ - \ - \ -, \text{Branch - } t' \ - \ - \ -, \text{Branch - } tx' \ - \ - \ -) \Rightarrow$
 $\text{compare-height } (RBT\text{-Impl.skip-black } sx') \ s' \ t' \ (RBT\text{-Impl.skip-black } tx')$
 $| (-, \text{rbt.Empty}, -, \text{Branch - - - -}) \Rightarrow RBT\text{-Impl.LT}$
 $| (\text{Branch - - - -}, -, \text{rbt.Empty}, -) \Rightarrow RBT\text{-Impl.GT}$
 $| (\text{Branch - } sx' \ - \ - \ -, \text{Branch - } s' \ - \ - \ -, \text{Branch - } t' \ - \ - \ -, \text{rbt.Empty}) \Rightarrow$
 $\text{compare-height } (RBT\text{-Impl.skip-black } sx') \ s' \ t' \ \text{rbt.Empty}$
 $| (\text{rbt.Empty}, \text{Branch - } s' \ - \ - \ -, \text{Branch - } t' \ - \ - \ -, \text{Branch - } tx' \ - \ - \ -) \Rightarrow$
 $\text{compare-height } \text{rbt.Empty } s' \ t' \ (RBT\text{-Impl.skip-black } tx')$
 $| - \Rightarrow RBT\text{-Impl.EQ}$
 $\langle\text{proof}\rangle$

lemma *skip-red-size*: $\text{size } (RBT\text{-Impl.skip-red } b) \leq \text{size } b$
 $\langle\text{proof}\rangle$

lemma *skip-black-size*: $\text{size } (RBT\text{-Impl.skip-black } b) \leq \text{size } b$
 $\langle\text{proof}\rangle$

termination
 $\langle\text{proof}\rangle$

lemmas $[\text{simp del}] = \text{compare-height.simps}$

lemma *compare-height-alt*:
 $RBT\text{-Impl.compare-height } sx \ s \ t \ tx = \text{compare-height } sx \ s \ t \ tx$
 $\langle\text{proof}\rangle$

term $RBT\text{-Impl.skip-red}$

lemma *param-skip-red*[*param*]: $(RBT\text{-Impl.skip-red}, RBT\text{-Impl.skip-red})$
 $\in \langle Rk, Rv \rangle \text{rbt-rel} \rightarrow \langle Rk, Rv \rangle \text{rbt-rel}$
 $\langle\text{proof}\rangle$

lemma *param-skip-black*[*param*]: $(RBT\text{-Impl.skip-black}, RBT\text{-Impl.skip-black})$
 $\in \langle Rk, Rv \rangle \text{rbt-rel} \rightarrow \langle Rk, Rv \rangle \text{rbt-rel}$

$\langle proof \rangle$

term *case-rbt*

lemma *param-case-rbt'*:

assumes $(t, t') \in \langle Rk, Rv \rangle rbt\text{-rel}$

assumes $\llbracket t = rbt.Empty; t' = rbt.Empty \rrbracket \implies (fl, fl') \in R$

assumes $\bigwedge c l k v r c' l' k' v' r'. \llbracket$

$t = Branch\ c\ l\ k\ v\ r; t' = Branch\ c'\ l'\ k'\ v'\ r';$

$(c, c') \in color\text{-rel};$

$(l, l') \in \langle Rk, Rv \rangle rbt\text{-rel}; (k, k') \in Rk; (v, v') \in Rv; (r, r') \in \langle Rk, Rv \rangle rbt\text{-rel}$

$\rrbracket \implies (fb\ c\ l\ k\ v\ r, fb'\ c'\ l'\ k'\ v'\ r') \in R$

shows $(case\text{-rbt}\ fl\ fb\ t, case\text{-rbt}\ fl'\ fb'\ t') \in R$

$\langle proof \rangle$

lemma *compare-height-param-aux*[*param*]:

$\llbracket (sx, sx') \in \langle Rk, Rv \rangle rbt\text{-rel}; (s, s') \in \langle Rk, Rv \rangle rbt\text{-rel};$

$(t, t') \in \langle Rk, Rv \rangle rbt\text{-rel}; (tx, tx') \in \langle Rk, Rv \rangle rbt\text{-rel} \rrbracket$

$\implies (compare\text{-height}\ sx\ s\ t\ tx, compare\text{-height}\ sx'\ s'\ t'\ tx') \in compare\text{-rel}$

$\langle proof \rangle$

lemma *compare-height-param*[*param*]:

$(RBT\text{-Impl}.compare\text{-height}, RBT\text{-Impl}.compare\text{-height}) \in$

$\langle Rk, Rv \rangle rbt\text{-rel} \rightarrow \langle Rk, Rv \rangle rbt\text{-rel} \rightarrow \langle Rk, Rv \rangle rbt\text{-rel} \rightarrow \langle Rk, Rv \rangle rbt\text{-rel}$

$\rightarrow compare\text{-rel}$

$\langle proof \rangle$

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

fixes *less*

assumes *param-less*[*param*]: $(less, less') \in Ra \rightarrow Ra \rightarrow Id$

shows $(ord.rbt\text{-union}\ less, ord.rbt\text{-union}\ less')$

$\in \langle Ra, Rb \rangle rbt\text{-rel} \rightarrow \langle Ra, Rb \rangle rbt\text{-rel} \rightarrow \langle Ra, Rb \rangle rbt\text{-rel}$

$\langle proof \rangle$

term *rm-iterateoi*

lemma *param-rm-iterateoi*[*param*]: $(rm\text{-iterateoi}, rm\text{-iterateoi})$

$\in \langle Ra, Rb \rangle rbt\text{-rel} \rightarrow (Rc \rightarrow Id) \rightarrow (\langle Ra, Rb \rangle prod\text{-rel} \rightarrow Rc \rightarrow Rc) \rightarrow Rc \rightarrow Rc$

$\langle proof \rangle$

lemma *param-rm-reverse-iterateoi*[*param*]:

$(rm\text{-reverse-iterateoi}, rm\text{-reverse-iterateoi})$

$\in \langle Ra, Rb \rangle rbt\text{-rel} \rightarrow (Rc \rightarrow Id) \rightarrow (\langle Ra, Rb \rangle prod\text{-rel} \rightarrow Rc \rightarrow Rc) \rightarrow Rc \rightarrow Rc$

$\langle proof \rangle$

lemma *param-color-eq*[*param*]:

$((=), (=)) \in color\text{-rel} \rightarrow color\text{-rel} \rightarrow Id$

$\langle proof \rangle$

lemma *param-color-of*[*param*]:

$(color-of, color-of) \in \langle Rk, Rv \rangle rbt-rel \rightarrow color-rel$
 $\langle proof \rangle$

term *bheight*

lemma *param-bheight*[*param*]:
 $(bheight, bheight) \in \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

lemma *inv1-param*[*param*]: $(inv1, inv1) \in \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

lemma *inv2-param*[*param*]: $(inv2, inv2) \in \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

term *ord.rbt-less*

lemma *rbt-less-param*[*param*]: $(ord.rbt-less, ord.rbt-less) \in$
 $(Rk \rightarrow Rk \rightarrow Id) \rightarrow Rk \rightarrow \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

term *ord.rbt-greater*

lemma *rbt-greater-param*[*param*]: $(ord.rbt-greater, ord.rbt-greater) \in$
 $(Rk \rightarrow Rk \rightarrow Id) \rightarrow Rk \rightarrow \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

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

$(ord.rbt-sorted, ord.rbt-sorted) \in (Rk \rightarrow Rk \rightarrow Id) \rightarrow \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

lemma *is-rbt-param*[*param*]: $(ord.is-rbt, ord.is-rbt) \in$

$(Rk \rightarrow Rk \rightarrow Id) \rightarrow \langle Rk, Rv \rangle rbt-rel \rightarrow Id$
 $\langle proof \rangle$

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

lemma (**in** *linorder*) *rbt-map-impl*:

$(rbt.Empty, Map.empty) \in rbt-map-rel' (<)$
 $(rbt.insert, \lambda k v m. m(k \mapsto v))$
 $\in Id \rightarrow Id \rightarrow rbt-map-rel' (<) \rightarrow rbt-map-rel' (<)$
 $(rbt.lookup, \lambda m k. m k) \in rbt-map-rel' (<) \rightarrow Id \rightarrow \langle Id \rangle option-rel$
 $(rbt.delete, \lambda k m. m|'(-\{k\})) \in Id \rightarrow rbt-map-rel' (<) \rightarrow rbt-map-rel' (<)$
 $(rbt.union, (++)$
 $\in rbt-map-rel' (<) \rightarrow rbt-map-rel' (<) \rightarrow rbt-map-rel' (<)$
 $\langle proof \rangle$

lemma *sorted-wrt-keys-true*[*simp*]: *sorted-wrt* $(\lambda(-, -) (-, -). True) l$

$\langle proof \rangle$

definition *rbt-map-rel-def-internal*:

$rbt\text{-map-rel } lt \ Rk \ Rv \equiv \langle Rk, Rv \rangle rbt\text{-rel } O \ rbt\text{-map-rel}' \ lt$

lemma *rbt-map-rel-def*:

$\langle Rk, Rv \rangle rbt\text{-map-rel } lt \equiv \langle Rk, Rv \rangle rbt\text{-rel } O \ rbt\text{-map-rel}' \ lt$
 $\langle proof \rangle$

lemma (**in** *linorder*) *autoref-gen-rbt-empty*:

$(rbt.Empty, Map.empty) \in \langle Rk, Rv \rangle rbt\text{-map-rel } (<)$
 $\langle proof \rangle$

lemma (**in** *linorder*) *autoref-gen-rbt-insert*:

fixes *less-impl*
assumes *param-less*: $(less\text{-impl}, (<)) \in Rk \rightarrow Rk \rightarrow Id$
shows $(ord.rbt\text{-insert } less\text{-impl}, \lambda k \ v \ m. \ m(k \mapsto v)) \in$
 $Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<) \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<)$
 $\langle proof \rangle$

lemma (**in** *linorder*) *autoref-gen-rbt-lookup*:

fixes *less-impl*
assumes *param-less*: $(less\text{-impl}, (<)) \in Rk \rightarrow Rk \rightarrow Id$
shows $(ord.rbt\text{-lookup } less\text{-impl}, \lambda m \ k. \ m \ k) \in$
 $\langle Rk, Rv \rangle rbt\text{-map-rel } (<) \rightarrow Rk \rightarrow \langle Rv \rangle option\text{-rel}$
 $\langle proof \rangle$

lemma (**in** *linorder*) *autoref-gen-rbt-delete*:

fixes *less-impl*
assumes *param-less*: $(less\text{-impl}, (<)) \in Rk \rightarrow Rk \rightarrow Id$
shows $(ord.rbt\text{-delete } less\text{-impl}, \lambda k \ m. \ m \setminus \{k\}) \in$
 $Rk \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<) \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<)$
 $\langle proof \rangle$

lemma (**in** *linorder*) *autoref-gen-rbt-union*:

fixes *less-impl*
assumes *param-less*: $(less\text{-impl}, (<)) \in Rk \rightarrow Rk \rightarrow Id$
shows $(ord.rbt\text{-union } less\text{-impl}, (++) \in$
 $\langle Rk, Rv \rangle rbt\text{-map-rel } (<) \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<) \rightarrow \langle Rk, Rv \rangle rbt\text{-map-rel } (<)$
 $\langle proof \rangle$

A linear ordering on red-black trees

abbreviation *rbt-to-list* $t \equiv it\text{-to-list } rm\text{-iterateoi } t$

lemma (**in** *linorder*) *rbt-to-list-correct*:

assumes *SORTED*: *rbt-sorted* t
shows $rbt\text{-to-list } t = sorted\text{-list-of-map } (rbt\text{-lookup } t) \ (\text{is } ?tl = -)$
 $\langle proof \rangle$

definition

$$cmp\text{-}rbt\ cmpk\ cmpv \equiv cmp\text{-}img\ rbt\text{-}to\text{-}list\ (cmp\text{-}lex\ (cmp\text{-}prod\ cmpk\ cmpv))$$

lemma (in *linorder*) *param-rbt-sorted-list-of-map*[*param*]:

shows (*rbt-to-list*, *sorted-list-of-map*) \in
 $\langle Rk, Rv \rangle rbt\text{-}map\text{-}rel\ (<) \rightarrow \langle \langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

lemma *param-rbt-sorted-list-of-map'*[*param*]:

assumes *ELO*: *eq-linorder* *cmp'*
shows (*rbt-to-list*, *linorder.sorted-list-of-map* (*comp2le* *cmp'*)) \in
 $\langle Rk, Rv \rangle rbt\text{-}map\text{-}rel\ (comp2lt\ cmp') \rightarrow \langle \langle Rk, Rv \rangle prod\text{-}rel \rangle list\text{-}rel$
 $\langle proof \rangle$

lemma *rbt-linorder-impl*:

assumes *ELO*: *eq-linorder* *cmp'*
assumes [*param*]: (*cmp*, *cmp'*) $\in Rk \rightarrow Rk \rightarrow Id$
shows
 $(cmp\text{-}rbt\ cmp, cmp\text{-}map\ cmp') \in$
 $(Rv \rightarrow Rv \rightarrow Id)$
 $\rightarrow \langle Rk, Rv \rangle rbt\text{-}map\text{-}rel\ (comp2lt\ cmp')$
 $\rightarrow \langle Rk, Rv \rangle rbt\text{-}map\text{-}rel\ (comp2lt\ cmp') \rightarrow Id$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *rbt-rel-sv-aux*:

assumes *SK*: *single-valued* *Rk*
assumes *SV*: *single-valued* *Rv*
assumes *I1*: (*a*, *b*) $\in (\langle Rk, Rv \rangle rbt\text{-}rel)$
assumes *I2*: (*a*, *c*) $\in (\langle Rk, Rv \rangle rbt\text{-}rel)$
shows *b* = *c*
 $\langle proof \rangle$

lemma *rbt-rel-sv*[*relator-props*]:

assumes *SK*: *single-valued* *Rk*
assumes *SV*: *single-valued* *Rv*
shows *single-valued* ($\langle Rk, Rv \rangle rbt\text{-}rel$)
 $\langle proof \rangle$

lemma *rbt-map-rel-sv*[*relator-props*]:

$\llbracket single\text{-}valued\ Rk; single\text{-}valued\ Rv \rrbracket$
 $\implies single\text{-}valued\ (\langle Rk, Rv \rangle rbt\text{-}map\text{-}rel\ lt)$
 $\langle proof \rangle$

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*) ∈
 $\langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\langle \text{proof} \rangle$

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*) ∈
 $Rk \rightarrow Rv \rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\langle \text{proof} \rangle$

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 ($\lambda k t. \text{ord.rbt-lookup (comp2lt cmp) } t k, \text{op-map-lookup}$) ∈
 $Rk \rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')} \rightarrow \langle Rv \rangle \text{option-rel}$
 $\langle \text{proof} \rangle$

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*) ∈
 $Rk \rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\langle \text{proof} \rangle$

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),(++)*) ∈
 $\langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')} \rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\rightarrow \langle Rk, Rv \rangle \text{rbt-map-rel (comp2lt cmp')}$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

lemma (in *linorder*) *dflt-cmp-id*:
 (*dflt-cmp* (\leq) ($<$), *dflt-cmp* (\leq) ($<$)) $\in Id \rightarrow Id \rightarrow Id$
<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* \rightarrow *Rk* \rightarrow *Id*)
assumes *GEN-OP* *cmpv cmpv'* (*Rv* \rightarrow *Rv* \rightarrow *Id*)
shows
 (*cmp-rbt cmpk cmpv*, *cmp-map cmpk' cmpv'*) \in
 $\langle Rk, Rv \rangle$ *rbt-map-rel* (*comp2lt cmpk'*)
 $\rightarrow \langle Rk, Rv \rangle$ *rbt-map-rel* (*comp2lt cmpk'*) $\rightarrow Id$
<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 ($\langle Rk, Rv \rangle$ *rbt-map-rel* ($<$))
<proof>

lemma *rbt-map-rel-finite*[*relator-props*]:
assumes *ELO*: *GEN-ALGO-tag* (*eq-linorder cmpk*)
shows *finite-map-rel* ($\langle Rk, Rv \rangle$ *rbt-map-rel* (*comp2lt cmpk*))
<proof>

abbreviation

dflt-rm-rel \equiv *rbt-map-rel* (*comp2lt* (*dflt-cmp* (\leq) ($<$)))

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* ($\langle < \rangle :: \text{linorder} \Rightarrow -$) = *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]*:
 $\llbracket \text{single-valued } R; \text{ Range } R = \text{UNIV} \rrbracket \implies \text{single-valued } (\langle R \rangle \text{fun-set-rel})$
<proof>

lemma *fun-set-rel-RUNIV*[*relator-props*]:

assumes *SV*: *single-valued R*

shows *Range* ($\langle R \rangle$ *fun-set-rel*) = *UNIV*

<proof>

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

lemma *fs-mem-refine*[*autoref-rules*]: $(\lambda x f. f x, (\in)) \in R \rightarrow \langle R \rangle$ *fun-set-rel* \rightarrow *bool-rel*

<proof>

lemma *fun-set-Collect-refine*[*autoref-rules*]:

$(\lambda x. x, \text{Collect}) \in (R \rightarrow \text{bool-rel}) \rightarrow \langle R \rangle$ *fun-set-rel*

<proof>

lemma *fun-set-empty-refine*[*autoref-rules*]:

$(\lambda-. \text{False}, \{\}) \in \langle R \rangle$ *fun-set-rel*

<proof>

lemma *fun-set-UNIV-refine*[*autoref-rules*]:

$(\lambda-. \text{True}, \text{UNIV}) \in \langle R \rangle$ *fun-set-rel*

<proof>

lemma *fun-set-union-refine*[*autoref-rules*]:

$(\lambda a b x. a x \vee b x, (\cup)) \in \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *fun-set-rel*

<proof>

lemma *fun-set-inter-refine*[*autoref-rules*]:

$(\lambda a b x. a x \wedge b x, (\cap)) \in \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *fun-set-rel*

<proof>

lemma *fun-set-diff-refine*[*autoref-rules*]:

$(\lambda a b x. a x \wedge \neg b x, (-)) \in \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *fun-set-rel* $\rightarrow \langle R \rangle$ *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 \Rightarrow nat \rightarrow 'v **where**
iam- α a i \equiv if i < array-length a then array-get a i else None

lemma [code]: *iam- α* a i \equiv array-get-oo None a i
 <proof>

abbreviation *iam-invar* :: 'v iam \Rightarrow bool **where** *iam-invar* \equiv λ -. True

definition *iam-empty* :: unit \Rightarrow 'v iam
where *iam-empty* \equiv λ ::unit. array-of-list []

definition *iam-lookup* :: nat \Rightarrow 'v iam \rightarrow 'v
where [code-unfold]: *iam-lookup* k a \equiv *iam- α* a k

definition *iam-increment* (l::nat) idx \equiv
 max (idx + 1 - l) (2 * l + 3)

lemma *incr-correct*: \neg idx < l \implies idx < l + *iam-increment* l idx
 <proof>

definition *iam-update* :: nat \Rightarrow 'v \Rightarrow 'v iam \Rightarrow 'v iam
where *iam-update* k v a \equiv 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 \equiv 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 \Rightarrow 'v iam \Rightarrow 'v iam
where *iam-delete* k a \equiv
 if k < array-length a then array-set a k None else a

lemma [code]: *iam-delete* k a \equiv array-set-oo (λ -. a) a k None
 <proof>

primrec *iam-iteratei-aux*

$:: \text{nat} \Rightarrow ('v \text{ iam}) \Rightarrow ('σ \Rightarrow \text{bool}) \Rightarrow (\text{nat} \times 'v \Rightarrow 'σ \Rightarrow 'σ) \Rightarrow 'σ \Rightarrow 'σ$
where
 $\text{iam-iteratei-aux } 0 \ a \ c \ f \ \sigma = \sigma$
 $| \text{iam-iteratei-aux } (\text{Suc } i) \ a \ c \ f \ \sigma = ($
 $\text{if } c \ \sigma \text{ then}$
 $\text{iam-iteratei-aux } i \ a \ c \ f \ ($
 $\text{case array-get } a \ i \ \text{of None} \Rightarrow \sigma \ | \ \text{Some } x \Rightarrow f \ (i, x) \ \sigma$
 $)$
 $\text{else } \sigma)$

definition $\text{iam-iteratei} :: 'v \text{ iam} \Rightarrow (\text{nat} \times 'v, 'σ) \text{ set-iterator}$ **where**
 $\text{iam-iteratei } a = \text{iam-iteratei-aux } (\text{array-length } a) \ a$

Parametricity

definition $\text{iam-rel-def-internal}$:

$\text{iam-rel } R \equiv \langle \langle R \rangle \text{ option-rel} \rangle \text{ array-rel}$

lemma iam-rel-def : $\langle R \rangle \text{ iam-rel} = \langle \langle R \rangle \text{ option-rel} \rangle \text{ array-rel}$
 $\langle \text{proof} \rangle$

lemma iam-rel-sv [relator-props]:

$\text{single-valued } Rv \Longrightarrow \text{single-valued } (\langle Rv \rangle \text{ iam-rel})$

$\langle \text{proof} \rangle$

lemma $\text{param-iam-}\alpha$ [param]:

$(\text{iam-}\alpha, \text{iam-}\alpha) \in \langle R \rangle \text{ iam-rel} \rightarrow \text{nat-rel} \rightarrow \langle R \rangle \text{ option-rel}$

$\langle \text{proof} \rangle$

lemma param-iam-invar [param]:

$(\text{iam-invar}, \text{iam-invar}) \in \langle R \rangle \text{ iam-rel} \rightarrow \text{bool-rel}$

$\langle \text{proof} \rangle$

lemma param-iam-empty [param]:

$(\text{iam-empty}, \text{iam-empty}) \in \text{unit-rel} \rightarrow \langle R \rangle \text{ iam-rel}$

$\langle \text{proof} \rangle$

lemma param-iam-lookup [param]:

$(\text{iam-lookup}, \text{iam-lookup}) \in \text{nat-rel} \rightarrow \langle R \rangle \text{ iam-rel} \rightarrow \langle R \rangle \text{ option-rel}$

$\langle \text{proof} \rangle$

lemma $\text{param-iam-increment}$ [param]:

$(\text{iam-increment}, \text{iam-increment}) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

$\langle \text{proof} \rangle$

lemma param-iam-update [param]:

$(\text{iam-update}, \text{iam-update}) \in \text{nat-rel} \rightarrow R \rightarrow \langle R \rangle \text{ iam-rel} \rightarrow \langle R \rangle \text{ iam-rel}$

$\langle \text{proof} \rangle$

lemma *param-iam-delete*[*param*]:
 $(iam_delete, iam_delete) \in nat_rel \rightarrow \langle R \rangle iam_rel \rightarrow \langle R \rangle iam_rel$
 $\langle proof \rangle$

lemma *param-iam-iteratei-aux*[*param*]:
assumes *I*: $i \leq array_length\ a$
assumes *IR*: $(i, i') \in nat_rel$
assumes *AR*: $(a, a') \in \langle Ra \rangle iam_rel$
assumes *CR*: $(c, c') \in Rb \rightarrow bool_rel$
assumes *FR*: $(f, f') \in \langle nat_rel, Ra \rangle prod_rel \rightarrow Rb \rightarrow Rb$
assumes σR : $(\sigma, \sigma') \in Rb$
shows $(iam_iteratei_aux\ i\ a\ c\ f\ \sigma, iam_iteratei_aux\ i'\ a'\ c'\ f'\ \sigma') \in Rb$
 $\langle proof \rangle$

lemma *param-iam-iteratei*[*param*]:
 $(iam_iteratei, iam_iteratei) \in \langle Ra \rangle iam_rel \rightarrow (Rb \rightarrow bool_rel) \rightarrow$
 $(\langle nat_rel, Ra \rangle prod_rel \rightarrow Rb \rightarrow Rb) \rightarrow Rb \rightarrow Rb$
 $\langle proof \rangle$

Correctness

definition $iam_rel' \equiv br\ iam_\alpha\ iam_invar$

lemma *iam-empty-correct*:
 $(iam_empty\ (), Map.empty) \in iam_rel'$
 $\langle proof \rangle$

lemma *iam-update-correct*:
 $(iam_update, op_map_update) \in nat_rel \rightarrow Id \rightarrow iam_rel' \rightarrow iam_rel'$
 $\langle proof \rangle$

lemma *iam-lookup-correct*:
 $(iam_lookup, op_map_lookup) \in Id \rightarrow iam_rel' \rightarrow \langle Id \rangle option_rel$
 $\langle proof \rangle$

lemma *array-get-set-iff*: $i < array_length\ a \implies$
 $array_get\ (array_set\ a\ i\ x)\ j = (if\ i=j\ then\ x\ else\ array_get\ a\ j)$
 $\langle proof \rangle$

lemma *iam-delete-correct*:
 $(iam_delete, op_map_delete) \in Id \rightarrow iam_rel' \rightarrow iam_rel'$
 $\langle proof \rangle$

definition *iam-map-rel-def-internal*:
 $iam_map_rel\ Rk\ Rv \equiv$
 $if\ Rk = nat_rel\ then\ \langle Rv \rangle iam_rel\ O\ iam_rel'\ else\ \{\}$

lemma *iam-map-rel-def*:

$\langle \text{nat-rel}, Rv \rangle \text{iam-map-rel} \equiv \langle Rv \rangle \text{iam-rel} \circ \text{iam-rel}'$
 $\langle \text{proof} \rangle$

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

lemma iam-map-rel-sv[relator-props]:
single-valued $Rv \implies \text{single-valued } (\langle \text{nat-rel}, Rv \rangle \text{iam-map-rel})$
 $\langle \text{proof} \rangle$

lemma iam-empty-impl:
 $(\text{iam-empty } (), \text{op-map-empty}) \in \langle \text{nat-rel}, R \rangle \text{iam-map-rel}$
 $\langle \text{proof} \rangle$

lemma iam-lookup-impl:
 $(\text{iam-lookup}, \text{op-map-lookup})$
 $\in \text{nat-rel} \rightarrow \langle \text{nat-rel}, R \rangle \text{iam-map-rel} \rightarrow \langle R \rangle \text{option-rel}$
 $\langle \text{proof} \rangle$

lemma iam-update-impl:
 $(\text{iam-update}, \text{op-map-update}) \in$
 $\text{nat-rel} \rightarrow R \rightarrow \langle \text{nat-rel}, R \rangle \text{iam-map-rel} \rightarrow \langle \text{nat-rel}, R \rangle \text{iam-map-rel}$
 $\langle \text{proof} \rangle$

lemma iam-delete-impl:
 $(\text{iam-delete}, \text{op-map-delete}) \in$
 $\text{nat-rel} \rightarrow \langle \text{nat-rel}, R \rangle \text{iam-map-rel} \rightarrow \langle \text{nat-rel}, R \rangle \text{iam-map-rel}$
 $\langle \text{proof} \rangle$

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 $a \equiv \text{it-to-list iam-iteratei } a$

lemma distinct-iam-to-list-aux:
shows $\llbracket \text{distinct } xs; \forall (i, -) \in \text{set } xs. i \geq n \rrbracket \implies$
 $\text{distinct } (\text{iam-iteratei-aux } n \ a$
 $\quad (\lambda -. \text{True}) (\lambda x \ y. y \ @ \ [x]) \ xs)$
(is $\llbracket -; \rrbracket \implies \text{distinct } (?iam-to-list-aux \ n \ xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-iam-to-list*:

distinct (iam-to-list a)

<proof>

lemma *iam-to-list-set-correct-aux*:

assumes $(a, m) \in \text{iam-rel}'$

shows $\llbracket n \leq \text{array-length } a; \text{map-to-set } m - \{(k,v). k < n\} = \text{set } xs \rrbracket$

$\implies \text{map-to-set } m =$

$\text{set } (\text{iam-iteratei-aux } n \ a \ (\lambda-. \text{True}) \ (\lambda x \ y. y @ [x]) \ xs)$

<proof>

lemma *iam-to-list-set-correct*:

assumes $(a, m) \in \text{iam-rel}'$

shows $\text{map-to-set } m = \text{set } (\text{iam-to-list } a)$

<proof>

lemma *iam-iteratei-aux-append*:

$n \leq \text{length } xs \implies \text{iam-iteratei-aux } n \ (\text{Array } (xs @ ys)) =$

$\text{iam-iteratei-aux } n \ (\text{Array } xs)$

<proof>

lemma *iam-iteratei-append*:

$\text{iam-iteratei } (\text{Array } (xs @ [\text{None}])) \ c \ f \ \sigma =$

$\text{iam-iteratei } (\text{Array } xs) \ c \ f \ \sigma$

$\text{iam-iteratei } (\text{Array } (xs @ [\text{Some } x])) \ c \ f \ \sigma =$

$\text{iam-iteratei } (\text{Array } xs) \ c \ f$

$(\text{if } c \ \sigma \ \text{then } (f \ (\text{length } xs, x) \ \sigma) \ \text{else } \sigma)$

<proof>

lemma *iam-iteratei-aux-Cons*:

$n < \text{array-length } a \implies$

$\text{iam-iteratei-aux } n \ a \ (\lambda-. \text{True}) \ (\lambda x \ l. l @ [x]) \ (x \# xs) =$

$x \# \text{iam-iteratei-aux } n \ a \ (\lambda-. \text{True}) \ (\lambda x \ l. l @ [x]) \ xs$

<proof>

lemma *iam-to-list-append*:

$\text{iam-to-list } (\text{Array } (xs @ [\text{None}])) = \text{iam-to-list } (\text{Array } xs)$

$\text{iam-to-list } (\text{Array } (xs @ [\text{Some } x])) =$

$(\text{length } xs, x) \# \text{iam-to-list } (\text{Array } xs)$

<proof>

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

shows $\text{is-map-to-list } \text{nat-rel } Rv \ \text{iam-map-rel } \text{iam-to-list}$

<proof>

lemmas [*autoref-ga-rules*] =

$\text{autoref-iam-is-iterator}[\text{unfolded } \text{is-map-to-list-def}]$

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- $\alpha$  = 's  $\Rightarrow$  'x set  
type-synonym ('x, 's) set-invar = 's  $\Rightarrow$  bool  
locale set =  
  — Abstraction to set  
  fixes  $\alpha$  :: 's  $\Rightarrow$  'x set  
  — Invariant  
  fixes invar :: 's  $\Rightarrow$  bool
```

```
locale set-no-invar = set +  
  assumes invar[simp, intro!]:  $\bigwedge s. \text{invar } s$ 
```

Basic Set Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

locale $set\text{-isSng} = set +$
constrains $\alpha :: 's \Rightarrow 'x \text{ set}$
fixes $isSng :: 's \Rightarrow bool$
assumes $isSng\text{-correct}: \text{invar } s \Longrightarrow isSng s \longleftrightarrow (\exists e. \alpha s = \{e\})$
begin
lemma $isSng\text{-correct}\text{-exists1} :$
 $\text{invar } s \Longrightarrow (isSng s \longleftrightarrow (\exists !e. (e \in \alpha s)))$
 $\langle proof \rangle$
lemma $isSng\text{-correct}\text{-card} :$
 $\text{invar } s \Longrightarrow (isSng s \longleftrightarrow (card (\alpha s) = 1))$
 $\langle proof \rangle$
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 $('x, 's) \text{ set}\text{-list}\text{-it}$

```

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

  abbreviation iterate s  $\equiv$  iteratei s (λ-. True)
end

locale poly-set-iteratei =
  finite-set + poly-set-iteratei-defs list-it
  for list-it :: 's  $\Rightarrow$  ('x, 'x list) set-iterator +
  constrains α :: 's  $\Rightarrow$  'x set
  assumes list-it-correct: invar s  $\Longrightarrow$  set-iterator (list-it s) (α s)
begin
  lemma iteratei-correct: invar S  $\Longrightarrow$  set-iterator (iteratei S) (α S)
    <proof>

  lemma pi-iteratei[icf-proper-iteratorI]:
    proper-it (iteratei S) (iteratei S)
    <proof>

  lemma iteratei-rule-P:
    [
      invar S;
      I (α S) σ 0;
      !!x it σ. [ c σ; x ∈ it; it ⊆ α S; I it σ ]  $\Longrightarrow$  I (it - {x}) (f x σ);
      !!σ. I {} σ  $\Longrightarrow$  P σ;
      !!σ it. [ it ⊆ α S; it ≠ {}; ¬ c σ; I it σ ]  $\Longrightarrow$  P σ
    ]  $\Longrightarrow$  P (iteratei S c f σ 0)
    <proof>

  lemma iteratei-rule-insert-P:
    [
      invar S;
      I {} σ 0;
      !!x it σ. [ c σ; x ∈ α S - it; it ⊆ α S; I it σ ]  $\Longrightarrow$  I (insert x it) (f x σ);
      !!σ. I (α S) σ  $\Longrightarrow$  P σ;
      !!σ it. [ it ⊆ α S; it ≠ α S; ¬ c σ; I it σ ]  $\Longrightarrow$  P σ
    ]  $\Longrightarrow$  P (iteratei S c f σ 0)
    <proof>

```

Versions without break condition.

```

lemma iterate-rule-P:
  [
    invar S;
    I (α S) σ 0;

```

$$\begin{aligned} & \llbracket x \in it; it \subseteq \alpha S; I it \sigma \rrbracket \implies I (it - \{x\}) (f x \sigma); \\ & \llbracket \sigma. I \{\} \sigma \implies P \sigma \\ & \rrbracket \implies P (\text{iteratei } S (\lambda-. \text{True}) f \sigma 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *iterate-rule-insert-P*:

$$\begin{aligned} & \llbracket \\ & \quad \text{invar } S; \\ & \quad I \{\} \sigma 0; \\ & \quad \llbracket x \in \alpha S - it; it \subseteq \alpha S; I it \sigma \rrbracket \implies I (\text{insert } x it) (f x \sigma); \\ & \quad \llbracket \sigma. I (\alpha S) \sigma \implies P \sigma \\ & \rrbracket \implies P (\text{iteratei } S (\lambda-. \text{True}) f \sigma 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

end

More Set Operations

Copy locale *set-copy* = *s1: set $\alpha 1$ invar1 + s2: set $\alpha 2$ invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a \text{ set}$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a \text{ set}$ **and** *invar2*
 +
fixes *copy* :: $'s1 \Rightarrow 's2$
assumes *copy-correct*:
 $\text{invar1 } s1 \implies \alpha 2 (\text{copy } s1) = \alpha 1 s1$
 $\text{invar1 } s1 \implies \text{invar2 } (\text{copy } s1)$

Union locale *set-union* = *s1: set $\alpha 1$ invar1 + s2: set $\alpha 2$ invar2 + s3: set $\alpha 3$ invar3*
for $\alpha 1 :: 's1 \Rightarrow 'a \text{ set}$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a \text{ set}$ **and** *invar2*
and $\alpha 3 :: 's3 \Rightarrow 'a \text{ set}$ **and** *invar3*
 +
fixes *union* :: $'s1 \Rightarrow 's2 \Rightarrow 's3$
assumes *union-correct*:
 $\text{invar1 } s1 \implies \text{invar2 } s2 \implies \alpha 3 (\text{union } s1 s2) = \alpha 1 s1 \cup \alpha 2 s2$
 $\text{invar1 } s1 \implies \text{invar2 } s2 \implies \text{invar3 } (\text{union } s1 s2)$

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

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

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

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

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

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

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

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

```

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

  lemmas  $\text{image-filter-correct} =$ 
     $\text{image-filter-correct-aux2 } \text{image-filter-correct-aux}$ 

end

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

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

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

```

Filter *locale set-filter* = *s1: set $\alpha1$ invar1 + s2: set $\alpha2$ invar2*
for $\alpha1 :: 's1 \Rightarrow 'a$ *set and invar1*
and $\alpha2 :: 's2 \Rightarrow 'a$ *set and invar2*
+
fixes *filter* :: $('a \Rightarrow \text{bool}) \Rightarrow 's1 \Rightarrow 's2$
assumes *filter-correct*:
invar1 s $\implies \alpha2$ (filter P s) = {e. e $\in \alpha1$ s $\wedge P$ e}
invar1 s \implies invar2 (filter P s)

Union of Set of Sets *locale set-Union-image* =
s1: set $\alpha1$ invar1 + s2: set $\alpha2$ invar2 + s3: set $\alpha3$ invar3
for $\alpha1 :: 's1 \Rightarrow 'a$ *set and invar1*
and $\alpha2 :: 's2 \Rightarrow 'b$ *set and invar2*
and $\alpha3 :: 's3 \Rightarrow 'b$ *set and invar3*
+
fixes *Union-image* :: $('a \Rightarrow 's2) \Rightarrow 's1 \Rightarrow 's3$
assumes *Union-image-correct*:
 $\llbracket \text{invar1 } s; \forall x. x \in \alpha1 \text{ } s \implies \text{invar2 } (f \text{ } x) \rrbracket \implies$
 $\alpha3 \text{ (Union-image } f \text{ } s) = \bigcup (\alpha2 \text{ } f \text{ } \alpha1 \text{ } s)$
 $\llbracket \text{invar1 } s; \forall x. x \in \alpha1 \text{ } s \implies \text{invar2 } (f \text{ } x) \rrbracket \implies \text{invar3 (Union-image } f \text{ } s)$

Disjointness Check *locale set-disjoint* = *s1: set $\alpha1$ invar1 + s2: set $\alpha2$ invar2*
for $\alpha1 :: 's1 \Rightarrow 'a$ *set and invar1*
and $\alpha2 :: 's2 \Rightarrow 'a$ *set and invar2*
+
fixes *disjoint* :: $'s1 \Rightarrow 's2 \Rightarrow \text{bool}$
assumes *disjoint-correct*:
invar1 s1 \implies invar2 s2 \implies disjoint s1 s2 $\longleftrightarrow \alpha1$ s1 $\cap \alpha2$ s2 = {}

Disjointness Check With Witness *locale set-disjoint-witness* = *s1: set $\alpha1$ invar1 + s2: set $\alpha2$ invar2*
for $\alpha1 :: 's1 \Rightarrow 'a$ *set and invar1*
and $\alpha2 :: 's2 \Rightarrow 'a$ *set and invar2*
+
fixes *disjoint-witness* :: $'s1 \Rightarrow 's2 \Rightarrow 'a$ *option*
assumes *disjoint-witness-correct*:
 $\llbracket \text{invar1 } s1; \text{invar2 } s2 \rrbracket$
 $\implies \text{disjoint-witness } s1 \text{ } s2 = \text{None} \implies \alpha1 \text{ } s1 \cap \alpha2 \text{ } s2 = \{\}$
 $\llbracket \text{invar1 } s1; \text{invar2 } s2; \text{disjoint-witness } s1 \text{ } s2 = \text{Some } a \rrbracket$
 $\implies a \in \alpha1 \text{ } s1 \cap \alpha2 \text{ } s2$
begin
lemma *disjoint-witness-None*: $\llbracket \text{invar1 } s1; \text{invar2 } s2 \rrbracket$
 $\implies \text{disjoint-witness } s1 \text{ } s2 = \text{None} \longleftrightarrow \alpha1 \text{ } s1 \cap \alpha2 \text{ } s2 = \{\}$
 $\langle \text{proof} \rangle$
lemma *disjoint-witnessI*: \llbracket
invar1 s1;
invar2 s2;

$\alpha 1 s1 \cap \alpha 2 s2 \neq \{\}$;
!!a. $\llbracket \text{disjoint-witness } s1 \ s2 = \text{Some } a \rrbracket \implies P$
 $\rrbracket \implies P$
 <proof>

end

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

lemma *sel-someD*:
 $\llbracket \text{invar } s; \text{sel } s \ f = \text{Some } r; \llbracket x \in \alpha \ s; f \ x = \text{Some } r \rrbracket \implies P \rrbracket \implies P$
 <proof>

lemma *sel-noneD*:
 $\llbracket \text{invar } s; \text{sel } s \ f = \text{None}; x \in \alpha \ s \rrbracket \implies f \ x = \text{None}$
 <proof>

end

— Selection of element (without mapping)

locale *set-sel'* = *set* +
constrains $\alpha :: 's \Rightarrow 'x \ \text{set}$
fixes *sel'* :: $'s \Rightarrow ('x \Rightarrow \text{bool}) \Rightarrow 'x \ \text{option}$
assumes *sel'E*:
 $\llbracket \text{invar } s; x \in \alpha \ s; P \ x; \llbracket \text{sel}' \ s \ P = \text{Some } x; x \in \alpha \ s; P \ x \rrbracket \implies Q$
 $\rrbracket \implies Q$
assumes *sel'I*: $\llbracket \text{invar } s; \forall x \in \alpha \ s. \neg P \ x \rrbracket \implies \text{sel}' \ s \ P = \text{None}$
begin

lemma *sel'-someD*:
 $\llbracket \text{invar } s; \text{sel}' \ s \ P = \text{Some } x \rrbracket \implies x \in \alpha \ s \wedge P \ x$
 <proof>

lemma *sel'-noneD*:
 $\llbracket \text{invar } s; \text{sel}' \ s \ P = \text{None}; x \in \alpha \ s \rrbracket \implies \neg P \ x$
 <proof>

end

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

assumes *to-list-correct*:
invar $s \implies \text{set } (\text{to-list } s) = \alpha \ s$
invar $s \implies \text{distinct } (\text{to-list } s)$

Conversion of List to Set **locale** *list-to-set* = *set* +
constrains $\alpha :: 's \Rightarrow 'x \ \text{set}$
fixes *to-set* :: $'x \ \text{list} \Rightarrow 's$
assumes *to-set-correct*:
 $\alpha \ (\text{to-set } l) = \text{set } l$
invar $(\text{to-set } l)$

Ordered Sets

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

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

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

abbreviation *iterateo* $s \equiv \text{iterateoi } s \ (\lambda-. \ \text{True})$
end

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

begin
lemma *iterateoi-correct*:
invar $S \implies \text{set-iterator-linord } (\text{iterateoi } S) \ (\alpha \ S)$
 $\langle \text{proof} \rangle$

lemma *pi-iterateoi*[*icf-proper-iteratorI*]:
proper-it $(\text{iterateoi } S) \ (\text{iterateoi } S)$
 $\langle \text{proof} \rangle$

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

assumes $IP: !!k \text{ it } \sigma. \llbracket$
 $c \ \sigma;$
 $k \in \text{it};$
 $\forall j \in \text{it}. k \leq j;$
 $\forall j \in \alpha \ s - \text{it}. j \leq k;$
 $\text{it} \subseteq \alpha \ s;$
 $I \ \text{it} \ \sigma$
 $\rrbracket \implies I (\text{it} - \{k\}) (f \ k \ \sigma)$
assumes $IF: !!\sigma. I \ \{\} \ \sigma \implies P \ \sigma$
assumes $II: !!\sigma \ \text{it}. \llbracket$
 $\text{it} \subseteq \alpha \ s;$
 $\text{it} \neq \{\};$
 $\neg c \ \sigma;$
 $I \ \text{it} \ \sigma;$
 $\forall k \in \text{it}. \forall j \in \alpha \ s - \text{it}. j \leq k$
 $\rrbracket \implies P \ \sigma$
shows $P (\text{iterateoi} \ s \ c \ f \ \sigma \ 0)$
 $\langle \text{proof} \rangle$

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

assumes $MINV: \text{invar} \ s$
assumes $I0: I ((\alpha \ s)) \ \sigma \ 0$
assumes $IP: !!k \ \text{it} \ \sigma. \llbracket k \in \text{it}; \forall j \in \text{it}. k \leq j;$
 $\forall j \in (\alpha \ s) - \text{it}. j \leq k; \text{it} \subseteq (\alpha \ s); I \ \text{it} \ \sigma \rrbracket$
 $\implies I (\text{it} - \{k\}) (f \ k \ \sigma)$
assumes $IF: !!\sigma. I \ \{\} \ \sigma \implies P \ \sigma$
shows $P (\text{iterateo} \ s \ f \ \sigma \ 0)$
 $\langle \text{proof} \rangle$

end

locale *poly-set-rev-iterateoi-defs* =

fixes $\text{list-rev-it} :: 's \Rightarrow ('x::\text{linorder}, 'x \ \text{list}) \ \text{set-iterator}$

begin

definition $\text{rev-iterateoi} :: 's \Rightarrow ('x, 's) \ \text{set-iterator}$

where $\text{rev-iterateoi} \ S \equiv \text{it-to-it} (\text{list-rev-it} \ S)$

abbreviation $\text{rev-iterateo} \ m \equiv \text{rev-iterateoi} \ m \ (\lambda-. \ \text{True})$

abbreviation $\text{reverse-iterateoi} \equiv \text{rev-iterateoi}$

abbreviation $\text{reverse-iterateo} \equiv \text{rev-iterateo}$

end

locale *poly-set-rev-iterateoi* =

$\text{finite-set} \ \alpha \ \text{invar} + \text{poly-set-rev-iterateoi-defs} \ \text{list-rev-it}$

for $\alpha :: 's \Rightarrow 'x::\text{linorder} \ \text{set}$

and invar

and $\text{list-rev-it} :: 's \Rightarrow ('x, 'x \ \text{list}) \ \text{set-iterator} +$

assumes $\text{list-rev-it-correct}:$

$\text{invar} \ s \implies \text{set-iterator-rev-linord} (\text{list-rev-it} \ s) (\alpha \ s)$

begin

lemma *rev-iterateoi-correct*:

invar S \implies *set-iterator-rev-linord* (*rev-iterateoi S*) (αS)
 \langle *proof* \rangle

lemma *pi-rev-iterateoi*[*icf-proper-iteratorI*]:

proper-it (*rev-iterateoi S*) (*rev-iterateoi S*)
 \langle *proof* \rangle

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

assumes *MINV*: *invar s*

assumes *I0*: $I ((\alpha s)) \sigma 0$

assumes *IP*: $!!k \text{ it } \sigma. \llbracket$

$c \sigma;$

$k \in \text{it};$

$\forall j \in \text{it}. k \geq j;$

$\forall j \in (\alpha s) - \text{it}. j \geq k;$

$\text{it} \subseteq (\alpha s);$

$I \text{ it } \sigma$

$\rrbracket \implies I (\text{it} - \{k\}) (f k \sigma)$

assumes *IF*: $!!\sigma. I \{\} \sigma \implies P \sigma$

assumes *II*: $!!\sigma \text{ it}. \llbracket$

$\text{it} \subseteq (\alpha s);$

$\text{it} \neq \{\};$

$\neg c \sigma;$

$I \text{ it } \sigma;$

$\forall k \in \text{it}. \forall j \in (\alpha s) - \text{it}. j \geq k$

$\rrbracket \implies P \sigma$

shows $P (\text{rev-iterateoi } s \ c \ f \ \sigma 0)$

\langle *proof* \rangle

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

assumes *MINV*: *invar s*

assumes *I0*: $I ((\alpha s)) \sigma 0$

assumes *IP*: $!!k \text{ it } \sigma. \llbracket$

$k \in \text{it};$

$\forall j \in \text{it}. k \geq j;$

$\forall j \in (\alpha s) - \text{it}. j \geq k;$

$\text{it} \subseteq (\alpha s);$

$I \text{ it } \sigma$

$\rrbracket \implies I (\text{it} - \{k\}) (f k \sigma)$

assumes *IF*: $!!\sigma. I \{\} \sigma \implies P \sigma$

shows $P (\text{rev-iterateoi } s \ f \ \sigma 0)$

\langle *proof* \rangle

end

Minimal and Maximal Element **locale** *set-min* = *ordered-set* +
constrains $\alpha :: 's \Rightarrow 'u::\text{linorder set}$

fixes $min :: 's \Rightarrow ('u \Rightarrow bool) \Rightarrow 'u \text{ option}$
assumes $min\text{-correct}$:
 $\llbracket invar\ s; x \in \alpha\ s; P\ x \rrbracket \Longrightarrow min\ s\ P \in Some\ \{x \in \alpha\ s. P\ x\}$
 $\llbracket invar\ s; x \in \alpha\ s; P\ x \rrbracket \Longrightarrow (the\ (min\ s\ P)) \leq x$
 $\llbracket invar\ s; \{x \in \alpha\ s. P\ x\} = \{\} \rrbracket \Longrightarrow min\ s\ P = None$

begin

lemma $minE$:

assumes $A: invar\ s\ \ x \in \alpha\ s\ \ P\ x$

obtains x' **where**

$min\ s\ P = Some\ x' \ \ x' \in \alpha\ s \ \ P\ x' \ \ \forall x \in \alpha\ s. P\ x \longrightarrow x' \leq x$
 $\langle proof \rangle$

lemmas $minI = min\text{-correct}(3)$

lemma $min\text{-Some}$:

$\llbracket invar\ s; min\ s\ P = Some\ x \rrbracket \Longrightarrow x \in \alpha\ s$

$\llbracket invar\ s; min\ s\ P = Some\ x \rrbracket \Longrightarrow P\ x$

$\llbracket invar\ s; min\ s\ P = Some\ x; x' \in \alpha\ s; P\ x' \rrbracket \Longrightarrow x \leq x'$

$\langle proof \rangle$

lemma $min\text{-None}$:

$\llbracket invar\ s; min\ s\ P = None \rrbracket \Longrightarrow \{x \in \alpha\ s. P\ x\} = \{\}$

$\langle proof \rangle$

end

locale $set\text{-max} = ordered\text{-set} +$

constrains $\alpha :: 's \Rightarrow 'u::linorder\ set$

fixes $max :: 's \Rightarrow ('u \Rightarrow bool) \Rightarrow 'u \text{ option}$

assumes $max\text{-correct}$:

$\llbracket invar\ s; x \in \alpha\ s; P\ x \rrbracket \Longrightarrow max\ s\ P \in Some\ \{x \in \alpha\ s. P\ x\}$

$\llbracket invar\ s; x \in \alpha\ s; P\ x \rrbracket \Longrightarrow the\ (max\ s\ P) \geq x$

$\llbracket invar\ s; \{x \in \alpha\ s. P\ x\} = \{\} \rrbracket \Longrightarrow max\ s\ P = None$

begin

lemma $maxE$:

assumes $A: invar\ s\ \ x \in \alpha\ s\ \ P\ x$

obtains x' **where**

$max\ s\ P = Some\ x' \ \ x' \in \alpha\ s \ \ P\ x' \ \ \forall x \in \alpha\ s. P\ x \longrightarrow x' \geq x$
 $\langle proof \rangle$

lemmas $maxI = max\text{-correct}(3)$

lemma $max\text{-Some}$:

$\llbracket invar\ s; max\ s\ P = Some\ x \rrbracket \Longrightarrow x \in \alpha\ s$

$\llbracket invar\ s; max\ s\ P = Some\ x \rrbracket \Longrightarrow P\ x$

$\llbracket invar\ s; max\ s\ P = Some\ x; x' \in \alpha\ s; P\ x' \rrbracket \Longrightarrow x \geq x'$

$\langle proof \rangle$

lemma $max\text{-None}$:

$\llbracket \text{invar } s; \text{max } s \ P = \text{None} \rrbracket \implies \{x \in \alpha \ s. \ P \ x\} = \{\}$
 $\langle \text{proof} \rangle$

end

Conversion to List

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

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

Record Based Interface

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

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

```

locale StdSetDefs =
  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  $\alpha$  invar ball +
    set-bex  $\alpha$  invar bex +
    set-size  $\alpha$  invar size +
    set-size-abort  $\alpha$  invar size-abort +
    set-union  $\alpha$  invar  $\alpha$  invar  $\alpha$  invar union +
    set-union-dj  $\alpha$  invar  $\alpha$  invar  $\alpha$  invar union-dj +
    set-diff  $\alpha$  invar  $\alpha$  invar diff +
    set-filter  $\alpha$  invar  $\alpha$  invar filter +
    set-inter  $\alpha$  invar  $\alpha$  invar  $\alpha$  invar inter +
    set-subset  $\alpha$  invar  $\alpha$  invar subset +
    set-equal  $\alpha$  invar  $\alpha$  invar equal +
    set-disjoint  $\alpha$  invar  $\alpha$  invar disjoint +
    set-disjoint-witness  $\alpha$  invar  $\alpha$  invar disjoint-witness +
    set-sel'  $\alpha$  invar sel +
    set-to-list  $\alpha$  invar to-list +
    list-to-set  $\alpha$  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  $\alpha$  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 $\alpha :: 's \Rightarrow 'x \text{ list}$
 — Invariant
fixes *invar* :: $'s \Rightarrow \text{bool}$

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

Functions

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

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

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

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

locale *list-size* = *list* +
constrains $\alpha :: 's \Rightarrow 'x \text{ list}$
fixes *size* :: $'s \Rightarrow \text{nat}$


```

assumes size-correct:
  invar s  $\implies$  size s = length ( $\alpha$  s)

locale list-appendl = list +
  constrains  $\alpha :: 's \Rightarrow 'x \text{ list}$ 
  fixes appendl ::  $'x \Rightarrow 's \Rightarrow 's$ 
  assumes appendl-correct:
    invar s  $\implies \alpha$  (appendl x s) = x# $\alpha$  s
    invar s  $\implies$  invar (appendl x s)
begin
  abbreviation (input) push  $\equiv$  appendl
  lemmas push-correct = appendl-correct
end

locale list-removel = list +
  constrains  $\alpha :: 's \Rightarrow 'x \text{ list}$ 
  fixes removel ::  $'s \Rightarrow ('x \times 's)$ 
  assumes removel-correct:
     $\llbracket \textit{invar } s; \alpha \textit{ s} \neq [] \rrbracket \implies \textit{fst} (\textit{removel } s) = \textit{hd} (\alpha \textit{ s})$ 
     $\llbracket \textit{invar } s; \alpha \textit{ s} \neq [] \rrbracket \implies \alpha (\textit{snd} (\textit{removel } s)) = \textit{tl} (\alpha \textit{ s})$ 
     $\llbracket \textit{invar } s; \alpha \textit{ s} \neq [] \rrbracket \implies \textit{invar} (\textit{snd} (\textit{removel } s))$ 
begin
  lemma removelE:
    assumes I: invar s  $\alpha$  s  $\neq []$ 
    obtains s' where removel s = (hd ( $\alpha$  s), s')  $\textit{invar } s'$   $\alpha$  s' = tl ( $\alpha$  s)
    <proof>
end

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

abbreviation (input) pop  $\equiv$  removel
lemmas pop-correct = removel-correct

abbreviation (input) dequeue  $\equiv$  removel
lemmas dequeue-correct = removel-correct
end

locale list-leftmost = list +
  constrains  $\alpha :: 's \Rightarrow 'x \text{ list}$ 
  fixes leftmost ::  $'s \Rightarrow 'x$ 
  assumes leftmost-correct:
     $\llbracket \textit{invar } s; \alpha \textit{ s} \neq [] \rrbracket \implies \textit{leftmost } s = \textit{hd} (\alpha \textit{ s})$ 
begin
  abbreviation (input) top where top  $\equiv$  leftmost
  lemmas top-correct = leftmost-correct
end

locale list-appendr = list +
  constrains  $\alpha :: 's \Rightarrow 'x \text{ list}$ 
  fixes appendr ::  $'x \Rightarrow 's \Rightarrow 's$ 

```

```

assumes appendr-correct:
  invar s  $\implies \alpha$  (appendr x s) =  $\alpha$  s @ [x]
  invar s  $\implies$  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:
   $\llbracket \textit{invar } s; \alpha s \neq [] \rrbracket \implies \alpha$  (fst (remover s)) = butlast ( $\alpha$  s)
   $\llbracket \textit{invar } s; \alpha s \neq [] \rrbracket \implies$  snd (remover s) = last ( $\alpha$  s)
   $\llbracket \textit{invar } s; \alpha s \neq [] \rrbracket \implies$  invar (fst (remover s))

locale list-rightmost = list +
constrains  $\alpha :: 's \Rightarrow 'x$  list
fixes rightmost ::  $'s \Rightarrow 'x$ 
assumes rightmost-correct:
   $\llbracket \textit{invar } s; \alpha s \neq [] \rrbracket \implies$  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 ::  $'s \Rightarrow \textit{nat} \Rightarrow 'x$ 
assumes get-correct:
   $\llbracket \textit{invar } s; i < \textit{length} (\alpha s) \rrbracket \implies$  get s i =  $\alpha$  s ! i

locale list-set = list +
constrains  $\alpha :: 's \Rightarrow 'x$  list
fixes set ::  $'s \Rightarrow \textit{nat} \Rightarrow 'x \Rightarrow 's$ 
assumes set-correct:
   $\llbracket \textit{invar } s; i < \textit{length} (\alpha s) \rrbracket \implies \alpha$  (set s i x) = ( $\alpha$  s) [i := x]
   $\llbracket \textit{invar } s; i < \textit{length} (\alpha s) \rrbracket \implies$  invar (set s i x)

record ( $'a, 's$ ) list-ops =
  list-op- $\alpha$  ::  $'s \Rightarrow 'a$  list
  list-op-invar ::  $'s \Rightarrow \textit{bool}$ 
  list-op-empty :: unit  $\Rightarrow 's$ 
  list-op-isEmpty ::  $'s \Rightarrow \textit{bool}$ 
  list-op-size ::  $'s \Rightarrow \textit{nat}$ 
  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  $\alpha$  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  $\alpha :: 's \Rightarrow ('e \times 'a::monoid-add) list$ 
  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  $\alpha :: 's \Rightarrow ('e \times 'a::monoid-add) list$ 
  fixes empty :: unit  $\Rightarrow 's$ 
  assumes empty-correct:
    invar (empty ())
     $\alpha$  (empty ()) = Nil

```

```

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

```

```

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

```

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

Appending an Element from the Right locale $al\text{-}consr = al +$
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add) list$
fixes $consr :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
assumes $consr\text{-}correct:$
 $invar\ s \Longrightarrow invar\ (consr\ s\ e\ a)$
 $invar\ s \Longrightarrow (\alpha\ (consr\ s\ e\ a)) = (\alpha\ s) @ [(e,a)]$

Take the First Element locale $al\text{-}head = al +$
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add) list$
fixes $head :: 's \Rightarrow ('e \times 'a)$
assumes $head\text{-}correct:$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow head\ s = hd\ (\alpha\ s)$

Drop the First Element locale $al\text{-}tail = al +$
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add) list$
fixes $tail :: 's \Rightarrow 's$
assumes $tail\text{-}correct:$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow \alpha\ (tail\ s) = tl\ (\alpha\ s)$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow invar\ (tail\ s)$

Take the Last Element locale $al\text{-}headR = al +$
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add) list$
fixes $headR :: 's \Rightarrow ('e \times 'a)$
assumes $headR\text{-}correct:$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow headR\ s = last\ (\alpha\ s)$

Drop the Last Element locale $al\text{-}tailR = al +$
constrains $\alpha :: 's \Rightarrow ('e \times 'a::monoid\text{-}add) list$
fixes $tailR :: 's \Rightarrow 's$
assumes $tailR\text{-}correct:$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow \alpha\ (tailR\ s) = butlast\ (\alpha\ s)$
 $\llbracket invar\ s; \alpha\ s \neq Nil \rrbracket \Longrightarrow invar\ (tailR\ s)$

Fold a Function over the Elements from the Left locale $al\text{-}foldl = al$
 $+$
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 $foldl\text{-}correct:$
 $invar\ s \Longrightarrow foldl\ f\ \sigma\ s = List.foldl\ f\ \sigma\ (\alpha\ s)$

Fold a Function over the Elements from the Right locale *al-foldr* =
al +

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

locale *poly-al-fold* = *al* +

constrains $\alpha :: 's \Rightarrow ('e \times 'a::\text{monoid-add}) \text{ list}$

begin

definition *foldl* **where**

foldl-correct[*code-unfold*]: $\text{foldl } f \ \sigma \ s = \text{List.foldl } f \ \sigma \ (\alpha \ s)$

definition *foldr* **where**

foldr-correct[*code-unfold*]: $\text{foldr } f \ s \ \sigma = \text{List.foldr } f \ (\alpha \ s) \ \sigma$

end

Concatenation of Two Annotated Lists locale *al-app* = *al* +

constrains $\alpha :: 's \Rightarrow ('e \times 'a::\text{monoid-add}) \text{ list}$

fixes *app* :: $'s \Rightarrow 's \Rightarrow 's$

assumes *app-correct*:

$\llbracket \text{invar } s; \text{invar } s' \rrbracket \Longrightarrow \alpha \ (\text{app } s \ s') = (\alpha \ s) \ @ \ (\alpha \ s')$

$\llbracket \text{invar } s; \text{invar } s' \rrbracket \Longrightarrow \text{invar } (\text{app } s \ s')$

Readout the Summed up Annotations locale *al-annot* = *al* +

constrains $\alpha :: 's \Rightarrow ('e \times 'a::\text{monoid-add}) \text{ list}$

fixes *annot* :: $'s \Rightarrow 'a$

assumes *annot-correct*:

$\text{invar } s \Longrightarrow (\text{annot } s) = (\text{sum-list } (\text{map snd } (\alpha \ s)))$

Split by Monotone Predicate locale *al-splits* = *al* +

constrains $\alpha :: 's \Rightarrow ('e \times 'a::\text{monoid-add}) \text{ list}$

fixes *splits* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 's \Rightarrow$
 $('s \times ('e \times 'a) \times 's)$

assumes *splits-correct*:

$\llbracket \text{invar } s;$
 $\forall a \ b. \ p \ a \ \longrightarrow \ p \ (a + b);$
 $\neg \ p \ i;$
 $p \ (i + \text{sum-list } (\text{map snd } (\alpha \ s)));$
 $(\text{splits } p \ i \ s) = (l, (e, a), r) \rrbracket$

\Longrightarrow

$(\alpha \ s) = (\alpha \ l) \ @ \ (e, a) \ \# \ (\alpha \ r) \ \wedge$
 $\neg \ p \ (i + \text{sum-list } (\text{map snd } (\alpha \ l))) \ \wedge$
 $p \ (i + \text{sum-list } (\text{map snd } (\alpha \ l)) + a) \ \wedge$
 $\text{invar } l \ \wedge$
 $\text{invar } r$

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:  $p (i + \text{sum-list } (\text{map snd } (\alpha s)))$ 
obtains l e a r where
  (splits p i s) = (l, (e,a), r)
  ( $\alpha s$ ) = ( $\alpha l$ ) @ (e,a) # ( $\alpha r$ )
   $\neg p (i + \text{sum-list } (\text{map snd } (\alpha l)))$ 
   $p (i + \text{sum-list } (\text{map snd } (\alpha l)) + a)$ 
  invar l
  invar r
  <proof>
end

```

Record Based Interface

```

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

locale StdALDefs = poly-al-fold alist-op- $\alpha$  ops  alist-op-invar ops
  for ops :: ('e,'a::monoid-add,'s,'more) alist-ops-scheme
begin
  abbreviation  $\alpha$  where  $\alpha ==$  alist-op- $\alpha$  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  $\alpha$  invar +
  al-empty  $\alpha$  invar empty +
  al-isEmpty  $\alpha$  invar isEmpty +
  al-count  $\alpha$  invar count +
  al-consl  $\alpha$  invar consl +
  al-consr  $\alpha$  invar consr +
  al-head  $\alpha$  invar head +
  al-tail  $\alpha$  invar tail +
  al-headR  $\alpha$  invar headR +
  al-tailR  $\alpha$  invar tailR +
  al-app  $\alpha$  invar app +
  al-annot  $\alpha$  invar annot +
  al-splits  $\alpha$  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  $\alpha$  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  $\alpha :: 'p \Rightarrow ('e \times 'a::linorder)$  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 $find-correct: \llbracket invar p; \alpha p \neq \{\#\} \rrbracket \Longrightarrow$
 $(find p) \in \# (\alpha p) \wedge (\forall y \in set-mset (\alpha p). snd (find p) \leq snd y)$

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 $meld-correct:$
 $\llbracket invar p; invar p' \rrbracket \Longrightarrow invar (meld p p')$
 $\llbracket invar p; invar p' \rrbracket \Longrightarrow \alpha (meld p p') = (\alpha p) + (\alpha p')$

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 $delete-correct:$
 $\llbracket invar p; \alpha p \neq \{\#\} \rrbracket \Longrightarrow invar (delete p)$
 $\llbracket invar p; \alpha p \neq \{\#\} \rrbracket \Longrightarrow \alpha (delete p) = (\alpha p) - \{\# (find p) \#}$

Record based interface

```

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

locale StdPrioDefs =
  fixes ops :: ('e,'a::linorder,'p) prio-ops
begin
  abbreviation  $\alpha$  where  $\alpha$  == prio-op- $\alpha$  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  $\alpha$  invar +
  prio-empty  $\alpha$  invar empty +
  prio-isEmpty  $\alpha$  invar isEmpty +
  prio-find  $\alpha$  invar find +
  prio-insert  $\alpha$  invar insert +
  prio-meld  $\alpha$  invar meld +
  prio-delete  $\alpha$  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  $\alpha$  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  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::linorder)$ 
  fixes  $invar :: 's \Rightarrow bool$ 

locale uprio-no-invar = uprio +
  assumes  $invar[simp, intro!]: \bigwedge s. invar\ s$ 

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

```

Basic Upriority Queue Functions

```

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

```

```

Emptiness Predicate locale uprio-isEmpty = uprio +
  constrains  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::linorder)$ 
  fixes  $isEmpty :: 's \Rightarrow bool$ 
  assumes  $isEmpty-correct:$ 
   $invar\ s \implies (isEmpty\ s) = (\alpha\ s = Map.empty)$ 

```

```

Find and Remove Minimal Element locale uprio-pop = uprio +
  constrains  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::linorder)$ 
  fixes  $pop :: 's \Rightarrow ('e \times 'a \times 's)$ 
  assumes  $pop-correct:$ 
   $[[invar\ s; \alpha\ s \neq Map.empty; pop\ s = (e, a, s')]] \implies$ 
   $invar\ s' \wedge$ 
   $\alpha\ s' = (\alpha\ s)(e := None) \wedge$ 
   $(\alpha\ s)\ e = Some\ a \wedge$ 
   $(\forall y \in ran\ (\alpha\ s). a \leq y)$ 

```

```

begin

```

```

  lemma popE:

```

```

assumes
  invar s
   $\alpha s \neq \text{Map.empty}$ 
obtains  $e a s'$  where
   $\text{pop } s = (e, a, s')$ 
  invar s'
   $\alpha s' = (\alpha s)(e := \text{None})$ 
   $(\alpha s) e = \text{Some } a$ 
   $(\forall y \in \text{ran } (\alpha s). a \leq y)$ 
  <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  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::\text{linorder})$ 
  fixes  $\text{insert} :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$ 
  assumes insert-correct:
   $\text{invar } s \Longrightarrow \text{invar } (\text{insert } s e a)$ 
   $\text{invar } s \Longrightarrow \alpha (\text{insert } s e a) = (\alpha s)(e \mapsto a)$ 

```

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

```

locale uprio-distinct-insert = uprio +
  constrains  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::\text{linorder})$ 
  fixes  $\text{insert} :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$ 
  assumes distinct-insert-correct:
   $\llbracket \text{invar } s; e \notin \text{dom } (\alpha s) \rrbracket \Longrightarrow \text{invar } (\text{insert } s e a)$ 
   $\llbracket \text{invar } s; e \notin \text{dom } (\alpha s) \rrbracket \Longrightarrow \alpha (\text{insert } s e a) = (\alpha s)(e \mapsto a)$ 

```

```

Looking up Priorities locale uprio-prio = uprio +
  constrains  $\alpha :: 's \Rightarrow ('e \rightarrow 'a::\text{linorder})$ 
  fixes  $\text{prio} :: 's \Rightarrow 'e \Rightarrow 'a \text{ option}$ 
  assumes prio-correct:
   $\text{invar } s \Longrightarrow \text{prio } s e = (\alpha s) e$ 

```

Record Based Interface

```

record ( $'e, 'a, 's$ ) uprio-ops =
   $\text{upr-}\alpha :: 's \Rightarrow ('e \rightarrow 'a)$ 
   $\text{upr-invar} :: 's \Rightarrow \text{bool}$ 
   $\text{upr-empty} :: \text{unit} \Rightarrow 's$ 
   $\text{upr-isEmpty} :: 's \Rightarrow \text{bool}$ 
   $\text{upr-insert} :: 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$ 
   $\text{upr-pop} :: 's \Rightarrow ('e \times 'a \times 's)$ 
   $\text{upr-prio} :: 's \Rightarrow 'e \Rightarrow 'a \text{ option}$ 

```

```

locale StdUprioDefs =
  fixes ops :: ('e,'a::linorder,'s, 'more) uprio-ops-scheme
begin
  abbreviation  $\alpha$  where  $\alpha == \text{upr-}\alpha$  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  $\alpha$  invar +
  uprio-empty  $\alpha$  invar empty +
  uprio-isEmpty  $\alpha$  invar isEmpty +
  uprio-insert  $\alpha$  invar insert +
  uprio-pop  $\alpha$  invar pop +
  uprio-prio  $\alpha$  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  $\alpha$  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) =

```

$it (\lambda-. True) (\lambda x \sigma. ins x \sigma) s$

lemma *iterate-add-to-set-correct* :
assumes *ins-OK*: *set-ins* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
shows α (*iterate-add-to-set s ins it*) = $S0 \cup \alpha s \wedge invar$ (*iterate-add-to-set s ins it*)
 $\langle proof \rangle$

lemma *iterate-add-to-set-dj-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins-dj*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: $S0 \cap \alpha s = \{\}$
shows α (*iterate-add-to-set s ins-dj it*) = $S0 \cup \alpha s \wedge invar$ (*iterate-add-to-set s ins-dj it*)
 $\langle proof \rangle$

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] :
 $iterate-to-set emp ins-dj (it::('x,'x-set) set-iterator) =$
 $it (\lambda-. True) (\lambda x \sigma. ins-dj x \sigma) (emp ())$
 $\langle proof \rangle$

lemma *iterate-to-set-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins-dj*
assumes *emp-OK*: *set-empty* α *invar emp*
assumes *it*: *set-iterator it S0*
shows α (*iterate-to-set emp ins-dj it*) = $S0 \wedge invar$ (*iterate-to-set emp ins-dj it*)
 $\langle proof \rangle$

Iterate image/filter add to Set

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

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

$\langle proof \rangle$

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)*) =
 $\{b . \exists a. a \in S0 \wedge f a = \text{Some } b\} \wedge$
invar (iterate-to-set emp ins (set-iterator-image-filter f it))
 $\langle proof \rangle$

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*: $\{y. \exists x. x \in S0 \wedge f x = \text{Some } y\} \cap \alpha s = \{\}$
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)*) =
 $\{b . \exists a. a \in S0 \wedge f a = \text{Some } b\} \cup \alpha s \wedge$
invar (iterate-add-to-set s ins (set-iterator-image-filter f it))
 $\langle proof \rangle$

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)*) =
 $\{b . \exists a. a \in S0 \wedge f a = \text{Some } b\} \wedge$
invar (iterate-to-set emp ins (set-iterator-image-filter f it))
 $\langle proof \rangle$

Iterate diff Set

definition *iterate-diff-set where*
iterate-diff-set s del (it::('x,'x-set) set-iterator) =
it (λ -. True) ($\lambda x \sigma. del x \sigma$) 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)*
 $\langle proof \rangle$

Iterate add to Map**definition** *iterate-add-to-map* **where**

$$\begin{aligned} & \textit{iterate-add-to-map } m \textit{ update } (it::('k \times 'v, 'kv\text{-map}) \textit{ set-iterator}) = \\ & it (\lambda-. \textit{ True}) (\lambda(k,v) \sigma. \textit{ update } k \ v \ \sigma) \ m \end{aligned}$$
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* \wedge *invar* (*iterate-add-to-map m upd it*)*<proof>***lemma** *iterate-add-to-map-dj-correct* :**assumes** *upd-OK*: *map-update-dj* α *invar upd***assumes** *m-OK*: *invar m***assumes** *it*: *map-iterator it M***assumes** *dj*: *dom M* \cap *dom* (α *m*) = {}**shows** α (*iterate-add-to-map m upd it*) = α *m* ++ *M* \wedge *invar* (*iterate-add-to-map m upd it*)*<proof>***Iterator to Map****definition** *iterate-to-map* **where**

$$\begin{aligned} & \textit{iterate-to-map } emp \textit{ upd-dj } (it::('k \times 'v, 'kv\text{-map}) \textit{ set-iterator}) = \\ & \textit{iterate-add-to-map } (emp ()) \textit{ upd-dj } it \end{aligned}$$
lemma *iterate-to-map-alt-def*[*code*] :*iterate-to-map emp upd-dj it* =*it* ($\lambda-. \textit{ True}$) ($\lambda(k, v) \sigma. \textit{ upd-dj } k \ v \ \sigma$) (*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* \wedge *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- $\alpha$  :: ('k,'v,'s) map- $\alpha$ 
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  $\Rightarrow$  ('k,'v,('k  $\times$  'v) list) map-iterator
  bmap-op-rev-list-it :: 's  $\Rightarrow$  ('k,'v,('k  $\times$  '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  $\alpha$  where  $\alpha ==$  bmap-op- $\alpha$  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
     $\equiv$  bmap-op-ordered-list-it ops
  abbreviation rev-list-it where rev-list-it
     $\equiv$  bmap-op-rev-list-it ops
end

```

```

locale StdBasicMap = StdBasicMapDefs ops +
  map  $\alpha$  invar +
  map-empty  $\alpha$  invar empty +
  map-lookup  $\alpha$  invar lookup +
  map-update  $\alpha$  invar update +
  map-update-dj  $\alpha$  invar update-dj +
  map-delete  $\alpha$  invar delete +
  poly-map-iteratei  $\alpha$  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  $\alpha$  invar ordered-list-it +
  poly-map-rev-iterateoi  $\alpha$  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  $\equiv$  update k v (empty ())

```

```

  definition g-add m1 m2  $\equiv$  iterate m2 ( $\lambda(k,v) \sigma. \text{update } k \ v \ \sigma$ ) m1

```

```

  definition

```

```

    g-sel m P  $\equiv$ 

```

```

    iteratei m ( $\lambda \sigma. \sigma = \text{None}$ ) ( $\lambda x \ \sigma. \text{if } P \ x \ \text{then } \text{Some } x \ \text{else } \text{None}$ ) None

```

```

  definition g-bex m P  $\equiv$  iteratei m ( $\lambda x. \neg x$ ) ( $\lambda kv \ \sigma. P \ kv$ ) False

```

```

  definition g-ball m P  $\equiv$  iteratei m id ( $\lambda kv \ \sigma. P \ kv$ ) True

```

```

  definition g-size m  $\equiv$  iterate m ( $\lambda-. \text{Suc}$ ) (0::nat)

```

```

  definition g-size-abort b m  $\equiv$  iteratei m ( $\lambda s. s < b$ ) ( $\lambda-. \text{Suc}$ ) (0::nat)

```

```

  definition g-isEmpty m  $\equiv$  g-size-abort 1 m = 0

```

```

  definition g-isSng m  $\equiv$  g-size-abort 2 m = 1

```

```

  definition g-to-list m  $\equiv$  iterate m (#) []

```

```

  definition g-list-to-map l  $\equiv$  foldl ( $\lambda m \ (k,v). \text{update } k \ v \ m$ ) (empty ())
    (rev l)

```

```

  definition g-add-dj m1 m2  $\equiv$  iterate m2 ( $\lambda(k,v) \sigma. \text{update-dj } k \ v \ \sigma$ ) m1

```

```

  definition g-restrict P m  $\equiv$  iterate m

```

```

    ( $\lambda(k,v) \sigma. \text{if } P \ (k,v) \ \text{then } \text{update-dj } k \ v \ \sigma \ \text{else } \sigma$ ) (empty ())

```

```

  definition dflt-ops :: ('k,'v,'s) map-ops

```

```

  where [icf-rec-def]:

```

```

    dflt-ops  $\equiv$ 

```

```

    (|

```

```

      map-op- $\alpha$  =  $\alpha$ ,

```

```

      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* \implies *map-iterator* *m it*

⟨proof⟩

lemma *map-rev-iterator-linord-is-it*:

map-iterator-rev-linord *m it* \implies *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 \equiv
iterateoi m ($\lambda\sigma. \sigma = \text{None}$) ($\lambda x \sigma. \text{if } P \ x \ \text{then } \text{Some } x \ \text{else } \text{None}$) None

definition

g-max m P \equiv
rev-iterateoi m ($\lambda\sigma. \sigma = \text{None}$) ($\lambda x \sigma. \text{if } P \ x \ \text{then } \text{Some } x \ \text{else } \text{None}$) None

definition *g-to-sorted-list m \equiv rev-iterateo m (#) []*

definition *g-to-rev-list m \equiv iterateo m (#) []*

definition *dflt-ops :: ('k,'v,'s) omap-ops*

where [*icf-rec-def*]:

dflt-ops \equiv 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  $\alpha$  invar g-min
  <proof>

  lemma g-max-impl: map-max  $\alpha$  invar g-max
  <proof>

  lemma g-to-sorted-list-impl: map-to-sorted-list  $\alpha$  invar g-to-sorted-list
  <proof>

  lemma g-to-rev-list-impl: map-to-rev-list  $\alpha$  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  $\equiv$  m1.iterate m1 ( $\lambda kv \sigma$ . case f kv of
    None =>  $\sigma$ 
    | Some (k',v') => m2.update-dj k' v'  $\sigma$ 
    ) (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  $\equiv$  s1.iterate s s2.ins-dj (s2.empty ())
```

```
  definition g-filter P s1  $\equiv$  s1.iterate s1
    ( $\lambda x \sigma$ . if P x then s2.ins-dj x  $\sigma$  else  $\sigma$ )
    (s2.empty ())
```

```
  definition g-union s1 s2  $\equiv$  s1.iterate s1 s2.ins s2
```

```
  definition g-diff s1 s2  $\equiv$  s2.iterate s2 s1.delete s1
```

```
  definition g-union-list where
```

```
    g-union-list l =
      foldl ( $\lambda s s'$ . g-union s' s) (s2.empty ()) l
```

```
  definition g-union-dj s1 s2  $\equiv$  s1.iterate s1 s2.ins-dj s2
```

```
  definition g-disjoint-witness s1 s2  $\equiv$ 
    s1.sel s1 ( $\lambda x$ . s2.memb x s2)
```

```
  definition g-disjoint s1 s2  $\equiv$ 
    s1.ball s1 ( $\lambda x$ .  $\neg$ 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. $\alpha$  s1.invar s2. $\alpha$  s2.invar g-copy
    <proof>
```

```
  lemma g-filter-impl: set-filter s1. $\alpha$  s1.invar s2. $\alpha$  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 (λx 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* $f\ s \equiv$
 $s1.iterate\ s$
 $(\lambda x\ res.\ case\ f\ x\ of\ Some\ v \Rightarrow s2.ins\ v\ res\ |\ - \Rightarrow res)$
 $(s2.empty\ ())$

definition *g-image* $f\ s \equiv$
 $s1.iterate\ s\ (\lambda x\ res.\ s2.ins\ (f\ x)\ res)\ (s2.empty\ ())$

definition *g-inj-image-filter* $f\ s \equiv$
 $s1.iterate\ s$
 $(\lambda x\ res.\ case\ f\ x\ of\ Some\ v \Rightarrow s2.ins-dj\ v\ res\ |\ - \Rightarrow res)$
 $(s2.empty\ ())$

definition *g-inj-image* $f\ s \equiv$
 $s1.iterate\ s\ (\lambda x\ res.\ s2.ins-dj\ (f\ x)\ res)\ (s2.empty\ ())$

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*:
 $set-image-filter\ s1.\alpha\ s1.invar\ s2.\alpha\ s2.invar\ g-image-filter$
 $\langle proof \rangle$

lemma *g-image-alt*: $g-image\ f\ s = g-image-filter\ (Some\ o\ f)\ s$
 $\langle proof \rangle$

lemma *g-image-impl*: $set-image\ s1.\alpha\ s1.invar\ s2.\alpha\ s2.invar\ g-image$
 $\langle proof \rangle$

lemma *g-inj-image-filter-impl*:
 $set-inj-image-filter\ s1.\alpha\ s1.invar\ s2.\alpha\ s2.invar\ g-inj-image-filter$
 $\langle proof \rangle$

lemma *g-inj-image-alt*: $g-inj-image\ f\ s = g-inj-image-filter\ (Some\ o\ f)\ s$
 $\langle proof \rangle$

lemma *g-inj-image-impl*:
 $set-inj-image\ s1.\alpha\ s1.invar\ s2.\alpha\ s2.invar\ g-inj-image$
 $\langle proof \rangle$

end

sublocale *g-set-xy-loc* < *set-image-filter* $s1.\alpha\ s1.invar\ s2.\alpha\ s2.invar$
 $g-image-filter\ \langle proof \rangle$

```
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 (λx 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  $\Rightarrow$  ('x,'x list) set-iterator
  bset-op-rev-list-it :: 's  $\Rightarrow$  ('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  $\alpha$  where  $\alpha ==$  bset-op- $\alpha$  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  $\equiv$  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  $\equiv$  bset-op-ordered-list-it ops
  abbreviation rev-list-it  $\equiv$  bset-op-rev-list-it ops
end

```

```

locale StdBasicSet = StdBasicSetDefs ops +
  set  $\alpha$  invar +
  set-empty  $\alpha$  invar empty +
  set-memb  $\alpha$  invar memb +
  set-ins  $\alpha$  invar ins +
  set-ins-dj  $\alpha$  invar ins-dj +
  set-delete  $\alpha$  invar delete +
  poly-set-iteratei  $\alpha$  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  $\alpha$  invar ordered-list-it +
poly-set-rev-iterateoi  $\alpha$  invar rev-list-it
for ops :: ('x::linorder,'s,'more) oset-basic-ops-scheme
begin
end

context StdBasicSetDefs
begin
definition g-sng  $x \equiv \text{ins } x \text{ (empty ())}$ 
definition g-isEmpty  $s \equiv \text{iteratei } s \text{ } (\lambda c. c) \text{ } (\lambda -. \text{False}) \text{ True}$ 
definition g-sel'  $s P \equiv \text{iteratei } s \text{ } ((=) \text{ None})$ 
   $(\lambda x -. \text{if } P x \text{ then Some } x \text{ else None}) \text{ None}$ 

definition g-ball  $s P \equiv \text{iteratei } s \text{ } (\lambda c. c) \text{ } (\lambda x \sigma. P x) \text{ True}$ 
definition g-bex  $s P \equiv \text{iteratei } s \text{ } (\lambda c. \neg c) \text{ } (\lambda x \sigma. P x) \text{ False}$ 
definition g-size  $s \equiv \text{iteratei } s \text{ } (\lambda -. \text{True}) \text{ } (\lambda x n. \text{Suc } n) \text{ 0}$ 
definition g-size-abort  $m s \equiv \text{iteratei } s \text{ } (\lambda \sigma. \sigma < m) \text{ } (\lambda x. \text{Suc } 0) \text{ 0}$ 
definition g-isSng  $s \equiv (\text{iteratei } s \text{ } (\lambda \sigma. \sigma < 2) \text{ } (\lambda x. \text{Suc } 0) \text{ 1})$ 

definition g-union  $s1 s2 \equiv \text{iterate } s1 \text{ ins } s2$ 
definition g-diff  $s1 s2 \equiv \text{iterate } s2 \text{ delete } s1$ 

definition g-subset  $s1 s2 \equiv \text{g-ball } s1 \text{ } (\lambda x. \text{memb } x s2)$ 

definition g-equal  $s1 s2 \equiv \text{g-subset } s1 s2 \wedge \text{g-subset } s2 s1$ 

definition g-to-list  $s \equiv \text{iterate } s \text{ } (\#) \text{ []}$ 

fun g-from-list-aux where
  g-from-list-aux  $\text{accs []} = \text{accs}$  |
  g-from-list-aux  $\text{accs } (x\#l) = \text{g-from-list-aux } (\text{ins } x \text{ accs}) l$ 
  — Tail recursive version

definition g-from-list  $l == \text{g-from-list-aux } (\text{empty ()}) l$ 

definition g-inter  $s1 s2 \equiv$ 
   $\text{iterate } s1 \text{ } (\lambda x s. \text{if } \text{memb } x s2 \text{ then ins-dj } x s \text{ else } s)$ 
   $(\text{empty ()})$ 

definition g-union-dj  $s1 s2 \equiv \text{iterate } s1 \text{ ins-dj } s2$ 
definition g-filter  $P s \equiv \text{iterate } s$ 
   $(\lambda x \sigma. \text{if } P x \text{ then ins-dj } x \sigma \text{ else } \sigma)$ 
   $(\text{empty ()})$ 

definition g-disjoint-witness  $s1 s2 \equiv \text{g-sel}' s1 \text{ } (\lambda x. \text{memb } x s2)$ 
definition g-disjoint  $s1 s2 \equiv \text{g-ball } s1 \text{ } (\lambda x. \neg \text{memb } x s2)$ 

definition dflt-ops

```

```

where [icf-rec-def]: dflt-ops  $\equiv$  (|
  set-op- $\alpha$  =  $\alpha$ ,
  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
|)

```

$\langle ML \rangle$

end

context StdBasicSet

begin

lemma g-sng-impl: set-sng α invar g-sng
 $\langle proof \rangle$

lemma g-ins-dj-impl: set-ins-dj α invar ins
 $\langle proof \rangle$

lemma g-isEmpty-impl: set-isEmpty α invar g-isEmpty
 $\langle proof \rangle$

lemma g-sel'-impl: set-sel' α invar g-sel'
 $\langle proof \rangle$

lemma g-ball-alt: g-ball s P = iterate-ball (iteratei s) P
 $\langle proof \rangle$

lemma *g-bex-alt*: $g\text{-bex } s P = \text{iterate-bex } (\text{iteratei } s) P$
 ⟨proof⟩

lemma *g-ball-impl*: $\text{set-ball } \alpha \text{ invar } g\text{-ball}$
 ⟨proof⟩

lemma *g-bex-impl*: $\text{set-bex } \alpha \text{ invar } g\text{-bex}$
 ⟨proof⟩

lemma *g-size-alt*: $g\text{-size } s = \text{iterate-size } (\text{iteratei } s)$
 ⟨proof⟩

lemma *g-size-abort-alt*: $g\text{-size-abort } m s = \text{iterate-size-abort } (\text{iteratei } s) m$
 ⟨proof⟩

lemma *g-size-impl*: $\text{set-size } \alpha \text{ invar } g\text{-size}$
 ⟨proof⟩

lemma *g-size-abort-impl*: $\text{set-size-abort } \alpha \text{ invar } g\text{-size-abort}$
 ⟨proof⟩

lemma *g-isSng-alt*: $g\text{-isSng } s = \text{iterate-is-sng } (\text{iteratei } s)$
 ⟨proof⟩

lemma *g-isSng-impl*: $\text{set-isSng } \alpha \text{ invar } g\text{-isSng}$
 ⟨proof⟩

lemma *g-union-impl*: $\text{set-union } \alpha \text{ invar } \alpha \text{ invar } \alpha \text{ invar } g\text{-union}$
 ⟨proof⟩

lemma *g-diff-impl*: $\text{set-diff } \alpha \text{ invar } \alpha \text{ invar } g\text{-diff}$
 ⟨proof⟩

lemma *g-subset-impl*: $\text{set-subset } \alpha \text{ invar } \alpha \text{ invar } g\text{-subset}$
 ⟨proof⟩

lemma *g-equal-impl*: $\text{set-equal } \alpha \text{ invar } \alpha \text{ invar } g\text{-equal}$
 ⟨proof⟩

lemma *g-to-list-impl*: $\text{set-to-list } \alpha \text{ invar } g\text{-to-list}$
 ⟨proof⟩

lemma *g-from-list-impl*: $\text{list-to-set } \alpha \text{ invar } g\text{-from-list}$
 ⟨proof⟩

lemma *g-inter-impl*: $\text{set-inter } \alpha \text{ invar } \alpha \text{ invar } \alpha \text{ invar } g\text{-inter}$
 ⟨proof⟩

lemma *g-union-dj-impl*: $\text{set-union-dj } \alpha \text{ invar } \alpha \text{ invar } \alpha \text{ invar } g\text{-union-dj}$
 ⟨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 \equiv iterateoi s ($\lambda x. x = \text{None}$)*
(λx -. if P x then Some x else None) None

definition *g-max s P \equiv rev-iterateoi s ($\lambda x. x = \text{None}$)*
(λx -. if P x then Some x else None) None

definition *g-to-sorted-list s \equiv rev-iterateo s (#) []*

definition *g-to-rev-list s \equiv iterateo s (#) []*

definition *dflt-ops :: ('x::linorder, 's) oset-ops*

where [*icf-rec-def*]:

dflt-ops \equiv 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 *dfft-oops-impl: StdOSet dfft-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 (λx res.

s2.iterate s2 (λy res.

case (f (x, y)) of

None \Rightarrow res

| Some z \Rightarrow (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 \equiv

image-filter-cartesian-product

$(\lambda xy. \text{if } P \ xy \text{ then } \text{Some } (f \ xy) \text{ else } \text{None}) \ s1 \ s2$

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 ($s1.iteratei$ $s1$) ($\lambda-. s2.iteratei$ $s2$)))
 ⟨*proof*⟩

definition *inj-image-filter-cp* **where**
inj-image-filter-cp f P $s1$ $s2$ \equiv
inj-image-filter-cartesian-product
 ($\lambda xy. \text{if } P \text{ } xy \text{ then } \text{Some } (f \text{ } xy) \text{ else } \text{None}$) $s1$ $s2$

end

locale *inj-image-filter-cp-loc* = *inj-image-filter-cp-defs-loc* $ops1$ $ops2$ $ops3$ +
 $s1: \text{StdSet } ops1$ +
 $s2: \text{StdSet } ops2$ +
 $s3: \text{StdSet } ops3$
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 $f :: 'x \times 'y \rightarrow 'z$
assumes $I[simp, intro!]: s1.invar \ s1 \quad s2.invar \ s2$
assumes $INJ: inj\text{-on } f \ (s1.\alpha \ s1 \times s2.\alpha \ s2 \cap dom \ f)$
shows $s3.\alpha \ (inj\text{-image-filter-cartesian-product } f \ s1 \ s2)$
 = $\{ z \mid \exists x \ y \ z. f \ (x, y) = \text{Some } z \wedge x \in s1.\alpha \ s1 \wedge y \in s2.\alpha \ s2 \}$ (**is** ? $T1$)
 $s3.invar \ (inj\text{-image-filter-cartesian-product } f \ s1 \ s2)$ (**is** ? $T2$)
 ⟨*proof*⟩

lemma *inj-image-filter-cp-correct*:
assumes $I: s1.invar \ s1 \quad s2.invar \ s2$
assumes $INJ: inj\text{-on } f \ \{x \in s1.\alpha \ s1 \times s2.\alpha \ s2. P \ x\}$
shows
 $s3.\alpha \ (inj\text{-image-filter-cp } f \ P \ s1 \ s2)$
 = $\{ f \ (x, y) \mid x \ y. P \ (x, y) \wedge x \in s1.\alpha \ s1 \wedge y \in s2.\alpha \ s2 \}$ (**is** ? $T1$)
 $s3.invar \ (inj\text{-image-filter-cp } f \ P \ s1 \ s2)$ (**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 $ops3$:: ($'x \times 'y, 's3, 'more3$) *set-ops-scheme*
begin

definition *cart* *s1 s2* \equiv
s1.iterate s1
 $(\lambda x. s2.iterate s2 (\lambda y res. s3.ins-dj (x,y) res))$
 $(s3.empty ())$

lemma *cart-alt*: *cart s1 s2* $==$
inj-image-filter-cartesian-product Some s1 s2
 $\langle proof \rangle$

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* :: $('x \times 'y, 's3, 'more3)$ *set-ops-scheme*
begin

lemma *cart-correct*:
assumes $I[simp, intro!]$: *s1.invar s1* *s2.invar s2*
shows $s3.\alpha (cart\ s1\ s2)$
 $= s1.\alpha\ s1 \times s2.\alpha\ s2$ (**is** ?*T1*)
s3.invar (cart s1 s2) (**is** ?*T2*)
 $\langle proof \rangle$

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 *ift-image* *ift f s* $==$ *ift* $(\lambda x. Some (f\ x))\ s$

lemma *ift-image-correct*:
assumes *set-image-filter* $\alpha1\ invar1\ \alpha2\ invar2\ ift$
shows *set-image* $\alpha1\ invar1\ \alpha2\ invar2$ (*ift-image ift*)
 $\langle proof \rangle$

Injective Image-Filter (by image-filter)

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

lemma *ift-inj-image-correct*:
assumes *set-inj-image-filter* $\alpha1\ invar1\ \alpha2\ invar2\ ift$
shows *set-inj-image* $\alpha1\ invar1\ \alpha2\ invar2$ (*ift-inj-image ift*)
 $\langle proof \rangle$

Filter (by image-filter)

definition *iflt-filter* $\text{iflt } P \ s == \text{iflt } (\lambda x. \text{ if } P \ x \text{ then } \text{Some } x \text{ else } \text{None}) \ s$

lemma *iflt-filter-correct*:

fixes $\alpha 1 :: 's1 \Rightarrow 'a \ \text{set}$

fixes $\alpha 2 :: 's2 \Rightarrow 'a \ \text{set}$

assumes *set-inj-image-filter* $\alpha 1 \ \text{invar1} \ \alpha 2 \ \text{invar2} \ \text{iflt}$

shows *set-filter* $\alpha 1 \ \text{invar1} \ \alpha 2 \ \text{invar2} \ (\text{iflt-filter } \text{iflt})$

<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* $l \ (\lambda x. \ \text{True}) \ (\lambda x \ l. \ l@[f \ x]) \ []$) $c \ f' \ \sigma 0$

$= \text{foldli } l \ c \ (f' \ o \ f) \ \sigma 0$

<proof>

locale *SetByMapDefs* =

map: *StdBasicMapDefs ops*

for *ops* :: $('x, \text{unit}, 's, 'more)$ *map-basic-ops-scheme*

begin

definition $\alpha \ s \equiv \text{dom } (\text{map}.\alpha \ s)$

definition *invar* $s \equiv \text{map.invar } s$

definition *empty* **where** *empty* $\equiv \text{map.empty}$

definition *memb* $x \ s \equiv \text{map.lookup } x \ s \neq \text{None}$

definition *ins* $x \ s \equiv \text{map.update } x \ () \ s$

definition *ins-dj* $x \ s \equiv \text{map.update-dj } x \ () \ s$

definition *delete* $x \ s \equiv \text{map.delete } x \ s$

definition *list-it* :: $'s \Rightarrow ('x, 'x \ \text{list}) \ \text{set-iterator}$

where *list-it* $s \ c \ f \ \sigma 0 \equiv \text{it-to-it } (\text{map.list-it } s) \ c \ (f \ o \ \text{fst}) \ \sigma 0$

<ML>

lemma *list-it-alt*: *list-it* $s = \text{map-iterator-dom } (\text{map.iteratei } s)$

<proof>

lemma *list-it-unfold*:

it-to-it (list-it s) c f $\sigma 0 = \text{map.iteratei s c (f o fst) } \sigma 0$
 <proof>

definition [*icf-rec-def*]: *dflt-basic-ops* \equiv (|

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-ops
  ⟨proof⟩

```

```

lemma proper-it'-map2set: proper-it' it it'
   $\implies$  proper-it' ( $\lambda s c f. it s c (f o fst)$ ) ( $\lambda s c f. it' s c (f o fst)$ )
  <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  $\alpha :: 's \Rightarrow 'a \text{ list}$ 
  assumes [simp]:  $\bigwedge s. \text{invar } s$ 
begin

```

```

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

```

```

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

```

```

lemma idx-iteratei-aux-simps[simp]:
   $i=0 \implies \text{idx-iteratei-aux } sz \ i \ l \ c \ f \ \sigma = \sigma$ 
   $\neg c \ \sigma \implies \text{idx-iteratei-aux } sz \ i \ l \ c \ f \ \sigma = \sigma$ 
   $\llbracket i \neq 0; c \ \sigma \rrbracket \implies \text{idx-iteratei-aux } sz \ i \ l \ c \ f \ \sigma = \text{idx-iteratei-aux } sz \ (i - 1) \ l \ c \ f$ 
  (f (get l (sz-i))  $\sigma$ )
  <proof>

```

```

definition idx-iteratei where

```

```

  idx-iteratei l c f  $\sigma \equiv \text{idx-iteratei-aux } (\text{size } l) \ (\text{size } l) \ l \ c \ f \ \sigma$ 

```

```

lemma idx-iteratei-correct:

```

```

  shows idx-iteratei s = foldli ( $\alpha$  s)
  <proof>

```

```

lemmas idx-iteratei-unfold[code-unfold] = idx-iteratei-correct[symmetric]

```

```

reverse_iteratei (by get, size) fun idx-reverse-iteratei-aux
  ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow 's \Rightarrow ('s \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 's \Rightarrow 's) \Rightarrow 's \Rightarrow 's$ 

```


where

```

idx-reverse-iteratei-aux sz i l c f  $\sigma$  = (
  if  $i=0 \vee \neg c \sigma$  then  $\sigma$ 
  else idx-reverse-iteratei-aux sz (i - 1) l c f (f (get l (i - 1))  $\sigma$ )
)

```

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

lemma *idx-reverse-iteratei-aux-simps*[*simp*]:

```

i=0  $\implies$  idx-reverse-iteratei-aux sz i l c f  $\sigma$  =  $\sigma$ 
 $\neg c \sigma \implies$  idx-reverse-iteratei-aux sz i l c f  $\sigma$  =  $\sigma$ 
[[i $\neq$ 0; c  $\sigma$ ]]  $\implies$  idx-reverse-iteratei-aux sz i l c f  $\sigma$ 
= idx-reverse-iteratei-aux sz (i - 1) l c f (f (get l (i - 1))  $\sigma$ )
<proof>

```

definition *idx-reverse-iteratei l c f σ*

```

== idx-reverse-iteratei-aux (size l) (size l) l c f  $\sigma$ 

```

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 $\alpha :: 's \Rightarrow 'a \text{ list}$

begin

definition *it-size* :: $'s \Rightarrow \text{nat}$

```

where it-size l == iterate l ( $\lambda x \text{ res. Suc res}$ ) (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 $\alpha :: 's \Rightarrow 'a \text{ list}$

begin

definition *rev-it-size* :: $'s \Rightarrow \text{nat}$

```

where rev-it-size l == rev-iterate l ( $\lambda x \text{ res. Suc res}$ ) (0::nat)

```

lemma *rev-it-size-impl*:

shows *list-size α invar rev-it-size*

<proof>

end

Get (by iterator)

locale *it-get-loc* = *poly-list-iteratei* +

constrains $\alpha :: 's \Rightarrow 'a \text{ list}$

begin

definition *it-get*:: $'s \Rightarrow \text{nat} \Rightarrow 'a$

where *it-get* *s* *i* \equiv

the (*snd* (*iteratei* *s*

$(\lambda(i,x). x = \text{None})$

$(\lambda x (i,-). \text{if } i=0 \text{ then } (0, \text{Some } x) \text{ else } (i - 1, \text{None}))$

$(i, \text{None}))$)

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* :: $('a \Rightarrow 'i) \Rightarrow 'a \text{ set} \Rightarrow 'i \Rightarrow 'a \text{ set}$

where *index* *f* *s* *i* == $\{ x \in s . f x = i \}$

lemma *indexI*: $\llbracket x \in s; f x = i \rrbracket \Longrightarrow x \in \text{index } f s i$ *<proof>*

lemma *indexD*:

$x \in \text{index } f s i \Longrightarrow x \in s$

$x \in \text{index } f s i \Longrightarrow f x = i$

<proof>

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

Indexing by Map

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

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

lemma *index-map-correct*: *im-α* (*index-map* f s) = *index* f s
 ⟨*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 ⇒ 'i) ⇒ 'x set ⇒ 'm ⇒ bool

where

is-index f s idx == *ci-invar* idx ∧ *ci-α'* idx = *index-map* f s

lemma *is-index-invar*: *is-index* f s idx ⇒ *ci-invar* idx

⟨*proof*⟩

lemma *is-index-correct*: *is-index* f s idx ⇒ *ci-α* idx = *index* f s

⟨*proof*⟩

definition *lookup* :: 'i ⇒ 'm ⇒ '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  $I[\text{simp}, \text{intro!}]$ : is-index f s idx
  shows
     $s.\alpha (\text{lookup } i \text{ idx}) = \text{index } f \ s \ i$ 
     $s.\text{invar } (\text{lookup } i \ \text{idx})$ 
  <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  $\Rightarrow$  'i)  $\Rightarrow$  'x  $\Rightarrow$  'm  $\Rightarrow$  'm where
  idx-build-stepfun f x m ==
    let i=f x in
      (case m.lookup i m of
        None  $\Rightarrow$  m.update i (s.ins x (s.empty ())) m |
        Some s  $\Rightarrow$  m.update i (s.ins x s) m
      )

```

```

definition idx-build :: ('x  $\Rightarrow$  'i)  $\Rightarrow$  't  $\Rightarrow$  '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  $ci\text{-}\alpha'$  (idx-build f t) = index-map f (t. $\alpha$  t) (is ?T1) and
  [simp]:  $ci\text{-invar}$  (idx-build f t) (is ?T2)
  <proof>

```

```

lemma idx-build-is-index:
   $t.\text{invar } t \Longrightarrow \text{is-index } f \ (t.\alpha \ t) \ (\text{idx-build } f \ t)$ 
  <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* \Rightarrow *bool*
where *inatseg* *s* == $\exists k. s = \{i::nat. i < k\}$

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* \Rightarrow '*m* **where**
map-to-nat *s* ==
snd (*s.iterate* *s* ($\lambda x (c,m). (c+1, m.update\ x\ c\ m)$) (0, *m.empty* ()))

lemma *map-to-nat-correct*:

assumes *INV*[*simp*]: *s.invar* *s*
shows
— All elements have got a number
 $dom\ (m.\alpha\ (map-to-nat\ s)) = s.\alpha\ s$ (**is** ?*T1*) **and**
— No two elements got the same number
[*rule-format*]: $inj-on\ (m.\alpha\ (map-to-nat\ s))\ (s.\alpha\ s)$ (**is** ?*T2*) **and**
— Numbering is inatseg
[*rule-format*]: $inatseg\ (ran\ (m.\alpha\ (map-to-nat\ s)))$ (**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 $('e, 'a) \text{ Prio} = \text{Infty} \mid \text{Prio } 'e \ 'a$

```

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]*: $p\text{-min } a \text{ Infty} = a$ \langle proof \rangle

lemma *p-min-le-neut[simp]*: $p\text{-min } \text{Infty } a = a$ \langle proof \rangle

lemma *p-min-asso*: $p\text{-min } (p\text{-min } a \ b) \ c = p\text{-min } a \ (p\text{-min } b \ c)$
 \langle proof \rangle

lemma *lp-mono*: *class.monoid-add* $p\text{-min } \text{Infty}$
 \langle proof \rangle

instantiation *Prio* :: (*type*, *linorder*) *monoid-add*

begin

definition *zero-def*: $0 == \text{Infty}$

definition *plus-def*: $a + b == p\text{-min } a \ b$

instance $\langle proof \rangle$
end

fun $p\text{-less}\text{-eq} :: ('e, 'a::linorder) Prio \Rightarrow ('e, 'a) Prio \Rightarrow bool$ **where**
 $p\text{-less}\text{-eq} (Prio\ e\ a) (Prio\ f\ b) = (a \leq b)$
 $p\text{-less}\text{-eq} - Infty = True$
 $p\text{-less}\text{-eq} Infty (Prio\ e\ a) = False$

fun $p\text{-less} :: ('e, 'a::linorder) Prio \Rightarrow ('e, 'a) Prio \Rightarrow bool$ **where**
 $p\text{-less} (Prio\ e\ a) (Prio\ f\ b) = (a < b)$
 $p\text{-less} (Prio\ e\ a) Infty = True$
 $p\text{-less} Infty - = False$

lemma $p\text{-less}\text{-le}\text{-not}\text{-le} : p\text{-less}\ x\ y \longleftrightarrow p\text{-less}\text{-eq}\ x\ y \wedge \neg (p\text{-less}\text{-eq}\ y\ x)$
 $\langle proof \rangle$

lemma $p\text{-order}\text{-refl} : p\text{-less}\text{-eq}\ x\ x$
 $\langle proof \rangle$

lemma $p\text{-le}\text{-inf} : p\text{-less}\text{-eq}\ Infty\ x \Longrightarrow x = Infty$
 $\langle proof \rangle$

lemma $p\text{-order}\text{-trans} : \llbracket p\text{-less}\text{-eq}\ x\ y; p\text{-less}\text{-eq}\ y\ z \rrbracket \Longrightarrow p\text{-less}\text{-eq}\ x\ z$
 $\langle proof \rangle$

lemma $p\text{-linear}2 : p\text{-less}\text{-eq}\ x\ y \vee p\text{-less}\text{-eq}\ y\ x$
 $\langle proof \rangle$

instantiation $Prio :: (type, linorder) preorder$
begin

definition $p\text{lesseq}\text{-def} : \text{less}\text{-eq} = p\text{-less}\text{-eq}$

definition $p\text{less}\text{-def} : \text{less} = p\text{-less}$

instance
 $\langle proof \rangle$

end

Operations **definition** $alprio\text{-}\alpha :: ('s \Rightarrow (unit \times ('e, 'a::linorder) Prio) list)$
 $\Rightarrow 's \Rightarrow ('e \times 'a::linorder) multiset$
where
 $alprio\text{-}\alpha\ \alpha\ al == (mset (map\ p\text{-unwrap} (map\ snd\ (\alpha\ al))))$

definition $alprio\text{-invar} :: ('s \Rightarrow (unit \times ('c, 'd::linorder) Prio) list)$
 $\Rightarrow ('s \Rightarrow bool) \Rightarrow 's \Rightarrow bool$
where
 $alprio\text{-invar}\ \alpha\ invar\ al == invar\ al \wedge (\forall\ x \in set\ (\alpha\ al). snd\ x \neq Infty)$

definition *alprio-empty* **where**

alprio-empty *empt* = *empt*

definition *alprio-isEmpty* **where**

alprio-isEmpty *isEmpty* = *isEmpty*

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

where

alprio-insert *consl e a s* = *consl* () (*Prio e a*) *s*

definition *alprio-find* :: (*'s* \Rightarrow (*'e, 'a::linorder*) *Prio*) \Rightarrow *'s* \Rightarrow (*'e* \times *'a*)

where

alprio-find *annot s* = *p-unwrap* (*annot s*)

definition *alprio-delete* :: (((*'e, 'a::linorder*) *Prio* \Rightarrow *bool*)

\Rightarrow (*'e, 'a*) *Prio* \Rightarrow *'s* \Rightarrow (*'s* \times (*unit* \times (*'e, 'a*) *Prio*) \times *'s*)

\Rightarrow (*'s* \Rightarrow (*'e, 'a*) *Prio*) \Rightarrow (*'s* \Rightarrow *'s* \Rightarrow *'s*) \Rightarrow *'s* \Rightarrow *'s*

where

alprio-delete *splits annot app s* = (*let* (*l, -, r*)

= *splits* ($\lambda x. x \leq (\text{annot } s)$) *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*) $\leq y \vee y \leq x$

<proof>

lemma *p-min-mon*: (*x::('e, 'a::linorder) Prio*) $\leq y \implies (z + x) \leq y$

<proof>

lemma *p-min-mon2*: $p\text{-less-eg } x \ y \implies p\text{-less-eg } (p\text{-min } z \ x) \ y$
 <proof>

lemma *ls-min*: $\forall x \in \text{set } (xs::('e,'a::\text{linorder}) \text{ Prio list}) . \text{sum-list } xs \leq x$
 <proof>

lemma *infadd*: $x \neq \text{Infty} \implies x + y \neq \text{Infty}$
 <proof>

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

lemma *sum-list-in-set*: $(l::('x \times ('e,'a::\text{linorder}) \text{ Prio}) \text{ list}) \neq [] \implies$
 $\text{sum-list } (\text{map snd } l) \in \text{set } (\text{map snd } l)$
 <proof>

lemma *p-unwrap-less-sum*: $\text{snd } (p\text{-unwrap } ((\text{Prio } e \ aa) + b)) \leq aa$
 <proof>

lemma *prio-add-alb*: $\neg b \leq (a::('e,'a::\text{linorder}) \text{ Prio}) \implies b + a = a$
 <proof>

lemma *prio-add-alb2*: $(a::('e,'a::\text{linorder}) \text{ Prio}) \leq a + b \implies a + b = a$
 <proof>

lemma *prio-add-abc*:
 assumes $(l::('e,'a::\text{linorder}) \text{ Prio}) + a \leq c$
 and $\neg l \leq c$
 shows $\neg l \leq a$
 <proof>

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

Empty lemma *alprio-empty-correct*:
 assumes *al-empty* α *invar empty*
 shows *prio-empty* $(\text{alprio-}\alpha \ \alpha)$ $(\text{alprio-invar } \alpha \ \text{invar})$ $(\text{alprio-empty empty})$
 <proof>

Is Empty lemma *alprio-isEmpty-correct*:
 assumes *al-isEmpty* α *invar isEmpty*
 shows *prio-isEmpty* $(\text{alprio-}\alpha \ \alpha)$ $(\text{alprio-invar } \alpha \ \text{invar})$ $(\text{alprio-isEmpty isEmpty})$
 <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*: $\forall x \in \text{set } xs. \text{snd } x \neq \text{Infty} \implies$

$\forall y \in \text{set-mset } (\text{mset } (\text{map } \text{p-unwrap } (\text{map } \text{snd } xs))).$
 $\text{snd } (\text{p-unwrap } (\text{sum-list } (\text{map } \text{snd } xs))) \leq \text{snd } y$
 ⟨*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*:

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

lemma *alpriodel-invar*:

assumes *alprio-invar* α *invar s*
and *al-annot* α *invar annot*
and *alprio- α* α *s* \neq $\{\#\}$
and *al-splits* α *invar splits*
and *al-app* α *invar app*
shows *alprio-invar* α *invar* (*alprio-delete splits annot app s*)
 ⟨*proof*⟩

lemma *sum-list-elim*:
assumes $ins = l @ (a::('e,'a::linorder)Prio) \# r$
and $\neg \text{sum-list } l \leq \text{sum-list } ins$
and $\text{sum-list } l + a \leq \text{sum-list } ins$
shows $a = \text{sum-list } ins$
 $\langle \text{proof} \rangle$

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 $s - \{\#p\text{-unwrap } (\text{annot } s)\#\}$
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

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 $ops :: (\text{unit}, ('e, 'a::linorder) Prio, 's) \text{alist-ops}$

begin
definition [*icf-rec-def*]: *alprio-ops* \equiv (
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*
 \rangle

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 priorities¹. 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* \Rightarrow (*'e* \times *'a*) **where**
p-unwrap (*LP e a*) = (*e*, *a*)

fun *p-min* :: (*'e*::*linorder*, *'a*::*linorder*) *LP* \Rightarrow (*'e*, *'a*) *LP* \Rightarrow (*'e*, *'a*) *LP* **where**
p-min Infty Infty = *Infty*

¹Technically, 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  $\Rightarrow$  ('e::linorder, 'a::linorder) LP  $\Rightarrow$  bool where
  e-less-eq e Infty = False|
  e-less-eq e (LP e' -) = (e  $\leq$  e')

```

Instantiation of classes

```

lemma p-min-re-neut[simp]: p-min a Infty = a <proof>

```

```

lemma p-min-le-neut[simp]: p-min Infty a = a <proof>

```

```

lemma p-min-asso: p-min (p-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  $\Rightarrow$  ('e, 'a) LP  $\Rightarrow$  bool where
  p-less-eq (LP e a) (LP f b) = (a  $\leq$  b)|
  p-less-eq - Infty = True|
  p-less-eq Infty (LP e a) = False

```

```

fun p-less :: ('e, 'a::linorder) LP  $\Rightarrow$  ('e, 'a) LP  $\Rightarrow$  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  $\longleftrightarrow$  p-less-eq x y  $\wedge$   $\neg$  (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  $\Longrightarrow$  x = Infty
  <proof>

```

```

lemma p-order-trans :  $\llbracket$ p-less-eq x y; p-less-eq y z $\rrbracket \Longrightarrow$  p-less-eq x z
  <proof>

```

```

lemma p-linear2 : p-less-eq x y  $\vee$  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

$\langle proof \rangle$

end

Operations definition $aluprio-\alpha :: ('s \Rightarrow (unit \times ('e::linorder, 'a::linorder) LP) list)$

$\Rightarrow 's \Rightarrow ('e::linorder \rightarrow 'a::linorder)$

where

$aluprio-\alpha \alpha ft == (map-of (map p-unwrap (map snd (\alpha ft))))$

definition $aluprio-invar :: ('s \Rightarrow (unit \times ('c::linorder, 'd::linorder) LP) list)$

$\Rightarrow ('s \Rightarrow bool) \Rightarrow 's \Rightarrow bool$

where

$aluprio-invar \alpha invar ft ==$

$invar ft$

$\wedge (\forall x \in set (\alpha ft). snd x \neq Infty)$

$\wedge sorted (map fst (map p-unwrap (map snd (\alpha ft))))$

$\wedge distinct (map fst (map p-unwrap (map snd (\alpha ft))))$

definition $aluprio-empty$ **where**

$aluprio-empty empt = empt$

definition $aluprio-isEmpty$ **where**

$aluprio-isEmpty isEmpty = isEmpty$

definition $aluprio-insert ::$

$((('e::linorder, 'a::linorder) LP \Rightarrow bool)$

$\Rightarrow ('e, 'a) LP \Rightarrow 's \Rightarrow ('s \times (unit \times ('e, 'a) LP) \times 's))$

$\Rightarrow ('s \Rightarrow ('e, 'a) LP)$

$\Rightarrow ('s \Rightarrow bool)$

$\Rightarrow ('s \Rightarrow 's \Rightarrow 's)$

$\Rightarrow ('s \Rightarrow unit \Rightarrow ('e, 'a) LP \Rightarrow 's)$

$\Rightarrow 's \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$

where

$aluprio-insert splits annot isEmpty app consr s e a =$

$(if e-less-eq e (annot s) \wedge \neg isEmpty s$

$then$

$(let (l, (-,lp), r) = splits (e-less-eq e) Infty s in$

$(if e < fst (p-unwrap lp)$

$then$

$app (consr (consr l () (LP e a)) () lp) r$

$else$

```

      app (consr l () (LP e a)) r ))
else
  consr s () (LP e a)

```

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

where

```

aluprio-pop splits annot app s =
  (let (l, (-,lp) , r) = splits ( $\lambda x. x \leq (annot s)$ ) Infty s
  in
    (case lp of
      (LP e a)  $\Rightarrow$ 
        (e, a, app l r) ))

```

definition *aluprio-prio* ::

```

 $((('e::linorder, 'a::linorder) LP \Rightarrow bool) \Rightarrow ('e, 'a) LP \Rightarrow 's \Rightarrow ('s \times (unit \times ('e, 'a) LP) \times 's)) \Rightarrow ('s \Rightarrow ('e, 'a) LP) \Rightarrow ('s \Rightarrow bool) \Rightarrow 's \Rightarrow 'e \Rightarrow 'a option$ 

```

where

```

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

```

lemmas *aluprio-defs* =

```

aluprio-invar-def
aluprio- $\alpha$ -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*: $e\text{-less-eq } e \ x \implies e\text{-less-eq } e \ (x + y)$
 ⟨proof⟩

lemma *e-less-eq-mon2*: $e\text{-less-eq } e \ y \implies e\text{-less-eq } e \ (x + y)$
 ⟨proof⟩

lemmas *e-less-eq-mon* =
e-less-eq-mon1
e-less-eq-mon2

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

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

lemma *infadd*: $x \neq \text{Infty} \implies x + y \neq \text{Infty}$
 ⟨proof⟩

lemma *e-less-eq-sum-list*:
 $\llbracket \neg e\text{-less-eq } e \ (\text{sum-list } xs) \rrbracket \implies \forall x \in \text{set } xs. \neg e\text{-less-eq } e \ x$
 ⟨proof⟩

lemma *e-less-eq-p-unwrap*:
 $\llbracket x \neq \text{Infty}; \neg e\text{-less-eq } e \ x \rrbracket \implies \text{fst } (p\text{-unwrap } x) < e$
 ⟨proof⟩

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

lemma *e-less-eq-sum-list2*:
assumes
 $\forall x \in \text{set } (\alpha s). \text{snd } x \neq \text{Infty}$
 $((), b) \in \text{set } (\alpha s)$
shows $e\text{-less-eq } (\text{fst } (p\text{-unwrap } b)) \ (\text{sum-list } (\text{map } \text{snd } (\alpha s)))$
 ⟨proof⟩

lemma *e-less-eq-lem1*:
 $\llbracket \neg e\text{-less-eq } e \ a; e\text{-less-eq } e \ (a + b) \rrbracket \implies e\text{-less-eq } e \ b$
 ⟨proof⟩

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

lemma *sum-list-less-elems*: $\forall x \in \text{set } xs. \text{snd } x \neq \text{Infty} \implies$
 $\forall y \in \text{set } (map\ \text{snd } (map\ p\text{-unwrap } (map\ \text{snd } xs)))$
 $\text{snd } (p\text{-unwrap } (sum\text{-list } (map\ \text{snd } xs))) \leq y$
 <proof>

lemma *distinct-sortet-list-app*:
 $\llbracket \text{sorted } xs; \text{distinct } xs; xs = as\ @\ b\ \# \ cs \rrbracket$
 $\implies \forall x \in \text{set } cs. b < x$
 <proof>

lemma *distinct-sorted-list-lem1*:

assumes
sorted xs
sorted ys
distinct xs
distinct ys
 $\forall x \in \text{set } xs. x < e$
 $\forall y \in \text{set } ys. e < y$
shows
sorted (xs @ e # ys)
distinct (xs @ e # ys)
 <proof>

lemma *distinct-sorted-list-lem2*:

assumes
sorted xs
sorted ys
distinct xs
distinct ys
 $e < e'$
 $\forall x \in \text{set } xs. x < e$
 $\forall y \in \text{set } ys. e' < y$
shows
sorted (xs @ e # e' # ys)
distinct (xs @ e # e' # ys)
 <proof>

lemma *map-of-distinct-upd*:

$x \notin \text{set } (map\ \text{fst } xs) \implies [x \mapsto y] ++ map\text{-of } xs = (map\text{-of } xs) (x \mapsto y)$
 <proof>

lemma *map-of-distinct-upd2*:

assumes $x \notin \text{set } (map\ \text{fst } xs)$
 $x \notin \text{set } (map\ \text{fst } ys)$
shows $map\text{-of } (xs\ @\ (x, y)\ \# \ ys) = (map\text{-of } (xs\ @\ ys))(x \mapsto y)$

<proof>

lemma *map-of-distinct-upd3:*

assumes $x \notin \text{set}(\text{map fst } xs)$

$x \notin \text{set}(\text{map fst } ys)$

shows $\text{map-of } (xs @ (x,y) \# ys) = (\text{map-of } (xs @ (x,y') \# ys))(x \mapsto y)$

<proof>

lemma *map-of-distinct-upd4:*

assumes $x \notin \text{set}(\text{map fst } xs)$

$x \notin \text{set}(\text{map fst } ys)$

shows $\text{map-of } (xs @ ys) = (\text{map-of } (xs @ (x,y) \# ys))(x := \text{None})$

<proof>

lemma *map-of-distinct-lookup:*

assumes $x \notin \text{set}(\text{map fst } xs)$

$x \notin \text{set}(\text{map fst } ys)$

shows $\text{map-of } (xs @ (x,y) \# ys) x = \text{Some } y$

<proof>

lemma *ran-distinct:*

assumes *dist: distinct* ($\text{map fst } al$)

shows $\text{ran } (\text{map-of } al) = \text{snd } \text{'set } al$

<proof>

Finite lemma *aluprio-finite-correct:* $\text{uprio-finite } (\text{aluprio-}\alpha \ \alpha) (\text{aluprio-invar } \alpha \ \text{invar})$

<proof>

Empty lemma *aluprio-empty-correct:*

assumes *al-empty* $\alpha \ \text{invar } \text{empty}$

shows $\text{uprio-empty } (\text{aluprio-}\alpha \ \alpha) (\text{aluprio-invar } \alpha \ \text{invar}) (\text{aluprio-empty } \text{empty})$

<proof>

Is Empty lemma *aluprio-isEmpty-correct:*

assumes *al-isEmpty* $\alpha \ \text{invar } \text{isEmpty}$

shows $\text{uprio-isEmpty } (\text{aluprio-}\alpha \ \alpha) (\text{aluprio-invar } \alpha \ \text{invar}) (\text{aluprio-isEmpty } \text{isEmpty})$

<proof>

Insert lemma *annot-inf:*

assumes $A: \text{invar } s \quad \forall x \in \text{set } (\alpha \ s). \ \text{snd } x \neq \text{Infy} \quad \text{al-annot } \alpha \ \text{invar } \text{annot}$

shows $\text{annot } s = \text{Infy} \longleftrightarrow \alpha \ s = []$

<proof>

lemma *e-less-eq-annot:*

assumes *al-annot* $\alpha \ \text{invar } \text{annot}$

$\text{invar } s \quad \forall x \in \text{set } (\alpha \ s). \ \text{snd } x \neq \text{Infy} \quad \neg \text{e-less-eq } e \ (\text{annot } s)$

shows $\forall x \in \text{set } (\text{map } (\text{fst} \circ (\text{p-unwrap} \circ \text{snd})) (\alpha s)). x < e$
 <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* \equiv ()

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
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 = (λk 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* \equiv (
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* ≡ λ*m*. *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 = (λ*k m*. *Map.map-of m k*),
bmap-op-update = *AList.update*,
bmap-op-update-dj = (λ*k 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.dftt-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 := (λ*l*. *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* :: $('k \times 'v)$ *list* $\Rightarrow ('k, 'v)$ *lmi* **where**
lmi-from-list-dj $\equiv id$

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

Code generator test

definition *test-codegen* \equiv (
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  $\equiv$  (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

$(\text{'k}, \text{'v}) \text{ abs-hashmap} = \text{hashcode} \rightarrow (\text{'k} \rightarrow \text{'v})$

— Map entry of map by function

abbreviation map-entry **where** $\text{map-entry } k \ f \ m == m(k := f (m \ k))$

— Invariant: Buckets only contain entries with the right hashcode and there are no empty buckets

definition $\text{ahm-invar} :: (\text{'k}::\text{hashable}, \text{'v}) \text{ abs-hashmap} \Rightarrow \text{bool}$

where $\text{ahm-invar } m ==$

$(\forall hc \ cm \ k. \ m \ hc = \text{Some } cm \wedge k \in \text{dom } cm \longrightarrow \text{hashcode } k = hc) \wedge$

$(\forall hc \ cm. \ m \ hc = \text{Some } cm \longrightarrow cm \neq \text{Map.empty})$

— Abstract a hashmap to the corresponding map

definition $\text{ahm-}\alpha$ **where**

$\text{ahm-}\alpha \ m \ k == \text{case } m \ (\text{hashcode } k) \ \text{of}$

$\text{None} \Rightarrow \text{None} \mid$

$\text{Some } cm \Rightarrow cm \ k$

— Lookup an entry

definition $\text{ahm-lookup} :: \text{'k}::\text{hashable} \Rightarrow (\text{'k}, \text{'v}) \text{ abs-hashmap} \Rightarrow \text{'v} \ \text{option}$

where $\text{ahm-lookup } k \ m == (\text{ahm-}\alpha \ m) \ k$

— The empty hashmap

definition $\text{ahm-empty} :: (\text{'k}::\text{hashable}, \text{'v}) \text{ abs-hashmap}$

where $\text{ahm-empty} = \text{Map.empty}$

— Update/insert an entry

definition ahm-update **where**

$\text{ahm-update } k \ v \ m ==$

$\text{case } m \ (\text{hashcode } k) \ \text{of}$

$\text{None} \Rightarrow m \ (\text{hashcode } k \mapsto [k \mapsto v]) \mid$

$\text{Some } cm \Rightarrow m \ (\text{hashcode } k \mapsto cm \ (k \mapsto v))$

— Delete an entry

definition ahm-delete **where**

$\text{ahm-delete } k \ m == \text{map-entry } (\text{hashcode } k)$

```

( $\lambda v$ . case v of
  None  $\Rightarrow$  None |
  Some bm  $\Rightarrow$  (
    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*: \llbracket

!!hc cm k. \llbracket m hc = Some cm; $k \in \text{dom cm}$ $\rrbracket \Longrightarrow \text{hashcode } k = \text{hc}$;

!!hc cm. \llbracket m hc = Some cm $\rrbracket \Longrightarrow \text{cm} \neq \text{Map.empty}$

$\rrbracket \Longrightarrow \text{ahm-invar } m$

<proof>

lemma *ahm-invarD*: \llbracket *ahm-invar* m; m hc = Some cm; $k \in \text{dom cm}$ $\rrbracket \Longrightarrow \text{hashcode } k = \text{hc}$

<proof>

lemma *ahm-invarDne*: \llbracket *ahm-invar* m; m hc = Some cm $\rrbracket \Longrightarrow \text{cm} \neq \text{Map.empty}$

<proof>

lemma *ahm-invar-bucket-not-empty[simp]*:

ahm-invar m \Longrightarrow m hc \neq 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 \mapsto v)

ahm-invar m \Longrightarrow *ahm-invar* (*ahm-update* k v m)

<proof>

lemma *fun-upd-apply-ne*: $x \neq y \Longrightarrow (f(x:=v)) y = f y$

<proof>

lemma *cancel-one-empty-simp*: $m \mid' (-\{k\}) = \text{Map.empty} \longleftrightarrow \text{dom } m \subseteq \{k\}$
 ⟨proof⟩

lemma *ahm-delete-correct*:
 $\text{ahm-}\alpha (\text{ahm-delete } k \ m) = (\text{ahm-}\alpha \ m) \mid' (-\{k\})$
 $\text{ahm-invar } m \implies \text{ahm-invar } (\text{ahm-delete } k \ m)$
 ⟨proof⟩

lemma *ahm-isEmpty-correct*: $\text{ahm-invar } m \implies \text{ahm-isEmpty } m \longleftrightarrow \text{ahm-}\alpha \ m = \text{Map.empty}$
 ⟨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*: $\text{ahm-invar } m$
assumes *A*: $m \ \text{hc} = \text{Some } bm \quad bm \ k = \text{Some } v$
shows $\text{ahm-}\alpha \ m \ k = \text{Some } v$
 ⟨proof⟩

lemma *ahm-e-is-be*: $\llbracket \text{ahm-}\alpha \ m \ k = \text{Some } v; \text{!!}bm. \llbracket m \ (\text{hashcode } k) = \text{Some } bm; bm \ k = \text{Some } v \rrbracket \implies P \rrbracket \implies P$
 ⟨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

$(k, v) \ \text{hm-impl} = (\text{hashcode}, (k, v) \ \text{lm}) \ \text{rm}$

Operations — Auxiliary function: Apply function to value of an entry

definition *rm-map-entry*

$:: \text{hashcode} \Rightarrow ('v \ \text{option} \Rightarrow 'v \ \text{option}) \Rightarrow (\text{hashcode}, 'v) \ \text{rm} \Rightarrow (\text{hashcode}, 'v) \ \text{rm}$

where

$\text{rm-map-entry } k \ f \ m ==$
 $\text{case } \text{rm.lookup } k \ m \ \text{of}$
 $\text{None} \Rightarrow (\text{case } f \ \text{None} \ \text{of}$
 $\text{None} \Rightarrow m \mid$
 $\text{Some } v \Rightarrow \text{rm.update } k \ v \ m$

```

) |
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\text{-}iterator\text{-}image\ snd\ (\$
 $set\text{-}iterator\text{-}product\ (rm.iteratei\ m)\ (\lambda hc\ lm.\ lm.iteratei\ (snd\ hc\ lm)))$
 $\langle proof \rangle$

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.

— Abstract concrete hashmap to abstract hashmap

definition $hm\text{-}\alpha'$ **where** $hm\text{-}\alpha'\ m == \lambda hc.\ case\ rm.\alpha\ m\ hc\ of$

$None \Rightarrow None\ |$

$Some\ lm \Rightarrow Some\ (lm.\alpha\ lm)$

— Invariant for concrete hashmap: The hashcode-map and bucket-maps satisfy their invariants and the invariant of the corresponding abstract hashmap is satisfied.

definition $invar\ m == ahm\text{-}invar\ (hm\text{-}\alpha'\ m)$

lemma $rm\text{-}map\text{-}entry\text{-}correct$:

$rm.\alpha\ (rm\text{-}map\text{-}entry\ k\ f\ m) = (rm.\alpha\ m)(k := f\ (rm.\alpha\ m\ k))$

$\langle proof \rangle$

lemma $empty\text{-}correct'$:

$hm\text{-}\alpha'\ (empty\ ()) = ahm\text{-}empty$

$invar\ (empty\ ())$

$\langle proof \rangle$

lemma $lookup\text{-}correct'$:

$invar\ m \implies lookup\ k\ m = ahm\text{-}lookup\ k\ (hm\text{-}\alpha'\ m)$

$\langle proof \rangle$

lemma $update\text{-}correct'$:

$invar\ m \implies hm\text{-}\alpha'\ (update\ k\ v\ m) = ahm\text{-}update\ k\ v\ (hm\text{-}\alpha'\ m)$

$invar\ m \implies invar\ (update\ k\ v\ m)$

$\langle proof \rangle$

lemma $delete\text{-}correct'$:

$invar\ m \implies hm\text{-}\alpha'\ (delete\ k\ m) = ahm\text{-}delete\ k\ (hm\text{-}\alpha'\ m)$

$invar\ m \implies invar\ (delete\ k\ m)$

$\langle proof \rangle$

lemma $isEmpty\text{-}correct'$:

$invar\ hm \implies isEmpty\ hm \longleftrightarrow ahm\text{-}\alpha\ (hm\text{-}\alpha'\ hm) = Map.empty$

$\langle proof \rangle$

```

lemma iteratei-correct':
  assumes invar: invar hm
  shows map-iterator (iteratei hm) (ahm- $\alpha$  (hm- $\alpha'$  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  $\Rightarrow$  ('k :: hashable, 'v) hashmap
where hm-empty = ( $\lambda$ -. hm-empty-const)

```

```

definition hm-lookup k hm == HashMap-Impl.lookup k (impl-of-RBT-HM hm)

```

```

definition hm-update :: ('k :: hashable)  $\Rightarrow$  'v  $\Rightarrow$  ('k, 'v) hashmap  $\Rightarrow$  ('k, 'v)
  hashmap
where hm-update k v hm = RBT-HM (HashMap-Impl.update k v (impl-of-RBT-HM
  hm))

```


definition $hm\text{-update-dj} :: ('k :: hashable) \Rightarrow 'v \Rightarrow ('k, 'v) \text{hashmap} \Rightarrow ('k, 'v) \text{hashmap}$

where $hm\text{-update-dj} = hm\text{-update}$

definition $hm\text{-delete} :: ('k :: hashable) \Rightarrow ('k, 'v) \text{hashmap} \Rightarrow ('k, 'v) \text{hashmap}$

where $hm\text{-delete} k hm = RBT\text{-HM} (HashMap\text{-Impl.delete } k (impl\text{-of-RBT-HM } hm))$

definition $hm\text{-isEmpty} :: ('k :: hashable, 'v) \text{hashmap} \Rightarrow bool$

where $hm\text{-isEmpty} hm = HashMap\text{-Impl.isEmpty} (impl\text{-of-RBT-HM } hm)$

definition $hm\text{-iteratei} hm == HashMap\text{-Impl.iteratei} (impl\text{-of-RBT-HM } hm)$

lemma $impl\text{-of-hm-empty}$ [*simp, code abstract*]:

$impl\text{-of-RBT-HM} (hm\text{-empty-const}) = HashMap\text{-Impl.empty} ()$
 ⟨*proof*⟩

lemma $impl\text{-of-hm-update}$ [*simp, code abstract*]:

$impl\text{-of-RBT-HM} (hm\text{-update } k v hm) = HashMap\text{-Impl.update } k v (impl\text{-of-RBT-HM } hm)$
 ⟨*proof*⟩

lemma $impl\text{-of-hm-delete}$ [*simp, code abstract*]:

$impl\text{-of-RBT-HM} (hm\text{-delete } k hm) = HashMap\text{-Impl.delete } k (impl\text{-of-RBT-HM } hm)$
 ⟨*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 $('k, 'v) hm = ('k, 'v) \text{hashmap}$

— Abstract concrete hashmap to map

definition $hm\text{-}\alpha == ahm\text{-}\alpha \circ hm\text{-}\alpha' \circ impl\text{-of-RBT-HM}$

abbreviation (*input*) $hm\text{-invar} :: ('k :: hashable, 'v) \text{hashmap} \Rightarrow bool$

where $hm\text{-invar} == \lambda\text{-}. True$

lemma $hm\text{-aux-correct}$:

$hm\text{-}\alpha (hm\text{-empty} ()) = Map.empty$

```

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* \equiv (
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

Interruptible iterator

fun *iteratei-postfixed* :: *'key list* \Rightarrow (*'key*, *'val*) *trie* \Rightarrow
('key list \times *'val*, *' σ*) *set-iterator*

where

iteratei-postfixed *ks* (*Trie vo ts*) *c f* σ =
(if *c* σ
 then *foldli* *ts* *c* ($\lambda(k, t)$ σ . *iteratei-postfixed* (*k* # *ks*) *t* *c f* σ)
 (case *vo* of *None* \Rightarrow σ | *Some* *v* \Rightarrow *f* (*ks*, *v*) σ)
 else σ)

definition *iteratei* :: ('key, 'val) trie \Rightarrow ('key list \times 'val, 'σ) set-iterator
where *iteratei* t c f σ = *iteratei-postfixed* [] t c f σ

lemma *iteratei-postfixed-interrupt*:
 \neg c σ \Longrightarrow *iteratei-postfixed* ks t c f σ = σ
 <proof>

lemma *iteratei-interrupt*:
 \neg c σ \Longrightarrow *iteratei* t c f σ = σ
 <proof>

lemma *iteratei-postfixed-alt-def* :
iteratei-postfixed ks ((Trie vo ts)::('key, 'val) trie) =
 (set-iterator-union
 (case-option set-iterator-emp (λv. set-iterator-sng (ks, v)) vo)
 (set-iterator-image snd
 (set-iterator-product (foldli ts
 (λ(k, t'). *iteratei-postfixed* (k # ks) t'))
))
 <proof>

lemma *iteratei-postfixed-correct* :
assumes *invar*: *invar-trie* (t :: ('key, 'val) trie)
shows set-iterator ((*iteratei-postfixed* ks0 t)::('key list \times 'val, 'σ) set-iterator)
 ((λksv. (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 \times '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* \Rightarrow *'val* \Rightarrow (*'key, 'val*) *trie* \Rightarrow (*'key, 'val*) *trie*)
where *update ks v t* = *Trie (update-trie ks v (impl-of t))*

definition *delete* :: (*'key list* \Rightarrow (*'key, 'val*) *trie* \Rightarrow (*'key, 'val*) *trie*)
where *delete ks t* = *Trie (delete-trie ks (impl-of t))*

definition *lookup* :: (*'key, 'val*) *trie* \Rightarrow *'key list* \Rightarrow *'val option*
where *lookup t* = *lookup-trie (impl-of t)*

definition *isEmpty* :: (*'key, 'val*) *trie* \Rightarrow *bool*
where *isEmpty t* = *is-empty-trie (impl-of t)*

definition *iteratei* :: (*'key, 'val*) *trie* \Rightarrow (*'key list* \times *'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 $(\text{'k}, \text{'v}) \text{tm} = (\text{'k}, \text{'v}) \text{trie}$

definition $[\text{icf-rec-def}]$: $\text{tm-basic-ops} \equiv (\langle$
 $\text{bmap-op-}\alpha = \text{Trie2.lookup},$
 $\text{bmap-op-invar} = \lambda-. \text{True},$
 $\text{bmap-op-empty} = (\lambda::\text{unit}. \text{Trie2.empty}),$
 $\text{bmap-op-lookup} = (\lambda k m. \text{Trie2.lookup } m \ k),$
 $\text{bmap-op-update} = \text{Trie2.update},$
 $\text{bmap-op-update-dj} = \text{Trie2.update},$
 $\text{bmap-op-delete} = \text{Trie2.delete},$
 $\text{bmap-op-list-it} = \text{Trie2.iteratei}$
 \rangle

$\langle \text{ML} \rangle$

interpretation tm-basic : $\text{StdBasicMap } \text{tm-basic-ops}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

definition $[\text{icf-rec-def}]$: $\text{tm-ops} \equiv \text{tm-basic.dflt-ops}$

$\langle \text{ML} \rangle$

interpretation tm : $\text{StdMap } \text{tm-ops}$

$\langle \text{proof} \rangle$

interpretation tm : $\text{StdMap-no-invar } \text{tm-ops}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma $\text{pi-trie-impl}[\text{proper-it}]$:

shows $\text{proper-it}'$

$((\text{Trie-Impl.iteratei}) :: - \Rightarrow (-, \sigma a) \text{ set-iterator})$

$((\text{Trie-Impl.iteratei}) :: - \Rightarrow (-, \sigma b) \text{ set-iterator})$

$\langle \text{proof} \rangle$

lemma $\text{pi-trie}[\text{proper-it}]$:

$\text{proper-it}' \text{Trie2.iteratei } \text{Trie2.iteratei}$

$\langle \text{proof} \rangle$

interpretation pi-trie : $\text{proper-it-loc } \text{Trie2.iteratei } \text{Trie2.iteratei}$

$\langle \text{proof} \rangle$

Code generator test

definition $\text{test-codegen} \equiv ($

$\text{tm.add} ,$

$\text{tm.add-dj} ,$

$\text{tm.ball} ,$

$\text{tm.bex} ,$

$\text{tm.delete} ,$

```

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  $\lambda$ -. 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  $\times$  'val) list array nat
```


Operations

definition *new-hashmap-with* :: $\text{nat} \Rightarrow ('key :: \text{hashable}, 'val) \text{hashmap}$
where $\wedge \text{size}. \text{new-hashmap-with } \text{size} = \text{HashMap } (\text{new-array } [] \text{ size}) 0$

definition *ahm-empty* :: $\text{unit} \Rightarrow ('key :: \text{hashable}, 'val) \text{hashmap}$
where *ahm-empty* $\equiv \lambda\text{-}. \text{new-hashmap-with } (\text{def-hashmap-size } \text{TYPE}('key))$

definition *bucket-ok* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (('key :: \text{hashable}) \times 'val) \text{list} \Rightarrow \text{bool}$
where *bucket-ok* $\text{len } h \text{ kvs} = (\forall k \in \text{fst } ' \text{ set } \text{kvs}. \text{bounded-hashcode-nat } \text{len } k = h)$

definition *ahm-invar-aux* :: $\text{nat} \Rightarrow (('key :: \text{hashable}) \times 'val) \text{list array} \Rightarrow \text{bool}$
where

ahm-invar-aux $n \ a \ \longleftrightarrow$
 $(\forall h. h < \text{array-length } a \longrightarrow \text{bucket-ok } (\text{array-length } a) \ h \ (\text{array-get } a \ h) \wedge$
distinct $(\text{map } \text{fst } (\text{array-get } a \ h))) \wedge$
 $\text{array-foldl } (\lambda\text{- } n \ \text{kvs}. n + \text{size } \text{kvs}) \ 0 \ a = n \wedge$
 $\text{array-length } a > 1$

primrec *ahm-invar* :: $('key :: \text{hashable}, 'val) \text{hashmap} \Rightarrow \text{bool}$
where *ahm-invar* $(\text{HashMap } a \ n) = \text{ahm-invar-aux } n \ a$

definition *ahm- α -aux* :: $(('key :: \text{hashable}) \times 'val) \text{list array} \Rightarrow 'key \Rightarrow 'val \text{option}$
where [*simp*]: *ahm- α -aux* $a \ k = \text{map-of } (\text{array-get } a \ (\text{bounded-hashcode-nat } (\text{array-length } a) \ k)) \ k$

primrec *ahm- α* :: $('key :: \text{hashable}, 'val) \text{hashmap} \Rightarrow 'key \Rightarrow 'val \text{option}$
where
ahm- α $(\text{HashMap } a \ -) = \text{ahm-}\alpha\text{-aux } a$

definition *ahm-lookup* :: $'key \Rightarrow ('key :: \text{hashable}, 'val) \text{hashmap} \Rightarrow 'val \text{option}$
where *ahm-lookup* $k \ \text{hm} = \text{ahm-}\alpha \ \text{hm } k$

primrec *ahm-iteratei-aux* :: $((('key :: \text{hashable}) \times 'val) \text{list array}) \Rightarrow ('key \times 'val, ' \sigma) \text{set-iterator}$
where *ahm-iteratei-aux* $(\text{Array } \text{xs}) \ c \ f = \text{foldli } (\text{concat } \text{xs}) \ c \ f$

primrec *ahm-iteratei* :: $(('key :: \text{hashable}, 'val) \text{hashmap}) \Rightarrow (('key \times 'val), ' \sigma) \text{set-iterator}$
where
ahm-iteratei $(\text{HashMap } a \ n) = \text{ahm-iteratei-aux } a$

definition *ahm-rehash-aux'* :: $\text{nat} \Rightarrow 'key \times 'val \Rightarrow (('key :: \text{hashable}) \times 'val) \text{list array} \Rightarrow ('key \times 'val) \text{list array}$
where

ahm-rehash-aux' $n \ kv \ a =$
 $(\text{let } h = \text{bounded-hashcode-nat } n \ (\text{fst } kv)$
 $\text{in } \text{array-set } a \ h \ (kv \ \# \ \text{array-get } a \ h))$

definition $ahm\text{-rehash-}aux :: ('key :: hashable) \times 'val \text{ list array} \Rightarrow nat \Rightarrow ('key \times 'val) \text{ list array}$

where

$ahm\text{-rehash-}aux\ a\ sz = ahm\text{-iteratei-}aux\ a\ (\lambda x. True)\ (ahm\text{-rehash-}aux'\ sz)\ (new\text{-array}\ []\ sz)$

primrec $ahm\text{-rehash} :: ('key :: hashable, 'val) \text{ hashmap} \Rightarrow nat \Rightarrow ('key, 'val) \text{ hashmap}$

where $ahm\text{-rehash}\ (HashMap\ a\ n)\ sz = HashMap\ (ahm\text{-rehash-}aux\ a\ sz)\ n$

primrec $hm\text{-grow} :: ('key :: hashable, 'val) \text{ hashmap} \Rightarrow nat$

where $hm\text{-grow}\ (HashMap\ a\ n) = 2 * \text{array-length}\ a + 3$

primrec $ahm\text{-filled} :: ('key :: hashable, 'val) \text{ hashmap} \Rightarrow bool$

where $ahm\text{-filled}\ (HashMap\ a\ n) = (\text{array-length}\ a * \text{load-factor} \leq n * 100)$

primrec $ahm\text{-update-}aux :: ('key :: hashable, 'val) \text{ hashmap} \Rightarrow 'key \Rightarrow 'val \Rightarrow ('key, 'val) \text{ hashmap}$

where

$ahm\text{-update-}aux\ (HashMap\ a\ n)\ k\ v =$
 $(let\ h = \text{bounded-hashcode-nat}\ (\text{array-length}\ a)\ k;$
 $m = \text{array-get}\ a\ h;$
 $insert = \text{map-of}\ m\ k = None$
 $in\ HashMap\ (\text{array-set}\ a\ h\ (AList.update\ k\ v\ m))\ (\text{if}\ insert\ \text{then}\ n + 1\ \text{else}\ n))$

definition $ahm\text{-update} :: 'key \Rightarrow 'val \Rightarrow ('key :: hashable, 'val) \text{ hashmap} \Rightarrow ('key, 'val) \text{ hashmap}$

where

$ahm\text{-update}\ k\ v\ hm =$
 $(let\ hm' = ahm\text{-update-}aux\ hm\ k\ v$
 $in\ (\text{if}\ ahm\text{-filled}\ hm'\ \text{then}\ ahm\text{-rehash}\ hm'\ (hm\text{-grow}\ hm')\ \text{else}\ hm'))$

primrec $ahm\text{-delete} :: 'key \Rightarrow ('key :: hashable, 'val) \text{ hashmap} \Rightarrow ('key, 'val) \text{ hashmap}$

where

$ahm\text{-delete}\ k\ (HashMap\ a\ n) =$
 $(let\ h = \text{bounded-hashcode-nat}\ (\text{array-length}\ a)\ k;$
 $m = \text{array-get}\ a\ h;$
 $deleted = (\text{map-of}\ m\ k \neq None)$
 $in\ HashMap\ (\text{array-set}\ a\ h\ (AList.delete\ k\ m))\ (\text{if}\ deleted\ \text{then}\ n - 1\ \text{else}\ n))$

lemma $hm\text{-grow-gt-1}$ [iff]:

$Suc\ 0 < hm\text{-grow}\ hm$

<proof>

lemma $bucket\text{-ok-}Nil$ [simp]: $bucket\text{-ok}\ len\ h\ [] = True$

<proof>

lemma *bucket-okD*:

[[*bucket-ok* *len* *h* *xs*; $(k, v) \in \text{set } xs$]]
 \implies *bounded-hashcode-nat* *len* $k = h$

<proof>

lemma *bucket-okI*:

$(\bigwedge k. k \in \text{fst } \text{'set } kvs \implies \text{bounded-hashcode-nat } \text{len } k = h) \implies \text{bucket-ok } \text{len } h$
kvs

<proof>

ahm-invar

lemma *ahm-invar-auxE*:

assumes *ahm-invar-aux* *n* *a*

obtains $\forall h. h < \text{array-length } a \longrightarrow \text{bucket-ok } (\text{array-length } a) h (\text{array-get } a h)$
 $\wedge \text{distinct } (\text{map } \text{fst } (\text{array-get } a h))$

and $n = \text{array-foldl } (\lambda - n kvs. n + \text{length } kvs) 0 a$ **and** $\text{array-length } a > 1$

<proof>

lemma *ahm-invar-auxI*:

[[$\bigwedge h. h < \text{array-length } a \implies \text{bucket-ok } (\text{array-length } a) h (\text{array-get } a h);$

$\bigwedge h. h < \text{array-length } a \implies \text{distinct } (\text{map } \text{fst } (\text{array-get } a h));$

$n = \text{array-foldl } (\lambda - n kvs. n + \text{length } kvs) 0 a; \text{array-length } a > 1$]]

\implies *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* \implies *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 \implies ahm-invar (new-hashmap-with n)
<proof>

lemma *ahm- α -new-hashmap-with*:
n > 1 \implies 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 $\sigma = a\text{-idx-it.idx-iteratei } a \ c \ (\lambda x. \text{foldli } x \ c \ f) \ \sigma$
 ⟨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: $k \notin \text{fst 'set (array-get a (bounded-hashcode-nat (array-length a) k))}$
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* :: $(('key :: \text{hashable}) \times 'val) \text{ list array}$
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* :: $(('key :: \text{hashable}, 'val) \text{ hashmap}$
assumes *inv*: *ahm-invar hm*
and $sz > 1$
shows *ahm-invar (ahm-rehash hm sz)* $\text{ahm-}\alpha \text{ (ahm-rehash hm sz) = ahm-}\alpha$
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 \mapsto 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 \mapsto 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 \equiv (HashMap (ArrayHashMap-Impl.ahm-empty ()))*

definition $ahm\text{-empty} :: unit \Rightarrow ('key :: hashable, 'val) \text{hashmap}$
where $ahm\text{-empty} \equiv \lambda\cdot. ahm\text{-empty}\text{-const}$

definition $ahm\text{-}\alpha :: ('key :: hashable, 'val) \text{hashmap} \Rightarrow 'key \Rightarrow 'val \text{option}$
where $ahm\text{-}\alpha \text{ hm} = \text{ArrayHashMap-Impl.ahm-}\alpha \text{ (impl-of hm)}$

definition $ahm\text{-lookup} :: 'key \Rightarrow ('key :: hashable, 'val) \text{hashmap} \Rightarrow 'val \text{option}$
where $ahm\text{-lookup} \text{ k hm} = \text{ArrayHashMap-Impl.ahm-lookup} \text{ k (impl-of hm)}$

definition $ahm\text{-iteratei} :: ('key :: hashable, 'val) \text{hashmap} \Rightarrow ('key \times 'val, 'a) \text{set-iterator}$
where $ahm\text{-iteratei} \text{ hm} = \text{ArrayHashMap-Impl.ahm-iteratei} \text{ (impl-of hm)}$

definition $ahm\text{-update} :: 'key \Rightarrow 'val \Rightarrow ('key :: hashable, 'val) \text{hashmap} \Rightarrow ('key, 'val) \text{hashmap}$
where
 $ahm\text{-update} \text{ k v hm} = \text{HashMap (ArrayHashMap-Impl.ahm-update} \text{ k v (impl-of hm))}$

definition $ahm\text{-delete} :: 'key \Rightarrow ('key :: hashable, 'val) \text{hashmap} \Rightarrow ('key, 'val) \text{hashmap}$
where
 $ahm\text{-delete} \text{ k hm} = \text{HashMap (ArrayHashMap-Impl.ahm-delete} \text{ k (impl-of hm))}$

lemma $impl\text{-of-ahm-empty}$ [code abstract]:
 $impl\text{-of} \text{ ahm-empty-const} = \text{ArrayHashMap-Impl.ahm-empty} \text{ ()}$
 $\langle proof \rangle$

lemma $impl\text{-of-ahm-update}$ [code abstract]:
 $impl\text{-of} \text{ (ahm-update} \text{ k v hm)} = \text{ArrayHashMap-Impl.ahm-update} \text{ k v (impl-of hm)}$
 $\langle proof \rangle$

lemma $impl\text{-of-ahm-delete}$ [code abstract]:
 $impl\text{-of} \text{ (ahm-delete} \text{ k hm)} = \text{ArrayHashMap-Impl.ahm-delete} \text{ k (impl-of hm)}$
 $\langle proof \rangle$

lemma $finite\text{-dom-ahm-}\alpha$ [simp]: $finite \text{ (dom (ahm-}\alpha \text{ hm))}$
 $\langle proof \rangle$

lemma $ahm\text{-empty-correct}$ [simp]: $ahm\text{-}\alpha \text{ (ahm-empty} \text{ ())} = \text{Map.empty}$
 $\langle proof \rangle$

lemma $ahm\text{-lookup-correct}$ [simp]: $ahm\text{-lookup} \text{ k m} = ahm\text{-}\alpha \text{ m k}$
 $\langle proof \rangle$

lemma $ahm\text{-update-correct}$ [simp]: $ahm\text{-}\alpha \text{ (ahm-update} \text{ k v hm)} = (ahm\text{-}\alpha \text{ hm})(k \mapsto v)$
 $\langle proof \rangle$

lemma *ahm-delete-correct*[simp]:
 $ahm-\alpha (ahm-delete\ k\ hm) = (ahm-\alpha\ hm) \uparrow (-\ \{k\})$
 ⟨proof⟩

lemma *ahm-iteratei-impl*[simp]: *map-iterator* (*ahm-iteratei* *m*) (*ahm-α* *m*)
 ⟨proof⟩

ICF Integration

definition [*icf-rec-def*]: *ahm-basic-ops* \equiv (
 $bmap-op-\alpha = ahm-\alpha,$
 $bmap-op-invar = \lambda-. True,$
 $bmap-op-empty = ahm-empty,$
 $bmap-op-lookup = ahm-lookup,$
 $bmap-op-update = ahm-update,$
 $bmap-op-update-dj = ahm-update,$ — 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* \equiv *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* \equiv (
 $ahm.add$,
 $ahm.add-dj$,
 $ahm.ball$,
 $ahm.bex$,


```

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 \Rightarrow nat \rightarrow 'v* **where**
iam- α a i \equiv if $i < \text{array-length } a$ then $\text{array-get } a \ i$ else None

lemma [*code*]: *iam- α a i \equiv array-get-oo None a i*
 <*proof*>

abbreviation (*input*) *iam-invar* :: *'v iam \Rightarrow bool*
where *iam-invar \equiv λ -. True*

definition *iam-empty* :: *unit \Rightarrow 'v iam*
where *iam-empty \equiv λ ::unit. array-of-list []*

definition *iam-lookup* :: *nat \Rightarrow 'v iam \rightarrow 'v*
where [*code-unfold*]: *iam-lookup k a \equiv iam- α a k*

definition *iam-increment (l::nat) idx \equiv*
*max (idx + 1 - l) (2 * l + 3)*

lemma *incr-correct*: $\neg \text{idx} < l \implies \text{idx} < l + \text{iam-increment } l \text{ idx}$
 <proof>

definition *iam-update* :: $\text{nat} \Rightarrow 'v \Rightarrow 'v \text{ iam} \Rightarrow 'v \text{ iam}$
where *iam-update* $k \ v \ a \equiv \text{let}$
 $l = \text{array-length } a;$
 $a = \text{if } k < l \text{ then } a \text{ else } \text{array-grow } a \ (\text{iam-increment } l \ k) \ \text{None}$
in
 $\text{array-set } a \ k \ (\text{Some } v)$

lemma [*code*]: *iam-update* $k \ v \ a \equiv \text{array-set-oo}$
 $(\lambda-. \text{array-set}$
 $(\text{array-grow } a \ (\text{iam-increment } (\text{array-length } a) \ k) \ \text{None}) \ k \ (\text{Some } v))$
 $a \ k \ (\text{Some } v)$
 <proof>

definition *iam-update-dj* $\equiv \text{iam-update}$

definition *iam-delete* :: $\text{nat} \Rightarrow 'v \text{ iam} \Rightarrow 'v \text{ iam}$
where *iam-delete* $k \ a \equiv$
 $\text{if } k < \text{array-length } a \text{ then } \text{array-set } a \ k \ \text{None} \text{ else } a$

lemma [*code*]: *iam-delete* $k \ a \equiv \text{array-set-oo } (\lambda-. a) \ a \ k \ \text{None}$
 <proof>

fun *iam-rev-iterateoi-aux*
 :: $\text{nat} \Rightarrow ('v \text{ iam}) \Rightarrow ('\sigma \Rightarrow \text{bool}) \Rightarrow (\text{nat} \times 'v \Rightarrow 'v \Rightarrow '\sigma) \Rightarrow 'v \Rightarrow '\sigma$
where
 $\text{iam-rev-iterateoi-aux } 0 \ a \ c \ f \ \sigma = \sigma$
 $|\ \text{iam-rev-iterateoi-aux } i \ a \ c \ f \ \sigma =$
 $\text{if } c \ \sigma \text{ then}$
 $\text{iam-rev-iterateoi-aux } (i - 1) \ a \ c \ f \ ($
 $\text{case } \text{array-get } a \ (i - 1) \ \text{of } \text{None} \Rightarrow \sigma \ | \ \text{Some } x \Rightarrow f \ (i - 1, x) \ \sigma$
 $)$
 $\text{else } \sigma)$

definition *iam-rev-iterateoi* :: $'v \text{ iam} \Rightarrow (\text{nat} \times 'v, '\sigma) \text{ set-iterator}$ **where**
 $\text{iam-rev-iterateoi } a \equiv \text{iam-rev-iterateoi-aux } (\text{array-length } a) \ a$

function *iam-iterateoi-aux*
 :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('v \text{ iam}) \Rightarrow ('\sigma \Rightarrow \text{bool}) \Rightarrow (\text{nat} \times 'v \Rightarrow 'v \Rightarrow '\sigma) \Rightarrow 'v \Rightarrow '\sigma$
where
 $\text{iam-iterateoi-aux } i \ \text{len} \ a \ c \ f \ \sigma =$
 $(\text{if } i \geq \text{len} \vee \neg c \ \sigma \text{ then } \sigma \text{ else let}$
 $\sigma' = (\text{case } \text{array-get } a \ i \ \text{of}$
 $\text{None} \Rightarrow \sigma$

| *Some* $x \Rightarrow f (i, x) \sigma$
 in *iam-iterateoi-aux* $(i + 1) \text{ len } a \text{ c } f \sigma'$
 <proof>

termination

<proof>

declare *iam-iterateoi-aux.simps*[*simp del*]

lemma *iam-iterateoi-aux-csimps*:

$i \geq \text{len} \Rightarrow \text{iam-iterateoi-aux } i \text{ len } a \text{ c } f \sigma = \sigma$
 $\neg c \sigma \Rightarrow \text{iam-iterateoi-aux } i \text{ len } a \text{ c } f \sigma = \sigma$
 $\llbracket i < \text{len}; c \sigma \rrbracket \Rightarrow \text{iam-iterateoi-aux } i \text{ len } a \text{ c } f \sigma =$
 (case *array-get* $a \ i$ of
 None $\Rightarrow \text{iam-iterateoi-aux } (i + 1) \text{ len } a \text{ c } f \sigma$
 | *Some* $x \Rightarrow \text{iam-iterateoi-aux } (i + 1) \text{ len } a \text{ c } f (f (i, x) \sigma)$)
 <proof>

definition *iam-iterateoi* :: '*v iam* $\Rightarrow (\text{nat} \times 'v, ' \sigma)$ *set-iterator* **where**
iam-iterateoi $a = \text{iam-iterateoi-aux } 0 (\text{array-length } a) a$

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*: $i < \text{array-length } a \Rightarrow$
array-get (*array-set* $a \ i \ x$) $j = (\text{if } i=j \text{ then } x \text{ else } \text{array-get } a \ j)$
 <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* $(\lambda n. \text{map-option } (\lambda v. (n, v)) (\text{array-get } a \ n))$) (*rev*
 $[0..<n]$)
 <proof>

lemma *iam-rev-iterateoi-foldli-conv* :
iam-rev-iterateoi $a =$
foldli (*List.map-filter*
 $(\lambda n. \text{map-option } (\lambda v. (n, v)) (\text{array-get } a \ n))$)

(*rev* [0..*(array-length a)*]))
 <proof>

lemma *iam-rev-iterateoi-correct* :
fixes *m::'a option array*
defines *kvs* \equiv *List.map-filter*
 ($\lambda n.$ *map-option* ($\lambda v.$ (*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-iterateoi-impl*:
poly-map-iterateoi 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 ($\lambda n.$ *map-option* ($\lambda v.$ (*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
 ($\lambda n.$ *map-option* ($\lambda v.$ (*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* \equiv *List.map-filter*
 ($\lambda n.$ *map-option* ($\lambda v.$ (*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.dfft-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 $ls-\alpha :: 'a\ ls \Rightarrow 'a\ set$ **where** $ls-\alpha\ l == set\ (list-of-dlist\ l)$

definition $ls-basic-ops :: ('a, 'a\ ls)\ set-basic-ops$ **where**

```

[icf-rec-def]: ls-basic-ops  $\equiv$  ()
  bset-op- $\alpha$  = ls- $\alpha$ ,
  bset-op-invar =  $\lambda$ -. True,
  bset-op-empty =  $\lambda$ -. Dlist.empty,
  bset-op-memb = ( $\lambda x\ s.$  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
)

```

$\langle ML \rangle$

interpretation $ls-basic: StdBasicSet\ ls-basic-ops$

$\langle proof \rangle$

$\langle ML \rangle$

definition [icf-rec-def]: $ls-ops \equiv ls-basic.dflt-ops()$

$set-op-to-list := list-of-dlist$

)

$\langle ML \rangle$

interpretation $ls: StdSetDefs\ ls-ops$ $\langle proof \rangle$

interpretation $ls: StdSet\ ls-ops$

$\langle proof \rangle$

interpretation $ls: StdSet-no-invar\ ls-ops$

$\langle proof \rangle$

$\langle ML \rangle$

lemma $pi-ls[proper-it]:$

$proper-it'\ dlist-iteratei\ dlist-iteratei$

$\langle proof \rangle$

lemma $pi-ls'[proper-it]:$

$proper-it'\ ls.iteratei\ ls.iteratei$

$\langle proof \rangle$

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* \equiv (

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 \Rightarrow 'a lsi \Rightarrow '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 = (λx s. List.member s x),
  bset-op-ins = lsi-ins,
  bset-op-ins-dj = (#),
  bset-op-delete = λx 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- $\alpha$  :: 'a lsnd  $\Rightarrow$  'a set where lsnd- $\alpha$  == set
```

```
abbreviation (input) lsnd-invar
```

```
  :: 'a lsnd  $\Rightarrow$  bool where lsnd-invar == ( $\lambda$ -. True)
```

```
definition lsnd-empty :: unit  $\Rightarrow$  'a lsnd where lsnd-empty == ( $\lambda$ -.unit. [])
```

```
definition lsnd-memb :: 'a  $\Rightarrow$  'a lsnd  $\Rightarrow$  bool where lsnd-memb x l == List.member l x
```

```
definition lsnd-ins :: 'a  $\Rightarrow$  'a lsnd  $\Rightarrow$  'a lsnd where lsnd-ins x l == x#l
```

```
definition lsnd-ins-dj :: 'a  $\Rightarrow$  'a lsnd  $\Rightarrow$  'a lsnd where lsnd-ins-dj x l == x#l
```

```
definition lsnd-delete :: 'a  $\Rightarrow$  'a lsnd  $\Rightarrow$  'a lsnd where lsnd-delete x l == remove-rev x l
```

```
definition lsnd-iteratei :: 'a lsnd  $\Rightarrow$  ('a,' $\sigma$ ) set-iterator
```

```
where lsnd-iteratei l = foldli (remdups l)
```

```
definition lsnd-isEmpty :: 'a lsnd  $\Rightarrow$  bool where lsnd-isEmpty s == s=[]
```

```
definition lsnd-union :: 'a lsnd  $\Rightarrow$  'a lsnd  $\Rightarrow$  'a lsnd
```

```
  where lsnd-union s1 s2 == revg s1 s2
```

```
definition lsnd-union-dj :: 'a lsnd  $\Rightarrow$  'a lsnd  $\Rightarrow$  'a lsnd
```

```
  where lsnd-union-dj s1 s2 == revg s1 s2 — Union of disjoint sets
```

definition *lsnd-to-list* :: 'a *lsnd* \Rightarrow 'a *list* **where** *lsnd-to-list* == *remdups*

definition *list-to-lsnd* :: 'a *list* \Rightarrow '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

$\langle proof \rangle$

lemma *lsnd-union-dj-impl*: *set-union-dj lsnd- α lsnd-invar lsnd- α lsnd-invar lsnd- α lsnd-invar lsnd-union-dj*
 $\langle proof \rangle$

lemma *lsnd-to-list-impl*: *set-to-list lsnd- α lsnd-invar lsnd-to-list*
 $\langle proof \rangle$

lemma *list-to-lsnd-impl*: *list-to-set lsnd- α lsnd-invar list-to-lsnd*
 $\langle proof \rangle$

definition *lsnd-basic-ops* :: (*'x, 'x lsnd*) *set-basic-ops*

where [*icf-rec-def*]: *lsnd-basic-ops* \equiv (
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*
 \rangle

$\langle ML \rangle$

interpretation *lsnd-basic*: *StdBasicSet lsnd-basic-ops*
 $\langle proof \rangle$
 $\langle ML \rangle$

definition [*icf-rec-def*]: *lsnd-ops* \equiv *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*
 \rangle

$\langle ML \rangle$

interpretation *lsnd*: *StdSetDefs lsnd-ops* $\langle proof \rangle$

interpretation *lsnd*: *StdSet lsnd-ops*

$\langle proof \rangle$

interpretation *lsnd*: *StdSet-no-invar lsnd-ops*

$\langle proof \rangle$

$\langle ML \rangle$

lemma *pi-lsnd*[*proper-it*]:

proper-it' lsnd-iteratei lsnd-iteratei

$\langle proof \rangle$

interpretation *pi-lsnd*: *proper-it-loc lsnd-iteratei lsnd-iteratei*
<proof>

definition *test-codegen* **where** *test-codegen* \equiv (
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 \Rightarrow 'a set **where** *lss- α* == set

definition *lss-invar* :: 'a::{\i>linorder} lss \Rightarrow bool **where** *lss-invar* *l* == distinct *l* \wedge sorted *l*

definition *lss-empty* :: *unit* \Rightarrow *'a lss* **where** *lss-empty* == (λ -.:unit. [])

definition *lss-memb* :: *'a::{\linorder}* \Rightarrow *'a lss* \Rightarrow *bool* **where** *lss-memb* *x l* == *Sorted-List-Operations.memb-sorted* *l x*

definition *lss-ins* :: *'a::{\linorder}* \Rightarrow *'a lss* \Rightarrow *'a lss* **where** *lss-ins* *x l* == *insertion-sort* *x l*

definition *lss-ins-dj* :: *'a::{\linorder}* \Rightarrow *'a lss* \Rightarrow *'a lss* **where** *lss-ins-dj* == *lss-ins*

definition *lss-delete* :: *'a::{\linorder}* \Rightarrow *'a lss* \Rightarrow *'a lss* **where** *lss-delete* *x l* == *delete-sorted* *x l*

definition *lss-iterateoi* :: *'a lss* \Rightarrow (*'a, 'σ*) *set-iterator*
where *lss-iterateoi* *l* = *foldli* *l*

definition *lss-reverse-iterateoi* :: *'a lss* \Rightarrow (*'a, 'σ*) *set-iterator*
where *lss-reverse-iterateoi* *l* = *foldli* (*rev* *l*)

definition *lss-iteratei* :: *'a lss* \Rightarrow (*'a, 'σ*) *set-iterator*
where *lss-iteratei* *l* = *foldli* *l*

definition *lss-isEmpty* :: *'a lss* \Rightarrow *bool* **where** *lss-isEmpty* *s* == *s* == []

definition *lss-union* :: *'a::{\linorder}* *lss* \Rightarrow *'a lss* \Rightarrow *'a lss*
where *lss-union* *s1 s2* == *Misc.merge* *s1 s2*

definition *lss-union-list* :: *'a::{\linorder}* *lss list* \Rightarrow *'a lss*
where *lss-union-list* *l* == *merge-list* [] *l*

definition *lss-inter* :: *'a::{\linorder}* *lss* \Rightarrow *'a lss* \Rightarrow *'a lss*
where *lss-inter* == *inter-sorted*

definition *lss-union-dj* :: *'a::{\linorder}* *lss* \Rightarrow *'a lss* \Rightarrow *'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* == *iftt-image* *lss-image-filter*

definition *lss-inj-image* == *iftt-inj-image* *lss-inj-image-filter*

definition *lss-to-list* :: *'a lss* \Rightarrow *'a list* **where** *lss-to-list* == *id*

definition *list-to-lss* :: *'a::{\linorder}* *list* \Rightarrow *'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* \equiv ()

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-ops* (

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 RBSetImpl
imports
  ../spec/SetSpec
  RBMapImpl
  ../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-ops

```

```

⟨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* \equiv (

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* \equiv (
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 \equiv *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* \equiv (
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  $\equiv$  ias-sbm.obasic.dflt-ops
```

```
<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  $\equiv$  (
```



```

    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* == $\lambda::unit. ([], [])$

— 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
 $(f1[\text{length } (fst F) - Suc (i - l2) := a], f2)$

Correctness

lemma *fifo-empty-impl: list-empty fifo- α fifo-invar fifo-empty*
 $\langle proof \rangle$

lemma *fifo-isEmpty-impl: list-isEmpty fifo- α fifo-invar fifo-isEmpty*
 $\langle proof \rangle$

lemma *fifo-size-impl: list-size fifo- α fifo-invar fifo-size*
 $\langle proof \rangle$

lemma *fifo-appendr-impl: list-appendr fifo- α fifo-invar fifo-appendr*
 $\langle proof \rangle$

lemma *fifo-appendl-impl: list-appendl fifo- α fifo-invar fifo-appendl*
 $\langle proof \rangle$

lemma *fifo-removel-impl: list-removel fifo- α fifo-invar fifo-removel*
 $\langle proof \rangle$

lemma *fifo-remover-impl: list-remover fifo- α fifo-invar fifo-remover*
 $\langle proof \rangle$

lemma *fifo-leftmost-impl: list-leftmost fifo- α fifo-invar fifo-leftmost*
 $\langle proof \rangle$

lemma *fifo-rightmost-impl: list-rightmost fifo- α fifo-invar fifo-rightmost*
 $\langle proof \rangle$

lemma *fifo-get-impl: list-get fifo- α fifo-invar fifo-get*
 $\langle proof \rangle$

lemma *fifo-set-impl: list-set fifo- α fifo-invar fifo-set*
 $\langle proof \rangle$

definition [*icf-rec-def*]: $fifo-ops \equiv ()$
 $list-op-\alpha = fifo-\alpha,$
 $list-op-invar = fifo-invar,$
 $list-op-empty = fifo-empty,$
 $list-op-isEmpty = fifo-isEmpty,$
 $list-op-size = fifo-size,$
 $list-op-appendl = fifo-appendl,$
 $list-op-removel = fifo-removel,$
 $list-op-leftmost = fifo-leftmost,$
 $list-op-appendr = fifo-appendr,$
 $list-op-remover = fifo-remover,$

```

list-op-rightmost = fifo-rightmost,
list-op-get = fifo-get,
list-op-set = fifo-set
)

```

$\langle ML \rangle$

interpretation *fifo*: *StdList fifo-ops*

$\langle proof \rangle$

interpretation *fifo*: *StdList-no-invar fifo-ops*

$\langle proof \rangle$

$\langle ML \rangle$

definition *test-codegen* **where** *test-codegen* \equiv

```

(
  fifo.empty,
  fifo.isEmpty,
  fifo.size,
  fifo.appendl,
  fifo.remove,
  fifo.leftmost,
  fifo.appendr,
  fifo.remove,
  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) *binom* = ('a,'b) *BinomialHeap*

Definitions

definition *binom- α* **where** *binom- α* $q \equiv$ *BinomialHeap.to-mset* q

definition *binoinsert* **where** *binoinsert* \equiv *BinomialHeap.insert*
abbreviation (*input*) *binoinvar* $:: ('a,'b)$ *BinomialHeap* \Rightarrow *bool*
where *binoinvar* $\equiv \lambda-. True$
definition *binofind* **where** *binofind* \equiv *BinomialHeap.findMin*
definition *binodelete* **where** *binodelete* \equiv *BinomialHeap.deleteMin*
definition *binomeld* **where** *binomeld* \equiv *BinomialHeap.meld*
definition *binomempty* **where** *binomempty* $\equiv \lambda::unit. BinomialHeap.empty$
definition *binoisEmpty* **where** *binoisEmpty* $= BinomialHeap.isEmpty$

definition [*icf-rec-def*]: *binops* $==$ (
prio-op- α $=$ *bino- α* ,
prio-op-invar $=$ *binoinvar*,
prio-op-empty $=$ *binomempty*,
prio-op-isEmpty $=$ *binoisEmpty*,
prio-op-insert $=$ *binoinsert*,
prio-op-find $=$ *binofind*,
prio-op-delete $=$ *binodelete*,
prio-op-meld $=$ *binomeld*
>)

lemmas *binodefs* $=$
bino- α -def
binoinsert-def
binofind-def
binodelete-def
binomeld-def
binomempty-def
binoisEmpty-def

Correctness

theorem *binomempty-impl*: *prio-empty bino- α binoinvar binomempty*
<proof>

theorem *binoisEmpty-impl*: *prio-isEmpty bino- α binoinvar binoisEmpty*
<proof>

theorem *binofind-impl*: *prio-find bino- α binoinvar binofind*
<proof>

lemma *binoinsert-impl*: *prio-insert bino- α binoinvar binoinsert*
<proof>

lemma *binomeld-impl*: *prio-meld bino- α binoinvar binomeld*
<proof>

lemma *binodelete-impl*:
prio-delete bino- α binoinvar binofind binodelete
<proof>

```

⟨ML⟩
interpretation binop: StdPrio binop-ops
  ⟨proof⟩
interpretation binop: StdPrio-no-invar binop-ops
  ⟨proof⟩
⟨ML⟩

```

```

definition test-codegen where test-codegen = (
  binop.empty,
  binop.isEmpty,
  binop.find,
  binop.insert,
  binop.meld,
  binop.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- $\alpha$ -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 \equiv (*
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- $\alpha$   $\equiv$  FingerTree.toList
```

abbreviation (*input*) *ft-invar* :: ('a,'b) FingerTree \Rightarrow bool **where**

```
ft-invar  $\equiv$   $\lambda$ -. True
```

definition *ft-empty* **where**

```
ft-empty  $\equiv$   $\lambda$ -.unit. FingerTree.empty
```

definition *ft-isEmpty* **where**

```
ft-isEmpty  $\equiv$  FingerTree.isEmpty
```

definition *ft-count* **where**

```
ft-count  $\equiv$  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  $\equiv$  FingerTree.head
```

definition *ft-tail* **where**

```
ft-tail  $\equiv$  FingerTree.tail
```

definition *ft-headR* **where**

```
ft-headR  $\equiv$  FingerTree.headR
```

definition *ft-tailR* **where**

```
ft-tailR  $\equiv$  FingerTree.tailR
```

definition *ft-foldl* **where**

```
ft-foldl  $\equiv$  FingerTree.foldl
```

definition *ft-foldr* **where**

```
ft-foldr  $\equiv$  FingerTree.foldr
```

definition *ft-app* **where**

ft-app \equiv *FingerTree.app*

definition *ft-annot* **where**

ft-annot \equiv *FingerTree.annot*

definition *ft-splits* **where**

ft-splits \equiv *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  $\equiv$  (
  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) alprio = (unit, ('a, 'p) Prio) FingerTree

<ML>
interpretation alprio-ga: alprio ft-ops <proof>
<ML>

definition [icf-rec-def]: alprio-ops  $\equiv$  alprio-ga.alprio-ops

<ML>
interpretation alprio: StdPrio alprio-ops
  <proof>
<ML>

definition test-codegen where test-codegen  $\equiv$  (
  alprio.empty,
  alprio.isEmpty,
  alprio.insert,
  alprio.find,

```

```

    alprio.delete,
    alprio.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) alprio = (unit, ('a, 'b) LP) FingerTree

```

```

<ML>

```

```

interpretation alprio-ga: alprio ft-ops <proof>

```

```

<ML>

```

```

definition [icf-rec-def]: alprio-ops ≡ alprio-ga.alprio-ops

```

```

<ML>

```

```

interpretation alprio: StdUprio alprio-ops

```

```

  <proof>

```

```

<ML>

```

```

definition test-codegen where test-codegen ≡ (

```

```

  alprio.empty,

```

```

  alprio.isEmpty,

```

```

  alprio.insert,

```

```

  alprio.pop,

```

```

  alprio.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** $\alpha :: 's \Rightarrow 'x \text{ set}$ **and** *invar* +
fixes *iteratei* :: $'s \Rightarrow ('x, 's) \text{ set-iterator}$

```

assumes iteratei-rule: invar  $S \Longrightarrow \text{set-iterator } (\text{iteratei } S) (\alpha S)$ 

```

```

begin

```

```

lemma iteratei-rule-P:

```

```

  [
    invar  $S$ ;
     $I (\alpha S) \sigma 0$ ;
     $!!x \text{ it } \sigma. [c \ \sigma; x \in \text{it}; \text{it} \subseteq \alpha S; I \ \text{it} \ \sigma] \Longrightarrow I (\text{it} - \{x\}) (f \ x \ \sigma)$ ;
     $!!\sigma. I \ \{\} \ \sigma \Longrightarrow P \ \sigma$ ;
     $!!\sigma \ \text{it}. [ \text{it} \subseteq \alpha S; \text{it} \neq \{\}; \neg c \ \sigma; I \ \text{it} \ \sigma ] \Longrightarrow P \ \sigma$ 
  ]  $\Longrightarrow P (\text{iteratei } S \ c \ f \ \sigma 0)$ 
  <proof>

```

```

lemma iteratei-rule-insert-P:

```

```

  [
    invar  $S$ ;
     $I \ \{\} \ \sigma 0$ ;
     $!!x \ \text{it} \ \sigma. [c \ \sigma; x \in \alpha S - \text{it}; \text{it} \subseteq \alpha S; I \ \text{it} \ \sigma] \Longrightarrow I (\text{insert } x \ \text{it}) (f \ x \ \sigma)$ ;
     $!!\sigma. I (\alpha S) \ \sigma \Longrightarrow P \ \sigma$ ;
  ]

```

$$\begin{aligned} & !!\sigma \text{ it. } \llbracket \text{it} \subseteq \alpha S; \text{it} \neq \alpha S; \neg c \sigma; I \text{ it } \sigma \rrbracket \implies P \sigma \\ & \rrbracket \implies P (\text{iteratei } S \text{ c f } \sigma 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

Versions without break condition.

lemma *iterate-rule-P*:

$$\begin{aligned} & \llbracket \\ & \quad \text{invar } S; \\ & \quad I (\alpha S) \sigma 0; \\ & \quad !!x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq \alpha S; I \text{ it } \sigma \rrbracket \implies I (\text{it} - \{x\}) (f x \sigma); \\ & \quad !!\sigma. I \{\} \sigma \implies P \sigma \\ & \rrbracket \implies P (\text{iteratei } S (\lambda-. \text{True}) f \sigma 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *iterate-rule-insert-P*:

$$\begin{aligned} & \llbracket \\ & \quad \text{invar } S; \\ & \quad I \{\} \sigma 0; \\ & \quad !!x \text{ it } \sigma. \llbracket x \in \alpha S - \text{it}; \text{it} \subseteq \alpha S; I \text{ it } \sigma \rrbracket \implies I (\text{insert } x \text{ it}) (f x \sigma); \\ & \quad !!\sigma. I (\alpha S) \sigma \implies P \sigma \\ & \rrbracket \implies P (\text{iteratei } S (\lambda-. \text{True}) f \sigma 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

end

lemma *set-iteratei-I* :

assumes $\bigwedge s. \text{invar } s \implies \text{set-iterator } (\text{iti } s) (\alpha s)$

shows *set-iteratei* α *invar* *iti*

$\langle \text{proof} \rangle$

locale *set-iterateoi* = *ordered-finite-set* α *invar*

for $\alpha :: 's \Rightarrow ('u::\text{linorder}) \text{ set}$ **and** *invar*

+

fixes *iterateoi* :: $'s \Rightarrow ('u, ' \sigma) \text{ set-iterator}$

assumes *iterateoi-rule*:

$\text{invar } s \implies \text{set-iterator-linord } (\text{iterateoi } s) (\alpha s)$

begin

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

assumes *MINV*: *invar* *m*

assumes *I0*: $I (\alpha m) \sigma 0$

assumes *IP*: $!!k \text{ it } \sigma. \llbracket$

$c \sigma;$

$k \in \text{it};$

$\forall j \in \text{it}. k \leq j;$

$\forall j \in \alpha m - \text{it}. j \leq k;$

$\text{it} \subseteq \alpha m;$

$I \text{ it } \sigma$

$\rrbracket \implies I (\text{it} - \{k\}) (f k \sigma)$

assumes *IF*: $!!\sigma. I \{\} \sigma \implies P \sigma$

assumes *II*: $!!\sigma \text{ it. } \llbracket$

```

    it  $\subseteq$   $\alpha$  m;
    it  $\neq$  {};
     $\neg$  c  $\sigma$ ;
    I it  $\sigma$ ;
     $\forall k \in it. \forall j \in \alpha m - it. j \leq k$ 
  ]  $\implies$  P  $\sigma$ 
  shows P (iterateoi m c f  $\sigma$  0)
  <proof>

```

```

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

```

```

lemma set-iterateoi-I :
  assumes  $\bigwedge s. invar s \implies set-iterator-linord (itoi s) (\alpha s)$ 
  shows set-iterateoi  $\alpha$  invar itoi
  <proof>

```

```

locale set-reverse-iterateoi = ordered-finite-set  $\alpha$  invar
  for  $\alpha :: 's \Rightarrow ('u::linorder)$  set and invar
  +
  fixes reverse-iterateoi :: ' $s \Rightarrow ('u, 's)$  set-iterator
  assumes reverse-iterateoi-rule:
    invar m  $\implies set-iterator-rev-linord (reverse-iterateoi m) (\alpha m)$ 

```

```

begin
lemma reverse-iterateoi-rule-P[case-names minv inv0 inv-pres i-complete i-inter]:
  assumes MINV: invar m
  assumes I0: I (( $\alpha$  m))  $\sigma$  0
  assumes IP: !!k it  $\sigma$ . [
    c  $\sigma$ ;
    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  $\sigma$ 
  ]  $\implies$  I (it - {k}) (f k  $\sigma$ )
  assumes IF: !! $\sigma$ . I {}  $\sigma \implies$  P  $\sigma$ 
  assumes II: !! $\sigma$  it. [
    it  $\subseteq$  ( $\alpha$  m);
    it  $\neq$  {};
     $\neg$  c  $\sigma$ ;

```



```

  I it  $\sigma$ ;
   $\forall k \in it. \forall j \in (\alpha m) - it. j \geq k$ 
 $\implies P \sigma$ 
shows  $P$  (reverse-iterateoi m c f  $\sigma$ )
<proof>

```

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

```

assumes MINV: invar m
assumes I0:  $I ((\alpha m)) \sigma$ 
assumes IP:  $\forall k \in it. \sigma$ . [
  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  $\sigma$ 
 $\implies I (it - \{k\}) (f k \sigma)$ 
assumes IF:  $\forall \sigma. I \{ \} \sigma \implies P \sigma$ 
shows  $P$  (reverse-iterateoi m ( $\lambda-. True$ ) f  $\sigma$ )
<proof>
end

```

lemma set-reverse-iterateoi- I :

```

assumes  $\bigwedge s. invar s \implies set-iterator-rev-linord (itoi s) (\alpha s)$ 
shows set-reverse-iterateoi  $\alpha$  invar itoi
<proof>

```

lemma (in poly-set-iteratei) v1-iteratei-impl:

```

set-iteratei  $\alpha$  invar iteratei
<proof>

```

lemma (in poly-set-iterateoi) v1-iterateoi-impl:

```

set-iterateoi  $\alpha$  invar iterateoi
<proof>

```

lemma (in poly-set-rev-iterateoi) v1-reverse-iterateoi-impl:

```

set-reverse-iterateoi  $\alpha$  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** $\alpha :: 's \Rightarrow 'u \rightarrow 'v$ **and** invar
+
fixes iteratei :: $'s \Rightarrow ('u \times 'v, ' \sigma)$ set-iterator

```

assumes iteratei-rule: invar m  $\implies map-iterator (iteratei m) (\alpha m)$ 
begin

```

lemma *iteratei-rule-P*:

assumes *invar m*
and *I0*: $I (\text{dom } (\alpha m)) \sigma 0$
and *IP*: $!!k v \text{ it } \sigma. \llbracket c \sigma; k \in \text{it}; \alpha m k = \text{Some } v; \text{it} \subseteq \text{dom } (\alpha m); I \text{ it } \sigma \rrbracket$
 $\implies I (\text{it} - \{k\}) (f (k, v) \sigma)$
and *IF*: $!!\sigma. I \{\} \sigma \implies P \sigma$
and *II*: $!!\sigma \text{ it}. \llbracket \text{it} \subseteq \text{dom } (\alpha m); \text{it} \neq \{\}; \neg c \sigma; I \text{ it } \sigma \rrbracket \implies P \sigma$
shows $P (\text{iteratei } m \ c \ f \ \sigma 0)$
 $\langle \text{proof} \rangle$

lemma *iteratei-rule-insert-P*:

assumes
invar m
 $I \{\} \sigma 0$
 $!!k v \text{ it } \sigma. \llbracket c \sigma; k \in (\text{dom } (\alpha m) - \text{it}); \alpha m k = \text{Some } v; \text{it} \subseteq \text{dom } (\alpha m); I \text{ it } \sigma \rrbracket$
 $\implies I (\text{insert } k \ \text{it}) (f (k, v) \sigma)$
 $!!\sigma. I (\text{dom } (\alpha m)) \sigma \implies P \sigma$
 $!!\sigma \text{ it}. \llbracket \text{it} \subseteq \text{dom } (\alpha m); \text{it} \neq \text{dom } (\alpha m);$
 $\neg (c \sigma);$
 $I \text{ it } \sigma \rrbracket \implies P \sigma$
shows $P (\text{iteratei } m \ c \ f \ \sigma 0)$
 $\langle \text{proof} \rangle$

lemma *iterate-rule-P*:

\llbracket
invar m;
 $I (\text{dom } (\alpha m)) \sigma 0$;
 $!!k v \text{ it } \sigma. \llbracket k \in \text{it}; \alpha m k = \text{Some } v; \text{it} \subseteq \text{dom } (\alpha m); I \text{ it } \sigma \rrbracket$
 $\implies I (\text{it} - \{k\}) (f (k, v) \sigma)$;
 $!!\sigma. I \{\} \sigma \implies P \sigma$
 $\rrbracket \implies P (\text{iteratei } m \ (\lambda-. \text{True}) \ f \ \sigma 0)$
 $\langle \text{proof} \rangle$

lemma *iterate-rule-insert-P*:

\llbracket
invar m;
 $I \{\} \sigma 0$;
 $!!k v \text{ it } \sigma. \llbracket k \in (\text{dom } (\alpha m) - \text{it}); \alpha m k = \text{Some } v; \text{it} \subseteq \text{dom } (\alpha m); I \text{ it } \sigma \rrbracket$
 \rrbracket
 $\implies I (\text{insert } k \ \text{it}) (f (k, v) \sigma)$;
 $!!\sigma. I (\text{dom } (\alpha m)) \sigma \implies P \sigma$
 $\rrbracket \implies P (\text{iteratei } m \ (\lambda-. \text{True}) \ f \ \sigma 0)$
 $\langle \text{proof} \rangle$

end

lemma *map-iteratei-I* :

assumes $\bigwedge m. \text{invar } m \implies \text{map-iterator } (\text{iti } m) (\alpha m)$
shows $\text{map-iteratei } \alpha \ \text{invar } \text{iti}$

<proof>

```

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

```

```

lemma iterateo-rule-P[case-names minv inv0 inv-pres i-complete]:
  assumes MINV: invar  $m$ 
  assumes I0:  $I (\text{dom } (\alpha \ m)) \ \sigma \ 0$ 
  assumes IP:  $!!k \ v \ it \ \sigma. \llbracket k \in it; \forall j \in it. \ k \leq j; \forall j \in \text{dom } (\alpha \ m) - it. \ j \leq k; \alpha \ m$ 
 $k = \text{Some } v; it \subseteq \text{dom } (\alpha \ m); I \ it \ \sigma \rrbracket$ 
     $\Longrightarrow I (it - \{k\}) (f \ (k, \ v) \ \sigma)$ 
  assumes IF:  $!!\sigma. \ I \ \{\} \ \sigma \Longrightarrow P \ \sigma$ 
  shows  $P (\text{iterateoi } m \ (\lambda-. \ \text{True}) \ f \ \sigma \ 0)$ 
<proof>
end

```

```

lemma map-iterateoi-I :
assumes  $\wedge m. \ \text{invar } m \Longrightarrow \text{map-iterator-linord } (\text{itoi } m) (\alpha \ m)$ 
shows  $\text{map-iterateoi } \alpha \ \text{invar } \text{itoi}$ 
<proof>

```

```

locale map-reverse-iterateoi = ordered-finite-map  $\alpha$  invar
  for  $\alpha :: 's \Rightarrow ('u::linorder) \rightarrow 'v$  and invar
  +
  fixes reverse-iterateoi ::  $'s \Rightarrow ('u \times 'v, 's)$  set-iterator
  assumes reverse-iterateoi-rule:
    invar m  $\Longrightarrow$  map-iterator-rev-linord (reverse-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:  $!!k v it \sigma. \llbracket$ 
     $c \sigma;$ 
     $k \in it;$ 
     $\forall j \in it. k \geq j;$ 
     $\forall j \in \text{dom} (\alpha m) - it. j \geq k;$ 
     $\alpha m k = \text{Some } v;$ 
     $it \subseteq \text{dom} (\alpha m);$ 
     $I it \sigma$ 
   $\rrbracket \Longrightarrow I (it - \{k\}) (f (k, v) \sigma)$ 
  assumes IF:  $!!\sigma. I \{ \} \sigma \Longrightarrow P \sigma$ 
  assumes II:  $!!\sigma it. \llbracket$ 
     $it \subseteq \text{dom} (\alpha m);$ 
     $it \neq \{ \};$ 
     $\neg c \sigma;$ 
     $I it \sigma;$ 
     $\forall k \in it. \forall j \in \text{dom} (\alpha m) - it. j \geq k$ 
   $\rrbracket \Longrightarrow P \sigma$ 
  shows  $P$  (reverse-iterateoi m c f  $\sigma$ 0)
  <proof>

lemma reverse-iteateo-rule-P[case-names minv inv0 inv-pres i-complete]:
  assumes MINV: invar m
  assumes I0:  $I$  (dom ( $\alpha$  m))  $\sigma$ 0
  assumes IP:  $!!k v it \sigma. \llbracket$ 
     $k \in it;$ 
     $\forall j \in it. k \geq j;$ 
     $\forall j \in \text{dom} (\alpha m) - it. j \geq k;$ 
     $\alpha m k = \text{Some } v;$ 
     $it \subseteq \text{dom} (\alpha m);$ 
     $I it \sigma$ 
   $\rrbracket \Longrightarrow I (it - \{k\}) (f (k, v) \sigma)$ 
  assumes IF:  $!!\sigma. I \{ \} \sigma \Longrightarrow P \sigma$ 
  shows  $P$  (reverse-iterateoi m ( $\lambda-. \text{True}$ ) f  $\sigma$ 0)
  <proof>
end

lemma map-reverse-iterateoi-I :
assumes  $\bigwedge m. \text{invar } m \Longrightarrow \text{map-iterator-rev-linord} (\text{ritoi } m) (\alpha m)$ 
shows map-reverse-iterateoi  $\alpha$  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 *binocorrect* = *binocorrect*

lemmas *fifocorrect* = *fifocorrect*

lemmas *ftcorrect* = *ftcorrect*

lemmas *alprioicorrect* = *alprioicorrect*

lemmas *aluprioicorrect* = *aluprioicorrect*

lemmas *skewcorrect* = *skewcorrect*

locale *list-enqueue* = *list-appendr*
locale *list-dequeue* = *list-remove*

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** $R=\langle Id \rangle$ *dflt-rs-rel*]

lemmas *tyrel-dflt-bool-set* =
ty-REL[**where** *'a=bool set* **and** $R=\langle Id \rangle$ *list-set-rel*]

lemmas *tyrel-dflt-nat-map* =
ty-REL[**where** $R=\langle nat-rel, Rv \rangle$ *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** $R=\langle Id::('a::linorder \times 'a) set \rangle$ *dflt-rs-rel*]

lemmas *tyrel-dflt-linorder-map* =
ty-REL[**where** $R=\langle Id::('a::linorder \times 'a) set, R \rangle$ *dflt-rm-rel*] **for** *R*

```

lemmas tyrel-dflt-bool-map =
  ty-REL[where  $R = \langle Id :: (bool \times bool) \text{ set}, R \rangle list\text{-map-rel}$ ] 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: ( $?c :: ?'c, \{1, 2, 3 :: int\} \in ?R$ )
  <proof> lemmas asm-rl[of  $-\in \langle int-rel \rangle dflt-rs-rel, OF test-dflt-tyrel1$ ]

private schematic-goal test-dflt-tyrel2: ( $?c :: ?'c, \{True, False\} \in ?R$ )
  <proof> lemmas asm-rl[of  $-\in \langle bool-rel \rangle list-set-rel, OF test-dflt-tyrel2$ ]

private schematic-goal test-dflt-tyrel3: ( $?c :: ?'c, [1 :: int \rightarrow 0 :: nat] \in ?R$ )
  <proof> lemmas asm-rl[of  $-\in \langle int-rel, nat-rel \rangle dflt-rm-rel, OF test-dflt-tyrel3$ ]

private schematic-goal test-dflt-tyrel4: ( $?c :: ?'c, [False \rightarrow 0 :: nat] \in ?R$ )
  <proof> lemmas asm-rl[of  $-\in \langle bool-rel, nat-rel \rangle list-map-rel, OF test-dflt-tyrel4$ ]

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

$\langle ML \rangle$

Fallbacks

 $\langle ML \rangle$ **class** *id-refine***instance** *nat* :: *id-refine* $\langle proof \rangle$ **instance** *bool* :: *id-refine* $\langle proof \rangle$ **instance** *int* :: *id-refine* $\langle proof \rangle$ **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*= $\langle Id \rangle rs.rel$]*ty-REL*[**where** 'a=*int set* **and** *R*= $\langle Id \rangle rs.rel$]*ty-REL*[**where** 'a=*bool set* **and** *R*= $\langle Id \rangle lsi.rel$]**lemmas** [*autoref-tyrel*] =*ty-REL*[**where** 'a=(*nat* \rightarrow 'b), **where** *R*= $\langle nat-rel, Rv \rangle dflt-rm-rel$]*ty-REL*[**where** 'a=(*int* \rightarrow 'b), **where** *R*= $\langle int-rel, Rv \rangle dflt-rm-rel$]*ty-REL*[**where** 'a=(*bool* \rightarrow 'b), **where** *R*= $\langle bool-rel, Rv \rangle dflt-rm-rel$]**for** *Rv***lemmas** [*autoref-tyrel*] =*ty-REL*[**where** 'a=(*nat* \rightarrow 'b::*id-refine*), **where** *R*= $\langle nat-rel, Id \rangle rm.rel$]*ty-REL*[**where** 'a=(*int* \rightarrow 'b::*id-refine*), **where** *R*= $\langle int-rel, Id \rangle rm.rel$]*ty-REL*[**where** 'a=(*bool* \rightarrow 'b::*id-refine*), **where** *R*= $\langle bool-rel, Id \rangle rm.rel$]**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

 $\langle ML \rangle$ **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 monadic 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* \Rightarrow (*nat* \times *nat*) *nres* **where**
sum-max *V* \equiv *do* {
 (*s,m*) \leftarrow *WHILE* ($\lambda(V,s,m). V \neq \{\}$) ($\lambda(V,s,m). do$ {
 x \leftarrow *SPEC* ($\lambda x. x \in V$);
 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* σ_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* σ_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* σ_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* Φ 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* $V_0 \equiv \lambda(V, s::nat, m)$.

$$\begin{aligned} & V \subseteq V_0 \\ \wedge \quad & s = \sum (V_0 - V) \\ \wedge \quad & m = (\text{if } (V_0 - V) = \{\} \text{ then } 0 \text{ else } \text{Max } (V_0 - V)) \\ \wedge \quad & \text{finite } (V_0 - V) \end{aligned}$$

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* $V_0 (V - \{x\}, s + x, \text{max } m \ x)$

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* $V \leq SPEC (\lambda(s, m). s = \sum V \wedge m = \text{Max } V)$

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 explicitly. 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 $sum-max'-invar V_0 \sigma \equiv$
 $sum-max-invar V_0 \sigma$
 $\wedge (let (V, -, -) = \sigma \text{ in } finite (V_0 - V))$

definition $sum-max' :: nat \text{ set} \Rightarrow (nat \times nat) \text{ nres}$ **where**
 $sum-max' V \equiv do \{$
 $(-, s, m) \leftarrow WHILE_T^{sum-max'-invar V} (\lambda(V, s, m). V \neq \{\}) (\lambda(V, s, m). do \{$
 $x \leftarrow SPEC (\lambda x. x \in V);$
 $let V = V - \{x\};$
 $let s = s + x;$
 $let m = max m x;$
 $RETURN (V, s, m)$
 $\}) (V, 0, 0);$
 $RETURN (s, m)$
 $\}$

theorem $sum-max'-correct:$

assumes $NE: V \neq \{\}$ **and** $FIN: finite V$
shows $sum-max' V \leq SPEC (\lambda(s, m). s = \sum V \wedge m = Max V)$
<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 [?, ?].

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* :: $nat\ ls \Rightarrow (nat \times nat)\ nres$ **where**
 $sum-max-impl\ V \equiv do\ \{$
 $(-,s,m) \leftarrow WHILE\ (\lambda(V,s,m). \neg ls.isEmpty\ V)\ (\lambda(V,s,m). do\ \{$
 $x \leftarrow RETURN\ (the\ (ls.sel\ V\ (\lambda x. True)));$
 $let\ V = ls.delete\ x\ V;$
 $let\ s = s + x;$
 $let\ m = max\ m\ x;$
 $RETURN\ (V,s,m)$
 $\})\ (V,0,0);$
 $RETURN\ (s,m)$
 $\}$

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 (λx. True))*, i.e., selection of an element that satisfies the predicate $\lambda x. True$. 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 program actually refines the abstract one.

theorem *sum-max-impl-refine*:
assumes $(V, V') \in build-rel\ ls.\alpha\ ls.invar$
shows $sum-max-impl\ V \leq \Downarrow Id\ (sum-max\ V')$

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.

¹Also called coupling invariant.

theorem *sum-max-impl-correct*:

assumes $(V, V') \in \text{build-rel } ls.\alpha \text{ } ls.\text{invar}$ **and** $V' \neq \{\}$

shows $\text{sum-max-impl } V \leq \text{SPEC } (\lambda(s, m). s = \sum V' \wedge m = \text{Max } V')$

<proof>

Just for completeness, we also refine the total correct program in the same way.

definition *sum-max'-impl* :: $\text{nat } ls \Rightarrow (\text{nat} \times \text{nat}) \text{ nres}$ **where**

```

sum-max'-impl V  $\equiv$  do {
  (-, s, m)  $\leftarrow$  WHILET ( $\lambda(V, s, m). \neg ls.\text{isEmpty } V$ ) ( $\lambda(V, s, m).$  do {
    x  $\leftarrow$  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*:

$(V, V') \in \text{build-rel } ls.\alpha \text{ } ls.\text{invar} \implies \text{sum-max'-impl } V \leq \Downarrow \text{Id } (\text{sum-max}' V')$

<proof>

theorem *sum-max'-impl-correct*:

assumes $(V, V') \in \text{build-rel } ls.\alpha \text{ } ls.\text{invar}$ **and** $V' \neq \{\}$

shows $\text{sum-max'-impl } V \leq \text{SPEC } (\lambda(s, m). s = \sum V' \wedge m = \text{Max } V')$

<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*: $\text{nres-of } ?\text{sum-max-code} \leq \text{sum-max-impl } V$

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 $sum-max-code V = dRETURN (s,m) \implies s = \sum (ls.\alpha V) \wedge m = Max (ls.\alpha V)$

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*:

$[[ls.\alpha V \neq \{\}]] \implies sum-max'-code V = (\sum (ls.\alpha V), Max (ls.\alpha V))$

<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*:

$$\llbracket ls.\alpha V \neq \{\} \rrbracket \implies \text{sum-max''-code } V = (\sum (ls.\alpha V), \text{Max } (ls.\alpha V))$$

<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* $S f \sigma_0$ iterates $f::'x \Rightarrow 's \Rightarrow 's$ for each element in $S::'x \text{ set}$, starting with state $\sigma_0::'s$.

With foreach-loops, we could have written our example as follows:

definition *sum-max-it* :: *nat set* \Rightarrow (*nat* \times *nat*) *nres* **where**
sum-max-it $V \equiv \text{FOREACH } V (\lambda x (s,m). \text{RETURN } (s+x, \text{max } m x)) (0,0)$

theorem *sum-max-it-correct*:

assumes *PRE*: $V \neq \{\}$ **and** *FIN*: *finite* V
shows *sum-max-it* $V \leq \text{SPEC } (\lambda (s,m). s = \sum V \wedge m = \text{Max } V)$
<proof>

definition *sum-max-it-impl* :: *nat ls* \Rightarrow (*nat* \times *nat*) *nres* **where**

sum-max-it-impl $V \equiv \text{FOREACH } (ls.\alpha V) (\lambda x (s,m). \text{RETURN } (s+x, \text{max } m x)) (0,0)$

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 $(V, V') \in \text{build-rel } ls.\alpha \text{ } ls.\text{invar}$
shows *sum-max-it-impl* $V \leq \Downarrow \text{Id } (\text{sum-max-it } V')$
<proof>

schematic-goal *sum-max-it-code-aux*:

RETURN ?*sum-max-it-code* \leq *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 $ls.\alpha V \neq \{\}$

shows *sum-max-it-code* $V = (\sum (ls.\alpha V), Max (ls.\alpha V))$
 ⟨*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* \equiv *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*:

$$(?S = ?S') = (nofail ?S = nofail ?S' \wedge (\forall x. inres ?S x = inres ?S' x))$$

$$(?S \leq ?S') = (nofail ?S' \longrightarrow nofail ?S \wedge (\forall x. inres ?S x \longrightarrow inres ?S' x))$$

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* ($\lambda x. x \leq (2::nat) * (fst p + snd p)$) }
 ≤ *do* { *let* (*x,y*)=*p*; *z*←*SPEC* ($\lambda z. z \leq x+y$);
 a←*SPEC* ($\lambda a. a \leq x+y$); *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 Φ ; f }*.

ASSUME Φ Assumption. Succeeds if Φ does not hold, otherwise returns (). Note that the default usage with the *do*-notation is: *do {ASSUME Φ ; 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*.

$WHILE^I b f \sigma_0$, $WHILE_T^I b f \sigma_0$ While-loop with annotated invariant. It is asserted that the invariant holds.

$FOREACH S f \sigma_0$ 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.

$FOREACH^I S f \sigma_0$ Foreach-loop with annotated invariant.

Alternative syntax: $FOREACH^I S f \sigma_0$.

The invariant is a predicate of type $I::'a \text{ set} \Rightarrow 'b \Rightarrow \text{bool}$, where $I \text{ it}$ σ means, that the invariant holds for the remaining set of elements it and current state σ .

$FOREACH_C S c f \sigma_0$ Foreach-loop with explicit continuation condition.

Alternative syntax: $FOREACH_C S c f \sigma_0$.

If $c::'\sigma \Rightarrow \text{bool}$ becomes false for the current state, the iteration immediately terminates.

$FOREACH_C^I S c f \sigma_0$ Foreach-loop with explicit continuation condition and annotated invariant.

Alternative syntax: $FOREACH_C^I S c f \sigma_0$.

partial-function (nrec) Mode of the partial function package for the non-determinism monad.

Refinement

$(\leq)::'a \text{ nres} \Rightarrow 'a \text{ nres} \Rightarrow \text{bool}$ 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 *SUCCESS* and greatest element *FAIL*.

$\Downarrow R$ Concretization. Takes a refinement relation $R::('c \times 'a) \text{ set}$ 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: $S \leq \Downarrow R S' \iff \Uparrow R S \leq S'$.

$br \alpha I$ 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 explicitly by instantiating the respective proof-rule for the loop construct, e.g., *intro WHILE-rule[where I=...]* *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 ($\lambda(a,b)(c,d). \dots$) is done, and the monotonicity prover is used on monotonicity goals.

There is a standard setup for $\alpha=RETURN$ (transfer to plain function for total correct code generation), and $\alpha=nres-of$ (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 [2] 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*-function²:

definition *rbt-restrict-list* :: *'a::linorder rs* \Rightarrow *'a list* \Rightarrow *'a list*
where *rbt-restrict-list* *s l* == [*x* \leftarrow *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* \leftarrow *l*. *x* \in *rs*. α *s*]
<proof>

The lemma *rs.memb-correct*:

²Note that Isabelle/HOL uses the list comprehension syntax [*x* \leftarrow *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.α* maps a red-black-tree to the set that it represents. Moreover, we have to explicitly 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*:

$$\begin{aligned}
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) & \\
\llbracket rs.invar\ s; x \notin rs.\alpha\ s \rrbracket \Longrightarrow rs.\alpha\ (rs.ins-dj\ x\ s) &= insert\ x\ (rs.\alpha\ s) \\
\llbracket rs.invar\ s; x \notin rs.\alpha\ s \rrbracket \Longrightarrow rs.invar\ (rs.ins-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-abort\ m\ s &= min\ m\ (card\ (rs.\alpha\ s)) \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.\alpha\ (rs.union\ s1\ s2) &= rs.\alpha\ s1 \cup rs.\alpha\ s2 \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.invar\ (rs.union\ s1\ s2) & \\
\llbracket rs.invar\ s1; rs.invar\ s2; rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\} \rrbracket & \\
\Longrightarrow rs.\alpha\ (rs.union-dj\ s1\ s2) &= rs.\alpha\ s1 \cup rs.\alpha\ s2 \\
\llbracket rs.invar\ s1; rs.invar\ s2; rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\} \rrbracket & \\
\Longrightarrow rs.invar\ (rs.union-dj\ s1\ s2) & \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.\alpha\ (rs.diff\ s1\ s2) &= rs.\alpha\ s1 - rs.\alpha\ s2 \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \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) & \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.\alpha\ (rs.inter\ s1\ s2) &= rs.\alpha\ s1 \cap rs.\alpha\ s2 \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.invar\ (rs.inter\ s1\ s2) & \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.subset\ s1\ s2 &= (rs.\alpha\ s1 \subseteq rs.\alpha\ s2) \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.equal\ s1\ s2 &= (rs.\alpha\ s1 = rs.\alpha\ s2) \\
\llbracket rs.invar\ s1; rs.invar\ s2 \rrbracket \Longrightarrow rs.disjoint\ s1\ s2 &= (rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}) \\
\llbracket rs.invar\ s1; rs.invar\ s2; rs.disjoint-witness\ s1\ s2 = None \rrbracket & \\
\Longrightarrow rs.\alpha\ s1 \cap rs.\alpha\ s2 &= \{\} \\
\llbracket rs.invar\ s1; rs.invar\ s2; rs.disjoint-witness\ s1\ s2 = Some\ a \rrbracket & \\
\Longrightarrow a \in rs.\alpha\ s1 \cap rs.\alpha\ s2 & \\
rs.invar\ s \Longrightarrow set\ (rs.to-list\ s) &= rs.\alpha\ s
\end{aligned}$$

```

rs.invar s  $\implies$  distinct (rs.to-list s)
rs. $\alpha$  (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  $\equiv$  (
  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])
)

```

$\langle ML \rangle$

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  $\implies$  set (rs.to-list s) = rs. $\alpha$  s
rs.invar s  $\implies$  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  $\implies$  set (rs.to-sorted-list s) = rs. $\alpha$  s
rs.invar s  $\implies$  distinct (rs.to-sorted-list s)
rs.invar s  $\implies$  sorted (rs.to-sorted-list s)

```

and *rs.to-rev-list-correct*:

```

rs.invar s  $\implies$  set (rs.to-rev-list s) = rs. $\alpha$  s
rs.invar s  $\implies$  distinct (rs.to-rev-list s)
rs.invar s  $\implies$  sorted (rev (rs.to-rev-list s))

```

```

definition restrict-list-test  $\equiv$  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]

```

$\langle ML \rangle$

```

definition big-test n = (rs.from-list [(1::int)..n])

```

$\langle ML \rangle$

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*:

$$\begin{aligned} & \llbracket rs.invar\ S; I\ (rs.\alpha\ S)\ \sigma 0; \\ & \bigwedge x\ it\ \sigma. \llbracket c\ \sigma; x \in it; it \subseteq rs.\alpha\ S; I\ it\ \sigma \rrbracket \implies I\ (it - \{x\})\ (f\ x\ \sigma); \\ & \bigwedge \sigma. I\ \{\} \sigma \implies P\ \sigma; \bigwedge \sigma\ it. \llbracket it \subseteq rs.\alpha\ S; it \neq \{\}; \neg c\ \sigma; I\ it\ \sigma \rrbracket \implies P\ \sigma \rrbracket \\ & \implies P\ (rs.iteratei\ S\ c\ f\ \sigma 0) \end{aligned}$$

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:

$$\bigwedge x \text{ it } \sigma. \llbracket x \in \text{it}; \text{it} \subseteq \text{rs}.\alpha S; I \text{ it } \sigma \rrbracket \Longrightarrow I (\text{it} - \{x\}) (f x \sigma)$$

And the proposition to be shown for the final state must be a consequence of the invariant for no elements remaining: $\bigwedge \sigma. I \{\} \sigma \Longrightarrow P \sigma$.

A generalization of iterators are *interruptible iterators* where iteration is only continues while some condition on the state holds. Reasoning over interruptible iterators is also done by invariants: *rs.iteratei-rule-P*:

$$\begin{aligned} & \llbracket \text{rs.invar } S; I (\text{rs}.\alpha S) \sigma \theta; \\ & \bigwedge x \text{ it } \sigma. \llbracket c \sigma; x \in \text{it}; \text{it} \subseteq \text{rs}.\alpha S; I \text{ it } \sigma \rrbracket \Longrightarrow I (\text{it} - \{x\}) (f x \sigma); \\ & \bigwedge \sigma. I \{\} \sigma \Longrightarrow P \sigma; \bigwedge \sigma \text{ it}. \llbracket \text{it} \subseteq \text{rs}.\alpha S; \text{it} \neq \{\}; \neg c \sigma; I \text{ it } \sigma \rrbracket \Longrightarrow P \sigma \rrbracket \\ & \Longrightarrow P (\text{rs.iteratei } S c f \sigma \theta) \end{aligned}$$

Here, interruption of the iteration is handled by the premise

$$\bigwedge \sigma \text{ it}. \llbracket \text{it} \subseteq \text{rs}.\alpha S; \text{it} \neq \{\}; \neg c \sigma; I \text{ it } \sigma \rrbracket \Longrightarrow P \sigma$$

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* ($\lambda-. \text{True}$) ($\#$) []

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* ($\lambda \sigma. \neg \sigma$) ($\lambda x \sigma. P x$) *False*

lemma *hs-bex-correct*:

hs.invar s \Longrightarrow *hs-bex s P* \longleftrightarrow ($\exists x \in \text{hs}.\alpha s. P x$)

<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 [3] matches the concept of Generic Algorithm quite well.

Implementation An implementation of an interface provides a data structure for that interface together with an abstraction mapping and an invariant. Moreover, it provides implementations for some (or all) functions of that interface. For example, red-black trees are an implementation of the set-interface, with the abstraction mapping *rs. α* and invariant *rs.invar*; and the constant *rs.ins* implements the insert-function, as 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 implicitly 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. *rml.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 loses a great amount of control over the refinement. For example, in *Executable-Set*, all sets have to be represented by lists, and there is no possibility to represent one set differently from another.

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 concise 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 framework 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 quietly 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 its 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 deficiencies¹ (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.

¹For 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.

²A simple example is the always-diverging function $f_{\text{div}} :: \text{bool} = \text{while } (\lambda x. \text{True}) \text{ id True}$ that is definable in HOL. The lemma $\forall x. x = \text{if } f_{\text{div}} \text{ then } x \text{ else } x$ 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 and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.
- [3] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, November 1995.