Isabelle Collections Framework

By Peter Lammich and Andreas Lochbihler

September 1, 2025

Abstract

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

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

Contents

Chapter 1

Introduction

This development provides an efficient, extensible, machine checked collections framework for use in Isabelle/HOL. The library adopts the concepts of interface, implementation and generic algorithm known from object oriented (collection) libraries like the C++ Standard Template Library[5] or the Java Collections Framework[1] and makes them available in the Isabelle/HOL environment.

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

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

For more documentation and introductory material refer to the userguide (Section 5.2) and the ITP-2010 paper [3].

Chapter 2

The Generic Collection Framework

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

2.1 Interfaces

2.1.1 Map Interface

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

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

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

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

SPEC (λ((*k*,*v*),*m*′). *m k* = *Some v* ∧ *m*′ = *m* |‘ (−{*k*}))

context begin interpretation *autoref-syn* **.**

lemma [*autoref-op-pat*]:
Map.empty ≡ *op-map-empty*
(*m*::′*k*⇀′*v*) *k* ≡ *op-map-lookup*$*k*$*m*
m(*k*↦*v*) ≡ *op-map-update*$*k*$*v*$*m*
m |‘ (−{*k*}) ≡ *op-map-delete*$*k*$*m*
m |‘ {*k*∈*dom m*. *P* (*k*, *the* (*m k*))} ≡ *op-map-restrict*$*P*$*m*

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

∃ *k v*. *m*=[*k*↦*v*] ≡ *op-map-isSng*$*m*
∃ *k v*. [*k*↦*v*]=*m* ≡ *op-map-isSng*$*m*
∃ *k*. *dom m*={*k*} ≡ *op-map-isSng*$*m*
∃ *k*. {*k*} = *dom m* ≡ *op-map-isSng*$*m*
1 = *card* (*dom m*) ≡ *op-map-isSng*$*m*

⋀*P*. *Ball* (*map-to-set m*) *P* ≡ *op-map-ball*$*m*$*P*
⋀*P*. *Bex* (*map-to-set m*) *P* ≡ *op-map-bex*$*m*$*P*

card (*dom m*) ≡ *op-map-size*$*m*

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

⋀*P*. *SPEC* (λ(*k*,*v*). *m k*=*Some v* ∧ *P k v*) ≡ *op-map-sel*$*m*$*P*
⋀*P*. *SPEC* (λ(*k*,*v*). *P k v* ∧ *m k*=*Some v*) ≡ *op-map-sel*$*m*$*P*

⋀*P*. *SPEC* (λ(*k*,*v*). *m k* = *Some v*) ≡ *op-map-pick*$*m*
⋀*P*. *SPEC* (λ(*k*,*v*). (*k*,*v*) ∈ *map-to-set m*) ≡ *op-map-pick*$*m*
by (*auto*
intro!: *eq-reflection ext*
simp: *restrict-map-def dom-eq-singleton-conv card-Suc-eq map-to-set-def*
dest!: *sym*[*of Suc 0* *card* (*dom m*)] *sym*[*of -* *dom m*]
)

lemma [*autoref-op-pat*]:
SPEC (λ((*k*,*v*),*m*′). *m k* = *Some v* ∧ *m*′ = *m* |‘ (−{*k*}))
≡ *op-map-pick-remove*$*m*
by *simp*

lemma *op-map-pick-remove-alt*:
do {((*k*,*v*),*m*) ← *op-map-pick-remove m*; *f k v m*}

= (
do {
(k,v)←*SPEC* (λ(k,v). m k = *Some* v);
let m=m |‘ (−{k});
f k v m
})
unfolding *op-map-pick-remove-def*
apply (*auto simp*: *pw-eq-iff refine-pw-simps*)
done

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}
unfolding *op-map-pick-remove-alt* **.**

end

lemma [*autoref-itype*]:
op-map-empty $::_i$ $\langle Ik,Iv\rangle_i$*i-map*
op-map-lookup $::_i$ $Ik \rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Iv\rangle_i$*i-option*
op-map-update $::_i$ $Ik \rightarrow_i Iv \rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map*
op-map-delete $::_i$ $Ik \rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map*
op-map-restrict
$::_i$ ($\langle Ik,Iv\rangle_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map*
op-map-isEmpty $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ *i-bool*
op-map-isSng $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ *i-bool*
op-map-ball $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ ($\langle Ik,Iv\rangle_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
op-map-bex $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ ($\langle Ik,Iv\rangle_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
op-map-size $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ *i-nat*
op-map-size-abort $::_i$ *i-nat* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ *i-nat*
(++) $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map*
map-of $::_i$ $\langle\langle Ik,Iv\rangle_i$*i-prod*$\rangle_i$*i-list* $\rightarrow_i$ $\langle Ik,Iv\rangle_i$*i-map*

op-map-sel $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ ($Ik \rightarrow_i Iv \rightarrow_i$ *i-bool*)
$\rightarrow_i$ $\langle\langle Ik,Iv\rangle_i$*i-prod*$\rangle_i$*i-nres*
op-map-pick $::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle\langle Ik,Iv\rangle_i$*i-prod*$\rangle_i$*i-nres*
op-map-pick-remove
$::_i$ $\langle Ik,Iv\rangle_i$*i-map* $\rightarrow_i$ $\langle\langle\langle Ik,Iv\rangle_i$*i-prod*,$\langle Ik,Iv\rangle_i$*i-map*$\rangle_i$*i-prod*$\rangle_i$*i-nres*
by *simp-all*

lemma *hom-map1* [*autoref-hom*]:
CONSTRAINT Map.empty ($\langle Rk,Rv\rangle Rm$)
CONSTRAINT map-of ($\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel* → $\langle Rk,Rv\rangle Rm$)
CONSTRAINT (++) ($\langle Rk,Rv\rangle Rm$ → $\langle Rk,Rv\rangle Rm$ → $\langle Rk,Rv\rangle Rm$)
by *simp-all*

term *op-map-restrict*
lemma *hom-map2*[*autoref-hom*]:
CONSTRAINT op-map-lookup ($Rk \to \langle Rk,Rv \rangle Rm \to \langle Rv \rangle option\text{-}rel$)
CONSTRAINT op-map-update ($Rk \to Rv \to \langle Rk,Rv \rangle Rm \to \langle Rk,Rv \rangle Rm$)
CONSTRAINT op-map-delete ($Rk \to \langle Rk,Rv \rangle Rm \to \langle Rk,Rv \rangle Rm$)
CONSTRAINT op-map-restrict ($(\langle Rk,Rv \rangle prod\text{-}rel \to Id) \to \langle Rk,Rv \rangle Rm \to \langle Rk,Rv \rangle Rm$)
CONSTRAINT op-map-isEmpty ($\langle Rk,Rv \rangle Rm \to Id$)
CONSTRAINT op-map-isSng ($\langle Rk,Rv \rangle Rm \to Id$)
CONSTRAINT op-map-ball ($\langle Rk,Rv \rangle Rm \to (\langle Rk,Rv \rangle prod\text{-}rel \to Id) \to Id$)
CONSTRAINT op-map-bex ($\langle Rk,Rv \rangle Rm \to (\langle Rk,Rv \rangle prod\text{-}rel \to Id) \to Id$)
CONSTRAINT op-map-size ($\langle Rk,Rv \rangle Rm \to Id$)
CONSTRAINT op-map-size-abort ($Id \to \langle Rk,Rv \rangle Rm \to Id$)

CONSTRAINT op-map-sel ($\langle Rk,Rv \rangle Rm \to (Rk \to Rv \to bool\text{-}rel) \to \langle Rk \times_r Rv \rangle nres\text{-}rel$)
CONSTRAINT op-map-pick ($\langle Rk,Rv \rangle Rm \to \langle Rk \times_r Rv \rangle nres\text{-}rel$)
CONSTRAINT op-map-pick-remove ($\langle Rk,Rv \rangle Rm \to \langle (Rk \times_r Rv) \times_r \langle Rk,Rv \rangle Rm \rangle nres\text{-}rel$)
by *simp-all*

definition *finite-map-rel R* $\equiv$ *Range R* $\subseteq$ *Collect* (*finite* $\circ$ *dom*)
lemma *finite-map-rel-trigger*: *finite-map-rel R* $\Longrightarrow$ *finite-map-rel R* **.**

declaration ‹*Tagged-Solver.add-triggers*
Relators.relator-props-solver @{*thms finite-map-rel-trigger*}›

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* **.**
lemma [*autoref-op-pat*]:
fixes *s a b* :: *'a set* **and** *x*::*'a* **and** *P* :: *'a* $\Rightarrow$ *bool*
shows
s $-$ {*x*} $\equiv$ *op-set-delete*\$*x*\$*s*

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

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

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

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

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

SPEC (λ*x*. *x*∈*s* ∧ *P x*) $\equiv$ *op-set-sel*\$*P*\$*s*
SPEC (λ*x*. *P x* ∧ *x*∈*s*) $\equiv$ *op-set-sel*\$*P*\$*s*

SPEC (λ*x*. *x*∈*s*) $\equiv$ *op-set-pick*\$*s*
by (*auto intro*!: *eq-reflection simp*: *card-Suc-eq*)

lemma [*autoref-op-pat*]:
a $\times$ *b* $\equiv$ *op-set-cart a b*
by (*auto intro*!: *eq-reflection simp*: *card-Suc-eq*)

lemma [*autoref-op-pat*]:
SPEC (λ(*u*,*v*). (*u*,*v*)∈*s*) $\equiv$ *op-set-pick*\$*s*
SPEC (λ(*u*,*v*). *P u v* ∧ (*u*,*v*)∈*s*) $\equiv$ *op-set-sel*\$(*case-prod P*)\$*s*
SPEC (λ(*u*,*v*). (*u*,*v*)∈*s* ∧ *P u v*) $\equiv$ *op-set-sel*\$(*case-prod P*)\$*s*
by (*auto intro*!: *eq-reflection*)

lemma [*autoref-op-pat*]:
SPEC (λ*l*. *set l* = *s* ∧ *distinct l* ∧ *sorted-wrt ordR l*)
$\equiv$ *OP* (*op-set-to-sorted-list ordR*)\$*s*
SPEC (λ*l*. *set l* = *s* ∧ *sorted-wrt ordR l* ∧ *distinct l*)
$\equiv$ *OP* (*op-set-to-sorted-list ordR*)\$*s*
SPEC (λ*l*. *distinct l* ∧ *set l* = *s* ∧ *sorted-wrt ordR l*)
$\equiv$ *OP* (*op-set-to-sorted-list ordR*)\$*s*
SPEC (λ*l*. *distinct l* ∧ *sorted-wrt ordR l* ∧ *set l* = *s*)
$\equiv$ *OP* (*op-set-to-sorted-list ordR*)\$*s*

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

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

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

SPEC ($\lambda l.$ $s =$ *set* $l \wedge$ *distinct* l) $\equiv$ *op-set-to-list*\$*s*
SPEC ($\lambda l.$ *distinct* $l \wedge s =$ *set* l) $\equiv$ *op-set-to-list*\$*s*
by (*auto intro*!: *eq-reflection*)

end

lemma [*autoref-itype*]:
{} $::_i \langle I\rangle_i$*i-set*
insert $::_i I \rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
op-set-delete $::_i I \rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
$(\in) ::_i I \rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
op-set-isEmpty $::_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
op-set-isSng $::_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
$(\cup) ::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
$(\cap) ::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
($(-)$ :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set*) $::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
($(=)$ :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*) $::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
$(\subseteq) ::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
op-set-disjoint $::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-bool*
Ball $::_i \langle I\rangle_i$*i-set* $\rightarrow_i$ ($I \rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
Bex $::_i \langle I\rangle_i$*i-set* $\rightarrow_i$ ($I \rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
op-set-filter $::_i$ ($I \rightarrow_i$ *i-bool*) $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-set*
card $::_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-nat*
op-set-size-abort $::_i$ *i-nat* $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i$ *i-nat*
set $::_i \langle I\rangle_i$*i-list* $\rightarrow_i \langle I\rangle_i$*i-set*
op-set-sel $::_i$ ($I \rightarrow_i$ *i-bool*) $\rightarrow_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-nres*
op-set-pick $::_i \langle I\rangle_i$*i-set* $\rightarrow_i \langle I\rangle_i$*i-nres*
Sigma $::_i \langle Ia\rangle_i$*i-set* $\rightarrow_i$ ($Ia \rightarrow_i \langle Ib\rangle_i$*i-set*) $\rightarrow_i \langle\langle Ia,Ib\rangle_i$*i-prod*$\rangle_i$*i-set*

$(`) ::_i (Ia \rightarrow_i Ib) \rightarrow_i \langle Ia\rangle_i i\text{-}set \rightarrow_i \langle Ib\rangle_i i\text{-}set$
$op\text{-}set\text{-}cart ::_i \langle Ix\rangle_i Isx \rightarrow_i \langle Iy\rangle_i Isy \rightarrow_i \langle\langle Ix,\ Iy\rangle_i i\text{-}prod\rangle_i Isp$
$Union ::_i \langle\langle I\rangle_i i\text{-}set\rangle_i i\text{-}set \rightarrow_i \langle I\rangle_i i\text{-}set$
$atLeastLessThan ::_i i\text{-}nat \rightarrow_i i\text{-}nat \rightarrow_i \langle i\text{-}nat\rangle_i i\text{-}set$
by *simp-all*

lemma *hom-set1*[*autoref-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 $(`)$ $((Ra \rightarrow Rb) \rightarrow \langle Ra\rangle Rs \rightarrow \langle Rb\rangle Rs)$
CONSTRAINT *Union* $(\langle\langle R\rangle Ri\rangle Ro \rightarrow \langle R\rangle Ri)$
by *simp-all*

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* $(\langle R\rangle Rs \rightarrow \langle R\rangle Rn)$
by *simp-all*

lemma *hom-set-Sigma*[*autoref-hom*]:
CONSTRAINT *Sigma* $(\langle Ra\rangle Rs \rightarrow (Ra \rightarrow \langle Rb\rangle Rs) \rightarrow \langle\langle Ra,Rb\rangle prod\text{-}rel\rangle Rs2)$
by *simp-all*

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

lemma *finite-set-rel-trigger*: *finite-set-rel* $R \Longrightarrow$ *finite-set-rel* R **.**

declaration ‹*Tagged-Solver.add-triggers*
Relators.relator-props-solver @{*thms finite-set-rel-trigger*}›

end

2.1.3 Hashable Interface

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

type-synonym $'a\ eq = 'a \Rightarrow 'a \Rightarrow bool$
type-synonym $'k\ bhc = nat \Rightarrow 'k \Rightarrow nat$

Abstract and concrete hash functions

definition *is-bounded-hashcode* :: $('c\times'a)\ set \Rightarrow 'c\ eq \Rightarrow 'c\ bhc \Rightarrow bool$
  where *is-bounded-hashcode R eq bhc* $\equiv$
    $((eq,(=)) \in R \rightarrow R \rightarrow bool\text{-}rel) \wedge$
    $(\forall n.\ \forall\ x \in Domain\ R.\ \forall\ y \in Domain\ R.\ eq\ x\ y \longrightarrow bhc\ n\ x = bhc\ n\ y) \wedge$
    $(\forall n\ x.\ 1 < n \longrightarrow bhc\ n\ x < n)$
definition *abstract-bounded-hashcode* :: $('c\times'a)\ set \Rightarrow 'c\ bhc \Rightarrow 'a\ bhc$
  where *abstract-bounded-hashcode Rk bhc n x'* $\equiv$
    if $x' \in Range\ Rk$
      then $THE\ c.\ \exists x.\ (x,x') \in Rk \wedge bhc\ n\ x = c$
      else 0

lemma *is-bounded-hashcodeI*[*intro*]:
  $((eq,(=)) \in R \rightarrow R \rightarrow bool\text{-}rel) \Longrightarrow$
  $(\bigwedge x\ y\ n.\ x \in Domain\ R \Longrightarrow y \in Domain\ R \Longrightarrow eq\ x\ y \Longrightarrow bhc\ n\ x = bhc\ n\ y) \Longrightarrow$
  $(\bigwedge x\ n.\ 1 < n \Longrightarrow bhc\ n\ x < n) \Longrightarrow$ *is-bounded-hashcode R eq bhc*
  unfolding *is-bounded-hashcode-def* **by** *force*

lemma *is-bounded-hashcodeD*[*dest*]:
  assumes *is-bounded-hashcode R eq bhc*
  shows $(eq,(=)) \in R \rightarrow R \rightarrow bool\text{-}rel$ **and**
    $\bigwedge n\ x\ y.\ x \in Domain\ R \Longrightarrow y \in Domain\ R \Longrightarrow eq\ x\ y \Longrightarrow bhc\ n\ x = bhc\ n\ y$ **and**
    $\bigwedge n\ x.\ 1 < n \Longrightarrow bhc\ n\ x < n$
  using *assms* **unfolding** *is-bounded-hashcode-def* **by** *simp-all*

lemma *bounded-hashcode-welldefined*:
  assumes *BHC*: *is-bounded-hashcode Rk eq bhc* **and**
    R1: $(x1,x') \in Rk$ **and** *R2*: $(x2,x') \in Rk$
  shows $bhc\ n\ x1 = bhc\ n\ x2$
proof−
  from *is-bounded-hashcodeD*[*OF BHC*] **have** $(eq,(=)) \in Rk \rightarrow Rk \rightarrow bool\text{-}rel$ **by** *simp*
  with *R1 R2* **have** *eq x1 x2* **by** (*force dest*: *fun-relD*)
  thus *?thesis* **using** *R1 R2 BHC* **by** *blast*

```
qed

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 (intro fun-relI)
  fix n n' x x'
  assume A: (n,n') ∈ nat-rel and B: (x,x') ∈ Rk
  hence C: n = n' and D: x' ∈ Range Rk by auto
  have ?bhc' n' x' = bhc n x
      unfolding abstract-bounded-hashcode-def
      apply (simp add: C D, rule)
      apply (intro exI conjI, fact B, rule refl)
      apply (elim exE conjE, hypsubst,
          erule bounded-hashcode-welldefined[OF assms - B])
      done
  thus (bhc n x, ?bhc' n' x') ∈ nat-rel by simp
qed

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
  fix x'::'a and y'::'a and n'::nat assume x' = y'
  thus ?bhc' n' x' = ?bhc' n' y' by simp
next
  fix x'::'a and n'::nat assume 1 < n'
  from abstract-bhc-correct[OF bhc] show ?bhc' n' x' < n'
  proof (cases x' ∈ Range Rk)
    case False
      with ‹1 < n'› show ?thesis
          unfolding abstract-bounded-hashcode-def by simp
  next
    case True
      then obtain x where (x,x') ∈ Rk ..
      have (n',n') ∈ nat-rel ..
      from abstract-bhc-correct[OF assms] have ?bhc' n' x' = bhc n' x
        apply −
        apply (drule fun-relD[OF - ‹(n',n') ∈ nat-rel›],
              drule fun-relD[OF - ‹(x,x') ∈ Rk›], simp)
        done
      also from ‹1 < n'› and bhc have ... < n' by blast
      finally show ?bhc' n' x' < n' .
  qed
qed simp
```

lemma *hashable-bhc-is-bhc*[*autoref-ga-rules*]:
⟦*STRUCT-EQ-tag eq* (=); *REL-FORCE-ID R*⟧ ⟹ *is-bounded-hashcode R eq bounded-hashcode-nat*
unfolding *is-bounded-hashcode-def*
by (*simp add*: *bounded-hashcode-nat-bounds*)

Default hash map size

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

lemma *hashable-def-size-is-def-size*[*autoref-ga-rules*]:
shows *is-valid-def-hm-size TYPE*(*'k*::*hashable*) (*def-hashmap-size TYPE*(*'k*))
unfolding *is-valid-def-hm-size-def* **by** (*fact def-hashmap-size*)

end

2.1.4 Orderings By Comparison Operator

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

Basic Definitions

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

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

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

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

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

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

$\llbracket$ $x{\in}D$; $y{\in}D$; $z{\in}D$; $cmp\ x\ y = LESS$; $cmp\ y\ z = LESS\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket$ $x{\in}D$; $y{\in}D$; $z{\in}D$; $cmp\ x\ y = LESS$; $cmp\ y\ z = EQUAL\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket$ $x{\in}D$; $y{\in}D$; $z{\in}D$; $cmp\ x\ y = EQUAL$; $cmp\ y\ z = LESS\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket$ $x{\in}D$; $y{\in}D$; $z{\in}D$; $cmp\ x\ y = EQUAL$; $cmp\ y\ z = EQUAL\rrbracket \Longrightarrow cmp\ x\ z = EQUAL$
begin
abbreviation *le* ≡ *comp2le cmp*
abbreviation *lt* ≡ *comp2lt cmp*

lemma *eq-sym*: $\llbracket x{\in}D;\ y{\in}D\rrbracket \Longrightarrow cmp\ x\ y = EQUAL \Longrightarrow cmp\ y\ x = EQUAL$
apply (*cases cmp y x*)
using *lt-eq lt-eq*[*symmetric*]
by *auto*
end

abbreviation *linorder* ≡ *linorder-on UNIV*

lemma *linorder-to-class*:
assumes *linorder cmp*
assumes [*simp*]: $\bigwedge x\ y.\ cmp\ x\ y = EQUAL \Longrightarrow x{=}y$
shows *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
proof −
interpret *linorder-on UNIV cmp* **by** *fact*
show *?thesis*
apply (*unfold-locales*)
unfolding *comp2le-def comp2lt-def*
apply (*auto split*: *comp-res.split comp-res.split-asm*)
using *lt-eq* **apply** *simp*
using *lt-eq* **apply** *simp*
using *lt-eq*[*symmetric*] **apply** *simp*
apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
using *lt-eq* **apply** *simp*
using *lt-eq* **apply** *simp*
using *lt-eq*[*symmetric*] **apply** *simp*
done
qed

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

lemma (**in** *linorder*) *class-to-linorder*:
linorder (*dflt-cmp* (≤) (<))
apply (*unfold-locales*)
unfolding *dflt-cmp-def*

by (*auto split*: *if-split-asm*)

lemma *restrict-linorder*: $[\![$*linorder-on D cmp* ; $D'\subseteq D]\!] \Longrightarrow$ *linorder-on* D' *cmp*
apply (*rule linorder-on.intro*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*2*) *linorder-on.lt-eq*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*1*) *linorder-on.refl*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *rev-subsetD*)+
apply (*erule* (*5*) *linorder-on.trans*)
done

Operations on Linear Orderings

Map with injective function

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

lemma *img-linorder*[*intro?*]:
assumes *LO*: *linorder-on* (*f'D*) *cmp*
shows *linorder-on D* (*cmp-img f cmp*)
apply *unfold-locales*
unfolding *cmp-img-def*
apply (*rule linorder-on.lt-eq*[*OF LO*], *auto*) []
apply (*rule linorder-on.refl*[*OF LO*], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* -2], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* -2], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* -2], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* -2], *auto*) []
done

Combine

definition *cmp-combine D1 cmp1 D2 cmp2 a b* ≡
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 $x\in A$ | $x\notin A$ $x\in B$
using *assms* **by** *blast*

lemma *combine-linorder*[*intro?*]:
  assumes *linorder-on D1 cmp1*
  assumes *linorder-on D2 cmp2*
  assumes *D* = *D1*∪*D2*
  shows *linorder-on D* (*cmp-combine D1 cmp1 D2 cmp2*)
  apply *unfold-locales*
  unfolding *cmp-combine-def*
  using *assms* **apply** −
  apply (*simp only*:)
  apply (*elim UnE*)
  apply (*auto dest*: *linorder-on.lt-eq*) [*4*]

  apply (*simp only*:)
  apply (*elim UnE*)
  apply (*auto dest*: *linorder-on.refl*) [*2*]

  apply (*simp only*:)
  apply (*elim UnE′*)
  apply *simp-all* [*8*]
  apply (*erule* (*5*) *linorder-on.trans*)
  apply (*erule* (*5*) *linorder-on.trans*)

  apply (*simp only*:)
  apply (*elim UnE′*)
  apply *simp-all* [*8*]
  apply (*erule* (*5*) *linorder-on.trans*)
  apply (*erule* (*5*) *linorder-on.trans*)

  apply (*simp only*:)
  apply (*elim UnE′*)
  apply *simp-all* [*8*]
  apply (*erule* (*5*) *linorder-on.trans*)
  apply (*erule* (*5*) *linorder-on.trans*)

  apply (*simp only*:)
  apply (*elim UnE′*)
  apply *simp-all* [*8*]
  apply (*erule* (*5*) *linorder-on.trans*)
  apply (*erule* (*5*) *linorder-on.trans*)
  done

Universal Linear Ordering

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

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

else GREATER

lemma *univ-wo*: *well-order-on UNIV univ-order-rel*
unfolding *univ-order-rel-def*
using *well-order-on*[*of UNIV*]
..

lemma *univ-linorder*[*intro?*]: *linorder univ-cmp*
apply *unfold-locales*
unfolding *univ-cmp-def*
apply (*auto split*: *if-split-asm*)
using *univ-wo*
apply −
unfolding *well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def*
apply (*auto simp add*: *antisym-def*) []
apply (*unfold total-on-def*, *fast*) []
apply (*unfold trans-def*, *fast*) []
apply (*auto simp add*: *antisym-def*) []
done

Extend any linear order to a universal order

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

lemma *extend-linorder*[*intro?*]:
linorder-on D cmp ⟹ *linorder* (*cmp-extend D cmp*)
unfolding *cmp-extend-def*
apply *rule*
apply *assumption*
apply *rule*
by *simp*

Lexicographic Order on Lists

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

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

```
  | GREATER ⇒ GREATER
 ))

lemma cmp-lex-alt: cmp-lex cmp l m = cmp-lex' cmp l m
 apply (induct l arbitrary: m)
 apply (auto split: comp-res.split list.split)
 done

lemma (in linorder-on) lex-linorder[intro?]:
 linorder-on (lists D) (cmp-lex cmp)
proof
 fix l m
 assume l∈lists D   m∈lists D
 thus (cmp-lex cmp l m = LESS) = (cmp-lex cmp m l = GREATER)
  apply (induct cmp≡cmp l m rule: cmp-lex.induct)
  apply (auto split: comp-res.split simp: lt-eq)
  apply (auto simp: lt-eq[symmetric])
  done
next
 fix x
 assume x∈lists D
 thus cmp-lex cmp x x = EQUAL
  by (induct x) auto
next
 fix x y z
 assume M: x∈lists D   y∈lists D   z∈lists D

 {
  assume cmp-lex cmp x y = LESS   cmp-lex cmp y z = LESS
  thus cmp-lex cmp x z = LESS
   using M
   apply (induct cmp≡cmp x y arbitrary: z rule: cmp-lex.induct)
   apply (auto split: comp-res.split-asm comp-res.split)
   apply (case-tac z, auto) []
   apply (case-tac z,
    auto split: comp-res.split-asm comp-res.split,
    (drule (4) trans, simp)+
   ) []
   apply (case-tac z,
    auto split: comp-res.split-asm comp-res.split,
    (drule (4) trans, simp)+
   ) []
   done
 }

 {
  assume cmp-lex cmp x y = LESS   cmp-lex cmp y z = EQUAL
  thus cmp-lex cmp x z = LESS
   using M
```

apply (*induct cmp≡cmp x y arbitrary*: *z rule*: *cmp-lex.induct*)
apply (*auto split*: *comp-res.split-asm comp-res.split*)
apply (*case-tac z*, *auto*) []
apply (*case-tac z*,
auto split: *comp-res.split-asm comp-res.split*,
(*drule* (*4*) *trans*, *simp*)+
) []
apply (*case-tac z*,
auto split: *comp-res.split-asm comp-res.split*,
(*drule* (*4*) *trans*, *simp*)+
) []
done
}

{
assume *cmp-lex cmp x y = EQUAL* *cmp-lex cmp y z = LESS*
thus *cmp-lex cmp x z = LESS*
using *M*
apply (*induct cmp≡cmp x y arbitrary*: *z rule*: *cmp-lex.induct*)
apply (*auto split*: *comp-res.split-asm comp-res.split*)
apply (*case-tac z*,
auto split: *comp-res.split-asm comp-res.split*,
(*drule* (*4*) *trans*, *simp*)+
) []
done
}

{
assume *cmp-lex cmp x y = EQUAL* *cmp-lex cmp y z = EQUAL*
thus *cmp-lex cmp x z = EQUAL*
using *M*
apply (*induct cmp≡cmp x y arbitrary*: *z rule*: *cmp-lex.induct*)
apply (*auto split*: *comp-res.split-asm comp-res.split*)
apply (*case-tac z*)
apply (*auto split*: *comp-res.split-asm comp-res.split*)
apply (*drule* (*4*) *trans*, *simp*)+
done
}
qed

Lexicographic Order on Pairs

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

lemma *cmp-prod-alt*: *cmp-prod* = (λ*cmp1 cmp2* (*a1*,*a2*) (*b1*,*b2*). (

case cmp1 a1 b1 of
  LESS ⇒ *LESS*
| *EQUAL* ⇒ *cmp2 a2 b2*
| *GREATER* ⇒ *GREATER*))
by (*auto intro*!: *ext*)

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 −
interpret *A*: *linorder-on A cmp1*
+ *B*: *linorder-on B cmp2* **by** *fact*+

show *?thesis*
apply *unfold-locales*
apply (*auto split*: *comp-res.split comp-res.split-asm*,
simp-all add: *A.lt-eq B.lt-eq*,
simp-all add: *A.lt-eq*[*symmetric*]
) []

apply (*auto split*: *comp-res.split comp-res.split-asm*) []

apply (*auto split*: *comp-res.split comp-res.split-asm*) []
apply (*drule* (*4*) *A.trans B.trans*, *simp*)+

apply (*auto split*: *comp-res.split comp-res.split-asm*) []
apply (*drule* (*4*) *A.trans B.trans*, *simp*)+

apply (*auto split*: *comp-res.split comp-res.split-asm*) []
apply (*drule* (*4*) *A.trans B.trans*, *simp*)+

apply (*auto split*: *comp-res.split comp-res.split-asm*) []
apply (*drule* (*4*) *A.trans B.trans*, *simp*)+
done
qed

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*
apply (*induct l*)
apply (*auto simp*: *set-to-map-insert*)
done

context *linorder* **begin**

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

```
  assumes finite A
  shows [] = sorted-list-of-set A ⟷ A={}
  using assms
  by (auto dest: sym)

lemma set-insort[simp]: set (insort x l) = insert x (set l)
  by (induct l) auto

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
  using assms
proof –
  from ‹finite B› have B = set (sorted-list-of-set B) by simp
  also from assms have ... = set (sorted-list-of-set (A))
    by simp
  also from ‹finite A›
  have set (sorted-list-of-set (A)) = A
    by simp
  finally show ?thesis by simp
qed

lemma sorted-list-of-set-inj: inj-on sorted-list-of-set (Collect finite)
  apply (rule inj-onI)
  using sorted-list-of-set-inj-aux
  by blast

definition sorted-list-of-map m ≡
  map (λk. (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 –
  have dom (map-of l) = set (map fst l) by (induct l) force+
  hence sorted-list-of-set (dom (map-of l)) = map fst l
    using sorted-list-of-set.idem-if-sorted-distinct[OF assms(2,1)] by simp
  hence sorted-list-of-map (map-of l)
    = map (λk. (k, the (map-of l k))) (map fst l)
    unfolding sorted-list-of-map-def by simp
  also have ... = l using ‹distinct (map fst l)›
  proof (induct l)
    case Nil thus ?case by simp
  next
    case (Cons a l)
```

```
    hence
      1: distinct (map fst l)
      and 2: fst a∉fst‘set l
      and 3: map (λk. (k, the (map-of l k))) (map fst l) = l
      by simp-all

    from 2 have [simp]: ¬(∃x∈set l. fst x = fst a)
      by (auto simp: image-iff)

    show ?case
      apply simp
      apply (subst (3) 3[symmetric])
      apply simp
      done
  qed
  finally show ?thesis .
qed

lemma map-of-sorted-list-of-map[simp]:
  assumes FIN: finite (dom m)
  shows map-of (sorted-list-of-map m) = m
  unfolding sorted-list-of-map-def
proof –
  have set (sorted-list-of-set (dom m)) = dom m
    and DIST: distinct (sorted-list-of-set (dom m))
    by (simp-all add: FIN)

  have [simp]: (fst ∘ (λk. (k, the (m k)))) = id by auto

  have [simp]: (λk. (k, the (m k))) ‘ dom m = map-to-set m
    by (auto simp: map-to-set-def)

  show map-of (map (λk. (k, the (m k))) (sorted-list-of-set (dom m))) = m
    apply (subst set-to-map-set-is-map-of[symmetric])
    apply (simp add: DIST)
    apply (subst set-map)
    apply (simp add: FIN map-to-set-inverse)
    done
qed

lemma sorted-list-of-map-inj-aux:
  fixes A B :: 'a⇀'b
  assumes [simp]: finite (dom A)
  assumes [simp]: finite (dom B)
  assumes E: sorted-list-of-map A = sorted-list-of-map B
  shows A=B
  using assms
proof –
  have A = map-of (sorted-list-of-map A) by simp
```

```
    also note E
    also have map-of (sorted-list-of-map B) = B by simp
    finally show ?thesis .
  qed

  lemma sorted-list-of-map-inj:
    inj-on sorted-list-of-map (Collect (finite o dom))
    apply (rule inj-onI)
    using sorted-list-of-map-inj-aux
    by auto
end

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

thm img-linorder

lemma set-ord-linear[intro?]:
  linorder cmp ⟹ linorder (cmp-set cmp)
  unfolding cmp-set-def
  apply rule
  apply rule
  apply (rule restrict-linorder)
  apply (erule linorder-on.lex-linorder)
  apply simp
  done

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


lemma map-to-set-inj[intro!]: inj map-to-set
  apply (rule inj-onI)
  unfolding map-to-set-def
  apply (rule ext)
  apply (case-tac x xa)
  apply (case-tac [!] y xa)
  apply force+
  done

corollary map-to-set-inj'[intro!]: inj-on map-to-set S
```

```
  by (metis map-to-set-inj subset-UNIV inj-on-subset)

lemma map-ord-linear[intro?]:
  assumes A: linorder cmpk
  assumes B: linorder cmpv
  shows linorder (cmp-map cmpk cmpv)
proof −
  interpret lk: linorder-on UNIV cmpk by fact
  interpret lv: linorder-on UNIV cmpv by fact

  show ?thesis
    unfolding cmp-map-def
    apply rule
    apply rule
    apply (rule restrict-linorder)
    apply (rule linorder-on.lex-linorder)
    apply (rule)
    apply fact
    apply fact
    apply simp
    done
qed


locale eq-linorder-on = linorder-on +
  assumes cmp-imp-equal: ⟦x∈D; y∈D⟧ ⟹ cmp x y = EQUAL ⟹ x = y
begin
  lemma cmp-eq[simp]: ⟦x∈D; y∈D⟧ ⟹ cmp x y = EQUAL ⟷ x = y
    by (auto simp: cmp-imp-equal)
end

abbreviation eq-linorder ≡ eq-linorder-on UNIV

lemma dflt-cmp-2inv[simp]:
  dflt-cmp (comp2le cmp) (comp2lt cmp) = cmp
  unfolding dflt-cmp-def[abs-def] comp2le-def[abs-def] comp2lt-def[abs-def]
  apply (auto split: comp-res.splits intro!: ext)
  done

lemma (in linorder) dflt-cmp-inv2[simp]:
  shows
  (comp2le (dflt-cmp (≤) (<)))= (≤)
  (comp2lt (dflt-cmp (≤) (<)))= (<)
proof −
  show (comp2lt (dflt-cmp (≤) (<)))= (<)
    unfolding dflt-cmp-def[abs-def] comp2le-def[abs-def] comp2lt-def[abs-def]
    apply (auto split: comp-res.splits intro!: ext)
    done
```

show (*comp2le* (*dflt-cmp* (≤) (<))) = (≤)
unfolding *dflt-cmp-def*[*abs-def*] *comp2le-def*[*abs-def*] *comp2lt-def*[*abs-def*]
apply (*auto split*: *comp-res.splits intro*!: *ext*)
done

qed

lemma *eq-linorder-class-conv*:
eq-linorder cmp ⟷ *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
proof
assume *eq-linorder cmp*
then interpret *eq-linorder-on UNIV cmp* **.**
have *linorder cmp* **by** *unfold-locales*
show *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
apply (*rule linorder-to-class*)
apply *fact*
by *simp*
next
assume *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
then interpret *linorder comp2le cmp comp2lt cmp* **.**

from *class-to-linorder* **interpret** *linorder-on UNIV cmp*
by *simp*
show *eq-linorder cmp*
proof
fix *x y*
assume *cmp x y* = *EQUAL*
hence *comp2le cmp x y* ¬*comp2lt cmp x y*
by (*auto simp*: *comp2le-def comp2lt-def*)
thus *x*=*y* **by** *simp*
qed
qed

lemma (**in** *linorder*) *class-to-eq-linorder*:
eq-linorder (*dflt-cmp* (≤) (<))
proof −
interpret *linorder-on UNIV dflt-cmp* (≤) (<)
by (*rule class-to-linorder*)

show *?thesis*
apply *unfold-locales*
apply (*auto simp*: *dflt-cmp-def split*: *if-split-asm*)
done
qed

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

interpret *eq-linorder-on UNIV cmp* **by** *fact*
show *?thesis*
apply (*intro ext*)
unfolding *comp2eq-def*
apply (*auto split*: *comp-res.split dest*: *refl*)
done
qed

lemma *restrict-eq-linorder*:
assumes *eq-linorder-on D cmp*
assumes *S*: $D' \subseteq D$
shows *eq-linorder-on* D' *cmp*
proof −
interpret *eq-linorder-on D cmp* **by** *fact*

show *?thesis*
apply (*rule eq-linorder-on.intro*)
apply (*rule restrict-linorder*[**where** *D*=*D*])
apply *unfold-locales* []
apply *fact*
apply *unfold-locales*
using *S*
apply −
apply (*drule* (*1*) *rev-subsetD*)+
apply *auto*
done
qed

lemma *combine-eq-linorder*[*intro?*]:
assumes *A*: *eq-linorder-on D1 cmp1*
assumes *B*: *eq-linorder-on D2 cmp2*
assumes *EQ*: $D=D1 \cup D2$
shows *eq-linorder-on D* (*cmp-combine D1 cmp1 D2 cmp2*)
proof −
interpret *A*: *eq-linorder-on D1 cmp1* **by** *fact*
interpret *B*: *eq-linorder-on D2 cmp2* **by** *fact*
interpret *linorder-on* (*D1* ∪ *D2*) (*cmp-combine D1 cmp1 D2 cmp2*)
apply *rule*
apply *unfold-locales*
by *simp*

show *?thesis*
apply (*simp only*: *EQ*)
apply *unfold-locales*
unfolding *cmp-combine-def*
by (*auto split*: *if-split-asm*)
qed

lemma *img-eq-linorder*[*intro?*]:

assumes *A*: *eq-linorder-on* (*f'D*) *cmp*
assumes *INJ*: *inj-on f D*
shows *eq-linorder-on D* (*cmp-img f cmp*)
proof −
interpret *eq-linorder-on f'D cmp* **by** *fact*
interpret *L*: *linorder-on* (*D*) (*cmp-img f cmp*)
apply *rule*
apply *unfold-locales*
done

show *?thesis*
apply *unfold-locales*
unfolding *cmp-img-def*
using *INJ*
apply (*auto dest*: *inj-onD*)
done
qed

lemma *univ-eq-linorder*[*intro?*]:
shows *eq-linorder univ-cmp*
apply (*rule eq-linorder-on.intro*)
apply *rule*
apply *unfold-locales*
unfolding *univ-cmp-def*
apply (*auto split*: *if-split-asm*)
done

lemma *extend-eq-linorder*[*intro?*]:
assumes *eq-linorder-on D cmp*
shows *eq-linorder* (*cmp-extend D cmp*)
proof −
interpret *eq-linorder-on D cmp* **by** *fact*
show *?thesis*
unfolding *cmp-extend-def*
apply (*rule*)
apply *fact*
apply *rule*
by *simp*
qed

lemma *lex-eq-linorder*[*intro?*]:
assumes *eq-linorder-on D cmp*
shows *eq-linorder-on* (*lists D*) (*cmp-lex cmp*)
proof −
interpret *eq-linorder-on D cmp* **by** *fact*
show *?thesis*
apply (*rule eq-linorder-on.intro*)
apply *rule*
apply *unfold-locales*

```
  subgoal for l m
    apply (induct cmp≡cmp l m rule: cmp-lex.induct)
    apply (auto split: comp-res.splits)
    done
  done
qed

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 –
 interpret A: eq-linorder-on D1 cmp1 by fact
 interpret B: eq-linorder-on D2 cmp2 by fact
 show ?thesis
  apply (rule eq-linorder-on.intro)
  apply rule
  apply unfold-locales
  apply (auto split: comp-res.splits)
  done
qed

lemma set-ord-eq-linorder[intro?]:
 eq-linorder cmp ⟹ eq-linorder (cmp-set cmp)
 unfolding cmp-set-def
 apply rule
 apply rule
 apply (rule restrict-eq-linorder)
 apply rule
 apply assumption
 apply simp

 apply (rule linorder.sorted-list-of-set-inj)
 apply (subst (asm) eq-linorder-class-conv)
 .

lemma map-ord-eq-linorder[intro?]:
 ⟦eq-linorder cmpk; eq-linorder cmpv⟧ ⟹ eq-linorder (cmp-map cmpk cmpv)
 unfolding cmp-map-def
 apply rule
 apply rule
 apply (rule restrict-eq-linorder)
 apply rule
 apply rule
 apply assumption
 apply assumption
 apply simp

 apply (rule linorder.sorted-list-of-map-inj)
```

apply (*subst* (*asm*) *eq-linorder-class-conv*)
.

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

lemma *cmp-unit-eq-linorder*:
eq-linorder cmp-unit
by *unfold-locales simp-all*

Parametricity

lemma *param-cmp-extend*[*param*]:
assumes $(cmp,cmp')\in R \rightarrow R \rightarrow Id$
assumes *Range* $R \subseteq D$
shows (*cmp*,*cmp-extend D cmp'*) $\in R \rightarrow R \rightarrow Id$
unfolding *cmp-extend-def cmp-combine-def*[*abs-def*]
using *assms*
apply *clarsimp*
by (*blast dest*!: *fun-relD*)

lemma *param-cmp-img*[*param*]:
(*cmp-img*,*cmp-img*) $\in (Ra\rightarrow Rb) \rightarrow (Rb\rightarrow Rb\rightarrow Rc) \rightarrow Ra \rightarrow Ra \rightarrow Rc$
unfolding *cmp-img-def*[*abs-def*]
by *parametricity*

lemma *param-comp-res*[*param*]:
(*LESS*,*LESS*)∈*Id*
(*EQUAL*,*EQUAL*)∈*Id*
(*GREATER*,*GREATER*)∈*Id*
(*case-comp-res*,*case-comp-res*)∈*Ra*→*Ra*→*Ra*→*Id*→*Ra*
by (*auto split*: *comp-res.split*)

term *cmp-lex*
lemma *param-cmp-lex*[*param*]:
(*cmp-lex*,*cmp-lex*)∈(*Ra*→*Rb*→*Id*)→⟨*Ra*⟩*list-rel*→⟨*Rb*⟩*list-rel*→*Id*
unfolding *cmp-lex-alt*[*abs-def*] *cmp-lex'-def*
by (*parametricity*)

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

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

lemma *param-comp2eq*[*param*]: (*comp2eq*,*comp2eq*)∈(*R*→*R*→*Id*)→*R*→*R*→*Id*
  unfolding *comp2eq-def*[*abs-def*]
  by (*parametricity*)

lemma *cmp-combine-paramD*:
  assumes (*cmp*,*cmp-combine D1 cmp1 D2 cmp2*)∈*R*→*R*→*Id*
  assumes *Range R* ⊆ *D1*
  shows (*cmp*,*cmp1*)∈*R*→*R*→*Id*
  using *assms*
  unfolding *cmp-combine-def*[*abs-def*]
  apply (*intro fun-relI*)
  apply (*drule-tac x*=*a* **in** *fun-relD*, *assumption*)
  apply (*drule-tac x*=*aa* **in** *fun-relD*, *assumption*)
  apply (*drule RangeI*, *drule* (*1*) *rev-subsetD*)
  apply (*drule RangeI*, *drule* (*1*) *rev-subsetD*)
  apply *simp*
  done

lemma *cmp-extend-paramD*:
  assumes (*cmp*,*cmp-extend D cmp'*)∈*R*→*R*→*Id*
  assumes *Range R* ⊆ *D*
  shows (*cmp*,*cmp'*)∈*R*→*R*→*Id*
  using *assms*
  unfolding *cmp-extend-def*
  apply (*rule cmp-combine-paramD*)
  done

Tuning of Generated Implementation

lemma [*autoref-post-simps*]: *comp2eq* (*dflt-cmp* (≤) ((<)::-::*linorder*⇒-)) = (=)
  by (*simp add*: *class-to-eq-linorder eq-linorder-comp2eq-eq*)

end

2.2 Generic Algorithms

2.2.1 Generic Set Algorithms

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

  lemma *foldli-union*: *det-fold-set X* (λ-. *True*) *insert a* ((∪) *a*)
  proof (*rule*, *goal-cases*)

```
  case (1 l) thus ?case
    by (induct l arbitrary: a) auto
qed

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

lemma gen-union[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes INS: GEN-OP ins Set.insert (Rk→⟨Rk⟩Rs2→⟨Rk⟩Rs2)
  assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 tsl)
  shows (gen-union (λx. foldli (tsl x)) ins,(∪))
  ∈ (⟨Rk⟩Rs1) → (⟨Rk⟩Rs2) → (⟨Rk⟩Rs2)
  apply (intro fun-relI)
  apply (subst Un-commute)
  unfolding gen-union-def
  apply (rule det-fold-set[OF
    foldli-union IT[unfolded autoref-tag-defs]])
  using INS
  unfolding autoref-tag-defs
  apply (parametricity)+
  done

lemma foldli-inter: det-fold-set X (λ-. True)
  (λx s. if x∈a then insert x s else s) {} (λs. s∩a)
  (is det-fold-set - - ?f - -)
proof −
  {
    fix l s0
    have foldli l (λ-. True)
      (λx s. if x∈a then insert x s else s) s0 = s0 ∪ (set l ∩ a)
      by (induct l arbitrary: s0) auto
  }
  from this[of - {}] show ?thesis apply − by rule simp
qed

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

lemma gen-inter[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 tsl)
  assumes MEMB:
```

GEN-OP memb2 (∈) (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
assumes *INS*:
GEN-OP ins3 Set.insert (*Rk*→⟨*Rk*⟩*Rs3*→⟨*Rk*⟩*Rs3*)
assumes *EMPTY*:
GEN-OP empty3 {} (⟨*Rk*⟩*Rs3*)
shows (*gen-inter* (λ*x*. *foldli* (*tsl x*)) *memb2 ins3 empty3*,(∩))
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs3*)
apply (*intro fun-relI*)
unfolding *gen-inter-def*
apply (*rule det-fold-set*[*OF foldli-inter IT*[*unfolded autoref-tag-defs*]])
using *MEMB INS EMPTY*
unfolding *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-diff*:
det-fold-set X (λ-. *True*) (λ*x s*. *op-set-delete x s*) *s* ((−) *s*)
proof (*rule*, *goal-cases*)
case (*1 l*) **thus** *?case*
by (*induct l arbitrary*: *s*) *auto*
qed

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

lemma *gen-diff*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *DEL*:
GEN-OP del1 op-set-delete (*Rk* → ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs1*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs2 it2*)
shows (*gen-diff del1* (λ*x*. *foldli* (*it2 x*)),(−))
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs1*)
apply (*intro fun-relI*)
unfolding *gen-diff-def*
apply (*rule det-fold-set*[*OF foldli-diff IT*[*unfolded autoref-tag-defs*]])
using *DEL*
unfolding *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-ball-aux*:
foldli l (λ*x*. *x*) (λ*x* -. *P x*) *b* ⟷ *b* ∧ *Ball* (*set l*) *P*
by (*induct l arbitrary*: *b*) *auto*

lemma *foldli-ball*: *det-fold-set X* (λ*x*. *x*) (λ*x* -. *P x*) *True* (λ*s*. *Ball s P*)
apply *rule* **using** *foldli-ball-aux*[**where** *b*=*True*] **by** *simp*

definition *gen-ball* :: - ⇒ '*s* ⇒ ('*k* ⇒ *bool*) ⇒ -

where *gen-ball it s P* ≡ *it s* (λ*x*. *x*) (λ*x* -. *P x*) *True*

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-ball* (λ*x*. *foldli* (*it x*)),*Ball*) ∈ ⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*
apply (*intro fun-relI*)
unfolding *gen-ball-def*
apply (*rule det-fold-set*[*OF foldli-ball IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *foldli-bex-aux*: *foldli l* (λ*x*. ¬*x*) (λ*x* -. *P x*) *b* ⟷ *b* ∨ *Bex* (*set l*) *P*
by (*induct l arbitrary*: *b*) *auto*

lemma *foldli-bex*: *det-fold-set X* (λ*x*. ¬*x*) (λ*x* -. *P x*) *False* (λ*s*. *Bex s P*)
apply *rule* **using** *foldli-bex-aux*[**where** *b*=*False*] **by** *simp*

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

lemma *gen-bex*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-bex* (λ*x*. *foldli* (*it x*)),*Bex*) ∈ ⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*
apply (*intro fun-relI*)
unfolding *gen-bex-def*
apply (*rule det-fold-set*[*OF foldli-bex IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *ball-subseteq*:
(*Ball s1* (λ*x*. *x*∈*s2*)) ⟷ *s1* ⊆ *s2*
by *blast*

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

lemma *gen-subseteq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball1 Ball* (⟨*Rk*⟩*Rs1* → (*Rk*→*Id*) → *Id*)
assumes *GEN-OP mem2* (∈) (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
shows (*gen-subseteq ball1 mem2*,(⊆)) ∈ ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs2* → *Id*
apply (*intro fun-relI*)
unfolding *gen-subseteq-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-subseteq*[*symmetric*])

apply *parametricity*
done

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*, $(=)$) $\in \langle Rk\rangle Rs1 \rightarrow \langle Rk\rangle Rs2 \rightarrow Id$
apply (*intro fun-relI*)
unfolding *gen-equal-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst set-eq-subset*)
apply (*parametricity*)
done

lemma *foldli-card-aux*: *distinct l* $\Longrightarrow$ *foldli l* (λ-. *True*)
(λ- *n*. *Suc n*) *n* = *n* + *card* (*set l*)
apply (*induct l arbitrary*: *n*)
apply *auto*
done

lemma *foldli-card*: *det-fold-set X* (λ-. *True*) (λ- *n*. *Suc n*) *0 card*
apply *rule* **using** *foldli-card-aux*[**where** *n=0*] **by** *simp*

definition *gen-card* **where**
gen-card it s $\equiv$ *it s* (λx. *True*) (λ- *n*. *Suc n*) *0*

lemma *gen-card*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-card* (λx. *foldli* (*it x*)),*card*) $\in \langle Rk\rangle Rs \rightarrow Id$
apply (*intro fun-relI*)
unfolding *gen-card-def*
apply (*rule det-fold-set*[*OF foldli-card IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *fold-set*: *fold Set.insert l s* = *s* $\cup$ *set l*
by (*induct l arbitrary*: *s*) *auto*

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

lemma *gen-set*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *EMPTY*:

GEN-OP emp {} ($\langle Rk\rangle Rs$)
assumes *INS*:
GEN-OP ins Set.insert ($Rk\to\langle Rk\rangle Rs\to\langle Rk\rangle Rs$)
shows (*gen-set emp ins*,*set*)$\in\langle Rk\rangle$*list-rel*$\to\langle Rk\rangle Rs$
apply (*intro fun-relI*)
unfolding *gen-set-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst fold-set*[**where** *s*={},*simplified*,*symmetric*])
apply *parametricity*
done

lemma *ball-isEmpty*: *op-set-isEmpty s* = ($\forall x\in s.$ *False*)
by *auto*

definition *gen-isEmpty* :: ($'s \Rightarrow ('k \Rightarrow bool) \Rightarrow bool) \Rightarrow 's \Rightarrow bool$ **where**
gen-isEmpty ball s $\equiv$ *ball s* (λ-. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball Ball* ($\langle Rk\rangle Rs \to (Rk\to Id) \to Id$)
shows (*gen-isEmpty ball*,*op-set-isEmpty*) $\in \langle Rk\rangle Rs \to Id$
apply (*intro fun-relI*)
unfolding *gen-isEmpty-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-isEmpty*)
apply *parametricity*
done

lemma *foldli-size-abort-aux*:
$[\![n0\le m;$ *distinct l*$]\!] \Longrightarrow$
foldli l ($\lambda n.\ n<m$) (λ- *n. Suc n*) *n0* = *min m* (*n0* + *card* (*set l*))
apply (*induct l arbitrary*: *n0*)
apply *auto*
done

lemma *foldli-size-abort*:
det-fold-set X ($\lambda n.\ n<m$) (λ- *n. Suc n*) *0* (*op-set-size-abort m*)
apply *rule*
using *foldli-size-abort-aux*[**where** *?n0.0=0*]
by *simp*

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

lemma *gen-size-abort*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)

shows (*gen-size-abort* ($\lambda x.$ *foldli* (*it x*)),*op-set-size-abort*)
$\in$ *Id* $\rightarrow$ $\langle Rk\rangle Rs$ $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-size-abort-def*
apply (*rule det-fold-set*[*OF foldli-size-abort IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *size-abort-isSng*: *op-set-isSng s* $\longleftrightarrow$ *op-set-size-abort 2 s* = *1*
by *auto*

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-set-size-abort* (*Id* $\rightarrow$ ($\langle Rk\rangle Rs$) $\rightarrow$ *Id*)
shows (*gen-isSng sizea*,*op-set-isSng*) $\in$ $\langle Rk\rangle Rs$ $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-isSng-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst size-abort-isSng*)
apply *parametricity*
done

lemma *foldli-disjoint-aux*:
foldli l1 ($\lambda x.$ *x*) (λx -. $\neg x \in s2$) *b* $\longleftrightarrow$ *b* $\wedge$ *op-set-disjoint* (*set l1*) *s2*
by (*induct l1 arbitrary*: *b*) *auto*

lemma *foldli-disjoint*:
det-fold-set X ($\lambda x.$ *x*) (λx -. $\neg x \in s2$) *True* ($\lambda s1.$ *op-set-disjoint s1 s2*)
apply *rule* **using** *foldli-disjoint-aux*[**where** *b*=*True*] **by** *simp*

definition *gen-disjoint*
:: - $\Rightarrow$ (*'k*$\Rightarrow$*'s2*$\Rightarrow$*bool*) $\Rightarrow$ -
where *gen-disjoint it1 mem2 s1 s2*
$\equiv$ *it1 s1* ($\lambda x.$ *x*) (λx -. $\neg$*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*
apply (*intro fun-relI*)
unfolding *gen-disjoint-def*
apply (*rule det-fold-set*[*OF foldli-disjoint IT*[*unfolded autoref-tag-defs*]])
using *MEM* **unfolding** *autoref-tag-defs*

apply (*parametricity*)+
done

lemma *foldli-filter-aux*:
foldli l (λ-. *True*) (λ*x s. if P x then insert x s else s*) *s0*
= *s0* ∪ *op-set-filter P* (*set l*)
by (*induct l arbitrary*: *s0*) *auto*

lemma *foldli-filter*:
det-fold-set X (λ-. *True*) (λ*x s. if P x then insert x s else s*) {}
(*op-set-filter P*)
apply *rule* **using** *foldli-filter-aux*[**where** *?s0.0*={}] **by** *simp*

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

lemma *gen-filter*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *INS*:
GEN-OP ins2 Set.insert (*Rk*→⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (⟨*Rk*⟩*Rs2*)
shows (*gen-filter* (λ*x. foldli* (*it1 x*)) *empty2 ins2*,*op-set-filter*)
∈ (*Rk*→*Id*) → (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*)
apply (*intro fun-relI*)
unfolding *gen-filter-def*
apply (*rule det-fold-set*[*OF foldli-filter IT*[*unfolded autoref-tag-defs*]])
using *INS EMPTY* **unfolding** *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-image-aux*:
foldli l (λ-. *True*) (λ*x s. insert* (*f x*) *s*) *s0*
= *s0* ∪ *f*‘(*set l*)
by (*induct l arbitrary*: *s0*) *auto*

lemma *foldli-image*:
det-fold-set X (λ-. *True*) (λ*x s. insert* (*f x*) *s*) {}
((‘) *f*)
apply *rule* **using** *foldli-image-aux*[**where** *?s0.0*={}] **by** *simp*

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

lemma *gen-image*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*

```
  assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 it1)
  assumes INS:
    GEN-OP ins2 Set.insert (Rk'→⟨Rk'⟩Rs2→⟨Rk'⟩Rs2)
  assumes EMPTY:
    GEN-OP empty2 {} (⟨Rk'⟩Rs2)
  shows (gen-image (λx. foldli (it1 x)) empty2 ins2,(`))
  ∈ (Rk→Rk') → (⟨Rk⟩Rs1) → (⟨Rk'⟩Rs2)
  apply (intro fun-relI)
  unfolding gen-image-def
  apply (rule det-fold-set[OF foldli-image IT[unfolded autoref-tag-defs]])
  using INS EMPTY unfolding autoref-tag-defs
  apply (parametricity)+
  done


lemma foldli-pick:
  assumes l≠[]
  obtains x where x∈set l
  and (foldli l (case-option True (λ-. False)) (λx -. Some x) None) = Some x
  using assms by (cases l) auto

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

context begin interpretation autoref-syn .
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'≠{})
  assumes SREF: (s,s')∈⟨Rk⟩Rs
  shows (RETURN (gen-pick (λx. foldli (it x)) s),
    (OP op-set-pick ::: ⟨Rk⟩Rs→⟨Rk⟩nres-rel)$s')∈⟨Rk⟩nres-rel
proof –
  obtain tsl' where
    [param]: (it s,tsl') ∈ ⟨Rk⟩list-rel
    and IT': RETURN tsl' ≤ it-to-sorted-list (λ- -. True) s'
    using IT[unfolded autoref-tag-defs is-set-to-list-def] SREF
    by (rule is-set-to-sorted-listE)

  from IT' NE have tsl'≠[] and [simp]: s'=set tsl'
    unfolding it-to-sorted-list-def by simp-all
  then obtain x where x∈s' and
    (foldli tsl' (case-option True (λ-. False)) (λx -. Some x) None) = Some x
    (is ?fld = -)
    by (blast elim: foldli-pick)
  moreover
  have (RETURN (gen-pick (λx. foldli (it x)) s), RETURN (the ?fld))
```

```
      ∈ ⟨Rk⟩nres-rel
      unfolding gen-pick-def
      apply (parametricity add: the-paramR)
      using ‹?fld = Some x›
      by simp
    ultimately show ?thesis
      unfolding autoref-tag-defs
      apply −
      apply (drule nres-relD)
      apply (rule nres-relI)
      apply (erule ref-two-step)
      by simp
  qed
end

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


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

    obtain l' where
      IT1L: (it1 s,l')∈⟨Rk⟩list-rel
      and SL: s' = set l'
      apply (rule
        is-set-to-sorted-listE[OF IT1[unfolded is-set-to-list-def] SR])
      by (auto simp: it-to-sorted-list-def)
```

```
  show ?thesis
    unfolding SL
    using IT1L S0R
  proof (induct arbitrary: s0 s0' rule: list-rel-induct)
    case Nil thus ?case by simp
  next
    case (Cons x x' l l')

    obtain l2' where
      IT2L: (it2 (f x),l2')∈⟨Rl⟩list-rel
      and FXL: f' x' = set l2'
      apply (rule
        is-set-to-sorted-listE[
          OF IT2[unfolded is-set-to-list-def], of f x   f' x'
        ])
      apply (parametricity add: Cons.hyps(1) FR)
      by (auto simp: it-to-sorted-list-def)

    have (foldli (it2 (f x)) (λ-. True) (λy. insX (x, y)) s0,
      s0' ∪ {x'}×f' x') ∈ ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
      unfolding FXL
      using IT2L ‹(s0, s0') ∈ ⟨⟨Rk, Rl⟩prod-rel⟩Rs3›
      apply (induct  arbitrary: s0 s0' rule: list-rel-induct)
      apply simp
      apply simp
      apply (subst Un-insert-left[symmetric])
      apply (rprems)
      apply (parametricity add: INS ‹(x,x')∈Rk›)
      done

    show ?case
      apply simp
      apply (subst S)
      apply (subst Un-assoc[symmetric])
      apply (rule Cons.hyps)
      apply fact
      done
  qed
qed


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

```
      (⟨Rk,Rl⟩prod-rel→⟨⟨Rk,Rl⟩prod-rel⟩Rs3→⟨⟨Rk,Rl⟩prod-rel⟩Rs3)
    shows (gen-Sigma (λx. foldli (it1 x)) (λx. foldli (it2 x)) empX insX,Sigma)
      ∈ (⟨Rk⟩Rs1) → (Rk → ⟨Rl⟩Rs2) → ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
    apply (intro fun-relI)
    unfolding gen-Sigma-def
    using foldli-Sigma-aux[OF
      IT1[unfolded autoref-tag-defs]
      IT2[unfolded autoref-tag-defs]
      INS[unfolded autoref-tag-defs]
      EMPTY[unfolded autoref-tag-defs]
    ]
    by simp

lemma gen-cart:
  assumes PRIO-TAG-GEN-ALGO
  assumes [param]: (sigma, Sigma) ∈ (⟨Rx⟩Rsx → (Rx → ⟨Ry⟩Rsy) → ⟨Rx ×r Ry⟩Rsp)
  shows (λx y. sigma x (λ-. y), op-set-cart) ∈ ⟨Rx⟩Rsx → ⟨Ry⟩Rsy → ⟨Rx ×r Ry⟩Rsp
  unfolding op-set-cart-def[abs-def]
  by parametricity
lemmas [autoref-rules] = gen-cart[OF - GEN-OP-D]


context begin interpretation autoref-syn .

  lemma op-set-to-sorted-list-autoref[autoref-rules]:
    assumes SIDE-GEN-ALGO (is-set-to-sorted-list ordR Rk Rs tsl)
    shows (λsi. RETURN (tsl si),  OP (op-set-to-sorted-list ordR))
      ∈ ⟨Rk⟩Rs → ⟨⟨Rk⟩list-rel⟩nres-rel
    using assms
    apply (intro fun-relI nres-relI)
    apply simp
    apply (rule RETURN-SPEC-refine)
    apply (auto simp: is-set-to-sorted-list-def it-to-sorted-list-def)
    done

  lemma op-set-to-list-autoref[autoref-rules]:
    assumes SIDE-GEN-ALGO (is-set-to-sorted-list ordR Rk Rs tsl)
    shows (λsi. RETURN (tsl si), op-set-to-list)
      ∈ ⟨Rk⟩Rs → ⟨⟨Rk⟩list-rel⟩nres-rel
    using assms
    apply (intro fun-relI nres-relI)
    apply simp
    apply (rule RETURN-SPEC-refine)
    apply (auto simp: is-set-to-sorted-list-def it-to-sorted-list-def)
    done

end
```

```
lemma foldli-Union: det-fold-set X (λ-. True) (∪) {} Union
proof (rule, goal-cases)
  case (1 l)
  have ∀ a. foldli l (λ-. True) (∪) a = a ∪ ⋃(set l)
    by (induct l) auto
  thus ?case by auto
qed

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

lemma gen-Union[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes EMP: GEN-OP emp {} (⟨Rk⟩Rs3)
  assumes UN: GEN-OP un (∪) (⟨Rk⟩Rs2→⟨Rk⟩Rs3→⟨Rk⟩Rs3)
  assumes IT: SIDE-GEN-ALGO (is-set-to-list (⟨Rk⟩Rs2) Rs1 tsl)
 shows (gen-Union (λx. foldli (tsl x)) emp un,Union) ∈ ⟨⟨Rk⟩Rs2⟩Rs1 → ⟨Rk⟩Rs3
  apply (intro fun-relI)
  unfolding gen-Union-def
  apply (rule det-fold-set[OF
    foldli-Union IT[unfolded autoref-tag-defs]])
  using EMP UN
  unfolding autoref-tag-defs
  apply (parametricity)+
  done

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

lemma atLeastLessThan-impl-correct:
  atLeastLessThan-impl a b ≤ SPEC (λr. r = {a..<b::nat})
  unfolding atLeastLessThan-impl-def
  apply (refine-rcg refine-vcg WHILET-rule[where
    I = λ(i,r). r = {a..<i} ∧ a≤i ∧ ((a<b ⟶ i≤b) ∧ (¬a<b ⟶ i=a))
    and R = measure (λ(i,-). b − i)
    ])
  by auto

schematic-goal atLeastLessThan-code-aux:
  notes [autoref-rules] = IdI[of a] IdI[of b]
  assumes [autoref-rules]: (emp,{})∈Rs
  assumes [autoref-rules]: (ins,insert)∈nat-rel → Rs → Rs
  shows (?c, atLeastLessThan-impl)
```

$\in$ *nat-rel* $\rightarrow$ *nat-rel* $\rightarrow$ $\langle Rs\rangle$*nres-rel*
unfolding *atLeastLessThan-impl-def*[*abs-def*]
apply (*autoref* (*keep-goal*))
done
concrete-definition *atLeastLessThan-code* **uses** *atLeastLessThan-code-aux*

schematic-goal *atLeastLessThan-tr-aux*:
RETURN ?c $\leq$ *atLeastLessThan-code emp ins a b*
unfolding *atLeastLessThan-code-def*
by (*refine-transfer* (*post*))
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*)
$\in$ *nat-rel* $\rightarrow$ *nat-rel* $\rightarrow$ *Rs*
proof (*intro fun-relI*, *simp*)
fix *a b*
from *assms* **have** *GEN*:
(*emp*,{})$\in$*Rs* (*ins*,*insert*)$\in$*nat-rel* $\rightarrow$ *Rs* $\rightarrow$ *Rs*
by *auto*

note *atLeastLessThan-tr.refine*[*of emp ins a b*]
also note
atLeastLessThan-code.refine[*OF GEN*,*param-fo*, *OF IdI IdI*, *THEN nres-relD*]
also note *atLeastLessThan-impl-correct*
finally show (*atLeastLessThan-tr emp ins a b*, {*a*..<*b*}) $\in$ *Rs*
by (*auto simp*: *pw-le-iff refine-pw-simps*)
qed

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'*)
apply (*rule ccontr*)
apply (*drule not-distinct-decomp*)
using *assms*
apply (*clarsimp elim*!: *map-eq-appendE*)
by (*metis* (*opaque-lifting*, *no-types*) *insert-iff map-to-set-inj*)

```
lemma foldli-add: det-fold-map X
  (λ-. True) (λ(k,v) m. op-map-update k v m) m ((++) m)
proof (rule, goal-cases)
  case (1 l) thus ?case
    apply (induct l arbitrary: m)
    apply (auto simp: map-of-distinct-upd[symmetric])
    done
qed

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

lemma gen-add[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
 assumes UPD: GEN-OP ins op-map-update (Rk→Rv→⟨Rk,Rv⟩Rs1→⟨Rk,Rv⟩Rs1)
  assumes IT: SIDE-GEN-ALGO (is-map-to-list Rk Rv Rs2 tsl)
  shows (gen-add (foldli o tsl) ins,(++))
    ∈ (⟨Rk,Rv⟩Rs1) → (⟨Rk,Rv⟩Rs2) → (⟨Rk,Rv⟩Rs1)
  apply (intro fun-relI)
  unfolding gen-add-def comp-def
  apply (rule det-fold-map[OF foldli-add IT[unfolded autoref-tag-defs]])
  apply (parametricity add: UPD[unfolded autoref-tag-defs])+
  done

lemma foldli-restrict: det-fold-map X (λ-. True)
  (λ(k,v) m. if P (k,v) then op-map-update k v m else m) Map.empty
    (op-map-restrict P ) (is det-fold-map - - ?f - -)
proof −
  {
    fix l m
    have distinct (map fst l) ⟹
      foldli l (λ-. True) ?f m = m ++ op-map-restrict P (map-of l)
    proof (induction l arbitrary: m)
      case Nil thus ?case by simp
    next
      case (Cons kv l)
      obtain k v where [simp]: kv = (k,v) by fastforce
      from Cons.prems have
        DL: distinct (map fst l) and KNI: k ∉ set (map fst l)
        by auto

      show ?case proof (cases P (k,v))
        case [simp]: True
        have foldli (kv#l) (λ-. True) ?f m = foldli l (λ-. True) ?f (m(k↦v))
          by simp
```

```
      also from Cons.IH[OF DL] have
        ... = m(k↦v) ++ op-map-restrict P (map-of l) .
      also have ... = m ++ op-map-restrict P (map-of (kv#l))
        using KNI
        by (auto
          split: option.splits
          intro!: ext
          simp: Map.restrict-map-def Map.map-add-def
          simp: map-of-eq-None-iff[symmetric])
      finally show ?thesis .
    next
      case [simp]: False
      have foldli (kv#l) (λ-. True) ?f m = foldli l (λ-. True) ?f m
        by simp
      also from Cons.IH[OF DL] have
        ... = m ++ op-map-restrict P (map-of l) .
      also have ... = m ++ op-map-restrict P (map-of (kv#l))
        using KNI
        by (auto
          intro!: ext
          simp: Map.restrict-map-def Map.map-add-def
          simp: map-of-eq-None-iff[symmetric]
        )
      finally show ?thesis .
    qed
  qed
 }
 from this[of - Map.empty] show ?thesis
   by (auto intro!: det-fold-mapI)
qed

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

lemma gen-restrict[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT: SIDE-GEN-ALGO (is-map-to-list Rk Rv Rs1 tsl)
  assumes INS:
    GEN-OP upd op-map-update (Rk→Rv→⟨Rk,Rv⟩Rs2→⟨Rk,Rv⟩Rs2)
  assumes EMPTY:
    GEN-OP emp Map.empty (⟨Rk,Rv⟩Rs2)
  shows (gen-restrict (foldli o tsl) upd emp,op-map-restrict)
  ∈ (⟨Rk,Rv⟩prod-rel → Id) → (⟨Rk,Rv⟩Rs1) → (⟨Rk,Rv⟩Rs2)
  apply (intro fun-relI)
  unfolding gen-restrict-def comp-def
  apply (rule det-fold-map[OF foldli-restrict IT[unfolded autoref-tag-defs]])
  using INS EMPTY unfolding autoref-tag-defs
  apply (parametricity)+
```

done

lemma *fold-map-of*:
fold (λ(*k*,*v*) *s*. *op-map-update k v s*) (*rev l*) *Map.empty* = *map-of l*
proof –
{
fix *m*
have *fold* (λ(*k*,*v*) *s*. *s*(*k*↦*v*)) (*rev l*) *m* = *m* ++ *map-of l*
apply (*induct l arbitrary*: *m*)
apply *auto*
done
} **thus** *?thesis* **by** *simp*
qed

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

lemma *gen-map-of*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *UPD*: *GEN-OP upd op-map-update* (*Rk*→*Rv*→⟨*Rk*,*Rv*⟩*Rm*→⟨*Rk*,*Rv*⟩*Rm*)
assumes *EMPTY*: *GEN-OP emp Map.empty* (⟨*Rk*,*Rv*⟩*Rm*)
shows (*gen-map-of emp upd*,*map-of*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨*Rk*,*Rv*⟩*Rm*
using *assms*
apply (*intro fun-relI*)
unfolding *gen-map-of-def*[*abs-def*]
unfolding *autoref-tag-defs*
apply (*subst fold-map-of*[*symmetric*])
apply *parametricity*
done

lemma *foldli-ball-aux*:
distinct (*map fst l*) ⟹ *foldli l* (λ*x*. *x*) (λ*x* -. *P x*) *b*
⟷ *b* ∧ *op-map-ball* (*map-of l*) *P*
apply (*induct l arbitrary*: *b*)
apply *simp*
apply (*force simp*: *map-to-set-map-of image-def*)
done

lemma *foldli-ball*:
det-fold-map X (λ*x*. *x*) (λ*x* -. *P x*) *True* (λ*m*. *op-map-ball m P*)
apply *rule*
using *foldli-ball-aux*[**where** *b*=*True*] **by** *auto*

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

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*

```
  assumes IT: SIDE-GEN-ALGO (is-map-to-list Rk Rv Rm tsl)
  shows (gen-ball (foldli o tsl),op-map-ball)
  ∈ ⟨Rk,Rv⟩Rm → (⟨Rk,Rv⟩prod-rel → Id) → Id
  apply (intro fun-relI)
  unfolding gen-ball-def comp-def
  apply (rule det-fold-map[OF foldli-ball IT[unfolded autoref-tag-defs]])
  apply (parametricity)+
  done

lemma foldli-bex-aux:
  distinct (map fst l) ⟹ foldli l (λx. ¬x) (λx -. P x) b
  ⟷ b ∨ op-map-bex (map-of l) P
  apply (induct l arbitrary: b)
  apply simp
  apply (force simp: map-to-set-map-of image-def)
  done

lemma foldli-bex:
  det-fold-map X (λx. ¬x) (λx -. P x) False (λm. op-map-bex m P)
  apply rule
  using foldli-bex-aux[where b=False] by auto

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
  apply (intro fun-relI)
  unfolding gen-bex-def comp-def
  apply (rule det-fold-map[OF foldli-bex IT[unfolded autoref-tag-defs]])
  apply (parametricity)+
  done

lemma ball-isEmpty: op-map-isEmpty m = op-map-ball m (λ-. False)
  apply (auto intro!: ext)
  by (metis map-to-set-simps(7) option.exhaust)

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
  apply (intro fun-relI)
```

unfolding *gen-isEmpty-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-isEmpty*)
apply *parametricity+*
done

lemma *foldli-size-aux*: *distinct* (*map fst l*)
$\Longrightarrow$ *foldli l* (λ-. *True*) (λ- *n*. *Suc n*) *n* = *n* + *op-map-size* (*map-of l*)
apply (*induct l arbitrary*: *n*)
apply (*auto simp*: *dom-map-of-conv-image-fst*)
done

lemma *foldli-size*: *det-fold-map X* (λ-. *True*) (λ- *n*. *Suc n*) *0 op-map-size*
apply *rule*
using *foldli-size-aux*[**where** *n=0*] **by** *simp*

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*
apply (*intro fun-relI*)
unfolding *gen-size-def comp-def*
apply (*rule det-fold-map*[*OF foldli-size IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *foldli-size-abort-aux*:
⟦*n0*≤*m*; *distinct* (*map fst l*)⟧ $\Longrightarrow$
foldli l (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *n0* = *min m* (*n0* + *card* (*dom* (*map-of l*)))
apply (*induct l arbitrary*: *n0*)
apply (*auto simp*: *dom-map-of-conv-image-fst*)
done

lemma *foldli-size-abort*:
det-fold-map X (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0* (*op-map-size-abort m*)
apply *rule*
using *foldli-size-abort-aux*[**where** *?n0.0=0*]
by *simp*

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

lemma *gen-size-abort*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)

shows (*gen-size-abort* (*foldli o tsl*),*op-map-size-abort*)
 $\in$ *Id* $\rightarrow$ $\langle$*Rk*,*Rv*$\rangle$*Rm* $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-size-abort-def comp-def*
apply (*rule det-fold-map*[*OF foldli-size-abort*
 IT[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *size-abort-isSng*: *op-map-isSng s* $\longleftrightarrow$ *op-map-size-abort 2 s* = *1*
by (*auto simp*: *dom-eq-singleton-conv min-def dest*!: *card-eq-SucD*)

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*
apply (*intro fun-relI*)
unfolding *gen-isSng-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst size-abort-isSng*)
apply *parametricity*
done

lemma *foldli-pick*:
assumes *l*$\neq$[]
obtains *k v* **where** (*k*,*v*)$\in$*set l*
and (*foldli l* (*case-option True* (λ-. *False*)) (λ*x* -. *Some x*) *None*)
 = *Some* (*k*,*v*)
using *assms* **by** (*cases l*) *auto*

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

context begin interpretation *autoref-syn* **.**

lemma *gen-pick*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm it*)

```
    assumes NE: SIDE-PRECOND (m'≠Map.empty)
    assumes SREF: (m,m')∈⟨Rk,Rv⟩Rm
    shows (RETURN (gen-pick (λx. foldli (it x)) m),
      (OP op-map-pick ::: ⟨Rk,Rv⟩Rm→⟨Rk×_r Rv⟩nres-rel)$m')∈⟨Rk×_r Rv⟩nres-rel
  proof −
    thm is-map-to-list-def is-map-to-sorted-listE

    obtain tsl' where
      [param]: (it m,tsl') ∈ ⟨Rk×_r Rv⟩list-rel
      and IT': RETURN tsl' ≤ it-to-sorted-list (λ- -. True) (map-to-set m')
      using IT[unfolded autoref-tag-defs is-map-to-list-def] SREF
      by (auto intro: is-map-to-sorted-listE)

    from IT' NE have tsl'≠[] and [simp]: m'=map-of tsl'
      and DIS': distinct (map fst tsl')
      unfolding it-to-sorted-list-def
      apply simp-all
      apply (metis empty-set map-to-set-empty-iff(1))
      apply (metis map-of-map-to-set map-to-set-distinct-conv)
      apply (metis map-to-set-distinct-conv)
      done

    then obtain k v where m' k = Some v and
      (foldli tsl' (case-option True (λ-. False)) (λx -. Some x) None)
        = Some (k,v)
      (is ?fld = -)
      by (cases rule: foldli-pick) auto
    moreover
    have (RETURN (gen-pick (λx. foldli (it x)) m), RETURN (the ?fld))
      ∈ ⟨Rk×_r Rv⟩nres-rel
      unfolding gen-pick-def
      apply (parametricity add: the-paramR)
      using ‹?fld = Some (k,v)›
      by simp
    ultimately show ?thesis
      unfolding autoref-tag-defs
      apply −
      apply (drule nres-relD)
      apply (rule nres-relI)
      apply (erule ref-two-step)
      by simp
  qed
end


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

```
  }

context begin interpretation autoref-syn .
  lemma gen-map-pick-remove
    [unfolded gen-map-pick-remove-def, autoref-rules-raw]:
    assumes PRIO-TAG-GEN-ALGO
    assumes PICK: SIDE-GEN-OP (
      (pick m,
      (OP op-map-pick ::: ⟨Rk,Rv⟩Rm → ⟨Rk×_r Rv⟩nres-rel)$m′) ∈
      ⟨Rk×_r Rv⟩nres-rel)
   assumes DEL: GEN-OP del op-map-delete (Rk → ⟨Rk,Rv⟩Rm → ⟨Rk,Rv⟩Rm)
    assumes [param]: (m,m′)∈⟨Rk,Rv⟩Rm
    shows (gen-map-pick-remove pick del m,
      (OP op-map-pick-remove
        ::: ⟨Rk,Rv⟩Rm → ⟨(Rk×_r Rv) ×_r ⟨Rk,Rv⟩Rm⟩nres-rel)$m′)
      ∈ ⟨(Rk×_r Rv) ×_r ⟨Rk,Rv⟩Rm⟩nres-rel
  proof −
    note [param] =
      PICK[unfolded autoref-tag-defs]
      DEL[unfolded autoref-tag-defs]

    have (gen-map-pick-remove pick del m,
      do {
        (k,v)←op-map-pick m′;
        let m′ = op-map-delete k m′;
        RETURN ((k,v),m′)
      }) ∈ ⟨(Rk×_r Rv) ×_r ⟨Rk,Rv⟩Rm⟩nres-rel (is (-,?h):-)
      unfolding gen-map-pick-remove-def[abs-def]
      apply parametricity
      done
    also have ?h = op-map-pick-remove m′
      by (auto simp add: pw-eq-iff refine-pw-simps)
    finally show ?thesis by simp
  qed
end


end
```

2.2.3 Generic Map To Set Converter

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

lemma *map-fst-unit-distinct-eq*[*simp*]:
fixes *l* :: (*'k*×*unit*) *list*
shows *distinct* (*map fst l*) ⟷ *distinct l*
by (*induct l*) *auto*

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

lemma *map2set-rel-def*: ⟨*Rk*⟩(*map2set-rel R*)
= ⟨*Rk*,*Id*::(*unit*×-) *set*⟩*R O* {(*m*,*dom m*)| *m*. *True*}
unfolding *map2set-rel-def-internal*[*abs-def*] **by** (*simp add*: *relAPP-def*)

lemma *map2set-relI*:
assumes (*s*,*m'*)∈⟨*Rk*,*Id*⟩*R* **and** *s'*=*dom m'*
shows (*s*,*s'*)∈⟨*Rk*⟩*map2set-rel R*
using *assms* **unfolding** *map2set-rel-def* **by** *blast*

lemma *map2set-relE*:
assumes (*s*,*s'*)∈⟨*Rk*⟩*map2set-rel R*
obtains *m'* **where** (*s*,*m'*)∈⟨*Rk*,*Id*⟩*R* **and** *s'*=*dom m'*
using *assms* **unfolding** *map2set-rel-def* **by** *blast*

lemma *map2set-rel-sv*[*relator-props*]:
single-valued (⟨*Rk*,*Id*⟩*Rm*) ⟹ *single-valued* (⟨*Rk*⟩*map2set-rel Rm*)
unfolding *map2set-rel-def*
by (*auto intro*: *single-valuedI dest*: *single-valuedD*)

lemma *map2set-empty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP e op-map-empty* (⟨*Rk*,*Id*⟩*R*)
shows (*e*,{})∈⟨*Rk*⟩*map2set-rel R*
using *assms*
unfolding *map2set-rel-def*
by *auto*

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

definition *map2set-insert i k s* ≡ *i k* () *s*
lemma *map2set-insert*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP i op-map-update* (*Rk* → *Id* → ⟨*Rk*,*Id*⟩*R* → ⟨*Rk*,*Id*⟩*R*)

shows
  ($map2set\text{-}insert\ i, Set.insert) \in Rk \to \langle Rk \rangle map2set\text{-}rel\ R \to \langle Rk \rangle map2set\text{-}rel\ R$
using *assms*
unfolding *map2set-rel-def map2set-insert-def*[*abs-def*]
by (*force dest*: *fun-relD*)

definition $map2set\text{-}memb\ l\ k\ s \equiv case\ l\ k\ s\ of\ None \Rightarrow False \mid Some\ \text{-} \Rightarrow True$
lemma *map2set-memb*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes $GEN\text{-}OP\ l\ op\text{-}map\text{-}lookup\ (Rk \to \langle Rk,Id \rangle R \to \langle Id \rangle option\text{-}rel)$
shows $(map2set\text{-}memb\ l\ ,(\in))$
  $\in Rk \to \langle Rk \rangle map2set\text{-}rel\ R \to Id$
using *assms*
unfolding *map2set-rel-def map2set-memb-def*[*abs-def*]
by (*force dest*: *fun-relD split*: *option.splits*)

lemma *map2set-delete*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes $GEN\text{-}OP\ d\ op\text{-}map\text{-}delete\ (Rk \to \langle Rk,Id \rangle R \to \langle Rk,Id \rangle R)$
shows $(d, op\text{-}set\text{-}delete) \in Rk \to \langle Rk \rangle map2set\text{-}rel\ R \to \langle Rk \rangle map2set\text{-}rel\ R$
using *assms*
unfolding *map2set-rel-def*
by (*force dest*: *fun-relD*)

lemma *map2set-to-sorted-list*[*autoref-ga-rules*]:
fixes $it :: 'm \Rightarrow ('k \times unit)\ list$
assumes *A*: $GEN\text{-}ALGO\text{-}tag\ (is\text{-}map\text{-}to\text{-}sorted\text{-}list\ ordR\ Rk\ Id\ R\ it)$
shows $is\text{-}set\text{-}to\text{-}sorted\text{-}list\ ordR\ Rk\ (map2set\text{-}rel\ R)$
  $(it\text{-}to\text{-}list\ (map\text{-}iterator\text{-}dom\ o\ (foldli\ o\ it)))$
proof −
{
  fix $l{::}('k \times unit)\ list$
  have $\bigwedge l0.\ foldli\ l\ (\lambda\text{-}.\ True)\ (\lambda x\ \sigma.\ \sigma\ @\ [fst\ x])\ l0 = l0 @ map\ fst\ l$
    by (*induct l*) *auto*
}
hence *S*: $it\text{-}to\text{-}list\ (map\text{-}iterator\text{-}dom\ o\ (foldli\ o\ it)) = map\ fst\ o\ it$
  unfolding *it-to-list-def*[*abs-def*] *map-iterator-dom-def*[*abs-def*]
    set-iterator-image-def set-iterator-image-filter-def
  by (*auto*)
show *?thesis*
  unfolding *S*
  using *assms*
  unfolding *is-map-to-sorted-list-def is-set-to-sorted-list-def*
  apply *clarsimp*
  apply (*erule map2set-relE*)
  apply (*drule spec*, *drule spec*)
  apply (*drule* (*1*) *mp*)
  apply (*elim exE conjE*)
  apply (*rule-tac* $x{=}map\ fst\ l'$ **in** *exI*)

apply (*rule conjI*)
apply *parametricity*

unfolding *it-to-sorted-list-def*
apply (*simp add*: *map-to-set-dom*)
apply (*simp add*: *sorted-wrt-map key-rel-def*[*abs-def*])
done
qed

lemma *map2set-to-list*[*autoref-ga-rules*]:
fixes *it* :: *'m* ⇒ (*'k*×*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*)))
using *assms* **unfolding** *is-set-to-list-def is-map-to-list-def*
by (*rule map2set-to-sorted-list*)

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

lemma *map2set-union*[*autoref-rules-raw*]:
assumes *MINOR-PRIO-TAG* (− *9*)
assumes *GEN-OP u* (++) (⟨*Rk*,*Id*⟩*R*→⟨*Rk*,*Id*⟩*R*→⟨*Rk*,*Id*⟩*R*)
shows (*u*,(∪))∈⟨*Rk*⟩*map2set-rel R*→⟨*Rk*⟩*map2set-rel R*→⟨*Rk*⟩*map2set-rel R*
using *assms*
unfolding *map2set-rel-def*
by (*force dest*: *fun-relD*)

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

lemma *cmp-lex-zip-unit*[*simp*]:
cmp-lex (*cmp-prod cmp cmp-unit*) (*map* (λ*k*. (*k*, ())) *l*)
(*map* (λ*k*. (*k*, ())) *m*) =
cmp-lex cmp l m
apply (*induct cmp l m rule*: *cmp-lex.induct*)
apply (*auto split*: *comp-res.split*)
done

lemma *cmp-img-zip-unit*[*simp*]:
cmp-img (λ*m*. *map* (λ*k*. (*k*,())) (*f m*)) (*cmp-lex* (*cmp-prod cmp1 cmp-unit*))
= *cmp-img f* (*cmp-lex cmp1*)
unfolding *cmp-img-def*[*abs-def*]
apply (*intro ext*)
apply *simp*
done

lemma *map2set-finite*[*relator-props*]:

assumes *finite-map-rel* (⟨*Rk*,*Id*⟩*R*)
shows *finite-set-rel* (⟨*Rk*⟩*map2set-rel R*)
using *assms*
unfolding *map2set-rel-def finite-set-rel-def finite-map-rel-def*
by *auto*

lemma *map2set-cmp*[*autoref-rules-raw*]:
assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmpk*)
assumes *MPAR*:
GEN-OP cmp (*cmp-map cmpk cmp-unit*) (⟨*Rk*,*Id*⟩*R* → ⟨*Rk*,*Id*⟩*R* → *Id*)
assumes *FIN*: *PREFER finite-map-rel* (⟨*Rk*, *Id*⟩*R*)
shows (*cmp*,*cmp-set cmpk*)∈⟨*Rk*⟩*map2set-rel R* → ⟨*Rk*⟩*map2set-rel R* → *Id*
proof −
interpret *linorder comp2le cmpk comp2lt cmpk*
using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)

show *?thesis*
using *MPAR*
unfolding *cmp-map-def cmp-set-def*
apply *simp*
apply *parametricity*
apply (*drule cmp-extend-paramD*)
apply (*insert FIN*, *fastforce simp add*: *finite-map-rel-def*) []
apply (*simp add*: *sorted-list-of-map-def*[*abs-def*])
apply (*auto simp*: *map2set-rel-def cmp-img-def*[*abs-def*] *dest*: *fun-relD*) []

apply (*insert map2set-finite*[*OF FIN*[*unfolded autoref-tag-defs*]],
fastforce simp add: *finite-set-rel-def*)
done
qed

end

2.2.4 Generic Compare Algorithms

theory *Gen-Comp*
imports
../Intf/Intf-Comp
Automatic-Refinement.Automatic-Refinement
HOL−Library.Product-Lexorder
begin

Order for Product

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

lemma *gen-prod-cmp-dflt*[*autoref-rules-raw*]:

assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP cmp1* (*dflt-cmp* (≤) (<)) (*R1* → *R1* → *Id*)
assumes *GEN-OP cmp2* (*dflt-cmp* (≤) (<)) (*R2* → *R2* → *Id*)
shows (*cmp-prod cmp1 cmp2*, *dflt-cmp* (≤) (<)) ∈
⟨*R1*,*R2*⟩*prod-rel* → ⟨*R1*,*R2*⟩*prod-rel* → *Id*
proof −
have *E*: *dflt-cmp* (≤) (<)
= *cmp-prod* (*dflt-cmp* (≤) (<)) (*dflt-cmp* (≤) (<))
by (*auto simp*: *dflt-cmp-def prod-less-def prod-le-def intro*!: *ext*)

show *?thesis*
using *assms*
unfolding *autoref-tag-defs E*
by *parametricity*
qed

end

2.3 Implementations

2.3.1 Stack by Array

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

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

term *Diff-Array.array-length*

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

definition *as-rel-def-internal*: *as-rel R* ≡ *br as-raw-α as-raw-invar O* ⟨*R*⟩*list-rel*
lemma *as-rel-def*: ⟨*R*⟩*as-rel* ≡ *br as-raw-α as-raw-invar O* ⟨*R*⟩*list-rel*
unfolding *as-rel-def-internal*[*abs-def*] **by** (*simp add*: *relAPP-def*)

lemma [*relator-props*]: *single-valued R* ⟹ *single-valued* (⟨*R*⟩*as-rel*)
unfolding *as-rel-def*
by *tagged-solver*

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

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

lemma *as-empty-refine*[*autoref-rules*]: (*as-empty* (),[]) ∈ ⟨*R*⟩*as-rel*
  unfolding *as-rel-def as-empty-def br-def*
  unfolding *as-raw-α-def as-raw-invar-def*
  by *auto*

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

lemma *as-push-refine*[*autoref-rules*]:
  (*as-push*,*op-list-append-elem*) ∈ ⟨*R*⟩*as-rel* → *R* → ⟨*R*⟩*as-rel*
  apply (*intro fun-relI*)
  apply (*simp add*: *as-push-def op-list-append-elem-def as-rel-def br-def*
    as-raw-α-def as-raw-invar-def)
  apply *clarsimp*
  apply *safe*
  apply (*rule*)
  apply *auto* []
  apply (*clarsimp simp*: *array-length-list*) []
  apply *parametricity*

  apply *rule*
  apply *auto* []
  apply (*auto simp*: *take-Suc-conv-app-nth array-length-list list-update-append*) []
  apply *parametricity*
  done

term *array-shrink*

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

lemma *as-shrink-id-refine*: (*as-shrink*,*id*) ∈ ⟨*R*⟩*as-rel* → ⟨*R*⟩*as-rel*
  apply (*intro fun-relI*)
  apply (*simp add*: *as-shrink-def as-rel-def br-def*
    as-raw-α-def as-raw-invar-def Let-def)
  apply *clarsimp*
  apply *safe*

```
  apply (rule)
  apply (auto simp: array-length-list)
  done

lemma as-shrinkI:
  assumes [param]: (s,a)∈⟨R⟩as-rel
  shows (as-shrink s,a)∈⟨R⟩as-rel
  apply (subst id-apply[of a,symmetric])
  apply (parametricity add: as-shrink-id-refine)
  done

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

lemma as-pop-refine[autoref-rules]: (as-pop,butlast) ∈ ⟨R⟩as-rel → ⟨R⟩as-rel
  apply (intro fun-relI)
  apply (clarsimp simp add: as-pop-def split: prod.split)
  apply (rule as-shrinkI)

  apply (simp add: as-pop-def as-rel-def br-def
    as-raw-α-def as-raw-invar-def Let-def)
  apply clarsimp

  apply rule
  apply (auto simp: array-length-list) []
  apply (clarsimp simp: array-length-list take-minus-one-conv-butlast) []
  apply parametricity
  done

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

lemma as-get-refine:
  assumes 1: i'<length l
  assumes 2: (a,l)∈⟨R⟩as-rel
  assumes 3[param]: (i,i')∈nat-rel
  shows (as-get a i,l!i')∈R
  using 2
  apply (clarsimp
    simp add: as-get-def as-rel-def br-def as-raw-α-def as-raw-invar-def
    split: prod.split)
  apply (rename-tac aa bb)
  apply (case-tac aa, simp)
proof −
  fix n cl
  assume TKR[param]: (take n cl, l) ∈ ⟨R⟩list-rel

  have (take n cl!i, l!i')∈R
    by parametricity (rule 1)
  also have take n cl!i = cl!i
```

```
    using 1 3 list-rel-imp-same-length[OF TKR]
    by simp
  finally show (cl!i,l!i')∈R .
qed

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

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

lemma as-set-refine[autoref-rules]:
  (as-set,list-update)∈⟨R⟩as-rel → nat-rel → R → ⟨R⟩as-rel
  apply (intro fun-relI)
  apply (clarsimp
    simp: as-set-def as-rel-def br-def as-raw-α-def as-raw-invar-def
    split: prod.split)
  apply rule
  apply auto []
  apply parametricity
  by simp

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

lemma as-length-refine[autoref-rules]:
  (as-length,length) ∈ ⟨R⟩as-rel → nat-rel
  by (auto
    simp: as-length-def as-rel-def br-def as-raw-α-def as-raw-invar-def
      array-length-list
    dest!: list-rel-imp-same-length
  )

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

lemma as-top-code[code]: as-top s = (let (a,n)=s in array-get a (n − 1))
  unfolding as-top-def as-get-def as-length-def
  by (auto split: prod.split)

lemma as-top-refine: ⟦l≠[]; (s,l)∈⟨R⟩as-rel⟧ ⟹ (as-top s,last l)∈R
  unfolding as-top-def
  apply (simp add: last-conv-nth)
  apply (rule as-get-refine)
  apply (auto simp: as-length-def as-rel-def br-def as-raw-α-def
    as-raw-invar-def array-length-list
```

```
    dest!: list-rel-imp-same-length)
  done

lemma as-top-autoref[autoref-rules]:
  assumes (l,l')∈⟨R⟩as-rel
  assumes SIDE-PRECOND (l' ≠ [])
  shows (as-top l,(OP last ::: ⟨R⟩as-rel → R)$l')∈R
  using assms by (simp add: as-top-refine)


definition as-is-empty s ≡ as-length s = 0
lemma as-is-empty-code[code]: as-is-empty s = (snd s = 0)
  unfolding as-is-empty-def as-length-def by simp

lemma as-is-empty-refine[autoref-rules]:
  (as-is-empty,is-Nil) ∈ ⟨R⟩as-rel → bool-rel
proof
  fix s l
  assume [param]: (s,l)∈⟨R⟩as-rel
  have (as-is-empty s,length l = 0) ∈ bool-rel
    unfolding as-is-empty-def
    by (parametricity add: as-length-refine)
  also have length l = 0 ⟷ is-Nil l
    by (cases l) auto
  finally show (as-is-empty s, is-Nil l) ∈ bool-rel .
qed

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

lemma as-take-refine[autoref-rules]:
  (as-take,take)∈nat-rel → ⟨R⟩as-rel → ⟨R⟩as-rel
  apply (intro fun-relI)
  apply (clarsimp simp add: as-take-def, safe)

  apply (rule as-shrinkI)
  apply (simp add: as-rel-def br-def as-raw-α-def as-raw-invar-def)
  apply rule
  apply auto []
  apply clarsimp
  apply (subgoal-tac take a' (list-of-array a) = take a' (take ba (list-of-array a)))
  apply (simp only: )
  apply (parametricity, rule IdI)
  apply simp

  apply (simp add: as-rel-def br-def as-raw-α-def as-raw-invar-def)
  apply rule
```

```
  apply auto []
  apply clarsimp
  apply (frule list-rel-imp-same-length)
  apply simp
  done

definition as-singleton x ≡ (array-of-list [x],1)
lemma as-singleton-refine[autoref-rules]:
  (as-singleton,op-list-singleton)∈R → ⟨R⟩as-rel
  apply (intro fun-relI)
  apply (simp add: as-singleton-def as-rel-def br-def as-raw-α-def
    as-raw-invar-def)
  apply rule
  apply (auto simp: array-length-list) []
  apply simp
  done

end

end
```

2.3.2 List Based Sets

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

  lemma list-all2-refl-conv:
    list-all2 P xs xs ⟷ (∀x∈set xs. P x x)
    by (induct xs) auto

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

  lemma param-glist-member[param]:
    (glist-member,glist-member)∈(Ra→Ra→Id) → Ra → ⟨Ra⟩list-rel → Id
    unfolding glist-member-def
    by (parametricity)

lemma list-member-alt: List.member = (λl x. glist-member (=) x l)
  proof (rule ext)+
    fix x l
    show List.member l x = glist-member (=) x l
      by (induct l) simp-all
  qed
```

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*
  unfolding *glist-insert-def*[*abs-def*]
  by (*parametricity*)

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*
  by (*induct l arbitrary*: *ac*) *auto*

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*
  unfolding *glist-delete-def*[*abs-def*]
    glist-delete-aux-def
    rev-append-def
  by (*parametricity*)

lemma *list-rel-Range*:
  $\forall x'\in set\ l'.\ x' \in Range\ R \Longrightarrow l' \in Range\ (\langle R\rangle list\text{-}rel)$
proof (*induction* l')
  case *Nil* **thus** *?case* **by** *force*
next
  case (*Cons* x' xs')
    then obtain *xs* **where** $(xs,xs') \in \langle R\rangle\ list\text{-}rel$ **by** *force*
    moreover from *Cons.prems* **obtain** *x* **where** $(x,x') \in R$ **by** *force*
    ultimately have $(x\#xs,\ x'\#xs') \in \langle R\rangle\ list\text{-}rel$ **by** *simp*
    thus *?case* **..**
qed

All finite sets can be represented

lemma *list-set-rel-range*:
  Range ($\langle R\rangle$*list-set-rel*) = { *S*. *finite S* ∧ *S*⊆*Range R* }
    (**is** *?A* = *?B*)
proof (*intro equalityI subsetI*)
  fix *s'* **assume** *s'* ∈ *?A*
  then obtain *l l'* **where** *A*: (*l*,*l'*) ∈ $\langle R\rangle$*list-rel* **and**
    B: *s'* = *set l'* **and** *C*: *distinct l'*
      unfolding *list-set-rel-def br-def* **by** *blast*
  moreover have *set l'* ⊆ *Range R*
      by (*induction rule*: *list-rel-induct*[*OF A*], *auto*)
  ultimately show *s'* ∈ *?B* **by** *simp*
next
  fix *s'* **assume** *A*: *s'* ∈ *?B*
  then obtain *l'* **where** *B*: *set l'* = *s'* **and** *C*: *distinct l'*
      using *finite-distinct-list* **by** *blast*
  hence (*l'*,*s'*) ∈ *br set distinct* **by** (*simp add*: *br-def*)

  moreover from *A* **and** *B* **have** ∀ *x*∈*set l'*. *x* ∈ *Range R* **by** *blast*
  from *list-rel-Range*[*OF this*] **obtain** *l*
      where (*l*,*l'*) ∈ $\langle R\rangle$*list-rel* **by** *blast*

  ultimately show *s'* ∈ *?A* **unfolding** *list-set-rel-def* **by** *fast*
qed

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

lemma *list-set-rel-finite*[*autoref-ga-rules*]:
  finite-set-rel ($\langle R\rangle$*list-set-rel*)
  unfolding *finite-set-rel-def list-set-rel-def*
  by (*auto simp*: *br-def*)

lemma *list-set-rel-sv*[*relator-props*]:
  single-valued R ⟹ *single-valued* ($\langle R\rangle$*list-set-rel*)
  unfolding *list-set-rel-def*
  by *tagged-solver*

lemma *Id-comp-Id*: *Id O Id* = *Id* **by** *simp*

lemma *glist-member-id-impl*:
  (*glist-member* (=), (∈)) ∈ *Id* → *br set distinct* → *Id*
proof (*intro fun-relI*, *goal-cases*)
  case (*1 x x' l s'*) **thus** *?case*
    by (*induct l arbitrary*: *s'*) (*auto simp*: *br-def*)
qed

lemma *glist-insert-id-impl*:
  (*glist-insert* (=), *Set.insert*) ∈ *Id* → *br set distinct* → *br set distinct*
proof –

```
    have IC: ⋀x s. insert x s = (if x∈s then s else insert x s) by auto

    show ?thesis
      apply (intro fun-relI)
      apply (subst IC)
      unfolding glist-insert-def
      apply (parametricity add: glist-member-id-impl)
      apply (auto simp: br-def)
      done
  qed

  lemma glist-delete-id-impl:
    (glist-delete (=), λx s. s−{x})
    ∈ Id→br set distinct → br set distinct
  proof (intro fun-relI)
    fix x x′:: ′a and s and s′ :: ′a set
    assume XREL: (x, x′) ∈ Id and SREL: (s, s′) ∈ br set distinct
    from XREL have [simp]: x′=x by simp

    {
      fix a and a′ :: ′a set
      assume (a,a′)∈br set distinct and s′ ∩ a′ = {}
      hence (glist-delete-aux (=) x s a, s′−{x′} ∪ a′)∈br set distinct
        using SREL
      proof (induction s arbitrary: a s′ a′)
        case Nil thus ?case by (simp add: br-def)
      next
        case (Cons y s)
        show ?case proof (cases x=y)
          case True with Cons show ?thesis
            by (auto simp add: br-def rev-append-eq)
        next
          case False
          have glist-delete-aux (=) x (y # s) a
            = glist-delete-aux (=) x s (y#a) by (simp add: False)
          also have (. . .,set s − {x′} ∪ insert y a′)∈br set distinct
            apply (rule Cons.IH[of y#a     insert y a′     set s])
            using Cons.prems by (auto simp: br-def)
          also have set s − {x′} ∪ insert y a′ = (s′ − {x′}) ∪ a′
          proof −
            from Cons.prems have [simp]: s′ = insert y (set s)
              by (auto simp: br-def)
            show ?thesis using False by auto
          qed
          finally show ?thesis .
        qed
      qed
    }
    from this[of []     {}]
```

show $(glist\text{-}delete\ (=)\ x\ s,\ s' - \{x'\}) \in br\ set\ distinct$
unfolding *glist-delete-def*
by (*simp add*: *br-def*)
qed

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

lemma *list-set-autoref-member*[*autoref-rules*]:
assumes $GEN\text{-}OP\ eq\ (=)\ (R \rightarrow R \rightarrow Id)$
shows $(glist\text{-}member\ eq,(\in)) \in R \rightarrow \langle R \rangle list\text{-}set\text{-}rel \rightarrow Id$
using *assms*
apply (*intro fun-relI*)
unfolding *list-set-rel-def*
apply (*erule relcompE*)
apply (*simp del*: *pair-in-Id-conv*)
apply (*subst Id-comp-Id*[*symmetric*])
apply (*rule relcompI*[*rotated*])
apply (*rule glist-member-id-impl*[*param-fo*])
apply (*rule IdI*)
apply *assumption*
apply *parametricity*
done

lemma *list-set-autoref-insert*[*autoref-rules*]:
assumes $GEN\text{-}OP\ eq\ (=)\ (R \rightarrow R \rightarrow Id)$
shows $(glist\text{-}insert\ eq, Set.insert)$
$\in R \rightarrow \langle R \rangle list\text{-}set\text{-}rel \rightarrow \langle R \rangle list\text{-}set\text{-}rel$
using *assms*
apply (*intro fun-relI*)
unfolding *list-set-rel-def*
apply (*erule relcompE*)
apply (*simp del*: *pair-in-Id-conv*)
apply (*rule relcompI*[*rotated*])
apply (*rule glist-insert-id-impl*[*param-fo*])
apply (*rule IdI*)
apply *assumption*
apply *parametricity*
done

lemma *list-set-autoref-delete*[*autoref-rules*]:
assumes $GEN\text{-}OP\ eq\ (=)\ (R \rightarrow R \rightarrow Id)$
shows $(glist\text{-}delete\ eq, op\text{-}set\text{-}delete)$
$\in R \rightarrow \langle R \rangle list\text{-}set\text{-}rel \rightarrow \langle R \rangle list\text{-}set\text{-}rel$
using *assms*
apply (*intro fun-relI*)
unfolding *list-set-rel-def*
apply (*erule relcompE*)

apply (*simp del*: *pair-in-Id-conv*)
apply (*rule relcompI*[*rotated*])
apply (*rule glist-delete-id-impl*[*param-fo*])
apply (*rule IdI*)
apply *assumption*
apply *parametricity*
done

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* (λ- -. *True*) *R list-set-rel id*
unfolding *is-set-to-list-def is-set-to-sorted-list-def*
it-to-sorted-list-def list-set-rel-def br-def
by *auto*

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

lemma *glist-insert-dj-id-impl*:
$[\![$ $x \notin s$; $(l,s) \in br$ *set distinct* $]\!] \Longrightarrow$ $(x\#l, insert\ x\ s) \in br$ *set distinct*
by (*auto simp*: *br-def*)

context begin interpretation *autoref-syn* **.**
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*
using *assms*
unfolding *autoref-tag-defs*
unfolding *list-set-rel-def*
apply −
apply (*erule relcompE*)
apply (*simp del*: *pair-in-Id-conv*)
apply (*rule relcompI*[*rotated*])
apply (*rule glist-insert-dj-id-impl*)
apply *assumption*
apply *assumption*
apply *parametricity*
done
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*
by (*auto simp*: *list-set-rel-def br-def split*: *list.split-asm*)

```
lemma list-set-autoref-filter[autoref-rules]:
  (filter,op-set-filter)
    ∈ (R → bool-rel) → ⟨R⟩list-set-rel → ⟨R⟩list-set-rel
proof −
  have (filter, op-set-filter)
    ∈ (Id → bool-rel) → ⟨Id⟩list-set-rel → ⟨Id⟩list-set-rel
    by (auto simp: list-set-rel-def br-def)
  note this[param-fo]
  moreover have (filter,filter)∈(R → bool-rel) → ⟨R⟩list-rel → ⟨R⟩list-rel
    unfolding List.filter-def
    by parametricity
  note this[param-fo]
  ultimately show ?thesis
    unfolding list-set-rel-def
    apply (intro fun-relI)
    apply (erule relcompE, simp)
    apply (rule relcompI)
    apply (rprems, assumption+)
    apply (rprems, simp+)
    done
qed


context begin interpretation autoref-syn .
lemma list-set-autoref-inj-image[autoref-rules]:
  assumes PRIO-TAG-OPTIMIZATION
  assumes INJ: SIDE-PRECOND-OPT (inj-on f s)
  assumes [param]: (fi,f)∈Ra→Rb
  assumes LP: (l,s)∈⟨Ra⟩list-set-rel
  shows (map fi l,
    (OP (`) ::: (Ra→Rb) → ⟨Ra⟩list-set-rel → ⟨Rb⟩list-set-rel)$f$s)
    ∈ ⟨Rb⟩list-set-rel
proof −
  from LP obtain l' where
    [param]: (l,l')∈⟨Ra⟩list-rel and L'S: (l',s)∈br set distinct
    unfolding list-set-rel-def by auto

  have (map fi l, map f l')∈⟨Rb⟩list-rel by parametricity
  also from INJ L'S have (map f l',f`s)∈br set distinct
    apply (induction l' arbitrary: s)
    apply (auto simp: br-def dest: injD)
    done
  finally (relcompI) show ?thesis
    unfolding autoref-tag-defs list-set-rel-def .
qed

end
```

lemma *list-set-cart-autoref* [*autoref-rules*]:
  fixes *Rx* :: ('*xi* × '*x*) *set*
  fixes *Ry* :: ('*yi* × '*y*) *set*
  shows (λ*xl yl*. [(*x*,*y*). *x*←*xl*, *y*←*yl*], *op-set-cart*)
  ∈ ⟨*Rx*⟩*list-set-rel* → ⟨*Ry*⟩*list-set-rel* → ⟨*Rx* $\times_r$ *Ry*⟩*list-set-rel*
proof (*intro fun-relI*)
  fix *xl xs yl ys*
  assume (*xl*, *xs*) ∈ ⟨*Rx*⟩*list-set-rel*   (*yl*, *ys*) ∈ ⟨*Ry*⟩*list-set-rel*
  then obtain *xl'* :: '*x list* **and** *yl'* :: '*y list* **where**
    [*param*]: (*xl*,*xl'*)∈⟨*Rx*⟩*list-rel*   (*yl*,*yl'*)∈⟨*Ry*⟩*list-rel*
    and *XLS*: (*xl'*,*xs*)∈*br set distinct* **and** *YLS*: (*yl'*,*ys*)∈*br set distinct*
    unfolding *list-set-rel-def*
    by *auto*

  have ([(*x*,*y*). *x*←*xl*, *y*←*yl*], [(*x*,*y*). *x*←*xl'*, *y*←*yl'*])
    ∈ ⟨*Rx* $\times_r$ *Ry*⟩*list-rel*
    by *parametricity*
  also have ([(*x*,*y*). *x*←*xl'*, *y*←*yl'*], *xs* × *ys*) ∈ *br set distinct*
    using *XLS YLS*
    apply (*auto simp*: *br-def*)
    apply *hypsubst-thin*
    apply (*induction xl'*)
    apply *simp*
    apply (*induction yl'*)
    apply *simp*
    apply *auto* []
    apply (*metis distinct-product product-concat-map*)
    done
  finally (*relcompI*)
  show ([(*x*,*y*). *x*←*xl*, *y*←*yl*], *op-set-cart xs ys*) ∈ ⟨*Rx* $\times_r$ *Ry*⟩*list-set-rel*
    unfolding *list-set-rel-def* **by** *simp*
qed

Optimizations

lemma *glist-delete-hd*: *eq x y* ⟹ *glist-delete eq x* (*y*#*s*) = *s*
  by (*simp add*: *glist-delete-def*)

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

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

lemma *LIST-SET-REV-TAG-autoref* [*autoref-rules*]:
  (*rev*,*LIST-SET-REV-TAG*) ∈ ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*
  unfolding *list-set-rel-def*
  apply (*intro fun-relI*)
  apply (*elim relcompE*)
  apply (*clarsimp simp*: *br-def*)

apply (*rule relcompI*)
apply (*rule param-rev*[*param-fo*], *assumption*)
apply *auto*
done

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

lemma *idx-iteratei-aux-array-get-Array-conv-nth*:
idx-iteratei-aux array-get sz i (*Array xs*) *c f* σ =
idx-iteratei-aux (!) *sz i xs c f* σ
apply(*induct get*≡(!) :: *'b list* ⇒ *nat* ⇒ *'b sz i xs c f* σ *rule*: *idx-iteratei-aux.induct*)
apply(*subst* (*1 2*) *idx-iteratei-aux.simps*)
apply *simp*
done

lemma *idx-iteratei-array-get-Array-conv-nth*:
idx-iteratei array-get array-length (*Array xs*) = *idx-iteratei nth length xs*
by(*simp add*: *idx-iteratei-def fun-eq-iff idx-iteratei-aux-array-get-Array-conv-nth*)

end

2.3.3 List Based Maps

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

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

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

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

lemma *list-map-rel-def*:
⟨*Rk*,*Rv*⟩*list-map-rel* = ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel O br map-of list-map-invar*
unfolding *list-map-rel-internal-def*[*abs-def*] **by** (*simp add*: *relAPP-def*)

```
lemma list-rel-Range:
   ∀x'∈set l'. x' ∈ Range R ⟹ l' ∈ Range (⟨R⟩list-rel)
proof (induction l')
  case Nil thus ?case by force
next
  case (Cons x' xs')
    then obtain xs where (xs,xs') ∈ ⟨R⟩ list-rel by force
    moreover from Cons.prems obtain x where (x,x') ∈ R by force
    ultimately have (x#xs, x'#xs') ∈ ⟨R⟩ list-rel by simp
    thus ?case ..
qed
```

All finite maps can be represented

```
lemma list-map-rel-range:
  Range (⟨Rk,Rv⟩list-map-rel) =
      {m. finite (dom m) ∧ dom m ⊆ Range Rk ∧ ran m ⊆ Range Rv}
      (is ?A = ?B)
proof (intro equalityI subsetI)
  fix m' assume m' ∈ ?A
  then obtain l l' where A: (l,l') ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel and
                         B: m' = map-of l' and C: list-map-invar l'
      unfolding list-map-rel-def br-def by blast
  {
    fix x' y' assume m' x' = Some y'
    with B have (x',y') ∈ set l' by (fast dest: map-of-SomeD)
    hence x' ∈ Range Rk and y' ∈ Range Rv
      by (induction rule: list-rel-induct[OF A], auto)
  }
  with B show m' ∈ ?B by (force dest: map-of-SomeD simp: ran-def)

next
  fix m' assume m' ∈ ?B
  hence A: finite (dom m') and B: dom m' ⊆ Range Rk and
        C: ran m' ⊆ Range Rv by simp-all
  from A have finite (map-to-set m') by (simp add: finite-map-to-set)
  from finite-distinct-list[OF this]
    obtain l' where l'-props: distinct l'    set l' = map-to-set m' by blast
  hence *: distinct (map fst l')
    by (force simp: distinct-map inj-on-def map-to-set-def)
  from map-of-map-to-set[OF this] and l'-props
    have  map-of l' = m' by simp
  with * have (l',m') ∈ br map-of list-map-invar
    unfolding br-def list-map-invar-def o-def by simp

  moreover from B and C and l'-props
    have ∀x ∈ set l'. x ∈ Range (⟨Rk,Rv⟩prod-rel)
    unfolding map-to-set-def ran-def prod-rel-def by force
  from list-rel-Range[OF this] obtain l where
    (l,l') ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel by force
```

ultimately show *m′* ∈ *?A* **unfolding** *list-map-rel-def* **by** *blast*
qed

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

lemma *list-map-rel-finite*[*autoref-ga-rules*]:
finite-map-rel (⟨*Rk*,*Rv*⟩*list-map-rel*)
unfolding *finite-map-rel-def list-map-rel-def*
by (*auto simp*: *br-def*)

lemma *list-map-rel-sv*[*relator-props*]:
single-valued Rk ⟹ *single-valued Rv* ⟹
single-valued (⟨*Rk*,*Rv*⟩*list-map-rel*)
unfolding *list-map-rel-def*
by *tagged-solver*

Implementation

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

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

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

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

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

definition *list-map-isEmpty* :: (*′k*,*′v*) *list-map* ⇒ *bool*

where *list-map-isEmpty* ≡ *List.null*

definition *list-map-isSng* :: ('k,'v) *list-map* ⇒ *bool*
where *list-map-isSng m* = (*case m of* [*x*] ⇒ *True* | - ⇒ *False*)

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

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

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

Parametricity

lemma *list-map-autoref-empty*[*autoref-rules*]:
([], *op-map-empty*)∈⟨*Rk*,*Rv*⟩*list-map-rel*
by (*auto simp*: *list-map-rel-def br-def list-map-invar-def*)

lemma *param-list-map-lookup*[*param*]:
(*list-map-lookup*,*list-map-lookup*) ∈ (*Rk* → *Rk* → *bool-rel*) →
Rk → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨*Rv*⟩*option-rel*
unfolding *list-map-lookup-def*[*abs-def*] **by** *parametricity*

lemma *list-map-autoref-lookup-aux*:
assumes *eq*: *GEN-OP eq* (=) (*Rk*→*Rk*→*Id*)
assumes *K*: (*k*, *k'*) ∈ *Rk*
assumes *M*: (*m*, *m'*) ∈ ⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel*
shows (*list-map-lookup eq k m*, *op-map-lookup k'* (*map-of m'*))
∈ ⟨*Rv*⟩*option-rel*
unfolding *op-map-lookup-def*
proof (*induction rule*: *list-rel-induct*[*OF M*, *case-names Nil Cons*])
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x x' xs xs'*)
from *eq* **have** *eq'*: (*eq*,(=)) ∈ *Rk* → *Rk* → *Id* **by** *simp*
with *eq'*[*param-fo*] **and** *K* **and** *Cons*
show *?case* **by** (*force simp*: *prod-rel-def*)
qed

lemma *list-map-autoref-lookup*[*autoref-rules*]:
assumes *GEN-OP eq* (=) (*Rk*→*Rk*→*Id*)
shows (*list-map-lookup eq*, *op-map-lookup*) ∈
Rk → ⟨*Rk*,*Rv*⟩*list-map-rel* → ⟨*Rv*⟩*option-rel*
by (*force simp*: *list-map-rel-def br-def*
dest: *list-map-autoref-lookup-aux*[*OF assms*])

```
lemma param-list-map-update-aux[param]:
  (list-map-update-aux,list-map-update-aux) ∈ (Rk → Rk → bool-rel) →
      Rk → Rv → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
      → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-update-aux-def[abs-def] by parametricity

lemma param-list-map-update[param]:
  (list-map-update,list-map-update) ∈ (Rk → Rk → bool-rel) →
      Rk → Rv → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-update-def[abs-def] by parametricity


lemma list-map-autoref-update-aux1:
  assumes eq: (eq,(=)) ∈ Rk → Rk → Id
  assumes K: (k, k′) ∈ Rk
  assumes V: (v, v′) ∈ Rv
  assumes A: (accu, accu′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  assumes M: (m, m′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  shows (list-map-update-aux eq k v m accu,
          list-map-update-aux (=) k′ v′ m′ accu′)
              ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
proof (insert A, induction arbitrary: accu accu′
          rule: list-rel-induct[OF M, case-names Nil Cons])
  case Nil
    thus ?case by (simp add: K V)
next
  case (Cons x x′ xs xs′)
    from eq have eq′: (eq,(=)) ∈ Rk → Rk → Id by simp
    from eq′[param-fo] Cons(1) K
        have [simp]: (eq (fst x) k) ⟷ ((fst x′) = k′)
        by (force simp: prod-rel-def)
    show ?case
    proof (cases eq (fst x) k)
      case False
        from Cons.prems and Cons.hyps have (x # accu, x′ # accu′) ∈
            ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
        from Cons.IH[OF this] and False show ?thesis by simp
    next
      case True
        from Cons.prems and Cons.hyps have (xs @ accu, xs′ @ accu′) ∈
            ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
        with K and V and True show ?thesis by simp
  qed
qed

lemma list-map-autoref-update1[param]:
```

assumes *eq*: $(eq,(=)) \in Rk \to Rk \to Id$
shows $(\textit{list-map-update eq}, \textit{list-map-update } (=)) \in Rk \to Rv \to \langle\langle Rk, Rv\rangle \textit{prod-rel}\rangle \textit{list-rel} \to \langle\langle Rk, Rv\rangle \textit{prod-rel}\rangle \textit{list-rel}$
unfolding *list-map-update-def*[*abs-def*]
by (*intro fun-relI*, *erule* (*1*) *list-map-autoref-update-aux1*[*OF eq*], *simp-all*)

lemma *map-add-sng-right*: $m \mathbin{++} [k \mapsto v] = m(k \mapsto v)$
unfolding *map-add-def* **by** *force*
lemma *map-add-sng-right′*:
$m \mathbin{++} (\lambda a.\ \textit{if } a = k \textit{ then Some } v \textit{ else None}) = m(k \mapsto v)$
unfolding *map-add-def* **by** *force*

lemma *list-map-autoref-update-aux2*:
assumes *K*: $(k, k') \in Id$
assumes *V*: $(v, v') \in Id$
assumes *A*: $(accu, accu') \in \textit{br map-of list-map-invar}$
assumes *A1*: *distinct* (*map fst* (m @ *accu*))
assumes *A2*: $k \notin \textit{set (map fst accu)}$
assumes *M*: $(m, m') \in \textit{br map-of list-map-invar}$
shows (*list-map-update-aux* (=) *k v m accu*,
accu′ ++ *op-map-update k′ v′ m′*)
$\in$ *br map-of list-map-invar* (**is** (*?f m accu*, -) ∈ -)
using *M A A1 A2*
proof (*induction m arbitrary*: *accu accu′ m′*)
case *Nil*
with *K V* **show** *?case* **by** (*auto simp*: *br-def list-map-invar-def map-add-sng-right′*)
next
case (*Cons x xs accu accu′ m′*)
from *Cons.prems* **have** *A*: *m′* = *map-of* (*x#xs*) *accu′* = *map-of accu*
unfolding *br-def* **by** *simp-all*
show *?case*
proof (*cases* (*fst x*) = *k*)
case *True*
hence ((*k*, *v*) # *xs* @ *accu*, *accu′* ++ *op-map-update k′ v′ m′*)
∈ *br map-of list-map-invar*
using *K V Cons.prems*(*3*,*4*) **unfolding** *br-def*
by (*force simp add*: *A list-map-invar-def*)
also from *True* **have** (*k*,*v*) # *xs* @ *accu* = *?f* (*x* # *xs*) *accu* **by** *simp*
finally show *?thesis* **.**
next
case *False*
from *Cons.prems*(*1*) **have** *B*: (*xs*, *map-of xs*) ∈ *br map-of list-map-invar***by** (*simp add*: *br-def list-map-invar-def*)
from *Cons.prems*(*2*,*3*) **have** *C*: (*x#accu*, *map-of* (*x#accu*)) ∈ *br map-of list-map-invar* **by** (*simp add*: *br-def list-map-invar-def*)

```
    from Cons.prems(3) have D: distinct (map fst (xs @ x # accu))
      by simp
    from Cons.prems(4) and False have E: k ∉ set (map fst (x # accu))
      by simp
    note Cons.IH[OF B C D E]
    also from False have ?f xs (x#accu) = ?f (x#xs) accu by simp
    also from distinct-map-fstD[OF D]
      have F: ⋀z. (fst x, z) ∈ set xs ⟹ z = snd x by force
    have map-of (x # accu) ++ op-map-update k′ v′ (map-of xs) =
          accu′ ++ op-map-update k′ v′ m′
      by (intro ext, auto simp: A F map-add-def
            dest: map-of-SomeD split: option.split)
    finally show ?thesis .
  qed
qed

lemma list-map-autoref-update2[param]:
  shows (list-map-update (=), op-map-update) ∈ Id → Id →
          br map-of list-map-invar → br map-of list-map-invar
unfolding list-map-update-def[abs-def]
apply (intro fun-relI)
apply (drule list-map-autoref-update-aux2
          [where accu=[] and accu′=Map.empty])
apply (auto simp: br-def list-map-invar-def)
done

lemma list-map-autoref-update[autoref-rules]:
  assumes eq: GEN-OP eq (=) (Rk→Rk→Id)
  shows (list-map-update eq, op-map-update) ∈
    Rk → Rv → ⟨Rk,Rv⟩list-map-rel → ⟨Rk,Rv⟩list-map-rel
unfolding list-map-rel-def
apply (intro fun-relI, elim relcompE, intro relcompI, clarsimp)
apply (erule (2) list-map-autoref-update1[param-fo, OF eq[simplified]])
apply (rule list-map-autoref-update2[param-fo], simp-all)
done

context begin interpretation autoref-syn .
lemma list-map-autoref-update-dj[autoref-rules]:
  assumes PRIO-TAG-OPTIMIZATION
  assumes new: SIDE-PRECOND-OPT (k′ ∉ dom m′)
  assumes K: (k,k′)∈Rk and V: (v,v′)∈Rv
  assumes M: (l,m′)∈⟨Rk, Rv⟩list-map-rel
  defines R-annot ≡ Rk → Rv → ⟨Rk,Rv⟩list-map-rel → ⟨Rk,Rv⟩list-map-rel
  shows
    ((k, v)#l,
      (OP op-map-update:::R-annot)$k′$v′$m′)
    ∈ ⟨Rk,Rv⟩list-map-rel
proof −
  from M obtain l′ where A: (l,l′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel and
```

```
    B: (l′,m′) ∈ br map-of list-map-invar
    unfolding list-map-rel-def by blast
  hence ((k,v)#l, (k′,v′)#l′)∈⟨⟨Rk, Rv⟩prod-rel⟩list-rel
    and ((k′,v′)#l′, m′(k′ ↦ v′)) ∈ br map-of list-map-invar
    using assms unfolding br-def list-map-invar-def
      by (simp-all add: dom-map-of-conv-image-fst)
  thus ?thesis
    unfolding autoref-tag-defs
    by (force simp: list-map-rel-def)
qed
end

lemma param-list-map-delete-aux[param]:
  (list-map-delete-aux,list-map-delete-aux) ∈ (Rk → Rk → bool-rel) →
     Rk → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
      → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-aux-def[abs-def] by parametricity

lemma param-list-map-delete[param]:
  (list-map-delete,list-map-delete) ∈ (Rk → Rk → bool-rel) →
     Rk → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-def[abs-def] by parametricity

lemma list-map-autoref-delete-aux1:
  assumes eq: (eq,(=)) ∈ Rk → Rk → Id
  assumes K: (k, k′) ∈ Rk
  assumes A: (accu, accu′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  assumes M: (m, m′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  shows (list-map-delete-aux eq k m accu,
         list-map-delete-aux (=) k′ m′ accu′)
           ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
proof (insert A, induction arbitrary: accu accu′
         rule: list-rel-induct[OF M, case-names Nil Cons])
  case Nil
    thus ?case by (simp add: K)
next
  case (Cons x x′ xs xs′)
   from eq have eq′: (eq,(=)) ∈ Rk → Rk → Id by simp
   from eq′[param-fo] Cons(1) K
     have [simp]: (eq (fst x) k) ⟷ ((fst x′) = k′)
     by (force simp: prod-rel-def)
   show ?case
   proof (cases eq (fst x) k)
     case False
       from Cons.prems and Cons.hyps have (x # accu, x′ # accu′) ∈
         ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
       from Cons.IH[OF this] and False show ?thesis by simp
   next
     case True
```

```
      from Cons.prems and Cons.hyps have (xs @ accu, xs' @ accu') ∈
          ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
      with K and True show ?thesis by simp
  qed
qed

lemma list-map-autoref-delete1[param]:
  assumes eq: (eq,(=)) ∈ Rk → Rk → Id
  shows (list-map-delete eq, list-map-delete (=)) ∈ Rk →
            ⟨⟨Rk, Rv⟩prod-rel⟩list-rel → ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-def[abs-def]
  by (intro fun-relI, erule list-map-autoref-delete-aux1[OF eq],
    simp-all)


lemma list-map-autoref-delete-aux2:
  assumes K: (k, k') ∈ Id
  assumes A: (accu, accu') ∈ br map-of list-map-invar
  assumes A1: distinct (map fst (m @ accu))
  assumes A2: k ∉ set (map fst accu)
  assumes M: (m, m') ∈ br map-of list-map-invar
  shows (list-map-delete-aux (=) k m accu,
          accu' ++ op-map-delete k' m')
              ∈ br map-of list-map-invar (is (?f m accu, -) ∈ -)
using M A A1 A2
proof (induction m arbitrary: accu accu' m')
  case Nil
    with K show ?case by (auto simp: br-def list-map-invar-def
      map-add-sng-right')
next
  case (Cons x xs accu accu' m')
    from Cons.prems have A: m' = map-of (x#xs)    accu' = map-of accu
      unfolding br-def by simp-all
    show ?case
    proof (cases (fst x) =  k)
      case True
        with Cons.prems(3) have map-of xs (fst x) = None
          by (induction xs, simp-all)
        with fun-upd-triv[of map-of xs    fst x]
        have map-of xs |` (− {fst x}) = map-of xs
          by (simp add: map-upd-eq-restrict)
        with True have(xs @ accu, accu' ++ op-map-delete k' m')
                        ∈ br map-of list-map-invar
            using K Cons.prems unfolding br-def
            by (auto simp add: A list-map-invar-def)
        thus ?thesis using True by simp
    next
      case False
        from False and K have [simp]: fst x ≠ k' by simp
```

```
        from Cons.prems(1) have B: (xs, map-of xs) ∈ br map-of
            list-map-invarby (simp add: br-def list-map-invar-def)
        from Cons.prems(2,3) have C: (x#accu, map-of (x#accu)) ∈ br map-of
            list-map-invar by (simp add: br-def list-map-invar-def)
        from Cons.prems(3) have D: distinct (map fst (xs @ x # accu))
            by simp
        from Cons.prems(4) and False have E: k ∉ set (map fst (x # accu))
            by simp
        note Cons.IH[OF B C D E]
        also from False have ?f xs (x#accu) = ?f (x#xs) accu by simp
        also from distinct-map-fstD[OF D]
            have F: ⋀z. (fst x, z) ∈ set xs ⟹ z = snd x by force

        from Cons.prems(3) have map-of xs (fst x) = None
            by (induction xs, simp-all)
        hence map-of (x # accu) ++ op-map-delete k' (map-of xs) =
               accu' ++ op-map-delete k' m'
            apply (intro ext, simp add: map-add-def A
                              split: option.split)
            apply (intro conjI impI allI)
            apply (auto simp: restrict-map-def)
            done
        finally show ?thesis .
    qed
qed

lemma list-map-autoref-delete2[param]:
  shows (list-map-delete (=), op-map-delete) ∈ Id →
            br map-of list-map-invar → br map-of list-map-invar
unfolding list-map-delete-def[abs-def]
apply (intro fun-relI)
apply (drule list-map-autoref-delete-aux2
                [where accu=[] and accu'=Map.empty])
apply (auto simp: br-def list-map-invar-def)
done

lemma list-map-autoref-delete[autoref-rules]:
  assumes eq: GEN-OP eq (=) (Rk→Rk→Id)
  shows (list-map-delete eq, op-map-delete) ∈
      Rk → ⟨Rk,Rv⟩list-map-rel → ⟨Rk,Rv⟩list-map-rel
unfolding list-map-rel-def
apply (intro fun-relI, elim relcompE, intro relcompI, clarsimp)
apply (erule (1) list-map-autoref-delete1[param-fo, OF eq[simplified]])
apply (rule list-map-autoref-delete2[param-fo], simp-all)
done

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

```
unfolding list-map-isEmpty-def op-map-isEmpty-def[abs-def]
  list-map-rel-def br-def by force

lemma param-list-map-isSng[param]:
 assumes (l,l') ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
 shows (list-map-isSng l, list-map-isSng l') ∈ bool-rel
unfolding list-map-isSng-def using assms by parametricity


lemma list-map-autoref-isSng-aux:
 assumes (l',m') ∈ br map-of list-map-invar
 shows (list-map-isSng l', op-map-isSng m') ∈ bool-rel
using assms
unfolding list-map-isSng-def op-map-isSng-def br-def list-map-invar-def
apply (clarsimp split: list.split)
apply (intro conjI impI allI)
apply (metis map-upd-nonempty)
apply blast
apply (simp, metis fun-upd-apply option.distinct(1))
done

lemma list-map-autoref-isSng[autoref-rules]:
 (list-map-isSng, op-map-isSng) ∈ ⟨Rk,Rv⟩list-map-rel → bool-rel
 unfolding list-map-rel-def
 by (blast dest!: param-list-map-isSng list-map-autoref-isSng-aux)

lemma list-map-autoref-size-aux:
 assumes distinct (map fst x)
 shows card (dom (map-of x)) = length x
proof−
 have card (dom (map-of x)) = card (map-to-set (map-of x))
   by (simp add: card-map-to-set)
 also from assms have ... = card (set x)
   by (simp add: map-to-set-map-of)
 also from assms have ... = length x
   by (force simp: distinct-card dest!: distinct-mapI)
 finally show ?thesis .
qed

lemma param-list-map-size[param]:
 (list-map-size, list-map-size) ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → nat-rel
 unfolding list-map-size-def[abs-def] by parametricity

lemma list-map-autoref-size[autoref-rules]:
 shows (list-map-size, op-map-size) ∈
       ⟨Rk,Rv⟩list-map-rel → nat-rel
unfolding list-map-size-def[abs-def] op-map-size-def[abs-def]
  list-map-rel-def br-def list-map-invar-def
  by (force simp: list-map-autoref-size-aux list-rel-imp-same-length)
```

lemma *autoref-list-map-is-iterator*[*autoref-ga-rules*]:
  shows *is-map-to-list Rk Rv list-map-rel list-map-to-list*
unfolding *is-map-to-list-def is-map-to-sorted-list-def*
proof (*clarify*)
  fix l m'
  assume $(l,m') \in \langle Rk,Rv\rangle$*list-map-rel*
  then obtain l' **where** $*$: $(l,l')\in\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel* $(l',m')\in br$ *map-of list-map-invar*
    unfolding *list-map-rel-def* **by** *blast*
  then have *RETURN* $l' \le$ *it-to-sorted-list* (*key-rel* (λ- -. *True*)) (*map-to-set* m')
      unfolding *it-to-sorted-list-def*
      apply (*intro refine-vcg*)
      unfolding *br-def list-map-invar-def key-rel-def*[*abs-def*]
      apply (*auto intro*: *distinct-mapI simp*: *map-to-set-map-of*)
      done
  with $*$ **show**
      $\exists\, l'$. (*list-map-to-list* l, l') $\in \langle\langle Rk,\ Rv\rangle$*prod-rel*$\rangle$*list-rel* $\wedge$
          RETURN $l' \le$ *it-to-sorted-list* (*key-rel* (λ- -. *True*))
                              (*map-to-set* m')
    unfolding *list-map-to-list-def* **by** *force*
qed

lemma *pi-list-map*[*icf-proper-iteratorI*]:
  proper-it (*list-map-iteratei m*) (*list-map-iteratei m*)
unfolding *proper-it-def list-map-iteratei-def* **by** *blast*

lemma *pi'-list-map*[*icf-proper-iteratorI*]:
  proper-it' list-map-iteratei list-map-iteratei
  by (*rule proper-it'I*, *rule pi-list-map*)

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

context begin interpretation *autoref-syn* **.**
  lemma *list-map-autoref-pick-remove*[*autoref-rules*]:
    assumes *NE*: *SIDE-PRECOND* ($m\neq$*Map.empty*)
    assumes *R*: $(l,m)\in\langle Rk,Rv\rangle$*list-map-rel*
    defines $Rres \equiv \langle(Rk\times_r Rv)\ \times_r\ \langle Rk,Rv\rangle$*list-map-rel*$\rangle$*nres-rel*
    shows (
        RETURN (*list-map-pick-remove l*),
        (*OP op-map-pick-remove* ::: $\langle Rk,Rv\rangle$*list-map-rel* $\rightarrow$ *Rres*)\$*m*
) $\in$ *Rres*
  proof −
    from *NE R* **obtain** *k v lr* **where**
      [*simp*]: *l*=(*k*,*v*)#*lr*

```
    by (cases l) (auto simp: list-map-rel-def br-def)

  thm list-map-rel-def
  from R obtain l' where
    LL': (l,l')∈⟨Rk×r Rv⟩list-rel and
    L'M: (l',m)∈br map-of list-map-invar
    unfolding list-map-rel-def by auto
  from LL' obtain k' v' lr' where
    [simp]: l' = (k',v')#lr' and
      KVR: (k,k')∈Rk    (v,v')∈Rv and
      LRR: (lr,lr')∈⟨Rk×r Rv⟩list-rel
    by (fastforce elim!: list-relE)

  from L'M have
    MKV': m k' = Some v' and
    LRR': (lr',m|`(−{k'}))∈br map-of list-map-invar
    by (auto
      simp: restrict-map-def map-of-eq-None-iff br-def list-map-invar-def
      intro!: ext
      intro: sym)

  from LRR LRR' have LM: (lr,m|`(−{k'}))∈⟨Rk,Rv⟩list-map-rel
    unfolding list-map-rel-def by auto

  show ?thesis
    apply (simp add: Rres-def)
    apply (refine-rcg SPEC-refine nres-relI refine-vcg)
    using LM KVR MKV'
    by auto
 qed
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/Diff-Array
  Impl-List-Map
begin
```

Type definition and primitive operations

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

where *load-factor* = *75*

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

Operations

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

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

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

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

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

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

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

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

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

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

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

definition *ahm-rel-def-internal*: *ahm-rel bhc Rk Rv* =
  $\langle$*Rk*,*Rv*$\rangle$ *ahm-map-rel O ahm-map-rel'* (*abstract-bounded-hashcode Rk bhc*)
lemma *ahm-rel-def*: $\langle$*Rk*, *Rv*$\rangle$ *ahm-rel bhc* $\equiv$
  $\langle$*Rk*,*Rv*$\rangle$ *ahm-map-rel O ahm-map-rel'* (*abstract-bounded-hashcode Rk bhc*)
  unfolding *relAPP-def ahm-rel-def-internal* **.**
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ahm-rel bhc i-map*] **for** *bhc*

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

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

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

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

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

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

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

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

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

$(if\ insert\ then\ n + 1\ else\ n))$

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

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

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

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

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

lemma *hm-grow-gt-1 [iff]*:
 Suc 0 < hm-grow hm
by(*cases hm*)(*simp*)

lemma *bucket-ok-Nil [simp]*: *bucket-ok bhc len h [] = True*
by(*simp add*: *bucket-ok-def*)

lemma *bucket-okD*:
 ⟦ *bucket-ok bhc len h xs*; *(k, v) ∈ set xs* ⟧
 ⟹ *bhc len k = h*
by(*auto simp add*: *bucket-ok-def*)

lemma *bucket-okI*:
 $(\bigwedge k.\ k \in fst\ `\ set\ kvs \Longrightarrow bhc\ len\ k = h) \Longrightarrow bucket\text{-}ok\ bhc\ len\ h\ kvs$
by(*simp add*: *bucket-ok-def*)

Parametricity

lemma *param-HashMap[param]*: *(HashMap, HashMap)* ∈
 $\langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel \to nat\text{-}rel \to \langle Rk,Rv\rangle ahm\text{-}map\text{-}rel$

unfolding *ahm-map-rel-def* **by** *force*

lemma *param-case-hashmap*[*param*]: (*case-hashmap*, *case-hashmap*) ∈
(⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* → *nat-rel* → *R*) →
⟨*Rk*,*Rv*⟩*ahm-map-rel* → *R*
unfolding *ahm-map-rel-def*[*abs-def*]
by (*force split*: *hashmap.split dest*: *fun-relD*)

lemma *rec-hashmap-is-case*[*simp*]: *rec-hashmap* = *case-hashmap*
by (*intro ext*, *simp split*: *hashmap.split*)

ahm-invar

lemma *ahm-invar-auxD*:
assumes *ahm-invar-aux bhc n a*
shows $\bigwedge$*h*. *h* < *array-length a* ⟹
bucket-ok bhc (*array-length a*) *h* (*array-get a h*) **and**
$\bigwedge$*h*. *h* < *array-length a* ⟹
list-map-invar (*array-get a h*) **and**
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and**
array-length a > *1*
using *assms* **unfolding** *ahm-invar-aux-def* **by** *auto*

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

lemma *ahm-invar-auxI*:
⟦ $\bigwedge$*h*. *h* < *array-length a* ⟹
bucket-ok bhc (*array-length a*) *h* (*array-get a h*);
$\bigwedge$*h*. *h* < *array-length a* ⟹ *list-map-invar* (*array-get a h*);
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*; *array-length a* > *1* ⟧
⟹ *ahm-invar-aux bhc n a*
unfolding *ahm-invar-aux-def* **by** *blast*

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux bhc n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
proof −
{ fix *h*
assume *h* < *length xs*
with *inv* **have** *bucket-ok bhc* (*length xs*) *h* (*xs* ! *h*) **and**
list-map-invar (*xs* ! *h*)
by(*simp-all add*: *ahm-invar-aux-def*) **}**

note *no-junk* = *this*

show *?thesis* **unfolding** *map-concat*
proof(*rule distinct-concat'*)
have *distinct* $[x{\leftarrow}xs \,.\, x \neq []]$ **unfolding** *distinct-conv-nth*
proof(*intro allI ballI impI*)
fix $i\ j$
assume $i < length\ [x{\leftarrow}xs \,.\, x \neq []] \quad j < length\ [x{\leftarrow}xs \,.\, x \neq []] \quad i \neq j$
from *filter-nth-ex-nth*$[OF\ \langle i < length\ [x{\leftarrow}xs \,.\, x \neq []]\rangle]$
obtain i' **where** $i' \geq i \quad i' < length\ xs$ **and** *ith*: $[x{\leftarrow}xs \,.\, x \neq []]\ !\ i = xs\ !\ i'$
and *eqi*: $[x{\leftarrow}take\ i'\ xs \,.\, x \neq []] = take\ i\ [x{\leftarrow}xs \,.\, x \neq []]$ **by** *blast*
from *filter-nth-ex-nth*$[OF\ \langle j < length\ [x{\leftarrow}xs \,.\, x \neq []]\rangle]$
obtain j' **where** $j' \geq j \quad j' < length\ xs$ **and** *jth*: $[x{\leftarrow}xs \,.\, x \neq []]\ !\ j = xs\ !\ j'$
and *eqj*: $[x{\leftarrow}take\ j'\ xs \,.\, x \neq []] = take\ j\ [x{\leftarrow}xs \,.\, x \neq []]$ **by** *blast*
show $[x{\leftarrow}xs \,.\, x \neq []]\ !\ i \neq [x{\leftarrow}xs \,.\, x \neq []]\ !\ j$
proof
assume $[x{\leftarrow}xs \,.\, x \neq []]\ !\ i = [x{\leftarrow}xs \,.\, x \neq []]\ !\ j$
hence *eq*: $xs\ !\ i' = xs\ !\ j'$ **using** *ith jth* **by** *simp*
from $\langle i < length\ [x{\leftarrow}xs \,.\, x \neq []]\rangle$
have $[x{\leftarrow}xs \,.\, x \neq []]\ !\ i \in set\ [x{\leftarrow}xs \,.\, x \neq []]$ **by**(*rule nth-mem*)
with *ith* **have** $xs\ !\ i' \neq []$ **by** *simp*
then obtain *kv* **where** $kv \in set\ (xs\ !\ i')$ **by**(*fastforce simp add*: *neq-Nil-conv*)
with *no-junk*$[OF\ \langle i' < length\ xs\rangle]$ **have** $bhc\ (length\ xs)\ (fst\ kv) = i'$
by(*simp add*: *bucket-ok-def*)
moreover from *eq* $\langle kv \in set\ (xs\ !\ i')\rangle$ **have** $kv \in set\ (xs\ !\ j')$ **by** *simp*
with *no-junk*$[OF\ \langle j' < length\ xs\rangle]$ **have** $bhc\ (length\ xs)\ (fst\ kv) = j'$
by(*simp add*: *bucket-ok-def*)
ultimately have [*simp*]: $i' = j'$ **by** *simp*
from $\langle i < length\ [x{\leftarrow}xs \,.\, x \neq []]\rangle$ **have** $i = length\ (take\ i\ [x{\leftarrow}xs \,.\, x \neq []])$ **by** *simp*
also from *eqi eqj* **have** $take\ i\ [x{\leftarrow}xs \,.\, x \neq []] = take\ j\ [x{\leftarrow}xs \,.\, x \neq []]$ **by** *simp*
finally show *False* **using** $\langle i \neq j\rangle\ \langle j < length\ [x{\leftarrow}xs \,.\, x \neq []]\rangle$ **by** *simp*
qed
qed
moreover have *inj-on* $(map\ fst)\ \{x \in set\ xs.\ x \neq []\}$
proof(*rule inj-onI*)
fix $x\ y$
assume $x \in \{x \in set\ xs.\ x \neq []\} \quad y \in \{x \in set\ xs.\ x \neq []\} \quad map\ fst\ x = map\ fst\ y$
hence $x \in set\ xs \quad y \in set\ xs \quad x \neq [] \quad y \neq []$ **by** *auto*
from $\langle x \in set\ xs\rangle$ **obtain** i **where** $xs\ !\ i = x \quad i < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from $\langle y \in set\ xs\rangle$ **obtain** j **where** $xs\ !\ j = y \quad j < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from $\langle x \neq []\rangle$ **obtain** $k\ v\ x'$ **where** $x = (k,\ v)\ \#\ x'$ **by**(*cases x*) *auto*
with *no-junk*$[OF\ \langle i < length\ xs\rangle]\ \langle xs\ !\ i = x\rangle$
have $bhc\ (length\ xs)\ k = i$ **by**(*auto simp add*: *bucket-ok-def*)
moreover from $\langle map\ fst\ x = map\ fst\ y\rangle\ \langle x = (k,\ v)\ \#\ x'\rangle$ **obtain** v' **where**

$(k, v') \in$ *set y* **by** *fastforce*
with *no-junk*[*OF* ‹$j <$ *length xs*›] ‹*xs* ! $j = y$›
have *bhc* (*length xs*) $k = j$ **by**(*auto simp add*: *bucket-ok-def*)
ultimately have $i = j$ **by** *simp*
with ‹*xs* ! $i = x$› ‹*xs* ! $j = y$› **show** $x = y$ **by** *simp*
qed
ultimately show *distinct* [*ys*←*map* (*map fst*) *xs* . *ys* $\neq$ []]
by(*simp add*: *filter-map o-def distinct-map*)
next
fix *ys*
have *A*: $\bigwedge$*xs. distinct* (*map fst xs*) = *list-map-invar xs*
by (*simp add*: *list-map-invar-def*)
assume *ys* $\in$ *set* (*map* (*map fst*) *xs*)
thus *distinct ys*
by(*clarsimp simp add*: *set-conv-nth A*) (*erule no-junk*(*2*))
next
fix *ys zs*
assume *ys* $\in$ *set* (*map* (*map fst*) *xs*) *zs* $\in$ *set* (*map* (*map fst*) *xs*) *ys* $\neq$ *zs*
then obtain *ys' zs'* **where** [*simp*]: *ys* = *map fst ys'* *zs* = *map fst zs'*
and *ys'* $\in$ *set xs* *zs'* $\in$ *set xs* **by** *auto*
have *fst* ‘ *set ys'* $\cap$ *fst* ‘ *set zs'* = {}
proof(*rule equals0I*)
fix *k*
assume $k \in$ *fst* ‘ *set ys'* $\cap$ *fst* ‘ *set zs'*
then obtain *v v'* **where** $(k, v) \in$ *set ys'* $(k, v') \in$ *set zs'* **by**(*auto*)
from ‹*ys'* $\in$ *set xs*› **obtain** *i* **where** *xs* ! i = *ys'* $i <$ *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v) \in$ *set ys'*› **have** *bhc* (*length xs*) $k = i$ **by**(*auto dest*: *no-junk bucket-okD*)
moreover
from ‹*zs'* $\in$ *set xs*› **obtain** *j* **where** *xs* ! j = *zs'* $j <$ *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v') \in$ *set zs'*› **have** *bhc* (*length xs*) $k = j$ **by**(*auto dest*: *no-junk bucket-okD*)
ultimately have $i = j$ **by** *simp*
with ‹*xs* ! i = *ys'*› ‹*xs* ! j = *zs'*› **have** *ys'* = *zs'* **by** *simp*
with ‹*ys* $\neq$ *zs*› **show** *False* **by** *simp*
qed
thus *set ys* $\cap$ *set zs* = {} **by** *simp*
qed
qed

ahm-α

lemma *list-map-lookup-is-map-of*:
list-map-lookup (=) *k l* = *map-of l k*
using *list-map-autoref-lookup-aux*[**where** *eq*=(=) **and**
Rk=*Id* **and** *Rv*=*Id*] **by** *force*
definition *ahm-α-aux bhc a* $\equiv$

(*λk. ahm-lookup-aux* (=) *bhc k a*)
lemma *ahm-α-aux-def2*: *ahm-α-aux bhc a* = (*λk. map-of* (*array-get a* (*bhc* (*array-length a*) *k*)) *k*)
unfolding *ahm-α-aux-def ahm-lookup-aux-def*
by (*simp add*: *list-map-lookup-is-map-of*)
lemma *ahm-α-def2*: *ahm-α bhc* (*HashMap a n*) = *ahm-α-aux bhc a*
unfolding *ahm-α-def ahm-α-aux-def* **by** *simp*

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 −
have *dom* (*ahm-α-aux bhc a*) ⊆ (⋃*h* ∈ *range* (*bhc* (*array-length a*) :: *'a* ⇒ *nat*). *dom* (*map-of* (*array-get a h*)))
unfolding *ahm-α-aux-def2*
by(*force simp add*: *dom-map-of-conv-image-fst dest*: *map-of-SomeD*)
moreover have *finite* ...
proof(*rule finite-UN-I*)
from ‹*ahm-invar-aux bhc n a*› **have** *array-length a* > *1* **by**(*simp add*: *ahm-invar-aux-def*)
hence *range* (*bhc* (*array-length a*) :: *'a* ⇒ *nat*) ⊆ {*0*..<*array-length a*}
using *assms* **by** *force*
thus *finite* (*range* (*bhc* (*array-length a*) :: *'a* ⇒ *nat*))
by(*rule finite-subset*) *simp*
qed(*rule finite-dom-map-of*)
ultimately show *?thesis* **by**(*rule finite-subset*)
qed

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*
using *is-bounded-hashcodeD*(*3*)[*OF assms*]
unfolding *ahm-α-aux-def ahm-lookup-aux-def* **by** *simp*

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
fix *k*
show *ahm-α-aux bhc* (*Array xs*) *k* = *map-of* (*concat xs*) *k*
proof(*cases map-of* (*concat xs*) *k*)
case *None*
hence *k* ∉ *fst* ‘ *set* (*concat xs*) **by**(*simp add*: *map-of-eq-None-iff*)
hence *k* ∉ *fst* ‘ *set* (*xs* ! *bhc* (*length xs*) *k*)
proof(*rule contrapos-nn*)
assume *k* ∈ *fst* ‘ *set* (*xs* ! *bhc* (*length xs*) *k*)
then obtain *v* **where** (*k*, *v*) ∈ *set* (*xs* ! *bhc* (*length xs*) *k*) **by** *auto*
moreover from *inv* **have** *bhc* (*length xs*) *k* < *length xs*
using *bhc* **by** (*force simp*: *ahm-invar-aux-def*)

```
      ultimately show k ∈ fst ` set (concat xs)
        by (force intro: rev-image-eqI)
    qed
    thus ?thesis unfolding None ahm-α-aux-def2
        by (simp add: map-of-eq-None-iff)
  next
    case (Some v)
    hence (k, v) ∈ set (concat xs) by(rule map-of-SomeD)
    then obtain ys where ys ∈ set xs    (k, v) ∈ set ys
      unfolding set-concat by blast
    from ‹ys ∈ set xs› obtain i j where i < length xs    xs ! i = ys
      unfolding set-conv-nth by auto
    with inv ‹(k, v) ∈ set ys›
    show ?thesis unfolding Some
      unfolding ahm-α-aux-def2
      by(force dest: bucket-okD simp add: ahm-invar-aux-def list-map-invar-def)
  qed
qed

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(cases a)
  case [simp]: (Array xs)
  from inv have card (dom (ahm-α-aux bhc (Array xs))) = card (dom (map-of
(concat xs)))
    by(simp add: ahm-α-aux-conv-map-of-concat[OF bhc])
  also from inv have distinct (map fst (concat xs))
    by(simp add: ahm-invar-distinct-fst-concatD)
  hence card (dom (map-of (concat xs))) = length (concat xs)
    by(rule card-dom-map-of)
  also have length (concat xs) = foldl (+) 0 (map length xs)
   by (simp add: length-concat foldl-conv-fold add.commute fold-plus-sum-list-rev)
  also from inv
  have ... = n unfolding foldl-map by(simp add: ahm-invar-aux-def array-foldl-foldl)
  finally show ?thesis by(simp)
qed

lemma finite-dom-ahm-α:
  assumes is-bounded-hashcode Id (=) bhc    ahm-invar bhc hm
  shows finite (dom (ahm-α bhc hm))
  using assms by (cases hm, force intro: finite-dom-ahm-α-aux
      simp: ahm-α-def2)
```

ahm-empty

```
lemma ahm-invar-aux-new-array:
  assumes n > 1
```

shows *ahm-invar-aux bhc 0* (*new-array* [] *n*)
proof −
have *foldl* (λb (k, v). b + *length* v) *0* (*zip* [*0*..<*n*] (*replicate n* [])) = *0*
by(*induct n*)(*simp-all add*: *replicate-Suc-conv-snoc del*: *replicate-Suc*)
with *assms* **show** *?thesis* **by**(*simp add*: *ahm-invar-aux-def array-foldl-new-array list-map-invar-def*)
qed

lemma *ahm-invar-new-hashmap-with*:
$n > 1 \Longrightarrow$ *ahm-invar bhc* (*new-hashmap-with n*)
by(*auto simp add*: *ahm-invar-def new-hashmap-with-def intro*: *ahm-invar-aux-new-array*)

lemma *ahm-α-new-hashmap-with*:
assumes *is-bounded-hashcode Id* (=) *bhc* **and** $n > 1$
shows *Map.empty* = *ahm-α bhc* (*new-hashmap-with n*)
unfolding *new-hashmap-with-def ahm-α-def*
using *is-bounded-hashcodeD*(*3*)[*OF assms*] **by** *force*

lemma *ahm-empty-impl*:
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
assumes *def-size*: *def-size* > *1*
shows (*ahm-empty def-size*, *Map.empty*) ∈ *ahm-map-rel′ bhc*
proof−
from *def-size* **and** *ahm-α-new-hashmap-with*[*OF bhc def-size*] **and**
ahm-invar-new-hashmap-with[*OF def-size*]
show *?thesis* **unfolding** *ahm-empty-def ahm-map-rel′-def br-def* **by** *force*
qed

lemma *param-ahm-empty*[*param*]:
assumes *def-size*: (*def-size*, *def-size′*) ∈ *nat-rel*
shows (*ahm-empty def-size* ,*ahm-empty def-size′*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
unfolding *ahm-empty-def*[*abs-def*] *new-hashmap-with-def*[*abs-def*]
new-array-def[*abs-def*]
using *assms* **by** *parametricity*

lemma *autoref-ahm-empty*[*autoref-rules*]:
fixes *Rk* :: (*′kc*×*′ka*) *set*
assumes *bhc*: *SIDE-GEN-ALGO* (*is-bounded-hashcode Rk eq bhc*)
assumes *def-size*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*′kc*) *def-size*)
shows (*ahm-empty def-size*, *op-map-empty*) ∈ ⟨*Rk*, *Rv*⟩*ahm-rel bhc*
proof−
from *bhc* **have** *eq′*: (*eq*,(=)) ∈ *Rk* → *Rk* → *bool-rel*
by (*simp add*: *is-bounded-hashcodeD*)
with *bhc* **have** *is-bounded-hashcode Id* (=)
(*abstract-bounded-hashcode Rk bhc*)
unfolding *autoref-tag-defs*
by *blast*
thus *?thesis* **using** *assms*

  unfolding *op-map-empty-def*
  unfolding *ahm-rel-def is-valid-def-hm-size-def autoref-tag-defs*
  apply (*intro relcompI*)
  apply (*rule param-ahm-empty*[*of def-size def-size*], *simp*)
  apply (*blast intro*: *ahm-empty-impl*)
  done
qed

ahm-lookup

lemma *param-ahm-lookup*[*param*]:
  assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
  defines *bhc′-def*: *bhc′* ≡ *abstract-bounded-hashcode Rk bhc*
  assumes *inv*: *ahm-invar bhc′ m′*
  assumes *K*: (*k*,*k′*) ∈ *Rk*
  assumes *M*: (*m*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
  shows (*ahm-lookup eq bhc k m*, *ahm-lookup* (=) *bhc′ k′ m′*) ∈
      ⟨*Rv*⟩*option-rel*
proof−
  from *bhc* **have** *eq′*: (*eq*,(=)) ∈ *Rk* → *Rk* → *bool-rel* **by** (*simp add*: *is-bounded-hashcodeD*)
  moreover from *abstract-bhc-correct*[*OF bhc*]
    have *bhc′*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel* **unfolding** *bhc′-def* **.**
  moreover from *M* **obtain** *a a′ n n′* **where**
    [*simp*]: *m* = *HashMap a n* **and** [*simp*]: *m′* = *HashMap a′ n′* **and**
    A: (*a*,*a′*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* **and** *N*: (*n*,*n′*) ∈ *Id*
      by (*cases m*, *cases m′*, *unfold ahm-map-rel-def*, *auto*)
  moreover from *inv* **and** *array-rel-imp-same-length*[*OF A*]
    have *array-length a* > *1* **by** (*simp add*: *ahm-invar-aux-def*)
  with *abstract-bhc-is-bhc*[*OF bhc*]
    have *bhc′* (*array-length a*) *k′* < *array-length a*
    unfolding *bhc′-def* **by** *blast*
  with *bhc′*[*param-fo*, *OF - K*]
    have *bhc* (*array-length a*) *k* < *array-length a* **by** *simp*
  ultimately show *?thesis* **using** *K*
    unfolding *ahm-lookup-def*[*abs-def*] *rec-hashmap-is-case*
    by (*simp*, *parametricity*)
qed

lemma *ahm-lookup-impl*:
  assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
  shows (*ahm-lookup* (=) *bhc*, *op-map-lookup*) ∈ *Id* → *ahm-map-rel′ bhc* → *Id*
unfolding *ahm-map-rel′-def br-def ahm-α-def* **by** *force*

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

$\in$ *Rk* $\rightarrow$ ⟨*Rk*,*Rv*⟩*ahm-rel bhc* $\rightarrow$ ⟨*Rv*⟩*option-rel*
proof (*intro fun-relI*)
  let *?bhc′* = *abstract-bounded-hashcode Rk bhc*
  fix *k k′ a m′*
  assume *K*: (*k*,*k′*) $\in$ *Rk*
  assume *M*: (*a*,*m′*) $\in$ ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
  from *bhc* **have** *bhc′*: *is-bounded-hashcode Id* (=) *?bhc′*
    by *blast*

  from *M* **obtain** *a′* **where** *M1*: (*a*,*a′*) $\in$ ⟨*Rk*,*Rv*⟩*ahm-map-rel* **and**
    M2: (*a′*,*m′*) $\in$ *ahm-map-rel′ ?bhc′* **unfolding** *ahm-rel-def* **by** *blast*
  hence *inv*: *ahm-invar ?bhc′ a′*
    unfolding *ahm-map-rel′-def br-def* **by** *simp*

  from *relcompI*[*OF param-ahm-lookup*[*OF bhc inv K M1*]
          ahm-lookup-impl[*param-fo*, *OF bhc′ - M2*]]
  show (*ahm-lookup eq bhc k a*, *op-map-lookup k′ m′*) $\in$ ⟨*Rv*⟩*option-rel*
    by *simp*
qed

ahm-iteratei

abbreviation *ahm-to-list* $\equiv$ *it-to-list ahm-iteratei*

lemma *param-ahm-iteratei-aux*[*param*]:
  (*ahm-iteratei-aux*,*ahm-iteratei-aux*) $\in$ ⟨⟨*Ra*⟩*list-rel*⟩*array-rel* $\rightarrow$
    (*Rb* $\rightarrow$ *bool-rel*) $\rightarrow$ (*Ra* $\rightarrow$ *Rb* $\rightarrow$ *Rb*) $\rightarrow$ *Rb* $\rightarrow$ *Rb*
unfolding *ahm-iteratei-aux-def*[*abs-def*] **by** *parametricity*

lemma *param-ahm-iteratei*[*param*]:
  (*ahm-iteratei*,*ahm-iteratei*) $\in$ ⟨*Rk*,*Rv*⟩*ahm-map-rel* $\rightarrow$
    (*Rb* $\rightarrow$ *bool-rel*) $\rightarrow$ (⟨*Rk*,*Rv*⟩*prod-rel* $\rightarrow$ *Rb* $\rightarrow$ *Rb*) $\rightarrow$ *Rb* $\rightarrow$ *Rb*
unfolding *ahm-iteratei-def*[*abs-def*] *rec-hashmap-is-case* **by** *parametricity*

lemma *param-ahm-to-list*[*param*]:
  (*ahm-to-list*,*ahm-to-list*) $\in$
    ⟨*Rk*, *Rv*⟩*ahm-map-rel* $\rightarrow$ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
unfolding *it-to-list-def*[*abs-def*] **by** *parametricity*

lemma *ahm-to-list-distinct*[*simp*,*intro*]:
  assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
  assumes *inv*: *ahm-invar bhc m*
  shows *distinct* (*ahm-to-list m*)
proof−
  obtain *n a* **where** [*simp*]: *m* = *HashMap a n* **by** (*cases m*)
  obtain *l* **where** [*simp*]: *a* = *Array l* **by** (*cases a*)
  from *inv* **show** *distinct* (*ahm-to-list m*) **unfolding** *it-to-list-def*
    by (*force intro*: *distinct-mapI dest*: *ahm-invar-distinct-fst-concatD*)
qed

lemma *set-ahm-to-list*:
  assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
  assumes *ref*: $(m,m') \in$ *ahm-map-rel'* *bhc*
  shows *map-to-set* m' = *set* (*ahm-to-list* m)
proof−
  obtain n a **where** [*simp*]: m = *HashMap* a n **by** (*cases* m)
  obtain l **where** [*simp*]: a = *Array* l **by** (*cases* a)
  from *ref* **have** α[*simp*]: m' = *ahm-*α *bhc* m **and**
    inv: *ahm-invar bhc* m
    unfolding *ahm-map-rel'-def br-def* **by** *auto*

  from *inv* **have** *length*: *length* $l > 1$
    unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*
  from *inv* **have** *buckets-ok*: $\bigwedge h\ x.\ h <$ *length* $l \Longrightarrow x \in$ *set* $(l!h) \Longrightarrow$
    bhc (*length* l) (*fst* x) = h
    $\bigwedge h.\ h <$ *length* $l \Longrightarrow$ *distinct* (*map fst* $(l!h)$)
    by (*simp-all add*: *ahm-invar-def ahm-invar-aux-def*
        bucket-ok-def list-map-invar-def)

  show *?thesis* **unfolding** *it-to-list-def* α *ahm-*α*-def ahm-iteratei-def*
    apply (*simp add*: *list-map-lookup-is-map-of*)
  proof (*intro equalityI subsetI*, *goal-cases*)
    case *prems*: (*1* x)
    let *?m* = λk. *map-of* (l ! *bhc* (*length* l) k) k
    obtain k v **where** [*simp*]: $x = (k, v)$ **by** (*cases* x)
    from *prems* **have** *set-to-map* (*map-to-set ?m*) k = *Some* v
      by (*simp add*: *set-to-map-simp inj-on-fst-map-to-set*)
    also note *map-to-set-inverse*
    finally have *map-of* (l ! *bhc* (*length* l) k) k = *Some* v **.**
    hence $(k,v) \in$ *set* (l ! *bhc* (*length* l) k)
      by (*simp add*: *map-of-SomeD*)
    moreover have *bhc* (*length* l) $k <$ *length* l **using** *bhc length* **..**
    ultimately show *?case* **by** *force*
  next
    case *prems*: (*2* x)
    obtain k v **where** [*simp*]: $x = (k, v)$ **by** (*cases* x)
    from *prems* **obtain** h **where** *h-props*: $(k,v) \in$ *set* $(l!h)$ $h <$ *length* l
      by (*force simp*: *set-conv-nth*)
    moreover from *h-props* **and** *buckets-ok*
    have *bhc* (*length* l) $k = h$ *distinct* (*map fst* $(l!h)$) **by** *auto*
    ultimately have *map-of* (l ! *bhc* (*length* l) k) k = *Some* v
      by (*force intro*: *map-of-is-SomeI*)
    thus *?case* **by** *simp*
  qed
qed

lemma *ahm-iteratei-aux-impl*:
  assumes *inv*: *ahm-invar-aux bhc n a*
  and *bhc*: *is-bounded-hashcode Id* (=) *bhc*
  shows *map-iterator* (*ahm-iteratei-aux a*) (*ahm-α-aux bhc a*)
proof (*cases a*, *rule*)
  fix *xs* **assume** [*simp*]: *a* = *Array xs*
  show *ahm-iteratei-aux a* = *foldli* (*concat xs*)
    by (*intro ext*, *simp*)
  from *ahm-invar-distinct-fst-concatD* **and** *inv*
    show *distinct* (*map fst* (*concat xs*)) **by** *simp*
  from *ahm-α-aux-conv-map-of-concat* **and** *assms*
    show *ahm-α-aux bhc a* = *map-of* (*concat xs*) **by** *simp*
qed

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*)
  by (*insert assms*, *cases m*, *simp add*: *ahm-α-def2*,
      erule (*1*) *ahm-iteratei-aux-impl*)

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*
  unfolding *is-map-to-list-def is-map-to-sorted-list-def*
proof (*intro allI impI*)
  let *?bhc′* = *abstract-bounded-hashcode Rk bhc*
  fix *a m′* **assume** *M*: (*a*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
  from *bhc* **have** *bhc′*: *is-bounded-hashcode Id* (=) *?bhc′*
   unfolding *autoref-tag-defs*
   apply (*rule-tac abstract-bhc-is-bhc*)
   by *simp-all*

  from *M* **obtain** *a′* **where** *M1*: (*a*,*a′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* **and**
    M2: (*a′*,*m′*) ∈ *ahm-map-rel′ ?bhc′* **unfolding** *ahm-rel-def* **by** *blast*
  hence *inv*: *ahm-invar ?bhc′ a′*
    unfolding *ahm-map-rel′-def br-def* **by** *simp*

  let *?l′* = *ahm-to-list a′*
  from *param-ahm-to-list*[*param-fo*, *OF M1*]
    have (*ahm-to-list a*, *?l′*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* **.**
  moreover from *ahm-to-list-distinct*[*OF bhc′ inv*]
    have *distinct* (*ahm-to-list a′*) **.**
  moreover from *set-ahm-to-list*[*OF bhc′ M2*]
    have *map-to-set m′* = *set* (*ahm-to-list a′*) **.**
  ultimately show ∃ *l′*. (*ahm-to-list a*, *l′*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* ∧

RETURN l′ ≤ it-to-sorted-list
(*key-rel* (λ- -. *True*)) (*map-to-set m′*)
by (*force simp*: *it-to-sorted-list-def key-rel-def*[*abs-def*])
qed

lemma *ahm-iteratei-aux-code*[*code*]:
ahm-iteratei-aux a c f σ = *idx-iteratei array-get array-length a c*
(λ*x*. *foldli x c f*) *σ*
proof(*cases a*)
case [*simp*]: (*Array xs*)
have *ahm-iteratei-aux a c f σ* = *foldli* (*concat xs*) *c f σ* **by** *simp*
also have ... = *foldli xs c* (λ*x*. *foldli x c f*) *σ* **by** (*simp add*: *foldli-concat*)
also have ... = *idx-iteratei* (!) *length xs c* (λ*x*. *foldli x c f*) *σ*
by (*simp add*: *idx-iteratei-nth-length-conv-foldli*)
also have ... = *idx-iteratei array-get array-length a c* (λ*x*. *foldli x c f*) *σ*
by(*simp add*: *idx-iteratei-array-get-Array-conv-nth*)
finally show *?thesis* **.**
qed

ahm-rehash

lemma *array-length-ahm-rehash-aux′*:
array-length (*ahm-rehash-aux′ bhc n kv a*) = *array-length a*
by(*simp add*: *ahm-rehash-aux′-def Let-def*)

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
assumes *inv*: *ahm-invar-aux bhc n a*
and *bhc*: *is-bounded-hashcode Id* (=) *bhc*
and *fresh*: *k* ∉ *fst* ‘ *set* (*array-get a* (*bhc* (*array-length a*) *k*))
shows *ahm-invar-aux bhc* (*Suc n*) (*ahm-rehash-aux′ bhc* (*array-length a*) (*k*, *v*)
a)
(**is** *ahm-invar-aux bhc - ?a*)
proof(*rule ahm-invar-auxI*)
note *invD* = *ahm-invar-auxD*[*OF inv*]
let *?l* = *array-length a*
fix *h*
assume *h* < *array-length ?a*
hence *hlen*: *h* < *?l* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
from *invD*(*1*,*2*)[*OF this*] **have** *bucket*: *bucket-ok bhc ?l h* (*array-get a h*)
and *dist*: *distinct* (*map fst* (*array-get a h*))
by (*simp-all add*: *list-map-invar-def*)
let *?h* = *bhc* (*array-length a*) *k*
from *hlen bucket* **show** *bucket-ok bhc* (*array-length ?a*) *h* (*array-get ?a h*)
by(*cases h* = *?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-length-ahm-rehash-aux′*
array-get-array-set-other dest: *bucket-okD intro*!: *bucket-okI*)
from *dist hlen fresh*
show *list-map-invar* (*array-get ?a h*)
unfolding *list-map-invar-def*

by(*cases h = ?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-get-array-set-other*)
next
let *?f* = λ*n kvs. n + length kvs*
{ fix *n* :: *nat* **and** *xs* :: (′*a* × ′*b*) *list list*
have *foldl ?f n xs = n + foldl ?f 0 xs*
by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
note *fold = this*
let *?h = bhc* (*array-length a*) *k*

obtain *xs* **where** *a* [*simp*]: *a = Array xs* **by**(*cases a*)
from *inv* **and** *bhc* **have** [*simp*]: *bhc* (*length xs*) *k < length xs*
by (*force simp add*: *ahm-invar-aux-def*)
have *xs*: *xs = take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs* **by**(*simp add*: *Cons-nth-drop-Suc*)
from *inv* **have** *n = array-foldl* (λ- *n kvs. n + length kvs*) *0 a*
by(*auto elim*: *ahm-invar-auxE*)
hence *n = foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc ?h*) *xs*)
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
thus *Suc n = array-foldl* (λ- *n kvs. n + length kvs*) *0 ?a*
by(*simp add*: *ahm-rehash-aux′-def Let-def array-foldl-foldl foldl-list-update*)(*subst* (*1 2 3 4*) *fold*, *simp*)
next
from *inv* **have** *1 < array-length a* **by**(*auto elim*: *ahm-invar-auxE*)
thus *1 < array-length ?a* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
qed

lemma *ahm-rehash-aux-correct*:
fixes *a* :: (′*k*×′*v*) *list array*
assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
and *inv*: *ahm-invar-aux bhc n a*
and *sz* > *1*
shows *ahm-invar-aux bhc n* (*ahm-rehash-aux bhc a sz*) (**is** *?thesis1*)
and *ahm-α-aux bhc* (*ahm-rehash-aux bhc a sz*) = *ahm-α-aux bhc a* (**is** *?thesis2*)
proof −
let *?a = ahm-rehash-aux bhc a sz*
define *I* **where** *I it a′* ⟷
ahm-invar-aux bhc (*n − card it*) *a′*
∧ *array-length a′ = sz*
∧ (∀ *k. if k* ∈ *it then*
ahm-α-aux bhc a′ k = None
else ahm-α-aux bhc a′ k = ahm-α-aux bhc a k) **for** *it a′*

note *iterator-rule = map-iterator-no-cond-rule-P*[
OF ahm-iteratei-aux-impl[*OF inv bhc*],
of I new-array [] *sz ahm-rehash-aux′ bhc sz I* {}]

from *inv* **have** *I* {} *?a* **unfolding** *ahm-rehash-aux-def*
proof(*intro iterator-rule*)
  from *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF bhc inv*]
    have *card* (*dom* (*ahm-α-aux bhc a*)) = *n* **.**
  moreover from *ahm-invar-aux-new-array*[*OF* ‹*1* < *sz*›]
    have *ahm-invar-aux bhc 0* (*new-array* ([]::(′*k*×′*v*) *list*) *sz*) **.**
  moreover {
   fix *k*
   assume *k* ∉ *dom* (*ahm-α-aux bhc a*)
   hence *ahm-α-aux bhc a k* = *None* **by** *auto*
   hence *ahm-α-aux bhc* (*new-array* [] *sz*) *k* = *ahm-α-aux bhc a k*
     using *assms* **by** *simp*
  }
  ultimately show *I* (*dom* (*ahm-α-aux bhc a*)) (*new-array* [] *sz*)
    using *assms* **by** (*simp add*: *I-def*)
next
  fix *k* :: ′*k*
   and *v* :: ′*v*
   and *it a*′
  assume *k* ∈ *it*   *ahm-α-aux bhc a k* = *Some v*
   and *it-sub*: *it* ⊆ *dom* (*ahm-α-aux bhc a*)
   and *I*: *I it a*′
  from *I* **have** *inv*′: *ahm-invar-aux bhc* (*n* − *card it*) *a*′
   and *a*′*-eq-a*: ⋀*k*. *k* ∉ *it* ⟹ *ahm-α-aux bhc a*′ *k* = *ahm-α-aux bhc a k*
   and *a*′*-None*: ⋀*k*. *k* ∈ *it* ⟹ *ahm-α-aux bhc a*′ *k* = *None*
   and [*simp*]: *sz* = *array-length a*′
   by (*auto split*: *if-split-asm simp*: *I-def*)
  from *it-sub finite-dom-ahm-α-aux*[*OF bhc inv*]
    have *finite it* **by**(*rule finite-subset*)
  moreover with ‹*k* ∈ *it*› **have** *card it* > *0* **by** (*auto simp add*: *card-gt-0-iff*)
  moreover from *finite-dom-ahm-α-aux*[*OF bhc inv*] *it-sub*
    have *card it* ≤ *card* (*dom* (*ahm-α-aux bhc a*)) **by** (*rule card-mono*)
  moreover have ... = *n* **using** *inv*
    by(*simp add*: *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF bhc*])
  ultimately have *n* − *card* (*it* − {*k*}) = (*n* − *card it*) + *1*
    using ‹*k* ∈ *it*› **by** *auto*
  moreover from ‹*k* ∈ *it*› **have** *ahm-α-aux bhc a*′ *k* = *None* **by** (*rule a*′*-None*)
  hence *k* ∉ *fst* ‘ *set* (*array-get a*′ (*bhc* (*array-length a*′) *k*))
    by (*simp add*: *ahm-α-aux-def2 map-of-eq-None-iff*)
  ultimately have *ahm-invar-aux bhc* (*n* − *card* (*it* − {*k*}))
    (*ahm-rehash-aux*′ *bhc sz* (*k*, *v*) *a*′)
    using *ahm-rehash-aux*′*-preserves-ahm-invar-aux*[*OF inv*′ *bhc*] **by** *simp*
  moreover have *array-length* (*ahm-rehash-aux*′ *bhc sz* (*k*, *v*) *a*′) = *sz*
    by (*simp add*: *array-length-ahm-rehash-aux*′)
  moreover {
   fix *k*′
   assume *k*′ ∈ *it* − {*k*}
   with *is-bounded-hashcodeD*(*3*)[*OF bhc* ‹*1* < *sz*›, *of k*′] *a*′*-None*[*of k*′]
   have *ahm-α-aux bhc* (*ahm-rehash-aux*′ *bhc sz* (*k*, *v*) *a*′) *k*′ = *None*

```
        unfolding ahm-α-aux-def2
        by (cases bhc sz k = bhc sz k') (simp-all add:
              array-get-array-set-other ahm-rehash-aux'-def Let-def)
  } moreover {
    fix k'
    assume k' ∉ it − {k}
    with is-bounded-hashcodeD(3)[OF bhc ‹1 < sz›, of k]
         is-bounded-hashcodeD(3)[OF bhc ‹1 < sz›, of k']
         a'-eq-a[of k'] ‹k ∈ it›
    have ahm-α-aux bhc (ahm-rehash-aux' bhc sz (k, v) a') k' =
             ahm-α-aux bhc a k'
        unfolding ahm-rehash-aux'-def Let-def
        using ‹ahm-α-aux bhc a k = Some v›
        unfolding ahm-α-aux-def2
      by(cases bhc sz k = bhc sz k') (case-tac [!]     k' = k,
         simp-all add: array-get-array-set-other)
  }
  ultimately show I (it − {k}) (ahm-rehash-aux' bhc sz (k, v) a')
      unfolding I-def by simp
 qed simp-all
 thus ?thesis1 ?thesis2 unfolding ahm-rehash-aux-def I-def by auto
qed

lemma ahm-rehash-correct:
 fixes hm :: ('k, 'v) hashmap
 assumes bhc: is-bounded-hashcode Id (=) bhc
 and inv: ahm-invar bhc hm
 and sz > 1
 shows ahm-invar bhc (ahm-rehash bhc hm sz)
       ahm-α bhc (ahm-rehash bhc hm sz) = ahm-α bhc hm
proof−
 obtain a n where [simp]: hm = HashMap a n by (cases hm)
 from inv have ahm-invar-aux bhc n a by simp
 from ahm-rehash-aux-correct[OF bhc this ‹sz > 1›]
     show ahm-invar bhc (ahm-rehash bhc hm sz) and
          ahm-α bhc (ahm-rehash bhc hm sz) = ahm-α bhc hm
     by (simp-all add: ahm-α-def2)
qed
```

ahm-update

```
lemma param-hm-grow[param]:
 (hm-grow, hm-grow) ∈ ⟨Rk,Rv⟩ahm-map-rel → nat-rel
unfolding hm-grow-def[abs-def] rec-hashmap-is-case by parametricity

lemma param-ahm-rehash-aux'[param]:
 assumes is-bounded-hashcode Rk eq bhc
 assumes 1 < n
 assumes (bhc,bhc') ∈ nat-rel → Rk → nat-rel
```

assumes $(n,n') \in$ *nat-rel* **and** $n =$ *array-length a*
assumes $(kv,kv') \in \langle Rk,Rv\rangle$*prod-rel*
assumes $(a,a') \in \langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*
shows (*ahm-rehash-aux′ bhc n kv a*, *ahm-rehash-aux′ bhc′ n′ kv′ a′*) $\in$
$\langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*
proof −
from *assms* **have** *bhc n* (*fst kv*) $<$ *array-length a* **by** *force*
thus *?thesis* **unfolding** *ahm-rehash-aux′-def*[*abs-def*]
rec-hashmap-is-case Let-def **using** *assms* **by** *parametricity*
qed

lemma *param-new-array*[*param*]:
(*new-array*, *new-array*) $\in R \to$ *nat-rel* $\to \langle R\rangle$*array-rel*
unfolding *new-array-def*[*abs-def*] **by** *parametricity*

lemma *param-foldli-induct*:
assumes *l*: $(l,l') \in \langle Ra\rangle$*list-rel*
assumes *c*: $(c,c') \in Rb \to$ *bool-rel*
assumes σ: $(\sigma,\sigma') \in Rb$
assumes $P\sigma$: $P\ \sigma\ \sigma'$
assumes *f*: $\bigwedge a\ a'\ b\ b'.\ (a,a')\in Ra \Longrightarrow (b,b')\in Rb \Longrightarrow c\ b \Longrightarrow c'\ b' \Longrightarrow$
$P\ b\ b' \Longrightarrow (f\ a\ b,\ f'\ a'\ b') \in Rb\ \wedge$
$P\ (f\ a\ b)\ (f'\ a'\ b')$
shows (*foldli l c f* σ, *foldli l′ c′ f′* σ') $\in Rb$
using $c\ \sigma\ P\sigma\ f$ **by** (*induction arbitrary*: $\sigma\ \sigma'$ *rule*: *list-rel-induct*[*OF l*],
auto dest!: *fun-relD*)

lemma *param-foldli-induct-nocond*:
assumes *l*: $(l,l') \in \langle Ra\rangle$*list-rel*
assumes σ: $(\sigma,\sigma') \in Rb$
assumes $P\sigma$: $P\ \sigma\ \sigma'$
assumes *f*: $\bigwedge a\ a'\ b\ b'.\ (a,a')\in Ra \Longrightarrow (b,b')\in Rb \Longrightarrow P\ b\ b' \Longrightarrow$
$(f\ a\ b,\ f'\ a'\ b') \in Rb \wedge P\ (f\ a\ b)\ (f'\ a'\ b')$
shows (*foldli l* (λ-. *True*) *f* σ, *foldli l′* (λ-. *True*) *f′* σ') $\in Rb$
using *assms* **by** (*blast intro*: *param-foldli-induct*)

lemma *param-ahm-rehash-aux*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc-rel*: $(bhc,bhc') \in$ *nat-rel* $\to Rk \to$ *nat-rel*
assumes *A*: $(a,a') \in \langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*
assumes *N*: $(n,n') \in$ *nat-rel* $\quad 1 < n$
shows (*ahm-rehash-aux bhc a n*, *ahm-rehash-aux bhc′ a′ n′*) $\in$
$\langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*
proof −
obtain *l l′* **where** [*simp*]: $a =$ *Array l* $\quad a' =$ *Array l′*
by (*cases a*, *cases a′*)
from *A* **have** *L*: $(l,l') \in \langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*list-rel*

unfolding *array-rel-def* **by** *simp*
hence L': (*concat l*, *concat l'*) $\in$ $\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*
by *parametricity*
let $?P = \lambda a\ a'.\ n = $ *array-length a*

note *induct-rule* = *param-foldli-induct-nocond*[*OF* L', **where** P=$?P$]

show *?thesis* **unfolding** *ahm-rehash-aux-def*
by (*simp*, *induction rule*: *induct-rule*, *insert N bhc bhc-rel*,
auto intro: *param-new-array*[*param-fo*]
param-ahm-rehash-aux'[*param-fo*]
simp: *array-length-ahm-rehash-aux'*)
qed

lemma *param-ahm-rehash*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc-rel*: (*bhc*,*bhc'*) $\in$ *nat-rel* $\rightarrow$ *Rk* $\rightarrow$ *nat-rel*
assumes *M*: (*m*,*m'*) $\in$ $\langle Rk,Rv\rangle$*ahm-map-rel*
assumes *N*: (*n*,*n'*) $\in$ *nat-rel* $1 < n$
shows (*ahm-rehash bhc m n*, *ahm-rehash bhc' m' n'*) $\in$
$\langle Rk,Rv\rangle$*ahm-map-rel*
proof−
obtain *a a' k k'* **where** [*simp*]: *m* = *HashMap a k* *m'* = *HashMap a' k'*
by (*cases m*, *cases m'*)
hence *K*: (*k*,*k'*) $\in$ *nat-rel* **and**
A: (*a*,*a'*) $\in$ $\langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*
using *M* **unfolding** *ahm-map-rel-def* **by** *simp-all*
show *?thesis* **unfolding** *ahm-rehash-def*
by (*simp*, *insert K A assms*, *parametricity*)
qed

lemma *param-load-factor*[*param*]:
(*load-factor*, *load-factor*) $\in$ *nat-rel*
unfolding *load-factor-def* **by** *simp*

lemma *param-ahm-filled*[*param*]:
(*ahm-filled*, *ahm-filled*) $\in$ $\langle Rk,Rv\rangle$*ahm-map-rel* $\rightarrow$ *bool-rel*
unfolding *ahm-filled-def*[*abs-def*] *rec-hashmap-is-case*
by *parametricity*

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

ahm-update-aux $(=)$ *bhc′ m′ k′ v′* $) \in \langle Rk,Rv\rangle$*ahm-map-rel*
proof−
from *bhc* **have** *eq*[*param*]: $(eq, (=)) \in Rk \to Rk \to$ *bool-rel* **by** (*simp add*: *is-bounded-hashcodeD*)
obtain *a a′ n n′* **where**
[*simp*]: m = *HashMap a n* **and** [*simp*]: m' = *HashMap a′ n′*
by (*cases m*, *cases m′*)
from *M* **have** *A*: $(a,a') \in \langle\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel* **and**
N: $(n,n') \in$ *nat-rel*
unfolding *ahm-map-rel-def* **by** *simp-all*

from *inv* **have** *1* < *array-length a′*
unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*
hence *1* < *array-length a*
by (*simp add*: *array-rel-imp-same-length*[*OF A*])
with *bhc* **have** *bhc-range*: *bhc* (*array-length a*) *k* < *array-length a* **by** *blast*

have *option-compare*: $\bigwedge a\ a'.\ (a,a') \in \langle Rv\rangle$*option-rel* $\Longrightarrow$
$(a = None, a' = None) \in$ *bool-rel* **by** *force*
have (*array-get a* (*bhc* (*array-length a*) *k*),
array-get a′ (*bhc′* (*array-length a′*) *k′*)) $\in$
$\langle\langle Rk,Rv\rangle$*prod-rel*$\rangle$*list-rel*
using *A K bhc-rel bhc-range* **by** *parametricity*
note *cmp* = *option-compare*[*OF param-list-map-lookup*[*param-fo*, *OF eq K this*]]

show *?thesis* **apply** *simp*
unfolding *ahm-update-aux-def Let-def rec-hashmap-is-case*
using *assms A N bhc-range cmp* **by** *parametricity*
qed

lemma *param-ahm-update*[*param*]:
assumes *bhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc-rel*: $(bhc,bhc') \in$ *nat-rel* $\to Rk \to$ *nat-rel*
assumes *inv*: *ahm-invar bhc′ m′*
assumes *K*: $(k,k') \in Rk$
assumes *V*: $(v,v') \in Rv$
assumes *M*: $(m,m') \in \langle Rk,Rv\rangle$*ahm-map-rel*
shows (*ahm-update eq bhc k v m*, *ahm-update* $(=)$ *bhc′ k′ v′ m′*) $\in$
$\langle Rk,Rv\rangle$*ahm-map-rel*
proof−
have *1* < *hm-grow* (*ahm-update-aux eq bhc m k v*) **by** *simp*
with *assms* **show** *?thesis* **unfolding** *ahm-update-def*[*abs-def*] *Let-def*
by *parametricity*
qed

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 (*cases list-map-lookup* (=) *k xs*, *simp-all*)
  case *None*
    hence $k \notin$ *dom* (*map-of xs*) **by** (*force simp*: *list-map-lookup-is-map-of*)
    hence $\bigwedge a$. *list-map-update-aux* (=) *k v xs a* = (*k*,*v*) # *rev xs* @ *a*
      by (*induction xs*, *auto*)
    thus *?l-new* = *Suc* (*length xs*) **unfolding** *list-map-update-def* **by** *simp*
next
  case (*Some v′*)
    hence (*k*,*v′*) ∈ *set xs* **unfolding** *list-map-lookup-is-map-of*
      by (*rule map-of-SomeD*)
    hence $\bigwedge a$. *length* (*list-map-update-aux* (=) *k v xs a*) =
      length xs + *length a* **by** (*induction xs*, *auto*)
    thus *?l-new* = *length xs* **unfolding** *list-map-update-def* **by** *simp*
qed

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 (*cases list-map-lookup* (=) *k xs*, *simp-all*)
  case *None*
    hence $k \notin$ *dom* (*map-of xs*) **by** (*force simp*: *list-map-lookup-is-map-of*)
    hence $\bigwedge a$. *list-map-delete-aux* (=) *k xs a* = *rev xs* @ *a*
      by (*induction xs*, *auto*)
    thus *?l-new* = *length xs* **unfolding** *list-map-delete-def* **by** *simp*
next
  case (*Some v′*)
    hence (*k*,*v′*) ∈ *set xs* **unfolding** *list-map-lookup-is-map-of*
      by (*rule map-of-SomeD*)
    hence $\bigwedge a$. $k \notin$ *fst‘set a* ⟹ *length* (*list-map-delete-aux* (=) *k xs a*) =
      length xs + *length a* − *1* **by** (*induction xs*, *auto*)
    thus *?l-new* = *length xs* − *Suc 0* **unfolding** *list-map-delete-def* **by** *simp*
qed

lemma *ahm-update-impl*:
  assumes *bhc*: *is-bounded-hashcode Id* (=) *bhc*
  shows (*ahm-update* (=) *bhc*, *op-map-update*) ∈ (*Id*::(′*k*×′*k*) *set*) →
        (*Id*::(′*v*×′*v*) *set*) → *ahm-map-rel′ bhc* → *ahm-map-rel′ bhc*
proof (*intro fun-relI*, *clarsimp*)
  fix *k*::′*k* **and** *v*::′*v* **and** *hm*::(′*k*,′*v*) *hashmap* **and** *m*::′*k*⇀′*v*
  assume (*hm*,*m*) ∈ *ahm-map-rel′ bhc*
  hence *α*: *m* = *ahm-α bhc hm* **and** *inv*: *ahm-invar bhc hm*
    unfolding *ahm-map-rel′-def br-def* **by** *simp-all*
  obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)

have *K*: (*k*,*k*) ∈ *Id* **and** *V*: (*v*,*v*) ∈ *Id* **by** *simp-all*

from *inv* **have** [*simp*]: *1* < *array-length a*
  unfolding *ahm-invar-def ahm-invar-aux-def* **by** *simp*
hence *bhc-range*[*simp*]: ⋀*k*. *bhc* (*array-length a*) *k* < *array-length a*
  using *bhc* **by** *blast*

let *?l* = *array-length a*
let *?h* = *bhc* (*array-length a*) *k*
let *?a′* = *array-set a ?h* (*list-map-update* (=) *k v* (*array-get a ?h*))
let *?n′* = *if list-map-lookup* (=) *k* (*array-get a ?h*) = *None*
      then n + *1 else n*

let *?list* = *array-get a* (*bhc ?l k*)
let *?list′* = *map-of ?list*
have *L*: (*?list*, *?list′*) ∈ *br map-of list-map-invar*
  using *inv* **unfolding** *ahm-invar-def ahm-invar-aux-def br-def* **by** *simp*
hence *list*: *list-map-invar ?list* **by** (*simp-all add*: *br-def*)
let *?list-new* = *list-map-update* (=) *k v ?list*
let *?list-new′* = *op-map-update k v* (*map-of* (*?list*))
from *list-map-autoref-update2*[*param-fo*, *OF K V L*]
  have *list-updated*: *map-of ?list-new* = *?list-new′*
    list-map-invar ?list-new **unfolding** *br-def* **by** *simp-all*

have *ahm-invar bhc* (*HashMap ?a′ ?n′*) **unfolding** *ahm-invar.simps*
proof(*rule ahm-invar-auxI*)
 fix *h*
 assume *h* < *array-length ?a′*
 hence *h-in-range*: *h* < *array-length a* **by** *simp*
 with *inv* **have** *bucket-ok*: *bucket-ok bhc ?l h* (*array-get a h*)
  by(*auto elim*: *ahm-invar-auxD*)
 thus *bucket-ok bhc* (*array-length ?a′*) *h* (*array-get ?a′ h*)
  proof (*cases h* = *bhc* (*array-length a*) *k*)
   case *False*
    with *bucket-ok* **show** *?thesis*
     by (*intro bucket-okI*, *force simp add*:
      array-get-array-set-other dest: *bucket-okD*)
  next
   case *True*
    show *?thesis*
    proof (*insert True*, *simp*, *intro bucket-okI*, *goal-cases*)
     case *prems*: (*1 k′*)
      show *?case*
      proof (*cases k* = *k′*)
       case *False*
        from *prems* **have** *k′* ∈ *dom ?list-new′*
         by (*simp only*: *dom-map-of-conv-image-fst*
          list-updated(*1*)[*symmetric*])
        hence *k′* ∈ *fst‘set ?list* **using** *False*

```
            by (simp add: dom-map-of-conv-image-fst)
          from imageE[OF this] obtain x where
            fst x = k′ and x ∈ set ?list by force
          then obtain v′ where (k′,v′) ∈ set ?list
             by (cases x, simp)
          with bucket-okD[OF bucket-ok] and
            ‹h = bhc (array-length a) k›
            show ?thesis by simp
        qed simp
      qed
    qed
    from ‹h < array-length a› inv have list-map-invar (array-get a h)
      by(auto dest: ahm-invar-auxD)
    with ‹h < array-length a›
    show list-map-invar (array-get ?a′ h)
      by (cases h = ?h, simp-all add:
         list-updated array-get-array-set-other)
  next

    obtain xs where a [simp]: a = Array xs by(cases a)

    let ?f = λn kvs. n + length kvs
    { fix n :: nat and xs :: ('k × 'v) list list
      have foldl ?f n xs = n + foldl ?f 0 xs
        by(induct xs arbitrary: rule: rev-induct) simp-all }
    note fold = this

    from inv have [simp]: bhc (length xs) k < length xs
        using bhc-range by simp
    have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs
        by(simp add: Cons-nth-drop-Suc)
    from inv have n = array-foldl (λ- n kvs. n + length kvs) 0 a
        by (force dest: ahm-invar-auxD)
    hence n = foldl ?f 0 (take ?h xs) + length (xs ! ?h) + foldl ?f 0 (drop (Suc
?h) xs)
      apply (simp add: array-foldl-foldl)
      apply (subst xs)
      apply simp
      apply (metis fold)
      done
    thus ?n′ = array-foldl (λ- n kvs. n + length kvs) 0 ?a′
     apply(simp add: ahm-rehash-aux′-def Let-def array-foldl-foldl foldl-list-update
map-of-eq-None-iff)
      apply(subst (1 2 3 4 5 6 7 8) fold)
      apply(simp add: length-list-map-update)
      done
  next
    from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
    thus 1 < array-length ?a′ by simp
```

next
qed

moreover have *ahm-*α *bhc* (*ahm-update-aux* (=) *bhc hm k v*) =
(*ahm-*α *bhc hm*)($k \mapsto v$)
proof
fix k'
show *ahm-*α *bhc* (*ahm-update-aux* (=) *bhc hm k v*) k' = ((*ahm-*α *bhc hm*)($k \mapsto$ v)) k'
proof (*cases bhc ?l k* = *bhc ?l* k')
case *False*
thus *?thesis* **by** (*force simp add*: *Let-def*
ahm-α*-def array-get-array-set-other*)
next
case *True*
hence *bhc ?l* k' = *bhc ?l k* **by** *simp*
thus *?thesis* **by** (*auto simp add*: *Let-def ahm-*α*-def*
list-map-lookup-is-map-of list-updated)
qed
qed

ultimately have *ref*: (*ahm-update-aux* (=) *bhc hm k v*,
$m(k \mapsto v)$) $\in$ *ahm-map-rel*$'$ *bhc* (**is** (*?hm*$'$,-)$\in$-)
unfolding *ahm-map-rel*$'$*-def br-def* **using** α **by** (*auto simp*: *Let-def*)

show (*ahm-update* (=) *bhc k v hm*, $m(k \mapsto v)$)
$\in$ *ahm-map-rel*$'$ *bhc*
proof (*cases ahm-filled ?hm*$'$)
case *False*
with *ref* **show** *?thesis* **unfolding** *ahm-update-def*
by (*simp del*: *ahm-update-aux.simps*)
next
case *True*
from *ref* **have** (*ahm-rehash bhc ?hm*$'$ (*hm-grow ?hm*$'$), $m(k \mapsto v)$) $\in$
ahm-map-rel$'$ *bhc* **unfolding** *ahm-map-rel*$'$*-def br-def*
by (*simp del*: *ahm-update-aux.simps*
add: *ahm-rehash-correct*[*OF bhc*])
thus *?thesis* **unfolding** *ahm-update-def* **using** *True*
by (*simp del*: *ahm-update-aux.simps add*: *Let-def*)
qed
qed

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*) $\in$
$Rk \to Rv \to \langle Rk,Rv \rangle$*ahm-rel bhc* $\to \langle Rk,Rv \rangle$*ahm-rel bhc*
proof (*intro fun-relI*)
let *?bhc*$'$ = *abstract-bounded-hashcode Rk bhc*

fix $k\ k'\ v\ v'\ a\ m'$
assume K: $(k,k') \in Rk$ **and** V: $(v,v') \in Rv$
assume M: $(a,m') \in \langle Rk,Rv\rangle ahm\text{-}rel\ bhc$

with *bhc* **have** *bhc'*: *is-bounded-hashcode Id* $(=)$ *?bhc'* **by** *blast*
from *abstract-bhc-correct*[*OF bhc*]
  have *bhc-rel*: $(bhc,?bhc') \in nat\text{-}rel \to Rk \to nat\text{-}rel$ **.**

from *M* **obtain** a' **where** *M1*: $(a,a') \in \langle Rk,Rv\rangle ahm\text{-}map\text{-}rel$ **and**
  M2: $(a',m') \in ahm\text{-}map\text{-}rel'\ ?bhc'$ **unfolding** *ahm-rel-def* **by** *blast*
hence *inv*: *ahm-invar ?bhc' a'*
  unfolding *ahm-map-rel'-def br-def* **by** *simp*

from *relcompI*[*OF param-ahm-update*[*OF bhc bhc-rel inv K V M1*]
    ahm-update-impl[*param-fo*, *OF bhc' - - M2*]]
show (*ahm-update eq bhc k v a*, *op-map-update k' v' m'*) $\in$
    $\langle Rk,Rv\rangle ahm\text{-}rel\ bhc$ **unfolding** *ahm-rel-def* **by** *simp*
qed

ahm-delete

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

assumes *isbhc*: *is-bounded-hashcode Rk eq bhc*
assumes *bhc*: $(bhc,bhc') \in nat\text{-}rel \to Rk \to nat\text{-}rel$
assumes *inv*: *ahm-invar bhc' m'*
assumes K: $(k,k') \in Rk$
assumes M: $(m,m') \in \langle Rk,Rv\rangle ahm\text{-}map\text{-}rel$
shows
(*ahm-delete eq bhc k m*, *ahm-delete* $(=)$ *bhc' k' m'*) $\in$
  $\langle Rk,Rv\rangle ahm\text{-}map\text{-}rel$
proof−
from *isbhc* **have** *eq*: $(eq,(=)) \in Rk \to Rk \to bool\text{-}rel$ **by** (*simp add*: *is-bounded-hashcodeD*)

obtain $a\ a'\ n\ n'$ **where**
  [*simp*]: $m = HashMap\ a\ n$ **and** [*simp*]: $m' = HashMap\ a'\ n'$
  by (*cases m*, *cases m'*)
from *M* **have** A: $(a,a') \in \langle\langle\langle Rk,Rv\rangle prod\text{-}rel\rangle list\text{-}rel\rangle array\text{-}rel$ **and**
    N: $(n,n') \in nat\text{-}rel$
  unfolding *ahm-map-rel-def* **by** *simp-all*

from *inv* **have** $1 < array\text{-}length\ a'$
  unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*
hence $1 < array\text{-}length\ a$
  by (*simp add*: *array-rel-imp-same-length*[*OF A*])
with *isbhc* **have** *bhc-range*: $bhc\ (array\text{-}length\ a)\ k < array\text{-}length\ a$ **by** *blast*

have *option-compare*: $\bigwedge a\ a'.\ (a,a') \in \langle Rv\rangle option\text{-}rel \Longrightarrow$

```
                          (a = None,a' = None) ∈ bool-rel by force
  have (array-get a (bhc (array-length a) k),
        array-get a' (bhc' (array-length a') k')) ∈
        ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
    using A K bhc bhc-range by parametricity
  note cmp = option-compare[OF param-list-map-lookup[param-fo, OF eq K this]]

  show ?thesis unfolding ‹m = HashMap a n› ‹m' = HashMap a' n'›
    by (simp only: ahm-delete.simps Let-def,
        insert eq isbhc bhc K A N bhc-range cmp, parametricity)
qed


lemma ahm-delete-impl:
  assumes bhc: is-bounded-hashcode Id (=) bhc
  shows (ahm-delete (=) bhc, op-map-delete) ∈ (Id::('k×'k) set) →
          ahm-map-rel' bhc → ahm-map-rel' bhc
proof (intro fun-relI, clarsimp)
  fix k::'k and hm::('k,'v) hashmap and  m::'k⇀'v
  assume (hm,m) ∈ ahm-map-rel' bhc
  hence α: m = ahm-α bhc hm and inv: ahm-invar bhc hm
    unfolding ahm-map-rel'-def br-def by simp-all
  obtain a n where [simp]: hm = HashMap a n by (cases hm)

  have K: (k,k) ∈ Id by simp

  from inv have [simp]: 1 < array-length a
    unfolding ahm-invar-def ahm-invar-aux-def by simp
  hence bhc-range[simp]: ⋀k. bhc (array-length a) k < array-length a
    using bhc by blast

  let ?l = array-length a
  let ?h = bhc ?l k
  let ?a' = array-set a ?h (list-map-delete (=) k (array-get a ?h))
  let ?n' = if list-map-lookup (=) k (array-get a ?h) = None then n else n − 1
  let ?list = array-get a ?h let ?list' = map-of ?list
  let ?list-new = list-map-delete (=) k ?list
  let ?list-new' = ?list' |` (−{k})
  from inv have (?list, ?list') ∈ br map-of list-map-invar
    unfolding br-def ahm-invar-def ahm-invar-aux-def by simp
  from list-map-autoref-delete2[param-fo, OF K this]
    have list-updated: map-of ?list-new = ?list-new'
      list-map-invar ?list-new by (simp-all add: br-def)

  have [simp]: array-length ?a' = ?l by simp

  have ahm-invar-aux bhc ?n' ?a'
  proof(rule ahm-invar-auxI)
   fix h
```

```
  assume h < array-length ?a'
  hence h-in-range[simp]: h < array-length a by simp
  with inv have inv': bucket-ok bhc ?l h (array-get a h)   1 < ?l
    list-map-invar (array-get a h) by (auto elim: ahm-invar-auxE)

  show bucket-ok bhc (array-length ?a') h (array-get ?a' h)
   proof (cases h = bhc ?l k)
    case False thus ?thesis using inv'
      by (simp add: array-get-array-set-other)
   next
    case True thus ?thesis
    proof (simp, intro bucket-okI, goal-cases)
     case prems: (1 k')
       show ?case
       proof (cases k = k')
        case False
         from prems have k' ∈ dom ?list-new'
           by (simp only: dom-map-of-conv-image-fst
             list-updated(1)[symmetric])
         hence k' ∈ fst`set ?list using False
           by (simp add: dom-map-of-conv-image-fst)
         from imageE[OF this] obtain x where
           fst x = k' and x ∈ set ?list by force
         then obtain v' where (k',v') ∈ set ?list
            by (cases x, simp)
         with bucket-okD[OF inv'(1)] and
           ‹h = bhc (array-length a) k›
           show ?thesis by blast
      qed simp
    qed
   qed

  from inv'(3) ‹h < array-length a›
  show list-map-invar (array-get ?a' h)
    by (cases h = ?h, simp-all add:
      list-updated array-get-array-set-other)
 next
  obtain xs where a [simp]: a = Array xs by(cases a)

  let ?f = λn kvs. n + length (kvs::('k×'v) list)
  { fix n :: nat and xs :: ('k×'v) list list
   have foldl ?f n xs = n + foldl ?f 0 xs
    by(induct xs arbitrary:  rule: rev-induct) simp-all }
  note fold = this

  from bhc-range have [simp]: bhc (length xs) k < length xs by simp
  moreover
   have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs by(simp add:
Cons-nth-drop-Suc)
```

from *inv* **have** $n = array\text{-}foldl\ (\lambda\text{-}\ n\ kvs.\ n + length\ kvs)\ 0\ a$
by(*auto elim*: *ahm-invar-auxE*)
hence $n = foldl\ ?f\ 0\ (take\ ?h\ xs) + length\ (xs\ !\ ?h) + foldl\ ?f\ 0\ (drop\ (Suc\ ?h)\ xs)$
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
moreover have $\bigwedge v\ a\ b.\ list\text{-}map\text{-}lookup\ (=)\ k\ (xs\ !\ ?h) = Some\ v$
$\Longrightarrow a + (length\ (xs\ !\ ?h) - 1) + b = a + length\ (xs\ !\ ?h) + b - 1$
by (*cases xs* ! *?h*, *simp-all*)
ultimately show $?n' = array\text{-}foldl\ (\lambda\text{-}\ n\ kvs.\ n + length\ kvs)\ 0\ ?a'$
apply(*simp add*: *array-foldl-foldl foldl-list-update map-of-eq-None-iff*)
apply(*subst* (*1 2 3 4 5 6 7 8*) *fold*)
apply (*intro conjI impI*)
apply(*auto simp add*: *length-list-map-delete*)
done
next

from *inv* **show** $1 < array\text{-}length\ ?a'$ **by**(*auto elim*: *ahm-invar-auxE*)
qed
hence *ahm-invar bhc* (*HashMap ?a′ ?n′*) **by** *simp*

moreover have $ahm\text{-}\alpha\text{-}aux\ bhc\ ?a' = ahm\text{-}\alpha\text{-}aux\ bhc\ a\ |`\ (-\ \{k\})$
proof
fix $k' :: 'k$
show $ahm\text{-}\alpha\text{-}aux\ bhc\ ?a'\ k' = (ahm\text{-}\alpha\text{-}aux\ bhc\ a\ |`\ (-\ \{k\}))\ k'$
proof (*cases bhc* $?l\ k' = ?h$)
case *False*
hence $k \neq k'$ **by** *force*
thus *?thesis* **using** *False* **unfolding** *ahm-α-aux-def*
by (*simp add*: *array-get-array-set-other*
list-map-lookup-is-map-of)
next
case *True*
thus *?thesis* **unfolding** *ahm-α-aux-def*
by (*simp add*: *list-map-lookup-is-map-of*
list-updated(*1*) *restrict-map-def*)
qed
qed
hence $ahm\text{-}\alpha\ bhc\ (HashMap\ ?a'\ ?n') = op\text{-}map\text{-}delete\ k\ m$
unfolding *op-map-delete-def* **by** (*simp add*: *ahm-α-def2 α*)

ultimately have $(HashMap\ ?a'\ ?n',\ op\text{-}map\text{-}delete\ k\ m) \in ahm\text{-}map\text{-}rel'\ bhc$
unfolding *ahm-map-rel′-def br-def* **by** *simp*

thus $(ahm\text{-}delete\ (=)\ bhc\ k\ hm,\ m\ |`\ (-\{k\})) \in ahm\text{-}map\text{-}rel'\ bhc$
by (*simp only*: ‹*hm* = *HashMap a n*› *ahm-delete.simps Let-def*
op-map-delete-def, *force*)
qed

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

assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode Rk eq bhc*)
shows (*ahm-delete eq bhc*, *op-map-delete*) ∈
$Rk \to \langle Rk,Rv\rangle$*ahm-rel bhc* $\to \langle Rk,Rv\rangle$*ahm-rel bhc*
proof (*intro fun-relI*)
let *?bhc′* = *abstract-bounded-hashcode Rk bhc*
fix *k k′ a m′*
assume *K*: (*k*,*k′*) ∈ *Rk*
assume *M*: (*a*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc*

with *bhc* **have** *bhc′*: *is-bounded-hashcode Id* (=) *?bhc′* **by** *blast*
from *abstract-bhc-correct*[*OF bhc*]
have *bhc-rel*: (*bhc*,*?bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel* **.**

from *M* **obtain** *a′* **where** *M1*: (*a*,*a′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* **and**
M2: (*a′*,*m′*) ∈ *ahm-map-rel′ ?bhc′* **unfolding** *ahm-rel-def* **by** *blast*
hence *inv*: *ahm-invar ?bhc′ a′*
unfolding *ahm-map-rel′-def br-def* **by** *simp*

from *relcompI*[*OF param-ahm-delete*[*OF bhc bhc-rel inv K M1*]
ahm-delete-impl[*param-fo*, *OF bhc′ - M2*]]
show (*ahm-delete eq bhc k a*, *op-map-delete k′ m′*) ∈
⟨*Rk*,*Rv*⟩*ahm-rel bhc* **unfolding** *ahm-rel-def* **by** *simp*
qed

Various simple operations

lemma *param-ahm-isEmpty*[*param*]:
(*ahm-isEmpty*, *ahm-isEmpty*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
unfolding *ahm-isEmpty-def*[*abs-def*] *rec-hashmap-is-case*
by *parametricity*

lemma *param-ahm-isSng*[*param*]:
(*ahm-isSng*, *ahm-isSng*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
unfolding *ahm-isSng-def*[*abs-def*] *rec-hashmap-is-case*
by *parametricity*

lemma *param-ahm-size*[*param*]:
(*ahm-size*, *ahm-size*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *nat-rel*
unfolding *ahm-size-def*[*abs-def*] *rec-hashmap-is-case*
by *parametricity*

lemma *ahm-isEmpty-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-isEmpty*, *op-map-isEmpty*) ∈ *ahm-map-rel′ bhc* → *bool-rel*
proof (*intro fun-relI*)
fix *hm m* **assume** *rel*: (*hm*,*m*) ∈ *ahm-map-rel′ bhc*
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)

from *rel* **have** α: m = *ahm-α-aux bhc a* **and** *inv*: *ahm-invar-aux bhc n a*
unfolding *ahm-map-rel′-def br-def* **by** (*simp-all add*: *ahm-α-def2*)
from *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF assms inv*,*symmetric*] **and**
finite-dom-ahm-α-aux[*OF assms inv*]
show (*ahm-isEmpty hm*, *op-map-isEmpty m*) $\in$ *bool-rel*
unfolding *ahm-isEmpty-def op-map-isEmpty-def*
by (*simp add*: α *card-eq-0-iff*)
qed

lemma *ahm-isSng-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-isSng*, *op-map-isSng*) $\in$ *ahm-map-rel′ bhc* $\rightarrow$ *bool-rel*
proof (*intro fun-relI*)
fix *hm m* **assume** *rel*: (*hm*,*m*) $\in$ *ahm-map-rel′ bhc*
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)
from *rel* **have** α: m = *ahm-α-aux bhc a* **and** *inv*: *ahm-invar-aux bhc n a*
unfolding *ahm-map-rel′-def br-def* **by** (*simp-all add*: *ahm-α-def2*)
note *n-props*[*simp*] = *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF assms inv*,*symmetric*]
note *finite-dom*[*simp*] = *finite-dom-ahm-α-aux*[*OF assms inv*]
show (*ahm-isSng hm*, *op-map-isSng m*) $\in$ *bool-rel*
by (*force simp add*: α[*symmetric*] *dom-eq-singleton-conv*
dest!: *card-eq-SucD*)
qed

lemma *ahm-size-impl*:
assumes *is-bounded-hashcode Id* (=) *bhc*
shows (*ahm-size*, *op-map-size*) $\in$ *ahm-map-rel′ bhc* $\rightarrow$ *nat-rel*
proof (*intro fun-relI*)
fix *hm m* **assume** *rel*: (*hm*,*m*) $\in$ *ahm-map-rel′ bhc*
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)
from *rel* **have** α: m = *ahm-α-aux bhc a* **and** *inv*: *ahm-invar-aux bhc n a*
unfolding *ahm-map-rel′-def br-def* **by** (*simp-all add*: *ahm-α-def2*)
from *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF assms inv*,*symmetric*]
show (*ahm-size hm*, *op-map-size m*) $\in$ *nat-rel*
unfolding *ahm-isEmpty-def op-map-isEmpty-def*
by (*simp add*: α *card-eq-0-iff*)
qed

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*) $\in$ $\langle Rk,Rv\rangle$*ahm-rel bhc* $\rightarrow$ *bool-rel*
proof (*intro fun-relI*)
let *?bhc′* = *abstract-bounded-hashcode Rk bhc*
fix *k k′ a m′*
assume *M*: (*a*,*m′*) $\in$ $\langle Rk,Rv\rangle$*ahm-rel bhc*

```
  from bhc have bhc': is-bounded-hashcode Id (=) ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-isEmpty[param-fo, OF M1]
                ahm-isEmpty-impl[param-fo, OF bhc' M2]]
  show (ahm-isEmpty a, op-map-isEmpty m') ∈ bool-rel by simp
qed

lemma autoref-ahm-isSng[autoref-rules]:
  assumes bhc[unfolded autoref-tag-defs]:
      SIDE-GEN-ALGO (is-bounded-hashcode Rk eq bhc)
  shows (ahm-isSng, op-map-isSng) ∈ ⟨Rk,Rv⟩ahm-rel bhc → bool-rel
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from bhc have bhc': is-bounded-hashcode Id (=) ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-isSng[param-fo, OF M1]
                ahm-isSng-impl[param-fo, OF bhc' M2]]
  show (ahm-isSng a, op-map-isSng m') ∈ bool-rel by simp
qed

lemma autoref-ahm-size[autoref-rules]:
  assumes bhc[unfolded autoref-tag-defs]:
      SIDE-GEN-ALGO (is-bounded-hashcode Rk eq bhc)
  shows (ahm-size, op-map-size) ∈ ⟨Rk,Rv⟩ahm-rel bhc → nat-rel
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from bhc have bhc': is-bounded-hashcode Id (=) ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-size[param-fo, OF M1]
                ahm-size-impl[param-fo, OF bhc' M2]]
  show (ahm-size a, op-map-size m') ∈ nat-rel by simp
qed
```

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*)
proof −
  from *SK SV* **have** *1*: *single-valued* ($\langle\langle\langle$*Rk*, *Rv*$\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel*)
    by (*tagged-solver*)

  thus *?thesis*
    unfolding *ahm-map-rel-def*
    by (*auto intro*: *single-valuedI dest*: *single-valuedD*[*OF 1*])
qed

lemma *ahm-rel-sv*[*relator-props*]:
  $[\![$*single-valued Rk*; *single-valued Rv*$]\!]$
  $\Longrightarrow$ *single-valued* ($\langle$*Rk*,*Rv*$\rangle$*ahm-rel bhc*)
  unfolding *ahm-rel-def ahm-map-rel′-def*
  by (*tagged-solver* (*keep*))

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*)
  unfolding *ahm-rel-def finite-map-rel-def ahm-map-rel′-def br-def*
  apply *auto*
  apply (*case-tac y*)
  apply (*auto simp*: *ahm-*α*-def2*)
  thm *finite-dom-ahm-*α*-aux*
  apply (*rule finite-dom-ahm-*α*-aux*)
  apply (*rule abstract-bhc-is-bhc*)
  apply (*rule A*)
  apply *assumption*
  done

Proper iterator proofs

lemma *pi-ahm*[*icf-proper-iteratorI*]:
  proper-it (*ahm-iteratei m*) (*ahm-iteratei m*)
proof−
  obtain *a n* **where** [*simp*]: *m* = *HashMap a n* **by** (*cases m*)
  then obtain *l* **where** [*simp*]: *a* = *Array l* **by** (*cases a*)
  thus *?thesis*
    unfolding *proper-it-def list-map-iteratei-def*
    by (*simp add*: *ahm-iteratei-aux-def*, *blast*)
qed

lemma *pi′-ahm*[*icf-proper-iteratorI*]:
  proper-it′ ahm-iteratei ahm-iteratei
  by (*rule proper-it′I*, *rule pi-ahm*)

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

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

end

2.3.5 Red-Black Tree based Maps

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

Standard Setup

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

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

thm *color-rel-elims*

lemma *param-color*[*param*]:
(*color.R*,*color.R*)∈*color-rel*
(*color.B*,*color.B*)∈*color-rel*
(*case-color*,*case-color*)∈*R* → *R* → *color-rel* → *R*
by (*auto*

```
    intro: color-rel.intros
    elim: color-rel.cases
    split: color.split)

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

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

definition rbt-rel ≡ rbt-rel-aux
lemma rbt-rel-aux-fold: rbt-rel-aux Ra Rb ≡ ⟨Ra,Rb⟩rbt-rel
  by (simp add: rbt-rel-def relAPP-def)

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

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

lemma param-case-rbt[param]:
  (case-rbt,case-rbt) ∈
    Ra → (color-rel → ⟨Rb,Rc⟩rbt-rel → Rb → Rc → ⟨Rb,Rc⟩rbt-rel → Ra)
      → ⟨Rb,Rc⟩rbt-rel → Ra
  apply clarsimp
  apply (erule rbt-rel-cases)
  apply simp
  apply simp
  apply parametricity
  done

lemma param-rec-rbt[param]: (rec-rbt, rec-rbt) ∈
  Ra → (color-rel → ⟨Rb,Rc⟩rbt-rel → Rb → Rc → ⟨Rb,Rc⟩rbt-rel
   → Ra → Ra → Ra) → ⟨Rb,Rc⟩rbt-rel → Ra
proof (intro fun-relI, goal-cases)
  case (1 s s' f f' t t') from this(3,1,2) show ?case
```

```
    apply (induct arbitrary: s s')
    apply simp
    apply simp
    apply parametricity
    done
qed

lemma param-paint[param]:
  (paint,paint)∈color-rel → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding paint-def
  by parametricity

lemma param-balance[param]:
  shows (balance,balance) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI, goal-cases)
  case (1 t1 t1' a a' b b' t2 t2')
  thus ?case
    apply (induct t1' a' b' t2' arbitrary: t1 a b t2 rule: balance.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance.simps)
    apply (parametricity)+
    done
qed


lemma param-rbt-ins[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows (ord.rbt-ins less,ord.rbt-ins less') ∈
          (Ra→Rb→Rb→Rb) → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI, goal-cases)
  case (1 f f' a a' b b' t t')
  thus ?case
    apply (induct f' a' b' t' arbitrary: f a b t rule: ord.rbt-ins.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: ord.rbt-ins.simps rbt-ins.simps)
    apply parametricity+
    done
qed

term rbt-insert
lemma param-rbt-insert[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows (ord.rbt-insert less,ord.rbt-insert less') ∈
    Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding rbt-insert-def ord.rbt-insert-def
  unfolding rbt-insert-with-key-def[abs-def]
```

```
    ord.rbt-insert-with-key-def[abs-def]
  by parametricity

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

term balance-left
lemma param-balance-left[param]:
  (balance-left, balance-left) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI, goal-cases)
  case (1 l l' a a' b b' r r')
  thus ?case
    apply (induct l a b r arbitrary: l' a' b' r' rule: balance-left.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance-left.simps)
    apply parametricity+
    done
qed

term balance-right
lemma param-balance-right[param]:
  (balance-right, balance-right) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI, goal-cases)
  case (1 l l' a a' b b' r r')
  thus ?case
    apply (induct l a b r arbitrary: l' a' b' r' rule: balance-right.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance-right.simps)
    apply parametricity+
    done
qed

lemma param-combine[param]:
  (combine,combine)∈⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI, goal-cases)
  case (1 t1 t1' t2 t2')
  thus ?case
    apply (induct t1 t2 arbitrary: t1' t2' rule: combine.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: combine.simps)
    apply parametricity+
    done
```

```
qed

lemma ih-aux1: ⟦ (a',b)∈R; a'=a ⟧ ⟹ (a,b)∈R by auto
lemma is-eq: a=b ⟹ a=b .

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

  apply (assumption
    | elim rbt-rel-elims color-rel-elims
    | simp (no-asm-use) only: rbt-del.simps ord.rbt-del.simps
        rbt-del-from-left.simps ord.rbt-del-from-left.simps
        rbt-del-from-right.simps ord.rbt-del-from-right.simps
    | parametricity
    | rule rbt-rel-intros
    | hypsubst
    | (simp, rule ih-aux1, rprems)
    | (rule is-eq, simp)
  ) +
  done

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

    Ra → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
  by (*intro fun-relI*, *blast intro*: *param-rbt-del-aux*[*OF param-less*])+

lemma *param-rbt-delete*[*param*]:
  fixes *less*
  assumes *param-less*[*param*]: (*less*,*less′*) ∈ *Ra* → *Ra* → *Id*
  shows (*ord.rbt-delete less*, *ord.rbt-delete less′*)
    ∈ *Ra* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
  unfolding *rbt-delete-def*[*abs-def*] *ord.rbt-delete-def*[*abs-def*]
  by *parametricity*

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

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

lemma *param-compare*[*param*]:
  (*RBT-Impl.LT*,*RBT-Impl.LT*)∈*compare-rel*
  (*RBT-Impl.GT*,*RBT-Impl.GT*)∈*compare-rel*
  (*RBT-Impl.EQ*,*RBT-Impl.EQ*)∈*compare-rel*
 (*RBT-Impl.case-compare*,*RBT-Impl.case-compare*)∈*R*→*R*→*R*→*compare-rel*→*R*
  by (*auto split*: *RBT-Impl.compare.split*)

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

  apply (*simp only*: *pair-in-Id-conv*)
  apply (*simp* (*no-asm-use*) *only*: *rbtreeify-f-simps rbtreeify-g-simps*)
  apply *parametricity*

  apply (*elim list-relE prod-relE*)
  apply (*simp only*: *pair-in-Id-conv*)
  apply *hypsubst*
  apply (*simp* (*no-asm-use*) *only*: *rbtreeify-f-simps rbtreeify-g-simps*)
  apply *parametricity*

  apply *clarsimp*
  apply (*subgoal-tac* (*rbtreeify-f n kvs*, *rbtreeify-f n kvs′a*)
    ∈ ⟨⟨*Ra*, *Rb*⟩*rbt-rel*, ⟨⟨*Ra*, *Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*)
  apply (*clarsimp elim*!: *list-relE prod-relE*)

apply *parametricity*
apply (*rule refl*)
apply *rprems*
apply (*rule refl*)
apply *assumption*

apply *clarsimp*
apply (*subgoal-tac* (*rbtreeify-f n kvs*, *rbtreeify-f n kvs′a*)
∈ ⟨⟨*Ra*, *Rb*⟩*rbt-rel*, ⟨⟨*Ra*, *Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*)
apply (*clarsimp elim*!: *list-relE prod-relE*)
apply *parametricity*
apply (*rule refl*)
apply *rprems*
apply (*rule refl*)
apply *assumption*

apply *simp*
apply *parametricity*

apply *clarsimp*
apply *parametricity*

apply *clarsimp*
apply (*subgoal-tac* (*rbtreeify-g n kvs*, *rbtreeify-g n kvs′a*)
∈ ⟨⟨*Ra*, *Rb*⟩*rbt-rel*, ⟨⟨*Ra*, *Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*)
apply (*clarsimp elim*!: *list-relE prod-relE*)
apply *parametricity*
apply (*rule refl*)
apply *parametricity*
apply (*rule refl*)

apply *clarsimp*
apply (*subgoal-tac* (*rbtreeify-f n kvs*, *rbtreeify-f n kvs′a*)
∈ ⟨⟨*Ra*, *Rb*⟩*rbt-rel*, ⟨⟨*Ra*, *Rb*⟩*prod-rel*⟩*list-rel*⟩*prod-rel*)
apply (*clarsimp elim*!: *list-relE prod-relE*)
apply *parametricity*
apply (*rule refl*)
apply *parametricity*
apply (*rule refl*)
done

lemma *param-rbtreeify*[*param*]:
(*rbtreeify*, *rbtreeify*) ∈ ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
unfolding *rbtreeify-def*[*abs-def*]
apply *parametricity*
by *simp*

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

```
  shows ⟦ (less,less') ∈ Ra → Ra → Id;
    (f,f')∈(Ra→Rb→Rb→Rb); (a,a')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel;
    (b,b')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel ⟧
  ⟹ (ord.sunion-with less f a b, ord.sunion-with less' f' a' b') ∈
    ⟨⟨Ra,Rb⟩prod-rel⟩list-rel
  apply (induct f' a' b' arbitrary: f a b
    rule: ord.sunion-with.induct[of less'])
  apply (elim-all list-relE prod-relE)
  apply (simp-all only: ord.sunion-with.simps)
  apply parametricity
  apply simp-all
  done

lemma skip-red-alt:
  RBT-Impl.skip-red t = (case t of
    (Branch color.R l k v r) ⇒ l
  | - ⇒ t)
  by (auto split: rbt.split color.split)

function compare-height ::
  ('a, 'b) RBT-Impl.rbt ⇒ ('a, 'b) RBT-Impl.rbt ⇒ ('a, 'b) RBT-Impl.rbt ⇒ ('a,
'b) RBT-Impl.rbt ⇒ RBT-Impl.compare
  where
  compare-height sx s t tx =
 (case (RBT-Impl.skip-red sx, RBT-Impl.skip-red s, RBT-Impl.skip-red t, RBT-Impl.skip-red
tx) of
    (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
      compare-height (RBT-Impl.skip-black sx') s' t' (RBT-Impl.skip-black tx')
  | (-, rbt.Empty, -, Branch - - - - -) ⇒ RBT-Impl.LT
  | (Branch - - - - -, -, rbt.Empty, -) ⇒ RBT-Impl.GT
  | (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, rbt.Empty) ⇒
      compare-height (RBT-Impl.skip-black sx') s' t' rbt.Empty
  | (rbt.Empty, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
      compare-height rbt.Empty s' t' (RBT-Impl.skip-black tx')
  | - ⇒ RBT-Impl.EQ)
  by pat-completeness auto

lemma skip-red-size: size (RBT-Impl.skip-red b) ≤ size b
  by (auto simp add: skip-red-alt split: rbt.split color.split)

lemma skip-black-size: size (RBT-Impl.skip-black b) ≤ size b
  unfolding RBT-Impl.skip-black-def
  apply (auto
    simp add: Let-def
    split: rbt.split color.split
  )
  using skip-red-size[of b]
  apply auto
  done
```

```
termination
proof -
  {
    fix s t x
    assume A: s = RBT-Impl.skip-red t
      and B: x < size s
    note B
    also note A
    also note skip-red-size
    finally have x < size t .
  } note AUX=this

  show All compare-height-dom
    apply (relation
      measure (λ(a, b, c, d). size a + size b + size c + size d))
    apply rule



    apply (clarsimp simp: Let-def split: rbt.splits color.splits)
    apply (intro add-less-mono trans-less-add2
      order-le-less-trans[OF skip-black-size], (erule AUX, simp)+) []

    apply (clarsimp simp: Let-def split: rbt.splits color.splits)
    apply (rule trans-less-add1)
    apply (intro add-less-mono trans-less-add2
      order-le-less-trans[OF skip-black-size], (erule AUX, simp)+) []

    apply (clarsimp simp: Let-def split: rbt.splits color.splits)
    apply (intro add-less-mono trans-less-add2
      order-le-less-trans[OF skip-black-size], (erule AUX, simp)+) []
    done
qed

lemmas [simp del] = compare-height.simps

lemma compare-height-alt:
  RBT-Impl.compare-height sx s t tx = compare-height sx s t tx
  apply (induct sx s t tx rule: compare-height.induct)
  apply (subst RBT-Impl.compare-height.simps)
  apply (subst compare-height.simps)
  apply (auto split: rbt.split)
  done

term RBT-Impl.skip-red
lemma param-skip-red[param]: (RBT-Impl.skip-red,RBT-Impl.skip-red)
  ∈ ⟨Rk,Rv⟩rbt-rel → ⟨Rk,Rv⟩rbt-rel
  unfolding skip-red-alt[abs-def] by parametricity
```

lemma *param-skip-black*[*param*]: (*RBT-Impl.skip-black*,*RBT-Impl.skip-black*)
$\in$ ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
unfolding *RBT-Impl.skip-black-def*[*abs-def*] **by** *parametricity*

term *case-rbt*
lemma *param-case-rbt′*:
assumes (*t*,*t′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*
assumes ⟦*t*=*rbt.Empty*; *t′*=*rbt.Empty*⟧ ⟹ (*fl*,*fl′*)∈*R*
assumes ⋀*c l k v r c′ l′ k′ v′ r′*. ⟦
t = *Branch c l k v r*; *t′* = *Branch c′ l′ k′ v′ r′*;
(*c*,*c′*)∈*color-rel*;
(*l*,*l′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*k*,*k′*)∈*Rk*; (*v*,*v′*)∈*Rv*; (*r*,*r′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*
⟧ ⟹ (*fb c l k v r*, *fb′ c′ l′ k′ v′ r′*) ∈ *R*
shows (*case-rbt fl fb t*, *case-rbt fl′ fb′ t′*) ∈ *R*
using *assms* **by** (*auto split*: *rbt.split elim*: *rbt-rel-elims*)

lemma *compare-height-param-aux*[*param*]:
⟦ (*sx*,*sx′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*s*,*s′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*;
(*t*,*t′*)∈⟨*Rk*,*Rv*⟩*rbt-rel*; (*tx*,*tx′*)∈⟨*Rk*,*Rv*⟩*rbt-rel* ⟧
⟹ (*compare-height sx s t tx*, *compare-height sx′ s′ t′ tx′*) ∈ *compare-rel*
apply (*induct sx′ s′ t′ tx′ arbitrary*: *sx s t tx*
rule: *compare-height.induct*)
apply (*subst* (*2*) *compare-height.simps*)
apply (*subst compare-height.simps*)
apply (*parametricity add*: *param-case-prod′ param-case-rbt′*, (*simp only*: *prod.inject*)+)
[]
done

lemma *compare-height-param*[*param*]:
(*RBT-Impl.compare-height*,*RBT-Impl.compare-height*) ∈
⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
→ *compare-rel*
unfolding *compare-height-alt*[*abs-def*]
by *parametricity*

lemma *rbt-rel-bheight*: (*t*, *t′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel* ⟹ *bheight t* = *bheight t′*
by (*induction t arbitrary*: *t′*) (*auto elim*!: *rbt-rel-elims color-rel.cases*)

lemma *param-rbt-baliL*[*param*]: (*rbt-baliL*,*rbt-baliL*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb*
→ ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
proof (*intro fun-relI*, *goal-cases*)
case (*1 l l′ a a′ b b′ r r′*)
thus *?case*
apply (*induct l a b r arbitrary*: *l′ a′ b′ r′ rule*: *rbt-baliL.induct*)
apply (*elim-all rbt-rel-elims color-rel-elims*)
apply (*simp-all only*: *rbt-baliL.simps*)
apply *parametricity*+
done

qed

lemma *param-rbt-baliR*[*param*]: (*rbt-baliR*,*rbt-baliR*) $\in$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ *Ra* $\rightarrow$ *Rb* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel*
proof (*intro fun-relI*, *goal-cases*)
  case (*1 l l′ a a′ b b′ r r′*)
  thus *?case*
    apply (*induction l a b r arbitrary*: *l′ a′ b′ r′ rule*: *rbt-baliR.induct*)
    apply (*elim-all rbt-rel-elims color-rel-elims*)
    apply (*simp-all only*: *rbt-baliR.simps*)
    apply *parametricity+*
    done
qed

lemma *param-rbt-joinL*[*param*]: (*rbt-joinL*,*rbt-joinL*) $\in$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ *Ra* $\rightarrow$ *Rb* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel*
proof (*intro fun-relI*, *goal-cases*)
  case (*1 l l′ a a′ b b′ r r′*)
  thus *?case*
  proof (*induction l a b r arbitrary*: *l′ a′ b′ r′ rule*: *rbt-joinL.induct*)
    case (*1 l a b r*)
    have *bheight l* $<$ *bheight r* $\Longrightarrow$ *r* = *RBT-Impl.MB ll k v rr* $\Longrightarrow$ (*ll*, *ll′*) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel* $\Longrightarrow$
      (*k*, *k′*) $\in$ *Ra* $\Longrightarrow$ (*v*, *v′*) $\in$ *Rb* $\Longrightarrow$ (*rr*, *rr′*) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel* $\Longrightarrow$
      (*rbt-baliL* (*rbt-joinL l a b ll*) *k v rr*, *rbt-baliL* (*rbt-joinL l′ a′ b′ ll′*) *k′ v′ rr′*) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel*
      for *ll ll′ k k′ v v′ rr rr′*
      by *parametricity* (*auto intro*: *1*)
    then show *?case*
      using *1*
        by (*auto simp*: *rbt-joinL.simps*[*of l a b r*] *rbt-joinL.simps*[*of l′ a′ b′ r′*] *rbt-rel-bheight*
          intro: *rbt-rel-intros color-rel.intros elim*!: *rbt-rel-elims color-rel-elims*
          split: *rbt.splits color.splits*)
  qed
qed

lemma *param-rbt-joinR*[*param*]:
 (*rbt-joinR*,*rbt-joinR*) $\in$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ *Ra* $\rightarrow$ *Rb* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel* $\rightarrow$ $\langle$*Ra*,*Rb*$\rangle$*rbt-rel*
proof (*intro fun-relI*, *goal-cases*)
  case (*1 l l′ a a′ b b′ r r′*)
  thus *?case*
  proof (*induction l a b r arbitrary*: *l′ a′ b′ r′ rule*: *rbt-joinR.induct*)
    case (*1 l a b r*)
    have *bheight r* $<$ *bheight l* $\Longrightarrow$ *l* = *RBT-Impl.MB ll k v rr* $\Longrightarrow$ (*ll*, *ll′*) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel* $\Longrightarrow$
      (*k*, *k′*) $\in$ *Ra* $\Longrightarrow$ (*v*, *v′*) $\in$ *Rb* $\Longrightarrow$ (*rr*, *rr′*) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel* $\Longrightarrow$
      (*rbt-baliR ll k v* (*rbt-joinR rr a b r*), *rbt-baliR ll′ k′ v′* (*rbt-joinR rr′ a′ b′ r′*)) $\in$ $\langle$*Ra*, *Rb*$\rangle$*rbt-rel*

for *ll ll′ k k′ v v′ rr rr′*
by *parametricity* (*auto intro*: *1*)
then show *?case*
using *1*
by (*auto simp*: *rbt-joinR.simps*[*of l*] *rbt-joinR.simps*[*of l′*] *rbt-rel-bheight*[*symmetric*]
intro: *rbt-rel-intros color-rel.intros elim*!: *rbt-rel-elims color-rel-elims*
split: *rbt.splits color.splits*)
qed
qed

lemma *param-rbt-join*[*param*]: (*rbt-join*,*rbt-join*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → *Rb* →
⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
by (*auto simp*: *rbt-join-def Let-def rbt-rel-bheight*) *parametricity*+

lemma *param-split*[*param*]:
fixes *less*
assumes [*param*]: (*less*,*less′*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-split less*, *ord.rbt-split less′*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → *Ra* → ⟨⟨*Ra*,*Rb*⟩*rbt-rel*,⟨⟨*Rb*⟩*option-rel*,⟨*R*
proof (*intro fun-relI*)
fix *t t′ a a′*
assume (*t*, *t′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*   (*a*, *a′*) ∈ *Ra*
then show (*ord.rbt-split less t a*, *ord.rbt-split less′ t′ a′*) ∈ ⟨⟨*Ra*,*Rb*⟩*rbt-rel*,⟨⟨*Rb*⟩*option-rel*,⟨*Ra*,*Rb*⟩*rbt-re*
proof (*induction t a arbitrary*: *t′ rule*: *ord.rbt-split.induct*[**where** *?less*=*less*])
case (*2 c l k b r a*)
obtain *c′ l′ k′ b′ r′* **where** *t′-def*: *t′* = *Branch c′ l′ k′ b′ r′*
using *2*(*3*)
by (*auto elim*: *rbt-rel-elims*)
have *sub-rel*: (*l*, *l′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*   (*k*, *k′*) ∈ *Ra*   (*b*, *b′*) ∈ *Rb*   (*r*, *r′*) ∈
⟨*Ra*, *Rb*⟩*rbt-rel*
using *2*(*3*)
by (*auto simp*: *t′-def elim*: *rbt-rel-elims*)
have *less-iff*: *less a k* ⟷ *less′ a′ k′*   *less k a* ⟷ *less′ k′ a′*
using *assms 2*(*4*) *sub-rel*(*2*)
by (*auto dest*: *fun-relD*)
show *?case*
using *2*(*1*)[*OF* - *sub-rel*(*1*) *2*(*4*)] *2*(*2*)[*OF* - - *sub-rel*(*4*) *2*(*4*)] *sub-rel*
unfolding *t′-def less-iff ord.rbt-split.simps*(*2*)[*of less c*] *ord.rbt-split.simps*(*2*)[*of*
less′ c′]
by (*auto split*: *prod.splits*) *parametricity*+
qed (*auto simp*: *ord.rbt-split.simps elim*!: *rbt-rel-elims intro*: *rbt-rel-intros*)
qed

lemma *param-rbt-union-swap-rec*[*param*]:
fixes *less*
assumes [*param*]: (*less*,*less′*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-union-swap-rec less*, *ord.rbt-union-swap-rec less′*) ∈
(*Ra* → *Rb* → *Rb* → *Rb*) → *Id* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
proof (*intro fun-relI*)
fix *f f′ b b′ t1 t1′ t2 t2′*

assume (*f*, *f′*) ∈ *Ra* → *Rb* → *Rb* → *Rb* (*b*, *b′*) ∈ *bool-rel* (*t1*, *t1′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel* (*t2*, *t2′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*
then show (*ord.rbt-union-swap-rec less f b t1 t2*, *ord.rbt-union-swap-rec less′ f′ b′ t1′ t2′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*
proof (*induction f b t1 t2 arbitrary*: *b′ t1′ t2′ rule*: *ord.rbt-union-swap-rec.induct*[**where** *?less=less*])
  case (*1 f b t1 t2*)
  obtain *γ s1 s2* **where** *flip1*: (*γ*, *s2*, *s1*) =
    (*if flip-rbt t2 t1 then* (¬*b*, *t1*, *t2*) *else* (*b*, *t2*, *t1*))
    by *fastforce*
  obtain *γ′ s1′ s2′* **where** *flip2*: (*γ′*, *s2′*, *s1′*) =
    (*if flip-rbt t2′ t1′ then* (¬*b′*, *t1′*, *t2′*) *else* (*b′*, *t2′*, *t1′*))
    by *fastforce*
  define *g* **where** *g* = (*if γ then λk v v′. f k v′ v else f*)
  define *g′* **where** *g′* = (*if γ′ then λk v v′. f′ k v′ v else f′*)
  note *bheight-cong* = *rbt-rel-bheight*[*OF 1*(*5*)] *rbt-rel-bheight*[*OF 1*(*6*)]
  have *flip-cong*: *flip-rbt t2 t1* ⟷ *flip-rbt t2′ t1′*
    by (*auto simp*: *flip-rbt-def bheight-cong*)
  have *gamma-cong*: *γ* = *γ′*
    using *flip1 flip2 1*(*4*)
    by (*auto simp*: *flip-cong split*: *if-splits*)
  have *small-rbt-cong*: *small-rbt s2* ⟷ *small-rbt s2′*
    using *flip1 flip2*
    by (*auto simp*: *small-rbt-def flip-cong bheight-cong split*: *if-splits*)
  have *rbt-rel-s*: (*s1*, *s1′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel* (*s2*, *s2′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*
    using *flip1 flip2 1*(*5*,*6*)
    by (*auto simp*: *flip-cong split*: *if-splits*)
  have *fun-rel-g*: (*g*, *g′*) ∈ *Ra* → *Rb* → *Rb* → *Rb*
    using *flip1 flip2 1*(*3*,*4*)
    by (*auto simp*: *flip-cong g-def g′-def intro*: *fun-relD*[*OF fun-relD*[*OF fun-relD*]] *split*: *if-splits*)
  have *rbt-rel-fold*: (*RBT-Impl.fold* (*ord.rbt-insert-with-key less g*) *s2 s1*, *RBT-Impl.fold* (*ord.rbt-insert-with-key less′ g′*) *s2′ s1′*) ∈ ⟨*Ra*, *Rb*⟩*rbt-rel*
    unfolding *RBT-Impl.fold-def RBT-Impl.entries-def ord.rbt-insert-with-key-def*
    using *rbt-rel-s fun-rel-g*
    by *parametricity*
  {
    fix *c l k v r c′ l′ k′ v′ r′ ll β rr ll′ β′ rr′*
    assume *defs*: *s1* = *Branch c l k v r* *s1′* = *Branch c′ l′ k′ v′ r′*
      ord.rbt-split less s2 k = (*ll*, *β*, *rr*) *ord.rbt-split less′ s2′ k′* = (*ll′*, *β′*, *rr′*)
      ¬*small-rbt s2*
    have *split-rel*: (*ord.rbt-split less s2 k*, *ord.rbt-split less′ s2′ k′*) ∈ ⟨⟨*Ra*,*Rb*⟩*rbt-rel*,⟨⟨*Rb*⟩*option-rel*,⟨*Ra*,*Rb*⟩*rbt-rel*⟩*pr*
      using *defs*(*1*,*2*) *param-split*[*OF assms*, *of Rb*] *rbt-rel-s*
      by (*auto elim*: *rbt-rel-elims dest*: *fun-relD*)
    have *IH1*: (*ord.rbt-union-swap-rec less f γ l ll*, *ord.rbt-union-swap-rec less′ f′ γ′ l′ ll′*) ∈ ⟨*Ra*,*Rb*⟩*rbt-rel*
      apply (*rule 1*(*1*)[*OF refl flip1 refl refl - - - refl*])
      using *1*(*3*) *split-rel rbt-rel-s*(*1*) *defs*
      by (*auto simp*: *gamma-cong elim*: *rbt-rel-elims*)

```
    have IH2: (ord.rbt-union-swap-rec less f γ r rr, ord.rbt-union-swap-rec less'
f' γ' r' rr') ∈ ⟨Ra,Rb⟩rbt-rel
      apply (rule 1(2)[OF refl flip1 refl refl - - - refl])
      using 1(3) split-rel rbt-rel-s(1) defs
      by (auto simp: gamma-cong elim: rbt-rel-elims)
    have fun-rel-g-app: (g k v, g' k' v') ∈ Rb → Rb
      using fun-rel-g rbt-rel-s
      by (auto simp: defs elim: rbt-rel-elims dest: fun-relD)
    have (rbt-join (ord.rbt-union-swap-rec less f γ l ll) k (case β of None ⇒ v |
Some x ⇒ g k v x) (ord.rbt-union-swap-rec less f γ r rr),
     rbt-join (ord.rbt-union-swap-rec less' f' γ' l' ll') k' (case β' of None ⇒ v' |
Some x ⇒ g' k' v' x) (ord.rbt-union-swap-rec less' f' γ' r' rr')) ∈ ⟨Ra, Rb⟩rbt-rel
      apply parametricity
      using IH1 IH2 rbt-rel-s fun-rel-g-app split-rel
      by (auto simp: defs elim!: rbt-rel-elims)
  }
  then show ?case
   unfolding ord.rbt-union-swap-rec.simps[of - - - t1] ord.rbt-union-swap-rec.simps[of
- - - t1']
    using rbt-rel-fold rbt-rel-s
   by (auto simp: flip1[symmetric] flip2[symmetric] g-def[symmetric] g'-def[symmetric]
small-rbt-cong
        split: rbt.splits prod.splits elim: rbt-rel-elims)
 qed
qed

 lemma param-rbt-union[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows (ord.rbt-union less, ord.rbt-union less')
   ∈ ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding ord.rbt-union-def[abs-def] ord.rbt-union-with-key-def[abs-def]
  by parametricity

term rm-iterateoi
lemma param-rm-iterateoi[param]: (rm-iterateoi,rm-iterateoi)
 ∈ ⟨Ra,Rb⟩rbt-rel → (Rc→Id) → (⟨Ra,Rb⟩prod-rel → Rc → Rc) → Rc → Rc
 unfolding rm-iterateoi-def
 by (parametricity)

lemma param-rm-reverse-iterateoi[param]:
 (rm-reverse-iterateoi,rm-reverse-iterateoi)
  ∈ ⟨Ra,Rb⟩rbt-rel → (Rc→Id) → (⟨Ra,Rb⟩prod-rel → Rc → Rc) → Rc → Rc
 unfolding rm-reverse-iterateoi-def
 by (parametricity)


lemma param-color-eq[param]:
 ((=), (=))∈color-rel→color-rel→Id
```

by (*auto elim*: *color-rel.cases*)

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

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

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

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

term *ord.rbt-less*
lemma *rbt-less-param*[*param*]: (*ord.rbt-less*,*ord.rbt-less*) ∈
(*Rk*→*Rk*→*Id*) → *Rk* → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
unfolding *ord.rbt-less-prop*[*abs-def*]
apply (*parametricity add*: *param-list-ball*)
unfolding *RBT-Impl.keys-def RBT-Impl.entries-def*
apply (*parametricity*)
done

term *ord.rbt-greater*
lemma *rbt-greater-param*[*param*]: (*ord.rbt-greater*,*ord.rbt-greater*) ∈
(*Rk*→*Rk*→*Id*) → *Rk* → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
unfolding *ord.rbt-greater-prop*[*abs-def*]
apply (*parametricity add*: *param-list-ball*)
unfolding *RBT-Impl.keys-def RBT-Impl.entries-def*
apply (*parametricity*)
done

lemma *rbt-sorted-param*[*param*]:
(*ord.rbt-sorted*,*ord.rbt-sorted*)∈(*Rk*→*Rk*→*Id*)→⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
unfolding *ord.rbt-sorted-def*[*abs-def*]
by (*parametricity*)

lemma *is-rbt-param*[*param*]: (*ord.is-rbt*,*ord.is-rbt*) ∈
(*Rk*→*Rk*→*Id*) → ⟨*Rk*,*Rv*⟩*rbt-rel* → *Id*
unfolding *ord.is-rbt-def*[*abs-def*]
by (*parametricity*)

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

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

lemma *sorted-wrt-keys-true*[*simp*]: *sorted-wrt* (λ(-,-) (-,-). *True*) *l*
  apply (*induct l*)
  apply *auto*
  done

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

lemma *rbt-map-rel-def*:
  ⟨*Rk*,*Rv*⟩*rbt-map-rel lt* ≡ ⟨*Rk*,*Rv*⟩*rbt-rel O rbt-map-rel′ lt*
  by (*simp add*: *rbt-map-rel-def-internal relAPP-def*)

lemma (**in** *linorder*) *autoref-gen-rbt-empty*:
  (*rbt.Empty*,*Map.empty*) ∈ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
  by (*auto simp*: *rbt-map-rel-def*
    intro!: *rbt-map-impl rbt-rel-intros*)

lemma (**in** *linorder*) *autoref-gen-rbt-insert*:
  fixes *less-impl*
  assumes *param-less*: (*less-impl*,(<)) ∈ *Rk* → *Rk* → *Id*
  shows (*ord.rbt-insert less-impl*,λ*k v m. m*(*k*↦*v*)) ∈
    Rk → *Rv* → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (<)
  apply (*intro fun-relI*)
  unfolding *rbt-map-rel-def*
  apply (*auto intro*!: *relcomp.intros*)
  apply (*rule param-rbt-insert*[*OF param-less*, *param-fo*])
  apply *assumption*+
  apply (*rule rbt-map-impl*[*param-fo*])
  apply (*rule IdI* | *assumption*)+
  done

lemma (**in** *linorder*) *autoref-gen-rbt-lookup*:
fixes *less-impl*
assumes *param-less*: $(less\text{-}impl,(<)) \in Rk \to Rk \to Id$
shows $(ord.rbt\text{-}lookup\ less\text{-}impl,\ \lambda m\ k.\ m\ k) \in$
$\langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<) \to Rk \to \langle Rv\rangle option\text{-}rel$
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*subst R-O-Id*[*symmetric*])
apply (*rule relcompI*)
apply (*rule param-rbt-lookup*[*OF param-less*, *param-fo*])
apply *assumption+*
apply (*subst option-rel-id-simp*[*symmetric*])
apply (*rule rbt-map-impl*[*param-fo*])
apply *assumption*
apply (*rule IdI*)
done

lemma (**in** *linorder*) *autoref-gen-rbt-delete*:
fixes *less-impl*
assumes *param-less*: $(less\text{-}impl,(<)) \in Rk \to Rk \to Id$
shows $(ord.rbt\text{-}delete\ less\text{-}impl,\ \lambda k\ m.\ m\ |`(-\{k\})) \in$
$Rk \to \langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<) \to \langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<)$
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*rule relcompI*)
apply (*rule param-rbt-delete*[*OF param-less*, *param-fo*])
apply *assumption+*
apply (*rule rbt-map-impl*[*param-fo*])
apply (*rule IdI*)
apply *assumption*
done

lemma (**in** *linorder*) *autoref-gen-rbt-union*:
fixes *less-impl*
assumes *param-less*: $(less\text{-}impl,(<)) \in Rk \to Rk \to Id$
shows $(ord.rbt\text{-}union\ less\text{-}impl,\ (++)) \in$
$\langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<) \to \langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<) \to \langle Rk,Rv\rangle rbt\text{-}map\text{-}rel\ (<)$
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*rule relcompI*)
apply (*rule param-rbt-union*[*OF param-less*, *param-fo*])
apply *assumption+*

apply (*rule rbt-map-impl*[*param-fo*])
apply *assumption*+
done

A linear ordering on red-black trees

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

lemma (**in** *linorder*) *rbt-to-list-correct*:
assumes *SORTED*: *rbt-sorted t*
shows *rbt-to-list t* = *sorted-list-of-map* (*rbt-lookup t*) (**is** *?tl* = -)
proof −
from *map-it-to-list-linord-correct*[**where** *it*=*rm-iterateoi*, *OF*
rm-iterateoi-correct[*OF SORTED*]
] **have**
M: *map-of ?tl* = *rbt-lookup t*
and *D*: *distinct* (*map fst ?tl*)
and *S*: *sorted* (*map fst ?tl*)
by (*simp-all*)

from *the-sorted-list-of-map*[*OF D S*] *M* **show** *?thesis*
by *simp*
qed

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

lemma (**in** *linorder*) *param-rbt-sorted-list-of-map*[*param*]:
shows (*rbt-to-list*, *sorted-list-of-map*) ∈
⟨*Rk*, *Rv*⟩*rbt-map-rel* (<) → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
apply (*auto simp*: *rbt-map-rel-def rbt-map-rel′-def br-def*
rbt-to-list-correct[*symmetric*]
)
by (*parametricity*)

lemma *param-rbt-sorted-list-of-map′*[*param*]:
assumes *ELO*: *eq-linorder cmp′*
shows (*rbt-to-list*,*linorder.sorted-list-of-map* (*comp2le cmp′*)) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
proof −
interpret *linorder comp2le cmp′* *comp2lt cmp′*
using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
show *?thesis*
by *parametricity*
qed

lemma *rbt-linorder-impl*:
assumes *ELO*: *eq-linorder cmp′*
assumes [*param*]: (*cmp*,*cmp′*)∈*Rk*→*Rk*→*Id*

```
  shows
  (cmp-rbt cmp, cmp-map cmp′) ∈
    (Rv→Rv→Id)
    → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′)
    → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′) → Id
proof −
  interpret linorder comp2le cmp′   comp2lt cmp′
    using ELO by (simp add: eq-linorder-class-conv)

  show ?thesis
    unfolding cmp-map-def[abs-def] cmp-rbt-def[abs-def]
    apply (parametricity add: param-cmp-extend param-cmp-img)
    unfolding rbt-map-rel-def[abs-def] rbt-map-rel′-def br-def
    by auto
qed

lemma color-rel-sv[relator-props]: single-valued color-rel
  by (auto intro!: single-valuedI elim: color-rel.cases)

lemma rbt-rel-sv-aux:
  assumes SK: single-valued Rk
  assumes SV: single-valued Rv
  assumes I1: (a,b)∈(⟨Rk, Rv⟩rbt-rel)
  assumes I2: (a,c)∈(⟨Rk, Rv⟩rbt-rel)
  shows b=c
  using I1 I2
  apply (induct arbitrary: c)
  apply (elim rbt-rel-elims)
  apply simp
  apply (elim rbt-rel-elims)
  apply (simp add: single-valuedD[OF color-rel-sv]
    single-valuedD[OF SK] single-valuedD[OF SV])
  done

lemma rbt-rel-sv[relator-props]:
  assumes SK: single-valued Rk
  assumes SV: single-valued Rv
  shows single-valued (⟨Rk, Rv⟩rbt-rel)
  by (auto intro: single-valuedI rbt-rel-sv-aux[OF SK SV])

lemma rbt-map-rel-sv[relator-props]:
  ⟦single-valued Rk; single-valued Rv⟧
  ⟹ single-valued (⟨Rk,Rv⟩rbt-map-rel lt)
  apply (auto simp: rbt-map-rel-def rbt-map-rel′-def)
  apply (rule single-valued-relcomp)
  apply (rule rbt-rel-sv, assumption+)
  apply (rule br-sv)
  done
```

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

Second Part: Binding

lemma *autoref-rbt-empty*[*autoref-rules*]:
  assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp'*)
  assumes [*simplified,param*]: *GEN-OP cmp cmp'* (*Rk*→*Rk*→*Id*)
  shows (*rbt.Empty,op-map-empty*) ∈
    ⟨*Rk,Rv*⟩*rbt-map-rel* (*comp2lt cmp'*)
proof −
  interpret *linorder comp2le cmp'*   *comp2lt cmp'*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    by (*simp*) (*rule autoref-gen-rbt-empty*)
qed

lemma *autoref-rbt-update*[*autoref-rules*]:
  assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp'*)
  assumes [*simplified,param*]: *GEN-OP cmp cmp'* (*Rk*→*Rk*→*Id*)
  shows (*ord.rbt-insert* (*comp2lt cmp*),*op-map-update*) ∈
    Rk→*Rv*→⟨*Rk,Rv*⟩*rbt-map-rel* (*comp2lt cmp'*)
    → ⟨*Rk,Rv*⟩*rbt-map-rel* (*comp2lt cmp'*)
proof −
  interpret *linorder comp2le cmp'*   *comp2lt cmp'*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    unfolding *op-map-update-def*[*abs-def*]
    apply (*rule autoref-gen-rbt-insert*)
    unfolding *comp2lt-def*[*abs-def*]
    by (*parametricity*)
qed

lemma *autoref-rbt-lookup*[*autoref-rules*]:
  assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp'*)
  assumes [*simplified,param*]: *GEN-OP cmp cmp'* (*Rk*→*Rk*→*Id*)
  shows (λ*k t*. *ord.rbt-lookup* (*comp2lt cmp*) *t k*, *op-map-lookup*) ∈
    Rk → ⟨*Rk,Rv*⟩*rbt-map-rel* (*comp2lt cmp'*) → ⟨*Rv*⟩*option-rel*
proof −
  interpret *linorder comp2le cmp'*   *comp2lt cmp'*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    unfolding *op-map-lookup-def*[*abs-def*]
    apply (*intro fun-relI*)
    apply (*rule autoref-gen-rbt-lookup*[*param-fo*])
    apply (*unfold comp2lt-def*[*abs-def*]) []
    apply (*parametricity*)
    apply *assumption+*
    done
qed

lemma *autoref-rbt-delete*[*autoref-rules*]:
  assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
  assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
  shows (*ord.rbt-delete* (*comp2lt cmp*),*op-map-delete*) ∈
    Rk → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
      → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
proof −
  interpret *linorder comp2le cmp′ comp2lt cmp′*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    unfolding *op-map-delete-def*[*abs-def*]
    apply (*intro fun-relI*)
    apply (*rule autoref-gen-rbt-delete*[*param-fo*])
    apply (*unfold comp2lt-def*[*abs-def*]) []
    apply (*parametricity*)
    apply *assumption+*
    done
qed

lemma *autoref-rbt-union*[*autoref-rules*]:
  assumes *ELO*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
  assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*Rk*→*Rk*→*Id*)
  shows (*ord.rbt-union* (*comp2lt cmp*),(++)) ∈
    ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
      → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
proof −
  interpret *linorder comp2le cmp′ comp2lt cmp′*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    apply (*intro fun-relI*)
    apply (*rule autoref-gen-rbt-union*[*param-fo*])
    apply (*unfold comp2lt-def*[*abs-def*]) []
    apply (*parametricity*)
    apply *assumption+*
    done
qed

lemma *autoref-rbt-is-iterator*[*autoref-ga-rules*]:
  assumes *ELO*: *GEN-ALGO-tag* (*eq-linorder cmp′*)
  shows *is-map-to-sorted-list* (*comp2le cmp′*) *Rk Rv* (*rbt-map-rel* (*comp2lt cmp′*))
    rbt-to-list
proof −
  interpret *linorder comp2le cmp′ comp2lt cmp′*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)

  show *?thesis*
    unfolding *is-map-to-sorted-list-def*
      it-to-sorted-list-def

apply *auto*
proof –
fix $r\ m'$
assume $(r,\ m') \in \langle Rk,\ Rv\rangle rbt\text{-}map\text{-}rel\ (comp2lt\ cmp')$
then obtain r' **where** $R1$: $(r,r')\in\langle Rk,Rv\rangle rbt\text{-}rel$
and $R2$: $(r',m') \in rbt\text{-}map\text{-}rel'\ (comp2lt\ cmp')$
unfolding *rbt-map-rel-def* **by** *blast*

from *R2* **have** $is\text{-}rbt\ r'$ **and** M': $m' = rbt\text{-}lookup\ r'$
unfolding *rbt-map-rel′-def*
by (*simp-all add*: *br-def*)
hence *SORTED*: $rbt\text{-}sorted\ r'$
by (*simp add*: *is-rbt-def*)

from *map-it-to-list-linord-correct*[**where** *it* = *rm-iterateoi*, *OF*
rm-iterateoi-correct[*OF SORTED*]
] **have**
M: $map\text{-}of\ (rbt\text{-}to\text{-}list\ r') = rbt\text{-}lookup\ r'$
and D: $distinct\ (map\ fst\ (rbt\text{-}to\text{-}list\ r'))$
and S: $sorted\ (map\ fst\ (rbt\text{-}to\text{-}list\ r'))$
by (*simp-all*)

show $\exists\, l'.\ (rbt\text{-}to\text{-}list\ r,\ l') \in \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel\ \wedge$
$distinct\ l'\ \wedge$
$map\text{-}to\text{-}set\ m' = set\ l'\ \wedge$
$sorted\text{-}wrt\ (key\text{-}rel\ (comp2le\ cmp'))\ l'$
proof (*intro exI conjI*)
from *D* **show** $distinct\ (rbt\text{-}to\text{-}list\ r')$ **by** (*rule distinct-mapI*)
from *S* **show** $sorted\text{-}wrt\ (key\text{-}rel\ (comp2le\ cmp'))\ (rbt\text{-}to\text{-}list\ r')$
unfolding *key-rel-def*[*abs-def*]
by *simp*
show $(rbt\text{-}to\text{-}list\ r,\ rbt\text{-}to\text{-}list\ r') \in \langle\langle Rk,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
by (*parametricity add*: *R1*)
from *M* **show** $map\text{-}to\text{-}set\ m' = set\ (rbt\text{-}to\text{-}list\ r')$
by (*simp add*: M' *map-of-map-to-set*[*OF D*])
qed
qed
qed

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

lemma (**in** *linorder*) *dflt-cmp-id*:
$(dflt\text{-}cmp\ (\le)\ (<),\ dflt\text{-}cmp\ (\le)\ (<))\in Id\to Id\to Id$
by *auto*

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

lemma *rbt-linorder-autoref*[*autoref-rules*]:
  assumes *SIDE-GEN-ALGO* (*eq-linorder cmpk′*)
  assumes *SIDE-GEN-ALGO* (*eq-linorder cmpv′*)
  assumes *GEN-OP cmpk cmpk′* (*Rk*→*Rk*→*Id*)
  assumes *GEN-OP cmpv cmpv′* (*Rv*→*Rv*→*Id*)
  shows
  (*cmp-rbt cmpk cmpv*, *cmp-map cmpk′ cmpv′*) ∈
    ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk′*)
   → ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk′*) → *Id*
  apply (*intro fun-relI*)
  apply (*rule rbt-linorder-impl*[*param-fo*])
  using *assms*
  apply *simp-all*
  done

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*)
  using *assms* **apply** *simp-all*
  using *map-ord-eq-linorder* **.**

lemma *set-linorder-impl*[*autoref-ga-rules*]:
  assumes *GEN-ALGO-tag* (*eq-linorder cmpk*)
  shows *eq-linorder* (*cmp-set cmpk*)
  using *assms* **apply** *simp-all*
  using *set-ord-eq-linorder* **.**

lemma (**in** *linorder*) *rbt-map-rel-finite-aux*:
  finite-map-rel (⟨*Rk*,*Rv*⟩*rbt-map-rel* (<))
  unfolding *finite-map-rel-def*
  by (*auto simp*: *rbt-map-rel-def rbt-map-rel′-def br-def*)

lemma *rbt-map-rel-finite*[*relator-props*]:
  assumes *ELO*: *GEN-ALGO-tag* (*eq-linorder cmpk*)
  shows *finite-map-rel* (⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmpk*))
proof −
  interpret *linorder comp2le cmpk   comp2lt cmpk*
    using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
  show *?thesis*
    using *rbt-map-rel-finite-aux* **.**
qed

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

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

lemma [*simp,autoref-post-simps*]: *ord.rbt-ins* (<) = *rbt-ins*
proof (*intro ext*, *goal-cases*)
  case (*1 x xa xb xc*) **thus** *?case*
    apply (*induct x xa xb xc rule*: *rbt-ins.induct*)
    apply (*simp-all add*: *ord.rbt-ins.simps*)
    done
qed

lemma [*autoref-post-simps*]: *ord.rbt-lookup* ((<)::-::*linorder*⇒-) = *rbt-lookup*
  unfolding *ord.rbt-lookup-def rbt-lookup-def* **..**

lemma [*simp,autoref-post-simps*]:
  ord.rbt-insert-with-key (<) = *rbt-insert-with-key*
  ord.rbt-insert (<) = *rbt-insert*
  unfolding
    ord.rbt-insert-with-key-def[*abs-def*] *rbt-insert-with-key-def*[*abs-def*]
    ord.rbt-insert-def[*abs-def*] *rbt-insert-def*[*abs-def*]
  by *simp-all*

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 −
  from *ELC* **have** *1*: *eq-linorder cmp′* **by** *simp*
  show *?thesis*
    apply (*subst eq-linorder-comp2eq-eq*[*OF 1*,*symmetric*])
    by *parametricity*
qed

lemma *pi′-rm*[*icf-proper-iteratorI*]:
  proper-it′ rm-iterateoi rm-iterateoi
  proper-it′ rm-reverse-iterateoi rm-reverse-iterateoi
  apply (*rule proper-it′I*)
  apply (*rule pi-rm*)
  apply (*rule proper-it′I*)
  apply (*rule pi-rm-rev*)
  done

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*)
by (*simp add*: *relAPP-def fun-set-rel-internal-def*)

lemma *fun-set-rel-sv*[*relator-props*]:
⟦*single-valued R*; *Range R* = *UNIV*⟧ ⟹ *single-valued* (⟨*R*⟩*fun-set-rel*)
unfolding *fun-set-rel-def*
by (*tagged-solver* (*keep*))

lemma *fun-set-rel-RUNIV*[*relator-props*]:
assumes *SV*: *single-valued R*
shows *Range* (⟨*R*⟩*fun-set-rel*) = *UNIV*
proof −
{
fix *b*
have ∃ *a*. (*a*,*b*)∈⟨*R*⟩*fun-set-rel* **unfolding** *fun-set-rel-def*
apply (*rule exI*)
apply (*rule relcompI*)
proof −
show ((λ*x*. *x*∈*b*),*b*)∈*br Collect* (λ-. *True*) **by** (*auto simp*: *br-def*)
show (λ*x'*. ∃ *x*. (*x'*,*x*)∈*R* ∧ *x*∈*b*,λ*x*. *x* ∈ *b*)∈*R* → *bool-rel*
by (*auto dest*: *single-valuedD*[*OF SV*])
qed
} **thus** *?thesis* **by** *blast*
qed

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

lemma *fs-mem-refine*[*autoref-rules*]: (λ*x f*. *f x*,(∈)) ∈ *R* → ⟨*R*⟩*fun-set-rel* → *bool-rel*
apply (*intro fun-relI*)
apply (*auto simp add*: *fun-set-rel-def br-def dest*: *fun-relD*)
done

lemma *fun-set-Collect-refine*[*autoref-rules*]:
  $(\lambda x.\ x,\ Collect)\in(R\rightarrow bool\text{-}rel) \rightarrow \langle R\rangle fun\text{-}set\text{-}rel$
  unfolding *fun-set-rel-def*
  by (*auto simp*: *br-def*)

lemma *fun-set-empty-refine*[*autoref-rules*]:
  $(\lambda\text{-}.\ False,\{\})\in\langle R\rangle fun\text{-}set\text{-}rel$
  by (*force simp add*: *fun-set-rel-def br-def*)

lemma *fun-set-UNIV-refine*[*autoref-rules*]:
  $(\lambda\text{-}.\ True, UNIV)\in\langle R\rangle fun\text{-}set\text{-}rel$
  by (*force simp add*: *fun-set-rel-def br-def*)

lemma *fun-set-union-refine*[*autoref-rules*]:
  $(\lambda a\ b\ x.\ a\ x \lor b\ x,(\cup))\in\langle R\rangle fun\text{-}set\text{-}rel \rightarrow \langle R\rangle fun\text{-}set\text{-}rel \rightarrow \langle R\rangle fun\text{-}set\text{-}rel$
proof −
  have *A*: $\bigwedge a\ b.\ (\lambda x.\ x\in a \lor x\in b,\ a \cup b) \in br\ Collect\ (\lambda\text{-}.\ True)$
    by (*auto simp*: *br-def*)

  show *?thesis*
    apply (*simp add*: *fun-set-rel-def*)
    apply (*intro fun-relI*)
    apply *clarsimp*
    apply *rule*
    defer
    apply (*rule A*)
    apply (*auto simp*: *br-def dest*: *fun-relD*)
    done
qed

lemma *fun-set-inter-refine*[*autoref-rules*]:
  $(\lambda a\ b\ x.\ a\ x \land b\ x,(\cap))\in\langle R\rangle fun\text{-}set\text{-}rel \rightarrow \langle R\rangle fun\text{-}set\text{-}rel \rightarrow \langle R\rangle fun\text{-}set\text{-}rel$
proof −
  have *A*: $\bigwedge a\ b.\ (\lambda x.\ x\in a \land x\in b,\ a \cap b) \in br\ Collect\ (\lambda\text{-}.\ True)$
    by (*auto simp*: *br-def*)

  show *?thesis*
    apply (*simp add*: *fun-set-rel-def*)
    apply (*intro fun-relI*)
    apply *clarsimp*
    apply *rule*
    defer
    apply (*rule A*)
    apply (*auto simp*: *br-def dest*: *fun-relD*)
    done
qed

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 −
  have *A*: $\bigwedge a\ b.\ (\lambda x.\ x \in a \wedge \neg x \in b,\ a - b) \in$ *br Collect* (λ-. *True*)
    by (*auto simp*: *br-def*)

  show *?thesis*
    apply (*simp add*: *fun-set-rel-def*)
    apply (*intro fun-relI*)
    apply *clarsimp*
    apply *rule*
    defer
    apply (*rule A*)
    apply (*auto simp*: *br-def dest*: *fun-relD*)
    done
qed

end

2.3.7 Array-Based Maps with Natural Number Keys

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

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

Definitions

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

lemma [*code*]: *iam-α a i* ≡ *array-get-oo None a i*
  unfolding *array-get-oo-def iam-α-def* **.**

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

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

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

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

lemma *incr-correct*: ¬ *idx* < *l* ⟹ *idx* < *l* + *iam-increment l idx*
  unfolding *iam-increment-def* **by** *auto*

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

lemma [*code*]: *iam-update k v a* ≡ *array-set-oo*
  (λ-. *let l*=*array-length a in*
        array-set (*array-grow a* (*iam-increment l k*) *None*) *k* (*Some v*))
  a k (*Some v*)
  unfolding *iam-update-def array-set-oo-def*
  apply (*rule eq-reflection*)
  by *auto*

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

lemma [*code*]: *iam-delete k a* ≡ *array-set-oo* (λ-. *a*) *a k None*
  unfolding *iam-delete-def array-set-oo-def* **by** *auto*

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

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

Parametricity

definition *iam-rel-def-internal*:
  iam-rel R ≡ ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
lemma *iam-rel-def*: ⟨*R*⟩ *iam-rel* = ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
    by (*simp add*: *iam-rel-def-internal relAPP-def*)

lemma *iam-rel-sv*[*relator-props*]:
single-valued Rv $\Longrightarrow$ *single-valued* ($\langle Rv\rangle$*iam-rel*)
unfolding *iam-rel-def*
by *tagged-solver*

lemma *param-iam-*α[*param*]:
(*iam-*α, *iam-*α) $\in$ $\langle R\rangle$ *iam-rel* $\to$ *nat-rel* $\to$ $\langle R\rangle$ *option-rel*
unfolding *iam-*α*-def*[*abs-def*] *iam-rel-def* **by** *parametricity*

lemma *param-iam-invar*[*param*]:
(*iam-invar*, *iam-invar*) $\in$ $\langle R\rangle$ *iam-rel* $\to$ *bool-rel*
unfolding *iam-rel-def* **by** *parametricity*

lemma *param-iam-empty*[*param*]:
(*iam-empty*, *iam-empty*) $\in$ *unit-rel* $\to$ $\langle R\rangle$*iam-rel*
unfolding *iam-empty-def*[*abs-def*] *iam-rel-def* **by** *parametricity*

lemma *param-iam-lookup*[*param*]:
(*iam-lookup*, *iam-lookup*) $\in$ *nat-rel* $\to$ $\langle R\rangle$*iam-rel* $\to$ $\langle R\rangle$*option-rel*
unfolding *iam-lookup-def*[*abs-def*]
by *parametricity*

lemma *param-iam-increment*[*param*]:
(*iam-increment*, *iam-increment*) $\in$ *nat-rel* $\to$ *nat-rel* $\to$ *nat-rel*
unfolding *iam-increment-def*[*abs-def*]
by *simp*

lemma *param-iam-update*[*param*]:
(*iam-update*, *iam-update*) $\in$ *nat-rel* $\to$ *R* $\to$ $\langle R\rangle$*iam-rel* $\to$ $\langle R\rangle$*iam-rel*
unfolding *iam-update-def*[*abs-def*] *iam-rel-def Let-def*
apply *parametricity*
done

lemma *param-iam-delete*[*param*]:
(*iam-delete*, *iam-delete*) $\in$ *nat-rel* $\to$ $\langle R\rangle$*iam-rel* $\to$ $\langle R\rangle$*iam-rel*
unfolding *iam-delete-def*[*abs-def*] *iam-rel-def* **by** *parametricity*

lemma *param-iam-iteratei-aux*[*param*]:
assumes *I*: $i \le$ *array-length a*
assumes *IR*: $(i,i') \in$ *nat-rel*
assumes *AR*: $(a,a') \in$ $\langle Ra\rangle$*iam-rel*
assumes *CR*: $(c,c') \in Rb \to$ *bool-rel*
assumes *FR*: $(f,f') \in$ $\langle$*nat-rel*,*Ra*$\rangle$*prod-rel* $\to Rb \to Rb$
assumes σR: $(\sigma,\sigma') \in Rb$
shows (*iam-iteratei-aux i a c f* σ, *iam-iteratei-aux* i' a' c' f' σ') $\in Rb$
using *assms*

unfolding *iam-rel-def*
apply (*induct* i' *arbitrary*: i σ σ')
apply (*simp-all only*: *pair-in-Id-conv iam-iteratei-aux.simps*)
apply *parametricity*
apply *simp-all*
done

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*
unfolding *iam-iteratei-def*[*abs-def*]
by *parametricity* (*simp-all add*: *iam-rel-def*)

Correctness

definition *iam-rel'* $\equiv$ *br iam-*α *iam-invar*

lemma *iam-empty-correct*:
(*iam-empty* (), *Map.empty*) $\in$ *iam-rel'*
by (*simp add*: *iam-rel'-def br-def iam-*α*-def*[*abs-def*] *iam-empty-def*)

lemma *iam-update-correct*:
(*iam-update*,*op-map-update*) $\in$ *nat-rel* $\rightarrow$ *Id* $\rightarrow$ *iam-rel'* $\rightarrow$ *iam-rel'*
by (*auto simp*: *iam-rel'-def br-def Let-def array-get-array-set-other*
incr-correct iam-α*-def*[*abs-def*] *iam-update-def*)

lemma *iam-lookup-correct*:
(*iam-lookup*,*op-map-lookup*) $\in$ *Id* $\rightarrow$ *iam-rel'* $\rightarrow$ $\langle Id\rangle$*option-rel*
by (*auto simp*: *iam-rel'-def br-def iam-lookup-def*[*abs-def*])

lemma *array-get-set-iff*: $i<$*array-length* $a \Longrightarrow$
array-get (*array-set* a i x) j = (*if* $i=j$ *then* x *else* *array-get* a j)
by (*auto simp*: *array-get-array-set-other*)

lemma *iam-delete-correct*:
(*iam-delete*,*op-map-delete*) $\in$ *Id* $\rightarrow$ *iam-rel'* $\rightarrow$ *iam-rel'*
unfolding *iam-*α*-def*[*abs-def*] *iam-delete-def*[*abs-def*] *iam-rel'-def br-def*
by (*auto simp*: *Let-def array-get-set-iff*)

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*:
$\langle$*nat-rel*,*Rv*$\rangle$*iam-map-rel* $\equiv$ $\langle Rv\rangle$*iam-rel O iam-rel'*
unfolding *iam-map-rel-def-internal relAPP-def* **by** *simp*

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

lemma *iam-map-rel-sv*[*relator-props*]:
  single-valued Rv ⟹ *single-valued* (⟨*nat-rel*,*Rv*⟩*iam-map-rel*)
  unfolding *iam-map-rel-def iam-rel′-def* **by** *tagged-solver*

lemma *iam-empty-impl*:
    (*iam-empty* (), *op-map-empty*) ∈ ⟨*nat-rel*,*R*⟩*iam-map-rel*
  unfolding *iam-map-rel-def op-map-empty-def*
  apply (*intro relcompI*)
  apply (*rule param-iam-empty*[*THEN fun-relD*], *simp*)
  apply (*rule iam-empty-correct*)
  done

lemma *iam-lookup-impl*:
    (*iam-lookup*, *op-map-lookup*)
  ∈ *nat-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*R*⟩*option-rel*
unfolding *iam-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcompE*)
apply (*frule iam-lookup-correct*[*param-fo*], *assumption*)
apply (*frule param-iam-lookup*[*param-fo*], *assumption*)
apply *simp*
done

lemma *iam-update-impl*:
   (*iam-update*, *op-map-update*) ∈
      nat-rel → *R* → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel*
  unfolding *iam-map-rel-def*
  apply (*intro fun-relI*, *elim relcompEpair*, *intro relcompI*)
  apply (*erule* (*2*) *param-iam-update*[*param-fo*])
  apply (*rule iam-update-correct*[*param-fo*])
  apply *simp-all*
  done

lemma *iam-delete-impl*:
    (*iam-delete*, *op-map-delete*) ∈
        nat-rel → ⟨*nat-rel*,*R*⟩*iam-map-rel* → ⟨*nat-rel*,*R*⟩*iam-map-rel*
  unfolding *iam-map-rel-def*
  apply (*intro fun-relI*, *elim relcompEpair*, *intro relcompI*)
  apply (*erule* (*1*) *param-iam-delete*[*param-fo*])
  apply (*rule iam-delete-correct*[*param-fo*])
  by *simp-all*

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

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

Iterator proofs

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

lemma *distinct-iam-to-list-aux*:
  shows ⟦*distinct xs*; ∀ (*i*,-)∈*set xs*. $i \geq n$⟧ ⟹
    distinct (*iam-iteratei-aux n a*
      (λ-. *True*) (λ*x y*. *y* @ [*x*]) *xs*)
  (**is** ⟦-;-⟧ ⟹ *distinct* (*?iam-to-list-aux n xs*))
proof (*induction n arbitrary*: *xs*)
  case (*0 xs*) **thus** *?case* **by** *simp*
next
  case (*Suc i xs*)
    show *?case*
    proof (*cases array-get a i*)
      case *None*
        with *Suc.IH*[*OF Suc.prems*(*1*)] *Suc.prems*(*2*)
          show *?thesis* **by** *force*
    next
      case (*Some x*)
        let *?xs'* = *xs* @ [(*i*,*x*)]
        from *Suc.prems* **have** *distinct ?xs'* **and**
          ∀ (*i'*,*x*)∈*set ?xs'*. $i' \geq i$ **by** *force*+
        from *Some* **and** *Suc.IH*[*OF this*] **show** *?thesis* **by** *simp*
  qed
qed

lemma *distinct-iam-to-list*:
  distinct (*iam-to-list a*)
unfolding *it-to-list-def iam-iteratei-def*
  by (*force intro*: *distinct-iam-to-list-aux*)

lemma *iam-to-list-set-correct-aux*:
  assumes (*a*, *m*) ∈ *iam-rel'*
  shows ⟦$n \leq$ *array-length a*; *map-to-set m* − {(*k*,*v*). $k < n$} = *set xs*⟧
    ⟹ *map-to-set m* =
      set (*iam-iteratei-aux n a* (λ-. *True*) (λ*x y*. *y* @ [*x*]) *xs*)
proof (*induction n arbitrary*: *xs*)
  case (*0 xs*)
    thus *?case* **by** *simp*
next
  case (*Suc n xs*)
    with *assms* **have** [*simp*]: *array-get a n* = *m n*
      unfolding *iam-rel'-def br-def iam-α-def*[*abs-def*] **by** *simp*
    show *?case*
    proof (*cases m n*)

```
    case None
      with Suc.prems(2) have map-to-set m − {(k,v). k < n} = set xs
      unfolding map-to-set-def by (fastforce simp: less-Suc-eq)
      from None and Suc.IH[OF - this] and Suc.prems(1)
        show ?thesis by simp
  next
    case (Some x)
      let ?xs′ = xs @ [(n,x)]
      from Some and Suc.prems(2)
        have map-to-set m − {(k,v). k < n} = set ?xs′
        unfolding map-to-set-def by (fastforce simp: less-Suc-eq)
      from Some and Suc.IH[OF - this] and Suc.prems(1)
        show ?thesis by simp
  qed
qed

lemma iam-to-list-set-correct:
 assumes (a, m) ∈ iam-rel′
 shows map-to-set m = set (iam-to-list a)
proof−
 from assms
    have A: map-to-set m − {(k, v). k < array-length a} = set []
    unfolding map-to-set-def iam-rel′-def br-def iam-α-def[abs-def]
    by (force split: if-split-asm)
 with iam-to-list-set-correct-aux[OF assms - A] show ?thesis
  unfolding it-to-list-def iam-iteratei-def by simp
qed

lemma iam-iteratei-aux-append:
 n ≤ length xs ⟹ iam-iteratei-aux n (Array (xs @ ys)) =
           iam-iteratei-aux n (Array xs)
apply (induction n)
apply force
apply (intro ext, auto split: option.split simp: nth-append)
done

lemma iam-iteratei-append:
 iam-iteratei (Array (xs @ [None])) c f σ =
     iam-iteratei (Array xs) c f σ
 iam-iteratei (Array (xs @ [Some x])) c f σ =
     iam-iteratei (Array xs) c f
     (if c σ then (f (length xs, x) σ) else σ)
unfolding iam-iteratei-def
apply (cases length xs)
apply (simp add: iam-iteratei-aux-append)
apply (force simp: nth-append iam-iteratei-aux-append) []
apply (cases length xs)
apply (simp add: iam-iteratei-aux-append)
apply (force split: option.split
```

simp: *nth-append iam-iteratei-aux-append*) []
done

lemma *iam-iteratei-aux-Cons*:
$n < array\text{-}length\ a \Longrightarrow$
$iam\text{-}iteratei\text{-}aux\ n\ a\ (\lambda\text{-}.\ True)\ (\lambda x\ l.\ l\ @\ [x])\ (x\#xs) =$
$x\ \#\ iam\text{-}iteratei\text{-}aux\ n\ a\ (\lambda\text{-}.\ True)\ (\lambda x\ l.\ l\ @\ [x])\ xs$
by (*induction n arbitrary*: *xs*, *auto split*: *option.split*)

lemma *iam-to-list-append*:
iam-to-list (*Array* (*xs* @ [*None*])) = *iam-to-list* (*Array xs*)
iam-to-list (*Array* (*xs* @ [*Some x*])) =
(*length xs*, *x*) # *iam-to-list* (*Array xs*)
unfolding *it-to-list-def iam-iteratei-def*
apply (*simp add*: *iam-iteratei-aux-append*)
apply (*simp add*: *iam-iteratei-aux-Cons*)
apply (*simp add*: *iam-iteratei-aux-append*)
done

lemma *autoref-iam-is-iterator*[*autoref-ga-rules*]:
shows *is-map-to-list nat-rel Rv iam-map-rel iam-to-list*
unfolding *is-map-to-list-def is-map-to-sorted-list-def*
proof (*clarify*)
fix $a\ m'$
assume $(a,m') \in \langle nat\text{-}rel,Rv\rangle iam\text{-}map\text{-}rel$
then obtain a' **where** [*param*]: $(a,a')\in\langle Rv\rangle iam\text{-}rel$
and $(a',m')\in iam\text{-}rel'$ **unfolding** *iam-map-rel-def* **by** *blast*

have (*iam-to-list a*, *iam-to-list a'*)
$\in \langle\langle nat\text{-}rel,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$ **by** *parametricity*

moreover from *distinct-iam-to-list* **and**
iam-to-list-set-correct[*OF* ‹$(a',m')\in iam\text{-}rel'$›]
have *RETURN* (*iam-to-list a'*) $\leq$ *it-to-sorted-list*
(*key-rel* (λ- -. *True*)) (*map-to-set m'*)
unfolding *it-to-sorted-list-def key-rel-def*[*abs-def*]
by (*force intro*: *refine-vcg*)

ultimately show $\exists\, l'.\ (iam\text{-}to\text{-}list\ a,\ l') \in$
$\langle\langle nat\text{-}rel,\ Rv\rangle prod\text{-}rel\rangle list\text{-}rel$
$\wedge\ RETURN\ l' \leq it\text{-}to\text{-}sorted\text{-}list\ ($
key-rel (λ- -. *True*)) (*map-to-set m'*) **by** *blast*
qed

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

```
lemma iam-iteratei-altdef:
  iam-iteratei a = foldli (iam-to-list a)
   (is ?f a = ?g (iam-to-list a))
proof−
 obtain l where a = Array l by (cases a)
 have ?f (Array l) = ?g (iam-to-list (Array l))
 proof (induction length l arbitrary: l)
  case (0 l)
   thus ?case by (intro ext,
     simp add: iam-iteratei-def it-to-list-def)
 next
  case (Suc n l)
   hence l ≠ [] and [simp]: length l = Suc n by force+
   with append-butlast-last-id have [simp]:
     butlast l @ [last l] = l by simp
   with Suc have [simp]: length (butlast l) = n by simp
   note IH = Suc(1)[OF this[symmetric]]
   let ?l' = iam-to-list (Array l)

   show ?case
   proof (cases last l)
    case None
     have ?f (Array l) =
       ?f (Array (butlast l @ [last l])) by simp
     also have ... = ?g (iam-to-list (Array (butlast l)))
       by (force simp: None iam-iteratei-append IH)
     also have iam-to-list (Array (butlast l)) =
       iam-to-list (Array (butlast l @ [last l]))
       using None by (simp add: iam-to-list-append)
     finally show ?f (Array l) = ?g ?l' by simp
   next
    case (Some x)
     have ?f (Array l) =
       ?f (Array (butlast l @ [last l])) by simp
     also have ... = ?g (iam-to-list
       (Array (butlast l @ [last l])))
       by (force simp: IH iam-iteratei-append
           iam-to-list-append Some)
     finally show ?thesis by simp
   qed
 qed
 thus ?thesis by (simp add: ‹a = Array l›)
qed


lemma pi-iam[icf-proper-iteratorI]:
 proper-it (iam-iteratei a) (iam-iteratei a)
unfolding proper-it-def by (force simp: iam-iteratei-altdef)
```

lemma *pi′-iam*[*icf-proper-iteratorI*]:
proper-it′ iam-iteratei iam-iteratei
by (*rule proper-it′I*, *rule pi-iam*)

end

Chapter 3

The Original Isabelle Collection Framework

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

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

3.1 Specifications

3.1.1 Specification of Sets

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

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

notation *insert* (‹*set′-ins*›)

type-synonym $('x,'s)$ *set-α* = $'s \Rightarrow 'x$ *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*!]: $\bigwedge s.$ *invar s*

Basic Set Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *isSng-correct-card* :
$invar\ s \Longrightarrow (isSng\ s \longleftrightarrow (card\ (\alpha\ s) = 1))$
by (*auto simp add*: *isSng-correct card-Suc-eq*)
end

Iterators

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

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

```
  = 's ⇒ ('x,'x list) set-iterator
locale poly-set-iteratei-defs =
  fixes list-it :: 's ⇒ ('x,'x list) set-iterator
begin
  definition iteratei :: 's ⇒ ('x,'σ) set-iterator
    where iteratei S ≡ it-to-it (list-it S)


  abbreviation iterate s ≡ iteratei s (λ-. True)
end

locale poly-set-iteratei =
  finite-set + poly-set-iteratei-defs list-it
  for list-it :: 's ⇒ ('x,'x list) set-iterator +
  constrains α :: 's ⇒ 'x set
  assumes list-it-correct: invar s ⟹ set-iterator (list-it s) (α s)
begin
  lemma iteratei-correct: invar S ⟹ set-iterator (iteratei S) (α S)
    unfolding iteratei-def
    apply (rule it-to-it-correct)
    by (rule list-it-correct)

  lemma pi-iteratei[icf-proper-iteratorI]:
    proper-it (iteratei S) (iteratei S)
    unfolding iteratei-def
    by (intro icf-proper-iteratorI)

  lemma iteratei-rule-P:
    ⟦
      invar S;
      I (α S) σ0;
      !!x it σ. ⟦ c σ; x ∈ it; it ⊆ α S; I it σ ⟧ ⟹ I (it − {x}) (f x σ);
      !!σ. I {} σ ⟹ P σ;
      !!σ it. ⟦ it ⊆ α S; it ≠ {}; ¬ c σ; I it σ ⟧ ⟹ P σ
    ⟧ ⟹ P (iteratei S c f σ0)
   apply (rule set-iterator-rule-P [OF iteratei-correct, of S I σ0 c f P])
   apply simp-all
  done

  lemma iteratei-rule-insert-P:
    ⟦
      invar S;
      I {} σ0;
      !!x it σ. ⟦ c σ; x ∈ α S − it; it ⊆ α S; I it σ ⟧ ⟹ I (insert x it) (f x σ);
      !!σ. I (α S) σ ⟹ P σ;
      !!σ it. ⟦ it ⊆ α S; it ≠ α S; ¬ c σ; I it σ ⟧ ⟹ P σ
    ⟧ ⟹ P (iteratei S c f σ0)
    apply (rule
      set-iterator-rule-insert-P[OF iteratei-correct, of S I σ0 c f P])
```

apply *simp-all*
done

Versions without break condition.

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

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

end

More Set Operations

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

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

$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow invar3\ (union\ s1\ s2)$

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

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

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

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

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

assumes *subset-correct*:
$invar1\ s1 \Longrightarrow invar2\ s2 \Longrightarrow subset\ s1\ s2 \longleftrightarrow \alpha 1\ s1 \subseteq \alpha 2\ s2$

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

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

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

end

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

Image **locale** $set\text{-}image = s1\colon set\ \alpha 1\ invar1 + s2\colon set\ \alpha 2\ invar2$
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** $invar1$
and $\alpha 2 :: 's2 \Rightarrow 'b\ set$ **and** $invar2$
+
fixes $image :: ('a \Rightarrow 'b) \Rightarrow 's1 \Rightarrow 's2$
assumes *image-correct*:

$invar1\ s \Longrightarrow \alpha 2\ (image\ f\ s) = f`\alpha 1\ s$
$invar1\ s \Longrightarrow invar2\ (image\ f\ s)$

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

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

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

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

Disjointness Check With Witness **locale** *set-disjoint-witness* = *s1*: *set* $\alpha 1$ *invar1* + *s2*: *set* $\alpha 2$ *invar2*
for $\alpha 1 :: 's1 \Rightarrow 'a\ set$ **and** *invar1*
and $\alpha 2 :: 's2 \Rightarrow 'a\ set$ **and** *invar2*
+
fixes $disjoint\text{-}witness :: 's1 \Rightarrow 's2 \Rightarrow 'a\ option$

assumes *disjoint-witness-correct*:
$[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \Longrightarrow \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}$
$[\![invar1\ s1;\ invar2\ s2;\ disjoint\text{-}witness\ s1\ s2 = Some\ a]\!]$
$\Longrightarrow a \in \alpha 1\ s1 \cap \alpha 2\ s2$
begin
lemma *disjoint-witness-None*: $[\![invar1\ s1;\ invar2\ s2]\!]$
$\Longrightarrow disjoint\text{-}witness\ s1\ s2 = None \longleftrightarrow \alpha 1\ s1 \cap \alpha 2\ s2 = \{\}$
by (*case-tac disjoint-witness s1 s2*)
(*auto dest*: *disjoint-witness-correct*)

lemma *disjoint-witnessI*: $[\![$
$invar1\ s1;$
$invar2\ s2;$
$\alpha 1\ s1 \cap \alpha 2\ s2 \neq \{\};$
$!!a.\ [\![disjoint\text{-}witness\ s1\ s2 = Some\ a]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*case-tac disjoint-witness s1 s2*)
(*auto dest*: *disjoint-witness-correct*)

end

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

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

lemma *sel-noneD*:
$[\![\ invar\ s;\ sel\ s\ f = None;\ x \in \alpha\ s\ ]\!] \Longrightarrow f\ x = None$
apply (*cases* $\exists x \in \alpha\ s.\ \exists r.\ f\ x = Some\ r$)
apply (*safe*)
apply (*erule-tac f=f* **and** *x=xa* **in** *selE*)
apply *auto*
done

end

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

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

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

Conversion of Set to List **locale** *set-to-list* = *set* +
  constrains α :: $'s \Rightarrow 'x$ *set*
  fixes *to-list* :: $'s \Rightarrow 'x$ *list*
  assumes *to-list-correct*:
    invar s $\Longrightarrow$ *set* (*to-list s*) = α *s*
    invar s $\Longrightarrow$ *distinct* (*to-list s*)

Conversion of List to Set **locale** *list-to-set* = *set* +
  constrains α :: $'s \Rightarrow 'x$ *set*
  fixes *to-set* :: $'x$ *list* $\Rightarrow 's$
  assumes *to-set-correct*:
    α (*to-set l*) = *set l*
    invar (*to-set l*)

Ordered Sets

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

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

Ordered Iteration **locale** *poly-set-iterateoi-defs* =
  fixes *olist-it* :: *'s* ⇒ (*'x*,*'x list*) *set-iterator*
begin
  definition *iterateoi* :: *'s* ⇒ (*'x*,*'σ*) *set-iterator*
    where *iterateoi S* ≡ *it-to-it* (*olist-it S*)

  abbreviation *iterateo s* ≡ *iterateoi s* (λ-. *True*)
end

locale *poly-set-iterateoi* =
  finite-set α *invar* + *poly-set-iterateoi-defs list-ordered-it*
  for α :: *'s* ⇒ *'x::linorder set*
  and *invar*
  and *list-ordered-it* :: *'s* ⇒ (*'x*,*'x list*) *set-iterator* +
  assumes *list-ordered-it-correct*: *invar x*
    ⟹ *set-iterator-linord* (*list-ordered-it x*) (α *x*)
begin
  lemma *iterateoi-correct*:
    invar S ⟹ *set-iterator-linord* (*iterateoi S*) (α *S*)
    unfolding *iterateoi-def*
    apply (*rule it-to-it-linord-correct*)
    by (*rule list-ordered-it-correct*)

  lemma *pi-iterateoi*[*icf-proper-iteratorI*]:
    proper-it (*iterateoi S*) (*iterateoi S*)
    unfolding *iterateoi-def*
    by (*intro icf-proper-iteratorI*)

  lemma *iterateoi-rule-P*[*case-names minv inv0 inv-pres i-complete i-inter*]:
    assumes *MINV*: *invar s*
    assumes *I0*: *I* (α *s*) *σ0*
    assumes *IP*: !!*k it σ*. ⟦
      c σ;
      k ∈ *it*;
      ∀ *j*∈*it*. *k*≤*j*;
      ∀ *j*∈α *s* − *it*. *j*≤*k*;
      it ⊆ α *s*;
      I it σ
    ⟧ ⟹ *I* (*it* − {*k*}) (*f k σ*)
    assumes *IF*: !!*σ*. *I* {} *σ* ⟹ *P σ*

```
    assumes II: !!σ it. ⟦
      it ⊆ α s;
      it ≠ {};
      ¬ c σ;
      I it σ;
      ∀ k∈it. ∀ j∈α s − it. j≤k
    ⟧ ⟹ P σ
    shows P (iterateoi s c f σ0)
  using set-iterator-linord-rule-P [OF iterateoi-correct,
    OF MINV, of I σ0 c f P, OF I0 - IF] IP II
  by simp

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

locale poly-set-rev-iterateoi-defs =
  fixes list-rev-it :: 's ⇒ ('x::linorder,'x list) set-iterator
begin
  definition rev-iterateoi :: 's ⇒ ('x,'σ) set-iterator
    where rev-iterateoi S ≡ it-to-it (list-rev-it S)


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

locale poly-set-rev-iterateoi =
  finite-set α invar + poly-set-rev-iterateoi-defs list-rev-it
  for α :: 's ⇒ 'x::linorder set
  and invar
  and list-rev-it :: 's ⇒ ('x,'x list) set-iterator +
  assumes list-rev-it-correct:
    invar s ⟹ set-iterator-rev-linord (list-rev-it s) (α s)
begin
  lemma rev-iterateoi-correct:
    invar S ⟹ set-iterator-rev-linord (rev-iterateoi S) (α S)
    unfolding rev-iterateoi-def
    apply (rule it-to-it-rev-linord-correct)
```

by (*rule list-rev-it-correct*)

lemma *pi-rev-iterateoi*[*icf-proper-iteratorI*]:
proper-it (*rev-iterateoi S*) (*rev-iterateoi S*)
unfolding *rev-iterateoi-def*
by (*intro icf-proper-iteratorI*)

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*: $!!k\ it\ \sigma.\ [\![$
$c\ \sigma;$
$k \in it;$
$\forall j{\in}it.\ k{\geq}j;$
$\forall j{\in}(\alpha\ s) - it.\ j{\geq}k;$
$it \subseteq (\alpha\ s);$
$I\ it\ \sigma$
$]\!] \Longrightarrow I\ (it - \{k\})\ (f\ k\ \sigma)$
assumes *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
assumes *II*: $!!\sigma\ it.\ [\![$
$it \subseteq (\alpha\ s);$
$it \neq \{\};$
$\neg\ c\ \sigma;$
$I\ it\ \sigma;$
$\forall k{\in}it.\ \forall j{\in}(\alpha\ s) - it.\ j{\geq}k$
$]\!] \Longrightarrow P\ \sigma$
shows P (*rev-iterateoi s c f* $\sigma 0$)
using *set-iterator-rev-linord-rule-P* [*OF rev-iterateoi-correct*,
OF MINV, *of I* $\sigma 0$ *c f P*, *OF I0 - IF*] *IP II*
by *simp*

lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar s*
assumes *I0*: $I\ ((\alpha\ s))\ \sigma 0$
assumes *IP*: $!!k\ it\ \sigma.\ [\![$
$k \in it;$
$\forall j{\in}it.\ k{\geq}j;$
$\forall j{\in}\ (\alpha\ s) - it.\ j{\geq}k;$
$it \subseteq (\alpha\ s);$
$I\ it\ \sigma$
$]\!] \Longrightarrow I\ (it - \{k\})\ (f\ k\ \sigma)$
assumes *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
shows P (*rev-iterateo s f* $\sigma 0$)
apply (*rule rev-iterateoi-rule-P* [**where** $I = I$])
apply (*simp-all add*: *assms*)
done

end

Minimal and Maximal Element **locale** *set-min* = *ordered-set* +
  constrains α :: $'s \Rightarrow 'u$::*linorder set*
  fixes *min* :: $'s \Rightarrow ('u \Rightarrow bool) \Rightarrow 'u$ *option*
  assumes *min-correct*:
    $\llbracket$ *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* $\longrightarrow x' \leq x$
 proof −
  from *min-correct*(*1*)[*of s x P*, *OF A*] **have**
   MIS: *min s P* $\in$ *Some* ‘ $\{x \in \alpha$ *s*. *P x*$\}$ **.**
  then obtain x' **where** *KV*: *min s P* = *Some* x' $x'{\in}\alpha$ *s* *P* x'
   by *auto*
  show *thesis*
   apply (*rule that*[*OF KV*])
   apply (*clarify*)
   apply (*drule* (*1*) *min-correct*(*2*)[*OF* ‹*invar s*›])
   apply (*simp add*: *KV*(*1*))
   done
 qed

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

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

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

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

 lemma min-None:
  ⟦ invar s; min s P = None ⟧ ⟹ {x∈α s. P x} = {}
  apply (cases {x∈α s. P x} = {})
  apply simp
  apply simp
  apply (erule exE)
  apply clarify
  apply (drule (2) min-correct(1)[where P=P])
  apply auto
  done

end

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

 lemmas maxI = max-correct(3)

 lemma max-Some:
  ⟦ invar s; max s P = Some x ⟧ ⟹ x∈α s
  ⟦ invar s; max s P = Some x ⟧ ⟹ P x
  ⟦ invar s; max s P = Some x; x'∈α s; P x'⟧ ⟹ x≥x'
```

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

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

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

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

end

Conversion to List

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

locale *set-to-rev-list* = *ordered-set* +
constrains α :: *'s* ⇒ *'x*::*linorder set*
fixes *to-rev-list* :: *'s* ⇒ *'x list*

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

Record Based Interface

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

locale *StdSetDefs* =
poly-set-iteratei-defs set-op-list-it ops
for *ops* :: ('*x*,'*s*,'*more*) *set-ops-scheme*
begin
abbreviation α **where** α == *set-op-α ops*
abbreviation *invar* **where** *invar* == *set-op-invar ops*
abbreviation *empty* **where** *empty* == *set-op-empty ops*
abbreviation *memb* **where** *memb* == *set-op-memb ops*
abbreviation *ins* **where** *ins* == *set-op-ins ops*
abbreviation *ins-dj* **where** *ins-dj* == *set-op-ins-dj ops*
abbreviation *delete* **where** *delete* == *set-op-delete ops*
abbreviation *list-it* **where** *list-it* ≡ *set-op-list-it ops*
abbreviation *sng* **where** *sng* == *set-op-sng ops*

abbreviation *isEmpty* **where** *isEmpty* == *set-op-isEmpty ops*
abbreviation *isSng* **where** *isSng* == *set-op-isSng ops*
abbreviation *ball* **where** *ball* == *set-op-ball ops*
abbreviation *bex* **where** *bex* == *set-op-bex ops*
abbreviation *size* **where** *size* == *set-op-size ops*
abbreviation *size-abort* **where** *size-abort* == *set-op-size-abort ops*
abbreviation *union* **where** *union* == *set-op-union ops*
abbreviation *union-dj* **where** *union-dj* == *set-op-union-dj ops*
abbreviation *diff* **where** *diff* == *set-op-diff ops*
abbreviation *filter* **where** *filter* == *set-op-filter ops*
abbreviation *inter* **where** *inter* == *set-op-inter ops*
abbreviation *subset* **where** *subset* == *set-op-subset ops*
abbreviation *equal* **where** *equal* == *set-op-equal ops*
abbreviation *disjoint* **where** *disjoint* == *set-op-disjoint ops*
abbreviation *disjoint-witness*
where *disjoint-witness* == *set-op-disjoint-witness ops*
abbreviation *sel* **where** *sel* == *set-op-sel ops*
abbreviation *to-list* **where** *to-list* == *set-op-to-list ops*
abbreviation *from-list* **where** *from-list* == *set-op-from-list ops*
end

locale *StdSet* = *StdSetDefs ops* +
set α *invar* +
set-empty α *invar empty* +
set-memb α *invar memb* +
set-ins α *invar ins* +
set-ins-dj α *invar ins-dj* +
set-delete α *invar delete* +
poly-set-iteratei α *invar list-it* +
set-sng α *invar sng* +
set-isEmpty α *invar isEmpty* +
set-isSng α *invar isSng* +
set-ball α *invar ball* +
set-bex α *invar bex* +
set-size α *invar size* +
set-size-abort α *invar size-abort* +
set-union α *invar* α *invar* α *invar union* +
set-union-dj α *invar* α *invar* α *invar union-dj* +
set-diff α *invar* α *invar diff* +
set-filter α *invar* α *invar filter* +
set-inter α *invar* α *invar* α *invar inter* +
set-subset α *invar* α *invar subset* +
set-equal α *invar* α *invar equal* +
set-disjoint α *invar* α *invar disjoint* +
set-disjoint-witness α *invar* α *invar disjoint-witness* +
set-sel' α *invar sel* +
set-to-list α *invar to-list* +
list-to-set α *invar from-list*
for *ops* :: *('x,'s,'more) set-ops-scheme*

begin

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

end

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

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

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

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

abbreviation *max* **where** *max* == *set-op-max ops*
abbreviation *to-sorted-list* **where**
to-sorted-list ≡ *set-op-to-sorted-list ops*
abbreviation *to-rev-list* **where** *to-rev-list* ≡ *set-op-to-rev-list ops*
end

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

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

no-notation *insert* (‹*set'-ins*›)

end

3.1.2 Specification of Sequences

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

Definition

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

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

Functions

locale *list-empty* = *list* +
constrains α :: '*s* ⇒ '*x list*
fixes *empty* :: *unit* ⇒ '*s*
assumes *empty-correct*:

α (*empty* ()) = []
invar (*empty* ())

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

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

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

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

locale *list-appendl* = *list* +
constrains α :: $'s \Rightarrow 'x$ *list*
fixes *appendl* :: $'x \Rightarrow 's \Rightarrow 's$
assumes *appendl-correct*:
invar $s \Longrightarrow \alpha$ (*appendl* $x\ s$) = $x\#\alpha\ s$
invar $s \Longrightarrow$ *invar* (*appendl* $x\ s$)
begin
abbreviation (*input*) *push* $\equiv$ *appendl*
lemmas *push-correct* = *appendl-correct*
end

locale *list-removel* = *list* +
constrains α :: $'s \Rightarrow 'x$ *list*

```
  fixes removel :: 's ⇒ ('x × 's)
  assumes removel-correct:
    ⟦invar s; α s ≠ []⟧ ⟹ fst (removel s) = hd (α s)
    ⟦invar s; α s ≠ []⟧ ⟹ α (snd (removel s)) = tl (α s)
    ⟦invar s; α s ≠ []⟧ ⟹ invar (snd (removel s))
begin
  lemma removelE:
    assumes I: invar s    α s ≠ []
    obtains s' where removel s = (hd (α s), s')    invar s'    α s' = tl (α s)
  proof −
    from removel-correct(1,2,3)[OF I] have
      C: fst (removel s) = hd (α s)
         α (snd (removel s)) = tl (α s)
         invar (snd (removel s)) .
    from that[of snd (removel s), OF - C(3,2), folded C(1)] show thesis
      by simp
  qed
```

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

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

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

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

locale list-appendr = list +
  constrains α :: 's ⇒ 'x list
  fixes appendr :: 'x ⇒ 's ⇒ 's
  assumes appendr-correct:
    invar s ⟹ α (appendr x s) = α s @ [x]
    invar s ⟹ invar (appendr x s)
begin
  abbreviation (input) enqueue ≡ appendr
  lemmas enqueue-correct = appendr-correct
end

locale list-remover = list +
```

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

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

Indexing

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

locale *list-set* = *list* +
constrains α :: *'s* $\Rightarrow$ *'x list*
fixes *set* :: *'s* $\Rightarrow$ *nat* $\Rightarrow$ *'x* $\Rightarrow$ *'s*
assumes *set-correct*:
$[\![invar\ s;\ i{<}length\ (\alpha\ s)]\!] \Longrightarrow \alpha\ (set\ s\ i\ x) = (\alpha\ s)\ [i := x]$
$[\![invar\ s;\ i{<}length\ (\alpha\ s)]\!] \Longrightarrow invar\ (set\ s\ i\ x)$

record (*'a*,*'s*) *list-ops* =
list-op-α :: *'s* $\Rightarrow$ *'a list*
list-op-invar :: *'s* $\Rightarrow$ *bool*
list-op-empty :: *unit* $\Rightarrow$ *'s*
list-op-isEmpty :: *'s* $\Rightarrow$ *bool*
list-op-size :: *'s* $\Rightarrow$ *nat*
list-op-appendl :: *'a* $\Rightarrow$ *'s* $\Rightarrow$ *'s*
list-op-removel :: *'s* $\Rightarrow$ *'a* $\times$ *'s*
list-op-leftmost :: *'s* $\Rightarrow$ *'a*
list-op-appendr :: *'a* $\Rightarrow$ *'s* $\Rightarrow$ *'s*
list-op-remover :: *'s* $\Rightarrow$ *'s* $\times$ *'a*
list-op-rightmost :: *'s* $\Rightarrow$ *'a*
list-op-get :: *'s* $\Rightarrow$ *nat* $\Rightarrow$ *'a*
list-op-set :: *'s* $\Rightarrow$ *nat* $\Rightarrow$ *'a* $\Rightarrow$ *'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** $\alpha \equiv$ *list-op-α ops*
  abbreviation *invar* **where** *invar* $\equiv$ *list-op-invar ops*
  abbreviation *empty* **where** *empty* $\equiv$ *list-op-empty ops*
  abbreviation *isEmpty* **where** *isEmpty* $\equiv$ *list-op-isEmpty ops*
  abbreviation *size* **where** *size* $\equiv$ *list-op-size ops*
  abbreviation *appendl* **where** *appendl* $\equiv$ *list-op-appendl ops*
  abbreviation *removel* **where** *removel* $\equiv$ *list-op-removel ops*
  abbreviation *leftmost* **where** *leftmost* $\equiv$ *list-op-leftmost ops*
  abbreviation *appendr* **where** *appendr* $\equiv$ *list-op-appendr ops*
  abbreviation *remover* **where** *remover* $\equiv$ *list-op-remover ops*
  abbreviation *rightmost* **where** *rightmost* $\equiv$ *list-op-rightmost ops*
  abbreviation *get* **where** *get* $\equiv$ *list-op-get ops*
  abbreviation *set* **where** *set* $\equiv$ *list-op-set ops*
end

locale *StdList* = *StdListDefs ops*
  + *list* α *invar*
  + *list-empty* α *invar empty*
  + *list-isEmpty* α *invar isEmpty*
  + *list-size* α *invar size*
  + *list-appendl* α *invar appendl*
  + *list-removel* α *invar removel*
  + *list-leftmost* α *invar leftmost*
  + *list-appendr* α *invar appendr*
  + *list-remover* α *invar remover*
  + *list-rightmost* α *invar rightmost*
  + *list-get* α *invar get*
  + *list-set* α *invar set*
  for *ops* :: ('a,'s,'more) *list-ops-scheme*
begin
  lemmas *correct* =
    empty-correct
    isEmpty-correct
    size-correct
    appendl-correct
    removel-correct
    leftmost-correct
    appendr-correct
    remover-correct
    rightmost-correct
    get-correct
    set-correct

end

locale *StdList-no-invar* = *StdList* + *list-no-invar* α *invar*

end

3.1.3 Specification of Annotated Lists

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

Introduction

We define lists with annotated elements. The annotations form a monoid. We provide standard list operations and the split-operation, that splits the list according to its annotations.

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

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

Basic Annotated List Operations

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

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

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

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

Appending an Element from the Right **locale** *al-consr* = *al* +
constrains $\alpha :: {'s} \Rightarrow ({'e} \times {'a}{::}monoid\text{-}add)\ list$
fixes $consr :: {'s} \Rightarrow {'e} \Rightarrow {'a} \Rightarrow {'s}$
assumes *consr-correct*:
$invar\ s \Longrightarrow invar\ (consr\ s\ e\ a)$
$invar\ s \Longrightarrow (\alpha\ (consr\ s\ e\ a)) = (\alpha\ s)\ @\ [(e,a)]$

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

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

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

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

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

Fold a Function over the Elements from the Right **locale** *al-foldr* =
al +
constrains $\alpha :: {'s} \Rightarrow ({'e} \times {'a}{::}monoid\text{-}add)\ list$
fixes $foldr :: ({'e} \times {'a} \Rightarrow {'z} \Rightarrow {'z}) \Rightarrow {'s} \Rightarrow {'z} \Rightarrow {'z}$
assumes *foldr-correct*:
$invar\ s \Longrightarrow foldr\ f\ s\ \sigma = List.foldr\ f\ (\alpha\ s)\ \sigma$

locale *poly-al-fold* = *al* +

constrains α :: $'s \Rightarrow ('e \times 'a$::*monoid-add*) *list*
begin
definition *foldl* **where**
foldl-correct[*code-unfold*]: $foldl\ f\ \sigma\ s = List.foldl\ f\ \sigma\ (\alpha\ s)$
definition *foldr* **where**
foldr-correct[*code-unfold*]: $foldr\ f\ s\ \sigma = List.foldr\ f\ (\alpha\ s)\ \sigma$
end

Concatenation of Two Annotated Lists **locale** *al-app* = *al* +
constrains α :: $'s \Rightarrow ('e \times 'a$::*monoid-add*) *list*
fixes *app* :: $'s \Rightarrow 's \Rightarrow 's$
assumes *app-correct*:
$[\![invar\ s;invar\ s']\!] \Longrightarrow \alpha\ (app\ s\ s') = (\alpha\ s)\ @\ (\alpha\ s')$
$[\![invar\ s;invar\ s']\!] \Longrightarrow invar\ (app\ s\ s')$

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

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

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

$p\ (i + \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{snd}\ (\alpha\ l)) + a)$
invar l
invar r
using *assms*
apply (*cases splits p i s*)
apply (*case-tac b*)
apply (*drule-tac* $i = i$ **and** $p = p$
and $l = a$ **and** $r = c$ **and** $e = aa$ **and** $a = ba$ **in** *splits-correct*)
apply (*simp-all*)
done
end

Record Based Interface

record ($'e$,$'a$,$'s$) *alist-ops* =
alist-op-α ::$'s \Rightarrow$ ($'e \times 'a$::*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 :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a \Rightarrow 's \Rightarrow ('s \times ('e \times 'a) \times 's)$

locale *StdALDefs* = *poly-al-fold alist-op-*α *ops* *alist-op-invar ops*
for *ops* :: ($'e$,$'a$::*monoid-add*,$'s$,$'more$) *alist-ops-scheme*
begin
abbreviation α **where** α == *alist-op-*α *ops*
abbreviation *invar* **where** *invar* == *alist-op-invar ops*
abbreviation *empty* **where** *empty* == *alist-op-empty ops*
abbreviation *isEmpty* **where** *isEmpty* == *alist-op-isEmpty ops*
abbreviation *count* **where** *count* == *alist-op-count ops*
abbreviation *consl* **where** *consl* == *alist-op-consl ops*
abbreviation *consr* **where** *consr* == *alist-op-consr ops*
abbreviation *head* **where** *head* == *alist-op-head ops*
abbreviation *tail* **where** *tail* == *alist-op-tail ops*
abbreviation *headR* **where** *headR* == *alist-op-headR ops*
abbreviation *tailR* **where** *tailR* == *alist-op-tailR ops*
abbreviation *app* **where** *app* == *alist-op-app ops*
abbreviation *annot* **where** *annot* == *alist-op-annot ops*
abbreviation *splits* **where** *splits* == *alist-op-splits ops*
end

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

locale *StdAL-no-invar* = *StdAL* + *al-no-invar* α *invar*

end

3.1.4 Specification of Priority Queues

theory *PrioSpec*
imports *ICF-Spec-Base HOL−Library.Multiset*
begin

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

locale *prio* =
fixes $\alpha :: {'p} \Rightarrow ({'e} \times {'a}{::}linorder)$ *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*: $[\![invar\ p;\ \alpha\ p \neq \{\#\}]\!] \Longrightarrow$
  $(find\ p) \in\#\ (\alpha\ p) \wedge (\forall\, y \in set\text{-}mset\ (\alpha\ p).\ snd\ (find\ p) \leq snd\ y)$

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

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

Delete Minimal Element Delete the same element that find will return

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

Record based interface

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

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

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

locale *StdPrio-no-invar* = *StdPrio* + *prio-no-invar α invar*

end

3.1.5 Specification of Unique Priority Queues

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

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

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

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

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

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

Basic Upriority Queue Functions

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

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

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

  lemma *popE*:

assumes
invar s
$\alpha\ s \neq Map.empty$
obtains $e\ a\ s'$ **where**
$pop\ s = (e,\ a,\ s')$
$invar\ s'$
$\alpha\ s' = (\alpha\ s)(e := None)$
$(\alpha\ s)\ e = Some\ a$
$(\forall\, y \in ran\ (\alpha\ s).\ a \leq y)$
using *assms*
apply (*cases pop s*)
apply (*drule* (*2*) *pop-correct*)
apply *blast*
done

end

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

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

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

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

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

Record Based Interface

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

upr-isEmpty :: $'s \Rightarrow$ *bool*
upr-insert :: $'s \Rightarrow 'e \Rightarrow 'a \Rightarrow 's$
upr-pop :: $'s \Rightarrow ('e \times 'a \times 's)$
upr-prio :: $'s \Rightarrow 'e \Rightarrow 'a$ *option*

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

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

locale *StdUprio-no-invar* = *StdUprio* + *uprio-no-invar* α *invar*

end

3.2 Generic Algorithms

3.2.1 General Algorithms for Iterators over Finite Sets

theory *SetIteratorCollectionsGA*
imports
../spec/SetSpec
../spec/MapSpec
begin

Iterate add to Set

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

lemma *iterate-add-to-set-correct* :
assumes *ins-OK*: *set-ins* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
shows α (*iterate-add-to-set s ins it*) = *S0* ∪ α *s* ∧ *invar* (*iterate-add-to-set s ins it*)
unfolding *iterate-add-to-set-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
where *?I*=λ*S* σ. α σ = *S* ∪ α *s* ∧ *invar* σ])
apply (*insert ins-OK s-OK*)
apply (*simp-all add*: *set-ins-def*)
done

lemma *iterate-add-to-set-dj-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins-dj*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: *S0* ∩ α *s* = {}
shows α (*iterate-add-to-set s ins-dj it*) = *S0* ∪ α *s* ∧ *invar* (*iterate-add-to-set s ins-dj it*)
unfolding *iterate-add-to-set-def*
apply (*rule set-iterator-no-cond-rule-insert-P* [*OF it*,
where *?I*=λ*S* σ. α σ = *S* ∪ α *s* ∧ *invar* σ])
apply (*insert ins-dj-OK s-OK dj*)
apply (*simp-all add*: *set-ins-dj-def set-eq-iff*)
done

Iterator to Set

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

lemma *iterate-to-set-alt-def*[*code*] :
iterate-to-set emp ins-dj (*it*::(′*x*,′*x-set*) *set-iterator*) =
it (λ-. *True*) (λ*x* σ. *ins-dj x* σ) (*emp* ())
unfolding *iterate-to-set-def iterate-add-to-set-def* **by** *simp*

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

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

Iterate image/filter add to Set

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

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

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

For completeness lets also consider injective versions.

lemma *iterate-inj-image-filter-add-to-set-correct* :
assumes *ins-dj-OK*: *set-ins-dj* α *invar ins*
assumes *s-OK*: *invar s*
assumes *it*: *set-iterator it S0*
assumes *dj*: $\{y.\ \exists\, x.\ x \in S0\ \wedge f\ x = Some\ y\} \cap \alpha\ s = \{\}$
assumes *f-inj-on*: *inj-on f* (*S0* ∩ *dom f*)
shows α (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*)) =
  $\{b\ .\ \exists\, a.\ a \in S0\ \wedge f\ a = Some\ b\} \cup \alpha\ s\ \wedge$
  invar (*iterate-add-to-set s ins* (*set-iterator-image-filter f it*))
proof −
  from *set-iterator-image-filter-correct* [*OF it f-inj-on*]
  have *it-f*: *set-iterator* (*set-iterator-image-filter f it*)

$\{y.\ \exists x.\ x \in S0 \wedge f\ x = Some\ y\}$ **by** *simp*

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

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

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

Iterate add to Map

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

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

using *assms*
unfolding *iterate-add-to-map-def*
apply (*rule-tac map-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=$\lambda d\ \sigma.\ (\alpha\ \sigma = \alpha\ m\ {+}{+}\ M\ |`\ d) \wedge invar\ \sigma$])
apply (*simp-all add*: *map-update-def restrict-map-insert*)
done

lemma *iterate-add-to-map-dj-correct* :
assumes *upd-OK*: *map-update-dj* α *invar upd*
assumes *m-OK*: *invar m*
assumes *it*: *map-iterator it M*
assumes *dj*: *dom M* $\cap$ *dom* (α *m*) = {}
shows α (*iterate-add-to-map m upd it*) = α *m* ++ *M* $\wedge$ *invar* (*iterate-add-to-map m upd it*)
using *assms*
unfolding *iterate-add-to-map-def*
apply (*rule-tac map-iterator-no-cond-rule-insert-P* [*OF it*,
    where *?I*=$\lambda d\ \sigma.\ (\alpha\ \sigma = \alpha\ m\ {+}{+}\ M\ |`\ d) \wedge invar\ \sigma$])
apply (*simp-all add*: *map-update-dj-def restrict-map-insert set-eq-iff*)
done

Iterator to Map

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

lemma *iterate-to-map-alt-def*[*code*] :
  iterate-to-map emp upd-dj it =
   it (λ-. *True*) (λ(*k*, *v*) σ. *upd-dj k v* σ) (*emp* ())
unfolding *iterate-to-map-def iterate-add-to-map-def* **by** *simp*

lemma *iterate-to-map-correct* :
assumes *upd-dj-OK*: *map-update-dj* α *invar upd-dj*
assumes *emp-OK*: *map-empty* α *invar emp*
assumes *it*: *map-iterator it M*
shows α (*iterate-to-map emp upd-dj it*) = *M* $\wedge$ *invar* (*iterate-to-map emp upd-dj it*)
unfolding *iterate-to-map-def*
using *iterate-add-to-map-dj-correct* [*OF upd-dj-OK - it*, *of emp* ()] *emp-OK*
by (*simp add*: *map-empty-def*)

end

3.2.2 Generic Algorithms for Maps

theory *MapGA*
imports *SetIteratorCollectionsGA*
begin record ('*k*,'*v*,'*s*) *map-basic-ops* =

bmap-op-α :: ('*k*,'*v*,'*s*) *map-α*
bmap-op-invar :: ('*k*,'*v*,'*s*) *map-invar*
bmap-op-empty :: ('*k*,'*v*,'*s*) *map-empty*
bmap-op-lookup :: ('*k*,'*v*,'*s*) *map-lookup*
bmap-op-update :: ('*k*,'*v*,'*s*) *map-update*
bmap-op-update-dj :: ('*k*,'*v*,'*s*) *map-update-dj*
bmap-op-delete :: ('*k*,'*v*,'*s*) *map-delete*
bmap-op-list-it :: ('*k*,'*v*,'*s*) *map-list-it*

record ('*k*,'*v*,'*s*) *omap-basic-ops* = ('*k*,'*v*,'*s*) *map-basic-ops* +
bmap-op-ordered-list-it :: '*s* ⇒ ('*k*,'*v*,('*k*×'*v*) *list*) *map-iterator*
bmap-op-rev-list-it :: '*s* ⇒ ('*k*,'*v*,('*k*×'*v*) *list*) *map-iterator*

locale *StdBasicMapDefs* =
poly-map-iteratei-defs bmap-op-list-it ops
for *ops* :: ('*k*,'*v*,'*s*,'*more*) *map-basic-ops-scheme*
begin
abbreviation *α* **where** *α* == *bmap-op-α ops*
abbreviation *invar* **where** *invar* == *bmap-op-invar ops*
abbreviation *empty* **where** *empty* == *bmap-op-empty ops*
abbreviation *lookup* **where** *lookup* == *bmap-op-lookup ops*
abbreviation *update* **where** *update* == *bmap-op-update ops*
abbreviation *update-dj* **where** *update-dj* == *bmap-op-update-dj ops*
abbreviation *delete* **where** *delete* == *bmap-op-delete ops*
abbreviation *list-it* **where** *list-it* == *bmap-op-list-it ops*
end

locale *StdBasicOMapDefs* = *StdBasicMapDefs ops*
+ *poly-map-iterateoi-defs bmap-op-ordered-list-it ops*
+ *poly-map-rev-iterateoi-defs bmap-op-rev-list-it ops*
for *ops* :: ('*k*::*linorder*,'*v*,'*s*,'*more*) *omap-basic-ops-scheme*
begin
abbreviation *ordered-list-it* **where** *ordered-list-it*
≡ *bmap-op-ordered-list-it ops*
abbreviation *rev-list-it* **where** *rev-list-it*
≡ *bmap-op-rev-list-it ops*
end

locale *StdBasicMap* = *StdBasicMapDefs ops* +
map α invar +
map-empty α invar empty +
map-lookup α invar lookup +
map-update α invar update +
map-update-dj α invar update-dj +
map-delete α invar delete +
poly-map-iteratei α invar list-it
for *ops* :: ('*k*,'*v*,'*s*,'*more*) *map-basic-ops-scheme*
begin
lemmas *correct*[*simp*] = *empty-correct lookup-correct update-correct*

update-dj-correct delete-correct
end

locale *StdBasicOMap* =
StdBasicOMapDefs ops +
StdBasicMap ops +
poly-map-iterateoi α *invar ordered-list-it* +
poly-map-rev-iterateoi α *invar rev-list-it*
for *ops* :: ('*k*::*linorder*,'*v*,'*s*,'*more*) *omap-basic-ops-scheme*
begin
end

context *StdBasicMapDefs* **begin**
definition *g-sng k v* ≡ *update k v* (*empty* ())
definition *g-add m1 m2* ≡ *iterate m2* (λ(*k*,*v*) σ. *update k v* σ) *m1*

definition
g-sel m P ≡
iteratei m ($\lambda\sigma$. σ = *None*) (λx σ. *if P x then Some x else None*) *None*

definition *g-bex m P* ≡ *iteratei m* (λx. ¬*x*) (λkv σ. *P kv*) *False*
definition *g-ball m P* ≡ *iteratei m id* (λkv σ. *P kv*) *True*

definition *g-size m* ≡ *iterate m* (λ-. *Suc*) (*0*::*nat*)
definition *g-size-abort b m* ≡ *iteratei m* (λs. $s<b$) (λ-. *Suc*) (*0*::*nat*)

definition *g-isEmpty m* ≡ *g-size-abort 1 m* = *0*
definition *g-isSng m* ≡ *g-size-abort 2 m* = *1*

definition *g-to-list m* ≡ *iterate m* (#) []

definition *g-list-to-map l* ≡ *foldl* (λm (*k*,*v*). *update k v m*) (*empty* ())
(*rev l*)

definition *g-add-dj m1 m2* ≡ *iterate m2* (λ(*k*,*v*) σ. *update-dj k v* σ) *m1*

definition *g-restrict P m* ≡ *iterate m*
(λ(*k*,*v*) σ. *if P* (*k*,*v*) *then update-dj k v* σ *else* σ) (*empty* ())

definition *dflt-ops* :: ('*k*,'*v*,'*s*) *map-ops*
where [*icf-rec-def*]:
dflt-ops ≡
(|
map-op-α = α,
map-op-invar = *invar*,
map-op-empty = *empty*,
map-op-lookup = *lookup*,
map-op-update = *update*,

```
    map-op-update-dj = update-dj,
    map-op-delete = delete,
    map-op-list-it = list-it,
    map-op-sng = g-sng,
    map-op-restrict = g-restrict,
    map-op-add = g-add,
    map-op-add-dj = g-add-dj,
    map-op-isEmpty = g-isEmpty,
    map-op-isSng = g-isSng,
    map-op-ball = g-ball,
    map-op-bex = g-bex,
    map-op-size = g-size,
    map-op-size-abort = g-size-abort,
    map-op-sel = g-sel,
    map-op-to-list = g-to-list,
    map-op-to-map = g-list-to-map
  |)

  local-setup ‹Locale-Code.lc-decl-del @{term dflt-ops}›

end

lemma update-dj-by-update:
  assumes map-update α invar update
  shows map-update-dj α invar update
proof −
  interpret map-update α invar update by fact
  show ?thesis
    apply (unfold-locales)
    apply (auto simp add: update-correct)
    done
qed

lemma map-iterator-linord-is-it:
  map-iterator-linord m it ⟹ map-iterator m it
  unfolding set-iterator-def set-iterator-map-linord-def
  apply (erule set-iterator-genord.set-iterator-weaken-R)
  ..

lemma map-rev-iterator-linord-is-it:
  map-iterator-rev-linord m it ⟹ map-iterator m it
  unfolding set-iterator-def set-iterator-map-rev-linord-def
  apply (erule set-iterator-genord.set-iterator-weaken-R)
  ..

context StdBasicMap
begin
  lemma g-sng-impl: map-sng α invar g-sng
    apply unfold-locales
```

```
  apply (simp-all add: update-correct empty-correct g-sng-def)
  done

lemma g-add-impl: map-add α invar g-add
proof
  fix m1 m2
  assume invar m1    invar m2

  have A: g-add m1 m2 = iterate-add-to-map m1 update (iteratei m2)
    unfolding g-add-def iterate-add-to-map-def by simp
  have α (g-add m1 m2) = α m1 ++ α m2 ∧ invar (g-add m1 m2)
    unfolding A
    apply (rule
      iterate-add-to-map-correct[of α invar update m1 iteratei m2    α m2])
    apply unfold-locales []
    apply fact
    apply (rule iteratei-correct, fact)
    done
  thus α (g-add m1 m2) = α m1 ++ α m2    invar (g-add m1 m2) by auto
qed

lemma g-sel-impl: map-sel′ α invar g-sel
proof −
  have A: ⋀m P. g-sel m P = iterate-sel-no-map (iteratei m) P
    unfolding g-sel-def iterate-sel-no-map-def iterate-sel-def by simp

  { fix m P
    assume I: invar m
    note iterate-sel-no-map-correct[OF iteratei-correct[OF I], of P]
  }
  thus ?thesis
    apply unfold-locales
    unfolding A
    apply (simp add: Bex-def Ball-def image-iff map-to-set-def)
    apply clarify
    apply (metis option.exhaust prod.exhaust)
    apply (simp add: Bex-def Ball-def image-iff map-to-set-def)
    done
qed

lemma g-bex-impl: map-bex α invar g-bex
  apply unfold-locales
  unfolding g-bex-def
  apply (rule-tac I=λit σ. σ ⟷ (∃kv∈it. P kv)
    in iteratei-rule-insert-P)
  by (auto simp: map-to-set-def)

lemma g-ball-impl: map-ball α invar g-ball
  apply unfold-locales
```

```
  unfolding g-ball-def
  apply (rule-tac I=λit σ. σ ⟷ (∀ kv∈it. P kv)
    in iteratei-rule-insert-P)
  apply (auto simp: map-to-set-def)
  done

lemma g-size-impl: map-size α invar g-size
proof
  fix m
  assume I: invar m
  have A: g-size m ≡ iterate-size (iteratei m)
    unfolding g-size-def iterate-size-def by simp

  from iterate-size-correct [OF iteratei-correct[OF I]]
  show g-size m = card (dom (α m))
    unfolding A
    by (simp-all add: card-map-to-set)
qed

lemma g-size-abort-impl: map-size-abort α invar g-size-abort
proof
  fix s m
  assume I: invar m
  have A: g-size-abort s m ≡ iterate-size-abort (iteratei m) s
    unfolding g-size-abort-def iterate-size-abort-def by simp

  from iterate-size-abort-correct [OF iteratei-correct[OF I]]
  show g-size-abort s m = min s (card (dom (α m)))
    unfolding A
    by (simp-all add: card-map-to-set)
qed

lemma g-isEmpty-impl: map-isEmpty α invar g-isEmpty
proof
  fix m
  assume I: invar m
  interpret map-size-abort α invar g-size-abort by (rule g-size-abort-impl)
  from size-abort-correct[OF I] have
    g-size-abort 1 m = min 1 (card (dom (α m))) .
  thus g-isEmpty m = (α m = Map.empty) unfolding g-isEmpty-def
    by (auto simp: min-def card-0-eq[OF finite] I)
qed

lemma g-isSng-impl: map-isSng α invar g-isSng
proof
  fix m
  assume I: invar m
  interpret map-size-abort α invar g-size-abort by (rule g-size-abort-impl)
  from size-abort-correct[OF I] have
```

```
    g-size-abort 2 m = min 2 (card (dom (α m))) .
  thus g-isSng m = (∃ k v. α m = [k ↦ v]) unfolding g-isSng-def
    by (auto simp: min-def I card-Suc-eq dom-eq-singleton-conv)
qed

lemma g-to-list-impl: map-to-list α invar g-to-list
proof
  fix m
  assume I: invar m

  have A: g-to-list m = iterate-to-list (iteratei m)
    unfolding g-to-list-def iterate-to-list-def by simp

  from iterate-to-list-correct [OF iteratei-correct[OF I]]
  have set-l-eq: set (g-to-list m) = map-to-set (α m) and
    dist-l: distinct (g-to-list m) unfolding A by simp-all

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

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

lemma g-list-to-map-impl: list-to-map α invar g-list-to-map
proof −
  {
    fix m0 l
    assume invar m0
    hence invar (foldl (λs (k,v). update k v s) m0 l) ∧
      α (foldl (λs (k,v). update k v s) m0 l) = α m0 ++ map-of (rev l)
    proof (induction l arbitrary: m0)
      case Nil thus ?case by simp
    next
      case (Cons kv l)
      obtain k v where [simp]: kv=(k,v) by (cases kv) auto
      have invar (foldl (λs (k, v). update k v s) m0 (kv # l))
        apply simp
        apply (rule conjunct1[OF Cons.IH])
        apply (simp add: update-correct Cons.prems)
        done
      moreover have α (foldl (λs (k, v). update k v s) m0 (kv # l)) =
        α m0 ++ map-of (rev (kv # l))
        apply simp
        apply (rule trans[OF conjunct2[OF Cons.IH]])
        apply (auto
          simp: update-correct Cons.prems Map.map-add-def[abs-def]
          split: option.split
        )
```

done
ultimately show *?case*
by *simp*
qed
} **thus** *?thesis*
apply *unfold-locales*
unfolding *g-list-to-map-def*
apply (*auto simp*: *empty-correct*)
done
qed

lemma *g-add-dj-impl*: *map-add-dj* α *invar g-add-dj*
proof
fix *m1 m2*
assume *invar m1*   *invar m2* **and** *DJ*: *dom* (α *m1*) $\cap$ *dom* (α *m2*) = {}

have *A*: *g-add-dj m1 m2* = *iterate-add-to-map m1 update-dj* (*iteratei m2*)
unfolding *g-add-dj-def iterate-add-to-map-def* **by** *simp*
have α (*g-add-dj m1 m2*) = α *m1* ++ α *m2* $\wedge$ *invar* (*g-add-dj m1 m2*)
unfolding *A*
apply (*rule*
iterate-add-to-map-dj-correct[
of α *invar update-dj m1 iteratei m2* α *m2*])
apply *unfold-locales* []
apply *fact*
apply (*rule iteratei-correct*, *fact*)
using *DJ* **apply** (*simp add*: *Int-ac*)
done
thus α (*g-add-dj m1 m2*) = α *m1* ++ α *m2*   *invar* (*g-add-dj m1 m2*) **by** *auto*
qed

lemma *g-restrict-impl*: *map-restrict* α *invar* α *invar g-restrict*
proof
fix *m P*
assume *I*: *invar m*

have *AUX*: $\bigwedge k\ v\ it\ \sigma.$
$[\![it \subseteq \{(k,\ v).\ \alpha\ m\ k = Some\ v\};\ \alpha\ m\ k = Some\ v;\ (k,\ v) \notin it;$
$\{(k,\ v).\ \alpha\ \sigma\ k = Some\ v\} = it \cap Collect\ P]\!]$
$\Longrightarrow k \notin dom\ (\alpha\ \sigma)$
proof (*rule ccontr*, *simp*)
fix *k v it* σ
assume $k \in dom\ (\alpha\ \sigma)$
then obtain v' **where** $\alpha\ \sigma\ k = Some\ v'$ **by** *auto*
moreover assume $\{(k,\ v).\ \alpha\ \sigma\ k = Some\ v\} = it \cap Collect\ P$
ultimately have *MEM*: $(k,v') \in it$ **by** *auto*
moreover assume $it \subseteq \{(k,\ v).\ \alpha\ m\ k = Some\ v\}$ **and** $\alpha\ m\ k = Some\ v$
ultimately have $v' = v$ **by** *auto*

```
      moreover assume (k,v)∉it
      moreover note MEM
      ultimately show False by simp
    qed

    have α (g-restrict P m) = α m |` {k. ∃v. α m k = Some v ∧ P (k, v)} ∧
      invar (g-restrict P m)
      unfolding g-restrict-def
      apply (rule-tac I=λit σ. invar σ
        ∧ map-to-set (α σ) = it ∩ Collect P
        in iterate-rule-insert-P)
      apply (auto simp: I empty-correct update-dj-correct map-to-set-def AUX)
      apply (auto split: if-split-asm)
      apply (rule ext)
      apply (auto simp: Map.restrict-map-def)
      apply force
      apply (rule ccontr)
      apply force
      done
    thus α (g-restrict P m) = α m |` {k. ∃v. α m k = Some v ∧ P (k, v)}
      invar (g-restrict P m) by auto
  qed

  lemma dflt-ops-impl: StdMap dflt-ops
    apply (rule StdMap-intro)
    apply icf-locales
    apply (simp-all add: icf-rec-unf)
    apply (rule g-sng-impl g-restrict-impl g-add-impl g-add-dj-impl
      g-isEmpty-impl g-isSng-impl g-ball-impl g-bex-impl g-size-impl
      g-size-abort-impl g-sel-impl g-to-list-impl g-list-to-map-impl)+
    done
end


context StdBasicOMapDefs
begin
  definition
    g-min m P ≡
      iterateoi m (λσ. σ = None) (λx σ. if P x then Some x else None) None

  definition
    g-max m P ≡
      rev-iterateoi m (λσ. σ = None) (λx σ. if P x then Some x else None) None

  definition g-to-sorted-list m ≡ rev-iterateo m (#) []
  definition g-to-rev-list m ≡ iterateo m (#) []

  definition dflt-oops :: ('k,'v,'s) omap-ops
    where [icf-rec-def]:
```

dflt-oops ≡ *map-ops.extend dflt-ops*
⦇
map-op-ordered-list-it = *ordered-list-it*,
map-op-rev-list-it = *rev-list-it*,
map-op-min = *g-min*,
map-op-max = *g-max*,
map-op-to-sorted-list = *g-to-sorted-list*,
map-op-to-rev-list = *g-to-rev-list*
⦈
local-setup ‹*Locale-Code.lc-decl-del* @{*term dflt-oops*}›

end

context *StdBasicOMap*
begin
lemma *g-min-impl*: *map-min* α *invar g-min*
proof
fix *m P*

assume *I*: *invar m*

from *iterateoi-correct*[*OF I*]
have *iti′*: *map-iterator-linord* (*iterateoi m*) (α *m*) **by** *simp*
note *sel-correct* = *iterate-sel-no-map-map-linord-correct*[*OF iti′*, *of P*]

have *A*: *g-min m P* = *iterate-sel-no-map* (*iterateoi m*) *P*
unfolding *g-min-def iterate-sel-no-map-def iterate-sel-def* **by** *simp*

{ assume *rel-of* (α *m*) *P* ≠ {}
with *sel-correct*
show *g-min m P* ∈ *Some* ‘ *rel-of* (α *m*) *P*
unfolding *A*
by (*auto simp add*: *image-iff rel-of-def*)
}

{ assume *rel-of* (α *m*) *P* = {}
with *sel-correct* **show** *g-min m P* = *None*
unfolding *A*
by (*auto simp add*: *image-iff rel-of-def*)
}

{ fix *k v*
assume (*k*, *v*) ∈ *rel-of* (α *m*) *P*
with *sel-correct* **show** *fst* (*the* (*g-min m P*)) ≤ *k*
unfolding *A*
by (*auto simp add*: *image-iff rel-of-def*)
}
qed

lemma *g-max-impl*: *map-max* α *invar g-max*
proof
  fix *m P*

  assume *I*: *invar m*

  from *rev-iterateoi-correct*[*OF I*]
  have *iti'*: *map-iterator-rev-linord* (*rev-iterateoi m*) (α *m*) **by** *simp*
  note *sel-correct* = *iterate-sel-no-map-map-rev-linord-correct*[*OF iti'*, *of P*]

  have *A*: *g-max m P* = *iterate-sel-no-map* (*rev-iterateoi m*) *P*
    unfolding *g-max-def iterate-sel-no-map-def iterate-sel-def* **by** *simp*

  { assume *rel-of* (α *m*) *P* $\neq$ {}
    with *sel-correct*
    show *g-max m P* $\in$ *Some* ‘ *rel-of* (α *m*) *P*
      unfolding *A*
      by (*auto simp add*: *image-iff rel-of-def*)
  }

  { assume *rel-of* (α *m*) *P* = {}
    with *sel-correct* **show** *g-max m P* = *None*
      unfolding *A*
      by (*auto simp add*: *image-iff rel-of-def*)
  }

  { fix *k v*
    assume (*k*, *v*) $\in$ *rel-of* (α *m*) *P*
    with *sel-correct* **show** *fst* (*the* (*g-max m P*)) $\geq$ *k*
      unfolding *A*
      by (*auto simp add*: *image-iff rel-of-def*)
  }
qed

lemma *g-to-sorted-list-impl*: *map-to-sorted-list* α *invar g-to-sorted-list*
proof
  fix *m*
  assume *I*: *invar m*
  note *iti* = *rev-iterateoi-correct*[*OF I*]
  from *iterate-to-list-map-rev-linord-correct*[*OF iti*]
  show *sorted* (*map fst* (*g-to-sorted-list m*))
      distinct (*map fst* (*g-to-sorted-list m*))
      map-of (*g-to-sorted-list m*) = α *m*
    unfolding *g-to-sorted-list-def iterate-to-list-def* **by** *simp-all*
qed

lemma *g-to-rev-list-impl*: *map-to-rev-list* α *invar g-to-rev-list*
proof
  fix *m*

```
    assume I: invar m
    note iti = iterateoi-correct[OF I]
    from iterate-to-list-map-linord-correct[OF iti]
    show sorted (rev (map fst (g-to-rev-list m)))
         distinct (map fst (g-to-rev-list m))
         map-of (g-to-rev-list m) = α m
      unfolding g-to-rev-list-def iterate-to-list-def
      by (simp-all add: rev-map)
  qed

  lemma dflt-oops-impl: StdOMap dflt-oops
  proof −
    interpret aux: StdMap dflt-ops by (rule dflt-ops-impl)

    show ?thesis
      apply (rule StdOMap-intro)
      apply icf-locales
      apply (simp-all add: icf-rec-unf)
      apply (rule g-min-impl)
      apply (rule g-max-impl)
      apply (rule g-to-sorted-list-impl)
      apply (rule g-to-rev-list-impl)
      done
  qed

end

locale g-image-filter-defs-loc =
  m1: StdMapDefs ops1 +
  m2: StdMapDefs ops2
  for ops1 :: ('k1,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k2,'v2,'s2,'m2) map-ops-scheme
begin
  definition g-image-filter f m1 ≡ m1.iterate m1 (λkv σ. case f kv of
      None => σ
    | Some (k',v') => m2.update-dj k' v' σ
    ) (m2.empty ())
end

locale g-image-filter-loc = g-image-filter-defs-loc ops1 ops2 +
  m1: StdMap ops1 +
  m2: StdMap ops2
  for ops1 :: ('k1,'v1,'s1,'m1) map-ops-scheme
  and ops2 :: ('k2,'v2,'s2,'m2) map-ops-scheme
begin
  lemma g-image-filter-impl:
    map-image-filter m1.α m1.invar m2.α m2.invar g-image-filter
  proof
    fix m k' v' and f :: ('k1 × 'v1) ⇒ ('k2 × 'v2) option
```

```
  assume invar-m: m1.invar m and
      unique-f: transforms-to-unique-keys (m1.α m) f

  have A: g-image-filter f m =
    iterate-to-map m2.empty m2.update-dj (
      set-iterator-image-filter f (m1.iteratei m))
    unfolding g-image-filter-def iterate-to-map-alt-def
      set-iterator-image-filter-def case-prod-beta
    by simp

  from m1.iteratei-correct[OF invar-m]
  have iti-m: map-iterator (m1.iteratei m) (m1.α m) by simp

  from unique-f have inj-on-f: inj-on f (map-to-set (m1.α m) ∩ dom f)
    unfolding transforms-to-unique-keys-def inj-on-def Ball-def map-to-set-def
    by auto (metis option.inject)

   define vP where vP k v ⟷ (∃ k' v'. m1.α m k' = Some v' ∧ f (k', v') =
Some (k, v)) for k v
  have vP-intro: ⋀k v. (∃ k' v'. m1.α m k' = Some v'
      ∧ f (k', v') = Some (k, v)) ⟷ vP k v
    unfolding vP-def by simp
  { fix k v
    have Eps-Opt (vP k) = Some v ⟷ vP k v
      using unique-f unfolding vP-def transforms-to-unique-keys-def
      apply (rule-tac Eps-Opt-eq-Some)
      apply (metis prod.inject option.inject)
    done
  } note Eps-vP-elim[simp] = this
  have map-intro: {y. ∃ x. x ∈ map-to-set (m1.α m) ∧ f x = Some y}
    = map-to-set (λk. Eps-Opt (vP k))
    by (simp add: map-to-set-def vP-intro set-eq-iff split: prod.splits)

  from set-iterator-image-filter-correct [OF iti-m, OF inj-on-f,
    unfolded map-intro]
  have iti-filter: map-iterator (set-iterator-image-filter f (m1.iteratei m))
      (λk. Eps-Opt (vP k)) by auto

  have upd: map-update-dj m2.α m2.invar m2.update-dj by unfold-locales
  have emp: map-empty m2.α m2.invar m2.empty by unfold-locales

  from iterate-to-map-correct[OF upd emp iti-filter] show
    map-op-invar ops2 (g-image-filter f m) ∧
      (map-op-α ops2 (g-image-filter f m) k' = Some v') =
      (∃ k v. map-op-α ops1 m k = Some v ∧ f (k, v) = Some (k', v'))
    unfolding A vP-def[symmetric]
    by (simp add: vP-intro)

qed
```

end

sublocale *g-image-filter-loc*
  $<$ *map-image-filter m1.α m1.invar m2.α m2.invar g-image-filter*
  by (*rule g-image-filter-impl*)

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*
  apply (*rule ext*)
  apply (*auto simp*: *restrict-map-def*)
  apply (*case-tac m x*)
  apply *auto*
  done

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
    apply *unfold-locales*
    unfolding *g-value-image-filter-def*
    apply (*rule-tac I*=λ*it* σ. *m2.invar* σ
      ∧ *m2.α* σ = (λ*k*. *Option.bind* (*map-op-α ops1 m k*) (*f k*)) |‘ *it*
      in *m1.old-iterate-rule-insert-P*)

    apply *auto* []
    apply (*auto simp*: *m2.empty-correct*) []
    defer
    apply *simp* []
    apply (*rule restrict-map-dom-subset*)

apply (*auto*) []
apply (*case-tac m1.α m x*)
apply (*auto*) [*2*]

apply (*auto split*: *option.split simp*: *m2.update-dj-correct intro*!: *ext*)
apply (*auto simp*: *restrict-map-def*)
done
end

sublocale *g-value-image-filter-loc*
< *map-value-image-filter m1.α m1.invar m2.α m2.invar g-value-image-filter*
by (*rule g-value-image-filter-impl*)

end

3.2.3 Generic Algorithms for Sets

theory *SetGA*
imports *../spec/SetSpec SetIteratorCollectionsGA*
begin

Generic Set Algorithms

locale *g-set-xx-defs-loc* =
s1: *StdSetDefs ops1* + *s2*: *StdSetDefs ops2*
for *ops1* :: (*'x*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x*,*'s2*,*'more2*) *set-ops-scheme*
begin
definition *g-copy s* ≡ *s1.iterate s s2.ins-dj* (*s2.empty* ())
definition *g-filter P s1* ≡ *s1.iterate s1*
(*λx σ. if P x then s2.ins-dj x σ else σ*)
(*s2.empty* ())

definition *g-union s1 s2* ≡ *s1.iterate s1 s2.ins s2*
definition *g-diff s1 s2* ≡ *s2.iterate s2 s1.delete s1*

definition *g-union-list* **where**
g-union-list l =
foldl (*λs s'. g-union s' s*) (*s2.empty* ()) *l*

definition *g-union-dj s1 s2* ≡ *s1.iterate s1 s2.ins-dj s2*

definition *g-disjoint-witness s1 s2* ≡
s1.sel s1 (*λx. s2.memb x s2*)

definition *g-disjoint s1 s2* ≡
s1.ball s1 (*λx.* ¬*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)
    unfolding iterate-to-set-alt-def g-copy-def ..

  lemma g-copy-impl: set-copy s1.α s1.invar s2.α s2.invar g-copy
  proof -

    have LIS:
      set-ins-dj s2.α s2.invar s2.ins-dj
      set-empty s2.α s2.invar s2.empty
      by unfold-locales

    from iterate-to-set-correct[OF LIS s1.iteratei-correct]
    show ?thesis
      apply unfold-locales
      unfolding g-copy-alt
      by simp-all
  qed

  lemma g-filter-impl: set-filter s1.α s1.invar s2.α s2.invar g-filter
  proof
    fix s P
    assume s1.invar s
    hence s2.α (g-filter P s) = {e ∈ s1.α s. P e} ∧
      s2.invar (g-filter P s) (is ?G1 ∧ ?G2)
      unfolding g-filter-def
      apply (rule-tac I=λit σ. s2.invar σ ∧ s2.α σ = {e ∈ it. P e}
        in s1.iterate-rule-insert-P)
      by (auto simp add: s2.empty-correct s2.ins-dj-correct)
    thus ?G1 ?G2 by auto
  qed

  lemma g-union-alt:
    g-union s1 s2 = iterate-add-to-set s2 s2.ins (s1.iteratei s1)
    unfolding iterate-add-to-set-def g-union-def ..

  lemma g-diff-alt:
    g-diff s1 s2 = iterate-diff-set s1 s1.delete (s2.iteratei s2)
    unfolding g-diff-def iterate-diff-set-def ..

  lemma g-union-impl:
    set-union s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union
  proof -
    have LIS: set-ins s2.α s2.invar s2.ins by unfold-locales
```

  from *iterate-add-to-set-correct*[*OF LIS - s1.iteratei-correct*]
  show *?thesis*
    apply *unfold-locales*
    unfolding *g-union-alt*
    by *simp-all*
qed

lemma *g-diff-impl*:
  set-diff s1.α s1.invar s2.α s2.invar g-diff
proof −
  have *LIS*: *set-delete s1.α s1.invar s1.delete* **by** *unfold-locales*
  from *iterate-diff-correct*[*OF LIS - s2.iteratei-correct*]
  show *?thesis*
    apply *unfold-locales*
    unfolding *g-diff-alt*
    by *simp-all*
qed

lemma *g-union-list-impl*:
  shows *set-union-list s1.α s1.invar s2.α s2.invar g-union-list*
proof
  fix *l*
  note *correct* = *s2.empty-correct set-union.union-correct*[*OF g-union-impl*]

  assume $\forall$ *s1*$\in$*set l*. *s1.invar s1*
  hence *aux*: $\bigwedge$*s*. *s2.invar s* $\Longrightarrow$
        s2.α (*foldl* (λ*s s'*. *g-union s' s*) *s l*)
        = $\bigcup$\{*s1.α s1* |*s1*. *s1* $\in$ *set l*\} $\cup$ *s2.α s* $\wedge$
        s2.invar (*foldl* (λ*s s'*. *g-union s' s*) *s l*)
    by (*induct l*) (*auto simp add*: *correct*)

  from *aux* [*of s2.empty* ()]
  show *s2.α* (*g-union-list l*) = $\bigcup$\{*s1.α s1* |*s1*. *s1* $\in$ *set l*\}
      s2.invar (*g-union-list l*)
    unfolding *g-union-list-def*
    by (*simp-all add*: *correct*)
qed

lemma *g-union-dj-impl*:
  set-union-dj s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union-dj
proof
  fix *s1 s2*
  assume *I*:
    s1.invar s1
    s2.invar s2
  assume *DJ*: *s1.α s1* $\cap$ *s2.α s2* = {}

  have *s2.α* (*g-union-dj s1 s2*)
    = *s1.α s1* $\cup$ *s2.α s2*

```
    ∧ s2.invar (g-union-dj s1 s2) (is ?G1 ∧ ?G2)
    unfolding g-union-dj-def

    apply (rule-tac I=λit σ. s2.invar σ ∧ s2.α σ = it ∪ s2.α s2
      in s1.iterate-rule-insert-P)
    using DJ
    apply (simp-all add: I)
    apply (subgoal-tac x∉s2.α σ)
    apply (simp add: s2.ins-dj-correct I)
    apply auto
    done
  thus ?G1 ?G2 by auto
qed

lemma g-disjoint-witness-impl:
  set-disjoint-witness s1.α s1.invar s2.α s2.invar g-disjoint-witness
proof –
  show ?thesis
    apply unfold-locales
    unfolding g-disjoint-witness-def
    by (auto dest: s1.sel′-noneD s1.sel′-someD simp: s2.memb-correct)
qed

lemma g-disjoint-impl:
  set-disjoint s1.α s1.invar s2.α s2.invar g-disjoint
proof –
  show ?thesis
    apply unfold-locales
    unfolding g-disjoint-def
    by (auto simp: s2.memb-correct s1.ball-correct)
qed
end

sublocale g-set-xx-loc <
  set-copy s1.α s1.invar s2.α s2.invar g-copy by (rule g-copy-impl)

sublocale g-set-xx-loc <
  set-filter s1.α s1.invar s2.α s2.invar g-filter by (rule g-filter-impl)

sublocale g-set-xx-loc <
  set-union s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union
  by (rule g-union-impl)

sublocale g-set-xx-loc <
  set-union-dj s1.α s1.invar s2.α s2.invar s2.α s2.invar g-union-dj
  by (rule g-union-dj-impl)

sublocale g-set-xx-loc <
  set-diff s1.α s1.invar s2.α s2.invar g-diff
```

```
  by (rule g-diff-impl)

sublocale g-set-xx-loc <
  set-disjoint-witness s1.α s1.invar s2.α s2.invar g-disjoint-witness
  by (rule g-disjoint-witness-impl)

sublocale g-set-xx-loc <
  set-disjoint s1.α s1.invar s2.α s2.invar g-disjoint by (rule g-disjoint-impl)
```

```
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
    fix s1 s2
    assume I:
      s1.invar s1
      s2.invar s2
    have s3.α (g-inter s1 s2) = s1.α s1 ∩ s2.α s2 ∧ s3.invar (g-inter s1 s2)
      unfolding g-inter-def
      apply (rule-tac I=λit σ. s3.α σ = it ∩ s2.α s2 ∧ s3.invar σ
        in s1.iterate-rule-insert-P)
      apply (simp-all add: I s3.empty-correct s3.ins-dj-correct s2.memb-correct)
      done
    thus s3.α (g-inter s1 s2) = s1.α s1 ∩ s2.α s2
```

and *s3.invar* (*g-inter s1 s2*) **by** *auto*
qed
end

sublocale *g-set-xxx-loc*
< *set-inter s1.α s1.invar s2.α s2.invar s3.α s3.invar g-inter*
by (*rule g-inter-impl*)

locale *g-set-xy-defs-loc* =
s1: *StdSet ops1* + *s2*: *StdSet ops2*
for *ops1* :: (*'x1*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x2*,*'s2*,*'more2*) *set-ops-scheme*
begin
definition *g-image-filter f s* ≡
s1.iterate s
(*λx res. case f x of Some v* ⇒ *s2.ins v res* | - ⇒ *res*)
(*s2.empty* ())

definition *g-image f s* ≡
s1.iterate s (*λx res. s2.ins* (*f x*) *res*) (*s2.empty* ())

definition *g-inj-image-filter f s* ≡
s1.iterate s
(*λx res. case f x of Some v* ⇒ *s2.ins-dj v res* | - ⇒ *res*)
(*s2.empty* ())

definition *g-inj-image f s* ≡
s1.iterate s (*λx res. s2.ins-dj* (*f x*) *res*) (*s2.empty* ())

end

locale *g-set-xy-loc* = *g-set-xy-defs-loc ops1 ops2* +
s1: *StdSet ops1* + *s2*: *StdSet ops2*
for *ops1* :: (*'x1*,*'s1*,*'more1*) *set-ops-scheme*
and *ops2* :: (*'x2*,*'s2*,*'more2*) *set-ops-scheme*
begin
lemma *g-image-filter-impl*:
set-image-filter s1.α s1.invar s2.α s2.invar g-image-filter
proof
fix *f s*
assume *I*: *s1.invar s*
have *A*: *g-image-filter f s* ==
iterate-to-set s2.empty s2.ins
(*set-iterator-image-filter f* (*s1.iteratei s*))
unfolding *g-image-filter-def*
iterate-to-set-alt-def set-iterator-image-filter-def

by *simp*

have *ins*: *set-ins s2.α s2.invar s2.ins*
and *emp*: *set-empty s2.α s2.invar s2.empty* **by** *unfold-locales*

from *iterate-image-filter-to-set-correct*[*OF ins emp s1.iteratei-correct*]
show *s2.α* (*g-image-filter f s*) =
{*b*. ∃ *a*∈*s1.α s*. *f a* = *Some b*}
s2.invar (*g-image-filter f s*)
unfolding *A* **using** *I* **by** *auto*
qed

lemma *g-image-alt*: *g-image f s* = *g-image-filter* (*Some o f*) *s*
unfolding *g-image-def g-image-filter-def*
by *auto*

lemma *g-image-impl*: *set-image s1.α s1.invar s2.α s2.invar g-image*
proof −
interpret *set-image-filter s1.α s1.invar s2.α s2.invar g-image-filter*
by (*rule g-image-filter-impl*)

show *?thesis*
apply *unfold-locales*
unfolding *g-image-alt*
by (*auto simp add*: *image-filter-correct*)
qed

lemma *g-inj-image-filter-impl*:
set-inj-image-filter s1.α s1.invar s2.α s2.invar g-inj-image-filter
proof
fix *f*::*'x1* ⇀ *'x2* **and** *s*
assume *I*: *s1.invar s* **and** *INJ*: *inj-on f* (*s1.α s* ∩ *dom f*)
have *A*: *g-inj-image-filter f s* ==
iterate-to-set s2.empty s2.ins-dj
(*set-iterator-image-filter f* (*s1.iteratei s*))
unfolding *g-inj-image-filter-def*
iterate-to-set-alt-def set-iterator-image-filter-def
by *simp*

have *ins-dj*: *set-ins-dj s2.α s2.invar s2.ins-dj*
and *emp*: *set-empty s2.α s2.invar s2.empty* **by** *unfold-locales*

from *set-iterator-image-filter-correct*[*OF s1.iteratei-correct*[*OF I*] *INJ*]
have *iti-s1-filter*: *set-iterator*
(*set-iterator-image-filter f* (*s1.iteratei s*))
{*y*. ∃ *x*. *x* ∈ *s1.α s* ∧ *f x* = *Some y*}
by *simp*

```
    from iterate-to-set-correct[OF ins-dj emp, OF iti-s1-filter]
    show s2.α (g-inj-image-filter f s) =
          {b. ∃ a∈s1.α s. f a = Some b}
        s2.invar (g-inj-image-filter f s)
      unfolding A by auto
  qed


  lemma g-inj-image-alt: g-inj-image f s = g-inj-image-filter (Some o f) s
    unfolding g-inj-image-def g-inj-image-filter-def
    by auto

  lemma g-inj-image-impl:
    set-inj-image s1.α s1.invar s2.α s2.invar g-inj-image
  proof −
    interpret set-inj-image-filter
      s1.α s1.invar s2.α s2.invar g-inj-image-filter
      by (rule g-inj-image-filter-impl)

    have AUX: ⋀S f. inj-on f S ⟹ inj-on (Some ∘ f) (S ∩ dom (Some ∘ f))
      by (auto intro!: inj-onI dest: inj-onD)

    show ?thesis
      apply unfold-locales
      unfolding g-inj-image-alt
      by (auto simp add: inj-image-filter-correct AUX)

  qed

end

sublocale g-set-xy-loc < set-image-filter s1.α s1.invar s2.α s2.invar
  g-image-filter by (rule g-image-filter-impl)

sublocale g-set-xy-loc < set-image s1.α s1.invar s2.α s2.invar
  g-image by (rule g-image-impl)

sublocale g-set-xy-loc < set-inj-image s1.α s1.invar s2.α s2.invar
  g-inj-image by (rule g-inj-image-impl)

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 –
  {
   fix s f
   have ⟦s0.invar s; ⋀x. x ∈ s0.α s ⟹ s1.invar (f x)⟧ ⟹
    s2.α (g-Union-image f s) = ⋃(s1.α ‘ f ‘ s0.α s)
    ∧ s2.invar (g-Union-image f s)
    apply (unfold g-Union-image-def)
    apply (rule-tac I=λit res. s2.invar res
     ∧ s2.α res = ⋃(s1.α‘f‘(s0.α s − it)) in s0.iterate-rule-P)
    apply (fastforce simp add: s2.empty-correct union-correct)+
    done
  }
  thus ?thesis
   apply unfold-locales
   apply auto
   done
 qed
end

sublocale g-set-xyy-loc <
 set-Union-image s0.α s0.invar s1.α s1.invar s2.α s2.invar g-Union-image
 by (rule g-Union-image-impl)
```

Default Set Operations

```
record ('x,'s) set-basic-ops =
 bset-op-α :: 's ⇒ 'x set
 bset-op-invar :: 's ⇒ bool
 bset-op-empty :: unit ⇒ 's
 bset-op-memb :: 'x ⇒ 's ⇒ bool
 bset-op-ins :: 'x ⇒ 's ⇒ 's
 bset-op-ins-dj :: 'x ⇒ 's ⇒ 's
 bset-op-delete :: 'x ⇒ 's ⇒ 's
 bset-op-list-it :: ('x,'s) set-list-it
```

record ('x,'s) *oset-basic-ops* = ('x::*linorder*,'s) *set-basic-ops* +
bset-op-ordered-list-it :: 's ⇒ ('x,'x *list*) *set-iterator*
bset-op-rev-list-it :: 's ⇒ ('x,'x *list*) *set-iterator*

locale *StdBasicSetDefs* =
poly-set-iteratei-defs bset-op-list-it ops
for *ops* :: ('x,'s,'*more*) *set-basic-ops-scheme*
begin
abbreviation α **where** α == *bset-op-*α *ops*
abbreviation *invar* **where** *invar* == *bset-op-invar ops*
abbreviation *empty* **where** *empty* == *bset-op-empty ops*
abbreviation *memb* **where** *memb* == *bset-op-memb ops*
abbreviation *ins* **where** *ins* == *bset-op-ins ops*
abbreviation *ins-dj* **where** *ins-dj* == *bset-op-ins-dj ops*
abbreviation *delete* **where** *delete* == *bset-op-delete ops*
abbreviation *list-it* **where** *list-it* ≡ *bset-op-list-it ops*
end

locale *StdBasicOSetDefs* = *StdBasicSetDefs ops*
\+ *poly-set-iterateoi-defs bset-op-ordered-list-it ops*
\+ *poly-set-rev-iterateoi-defs bset-op-rev-list-it ops*
for *ops* :: ('x::*linorder*,'s,'*more*) *oset-basic-ops-scheme*
begin
abbreviation *ordered-list-it* ≡ *bset-op-ordered-list-it ops*
abbreviation *rev-list-it* ≡ *bset-op-rev-list-it ops*
end

locale *StdBasicSet* = *StdBasicSetDefs ops* +
set α *invar* +
set-empty α *invar empty* +
set-memb α *invar memb* +
set-ins α *invar ins* +
set-ins-dj α *invar ins-dj* +
set-delete α *invar delete* +
poly-set-iteratei α *invar list-it*
for *ops* :: ('x,'s,'*more*) *set-basic-ops-scheme*
begin

lemmas *correct*[*simp*] =
empty-correct
memb-correct
ins-correct
ins-dj-correct
delete-correct

end

locale *StdBasicOSet* =
StdBasicOSetDefs ops +

StdBasicSet ops +
poly-set-iterateoi α *invar ordered-list-it* +
poly-set-rev-iterateoi α *invar rev-list-it*
for *ops* :: (*'x::linorder,'s,'more*) *oset-basic-ops-scheme*
begin
end

context *StdBasicSetDefs*
begin
definition *g-sng x* ≡ *ins x* (*empty* ())
definition *g-isEmpty s* ≡ *iteratei s* (λ*c. c*) (λ- -. *False*) *True*
definition *g-sel' s P* ≡ *iteratei s* ((=) *None*)
(λ*x* -. *if P x then Some x else None*) *None*

definition *g-ball s P* ≡ *iteratei s* (λ*c. c*) (λ*x σ. P x*) *True*
definition *g-bex s P* ≡ *iteratei s* (λ*c.* ¬*c*) (λ*x σ. P x*) *False*
definition *g-size s* ≡ *iteratei s* (λ-. *True*) (λ*x n . Suc n*) *0*
definition *g-size-abort m s* ≡ *iteratei s* (λσ. σ < *m*) (λ*x. Suc*) *0*
definition *g-isSng s* ≡ (*iteratei s* (λσ. σ < *2*) (λ*x. Suc*) *0* = *1*)

definition *g-union s1 s2* ≡ *iterate s1 ins s2*
definition *g-diff s1 s2* ≡ *iterate s2 delete s1*

definition *g-subset s1 s2* ≡ *g-ball s1* (λ*x. memb x s2*)

definition *g-equal s1 s2* ≡ *g-subset s1 s2* ∧ *g-subset s2 s1*

definition *g-to-list s* ≡ *iterate s* (#) []

fun *g-from-list-aux* **where**
g-from-list-aux accs [] = *accs* |
g-from-list-aux accs (*x*#*l*) = *g-from-list-aux* (*ins x accs*) *l*
— Tail recursive version

definition *g-from-list l* == *g-from-list-aux* (*empty* ()) *l*

definition *g-inter s1 s2* ≡
iterate s1 (λ*x s. if memb x s2 then ins-dj x s else s*)
(*empty* ())

definition *g-union-dj s1 s2* ≡ *iterate s1 ins-dj s2*
definition *g-filter P s* ≡ *iterate s*
(λ*x σ. if P x then ins-dj x σ else σ*)
(*empty* ())

definition *g-disjoint-witness s1 s2* ≡ *g-sel' s1* (λ*x. memb x s2*)
definition *g-disjoint s1 s2* ≡ *g-ball s1* (λ*x.* ¬*memb x s2*)

definition *dflt-ops*

where [*icf-rec-def*]: *dflt-ops* ≡ (|
set-op-α = *α*,
set-op-invar = *invar*,
set-op-empty = *empty*,
set-op-memb = *memb*,
set-op-ins = *ins*,
set-op-ins-dj = *ins-dj*,
set-op-delete = *delete*,
set-op-list-it = *list-it*,
set-op-sng = *g-sng* ,
set-op-isEmpty = *g-isEmpty* ,
set-op-isSng = *g-isSng* ,
set-op-ball = *g-ball* ,
set-op-bex = *g-bex* ,
set-op-size = *g-size* ,
set-op-size-abort = *g-size-abort* ,
set-op-union = *g-union* ,
set-op-union-dj = *g-union-dj* ,
set-op-diff = *g-diff* ,
set-op-filter = *g-filter* ,
set-op-inter = *g-inter* ,
set-op-subset = *g-subset* ,
set-op-equal = *g-equal* ,
set-op-disjoint = *g-disjoint* ,
set-op-disjoint-witness = *g-disjoint-witness* ,
set-op-sel = *g-sel′* ,
set-op-to-list = *g-to-list* ,
set-op-from-list = *g-from-list*
|)

local-setup ‹*Locale-Code.lc-decl-del* @{*term dflt-ops*}›
end

context *StdBasicSet*
begin
lemma *g-sng-impl*: *set-sng α invar g-sng*
apply *unfold-locales*
unfolding *g-sng-def*
apply (*auto simp*: *ins-correct empty-correct*)
done

lemma *g-ins-dj-impl*: *set-ins-dj α invar ins*
by *unfold-locales* (*auto simp*: *ins-correct*)

lemma *g-isEmpty-impl*: *set-isEmpty α invar g-isEmpty*
proof
fix *s* **assume** *I*: *invar s*
have *A*: *g-isEmpty s* = *iterate-is-empty* (*iteratei s*)
unfolding *g-isEmpty-def iterate-is-empty-def* **..**

from *iterate-is-empty-correct*[*OF iteratei-correct*[*OF I*]]
show *g-isEmpty s* $\longleftrightarrow \alpha\ s = \{\}$ **unfolding** *A* **.**
qed

lemma *g-sel′-impl*: *set-sel′ α invar g-sel′*
proof −
have *A*: $\bigwedge s\ P.$ *g-sel′ s P* = *iterate-sel-no-map* (*iteratei s*) *P*
unfolding *g-sel′-def iterate-sel-no-map-alt-def*
apply (*rule arg-cong*[**where** *f*=*iteratei*, *THEN cong*, *THEN cong*, *THEN cong*])
by *auto*

show *?thesis*
unfolding *set-sel′-def A*
using *iterate-sel-no-map-correct*[*OF iteratei-correct*]
apply (*simp add*: *Bex-def Ball-def*)
apply (*metis option.exhaust*)
done
qed

lemma *g-ball-alt*: *g-ball s P* = *iterate-ball* (*iteratei s*) *P*
unfolding *g-ball-def iterate-ball-def* **by** (*simp add*: *id-def*)
lemma *g-bex-alt*: *g-bex s P* = *iterate-bex* (*iteratei s*) *P*
unfolding *g-bex-def iterate-bex-def* **..**

lemma *g-ball-impl*: *set-ball α invar g-ball*
apply *unfold-locales*
unfolding *g-ball-alt*
apply (*rule iterate-ball-correct*)
by (*rule iteratei-correct*)

lemma *g-bex-impl*: *set-bex α invar g-bex*
apply *unfold-locales*
unfolding *g-bex-alt*
apply (*rule iterate-bex-correct*)
by (*rule iteratei-correct*)

lemma *g-size-alt*: *g-size s* = *iterate-size* (*iteratei s*)
unfolding *g-size-def iterate-size-def* **..**
lemma *g-size-abort-alt*: *g-size-abort m s* = *iterate-size-abort* (*iteratei s*) *m*
unfolding *g-size-abort-def iterate-size-abort-def* **..**

lemma *g-size-impl*: *set-size α invar g-size*
apply *unfold-locales*
unfolding *g-size-alt*
apply (*rule conjunct1*[*OF iterate-size-correct*])
by (*rule iteratei-correct*)

lemma *g-size-abort-impl*: *set-size-abort α invar g-size-abort*

apply *unfold-locales*
unfolding *g-size-abort-alt*
apply (*rule conjunct1*[*OF iterate-size-abort-correct*])
by (*rule iteratei-correct*)

lemma *g-isSng-alt*: *g-isSng s* = *iterate-is-sng* (*iteratei s*)
unfolding *g-isSng-def iterate-is-sng-def iterate-size-abort-def* **..**

lemma *g-isSng-impl*: *set-isSng* α *invar g-isSng*
apply *unfold-locales*
unfolding *g-isSng-alt*
apply (*drule iterate-is-sng-correct*[*OF iteratei-correct*])
apply (*simp add*: *card-Suc-eq*)
done

lemma *g-union-impl*: *set-union* α *invar* α *invar* α *invar g-union*
unfolding *g-union-def*[*abs-def*]
apply *unfold-locales*
apply (*rule-tac I*=λ*it* σ. *invar* $\sigma \wedge \alpha\ \sigma = it \cup \alpha\ s2$
in *iterate-rule-insert-P*, *simp-all*)+
done

lemma *g-diff-impl*: *set-diff* α *invar* α *invar g-diff*
unfolding *g-diff-def*[*abs-def*]
apply (*unfold-locales*)
apply (*rule-tac I*=λ*it* σ. *invar* $\sigma \wedge \alpha\ \sigma = \alpha\ s1 - it$
in *iterate-rule-insert-P*, *auto*)+
done

lemma *g-subset-impl*: *set-subset* α *invar* α *invar g-subset*
proof −
interpret *set-ball* α *invar g-ball* **by** (*rule g-ball-impl*)

show *?thesis*
apply *unfold-locales*
unfolding *g-subset-def*
by (*auto simp add*: *ball-correct memb-correct*)
qed

lemma *g-equal-impl*: *set-equal* α *invar* α *invar g-equal*
proof −
interpret *set-subset* α *invar* α *invar g-subset* **by** (*rule g-subset-impl*)

show *?thesis*
apply *unfold-locales*
unfolding *g-equal-def*
by (*auto simp add*: *subset-correct*)
qed

```
lemma g-to-list-impl: set-to-list α invar g-to-list
proof
  fix s
  assume I: invar s
  have A: g-to-list s = iterate-to-list (iteratei s)
    unfolding g-to-list-def iterate-to-list-def ..

  from iterate-to-list-correct [OF iteratei-correct[OF I]]
  show set (g-to-list s) = α s and distinct (g-to-list s)
    unfolding A
    by auto
qed

lemma g-from-list-impl: list-to-set α invar g-from-list
proof −
  { — Show a generalized lemma
    fix l accs
    have invar accs ⟹ α (g-from-list-aux accs l) = set l ∪ α accs
          ∧ invar (g-from-list-aux accs l)
      by (induct l arbitrary: accs)
         (auto simp add: ins-correct)
  } thus ?thesis
    apply (unfold-locales)
    apply (unfold g-from-list-def)
    apply (auto simp add: empty-correct)
    done
qed

lemma g-inter-impl: set-inter α invar α invar α invar g-inter
  unfolding g-inter-def[abs-def]
  apply unfold-locales
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∩ α s2
    in iterate-rule-insert-P, auto) []
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∩ α s2
    in iterate-rule-insert-P, auto)
  done

lemma g-union-dj-impl: set-union-dj α invar α invar α invar g-union-dj
  unfolding g-union-dj-def[abs-def]
  apply unfold-locales
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∪ α s2
    in iterate-rule-insert-P)
  apply simp
  apply simp
  apply (subgoal-tac x∉α σ)
  apply simp
  apply blast
  apply simp
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∪ α s2
```

```
    in iterate-rule-insert-P)
  apply simp
  apply simp
  apply (subgoal-tac x∉α σ)
  apply simp
  apply blast
  apply simp
  done

lemma g-filter-impl: set-filter α invar α invar g-filter
  unfolding g-filter-def[abs-def]
  apply (unfold-locales)
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∩ Collect P
    in iterate-rule-insert-P, auto)
  apply (rule-tac I=λit σ. invar σ ∧ α σ = it ∩ Collect P
    in iterate-rule-insert-P, auto)
  done

lemma g-disjoint-witness-impl: set-disjoint-witness
  α invar α invar g-disjoint-witness
proof –
  interpret set-sel' α invar g-sel' by (rule g-sel'-impl)
  show ?thesis
    unfolding g-disjoint-witness-def[abs-def]
    apply unfold-locales
    by (auto dest: sel'-noneD sel'-someD simp: memb-correct)
qed

lemma g-disjoint-impl: set-disjoint
  α invar α invar g-disjoint
proof –
  interpret set-ball α invar g-ball by (rule g-ball-impl)
  show ?thesis
    apply unfold-locales
    unfolding g-disjoint-def
    by (auto simp: memb-correct ball-correct)
qed

end

context StdBasicSet
begin
  lemma dflt-ops-impl: StdSet dflt-ops
    apply (rule StdSet-intro)
    apply icf-locales
    apply (simp-all add: icf-rec-unf)
    apply (rule g-sng-impl g-isEmpty-impl g-isSng-impl g-ball-impl
      g-bex-impl g-size-impl g-size-abort-impl g-union-impl g-union-dj-impl
      g-diff-impl g-filter-impl g-inter-impl
```

```
      g-subset-impl g-equal-impl g-disjoint-impl
      g-disjoint-witness-impl g-sel'-impl g-to-list-impl
      g-from-list-impl
    )+
    done
end

context StdBasicOSetDefs
begin
  definition g-min s P ≡ iterateoi s (λx. x = None)
    (λx -. if P x then Some x else None) None
  definition g-max s P ≡ rev-iterateoi s (λx. x = None)
    (λx -. if P x then Some x else None) None

  definition g-to-sorted-list s ≡ rev-iterateo s (#) []
  definition g-to-rev-list s ≡ iterateo s (#) []

  definition dflt-oops :: ('x::linorder,'s) oset-ops
    where [icf-rec-def]:
    dflt-oops ≡ set-ops.extend
      dflt-ops
      (|
        set-op-ordered-list-it = ordered-list-it,
        set-op-rev-list-it = rev-list-it,
        set-op-min = g-min,
        set-op-max = g-max,
        set-op-to-sorted-list = g-to-sorted-list,
        set-op-to-rev-list = g-to-rev-list
      |)
  local-setup ‹Locale-Code.lc-decl-del @{term dflt-oops}›

end

context StdBasicOSet
begin
  lemma g-min-impl: set-min α invar g-min
  proof
    fix s P

    assume I: invar s

    from iterateoi-correct[OF I]
    have iti': set-iterator-linord (iterateoi s) (α s) by simp
    note sel-correct = iterate-sel-no-map-linord-correct[OF iti', of P]

    have A: g-min s P = iterate-sel-no-map (iterateoi s) P
      unfolding g-min-def iterate-sel-no-map-def iterate-sel-def by simp

    { fix x
```

assume $x \in \alpha\ s$ $P\ x$
with *sel-correct*
show *g-min s P* $\in$ *Some* ‘ $\{x \in \alpha\ s.\ P\ x\}$ **and** *the* (*g-min s P*) $\leq x$
unfolding A **by** *auto*
}

{ **assume** $\{x \in \alpha\ s.\ P\ x\} = \{\}$
with *sel-correct* **show** *g-min s P* = *None*
unfolding A **by** *auto*
}
qed

lemma *g-max-impl*: *set-max* α *invar g-max*
proof
fix *s P*

assume *I*: *invar s*

from *rev-iterateoi-correct*[*OF I*]
have *iti′*: *set-iterator-rev-linord* (*rev-iterateoi s*) (α *s*) **by** *simp*
note *sel-correct* = *iterate-sel-no-map-rev-linord-correct*[*OF iti′*, *of P*]

have *A*: *g-max s P* = *iterate-sel-no-map* (*rev-iterateoi s*) *P*
unfolding *g-max-def iterate-sel-no-map-def iterate-sel-def* **by** *simp*

{ **fix** *x*
assume $x \in \alpha\ s$ $P\ x$
with *sel-correct*
show *g-max s P* $\in$ *Some* ‘ $\{x \in \alpha\ s.\ P\ x\}$ **and** *the* (*g-max s P*) $\geq x$
unfolding A **by** *auto*
}

{ **assume** $\{x \in \alpha\ s.\ P\ x\} = \{\}$
with *sel-correct* **show** *g-max s P* = *None*
unfolding A **by** *auto*
}
qed

lemma *g-to-sorted-list-impl*: *set-to-sorted-list* α *invar g-to-sorted-list*
proof
fix *s*
assume *I*: *invar s*
note *iti* = *rev-iterateoi-correct*[*OF I*]
from *iterate-to-list-rev-linord-correct*[*OF iti*]
show *sorted* (*g-to-sorted-list s*)
distinct (*g-to-sorted-list s*)
set (*g-to-sorted-list s*) = α *s*
unfolding *g-to-sorted-list-def iterate-to-list-def* **by** *simp-all*
qed

lemma *g-to-rev-list-impl*: *set-to-rev-list* α *invar g-to-rev-list*
proof
  fix *s*
  assume *I*: *invar s*
  note *iti* = *iterateoi-correct*[*OF I*]
  from *iterate-to-list-linord-correct*[*OF iti*]
  show *sorted* (*rev* (*g-to-rev-list s*))
      distinct (*g-to-rev-list s*)
      set (*g-to-rev-list s*) = α *s*
    unfolding *g-to-rev-list-def iterate-to-list-def*
    by (*simp-all*)
qed

lemma *dflt-oops-impl*: *StdOSet dflt-oops*
proof −
  interpret *aux*: *StdSet dflt-ops* **by** (*rule dflt-ops-impl*)

  show *?thesis*
    apply (*rule StdOSet-intro*)
    apply *icf-locales*
    apply (*simp-all add*: *icf-rec-unf*)
    apply (*rule g-min-impl*)
    apply (*rule g-max-impl*)
    apply (*rule g-to-sorted-list-impl*)
    apply (*rule g-to-rev-list-impl*)
    done
qed

end

More Generic Set Algorithms

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

Image and Filter of Cartesian Product

locale *image-filter-cp-defs-loc* =
  s1: *StdSetDefs ops1* +
  s2: *StdSetDefs ops2* +
  s3: *StdSetDefs ops3*
  for *ops1* :: ('*x*,'*s1*,'*more1*) *set-ops-scheme*
  and *ops2* :: ('*y*,'*s2*,'*more2*) *set-ops-scheme*
  and *ops3* :: ('*z*,'*s3*,'*more3*) *set-ops-scheme*
begin

  definition *image-filter-cartesian-product f s1 s2* ==
    s1.iterate s1 (λ*x res.*
      s2.iterate s2 (λ*y res.*

```
      case (f (x, y)) of
        None ⇒ res
      | Some z ⇒ (s3.ins z res)
    ) res
  ) (s3.empty ())

 lemma image-filter-cartesian-product-alt:
   image-filter-cartesian-product f s1 s2 ==
    iterate-to-set s3.empty s3.ins (set-iterator-image-filter f (
      set-iterator-product (s1.iteratei s1) (λ-. s2.iteratei s2)))
   unfolding image-filter-cartesian-product-def iterate-to-set-alt-def
     set-iterator-image-filter-def set-iterator-product-def
   by simp

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

end

locale image-filter-cp-loc = image-filter-cp-defs-loc ops1 ops2 ops3 +
 s1: StdSet ops1 +
 s2: StdSet ops2 +
 s3: StdSet ops3
 for ops1 :: ('x,'s1,'more1) set-ops-scheme
 and ops2 :: ('y,'s2,'more2) set-ops-scheme
 and ops3 :: ('z,'s3,'more3) set-ops-scheme
begin

 lemma image-filter-cartesian-product-correct:
   fixes f :: 'x × 'y ⇀ 'z
   assumes I[simp, intro!]: s1.invar s1   s2.invar s2
   shows s3.α (image-filter-cartesian-product f s1 s2)
    = { z | x y z. f (x,y) = Some z ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (is ?T1)
   s3.invar (image-filter-cartesian-product f s1 s2) (is ?T2)
 proof −
   from set-iterator-product-correct
     [OF s1.iteratei-correct[OF I(1)] s2.iteratei-correct[OF I(2)]]
     have it-s12: set-iterator
       (set-iterator-product (s1.iteratei s1) (λ-. s2.iteratei s2))
       (s1.α s1 × s2.α s2)
       by simp

     have LIS:
       set-ins s3.α s3.invar s3.ins
       set-empty s3.α s3.invar s3.empty
       by unfold-locales
```

from *iterate-image-filter-to-set-correct*[*OF LIS it-s12*, *of f*]
show *?T1 ?T2*
unfolding *image-filter-cartesian-product-alt* **by** *auto*
qed

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*) $\wedge$ *x*$\in$*s1*.α *s1* $\wedge$ *y*$\in$*s2*.α *s2* } (**is** *?T1*)
s3.invar (*image-filter-cp f P s1 s2*) (**is** *?T2*)
proof −
from *image-filter-cartesian-product-correct* [*OF I*]
show *?T1* *?T2*
unfolding *image-filter-cp-def*
by *auto*
qed

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 (λ*x res*.
s2.iterate s2 (λ*y res*.
case (*f* (*x*, *y*)) *of*
None $\Rightarrow$ *res*
| *Some z* $\Rightarrow$ (*s3.ins-dj z res*)
) *res*
) (*s3.empty* ())

lemma *inj-image-filter-cartesian-product-alt*:
inj-image-filter-cartesian-product f s1 s2 ==
iterate-to-set s3.empty s3.ins-dj (*set-iterator-image-filter f* (
set-iterator-product (*s1.iteratei s1*) (λ-. *s2.iteratei s2*)))
unfolding *inj-image-filter-cartesian-product-def iterate-to-set-alt-def*
set-iterator-image-filter-def set-iterator-product-def
by *simp*

definition *inj-image-filter-cp* **where**
inj-image-filter-cp f P s1 s2 $\equiv$
inj-image-filter-cartesian-product

```
    (λxy. if P xy then Some (f xy) else None) s1 s2

end

locale inj-image-filter-cp-loc = inj-image-filter-cp-defs-loc ops1 ops2 ops3 +
  s1: StdSet ops1 +
  s2: StdSet ops2 +
  s3: StdSet ops3
  for ops1 :: ('x,'s1,'more1) set-ops-scheme
  and ops2 :: ('y,'s2,'more2) set-ops-scheme
  and ops3 :: ('z,'s3,'more3) set-ops-scheme
begin

  lemma inj-image-filter-cartesian-product-correct:
    fixes f :: 'x × 'y ⇀ 'z
    assumes I[simp, intro!]: s1.invar s1   s2.invar s2
    assumes INJ: inj-on f (s1.α s1 × s2.α s2 ∩ dom f)
    shows s3.α (inj-image-filter-cartesian-product f s1 s2)
     = { z | x y z. f (x,y) = Some z ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (is ?T1)
    s3.invar (inj-image-filter-cartesian-product f s1 s2) (is ?T2)
  proof -
    from set-iterator-product-correct
      [OF s1.iteratei-correct[OF I(1)] s2.iteratei-correct[OF I(2)]]
      have it-s12: set-iterator
        (set-iterator-product (s1.iteratei s1) (λ-. s2.iteratei s2))
        (s1.α s1 × s2.α s2)
        by simp

      have LIS:
        set-ins-dj s3.α s3.invar s3.ins-dj
        set-empty s3.α s3.invar s3.empty
        by unfold-locales

      from iterate-inj-image-filter-to-set-correct[OF LIS it-s12 INJ]
      show ?T1 ?T2
        unfolding inj-image-filter-cartesian-product-alt by auto
  qed

  lemma inj-image-filter-cp-correct:
    assumes I: s1.invar s1   s2.invar s2
    assumes INJ: inj-on f {x∈s1.α s1 × s2.α s2. P x}
    shows
    s3.α (inj-image-filter-cp f P s1 s2)
     = { f (x, y) | x y. P (x, y) ∧ x∈s1.α s1 ∧ y∈s2.α s2 } (is ?T1)
    s3.invar (inj-image-filter-cp f P s1 s2) (is ?T2)
  proof -
    let ?f = λxy. if P xy then Some (f xy) else None
    from INJ have INJ': inj-on ?f (s1.α s1 × s2.α s2 ∩ dom ?f)
      by (force intro!: inj-onI dest: inj-onD split: if-split-asm)
```

```
    from inj-image-filter-cartesian-product-correct [OF I INJ']
    show ?T1  ?T2
      unfolding inj-image-filter-cp-def
      by auto
  qed

end
```

Cartesian Product

```
locale cart-defs-loc = inj-image-filter-cp-defs-loc ops1 ops2
ops3
  for ops1 :: ('x,'s1,'more1) set-ops-scheme
  and ops2 :: ('y,'s2,'more2) set-ops-scheme
  and ops3 :: ('x×'y,'s3,'more3) set-ops-scheme
begin

  definition cart s1 s2 ≡
    s1.iterate s1
      (λx. s2.iterate s2 (λy res. s3.ins-dj (x,y) res))
      (s3.empty ())

  lemma cart-alt: cart s1 s2 ==
    inj-image-filter-cartesian-product Some s1 s2
    unfolding cart-def inj-image-filter-cartesian-product-def
    by simp

end

locale cart-loc = cart-defs-loc ops1 ops2 ops3
  + inj-image-filter-cp-loc ops1 ops2 ops3
  for ops1 :: ('x,'s1,'more1) set-ops-scheme
  and ops2 :: ('y,'s2,'more2) set-ops-scheme
  and ops3 :: ('x×'y,'s3,'more3) set-ops-scheme
begin

  lemma cart-correct:
    assumes I[simp, intro!]: s1.invar s1  s2.invar s2
    shows s3.α (cart s1 s2)
          = s1.α s1 × s2.α s2 (is ?T1)
    s3.invar (cart s1 s2) (is ?T2)
    unfolding cart-alt
    by (auto simp add:
      inj-image-filter-cartesian-product-correct[OF I, where f=Some])

end
```

Generic Algorithms outside basic-set

In this section, we present some generic algorithms that are not formulated in terms of basic-set. They are useful for setting up some data structures.

Image (by image-filter)

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

lemma *iflt-image-correct*:
  assumes *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt*
  shows *set-image* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*iflt-image iflt*)
proof −
  interpret *set-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*unfold iflt-image-def*)
    apply (*auto simp add*: *image-filter-correct*)
    done
qed

Injective Image-Filter (by image-filter)

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

lemma *iflt-inj-image-correct*:
  assumes *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt*
  shows *set-inj-image* $\alpha 1$ *invar1* $\alpha 2$ *invar2* (*iflt-inj-image iflt*)
proof −
  interpret *set-inj-image-filter* $\alpha 1$ *invar1* $\alpha 2$ *invar2 iflt* **by** *fact*

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

Filter (by image-filter)

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

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

shows *set-filter α1 invar1 α2 invar2* (*iflt-filter iflt*)
proof (*rule set-filter.intro*)
fix *s P*
assume *invar-s*: *invar1 s*
interpret *S*: *set-inj-image-filter α1 invar1 α2 invar2 iflt* **by** *fact*

let *?f′* = *λx::′a. if P x then Some x else None*
have *inj-f′*: *inj-on ?f′* (*α1 s* ∩ *dom ?f′*)
by (*simp add*: *inj-on-def Ball-def domIff*)
note *correct′* = *S.inj-image-filter-correct* [*OF invar-s inj-f′*, *folded iflt-filter-def*]

show *invar2* (*iflt-filter iflt P s*)
α2 (*iflt-filter iflt P s*) = {*e* ∈ *α1 s*. *P e*}
by (*auto simp add*: *correct′*)
qed

end

3.2.4 Implementing Sets by Maps

theory *SetByMap*
imports
../spec/SetSpec
../spec/MapSpec
SetGA
MapGA
begin

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

Auxiliary lemma

lemma *foldli-foldli-map-eq*:
foldli (*foldli l* (*λx. True*) (*λx l. l*@[*f x*]) []) *c f′ σ0*
= *foldli l c* (*f′ o f*) *σ0*
proof −
show *?thesis*
apply (*simp add*: *map-by-foldl foldli-map foldli-foldl*)
done
qed

locale *SetByMapDefs* =
map: *StdBasicMapDefs ops*
for *ops* :: (*′x*,*unit*,*′s*,*′more*) *map-basic-ops-scheme*
begin
definition *α s* ≡ *dom* (*map.α s*)
definition *invar s* ≡ *map.invar s*
definition *empty* **where** *empty* ≡ *map.empty*
definition *memb x s* ≡ *map.lookup x s* ≠ *None*
definition *ins x s* ≡ *map.update x* () *s*

```
definition ins-dj x s ≡ map.update-dj x () s
definition delete x s ≡ map.delete x s
definition list-it :: 's ⇒ ('x,'x list) set-iterator
  where list-it s c f σ0 ≡ it-to-it (map.list-it s) c (f o fst) σ0

local-setup ‹Locale-Code.lc-decl-del @{term list-it}›

lemma list-it-alt: list-it s = map-iterator-dom (map.iteratei s)
proof –
  have A: ⋀f. (λ(x,-). f x) = (λx. f (fst x)) by auto
  show ?thesis
    unfolding list-it-def[abs-def] map-iterator-dom-def
      poly-map-iteratei-defs.iteratei-def
      set-iterator-image-def set-iterator-image-filter-def
    by (auto simp: comp-def)
qed

lemma list-it-unfold:
  it-to-it (list-it s) c f σ0 = map.iteratei s c (f o fst) σ0
  unfolding list-it-def[abs-def] it-to-it-def
  unfolding poly-map-iteratei-defs.iteratei-def it-to-it-def
  by (simp add: foldli-foldli-map-eq comp-def)

definition [icf-rec-def]: dflt-basic-ops ≡ (|
  bset-op-α = α,
  bset-op-invar = invar,
  bset-op-empty = empty,
  bset-op-memb = memb,
  bset-op-ins = ins,
  bset-op-ins-dj = ins-dj,
  bset-op-delete = delete,
  bset-op-list-it = list-it
  |)
local-setup ‹Locale-Code.lc-decl-del @{term dflt-basic-ops}›

end

setup ‹
  (Record-Intf.add-unf-thms-global @{thms
    SetByMapDefs.list-it-def[abs-def]
  })
›



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
    apply unfold-locales
    unfolding α-def invar-def empty-def
    by (auto simp: map.empty-correct)

  lemma memb-impl: set-memb α invar memb
    apply unfold-locales
    unfolding α-def invar-def memb-def
    by (auto simp: map.lookup-correct)

  lemma ins-impl: set-ins α invar ins
    apply unfold-locales
    unfolding α-def invar-def ins-def
    by (auto simp: map.update-correct)

  lemma ins-dj-impl: set-ins-dj α invar ins-dj
    apply unfold-locales
    unfolding α-def invar-def ins-dj-def
    by (auto simp: map.update-dj-correct)

  lemma delete-impl: set-delete α invar delete
    apply unfold-locales
    unfolding α-def invar-def delete-def
    by (auto simp: map.delete-correct)

  lemma list-it-impl: poly-set-iteratei α invar list-it
  proof
    fix s
    assume I: invar s
    hence I': map.invar s unfolding invar-def .

    have S: ⋀f. (λ(x,-). f x) = (λxy. f (fst xy))
      by auto

    from map-iterator-dom-correct[OF map.iteratei-correct[OF I']]
    show set-iterator (list-it s) (α s)
      unfolding α-def list-it-alt .

    show finite (α s)
      unfolding α-def by (simp add: map.finite[OF I'])
  qed

  lemma dflt-basic-ops-impl: StdBasicSet dflt-basic-ops
    apply (rule StdBasicSet.intro)
    apply (simp-all add: icf-rec-unf)
    apply (rule empty-impl memb-impl ins-impl
      ins-dj-impl delete-impl
      list-it-impl[unfolded SetByMapDefs.list-it-def[abs-def]]
```

```
    )+
    done
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

  local-setup ‹Locale-Code.lc-decl-del @{term ordered-list-it}›

  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

  local-setup ‹Locale-Code.lc-decl-del @{term rev-list-it}›


  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
      ⦈
  local-setup ‹Locale-Code.lc-decl-del @{term dflt-basic-oops}›

end

setup ‹
  (Record-Intf.add-unf-thms-global @{thms
    OSetByOMapDefs.ordered-list-it-def[abs-def]
    OSetByOMapDefs.rev-list-it-def[abs-def]
  })
›



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
    fix s
    assume I: invar s
    hence I': map.invar s unfolding invar-def .
```

```
  have S: ⋀f. (λ(x,-). f x) = (λxy. f (fst xy))
    by auto

  have A: ⋀s. ordered-list-it s = map-iterator-dom (map.iterateoi s)
    unfolding ordered-list-it-def[abs-def]
      map-iterator-dom-def set-iterator-image-alt-def map.iterateoi-def
    by (simp add: S comp-def)

  from map-iterator-linord-dom-correct[OF map.iterateoi-correct[OF I']]
  show set-iterator-linord (ordered-list-it s) (α s)
    unfolding α-def A .

  show finite (α s)
    unfolding α-def by (simp add: map.finite[OF I'])
qed

lemma rev-list-it-impl: poly-set-rev-iterateoi α invar rev-list-it
proof
  fix s
  assume I: invar s
  hence I': map.invar s unfolding invar-def .

  have S: ⋀f. (λ(x,-). f x) = (λxy. f (fst xy))
    by auto

  have A: ⋀s. rev-list-it s = map-iterator-dom (map.rev-iterateoi s)
    unfolding rev-list-it-def[abs-def]
      map-iterator-dom-def set-iterator-image-alt-def map.rev-iterateoi-def
    by (simp add: S comp-def)

  from map-iterator-rev-linord-dom-correct[
    OF map.rev-iterateoi-correct[OF I']]
  show set-iterator-rev-linord (rev-list-it s) (α s)
    unfolding α-def A .

  show finite (α s)
    unfolding α-def by (simp add: map.finite[OF I'])

qed

lemma dflt-basic-oops-impl: StdBasicOSet dflt-basic-oops
proof –
  interpret aux: StdBasicSet dflt-basic-ops by (rule dflt-basic-ops-impl)

  show ?thesis
    apply (rule StdBasicOSet.intro)
    apply (rule StdBasicSet.intro)
    apply icf-locales
    apply (simp-all add: icf-rec-unf)
```

apply (*rule*
ordered-list-it-impl[*unfolded ordered-list-it-def*[*abs-def*]]
rev-list-it-impl[*unfolded rev-list-it-def*[*abs-def*]]
)+
done
qed
end

sublocale *SetByMap* < *basic*: *StdBasicSet dflt-basic-ops*
by (*rule dflt-basic-ops-impl*)

sublocale *OSetByOMap* < *obasic*: *StdBasicOSet dflt-basic-oops*
by (*rule dflt-basic-oops-impl*)

lemma *proper-it'-map2set*: *proper-it' it it'*
$\Longrightarrow$ *proper-it'* ($\lambda s\ c\ f.\ it\ s\ c\ (f\ o\ fst)$) ($\lambda s\ c\ f.\ it'\ s\ c\ (f\ o\ fst)$)
unfolding *proper-it'-def*
apply *clarsimp*
apply (*drule-tac x=s* **in** *spec*)
apply (*erule proper-itE*)
apply (*rule proper-itI*)
apply (*auto simp add*: *foldli-map*[*symmetric*] *intro*!: *ext*)
done

end

3.2.5 Generic Algorithms for Sequences

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

Iterators

iteratei (by get, size) **locale** *idx-iteratei-loc* =
list-size + *list-get* +
constrains α :: $'s \Rightarrow 'a\ list$
assumes [*simp*]: $\bigwedge s.\ invar\ s$
begin

fun *idx-iteratei-aux*
:: $nat \Rightarrow nat \Rightarrow 's \Rightarrow ('\sigma \Rightarrow bool) \Rightarrow ('a \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow '\sigma \Rightarrow '\sigma$
where
idx-iteratei-aux sz i l c f σ = (
if $i=0 \vee \neg\ c\ \sigma$ *then* σ
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 $\Longrightarrow$ *idx-iteratei-aux sz i l c f* $\sigma = \sigma$
  $\neg c\ \sigma \Longrightarrow$ *idx-iteratei-aux sz i l c f* $\sigma = \sigma$
  $[\![i{\neq}0;\ c\ \sigma]\!] \Longrightarrow$ *idx-iteratei-aux sz i l c f* $\sigma =$ *idx-iteratei-aux sz* (*i* − *1*) *l c f* (*f* (*get l* (*sz*−*i*)) σ)
  apply −
  apply (*subst idx-iteratei-aux.simps*, *simp*)+
  done

definition *idx-iteratei* **where**
  idx-iteratei l c f $\sigma \equiv$ *idx-iteratei-aux* (*size l*) (*size l*) *l c f* σ

lemma *idx-iteratei-correct*:
  shows *idx-iteratei s* = *foldli* (α *s*)
proof −
  {
    fix *n l*
    assume *A*: *Suc n* $\leq$ *length l*
    hence *B*: *length l* − *Suc n* < *length l* **by** *simp*
    from *A* **have** [*simp*]: *Suc* (*length l* − *Suc n*) = *length l* − *n* **by** *simp*
    from *Cons-nth-drop-Suc*[*OF B*, *simplified*] **have**
      drop (*length l* − *Suc n*) *l* = *l*!(*length l* − *Suc n*)#*drop* (*length l* − *n*) *l*
      by *simp*
  } **note** *drop-aux=this*

  {
    fix *s c f* σ *i*
    assume *invar s*   *i*$\leq$*size s*
    hence *idx-iteratei-aux* (*size s*) *i s c f* σ
      = *foldli* (*drop* (*size s* − *i*) (α *s*)) *c f* σ
    proof (*induct i arbitrary*: σ)
      case *0* **with** *size-correct*[*of s*] **show** *?case* **by** *simp*
    next
      case (*Suc n*)
      note [*simp*, *intro*!] = *Suc.prems*(*1*)
      show *?case* **proof** (*cases c* σ)
        case *False* **thus** *?thesis* **by** *simp*
      next
        case [*simp*, *intro*!]: *True*
        show *?thesis* **using** *Suc* **by** (*simp add*: *get-correct size-correct drop-aux*)
      qed
    qed
  } **note** *aux=this*

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

lemmas *idx-iteratei-unfold*[*code-unfold*] = *idx-iteratei-correct*[*symmetric*]

reverse_iteratei (by get, size) **fun** *idx-reverse-iteratei-aux*
:: $nat \Rightarrow nat \Rightarrow 's \Rightarrow ('\sigma \Rightarrow bool) \Rightarrow ('a \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow '\sigma \Rightarrow '\sigma$
where
idx-reverse-iteratei-aux sz i l c f σ = (
if $i=0 \vee \neg\ c\ \sigma$ *then* σ
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*]:
$i=0 \Longrightarrow$ *idx-reverse-iteratei-aux sz i l c f* $\sigma = \sigma$
$\neg c\ \sigma \Longrightarrow$ *idx-reverse-iteratei-aux sz i l c f* $\sigma = \sigma$
$[\![i \neq 0;\ c\ \sigma]\!] \Longrightarrow$ *idx-reverse-iteratei-aux sz i l c f* σ
= *idx-reverse-iteratei-aux sz* $(i - 1)$ *l c f* $(f\ (get\ l\ (i - 1))\ \sigma)$
by (*subst idx-reverse-iteratei-aux.simps*, *simp*)+

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

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

show *?thesis*
unfolding *idx-reverse-iteratei-def*[*abs-def*]
apply (*simp add*: *fun-eq-iff aux*[*of - size s*, *simplified*])

```
    apply (simp add: size-correct)
   done
  qed

  lemmas idx-reverse-iteratei-unfold[code-unfold]
   = idx-reverse-iteratei-correct[symmetric]

end
```

Size (by iterator)

```
locale it-size-loc = poly-list-iteratei +
  constrains α :: 's ⇒ 'a list
begin

  definition it-size :: 's ⇒ nat
    where it-size l == iterate l (λx res. Suc res) (0::nat)

  lemma it-size-impl: shows list-size α invar it-size
    apply (unfold-locales)
    apply (unfold it-size-def)
    apply (simp add: iterate-correct foldli-foldl)
    done
end
```

Size (by reverse_iterator) **locale** *rev-it-size-loc* = *poly-list-rev-iteratei* +

```
  constrains α :: 's ⇒ 'a list
begin

  definition rev-it-size :: 's ⇒ nat
    where rev-it-size l == rev-iterate l (λx res. Suc res) (0::nat)

  lemma rev-it-size-impl:
    shows list-size α invar rev-it-size
    apply (unfold-locales)
    apply (unfold rev-it-size-def)
    apply (simp add: rev-iterate-correct foldri-foldr)
    done

end
```

Get (by iteratori)

```
locale it-get-loc = poly-list-iteratei +
  constrains α :: 's ⇒ 'a list
begin

  definition it-get:: 's ⇒ nat ⇒ 'a
    where it-get s i ≡
      the (snd (iteratei s
```

```
        (λ(i,x). x=None)
        (λx (i,-). if i=0 then (0,Some x) else (i − 1,None))
        (i,None)))

lemma it-get-correct:
  shows list-get α invar it-get
proof
  fix s i
  assume invar s   i < length (α s)

  define l where l = α s
  from ‹i < length (α s)›
  show it-get s i = α s ! i
    unfolding it-get-def iteratei-correct l-def[symmetric]
  proof (induct i arbitrary: l)
    case 0
    then obtain x xs where l-eq[simp]: l = x # xs by (cases l, auto)
    thus ?case by simp
  next
    case (Suc i)
    note ind-hyp = Suc(1)
    note Suc-i-le = Suc(2)
    from Suc-i-le obtain x xs
      where l-eq[simp]: l = x # xs by (cases l, auto)

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

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

lemma *indexI*: ⟦ $x \in s$; $f\ x = i$ ⟧ $\Longrightarrow$ $x \in index\ f\ s\ i$ **by** (*unfold index-def*) *auto*
lemma *indexD*:
  $x \in index\ f\ s\ i \Longrightarrow x \in s$
  $x \in index\ f\ s\ i \Longrightarrow f\ x = i$
  by (*unfold index-def*) *auto*

lemma *index-iff* [*simp*]: $x \in index\ f\ s\ i \longleftrightarrow x \in s \land f\ x = i$ **by** (*simp add*: *index-def*)

Indexing by Map

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

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

lemma *index-map-correct*: *im-α* (*index-map f s*) = *index f s*
  apply (*rule ext*)
  apply (*unfold index-def index-map-def im-α-def*)
  apply *auto*
  done

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

  definition *is-index* :: $('x \Rightarrow 'i) \Rightarrow 'x\ set \Rightarrow 'm \Rightarrow bool$
  where

```
    is-index f s idx == ci-invar idx ∧ ci-α' idx = index-map f s

  lemma is-index-invar: is-index f s idx ⟹ ci-invar idx
    by (simp add: is-index-def)

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

  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)
    apply (unfold ci-invar-def lookup-def)
    apply (auto split: option.split simp add: m.lookup-correct s.empty-correct)
    done

  lemma lookup-correct:
    assumes I[simp, intro!]: is-index f s idx
    shows
      s.α (lookup i idx) = index f s i
      s.invar (lookup i idx)
  proof goal-cases
    case 2 thus ?case using I by (simp add: is-index-def lookup-invar')
  next
    case 1
    have [simp, intro!]: m.invar idx
      using ci-impl-minvar[OF is-index-invar[OF I]]
      by simp
    thus ?case
    proof (cases m.lookup i idx)
      case None
      hence [simp]: m.α idx i = None by (simp add: m.lookup-correct)
      from is-index-correct[OF I] have index f s i = ci-α idx i by simp
      also have ... = {} by (simp add: ci-α-def ci-α'-def im-α-def)
      finally show ?thesis by (simp add: lookup-def None s.empty-correct)
    next
      case (Some si)
      hence [simp]: m.α idx i = Some si by (simp add: m.lookup-correct)
      from is-index-correct[OF I] have index f s i = ci-α idx i by simp
      also have ... = s.α si by (simp add: ci-α-def ci-α'-def im-α-def)
      finally show ?thesis by (simp add: lookup-def Some s.empty-correct)
    qed
  qed

end

locale build-index-loc = index-loc m-ops s-ops +
  t: StdSet t-ops
```

for *m-ops* :: ('*i*,'*s*,'*m*,'*more1*) *map-ops-scheme*
and *s-ops* :: ('*x*,'*s*,'*more3*) *set-ops-scheme*
and *t-ops* :: ('*x*,'*t*,'*more2*) *set-ops-scheme*
begin

Building indices

```
definition idx-build-stepfun :: ('x ⇒ 'i) ⇒ 'x ⇒ 'm ⇒ 'm where
  idx-build-stepfun f x m ==
    let i=f x in
      (case m.lookup i m of
        None ⇒ m.update i (s.ins x (s.empty ())) m |
        Some s ⇒ m.update i (s.ins x s) m
    )

definition idx-build :: ('x ⇒ 'i) ⇒ 't ⇒ 'm where
  idx-build f t == t.iterate t (idx-build-stepfun f) (m.empty ())

lemma idx-build-correct:
  assumes I: t.invar t
  shows ci-α' (idx-build f t) = index-map f (t.α t) (is ?T1) and
  [simp]: ci-invar (idx-build f t) (is ?T2)
proof −
  have t.invar t ⟹
    ci-α' (idx-build f t) = index-map f (t.α t) ∧ ci-invar (idx-build f t)
    apply (unfold idx-build-def)
    apply (rule-tac
        I=λit m. ci-α' m = index-map f (t.α t − it) ∧ ci-invar m
        in t.iterate-rule-P)
    apply assumption
    apply (simp add: ci-invar-def m.empty-correct)
    apply (rule ext)
    apply (unfold ci-α'-def index-map-def index-def)[1]
    apply (simp add: m.empty-correct)
    defer
    apply simp
    apply (rule conjI)
    defer
    apply (unfold idx-build-stepfun-def)[1]
    apply (auto
      simp add: ci-invar-def m.update-correct m.lookup-correct
              s.empty-correct s.ins-correct Let-def
      split: option.split) [1]

    apply (rule ext)
  proof goal-cases
    case prems: (1 x it m i)
    hence INV[simp, intro!]: m.invar m by (simp add: ci-invar-def)
    from prems have
      INVS[simp, intro]: !!q s. m.α m q = Some s ⟹ s.invar s
```

```
    by (simp add: ci-invar-def)

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

```
        qed
      next
        case False hence C: i≠f x   f x≠i by simp-all
        have ci-α' (idx-build-stepfun f x m) i = ci-α' m i
          apply (unfold ci-α'-def idx-build-stepfun-def)
          apply (simp
            split: option.split-asm option.split
            add: Let-def m.lookup-correct m.update-correct
                 s.ins-correct s.empty-correct C)
          done
        also from prems(4) have ci-α' m i = index-map f (t.α t − it) i
          by simp
        also have
          {xa ∈ t.α t − (it − {x}). f xa = i} = {xa ∈ t.α t − it. f xa = i}
          using prems(2,3) C by auto
        hence index-map f (t.α t − it) i = index-map f (t.α t − (it−{x})) i
          by (unfold index-map-def index-def) simp
        finally show ?thesis .
      qed
    qed
    with I show ?T1 ?T2 by auto
  qed

  lemma idx-build-is-index:
    t.invar t ⟹ is-index f (t.α t) (idx-build f t)
    by (simp add: idx-build-correct index-map-correct ci-α-def is-index-def)

end

end
```

3.2.7 More Generic Algorithms

```
theory Algos
imports
  ../spec/SetSpec
  ../spec/MapSpec
  ../spec/ListSpec
begin
```

Injective Map to Naturals

Whether a set is an initial segment of the natural numbers

```
definition inatseg :: nat set ⇒ bool
  where inatseg s == ∃ k. s = {i::nat. i<k}

lemma inatseg-simps[simp]:
  inatseg {}
  inatseg {0}
```

```
  by (unfold inatseg-def)
    auto
```

Compute an injective map from objects into an initial segment of the natural numbers

```
locale map-to-nat-loc =
  s: StdSet s-ops +
  m: StdMap m-ops
  for s-ops :: ('x,'s,'more1) set-ops-scheme
  and m-ops :: ('x,nat,'m,'more2) map-ops-scheme
begin

  definition map-to-nat
    :: 's ⇒ 'm where
    map-to-nat s ==
      snd (s.iterate s (λx (c,m). (c+1,m.update x c m)) (0,m.empty ()))

  lemma map-to-nat-correct:
    assumes INV[simp]: s.invar s
    shows
      — All elements have got a number
      dom (m.α (map-to-nat s)) = s.α s (is ?T1) and
      — No two elements got the same number
      [rule-format]: inj-on (m.α (map-to-nat s)) (s.α s) (is ?T2) and
      — Numbering is inatseg
      [rule-format]: inatseg (ran (m.α (map-to-nat s))) (is ?T3) and
      — The result satisfies the map invariant
      m.invar (map-to-nat s) (is ?T4)
    proof −
      have i-aux: !!m S S' k v. ⟦inj-on m S; S' = insert k S; v∉ran m⟧
                              ⟹ inj-on (m(k↦v)) S'
        apply (rule inj-onI)
        apply (simp split: if-split-asm)
        apply (simp add: ran-def)
        apply (simp add: ran-def)
        apply blast
        apply (blast dest: inj-onD)
        done

      have ?T1 ∧ ?T2 ∧ ?T3 ∧ ?T4
        apply (unfold map-to-nat-def)
        apply (rule-tac I=λit (c,m).
          m.invar m ∧
          dom (m.α m) = s.α s − it ∧
          inj-on (m.α m) (s.α s − it) ∧
          (ran (m.α m) = {i. i<c})
           in s.iterate-rule-P)
        apply simp
        apply (simp add: m.empty-correct)
```

```
      apply (case-tac σ)
      apply (simp add: m.empty-correct m.update-correct)
      apply (intro conjI)
      apply blast
      apply clarify
      apply (rule-tac m2=m.α ba and
                k2=x and v2=aa and
                S'2=(s.α s − (it − {x})) and
                S2=(s.α s − it)
                in i-aux)
      apply auto [3]
      apply auto [1]
      apply (case-tac σ)
      apply (auto simp add: inatseg-def)
      done
    thus ?T1 ?T2 ?T3 ?T4 by auto
  qed

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 −
    have m.α (it-dom-fun-to-map s f) k
      = (if k ∈ s.α s then Some (f k) else None) ∧
      m.invar (it-dom-fun-to-map s f)
      unfolding it-dom-fun-to-map-def
      apply (rule s.iterate-rule-P[where
        I = λit res. m.invar res
        ∧ (∀ k. m.α res k = (if (k ∈ (s.α s) − it) then Some (f k) else None))
        ])
```

```
      apply (simp add: INV)

      apply (simp add: m.empty-correct)

      apply (subgoal-tac x∉dom (m.α σ))

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

      apply auto [2]
    done
  thus ?G1 and ?G2
    by auto
qed

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))
    using I apply −
    unfolding g-set-to-listl-def
    apply (rule-tac I=λit σ. l.invar σ ∧ List.set (l.α σ) = it
      ∧ distinct (l.α σ)
      in s.iterate-rule-insert-P, auto simp: l.correct)+
    done

  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* ($l.\alpha$ (*g-set-to-listr s*))
using *I* **apply** −
unfolding *g-set-to-listr-def*
apply (*rule-tac I*=λ*it* σ. *l.invar* σ $\wedge$ *List.set* ($l.\alpha$ σ) = *it*
$\wedge$ *distinct* ($l.\alpha$ σ)
in *s.iterate-rule-insert-P*, *auto simp*: *l.correct*)+
done

end

end

3.2.8 Implementing Priority Queues by Annotated Lists

theory *PrioByAnnotatedList*
imports
../spec/AnnotatedListSpec
../spec/PrioSpec
begin

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

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

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

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

Definitions

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

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

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

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

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

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

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

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

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

lemma *p-less-le-not-le* : *p-less x y* ⟷ *p-less-eq x y* ∧ ¬ (*p-less-eq y x*)
by (*induct x y rule*: *p-less.induct*) *auto*

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

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

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

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

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

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

end

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

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

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

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

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

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

definition *alprio-delete* :: (((′*e*,′*a*::*linorder*) *Prio* ⇒ *bool*)
  ⇒ (′*e*,′*a*) *Prio* ⇒ ′*s* ⇒ (′*s* × (*unit* × (′*e*,′*a*) *Prio*) × ′*s*))
      ⇒ (′*s* ⇒ (′*e*,′*a*) *Prio*) ⇒ (′*s* ⇒ ′*s* ⇒ ′*s*) ⇒ ′*s* ⇒ ′*s*

where
alprio-delete splits annot app s = (*let* (*l*, - , *r*)
= *splits* (λ *x*. *x*$\leq$(*annot s*)) *Infty s in app l r*)

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

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

Correctness

Auxiliary Lemmas **lemma** *sum-list-split*: *sum-list* (*l* @ (*a*::'*a*::*monoid-add*) # *r*) = (*sum-list l*) + *a* + (*sum-list r*)
by (*induct l*) (*auto simp add*: *add.assoc*)

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

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

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

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

apply (*cases a*)
apply *auto*
apply (*cases sum-list ins*)
apply *auto*
apply (*case-tac x*)
apply *auto*
apply (*cases a*)
apply *auto*
apply (*cases sum-list ins*)
apply *auto*
done
qed

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

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

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

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

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

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

by (*auto simp add*: *plus-def*, *cases* (*a*,*b*) *rule*: *p-min.cases*) (*auto simp add*: *plesseq-def*)

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

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

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

Is Empty **lemma** *alprio-isEmpty-correct*:
  assumes *al-isEmpty* α *invar isEmpty*

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

Insert

lemma *alprio-insert-correct*:
assumes *al-consl α invar consl*
shows *prio-insert* (*alprio-α α*) (*alprio-invar α invar*) (*alprio-insert consl*)
proof –
interpret *al-consl α invar consl* **by** *fact*
show *?thesis* **by** *unfold-locales* (*auto simp add*: *alprio-defs consl-correct*)
qed

Meld

lemma *alprio-meld-correct*:
assumes *al-app α invar app*
shows *prio-meld* (*alprio-α α*) (*alprio-invar α invar*) (*alprio-meld app*)
proof –
interpret *al-app α invar app* **by** *fact*
show *?thesis* **by** *unfold-locales* (*auto simp add*: *alprio-defs app-correct*)
qed

Find

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

lemma *annot-in-set*:
  assumes (*alprio-invar* α *invar*) *s*
  and (*alprio-*α α) *s* $\neq$ {#}
  and *al-annot* α *invar annot*
  shows *p-unwrap* (*annot s*) $\in$# ((*alprio-*α α) *s*)
proof –
  interpret *al-annot* α *invar annot* **by** *fact*
  from *assms*(*2*) **have** *snn*: α *s* $\neq$ [] **by** (*auto simp add*: *alprio-defs*)
  from *assms*(*1*) **have** *invs*: *invar s* **by** (*simp add*: *alprio-defs*)
  hence *ans*: *annot s* = *sum-list* (*map snd* (α *s*)) **by** (*simp add*: *annot-correct*)
  let *?P* = *map snd* (α *s*)
  have *annot s* $\in$ *set ?P*
    by (*unfold ans*) (*rule sum-list-in-set*[*OF snn*])
  then show *?thesis*
    by (*auto intro*!: *image-eqI simp add*: *alprio-*α*-def*)
qed

lemma *sum-list-less-elems*: $\forall$ *x*$\in$*set xs*. *snd x* $\neq$ *Infty* $\Longrightarrow$
  $\forall$ *y*$\in$*set-mset* (*mset* (*map p-unwrap* (*map snd xs*))).
    snd (*p-unwrap* (*sum-list* (*map snd xs*))) $\leq$ *snd y*
  proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
  next
  case (*Cons a as*) **thus** *?case*
    apply *auto*
    apply (*cases* (*snd a*) *rule*: *p-unwrap.cases*)
    apply *auto*
    apply (*cases sum-list* (*map snd as*))
    apply *auto*
    apply (*metis linorder-linear p-min-re-neut*
      p-unwrap.simps plus-def [*abs-def*] *snd-eqD*)
    apply (*auto simp add*: *p-unwrap-less-sum*)
    apply (*unfold plus-def*)
    apply (*cases* (*snd a*, *sum-list* (*map snd as*)) *rule*: *p-min.cases*)
    apply *auto*
    apply (*cases map snd as*)
    apply (*auto simp add*: *infadd*)
    done
qed

lemma *alprio-find-correct*:
  assumes *al-annot* α *invar annot*
  shows *prio-find* (*alprio-*α α) (*alprio-invar* α *invar*) (*alprio-find annot*)
proof –
  interpret *al-annot* α *invar annot* **by** *fact*
  show *?thesis*
    apply *unfold-locales*
    apply (*rule conjI*)

```
    apply (insert assms)
    apply (unfold alprio-find-def)
    apply (simp add:annot-in-set)
    apply (unfold alprio-defs)
    apply (simp add: annot-correct)
    apply (auto simp add: sum-list-less-elems)
    done
qed
```

Delete

```
lemma delpred-mon:
 ∀(a:: ('e, 'a::linorder) Prio) b. ((λ x. x ≤ y) a
    ⟶ (λ x. x ≤ y) (a + b))
proof (intro impI allI)
  fix a b
  show a ≤ y ⟹ a + b ≤ y
    apply (induct a b rule: p-less.induct)
    apply (auto simp add: plus-def)
    apply (metis linorder-linear order-trans
      p-linear p-min.simps(4) p-min-mon plus-def prio-selects-one)
    apply (metis order-trans p-linear p-min-mon p-min-re-neut plus-def)
    done
qed
```

```
lemma alpriodel-invar:
  assumes alprio-invar α invar s
  and al-annot α invar annot
  and alprio-α α s ≠ {#}
  and al-splits α invar splits
  and al-app α invar app
  shows alprio-invar α invar (alprio-delete splits annot app s)
proof −
  interpret al-splits α invar splits by fact
  let ?P = λx. x ≤ annot s
  obtain l p r where
    [simp]:splits ?P Infty s = (l, p, r)
    by (cases splits ?P Infty s)  auto
  obtain e a where
    p = (e, a)
    by (cases p, blast)
  hence
    lear:splits ?P Infty s = (l, (e,a), r)
    by simp
  from annot-not-inf[OF assms(1) assms(3) assms(2)] have
    annot s ≠ Infty .
  hence
    sv1: ¬ Infty ≤ annot s
    by (simp add: plesseq-def, cases annot s, auto)
  from assms(1) have
```

invs: *invar s*
unfolding *alprio-invar-def* **by** *simp*
interpret *al-annot* α *invar annot* **by** *fact*
from *invs* **have**
sv2: *Infty* + *sum-list* (*map snd* (α *s*)) $\leq$ *annot s*
by (*auto simp add*: *annot-correct plus-def*
plesseq-def p-min-le-neut p-order-refl)
note *sp* = *splits-correct*[*of s ?P Infty l e a r*]
note *dp* = *delpred-mon*[*of annot s*]
from *sp*[*OF invs dp sv1 sv2 lear*] **have**
invlr: *invar l* $\wedge$ *invar r* **and**
alr: α *s* = α *l* @ (*e*, *a*) # α *r*
by *auto*
interpret *al-app* α *invar app* **by** *fact*
from *invlr app-correct* **have**
invapplr: *invar* (*app l r*)
by *simp*
from *invlr app-correct* **have**
sr: α (*app l r*) = (α *l*) @ (α *r*)
by *simp*
from *alr* **have**
set (α *s*) $\supseteq$ (*set* (α *l*) *Un set* (α *r*))
by *auto*
with *app-correct*[*of l r*] *invlr* **have**
set (α *s*) $\supseteq$ *set* (α (*app l r*)) **by** *auto*
with *invapplr assms*(*1*)
show *?thesis*
unfolding *alprio-defs* **by** *auto*
qed

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

lemma *alpriodel-right*:
assumes *alprio-invar* α *invar s*
and *al-annot* α *invar annot*

and *alprio-α α s ≠ {#}*
and *al-splits α invar splits*
and *al-app α invar app*
shows *alprio-α α (alprio-delete splits annot app s) =*
alprio-α α s − {#p-unwrap (annot s)#}
proof −
interpret *al-splits α invar splits* **by** *fact*
let *?P = λx. x ≤ annot s*
obtain *l p r* **where**
[simp]:splits ?P Infty s = (l, p, r)
by *(cases splits ?P Infty s) auto*
obtain *e a* **where**
p = (e, a)
by *(cases p, blast)*
hence
lear:splits ?P Infty s = (l, (e,a), r)
by *simp*
from *annot-not-inf[OF assms(1) assms(3) assms(2)]* **have**
annot s ≠ Infty **.**
hence
sv1: ¬ Infty ≤ annot s
by *(simp add: plesseq-def, cases annot s, auto)*
from *assms(1)* **have**
invs: invar s
unfolding *alprio-invar-def* **by** *simp*
interpret *al-annot α invar annot* **by** *fact*
from *invs* **have**
sv2: Infty + sum-list (map snd (α s)) ≤ annot s
by *(auto simp add: annot-correct plus-def*
plesseq-def p-min-le-neut p-order-refl)
note *sp = splits-correct[of s ?P Infty l e a r]*
note *dp = delpred-mon[of annot s]*

from *sp[OF invs dp sv1 sv2 lear]* **have**
invlr: invar l ∧ invar r **and**
alr: α s = α l @ (e, a) # α r **and**
anlel: ¬ sum-list (map snd (α l)) ≤ annot s **and**
aneqa: (sum-list (map snd (α l)) + a) ≤ annot s
by *(auto simp add: plus-def zero-def)*
have *mapalr: map snd (α s) = (map snd (α l)) @ a # (map snd (α r))*
using *alr* **by** *simp*
note *lsa = sum-list-elem[of map snd (α s) map snd (α l) a map snd (α r)]*
note *lsa2 = lsa[OF mapalr]*
hence *a-is-annot: a = annot s*
using *annot-correct[OF invs] anlel aneqa* **by** *auto*
have *map p-unwrap (map snd (α s)) =*
(map p-unwrap (map snd (α l))) @ (p-unwrap a)
 # (map p-unwrap (map snd (α r)))
using *alr* **by** *simp*

hence *alpriolst*: (*alprio-α α s*) = (*alprio-α α l*) +{# *p-unwrap a* #}+ (*alprio-α α r*)
  unfolding *alprio-defs*
  by (*simp add*: *algebra-simps*)
interpret *al-app α invar app* **by** *fact*
from *alpriolst* **show** *?thesis* **using** *app-correct*[*of l r*] *invlr a-is-annot*
  by (*auto simp add*: *alprio-defs algebra-simps*)
qed

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

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

locale *alprio-defs* = *StdALDefs ops*
  for *ops* :: (*unit*,(*'e*,*'a*::*linorder*) *Prio*,*'s*) *alist-ops*
begin
  definition [*icf-rec-def*]: *alprio-ops* ≡ (|
    prio-op-α = *alprio-α α*,
    prio-op-invar = *alprio-invar α invar*,
    prio-op-empty = *alprio-empty empty*,
    prio-op-isEmpty = *alprio-isEmpty isEmpty*,
    prio-op-insert = *alprio-insert consl*,
    prio-op-find = *alprio-find annot*,
    prio-op-delete = *alprio-delete splits annot app*,
    prio-op-meld = *alprio-meld app*

```
  ⟩
end

locale alprio = alprio-defs ops + StdAL ops
  for ops :: (unit,('e,'a::linorder) Prio,'s) alist-ops
begin
  lemma alprio-ops-impl: StdPrio alprio-ops
    apply (rule StdPrio.intro)
    apply (simp-all add: icf-rec-unf)
    apply (rule alprio-correct, unfold-locales) []
    apply (rule alprio-correct, unfold-locales) []
    apply (rule alprio-correct, unfold-locales) []
    apply (rule alprio-correct, unfold-locales) []
    apply (rule alprio-correct, unfold-locales) []
    apply (rule alprio-correct, unfold-locales) []
    done
end

end
```

3.2.9 Implementing Unique Priority Queues by Annotated Lists

```
theory PrioUniqueByAnnotatedList
imports
  ../spec/AnnotatedListSpec
  ../spec/PrioUniqueSpec
begin
```

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

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

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

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

The minimum priority can be read out as the sum over the whole sequence. Finding the element with minimum priority is done by splitting the sequence

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

at the point where the minimum priority of the elements read so far becomes equal to the minimum priority of the whole sequence.

Definitions

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

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

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

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

Instantiation of classes

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

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

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

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

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

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

lemma *p-less-le-not-le* : *p-less x y* ⟷ *p-less-eq x y* ∧ ¬ (*p-less-eq y x*)
by (*induct x y rule*: *p-less.induct*) *auto*

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

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

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

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

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

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

end

Operations

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

$\Rightarrow$ *'s* $\Rightarrow$ (*'e*::*linorder* $\rightharpoonup$ *'a*::*linorder*)
where
aluprio-α α ft == (*map-of* (*map p-unwrap* (*map snd* (*α ft*))))

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

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

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

definition *aluprio-insert* ::
(((*'e*::*linorder*,*'a*::*linorder*) *LP* $\Rightarrow$ *bool*)
$\Rightarrow$ (*'e*,*'a*) *LP* $\Rightarrow$ *'s* $\Rightarrow$ (*'s* × (*unit* × (*'e*,*'a*) *LP*) × *'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*) ∧ ¬ *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* × (*unit* × (*'e*,*'a*) *LP*) × *'s*))
$\Rightarrow$ (*'s* $\Rightarrow$ (*'e*,*'a*) *LP*)
$\Rightarrow$ (*'s* $\Rightarrow$ *'s* $\Rightarrow$ *'s*)
$\Rightarrow$ *'s*
$\Rightarrow$ *'e* ×*'a* ×*'s*
where

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

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

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

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

Correctness

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

lemma *e-less-eq-mon1*: *e-less-eq e x* $\Longrightarrow$ *e-less-eq e* (*x* + *y*)
 apply (*cases x*)
 apply (*auto simp add*: *plus-def*)
 apply (*cases y*)
 apply (*auto simp add*: *max.coboundedI1*)
 done

lemma *e-less-eq-mon2*: *e-less-eq e y* $\Longrightarrow$ *e-less-eq e* $(x + y)$
apply (*cases x*)
apply (*auto simp add*: *plus-def*)
apply (*cases y*)
apply (*auto simp add*: *max.coboundedI2*)
done
lemmas *e-less-eq-mon* =
e-less-eq-mon1
e-less-eq-mon2

lemma *p-less-eq-mon*:
$(x::('e::linorder,'a::linorder)\ LP) \leq z \Longrightarrow (x + y) \leq z$
apply(*cases y*)
apply(*auto simp add*: *plus-def*)
apply (*cases x*)
apply (*cases z*)
apply (*auto simp add*: *plesseq-def*)
apply (*cases z*)
apply (*auto simp add*: *min.coboundedI1*)
done

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

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

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

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

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

lemma *e-less-eq-sum-list2*:
  assumes
  $\forall x \in set\ (\alpha s).\ snd\ x \neq Infty$
  $((),\ b) \in set\ (\alpha s)$
  shows *e-less-eq* (*fst* (*p-unwrap b*)) (*sum-list* (*map snd* (αs)))
  apply(*insert assms*)
  apply (*induct* αs)
  apply (*auto simp add*: *zero-def e-less-eq-mon e-less-eq-refl*)
  done

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

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

lemma *sum-list-less-elems*: $\forall x \in set\ xs.\ snd\ x \neq Infty \Longrightarrow$
  $\forall y \in set$ (*map snd* (*map p-unwrap* (*map snd xs*))).
      snd (*p-unwrap* (*sum-list* (*map snd xs*))) $\leq y$
    proof (*induct xs*)
    case *Nil* **thus** *?case* **by** *simp*
    next
    case (*Cons a as*) **thus** *?case*
      apply *auto*
      apply (*cases* (*snd a*) *rule*: *p-unwrap.cases*)
      apply *auto*
      apply (*cases sum-list* (*map snd as*))
      apply *auto*
      apply (*metis linorder-linear p-min-re-neut p-unwrap.simps*
        plus-def[*abs-def*] *snd-eqD*)
      apply (*auto simp add*: *p-unwrap-less-sum*)

apply (*unfold plus-def*)
apply (*cases* (*snd a*, *sum-list* (*map snd as*)) *rule*: *p-min.cases*)
apply *auto*
apply (*cases map snd as*)
apply (*auto simp add*: *infadd*)
apply (*metis min.coboundedI2 snd-conv*)
done
qed

lemma *distinct-sortet-list-app*:
$[\![$*sorted xs*; *distinct xs*; $xs = as$ @ b # $cs]\!]$
$\Longrightarrow \forall\ x\in$ *set cs*. $b < x$
by (*metis distinct.simps*(2) *distinct-append*
antisym-conv2 sorted-wrt.simps(2) *sorted-append*)

lemma *distinct-sorted-list-lem1*:
assumes
sorted xs
sorted ys
distinct xs
distinct ys
$\forall\ x \in$ *set xs*. $x < e$
$\forall\ y \in$ *set ys*. $e < y$
shows
sorted (*xs* @ *e* # *ys*)
distinct (*xs* @ *e* # *ys*)
proof −
from *assms* (5,6)
have $\forall x\in$*set xs*. $\forall y\in$*set ys*. $x \le y$ **by** *force*
thus *sorted* (*xs* @ *e* # *ys*)
using *assms*
by (*auto simp add*: *sorted-append*)
have *set xs* $\cap$ *set ys* $= \{\}$ **using** *assms* (5,6) **by** *force*
thus *distinct* (*xs* @ *e* # *ys*)
using *assms*
by (*auto*)
qed

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

distinct (*xs* @ *e* # *e'* # *ys*)
proof −
have *sorted* (*e'* # *ys*)
distinct (*e'* # *ys*)
∀ *y* ∈ *set* (*e'* # *ys*). *e* < *y*
using *assms*(*2*,*4*,*5*,*7*)
by (*auto*)
thus *sorted* (*xs* @ *e* # *e'* # *ys*)
distinct (*xs* @ *e* # *e'* # *ys*)
using *assms*(*1*,*3*,*6*) *distinct-sorted-list-lem1*[*of xs e'* # *ys e*]
by *auto*
qed

lemma *map-of-distinct-upd*:
x ∉ *set* (*map fst xs*) ⟹ [*x* ↦ *y*] ++ *map-of xs* = (*map-of xs*) (*x* ↦ *y*)
by (*induct xs*) (*auto simp add*: *fun-upd-twist*)

lemma *map-of-distinct-upd2*:
assumes *x* ∉ *set*(*map fst xs*)
x ∉ *set* (*map fst ys*)
shows *map-of* (*xs* @ (*x*,*y*) # *ys*) = (*map-of* (*xs* @ *ys*))(*x* ↦ *y*)
apply(*insert assms*)
apply(*induct xs*)
apply (*auto intro*: *ext*)
done

lemma *map-of-distinct-upd3*:
assumes *x* ∉ *set*(*map fst xs*)
x ∉ *set* (*map fst ys*)
shows *map-of* (*xs* @ (*x*,*y*) # *ys*) = (*map-of* (*xs* @ (*x*,*y'*) # *ys*))(*x* ↦ *y*)
apply(*insert assms*)
apply(*induct xs*)
apply (*auto intro*: *ext*)
done

lemma *map-of-distinct-upd4*:
assumes *x* ∉ *set*(*map fst xs*)
x ∉ *set* (*map fst ys*)
shows *map-of* (*xs* @ *ys*) = (*map-of* (*xs* @ (*x*,*y*) # *ys*))(*x* := *None*)
apply(*insert assms*)
apply(*induct xs*)

apply *clarsimp*
apply (*metis dom-map-of-conv-image-fst fun-upd-None-restrict*
restrict-complement-singleton-eq restrict-map-self)

apply (*auto simp add*: *map-of-eq-None-iff*) []
done

lemma *map-of-distinct-lookup*:
  assumes $x \notin$ *set*(*map fst xs*)
  $x \notin$ *set* (*map fst ys*)
  shows *map-of* (*xs* @ (*x*,*y*) # *ys*) *x* = *Some y*
proof −
  have *map-of* (*xs* @ (*x*,*y*) # *ys*) = (*map-of* (*xs* @ *ys*)) ($x \mapsto y$)
    using *assms map-of-distinct-upd2* **by** *simp*
  thus *?thesis*
    by *simp*
qed

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

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

Empty **lemma** *aluprio-empty-correct*:
  assumes *al-empty α invar empt*
  shows *uprio-empty* (*aluprio-α α*) (*aluprio-invar α invar*) (*aluprio-empty empt*)
proof −
  interpret *al-empty α invar empt* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*auto simp add*: *empty-correct aluprio-defs*)
    done
qed

Is Empty **lemma** *aluprio-isEmpty-correct*:
  assumes *al-isEmpty α invar isEmpty*
  shows *uprio-isEmpty* (*aluprio-α α*) (*aluprio-invar α invar*) (*aluprio-isEmpty isEmpty*)
proof −
  interpret *al-isEmpty α invar isEmpty* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    apply (*auto simp add*: *aluprio-defs isEmpty-correct*)
    done

qed

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

lemma *e-less-eq-annot*:

  assumes *al-annot* α *invar annot*
   invar s $\forall x \in set\ (\alpha\ s).\ snd\ x \neq Infty$ $\neg$ *e-less-eq e* (*annot s*)
  shows $\forall x \in set\ (map\ (fst \circ (p\text{-}unwrap \circ snd))\ (\alpha\ s)).\ x < e$
proof −
  interpret *al-annot* α *invar annot* **by** *fact*
  from *assms*(*2*) **have** *annot s* = *sum-list* (*map snd* (α *s*))
    by (*auto simp add*: *annot-correct*)
  with *assms*(*4*) **have**
    $\forall x \in set\ (map\ snd\ (\alpha\ s)).\ \neg$ *e-less-eq e x*
    by (*metis e-less-eq-sum-list*)
  with *assms*(*3*)
  show *?thesis*
    by (*auto simp add*: *e-less-eq-p-unwrap*)
qed

lemma *aluprio-insert-correct*:
  assumes
  al-splits α *invar splits*
  al-annot α *invar annot*
  al-isEmpty α *invar isEmpty*
  al-app α *invar app*

```
  al-consr α invar consr
  shows
  uprio-insert (aluprio-α α) (aluprio-invar α invar)
    (aluprio-insert splits annot isEmpty app consr)
proof −
  interpret al-splits α invar splits by fact
  interpret al-annot α invar annot by fact
  interpret al-isEmpty α invar isEmpty by fact
  interpret al-app α invar app by fact
  interpret al-consr α invar consr by fact
  show ?thesis
  proof (unfold-locales, unfold aluprio-defs, goal-cases)
    case g1asms: (1 s e a)
    thus ?case proof (cases e-less-eq e (annot s) ∧ ¬ isEmpty s)
      case False with g1asms show  ?thesis
        apply (auto simp add: consr-correct )
      proof goal-cases
        case prems: 1
        with assms(2) have
          ∀ x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α s)). x < e
          by (simp add: e-less-eq-annot)
        with prems(3) show ?case
          by(auto simp add: sorted-append)
      next
        case prems: 2
        hence annot s = sum-list (map snd (α s))
          by (simp add: annot-correct)
        with prems
        show ?case
          by (auto simp add: e-less-eq-sum-list2)
      next
        case prems: 3
        hence α s = [] by (auto simp add: isEmpty-correct)
        thus ?case by simp
      next
        case prems: 4
        hence α s = [] by (auto simp add: isEmpty-correct)
        with prems show ?case by simp
      qed
    next
      case True note T1 = this
      obtain l uu lp r where
        l-lp-r: (splits (e-less-eq e) Infty s) = (l, ((), lp), r)
        by (cases splits (e-less-eq e) Infty s, auto)
      note v2 = splits-correct[of s e-less-eq e Infty l   () lp r]
      have
        v3: invar s
        ¬ e-less-eq e Infty
        e-less-eq e (Infty + sum-list (map snd (α s)))
```

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

from *l-lp-r T1 g1asms* **show** *?thesis*
proof (*fold aluprio-insert-def*, *cases e* < *fst* (*p-unwrap lp*))
case *True*
hence *v11*:
aluprio-insert splits annot isEmpty app consr s e a
= *app* (*consr* (*consr l* () (*LP e a*)) () *lp*) *r*
using *l-lp-r T1*

```
      by (auto simp add: aluprio-defs)
    have v12: invar (app (consr (consr l () (LP e a)) () lp) r)
      using v4(4,5)
      by (auto simp add: app-correct consr-correct)
    have v13:
      α (app (consr (consr l () (LP e a)) () lp) r)
        = α l @ ((),(LP e a)) # ((), lp) # α r
      using v4(4,5) by (auto simp add: app-correct consr-correct)
    hence v14:
      (∀ x∈set (α (app (consr (consr l () (LP e a)) () lp) r)).
        snd x ≠ Infty)
      using g1asms v4(1)
      by auto
    have v15: e = fst(p-unwrap (LP e a)) by simp
    hence v16:
      sorted (map fst (map p-unwrap
        (map snd (α l @ ((),(LP e a)) # ((), lp) # α r))))
      distinct (map fst (map p-unwrap
        (map snd (α l @ ((),(LP e a)) # ((), lp) # α r))))
      using v10(1,3) v7 True v9 v4(1) g1asms distinct-sorted-list-lem2
      by (auto simp add: sorted-append)
    thus invar (aluprio-insert splits annot isEmpty app consr s e a) ∧
      (∀ x∈set (α (aluprio-insert splits annot isEmpty app consr s e a)).
        snd x ≠ Infty) ∧
      sorted (map fst (map p-unwrap (map snd (α
        (aluprio-insert splits annot isEmpty app consr s e a))))) ∧
      distinct (map fst (map p-unwrap (map snd (α
        (aluprio-insert splits annot isEmpty app consr s e a)))))
      using v11 v12 v13 v14
      by simp
  next
    case False
    hence v11:
      aluprio-insert splits annot isEmpty app consr s e a
        = app (consr l () (LP e a)) r
      using l-lp-r T1
      by (auto simp add: aluprio-defs)
    have v12: invar (app (consr l () (LP e a)) r) using v4(4,5)
      by (auto simp add: app-correct consr-correct)
    have v13: α (app (consr l () (LP e a)) r) = α l @ ((),(LP e a)) # α r
      using v4(4,5) by (auto simp add: app-correct consr-correct)
    hence v14: (∀ x∈set (α (app (consr l () (LP e a)) r)). snd x ≠ Infty)
      using g1asms v4(1)
      by auto
    have v15: e = fst(p-unwrap (LP e a)) by simp
    have v16: e = fst(p-unwrap lp)
      using False v5 by (cases lp) auto
    hence v17:
      sorted (map fst (map p-unwrap
```

```
          (map snd (α l @ ((),(LP e a)) # α r))))
        distinct (map fst (map p-unwrap
          (map snd (α l @ ((),(LP e a)) # α r))))
        using v16 v15 v10(1,3) v7 True v9 v4(1)
          g1asms distinct-sorted-list-lem1
        by (auto simp add: sorted-append)
      thus invar (aluprio-insert splits annot isEmpty app consr s e a) ∧
        (∀ x∈set (α (aluprio-insert splits annot isEmpty app consr s e a)).
          snd x ≠ Infty) ∧
        sorted (map fst (map p-unwrap (map snd (α
          (aluprio-insert splits annot isEmpty app consr s e a))))) ∧
        distinct (map fst (map p-unwrap (map snd (α
          (aluprio-insert splits annot isEmpty app consr s e a)))))
        using v11 v12 v13 v14
        by simp
    qed
  qed
next
  case g1asms: (2 s e a)
  thus ?case proof (cases e-less-eq e (annot s) ∧ ¬ isEmpty s)
    case False with g1asms show  ?thesis
      apply (auto simp add: consr-correct)
    proof goal-cases
      case prems: 1
      with assms(2) have
        ∀ x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α s)). x < e
        by (simp add: e-less-eq-annot)
      hence e ∉ set (map fst ((map (p-unwrap ∘ snd)) (α s)))
        by auto
      thus ?case
        by (auto simp add: map-of-distinct-upd)
    next
      case prems: 2
      hence α s = [] by (auto simp add: isEmpty-correct)
      thus ?case
        by simp
    qed
  next
    case True note T1 = this
    obtain l lp r where
      l-lp-r: (splits (e-less-eq e) Infty s) = (l, ((), lp), r)
      by (cases splits (e-less-eq e) Infty s, auto)
    note v2 = splits-correct[of s e-less-eq e Infty l   () lp r]
    have
      v3: invar s
      ¬ e-less-eq e Infty
      e-less-eq e (Infty + sum-list (map snd (α s)))
      using T1 g1asms annot-correct
      by (auto simp add: plus-def)
```

```
have
  v4: α s = α l @ ((), lp) # α r
  ¬ e-less-eq e (Infty + sum-list (map snd (α l)))
  e-less-eq e (Infty + sum-list (map snd (α l)) + lp)
  invar l
  invar r
  using v2[OF v3(1) - v3(2) v3(3) l-lp-r] e-less-eq-mon(1) by auto
hence v5: e-less-eq e lp
  by (metis e-less-eq-lem1)
hence v6: e ≤ (fst (p-unwrap lp))
  by (cases lp) auto
have (Infty + sum-list (map snd (α l))) = (annot l)
  by (metis add-0-left annot-correct v4(4) zero-def)
hence v7:¬ e-less-eq e (annot l)
  using v4(2) by simp
have ∀x∈set (α l). snd x ≠ Infty
  using g1asms v4(1) by simp
hence v7: ∀x ∈ set (map (fst ∘ (p-unwrap ∘ snd)) (α l)). x < e
  using v4(4) v7 assms(2)
  by(simp add: e-less-eq-annot)
have v8:map fst (map p-unwrap (map snd (α s))) =
  map fst (map p-unwrap (map snd (α l))) @ fst(p-unwrap lp) #
  map fst (map p-unwrap (map snd (α r)))
  using v4(1)
  by simp
note distinct-sortet-list-app[of map fst (map p-unwrap (map snd (α s)))
  map fst (map p-unwrap (map snd (α l)))    fst(p-unwrap lp)
  map fst (map p-unwrap (map snd (α r)))]
hence v9:
  ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). fst(p-unwrap lp) < x
  using v4(1) g1asms v8
  by auto
hence v10: ∀ x∈set (map (fst ∘ (p-unwrap ∘ snd)) (α r)). e < x
  using v6 by auto
have v11:
  e ∉ set (map fst (map p-unwrap (map snd (α l))))
  e ∉ set (map fst (map p-unwrap (map snd (α r))))
  using v7 v10 v8 g1asms
  by auto
from l-lp-r T1 g1asms show ?thesis
proof (fold aluprio-insert-def, cases e < fst (p-unwrap lp))
  case True
  hence v12:
    aluprio-insert splits annot isEmpty app consr s e a
      = app (consr (consr l () (LP e a)) () lp) r
    using l-lp-r T1
    by (auto simp add: aluprio-defs)
  have v13:
    α (app (consr (consr l () (LP e a)) () lp) r)
```

```
        = α l @ ((),(LP e a)) # ((), lp) # α r
        using v4(4,5) by (auto simp add: app-correct consr-correct)
      have v14: e = fst(p-unwrap (LP e a)) by simp
      have v15: e ∉ set (map fst (map p-unwrap (map snd(((),lp)#α r))))
        using v11(2) True by auto
      note map-of-distinct-upd2[OF v11(1) v15]
      thus
        map-of (map p-unwrap (map snd (α
            (aluprio-insert splits annot isEmpty app consr s e a))))
          = (map-of (map p-unwrap (map snd (α s))))(e ↦ a)
        using v12 v13 v4(1)
        by simp
    next
      case False
      hence v12:
        aluprio-insert splits annot isEmpty app consr s e a
          = app (consr l () (LP e a)) r
        using l-lp-r T1
        by (auto simp add: aluprio-defs)
      have v13:
        α (app (consr l () (LP e a)) r) = α l @ ((),(LP e a)) # α r
        using v4(4,5) by (auto simp add: app-correct consr-correct)
      have v14: e = fst(p-unwrap lp)
        using False v5 by (cases lp) auto
      note v15 = map-of-distinct-upd3[OF v11(1) v11(2)]
      have v16:(map p-unwrap (map snd (α s))) =
        (map p-unwrap (map snd (α l))) @ (e,snd(p-unwrap lp)) #
        (map p-unwrap (map snd (α r)))
        using v4(1) v14
        by simp
      note v15[of a snd(p-unwrap lp)]
      thus
        map-of (map p-unwrap (map snd (α
            (aluprio-insert splits annot isEmpty app consr s e a))))
          = (map-of (map p-unwrap (map snd (α s))))(e ↦ a)
        using v12 v13 v16
        by simp
    qed
  qed
 qed
qed
```

Prio **lemma** *aluprio-prio-correct*:

```
 assumes
  al-splits α invar splits
  al-annot α invar annot
  al-isEmpty α invar isEmpty
 shows
  uprio-prio (aluprio-α α) (aluprio-invar α invar) (aluprio-prio splits annot isEmpty)
```

proof −
  interpret *al-splits* α *invar splits* **by** *fact*
  interpret *al-annot* α *invar annot* **by** *fact*
  interpret *al-isEmpty* α *invar isEmpty* **by** *fact*
  show *?thesis*
  proof (*unfold-locales*)
    fix *s e*
    assume *inv1*: *aluprio-invar* α *invar s*
    hence *sinv*: *invar s*
      ($\forall$ $x \in set$ (α *s*). *snd* $x \neq Infty$)
      sorted (*map fst* (*map p-unwrap* (*map snd* (α *s*))))
      distinct (*map fst* (*map p-unwrap* (*map snd* (α *s*))))
      by (*auto simp add*: *aluprio-defs*)
    show *aluprio-prio splits annot isEmpty s e* = *aluprio-*α α *s e*
    proof(*cases e-less-eq e* (*annot s*) $\wedge$ $\neg$ *isEmpty s*)
      case *False* **note** *F1* = *this*
      thus *?thesis*
      proof(*cases isEmpty s*)
        case *True*
        hence α *s* = []
          using *sinv isEmpty-correct* **by** *simp*
        hence *aluprio-*α α *s* = *Map.empty* **by** (*simp add*:*aluprio-defs*)
        hence *aluprio-*α α *s e* = *None* **by** *simp*
        thus *aluprio-prio splits annot isEmpty s e* = *aluprio-*α α *s e*
          using *F1*
          by (*auto simp add*: *aluprio-defs*)
      next
        case *False*
        hence *v3*:$\neg$ *e-less-eq e* (*annot s*) **using** *F1* **by** *simp*
        note *v4*=*e-less-eq-annot*[*OF assms*(*2*)]
        note *v4*[*OF sinv*(*1*) *sinv*(*2*) *v3*]
        hence *v5*:$e \notin set$ (*map* (*fst* $\circ$ (*p-unwrap* $\circ$ *snd*)) (α *s*))
          by *auto*
        hence *map-of* (*map* (*p-unwrap* $\circ$ *snd*) (α *s*)) *e* = *None*
          using *map-of-eq-None-iff*
          by (*metis map-map map-of-eq-None-iff set-map v5*)
        thus *aluprio-prio splits annot isEmpty s e* = *aluprio-*α α *s e*
          using *F1*
          by (*auto simp add*: *aluprio-defs*)
      qed
    next
      case *True* **note** *T1* = *this*
      obtain *l uu lp r* **where**
        l-lp-r: (*splits* (*e-less-eq e*) *Infty s*) = (*l*, ((), *lp*), *r*)
        by (*cases splits* (*e-less-eq e*) *Infty s*, *auto*)
      note *v2* = *splits-correct*[*of s e-less-eq e Infty l* () *lp r*]
      have
        v3: *invar s*
        $\neg$ *e-less-eq e Infty*

e-less-eq e (*Infty* + *sum-list* (*map snd* (α *s*)))
using *T1 sinv annot-correct*
by (*auto simp add*: *plus-def*)
have
v4: α *s* = α *l* @ ((), *lp*) # α *r*
¬ *e-less-eq e* (*Infty* + *sum-list* (*map snd* (α *l*)))
e-less-eq e (*Infty* + *sum-list* (*map snd* (α *l*)) + *lp*)
invar l
invar r
using *v2*[*OF v3*(*1*) - *v3*(*2*) *v3*(*3*) *l-lp-r*] *e-less-eq-mon*(*1*) **by** *auto*
hence *v5*: *e-less-eq e lp*
by (*metis e-less-eq-lem1*)
hence *v6*: *e* ≤ (*fst* (*p-unwrap lp*))
by (*cases lp*) *auto*
have (*Infty* + *sum-list* (*map snd* (α *l*))) = (*annot l*)
by (*metis add-0-left annot-correct v4*(*4*) *zero-def*)
hence *v7*:¬ *e-less-eq e* (*annot l*)
using *v4*(*2*) **by** *simp*
have ∀ *x*∈*set* (α *l*). *snd x* ≠ *Infty*
using *sinv v4*(*1*) **by** *simp*
hence *v7*: ∀ *x* ∈ *set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (α *l*)). *x* < *e*
using *v4*(*4*) *v7 assms*(*2*)
by(*simp add*: *e-less-eq-annot*)
have *v8*:*map fst* (*map p-unwrap* (*map snd* (α *s*))) =
map fst (*map p-unwrap* (*map snd* (α *l*))) @ *fst*(*p-unwrap lp*) #
map fst (*map p-unwrap* (*map snd* (α *r*)))
using *v4*(*1*)
by *simp*
note *distinct-sortet-list-app*[*of map fst* (*map p-unwrap* (*map snd* (α *s*)))
map fst (*map p-unwrap* (*map snd* (α *l*))) *fst*(*p-unwrap lp*)
map fst (*map p-unwrap* (*map snd* (α *r*)))]
hence *v9*:
∀ *x*∈*set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (α *r*)). *fst*(*p-unwrap lp*) < *x*
using *v4*(*1*) *sinv v8*
by *auto*
hence *v10*: ∀ *x*∈*set* (*map* (*fst* ∘ (*p-unwrap* ∘ *snd*)) (α *r*)). *e* < *x*
using *v6* **by** *auto*
have *v11*:
e ∉ *set* (*map fst* (*map p-unwrap* (*map snd* (α *l*))))
e ∉ *set* (*map fst* (*map p-unwrap* (*map snd* (α *r*))))
using *v7 v10 v8 sinv*
by *auto*
from *l-lp-r T1 sinv* **show** *?thesis*
proof (*cases e* = *fst* (*p-unwrap lp*))
case *False*
have *v12*: *e* ∉ *set* (*map fst* (*map p-unwrap* (*map snd*(α *s*))))
using *v11 False v4*(*1*) **by** *auto*
hence *map-of* (*map* (*p-unwrap* ∘ *snd*) (α *s*)) *e* = *None*
using *map-of-eq-None-iff*

```
        by (metis map-map map-of-eq-None-iff set-map v12)
      thus ?thesis
        using T1 False l-lp-r
        by (auto simp add: aluprio-defs)
    next
      case True
      have v12: map (p-unwrap ∘ snd) (α s) =
        map p-unwrap (map snd (α l)) @ (e,snd (p-unwrap lp)) #
        map p-unwrap (map snd (α r))
        using v4(1) True by simp
      note map-of-distinct-lookup[OF v11]
      hence
        map-of (map (p-unwrap ∘ snd) (α s)) e = Some (snd (p-unwrap lp))
        using v12 by simp
      thus ?thesis
        using T1 True l-lp-r
        by (auto simp add: aluprio-defs)
    qed
  qed
 qed
qed
```

Pop **lemma** *aluprio-pop-correct*:

```
 assumes al-splits α invar splits
 al-annot α invar annot
 al-app α invar app
 shows
 uprio-pop (aluprio-α α) (aluprio-invar α invar) (aluprio-pop splits annot app)
proof −
 interpret al-splits α invar splits by fact
 interpret al-annot α invar annot by fact
 interpret al-app α invar app by fact
 show ?thesis
 proof (unfold-locales)
   fix s e a s'
   assume A: aluprio-invar α invar s
     aluprio-α α s ≠ Map.empty
     aluprio-pop splits annot app s = (e, a, s')
   hence v1: α s ≠ []
     by (auto simp add: aluprio-defs)
   obtain l lp r where
     l-lp-r: splits (λ x. x≤annot s) Infty s = (l,((),lp),r)
     by (cases splits (λ x. x≤annot s) Infty s, auto)
   have invs:
     invar s
     (∀ x∈set (α s). snd x ≠ Infty)
     sorted (map fst (map p-unwrap (map snd (α s))))
     distinct (map fst (map p-unwrap (map snd (α s))))
     using A by (auto simp add:aluprio-defs)
```

```
note a1 = annot-inf[of invar s α annot]
note a1[OF invs(1) invs(2) assms(2)]
hence v2: annot s ≠ Infty
  using v1 by simp
hence v3:
  ¬ Infty ≤ annot s
  by(cases annot s) (auto simp add: plesseq-def)
have v4: annot s = sum-list (map snd (α s))
  by (auto simp add: annot-correct invs(1))
hence
  v5:
  (Infty + sum-list (map snd (α s))) ≤ annot s
  by (auto simp add: plus-def)
note p-mon = p-less-eq-mon[of - annot s]
note v6 = splits-correct[OF invs(1)]
note v7 = v6[of λ x. x ≤ annot s]
note v7[OF - v3 v5 l-lp-r] p-mon
hence v8:
   α s = α l @ ((), lp) # α r
  ¬ Infty + sum-list (map snd (α l)) ≤ annot s
  Infty + sum-list (map snd (α l)) + lp ≤ annot s
  invar l
  invar r
  by auto
hence v9: lp ≠ Infty
  using invs(2) by auto
hence v10:
  s' = app l r
  (e,a) = p-unwrap lp
  using l-lp-r A(3)
  apply (auto simp add: aluprio-defs)
  apply (cases lp)
  apply auto
  apply (cases lp)
  apply auto
  done
have lp ≤ annot s
  using v8(2,3) p-less-eq-lem1
  by auto
hence v11: a ≤ snd (p-unwrap (annot s))
  using v10(2) v2 v9
  apply (cases annot s)
  apply auto
  apply (cases lp)
  apply (auto simp add: plesseq-def)
  done
note sum-list-less-elems[OF invs(2)]
hence v12: ∀ y∈set (map snd (map p-unwrap (map snd (α s)))). a ≤ y
  using v4 v11 by auto
```

```
    have ran (aluprio-α α s) = set (map snd (map p-unwrap (map snd (α s))))
      using ran-distinct[OF invs(4)]
      apply (unfold aluprio-defs)
      apply (simp only: set-map)
      done
    hence ziel1: ∀ y∈ran (aluprio-α α s). a ≤ y
      using v12 by simp
    have v13:
      map p-unwrap (map snd (α s))
        = map p-unwrap (map  snd (α l)) @ (e,a) # map p-unwrap (map snd (α
r))
      using v8(1) v10 by auto
     hence v14:
      map fst (map p-unwrap (map snd (α s)))
        = map fst (map p-unwrap (map snd (α l))) @ e
            # map fst (map p-unwrap (map snd (α r)))
       by auto
    hence v15:
      e ∉ set (map fst (map p-unwrap (map snd (α l))))
      e ∉ set (map fst (map p-unwrap (map snd (α r))))
      using invs(4) by auto
    note map-of-distinct-lookup[OF v15]
    note this[of a]
    hence ziel2: aluprio-α α s e = Some a
      using  v13
      by (unfold aluprio-defs, auto)
    have v16:
      α s' = α l @ α r
      invar s'
      using v8(4,5) app-correct v10 by auto
    note map-of-distinct-upd4[OF v15]
    note this[of a]
    hence
      ziel3: aluprio-α α s' = (aluprio-α α s)(e := None)
      unfolding aluprio-defs
      using v16(1) v13 by auto
    have ziel4: aluprio-invar α invar s'
      using v16 v8(1) invs(2,3,4)
      unfolding aluprio-defs
      by (auto simp add: sorted-append)

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

lemmas aluprio-correct =
```

aluprio-finite-correct
aluprio-empty-correct
aluprio-isEmpty-correct
aluprio-insert-correct
aluprio-pop-correct
aluprio-prio-correct

locale *aluprio-defs* = *StdALDefs ops*
for *ops* :: (*unit*,('*e*::*linorder*,'*a*::*linorder*) *LP*,'*s*) *alist-ops*
begin
definition [*icf-rec-def*]: *aluprio-ops* ≡ (|
upr-α = *aluprio-α α*,
upr-invar = *aluprio-invar α invar*,
upr-empty = *aluprio-empty empty*,
upr-isEmpty = *aluprio-isEmpty isEmpty*,
upr-insert = *aluprio-insert splits annot isEmpty app consr*,
upr-pop = *aluprio-pop splits annot app*,
upr-prio = *aluprio-prio splits annot isEmpty*
|)

end

locale *aluprio* = *aluprio-defs ops* + *StdAL ops*
for *ops* :: (*unit*,('*e*::*linorder*,'*a*::*linorder*) *LP*,'*s*) *alist-ops*
begin
lemma *aluprio-ops-impl*: *StdUprio aluprio-ops*
apply (*rule StdUprio.intro*)
apply (*simp-all add*: *icf-rec-unf*)
apply (*rule aluprio-correct*)
apply (*rule aluprio-correct*, *unfold-locales*) []
apply (*rule aluprio-correct*, *unfold-locales*) []
apply (*rule aluprio-correct*, *unfold-locales*) []
apply (*rule aluprio-correct*, *unfold-locales*) []
apply (*rule aluprio-correct*, *unfold-locales*) []
done
end

end

3.3 Implementations

3.3.1 Map Implementation by Associative Lists

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

definition [*icf-rec-def*]: *lm-basic-ops* ≡ ⦇
  bmap-op-α = *Assoc-List.lookup*,
  bmap-op-invar = λ-. *True*,
  bmap-op-empty = (λ-::*unit*. *Assoc-List.empty*),
  bmap-op-lookup = (λ*k m*. *Assoc-List.lookup m k*),
  bmap-op-update = *Assoc-List.update*,
  bmap-op-update-dj = *Assoc-List.update*,
  bmap-op-delete = *Assoc-List.delete*,
  bmap-op-list-it = *Assoc-List.iteratei*
⦈

setup *Locale-Code.open-block*
interpretation *lm-basic*: *StdBasicMapDefs lm-basic-ops* **.**
interpretation *lm-basic*: *StdBasicMap lm-basic-ops*
  apply *unfold-locales*
  apply (*simp-all add*: *icf-rec-unf*
    Assoc-List.lookup-empty′ Assoc-List.iteratei-correct map-upd-eq-restrict)
  done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *lm-ops* ≡ *lm-basic.dflt-ops*
setup *Locale-Code.open-block*
interpretation *lm*: *StdMapDefs lm-ops* **.**
interpretation *lm*: *StdMap lm-ops*
  unfolding *lm-ops-def*
  by (*rule lm-basic.dflt-ops-impl*)
interpretation *lm*: *StdMap-no-invar lm-ops*
  by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs lm*›

lemma *pi-lm*[*proper-it*]:
  proper-it′ Assoc-List.iteratei Assoc-List.iteratei
  unfolding *Assoc-List.iteratei-def*[*abs-def*]
  by (*intro icf-proper-iteratorI proper-it′I*)

interpretation *pi-lm*: *proper-it-loc Assoc-List.iteratei Assoc-List.iteratei*
  apply *unfold-locales*
  apply (*rule pi-lm*)
  done

lemma *pi-lm′*[*proper-it*]:
  proper-it′ lm.iteratei lm.iteratei
  unfolding *lm.iteratei-def*[*abs-def*]
  by (*intro icf-proper-iteratorI proper-it′I*)

interpretation *pi-lm′*: *proper-it-loc lm.iteratei lm.iteratei*

apply *unfold-locales*
apply (*rule pi-lm'*)
done

Code generator test

definition *test-codegen* ≡ (
lm.add ,
lm.add-dj ,
lm.ball ,
lm.bex ,
lm.delete ,
lm.empty ,
lm.isEmpty ,
lm.isSng ,
lm.iterate ,
lm.iteratei ,
lm.list-it ,
lm.lookup ,
lm.restrict ,
lm.sel ,
lm.size ,
lm.size-abort ,
lm.sng ,
lm.to-list ,
lm.to-map ,
lm.update ,
lm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.2 Map Implementation by Association Lists with explicit invariants

theory *ListMapImpl-Invar*
imports
../spec/MapSpec
../../Lib/Assoc-List
../gen-algo/MapGA
begin type-synonym $('k,'v)$ *lmi* = $('k \times 'v)$ *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 = (λk m. Map.map-of m k),
  bmap-op-update = AList.update,
  bmap-op-update-dj = (λk v m. (k, v) # m),
  bmap-op-delete = AList.delete-aux,
  bmap-op-list-it = foldli
⦈


setup Locale-Code.open-block
interpretation lmi-basic: StdBasicMapDefs lmi-basic-ops .
interpretation lmi-basic: StdBasicMap lmi-basic-ops
  unfolding lmi-basic-ops-def
  apply unfold-locales
  apply (simp-all
    add: icf-rec-unf lmi-α-def lmi-invar-def
    add: AList.update-conv' AList.distinct-update AList.map-of-delete-aux'
      map-iterator-foldli-correct dom-map-of-conv-image-fst map-upd-eq-restrict
  )
  done
setup Locale-Code.close-block


definition [icf-rec-def]: lmi-ops ≡ lmi-basic.dflt-ops ⦇
  map-op-add-dj := revg,
  map-op-to-list := id,
  map-op-size := length,
  map-op-isEmpty := case-list True (λ- -. False),
  map-op-isSng := (λl. case l of [-] ⇒ True | - ⇒ False)
⦈

setup Locale-Code.open-block
interpretation lmi: StdMapDefs lmi-ops .
interpretation lmi: StdMap lmi-ops
proof −
  interpret aux: StdMap lmi-basic.dflt-ops by (rule lmi-basic.dflt-ops-impl)

  have [simp]: map-add-dj lmi-α lmi-invar revg
    apply (unfold-locales)
    apply (auto simp: lmi-α-def lmi-invar-def)
    apply (blast intro: map-add-comm)
    apply (simp add: rev-map[symmetric])
    apply fastforce
    done

  have [simp]: map-to-list lmi-α lmi-invar id
    apply unfold-locales
```

```
    by (simp-all add: lmi-α-def lmi-invar-def)

  have [simp]: map-isEmpty lmi-α lmi-invar (case-list True (λ- -. False))
    apply unfold-locales
    unfolding lmi-α-def lmi-invar-def
    by (simp split: list.split)

  have [simp]: map-isSng lmi-α lmi-invar
     (λl. case l of [-] ⇒ True | - ⇒ False)
    apply unfold-locales
    unfolding lmi-α-def lmi-invar-def
    apply (auto split: list.split)
    apply (metis (no-types) map-upd-nonempty)
    by (metis fun-upd-other fun-upd-same option.simps(3))

  have [simp]: map-size-axioms lmi-α lmi-invar length
    apply unfold-locales
    unfolding lmi-α-def lmi-invar-def
    by (metis card-dom-map-of)

  show StdMap lmi-ops
    unfolding lmi-ops-def
    apply (rule StdMap-intro)
    apply (simp-all)
    apply intro-locales
    apply (simp-all add: icf-rec-unf)
    done
qed
setup Locale-Code.close-block

setup ‹ICF-Tools.revert-abbrevs lmi›

lemma pi-lmi[proper-it]:
  proper-it' foldli foldli
  by (intro proper-it'I icf-proper-iteratorI)

interpretation pi-lmi: proper-it-loc foldli foldli
  apply unfold-locales
  apply (rule pi-lmi)
  done

definition lmi-from-list-dj :: ('k×'v) list ⇒ ('k,'v) lmi where
  lmi-from-list-dj ≡ id

lemma lmi-from-list-dj-correct:
  assumes [simp]: distinct (map fst l)
  shows lmi.α (lmi-from-list-dj l) = map-of l
        lmi.invar (lmi-from-list-dj l)
  by (auto simp add: lmi-from-list-dj-def icf-rec-unf lmi-α-def lmi-invar-def)
```

Code generator test

definition *test-codegen* ≡ (
lmi.add ,
lmi.add-dj ,
lmi.ball ,
lmi.bex ,
lmi.delete ,
lmi.empty ,
lmi.isEmpty ,
lmi.isSng ,
lmi.iterate ,
lmi.iteratei ,
lmi.list-it ,
lmi.lookup ,
lmi.restrict ,
lmi.sel ,
lmi.size ,
lmi.size-abort ,
lmi.sng ,
lmi.to-list ,
lmi.to-map ,
lmi.update ,
lmi.update-dj,
lmi-from-list-dj
)

export-code *test-codegen* **checking** *SML*

end

3.3.3 Map Implementation by Red-Black-Trees

theory *RBTMapImpl*
imports
../spec/MapSpec
../../Lib/RBT-add
HOL−Library.RBT
../gen-algo/MapGA
begin **hide-const** (**open**) *RBT.map RBT.fold RBT.foldi RBT.empty RBT.insert*

type-synonym ('*k*,'*v*) *rm* = ('*k*,'*v*) *RBT.rbt*

definition *rm-basic-ops* :: ('*k*::*linorder*,'*v*,('*k*,'*v*) *rm*) *omap-basic-ops*
where [*icf-rec-def*]: *rm-basic-ops* ≡ (|
bmap-op-α = *RBT.lookup*,
bmap-op-invar = λ-. *True*,
bmap-op-empty = (λ-::*unit*. *RBT.empty*),

bmap-op-lookup = (λ*k 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 = (λ*r*. *RBT-add.rm-iterateoi* (*RBT.impl-of r*)),
bmap-op-ordered-list-it = (λ*r*. *RBT-add.rm-iterateoi* (*RBT.impl-of r*)),
bmap-op-rev-list-it = (λ*r*. *RBT-add.rm-reverse-iterateoi* (*RBT.impl-of r*))
⦈

setup *Locale-Code.open-block*
interpretation *rm-basic*: *StdBasicOMap rm-basic-ops*
apply *unfold-locales*
apply (*simp-all add*: *rm-basic-ops-def map-upd-eq-restrict*)
apply (*rule map-iterator-linord-is-it*)
apply *dup-subgoals*
unfolding *RBT.lookup-def*
apply *simp-all*
apply (*rule rm-iterateoi-correct*)
apply *simp*
apply (*rule rm-reverse-iterateoi-correct*)
apply *simp*
done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *rm-ops* ≡ *rm-basic.dflt-oops*⦇*map-op-add* := *RBT.union*⦈
setup *Locale-Code.open-block*
interpretation *rm*: *StdOMap rm-ops*
proof −
interpret *aux1*: *StdOMap rm-basic.dflt-oops*
unfolding *rm-ops-def*
by (*rule rm-basic.dflt-oops-impl*)
interpret *aux2*: *map-add RBT.lookup* λ-. *True RBT.union*
apply *unfold-locales*
apply (*rule lookup-union*)
.

show *StdOMap rm-ops*
apply (*rule StdOMap-intro*)
apply *icf-locales*
done
qed

interpretation *rm*: *StdMap-no-invar rm-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs rm*›

lemma *pi-rm*[*proper-it*]:

proper-it′ RBT-add.rm-iterateoi RBT-add.rm-iterateoi
apply (*rule proper-it′I*)
by (*induct-tac s*) (*simp-all add*: *rm-iterateoi-alt-def icf-proper-iteratorI*)
lemma *pi-rm-rev*[*proper-it*]:
proper-it′ RBT-add.rm-reverse-iterateoi RBT-add.rm-reverse-iterateoi
apply (*rule proper-it′I*)
by (*induct-tac s*) (*simp-all add*: *rm-reverse-iterateoi-alt-def*
icf-proper-iteratorI)

interpretation *pi-rm*: *proper-it-loc RBT-add.rm-iterateoi RBT-add.rm-iterateoi*
apply *unfold-locales* **by** (*rule pi-rm*)
interpretation *pi-rm-rev*: *proper-it-loc RBT-add.rm-reverse-iterateoi*
RBT-add.rm-reverse-iterateoi
apply *unfold-locales* **by** (*rule pi-rm-rev*)

Code generator test

definition *test-codegen* ≡ (*rm.add* ,
rm.add-dj ,
rm.ball ,
rm.bex ,
rm.delete ,
rm.empty ,
rm.isEmpty ,
rm.isSng ,
rm.iterate ,
rm.iteratei ,
rm.iterateo ,
rm.iterateoi ,
rm.list-it ,
rm.lookup ,
rm.max ,
rm.min ,
rm.restrict ,
rm.rev-iterateo ,
rm.rev-iterateoi ,
rm.rev-list-it ,
rm.reverse-iterateo ,
rm.reverse-iterateoi ,
rm.sel ,
rm.size ,
rm.size-abort ,
rm.sng ,
rm.to-list ,
rm.to-map ,
rm.to-rev-list ,
rm.to-sorted-list ,
rm.update ,
rm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.4 Hash maps implementation

theory *HashMap-Impl*
imports
  RBTMapImpl
  ListMapImpl
  ../../Lib/HashCode
begin

We use a red-black tree instead of an indexed array. This has the disadvantage of being more complex, however we need not bother about a fixed-size array and rehashing if the array becomes too full.

The entries of the red-black tree are lists of (key,value) pairs.

Abstract Hashmap

We first specify the behavior of our hashmap on the level of maps. We will then show that our implementation based on hashcode-map and bucket-map is a correct implementation of this specification.

type-synonym
  $('k,'v)$ *abs-hashmap* $=$ *hashcode* $\rightharpoonup ('k \rightharpoonup 'v)$

— Map entry of map by function
abbreviation *map-entry* **where** *map-entry* $k\ f\ m == m(k := f\ (m\ k))$

— Invariant: Buckets only contain entries with the right hashcode and there are no empty buckets
definition *ahm-invar*:: $('k{::}hashable,'v)$ *abs-hashmap* $\Rightarrow$ *bool*
  where *ahm-invar* $m ==$
    $(\forall\ hc\ cm\ k.\ m\ hc = Some\ cm \wedge k{\in}dom\ cm \longrightarrow hashcode\ k = hc) \wedge$
    $(\forall\ hc\ cm.\ m\ hc = Some\ cm \longrightarrow cm \neq Map.empty)$

— Abstract a hashmap to the corresponding map
definition *ahm-α* **where**
  ahm-α $m\ k ==$ *case* $m\ (hashcode\ k)$ *of*
    $None \Rightarrow None\ |$
    $Some\ cm \Rightarrow cm\ k$

— Lookup an entry
definition *ahm-lookup* $::\ 'k{::}hashable \Rightarrow ('k,'v)$ *abs-hashmap* $\Rightarrow\ 'v$ *option*
  where *ahm-lookup* $k\ m == (ahm\text{-}\alpha\ m)\ k$

— The empty hashmap
definition *ahm-empty* :: ('k::hashable,'v) *abs-hashmap*
  where *ahm-empty* = *Map.empty*

— Update/insert an entry
definition *ahm-update* **where**
  ahm-update k v m ==
    case m (*hashcode k*) *of*
      None ⇒ *m* (*hashcode k* ↦ [*k* ↦ *v*]) |
      Some cm ⇒ *m* (*hashcode k* ↦ *cm* (*k* ↦ *v*))

— Delete an entry
definition *ahm-delete* **where**
  ahm-delete k m == *map-entry* (*hashcode k*)
    (λ*v*. *case v of*
      None ⇒ *None* |
      Some bm ⇒ (
        if bm |` (− {*k*}) = *Map.empty then*
          None
        else
          Some (*bm* |` (− {*k*}))
)
) *m*

definition *ahm-isEmpty* **where**
  ahm-isEmpty m == *m*=*Map.empty*

Now follow correctness lemmas, that relate the hashmap operations to operations on the corresponding map. Those lemmas are named op_correct, where (is) the operation.

lemma *ahm-invarI*: ⟦
  !!*hc cm k*. ⟦*m hc* = *Some cm*; *k*∈*dom cm*⟧ ⟹ *hashcode k* = *hc*;
  !!*hc cm*. ⟦ *m hc* = *Some cm* ⟧ ⟹ *cm* ≠ *Map.empty*
  ⟧ ⟹ *ahm-invar m*
  by (*unfold ahm-invar-def*) *blast*

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

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

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

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

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

lemma *ahm-update-correct*:
  ahm-α (*ahm-update k v m*) = (*ahm-α m*)($k \mapsto v$)
  ahm-invar m $\Longrightarrow$ *ahm-invar* (*ahm-update k v m*)
  apply (*rule ext*)
  apply (*unfold ahm-update-def*)
  apply (*auto simp add*: *ahm-α-def split*: *option.split*)
  apply (*rule ahm-invarI*)
  apply (*auto dest*: *ahm-invarD ahm-invarDne split*: *if-split-asm*)
  apply (*rule ahm-invarI*)
  apply (*auto dest*: *ahm-invarD split*: *if-split-asm*)
  apply (*drule* (*1*) *ahm-invarD*)
  apply *auto*
  done

lemma *fun-upd-apply-ne*: $x \neq y \Longrightarrow (f(x:=v))\ y = f\ y$
  by *simp*

lemma *cancel-one-empty-simp*: m |‘ $(-\{k\})$ = *Map.empty* $\longleftrightarrow$ *dom* $m \subseteq \{k\}$
proof
  assume m |‘ $(-\ \{k\})$ = *Map.empty*
  hence $\{\}$ = *dom* (m |‘ $(-\{k\})$) **by** *auto*
  also have ... = *dom* $m - \{k\}$ **by** *auto*
  finally show *dom* $m \subseteq \{k\}$ **by** *blast*
next
  assume *dom* $m \subseteq \{k\}$
  hence *dom* $m - \{k\} = \{\}$ **by** *auto*
  hence *dom* (m |‘ $(-\{k\})$) = $\{\}$ **by** *auto*
  thus m |‘ $(-\{k\})$ = *Map.empty* **by** *blast*
qed

lemma *ahm-delete-correct*:
  ahm-α (*ahm-delete k m*) = (*ahm-α m*) |‘ $(-\{k\})$
  ahm-invar m $\Longrightarrow$ *ahm-invar* (*ahm-delete k m*)
  apply (*rule ext*)
  apply (*unfold ahm-delete-def*)
  apply (*auto simp add*: *ahm-α-def Let-def Map.restrict-map-def*
        split: *option.split*)[*1*]

apply (*drule-tac x=x* **in** *fun-cong*)
apply (*auto*)[*1*]
apply (*rule ahm-invarI*)
apply (*auto split*: *if-split-asm option.split-asm dest*: *ahm-invarD*)
apply (*drule* (*1*) *ahm-invarD*)
apply (*auto simp add*: *restrict-map-def split*: *if-split-asm option.split-asm*)
done

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

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

lemmas *ahm-correct* = *ahm-empty-correct ahm-lookup-correct ahm-update-correct*

ahm-delete-correct ahm-isEmpty-correct

— Bucket entries correspond to map entries
lemma *ahm-be-is-e*:
assumes *I*: *ahm-invar m*
assumes *A*: *m hc* = *Some bm* *bm k* = *Some v*
shows *ahm-*α *m k* = *Some v*
using *A*
apply (*auto simp add*: *ahm-*α*-def split*: *option.split dest*: *ahm-invarD*[*OF I*])
apply (*frule ahm-invarD*[*OF I*, **where** *k=k*])
apply *auto*
done

— Map entries correspond to bucket entries

lemma *ahm-e-is-be*: ⟦
  $ahm\text{-}\alpha\ m\ k = Some\ v$;
  $!!bm.\ ⟦m\ (hashcode\ k) = Some\ bm;\ bm\ k = Some\ v\ ⟧ \Longrightarrow P$
  $⟧ \Longrightarrow P$
  by (*unfold ahm-α-def*)
    (*auto split*: *option.split-asm*)

Concrete Hashmap

In this section, we define the concrete hashmap that is made from the hashcode map and the bucket map.

We then show the correctness of the operations w.r.t. the abstract hashmap, and thus, indirectly, w.r.t. the corresponding map.

type-synonym
  ('k,'v) *hm-impl* = (*hashcode*, ('k,'v) *lm*) *rm*

Operations **definition** *rm-map-entry*
  :: *hashcode* ⇒ ('v *option* ⇒ 'v *option*) ⇒ (*hashcode*, 'v) *rm* ⇒ (*hashcode*,'v) *rm*
  where
  rm-map-entry k f m ==
    case rm.lookup k m of
      None ⇒ (
        case f None of
          None ⇒ *m* |
          Some v ⇒ *rm.update k v m*
) |
      Some v ⇒ (
        case f (*Some v*) *of*
          None ⇒ *rm.delete k m* |
          Some v' ⇒ *rm.update k v' m*
)

— Empty hashmap

definition *empty* :: *unit* ⇒ ('k :: *hashable*, 'v) *hm-impl* **where** *empty* == *rm.empty*

— Update/insert entry

definition *update* :: 'k::*hashable* ⇒ 'v ⇒ ('k,'v) *hm-impl* ⇒ ('k,'v) *hm-impl*
  where
  update k v m ==
   let hc = *hashcode k in*
    case rm.lookup hc m of
      None ⇒ *rm.update hc* (*lm.update k v* (*lm.empty* ())) *m* |
      Some bm ⇒ *rm.update hc* (*lm.update k v bm*) *m*

— Lookup value by key

definition *lookup* :: $'k$::*hashable* $\Rightarrow$ $('k,'v)$ *hm-impl* $\Rightarrow$ $'v$ *option* **where**
lookup k m ==
case rm.lookup (*hashcode k*) *m of*
None $\Rightarrow$ *None* |
Some lm $\Rightarrow$ *lm.lookup k lm*

— Delete entry by key
definition *delete* :: $'k$::*hashable* $\Rightarrow$ $('k,'v)$ *hm-impl* $\Rightarrow$ $('k,'v)$ *hm-impl* **where**
delete k m ==
rm-map-entry (*hashcode k*)
(λv. *case v of*
None $\Rightarrow$ *None* |
Some lm $\Rightarrow$ (
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* $\sigma 0$ ==
rm.iteratei m c (λ(*hc*, *lm*) σ.
lm.iteratei lm c f σ
) $\sigma 0$

lemma *iteratei-alt-def* :
iteratei m = *set-iterator-image snd* (
set-iterator-product (*rm.iteratei m*) (λ*hclm*. *lm.iteratei* (*snd hclm*)))
proof −
have *aux*: $\bigwedge c\ f$. (λ(*hc*, *lm*). *lm.iteratei lm c f*) = (λm. *lm.iteratei* (*snd m*) *c f*)
by *auto*
show *?thesis*
unfolding *set-iterator-product-def set-iterator-image-alt-def*
iteratei-def[*abs-def*] *aux*
by (*simp add*: *split-beta*)
qed

Correctness w.r.t. Abstract HashMap The following lemmas establish the correctness of the operations w.r.t. the abstract hashmap.

They have the naming scheme op_correct', where (is) the name of the operation.

definition *hm-*α' **where** *hm-*α' *m* == λ*hc*. *case rm.*α *m hc of*
None $\Rightarrow$ *None* |
Some lm $\Rightarrow$ *Some* (*lm.*α *lm*)

— Invariant for concrete hashmap: The hashcode-map and bucket-maps satisfy their invariants and the invariant of the corresponding abstract hashmap is satisfied.

definition *invar m* == *ahm-invar* (*hm-α′ m*)

lemma *rm-map-entry-correct*:
rm.α (*rm-map-entry k f m*) = (*rm.α m*)(*k* := *f* (*rm.α m k*))
apply (*auto*
simp add: *rm-map-entry-def rm.delete-correct rm.lookup-correct rm.update-correct*

split: *option.split*)
done

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

lemma *lookup-correct′*:
invar m ⟹ *lookup k m* = *ahm-lookup k* (*hm-α′ m*)
apply (*unfold lookup-def invar-def*)
apply (*auto split*: *option.split*
simp add: *ahm-lookup-def ahm-α-def hm-α′-def*
rm.correct lm.correct)
done

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

lemma *delete-correct′*:
invar m ⟹ *hm-α′* (*delete k m*) = *ahm-delete k* (*hm-α′ m*)
invar m ⟹ *invar* (*delete k m*)
proof −
assume *invar m*
thus *hm-α′* (*delete k m*) = *ahm-delete k* (*hm-α′ m*)
apply (*unfold invar-def*)

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

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



lemma iteratei-correct':
  assumes invar: invar hm
  shows map-iterator (iteratei hm) (ahm-α (hm-α' hm))
proof -
  from rm.iteratei-correct
  have it-rm: map-iterator (rm.iteratei hm) (rm.α hm) by simp

  from lm.iteratei-correct
  have it-lm: ⋀lm. map-iterator (lm.iteratei lm) (lm.α lm) by simp

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

  show ?thesis unfolding iteratei-alt-def
  proof (rule set-iterator-image-correct[OF it-prod])
    from invar
    show inj-on snd
       (SIGMA hclm:map-to-set (rm.α hm). map-to-set (lm.α (snd hclm)))
      apply (simp add: inj-on-def invar-def ahm-invar-def hm-α'-def Ball-def
                  map-to-set-def split: option.splits)
      apply (metis domI option.inject)
    done
  next
    from invar
    show map-to-set (ahm-α (hm-α' hm)) =
       snd ` (SIGMA hclm:map-to-set (rm.α hm). map-to-set (lm.α (snd hclm)))
```

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


lemmas hm-correct' = empty-correct' lookup-correct' update-correct'
          delete-correct' isEmpty-correct'
          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
  show HashMap-Impl.empty () ∈ {hm. HashMap-Impl.invar hm}
   by(simp add: HashMap-Impl.empty-correct')
qed

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

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

definition hm-empty-const :: ('k :: hashable, 'v) hashmap
where hm-empty-const = RBT-HM (HashMap-Impl.empty ())

definition hm-empty :: unit ⇒ ('k :: hashable, 'v) hashmap
where hm-empty = (λ-. hm-empty-const)

definition hm-lookup k hm == HashMap-Impl.lookup k (impl-of-RBT-HM hm)
```

definition *hm-update* :: ('*k* :: *hashable*) ⇒ '*v* ⇒ ('*k*, '*v*) *hashmap* ⇒ ('*k*, '*v*) *hashmap*
where *hm-update k v hm* = *RBT-HM* (*HashMap-Impl.update k v* (*impl-of-RBT-HM hm*))

definition *hm-update-dj* :: ('*k* :: *hashable*) ⇒ '*v* ⇒ ('*k*, '*v*) *hashmap* ⇒ ('*k*, '*v*) *hashmap*
where *hm-update-dj* = *hm-update*

definition *hm-delete* :: ('*k* :: *hashable*) ⇒ ('*k*, '*v*) *hashmap* ⇒ ('*k*, '*v*) *hashmap*
where *hm-delete k hm* = *RBT-HM* (*HashMap-Impl.delete k* (*impl-of-RBT-HM hm*))

definition *hm-isEmpty* :: ('*k* :: *hashable*, '*v*) *hashmap* ⇒ *bool*
where *hm-isEmpty hm* = *HashMap-Impl.isEmpty* (*impl-of-RBT-HM hm*)

definition *hm-iteratei hm* == *HashMap-Impl.iteratei* (*impl-of-RBT-HM hm*)

lemma *impl-of-hm-empty* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-empty-const*) = *HashMap-Impl.empty* ()
by(*simp add*: *hm-empty-const-def empty-correct' RBT-HM-inverse*)

lemma *impl-of-hm-update* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-update k v hm*) = *HashMap-Impl.update k v* (*impl-of-RBT-HM hm*)
by(*simp add*: *hm-update-def update-correct' RBT-HM-inverse*)

lemma *impl-of-hm-delete* [*simp*, *code abstract*]:
impl-of-RBT-HM (*hm-delete k hm*) = *HashMap-Impl.delete k* (*impl-of-RBT-HM hm*)
by(*simp add*: *hm-delete-def delete-correct' RBT-HM-inverse*)

Correctness w.r.t. Map

The next lemmas establish the correctness of the hashmap operations w.r.t. the associated map. This is achieved by chaining the correctness lemmas of the concrete hashmap w.r.t. the abstract hashmap and the correctness lemmas of the abstract hashmap w.r.t. maps.

type-synonym ('*k*, '*v*) *hm* = ('*k*, '*v*) *hashmap*

— Abstract concrete hashmap to map
definition *hm-α* == *ahm-α* ∘ *hm-α'* ∘ *impl-of-RBT-HM*

abbreviation (*input*) *hm-invar* :: ('*k* :: *hashable*, '*v*) *hashmap* ⇒ *bool*

where *hm-invar* == λ-. *True*

lemma *hm-aux-correct*:
  hm-α (*hm-empty* ()) = *Map.empty*
  hm-lookup k m = *hm-α m k*
  hm-α (*hm-update k v m*) = (*hm-α m*)(*k*↦*v*)
  hm-α (*hm-delete k m*) = (*hm-α m*) |‘ (−{*k*})
by(*auto simp add*: *hm-α-def hm-correct′ hm-empty-def ahm-correct hm-lookup-def*)

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

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

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*
⦈

setup *Locale-Code.open-block*
interpretation *hm-basic*: *StdBasicMap hm-basic-ops*
apply *unfold-locales*
apply (*simp-all add*: *icf-rec-unf hm-aux-correct hm-iteratei-impl*)
done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *hm-ops* ≡ *hm-basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *hm*: *StdMapDefs hm-ops* **.**
interpretation *hm*: *StdMap hm-ops*
unfolding *hm-ops-def*
by (*rule hm-basic.dflt-ops-impl*)
interpretation *hm*: *StdMap-no-invar hm-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs hm*›

lemma *pi-hm*[*proper-it*]:
shows *proper-it′ hm-iteratei hm-iteratei*
apply (*rule proper-it′I*)
unfolding *hm-iteratei-def HashMap-Impl.iteratei-alt-def*
by (*intro icf-proper-iteratorI*)

interpretation *pi-hm*: *proper-it-loc hm-iteratei hm-iteratei*
apply *unfold-locales*
apply (*rule pi-hm*)
done

Code generator test

definition *test-codegen* **where** *test-codegen* ≡ (
hm.add ,
hm.add-dj ,
hm.ball ,
hm.bex ,
hm.delete ,
hm.empty ,
hm.isEmpty ,

hm.isSng ,
hm.iterate ,
hm.iteratei ,
hm.list-it ,
hm.lookup ,
hm.restrict ,
hm.sel ,
hm.size ,
hm.size-abort ,
hm.sng ,
hm.to-list ,
hm.to-map ,
hm.update ,
hm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.6 Implementation of a trie with explicit invariants

theory *Trie-Impl* **imports**
../../Lib/Assoc-List
Trie.Trie
begin

Interuptible iterator

fun *iteratei-postfixed* :: $'key\ list \Rightarrow ('key, 'val)\ trie \Rightarrow$
$('key\ list \times 'val, '\sigma)\ set\text{-}iterator$
where
iteratei-postfixed ks $(Trie\ vo\ ts)\ c\ f\ \sigma =$
$(if\ c\ \sigma$
$then\ foldli\ ts\ c\ (\lambda(k, t)\ \sigma.\ iteratei\text{-}postfixed\ (k \# ks)\ t\ c\ f\ \sigma)$
$(case\ vo\ of\ None \Rightarrow \sigma \mid Some\ v \Rightarrow f\ (ks, v)\ \sigma)$
$else\ \sigma)$

definition *iteratei* :: $('key, 'val)\ trie \Rightarrow ('key\ list \times 'val, '\sigma)\ set\text{-}iterator$
where $iteratei\ t\ c\ f\ \sigma = iteratei\text{-}postfixed\ []\ t\ c\ f\ \sigma$

lemma *iteratei-postfixed-interrupt*:
$\neg\ c\ \sigma \Longrightarrow iteratei\text{-}postfixed\ ks\ t\ c\ f\ \sigma = \sigma$
by(*cases t*) *simp*

lemma *iteratei-interrupt*:
$\neg\ c\ \sigma \Longrightarrow iteratei\ t\ c\ f\ \sigma = \sigma$
unfolding *iteratei-def* **by** (*simp add*: *iteratei-postfixed-interrupt*)

lemma *iteratei-postfixed-alt-def* :

```
  iteratei-postfixed ks ((Trie vo ts)::('key, 'val) trie) =
   (set-iterator-union
     (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 −
  have aux: ⋀c f. (λ(k, t). iteratei-postfixed (k # ks) t c f) =
             (λa. iteratei-postfixed (fst a # ks) (snd a) c f)
    by auto

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

lemma iteratei-postfixed-correct :
  assumes invar: invar-trie (t :: ('key, 'val) trie)
  shows set-iterator ((iteratei-postfixed ks0 t)::('key list × 'val, 'σ) set-iterator)
         ((λksv. (rev (fst ksv) @ ks0, (snd ksv))) ` (map-to-set (lookup-trie t)))
using invar
proof (induct t arbitrary: ks0)
  case (Trie vo kvs)
  note ind-hyp = Trie(1)
  note invar = Trie(2)

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

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

— children iterator
define *it-prod* :: $(('key \times ('key, 'val)\ trie) \times 'key\ list \times 'val, '\sigma)$ *set-iterator*
where *it-prod* = *set-iterator-product* (*foldli kvs*) ($\lambda(k, y)$. *iteratei-postfixed* (*k* # *ks0*) *y*)

define *it-prod-S* **where** *it-prod-S* = (*SIGMA kt*:*set kvs*.
(λ*ksv*. (*rev* (*fst ksv*) @ ((*fst kt*) # *ks0*), *snd ksv*)) ‘
map-to-set (*lookup-trie* (*snd kt*)))

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

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

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

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

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

— overall iterator
have *it-all-OK*: *set-iterator*
((*iteratei-postfixed ks0* (*Trie vo kvs*)):: $('key\ list \times 'val, '\sigma)$ *set-iterator*)
(*vo-S* ∪ *snd* ‘ *it-prod-S*)
unfolding *iteratei-postfixed-alt-def*
it-vo-def[*symmetric*]
it-prod-def[*symmetric*]
proof (*rule set-iterator-union-correct* [*OF it-vo-OK it-image-OK*])

show *vo-S* ∩ *snd* ‘ *it-prod-S* = {}
unfolding *vo-S-def it-prod-S-def*
by (*simp split*: *option.split add*: *set-eq-iff image-iff*)
qed

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

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

definition *trie-reverse-key* **where**
trie-reverse-key ksv = (*rev* (*fst ksv*), (*snd ksv*))

lemma *trie-reverse-key-alt-def* [*code*] :
trie-reverse-key (*ks*, *v*) = (*rev ks*, *v*)
unfolding *trie-reverse-key-def* **by** *auto*

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

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

hide-const (**open**) *iteratei*
hide-type (**open**) *trie*

end

3.3.7 Tries without invariants

theory *Trie2* **imports**

Trie-Impl
begin

Abstract type definition

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

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

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

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* $\times$ $'val$, $'\sigma$) *set-iterator*
where *iteratei t* = *set-iterator-image trie-reverse-key* (*Trie-Impl.iteratei* (*impl-of* t))

lemma *iteratei-code*[*code*] :
iteratei t c f = *Trie-Impl.iteratei* (*impl-of* t) c (λ(*ks*, v). f (*rev ks*, v))
unfolding *iteratei-def set-iterator-image-alt-def*
apply (*subgoal-tac* (λx. f (*trie-reverse-key* x)) = (λ(*ks*, v). f (*rev ks*, v)))
apply (*auto simp add*: *trie-reverse-key-def*)
done

lemma *impl-of-empty* [*code abstract*]: *impl-of empty* = *empty-trie*
by(*simp add*: *empty-def Trie-inverse*)

lemma *impl-of-update* [*code abstract*]: *impl-of* (*update ks v t*) = *update-trie ks v* (*impl-of t*)
by(*simp add*: *update-def Trie-inverse invar-trie-update*)

lemma *impl-of-delete* [*code abstract*]: *impl-of* (*delete ks t*) = *delete-trie ks* (*impl-of t*)
by(*simp add*: *delete-def Trie-inverse invar-trie-delete*)

Correctness of primitive operations

lemma *lookup-empty* [*simp*]: *lookup empty* = *Map.empty*
by(*simp add*: *lookup-def empty-def Trie-inverse*)

lemma *lookup-update* [*simp*]: *lookup* (*update ks v t*) = (*lookup t*)(*ks* $\mapsto$ *v*)
by(*simp add*: *lookup-def update-def Trie-inverse invar-trie-update lookup-update'*)

lemma *lookup-delete* [*simp*]: *lookup* (*delete ks t*) = (*lookup t*)(*ks* := *None*)
by(*simp add*: *lookup-def delete-def Trie-inverse invar-trie-delete lookup-delete'*)

lemma *isEmpty-lookup*: *isEmpty t* $\longleftrightarrow$ *lookup t* = *Map.empty*
by(*simp add*: *isEmpty-def lookup-def is-empty-lookup-empty*)

lemma *finite-dom-lookup*: *finite* (*dom* (*lookup t*))
by(*simp add*: *lookup-def finite-dom-lookup*)

lemma *iteratei-correct*:
  map-iterator (*iteratei m*) (*lookup m*)
proof −
  note *it-base* = *Trie-Impl.trie-iteratei-correct* [*of impl-of m*]
  show *?thesis*
    unfolding *iteratei-def lookup-def*
    apply (*rule set-iterator-image-correct* [*OF it-base*])
    apply (*simp-all add*: *set-eq-iff image-iff inj-on-def*)
  done
qed

Type classes

instantiation *trie* :: (*equal*, *equal*) *equal* **begin**

definition *equal-class.equal* (*t* :: (*'a*, *'b*) *trie*) *t'* = (*impl-of t* = *impl-of t'*)

instance
proof
qed(*simp add*: *equal-trie-def impl-of-inject*)
end

hide-const (**open**) *empty lookup update delete iteratei isEmpty*

end

3.3.8 Map implementation via tries

theory *TrieMapImpl* **imports**
  Trie2
  ../gen-algo/MapGA
begin

Operations

type-synonym ('k, 'v) *tm* = ('k, 'v) *trie*

definition [*icf-rec-def*]: *tm-basic-ops* ≡ ⦇
  bmap-op-α = *Trie2.lookup*,
  bmap-op-invar = λ-. *True*,
  bmap-op-empty = (λ-::*unit*. *Trie2.empty*),
  bmap-op-lookup = (λ*k m*. *Trie2.lookup m k*),
  bmap-op-update = *Trie2.update*,
  bmap-op-update-dj = *Trie2.update*,
  bmap-op-delete = *Trie2.delete*,
  bmap-op-list-it = *Trie2.iteratei*
⦈

setup *Locale-Code.open-block*
interpretation *tm-basic*: *StdBasicMap tm-basic-ops*
  apply *unfold-locales*
  apply (*simp-all*
    add: *icf-rec-unf Trie2.finite-dom-lookup Trie2.iteratei-correct*
    add: *map-upd-eq-restrict*)
  done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *tm-ops* ≡ *tm-basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *tm*: *StdMap tm-ops*
  unfolding *tm-ops-def*
  by (*rule tm-basic.dflt-ops-impl*)
interpretation *tm*: *StdMap-no-invar tm-ops*
  by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs tm*›

lemma *pi-trie-impl*[*proper-it*]:
  shows *proper-it'*
    ((*Trie-Impl.iteratei*) :: - ⇒ (-,'σa) *set-iterator*)
    ((*Trie-Impl.iteratei*) :: - ⇒ (-,'σb) *set-iterator*)

```
  unfolding Trie-Impl.iteratei-def[abs-def]
proof (rule proper-it'I)

  fix t :: ('k,'v) Trie.trie
  {
    fix l and t :: ('k,'v) Trie.trie
    have proper-it ((Trie-Impl.iteratei-postfixed l t)
        :: (-,'σa) set-iterator)
      ((Trie-Impl.iteratei-postfixed l t)
        :: (-,'σb) set-iterator)
    proof (induct t arbitrary: l)
      case (Trie vo kvs l)

      let ?ITA = λl t. (Trie-Impl.iteratei-postfixed l t)
        :: (-,'σa) set-iterator
      let ?ITB = λl t. (Trie-Impl.iteratei-postfixed l t)
        :: (-,'σb) set-iterator

      show ?case
        unfolding Trie-Impl.iteratei-postfixed-alt-def
        apply (rule pi-union)
        apply (auto split: option.split intro: icf-proper-iteratorI) []
      proof (rule pi-image)
        define bs where bs = (λ(k,t). SOME l'::('k list × 'v) list.
          ?ITA (k#l) t = foldli l' ∧ ?ITB (k#l) t = foldli l')

        have EQ1: ∀(k,t)∈set kvs. ?ITA (k#l) t = foldli (bs (k,t)) and
          EQ2: ∀(k,t)∈set kvs. ?ITB (k#l) t = foldli (bs (k,t))
        proof (safe)
          fix k t
          assume A: (k,t) ∈ set kvs
          from Trie.hyps[OF A, of k#l] have
            PI: proper-it (?ITA (k#l) t) (?ITB (k#l) t)
            by assumption
          obtain l' where
            ?ITA (k#l) t = foldli l'
          ∧ (?ITB (k#l) t) = foldli l'
            by (blast intro: proper-itE[OF PI])
          thus ?ITA (k#l) t = foldli (bs (k,t))
            ?ITB (k#l) t = foldli (bs (k,t))
            unfolding bs-def
            apply auto
            apply (metis (lifting, full-types) someI-ex)
            apply (metis (lifting, full-types) someI-ex)
            done
        qed

        have PEQ1: set-iterator-product (foldli kvs) (λ(k,t). ?ITA (k#l) t)
          = set-iterator-product (foldli kvs) (λkt. foldli (bs kt))
```

```
          apply (rule set-iterator-product-eq2)
          using EQ1 by auto
        have PEQ2: set-iterator-product (foldli kvs) (λ(k,t). ?ITB (k#l) t)
          = set-iterator-product (foldli kvs) (λkt. foldli (bs kt))
          apply (rule set-iterator-product-eq2)
          using EQ2 by auto
        show proper-it
          (set-iterator-product (foldli kvs) (λ(k,t). ?ITA (k#l) t))
          (set-iterator-product (foldli kvs) (λ(k,t). ?ITB (k#l) t))
          apply (subst PEQ1)
          apply (subst PEQ2)
          apply (auto simp: set-iterator-product-foldli-conv)
          by (blast intro: proper-itI)
      qed
    qed
  } thus proper-it
      (iteratei-postfixed [] t :: (-,'σa) set-iterator)
      (iteratei-postfixed [] t :: (-,'σb) set-iterator) .
qed

lemma pi-trie[proper-it]:
  proper-it' Trie2.iteratei Trie2.iteratei
  unfolding Trie2.iteratei-def[abs-def]
  apply (rule proper-it'I)
  apply (intro icf-proper-iteratorI)
  apply (rule proper-it'D)
  by (rule pi-trie-impl)

interpretation pi-trie: proper-it-loc Trie2.iteratei Trie2.iteratei
  apply unfold-locales
  apply (rule pi-trie)
  done
```

Code generator test

```
definition test-codegen ≡ (
  tm.add ,
  tm.add-dj ,
  tm.ball ,
  tm.bex ,
  tm.delete ,
  tm.empty ,
  tm.isEmpty ,
  tm.isSng ,
  tm.iterate ,
  tm.iteratei ,
  tm.list-it ,
  tm.lookup ,
  tm.restrict ,
  tm.sel ,
```

tm.size ,
tm.size-abort ,
tm.sng ,
tm.to-list ,
tm.to-map ,
tm.update ,
tm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.9 Array-based hash map implementation

theory *ArrayHashMap-Impl* **imports**
../../Lib/HashCode
../../Lib/Diff-Array
../gen-algo/ListGA
ListMapImpl
../../Iterator/Array-Iterator
begin

Misc.

setup *Locale-Code.open-block*
interpretation *a-idx-it*:
idx-iteratei-loc list-of-array λ-. *True array-length array-get*
apply *unfold-locales*
apply (*case-tac* [!] *s*) [*2*]
apply *auto*
done
setup *Locale-Code.close-block*

Type definition and primitive operations

definition *load-factor* :: *nat* — in percent
where *load-factor* = *75*

We do not use ('*k*, '*v*) *assoc-list* for the buckets but plain lists of key-value pairs. This speeds up rehashing because we then do not have to go through the abstract operations.

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

Operations

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

definition *ahm-empty* :: *unit* ⇒ ('*key* :: *hashable*, '*val*) *hashmap*

where *ahm-empty* ≡ λ-. *new-hashmap-with* (*def-hashmap-size TYPE*(′*key*))

definition *bucket-ok* :: *nat* ⇒ *nat* ⇒ ((′*key* :: *hashable*) × ′*val*) *list* ⇒ *bool*
where *bucket-ok len h kvs* = (∀ *k* ∈ *fst* ‘ *set kvs*. *bounded-hashcode-nat len k* = *h*)

definition *ahm-invar-aux* :: *nat* ⇒ ((′*key* :: *hashable*) × ′*val*) *list array* ⇒ *bool*
where
ahm-invar-aux n a ⟷
(∀ *h*. *h* < *array-length a* ⟶ *bucket-ok* (*array-length a*) *h* (*array-get a h*) ∧ *distinct* (*map fst* (*array-get a h*))) ∧
array-foldl (λ- *n kvs*. *n* + *size kvs*) *0 a* = *n* ∧
array-length a > *1*

primrec *ahm-invar* :: (′*key* :: *hashable*, ′*val*) *hashmap* ⇒ *bool*
where *ahm-invar* (*HashMap a n*) = *ahm-invar-aux n a*

definition *ahm-α-aux* :: ((′*key* :: *hashable*) × ′*val*) *list array* ⇒ ′*key* ⇒ ′*val option*
where [*simp*]: *ahm-α-aux a k* = *map-of* (*array-get a* (*bounded-hashcode-nat* (*array-length a*) *k*)) *k*

primrec *ahm-α* :: (′*key* :: *hashable*, ′*val*) *hashmap* ⇒ ′*key* ⇒ ′*val option*
where
ahm-α (*HashMap a* -) = *ahm-α-aux a*

definition *ahm-lookup* :: ′*key* ⇒ (′*key* :: *hashable*, ′*val*) *hashmap* ⇒ ′*val option*
where *ahm-lookup k hm* = *ahm-α hm k*

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

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

definition *ahm-rehash-aux′* :: *nat* ⇒ ′*key* × ′*val* ⇒ ((′*key* :: *hashable*) × ′*val*) *list array* ⇒ (′*key* × ′*val*) *list array*
where
ahm-rehash-aux′ n kv a =
(*let h* = *bounded-hashcode-nat n* (*fst kv*)
in array-set a h (*kv* # *array-get a h*))

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

primrec *ahm-rehash* :: ('*key* :: *hashable*, '*val*) *hashmap* ⇒ *nat* ⇒ ('*key*, '*val*) *hashmap*
where *ahm-rehash* (*HashMap a n*) *sz* = *HashMap* (*ahm-rehash-aux a sz*) *n*

primrec *hm-grow* :: ('*key* :: *hashable*, '*val*) *hashmap* ⇒ *nat*
where *hm-grow* (*HashMap a n*) = *2* ∗ *array-length a* + *3*

primrec *ahm-filled* :: ('*key* :: *hashable*, '*val*) *hashmap* ⇒ *bool*
where *ahm-filled* (*HashMap a n*) = (*array-length a* ∗ *load-factor* ≤ *n* ∗ *100*)

primrec *ahm-update-aux* :: ('*key* :: *hashable*, '*val*) *hashmap* ⇒ '*key* ⇒ '*val* ⇒ ('*key*, '*val*) *hashmap*
where
  ahm-update-aux (*HashMap a n*) *k v* =
  (*let h* = *bounded-hashcode-nat* (*array-length a*) *k*;
      m = *array-get a h*;
      insert = *map-of m k* = *None*
   in HashMap (*array-set a h* (*AList.update k v m*)) (*if insert then n* + *1 else n*))

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

primrec *ahm-delete* :: '*key* ⇒ ('*key* :: *hashable*, '*val*) *hashmap* ⇒ ('*key*, '*val*) *hashmap*
where
  ahm-delete k (*HashMap a n*) =
  (*let h* = *bounded-hashcode-nat* (*array-length a*) *k*;
      m = *array-get a h*;
      deleted = (*map-of m k* ≠ *None*)
   in HashMap (*array-set a h* (*AList.delete k m*)) (*if deleted then n* − *1 else n*))

lemma *hm-grow-gt-1* [*iff*]:
  Suc 0 < *hm-grow hm*
by(*cases hm*)(*simp*)

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok len h* [] = *True*
by(*simp add*: *bucket-ok-def*)

lemma *bucket-okD*:
  ⟦ *bucket-ok len h xs*; (*k*, *v*) ∈ *set xs* ⟧
  ⟹ *bounded-hashcode-nat len k* = *h*
by(*auto simp add*: *bucket-ok-def*)

lemma *bucket-okI*:
($\bigwedge k.\ k \in fst$ ‘ $set\ kvs \Longrightarrow bounded\text{-}hashcode\text{-}nat\ len\ k = h) \Longrightarrow bucket\text{-}ok\ len\ h\ kvs$
by(*simp add*: *bucket-ok-def*)

ahm-invar

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux n a*
obtains $\forall h.\ h < array\text{-}length\ a \longrightarrow bucket\text{-}ok\ (array\text{-}length\ a)\ h\ (array\text{-}get\ a\ h) \wedge distinct\ (map\ fst\ (array\text{-}get\ a\ h))$
and $n = array\text{-}foldl\ (\lambda\text{-}\ n\ kvs.\ n + length\ kvs)\ 0\ a$ **and** $array\text{-}length\ a > 1$
using *assms* **unfolding** *ahm-invar-aux-def* **by** *blast*

lemma *ahm-invar-auxI*:
$[\![\ \bigwedge h.\ h < array\text{-}length\ a \Longrightarrow bucket\text{-}ok\ (array\text{-}length\ a)\ h\ (array\text{-}get\ a\ h);$
$\bigwedge h.\ h < array\text{-}length\ a \Longrightarrow distinct\ (map\ fst\ (array\text{-}get\ a\ h));$
$n = array\text{-}foldl\ (\lambda\text{-}\ n\ kvs.\ n + length\ kvs)\ 0\ a;\ array\text{-}length\ a > 1\ ]\!]$
$\Longrightarrow ahm\text{-}invar\text{-}aux\ n\ a$
unfolding *ahm-invar-aux-def* **by** *blast*

lemma *ahm-invar-distinct-fst-concatD*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *distinct* (*map fst* (*concat xs*))
proof −
{ fix *h*
assume $h < length\ xs$
with *inv* **have** *bucket-ok* (*length xs*) *h* (*xs* ! *h*)   *distinct* (*map fst* (*xs* ! *h*))
by(*simp-all add*: *ahm-invar-aux-def*) **}**
note *no-junk* = *this*

show *?thesis* **unfolding** *map-concat*
proof(*rule distinct-concat′*)
have *distinct* $[x{\leftarrow}xs\ .\ x \neq []]$ **unfolding** *distinct-conv-nth*
proof(*intro allI ballI impI*)
fix *i j*
assume $i < length\ [x{\leftarrow}xs\ .\ x \neq []]$ $j < length\ [x{\leftarrow}xs\ .\ x \neq []]$ $i \neq j$
from *filter-nth-ex-nth*[*OF* ‹$i < length\ [x{\leftarrow}xs\ .\ x \neq []]$›]
obtain i' **where** $i' \geq i$ $i' < length\ xs$ **and** *ith*: $[x{\leftarrow}xs\ .\ x \neq []]\ !\ i = xs\ !\ i'$
and *eqi*: $[x{\leftarrow}take\ i'\ xs\ .\ x \neq []] = take\ i\ [x{\leftarrow}xs\ .\ x \neq []]$ **by** *blast*
from *filter-nth-ex-nth*[*OF* ‹$j < length\ [x{\leftarrow}xs\ .\ x \neq []]$›]
obtain j' **where** $j' \geq j$ $j' < length\ xs$ **and** *jth*: $[x{\leftarrow}xs\ .\ x \neq []]\ !\ j = xs\ !\ j'$
and *eqj*: $[x{\leftarrow}take\ j'\ xs\ .\ x \neq []] = take\ j\ [x{\leftarrow}xs\ .\ x \neq []]$ **by** *blast*
show $[x{\leftarrow}xs\ .\ x \neq []]\ !\ i \neq [x{\leftarrow}xs\ .\ x \neq []]\ !\ j$
proof
assume $[x{\leftarrow}xs\ .\ x \neq []]\ !\ i = [x{\leftarrow}xs\ .\ x \neq []]\ !\ j$
hence *eq*: $xs\ !\ i' = xs\ !\ j'$ **using** *ith jth* **by** *simp*
from ‹$i < length\ [x{\leftarrow}xs\ .\ x \neq []]$›
have $[x{\leftarrow}xs\ .\ x \neq []]\ !\ i \in set\ [x{\leftarrow}xs\ .\ x \neq []]$ **by**(*rule nth-mem*)

with *ith* **have** $xs\ !\ i' \neq []$ **by** *simp*
then obtain *kv* **where** $kv \in set\ (xs\ !\ i')$ **by**(*fastforce simp add*: *neq-Nil-conv*)
with *no-junk*[*OF* ‹$i' < length\ xs$›] **have** *bounded-hashcode-nat* (*length xs*) (*fst kv*) $= i'$
by(*simp add*: *bucket-ok-def*)
moreover from *eq* ‹$kv \in set\ (xs\ !\ i')$› **have** $kv \in set\ (xs\ !\ j')$ **by** *simp*
with *no-junk*[*OF* ‹$j' < length\ xs$›] **have** *bounded-hashcode-nat* (*length xs*) (*fst kv*) $= j'$
by(*simp add*: *bucket-ok-def*)
ultimately have [*simp*]: $i' = j'$ **by** *simp*
from ‹$i < length\ [x{\leftarrow}xs\ .\ x \neq []]$› **have** $i = length\ (take\ i\ [x{\leftarrow}xs\ .\ x \neq []])$ **by** *simp*
also from *eqi eqj* **have** $take\ i\ [x{\leftarrow}xs\ .\ x \neq []] = take\ j\ [x{\leftarrow}xs\ .\ x \neq []]$ **by** *simp*
finally show *False* **using** ‹$i \neq j$› ‹$j < length\ [x{\leftarrow}xs\ .\ x \neq []]$› **by** *simp*
qed
qed
moreover have *inj-on* (*map fst*) $\{x \in set\ xs.\ x \neq []\}$
proof(*rule inj-onI*)
fix *x y*
assume $x \in \{x \in set\ xs.\ x \neq []\}$ $\quad y \in \{x \in set\ xs.\ x \neq []\}$ $\quad map\ fst\ x = map\ fst\ y$
hence $x \in set\ xs$ $\quad y \in set\ xs$ $\quad x \neq []$ $\quad y \neq []$ **by** *auto*
from ‹$x \in set\ xs$› **obtain** *i* **where** $xs\ !\ i = x$ $\quad i < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$y \in set\ xs$› **obtain** *j* **where** $xs\ !\ j = y$ $\quad j < length\ xs$ **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$x \neq []$› **obtain** $k\ v\ x'$ **where** $x = (k,\ v)\ \#\ x'$ **by**(*cases x*) *auto*
with *no-junk*[*OF* ‹$i < length\ xs$›] ‹$xs\ !\ i = x$›
have *bounded-hashcode-nat* (*length xs*) $k = i$ **by**(*auto simp add*: *bucket-ok-def*)
moreover from ‹$map\ fst\ x = map\ fst\ y$› ‹$x = (k,\ v)\ \#\ x'$› **obtain** v' **where** $(k,\ v') \in set\ y$ **by** *fastforce*
with *no-junk*[*OF* ‹$j < length\ xs$›] ‹$xs\ !\ j = y$›
have *bounded-hashcode-nat* (*length xs*) $k = j$ **by**(*auto simp add*: *bucket-ok-def*)
ultimately have $i = j$ **by** *simp*
with ‹$xs\ !\ i = x$› ‹$xs\ !\ j = y$› **show** $x = y$ **by** *simp*
qed
ultimately show *distinct* $[ys{\leftarrow}map\ (map\ fst)\ xs\ .\ ys \neq []]$
by(*simp add*: *filter-map o-def distinct-map*)
next
fix *ys*
assume $ys \in set\ (map\ (map\ fst)\ xs)$
thus *distinct ys* **by**(*clarsimp simp add*: *set-conv-nth*)(*rule no-junk*)
next
fix *ys zs*
assume $ys \in set\ (map\ (map\ fst)\ xs)$ $\quad zs \in set\ (map\ (map\ fst)\ xs)$ $\quad ys \neq zs$
then obtain $ys'\ zs'$ **where** [*simp*]: $ys = map\ fst\ ys'$ $\quad zs = map\ fst\ zs'$
and $ys' \in set\ xs$ $\quad zs' \in set\ xs$ **by** *auto*
have $fst\ `\ set\ ys' \cap fst\ `\ set\ zs' = \{\}$

proof(*rule equals0I*)
fix *k*
assume $k \in$ *fst* ‘ *set ys*′ ∩ *fst* ‘ *set zs*′
then obtain *v v*′ **where** $(k, v) \in$ *set ys*′ $(k, v') \in$ *set zs*′ **by**(*auto*)
from ‹*ys*′ ∈ *set xs*› **obtain** *i* **where** *xs* ! *i* = *ys*′ *i* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v) \in$ *set ys*′› **have** *bounded-hashcode-nat* (*length xs*) *k* = *i* **by**(*auto dest*: *no-junk bucket-okD*)
moreover
from ‹*zs*′ ∈ *set xs*› **obtain** *j* **where** *xs* ! *j* = *zs*′ *j* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹$(k, v') \in$ *set zs*′› **have** *bounded-hashcode-nat* (*length xs*) *k* = *j* **by**(*auto dest*: *no-junk bucket-okD*)
ultimately have *i* = *j* **by** *simp*
with ‹*xs* ! *i* = *ys*′› ‹*xs* ! *j* = *zs*′› **have** *ys*′ = *zs*′ **by** *simp*
with ‹*ys* ≠ *zs*› **show** *False* **by** *simp*
qed
thus *set ys* ∩ *set zs* = {} **by** *simp*
qed
qed

ahm-α

lemma *finite-dom-ahm-α-aux*:
assumes *ahm-invar-aux n a*
shows *finite* (*dom* (*ahm-α-aux a*))
proof −
have *dom* (*ahm-α-aux a*) ⊆ (⋃*h* ∈ *range* (*bounded-hashcode-nat* (*array-length a*) :: ′*a* ⇒ *nat*). *dom* (*map-of* (*array-get a h*)))
by(*force simp add*: *dom-map-of-conv-image-fst ahm-α-aux-def dest*: *map-of-SomeD*)
moreover have *finite* . . .
proof(*rule finite-UN-I*)
from ‹*ahm-invar-aux n a*› **have** *array-length a* > *1* **by**(*simp add*: *ahm-invar-aux-def*)
hence *range* (*bounded-hashcode-nat* (*array-length a*) :: ′*a* ⇒ *nat*) ⊆ {*0*..<*array-length a*}
by(*auto simp add*: *bounded-hashcode-nat-bounds*)
thus *finite* (*range* (*bounded-hashcode-nat* (*array-length a*) :: ′*a* ⇒ *nat*))
by(*rule finite-subset*) *simp*
qed(*rule finite-dom-map-of*)
ultimately show *?thesis* **by**(*rule finite-subset*)
qed

lemma *ahm-α-aux-conv-map-of-concat*:
assumes *inv*: *ahm-invar-aux n* (*Array xs*)
shows *ahm-α-aux* (*Array xs*) = *map-of* (*concat xs*)
proof
fix *k*
show *ahm-α-aux* (*Array xs*) *k* = *map-of* (*concat xs*) *k*
proof(*cases map-of* (*concat xs*) *k*)

case *None*
hence $k \notin$ *fst* ‘ *set* (*concat xs*) **by**(*simp add*: *map-of-eq-None-iff*)
hence $k \notin$ *fst* ‘ *set* (*xs* ! *bounded-hashcode-nat* (*length xs*) *k*)
proof(*rule contrapos-nn*)
assume $k \in$ *fst* ‘ *set* (*xs* ! *bounded-hashcode-nat* (*length xs*) *k*)
then obtain *v* **where** $(k, v) \in$ *set* (*xs* ! *bounded-hashcode-nat* (*length xs*) *k*)
by *auto*
moreover from *inv* **have** *bounded-hashcode-nat* (*length xs*) $k <$ *length xs*
by(*simp add*: *bounded-hashcode-nat-bounds ahm-invar-aux-def*)
ultimately show $k \in$ *fst* ‘ *set* (*concat xs*)
by(*force intro*: *rev-image-eqI*)
qed
thus *?thesis* **unfolding** *None* **by**(*simp add*: *map-of-eq-None-iff*)
next
case (*Some v*)
hence $(k, v) \in$ *set* (*concat xs*) **by**(*rule map-of-SomeD*)
then obtain *ys* **where** *ys* $\in$ *set xs* $(k, v) \in$ *set ys*
unfolding *set-concat* **by** *blast*
from ‹*ys* $\in$ *set xs*› **obtain** *i j* **where** $i <$ *length xs* *xs* ! $i =$ *ys*
unfolding *set-conv-nth* **by** *auto*
with *inv* ‹$(k, v) \in$ *set ys*›
show *?thesis* **unfolding** *Some*
by(*force dest*: *bucket-okD simp add*: *ahm-invar-aux-def*)
qed
qed

lemma *ahm-invar-aux-card-dom-ahm-α-auxD*:
assumes *inv*: *ahm-invar-aux n a*
shows *card* (*dom* (*ahm-α-aux a*)) $=$ *n*
proof(*cases a*)
case [*simp*]: (*Array xs*)
from *inv* **have** *card* (*dom* (*ahm-α-aux* (*Array xs*))) $=$ *card* (*dom* (*map-of* (*concat xs*)))
by(*simp add*: *ahm-α-aux-conv-map-of-concat*)
also from *inv* **have** *distinct* (*map fst* (*concat xs*))
by(*simp add*: *ahm-invar-distinct-fst-concatD*)
hence *card* (*dom* (*map-of* (*concat xs*))) $=$ *length* (*concat xs*)
by(*rule card-dom-map-of*)
also have *length* (*concat xs*) $=$ *foldl* $(+)$ *0* (*map length xs*)
by (*simp add*: *length-concat foldl-conv-fold add.commute fold-plus-sum-list-rev*)
also from *inv*
have $\ldots = n$ **unfolding** *foldl-map* **by**(*simp add*: *ahm-invar-aux-def array-foldl-foldl*)
finally show *?thesis* **by**(*simp*)
qed

lemma *finite-dom-ahm-α*:
ahm-invar hm $\Longrightarrow$ *finite* (*dom* (*ahm-α hm*))
by(*cases hm*)(*auto intro*: *finite-dom-ahm-α-aux*)

lemma *finite-map-ahm-α-aux*:
finite-map ahm-α-aux (*ahm-invar-aux n*)
by(*unfold-locales*)(*rule finite-dom-ahm-α-aux*)

lemma *finite-map-ahm-α*:
finite-map ahm-α ahm-invar
by(*unfold-locales*)(*rule finite-dom-ahm-α*)

ahm-empty

lemma *ahm-invar-aux-new-array*:
assumes $n > 1$
shows *ahm-invar-aux 0* (*new-array* [] *n*)
proof −
have *foldl* (λ*b* (*k*, *v*). *b* + *length v*) *0* (*zip* [*0..<n*] (*replicate n* [])) = *0*
by(*induct n*)(*simp-all add*: *replicate-Suc-conv-snoc del*: *replicate-Suc*)
with *assms* **show** *?thesis* **by**(*simp add*: *ahm-invar-aux-def array-foldl-new-array*)
qed

lemma *ahm-invar-new-hashmap-with*:
$n > 1$ ⟹ *ahm-invar* (*new-hashmap-with n*)
by(*auto simp add*: *ahm-invar-def new-hashmap-with-def intro*: *ahm-invar-aux-new-array*)

lemma *ahm-α-new-hashmap-with*:
$n > 1$ ⟹ *ahm-α* (*new-hashmap-with n*) = *Map.empty*
by(*simp add*: *new-hashmap-with-def bounded-hashcode-nat-bounds fun-eq-iff*)

lemma *ahm-invar-ahm-empty* [*simp*]: *ahm-invar* (*ahm-empty* ())
using *def-hashmap-size*[**where** *?'a* = *'a*]
by(*auto intro*: *ahm-invar-new-hashmap-with simp add*: *ahm-empty-def*)

lemma *ahm-empty-correct* [*simp*]: *ahm-α* (*ahm-empty* ()) = *Map.empty*
using *def-hashmap-size*[**where** *?'a* = *'a*]
by(*auto intro*: *ahm-α-new-hashmap-with simp add*: *ahm-empty-def*)

lemma *ahm-empty-impl*: *map-empty ahm-α ahm-invar ahm-empty*
by(*unfold-locales*)(*auto*)

ahm-lookup

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

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 −
obtain *ms* **where** *m-eq*[*simp*]: *m* = *Array ms* **by** (*cases m*)

from *ahm-invar-distinct-fst-concatD*[*of n ms*] *invar-m*
have *dist*: *distinct* (*map fst* (*concat ms*)) **by** *simp*

show *map-iterator* (*ahm-iteratei-aux m*) (*ahm-α-aux m*)
using *set-iterator-foldli-correct*[*of concat ms*] *dist*
by (*simp add*: *ahm-α-aux-conv-map-of-concat*[*OF invar-m*[*unfolded m-eq*]]
ahm-iteratei-aux-def map-to-set-map-of[*OF dist*] *distinct-map*)
qed

lemma *ahm-iteratei-correct*:
assumes *invar-hm*: *ahm-invar hm*
shows *map-iterator* (*ahm-iteratei hm*) (*ahm-α hm*)
proof −
obtain *A n* **where** *hm-eq* [*simp*]: *hm* = *HashMap A n* **by**(*cases hm*)

from *ahm-iteratei-aux-impl*[*of n A*] *invar-hm*
show *map-it*: *map-iterator* (*ahm-iteratei hm*) (*ahm-α hm*) **by** *simp*
qed

lemma *ahm-iteratei-aux-code* [*code*]:
ahm-iteratei-aux a c f σ = *a-idx-it.idx-iteratei a c* (*λx. foldli x c f*) *σ*
proof(*cases a*)
case [*simp*]: (*Array xs*)

have *ahm-iteratei-aux a c f σ* = *foldli* (*concat xs*) *c f σ* **by** *simp*
also have ... = *foldli xs c* (*λx. foldli x c f*) *σ* **by** (*simp add*: *foldli-concat*)
thm *a-idx-it.idx-iteratei-correct*
also have ... = *a-idx-it.idx-iteratei a c* (*λx. foldli x c f*) *σ*
by (*simp add*: *a-idx-it.idx-iteratei-correct*)
finally show *?thesis* **.**
qed

ahm-rehash

lemma *array-length-ahm-rehash-aux′*:
array-length (*ahm-rehash-aux′ n kv a*) = *array-length a*
by(*simp add*: *ahm-rehash-aux′-def Let-def*)

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
assumes *inv*: *ahm-invar-aux n a*
and *fresh*: *k* ∉ *fst* ‘ *set* (*array-get a* (*bounded-hashcode-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(*rule ahm-invar-auxI*)
fix *h*
assume *h* < *array-length ?a*
hence *hlen*: *h* < *array-length a* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
with *inv* **have** *bucket*: *bucket-ok* (*array-length a*) *h* (*array-get a h*)

and *dist*: *distinct* (*map fst* (*array-get a h*))
by(*auto elim*: *ahm-invar-auxE*)
let *?h* = *bounded-hashcode-nat* (*array-length a*) *k*
from *hlen bucket* **show** *bucket-ok* (*array-length ?a*) *h* (*array-get ?a h*)
by(*cases h* = *?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-length-ahm-rehash-aux′ array-get-array-set-other dest*: *bucket-okD intro*!: *bucket-okI*)
from *dist hlen fresh*
show *distinct* (*map fst* (*array-get ?a h*))
by(*cases h* = *?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-get-array-set-other*)
next
let *?f* = *λn kvs. n* + *length kvs*
{ fix *n* :: *nat* **and** *xs* :: (*′a* × *′b*) *list list*
have *foldl ?f n xs* = *n* + *foldl ?f 0 xs*
by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
note *fold* = *this*
let *?h* = *bounded-hashcode-nat* (*array-length a*) *k*

obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)
from *inv* **have** [*simp*]: *bounded-hashcode-nat* (*length xs*) *k* < *length xs*
by(*simp add*: *ahm-invar-aux-def bounded-hashcode-nat-bounds*)
have *xs*: *xs* = *take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs* **by**(*simp add*: *Cons-nth-drop-Suc*)
from *inv* **have** *n* = *array-foldl* (*λ- n kvs. n* + *length kvs*) *0 a*
by(*auto elim*: *ahm-invar-auxE*)
hence *n* = *foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc ?h*) *xs*)
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
thus *Suc n* = *array-foldl* (*λ- n kvs. n* + *length kvs*) *0 ?a*
by(*simp add*: *ahm-rehash-aux′-def Let-def array-foldl-foldl foldl-list-update*)(*subst* (*1 2 3 4*) *fold*, *simp*)
next
from *inv* **have** *1* < *array-length a* **by**(*auto elim*: *ahm-invar-auxE*)
thus *1* < *array-length ?a* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
qed

declare [[*coercion-enabled* = *false*]]

lemma *ahm-rehash-aux-correct*:
fixes *a* :: ((*′key* :: *hashable*) × *′val*) *list array*
assumes *inv*: *ahm-invar-aux n a*
and *sz* > *1*
shows *ahm-invar-aux n* (*ahm-rehash-aux a sz*) (**is** *?thesis1*)
and *ahm-α-aux* (*ahm-rehash-aux a sz*) = *ahm-α-aux a* (**is** *?thesis2*)
proof −

let *?a* = *ahm-rehash-aux a sz*
let *?I* = *λit a′. ahm-invar-aux* (*n* − *card it*) *a′* ∧ *array-length a′* = *sz* ∧ (∀ *k. if k* ∈ *it then ahm-α-aux a′ k* = *None else ahm-α-aux a′ k* = *ahm-α-aux a k*)
have *?I* {} *?a* ∨ (∃ *it*⊆*dom* (*ahm-α-aux a*). *it* ≠ {} ∧ ¬ *True* ∧ *?I it ?a*)
unfolding *ahm-rehash-aux-def*

proof (*rule map-iterator-rule-P*[*OF ahm-iteratei-aux-impl*[*OF inv*], **where**
$c = \lambda$-. *True* **and** f=*ahm-rehash-aux'* *sz* **and** *?σ0.0* = *new-array* [] *sz*
and *I*=*?I*]
)

from *inv* **have** *card* (*dom* (*ahm-α-aux a*)) = *n* **by**(*rule ahm-invar-aux-card-dom-ahm-α-auxD*)
moreover from ‹*1* < *sz*› **have** *ahm-invar-aux 0* (*new-array* ([] :: ('*key* × '*val*)
list) *sz*)
by(*rule ahm-invar-aux-new-array*)
moreover {
fix *k*
assume *k* ∉ *dom* (*ahm-α-aux a*)
hence *ahm-α-aux a k* = *None* **by** *auto*
moreover have *bounded-hashcode-nat sz k* < *sz* **using** ‹*1* < *sz*›
by(*simp add*: *bounded-hashcode-nat-bounds*)
ultimately have *ahm-α-aux* (*new-array* [] *sz*) *k* = *ahm-α-aux a k* **by** *simp* **}**
ultimately show *?I* (*dom* (*ahm-α-aux a*)) (*new-array* [] *sz*)
by(*auto simp add*: *bounded-hashcode-nat-bounds*[*OF* ‹*1* < *sz*›])
next
fix *k* :: '*key*
and *v* :: '*val*
and *it a'*
assume *k* ∈ *it* *ahm-α-aux a k* = *Some v*
and *it-sub*: *it* ⊆ *dom* (*ahm-α-aux a*)
and *I*: *?I it a'*
from *I* **have** *inv'*: *ahm-invar-aux* (*n* − *card it*) *a'*
and *a'-eq-a*: ⋀*k*. *k* ∉ *it* ⟹ *ahm-α-aux a' k* = *ahm-α-aux a k*
and *a'-None*: ⋀*k*. *k* ∈ *it* ⟹ *ahm-α-aux a' k* = *None*
and [*simp*]: *sz* = *array-length a'* **by**(*auto split*: *if-split-asm*)
from *it-sub finite-dom-ahm-α-aux*[*OF inv*] **have** *finite it* **by**(*rule finite-subset*)
moreover with ‹*k* ∈ *it*› **have** *card it* > *0* **by**(*auto simp add*: *card-gt-0-iff*)
moreover from *finite-dom-ahm-α-aux*[*OF inv*] *it-sub*
have *card it* ≤ *card* (*dom* (*ahm-α-aux a*)) **by**(*rule card-mono*)
moreover have ... = *n* **using** *inv*
by(*simp add*: *ahm-invar-aux-card-dom-ahm-α-auxD*)
ultimately have *n* − *card* (*it* − {*k*}) = (*n* − *card it*) + *1* **using** ‹*k* ∈ *it*› **by**
auto
moreover from ‹*k* ∈ *it*› **have** *ahm-α-aux a' k* = *None* **by**(*rule a'-None*)
hence *k* ∉ *fst* ‘ *set* (*array-get a'* (*bounded-hashcode-nat* (*array-length a'*) *k*))
by(*simp add*: *map-of-eq-None-iff*)
ultimately have *ahm-invar-aux* (*n* − *card* (*it* − {*k*})) (*ahm-rehash-aux' sz*
(*k*, *v*) *a'*)
using *inv'* **by**(*auto intro*: *ahm-rehash-aux'-preserves-ahm-invar-aux*)
moreover have *array-length* (*ahm-rehash-aux' sz* (*k*, *v*) *a'*) = *sz*
by(*simp add*: *array-length-ahm-rehash-aux'*)
moreover {
fix *k'*
assume *k'* ∈ *it* − {*k*}

with *bounded-hashcode-nat-bounds*[*OF* ‹*1* < *sz*›, *of k*′] *a*′*-None*[*of k*′]
have *ahm-α-aux* (*ahm-rehash-aux*′ *sz* (*k*, *v*) *a*′) *k*′ = *None*
by(*cases bounded-hashcode-nat sz k* = *bounded-hashcode-nat sz k*′)(*auto simp add*: *array-get-array-set-other ahm-rehash-aux*′*-def Let-def*)
} moreover {
fix *k*′
assume *k*′ ∉ *it* − {*k*}
with *bounded-hashcode-nat-bounds*[*OF* ‹*1* < *sz*›, *of k*′] *bounded-hashcode-nat-bounds*[*OF* ‹*1* < *sz*›, *of k*] *a*′*-eq-a*[*of k*′] ‹*k* ∈ *it*›
have *ahm-α-aux* (*ahm-rehash-aux*′ *sz* (*k*, *v*) *a*′) *k*′ = *ahm-α-aux a k*′
unfolding *ahm-rehash-aux*′*-def Let-def* **using** ‹*ahm-α-aux a k* = *Some v*›
by(*cases bounded-hashcode-nat sz k* = *bounded-hashcode-nat sz k*′)(*case-tac* [!] *k*′ = *k*, *simp-all add*: *array-get-array-set-other*) **}**
ultimately show *?I* (*it* − {*k*}) (*ahm-rehash-aux*′ *sz* (*k*, *v*) *a*′) **by** *simp*
qed *auto*
thus *?thesis1 ?thesis2* **unfolding** *ahm-rehash-aux-def*
by(*auto intro*: *ext*)
qed

lemma *ahm-rehash-correct*:
fixes *hm* :: (′*key* :: *hashable*, ′*val*) *hashmap*
assumes *inv*: *ahm-invar hm*
and *sz* > *1*
shows *ahm-invar* (*ahm-rehash hm sz*) *ahm-α* (*ahm-rehash hm sz*) = *ahm-α hm*
using *assms*
by −(*case-tac* [!] *hm*, *auto intro*: *ahm-rehash-aux-correct*)

ahm-update

lemma *ahm-update-aux-correct*:
assumes *inv*: *ahm-invar hm*
shows *ahm-invar* (*ahm-update-aux hm k v*) (**is** *?thesis1*)
and *ahm-α* (*ahm-update-aux hm k v*) = (*ahm-α hm*)(*k* ↦ *v*) (**is** *?thesis2*)
proof −
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by**(*cases hm*)

let *?h* = *bounded-hashcode-nat* (*array-length a*) *k*
let *?a*′ = *array-set a ?h* (*AList.update k v* (*array-get a ?h*))
let *?n*′ = *if map-of* (*array-get a ?h*) *k* = *None then n* + *1 else n*

have *ahm-invar* (*HashMap ?a*′ *?n*′) **unfolding** *ahm-invar.simps*
proof(*rule ahm-invar-auxI*)
fix *h*
assume *h* < *array-length ?a*′
hence *h* < *array-length a* **by** *simp*
with *inv* **have** *bucket-ok* (*array-length a*) *h* (*array-get a h*)
by(*auto elim*: *ahm-invar-auxE*)
thus *bucket-ok* (*array-length ?a*′) *h* (*array-get ?a*′ *h*)

```
    using ‹h < array-length a›
    apply(cases h = bounded-hashcode-nat (array-length a) k)
   apply(fastforce intro!: bucket-okI simp add: dom-update array-get-array-set-other
dest: bucket-okD del: imageE elim: imageE)+
    done
  from ‹h < array-length a› inv have distinct (map fst (array-get a h))
    by(auto elim: ahm-invar-auxE)
  with ‹h < array-length a›
  show distinct (map fst (array-get ?a' h))
     by(cases h = bounded-hashcode-nat (array-length a) k)(auto simp add: ar-
ray-get-array-set-other intro: distinct-update)
 next
  obtain xs where a [simp]: a = Array xs by(cases a)

  let ?f = λn kvs. n + length kvs
  { fix n :: nat and xs :: ('a × 'b) list list
    have foldl ?f n xs = n + foldl ?f 0 xs
      by(induct xs arbitrary:  rule: rev-induct) simp-all }
  note fold = this

  from inv have [simp]: bounded-hashcode-nat (length xs) k < length xs
    by(simp add: ahm-invar-aux-def bounded-hashcode-nat-bounds)
    have xs: xs = take ?h xs @ (xs ! ?h) # drop (Suc ?h) xs by(simp add:
Cons-nth-drop-Suc)
  from inv have n = array-foldl (λ- n kvs. n + length kvs) 0 a
    by(auto elim: ahm-invar-auxE)
   hence n = foldl ?f 0 (take ?h xs) + length (xs ! ?h) + foldl ?f 0 (drop (Suc
?h) xs)
    by(simp add: array-foldl-foldl)(subst xs, simp, subst (1 2 3 4) fold, simp)
  thus ?n' = array-foldl (λ- n kvs. n + length kvs) 0 ?a'
    apply(simp add: ahm-rehash-aux'-def Let-def array-foldl-foldl foldl-list-update
map-of-eq-None-iff)
    apply(subst (1 2 3 4 5 6 7 8) fold)
    apply(simp add: length-update)
    done
 next
  from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
  thus 1 < array-length ?a' by simp
 qed
 moreover have ahm-α (ahm-update-aux hm k v) = (ahm-α hm)(k ↦ v)
 proof
  fix k'
  from inv have 1 < array-length a by(auto elim: ahm-invar-auxE)
  with bounded-hashcode-nat-bounds[OF this, of k]
  show ahm-α (ahm-update-aux hm k v) k' = ((ahm-α hm)(k ↦ v)) k'
     by(cases bounded-hashcode-nat (array-length a) k = bounded-hashcode-nat
(array-length a) k')(auto simp add: Let-def update-conv array-get-array-set-other)
 qed
 ultimately show ?thesis1 ?thesis2 by(simp-all add: Let-def)
```

qed

lemma *ahm-update-correct*:
  assumes *inv*: *ahm-invar hm*
  shows *ahm-invar* (*ahm-update k v hm*)
  and *ahm-α* (*ahm-update k v hm*) = (*ahm-α hm*)(*k* ↦ *v*)
using *assms*
by(*simp-all add*: *ahm-update-def Let-def ahm-rehash-correct ahm-update-aux-correct*)

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

ahm-delete

lemma *ahm-delete-correct*:
  assumes *inv*: *ahm-invar hm*
  shows *ahm-invar* (*ahm-delete k hm*) (**is** *?thesis1*)
  and *ahm-α* (*ahm-delete k hm*) = (*ahm-α hm*) |‘ (− {*k*}) (**is** *?thesis2*)
proof −
  obtain *a n* **where** *hm* [*simp*]: *hm* = *HashMap a n* **by**(*cases hm*)

  let *?h* = *bounded-hashcode-nat* (*array-length a*) *k*
  let *?a′* = *array-set a ?h* (*AList.delete k* (*array-get a ?h*))
  let *?n′* = *if map-of* (*array-get a* (*bounded-hashcode-nat* (*array-length a*) *k*)) *k* = *None then n else n* − *1*

  have *ahm-invar-aux ?n′ ?a′*
  proof(*rule ahm-invar-auxI*)
    fix *h*
    assume *h* < *array-length ?a′*
    hence *h* < *array-length a* **by** *simp*
    with *inv* **have** *bucket-ok* (*array-length a*) *h* (*array-get a h*)
      and *1* < *array-length a*
      and *distinct* (*map fst* (*array-get a h*)) **by**(*auto elim*: *ahm-invar-auxE*)
    thus *bucket-ok* (*array-length ?a′*) *h* (*array-get ?a′ h*)
      and *distinct* (*map fst* (*array-get ?a′ h*))
      using *bounded-hashcode-nat-bounds*[*of array-length a k*]
      by−(*case-tac* [!] *h* = *bounded-hashcode-nat* (*array-length a*) *k*, *auto simp add*: *array-get-array-set-other set-delete-conv intro*!: *bucket-okI dest*: *bucket-okD intro*: *distinct-delete*)
  next
    obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)

    let *?f* = *λn kvs*. *n* + *length kvs*
    { fix *n* :: *nat* **and** *xs* :: (*′a* × *′b*) *list list*
      have *foldl ?f n xs* = *n* + *foldl ?f 0 xs*
        by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
    note *fold* = *this*

from *inv* **have** [*simp*]: *bounded-hashcode-nat* (*length xs*) $k <$ *length xs*
by(*simp add*: *ahm-invar-aux-def bounded-hashcode-nat-bounds*)
from *inv* **have** *distinct* (*map fst* (*array-get a ?h*)) **by**(*auto elim*: *ahm-invar-auxE*)
moreover
have *xs*: *xs* = *take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs* **by**(*simp add*: *Cons-nth-drop-Suc*)
from *inv* **have** n = *array-foldl* (λ- *n kvs*. n + *length kvs*) *0 a*
by(*auto elim*: *ahm-invar-auxE*)
hence n = *foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc ?h*) *xs*)
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
ultimately show *?n′* = *array-foldl* (λ- *n kvs*. n + *length kvs*) *0 ?a′*
apply(*simp add*: *array-foldl-foldl foldl-list-update map-of-eq-None-iff*)
apply(*subst* (*1 2 3 4 5 6 7 8*) *fold*)
apply(*auto simp add*: *length-distinct in-set-conv-nth*)
done
next
from *inv* **show** *1* < *array-length ?a′* **by**(*auto elim*: *ahm-invar-auxE*)
qed
thus *?thesis1* **by**(*auto simp add*: *Let-def*)

have *ahm-α-aux ?a′* = *ahm-α-aux a* |‘ (− {k})
proof
fix *k′* :: *′a*
from *inv* **have** *bounded-hashcode-nat* (*array-length a*) $k <$ *array-length a*
by(*auto elim*: *ahm-invar-auxE simp add*: *bounded-hashcode-nat-bounds*)
thus *ahm-α-aux ?a′ k′* = (*ahm-α-aux a* |‘ (− {k})) *k′*
by(*cases ?h* = *bounded-hashcode-nat* (*array-length a*) *k′*)(*auto simp add*: *restrict-map-def array-get-array-set-other delete-conv*)
qed
thus *?thesis2* **by**(*simp add*: *Let-def*)
qed

lemma *ahm-delete-impl*:
map-delete ahm-α ahm-invar ahm-delete
by(*unfold-locales*)(*blast intro*: *ahm-delete-correct*)+

hide-const (**open**) *HashMap ahm-empty bucket-ok ahm-invar ahm-α ahm-lookup ahm-iteratei ahm-rehash hm-grow ahm-filled ahm-update ahm-delete*
hide-type (**open**) *hashmap*

end

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
interpret *map-empty ArrayHashMap-Impl.ahm-α ArrayHashMap-Impl.ahm-invar ArrayHashMap-Impl.ahm-empty*
by(*rule ahm-empty-impl*)
show *ArrayHashMap-Impl.ahm-empty* () ∈ *?hashmap*
by(*simp add*: *empty-correct*)
qed

type-synonym (*'k*,*'v*) *ahm* = (*'k*,*'v*) *hashmap*

lemma *ahm-invar-impl-of* [*simp*, *intro*]: *ArrayHashMap-Impl.ahm-invar* (*impl-of hm*)
using *impl-of*[*of hm*] **by** *simp*

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

Primitive operations

definition *ahm-empty-const* :: (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty-const* ≡ (*HashMap* (*ArrayHashMap-Impl.ahm-empty* ()))

definition *ahm-empty* :: *unit* ⇒ (*'key* :: *hashable*, *'val*) *hashmap*
where *ahm-empty* ≡ λ-. *ahm-empty-const*

definition *ahm-α* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'key* ⇒ *'val option*
where *ahm-α hm* = *ArrayHashMap-Impl.ahm-α* (*impl-of hm*)

definition *ahm-lookup* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ *'val option*
where *ahm-lookup k hm* = *ArrayHashMap-Impl.ahm-lookup k* (*impl-of hm*)

definition *ahm-iteratei* :: (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key* × *'val*, *'σ*) *set-iterator*
where *ahm-iteratei hm* = *ArrayHashMap-Impl.ahm-iteratei* (*impl-of hm*)

definition *ahm-update* :: *'key* ⇒ *'val* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where
ahm-update k v hm = *HashMap* (*ArrayHashMap-Impl.ahm-update k v* (*impl-of hm*))

definition *ahm-delete* :: *'key* ⇒ (*'key* :: *hashable*, *'val*) *hashmap* ⇒ (*'key*, *'val*) *hashmap*
where

ahm-delete k hm = *HashMap* (*ArrayHashMap-Impl.ahm-delete k* (*impl-of hm*))

lemma *impl-of-ahm-empty* [*code abstract*]:
impl-of ahm-empty-const = *ArrayHashMap-Impl.ahm-empty* ()
by(*simp add*: *ahm-empty-const-def HashMap-inverse*)

lemma *impl-of-ahm-update* [*code abstract*]:
impl-of (*ahm-update k v hm*) = *ArrayHashMap-Impl.ahm-update k v* (*impl-of hm*)
by(*simp add*: *ahm-update-def HashMap-inverse ahm-update-correct*)

lemma *impl-of-ahm-delete* [*code abstract*]:
impl-of (*ahm-delete k hm*) = *ArrayHashMap-Impl.ahm-delete k* (*impl-of hm*)
by(*simp add*: *ahm-delete-def HashMap-inverse ahm-delete-correct*)

lemma *finite-dom-ahm-*α[*simp*]: *finite* (*dom* (*ahm-*α *hm*))
by (*simp add*: *ahm-*α*-def finite-dom-ahm-*α)

lemma *ahm-empty-correct*[*simp*]: *ahm-*α (*ahm-empty* ()) = *Map.empty*
by(*simp add*: *ahm-*α*-def ahm-empty-def ahm-empty-const-def HashMap-inverse*)

lemma *ahm-lookup-correct*[*simp*]: *ahm-lookup k m* = *ahm-*α *m k*
by (*simp add*: *ahm-lookup-def ArrayHashMap-Impl.ahm-lookup-def ahm-*α*-def*)

lemma *ahm-update-correct*[*simp*]: *ahm-*α (*ahm-update k v hm*) = (*ahm-*α *hm*)(*k*
$\mapsto$ *v*)
by (*simp add*: *ahm-*α*-def ahm-update-def ahm-update-correct HashMap-inverse*)

lemma *ahm-delete-correct*[*simp*]:
ahm-α (*ahm-delete k hm*) = (*ahm-*α *hm*) |‘ (− {*k*})
by (*simp add*: *ahm-*α*-def ahm-delete-def HashMap-inverse ahm-delete-correct*)

lemma *ahm-iteratei-impl*[*simp*]: *map-iterator* (*ahm-iteratei m*) (*ahm-*α *m*)
unfolding *ahm-iteratei-def ahm-*α*-def*
apply (*rule ahm-iteratei-correct*)
by *simp*

ICF Integration

definition [*icf-rec-def*]: *ahm-basic-ops* ≡ ⦇
bmap-op-α = *ahm-*α,
bmap-op-invar = λ-. *True*,
bmap-op-empty = *ahm-empty*,
bmap-op-lookup = *ahm-lookup*,
bmap-op-update = *ahm-update*,
bmap-op-update-dj = *ahm-update*, — TODO: We could use a more efficient bucket update here
bmap-op-delete = *ahm-delete*,
bmap-op-list-it = *ahm-iteratei*
⦈

setup *Locale-Code.open-block*
interpretation *ahm-basic*: *StdBasicMap ahm-basic-ops*
  apply *unfold-locales*
  apply (*simp-all add*: *icf-rec-unf*)
  done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *ahm-ops* ≡ *ahm-basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *ahm*: *StdMap ahm-ops*
  unfolding *ahm-ops-def*
  by (*rule ahm-basic.dflt-ops-impl*)
interpretation *ahm*: *StdMap-no-invar ahm-ops*
  apply *unfold-locales*
  unfolding *icf-rec-unf* **..**
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs ahm*›

lemma *pi-ahm*[*proper-it*]:
  proper-it′ ahm-iteratei ahm-iteratei
  unfolding *ahm-iteratei-def*[*abs-def*]
    ArrayHashMap-Impl.ahm-iteratei-def ArrayHashMap-Impl.ahm-iteratei-aux-def
  apply (*rule proper-it′I*)
  apply (*case-tac impl-of s*)
  apply *simp*
  apply (*rename-tac array nat*)
  apply (*case-tac array*)
  apply *simp*
  by (*intro icf-proper-iteratorI*)

interpretation *pi-ahm*: *proper-it-loc ahm-iteratei ahm-iteratei*
  apply *unfold-locales*
  apply (*rule pi-ahm*)
  done

Code generator test

definition *test-codegen* **where** *test-codegen* ≡ (
  ahm.add ,
  ahm.add-dj ,
  ahm.ball ,
  ahm.bex ,
  ahm.delete ,
  ahm.empty ,
  ahm.isEmpty ,
  ahm.isSng ,
  ahm.iterate ,

ahm.iteratei ,
ahm.list-it ,
ahm.lookup ,
ahm.restrict ,
ahm.sel ,
ahm.size ,
ahm.size-abort ,
ahm.sng ,
ahm.to-list ,
ahm.to-map ,
ahm.update ,
ahm.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.11 Maps from Naturals by Arrays

theory *ArrayMapImpl*
imports
../spec/MapSpec
../gen-algo/MapGA
../../Lib/Diff-Array
begin **type-synonym** *'v iam* = *'v option array*

Definitions

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

lemma [*code*]: *iam-α a i* ≡ *array-get-oo None a i*
unfolding *iam-α-def array-get-oo-def* **.**

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

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

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

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

lemma *incr-correct*: ¬ *idx* < *l* ⟹ *idx* < *l* + *iam-increment l idx*
unfolding *iam-increment-def* **by** *auto*

definition *iam-update* :: *nat* ⇒ *'v* ⇒ *'v iam* ⇒ *'v iam*

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

lemma [*code*]: *iam-update k v a* ≡ *array-set-oo*
(*λ-. array-set*
(*array-grow a* (*iam-increment* (*array-length a*) *k*) *None*) *k* (*Some v*))
a k (*Some v*)

unfolding *iam-update-def array-set-oo-def*
apply (*rule eq-reflection*)
apply (*auto simp add*: *Let-def*)
done

definition *iam-update-dj* ≡ *iam-update*

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

lemma [*code*]: *iam-delete k a* ≡ *array-set-oo* (*λ-. a*) *a k None*
unfolding *iam-delete-def array-set-oo-def* **by** *auto*

fun *iam-rev-iterateoi-aux*
:: *nat* ⇒ (*'v iam*) ⇒ (*'σ*⇒*bool*) ⇒ (*nat* × *'v*⇒*'σ*⇒*'σ*) ⇒ *'σ* ⇒ *'σ*
where
iam-rev-iterateoi-aux 0 a c f σ = *σ*
| *iam-rev-iterateoi-aux i a c f σ* = (
if c σ then
iam-rev-iterateoi-aux (*i* − *1*) *a c f* (
case array-get a (*i* − *1*) *of None* ⇒ *σ* | *Some x* ⇒ *f* (*i* − *1*, *x*) *σ*
)
else σ)

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

function *iam-iterateoi-aux*
:: *nat* ⇒ *nat* ⇒ (*'v iam*) ⇒ (*'σ*⇒*bool*) ⇒ (*nat* × *'v*⇒*'σ*⇒*'σ*) ⇒ *'σ* ⇒ *'σ*
where
iam-iterateoi-aux i len a c f σ =
(*if i* ≥ *len* ∨ ¬ *c σ then σ else let*
σ' = (*case array-get a i of*
None ⇒ *σ*
| *Some x* ⇒ *f* (*i*,*x*) *σ*)
in iam-iterateoi-aux (*i* + *1*) *len a c f σ'*)

by *pat-completeness auto*
termination
by (*relation measure* (λ(*i*,*l*,-). *l* − *i*)) *auto*

declare *iam-iterateoi-aux.simps*[*simp del*]

lemma *iam-iterateoi-aux-csimps*:
i ≥ *len* ⟹ *iam-iterateoi-aux i len a c f* σ = σ
¬ *c* σ ⟹ *iam-iterateoi-aux i len a c f* σ = σ
⟦ *i*< *len*; *c* σ ⟧ ⟹ *iam-iterateoi-aux i len a c f* σ =
(*case array-get a i of*
None ⇒ *iam-iterateoi-aux* (*i* + *1*) *len a c f* σ
| *Some x* ⇒ *iam-iterateoi-aux* (*i* + *1*) *len a c f* (*f* (*i*,*x*) σ))
apply (*subst iam-iterateoi-aux.simps*, *simp*)
apply (*subst iam-iterateoi-aux.simps*, *simp*)
apply (*subst iam-iterateoi-aux.simps*)
apply (*auto split*: *option.split-asm option.split*)
done

definition *iam-iterateoi* :: ′*v iam* ⇒ (*nat* × ′*v*,′σ) *set-iterator* **where**
iam-iterateoi a = *iam-iterateoi-aux 0* (*array-length a*) *a*

lemma *iam-empty-impl*: *map-empty iam-*α *iam-invar iam-empty*
apply *unfold-locales*
unfolding *iam-*α*-def*[*abs-def*] *iam-empty-def*
by *auto*

lemma *iam-lookup-impl*: *map-lookup iam-*α *iam-invar iam-lookup*
apply *unfold-locales*
unfolding *iam-*α*-def*[*abs-def*] *iam-lookup-def*
by *auto*

lemma *array-get-set-iff*: *i*<*array-length a* ⟹
array-get (*array-set a i x*) *j* = (*if i*=*j then x else array-get a j*)
by (*auto simp*: *array-get-array-set-other*)

lemma *iam-update-impl*: *map-update iam-*α *iam-invar iam-update*
apply *unfold-locales*
unfolding *iam-*α*-def*[*abs-def*] *iam-update-def*
apply (*rule ext*)
apply (*auto simp*: *Let-def array-get-set-iff incr-correct*)
done

lemma *iam-update-dj-impl*: *map-update-dj iam-*α *iam-invar iam-update-dj*
apply (*unfold iam-update-dj-def*)
apply (*rule update-dj-by-update*)
apply (*rule iam-update-impl*)
done

lemma *iam-delete-impl*: *map-delete iam-*α *iam-invar iam-delete*
apply *unfold-locales*
unfolding *iam-*α*-def*[*abs-def*] *iam-delete-def*
apply (*rule ext*)
apply (*auto simp*: *Let-def array-get-set-iff*)
done

lemma *iam-rev-iterateoi-aux-foldli-conv* :
iam-rev-iterateoi-aux n a =
foldli (*List.map-filter* (λ*n. map-option* (λ*v.* (*n, v*)) (*array-get a n*)) (*rev* [*0..<n*]))
by (*induct n*) (*auto simp add*: *List.map-filter-def fun-eq-iff*)

lemma *iam-rev-iterateoi-foldli-conv* :
iam-rev-iterateoi a =
foldli (*List.map-filter*
(λ*n. map-option* (λ*v.* (*n, v*)) (*array-get a n*))
(*rev* [*0..<*(*array-length a*)]))
unfolding *iam-rev-iterateoi-def iam-rev-iterateoi-aux-foldli-conv* **by** *simp*

lemma *iam-rev-iterateoi-correct* :
fixes *m*::'*a option array*
defines *kvs* ≡ *List.map-filter*
(λ*n. map-option* (λ*v.* (*n, v*)) (*array-get m n*)) (*rev* [*0..<*(*array-length m*)])
shows *map-iterator-rev-linord* (*iam-rev-iterateoi m*) (*iam-*α *m*)
proof (*rule map-iterator-rev-linord-I* [*of kvs*])
show *iam-rev-iterateoi m* = *foldli kvs*
unfolding *iam-rev-iterateoi-foldli-conv kvs-def* **by** *simp*
next
define *al* **where** *al* = *array-length m*
show *dist-kvs*: *distinct* (*map fst kvs*) **and** *sorted* (*rev* (*map fst kvs*))
unfolding *kvs-def al-def*[*symmetric*]
by (*induct al*) (*simp-all add*: *set-map-filter image-iff sorted-append*
split: *option.split*)

from *dist-kvs*
have $\bigwedge$*i. map-of kvs i* = *iam-*α *m i*
unfolding *kvs-def*
apply (*case-tac array-get m i*)
apply (*simp-all*
add: *iam-*α*-def map-of-eq-None-iff set-map-filter image-iff*)
done
thus *iam-*α *m* = *map-of kvs* **by** *auto*
qed

lemma *iam-rev-iterateoi-impl*:
poly-map-rev-iterateoi iam-α *iam-invar iam-rev-iterateoi*
apply *unfold-locales*
apply (*simp add*: *iam-*α*-def*[*abs-def*] *dom-def*)

```
  apply (simp add: iam-rev-iterateoi-correct)
  done

lemma iam-iteratei-impl:
  poly-map-iteratei iam-α iam-invar iam-rev-iterateoi
proof –
  interpret aux: poly-map-rev-iterateoi iam-α iam-invar iam-rev-iterateoi
    by (rule iam-rev-iterateoi-impl)

  show ?thesis
    apply unfold-locales
    apply (rule map-rev-iterator-linord-is-it)
    by (rule aux.list-rev-it-correct)
qed

lemma iam-iterateoi-aux-foldli-conv :
  iam-iterateoi-aux n (array-length a) a c f σ =
   foldli (List.map-filter (λn. map-option (λv. (n, v)) (array-get a n))
     ([n..<array-length a])) c f σ
  by (induct n array-length a a c f σ rule: iam-iterateoi-aux.induct)
    (auto simp add: upt-conv-Cons iam-iterateoi-aux.simps split: option.split)

lemma iam-iterateoi-foldli-conv :
  iam-iterateoi a =
   foldli (List.map-filter
     (λn. map-option (λv. (n, v)) (array-get a n))
     ([0..<(array-length a)]))
  apply (intro ext)
  unfolding iam-iterateoi-def iam-iterateoi-aux-foldli-conv
  by simp


lemma map-filter-append[simp]: List.map-filter f (la@lb)
  = List.map-filter f la @ List.map-filter f lb
  by (induct la) (auto split: option.split)

lemma iam-iterateoi-correct:
fixes m::'a option array
defines kvs ≡ List.map-filter
  (λn. map-option (λv. (n, v)) (array-get m n)) ([0..<(array-length m)])
shows map-iterator-linord (iam-iterateoi m) (iam-α m)
proof (rule map-iterator-linord-I [of kvs])
  show iam-iterateoi m = foldli kvs
    unfolding iam-iterateoi-foldli-conv kvs-def by simp
next
  define al where al = array-length m
  show dist-kvs: distinct (map fst kvs) and sorted (map fst kvs)
    unfolding kvs-def al-def[symmetric]
    apply (induct al)
```

apply (*simp-all*
add: *set-map-filter image-iff sorted-append*
split: *option.split*)
done

from *dist-kvs*
have $\bigwedge i.$ *map-of kvs i* = *iam-α m i*
unfolding *kvs-def*
apply (*case-tac array-get m i*)
apply (*simp-all*
add: *iam-α-def map-of-eq-None-iff set-map-filter image-iff*)
done
thus *iam-α m* = *map-of kvs* **by** *auto*
qed

lemma *iam-iterateoi-impl*:
poly-map-iterateoi iam-α iam-invar iam-iterateoi
apply *unfold-locales*
apply (*simp add*: *iam-α-def*[*abs-def*] *dom-def*)
apply (*simp add*: *iam-iterateoi-correct*)
done

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*
|)

setup *Locale-Code.open-block*
interpretation *iam-basic*: *StdBasicOMap iam-basic-ops*
apply (*rule StdBasicOMap.intro*)
apply (*rule StdBasicMap.intro*)
apply (*simp-all add*: *icf-rec-unf*)
apply (*rule iam-empty-impl iam-lookup-impl iam-update-impl*
iam-update-dj-impl iam-delete-impl iam-iteratei-impl
iam-iterateoi-impl iam-rev-iterateoi-impl)+
done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *iam-ops* ≡ *iam-basic.dflt-oops*

setup *Locale-Code.open-block*
interpretation *iam*: *StdOMap iam-ops*
  unfolding *iam-ops-def*
  by (*rule iam-basic.dflt-oops-impl*)
interpretation *iam*: *StdMap-no-invar iam-ops*
  by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*
setup ‹*ICF-Tools.revert-abbrevs iam*›

lemma *pi-iam*[*proper-it*]:
  proper-it′ iam-iterateoi iam-iterateoi
  apply (*rule proper-it′I*)
  unfolding *iam-iterateoi-foldli-conv*
  by (*rule icf-proper-iteratorI*)

lemma *pi-iam-rev*[*proper-it*]:
  proper-it′ iam-rev-iterateoi iam-rev-iterateoi
  apply (*rule proper-it′I*)
  unfolding *iam-rev-iterateoi-foldli-conv*
  by (*rule icf-proper-iteratorI*)

interpretation *pi-iam*: *proper-it-loc iam-iterateoi iam-iterateoi*
  apply *unfold-locales* **by** (*rule pi-iam*)

interpretation *pi-iam-rev*: *proper-it-loc iam-rev-iterateoi iam-rev-iterateoi*
  apply *unfold-locales* **by** (*rule pi-iam-rev*)

Code generator test

definition *test-codegen* ≡ (
  iam.add ,
  iam.add-dj ,
  iam.ball ,
  iam.bex ,
  iam.delete ,
  iam.empty ,
  iam.isEmpty ,
  iam.isSng ,
  iam.iterate ,
  iam.iteratei ,
  iam.iterateo ,
  iam.iterateoi ,
  iam.list-it ,
  iam.lookup ,
  iam.max ,
  iam.min ,
  iam.restrict ,
  iam.rev-iterateo ,
  iam.rev-iterateoi ,
  iam.rev-list-it ,

iam.reverse-iterateo ,
iam.reverse-iterateoi ,
iam.sel ,
iam.size ,
iam.size-abort ,
iam.sng ,
iam.to-list ,
iam.to-map ,
iam.to-rev-list ,
iam.to-sorted-list ,
iam.update ,
iam.update-dj)

export-code *test-codegen* **checking** *SML*

end

3.3.12 Standard Implementations of Maps

theory *MapStdImpl*
imports
ListMapImpl
ListMapImpl-Invar
RBTMapImpl
HashMap
TrieMapImpl
ArrayHashMap
ArrayMapImpl
begin

This theory summarizes various standard implementation of maps, namely list-maps, RB-tree-maps, trie-maps, hashmaps, indexed array maps.

end

3.3.13 Set Implementation by List

theory *ListSetImpl*
imports *../spec/SetSpec* *../gen-algo/SetGA* *../../Lib/Dlist-add*
begin type-synonym
'a ls = *'a dlist*

Definitions

definition *ls-*α :: *'a ls* ⇒ *'a set* **where** *ls-*α *l* == *set* (*list-of-dlist l*)

definition *ls-basic-ops* :: (*'a*,*'a ls*) *set-basic-ops* **where**
[*icf-rec-def*]: *ls-basic-ops* ≡ (|
bset-op-α = *ls-*α,
bset-op-invar = λ-. *True*,

```
    bset-op-empty = λ-. Dlist.empty,
    bset-op-memb = (λx 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
    |)

setup Locale-Code.open-block
interpretation ls-basic: StdBasicSet ls-basic-ops
  apply unfold-locales
  unfolding ls-basic-ops-def ls-α-def[abs-def]
  apply (auto simp: dlist-member-empty Dlist.member-def      dlist-iteratei-correct
    dlist-remove′-correct set-dlist-remove1′
  )
  done
setup Locale-Code.close-block

definition [icf-rec-def]: ls-ops ≡ ls-basic.dflt-ops(|
  set-op-to-list := list-of-dlist
  |)

setup Locale-Code.open-block
interpretation ls: StdSetDefs ls-ops .
interpretation ls: StdSet ls-ops
proof −
  interpret aux: StdSet ls-basic.dflt-ops
    by (rule ls-basic.dflt-ops-impl)

  show StdSet ls-ops
    unfolding ls-ops-def
    apply (rule StdSet-intro)
    apply icf-locales
    apply (simp-all add: icf-rec-unf)
    apply (unfold-locales)
    apply (simp-all add: ls-α-def)
    done
qed

interpretation ls: StdSet-no-invar ls-ops
  by unfold-locales (simp add: icf-rec-unf)
setup Locale-Code.close-block

setup ‹ICF-Tools.revert-abbrevs ls›

lemma pi-ls[proper-it]:
  proper-it′ dlist-iteratei dlist-iteratei
  apply (rule proper-it′I)
  unfolding dlist-iteratei-def
```

by (*intro icf-proper-iteratorI*)

lemma *pi-ls'*[*proper-it*]:
proper-it' ls.iteratei ls.iteratei
apply (*rule proper-it'I*)
unfolding *ls.iteratei-def*
by (*intro icf-proper-iteratorI*)

interpretation *pi-ls*: *proper-it-loc dlist-iteratei dlist-iteratei*
apply *unfold-locales* **by** (*rule pi-ls*)

interpretation *pi-ls'*: *proper-it-loc ls.iteratei ls.iteratei*
apply *unfold-locales* **by** (*rule pi-ls'*)

definition *test-codegen* **where** *test-codegen* ≡ (
ls.empty,
ls.memb,
ls.ins,
ls.delete,
ls.list-it,
ls.sng,
ls.isEmpty,
ls.isSng,
ls.ball,
ls.bex,
ls.size,
ls.size-abort,
ls.union,
ls.union-dj,
ls.diff,
ls.filter,
ls.inter,
ls.subset,
ls.equal,
ls.disjoint,
ls.disjoint-witness,
ls.sel,
ls.to-list,
ls.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.14 Set Implementation by List with explicit invariants

theory *ListSetImpl-Invar*
imports

../spec/SetSpec
../gen-algo/SetGA
../../Lib/Dlist-add
begin type-synonym
'a lsi = 'a list

Definitions

definition *lsi-ins* :: *'a* ⇒ *'a lsi* ⇒ *'a lsi* **where** *lsi-ins x l == if List.member l x then l else x#l*

definition *lsi-basic-ops* :: (*'a*,*'a lsi*) *set-basic-ops* **where**
[*icf-rec-def*]: *lsi-basic-ops* ≡ ⦇
bset-op-α = set,
bset-op-invar = distinct,
bset-op-empty = λ-. [],
bset-op-memb = (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
⦈

setup *Locale-Code.open-block*
interpretation *lsi-basic*: *StdBasicSet lsi-basic-ops*
apply *unfold-locales*
unfolding *lsi-basic-ops-def lsi-ins-def*[*abs-def*]
apply (*auto simp*: *set-dlist-remove1'*
distinct-remove1')
done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *lsi-ops* ≡ *lsi-basic.dflt-ops* ⦇
set-op-union-dj := (@),
set-op-to-list := *id*
⦈

setup *Locale-Code.open-block*
interpretation *lsi*: *StdSet lsi-ops*
proof −
interpret *aux*: *StdSet lsi-basic.dflt-ops* **by** (*rule lsi-basic.dflt-ops-impl*)

show *StdSet lsi-ops*
unfolding *lsi-ops-def*
apply (*rule StdSet-intro*)
apply *icf-locales*
apply (*simp-all add*: *icf-rec-unf*)
apply (*unfold-locales*, *auto*)
done

qed
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs lsi*›

lemma *pi-lsi*[*proper-it*]:
  proper-it′ foldli foldli
  by (*intro icf-proper-iteratorI proper-it′I*)

interpretation *pi-lsi*: *proper-it-loc foldli foldli*
  apply *unfold-locales* **by** (*rule pi-lsi*)

definition *test-codegen* **where** *test-codegen* ≡ (
  lsi.empty,
  lsi.memb,
  lsi.ins,
  lsi.delete,
  lsi.list-it,
  lsi.sng,
  lsi.isEmpty,
  lsi.isSng,
  lsi.ball,
  lsi.bex,
  lsi.size,
  lsi.size-abort,
  lsi.union,
  lsi.union-dj,
  lsi.diff,
  lsi.filter,
  lsi.inter,
  lsi.subset,
  lsi.equal,
  lsi.disjoint,
  lsi.disjoint-witness,
  lsi.sel,
  lsi.to-list,
  lsi.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.15 Set Implementation by non-distinct Lists

theory *ListSetImpl-NotDist*
imports
  ../spec/SetSpec
  ../gen-algo/SetGA

begin type-synonym
$'a$ *lsnd* = $'a$ *list*

Definitions

definition *lsnd-*α :: $'a$ *lsnd* $\Rightarrow$ $'a$ *set* **where** *lsnd-*α == *set*
abbreviation (*input*) *lsnd-invar*
:: $'a$ *lsnd* $\Rightarrow$ *bool* **where** *lsnd-invar* == (λ-. *True*)
definition *lsnd-empty* :: *unit* $\Rightarrow$ $'a$ *lsnd* **where** *lsnd-empty* == (λ-::*unit*. [])
definition *lsnd-memb* :: $'a \Rightarrow 'a$ *lsnd* $\Rightarrow$ *bool* **where** *lsnd-memb x l* == *List.member l x*
definition *lsnd-ins* :: $'a \Rightarrow 'a$ *lsnd* $\Rightarrow$ $'a$ *lsnd* **where** *lsnd-ins x l* == *x#l*
definition *lsnd-ins-dj* :: $'a \Rightarrow 'a$ *lsnd* $\Rightarrow$ $'a$ *lsnd* **where** *lsnd-ins-dj x l* == *x#l*

definition *lsnd-delete* :: $'a \Rightarrow 'a$ *lsnd* $\Rightarrow$ $'a$ *lsnd* **where** *lsnd-delete x l* == *remove-rev x l*

definition *lsnd-iteratei* :: $'a$ *lsnd* $\Rightarrow$ ($'a$,$'\sigma$) *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*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-memb-impl*: *set-memb lsnd-α lsnd-invar lsnd-memb*
by (*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *lsnd-ins-impl*: *set-ins lsnd-α lsnd-invar lsnd-ins*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-ins-dj-impl*: *set-ins-dj lsnd-α lsnd-invar lsnd-ins-dj*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-delete-impl*: *set-delete lsnd-α lsnd-invar lsnd-delete*
by (*unfold-locales*) (*auto simp add*: *lsnd-delete-def lsnd-α-def remove-rev-alt-def*)

lemma *lsnd-α-finite*[*simp*, *intro*!]: *finite* (*lsnd-α l*)
  by (*auto simp add*: *lsnd-defs*)

lemma *lsnd-is-finite-set*: *finite-set lsnd-α lsnd-invar*
by (*unfold-locales*) (*auto simp add*: *lsnd-defs*)

lemma *lsnd-iteratei-impl*: *poly-set-iteratei lsnd-α lsnd-invar lsnd-iteratei*
proof
  fix *l* :: *'a list*
  show *finite* (*lsnd-α l*)
    unfolding *lsnd-α-def* **by** *simp*

  show *set-iterator* (*lsnd-iteratei l*) (*lsnd-α l*)
    apply (*rule set-iterator-I* [*of remdups l*])
    apply (*simp-all add*: *lsnd-α-def lsnd-iteratei-def*)
  done
qed

lemma *lsnd-isEmpty-impl*: *set-isEmpty lsnd-α lsnd-invar lsnd-isEmpty*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *lsnd-union-impl*: *set-union lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *lsnd-union-dj-impl*: *set-union-dj lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-α lsnd-invar lsnd-union-dj*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *lsnd-to-list-impl*: *set-to-list lsnd-α lsnd-invar lsnd-to-list*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

lemma *list-to-lsnd-impl*: *list-to-set lsnd-α lsnd-invar list-to-lsnd*
by(*unfold-locales*)(*auto simp add*: *lsnd-defs*)

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
⦈

setup *Locale-Code.open-block*
interpretation *lsnd-basic: StdBasicSet lsnd-basic-ops*
apply (*rule StdBasicSet.intro*)
apply (*simp-all add: icf-rec-unf*)
apply (*rule lsnd-empty-impl lsnd-memb-impl lsnd-ins-impl lsnd-ins-dj-impl lsnd-delete-impl lsnd-iteratei-impl*)+
done
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *lsnd-ops* ≡ *lsnd-basic.dflt-ops* ⦇
set-op-isEmpty := lsnd-isEmpty,
set-op-union := lsnd-union,
set-op-union-dj := lsnd-union-dj,
set-op-to-list := lsnd-to-list,
set-op-from-list := list-to-lsnd
⦈

setup *Locale-Code.open-block*
interpretation *lsnd: StdSetDefs lsnd-ops* **.**
interpretation *lsnd: StdSet lsnd-ops*
proof −
interpret *aux: StdSet lsnd-basic.dflt-ops*
by (*rule lsnd-basic.dflt-ops-impl*)

show *StdSet lsnd-ops*
unfolding *lsnd-ops-def*
apply (*rule StdSet-intro*)
apply *icf-locales*
apply (*simp-all add: icf-rec-unf lsnd-isEmpty-impl lsnd-union-impl lsnd-union-dj-impl lsnd-to-list-impl list-to-lsnd-impl*
)
done
qed
interpretation *lsnd: StdSet-no-invar lsnd-ops*
by *unfold-locales* (*simp add: icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs lsnd*›

lemma *pi-lsnd*[*proper-it*]:
  proper-it′ lsnd-iteratei lsnd-iteratei
  apply (*rule proper-it′I*)
  unfolding *lsnd-iteratei-def*
  by (*intro icf-proper-iteratorI*)

interpretation *pi-lsnd*: *proper-it-loc lsnd-iteratei lsnd-iteratei*
  apply *unfold-locales* **by** (*rule pi-lsnd*)

definition *test-codegen* **where** *test-codegen* ≡ (
  lsnd.empty,
  lsnd.memb,
  lsnd.ins,
  lsnd.delete,
  lsnd.list-it,
  lsnd.sng,
  lsnd.isEmpty,
  lsnd.isSng,
  lsnd.ball,
  lsnd.bex,
  lsnd.size,
  lsnd.size-abort,
  lsnd.union,
  lsnd.union-dj,
  lsnd.diff,
  lsnd.filter,
  lsnd.inter,
  lsnd.subset,
  lsnd.equal,
  lsnd.disjoint,
  lsnd.disjoint-witness,
  lsnd.sel,
  lsnd.to-list,
  lsnd.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.16 Set Implementation by sorted Lists

theory *ListSetImpl-Sorted*
imports
  ../spec/SetSpec
  ../gen-algo/SetGA
  ../../Lib/Sorted-List-Operations

begin type-synonym
'a lss = 'a list

Definitions

definition *lss-α* :: *'a lss* ⇒ *'a set* **where** *lss-α* == *set*
definition *lss-invar* :: *'a*::{*linorder*} *lss* ⇒ *bool* **where** *lss-invar l* == *distinct l* ∧ *sorted l*
definition *lss-empty* :: *unit* ⇒ *'a lss* **where** *lss-empty* == (λ-::*unit*. [])
definition *lss-memb* :: *'a*::{*linorder*} ⇒ *'a lss* ⇒ *bool* **where** *lss-memb x l* == *Sorted-List-Operations.memb-sorted l x*
definition *lss-ins* :: *'a*::{*linorder*} ⇒ *'a lss* ⇒ *'a lss* **where** *lss-ins x l* == *insertion-sort x l*
definition *lss-ins-dj* :: *'a*::{*linorder*} ⇒ *'a lss* ⇒ *'a lss* **where** *lss-ins-dj* == *lss-ins*

definition *lss-delete* :: *'a*::{*linorder*} ⇒ *'a lss* ⇒ *'a lss* **where** *lss-delete x l* == *delete-sorted x l*

definition *lss-iterateoi* :: *'a lss* ⇒ (*'a*,*'σ*) *set-iterator*
where *lss-iterateoi l* = *foldli l*

definition *lss-reverse-iterateoi* :: *'a lss* ⇒ (*'a*,*'σ*) *set-iterator*
where *lss-reverse-iterateoi l* = *foldli* (*rev l*)

definition *lss-iteratei* :: *'a lss* ⇒ (*'a*,*'σ*) *set-iterator*
where *lss-iteratei l* = *foldli l*

definition *lss-isEmpty* :: *'a lss* ⇒ *bool* **where** *lss-isEmpty s* == *s*=[]

definition *lss-union* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
  where *lss-union s1 s2* == *Misc.merge s1 s2*
definition *lss-union-list* :: *'a*::{*linorder*} *lss list* ⇒ *'a lss*
  where *lss-union-list l* == *merge-list* [] *l*
definition *lss-inter* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
  where *lss-inter* == *inter-sorted*
definition *lss-union-dj* :: *'a*::{*linorder*} *lss* ⇒ *'a lss* ⇒ *'a lss*
  where *lss-union-dj* == *lss-union* — Union of disjoint sets

definition *lss-image-filter*
  where *lss-image-filter f l* =
    mergesort-remdups (*List.map-filter f l*)

definition *lss-filter* **where** [*code-unfold*]: *lss-filter* = *List.filter*

definition *lss-inj-image-filter*
  where *lss-inj-image-filter* == *lss-image-filter*

definition *lss-image* == *iflt-image lss-image-filter*
definition *lss-inj-image* == *iflt-inj-image lss-inj-image-filter*

definition *lss-to-list* :: *'a lss* ⇒ *'a list* **where** *lss-to-list* == *id*
definition *list-to-lss* :: *'a*::{*linorder*} *list* ⇒ *'a lss* **where** *list-to-lss* == *merge-sort-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*
by (*unfold-locales*) (*auto simp add*: *lss-defs*)

lemma *lss-memb-impl*: *set-memb lss-α lss-invar lss-memb*
by (*unfold-locales*) (*auto simp add*: *lss-defs memb-sorted-correct*)

lemma *lss-ins-impl*: *set-ins lss-α lss-invar lss-ins*
by (*unfold-locales*) (*auto simp add*: *lss-defs insertion-sort-correct*)

lemma *lss-ins-dj-impl*: *set-ins-dj lss-α lss-invar lss-ins-dj*
by (*unfold-locales*) (*auto simp add*: *lss-defs insertion-sort-correct*)

lemma *lss-delete-impl*: *set-delete lss-α lss-invar lss-delete*
by(*unfold-locales*)(*auto simp add*: *lss-delete-def lss-α-def lss-invar-def delete-sorted-correct*)

lemma *lss-α-finite*[*simp*, *intro*!]: *finite* (*lss-α l*)
by (*auto simp add*: *lss-defs*)

lemma *lss-is-finite-set*: *finite-set lss-α lss-invar*
by (*unfold-locales*) (*auto simp add*: *lss-defs*)

lemma *lss-iterateoi-impl*: *poly-set-iterateoi lss-α lss-invar lss-iterateoi*
proof
  fix *l* :: *'a*::{*linorder*} *list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator-linord* (*lss-iterateoi l*) (*lss-α l*)
    apply (*rule-tac set-iterator-linord-I* [*of l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-iterateoi-def*)
  done
qed

lemma *lss-reverse-iterateoi-impl*: *poly-set-rev-iterateoi lss-α lss-invar lss-reverse-iterateoi*
proof
  fix *l* :: *'a list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator-rev-linord* (*lss-reverse-iterateoi l*) (*lss-α l*)
    apply (*rule-tac set-iterator-rev-linord-I* [*of rev l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-reverse-iterateoi-def*)
  done
qed

lemma *lss-iteratei-impl*: *poly-set-iteratei lss-α lss-invar lss-iteratei*
proof
  fix *l* :: *'a list*
  assume *invar-l*: *lss-invar l*
  show *finite* (*lss-α l*)
    unfolding *lss-α-def* **by** *simp*

  from *invar-l*
  show *set-iterator* (*lss-iteratei l*) (*lss-α l*)
    apply (*rule-tac set-iterator-I* [*of l*])
    apply (*simp-all add*: *lss-α-def lss-invar-def lss-iteratei-def*)
  done
qed

lemma *lss-isEmpty-impl*: *set-isEmpty lss-α lss-invar lss-isEmpty*
by(*unfold-locales*)(*auto simp add*: *lss-defs*)

lemma *lss-inter-impl*: *set-inter lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-inter*
by (*unfold-locales*) (*auto simp add*: *lss-defs inter-sorted-correct*)

lemma *lss-union-impl*: *set-union lss-α lss-invar lss-α lss-invar lss-α lss-invar lss-union*

by (*unfold-locales*) (*auto simp add*: *lss-defs merge-correct*)

lemma *lss-union-list-impl*: *set-union-list lss-*α *lss-invar lss-*α *lss-invar lss-union-list*
proof
  fix *l* :: *'a*::{*linorder*} *lss list*
  assume $\forall$ *s1*$\in$*set l*. *lss-invar s1*

  with *merge-list-correct* [*of l* []]
  show *lss-*α (*lss-union-list l*) = $\bigcup${*lss-*α *s1* |*s1*. *s1* $\in$ *set l*}
      lss-invar (*lss-union-list l*)
    by (*auto simp add*: *lss-defs*)
qed

lemma *lss-union-dj-impl*: *set-union-dj lss-*α *lss-invar lss-*α *lss-invar lss-*α *lss-invar lss-union-dj*
by (*unfold-locales*) (*auto simp add*: *lss-defs merge-correct*)

lemma *lss-image-filter-impl* : *set-image-filter lss-*α *lss-invar lss-*α *lss-invar lss-image-filter*
apply (*unfold-locales*)
apply (*simp-all add*:
  lss-invar-def lss-image-filter-def lss-α*-def mergesort-remdups-correct*
  set-map-filter Bex-def)
done

lemma *lss-inj-image-filter-impl* : *set-inj-image-filter lss-*α *lss-invar lss-*α *lss-invar lss-inj-image-filter*
apply (*unfold-locales*)
apply (*simp-all add*: *lss-invar-def lss-inj-image-filter-def lss-image-filter-def*
          mergesort-remdups-correct lss-α*-def*
          set-map-filter Bex-def)
done

lemma *lss-filter-impl* : *set-filter lss-*α *lss-invar lss-*α *lss-invar lss-filter*
apply (*unfold-locales*)
apply (*simp-all add*: *lss-invar-def lss-filter-def sorted-filter lss-*α*-def*
          set-map-filter Bex-def sorted-filter')
done

lemmas *lss-image-impl* = *iflt-image-correct*[*OF lss-image-filter-impl*, *folded lss-image-def*]
lemmas *lss-inj-image-impl* = *iflt-inj-image-correct*[*OF lss-inj-image-filter-impl*, *folded lss-inj-image-def*]

lemma *lss-to-list-impl*: *set-to-list lss-*α *lss-invar lss-to-list*
by(*unfold-locales*)(*auto simp add*: *lss-defs*)

lemma *list-to-lss-impl*: *list-to-set lss-*α *lss-invar list-to-lss*
  by (*unfold-locales*) (*auto simp add*: *lss-defs mergesort-remdups-correct*)

```
definition lss-basic-ops :: ('x::linorder,'x lss) oset-basic-ops
  where [icf-rec-def]: lss-basic-ops ≡ (|
    bset-op-α = lss-α,
    bset-op-invar = lss-invar,
    bset-op-empty = lss-empty,
    bset-op-memb = lss-memb,
    bset-op-ins = lss-ins,
    bset-op-ins-dj = lss-ins-dj,
    bset-op-delete = lss-delete,
    bset-op-list-it = lss-iteratei,
    bset-op-ordered-list-it = lss-iterateoi,
    bset-op-rev-list-it = lss-reverse-iterateoi
  |)

setup Locale-Code.open-block
interpretation lss-basic: StdBasicOSet lss-basic-ops
  apply (rule StdBasicOSet.intro)
  apply (rule StdBasicSet.intro)
  apply (simp-all add: icf-rec-unf)
  apply (rule lss-empty-impl lss-memb-impl lss-ins-impl lss-ins-dj-impl
    lss-delete-impl lss-iteratei-impl lss-iterateoi-impl
    lss-reverse-iterateoi-impl)+
  done
setup Locale-Code.close-block

definition [icf-rec-def]: lss-ops ≡ lss-basic.dflt-oops (|
  set-op-isEmpty := lss-isEmpty,
  set-op-union := lss-union,
  set-op-union-dj := lss-union-dj,
  set-op-filter := lss-filter,
  set-op-to-list := lss-to-list,
  set-op-from-list := list-to-lss
  |)

setup Locale-Code.open-block
interpretation lss: StdOSetDefs lss-ops .
interpretation lss: StdOSet lss-ops
proof −
  interpret aux: StdOSet lss-basic.dflt-oops
    by (rule lss-basic.dflt-oops-impl)

  show StdOSet lss-ops
    unfolding lss-ops-def
    apply (rule StdOSet-intro)
    apply icf-locales
    apply (simp-all add: icf-rec-unf
      lss-isEmpty-impl lss-union-impl lss-union-dj-impl lss-to-list-impl
      lss-filter-impl
      list-to-lss-impl
```

)
done
qed
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs lss*›

lemma *pi-lss*[*proper-it*]:
proper-it′ lss-iteratei lss-iteratei
apply (*rule proper-it′I*)
unfolding *lss-iteratei-def*
by (*intro icf-proper-iteratorI*)

interpretation *pi-lss*: *proper-it-loc lss-iteratei lss-iteratei*
apply *unfold-locales* **by** (*rule pi-lss*)

definition *test-codegen* **where** *test-codegen* ≡ (
lss.empty,
lss.memb,
lss.ins,
lss.delete,
lss.list-it,
lss.sng,
lss.isEmpty,
lss.isSng,
lss.ball,
lss.bex,
lss.size,
lss.size-abort,
lss.union,
lss.union-dj,
lss.diff,
lss.filter,
lss.inter,
lss.subset,
lss.equal,
lss.disjoint,
lss.disjoint-witness,
lss.sel,
lss.to-list,
lss.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.17 Set Implementation by Red-Black-Tree

theory *RBTSetImpl*
imports
  ../spec/SetSpec
  RBTMapImpl
  ../gen-algo/SetByMap
  ../gen-algo/SetGA
begin

Definitions

type-synonym
  'a rs = (*'a*::*linorder*,*unit*) *rm*

setup *Locale-Code.open-block*
interpretation *rs-sbm*: *OSetByOMap rm-basic-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition *rs-ops* :: (*'x*::*linorder*,*'x rs*) *oset-ops*
  where [*icf-rec-def*]: *rs-ops* ≡ *rs-sbm.obasic.dflt-oops*

setup *Locale-Code.open-block*
interpretation *rs*: *StdOSetDefs rs-ops* **.**
interpretation *rs*: *StdOSet rs-ops*
  unfolding *rs-ops-def*
  by (*rule rs-sbm.obasic.dflt-oops-impl*)

interpretation *rs*: *StdSet-no-invar rs-ops*
  by *unfold-locales* (*simp add*: *icf-rec-unf SetByMapDefs.invar-def*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs rs*›

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
  unfolding *rs.iteratei-def*[*abs-def*] *rs.iterateoi-def*[*abs-def*]
    rs.rev-iterateoi-def[*abs-def*]
  by (*rule proper-it'I icf-proper-iteratorI*)+

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*

by *unfold-locales* (*rule pi-rs*)+

definition *test-codegen* **where** *test-codegen* ≡ (
rs.empty,
rs.memb,
rs.ins,
rs.delete,
rs.list-it,
rs.sng,
rs.isEmpty,
rs.isSng,
rs.ball,
rs.bex,
rs.size,
rs.size-abort,
rs.union,
rs.union-dj,
rs.diff,
rs.filter,
rs.inter,
rs.subset,
rs.equal,
rs.disjoint,
rs.disjoint-witness,
rs.sel,
rs.to-list,
rs.from-list,

rs.ordered-list-it,
rs.rev-list-it,
rs.min,
rs.max,
rs.to-sorted-list,
rs.to-rev-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.18 Hash Set

theory *HashSet*
imports
../spec/SetSpec
HashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
'a hs = (*'a*::*hashable*,*unit*) *hm*

setup *Locale-Code.open-block*
interpretation *hs-sbm*: *SetByMap hm-basic-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition *hs-ops* :: (*'a*::*hashable*,*'a hs*) *set-ops*
where [*icf-rec-def*]:
hs-ops ≡ *hs-sbm.basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *hs*: *StdSet hs-ops*
unfolding *hs-ops-def* **by** (*rule hs-sbm.basic.dflt-ops-impl*)
interpretation *hs*: *StdSet-no-invar hs-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf SetByMapDefs.invar-def*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs hs*›

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*
unfolding *hs.iteratei-def*[*abs-def*]
by (*rule proper-it'I icf-proper-iteratorI*)+

interpretation *pi-hs*: *proper-it-loc hs.iteratei hs.iteratei*
by *unfold-locales* (*rule pi-hs*)

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
$'a\ ts = ('a,\ unit)\ trie$

setup *Locale-Code.open-block*
interpretation *ts-sbm*: *SetByMap tm-basic-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition *ts-ops* :: $('a\ list,'a\ ts)$ *set-ops*
where [*icf-rec-def*]:
ts-ops ≡ *ts-sbm.basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *ts*: *StdSet ts-ops*
unfolding *ts-ops-def* **by** (*rule ts-sbm.basic.dflt-ops-impl*)
interpretation *ts*: *StdSet-no-invar ts-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf SetByMapDefs.invar-def*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs ts*›

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*
unfolding *ts.iteratei-def*[*abs-def*]
by (*rule proper-it'I icf-proper-iteratorI*)+

interpretation *pi-ts*: *proper-it-loc ts.iteratei ts.iteratei*
by *unfold-locales* (*rule pi-ts*)

definition *test-codegen* **where** *test-codegen* ≡ (
ts.empty,
ts.memb,
ts.ins,
ts.delete,
ts.list-it,
ts.sng,
ts.isEmpty,
ts.isSng,
ts.ball,
ts.bex,
ts.size,
ts.size-abort,
ts.union,
ts.union-dj,
ts.diff,
ts.filter,
ts.inter,
ts.subset,
ts.equal,
ts.disjoint,
ts.disjoint-witness,
ts.sel,
ts.to-list,
ts.from-list
)

export-code *test-codegen* **checking** *SML*

end

theory *ArrayHashSet*
imports
ArrayHashMap
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym
$'a$ *ahs* = ($'a$::*hashable*,*unit*) *ahm*

setup *Locale-Code.open-block*
interpretation *ahs-sbm*: *SetByMap ahm-basic-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition *ahs-ops* :: ('*a*::*hashable*,'*a ahs*) *set-ops*
  where [*icf-rec-def*]:
  ahs-ops ≡ *ahs-sbm.basic.dflt-ops*

setup *Locale-Code.open-block*
interpretation *ahs*: *StdSet ahs-ops*
  unfolding *ahs-ops-def* **by** (*rule ahs-sbm.basic.dflt-ops-impl*)
interpretation *ahs*: *StdSet-no-invar ahs-ops*
  apply *unfold-locales*
  by (*simp add*: *icf-rec-unf SetByMapDefs.invar-def*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs ahs*›

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*
  unfolding *ahs.iteratei-def*[*abs-def*]
  by (*rule proper-it'I icf-proper-iteratorI*)+

interpretation *pi-ahs*: *proper-it-loc ahs.iteratei ahs.iteratei*
  by *unfold-locales* (*rule pi-ahs*)

definition *test-codegen* **where** *test-codegen* ≡ (
  ahs.empty,
  ahs.memb,
  ahs.ins,
  ahs.delete,
  ahs.list-it,
  ahs.sng,
  ahs.isEmpty,
  ahs.isSng,
  ahs.ball,
  ahs.bex,
  ahs.size,
  ahs.size-abort,
  ahs.union,
  ahs.union-dj,
  ahs.diff,
  ahs.filter,
  ahs.inter,
  ahs.subset,
  ahs.equal,
  ahs.disjoint,
  ahs.disjoint-witness,
  ahs.sel,
  ahs.to-list,

ahs.from-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.20 Set Implementation by Arrays

theory *ArraySetImpl*
imports
../spec/SetSpec
ArrayMapImpl
../gen-algo/SetByMap
../gen-algo/SetGA
begin

Definitions

type-synonym *ias* = (*unit*) *iam*

setup *Locale-Code.open-block*
interpretation *ias-sbm*: *OSetByOMap iam-basic-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*
definition *ias-ops* :: (*nat*,*ias*) *oset-ops*
where [*icf-rec-def*]:
ias-ops ≡ *ias-sbm.obasic.dflt-oops*

setup *Locale-Code.open-block*
interpretation *ias*: *StdOSet ias-ops*
unfolding *ias-ops-def* **by** (*rule ias-sbm.obasic.dflt-oops-impl*)
interpretation *ias*: *StdSet-no-invar ias-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf SetByMapDefs.invar-def*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs ias*›

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
unfolding *ias.iteratei-def*[*abs-def*] *ias.iterateoi-def*[*abs-def*]
ias.rev-iterateoi-def[*abs-def*]
apply (*rule proper-it′I icf-proper-iteratorI*)+
done

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*
  apply *unfold-locales* **by** (*rule pi-ias*)+

definition *test-codegen* **where** *test-codegen* ≡ (
  ias.empty,
  ias.memb,
  ias.ins,
  ias.delete,
  ias.list-it,
  ias.sng,
  ias.isEmpty,
  ias.isSng,
  ias.ball,
  ias.bex,
  ias.size,
  ias.size-abort,
  ias.union,
  ias.union-dj,
  ias.diff,
  ias.filter,
  ias.inter,
  ias.subset,
  ias.equal,
  ias.disjoint,
  ias.disjoint-witness,
  ias.sel,
  ias.to-list,
  ias.from-list,

  ias.ordered-list-it,
  ias.rev-list-it,
  ias.min,
  ias.max,
  ias.to-sorted-list,
  ias.to-rev-list
)

export-code *test-codegen* **checking** *SML*

end

3.3.21 Standard Set Implementations

theory *SetStdImpl*
imports
  ListSetImpl

ListSetImpl-Invar
ListSetImpl-NotDist
ListSetImpl-Sorted
RBTSetImpl HashSet
TrieSetImpl
ArrayHashSet
ArraySetImpl
begin

This theory summarizes standard set implementations, namely list-sets RB-tree-sets, trie-sets and hashsets.

end

3.3.22 Fifo Queue by Pair of Lists

theory *Fifo*
imports
../gen-algo/ListGA
../tools/Record-Intf
../tools/Locale-Code
begin lemma *rev-tl-rev*: $rev\ (tl\ (rev\ l)) = butlast\ l$
by (*induct l*) *auto*

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 $'a\ fifo = 'a\ list \times 'a\ list$

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 $fifo\text{-}\alpha :: 'a\ fifo \Rightarrow 'a\ list$
where $fifo\text{-}\alpha\ F == snd\ F\ @\ rev\ (fst\ F)$

This fifo implementation has no invariants, any pair of lists is a valid fifo

definition [*simp*, *intro*!]: $fifo\text{-}invar\ x = True$

— The empty fifo
definition $fifo\text{-}empty :: unit \Rightarrow 'a\ fifo$

where *fifo-empty* == λ-::*unit*. ([],[])

— True, iff the fifo is empty
definition *fifo-isEmpty* :: *'a fifo* ⇒ *bool* **where** *fifo-isEmpty F* == *F*=([],[])

definition *fifo-size* :: *'a fifo* ⇒ *nat* **where**
fifo-size F ≡ *length* (*fst F*) + *length* (*snd F*)

— Add an element to the fifo
definition *fifo-appendr* :: *'a* ⇒ *'a fifo* ⇒ *'a fifo*
where *fifo-appendr a F* == (*a*#*fst F*, *snd F*)

definition *fifo-appendl* :: *'a* ⇒ *'a fifo* ⇒ *'a fifo*
where *fifo-appendl x F* == *case F of* (*e*,*d*) ⇒ (*e*,*x*#*d*)

— Get an element from the fifo
definition *fifo-remover* :: *'a fifo* ⇒ (*'a fifo* × *'a*) **where**
fifo-remover F ==
  case fst F of
    (*a*#*l*) ⇒ ((*l*,*snd F*),*a*) |
    [] ⇒ *let rp*=*rev* (*snd F*) *in*
      ((*tl rp*,[]),*hd rp*)

definition *fifo-removel* :: *'a fifo* ⇒ (*'a* × *'a fifo*) **where**
fifo-removel F ==
  case snd F of
    (*a*#*l*) ⇒ (*a*, (*fst F*, *l*)) |
    [] ⇒ *let rp*=*rev* (*fst F*) *in*
      (*hd rp*, ([], *tl rp*))

definition *fifo-leftmost* :: *'a fifo* ⇒ *'a* **where**
fifo-leftmost F ≡ *case F of* (-,*x*#-) ⇒ *x* | (*l*,[]) ⇒ *last l*

definition *fifo-rightmost* :: *'a fifo* ⇒ *'a* **where**
fifo-rightmost F ≡ *case F of* (*x*#-,-) ⇒ *x* | ([],*l*) ⇒ *last l*

definition *fifo-iteratei F* ≡ *foldli* (*fifo-α F*)
definition *fifo-rev-iteratei F* ≡ *foldri* (*fifo-α F*)

definition *fifo-get F i* ≡
let
  l2 = *length* (*snd F*)
in
  if i < *l2 then*
    snd F!*i*
  else
    (*fst F*)!(*length* (*fst F*) − *Suc* (*i* − *l2*))

definition *fifo-set F i a* ≡ *case F of* (*f1*,*f2*) ⇒
  let
    l2 = *length f2*
  in
    if i < *l2 then*
      (*f1*,*f2*[*i*:=*a*])
    else
      (*f1*[*length* (*fst F*) − *Suc* (*i* − *l2*) := *a*],*f2*)

Correctness

lemma *fifo-empty-impl*: *list-empty fifo-α fifo-invar fifo-empty*
  apply (*unfold-locales*)
  by (*auto simp add*: *fifo-α-def fifo-empty-def*)

lemma *fifo-isEmpty-impl*: *list-isEmpty fifo-α fifo-invar fifo-isEmpty*
  apply (*unfold-locales*)
  by (*case-tac s*) (*auto simp add*: *fifo-isEmpty-def fifo-α-def*)

lemma *fifo-size-impl*: *list-size fifo-α fifo-invar fifo-size*
  apply *unfold-locales*
  by (*auto simp add*: *fifo-size-def fifo-α-def*)

lemma *fifo-appendr-impl*: *list-appendr fifo-α fifo-invar fifo-appendr*
  apply (*unfold-locales*)
  by (*auto simp add*: *fifo-appendr-def fifo-α-def*)

lemma *fifo-appendl-impl*: *list-appendl fifo-α fifo-invar fifo-appendl*
  apply (*unfold-locales*)
  by (*auto simp add*: *fifo-appendl-def fifo-α-def*)

lemma *fifo-removel-impl*: *list-removel fifo-α fifo-invar fifo-removel*
  apply (*unfold-locales*)
  apply (*case-tac s*)
  apply (*case-tac b*)
  apply (*auto simp add*: *fifo-removel-def fifo-α-def Let-def*) [*2*]
  apply (*case-tac s*)
  apply (*case-tac b*)
  apply (*auto simp add*: *fifo-removel-def fifo-α-def Let-def*)
  done

lemma *fifo-remover-impl*: *list-remover fifo-α fifo-invar fifo-remover*
  apply (*unfold-locales*)
  unfolding *fifo-remover-def fifo-α-def Let-def*
  by (*auto split*: *list.split simp*: *hd-rev rev-tl-rev butlast-append*)

lemma *fifo-leftmost-impl*: *list-leftmost fifo-α fifo-invar fifo-leftmost*
  apply *unfold-locales*

by (*auto simp*: *fifo-leftmost-def fifo-α-def hd-rev split*: *list.split*)

lemma *fifo-rightmost-impl*: *list-rightmost fifo-α fifo-invar fifo-rightmost*
apply *unfold-locales*
by (*auto simp*: *fifo-rightmost-def fifo-α-def hd-rev split*: *list.split*)

lemma *fifo-get-impl*: *list-get fifo-α fifo-invar fifo-get*
apply *unfold-locales*
apply (*auto simp*: *fifo-α-def fifo-get-def Let-def nth-append rev-nth*)
done

lemma *fifo-set-impl*: *list-set fifo-α fifo-invar fifo-set*
apply *unfold-locales*
apply (*auto simp*: *fifo-α-def fifo-set-def Let-def list-update-append rev-update*)
done

definition [*icf-rec-def*]: *fifo-ops* $\equiv$ (|
list-op-α = *fifo-α*,
list-op-invar = *fifo-invar*,
list-op-empty = *fifo-empty*,
list-op-isEmpty = *fifo-isEmpty*,
list-op-size = *fifo-size*,
list-op-appendl = *fifo-appendl*,
list-op-removel = *fifo-removel*,
list-op-leftmost = *fifo-leftmost*,
list-op-appendr = *fifo-appendr*,
list-op-remover = *fifo-remover*,
list-op-rightmost = *fifo-rightmost*,
list-op-get = *fifo-get*,
list-op-set = *fifo-set*
|)

setup *Locale-Code.open-block*
interpretation *fifo*: *StdList fifo-ops*
apply (*rule StdList.intro*)
apply (*simp-all add*: *icf-rec-unf*)
apply (*rule*
fifo-empty-impl
fifo-isEmpty-impl
fifo-size-impl
fifo-appendl-impl
fifo-removel-impl
fifo-leftmost-impl
fifo-appendr-impl
fifo-remover-impl
fifo-rightmost-impl
fifo-get-impl
fifo-set-impl)+

done
interpretation *fifo*: *StdList-no-invar fifo-ops*
by *unfold-locales* (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*
setup ‹*ICF-Tools.revert-abbrevs fifo*›

definition *test-codegen* **where** *test-codegen* ≡
(
fifo.empty,
fifo.isEmpty,
fifo.size,
fifo.appendl,
fifo.removel,
fifo.leftmost,
fifo.appendr,
fifo.remover,
fifo.rightmost,
fifo.get,
fifo.set,
fifo.iteratei,
fifo.rev-iteratei
)

export-code *test-codegen* **checking** *SML*

end

3.3.23 Implementation of Priority Queues by Binomial Heap

theory *BinoPrioImpl*
imports
Binomial−Heaps.BinomialHeap
../spec/PrioSpec
../tools/Record-Intf
../tools/Locale-Code
begin

type-synonym ($'a$,$'b$) *bino* = ($'a$,$'b$) *BinomialHeap*

Definitions

definition *bino-α* **where** *bino-α q* ≡ *BinomialHeap.to-mset q*
definition *bino-insert* **where** *bino-insert* ≡ *BinomialHeap.insert*
abbreviation (*input*) *bino-invar* :: ($'a$,$'b$) *BinomialHeap* ⇒ *bool*
where *bino-invar* ≡ λ-. *True*
definition *bino-find* **where** *bino-find* ≡ *BinomialHeap.findMin*
definition *bino-delete* **where** *bino-delete* ≡ *BinomialHeap.deleteMin*
definition *bino-meld* **where** *bino-meld* ≡ *BinomialHeap.meld*

definition *bino-empty* **where** *bino-empty* ≡ λ-::*unit*. *BinomialHeap.empty*
definition *bino-isEmpty* **where** *bino-isEmpty* = *BinomialHeap.isEmpty*

definition [*icf-rec-def*]: *bino-ops* == (|
prio-op-α = *bino-α*,
prio-op-invar = *bino-invar*,
prio-op-empty = *bino-empty*,
prio-op-isEmpty = *bino-isEmpty*,
prio-op-insert = *bino-insert*,
prio-op-find = *bino-find*,
prio-op-delete = *bino-delete*,
prio-op-meld = *bino-meld*
|)

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

Correctness

theorem *bino-empty-impl*: *prio-empty bino-α bino-invar bino-empty*
by (*unfold-locales*, *auto simp add*: *bino-defs*)

theorem *bino-isEmpty-impl*: *prio-isEmpty bino-α bino-invar bino-isEmpty*
by *unfold-locales*
(*simp add*: *bino-defs BinomialHeap.isEmpty-correct BinomialHeap.empty-correct*)

theorem *bino-find-impl*: *prio-find bino-α bino-invar bino-find*
apply *unfold-locales*
apply (*simp add*: *bino-defs BinomialHeap.empty-correct BinomialHeap.findMin-correct*)
done

lemma *bino-insert-impl*: *prio-insert bino-α bino-invar bino-insert*
apply(*unfold-locales*)
apply(*unfold bino-defs*)
apply (*simp-all add*: *BinomialHeap.insert-correct*)
done

lemma *bino-meld-impl*: *prio-meld bino-α bino-invar bino-meld*
apply(*unfold-locales*)
apply(*unfold bino-defs*)
apply(*simp-all add*: *BinomialHeap.meld-correct*)
done

lemma *bino-delete-impl*:
  prio-delete bino-α bino-invar bino-find bino-delete
  apply *intro-locales*
  apply (*rule bino-find-impl*)
  apply(*unfold-locales*)
 apply(*simp-all add*: *bino-defs BinomialHeap.empty-correct BinomialHeap.deleteMin-correct*)
done

setup *Locale-Code.open-block*
interpretation *bino*: *StdPrio bino-ops*
  apply (*rule StdPrio.intro*)
  apply (*simp-all add*: *icf-rec-unf*)
  apply (*rule*
    bino-empty-impl
    bino-isEmpty-impl
    bino-find-impl
    bino-insert-impl
    bino-meld-impl
    bino-delete-impl
)+
  done
interpretation *bino*: *StdPrio-no-invar bino-ops*
  apply *unfold-locales*
  apply (*simp add*: *icf-rec-unf*)
  done
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs bino*›

definition *test-codegen* **where** *test-codegen* = (
  bino.empty,
  bino.isEmpty,
  bino.find,
  bino.insert,
  bino.meld,
  bino.delete
)

export-code *test-codegen* **checking** *SML*

end

3.3.24 Implementation of Priority Queues by Skew Binomial Heaps

theory *SkewPrioImpl*
imports
  Binomial−Heaps.SkewBinomialHeap
  ../spec/PrioSpec

../tools/Record-Intf
../tools/Locale-Code
begin

Definitions

type-synonym $('a,'b)$ *skew* = $('a, 'b)$ *SkewBinomialHeap*

definition *skew-α* **where** *skew-α q* ≡ *SkewBinomialHeap.to-mset q*
definition *skew-insert* **where** *skew-insert* ≡ *SkewBinomialHeap.insert*
abbreviation (*input*) *skew-invar* :: $('a, 'b)$ *SkewBinomialHeap* ⇒ *bool*
where *skew-invar* ≡ λ-. *True*
definition *skew-find* **where** *skew-find* ≡ *SkewBinomialHeap.findMin*
definition *skew-delete* **where** *skew-delete* ≡ *SkewBinomialHeap.deleteMin*
definition *skew-meld* **where** *skew-meld* ≡ *SkewBinomialHeap.meld*
definition *skew-empty* **where** *skew-empty* ≡ λ-::*unit*. *SkewBinomialHeap.empty*
definition *skew-isEmpty* **where** *skew-isEmpty* = *SkewBinomialHeap.isEmpty*

definition [*icf-rec-def*]: *skew-ops* == (|
prio-op-α = *skew-α*,
prio-op-invar = *skew-invar*,
prio-op-empty = *skew-empty*,
prio-op-isEmpty = *skew-isEmpty*,
prio-op-insert = *skew-insert*,
prio-op-find = *skew-find*,
prio-op-delete = *skew-delete*,
prio-op-meld = *skew-meld*
|)

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

Correctness

theorem *skew-empty-impl*: *prio-empty skew-α skew-invar skew-empty*
by (*unfold-locales*, *auto simp add*: *skew-defs*)

theorem *skew-isEmpty-impl*: *prio-isEmpty skew-α skew-invar skew-isEmpty*
by *unfold-locales*
(*simp add*: *skew-defs SkewBinomialHeap.isEmpty-correct SkewBinomialHeap.empty-correct*)

theorem *skew-find-impl*: *prio-find skew-α skew-invar skew-find*
apply *unfold-locales*
apply (*simp add*: *skew-defs SkewBinomialHeap.empty-correct SkewBinomialHeap.findMin-correct*)

```
  done

lemma skew-insert-impl: prio-insert skew-α skew-invar skew-insert
  apply(unfold-locales)
  apply(unfold skew-defs)
  apply (simp-all add: SkewBinomialHeap.insert-correct)
  done

lemma skew-meld-impl: prio-meld skew-α skew-invar skew-meld
  apply(unfold-locales)
  apply(unfold skew-defs)
  apply(simp-all add: SkewBinomialHeap.meld-correct)
  done

lemma skew-delete-impl:
  prio-delete skew-α skew-invar skew-find skew-delete
  apply intro-locales
  apply (rule skew-find-impl)
  apply(unfold-locales)
  apply(simp-all add: skew-defs SkewBinomialHeap.empty-correct SkewBinomial-
Heap.deleteMin-correct)
done

setup Locale-Code.open-block
interpretation skew: StdPrio skew-ops
  apply (rule StdPrio.intro)
  apply (simp-all add: icf-rec-unf)
  apply (rule
    skew-empty-impl
    skew-isEmpty-impl
    skew-find-impl
    skew-insert-impl
    skew-meld-impl
    skew-delete-impl
  )+
  done
interpretation skew: StdPrio-no-invar skew-ops
  by unfold-locales (simp add: icf-rec-unf)
setup Locale-Code.close-block

definition test-codegen where test-codegen ≡ (
  skew.empty,
  skew.isEmpty,
  skew.find,
  skew.insert,
  skew.meld,
  skew.delete
)
```

export-code *test-codegen* **checking** *SML*

end

3.3.25 Implementation of Annotated Lists by 2-3 Finger Trees

theory *FTAnnotatedListImpl*
imports
 Finger−Trees.FingerTree
 ../tools/Locale-Code
 ../tools/Record-Intf
 ../spec/AnnotatedListSpec
begin

type-synonym $('a,'b)$ *ft* = $('a,'b)$ *FingerTree*

Definitions

definition *ft-α* **where**
 ft-α ≡ *FingerTree.toList*
abbreviation (*input*) *ft-invar* :: $('a,'b)$ *FingerTree* ⇒ *bool* **where**
 ft-invar ≡ λ-. *True*
definition *ft-empty* **where**
 ft-empty ≡ λ-::*unit*. *FingerTree.empty*
definition *ft-isEmpty* **where**
 ft-isEmpty ≡ *FingerTree.isEmpty*
definition *ft-count* **where**
 ft-count ≡ *FingerTree.count*
definition *ft-consl* **where**
 ft-consl e a s = *FingerTree.lcons* (*e*,*a*) *s*
definition *ft-consr* **where**
 ft-consr s e a = *FingerTree.rcons s* (*e*,*a*)
definition *ft-head* **where**
 ft-head ≡ *FingerTree.head*
definition *ft-tail* **where**
 ft-tail ≡ *FingerTree.tail*
definition *ft-headR* **where**
 ft-headR ≡ *FingerTree.headR*
definition *ft-tailR* **where**
 ft-tailR ≡ *FingerTree.tailR*
definition *ft-foldl* **where**
 ft-foldl ≡ *FingerTree.foldl*
definition *ft-foldr* **where**
 ft-foldr ≡ *FingerTree.foldr*
definition *ft-app* **where**
 ft-app ≡ *FingerTree.app*
definition *ft-annot* **where**
 ft-annot ≡ *FingerTree.annot*

definition *ft-splits* **where**
ft-splits ≡ *FingerTree.splitTree*

lemmas *ft-defs* =
ft-α-def
ft-empty-def
ft-isEmpty-def
ft-count-def
ft-consl-def
ft-consr-def
ft-head-def
ft-tail-def
ft-headR-def
ft-tailR-def
ft-foldl-def
ft-foldr-def
ft-app-def
ft-annot-def
ft-splits-def

lemma *ft-empty-impl*: *al-empty ft-α ft-invar ft-empty*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.empty-correct*)
done

lemma *ft-consl-impl*: *al-consl ft-α ft-invar ft-consl*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.lcons-correct*)
done

lemma *ft-consr-impl*: *al-consr ft-α ft-invar ft-consr*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.rcons-correct*)
done

lemma *ft-isEmpty-impl*: *al-isEmpty ft-α ft-invar ft-isEmpty*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.isEmpty-correct FingerTree.empty-correct*)
done

lemma *ft-count-impl*: *al-count ft-α ft-invar ft-count*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.count-correct*)
done

lemma *ft-head-impl*: *al-head ft-α ft-invar ft-head*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.head-correct FingerTree.empty-correct*)
done

lemma *ft-tail-impl*: *al-tail ft-α ft-invar ft-tail*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.tail-correct FingerTree.empty-correct*)
done

lemma *ft-headR-impl*: *al-headR ft-α ft-invar ft-headR*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.headR-correct FingerTree.empty-correct*)
done

lemma *ft-tailR-impl*: *al-tailR ft-α ft-invar ft-tailR*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.tailR-correct FingerTree.empty-correct*)
done

lemma *ft-foldl-impl*: *al-foldl ft-α ft-invar ft-foldl*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.foldl-correct*)
done

lemma *ft-foldr-impl*: *al-foldr ft-α ft-invar ft-foldr*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.foldr-correct*)
done

lemma *ft-foldl-cunfold*[*code-unfold*]:
List.foldl f σ (*ft-α t*) = *ft-foldl f σ t*
apply (*auto simp add*: *ft-defs FingerTree.foldl-correct*)
done

lemma *ft-foldr-cunfold*[*code-unfold*]:
List.foldr f (*ft-α t*) *σ* = *ft-foldr f t σ*
apply (*auto simp add*: *ft-defs FingerTree.foldr-correct*)
done

lemma *ft-app-impl*: *al-app ft-α ft-invar ft-app*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.app-correct*)
done

lemma *ft-annot-impl*: *al-annot ft-α ft-invar ft-annot*
apply *unfold-locales*
apply (*auto simp add*: *ft-defs FingerTree.annot-correct*)
done

lemma *ft-splits-impl*: *al-splits ft-α ft-invar ft-splits*
apply *unfold-locales*
apply (*unfold ft-defs*)

apply (*simp only*: *FingerTree.annot-correct*[*symmetric*])
apply (*frule* (*3*) *FingerTree.splitTree-correct*(*1*))
apply (*frule* (*3*) *FingerTree.splitTree-correct*(*2*))
apply (*frule* (*3*) *FingerTree.splitTree-correct*(*3*))
apply (*simp only*: *FingerTree.annot-correct*[*symmetric*])
apply *simp*
done

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*
|)

setup *Locale-Code.open-block*
interpretation *ft*: *StdAL ft-ops*
apply (*rule StdAL.intro*)
apply (*simp-all add*: *icf-rec-unf*)
apply (*rule*
ft-empty-impl
ft-consl-impl
ft-consr-impl
ft-isEmpty-impl
ft-count-impl
ft-head-impl
ft-tail-impl
ft-headR-impl
ft-tailR-impl
ft-foldl-impl
ft-foldr-impl
ft-app-impl
ft-annot-impl
ft-splits-impl
)+
done
interpretation *ft*: *StdAL-no-invar ft-ops*
by (*unfold-locales*) (*simp add*: *icf-rec-unf*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs ft*›

definition *test-codegen* ≡ (
ft.empty,
ft.isEmpty,
ft.count,
ft.consl,
ft.consr,
ft.head,
ft.tail,
ft.headR,
ft.tailR,
ft.app,
ft.annot,
ft.splits,
ft.foldl,
ft.foldr
)

export-code *test-codegen* **checking** *SML*

end

3.3.26 Implementation of Priority Queues by Finger Trees

theory *FTPrioImpl*
imports *FTAnnotatedListImpl*
../gen-algo/PrioByAnnotatedList
begin

type-synonym (*'a*,*'p*) *alprioi* = (*unit*, (*'a*, *'p*) *Prio*) *FingerTree*

setup *Locale-Code.open-block*
interpretation *alprio-ga*: *alprio ft-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *alprioi-ops* ≡ *alprio-ga.alprio-ops*

setup *Locale-Code.open-block*
interpretation *alprioi*: *StdPrio alprioi-ops*
unfolding *alprioi-ops-def*
by (*rule alprio-ga.alprio-ops-impl*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs alprioi*›

definition *test-codegen* **where** *test-codegen* ≡ (
  alprioi.empty,
  alprioi.isEmpty,
  alprioi.insert,
  alprioi.find,
  alprioi.delete,
  alprioi.meld
)

export-code *test-codegen* **checking** *SML*

end

3.3.27 Implementation of Unique Priority Queues by Finger Trees

theory *FTPrioUniqueImpl*
imports
  FTAnnotatedListImpl
  ../gen-algo/PrioUniqueByAnnotatedList
begin

Definitions

type-synonym ('a,'b) *aluprioi* = (*unit*, ('a, 'b) *LP*) *FingerTree*

setup *Locale-Code.open-block*
interpretation *aluprio-ga*: *aluprio ft-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*

definition [*icf-rec-def*]: *aluprioi-ops* ≡ *aluprio-ga.aluprio-ops*

setup *Locale-Code.open-block*
interpretation *aluprioi*: *StdUprio aluprioi-ops*
  unfolding *aluprioi-ops-def*
  by (*rule aluprio-ga.aluprio-ops-impl*)
setup *Locale-Code.close-block*

setup ‹*ICF-Tools.revert-abbrevs aluprioi*›

definition *test-codegen* **where** *test-codegen* ≡ (
  aluprioi.empty,
  aluprioi.isEmpty,
  aluprioi.insert,
  aluprioi.pop,
  aluprioi.prio
)

export-code *test-codegen* **checking** *SML*

end

3.4 Entry Points

3.4.1 Standard Collections

theory *Collections*
imports
  ICF-Impl
  ICF-Refine-Monadic
  ICF-Autoref

  DatRef

begin

This theory summarizes the components of the Isabelle Collection Framework.

end

3.4.2 Backwards Compatibility for Version 1

theory *CollectionsV1*
imports *Collections*
begin

This theory defines some stuff to establish (partial) backwards compatibility with ICF Version 1.

  attribute-setup *locale-witness-add* = ‹
    Scan.succeed (*Locale.witness-add*)
› *Add witness for locale instantiation. HACK*, *use*
      sublocale or interpretation whereever possible!

Iterators

We define all the monomorphic iterator locales

Set **locale** *set-iteratei* = *finite-set* α *invar* **for** α :: $'s \Rightarrow 'x$ *set* **and** *invar* +
  fixes *iteratei* :: $'s \Rightarrow ('x, '\sigma)$ *set-iterator*

  assumes *iteratei-rule*: *invar* $S \Longrightarrow$ *set-iterator* (*iteratei* S) (α S)
begin
  lemma *iteratei-rule-P*:
    ⟦
      invar S;
      I (α S) $\sigma 0$;
      $!!x\ it\ \sigma.$ ⟦ $c\ \sigma$; $x \in it$; $it \subseteq \alpha\ S$; $I\ it\ \sigma$ ⟧ $\Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$;

```
    !!σ. I {} σ ⟹ P σ;
    !!σ it. ⟦ it ⊆ α S; it ≠ {}; ¬ c σ; I it σ ⟧ ⟹ P σ
  ⟧ ⟹ P (iteratei S c f σ0)
 apply (rule set-iterator-rule-P [OF iteratei-rule, of S I σ0 c f P])
 apply simp-all
done

lemma iteratei-rule-insert-P:
  ⟦
    invar S;
    I {} σ0;
    !!x it σ. ⟦ c σ; x ∈ α S − it; it ⊆ α S; I it σ ⟧ ⟹ I (insert x it) (f x σ);
    !!σ. I (α S) σ ⟹ P σ;
    !!σ it. ⟦ it ⊆ α S; it ≠ α S; ¬ c σ; I it σ ⟧ ⟹ P σ
  ⟧ ⟹ P (iteratei S c f σ0)
  apply (rule set-iterator-rule-insert-P [OF iteratei-rule, of S I σ0 c f P])
  apply simp-all
done
```

Versions without break condition.

```
lemma iterate-rule-P:
  ⟦
    invar S;
    I (α S) σ0;
    !!x it σ. ⟦ x ∈ it; it ⊆ α S; I it σ ⟧ ⟹ I (it − {x}) (f x σ);
    !!σ. I {} σ ⟹ P σ
  ⟧ ⟹ P (iteratei S (λ-. True) f σ0)
 apply (rule set-iterator-no-cond-rule-P [OF iteratei-rule, of S I σ0 f P])
 apply simp-all
done

lemma iterate-rule-insert-P:
  ⟦
    invar S;
    I {} σ0;
    !!x it σ. ⟦ x ∈ α S − it; it ⊆ α S; I it σ ⟧ ⟹ I (insert x it) (f x σ);
    !!σ. I (α S) σ ⟹ P σ
  ⟧ ⟹ P (iteratei S (λ-. True) f σ0)
  apply (rule set-iterator-no-cond-rule-insert-P [OF iteratei-rule, of S I σ0 f P])
  apply simp-all
done
end

lemma set-iteratei-I :
assumes ⋀s. invar s ⟹ set-iterator (iti s) (α s)
shows set-iteratei α invar iti
proof
  fix s
  assume invar-s: invar s
```

from *assms(1)[OF invar-s]* **show** *it-OK*: *set-iterator* (*iti s*) (α *s*) **.**

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

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

lemma *iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
assumes *MINV*: *invar m*
assumes *I0*: *I* ((α *m*)) $\sigma 0$
assumes *IP*: !!*k it* σ. ⟦ *k* $\in$ *it*; $\forall j{\in}it.\ k{\le}j$; $\forall j{\in}(\alpha\ m) - it.\ j{\le}k$; *it* $\subseteq$ (α *m*);
I it σ ⟧
$\Longrightarrow$ *I* (*it* $-$ {*k*}) (*f k* σ)
assumes *IF*: !!σ. *I* {} σ $\Longrightarrow$ *P* σ
shows *P* (*iterateoi m* (λ-. *True*) *f* $\sigma 0$)
apply (*rule iterateoi-rule-P* [**where** *I* = *I*])
apply (*simp-all add*: *assms*)
done
end

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

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

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

  lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
    assumes *MINV*: *invar m*
    assumes *I0*: *I* ((α *m*)) $\sigma 0$

assumes *IP*: !!k *it* σ. $\llbracket$
$k \in$ *it*;
$\forall j \in$*it*. $k \geq j$;
$\forall j \in (\alpha\ m) -$ *it*. $j \geq k$;
it $\subseteq (\alpha\ m)$;
I it σ
$\rrbracket \Longrightarrow I$ (*it* $- \{k\}$) ($f\ k\ \sigma$)
assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow P\ \sigma$
shows *P* (*reverse-iterateoi m* (λ-. *True*) *f* $\sigma 0$)
apply (*rule reverse-iterateoi-rule-P* [**where** $I = I$])
apply (*simp-all add*: *assms*)
done
end

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

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

lemma (**in** *poly-set-iteratei*) *v1-iteratei-impl*:
set-iteratei α *invar iteratei*
by *unfold-locales* (*rule iteratei-correct*)
lemma (**in** *poly-set-iterateoi*) *v1-iterateoi-impl*:
set-iterateoi α *invar iterateoi*
by *unfold-locales* (*rule iterateoi-correct*)
lemma (**in** *poly-set-rev-iterateoi*) *v1-reverse-iterateoi-impl*:
set-reverse-iterateoi α *invar rev-iterateoi*
by *unfold-locales* (*rule rev-iterateoi-correct*)

declare (**in** *poly-set-iteratei*) *v1-iteratei-impl*[*locale-witness-add*]
declare (**in** *poly-set-iterateoi*) *v1-iterateoi-impl*[*locale-witness-add*]
declare (**in** *poly-set-rev-iterateoi*)
v1-reverse-iterateoi-impl[*locale-witness-add*]

Map **locale** *map-iteratei* = *finite-map* α *invar* **for** α :: $'s \Rightarrow 'u \rightharpoonup 'v$ **and** *invar*
+
fixes *iteratei* :: $'s \Rightarrow ('u \times 'v,'\sigma)$ *set-iterator*

assumes *iteratei-rule*: *invar m* $\Longrightarrow$ *map-iterator* (*iteratei m*) (α *m*)
begin

lemma *iteratei-rule-P*:
  assumes *invar m*
    and *I0*: $I\ (dom\ (\alpha\ m))\ \sigma 0$
    and *IP*: $!!k\ v\ it\ \sigma.\ \llbracket\ c\ \sigma;\ k \in it;\ \alpha\ m\ k = Some\ v;\ it \subseteq dom\ (\alpha\ m);\ I\ it\ \sigma\ \rrbracket$
        $\Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
    and *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
    and *II*: $!!\sigma\ it.\ \llbracket\ it \subseteq dom\ (\alpha\ m);\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma\ \rrbracket \Longrightarrow P\ \sigma$
  shows $P\ (iteratei\ m\ c\ f\ \sigma 0)$
  using *map-iterator-rule-P* [*OF iteratei-rule*, *of m I* $\sigma 0$ *c f P*]
  by (*simp-all add*: *assms*)

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

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

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

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

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

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

    lemma *iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
      assumes *MINV*: *invar m*
      assumes *I0*: *I* (*dom* (α *m*)) $\sigma 0$
      assumes *IP*: !!*k v it* σ. $[\![$ *k* $\in$ *it*; $\forall j\in it.\ k\leq j$; $\forall j\in dom\ (\alpha\ m) - it.\ j\leq k$; α *m*
k = *Some v*; *it* $\subseteq$ *dom* (α *m*); *I it* σ $]\!]$

$\Longrightarrow I\ (it - \{k\})\ (f\ (k,\ v)\ \sigma)$
assumes *IF*: $!!\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$
shows P (*iterateoi m* (λ-. *True*) $f\ \sigma 0$)
using *map-iterator-linord-rule-P* [*OF iterateoi-rule*, *of m I* $\sigma 0$ λ-. *True f P*]
assms
by *simp*
end

lemma *map-iterateoi-I* :
assumes $\bigwedge m$. *invar m* $\Longrightarrow$ *map-iterator-linord* (*itoi m*) (α *m*)
shows *map-iterateoi* α *invar itoi*
proof
fix *m*
assume *invar-m*: *invar m*
from *assms*(*1*)[*OF invar-m*] **show** *it-OK*: *map-iterator-linord* (*itoi m*) (α *m*) **.**

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

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

P] *assms*
  by *simp*

  lemma *reverse-iterateo-rule-P*[*case-names minv inv0 inv-pres i-complete*]:
    assumes *MINV*: *invar m*
    assumes *I0*: *I* (*dom* (α *m*)) $\sigma 0$
    assumes *IP*: !!*k v it* σ. ⟦
      $k \in it$;
      $\forall j \in it.\ k \geq j$;
      $\forall j \in dom\ (\alpha\ m) - it.\ j \geq k$;
      $\alpha\ m\ k = Some\ v$;
      $it \subseteq dom\ (\alpha\ m)$;
      I it σ
    ⟧ $\Longrightarrow$ *I* (*it* $-$ {*k*}) (*f* (*k*, *v*) σ)
    assumes *IF*: !!σ. *I* {} $\sigma \Longrightarrow$ *P* σ
    shows *P* (*reverse-iterateoi m* (λ-. *True*) *f* $\sigma 0$)
  using *map-iterator-rev-linord-rule-P*[*OF reverse-iterateoi-rule*, *of m I* $\sigma 0$ λ-. *True f P*] *assms*
  by *simp*
end

lemma *map-reverse-iterateoi-I* :
assumes $\bigwedge$*m*. *invar m* $\Longrightarrow$ *map-iterator-rev-linord* (*ritoi m*) (α *m*)
shows *map-reverse-iterateoi* α *invar ritoi*
proof
  fix *m*
  assume *invar-m*: *invar m*
  from *assms*(*1*)[*OF invar-m*] **show** *it-OK*: *map-iterator-rev-linord* (*ritoi m*) (α *m*) **.**

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

lemma (**in** *poly-map-iteratei*) *v1-iteratei-impl*:
  map-iteratei α *invar iteratei*
  by *unfold-locales* (*rule iteratei-correct*)
lemma (**in** *poly-map-iterateoi*) *v1-iterateoi-impl*:
  map-iterateoi α *invar iterateoi*
  by *unfold-locales* (*rule iterateoi-correct*)
lemma (**in** *poly-map-rev-iterateoi*) *v1-reverse-iterateoi-impl*:
  map-reverse-iterateoi α *invar rev-iterateoi*
  by *unfold-locales* (*rule rev-iterateoi-correct*)

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

```
local-setup ‹let
  val thy = @{theory}
  val ctxt = Proof-Context.init-global thy;
  val pats = [
  hs,hm,
  rs,rm,
  ls,lm,lsi,lmi,lsnd,lss,
  ts,tm,
  ias,iam,
  ahs,ahm,
  bino,
  fifo,
  ft,
  alprioi,
  aluprioi,
  skew
  ];

  val {const-space, constants, ...} = Sign.consts-of thy |> Consts.dest
   val clist = Name-Space.extern-entries true ctxt const-space constants |> map
(apfst #1)

  fun abbrevs-for pat = clist
  |> map-filter (fn (n,-) => case Long-Name.explode n of
      [-,prefix,opname] =>
        if prefix = pat then let
            val aname = prefix ^ - ^ opname
            val rhs = Proof-Context.read-term-abbrev ctxt n
          in SOME (aname,rhs) end
        else NONE
    | - => NONE);

  fun do-abbrevs pat lthy = let
    val abbrevs = abbrevs-for pat;
  in
    case abbrevs of [] => (warning (No stuff found for ^pat); lthy)
    | - => let
      (*val - = tracing (Defining  ^ pat ^ -xxx ...);*)
      val lthy = fold (fn (name,rhs) =>
        Local-Theory.abbrev
          Syntax.mode-input
          ((Binding.name name,NoSyn),rhs) #> #2
      ) abbrevs lthy
      (*val - = tracing Done;*)
```

in lthy end
end
in
fold do-abbrevs pats
end

›

lemmas *hs-correct* = *hs.correct*
lemmas *hm-correct* = *hm.correct*
lemmas *rs-correct* = *rs.correct*
lemmas *rm-correct* = *rm.correct*
lemmas *ls-correct* = *ls.correct*
lemmas *lm-correct* = *lm.correct*
lemmas *lsi-correct* = *lsi.correct*
lemmas *lmi-correct* = *lmi.correct*
lemmas *lsnd-correct* = *lsnd.correct*
lemmas *lss-correct* = *lss.correct*
lemmas *ts-correct* = *ts.correct*
lemmas *tm-correct* = *tm.correct*
lemmas *ias-correct* = *ias.correct*
lemmas *iam-correct* = *iam.correct*
lemmas *ahs-correct* = *ahs.correct*
lemmas *ahm-correct* = *ahm.correct*
lemmas *bino-correct* = *bino.correct*
lemmas *fifo-correct* = *fifo.correct*
lemmas *ft-correct* = *ft.correct*
lemmas *alprioi-correct* = *alprioi.correct*
lemmas *aluprioi-correct* = *aluprioi.correct*
lemmas *skew-correct* = *skew.correct*

locale *list-enqueue* = *list-appendr*
locale *list-dequeue* = *list-removel*

locale *list-push* = *list-appendl*
locale *list-pop* = *list-remover*
locale *list-top* = *list-leftmost*
locale *list-bot* = *list-rightmost*

instantiation *rbt* :: ({*equal*,*linorder*},*equal*) *equal*
begin

definition *equal-class.equal* (*r* :: ('*a*, '*b*) *rbt*) *r'*
== *RBT.impl-of r* = *RBT.impl-of r'*

```
  instance
    apply intro-classes
    apply (simp add: equal-rbt-def RBT.impl-of-inject)
    done
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
begin

Configurations

lemmas *tyrel-dflt-nat-set* =
  ty-REL[**where** *'a=nat set* **and** *R=⟨Id⟩dflt-rs-rel*]

lemmas *tyrel-dflt-bool-set* =
  ty-REL[**where** *'a=bool set* **and** *R=⟨Id⟩list-set-rel*]

lemmas *tyrel-dflt-nat-map* =
  ty-REL[**where** *R=⟨nat-rel,Rv⟩dflt-rm-rel*] **for** *Rv*

lemmas *tyrel-dflt-old* = *tyrel-dflt-nat-set tyrel-dflt-bool-set tyrel-dflt-nat-map*

lemmas *tyrel-dflt-linorder-set* =
  ty-REL[**where** *R=⟨Id::('a::linorder×'a) set⟩dflt-rs-rel*]

lemmas *tyrel-dflt-linorder-map* =
  ty-REL[**where** *R=⟨Id::('a::linorder×'a) set,R⟩dflt-rm-rel*] **for** *R*

lemmas *tyrel-dflt-bool-map* =
ty-REL[**where** *R*=⟨*Id*::(*bool*×*bool*) *set*,*R*⟩*list-map-rel*] **for** *R*

lemmas *tyrel-dflt* = *tyrel-dflt-linorder-set tyrel-dflt-bool-set tyrel-dflt-linorder-map tyrel-dflt-bool-map*

declare *tyrel-dflt*[*autoref-tyrel*]

local-setup ‹
let open Autoref-Fix-Rel in
declare-prio Gen−AHM−map−hashable
@{*term* ⟨*Rk*::(-×-::*hashable*) *set*,*Rv*⟩*ahm-rel bhc*} *PR-LAST* #>
declare-prio Gen−RBT−map−linorder
@{*term* ⟨*Rk*::(-×-::*linorder*) *set*,*Rv*⟩*rbt-map-rel lt*} *PR-LAST* #>
declare-prio Gen−AHM−map @{*term* ⟨*Rk*,*Rv*⟩*ahm-rel bhc*} *PR-LAST* #>
declare-prio Gen−RBT−map @{*term* ⟨*Rk*,*Rv*⟩*rbt-map-rel lt*} *PR-LAST*
end
›

Fallbacks

local-setup ‹
let open Autoref-Fix-Rel in
declare-prio Gen−List−Set @{*term* ⟨*R*⟩*list-set-rel*} *PR-LAST* #>
declare-prio Gen−List−Map @{*term* ⟨*R*⟩*list-map-rel*} *PR-LAST*
end
›

Quick test of setup:

context begin
private schematic-goal *test-dflt-tyrel1*: (*?c*::*?'c*,{*1*,*2*,*3*::*int*})∈*?R*
by *autoref*
private lemmas *asm-rl*[*of* -∈⟨*int-rel*⟩*dflt-rs-rel*, *OF test-dflt-tyrel1*]

private schematic-goal *test-dflt-tyrel2*: (*?c*::*?'c*,{*True*, *False*})∈*?R*
by *autoref*
private lemmas *asm-rl*[*of* -∈⟨*bool-rel*⟩*list-set-rel*, *OF test-dflt-tyrel2*]

private schematic-goal *test-dflt-tyrel3*: (*?c*::*?'c*,[*1*::*int*↦*0*::*nat*])∈*?R*
by *autoref*
private lemmas *asm-rl*[*of* -∈⟨*int-rel*,*nat-rel*⟩*dflt-rm-rel*, *OF test-dflt-tyrel3*]

private schematic-goal *test-dflt-tyrel4*: (*?c*::*?'c*,[*False*↦*0*::*nat*])∈*?R*
by *autoref*
private lemmas *asm-rl*[*of* -∈⟨*bool-rel*,*nat-rel*⟩*list-map-rel*, *OF test-dflt-tyrel4*]

end

end

4.1.2 Entry Point with genCF and original ICF

theory *Refine-Dflt-ICF*
imports
Refine-Monadic.Refine-Monadic
GenCF/GenCF
ICF/Collections
begin

Contains the Monadic Refinement Framework, the generic collection framework and the original Isabelle Collection Framework

```
local-setup ‹
  let open Autoref-Fix-Rel in
    declare-prio Gen−RBT−set @{term ⟨R⟩dflt-rs-rel} PR-LAST #>
    declare-prio RBT−set @{term ⟨R⟩rs.rel} PR-LAST #>
    declare-prio Hash−set @{term ⟨R⟩hs.rel} PR-LAST #>
    declare-prio List−set @{term ⟨R⟩lsi.rel} PR-LAST
  end
›

local-setup ‹
  let open Autoref-Fix-Rel in
    declare-prio RBT−map @{term ⟨Rk,Rv⟩rm.rel} PR-LAST #>
    declare-prio Hash−map @{term ⟨Rk,Rv⟩hm.rel} PR-LAST #>
    (∗ declare-prio Gen−RBT−map @{term ⟨Rk,Rv⟩rbt-map-rel ?cmp} PR-LAST
#>∗)
    declare-prio List−map @{term ⟨Rk,Rv⟩lmi.rel} PR-LAST
  end
›
```

Fallbacks

```
local-setup ‹
  let open Autoref-Fix-Rel in
    declare-prio Gen−List−Set @{term ⟨R⟩list-set-rel} PR-LAST #>
    declare-prio Gen−List−Map @{term ⟨Rk,Rv⟩list-map-rel} PR-LAST
  end
›
```

class *id-refine*

instance *nat* :: *id-refine* **..**
instance *bool* :: *id-refine* **..**
instance *int* :: *id-refine* **..**

lemmas [*autoref-tyrel*] =
  ty-REL[**where** $'a$=*nat* **and** *R*=*nat-rel*]
  ty-REL[**where** $'a$=*int* **and** *R*=*int-rel*]
  ty-REL[**where** $'a$=*bool* **and** *R*=*bool-rel*]
lemmas [*autoref-tyrel*] =
  ty-REL[**where** $'a$=*nat set* **and** *R*=⟨*Id*⟩*rs.rel*]
  ty-REL[**where** $'a$=*int set* **and** *R*=⟨*Id*⟩*rs.rel*]
  ty-REL[**where** $'a$=*bool set* **and** *R*=⟨*Id*⟩*lsi.rel*]
lemmas [*autoref-tyrel*] =
  ty-REL[**where** $'a$=(*nat* ⇀ $'b$), **where** *R*=⟨*nat-rel*,*Rv*⟩*dflt-rm-rel*]
  ty-REL[**where** $'a$=(*int* ⇀ $'b$), **where** *R*=⟨*int-rel*,*Rv*⟩*dflt-rm-rel*]
  ty-REL[**where** $'a$=(*bool* ⇀ $'b$), **where** *R*=⟨*bool-rel*,*Rv*⟩*dflt-rm-rel*]
  for *Rv*

lemmas [*autoref-tyrel*] =
  ty-REL[**where** $'a$=(*nat* ⇀ $'b$::*id-refine*), **where** *R*=⟨*nat-rel*,*Id*⟩*rm.rel*]
  ty-REL[**where** $'a$=(*int* ⇀ $'b$::*id-refine*), **where** *R*=⟨*int-rel*,*Id*⟩*rm.rel*]
  ty-REL[**where** $'a$=(*bool* ⇀ $'b$::*id-refine*), **where** *R*=⟨*bool-rel*,*Id*⟩*rm.rel*]

end

4.1.3 Entry Point with only the ICF

theory *Refine-Dflt-Only-ICF*
imports
  Refine-Monadic.Refine-Monadic
  ICF/Collections
begin

Contains the Monadic Refinement Framework and the original Isabelle Collection Framework. The generic collection framework is not included

local-setup ‹
  let open Autoref-Fix-Rel in
    declare-prio RBT−set @{*term* ⟨*R*⟩*rs.rel*} *PR-LAST* #>
    declare-prio Hash−set @{*term* ⟨*R*⟩*hs.rel*} *PR-LAST* #>
    declare-prio List−set @{*term* ⟨*R*⟩*lsi.rel*} *PR-LAST*
  end
›

local-setup ‹
  let open Autoref-Fix-Rel in
    declare-prio RBT−map @{*term* ⟨*Rk*,*Rv*⟩*rm.rel*} *PR-LAST* #>
    declare-prio Hash−map @{*term* ⟨*Rk*,*Rv*⟩*hm.rel*} *PR-LAST* #>
    declare-prio List−map @{*term* ⟨*Rk*,*Rv*⟩*lmi.rel*} *PR-LAST*
  end
›

end

Chapter 5

Userguides

This chapter contains various userguides.

5.1 Old Monadic Refinement Framework Userguide

5.1.1 Introduction

This is the old userguide from Refine-Monadic. It contains the manual approach of using the mondaic refinement framework with the Isabelle Collection Framework. An alternative, more simple approach is provided by the Automatic Refinement Framework and the Generic Collection Framework.

The Isabelle/HOL refinement framework is a library that supports program and data refinement.

Programs are specified using a nondeterminism monad: An element of the monad type is either a set of results, or the special element *FAIL*, that indicates a failed assertion.

The bind-operation of the monad applies a function to all elements of the result-set, and joins all possible results.

On the monad type, an ordering $\leq$ is defined, that is lifted subset ordering, where *FAIL* is the greatest element. Intuitively, $S \leq S'$ means that program S refines program S', i.e., all results of S are also results of S', and S may only fail if S' also fails.

5.1.2 Guided Tour

In this section, we provide a small example program development in our framework. All steps of the development are heavily commented.

Defining Programs

A program is defined using the Haskell-like do-notation, that is provided by the Isabelle/HOL library. We start with a simple example, that iterates over a set of numbers, and computes the maximum value and the sum of all elements.

definition *sum-max* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max V ≡ *do* {
    (-,*s*,*m*) ← *WHILE* (λ(*V*,*s*,*m*). *V*≠{}) (λ(*V*,*s*,*m*). *do* {
      x←*SPEC* (λ*x*. *x*∈*V*);
      let V=*V*−{*x*};
      let s=*s*+*x*;
      let m=*max m x*;
      RETURN (*V*,*s*,*m*)
    }) (*V*,*0*,*0*);
    RETURN (*s*,*m*)
  }

The type of the nondeterminism monad is *′a nres*, where *′a* is the type of the results. Note that this program has only one possible result, however, the order in which we iterate over the elements of the set is unspecified.

This program uses the following statements provided by our framework: While-loops, bindings, return, and specification. We briefly explain the statements here. A complete reference can be found in Section 5.1.5.

A while-loop has the form *WHILE* b f 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←(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* ⇒ *′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* $V_0 \equiv \lambda(V,s::nat,m).$
$V \subseteq V_0$
$\wedge\ s=\sum (V_0 - V)$
$\wedge\ m=(\mathit{if}\ (V_0 - V)=\{\}\ \mathit{then}\ 0\ \mathit{else}\ Max\ (V_0 - V))$
$\wedge\ \mathit{finite}\ (V_0 - V)$

We have extracted the most complex verification condition — that the invariant is preserved by the loop body — to an own lemma. For complex proofs, it is always a good idea to do that, as it makes the proof more readable.

lemma *sum-max-invar-step*:
assumes $x \in V$ $\quad$ *sum-max-invar* $V_0\ (V,s,m)$
shows *sum-max-invar* $V_0\ (V - \{x\}, s+x, max\ m\ x)$

In our case the proof is rather straightforward, it only requires the lemma *it-step-insert-iff*, that handles the $V_0 - (V - \{x\})$ terms that occur in the invariant.

using *assms* **unfolding** *sum-max-invar-def* **by** (*auto simp*: *it-step-insert-iff*)

The correctness is now proved by first invoking the verification condition generator, and then discharging the verification conditions by *auto*. Note that we have to apply the *sum-max-invar-step* lemma, *before* we unfold the definition of the invariant to discharge the remaining verification conditions.

theorem *sum-max-correct*:
assumes *PRE*: $V \neq \{\}$
shows *sum-max* $V \leq SPEC\ (\lambda(s,m).\ s=\sum V \wedge m=Max\ V)$

The precondition $V \neq \{\}$ is necessary, as the *Max*-operator from Isabelle/HOL's standard library is not defined for empty sets.

using *PRE* **unfolding** *sum-max-def*
apply (*intro WHILE-rule*[**where** *I=sum-max-invar V*] *refine-vcg*) — Invoke vcg

Note that we have explicitely instantiated the rule for the while-loop with the invariant. If this is not done, the verification condition generator will stop at the WHILE-loop.

apply (*auto intro*: *sum-max-invar-step*) — Discharge step
unfolding *sum-max-invar-def* — Unfold invariant definition
apply (*auto*) — Discharge remaining goals
done

In this proof, we specified the invariant explicitely. Alternatively, we may annotate the invariant at the while loop, using the syntax $WHILE^{I}\ b\ f\ \sigma_0$. Then, the verification condition generator will use the annotated invariant automatically.

Total Correctness Now, we reformulate our program to use a total correct while loop, and annotate the invariant at the loop. The invariant is strengthened by stating that the set of elements is finite.

definition *sum-max′-invar* V_0 σ ≡
  sum-max-invar V_0 σ
  ∧ (*let* (*V*,-,-)=σ *in finite* (V_0−*V*))

definition *sum-max′* :: *nat set* ⇒ (*nat*×*nat*) *nres* **where**
  sum-max′ V ≡ *do* {
    (-,*s*,*m*) ← $WHILE_T{}^{sum\text{-}max'\text{-}invar\ V}$ (λ(*V*,*s*,*m*). *V*≠{}) (λ(*V*,*s*,*m*). *do* {
      x←*SPEC* (λ*x*. *x*∈*V*);
      let V=*V*−{*x*};
      let s=*s*+*x*;
      let m=*max m x*;
      RETURN (*V*,*s*,*m*)
    }) (*V*,*0*,*0*);
    RETURN (*s*,*m*)
  }

theorem *sum-max′-correct*:
  assumes *NE*: *V*≠{} **and** *FIN*: *finite V*
  shows *sum-max′ V* ≤ *SPEC* (λ(*s*,*m*). *s*=∑ *V* ∧ *m*=*Max V*)
  using *NE FIN* **unfolding** *sum-max′-def*
  apply (*intro refine-vcg*) — Invoke vcg

This time, the verification condition generator uses the annotated invariant. Moreover, it leaves us with a variant. We have to specify a well-founded relation, and show that the loop body respects this relation. In our case, the set *V* decreases in each step, and is initially finite. We use the relation *finite-psubset* and the *inv-image* combinator from the Isabelle/HOL standard library.

apply (*subgoal-tac wf* (*inv-image finite-psubset fst*),
assumption) — Instantiate variant
apply *simp* — Show variant well-founded

unfolding *sum-max′-invar-def* — Unfold definition of invariant
apply (*auto intro*: *sum-max-invar-step*) — Discharge step

unfolding *sum-max-invar-def* — Unfold definition of invariant completely
apply (*auto intro*: *finite-subset*) — Discharge remaining goals
done

Refinement

The next step in the program development is to refine the initial program towards an executable program. This usually involves both, program refinement and data refinement. Program refinement means changing the structure of the program. Usually, some specification statements are replaced by more concrete implementations. Data refinement means changing the used data types towards implementable data types.

In our example, we implement the set V with a distinct list, and replace the specification statement *SPEC* ($\lambda x.\ x \in V$) by the head operation on distinct lists. For the lists, we use the list-set data structure provided by the Isabelle Collection Framework [2, 4].

For this example, we write the refined program ourselves. An automation of this task can be achieved with the automatic refinement tool, which is available as a prototype in Refine-Autoref. Usage examples are in ex/Automatic-Refinement.

definition *sum-max-impl* :: *nat ls* ⇒ (*nat*×*nat*) *nres* **where**
sum-max-impl V ≡ *do* {
(-,*s*,*m*) ← *WHILE* (λ(*V*,*s*,*m*). ¬*ls.isEmpty V*) (λ(*V*,*s*,*m*). *do* {
x←*RETURN* (*the* (*ls.sel V* (λ*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* (λ*x*. *True*)), i.e., selection of an element that satisfies the predicate λ*x*. *True*. As *ls.sel* returns an option datatype, we extract the value with *the*. Moreover, we omitted the loop invariant, as we don't need it any more.

Next, we have to show that our concrete pogram actually refines the abstract one.

theorem *sum-max-impl-refine*:
assumes $(V,V')\in$*build-rel ls.α 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.

using *assms* **unfolding** *sum-max-impl-def sum-max-def*
apply (*refine-rcg*) — Decompose combinators, generate data refinement goals

apply (*refine-dref-type*) — Type-based heuristics to instantiate data refinement goals
apply (*auto simp add*:
ls.correct refine-hsimp refine-rel-defs) — Discharge proof obligations
done

Refinement is transitive, so it is easy to show that the concrete program meets the specification.

theorem *sum-max-impl-correct*:
assumes $(V,V')\in$*build-rel ls.α ls.invar* **and** $V'\neq\{\}$
shows *sum-max-impl* $V \leq$ *SPEC* $(\lambda(s,m).\ s=\sum V' \wedge m=Max\ V')$
proof –
note *sum-max-impl-refine*
also note *sum-max-correct*
finally show *?thesis* **using** *assms* **.**
qed

Just for completeness, we also refine the total correct program in the same way.

definition *sum-max'-impl* :: *nat ls* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
sum-max'-impl $V \equiv$ *do* {
(-,*s*,*m*) $\leftarrow$ $WHILE_T$ $(\lambda(V,s,m).\ \neg ls.isEmpty\ V)$ $(\lambda(V,s,m).$ *do* {

[1] Also called coupling invariant.

```
    x←RETURN (the (ls.sel V (λx. True)));
    let V=ls.delete x V;
    let s=s+x;
    let m=max m x;
    RETURN (V,s,m)
  }) (V,0,0);
  RETURN (s,m)
}
```

theorem *sum-max′-impl-refine*:
  $(V, V')\in$*build-rel ls.α ls.invar* $\Longrightarrow$ *sum-max′-impl V* $\leq \Downarrow$*Id* (*sum-max′ V′*)
  unfolding *sum-max′-impl-def sum-max′-def*
  apply *refine-rcg*
  apply *refine-dref-type*
  apply (*auto simp*: *refine-hsimp ls.correct refine-rel-defs*)
  done

theorem *sum-max′-impl-correct*:
  assumes $(V, V')\in$*build-rel ls.α ls.invar* **and** $V'\neq\{\}$
  shows *sum-max′-impl V* $\leq$ *SPEC* ($\lambda(s,m)$. $s=\sum V' \wedge m=$*Max* V')
  using *ref-two-step*[*OF sum-max′-impl-refine sum-max′-correct*] *assms*

Note that we do not need the finiteness precondition, as list-sets are always finite. However, in order to exploit this, we have to unfold the *build-rel* construct, that relates the list-set on the concrete side to the set on the abstract side.

  apply (*auto simp*: *build-rel-def*)
  done

Code Generation

In order to generate code from the above definitions, we convert the function defined in our monad to an ordinary, deterministic function, for that the Isabelle/HOL code generator can generate code.

For partial correct algorithms, we can generate code inside a deterministic result monad. The domain of this monad is a flat complete lattice, where top means a failed assertion and bottom means nontermination. (Note that executing a function in this monad will never return bottom, but just diverge). The construct *nres-of x* embeds the deterministic into the nondeterministic monad.

Thus, we have to construct a function *?sum-max-code* such that:

schematic-goal *sum-max-code-aux*: *nres-of ?sum-max-code* $\leq$ *sum-max-impl V*

This is done automatically by the transfer procedure of our framework.

  unfolding *sum-max-impl-def*
  apply (*refine-transfer*)
  done

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 $sum\text{-}max\text{-}code\ V = dRETURN\ (s,m) \Longrightarrow s{=}\sum (ls.\alpha\ V) \wedge m{=}Max\ (ls.\alpha\ V)$
    and $sum\text{-}max\text{-}code\ V \neq dFAIL$

The proof is done by transitivity, and unfolding some definitions:

using *nres-correctD*[*OF order-trans*[*OF sum-max-code.refine sum-max-impl-correct*,
    of V ls.α V]] *assms*
  by (*auto simp*: *refine-rel-defs*)

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*:
  $RETURN\ ?sum\text{-}max'\text{-}code \leq sum\text{-}max'\text{-}impl\ V$
  unfolding *sum-max′-impl-def*
  apply (*refine-transfer*)
  done

concrete-definition *sum-max′-code* **for** *V* **uses** *sum-max′-code-aux*

theorem *sum-max′-code-correct*:
  $[\![ls.\alpha\ V \neq \{\}]\!] \Longrightarrow sum\text{-}max'\text{-}code\ V = (\sum (ls.\alpha\ V),\ Max\ (ls.\alpha\ V))$
  using *order-trans*[*OF sum-max′-code.refine sum-max′-impl-correct*,
    of V ls.α V]
  by (*auto simp*: *refine-rel-defs*)

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*
  unfolding *sum-max'-impl-def*
  apply (*refine-transfer the-resI*) — Using *the-resI* for internal monad and result extraction
  done

concrete-definition *sum-max''-code* **for** *V* **uses** *sum-max''-code-aux*

theorem *sum-max''-code-correct*:
  $[\![ls.\alpha\ V \neq \{\}]\!] \Longrightarrow$ *sum-max''-code* $V = (\sum (ls.\alpha\ V),\ Max\ (ls.\alpha\ V))$
  using *order-trans*[*OF sum-max''-code.refine sum-max'-impl-correct*,
    of V ls.α *V*]
  by (*auto simp*: *refine-rel-defs*)

Now, we can generate verified code with the Isabelle/HOL code generator:

export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *SML*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *OCaml?*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *Haskell?*
export-code *sum-max-code sum-max'-code sum-max''-code* **checking** *Scala*

Foreach-Loops

In the *sum-max* example above, we used a while-loop to iterate over the elements of a set. As this pattern is used commonly, there is an abbreviation for it in the refinement framework. The construct *FOREACH S f* σ_0 iterates $f::'x \Rightarrow 's \Rightarrow 's$ for each element in $S::'x\ set$, starting with state $\sigma_0::'s$.

With foreach-loops, we could have written our example as follows:

definition *sum-max-it* :: *nat set* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
  sum-max-it V $\equiv$ *FOREACH V* (λx (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

theorem *sum-max-it-correct*:
  assumes *PRE*: $V \neq \{\}$ **and** *FIN*: *finite V*
  shows *sum-max-it V* $\leq$ *SPEC* (λ(*s*,*m*). $s=\sum V \wedge m = Max\ V$)
  using *PRE* **unfolding** *sum-max-it-def*
  apply (*intro FOREACH-rule*[**where** $I=\lambda it\ \sigma$. *sum-max-invar V* (*it*,σ)] *refine-vcg*)
  apply (*rule FIN*) — Discharge finiteness of iterated set
  apply (*auto intro*: *sum-max-invar-step*) — Discharge step
  unfolding *sum-max-invar-def* — Unfold invariant definition
  apply (*auto*) — Discharge remaining goals
  done

definition *sum-max-it-impl* :: *nat ls* $\Rightarrow$ (*nat*$\times$*nat*) *nres* **where**
  sum-max-it-impl V $\equiv$ *FOREACH* (*ls.*α *V*) (λx (*s*,*m*). *RETURN* (*s*+*x*,*max m x*)) (*0*,*0*)

Note: The nondeterminism for iterators is currently resolved at transfer phase, where they are replaced by iterators from the ICF.

lemma *sum-max-it-impl-refine*:
  notes [*refine*] = *inj-on-id*
  assumes $(V,V')\in$*build-rel ls.*α *ls.invar*
  shows *sum-max-it-impl* $V \le \Downarrow$*Id* (*sum-max-it* V')
  unfolding *sum-max-it-impl-def sum-max-it-def*

Note that we specified *inj-on-id* as additional introduction rule. This is due to the very general iterator refinement rule, that may also change the set over that is iterated.

  using *assms*
  apply *refine-rcg* — This time, we don't need the *refine-dref-type* heuristics, as no schematic refinement relations are generated.
  apply (*auto simp*: *refine-hsimp refine-rel-defs*)
  done

schematic-goal *sum-max-it-code-aux*:
  RETURN ?sum-max-it-code $\le$ *sum-max-it-impl V*
  unfolding *sum-max-it-impl-def*
  apply (*refine-transfer*)
  done

Note that the transfer method has replaced the iterator by an iterator from the Isabelle Collection Framework.

thm *sum-max-it-code-aux*
concrete-definition *sum-max-it-code* **for** *V* **uses** *sum-max-it-code-aux*

theorem *sum-max-it-code-correct*:
  assumes *ls.*α $V \neq \{\}$
  shows *sum-max-it-code* $V = (\sum$ (*ls.*α V), *Max* (*ls.*α V))
proof −
  note *sum-max-it-code.refine*[*of V*]
  also note *sum-max-it-impl-refine*[*of V ls.*α *V*]
  also note *sum-max-it-correct*
  finally show *?thesis* **using** *assms* **by** (*auto simp*: *refine-rel-defs*)
qed

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-val ‹
  @{*code sum-max-it-list*} (*map* @{*code nat-of-integer*} [*1*,*2*,*3*,*4*,*5*])
›

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 (λx. x≤(2::nat)*(fst p + snd p)) }
  ≤ do { let (x,y)=p; z←SPEC (λz. z≤x+y);
         a←SPEC (λa. a≤x+y); ASSERT (x>2); RETURN (a+z)}
  apply (rule pw-leI)
  apply (auto simp add: refine-pw-simps split: prod.split)

  apply (rename-tac a b x)
  apply (case-tac x≤a+b)
  apply (rule-tac x=0 in exI)
  apply simp
  apply (rule-tac x=a+b in exI)
  apply (simp)
  apply (rule-tac x=x−(a+b) in exI)
  apply simp
  done
```

5.1.4 Arbitrary Recursion (TBD)

While-loops are suited to express tail-recursion. In order to express arbitrary recursion, the refinement framework provides the nrec-mode for the *partial-function* command, as well as the fixed point combinators *REC* (partial correctness) and REC_T (total correctness).

Examples for *partial-function* can be found in *ex/Refine-Fold*. Examples for the recursion combinators can be found in *ex/Recursion* and *ex/Nested-DFS*.

5.1.5 Reference

Statements

SUCCEED The empty set of results. Least element of the refinement ordering.

FAIL Result that indicates a failing assertion. Greatest element of the refinement ordering.

RES X All results from set *X*.

RETURN x Return single result *x*. Defined in terms of *RES*: *RETURN x* = *RES* $\{x\}$.

EMBED r Embed partial-correctness option type, i.e., succeed if *r*=*None*, otherwise return value of *r*.

SPEC Φ Specification. All results that satisfy predicate Φ. Defined in terms of *RES*: *SPEC* Φ = *SPEC* Φ

bind M f Binding. Nondeterministically choose a result from *M* and apply *f* to it. Note that usually the *do*-notation is used, i.e., *do* $\{x \leftarrow M; f\, x\}$ or *do* $\{M;f\}$ if the result of *M* is not important. If *M* fails, *bind M f* also fails.

ASSERT Φ Assertion. Fails if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSERT\ \Phi;\ f\}$.

ASSUME Φ Assumption. Succeeds if Φ does not hold, otherwise returns (). Note that the default usage with the do-notation is: *do* $\{ASSUME\ \Phi;\ f\}$.

REC body Recursion for partial correctness. May be used to express arbitrary recursion. Returns *SUCCEED* on nontermination.

REC_T *body* Recursion for total correctness. Returns *FAIL* on nontermination.

WHILE $b\ f\ \sigma_0$ 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.

$WHILE_T$ $b\ f\ \sigma_0$ 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 $I::'a\ set \Rightarrow 'b \Rightarrow bool$, where $I\ it\ \sigma$ means, that the invariant holds for the remaining set of elements *it* and current state σ.

FOREACH$_C$ $S\ c\ f\ \sigma_0$ Foreach-loop with explicit continuation condition.

Alternative syntax: *FOREACH*$_C$ $S\ c\ f\ \sigma_0$.

If $c::'\sigma \Rightarrow bool$ 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 nondeterminism monad.

Refinement

($\leq$) Refinement ordering. $S \leq S'$ means, that every result in S is also a result in S'. Moreover, S may only fail if S' fails. $\leq$ forms a complete lattice, with least element *SUCCEED* and greatest element *FAIL*.

$\Downarrow R$ Concretization. Takes a refinement relation $R::('c \times 'a)\ set$ that relates concrete to abstract values, and returns a concretization function $\Downarrow R$.

$\Uparrow R$ Abstraction. Takes a refinement relation and returns an abstraction function. The functions $\Downarrow R$ and $\Uparrow R$ form a Galois-connection, i.e., we have: $S \leq \Downarrow R\ S' \longleftrightarrow \Uparrow R\ S \leq S'$.

br $\alpha\ I$ 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 explicitely 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 ($\lambda(a,b)$ $(c,d). \ldots$) is done, and the monotonicity prover is used on monotonicity goals.

There is a standard setup for α=*RETURN* (transfer to plain function for total correct code generation), and α=*nres-of* (transfer to deterministic result monad, for partial correct code generation).

Automatic Refinement:

Method: *refine-autoref*

Attributes: ...

See automatic refinement package for documentation (TBD)

Concrete Definition:

Command: *concrete-definition name* [*attribs*] *for params uses thm* where *attribs* and the *for*-part are optional.

Declares a new constant from the left-hand side of a refinement lemma. Has special handling for left-hand sides of the forms *RETURN* - and *nres-of*, in which cases those topmost functions are not included in the defined constant.

The refinement lemma is folded with the new constant and registered as *name.refine*.

Command: *prepare-code-thms thms* takes a list of definitional theorems and sets up lemmas for the code generator for those definitions. This includes handling of recursion combinators.

Packages

The following parts of the refinement framework are not included by default, but can be imported if necessary:

Collection-Bindings: Sets up refinement rules for the Isabelle Collection Framework. With this theory loaded, the refinement condition generator will discharge most data refinements using the ICF automatically. Moreover, the transfer procedure will replace *FOREACH*-statements by the corresponding ICF-iterators.

end

5.2 Isabelle Collections Framework Userguide

5.2.1 Introduction

This is the Userguide for the (old) Isabelle Collection Framework. It does not cover the Generic Collection Framework, nor the Automatic Refinement Framework.

The Isabelle Collections Framework defines interfaces of various collection types and provides some standard implementations and generic algorithms.

The relation between the data structures of the collection framework and standard Isabelle types (e.g. for sets and maps) is established by abstraction functions.

Amongst others, the following interfaces and data-structures are provided by the Isabelle Collections Framework (For a complete list, see the overview section in the implementations chapter of the proof document):

- Set and map implementations based on (associative) lists, red-black trees, hashing and tries.

- An implementation of a FIFO-queue based on two stacks.
- Annotated lists implemented by finger trees.
- Priority queues implemented by binomial heaps, skew binomial heaps, and annotated lists (via finger trees).

The red-black trees are imported from the standard isabelle library. The binomial and skew binomial heaps are imported from the *Binomial-Heaps* entry of the archive of formal proofs. The finger trees are imported from the *Finger-Trees* entry of the archive of formal proofs.

Getting Started

To get started with the Isabelle Collections Framework (assuming that you are already familiar with Isabelle/HOL and Isar), you should first read the introduction (this section), that provides many basic examples. More examples are in the examples/ subdirectory of the collection framework. Section 5.2.2 explains the concepts of the Isabelle Collections Framework in more detail. Section 5.2.3 provides information on extending the framework along with detailed examples, and Section 5.2.4 contains a discussion on the design of this framework. There is also a paper [3] on the design of the Isabelle Collections Framework available.

Introductory Example

We introduce the Isabelle Collections Framework by a simple example.

Given a set of elements represented by a red-black tree, and a list, we want to filter out all elements that are not contained in the set. This can be done by Isabelle/HOL's *filter*-function[2]:

definition *rbt-restrict-list* :: $'a$::*linorder rs* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*
where *rbt-restrict-list s l* == [x←l. *rs.memb x s*]

The type $'a$ *rs* is the type of sets backed by red-black trees. Note that the element type of sets backed by red-black trees must be of sort *linorder*. The function *rs.memb* tests membership on such sets.

Next, we show correctness of our function:

lemma *rbt-restrict-list-correct*:
  assumes [*simp*]: *rs.invar s*
  shows *rbt-restrict-list s l* = [x←l. $x \in rs.\alpha\ s$]
  by (*simp add*: *rbt-restrict-list-def rs.memb-correct*)

The lemma *rs.memb-correct*:

[2]Note that Isabelle/HOL uses the list comprehension syntax [x←l. $P\ x$] as syntactic sugar for filtering a list.

$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$

states correctness of the *rs.memb*-function. The function $rs.\alpha$ maps a red-black-tree to the set that it represents. Moreover, we have to explicitely keep track of the invariants of the used data structure, in this case red-black trees. The premise *rs.invar ?s* represents the invariant assumption for the collection data structure. Red-black-trees are invariant-free, so this defaults to *True*. For uniformity reasons, these (unnecessary) invariant assumptions are present in all correctness lemmata.

Many of the correctness lemmas for standard RBT-set-operations are summarized by the lemma *rs.correct*:

$rs.\alpha\ (rs.empty\ ()) = \{\}$
$rs.invar\ (rs.empty\ ())$
$rs.\alpha\ (rs.sng\ x) = \{x\}$
$rs.invar\ (rs.sng\ x)$
$rs.invar\ s \Longrightarrow rs.memb\ x\ s = (x \in rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.ins\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.ins\ x\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.\alpha\ (rs.ins\text{-}dj\ x\ s) = insert\ x\ (rs.\alpha\ s)$
$[\![rs.invar\ s;\ x \notin rs.\alpha\ s]\!] \Longrightarrow rs.invar\ (rs.ins\text{-}dj\ x\ s)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.delete\ x\ s) = rs.\alpha\ s - \{x\}$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.delete\ x\ s)$
$rs.invar\ s \Longrightarrow rs.isEmpty\ s = (rs.\alpha\ s = \{\})$
$rs.invar\ s \Longrightarrow rs.isSng\ s = (\exists\, e.\ rs.\alpha\ s = \{e\})$
$rs.invar\ S \Longrightarrow rs.ball\ S\ P = (\forall\, x{\in}rs.\alpha\ S.\ P\ x)$
$rs.invar\ S \Longrightarrow rs.bex\ S\ P = (\exists\, x{\in}rs.\alpha\ S.\ P\ x)$
$rs.invar\ s \Longrightarrow rs.size\ s = card\ (rs.\alpha\ s)$
$rs.invar\ s \Longrightarrow rs.size\text{-}abort\ m\ s = min\ m\ (card\ (rs.\alpha\ s))$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.union\ s1\ s2) = rs.\alpha\ s1 \cup rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.union\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow rs.\alpha\ (rs.union\text{-}dj\ s1\ s2) = rs.\alpha\ s1 \cup rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}]\!]$
$\Longrightarrow rs.invar\ (rs.union\text{-}dj\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.diff\ s1\ s2) = rs.\alpha\ s1 - rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.diff\ s1\ s2)$
$rs.invar\ s \Longrightarrow rs.\alpha\ (rs.filter\ P\ s) = \{e \in rs.\alpha\ s.\ P\ e\}$
$rs.invar\ s \Longrightarrow rs.invar\ (rs.filter\ P\ s)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.\alpha\ (rs.inter\ s1\ s2) = rs.\alpha\ s1 \cap rs.\alpha\ s2$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.invar\ (rs.inter\ s1\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.subset\ s1\ s2 = (rs.\alpha\ s1 \subseteq rs.\alpha\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.equal\ s1\ s2 = (rs.\alpha\ s1 = rs.\alpha\ s2)$
$[\![rs.invar\ s1;\ rs.invar\ s2]\!] \Longrightarrow rs.disjoint\ s1\ s2 = (rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\})$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.disjoint\text{-}witness\ s1\ s2 = None]\!]$
$\Longrightarrow rs.\alpha\ s1 \cap rs.\alpha\ s2 = \{\}$
$[\![rs.invar\ s1;\ rs.invar\ s2;\ rs.disjoint\text{-}witness\ s1\ s2 = Some\ a]\!]$
$\Longrightarrow a \in rs.\alpha\ s1 \cap rs.\alpha\ s2$
$rs.invar\ s \Longrightarrow set\ (rs.to\text{-}list\ s) = rs.\alpha\ s$

rs.invar s $\Longrightarrow$ *distinct* (*rs.to-list s*)
rs.α (*rs.from-list l*) = *set l*
rs.invar (*rs.from-list l*)

All implementations provided by this library are compatible with the Isabelle/HOL code-generator. Now follow some examples of using the code-generator. Note that the code generator can only generate code for plain constants without arguments, while the operations like *rs.memb* have arguments, that are only hidden by an abbreviation.

There are conversion functions from lists to sets and, vice-versa, from sets to lists:

definition *conv-tests* ≡ (
  rs.from-list [*1*::*int* .. *10*],
  rs.to-list (*rs.from-list* [*1*::*int* .. *10*]),
  rs.to-sorted-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*]),
  rs.to-rev-list (*rs.from-list* [*1*::*int*,*5*,*6*,*7*,*3*,*4*,*9*,*8*,*2*,*7*,*6*])
)

ML-val ‹@{*code conv-tests*}›

Note that sets make no guarantee about ordering, hence the only thing we can prove about conversion from sets to lists is: *rs.to-list-correct*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-list s*)

Some sets, like red-black-trees, also support conversion to sorted lists, and we have: *rs.to-sorted-list-correct*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-sorted-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-sorted-list s*)
rs.invar s $\Longrightarrow$ *sorted* (*rs.to-sorted-list s*)

and *rs.to-rev-list-correct*:

rs.invar s $\Longrightarrow$ *set* (*rs.to-rev-list s*) = *rs.α s*
rs.invar s $\Longrightarrow$ *distinct* (*rs.to-rev-list s*)
rs.invar s $\Longrightarrow$ *sorted* (*rev* (*rs.to-rev-list s*))

definition *restrict-list-test* ≡ *rbt-restrict-list* (*rs.from-list* [*1*::*nat*,*2*,*3*,*4*,*5*]) [*1*::*nat*,*9*,*2*,*3*,*4*,*5*,*6*,*5*,*4*,*3*,*6*,*7*,*8*,*9*]

ML-val ‹@{*code restrict-list-test*}›

definition *big-test n* = (*rs.from-list* [(*1*::*int*)..*n*])

ML-val ‹@{*code big-test*} (@{*code int-of-integer*} *9000*)›

Theories

To make available the whole collections framework to your formalization, import the theory *Collections.Collections* which includes everything. Here is a small selection:

Collections.SetSpec Specification of sets and set functions

Collections.SetGA Generic algorithms for sets

Collections.SetStdImpl Standard set implementations (list, rb-tree, hashing, tries)

Collections.MapSpec Specification of maps

Collections.MapGA Generic algorithms for maps

Collections.MapStdImpl Standard map implementations (list,rb-tree, hashing, tries)

Collections.ListSpec Specification of lists

Collections.Fifo Amortized fifo queue

Collections.DatRef Data refinement for the while combinator

Iterators

An important concept when using collections are iterators. An iterator is a kind of generalized fold-functional. Like the fold-functional, it applies a function to all elements of a set and modifies a state. There are no guarantees about the iteration order. But, unlike the fold functional, you can prove useful properties of iterations even if the function is not left-commutative. Proofs about iterations are done in invariant style, establishing an invariant over the iteration.

The iterator combinator for red-black tree sets is *rs.iterate*, and the proof-rule that is usually used is: *rs.iteratei-rule-P*:

$$
\begin{array}{l}
[\![rs.invar\ S;\ I\ (rs.\alpha\ S)\ \sigma 0; \\
\bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma); \\
\bigwedge \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge \sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!] \\
\Longrightarrow P\ (rs.iteratei\ S\ c\ f\ \sigma 0)
\end{array}
$$

The invariant I is parameterized with the set of remaining elements that have not yet been iterated over and the current state. The invariant has to hold for all elements remaining and the initial state: $I\ (rs.\alpha\ S)\ \sigma 0$. Moreover, the invariant has to be preserved by an iteration step:

$\bigwedge x\ it\ \sigma.\ [\![x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma)$

And the proposition to be shown for the final state must be a consequence of the invarant for no elements remaining: $\bigwedge\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma$.

A generalization of iterators are *interruptible iterators* where iteration is only continues while some condition on the state holds. Reasoning over interruptible iterators is also done by invariants: *rs.iteratei-rule-P*:

$[\![rs.invar\ S;\ I\ (rs.\alpha\ S)\ \sigma 0;$
$\bigwedge x\ it\ \sigma.\ [\![c\ \sigma;\ x \in it;\ it \subseteq rs.\alpha\ S;\ I\ it\ \sigma]\!] \Longrightarrow I\ (it - \{x\})\ (f\ x\ \sigma);$
$\bigwedge\sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma;\ \bigwedge\sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma]\!]$
$\Longrightarrow P\ (rs.iteratei\ S\ c\ f\ \sigma 0)$

Here, interruption of the iteration is handled by the premise

$\bigwedge\sigma\ it.\ [\![it \subseteq rs.\alpha\ S;\ it \neq \{\};\ \neg\ c\ \sigma;\ I\ it\ \sigma]\!] \Longrightarrow P\ \sigma$

that shows the proposition from the invariant for any intermediate state of the iteration where the continuation condition does not hold (and thus the iteration is interrupted).

As an example of reasoning about results of iterators, we implement a function that converts a hashset to a list that contains precisely the elements of the set.

definition *hs-to-list′ s == hs.iteratei s* (λ-. *True*) (#) []

The correctness proof works by establishing the invariant that the list contains all elements that have already been iterated over. Again *hs.invar s* denotes the invariant for hashsets which defaults to *True*.

lemma *hs-to-list′-correct*:
  assumes *INV*: *hs.invar s*
  shows *set* (*hs-to-list′ s*) = *hs.α s*
  apply (*unfold hs-to-list′-def*)
  apply (*rule-tac*
    I=λ*it σ. set σ = hs.α s − it*
    in *hs.iterate-rule-P*[*OF INV*])

The resulting proof obligations are easily discharged using auto:

  apply *auto*
  done

As an example for an interruptible iterator, we define a bounded existential-quantification over the list elements. As soon as the first element is found that fulfills the predicate, the iteration is interrupted. The state of the iteration is simply a boolean, indicating the (current) result of the quantification:

definition *hs-bex s P == hs.iteratei s* (λσ. ¬ σ) (λ*x σ. P x*) *False*

lemma *hs-bex-correct*:
$hs.invar\ s \Longrightarrow hs\text{-}bex\ s\ P \longleftrightarrow (\exists\, x{\in}hs.\alpha\ s.\ P\ x)$
apply (*unfold hs-bex-def*)

The invariant states that the current result matches the result of the quantification over the elements already iterated over:

apply (*rule-tac*
$I{=}\lambda it\ \sigma.\ \sigma \longleftrightarrow (\exists\, x{\in}hs.\alpha\ s - it.\ P\ x)$
in *hs.iteratei-rule-P*)

The resulting proof obligations are easily discharged by auto:

apply *auto*
done

5.2.2 Structure of the Framework

The concepts of the framework are roughly based on the object-oriented concepts of interfaces, implementations and generic algorithms.

The concepts used in the framework are the following:

Interfaces An interface describes some concept by providing an abstraction mapping α to a related Isabelle/HOL-concept. The definition is generic in the datatype used to implement the concept (i.e. the concrete data structure). An interface is specified by means of a locale that fixes the abstraction mapping and an invariant. For example, the set-interface contains an abstraction mapping to sets, and is specified by the locale *SetSpec.set*. An interface roughly matches the concept of a (collection) interface in Java, e.g. *java.util.Set*.

Functions A function specifies some functionality involving interfaces. A function is specified by means of a locale. For example, membership query for a set is specified by the locale *SetSpec.set-memb* and equality test between two sets is a function specified by *SetSpec.set-equal*. A function roughly matches a method declared in an interface, e.g. *java.util.Set#contains, java.util.Set#equals*.

Operation Records In order to reference an interface with a standard set of operations, those operations are summarized in a record, and there is a locale that fixes this record, and makes available all operations. For example, the locale *SetSpec.StdSet* fixes a record of standard set operations and assumes their correctness. It also defines abbreviations to easily access the members of the record. Internally, all the standard operations, like *hs.memb*, are introduced by interpretation of such an operation locale.

Generic Algorithms A generic algorithm specifies, in a generic way, how to implement a function using other functions. Usually, a generic algorithm lives in a locale that imports the necessary operation locales. For example, the locale *cart-loc* defines a generic algorithm for the cartesian product between two sets.

There is no direct match of generic algorithms in the Java Collections Framework. The most related concept are abstract collection interfaces, that provide some default algorithms, e.g. *java.util.AbstractSet*. The concept of *Algorithm* in the C++ Standard Template Library [5] matches the concept of Generic Algorithm quite well.

Implementation An implementation of an interface provides a data structure for that interface together with an abstraction mapping and an invariant. Moreover, it provides implementations for some (or all) functions of that interface. For example, red-black trees are an implementation of the set-interface, with the abstraction mapping $rs.\alpha$ and invariant *rs.invar*; and the constant *rs.ins* implements the insert-function, as can be verified by *set-ins* $rs.\alpha$ *rs.invar rs.ins*. An implementation matches a concrete collection interface in Java, e.g. *java.util.TreeSet*, and the methods implemented by such an interface, e.g. *java.util.TreeSet#add*.

Instantiation An instantiation of a generic algorithm provides actual implementations for the used functions. For example, the generic cartesian-product algorithm can be instantiated to use red-black-trees for both arguments, and output a list, as will be illustrated below in Section 5.2.2. While some of the functions of an implementation need to be implemented specifically, many functions may be obtained by instantiating generic algorithms. In Java, instantiation of a generic algorithm is matched most closely by inheriting from an abstract collection interface. In the C++ Standard Template Library instantiation of generic algorithms is done implicitely by the compiler.

Instantiation of Generic Algorithms

A generic algorithm is instantiated by interpreting its locale with the wanted implementations. For example, to obtain a cartesian product between two red-black trees, yielding a list, we can do the following:

setup *Locale-Code.open-block*
interpretation *rrl*: *cart-loc rs-ops rs-ops ls-ops* **by** *unfold-locales*
setup *Locale-Code.close-block*
setup ‹*ICF-Tools.revert-abbrevs rrl*›

It is then available under the expected name:

term *rrl.cart*

Note the three lines of boilerplate code, that work around some technical problems of Isabelle/HOL: The *Locale-Code.open-block* and *Locale-Code.close-block* commands set up code generation for any locale that is interpreted in between them. They also have to be specified if an existing locale that already has interpretations is extended by new definitions.

The *ICF-Tools.revert-abbrevs rrl* reverts all abbreviations introduced by the locale, such that the displayed information becomes nicer.

Naming Conventions

The Isabelle Collections Framework follows these general naming conventions. Each implementation has a two-letter (or three-letter) and a one-letter (or two-letter) abbreviation, that are used as prefixes for the related constants, lemmas and instantiations.

The two-letter and three-letter abbreviations should be unique over all interfaces and instantiations, the one-letter abbreviations should be unique over all implementations of the same interface. Names that reference the implementation of only one interface are prefixed with that implementation's two-letter abbreviation (e.g. *hs.ins* for insertion into a HashSet (hs,h)), names that reference more than one implementation are prefixed with the one-letter (or two-letter) abbreviations (e.g. *rrl.cart* for the cartesian product between two RBT-Sets, yielding a list-set)

The most important abbreviations are:

lm,l List Map

lmi,li List Map with explicit invariant

rm,r RB-Tree Map

hm,h Hash Map

ahm,a Array-based hash map

tm,t Trie Map

ls,l List Set

lsi,li List Set with explicit invariant

rs,r RB-Tree Set

hs,h Hash Set

ahs,a Array-based hash map

ts,t Trie Set

Each function *name* of an interface *interface* is declared in a locale *interface-name*. This locale provides a fact *name-correct*. For example, there is the locale *set-ins* providing the fact *set-ins.ins-correct*. An implementation instantiates the locales of all implemented functions, using its two-letter abbreviation as instantiation prefix. For example, the HashSet-implementation instantiates the locale *set-ins* with the prefix *hs*, yielding the lemma *hs.ins-correct*. Moreover, an implementation with two-letter abbreviation *aa* provides a lemma *aa.correct* that summarizes the correctness facts for the basic operations. It should only contain those facts that are safe to be used with the simplifier. E.g., the correctness facts for basic operations on hash sets are available via the lemma *hs.correct*.

5.2.3 Extending the Framework

The best way to add new features, i.e., interfaces, functions, generic algorithms, or implementations to the collection framework is to use one of the existing items as example.

5.2.4 Design Issues

In this section, we motivate some of the design decisions of the Isabelle Collections Framework and report our experience with alternatives. Many of the design decisions are justified by restrictions of Isabelle/HOL and the code generator, so that there may be better options if those restrictions should vanish from future releases of Isabelle/HOL.

The main design goals of this development are:

1. Make available various implementations of collections under a unified interface.

2. It should be easy to extend the framework by new interfaces, functions, algorithms, and implementations.

3. Allow simple and concise reasoning over functions using collections.

4. Allow generic algorithms, that are independent of the actual data structure that is used.

5. Support generation of executable code.

6. Let the user precisely control what data structures are used in the implementation.

Data Refinement

In order to allow simple reasoning over collections, we use a data refinement approach. Each collection interface has an abstraction function that maps it on a related Isabelle/HOL concept (abstract level). The specification of functions are also relative to the abstraction. This allows most of the correctness reasoning to be done on the abstract level. On this level, the tool support is more elaborated and one is not yet fixed to a concrete implementation. In a next step, the abstract specification is refined to use an actual implementation (concrete level). The correctness properties proven on the abstract level usually transfer easily to the concrete level.

Moreover, the user has precise control how the refinement is done, i.e. what data structures are used. An alternative would be to do refinement completely automatic, as e.g. done in the code generator setup of the Theory *Executable-Set*. This has the advantage that it induces less writing overhead. The disadvantage is that the user looses a great amount of control over the refinement. For example, in *Executable-Set*, all sets have to be represented by lists, and there is no possibility to represent one set differently from another.

For a more detailed discussion of the data refinement issue, we refer to the monadic refinement framework, that is available in the AFP (http://isa-afp.org/entries/Refine_Monadic.shtml)

Operation Records

In order to allow convenient access to the most frequently used functions of an interface, we have grouped them together in a record, and defined a locale that only fixes this record. This greatly reduces the boilerplate required to define a new (generic) algorithm, as only the operation locale (instead of every single function) has to be included in the locale for the generic algorithm.

Note however, that parameters of locales are monomorphic inside the locale. Thus, we have to import an own instance for the locale for every element type of a set, or key/value type of a map. For iterators, where this problem was most annoying, we have installed a workaround that allows polymorphic iterators even inside locales.

Locales for Generic Algorithms

A generic algorithm is defined within a locale, that includes the required functions (or operation locales). If many instances of the same interface are required, prefixes are used to distinguish between them. This makes the code for a generic algorithm quite consise and readable.

However, there are some technical issues that one has to consider:

- When fixing parameters in the declaration of the locale, their types will be inferred independently of the definitions later done in the locale context. In order to get the correct types, one has to add explicit type constraints.

- The code generator has problems with generating code from definitions inside a locale. Currently, the *Locale-Code*-package provides a rather convenient workaround for that issue: It requires the user to enclose interpretations and definitions of new constants inside already interpreted locales within two special commands, that set up the code generator appropriately.

Explicit Invariants vs Typedef

The interfaces of this framework use explicit invariants. This provides a more general specification which allows some operations to be implemented more efficiently, cf. *lsi.ins-dj* in *Collections.ListSetImpl-Invar*.

Most implementations, however, hide the invariant in a typedef and setup the code generator appropriately. In that case, the invariant is just λ-. *True*, and removed automatically by the simplifier and classical reasoner. However, it still shows up in some premises and conclusions due to uniformity reasons.

Chapter 6

Conclusion

This work presented the Isabelle Collections Framework, an efficient and extensible collections framework for Isabelle/HOL. The framework features data-refinement techniques to refine algorithms to use concrete collection datastructures, and is compatible with the Isabelle/HOL code generator, such that efficient code can be generated for all supported target languages. Finally, we defined a data refinement framework for the while-combinator, and used it to specify a state-space exploration algorithm and stepwise refined the specification to an executable DFS-algorithm using a hashset to store the set of already known states.

Up to now, interfaces for sets and maps are specified and implemented using lists, red-black-trees, and hashing. Moreover, an amortized constant time fifo-queue (based on two stacks) has been implemented. However, the 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 quitely accept a false theorem and behave normal otherwise. The theorem prover itself is a bit more critical in this aspect. However, Isabelle/HOL is implemented in LCF-style, i.e. all the proofs are eventually checked by a small kernel of trusted code, containing rather simple operations. HOL is the logic that is most frequently used with Isabelle, and it is unlikely that it's axiomatization in Isabelle is inconsistent and no one has found and reported this inconsistency yet.

The next crucial point is the code generator of Isabelle. We derive executable code from our specifications. The code generator contains another (thin) layer of untrusted code. This layer has some known deficiencies[1] (as of Isabelle2009) in the sense that invalid code is generated. This code is then rejected by the target language's compiler or interpreter, but does not silently compute the wrong thing.

Moreover, assuming correctness of the code generator, the generated code is only guaranteed to be *partially* correct[2], i.e. there are no formal termination guarantees.

Furthermore, manual adaptations of the code generator setup are also part of the trusted code base. For array-based hash maps, the Isabelle Collections Framework provides an ML implementation for arrays with in-place updates that is unverified; for Haskell, we use the DiffArray implementation from the Haskell library. Other than this, the Isabelle Collections Framework does not add any adaptations other than those available in the Isabelle/HOL library, in particular Efficient_Nat.

6.2 Acknowledgement

We thank Tobias Nipkow for encouraging us to make the collections framework an independent development. Moreover, we thank Markus Müller-Olm for discussion about data-refinement. Finally, we thank the people on the Isabelle mailing list for quick and useful response to any Isabelle-related questions.

[1]For example, the Haskell code generator may generate variables starting with uppercase letters, while the Haskell-specification requires variables to start with lowercase letters. Moreover, the ML code generator does not know the ML value restriction, and may generate code that violates this restriction.

[2]A simple example is the always-diverging function $\mathsf{f_{div}} :: \mathsf{bool} = \mathsf{while}\ (\lambda x.\ \mathsf{True})\ \mathsf{id}\ \mathsf{True}$ that is definable in HOL. The lemma $\forall x.\ x = \mathsf{if}\ \mathsf{f_{div}}\ \mathsf{then}\ x\ \mathsf{else}\ x$ is provable in Isabelle and rewriting based on it could, theoretically, be inserted before the code generation process, resulting in code that always diverges

Bibliography

[1] Java: The collections framework. Available on: http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html.

[2] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/collections.shtml, Dec. 2009. Formal proof development.

[3] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[4] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[5] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, November 1995.