Verifying Imperative Programs using Auto2

Bohua Zhan

March 17, 2025

Abstract

This entry contains the application of auto2 to verifying functional and imperative programs. Algorithms and data structures that are verified include linked lists, binary search trees, red-black trees, interval trees, priority queue, quicksort, union-find, Dijkstra's algorithm, and a sweep-line algorithm for detecting rectangle intersection. The imperative verification is based on Imperative HOL and its separation logic framework. A major goal of this work is to set up automation in order to reduce the length of proof that the user needs to provide, both for verifying functional programs and for working with separation logic.

Contents

1 Introduction

This AFP entry contains the applications of auto2 to verifying functional and imperative programs. These examples are published in [9].

- Functional programs (in directory Functional): we verify several functional algorithms and data structures, including: linked lists, binary search trees, red-black trees, interval trees, priority queue, quicksort, union-find, Dijkstra's algorithm, and a sweep-line algorithm for detecting rectangle intersection.
- Imperative programs (in directory Imperative): we verify imperative versions of the above algorithms and data structures, using Isabelle's Imperative HOL framework [1]. We make use of separation logic, following the framework set up by Lammich and Reis [5]. The general outline of some of the examples also come from there.

2 Mapping

theory *Mapping-Str*
  imports *Auto2-HOL.Auto2-Main*
begin

Basic definitions of a mapping. Here, we enclose the mapping inside a structure, to make evaluation a first-order concept.

datatype ('a, 'b) *map* = *Map* 'a ⇒ 'b *option*

fun *meval* :: ('a, 'b) *map* ⇒ 'a ⇒ 'b *option* (‹-⟨-⟩› [90]) **where**
  (*Map f*) ⟨*h*⟩ = *f h*
⟨*ML*⟩

lemma *meval-ext*: ∀ *x*. *M*⟨*x*⟩ = *N*⟨*x*⟩ ⟹ *M* = *N*
  ⟨*proof*⟩
⟨*ML*⟩

definition *empty-map* :: ('a, 'b) *map* **where**
  empty-map = *Map* (λ*x*. *None*)
⟨*ML*⟩

definition *update-map* :: ('a, 'b) *map* ⇒ 'a ⇒ 'b ⇒ ('a ,'b) *map* (‹ - { - → - }› [89,90,90] 90) **where**
  M {*k* → *v*} = *Map* (λ*x*. *if x* = *k then Some v else M*⟨*x*⟩)
⟨*ML*⟩

definition *delete-map* :: 'a ⇒ ('a, 'b) *map* ⇒ ('a, 'b) *map* **where**
  delete-map k M = *Map* (λ*x*. *if x* = *k then None else M*⟨*x*⟩)
⟨*ML*⟩

2.1 Map from an AList

fun *map-of-alist* :: $('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *map* **where**
map-of-alist [] = *empty-map*
| *map-of-alist* $(x \# xs)$ = (*map-of-alist* *xs*) $\{$*fst* $x \rightarrow$ *snd* $x\}$
⟨*ML*⟩

definition *has-key-alist* :: $('a \times 'b)$ *list* $\Rightarrow$ $'a \Rightarrow$ *bool* **where** [*rewrite*]:
has-key-alist *xs* *a* $\longleftrightarrow$ ($\exists$ *p*$\in$*set* *xs*. *fst* *p* = *a*)

lemma *map-of-alist-nil* [*rewrite-back*]:
has-key-alist *ys* *x* $\longleftrightarrow$ (*map-of-alist* *ys*)⟨*x*⟩ $\neq$ *None*
⟨*proof*⟩
⟨*ML*⟩

lemma *map-of-alist-some* [*forward*]:
(*map-of-alist* *xs*)⟨*k*⟩ = *Some* *v* $\Longrightarrow$ (*k*, *v*) $\in$ *set* *xs*
⟨*proof*⟩

lemma *map-of-alist-nil'*:
x $\in$ *set* (*map* *fst* *ys*) $\longleftrightarrow$ (*map-of-alist* *ys*)⟨*x*⟩ $\neq$ *None*
⟨*proof*⟩
⟨*ML*⟩

2.2 Mapping defined by a set of key-value pairs

definition *unique-keys-set* :: $('a \times 'b)$ *set* $\Rightarrow$ *bool* **where** [*rewrite*]:
unique-keys-set *S* = ($\forall$ *i* *x* *y*. (*i*, *x*) $\in$ *S* $\longrightarrow$ (*i*, *y*) $\in$ *S* $\longrightarrow$ *x* = *y*)

lemma *unique-keys-setD* [*forward*]: *unique-keys-set* *S* $\Longrightarrow$ (*i*, *x*) $\in$ *S* $\Longrightarrow$ (*i*, *y*) $\in$ *S* $\Longrightarrow$ *x* = *y* ⟨*proof*⟩
⟨*ML*⟩

definition *map-of-aset* :: $('a \times 'b)$ *set* $\Rightarrow$ $('a, 'b)$ *map* **where**
map-of-aset *S* = *Map* (λ*a*. *if* $\exists$ *b*. (*a*, *b*) $\in$ *S* *then* *Some* (*THE* *b*. (*a*, *b*) $\in$ *S*) *else* *None*)
⟨*ML*⟩

lemma *map-of-asetI1* [*rewrite*]: *unique-keys-set* *S* $\Longrightarrow$ (*a*, *b*) $\in$ *S* $\Longrightarrow$ (*map-of-aset* *S*)⟨*a*⟩ = *Some* *b*
⟨*proof*⟩

lemma *map-of-asetI2* [*rewrite*]: $\forall$ *b*. (*a*, *b*) $\notin$ *S* $\Longrightarrow$ (*map-of-aset* *S*)⟨*a*⟩ = *None*
⟨*proof*⟩

lemma *map-of-asetD1* [*forward*]: (*map-of-aset* *S*)⟨*a*⟩ = *None* $\Longrightarrow$ $\forall$ *b*. (*a*, *b*) $\notin$ *S*
⟨*proof*⟩

lemma *map-of-asetD2* [*forward*]:
unique-keys-set *S* $\Longrightarrow$ (*map-of-aset* *S*)⟨*a*⟩ = *Some* *b* $\Longrightarrow$ (*a*, *b*) $\in$ *S* ⟨*proof*⟩

$\langle ML \rangle$

lemma *map-of-aset-insert* [*rewrite*]:
$unique\text{-}keys\text{-}set\ (S \cup \{(k, v)\}) \Longrightarrow map\text{-}of\text{-}aset\ (S \cup \{(k, v)\}) = (map\text{-}of\text{-}aset\ S)\ \{k \rightarrow v\}$
$\langle proof \rangle$

lemma *map-of-alist-to-aset* [*rewrite*]:
$unique\text{-}keys\text{-}set\ (set\ xs) \Longrightarrow map\text{-}of\text{-}aset\ (set\ xs) = map\text{-}of\text{-}alist\ xs$
$\langle proof \rangle$

lemma *map-of-aset-delete* [*rewrite*]:
$unique\text{-}keys\text{-}set\ S \Longrightarrow (k, v) \in S \Longrightarrow map\text{-}of\text{-}aset\ (S - \{(k, v)\}) = delete\text{-}map\ k\ (map\text{-}of\text{-}aset\ S)$
$\langle proof \rangle$

lemma *map-of-aset-update* [*rewrite*]:
$unique\text{-}keys\text{-}set\ S \Longrightarrow (k, v) \in S \Longrightarrow$
$map\text{-}of\text{-}aset\ (S - \{(k, v)\} \cup \{(k, v')\}) = (map\text{-}of\text{-}aset\ S)\ \{k \rightarrow v'\}$ $\langle proof \rangle$

lemma *map-of-alist-delete* [*rewrite*]:
$set\ xs' = set\ xs - \{x\} \Longrightarrow unique\text{-}keys\text{-}set\ (set\ xs) \Longrightarrow x \in set\ xs \Longrightarrow$
$map\text{-}of\text{-}alist\ xs' = delete\text{-}map\ (fst\ x)\ (map\text{-}of\text{-}alist\ xs)$
$\langle proof \rangle$

lemma *map-of-alist-insert* [*rewrite*]:
$set\ xs' = set\ xs \cup \{x\} \Longrightarrow unique\text{-}keys\text{-}set\ (set\ xs') \Longrightarrow$
$map\text{-}of\text{-}alist\ xs' = (map\text{-}of\text{-}alist\ xs)\ \{fst\ x \rightarrow snd\ x\}$
$\langle proof \rangle$

lemma *map-of-alist-update* [*rewrite*]:
$set\ xs' = set\ xs - \{(k, v)\} \cup \{(k, v')\} \Longrightarrow unique\text{-}keys\text{-}set\ (set\ xs) \Longrightarrow (k, v) \in set\ xs \Longrightarrow$
$map\text{-}of\text{-}alist\ xs' = (map\text{-}of\text{-}alist\ xs)\ \{k \rightarrow v'\}$
$\langle proof \rangle$

2.3 Set of keys of a mapping

definition $keys\text{-}of :: ('a, 'b)\ map \Rightarrow 'a\ set$ **where** [*rewrite*]:
$keys\text{-}of\ M = \{x.\ M\langle x \rangle \neq None\}$

lemma *keys-of-iff* [*rewrite-bidir*]: $x \in keys\text{-}of\ M \longleftrightarrow M\langle x \rangle \neq None$ $\langle proof \rangle$
$\langle ML \rangle$

lemma *keys-of-empty* [*rewrite*]: $keys\text{-}of\ empty\text{-}map = \{\}$ $\langle proof \rangle$

lemma *keys-of-delete* [*rewrite*]:
$keys\text{-}of\ (delete\text{-}map\ x\ M) = keys\text{-}of\ M - \{x\}$ $\langle proof \rangle$

2.4 Minimum of a mapping, relevant for heaps (priority queues)

definition *is-heap-min* :: $'a \Rightarrow ('a, 'b{::}linorder)$ *map* $\Rightarrow$ *bool* **where** [*rewrite*]:
is-heap-min $x\ M \longleftrightarrow x \in$ *keys-of* $M \wedge (\forall k{\in}$*keys-of* $M.$ *the* $(M\langle x\rangle) \leq$ *the* $(M\langle k\rangle))$

2.5 General construction and update of maps

fun *map-constr* :: $(nat \Rightarrow bool) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow (nat, 'a)$ *map* **where**
map-constr $S\ f\ 0 =$ *empty-map*
| *map-constr* $S\ f\ (Suc\ k) =$ (*let* $M =$ *map-constr* $S\ f\ k$ *in if* $S\ k$ *then* $M\ \{k \rightarrow f\ k\}$ *else* M)
⟨*ML*⟩

lemma *map-constr-eval* [*rewrite*]:
map-constr $S\ f\ n =$ *Map* ($\lambda i.$ *if* $i < n$ *then if* $S\ i$ *then Some* $(f\ i)$ *else None else None*)
⟨*proof*⟩

lemma *keys-of-map-constr* [*rewrite*]:
$i \in$ *keys-of* (*map-constr* $S\ f\ n$) $\longleftrightarrow (S\ i \wedge i < n)$ ⟨*proof*⟩

definition *map-update-all* :: $(nat \Rightarrow 'a) \Rightarrow (nat, 'a)$ *map* $\Rightarrow (nat, 'a)$ *map* **where** [*rewrite*]:
map-update-all $f\ M =$ *Map* ($\lambda i.$ *if* $i \in$ *keys-of* M *then Some* $(f\ i)$ *else* $M\langle i\rangle$)

fun *map-update-all-impl* :: $(nat \Rightarrow 'a) \Rightarrow (nat, 'a)$ *map* $\Rightarrow nat \Rightarrow (nat, 'a)$ *map* **where**
map-update-all-impl $f\ M\ 0 = M$
| *map-update-all-impl* $f\ M\ (Suc\ k) =$
(*let* $M' =$ *map-update-all-impl* $f\ M\ k$ *in if* $k \in$ *keys-of* M *then* $M'\ \{k \rightarrow f\ k\}$ *else* M')
⟨*ML*⟩

lemma *map-update-all-impl-ind* [*rewrite*]:
map-update-all-impl $f\ M\ n =$ *Map* ($\lambda i.$ *if* $i < n$ *then if* $i \in$ *keys-of* M *then Some* $(f\ i)$ *else None else* $M\langle i\rangle$)
⟨*proof*⟩

lemma *map-update-all-impl-correct* [*rewrite*]:
$\forall i{\in}$*keys-of* $M.\ i < n \Longrightarrow$ *map-update-all-impl* $f\ M\ n =$ *map-update-all* $f\ M$ ⟨*proof*⟩

lemma *keys-of-map-update-all* [*rewrite*]:
keys-of (*map-update-all* $f\ M$) = *keys-of* M ⟨*proof*⟩

end

3 Lists

theory *Lists-Ex*

imports *Mapping-Str*
begin

Examples on lists. The itrev example comes from [7, Section 2.4].

The development here of insertion and deletion on lists is essential for verifying functional binary search trees and red-black trees. The idea, following Nipkow [6], is that showing sorted-ness and preservation of multisets for trees should be done on the in-order traversal of the tree.

3.1 Linear time version of rev

fun *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
itrev [] *ys* = *ys*
| *itrev* (*x* # *xs*) *ys* = *itrev xs* (*x* # *ys*)
⟨*ML*⟩

lemma *itrev-eq-rev*: *itrev x* [] = *rev x*
⟨*proof*⟩

3.2 Strict sorted

fun *strict-sorted* :: *'a::linorder list* ⇒ *bool* **where**
strict-sorted [] = *True*
| *strict-sorted* (*x* # *ys*) = ((∀ *y*∈*set ys*. *x* < *y*) ∧ *strict-sorted ys*)
⟨*ML*⟩

lemma *strict-sorted-appendI* [*backward*]:
strict-sorted xs ∧ *strict-sorted ys* ∧ (∀ *x*∈*set xs*. ∀ *y*∈*set ys*. *x* < *y*) ⟹ *strict-sorted* (*xs* @ *ys*)
⟨*proof*⟩

lemma *strict-sorted-appendE1* [*forward*]:
strict-sorted (*xs* @ *ys*) ⟹ *strict-sorted xs* ∧ *strict-sorted ys*
⟨*proof*⟩

lemma *strict-sorted-appendE2* [*forward*]:
strict-sorted (*xs* @ *ys*) ⟹ *x* ∈ *set xs* ⟹ ∀ *y*∈*set ys*. *x* < *y*
⟨*proof*⟩

lemma *strict-sorted-distinct* [*forward*]: *strict-sorted l* ⟹ *distinct l*
⟨*proof*⟩

3.3 Ordered insert

fun *ordered-insert* :: *'a::ord* ⇒ *'a list* ⇒ *'a list* **where**
ordered-insert x [] = [*x*]
| *ordered-insert x* (*y* # *ys*) = (
if x = *y then* (*y* # *ys*)
else if x < *y then x* # (*y* # *ys*)

else y # ordered-insert x ys)
⟨*ML*⟩

lemma *ordered-insert-set* [*rewrite*]:
set (*ordered-insert x ys*) = {*x*} ∪ *set ys*
⟨*proof*⟩

lemma *ordered-insert-sorted* [*forward*]:
strict-sorted ys ⟹ *strict-sorted* (*ordered-insert x ys*)
⟨*proof*⟩

lemma *ordered-insert-binary* [*rewrite*]:
strict-sorted (*xs* @ *a* # *ys*) ⟹ *ordered-insert x* (*xs* @ *a* # *ys*) =
(*if x* < *a then ordered-insert x xs* @ *a* # *ys*
else if x > *a then xs* @ *a* # *ordered-insert x ys*
else xs @ *a* # *ys*)
⟨*proof*⟩

3.4 Deleting an element

fun *remove-elt-list* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
remove-elt-list x [] = []
| *remove-elt-list x* (*y* # *ys*) = (*if y* = *x then remove-elt-list x ys else y* # *remove-elt-list x ys*)
⟨*ML*⟩

lemma *remove-elt-list-set* [*rewrite*]:
set (*remove-elt-list x ys*) = *set ys* − {*x*}
⟨*proof*⟩

lemma *remove-elt-list-sorted* [*forward*]:
strict-sorted ys ⟹ *strict-sorted* (*remove-elt-list x ys*)
⟨*proof*⟩

lemma *remove-elt-idem* [*rewrite*]:
x ∉ *set ys* ⟹ *remove-elt-list x ys* = *ys*
⟨*proof*⟩

lemma *remove-elt-list-binary* [*rewrite*]:
strict-sorted (*xs* @ *a* # *ys*) ⟹ *remove-elt-list x* (*xs* @ *a* # *ys*) =
(*if x* < *a then remove-elt-list x xs* @ *a* # *ys*
else if x > *a then xs* @ *a* # *remove-elt-list x ys else xs* @ *ys*)
⟨*proof*⟩

3.5 Ordered insertion into list of pairs

fun *ordered-insert-pairs* :: *'a*::*ord* ⇒ *'b* ⇒ (*'a* × *'b*) *list* ⇒ (*'a* × *'b*) *list* **where**
ordered-insert-pairs x v [] = [(*x*, *v*)]
| *ordered-insert-pairs x v* (*y* # *ys*) = (
if x = *fst y then* ((*x*, *v*) # *ys*)

else if $x < $ *fst y then* (x, v) # $(y$ # *ys*)
else y # *ordered-insert-pairs x v ys*)
⟨*ML*⟩

lemma *ordered-insert-pairs-map* [*rewrite*]:
map-of-alist (*ordered-insert-pairs x v ys*) = *update-map* (*map-of-alist ys*) *x v*
⟨*proof*⟩

lemma *ordered-insert-pairs-set* [*rewrite*]:
set (*map fst* (*ordered-insert-pairs x v ys*)) = $\{x\} \cup$ *set* (*map fst ys*)
⟨*proof*⟩

lemma *ordered-insert-pairs-sorted* [*backward*]:
strict-sorted (*map fst ys*) $\Longrightarrow$ *strict-sorted* (*map fst* (*ordered-insert-pairs x v ys*))
⟨*proof*⟩

lemma *ordered-insert-pairs-binary* [*rewrite*]:
strict-sorted (*map fst* (*xs* @ *a* # *ys*)) $\Longrightarrow$ *ordered-insert-pairs x v* (*xs* @ *a* # *ys*) =
(*if* $x <$ *fst a then ordered-insert-pairs x v xs* @ *a* # *ys*
else if $x >$ *fst a then xs* @ *a* # *ordered-insert-pairs x v ys*
else xs @ (x, v) # *ys*)
⟨*proof*⟩

3.6 Deleting from a list of pairs

fun *remove-elt-pairs* :: $'a \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b)$ *list* **where**
remove-elt-pairs x [] = []
| *remove-elt-pairs x* (*y* # *ys*) = (*if fst y* = *x then ys else y* # *remove-elt-pairs x ys*)
⟨*ML*⟩

lemma *remove-elt-pairs-map* [*rewrite*]:
strict-sorted (*map fst ys*) $\Longrightarrow$ *map-of-alist* (*remove-elt-pairs x ys*) = *delete-map x* (*map-of-alist ys*)
⟨*proof*⟩

lemma *remove-elt-pairs-on-set* [*rewrite*]:
strict-sorted (*map fst ys*) $\Longrightarrow$ *set* (*map fst* (*remove-elt-pairs x ys*)) = *set* (*map fst ys*) $- \{x\}$
⟨*proof*⟩

lemma *remove-elt-pairs-sorted* [*backward*]:
strict-sorted (*map fst ys*) $\Longrightarrow$ *strict-sorted* (*map fst* (*remove-elt-pairs x ys*))
⟨*proof*⟩

lemma *remove-elt-pairs-idem* [*rewrite*]:
$x \notin$ *set* (*map fst ys*) $\Longrightarrow$ *remove-elt-pairs x ys* = *ys*
⟨*proof*⟩

lemma *remove-elt-pairs-binary* [*rewrite*]:
strict-sorted (*map fst* (*xs* @ *a* # *ys*)) ⟹ *remove-elt-pairs x* (*xs* @ *a* # *ys*) =
(*if x* < *fst a then remove-elt-pairs x xs* @ *a* # *ys*
else if x > *fst a then xs* @ *a* # *remove-elt-pairs x ys else xs* @ *ys*)
⟨*proof*⟩

3.7 Search in a list of pairs

lemma *map-of-alist-binary* [*rewrite*]:
strict-sorted (*map fst* (*xs* @ *a* # *ys*)) ⟹ (*map-of-alist* (*xs* @ *a* # *ys*))⟨*x*⟩ =
(*if x* < *fst a then* (*map-of-alist xs*)⟨*x*⟩
else if x > *fst a then* (*map-of-alist ys*)⟨*x*⟩ *else Some* (*snd a*))
⟨*proof*⟩

end

4 Binary search tree

theory *BST*
imports *Lists-Ex*
begin

Verification of functional programs on binary search trees. For basic technique, see comments in Lists_Ex.thy.

4.1 Definition and setup for trees

datatype (′*a*, ′*b*) *tree* =
Tip | *Node* (*lsub*: (′*a*, ′*b*) *tree*) (*key*: ′*a*) (*nval*: ′*b*) (*rsub*: (′*a*, ′*b*) *tree*)

⟨*ML*⟩

4.2 Inorder traversal, and set of elements of a tree

fun *in-traverse* :: (′*a*, ′*b*) *tree* ⇒ ′*a list* **where**
in-traverse Tip = []
| *in-traverse* (*Node l k v r*) = *in-traverse l* @ *k* # *in-traverse r*
⟨*ML*⟩

fun *tree-set* :: (′*a*, ′*b*) *tree* ⇒ ′*a set* **where**
tree-set Tip = {}
| *tree-set* (*Node l k v r*) = {*k*} ∪ *tree-set l* ∪ *tree-set r*
⟨*ML*⟩

fun *in-traverse-pairs* :: (′*a*, ′*b*) *tree* ⇒ (′*a* × ′*b*) *list* **where**
in-traverse-pairs Tip = []
| *in-traverse-pairs* (*Node l k v r*) = *in-traverse-pairs l* @ (*k*, *v*) # *in-traverse-pairs*
r
⟨*ML*⟩

lemma *in-traverse-fst* [*rewrite*]:
  map fst (*in-traverse-pairs t*) = *in-traverse t*
⟨*proof*⟩

definition *tree-map* :: ('*a*, '*b*) *tree* ⇒ ('*a*, '*b*) *map* **where**
  tree-map t = *map-of-alist* (*in-traverse-pairs t*)
⟨*ML*⟩

4.3 Sortedness on trees

fun *tree-sorted* :: ('*a*::*linorder*, '*b*) *tree* ⇒ *bool* **where**
  tree-sorted Tip = *True*
| *tree-sorted* (*Node l k v r*) = ((∀ *x*∈*tree-set l*. *x* < *k*) ∧ (∀ *x*∈*tree-set r*. *k* < *x*)
    ∧ *tree-sorted l* ∧ *tree-sorted r*)
⟨*ML*⟩

lemma *tree-sorted-lr* [*forward*]:
  tree-sorted (*Node l k v r*) ⟹ *tree-sorted l* ∧ *tree-sorted r* ⟨*proof*⟩

lemma *inorder-preserve-set* [*rewrite*]:
  tree-set t = *set* (*in-traverse t*)
⟨*proof*⟩

lemma *inorder-pairs-sorted* [*rewrite*]:
  tree-sorted t ⟷ *strict-sorted* (*map fst* (*in-traverse-pairs t*))
⟨*proof*⟩

Use definition in terms of in_traverse from now on.

⟨*ML*⟩

4.4 Rotation on trees

definition *rotateL* :: ('*a*, '*b*) *tree* ⇒ ('*a*, '*b*) *tree* **where** [*rewrite*]:
  rotateL t = (*if t* = *Tip then t else if rsub t* = *Tip then t else*
    (*let rt* = *rsub t in*
     Node (*Node* (*lsub t*) (*key t*) (*nval t*) (*lsub rt*)) (*key rt*) (*nval rt*) (*rsub rt*)))

definition *rotateR* :: ('*a*, '*b*) *tree* ⇒ ('*a*, '*b*) *tree* **where** [*rewrite*]:
  rotateR t = (*if t* = *Tip then t else if lsub t* = *Tip then t else*
    (*let lt* = *lsub t in*
     Node (*lsub lt*) (*key lt*) (*nval lt*) (*Node* (*rsub lt*) (*key t*) (*nval t*) (*rsub t*))))

lemma *rotateL-in-trav* [*rewrite*]: *in-traverse* (*rotateL t*) = *in-traverse t* ⟨*proof*⟩
lemma *rotateR-in-trav* [*rewrite*]: *in-traverse* (*rotateR t*) = *in-traverse t* ⟨*proof*⟩

lemma *rotateL-sorted* [*forward*]: *tree-sorted t* ⟹ *tree-sorted* (*rotateL t*) ⟨*proof*⟩
lemma *rotateR-sorted* [*forward*]: *tree-sorted t* ⟹ *tree-sorted* (*rotateR t*) ⟨*proof*⟩

4.5 Insertion on trees

fun *tree-insert* :: *'a::ord* ⇒ *'b* ⇒ (*'a*, *'b*) *tree* ⇒ (*'a*, *'b*) *tree* **where**
tree-insert x v Tip = *Node Tip x v Tip*
| *tree-insert x v* (*Node l y w r*) =
(*if x* = *y then Node l x v r*
else if x < *y then Node* (*tree-insert x v l*) *y w r*
else Node l y w (*tree-insert x v r*))
⟨*ML*⟩

lemma *insert-in-traverse-pairs* [*rewrite*]:
tree-sorted t ⟹ *in-traverse-pairs* (*tree-insert x v t*) = *ordered-insert-pairs x v* (*in-traverse-pairs t*)
⟨*proof*⟩

Correctness results for insertion.

theorem *insert-sorted* [*forward*]:
tree-sorted t ⟹ *tree-sorted* (*tree-insert x v t*) ⟨*proof*⟩

theorem *insert-on-map*:
tree-sorted t ⟹ *tree-map* (*tree-insert x v t*) = (*tree-map t*) {*x* → *v*} ⟨*proof*⟩

4.6 Deletion on trees

fun *del-min* :: (*'a*, *'b*) *tree* ⇒ (*'a* × *'b*) × (*'a*, *'b*) *tree* **where**
del-min Tip = *undefined*
| *del-min* (*Node lt x v rt*) =
(*if lt* = *Tip then* ((*x*, *v*), *rt*) *else*
(*fst* (*del-min lt*), *Node* (*snd* (*del-min lt*)) *x v rt*))
⟨*ML*⟩

lemma *delete-min-del-hd-pairs* [*rewrite*]:
t ≠ *Tip* ⟹ *fst* (*del-min t*) # *in-traverse-pairs* (*snd* (*del-min t*)) = *in-traverse-pairs* *t*
⟨*proof*⟩

fun *delete-elt-tree* :: (*'a*, *'b*) *tree* ⇒ (*'a*, *'b*) *tree* **where**
delete-elt-tree Tip = *undefined*
| *delete-elt-tree* (*Node lt x v rt*) =
(*if lt* = *Tip then rt else if rt* = *Tip then lt else*
Node lt (*fst* (*fst* (*del-min rt*))) (*snd* (*fst* (*del-min rt*))) (*snd* (*del-min rt*)))
⟨*ML*⟩

lemma *delete-elt-in-traverse-pairs* [*rewrite*]:
in-traverse-pairs (*delete-elt-tree* (*Node lt x v rt*)) = *in-traverse-pairs lt* @ *in-traverse-pairs* *rt* ⟨*proof*⟩

fun *tree-delete* :: *'a::ord* ⇒ (*'a*, *'b*) *tree* ⇒ (*'a*, *'b*) *tree* **where**
tree-delete x Tip = *Tip*
| *tree-delete x* (*Node l y w r*) =

(*if* $x = y$ *then delete-elt-tree* (*Node l y w r*)
else if $x < y$ *then Node* (*tree-delete x l*) *y w r*
else Node l y w (*tree-delete x r*))
⟨*ML*⟩

lemma *tree-delete-in-traverse-pairs* [*rewrite*]:
tree-sorted t ⟹ *in-traverse-pairs* (*tree-delete x t*) = *remove-elt-pairs x* (*in-traverse-pairs t*)
⟨*proof*⟩

Correctness results for deletion.

theorem *tree-delete-sorted* [*forward*]:
tree-sorted t ⟹ *tree-sorted* (*tree-delete x t*) ⟨*proof*⟩

theorem *tree-delete-map* [*rewrite*]:
tree-sorted t ⟹ *tree-map* (*tree-delete x t*) = *delete-map x* (*tree-map t*) ⟨*proof*⟩

4.7 Search on sorted trees

fun *tree-search* :: ('*a*::*ord*, '*b*) *tree* ⇒ '*a* ⇒ '*b option* **where**
tree-search Tip x = *None*
| *tree-search* (*Node l k v r*) *x* =
(*if* $x = k$ *then Some v*
else if $x < k$ *then tree-search l x*
else tree-search r x)
⟨*ML*⟩

Correctness of search.

theorem *tree-search-correct* [*rewrite*]:
tree-sorted t ⟹ *tree-search t x* = (*tree-map t*)⟨*x*⟩
⟨*proof*⟩

end

5 Partial equivalence relation

theory *Partial-Equiv-Rel*
imports *Auto2-HOL.Auto2-Main*
begin

Partial equivalence relations, following theory Lib/Partial_Equivalence_Relation in [3].

definition *part-equiv* :: ('*a* × '*a*) *set* ⇒ *bool* **where** [*rewrite*]:
part-equiv R ⟷ *sym R* ∧ *trans R*

lemma *part-equivI* [*forward*]: *sym R* ⟹ *trans R* ⟹ *part-equiv R* ⟨*proof*⟩
lemma *part-equivD1* [*forward*]: *part-equiv R* ⟹ *sym R* ⟨*proof*⟩
lemma *part-equivD2* [*forward*]: *part-equiv R* ⟹ *trans R* ⟨*proof*⟩
⟨*ML*⟩

5.1 Combining two elements in a partial equivalence relation

definition *per-union* :: $('a \times 'a)$ *set* $\Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'a)$ *set* **where** [*rewrite*]:
per-union $R\ a\ b = R \cup \{\ (x,y).\ (x,a){\in}R \wedge (b,y){\in}R\ \} \cup \{\ (x,y).\ (x,b){\in}R \wedge (a,y){\in}R\ \}$

lemma *per-union-memI1* [*backward*]:
$(x,\ y) \in R \Longrightarrow (x,\ y) \in$ *per-union* $R\ a\ b$ ⟨*proof*⟩
⟨*ML*⟩

lemma *per-union-memI2* [*backward*]:
$(x,\ a) \in R \Longrightarrow (b,\ y) \in R \Longrightarrow (x,\ y) \in$ *per-union* $R\ a\ b$ ⟨*proof*⟩

lemma *per-union-memI3* [*backward*]:
$(x,\ b) \in R \Longrightarrow (a,\ y) \in R \Longrightarrow (x,\ y) \in$ *per-union* $R\ a\ b$ ⟨*proof*⟩

lemma *per-union-memD*:
$(x,\ y) \in$ *per-union* $R\ a\ b \Longrightarrow (x,\ y) \in R \vee ((x,\ a) \in R \wedge (b,\ y) \in R) \vee ((x,\ b) \in R \wedge (a,\ y) \in R)$
⟨*proof*⟩
⟨*ML*⟩

lemma *per-union-is-trans* [*forward*]:
trans $R \Longrightarrow$ *trans* (*per-union* $R\ a\ b$) ⟨*proof*⟩

lemma *per-union-is-part-equiv* [*forward*]:
part-equiv $R \Longrightarrow$ *part-equiv* (*per-union* $R\ a\ b$) ⟨*proof*⟩

end

6 Union find

theory *Union-Find*
imports *Partial-Equiv-Rel*
begin

Development follows theory Union_Find in [5].

6.1 Representing a partial equivalence relation using rep_of array

function (*domintros*) *rep-of* **where**
rep-of $l\ i = ($*if* $l\ !\ i = i$ *then* i *else rep-of* $l\ (l\ !\ i))$ ⟨*proof*⟩

⟨*ML*⟩

definition *ufa-invar* :: *nat list* $\Rightarrow$ *bool* **where** [*rewrite*]:
ufa-invar $l = (\forall\ i{<}$*length* l. *rep-of-dom* $(l,\ i) \wedge l\ !\ i <$ *length* $l)$

lemma *ufa-invarD*:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of-dom* (*l*, *i*) ∧ *l* ! *i* < *length l* ⟨*proof*⟩
⟨*ML*⟩

lemma *rep-of-id* [*rewrite*]: *ufa-invar l* ⟹ *i* < *length l* ⟹ *l* ! *i* = *i* ⟹ *rep-of l i* = *i* ⟨*proof*⟩

lemma *rep-of-iff* [*rewrite*]:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of l i* = (*if l* ! *i* = *i then i else rep-of l* (*l* ! *i*)) ⟨*proof*⟩
⟨*ML*⟩

lemma *rep-of-min* [*rewrite*]:
ufa-invar l ⟹ *i* < *length l* ⟹ *l* ! (*rep-of l i*) = *rep-of l i*
⟨*proof*⟩

lemma *rep-of-induct*:
ufa-invar l ∧ *i* < *length l* ⟹
∀ *i*<*length l*. *l* ! *i* = *i* ⟶ *P l i* ⟹
∀ *i*<*length l*. *l* ! *i* ≠ *i* ⟶ *P l* (*l* ! *i*) ⟶ *P l i* ⟹ *P l i*
⟨*proof*⟩
⟨*ML*⟩

lemma *rep-of-bound* [*forward-arg1*]:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of l i* < *length l*
⟨*proof*⟩

lemma *rep-of-idem* [*rewrite*]:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of l* (*rep-of l i*) = *rep-of l i* ⟨*proof*⟩

lemma *rep-of-idx* [*rewrite*]:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of l* (*l* ! *i*) = *rep-of l i* ⟨*proof*⟩

definition *ufa-α* :: *nat list* ⇒ (*nat* × *nat*) *set* **where** [*rewrite*]:
ufa-α l = {(*x*, *y*). *x* < *length l* ∧ *y* < *length l* ∧ *rep-of l x* = *rep-of l y*}

lemma *ufa-α-memI* [*backward*, *forward-arg*]:
x < *length l* ⟹ *y* < *length l* ⟹ *rep-of l x* = *rep-of l y* ⟹ (*x*, *y*) ∈ *ufa-α l*
⟨*proof*⟩

lemma *ufa-α-memD* [*forward*]:
(*x*, *y*) ∈ *ufa-α l* ⟹ *x* < *length l* ∧ *y* < *length l* ∧ *rep-of l x* = *rep-of l y*
⟨*proof*⟩
⟨*ML*⟩

lemma *ufa-α-equiv* [*forward*]: *part-equiv* (*ufa-α l*) ⟨*proof*⟩

lemma *ufa-α-refl* [*rewrite*]: (*i*, *i*) ∈ *ufa-α l* ⟷ *i* < *length l* ⟨*proof*⟩

6.2 Operations on rep_of array

definition *uf-init-rel* :: *nat* ⇒ (*nat* × *nat*) *set* **where** [*rewrite*]:
  $uf\text{-}init\text{-}rel\ n = ufa\text{-}\alpha\ [0..{<}n]$

lemma *ufa-init-invar* [*resolve*]: $ufa\text{-}invar\ [0..{<}n]$ ⟨*proof*⟩

lemma *ufa-init-correct* [*rewrite*]:
  $(x, y) \in uf\text{-}init\text{-}rel\ n \longleftrightarrow (x = y \wedge x < n)$
⟨*proof*⟩

abbreviation *ufa-union* :: *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat list* **where**
  $ufa\text{-}union\ l\ x\ y \equiv l[rep\text{-}of\ l\ x := rep\text{-}of\ l\ y]$

lemma *ufa-union-invar* [*forward-arg*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow y < length\ l \Longrightarrow l' = ufa\text{-}union\ l\ x\ y \Longrightarrow ufa\text{-}invar\ l'$
⟨*proof*⟩

lemma *ufa-union-aux* [*rewrite*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow y < length\ l \Longrightarrow l' = ufa\text{-}union\ l\ x\ y \Longrightarrow$
  $i < length\ l' \Longrightarrow rep\text{-}of\ l'\ i = (if\ rep\text{-}of\ l\ i = rep\text{-}of\ l\ x\ then\ rep\text{-}of\ l\ y\ else\ rep\text{-}of\ l\ i)$
⟨*proof*⟩

Correctness of union operation.

theorem *ufa-union-correct* [*rewrite*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow y < length\ l \Longrightarrow l' = ufa\text{-}union\ l\ x\ y \Longrightarrow$
  $ufa\text{-}\alpha\ l' = per\text{-}union\ (ufa\text{-}\alpha\ l)\ x\ y$
⟨*proof*⟩

abbreviation *ufa-compress* :: *nat list* ⇒ *nat* ⇒ *nat list* **where**
  $ufa\text{-}compress\ l\ x \equiv l[x := rep\text{-}of\ l\ x]$

lemma *ufa-compress-invar* [*forward-arg*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow l' = ufa\text{-}compress\ l\ x \Longrightarrow ufa\text{-}invar\ l'$
⟨*proof*⟩

lemma *ufa-compress-aux* [*rewrite*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow l' = ufa\text{-}compress\ l\ x \Longrightarrow i < length\ l' \Longrightarrow$
  $rep\text{-}of\ l'\ i = rep\text{-}of\ l\ i$
⟨*proof*⟩

Correctness of compress operation.

theorem *ufa-compress-correct* [*rewrite*]:
  $ufa\text{-}invar\ l \Longrightarrow x < length\ l \Longrightarrow ufa\text{-}\alpha\ (ufa\text{-}compress\ l\ x) = ufa\text{-}\alpha\ l$ ⟨*proof*⟩

⟨*ML*⟩

end

7 Connectedness for a set of undirected edges.

theory *Connectivity*
  imports *Union-Find*
begin

A simple application of union-find for graph connectivity.

fun *is-path* :: *nat* ⇒ (*nat* × *nat*) *set* ⇒ *nat list* ⇒ *bool* **where**
  is-path n S [] = *False*
| *is-path n S* (*x* # *xs*) =
   (*if xs* = [] *then x* < *n else* ((*x*, *hd xs*) ∈ *S* ∨ (*hd xs*, *x*) ∈ *S*) ∧ *is-path n S xs*)
⟨*ML*⟩

definition *has-path* :: *nat* ⇒ (*nat* × *nat*) *set* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
  has-path n S i j ⟷ (∃ *p*. *is-path n S p* ∧ *hd p* = *i* ∧ *last p* = *j*)

lemma *is-path-nonempty* [*forward*]: *is-path n S p* ⟹ *p* ≠ [] ⟨*proof*⟩
lemma *nonempty-is-not-path* [*resolve*]: ¬*is-path n S* [] ⟨*proof*⟩

lemma *is-path-extend* [*forward*]:
  is-path n S p ⟹ *S* ⊆ *T* ⟹ *is-path n T p*
⟨*proof*⟩

lemma *has-path-extend* [*forward*]:
  has-path n S i j ⟹ *S* ⊆ *T* ⟹ *has-path n T i j* ⟨*proof*⟩

definition *joinable* :: *nat list* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:
  joinable p q ⟷ (*last p* = *hd q*)

definition *path-join* :: *nat list* ⇒ *nat list* ⇒ *nat list* **where** [*rewrite*]:
  path-join p q = *p* @ *tl q*
⟨*ML*⟩

lemma *path-join-hd* [*rewrite*]: *p* ≠ [] ⟹ *hd* (*path-join p q*) = *hd p* ⟨*proof*⟩

lemma *path-join-last* [*rewrite*]: *joinable p q* ⟹ *q* ≠ [] ⟹ *last* (*path-join p q*) = *last q*
⟨*proof*⟩

lemma *path-join-is-path* [*backward*]:
  joinable p q ⟹ *is-path n S p* ⟹ *is-path n S q* ⟹ *is-path n S* (*path-join p q*)
⟨*proof*⟩

lemma *has-path-trans* [*forward*]:
  has-path n S i j ⟹ *has-path n S j k* ⟹ *has-path n S i k*
⟨*proof*⟩

definition *is-valid-graph* :: *nat* ⇒ (*nat* × *nat*) *set* ⇒ *bool* **where** [*rewrite*]:
  is-valid-graph n S ⟷ (∀ *p*∈*S*. *fst p* < *n* ∧ *snd p* < *n*)

lemma *has-path-single1* [*backward1*]:
is-valid-graph n S $\Longrightarrow$ $(a,\ b) \in S$ $\Longrightarrow$ *has-path n S a b*
⟨*proof*⟩

lemma *has-path-single2* [*backward1*]:
is-valid-graph n S $\Longrightarrow$ $(a,\ b) \in S$ $\Longrightarrow$ *has-path n S b a*
⟨*proof*⟩

lemma *has-path-refl* [*backward2*]:
is-valid-graph n S $\Longrightarrow$ $a < n$ $\Longrightarrow$ *has-path n S a a*
⟨*proof*⟩

definition *connected-rel* :: *nat* ⇒ (*nat* × *nat*) *set* ⇒ (*nat* × *nat*) *set* **where**
connected-rel n S = {(*a*,*b*). *has-path n S a b*}

lemma *connected-rel-iff* [*rewrite*]:
(*a*, *b*) ∈ *connected-rel n S* ⟷ *has-path n S a b* ⟨*proof*⟩

lemma *connected-rel-trans* [*forward*]:
trans (*connected-rel n S*) ⟨*proof*⟩

lemma *connected-rel-refl* [*backward2*]:
is-valid-graph n S $\Longrightarrow$ $a < n$ $\Longrightarrow$ (*a*, *a*) ∈ *connected-rel n S* ⟨*proof*⟩

lemma *is-path-per-union* [*rewrite*]:
is-valid-graph n (*S* ∪ {(*a*, *b*)}) $\Longrightarrow$
has-path n (*S* ∪ {(*a*, *b*)}) *i j* ⟷ (*i*, *j*) ∈ *per-union* (*connected-rel n S*) *a b*
⟨*proof*⟩

lemma *connected-rel-union* [*rewrite*]:
is-valid-graph n (*S* ∪ {(*a*, *b*)}) $\Longrightarrow$
connected-rel n (*S* ∪ {(*a*, *b*)}) = *per-union* (*connected-rel n S*) *a b* ⟨*proof*⟩

lemma *connected-rel-init* [*rewrite*]:
connected-rel n {} = *uf-init-rel n*
⟨*proof*⟩

fun *connected-rel-ind* :: *nat* ⇒ (*nat* × *nat*) *list* ⇒ *nat* ⇒ (*nat* × *nat*) *set* **where**
connected-rel-ind n es 0 = *uf-init-rel n*
| *connected-rel-ind n es* (*Suc k*) =
(*let R* = *connected-rel-ind n es k*; *p* = *es* ! *k in*
per-union R (*fst p*) (*snd p*))
⟨*ML*⟩

lemma *connected-rel-ind-rule* [*rewrite*]:
is-valid-graph n (*set es*) $\Longrightarrow$ $k \le$ *length es* $\Longrightarrow$
connected-rel-ind n es k = *connected-rel n* (*set* (*take k es*))
⟨*proof*⟩

Correctness of the functional algorithm.

theorem *connected-rel-ind-compute* [*rewrite*]:
  is-valid-graph n (*set es*) $\Longrightarrow$
   connected-rel-ind n es (*length es*) = *connected-rel n* (*set es*) ⟨*proof*⟩

end

8 Arrays

theory *Arrays-Ex*
  imports *Auto2-HOL.Auto2-Main*
begin

Basic examples for arrays.

8.1 List swap

definition *list-swap* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a list* **where** [*rewrite*]:
  list-swap xs i j = *xs*[*i* := *xs* ! *j*, *j* := *xs* ! *i*]
⟨*ML*⟩

lemma *list-swap-eval*:
  i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$
   (*list-swap xs i j*) ! *k* = (*if k* = *i then xs* ! *j else if k* = *j then xs* ! *i else xs* ! *k*)
⟨*proof*⟩
⟨*ML*⟩

lemma *list-swap-eval-triv* [*rewrite*]:
  i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ (*list-swap xs i j*) ! *i* = *xs* ! *j*
  i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ (*list-swap xs i j*) ! *j* = *xs* ! *i* ⟨*proof*⟩

lemma *length-list-swap* [*rewrite-arg*]:
  length (*list-swap xs i j*) = *length xs* ⟨*proof*⟩

lemma *mset-list-swap* [*rewrite*]:
  i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *mset* (*list-swap xs i j*) = *mset xs* ⟨*proof*⟩

lemma *set-list-swap* [*rewrite*]:
  i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *set* (*list-swap xs i j*) = *set xs* ⟨*proof*⟩
⟨*ML*⟩

8.2 Reverse

lemma *rev-nth* [*rewrite*]:
  n < *length xs* $\Longrightarrow$ *rev xs* ! *n* = *xs* ! (*length xs* − *1* − *n*)
⟨*proof*⟩

fun *rev-swap* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a list* **where**

rev-swap xs i j = (*if i* < *j then rev-swap* (*list-swap xs i j*) (*i* + *1*) (*j* − *1*) *else xs*)
⟨*ML*⟩

lemma *rev-swap-length* [*rewrite-arg*]:
$j < length\ xs \Longrightarrow length\ (rev\text{-}swap\ xs\ i\ j) = length\ xs$
⟨*proof*⟩

lemma *rev-swap-eval* [*rewrite*]:
$j < length\ xs \Longrightarrow (rev\text{-}swap\ xs\ i\ j)\ !\ k =$
$(if\ k < i\ then\ xs\ !\ k\ else\ if\ k > j\ then\ xs\ !\ k\ else\ xs\ !\ (j - (k - i)))$
⟨*proof*⟩

lemma *rev-swap-is-rev* [*rewrite*]:
$length\ xs \geq 1 \Longrightarrow rev\text{-}swap\ xs\ 0\ (length\ xs - 1) = rev\ xs$ ⟨*proof*⟩

8.3 Copy one array to the beginning of another

fun *array-copy* :: *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *'a list* **where**
array-copy xs xs' 0 = *xs'*
| *array-copy xs xs'* (*Suc n*) = *list-update* (*array-copy xs xs' n*) *n* (*xs* ! *n*)
⟨*ML*⟩

lemma *array-copy-length* [*rewrite-arg*]:
$n \leq length\ xs \Longrightarrow n \leq length\ xs' \Longrightarrow length\ (array\text{-}copy\ xs\ xs'\ n) = length\ xs'$
⟨*proof*⟩

lemma *array-copy-ind* [*rewrite*]:
$n \leq length\ xs \Longrightarrow n \leq length\ xs' \Longrightarrow k < n \Longrightarrow (array\text{-}copy\ xs\ xs'\ n)\ !\ k = xs\ !\ k$
⟨*proof*⟩

lemma *array-copy-correct* [*rewrite*]:
$n \leq length\ xs \Longrightarrow n \leq length\ xs' \Longrightarrow take\ n\ (array\text{-}copy\ xs\ xs'\ n) = take\ n\ xs$
⟨*proof*⟩

8.4 Sublist

definition *sublist* :: *nat* ⇒ *nat* ⇒ *'a list* ⇒ *'a list* **where** [*rewrite*]:
sublist l r xs = *drop l* (*take r xs*)
⟨*ML*⟩

lemma *length-sublist* [*rewrite-arg*]:
$r \leq length\ xs \Longrightarrow length\ (sublist\ l\ r\ xs) = r - l$ ⟨*proof*⟩

lemma *nth-sublist* [*rewrite*]:
$r \leq length\ xs \Longrightarrow xs' = sublist\ l\ r\ xs \Longrightarrow i < length\ xs' \Longrightarrow xs'\ !\ i = xs\ !\ (i + l)$ ⟨*proof*⟩

lemma *sublist-nil* [*rewrite*]:

$r \leq length\ xs \Longrightarrow r \leq l \Longrightarrow sublist\ l\ r\ xs = []$ ⟨*proof*⟩

lemma *sublist-0* [*rewrite*]:
$sublist\ 0\ l\ xs = take\ l\ xs$ ⟨*proof*⟩

lemma *sublist-drop* [*rewrite*]:
$sublist\ l\ r\ (drop\ n\ xs) = sublist\ (l + n)\ (r + n)\ xs$ ⟨*proof*⟩

⟨*ML*⟩

lemma *sublist-single* [*rewrite*]:
$l + 1 \leq length\ xs \Longrightarrow sublist\ l\ (l + 1)\ xs = [xs\ !\ l]$
⟨*proof*⟩

lemma *sublist-append* [*rewrite*]:
$l \leq m \Longrightarrow m \leq r \Longrightarrow r \leq length\ xs \Longrightarrow sublist\ l\ m\ xs\ @\ sublist\ m\ r\ xs = sublist\ l\ r\ xs$
⟨*proof*⟩

lemma *sublist-Cons* [*rewrite*]:
$r \leq length\ xs \Longrightarrow l < r \Longrightarrow xs\ !\ l\ \#\ sublist\ (l + 1)\ r\ xs = sublist\ l\ r\ xs$
⟨*proof*⟩

lemma *sublist-equalityI*:
$i \leq j \Longrightarrow j \leq length\ xs \Longrightarrow length\ xs = length\ ys \Longrightarrow$
$\forall k.\ i \leq k \longrightarrow k < j \longrightarrow xs\ !\ k = ys\ !\ k \Longrightarrow sublist\ i\ j\ xs = sublist\ i\ j\ ys$ ⟨*proof*⟩
⟨*ML*⟩

lemma *set-sublist* [*resolve*]:
$j \leq length\ xs \Longrightarrow x \in set\ (sublist\ i\ j\ xs) \Longrightarrow \exists k.\ k \geq i \wedge k < j \wedge x = xs\ !\ k$
⟨*proof*⟩

lemma *list-take-sublist-drop-eq* [*rewrite*]:
$l \leq r \Longrightarrow r \leq length\ xs \Longrightarrow take\ l\ xs\ @\ sublist\ l\ r\ xs\ @\ drop\ r\ xs = xs$
⟨*proof*⟩

8.5 Updating a set of elements in an array

definition *list-update-set* :: $(nat \Rightarrow bool) \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where** [*rewrite*]:
$list\text{-}update\text{-}set\ S\ f\ xs = list\ (\lambda i.\ if\ S\ i\ then\ f\ i\ else\ xs\ !\ i)\ (length\ xs)$

lemma *list-update-set-length* [*rewrite-arg*]:
$length\ (list\text{-}update\text{-}set\ S\ f\ xs) = length\ xs$ ⟨*proof*⟩

lemma *list-update-set-nth* [*rewrite*]:
$xs' = list\text{-}update\text{-}set\ S\ f\ xs \Longrightarrow i < length\ xs' \Longrightarrow xs'\ !\ i = (if\ S\ i\ then\ f\ i\ else\ xs\ !\ i)$ ⟨*proof*⟩
⟨*ML*⟩

fun *list-update-set-impl* :: $(nat \Rightarrow bool) \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ list$
where
list-update-set-impl S f xs 0 = xs
| *list-update-set-impl S f xs (Suc k) =*
(let xs′ = list-update-set-impl S f xs k in
if S k then xs′ [k := f k] else xs′)
⟨*ML*⟩

lemma *list-update-set-impl-ind* [*rewrite*]:
n ≤ length xs ⟹ list-update-set-impl S f xs n =
list (λi. if i < n then if S i then f i else xs ! i else xs ! i) (length xs)
⟨*proof*⟩

lemma *list-update-set-impl-correct* [*rewrite*]:
list-update-set-impl S f xs (length xs) = list-update-set S f xs ⟨*proof*⟩

end

9 Dijkstra's algorithm for shortest paths

theory *Dijkstra*
imports *Mapping-Str Arrays-Ex*
begin

Verification of Dijkstra's algorithm: function part.

The algorithm is also verified by Nordhoff and Lammich in [8].

9.1 Graphs

datatype *graph = Graph nat list list*

fun *size* :: *graph ⇒ nat* **where**
size (Graph G) = length G

fun *weight* :: *graph ⇒ nat ⇒ nat ⇒ nat* **where**
weight (Graph G) m n = (G ! m) ! n

fun *valid-graph* :: *graph ⇒ bool* **where**
valid-graph (Graph G) ⟷ (∀ i<length G. length (G ! i) = length G)
⟨*ML*⟩

9.2 Paths on graphs

The set of vertices less than n.

definition *verts* :: *graph ⇒ nat set* **where**
verts G = {i. i < size G}

lemma *verts-mem* [*rewrite*]: $i \in verts\ G \longleftrightarrow i < size\ G$ ⟨*proof*⟩
lemma *card-verts* [*rewrite*]: $card\ (verts\ G) = size\ G$ ⟨*proof*⟩
lemma *finite-verts* [*forward*]: $finite\ (verts\ G)$ ⟨*proof*⟩

definition *is-path* :: *graph* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:
  $is\text{-}path\ G\ p \longleftrightarrow p \neq [] \wedge set\ p \subseteq verts\ G$

lemma *is-path-to-in-verts* [*forward*]: $is\text{-}path\ G\ p \Longrightarrow hd\ p \in verts\ G \wedge last\ p \in verts\ G$
⟨*proof*⟩

definition *joinable* :: *graph* ⇒ *nat list* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:
  $joinable\ G\ p\ q \longleftrightarrow (is\text{-}path\ G\ p \wedge is\text{-}path\ G\ q \wedge last\ p = hd\ q)$

definition *path-join* :: *graph* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* **where** [*rewrite*]:
  $path\text{-}join\ G\ p\ q = p\ @\ tl\ q$
⟨*ML*⟩

lemma *path-join-is-path*:
  $joinable\ G\ p\ q \Longrightarrow is\text{-}path\ G\ (path\text{-}join\ G\ p\ q)$
⟨*proof*⟩
⟨*ML*⟩

fun *path-weight* :: *graph* ⇒ *nat list* ⇒ *nat* **where**
  $path\text{-}weight\ G\ [] = 0$
| $path\text{-}weight\ G\ (x\ \#\ xs) = (if\ xs = []\ then\ 0\ else\ weight\ G\ x\ (hd\ xs) + path\text{-}weight\ G\ xs)$
⟨*ML*⟩

lemma *path-weight-singleton* [*rewrite*]: $path\text{-}weight\ G\ [x] = 0$ ⟨*proof*⟩
lemma *path-weight-doubleton* [*rewrite*]: $path\text{-}weight\ G\ [m, n] = weight\ G\ m\ n$
⟨*proof*⟩

lemma *path-weight-sum* [*rewrite*]:
  $joinable\ G\ p\ q \Longrightarrow path\text{-}weight\ G\ (path\text{-}join\ G\ p\ q) = path\text{-}weight\ G\ p + path\text{-}weight\ G\ q$
⟨*proof*⟩

fun *path-set* :: *graph* ⇒ *nat* ⇒ *nat* ⇒ *nat list set* **where**
  $path\text{-}set\ G\ m\ n = \{p.\ is\text{-}path\ G\ p \wedge hd\ p = m \wedge last\ p = n\}$

lemma *path-set-mem* [*rewrite*]:
  $p \in path\text{-}set\ G\ m\ n \longleftrightarrow is\text{-}path\ G\ p \wedge hd\ p = m \wedge last\ p = n$ ⟨*proof*⟩

lemma *path-join-set*: $joinable\ G\ p\ q \Longrightarrow path\text{-}join\ G\ p\ q \in path\text{-}set\ G\ (hd\ p)\ (last\ q)$
⟨*proof*⟩
⟨*ML*⟩

9.3 Shortest paths

definition *is-shortest-path* :: *graph* ⇒ *nat* ⇒ *nat* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:
is-shortest-path G m n p ⟷
(*p* ∈ *path-set G m n* ∧ (∀ *p*′∈*path-set G m n*. *path-weight G p*′ ≥ *path-weight G p*))

lemma *is-shortest-pathD1* [*forward*]:
is-shortest-path G m n p ⟹ *p* ∈ *path-set G m n* ⟨*proof*⟩

lemma *is-shortest-pathD2* [*forward*]:
is-shortest-path G m n p ⟹ *p*′ ∈ *path-set G m n* ⟹ *path-weight G p*′ ≥ *path-weight G p* ⟨*proof*⟩
⟨*ML*⟩

definition *has-dist* :: *graph* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
has-dist G m n ⟷ (∃ *p*. *is-shortest-path G m n p*)

lemma *has-distI* [*forward*]: *is-shortest-path G m n p* ⟹ *has-dist G m n* ⟨*proof*⟩
lemma *has-distD* [*resolve*]: *has-dist G m n* ⟹ ∃ *p*. *is-shortest-path G m n p* ⟨*proof*⟩
lemma *has-dist-to-in-verts* [*forward*]: *has-dist G u v* ⟹ *u* ∈ *verts G* ∧ *v* ∈ *verts G* ⟨*proof*⟩
⟨*ML*⟩

definition *dist* :: *graph* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where** [*rewrite*]:
dist G m n = *path-weight G* (*SOME p*. *is-shortest-path G m n p*)
⟨*ML*⟩

lemma *dist-eq* [*rewrite*]:
is-shortest-path G m n p ⟹ *dist G m n* = *path-weight G p* ⟨*proof*⟩

lemma *distD* [*forward*]:
has-dist G m n ⟹ *p* ∈ *path-set G m n* ⟹ *path-weight G p* ≥ *dist G m n* ⟨*proof*⟩
⟨*ML*⟩

lemma *shortest-init* [*resolve*]: *n* ∈ *verts G* ⟹ *is-shortest-path G n n* [*n*] ⟨*proof*⟩

9.4 Interior points

List of interior points

definition *int-pts* :: *nat list* ⇒ *nat set* **where** [*rewrite*]:
int-pts p = *set* (*butlast p*)

lemma *int-pts-singleton* [*rewrite*]: *int-pts* [*x*] = {} ⟨*proof*⟩
lemma *int-pts-doubleton* [*rewrite*]: *int-pts* [*x*, *y*] = {*x*} ⟨*proof*⟩

definition *path-set-on* :: *graph* ⇒ *nat* ⇒ *nat* ⇒ *nat set* ⇒ *nat list set* **where**

$path\text{-}set\text{-}on\ G\ m\ n\ V = \{p.\ p \in path\text{-}set\ G\ m\ n \wedge int\text{-}pts\ p \subseteq V\}$

lemma *path-set-on-mem* [*rewrite*]:
$p \in path\text{-}set\text{-}on\ G\ m\ n\ V \longleftrightarrow p \in path\text{-}set\ G\ m\ n \wedge int\text{-}pts\ p \subseteq V$ ⟨*proof*⟩

Version of shortest path on a set of points

definition *is-shortest-path-on* :: $graph \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat\ set \Rightarrow bool$
where [*rewrite*]:
$is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V \longleftrightarrow$
$(p \in path\text{-}set\text{-}on\ G\ m\ n\ V \wedge (\forall p' \in path\text{-}set\text{-}on\ G\ m\ n\ V.\ path\text{-}weight\ G\ p' \geq path\text{-}weight\ G\ p))$

lemma *is-shortest-path-onD1* [*forward*]:
$is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V \Longrightarrow p \in path\text{-}set\text{-}on\ G\ m\ n\ V$ ⟨*proof*⟩

lemma *is-shortest-path-onD2* [*forward*]:
$is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V \Longrightarrow p' \in path\text{-}set\text{-}on\ G\ m\ n\ V \Longrightarrow path\text{-}weight\ G\ p' \geq path\text{-}weight\ G\ p$ ⟨*proof*⟩
⟨*ML*⟩

definition *has-dist-on* :: $graph \Rightarrow nat \Rightarrow nat \Rightarrow nat\ set \Rightarrow bool$ **where** [*rewrite*]:
$has\text{-}dist\text{-}on\ G\ m\ n\ V \longleftrightarrow (\exists p.\ is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V)$

lemma *has-dist-onI* [*forward*]: $is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V \Longrightarrow has\text{-}dist\text{-}on\ G\ m\ n\ V$ ⟨*proof*⟩
lemma *has-dist-onD* [*resolve*]: $has\text{-}dist\text{-}on\ G\ m\ n\ V \Longrightarrow \exists p.\ is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V$ ⟨*proof*⟩
⟨*ML*⟩

definition *dist-on* :: $graph \Rightarrow nat \Rightarrow nat \Rightarrow nat\ set \Rightarrow nat$ **where** [*rewrite*]:
$dist\text{-}on\ G\ m\ n\ V = path\text{-}weight\ G\ (SOME\ p.\ is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V)$
⟨*ML*⟩

lemma *dist-on-eq* [*rewrite*]:
$is\text{-}shortest\text{-}path\text{-}on\ G\ m\ n\ p\ V \Longrightarrow dist\text{-}on\ G\ m\ n\ V = path\text{-}weight\ G\ p$ ⟨*proof*⟩

lemma *dist-onD* [*forward*]:
$has\text{-}dist\text{-}on\ G\ m\ n\ V \Longrightarrow p \in path\text{-}set\text{-}on\ G\ m\ n\ V \Longrightarrow path\text{-}weight\ G\ p \geq dist\text{-}on\ G\ m\ n\ V$ ⟨*proof*⟩
⟨*ML*⟩

9.5 Two splitting lemmas

lemma *path-split1* [*backward*]: $is\text{-}path\ G\ p \Longrightarrow hd\ p \in V \Longrightarrow last\ p \notin V \Longrightarrow$
$\exists p1\ p2.\ joinable\ G\ p1\ p2 \wedge p = path\text{-}join\ G\ p1\ p2 \wedge int\text{-}pts\ p1 \subseteq V \wedge hd\ p2 \notin V$
⟨*proof*⟩

lemma *path-split2* [*backward*]: $is\text{-}path\ G\ p \Longrightarrow hd\ p \neq last\ p \Longrightarrow$

$\exists\, q\ n.\ joinable\ G\ q\ [n,\ last\ p] \wedge p = path\text{-}join\ G\ q\ [n,\ last\ p]$
⟨*proof*⟩

9.6 Deriving has_dist and has_dist_on

definition *known-dists* :: *graph* ⇒ *nat set* ⇒ *bool* **where** [*rewrite*]:
$known\text{-}dists\ G\ V \longleftrightarrow (V \subseteq verts\ G \wedge 0 \in V \wedge$
$(\forall i{\in}verts\ G.\ has\text{-}dist\text{-}on\ G\ 0\ i\ V) \wedge$
$(\forall i{\in}V.\ has\text{-}dist\ G\ 0\ i \wedge dist\ G\ 0\ i = dist\text{-}on\ G\ 0\ i\ V))$

lemma *derive-dist* [*backward2*]:
$known\text{-}dists\ G\ V \Longrightarrow$
$m \in verts\ G - V \Longrightarrow$
$\forall i{\in}verts\ G - V.\ dist\text{-}on\ G\ 0\ i\ V \geq dist\text{-}on\ G\ 0\ m\ V \Longrightarrow$
$has\text{-}dist\ G\ 0\ m \wedge dist\ G\ 0\ m = dist\text{-}on\ G\ 0\ m\ V$
⟨*proof*⟩

lemma *join-def'* [*resolve*]: $joinable\ G\ p\ q \Longrightarrow path\text{-}join\ G\ p\ q = butlast\ p\ @\ q$
⟨*proof*⟩

lemma *int-pts-join* [*rewrite*]:
$joinable\ G\ p\ q \Longrightarrow int\text{-}pts\ (path\text{-}join\ G\ p\ q) = int\text{-}pts\ p \cup int\text{-}pts\ q$
⟨*proof*⟩

lemma *dist-on-triangle-ineq* [*backward*]:
$has\text{-}dist\text{-}on\ G\ k\ m\ V \Longrightarrow has\text{-}dist\text{-}on\ G\ k\ n\ V \Longrightarrow V \subseteq verts\ G \Longrightarrow n \in verts\ G \Longrightarrow m \in V \Longrightarrow$
$dist\text{-}on\ G\ k\ m\ V + weight\ G\ m\ n \geq dist\text{-}on\ G\ k\ n\ V$
⟨*proof*⟩

lemma *derive-dist-on* [*backward2*]:
$known\text{-}dists\ G\ V \Longrightarrow$
$m \in verts\ G - V \Longrightarrow$
$\forall i{\in}verts\ G - V.\ dist\text{-}on\ G\ 0\ i\ V \geq dist\text{-}on\ G\ 0\ m\ V \Longrightarrow$
$V' = V \cup \{m\} \Longrightarrow$
$n \in verts\ G - V' \Longrightarrow$
$has\text{-}dist\text{-}on\ G\ 0\ n\ V' \wedge dist\text{-}on\ G\ 0\ n\ V' = min\ (dist\text{-}on\ G\ 0\ n\ V)\ (dist\text{-}on\ G\ 0\ m\ V + weight\ G\ m\ n)$
⟨*proof*⟩

9.7 Invariant for the Dijkstra's algorithm

The state consists of an array maintaining the best estimates, and a heap containing estimates for the unknown vertices.

datatype *state* = *State* (*est*: *nat list*) (*heap*: (*nat*, *nat*) *map*)
⟨*ML*⟩

definition *unknown-set* :: *state* ⇒ *nat set* **where** [*rewrite*]:
$unknown\text{-}set\ S = keys\text{-}of\ (heap\ S)$

definition *known-set* :: *state* ⇒ *nat set* **where** [*rewrite*]:
$\mathit{known\text{-}set}\ S = \{..<\mathit{length}\ (\mathit{est}\ S)\} - \mathit{unknown\text{-}set}\ S$

Invariant: for every vertex, the estimate is at least the shortest distance. Furthermore, for the known vertices the estimate is exact.

definition *inv* :: *graph* ⇒ *state* ⇒ *bool* **where** [*rewrite*]:
$\mathit{inv}\ G\ S \longleftrightarrow (\mathit{let}\ V = \mathit{known\text{-}set}\ S;\ W = \mathit{unknown\text{-}set}\ S;\ M = \mathit{heap}\ S\ \mathit{in}$
$(\mathit{length}\ (\mathit{est}\ S) = \mathit{size}\ G \wedge \mathit{known\text{-}dists}\ G\ V \wedge$
$\mathit{keys\text{-}of}\ M \subseteq \mathit{verts}\ G \wedge$
$(\forall i{\in}W.\ M\langle i\rangle = \mathit{Some}\ (\mathit{est}\ S\ !\ i)) \wedge$
$(\forall i{\in}V.\ \mathit{est}\ S\ !\ i = \mathit{dist}\ G\ 0\ i) \wedge$
$(\forall i{\in}\mathit{verts}\ G.\ \mathit{est}\ S\ !\ i = \mathit{dist\text{-}on}\ G\ 0\ i\ V)))$

lemma *invE1* [*forward*]: $\mathit{inv}\ G\ S \Longrightarrow \mathit{length}\ (\mathit{est}\ S) = \mathit{size}\ G \wedge \mathit{known\text{-}dists}\ G\ (\mathit{known\text{-}set}\ S) \wedge \mathit{unknown\text{-}set}\ S \subseteq \mathit{verts}\ G$ ⟨*proof*⟩
lemma *invE2* [*forward*]: $\mathit{inv}\ G\ S \Longrightarrow i \in \mathit{known\text{-}set}\ S \Longrightarrow \mathit{est}\ S\ !\ i = \mathit{dist}\ G\ 0\ i$ ⟨*proof*⟩
lemma *invE3* [*forward*]: $\mathit{inv}\ G\ S \Longrightarrow i \in \mathit{verts}\ G \Longrightarrow \mathit{est}\ S\ !\ i = \mathit{dist\text{-}on}\ G\ 0\ i\ (\mathit{known\text{-}set}\ S)$ ⟨*proof*⟩
lemma *invE4* [*rewrite*]: $\mathit{inv}\ G\ S \Longrightarrow i \in \mathit{unknown\text{-}set}\ S \Longrightarrow (\mathit{heap}\ S)\langle i\rangle = \mathit{Some}\ (\mathit{est}\ S\ !\ i)$ ⟨*proof*⟩
⟨*ML*⟩

lemma *inv-unknown-set* [*rewrite*]:
$\mathit{inv}\ G\ S \Longrightarrow \mathit{unknown\text{-}set}\ S = \mathit{verts}\ G - \mathit{known\text{-}set}\ S$ ⟨*proof*⟩

lemma *dijkstra-end-inv* [*forward*]:
$\mathit{inv}\ G\ S \Longrightarrow \mathit{unknown\text{-}set}\ S = \{\} \Longrightarrow \forall i{\in}\mathit{verts}\ G.\ \mathit{has\text{-}dist}\ G\ 0\ i \wedge \mathit{est}\ S\ !\ i = \mathit{dist}\ G\ 0\ i$ ⟨*proof*⟩

9.8 Starting state

definition *dijkstra-start-state* :: *graph* ⇒ *state* **where** [*rewrite*]:
$\mathit{dijkstra\text{-}start\text{-}state}\ G =$
$\mathit{State}\ (\mathit{list}\ (\lambda i.\ \mathit{if}\ i = 0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{weight}\ G\ 0\ i)\ (\mathit{size}\ G))$
$(\mathit{map\text{-}constr}\ (\lambda i.\ i > 0)\ (\lambda i.\ \mathit{weight}\ G\ 0\ i)\ (\mathit{size}\ G))$
⟨*ML*⟩

lemma *dijkstra-start-known-set* [*rewrite*]:
$\mathit{size}\ G > 0 \Longrightarrow \mathit{known\text{-}set}\ (\mathit{dijkstra\text{-}start\text{-}state}\ G) = \{0\}$ ⟨*proof*⟩

lemma *dijkstra-start-unknown-set* [*rewrite*]:
$\mathit{size}\ G > 0 \Longrightarrow \mathit{unknown\text{-}set}\ (\mathit{dijkstra\text{-}start\text{-}state}\ G) = \mathit{verts}\ G - \{0\}$ ⟨*proof*⟩

lemma *card-start-state* [*rewrite*]:
$\mathit{size}\ G > 0 \Longrightarrow \mathit{card}\ (\mathit{unknown\text{-}set}\ (\mathit{dijkstra\text{-}start\text{-}state}\ G)) = \mathit{size}\ G - 1$
⟨*proof*⟩

Starting start of Dijkstra's algorithm satisfies the invariant.

theorem *dijkstra-start-inv* [*backward*]:
size G > *0* ⟹ *inv G* (*dijkstra-start-state G*)
⟨*proof*⟩

9.9 Step of Dijkstra's algorithm

fun *dijkstra-step* :: *graph* ⇒ *nat* ⇒ *state* ⇒ *state* **where**
dijkstra-step G m (*State e M*) =
(*let M′* = *delete-map m M*;
e′ = *list-update-set* (*λi. i* ∈ *keys-of M′*) (*λi. min* (*e* ! *m* + *weight G m i*) (*e* ! *i*)) *e*;
M″ = *map-update-all* (*λi. e′* ! *i*) *M′*
in State e′ M″)
⟨*ML*⟩

lemma *has-dist-on-larger* [*backward1*]:
has-dist G m n ⟹ *has-dist-on G m n V* ⟹ *dist-on G m n V* = *dist G m n* ⟹
has-dist-on G m n (*V* ∪ {*x*}) ∧ *dist-on G m n* (*V* ∪ {*x*}) = *dist G m n*
⟨*proof*⟩

lemma *dijkstra-step-unknown-set* [*rewrite*]:
inv G S ⟹ *m* ∈ *unknown-set S* ⟹ *unknown-set* (*dijkstra-step G m S*) = *unknown-set S* − {*m*} ⟨*proof*⟩

lemma *dijkstra-step-known-set* [*rewrite*]:
inv G S ⟹ *m* ∈ *unknown-set S* ⟹ *known-set* (*dijkstra-step G m S*) = *known-set S* ∪ {*m*} ⟨*proof*⟩

One step of Dijkstra's algorithm preserves the invariant.

theorem *dijkstra-step-preserves-inv* [*backward*]:
inv G S ⟹ *is-heap-min m* (*heap S*) ⟹ *inv G* (*dijkstra-step G m S*)
⟨*proof*⟩

definition *is-dijkstra-step* :: *graph* ⇒ *state* ⇒ *state* ⇒ *bool* **where** [*rewrite*]:
is-dijkstra-step G S S′ ⟷ (∃ *m. is-heap-min m* (*heap S*) ∧ *S′* = *dijkstra-step G m S*)

lemma *is-dijkstra-stepI* [*backward2*]:
is-heap-min m (*heap S*) ⟹ *dijkstra-step G m S* = *S′* ⟹ *is-dijkstra-step G S S′*
⟨*proof*⟩

lemma *is-dijkstra-stepD1* [*forward*]:
inv G S ⟹ *is-dijkstra-step G S S′* ⟹ *inv G S′* ⟨*proof*⟩

lemma *is-dijkstra-stepD2* [*forward*]:
inv G S ⟹ *is-dijkstra-step G S S′* ⟹ *card* (*unknown-set S′*) = *card* (*unknown-set S*) − *1* ⟨*proof*⟩
⟨*ML*⟩

end

10 Intervals

theory *Interval*
  imports *Auto2-HOL.Auto2-Main*
begin

Basic definition of intervals.

10.1 Definition of interval

datatype *'a interval* = *Interval* (*low*: *'a*) (*high*: *'a*)
⟨*ML*⟩

instantiation *interval* :: (*linorder*) *linorder* **begin**

definition *int-less*: $(a < b) = (low\ a < low\ b \mid (low\ a = low\ b \wedge high\ a < high\ b))$
definition *int-less-eq*: $(a \leq b) = (low\ a < low\ b \mid (low\ a = low\ b \wedge high\ a \leq high\ b))$

instance ⟨*proof*⟩ **end**

definition *is-interval* :: (*'a*::*linorder*) *interval* ⇒ *bool* **where** [*rewrite*]:
  $is\text{-}interval\ it \longleftrightarrow (low\ it \leq high\ it)$

10.2 Definition of interval with an index

datatype *'a idx-interval* = *IdxInterval* (*int*: *'a interval*) (*idx*: *nat*)
⟨*ML*⟩

instantiation *idx-interval* :: (*linorder*) *linorder* **begin**

definition *iint-less*: $(a < b) = (int\ a < int\ b \mid (int\ a = int\ b \wedge idx\ a < idx\ b))$
definition *iint-less-eq*: $(a \leq b) = (int\ a < int\ b \mid (int\ a = int\ b \wedge idx\ a \leq idx\ b))$

instance ⟨*proof*⟩ **end**

lemma *interval-less-to-le-low* [*forward*]:
  $(a::('a::linorder\ idx\text{-}interval)) < b \Longrightarrow low\ (int\ a) \leq low\ (int\ b)$
  ⟨*proof*⟩

10.3 Overlapping intervals

definition *is-overlap* :: (*'a*::*linorder*) *interval* ⇒ *'a interval* ⇒ *bool* **where** [*rewrite*]:
  $is\text{-}overlap\ x\ y \longleftrightarrow (high\ x \geq low\ y \wedge high\ y \geq low\ x)$

definition *has-overlap* :: ('*a*::*linorder*) *idx-interval set* ⇒ '*a interval* ⇒ *bool* **where**
[*rewrite*]:
has-overlap xs y ⟷ (∃ *x*∈*xs*. *is-overlap* (*int x*) *y*)

end

11 Interval tree

theory *Interval-Tree*
imports *Lists-Ex Interval*
begin

Functional version of interval tree. This is an augmented data structure on top of regular binary search trees (see BST.thy). See [2, Section 14.3] for a reference.

11.1 Definition of an interval tree

datatype *interval-tree* =
Tip
| *Node* (*lsub*: *interval-tree*) (*val*: *nat idx-interval*) (*tmax*: *nat*) (*rsub*: *interval-tree*)
where
tmax Tip = *0*

⟨*ML*⟩

11.2 Inorder traversal, and set of elements of a tree

fun *in-traverse* :: *interval-tree* ⇒ *nat idx-interval list* **where**
in-traverse Tip = []
| *in-traverse* (*Node l it m r*) = *in-traverse l* @ *it* # *in-traverse r*
⟨*ML*⟩

fun *tree-set* :: *interval-tree* ⇒ *nat idx-interval set* **where**
tree-set Tip = {}
| *tree-set* (*Node l it m r*) = {*it*} ∪ *tree-set l* ∪ *tree-set r*
⟨*ML*⟩

fun *tree-sorted* :: *interval-tree* ⇒ *bool* **where**
tree-sorted Tip = *True*
| *tree-sorted* (*Node l it m r*) = ((∀ *x*∈*tree-set l*. *x* < *it*) ∧ (∀ *x*∈*tree-set r*. *it* < *x*)
∧ *tree-sorted l* ∧ *tree-sorted r*)
⟨*ML*⟩

lemma *tree-sorted-lr* [*forward*]:
tree-sorted (*Node l it m r*) ⟹ *tree-sorted l* ∧ *tree-sorted r* ⟨*proof*⟩

lemma *tree-sortedD1* [*forward*]:
tree-sorted (*Node l it m r*) ⟹ *x* ∈ *tree-set l* ⟹ *x* < *it* ⟨*proof*⟩

lemma *tree-sortedD2* [*forward*]:
tree-sorted (*Node l it m r*) $\Longrightarrow$ *x* $\in$ *tree-set r* $\Longrightarrow$ *x* > *it* ⟨*proof*⟩

lemma *inorder-preserve-set* [*rewrite*]:
tree-set t = *set* (*in-traverse t*)
⟨*proof*⟩

lemma *inorder-sorted* [*rewrite*]:
tree-sorted t $\longleftrightarrow$ *strict-sorted* (*in-traverse t*)
⟨*proof*⟩

Use definition in terms of in_traverse from now on.

⟨*ML*⟩

11.3 Invariant on the maximum

definition *max3* :: *nat idx-interval* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where** [*rewrite*]:
max3 it b c = *max* (*high* (*int it*)) (*max b c*)

fun *tree-max-inv* :: *interval-tree* ⇒ *bool* **where**
tree-max-inv Tip = *True*
| *tree-max-inv* (*Node l it m r*) $\longleftrightarrow$ (*tree-max-inv l* ∧ *tree-max-inv r* ∧ *m* = *max3 it* (*tmax l*) (*tmax r*))
⟨*ML*⟩

lemma *tree-max-is-max* [*resolve*]:
tree-max-inv t $\Longrightarrow$ *it* ∈ *tree-set t* $\Longrightarrow$ *high* (*int it*) ≤ *tmax t*
⟨*proof*⟩

lemma *tmax-exists* [*backward*]:
tree-max-inv t $\Longrightarrow$ *t* ≠ *Tip* $\Longrightarrow$ ∃ *p*∈*tree-set t*. *high* (*int p*) = *tmax t*
⟨*proof*⟩

For insertion

lemma *max3-insert* [*rewrite*]: *max3 it 0 0* = *high* (*int it*) ⟨*proof*⟩

⟨*ML*⟩

11.4 Condition on the values

definition *tree-interval-inv* :: *interval-tree* ⇒ *bool* **where** [*rewrite*]:
tree-interval-inv t $\longleftrightarrow$ (∀ *p*∈*tree-set t*. *is-interval* (*int p*))

definition *is-interval-tree* :: *interval-tree* ⇒ *bool* **where** [*rewrite*]:
is-interval-tree t $\longleftrightarrow$ (*tree-sorted t* ∧ *tree-max-inv t* ∧ *tree-interval-inv t*)

lemma *is-interval-tree-lr* [*forward*]:
is-interval-tree (*Node l x m r*) $\Longrightarrow$ *is-interval-tree l* ∧ *is-interval-tree r* ⟨*proof*⟩

11.5 Insertion on trees

fun *insert* :: *nat idx-interval* ⇒ *interval-tree* ⇒ *interval-tree* **where**
  insert x Tip = *Node Tip x* (*high* (*int x*)) *Tip*
| *insert x* (*Node l y m r*) =
    (*if x* = *y then Node l y m r*
     else if x < *y then*
       let l' = *insert x l in*
           Node l' *y* (*max3 y* (*tmax* l') (*tmax r*)) *r*
     else
       let r' = *insert x r in*
           Node l y (*max3 y* (*tmax l*) (*tmax* r')) r')
⟨*ML*⟩

lemma *tree-insert-in-traverse* [*rewrite*]:
  tree-sorted t ⟹ *in-traverse* (*insert x t*) = *ordered-insert x* (*in-traverse t*)
⟨*proof*⟩

lemma *tree-insert-max-inv* [*forward*]:
  tree-max-inv t ⟹ *tree-max-inv* (*insert x t*)
⟨*proof*⟩

Correctness of insertion.

theorem *tree-insert-all-inv* [*forward*]:
  is-interval-tree t ⟹ *is-interval* (*int it*) ⟹ *is-interval-tree* (*insert it t*) ⟨*proof*⟩

theorem *tree-insert-on-set* [*rewrite*]:
  tree-sorted t ⟹ *tree-set* (*insert it t*) = {*it*} ∪ *tree-set t* ⟨*proof*⟩

11.6 Deletion on trees

fun *del-min* :: *interval-tree* ⇒ *nat idx-interval* × *interval-tree* **where**
  del-min Tip = *undefined*
| *del-min* (*Node lt v m rt*) =
   (*if lt* = *Tip then* (*v*, *rt*) *else*
    let lt' = *snd* (*del-min lt*) *in*
    (*fst* (*del-min lt*), *Node* lt' *v* (*max3 v* (*tmax* lt') (*tmax rt*)) *rt*))
⟨*ML*⟩

lemma *delete-min-del-hd*:
  t ≠ *Tip* ⟹ *fst* (*del-min t*) # *in-traverse* (*snd* (*del-min t*)) = *in-traverse t*
⟨*proof*⟩
⟨*ML*⟩

lemma *delete-min-max-inv* [*forward-arg*]:
  tree-max-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-max-inv* (*snd* (*del-min t*))
⟨*proof*⟩

lemma *delete-min-on-set*:
  t ≠ *Tip* ⟹ {*fst* (*del-min t*)} ∪ *tree-set* (*snd* (*del-min t*)) = *tree-set t* ⟨*proof*⟩

⟨*ML*⟩

lemma *delete-min-interval-inv* [*forward-arg*]:
tree-interval-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-interval-inv* (*snd* (*del-min t*)) ⟨*proof*⟩

lemma *delete-min-all-inv* [*forward-arg*]:
is-interval-tree t ⟹ *t* ≠ *Tip* ⟹ *is-interval-tree* (*snd* (*del-min t*)) ⟨*proof*⟩

fun *delete-elt-tree* :: *interval-tree* ⇒ *interval-tree* **where**
delete-elt-tree Tip = *undefined*
| *delete-elt-tree* (*Node lt x m rt*) =
(*if lt* = *Tip then rt else if rt* = *Tip then lt else*
let x′ = *fst* (*del-min rt*);
rt′ = *snd* (*del-min rt*);
m′ = *max3 x′* (*tmax lt*) (*tmax rt′*) *in*
Node lt (*fst* (*del-min rt*)) *m′ rt′*)
⟨*ML*⟩

lemma *delete-elt-in-traverse* [*rewrite*]:
in-traverse (*delete-elt-tree* (*Node lt x m rt*)) = *in-traverse lt* @ *in-traverse rt*
⟨*proof*⟩

lemma *delete-elt-max-inv* [*forward-arg*]:
tree-max-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-max-inv* (*delete-elt-tree t*) ⟨*proof*⟩

lemma *delete-elt-on-set* [*rewrite*]:
t ≠ *Tip* ⟹ *tree-set* (*delete-elt-tree* (*Node lt x m rt*)) = *tree-set lt* ∪ *tree-set rt*
⟨*proof*⟩

lemma *delete-elt-interval-inv* [*forward-arg*]:
tree-interval-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-interval-inv* (*delete-elt-tree t*) ⟨*proof*⟩

lemma *delete-elt-all-inv* [*forward-arg*]:
is-interval-tree t ⟹ *t* ≠ *Tip* ⟹ *is-interval-tree* (*delete-elt-tree t*) ⟨*proof*⟩

fun *delete* :: *nat idx-interval* ⇒ *interval-tree* ⇒ *interval-tree* **where**
delete x Tip = *Tip*
| *delete x* (*Node l y m r*) =
(*if x* = *y then delete-elt-tree* (*Node l y m r*)
else if x < *y then*
let l′ = *delete x l*;
m′ = *max3 y* (*tmax l′*) (*tmax r*) *in Node l′ y m′ r*
else
let r′ = *delete x r*;
m′ = *max3 y* (*tmax l*) (*tmax r′*) *in Node l y m′ r′*)
⟨*ML*⟩

lemma *tree-delete-in-traverse* [*rewrite*]:
tree-sorted t ⟹ *in-traverse* (*delete x t*) = *remove-elt-list x* (*in-traverse t*)

⟨*proof*⟩

lemma *tree-delete-max-inv* [*forward*]:
 tree-max-inv t ⟹ *tree-max-inv* (*delete x t*)
⟨*proof*⟩

Correctness of deletion.

theorem *tree-delete-all-inv* [*forward*]:
 is-interval-tree t ⟹ *is-interval-tree* (*delete x t*)
⟨*proof*⟩

theorem *tree-delete-on-set* [*rewrite*]:
 tree-sorted t ⟹ *tree-set* (*delete x t*) = *tree-set t* − {*x*} ⟨*proof*⟩

11.7 Search on interval trees

fun *search* :: *interval-tree* ⇒ *nat interval* ⇒ *bool* **where**
 search Tip x = *False*
| *search* (*Node l y m r*) *x* =
 (*if is-overlap* (*int y*) *x then True*
 else if l ≠ *Tip* ∧ *tmax l* ≥ *low x then search l x*
 else search r x)
⟨*ML*⟩

Correctness of search

theorem *search-correct* [*rewrite*]:
 is-interval-tree t ⟹ *is-interval x* ⟹ *search t x* ⟷ *has-overlap* (*tree-set t*) *x*
⟨*proof*⟩

end

12 Quicksort

theory *Quicksort*
 imports *Arrays-Ex*
begin

Functional version of quicksort.

Implementation of quicksort is largely based on theory Imperative_Quicksort in HOL/Imperative_HOL/ex in the Isabelle library.

12.1 Outer remains

definition *outer-remains* :: *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
 outer-remains xs xs' l r ⟷ (*length xs* = *length xs'* ∧ (∀ *i*. *i* < *l* ∨ *r* < *i* ⟶ *xs* ! *i* = *xs'* ! *i*))

lemma *outer-remains-length* [*forward*]:

outer-remains xs xs′ l r ⟹ *length xs* = *length xs′* ⟨*proof*⟩

lemma *outer-remains-eq* [*rewrite-back*]:
outer-remains xs xs′ l r ⟹ *i* < *l* ⟹ *xs* ! *i* = *xs′* ! *i*
outer-remains xs xs′ l r ⟹ *r* < *i* ⟹ *xs* ! *i* = *xs′* ! *i* ⟨*proof*⟩

lemma *outer-remains-sublist* [*backward2*]:
outer-remains xs xs′ l r ⟹ *i* < *l* ⟹ *take i xs* = *take i xs′*
outer-remains xs xs′ l r ⟹ *r* < *i* ⟹ *drop i xs* = *drop i xs′*
i ≤ *j* ⟹ *j* ≤ *length xs* ⟹ *outer-remains xs xs′ l r* ⟹ *j* ≤ *l* ⟹ *sublist i j xs* = *sublist i j xs′*
i ≤ *j* ⟹ *j* ≤ *length xs* ⟹ *outer-remains xs xs′ l r* ⟹ *i* > *r* ⟹ *sublist i j xs* = *sublist i j xs′* ⟨*proof*⟩
⟨*ML*⟩

12.2 part1 function

function *part1* :: (′*a*::*linorder*) *list* ⇒ *nat* ⇒ *nat* ⇒ ′*a* ⇒ (*nat* × ′*a list*) **where**
part1 xs l r a = (
if r ≤ *l then* (*r*, *xs*)
else if xs ! *l* ≤ *a then part1 xs* (*l* + *1*) *r a*
else part1 (*list-swap xs l r*) *l* (*r* − *1*) *a*)
⟨*proof*⟩
termination ⟨*proof*⟩
⟨*ML*⟩

lemma *part1-basic*:
r < *length xs* ⟹ *l* ≤ *r* ⟹ (*rs*, *xs′*) = *part1 xs l r a* ⟹
outer-remains xs xs′ l r ∧ *mset xs′* = *mset xs* ∧ *l* ≤ *rs* ∧ *rs* ≤ *r*
⟨*proof*⟩
⟨*ML*⟩

lemma *part1-partitions1* [*backward*]:
r < *length xs* ⟹ (*rs*, *xs′*) = *part1 xs l r a* ⟹ *l* ≤ *i* ⟹ *i* < *rs* ⟹ *xs′* ! *i* ≤ *a*
⟨*proof*⟩

lemma *part1-partitions2* [*backward*]:
r < *length xs* ⟹ (*rs*, *xs′*) = *part1 xs l r a* ⟹ *rs* < *i* ⟹ *i* ≤ *r* ⟹ *xs′* ! *i* ≥ *a*
⟨*proof*⟩

12.3 Paritition function

definition *partition* :: (′*a*::*linorder list*) ⇒ *nat* ⇒ *nat* ⇒ (*nat* × ′*a list*) **where** [*rewrite*]:
partition xs l r = (
let p = *xs* ! *r*;
(*m*, *xs′*) = *part1 xs l* (*r* − *1*) *p*;
m′ = *if xs′* ! *m* ≤ *p then m* + *1 else m*
in
(*m′*, *list-swap xs′ m′ r*))

⟨*ML*⟩

lemma *partition-basic*:
$l < r \Longrightarrow r < length\ xs \Longrightarrow (rs,\ xs') = partition\ xs\ l\ r \Longrightarrow$
$outer\text{-}remains\ xs\ xs'\ l\ r \wedge mset\ xs' = mset\ xs \wedge l \leq rs \wedge rs \leq r$ ⟨*proof*⟩
⟨*ML*⟩

lemma *partition-partitions1* [*forward*]:
$l < r \Longrightarrow r < length\ xs \Longrightarrow (rs,\ xs') = partition\ xs\ l\ r \Longrightarrow$
$x \in set\ (sublist\ l\ rs\ xs') \Longrightarrow x \leq xs'\ !\ rs$
⟨*proof*⟩

lemma *partition-partitions2* [*forward*]:
$l < r \Longrightarrow r < length\ xs \Longrightarrow (rs,\ xs'') = partition\ xs\ l\ r \Longrightarrow$
$x \in set\ (sublist\ (rs + 1)\ (r + 1)\ xs'') \Longrightarrow x \geq xs''\ !\ rs$
⟨*proof*⟩
⟨*ML*⟩

lemma *quicksort-term1*:
$\neg r \leq l \Longrightarrow \neg\ length\ xs \leq r \Longrightarrow x = partition\ xs\ l\ r \Longrightarrow (p,\ xs1) = x \Longrightarrow p - Suc\ l < r - l$
⟨*proof*⟩

lemma *quicksort-term2*:
$\neg r \leq l \Longrightarrow \neg\ length\ xs \leq r \Longrightarrow x = partition\ xs\ l\ r \Longrightarrow (p,\ xs2) = x \Longrightarrow r - Suc\ p < r - l$
⟨*proof*⟩

12.4 Quicksort function

function *quicksort* :: (*'a*::*linorder*) *list* ⇒ *nat* ⇒ *nat* ⇒ *'a list* **where**
quicksort xs l r = (
if $l \geq r$ *then xs*
else if $r \geq length\ xs$ *then xs*
else let
$(p,\ xs1) = partition\ xs\ l\ r$;
$xs2 = quicksort\ xs1\ l\ (p - 1)$
in
$quicksort\ xs2\ (p + 1)\ r$)
⟨*proof*⟩ **termination** ⟨*proof*⟩

lemma *quicksort-basic* [*rewrite-arg*]:
$mset\ (quicksort\ xs\ l\ r) = mset\ xs \wedge outer\text{-}remains\ xs\ (quicksort\ xs\ l\ r)\ l\ r$
⟨*proof*⟩

lemma *quicksort-trivial1* [*rewrite*]:
$l \geq r \Longrightarrow quicksort\ xs\ l\ r = xs$
⟨*proof*⟩

lemma *quicksort-trivial2* [*rewrite*]:
  $r \geq length\ xs \Longrightarrow quicksort\ xs\ l\ r = xs$
⟨*proof*⟩

lemma *quicksort-permutes* [*resolve*]:
  $xs' = quicksort\ xs\ l\ r \Longrightarrow set\ (sublist\ l\ (r + 1)\ xs') = set\ (sublist\ l\ (r + 1)\ xs)$
⟨*proof*⟩

lemma *quicksort-sorts* [*forward-arg*]:
  $r < length\ xs \Longrightarrow sorted\ (sublist\ l\ (r + 1)\ (quicksort\ xs\ l\ r))$
⟨*proof*⟩

Main result: correctness of functional quicksort.

theorem *quicksort-sorts-all* [*rewrite*]:
  $xs \neq [] \Longrightarrow quicksort\ xs\ 0\ (length\ xs - 1) = sort\ xs$
⟨*proof*⟩

end

13 Indexed priority queues

theory *Indexed-PQueue*
  imports *Arrays-Ex Mapping-Str*
begin

Verification of indexed priority queue: functional part. The data structure is also verified by Lammich in [4].

13.1 Successor functions, eq-pred predicate

fun *s1* :: $nat \Rightarrow nat$ **where** $s1\ m = 2 * m + 1$
fun *s2* :: $nat \Rightarrow nat$ **where** $s2\ m = 2 * m + 2$

lemma *s-inj* [*forward*]:
  $s1\ m = s1\ m' \Longrightarrow m = m'$ $s2\ m = s2\ m' \Longrightarrow m = m'$ ⟨*proof*⟩
lemma *s-neq* [*resolve*]:
  $s1\ m \neq s2\ m'$ $s1\ m > m$ $s2\ m > m$ $s2\ m > s1\ m$ ⟨*proof*⟩
⟨*ML*⟩

inductive *eq-pred* :: $nat \Rightarrow nat \Rightarrow bool$ **where**
  eq-pred n n
| $eq\text{-}pred\ n\ m \Longrightarrow eq\text{-}pred\ n\ (s1\ m)$
| $eq\text{-}pred\ n\ m \Longrightarrow eq\text{-}pred\ n\ (s2\ m)$
⟨*ML*⟩

lemma *eq-pred-parent1* [*forward*]:
  $eq\text{-}pred\ i\ (s1\ k) \Longrightarrow i \neq s1\ k \Longrightarrow eq\text{-}pred\ i\ k$
⟨*proof*⟩

lemma *eq-pred-parent2* [*forward*]:
eq-pred i (*s2 k*) $\Longrightarrow$ *i* $\neq$ *s2 k* $\Longrightarrow$ *eq-pred i k*
⟨*proof*⟩

lemma *eq-pred-cases*:
eq-pred i j $\Longrightarrow$ *eq-pred* (*s1 i*) *j* $\vee$ *eq-pred* (*s2 i*) *j* $\vee$ *j* = *i* $\vee$ *j* = *s1 i* $\vee$ *j* = *s2 i*
⟨*proof*⟩
⟨*ML*⟩

lemma *eq-pred-le* [*forward*]: *eq-pred i j* $\Longrightarrow$ *i* $\leq$ *j*
⟨*proof*⟩

13.2 Heap property

The corresponding tree is a heap

definition *is-heap* :: (*'a* $\times$ *'b*::*linorder*) *list* $\Rightarrow$ *bool* **where** [*rewrite*]:
is-heap xs = ($\forall$ *i j*. *eq-pred i j* $\longrightarrow$ *j* < *length xs* $\longrightarrow$ *snd* (*xs* ! *i*) $\leq$ *snd* (*xs* ! *j*))

lemma *is-heapD*:
is-heap xs $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *eq-pred i j* $\Longrightarrow$ *snd* (*xs* ! *i*) $\leq$ *snd* (*xs* ! *j*) ⟨*proof*⟩
⟨*ML*⟩

13.3 Bubble-down

The corresponding tree is a heap, except k is not necessarily smaller than its descendents.

definition *is-heap-partial1* :: (*'a* $\times$ *'b*::*linorder*) *list* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where** [*rewrite*]:
is-heap-partial1 xs k = ($\forall$ *i j*. *eq-pred i j* $\longrightarrow$ *i* $\neq$ *k* $\longrightarrow$ *j* < *length xs* $\longrightarrow$ *snd* (*xs* ! *i*) $\leq$ *snd* (*xs* ! *j*))

Two cases of switching with s1 k.

lemma *bubble-down1*:
s1 k < *length xs* $\Longrightarrow$ *is-heap-partial1 xs k* $\Longrightarrow$ *snd* (*xs* ! *k*) > *snd* (*xs* ! *s1 k*) $\Longrightarrow$
snd (*xs* ! *s1 k*) $\leq$ *snd* (*xs* ! *s2 k*) $\Longrightarrow$ *is-heap-partial1* (*list-swap xs k* (*s1 k*)) (*s1 k*) ⟨*proof*⟩
⟨*ML*⟩

lemma *bubble-down2*:
s1 k < *length xs* $\Longrightarrow$ *is-heap-partial1 xs k* $\Longrightarrow$ *snd* (*xs* ! *k*) > *snd* (*xs* ! *s1 k*) $\Longrightarrow$
s2 k $\geq$ *length xs* $\Longrightarrow$ *is-heap-partial1* (*list-swap xs k* (*s1 k*)) (*s1 k*) ⟨*proof*⟩
⟨*ML*⟩

One case of switching with s2 k.

lemma *bubble-down3*:
s2 k < *length xs* $\Longrightarrow$ *is-heap-partial1 xs k* $\Longrightarrow$ *snd* (*xs* ! *s1 k*) > *snd* (*xs* ! *s2 k*) $\Longrightarrow$

snd (*xs* ! *k*) > *snd* (*xs* ! *s2 k*) ⟹ *xs'* = *list-swap xs k* (*s2 k*) ⟹ *is-heap-partial1 xs'* (*s2 k*) ⟨*proof*⟩
⟨*ML*⟩

13.4 Bubble-up

fun *par* :: *nat* ⇒ *nat* **where**
par m = (*m* − *1*) *div 2*
⟨*ML*⟩

lemma *ps-inverse* [*rewrite*]: *par* (*s1 k*) = *k par* (*s2 k*) = *k* ⟨*proof*⟩

lemma *p-basic*: *m* ≠ *0* ⟹ *par m* < *m* ⟨*proof*⟩
⟨*ML*⟩

lemma *p-cases*: *m* ≠ *0* ⟹ *m* = *s1* (*par m*) ∨ *m* = *s2* (*par m*) ⟨*proof*⟩
⟨*ML*⟩

lemma *eq-pred-p-next*:
i ≠ *0* ⟹ *eq-pred i j* ⟹ *eq-pred* (*par i*) *j*
⟨*proof*⟩
⟨*ML*⟩

lemma *heap-implies-hd-min* [*resolve*]:
is-heap xs ⟹ *i* < *length xs* ⟹ *xs* ≠ [] ⟹ *snd* (*hd xs*) ≤ *snd* (*xs* ! *i*)
⟨*proof*⟩

The corresponding tree is a heap, except k is not necessarily greater than its ancestors.

definition *is-heap-partial2* :: (*'a* × *'b*::*linorder*) *list* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
is-heap-partial2 xs k = (∀ *i j*. *eq-pred i j* ⟶ *j* < *length xs* ⟶ *j* ≠ *k* ⟶ *snd* (*xs* ! *i*) ≤ *snd* (*xs* ! *j*))

lemma *bubble-up1* [*forward*]:
k < *length xs* ⟹ *is-heap-partial2 xs k* ⟹ *snd* (*xs* ! *k*) < *snd* (*xs* ! *par k*) ⟹ *k* ≠ *0* ⟹
is-heap-partial2 (*list-swap xs k* (*par k*)) (*par k*) ⟨*proof*⟩

lemma *bubble-up2* [*forward*]:
k < *length xs* ⟹ *is-heap-partial2 xs k* ⟹ *snd* (*xs* ! *k*) ≥ *snd* (*xs* ! *par k*) ⟹ *k* ≠ *0* ⟹
is-heap xs ⟨*proof*⟩
⟨*ML*⟩

13.5 Indexed priority queue

type-synonym *'a idx-pqueue* = (*nat* × *'a*) *list* × *nat option list*

fun *index-of-pqueue* :: *'a idx-pqueue* ⇒ *bool* **where**

index-of-pqueue $(xs, m) = ($
  $(\forall i<length\ xs.\ fst\ (xs\ !\ i) < length\ m \wedge m\ !\ (fst\ (xs\ !\ i)) = Some\ i) \wedge$
  $(\forall i.\ \forall k<length\ m.\ m\ !\ k = Some\ i \longrightarrow i < length\ xs \wedge fst\ (xs\ !\ i) = k))$
⟨*ML*⟩

lemma *index-of-pqueueD1*:
  $i < length\ xs \Longrightarrow$ *index-of-pqueue* $(xs, m) \Longrightarrow$
  $fst\ (xs\ !\ i) < length\ m \wedge m\ !\ (fst\ (xs\ !\ i)) = Some\ i$ ⟨*proof*⟩
⟨*ML*⟩

lemma *index-of-pqueueD2* [*forward*]:
  $k < length\ m \Longrightarrow$ *index-of-pqueue* $(xs, m) \Longrightarrow$
  $m\ !\ k = Some\ i \Longrightarrow i < length\ xs \wedge fst\ (xs\ !\ i) = k$ ⟨*proof*⟩

lemma *index-of-pqueueD3* [*forward*]:
  index-of-pqueue $(xs, m) \Longrightarrow p \in set\ xs \Longrightarrow fst\ p < length\ m$
⟨*proof*⟩
⟨*ML*⟩

lemma *has-index-unique-key* [*forward*]:
  index-of-pqueue $(xs, m) \Longrightarrow$ *unique-keys-set* $(set\ xs)$
⟨*proof*⟩

lemma *has-index-keys-of* [*rewrite*]:
  index-of-pqueue $(xs, m) \Longrightarrow$ *has-key-alist* $xs\ k \longleftrightarrow (k < length\ m \wedge m\ !\ k \neq None)$
⟨*proof*⟩

lemma *has-index-distinct* [*forward*]:
  index-of-pqueue $(xs, m) \Longrightarrow distinct\ xs$
⟨*proof*⟩

13.6 Basic operations on indexed_queue

fun *idx-pqueue-swap-fun* :: $(nat \times 'a)\ list \times nat\ option\ list \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times 'a)\ list \times nat\ option\ list$ **where**
  idx-pqueue-swap-fun $(xs, m)\ i\ j = ($
  list-swap $xs\ i\ j, ((m\ [fst\ (xs\ !\ i) := Some\ j])\ [fst\ (xs\ !\ j) := Some\ i]))$

lemma *index-of-pqueue-swap* [*forward-arg*]:
  $i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow$ *index-of-pqueue* $(xs, m) \Longrightarrow$
  index-of-pqueue (*idx-pqueue-swap-fun* $(xs, m)\ i\ j$)
⟨*proof*⟩

lemma *fst-idx-pqueue-swap* [*rewrite*]:
  fst (*idx-pqueue-swap-fun* $(xs, m)\ i\ j$) = *list-swap* $xs\ i\ j$
⟨*proof*⟩

lemma *snd-idx-pqueue-swap* [*rewrite*]:

length (*snd* (*idx-pqueue-swap-fun* (*xs*, *m*) *i j*)) = *length m*
⟨*proof*⟩

fun *idx-pqueue-push-fun* :: *nat* ⇒ *'a* ⇒ *'a idx-pqueue* ⇒ *'a idx-pqueue* **where**
idx-pqueue-push-fun k v (*xs*, *m*) = (*xs* @ [(*k*, *v*)], *list-update m k* (*Some* (*length xs*)))

lemma *idx-pqueue-push-correct* [*forward-arg*]:
index-of-pqueue (*xs*, *m*) ⟹ *k* < *length m* ⟹ ¬*has-key-alist xs k* ⟹
r = *idx-pqueue-push-fun k v* (*xs*, *m*) ⟹
index-of-pqueue r ∧ *fst r* = *xs* @ [(*k*, *v*)] ∧ *length* (*snd r*) = *length m*
⟨*proof*⟩

fun *idx-pqueue-pop-fun* :: *'a idx-pqueue* ⇒ *'a idx-pqueue* **where**
idx-pqueue-pop-fun (*xs*, *m*) = (*butlast xs*, *list-update m* (*fst* (*last xs*)) *None*)

lemma *idx-pqueue-pop-correct* [*forward-arg*]:
index-of-pqueue (*xs*, *m*) ⟹ *xs* ≠ [] ⟹ *r* = *idx-pqueue-pop-fun* (*xs*, *m*) ⟹
index-of-pqueue r ∧ *fst r* = *butlast xs* ∧ *length* (*snd r*) = *length m*
⟨*proof*⟩

13.7 Bubble up and down

function *idx-bubble-down-fun* :: *'a*::*linorder idx-pqueue* ⇒ *nat* ⇒ *'a idx-pqueue* **where**

```
  idx-bubble-down-fun (xs, m) k = (
    if s2 k < length xs then
      if snd (xs ! s1 k) ≤ snd (xs ! s2 k) then
        if snd (xs ! k) > snd (xs ! s1 k) then
          idx-bubble-down-fun (idx-pqueue-swap-fun (xs, m) k (s1 k)) (s1 k)
        else (xs, m)
      else
        if snd (xs ! k) > snd (xs ! s2 k) then
          idx-bubble-down-fun (idx-pqueue-swap-fun (xs, m) k (s2 k)) (s2 k)
        else (xs, m)
    else if s1 k < length xs then
      if snd (xs ! k) > snd (xs ! s1 k) then
        idx-bubble-down-fun (idx-pqueue-swap-fun (xs, m) k (s1 k)) (s1 k)
      else (xs, m)
    else (xs, m))
  ⟨proof⟩
  termination ⟨proof⟩
```

lemma *idx-bubble-down-fun-correct*:
r = *idx-bubble-down-fun x k* ⟹ *is-heap-partial1* (*fst x*) *k* ⟹
is-heap (*fst r*) ∧ *mset* (*fst r*) = *mset* (*fst x*) ∧ *length* (*snd r*) = *length* (*snd x*)
⟨*proof*⟩
⟨*ML*⟩

lemma *idx-bubble-down-fun-correct2* [*forward*]:
index-of-pqueue x $\Longrightarrow$ *index-of-pqueue* (*idx-bubble-down-fun x k*)
⟨*proof*⟩

fun *idx-bubble-up-fun* :: *'a*::*linorder idx-pqueue* ⇒ *nat* ⇒ *'a idx-pqueue* **where**
idx-bubble-up-fun (*xs*, *m*) *k* = (
if k = *0 then* (*xs*, *m*)
else if k < *length xs then*
if snd (*xs* ! *k*) < *snd* (*xs* ! *par k*) *then*
idx-bubble-up-fun (*idx-pqueue-swap-fun* (*xs*, *m*) *k* (*par k*)) (*par k*)
else (*xs*, *m*)
else (*xs*, *m*))

lemma *idx-bubble-up-fun-correct*:
r = *idx-bubble-up-fun x k* $\Longrightarrow$ *is-heap-partial2* (*fst x*) *k* $\Longrightarrow$
is-heap (*fst r*) ∧ *mset* (*fst r*) = *mset* (*fst x*) ∧ *length* (*snd r*) = *length* (*snd x*)
⟨*proof*⟩
⟨*ML*⟩

lemma *idx-bubble-up-fun-correct2* [*forward*]:
index-of-pqueue x $\Longrightarrow$ *index-of-pqueue* (*idx-bubble-up-fun x k*)
⟨*proof*⟩

13.8 Main operations

fun *delete-min-idx-pqueue-fun* :: *'a*::*linorder idx-pqueue* ⇒ (*nat* × *'a*) × *'a idx-pqueue*
where
delete-min-idx-pqueue-fun (*xs*, *m*) = (
let (*xs'*, *m'*) = *idx-pqueue-swap-fun* (*xs*, *m*) *0* (*length xs* − *1*);
a'' = *idx-pqueue-pop-fun* (*xs'*, *m'*)
in (*last xs'*, *idx-bubble-down-fun a'' 0*))

lemma *delete-min-idx-pqueue-correct*:
index-of-pqueue (*xs*, *m*) $\Longrightarrow$ *xs* ≠ [] $\Longrightarrow$ *res* = *delete-min-idx-pqueue-fun* (*xs*, *m*) $\Longrightarrow$
index-of-pqueue (*snd res*)
⟨*proof*⟩
⟨*ML*⟩

lemma *hd-last-swap-eval-last* [*rewrite*]:
xs ≠ [] $\Longrightarrow$ *last* (*list-swap xs 0* (*length xs* − *1*)) = *hd xs*
⟨*proof*⟩

Correctness of delete-min.

theorem *delete-min-idx-pqueue-correct2*:
is-heap xs $\Longrightarrow$ *xs* ≠ [] $\Longrightarrow$ *res* = *delete-min-idx-pqueue-fun* (*xs*, *m*) $\Longrightarrow$ *index-of-pqueue* (*xs*, *m*) $\Longrightarrow$
is-heap (*fst* (*snd res*)) ∧ *fst res* = *hd xs* ∧ *length* (*snd* (*snd res*)) = *length m* ∧
map-of-alist (*fst* (*snd res*)) = *delete-map* (*fst* (*fst res*)) (*map-of-alist xs*)

⟨*proof*⟩
⟨*ML*⟩

fun *insert-idx-pqueue-fun* :: *nat* ⇒ *'a*::*linorder* ⇒ *'a idx-pqueue* ⇒ *'a idx-pqueue*
where
insert-idx-pqueue-fun k v x = (
let x' = *idx-pqueue-push-fun k v x in*
idx-bubble-up-fun x' (*length* (*fst x'*) − *1*))

lemma *insert-idx-pqueue-correct* [*forward-arg*]:
index-of-pqueue (*xs*, *m*) ⟹ *k* < *length m* ⟹ ¬*has-key-alist xs k* ⟹
index-of-pqueue (*insert-idx-pqueue-fun k v* (*xs*, *m*))
⟨*proof*⟩

Correctness of insertion.

theorem *insert-idx-pqueue-correct2*:
index-of-pqueue (*xs*, *m*) ⟹ *is-heap xs* ⟹ *k* < *length m* ⟹ ¬*has-key-alist xs k*
⟹
r = *insert-idx-pqueue-fun k v* (*xs*, *m*) ⟹
is-heap (*fst r*) ∧ *length* (*snd r*) = *length m* ∧
map-of-alist (*fst r*) = *map-of-alist xs* { *k* → *v* }
⟨*proof*⟩
⟨*ML*⟩

fun *update-idx-pqueue-fun* :: *nat* ⇒ *'a*::*linorder* ⇒ *'a idx-pqueue* ⇒ *'a idx-pqueue*
where
update-idx-pqueue-fun k v (*xs*, *m*) = (
if m ! *k* = *None then*
insert-idx-pqueue-fun k v (*xs*, *m*)
else let
i = *the* (*m* ! *k*);
xs' = *list-update xs i* (*k*, *v*)
in
if snd (*xs* ! *i*) ≤ *v then idx-bubble-down-fun* (*xs'*, *m*) *i*
else idx-bubble-up-fun (*xs'*, *m*) *i*)

lemma *update-idx-pqueue-correct* [*forward-arg*]:
index-of-pqueue (*xs*, *m*) ⟹ *k* < *length m* ⟹
index-of-pqueue (*update-idx-pqueue-fun k v* (*xs*, *m*))
⟨*proof*⟩

Correctness of update.

theorem *update-idx-pqueue-correct2*:
index-of-pqueue (*xs*, *m*) ⟹ *is-heap xs* ⟹ *k* < *length m* ⟹
r = *update-idx-pqueue-fun k v* (*xs*, *m*) ⟹
is-heap (*fst r*) ∧ *length* (*snd r*) = *length m* ∧
map-of-alist (*fst r*) = *map-of-alist xs* { *k* → *v* }
⟨*proof*⟩
⟨*ML*⟩

end

14 Red-black trees

theory *RBTree*
  imports *Lists-Ex*
begin

Verification of functional red-black trees. For general technique, see Lists_Ex.thy.

14.1 Definition of RBT

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* =
    Leaf
  | *Node* (*lsub*: ('*a*, '*b*) *rbt*) (*cl*: *color*) (*key*: '*a*) (*val*: '*b*) (*rsub*: ('*a*, '*b*) *rbt*)
where
  cl Leaf = *B*

⟨*ML*⟩

lemma *not-R* [*forward*]: $c \neq R \Longrightarrow c = B$ ⟨*proof*⟩
lemma *not-B* [*forward*]: $c \neq B \Longrightarrow c = R$ ⟨*proof*⟩
lemma *red-not-leaf* [*forward*]: *cl t* = *R* ⟹ *t* ≠ *Leaf* ⟨*proof*⟩

14.2 RBT invariants

fun *black-depth* :: ('*a*, '*b*) *rbt* ⇒ *nat* **where**
  black-depth Leaf = *0*
| *black-depth* (*Node l R k v r*) = *black-depth l*
| *black-depth* (*Node l B k v r*) = *black-depth l* + *1*
⟨*ML*⟩

fun *cl-inv* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
  cl-inv Leaf = *True*
| *cl-inv* (*Node l R k v r*) = (*cl-inv l* ∧ *cl-inv r* ∧ *cl l* = *B* ∧ *cl r* = *B*)
| *cl-inv* (*Node l B k v r*) = (*cl-inv l* ∧ *cl-inv r*)
⟨*ML*⟩

fun *bd-inv* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
  bd-inv Leaf = *True*
| *bd-inv* (*Node l c k v r*) = (*bd-inv l* ∧ *bd-inv r* ∧ *black-depth l* = *black-depth r*)
⟨*ML*⟩

definition *is-rbt* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where** [*rewrite*]:
  is-rbt t = (*cl-inv t* ∧ *bd-inv t*)

lemma *cl-invI*: *cl-inv l* ⟹ *cl-inv r* ⟹ *cl-inv* (*Node l B k v r*) ⟨*proof*⟩

⟨*ML*⟩

lemma *bd-invI*: *bd-inv l* $\Longrightarrow$ *bd-inv r* $\Longrightarrow$ *black-depth l* = *black-depth r* $\Longrightarrow$ *bd-inv* (*Node l c k v r*) ⟨*proof*⟩
⟨*ML*⟩

lemma *is-rbt-rec* [*forward*]: *is-rbt* (*Node l c k v r*) $\Longrightarrow$ *is-rbt l* $\wedge$ *is-rbt r*
⟨*proof*⟩

14.3 Balancedness of RBT

lemma *two-distrib* [*rewrite*]: (*2*::*nat*) $*$ (*a* + *1*) = *2* $*$ *a* + *2* ⟨*proof*⟩

fun *min-depth* :: (′*a*, ′*b*) *rbt* $\Rightarrow$ *nat* **where**
min-depth Leaf = *0*
| *min-depth* (*Node l c k v r*) = *min* (*min-depth l*) (*min-depth r*) + *1*
⟨*ML*⟩

fun *max-depth* :: (′*a*, ′*b*) *rbt* $\Rightarrow$ *nat* **where**
max-depth Leaf = *0*
| *max-depth* (*Node l c k v r*) = *max* (*max-depth l*) (*max-depth r*) + *1*
⟨*ML*⟩

Balancedness of red-black trees.

theorem *rbt-balanced*: *is-rbt t* $\Longrightarrow$ *max-depth t* $\leq$ *2* $*$ *min-depth t* + *1*
⟨*proof*⟩

14.4 Definition and basic properties of cl_inv’

fun *cl-inv*′ :: (′*a*, ′*b*) *rbt* $\Rightarrow$ *bool* **where**
cl-inv′ *Leaf* = *True*
| *cl-inv*′ (*Node l c k v r*) = (*cl-inv l* $\wedge$ *cl-inv r*)
⟨*ML*⟩

lemma *cl-inv*′*B* [*forward*, *backward1*]:
cl-inv′ *t* $\Longrightarrow$ *cl t* = *B* $\Longrightarrow$ *cl-inv t*
⟨*proof*⟩

lemma *cl-inv*′*R* [*forward*]:
cl-inv′ (*Node l R k v r*) $\Longrightarrow$ *cl l* = *B* $\Longrightarrow$ *cl r* = *B* $\Longrightarrow$ *cl-inv* (*Node l R k v r*)
⟨*proof*⟩

lemma *cl-inv-to-cl-inv*′ [*forward*]: *cl-inv t* $\Longrightarrow$ *cl-inv*′ *t*
⟨*proof*⟩

lemma *cl-inv*′*I* [*forward-arg*]:
cl-inv l $\Longrightarrow$ *cl-inv r* $\Longrightarrow$ *cl-inv*′ (*Node l c k v r*) ⟨*proof*⟩

14.5 Set of keys, sortedness

fun *rbt-in-traverse* :: ('a, 'b) *rbt* ⇒ 'a *list* **where**
rbt-in-traverse Leaf = []
| *rbt-in-traverse* (*Node l c k v r*) = *rbt-in-traverse l* @ *k* # *rbt-in-traverse r*
⟨*ML*⟩

fun *rbt-set* :: ('a, 'b) *rbt* ⇒ 'a *set* **where**
rbt-set Leaf = {}
| *rbt-set* (*Node l c k v r*) = {*k*} ∪ *rbt-set l* ∪ *rbt-set r*
⟨*ML*⟩

fun *rbt-in-traverse-pairs* :: ('a, 'b) *rbt* ⇒ ('a × 'b) *list* **where**
rbt-in-traverse-pairs Leaf = []
| *rbt-in-traverse-pairs* (*Node l c k v r*) = *rbt-in-traverse-pairs l* @ (*k*, *v*) # *rbt-in-traverse-pairs r*
⟨*ML*⟩

lemma *rbt-in-traverse-fst* [*rewrite*]: *map fst* (*rbt-in-traverse-pairs t*) = *rbt-in-traverse t*
⟨*proof*⟩

definition *rbt-map* :: ('a, 'b) *rbt* ⇒ ('a, 'b) *map* **where**
rbt-map t = *map-of-alist* (*rbt-in-traverse-pairs t*)
⟨*ML*⟩

fun *rbt-sorted* :: ('a::*linorder*, 'b) *rbt* ⇒ *bool* **where**
rbt-sorted Leaf = *True*
| *rbt-sorted* (*Node l c k v r*) = ((∀ *x*∈*rbt-set l*. *x* < *k*) ∧ (∀ *x*∈*rbt-set r*. *k* < *x*) ∧ *rbt-sorted l* ∧ *rbt-sorted r*)
⟨*ML*⟩

lemma *rbt-sorted-lr* [*forward*]:
rbt-sorted (*Node l c k v r*) ⟹ *rbt-sorted l* ∧ *rbt-sorted r* ⟨*proof*⟩

lemma *rbt-inorder-preserve-set* [*rewrite*]:
rbt-set t = *set* (*rbt-in-traverse t*)
⟨*proof*⟩

lemma *rbt-inorder-sorted* [*rewrite*]:
rbt-sorted t ⟷ *strict-sorted* (*map fst* (*rbt-in-traverse-pairs t*))
⟨*proof*⟩

⟨*ML*⟩

14.6 Balance function

definition *balanceR* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where** [*rewrite*]:
balanceR l k v r =

(*if cl r = R then*
let lr = lsub r; rr = rsub r in
if cl lr = R then Node (Node l B k v (lsub lr)) R (key lr) (val lr) (Node (rsub lr) B (key r) (val r) rr)
else if cl rr = R then Node (Node l B k v lr) R (key r) (val r) (Node (lsub rr) B (key rr) (val rr) (rsub rr))
else Node l B k v r
else Node l B k v r)

definition *balance* :: (*'a*, *'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where** [*rewrite*]:
balance l k v r =
(*if cl l = R then*
let ll = lsub l; rl = rsub l in
if cl ll = R then Node (Node (lsub ll) B (key ll) (val ll) (rsub ll)) R (key l) (val l) (Node (rsub l) B k v r)
else if cl rl = R then Node (Node (lsub l) B (key l) (val l) (lsub rl)) R (key rl) (val rl) (Node (rsub rl) B k v r)
else balanceR l k v r
else balanceR l k v r)
⟨*ML*⟩

lemma *balance-non-Leaf* [*resolve*]: *balance l k v r* ≠ *Leaf* ⟨*proof*⟩

lemma *balance-bdinv* [*forward-arg*]:
bd-inv l ⟹ *bd-inv r* ⟹ *black-depth l* = *black-depth r* ⟹ *bd-inv* (*balance l k v r*)
⟨*proof*⟩

lemma *balance-bd* [*rewrite*]:
bd-inv l ⟹ *bd-inv r* ⟹ *black-depth l* = *black-depth r* ⟹
black-depth (*balance l k v r*) = *black-depth l* + *1*
⟨*proof*⟩

lemma *balance-cl1* [*forward*]:
cl-inv' l ⟹ *cl-inv r* ⟹ *cl-inv* (*balance l k v r*) ⟨*proof*⟩

lemma *balance-cl2* [*forward*]:
cl-inv l ⟹ *cl-inv' r* ⟹ *cl-inv* (*balance l k v r*) ⟨*proof*⟩

lemma *balanceR-inorder-pairs* [*rewrite*]:
rbt-in-traverse-pairs (*balanceR l k v r*) = *rbt-in-traverse-pairs l* @ (*k*, *v*) # *rbt-in-traverse-pairs r* ⟨*proof*⟩

lemma *balance-inorder-pairs* [*rewrite*]:
rbt-in-traverse-pairs (*balance l k v r*) = *rbt-in-traverse-pairs l* @ (*k*, *v*) # *rbt-in-traverse-pairs r* ⟨*proof*⟩

⟨*ML*⟩

14.7 ins function

fun *ins* :: *'a::order* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
  ins x v Leaf = *Node Leaf R x v Leaf*
| *ins x v* (*Node l c y w r*) =
   (*if c* = *B then*
     (*if x* = *y then Node l B x v r*
      else if x < *y then balance* (*ins x v l*) *y w r*
      else balance l y w (*ins x v r*))
    else
     (*if x* = *y then Node l R x v r*
      else if x < *y then Node* (*ins x v l*) *R y w r*
      else Node l R y w (*ins x v r*)))
⟨*ML*⟩

lemma *ins-non-Leaf* [*resolve*]: *ins x v t* ≠ *Leaf*
⟨*proof*⟩

lemma *cl-inv-ins* [*forward*]:
  cl-inv t ⟹ *cl-inv'* (*ins x v t*)
⟨*proof*⟩

lemma *bd-inv-ins*:
  bd-inv t ⟹ *bd-inv* (*ins x v t*) ∧ *black-depth t* = *black-depth* (*ins x v t*)
⟨*proof*⟩
⟨*ML*⟩

lemma *ins-inorder-pairs* [*rewrite*]:
  rbt-sorted t ⟹ *rbt-in-traverse-pairs* (*ins x v t*) = *ordered-insert-pairs x v* (*rbt-in-traverse-pairs t*)
⟨*proof*⟩

14.8 Paint function

fun *paint* :: *color* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
  paint c Leaf = *Leaf*
| *paint c* (*Node l c' x v r*) = *Node l c x v r*
⟨*ML*⟩

lemma *paint-cl-inv'* [*forward*]: *cl-inv' t* ⟹ *cl-inv'* (*paint c t*) ⟨*proof*⟩

lemma *paint-bd-inv* [*forward*]: *bd-inv t* ⟹ *bd-inv* (*paint c t*) ⟨*proof*⟩

lemma *paint-bd* [*rewrite*]:
  bd-inv t ⟹ *t* ≠ *Leaf* ⟹ *cl t* = *B* ⟹ *black-depth* (*paint R t*) = *black-depth t* − *1* ⟨*proof*⟩

lemma *paint-in-traverse-pairs* [*rewrite*]:
  rbt-in-traverse-pairs (*paint c t*) = *rbt-in-traverse-pairs t* ⟨*proof*⟩

14.9 Insert function

definition *rbt-insert* :: *'a::order* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where** [*rewrite*]:
rbt-insert x v t = *paint B* (*ins x v t*)

Correctness results for insertion.

theorem *insert-is-rbt* [*forward*]:
is-rbt t ⟹ *is-rbt* (*rbt-insert x v t*) ⟨*proof*⟩

theorem *insert-sorted* [*forward*]:
rbt-sorted t ⟹ *rbt-sorted* (*rbt-insert x v t*) ⟨*proof*⟩

theorem *insert-rbt-map* [*rewrite*]:
rbt-sorted t ⟹ *rbt-map* (*rbt-insert x v t*) = (*rbt-map t*) {*x* → *v*} ⟨*proof*⟩

14.10 Search on sorted trees and its correctness

fun *rbt-search* :: (*'a::ord*, *'b*) *rbt* ⇒ *'a* ⇒ *'b option* **where**
rbt-search Leaf x = *None*
| *rbt-search* (*Node l c y w r*) *x* =
(*if x* = *y then Some w*
else if x < *y then rbt-search l x*
else rbt-search r x)
⟨*ML*⟩

Correctness of search

theorem *rbt-search-correct* [*rewrite*]:
rbt-sorted t ⟹ *rbt-search t x* = (*rbt-map t*)⟨*x*⟩
⟨*proof*⟩

14.11 balL and balR

definition *balL* :: (*'a*, *'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where** [*rewrite*]:
balL l k v r = (*let lr* = *lsub r in*
if cl l = *R then Node* (*Node* (*lsub l*) *B* (*key l*) (*val l*) (*rsub l*)) *R k v r*
else if r = *Leaf then Node l R k v r*
else if cl r = *B then balance l k v* (*Node* (*lsub r*) *R* (*key r*) (*val r*) (*rsub r*))
else if lr = *Leaf then Node l R k v r*
else if cl lr = *B then*
Node (*Node l B k v* (*lsub lr*)) *R* (*key lr*) (*val lr*) (*balance* (*rsub lr*) (*key r*) (*val r*) (*paint R* (*rsub r*)))
else Node l R k v r)
⟨*ML*⟩

definition *balR* :: (*'a*, *'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where** [*rewrite*]:
balR l k v r = (*let rl* = *rsub l in*
if cl r = *R then Node l R k v* (*Node* (*lsub r*) *B* (*key r*) (*val r*) (*rsub r*))
else if l = *Leaf then Node l R k v r*

else if cl l = B then balance (Node (lsub l) R (key l) (val l) (rsub l)) k v r
else if rl = Leaf then Node l R k v r
else if cl rl = B then
Node (balance (paint R (lsub l)) (key l) (val l) (lsub rl)) R (key rl) (val rl) (Node (rsub rl) B k v r)
else Node l R k v r)
⟨*ML*⟩

lemma *balL-bd* [*forward-arg*]:
bd-inv l ⟹ bd-inv r ⟹ cl r = B ⟹ black-depth l + 1 = black-depth r ⟹
bd-inv (balL l k v r) ∧ black-depth (balL l k v r) = black-depth l + 1 ⟨*proof*⟩

lemma *balL-bd′* [*forward-arg*]:
bd-inv l ⟹ bd-inv r ⟹ cl-inv r ⟹ black-depth l + 1 = black-depth r ⟹
bd-inv (balL l k v r) ∧ black-depth (balL l k v r) = black-depth l + 1 ⟨*proof*⟩

lemma *balL-cl* [*forward-arg*]:
cl-inv′ l ⟹ cl-inv r ⟹ cl r = B ⟹ cl-inv (balL l k v r) ⟨*proof*⟩

lemma *balL-cl′* [*forward*]:
cl-inv′ l ⟹ cl-inv r ⟹ cl-inv′ (balL l k v r) ⟨*proof*⟩

lemma *balR-bd* [*forward-arg*]:
bd-inv l ⟹ bd-inv r ⟹ cl-inv l ⟹ black-depth l = black-depth r + 1 ⟹
bd-inv (balR l k v r) ∧ black-depth (balR l k v r) = black-depth l ⟨*proof*⟩

lemma *balR-cl* [*forward-arg*]:
cl-inv l ⟹ cl-inv′ r ⟹ cl l = B ⟹ cl-inv (balR l k v r) ⟨*proof*⟩

lemma *balR-cl′* [*forward*]:
cl-inv l ⟹ cl-inv′ r ⟹ cl-inv′ (balR l k v r) ⟨*proof*⟩

lemma *balL-in-traverse-pairs* [*rewrite*]:
rbt-in-traverse-pairs (balL l k v r) = rbt-in-traverse-pairs l @ (k, v) # rbt-in-traverse-pairs r ⟨*proof*⟩

lemma *balR-in-traverse-pairs* [*rewrite*]:
rbt-in-traverse-pairs (balR l k v r) = rbt-in-traverse-pairs l @ (k, v) # rbt-in-traverse-pairs r ⟨*proof*⟩

⟨*ML*⟩

14.12 Combine

fun *combine* :: (′*a*, ′*b*) *rbt* ⇒ (′*a*, ′*b*) *rbt* ⇒ (′*a*, ′*b*) *rbt* **where**
combine Leaf t = t
| *combine t Leaf = t*
| *combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2 r2) = (*
if c1 = R then

if c2 = R then
let tm = combine r1 l2 in
if cl tm = R then
Node (Node l1 R k1 v1 (lsub tm)) R (key tm) (val tm) (Node (rsub tm) R k2 v2 r2)
else
Node l1 R k1 v1 (Node tm R k2 v2 r2)
else
Node l1 R k1 v1 (combine r1 (Node l2 c2 k2 v2 r2))
else
if c2 = B then
let tm = combine r1 l2 in
if cl tm = R then
Node (Node l1 B k1 v1 (lsub tm)) R (key tm) (val tm) (Node (rsub tm) B k2 v2 r2)
else
balL l1 k1 v1 (Node tm B k2 v2 r2)
else
Node (combine (Node l1 c1 k1 v1 r1) l2) R k2 v2 r2)
⟨*ML*⟩

lemma *combine-bd* [*forward-arg*]:
bd-inv lt ⟹ *bd-inv rt* ⟹ *black-depth lt = black-depth rt* ⟹
bd-inv (combine lt rt) ∧ *black-depth (combine lt rt) = black-depth lt*
⟨*proof*⟩

lemma *combine-cl*:
cl-inv lt ⟹ *cl-inv rt* ⟹
(*cl lt = B* ⟶ *cl rt = B* ⟶ *cl-inv (combine lt rt)*) ∧ *cl-inv′ (combine lt rt)*
⟨*proof*⟩
⟨*ML*⟩

lemma *combine-in-traverse-pairs* [*rewrite*]:
rbt-in-traverse-pairs (combine lt rt) = rbt-in-traverse-pairs lt @ rbt-in-traverse-pairs rt
⟨*proof*⟩

14.13 Deletion

fun *del* :: *′a::linorder ⇒ (′a, ′b) rbt ⇒ (′a, ′b) rbt* **where**
del x Leaf = Leaf
| *del x (Node l - k v r) =*
(if x = k then combine l r
else if x < k then
if l = Leaf then Node Leaf R k v r
else if cl l = B then balL (del x l) k v r
else Node (del x l) R k v r
else
if r = Leaf then Node l R k v Leaf

else if cl r = B then balR l k v (del x r)
else Node l R k v (del x r))
⟨*ML*⟩

lemma *del-bd* [*forward-arg*]:
bd-inv t ⟹ cl-inv t ⟹ bd-inv (del x t) ∧ (
if cl t = R then black-depth (del x t) = black-depth t
else black-depth (del x t) = black-depth t − 1)
⟨*proof*⟩

lemma *del-cl*:
cl-inv t ⟹ if cl t = R then cl-inv (del x t) else cl-inv′ (del x t)
⟨*proof*⟩
⟨*ML*⟩

lemma *del-in-traverse-pairs* [*rewrite*]:
rbt-sorted t ⟹ rbt-in-traverse-pairs (del x t) = remove-elt-pairs x (rbt-in-traverse-pairs t)
⟨*proof*⟩

definition *delete* :: *′a::linorder ⇒ (′a, ′b) rbt ⇒ (′a, ′b) rbt* **where** [*rewrite*]:
delete x t = paint B (del x t)

Correctness results for deletion.

theorem *delete-is-rbt* [*forward*]:
is-rbt t ⟹ is-rbt (delete x t) ⟨*proof*⟩

theorem *delete-sorted* [*forward*]:
rbt-sorted t ⟹ rbt-sorted (delete x t) ⟨*proof*⟩

theorem *delete-rbt-map* [*rewrite*]:
rbt-sorted t ⟹ rbt-map (delete x t) = delete-map x (rbt-map t) ⟨*proof*⟩

⟨*ML*⟩

end

15 Rectangle intersection

theory *Rect-Intersect*
imports *Interval-Tree*
begin

Functional version of algorithm for detecting rectangle intersection. See [2, Exercise 14.3-7] for a reference.

15.1 Definition of rectangles

datatype *′a rectangle = Rectangle (xint: ′a interval) (yint: ′a interval)*

⟨*ML*⟩

definition *is-rect* :: (*'a::linorder*) *rectangle* ⇒ *bool* **where** [*rewrite*]:
is-rect rect ⟷ *is-interval* (*xint rect*) ∧ *is-interval* (*yint rect*)

definition *is-rect-list* :: (*'a::linorder*) *rectangle list* ⇒ *bool* **where** [*rewrite*]:
is-rect-list rects ⟷ (∀ *i*<*length rects. is-rect* (*rects* ! *i*))

lemma *is-rect-listD*: *is-rect-list rects* ⟹ *i* < *length rects* ⟹ *is-rect* (*rects* ! *i*)
⟨*proof*⟩
⟨*ML*⟩

definition *is-rect-overlap* :: (*'a::linorder*) *rectangle* ⇒ (*'a::linorder*) *rectangle* ⇒ *bool* **where** [*rewrite*]:
is-rect-overlap A B ⟷ (*is-overlap* (*xint A*) (*xint B*) ∧ *is-overlap* (*yint A*) (*yint B*))

definition *has-rect-overlap* :: (*'a::linorder*) *rectangle list* ⇒ *bool* **where** [*rewrite*]:
has-rect-overlap As ⟷ (∃ *i*<*length As.* ∃ *j*<*length As. i* ≠ *j* ∧ *is-rect-overlap* (*As* ! *i*) (*As* ! *j*))

15.2 INS / DEL operations

datatype *'a operation* =
INS (*pos*: *'a*) (*op-idx*: *nat*) (*op-int*: *'a interval*)
| *DEL* (*pos*: *'a*) (*op-idx*: *nat*) (*op-int*: *'a interval*)
⟨*ML*⟩

instantiation *operation* :: (*linorder*) *linorder* **begin**

definition *less*: (*a* < *b*) = (*if pos a* ≠ *pos b then pos a* < *pos b else*
if is-INS a ≠ *is-INS b then is-INS a* ∧ ¬*is-INS b*
else if op-idx a ≠ *op-idx b then op-idx a* < *op-idx b else op-int a* < *op-int b*)
definition *less-eq*: (*a* ≤ *b*) = (*if pos a* ≠ *pos b then pos a* < *pos b else*
if is-INS a ≠ *is-INS b then is-INS a* ∧ ¬*is-INS b*
else if op-idx a ≠ *op-idx b then op-idx a* < *op-idx b else op-int a* ≤ *op-int b*)

instance ⟨*proof*⟩ **end**

⟨*ML*⟩

lemma *operation-leD* [*forward*]:
(*a*::(*'a::linorder operation*)) ≤ *b* ⟹ *pos a* ≤ *pos b* ⟨*proof*⟩

lemma *operation-lessI* [*backward*]:
p1 ≤ *p2* ⟹ *INS p1 n1 i1* < *DEL p2 n2 i2*
⟨*proof*⟩

⟨*ML*⟩

15.3 Set of operations corresponding to a list of rectangles

fun *ins-op* :: *'a rectangle list* ⇒ *nat* ⇒ (*'a*::*linorder*) *operation* **where**
ins-op rects i = *INS* (*low* (*yint* (*rects* ! *i*))) *i* (*xint* (*rects* ! *i*))
⟨*ML*⟩

fun *del-op* :: *'a rectangle list* ⇒ *nat* ⇒ (*'a*::*linorder*) *operation* **where**
del-op rects i = *DEL* (*high* (*yint* (*rects* ! *i*))) *i* (*xint* (*rects* ! *i*))
⟨*ML*⟩

definition *ins-ops* :: *'a rectangle list* ⇒ (*'a*::*linorder*) *operation list* **where** [*rewrite*]:
ins-ops rects = *list* (λ*i*. *ins-op rects i*) (*length rects*)

definition *del-ops* :: *'a rectangle list* ⇒ (*'a*::*linorder*) *operation list* **where** [*rewrite*]:
del-ops rects = *list* (λ*i*. *del-op rects i*) (*length rects*)

lemma *ins-ops-distinct* [*forward*]: *distinct* (*ins-ops rects*)
⟨*proof*⟩

lemma *del-ops-distinct* [*forward*]: *distinct* (*del-ops rects*)
⟨*proof*⟩

lemma *set-ins-ops* [*rewrite*]:
oper ∈ *set* (*ins-ops rects*) ⟷ *op-idx oper* < *length rects* ∧ *oper* = *ins-op rects* (*op-idx oper*)
⟨*proof*⟩

lemma *set-del-ops* [*rewrite*]:
oper ∈ *set* (*del-ops rects*) ⟷ *op-idx oper* < *length rects* ∧ *oper* = *del-op rects* (*op-idx oper*)
⟨*proof*⟩

definition *all-ops* :: *'a rectangle list* ⇒ (*'a*::*linorder*) *operation list* **where** [*rewrite*]:
all-ops rects = *sort* (*ins-ops rects* @ *del-ops rects*)

lemma *all-ops-distinct* [*forward*]: *distinct* (*all-ops rects*)
⟨*proof*⟩

lemma *set-all-ops-idx* [*forward*]:
oper ∈ *set* (*all-ops rects*) ⟹ *op-idx oper* < *length rects* ⟨*proof*⟩

lemma *set-all-ops-ins* [*forward*]:
INS p n i ∈ *set* (*all-ops rects*) ⟹ *INS p n i* = *ins-op rects n* ⟨*proof*⟩

lemma *set-all-ops-del* [*forward*]:
DEL p n i ∈ *set* (*all-ops rects*) ⟹ *DEL p n i* = *del-op rects n* ⟨*proof*⟩

lemma *ins-in-set-all-ops*:
$i < length\ rects \Longrightarrow ins\text{-}op\ rects\ i \in set\ (all\text{-}ops\ rects)$ ⟨*proof*⟩
⟨*ML*⟩

lemma *del-in-set-all-ops*:
$i < length\ rects \Longrightarrow del\text{-}op\ rects\ i \in set\ (all\text{-}ops\ rects)$ ⟨*proof*⟩
⟨*ML*⟩

lemma *all-ops-sorted* [*forward*]: *sorted* (*all-ops rects*) ⟨*proof*⟩

lemma *all-ops-nonempty* [*backward*]: $rects \neq [] \Longrightarrow all\text{-}ops\ rects \neq []$
⟨*proof*⟩

⟨*ML*⟩

15.4 Applying a set of operations

definition *apply-ops-k* :: (*'a::linorder*) *rectangle list* ⇒ *nat* ⇒ *nat set* **where** [*rewrite*]:
apply-ops-k rects k = (*let ops* = *all-ops rects in*
$\{i.\ i < length\ rects \wedge (\exists j<k.\ ins\text{-}op\ rects\ i = ops\ !\ j) \wedge \neg(\exists j<k.\ del\text{-}op\ rects\ i = ops\ !\ j)\})$
⟨*ML*⟩

lemma *apply-ops-set-mem* [*rewrite*]:
$ops = all\text{-}ops\ rects \Longrightarrow$
$i \in apply\text{-}ops\text{-}k\ rects\ k \longleftrightarrow (i < length\ rects \wedge (\exists j<k.\ ins\text{-}op\ rects\ i = ops\ !\ j) \wedge \neg(\exists j<k.\ del\text{-}op\ rects\ i = ops\ !\ j))$
⟨*proof*⟩
⟨*ML*⟩

definition *xints-of* :: *'a rectangle list* ⇒ *nat set* ⇒ ((*'a::linorder*) *idx-interval*) *set* **where** [*rewrite*]:
xints-of rect is = (λ*i*. *IdxInterval* (*xint* (*rect* ! *i*)) *i*) ‘ *is*

lemma *xints-of-mem* [*rewrite*]:
$IdxInterval\ it\ i \in xints\text{-}of\ rect\ is \longleftrightarrow (i \in is \wedge xint\ (rect\ !\ i) = it)$ ⟨*proof*⟩

lemma *xints-diff* [*rewrite*]:
xints-of rects (*A* − *B*) = *xints-of rects A* − *xints-of rects B*
⟨*proof*⟩

definition *has-overlap-at-k* :: (*'a::linorder*) *rectangle list* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
has-overlap-at-k rects k ⟷ (
let S = *apply-ops-k rects k*; *ops* = *all-ops rects in*
is-INS (*ops* ! *k*) ∧ *has-overlap* (*xints-of rects S*) (*op-int* (*ops* ! *k*)))
⟨*ML*⟩

lemma *has-overlap-at-k-equiv* [*forward*]:
is-rect-list rects $\Longrightarrow$ *ops* = *all-ops rects* $\Longrightarrow$ *k* < *length ops* $\Longrightarrow$
has-overlap-at-k rects k $\Longrightarrow$ *has-rect-overlap rects*
⟨*proof*⟩

lemma *has-overlap-at-k-equiv2* [*resolve*]:
is-rect-list rects $\Longrightarrow$ *ops* = *all-ops rects* $\Longrightarrow$ *has-rect-overlap rects* $\Longrightarrow$
∃ *k*<*length ops*. *has-overlap-at-k rects k*
⟨*proof*⟩

definition *has-overlap-lst* :: (′*a*::*linorder*) *rectangle list* ⇒ *bool* **where** [*rewrite*]:
has-overlap-lst rects = (*let ops* = *all-ops rects in* (∃ *k*<*length ops*. *has-overlap-at-k rects k*))

lemma *has-overlap-equiv* [*rewrite*]:
is-rect-list rects $\Longrightarrow$ *has-overlap-lst rects* ⟷ *has-rect-overlap rects* ⟨*proof*⟩

15.5 Implementation of apply_ops_k

lemma *apply-ops-k-next1* [*rewrite*]:
is-rect-list rects $\Longrightarrow$ *ops* = *all-ops rects* $\Longrightarrow$ *n* < *length ops* $\Longrightarrow$ *is-INS* (*ops* ! *n*) $\Longrightarrow$
apply-ops-k rects (*n* + *1*) = *apply-ops-k rects n* ∪ {*op-idx* (*ops* ! *n*)}
⟨*proof*⟩

lemma *apply-ops-k-next2* [*rewrite*]:
is-rect-list rects $\Longrightarrow$ *ops* = *all-ops rects* $\Longrightarrow$ *n* < *length ops* $\Longrightarrow$ ¬*is-INS* (*ops* ! *n*) $\Longrightarrow$
apply-ops-k rects (*n* + *1*) = *apply-ops-k rects n* − {*op-idx* (*ops* ! *n*)} ⟨*proof*⟩

definition *apply-ops-k-next* :: (′*a*::*linorder*) *rectangle list* ⇒ ′*a idx-interval set* ⇒ *nat* ⇒ ′*a idx-interval set* **where**
apply-ops-k-next rects S k = (*let ops* = *all-ops rects in*
(*case ops* ! *k of*
INS p n i ⇒ *S* ∪ {*IdxInterval i n*}
| *DEL p n i* ⇒ *S* − {*IdxInterval i n*}))
⟨*ML*⟩

lemma *apply-ops-k-next-is-correct* [*rewrite*]:
is-rect-list rects $\Longrightarrow$ *ops* = *all-ops rects* $\Longrightarrow$ *n* < *length ops* $\Longrightarrow$
S = *xints-of rects* (*apply-ops-k rects n*) $\Longrightarrow$
xints-of rects (*apply-ops-k rects* (*n* + *1*)) = *apply-ops-k-next rects S n*
⟨*proof*⟩

function *rect-inter* :: *nat rectangle list* ⇒ *nat idx-interval set* ⇒ *nat* ⇒ *bool* **where**
rect-inter rects S k = (*let ops* = *all-ops rects in*
if k ≥ *length ops then False*
else if is-INS (*ops* ! *k*) *then*

if has-overlap S (op-int (ops ! k)) then True
else if k = length ops − 1 then False
else rect-inter rects (apply-ops-k-next rects S k) (k + 1)
else if k = length ops − 1 then False
else rect-inter rects (apply-ops-k-next rects S k) (k + 1))
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *rect-inter-correct-ind* [*rewrite*]:
is-rect-list rects ⟹ *ops* = *all-ops rects* ⟹ *n* < *length ops* ⟹
rect-inter rects (*xints-of rects* (*apply-ops-k rects n*)) *n* ⟷
(∃ *k*<*length ops*. *k* ≥ *n* ∧ *has-overlap-at-k rects k*)
⟨*proof*⟩

Correctness of functional algorithm.

theorem *rect-inter-correct* [*rewrite*]:
is-rect-list rects ⟹ *rect-inter rects* {} *0* ⟷ *has-rect-overlap rects*
⟨*proof*⟩

end

theory *SepLogic-Base*
imports *Auto2-HOL.Auto2-Main*
begin

General auto2 setup for separation logic. The automation defined here can be instantiated for different variants of separation logic.

⟨*ML*⟩

end

16 Separation logic

theory *SepAuto*
imports *SepLogic-Base HOL−Imperative-HOL.Imperative-HOL*
begin

Separation logic for Imperative_HOL, and setup of auto2. The development of separation logic here follows [5] by Lammich and Meis.

16.1 Partial Heaps

datatype *pheap* = *pHeap* (*heapOf*: *heap*) (*addrOf*: *addr set*)
⟨*ML*⟩

fun *in-range* :: (*heap* × *addr set*) ⇒ *bool* **where**
in-range (*h*,*as*) ⟷ (∀ *a*∈*as*. *a* < *lim h*)

⟨*ML*⟩

Two heaps agree on a set of addresses.

definition *relH* :: *addr set* ⇒ *heap* ⇒ *heap* ⇒ *bool* **where** [*rewrite*]:
  relH as h h′ = (*in-range* (*h*, *as*) ∧ *in-range* (*h′*, *as*) ∧
    (∀ *t*. ∀ *a*∈*as*. *refs h t a* = *refs h′ t a* ∧ *arrays h t a* = *arrays h′ t a*))

lemma *relH-D* [*forward*]:
  relH as h h′ ⟹ *in-range* (*h*, *as*) ∧ *in-range* (*h′*, *as*) ⟨*proof*⟩

lemma *relH-D2* [*rewrite*]:
  relH as h h′ ⟹ *a* ∈ *as* ⟹ *refs h t a* = *refs h′ t a*
  relH as h h′ ⟹ *a* ∈ *as* ⟹ *arrays h t a* = *arrays h′ t a* ⟨*proof*⟩
⟨*ML*⟩

lemma *relH-dist-union* [*forward*]:
  relH (*as* ∪ *as′*) *h h′* ⟹ *relH as h h′* ∧ *relH as′ h h′* ⟨*proof*⟩

lemma *relH-ref* [*rewrite*]:
  relH as h h′ ⟹ *addr-of-ref r* ∈ *as* ⟹ *Ref.get h r* = *Ref.get h′ r*
  ⟨*proof*⟩

lemma *relH-array* [*rewrite*]:
  relH as h h′ ⟹ *addr-of-array r* ∈ *as* ⟹ *Array.get h r* = *Array.get h′ r*
  ⟨*proof*⟩

lemma *relH-set-ref* [*resolve*]:
  relH {*a*. *a* < *lim h* ∧ *a* ∉ {*addr-of-ref r*}} *h* (*Ref.set r x h*)
  ⟨*proof*⟩

lemma *relH-set-array* [*resolve*]:
  relH {*a*. *a* < *lim h* ∧ *a* ∉ {*addr-of-array r*}} *h* (*Array.set r x h*)
  ⟨*proof*⟩

16.2 Assertions

datatype *assn-raw* = *Assn* (*assn-fn*: *pheap* ⇒ *bool*)

fun *aseval* :: *assn-raw* ⇒ *pheap* ⇒ *bool* **where**
  aseval (*Assn f*) *h* = *f h*
⟨*ML*⟩

definition *proper* :: *assn-raw* ⇒ *bool* **where** [*rewrite*]:
  proper P = (
    (∀ *h as*. *aseval P* (*pHeap h as*) ⟶ *in-range* (*h*,*as*)) ∧
    (∀ *h h′ as*. *aseval P* (*pHeap h as*) ⟶ *relH as h h′* ⟶ *in-range* (*h′*,*as*) ⟶
aseval P (*pHeap h′ as*)))

fun *in-range-assn* :: *pheap* ⇒ *bool* **where**

in-range-assn (*pHeap h as*) ⟷ (∀ *a*∈*as*. *a* < *lim h*)
⟨*ML*⟩

typedef *assn* = *Collect proper*
⟨*proof*⟩

⟨*ML*⟩

lemma *Abs-assn-inverse′* [*rewrite*]: *proper y* ⟹ *Rep-assn* (*Abs-assn y*) = *y*
⟨*proof*⟩

lemma *proper-Rep-assn* [*forward*]: *proper* (*Rep-assn P*) ⟨*proof*⟩

definition *models* :: *pheap* ⇒ *assn* ⇒ *bool* (**infix** ‹⊨› *50*) **where** [*rewrite-bidir*]:
h ⊨ *P* ⟷ *aseval* (*Rep-assn P*) *h*

lemma *models-in-range* [*resolve*]: *pHeap h as* ⊨ *P* ⟹ *in-range* (*h*,*as*) ⟨*proof*⟩

lemma *mod-relH* [*forward*]: *relH as h h′* ⟹ *pHeap h as* ⊨ *P* ⟹ *pHeap h′ as* ⊨ *P* ⟨*proof*⟩

instantiation *assn* :: *one* **begin**
definition *one-assn* :: *assn* **where** [*rewrite*]:
1 ≡ *Abs-assn* (*Assn* (λ*h*. *addrOf h* = {}))
instance ⟨*proof*⟩ **end**

abbreviation *one-assn* :: *assn* (‹*emp*›) **where** *one-assn* ≡ *1*

lemma *one-assn-rule* [*rewrite*]: *h* ⊨ *emp* ⟷ *addrOf h* = {} ⟨*proof*⟩
⟨*ML*⟩

instantiation *assn* :: *times* **begin**
definition *times-assn* **where** [*rewrite*]:
P ∗ *Q* = *Abs-assn* (*Assn* (
λ*h*. (∃ *as1 as2*. *addrOf h* = *as1* ∪ *as2* ∧ *as1* ∩ *as2* = {} ∧
aseval (*Rep-assn P*) (*pHeap* (*heapOf h*) *as1*) ∧ *aseval* (*Rep-assn Q*) (*pHeap* (*heapOf h*) *as2*))))
instance ⟨*proof*⟩ **end**

lemma *mod-star-conv* [*rewrite*]:
pHeap h as ⊨ *A* ∗ *B* ⟷ (∃ *as1 as2*. *as* = *as1* ∪ *as2* ∧ *as1* ∩ *as2* = {} ∧ *pHeap h as1* ⊨ *A* ∧ *pHeap h as2* ⊨ *B*) ⟨*proof*⟩
⟨*ML*⟩

lemma *aseval-ext* [*backward*]: ∀ *h*. *aseval P h* = *aseval P′ h* ⟹ *P* = *P′*
⟨*proof*⟩

lemma *assn-ext*: ∀ *h as*. *pHeap h as* ⊨ *P* ⟷ *pHeap h as* ⊨ *Q* ⟹ *P* = *Q*
⟨*proof*⟩

⟨*ML*⟩

lemma *assn-one-left*: $1 * P = (P{::}assn)$
⟨*proof*⟩

lemma *assn-times-comm*: $P * Q = Q * (P{::}assn)$
⟨*proof*⟩

lemma *assn-times-assoc*: $(P * Q) * R = P * (Q * (R{::}assn))$
⟨*proof*⟩

instantiation *assn* :: *comm-monoid-mult* **begin**
  instance ⟨*proof*⟩
end

16.2.1 Existential Quantification

definition *ex-assn* :: $('a \Rightarrow assn) \Rightarrow assn$ (**binder** ‹$\exists_A$› *11*) **where** [*rewrite*]:
  $(\exists_A x.\ P\ x) = Abs\text{-}assn\ (Assn\ (\lambda h.\ \exists x.\ h \models P\ x))$

lemma *mod-ex-dist* [*rewrite*]: $(h \models (\exists_A x.\ P\ x)) \longleftrightarrow (\exists x.\ h \models P\ x)$ ⟨*proof*⟩
⟨*ML*⟩

lemma *ex-distrib-star*: $(\exists_A x.\ P\ x * Q) = (\exists_A x.\ P\ x) * Q$
⟨*proof*⟩

16.2.2 Pointers

definition *sngr-assn* :: $'a{::}heap\ ref \Rightarrow 'a \Rightarrow assn$ (**infix** ‹$\mapsto_r$› *82*) **where** [*rewrite*]:
  $r \mapsto_r x = Abs\text{-}assn\ (Assn\ ($
    $\lambda h.\ Ref.get\ (heapOf\ h)\ r = x \wedge addrOf\ h = \{addr\text{-}of\text{-}ref\ r\} \wedge addr\text{-}of\text{-}ref\ r < lim\ (heapOf\ h)))$

lemma *sngr-assn-rule* [*rewrite*]:
  $pHeap\ h\ as \models r \mapsto_r x \longleftrightarrow (Ref.get\ h\ r = x \wedge as = \{addr\text{-}of\text{-}ref\ r\} \wedge addr\text{-}of\text{-}ref\ r < lim\ h)$ ⟨*proof*⟩
⟨*ML*⟩

definition *snga-assn* :: $'a{::}heap\ array \Rightarrow 'a\ list \Rightarrow assn$ (**infix** ‹$\mapsto_a$› *82*) **where** [*rewrite*]:
  $r \mapsto_a x = Abs\text{-}assn\ (Assn\ ($
    $\lambda h.\ Array.get\ (heapOf\ h)\ r = x \wedge addrOf\ h = \{addr\text{-}of\text{-}array\ r\} \wedge addr\text{-}of\text{-}array\ r < lim\ (heapOf\ h)))$

lemma *snga-assn-rule* [*rewrite*]:
  $pHeap\ h\ as \models r \mapsto_a x \longleftrightarrow (Array.get\ h\ r = x \wedge as = \{addr\text{-}of\text{-}array\ r\} \wedge addr\text{-}of\text{-}array\ r < lim\ h)$ ⟨*proof*⟩
⟨*ML*⟩

16.2.3 Pure Assertions

definition *pure-assn* :: *bool* ⇒ *assn* (‹↑›) **where** [*rewrite*]:
$\uparrow b = \mathit{Abs\text{-}assn}\ (\mathit{Assn}\ (\lambda h.\ \mathit{addrOf}\ h = \{\} \wedge b))$

lemma *pure-assn-rule* [*rewrite*]: $h \models \uparrow b \longleftrightarrow (\mathit{addrOf}\ h = \{\} \wedge b)$ ⟨*proof*⟩
⟨*ML*⟩

definition *top-assn* :: *assn* (‹*true*›) **where** [*rewrite*]:
$\mathit{top\text{-}assn} = \mathit{Abs\text{-}assn}\ (\mathit{Assn}\ \mathit{in\text{-}range\text{-}assn})$

lemma *top-assn-rule* [*rewrite*]: $\mathit{pHeap}\ h\ \mathit{as} \models \mathit{true} \longleftrightarrow \mathit{in\text{-}range}\ (h, \mathit{as})$ ⟨*proof*⟩
⟨*ML*⟩

16.2.4 Properties of assertions

abbreviation *bot-assn* :: *assn* (‹*false*›) **where** $\mathit{bot\text{-}assn} \equiv \uparrow \mathit{False}$

lemma *top-assn-reduce*: $\mathit{true} * \mathit{true} = \mathit{true}$
⟨*proof*⟩

lemma *mod-pure-star-dist* [*rewrite*]:
$h \models P * \uparrow b \longleftrightarrow (h \models P \wedge b)$
⟨*proof*⟩

lemma *pure-conj*: $\uparrow(P \wedge Q) = \uparrow P * \uparrow Q$ ⟨*proof*⟩

16.2.5 Entailment and its properties

definition *entails* :: *assn* ⇒ *assn* ⇒ *bool* (**infix** ‹$\Longrightarrow_A$› *10*) **where** [*rewrite*]:
$(P \Longrightarrow_A Q) \longleftrightarrow (\forall h.\ h \models P \longrightarrow h \models Q)$

lemma *entails-triv*: $A \Longrightarrow_A A$ ⟨*proof*⟩
lemma *entails-true*: $A \Longrightarrow_A \mathit{true}$ ⟨*proof*⟩
lemma *entails-frame* [*backward*]: $P \Longrightarrow_A Q \Longrightarrow P * R \Longrightarrow_A Q * R$ ⟨*proof*⟩
lemma *entails-frame'*: $\neg (A * F \Longrightarrow_A Q) \Longrightarrow A \Longrightarrow_A B \Longrightarrow \neg (B * F \Longrightarrow_A Q)$
⟨*proof*⟩
lemma *entails-frame''*: $\neg (P \Longrightarrow_A B * F) \Longrightarrow A \Longrightarrow_A B \Longrightarrow \neg (P \Longrightarrow_A A * F)$
⟨*proof*⟩
lemma *entails-equiv-forward*: $P = Q \Longrightarrow P \Longrightarrow_A Q$ ⟨*proof*⟩
lemma *entails-equiv-backward*: $P = Q \Longrightarrow Q \Longrightarrow_A P$ ⟨*proof*⟩
lemma *entailsD* [*forward*]: $P \Longrightarrow_A Q \Longrightarrow h \models P \Longrightarrow h \models Q$ ⟨*proof*⟩
lemma *entails-trans2*: $A \Longrightarrow_A D * B \Longrightarrow B \Longrightarrow_A C \Longrightarrow A \Longrightarrow_A D * C$ ⟨*proof*⟩

lemma *entails-pure'*: $\neg(\uparrow b \Longrightarrow_A Q) \longleftrightarrow (\neg(\mathit{emp} \Longrightarrow_A Q) \wedge b)$ ⟨*proof*⟩
lemma *entails-pure*: $\neg(P * \uparrow b \Longrightarrow_A Q) \longleftrightarrow (\neg(P \Longrightarrow_A Q) \wedge b)$ ⟨*proof*⟩
lemma *entails-ex*: $\neg((\exists_A x.\ P\ x) \Longrightarrow_A Q) \longleftrightarrow (\exists x.\ \neg(P\ x \Longrightarrow_A Q))$ ⟨*proof*⟩
lemma *entails-ex-post*: $\neg(P \Longrightarrow_A (\exists_A x.\ Q\ x)) \Longrightarrow \forall x.\ \neg(P \Longrightarrow_A Q\ x)$ ⟨*proof*⟩
lemma *entails-pure-post*: $\neg(P \Longrightarrow_A Q * \uparrow b) \Longrightarrow P \Longrightarrow_A Q \Longrightarrow \neg b$ ⟨*proof*⟩

⟨ML⟩

16.3 Definition of the run predicate

inductive *run* :: *'a Heap ⇒ heap option ⇒ heap option ⇒ 'a ⇒ bool* **where**
run c None None r
| *execute c h = None ⟹ run c (Some h) None r*
| *execute c h = Some (r, h′) ⟹ run c (Some h) (Some h′) r*
⟨ML⟩

lemma *run-complete* [*resolve*]:
∃σ′ r. *run c σ σ′* (*r*::*'a*)
⟨*proof*⟩

lemma *run-to-execute* [*forward*]:
run c (*Some h*) *σ′ r ⟹ if σ′ = None then execute c h = None else execute c h = Some* (*r*, *the σ′*)
⟨*proof*⟩

⟨ML⟩
lemma *runE* [*forward*]:
run f (*Some h*) (*Some h′*) *r′ ⟹ run* (*f* ⤜ *g*) (*Some h*) *σ r ⟹ run* (*g r′*) (*Some h′*) *σ r* ⟨*proof*⟩

⟨ML⟩

16.4 Definition of hoare triple, and the frame rule.

definition *new-addrs* :: *heap ⇒ addr set ⇒ heap ⇒ addr set* **where** [*rewrite*]:
new-addrs h as h′ = as ∪ {*a. lim h ≤ a ∧ a < lim h′*}

definition *hoare-triple* :: *assn ⇒ 'a Heap ⇒* (*'a ⇒ assn*) *⇒ bool* (‹<->/ -/ <->›) **where** [*rewrite*]:
<P> c <Q> ⟷ (∀*h as σ r. pHeap h as* ⊨ *P* ⟶ *run c* (*Some h*) *σ r* ⟶
(*σ ≠ None ∧ pHeap* (*the σ*) (*new-addrs h as* (*the σ*)) ⊨ *Q r ∧ relH* {*a . a < lim h ∧ a ∉ as*} *h* (*the σ*) ∧
lim h ≤ lim (*the σ*)))

lemma *hoare-tripleD* [*forward*]:
<P> c <Q> ⟹ run c (*Some h*) *σ r ⟹* ∀*as. pHeap h as* ⊨ *P* ⟶
(*σ ≠ None ∧ pHeap* (*the σ*) (*new-addrs h as* (*the σ*)) ⊨ *Q r ∧ relH* {*a . a < lim h ∧ a ∉ as*} *h* (*the σ*) ∧
lim h ≤ lim (*the σ*))
⟨*proof*⟩
⟨ML⟩

abbreviation *hoare-triple′* :: *assn ⇒ 'r Heap ⇒* (*'r ⇒ assn*) *⇒ bool* (‹<-> - <->$_t$›) **where**
<P> c <Q>$_t$ ≡ *<P> c <λr. Q r ∗ true>*

theorem *frame-rule* [*backward*]:
$<P>\ c\ <Q> \Longrightarrow\ <P * R>\ c\ <\lambda x.\ Q\ x * R>$
$\langle proof \rangle$

This is the last use of the definition of separating conjunction.

$\langle ML \rangle$

theorem *bind-rule*:
$<P>\ f\ <Q> \Longrightarrow \forall x.\ <Q\ x>\ g\ x\ <R> \Longrightarrow\ <P>\ f \gg= g\ <R>$
$\langle proof \rangle$

Actual statement used:

lemma *bind-rule'*:
$<P>\ f\ <Q> \Longrightarrow \neg <P>\ f \gg= g\ <R> \Longrightarrow \exists x.\ \neg <Q\ x>\ g\ x\ <R>$ $\langle proof \rangle$

lemma *pre-rule'*:
$\neg <P * R>\ f\ <Q> \Longrightarrow P \Longrightarrow_A P' \Longrightarrow \neg <P' * R>\ f\ <Q>$
$\langle proof \rangle$

lemma *pre-rule''*:
$<P>\ f\ <Q> \Longrightarrow P' \Longrightarrow_A P * R \Longrightarrow\ <P'>\ f\ <\lambda x.\ Q\ x * R>$
$\langle proof \rangle$

lemma *pre-ex-rule*:
$\neg <\exists_A x.\ P\ x>\ f\ <Q> \longleftrightarrow (\exists x.\ \neg <P\ x>\ f\ <Q>)$ $\langle proof \rangle$

lemma *pre-pure-rule*:
$\neg <P * \uparrow b>\ f\ <Q> \longleftrightarrow \neg <P>\ f\ <Q> \wedge b$ $\langle proof \rangle$

lemma *pre-pure-rule'*:
$\neg <\uparrow b>\ f\ <Q> \longleftrightarrow \neg <emp>\ f\ <Q> \wedge b$ $\langle proof \rangle$

lemma *post-rule*:
$<P>\ f\ <Q> \Longrightarrow \forall x.\ Q\ x \Longrightarrow_A R\ x \Longrightarrow\ <P>\ f\ <R>$ $\langle proof \rangle$

$\langle ML \rangle$

Actual statement used:

lemma *post-rule'*:
$<P>\ f\ <Q> \Longrightarrow \neg <P>\ f\ <R> \Longrightarrow \exists x.\ \neg (Q\ x \Longrightarrow_A R\ x)$ $\langle proof \rangle$

lemma *norm-pre-pure-iff*: $<P * \uparrow b>\ c\ <Q> \longleftrightarrow (b \longrightarrow\ <P>\ c\ <Q>)$ $\langle proof \rangle$
lemma *norm-pre-pure-iff2*: $<\uparrow b>\ c\ <Q> \longleftrightarrow (b \longrightarrow\ <emp>\ c\ <Q>)$ $\langle proof \rangle$

16.5 Hoare triples for atomic commands

First, those that do not modify the heap.

$\langle ML \rangle$

lemma *assert-rule*:
<↑(R x)> assert R x <λr. ↑(r = x)> ⟨proof⟩

lemma *execute-return′* [rewrite]: execute (return x) h = Some (x, h) ⟨proof⟩
lemma *return-rule*:
<emp> return x <λr. ↑(r = x)> ⟨proof⟩

⟨ML⟩
lemma *nth-rule*:
<a $\mapsto_a$ xs * ↑(i < length xs)> Array.nth a i <λr. a $\mapsto_a$ xs * ↑(r = xs ! i)>
⟨proof⟩

⟨ML⟩
lemma *length-rule*:
<a $\mapsto_a$ xs> Array.len a <λr. a $\mapsto_a$ xs * ↑(r = length xs)> ⟨proof⟩

⟨ML⟩
lemma *lookup-rule*:
<p $\mapsto_r$ x> !p <λr. p $\mapsto_r$ x * ↑(r = x)> ⟨proof⟩

⟨ML⟩
lemma *freeze-rule*:
<a $\mapsto_a$ xs> Array.freeze a <λr. a $\mapsto_a$ xs * ↑(r = xs)> ⟨proof⟩

Next, the update rules.

⟨ML⟩
lemma *Array-lim-set* [rewrite]: lim (Array.set p xs h) = lim h ⟨proof⟩

⟨ML⟩
lemma *upd-rule*:
<a $\mapsto_a$ xs * ↑(i < length xs)> Array.upd i x a <λr. a $\mapsto_a$ list-update xs i x * ↑(r = a)> ⟨proof⟩

⟨ML⟩
lemma *update-rule*:
<p $\mapsto_r$ y> p := x <λr. p $\mapsto_r$ x> ⟨proof⟩

Finally, the allocation rules.

lemma *lim-set-gen* [rewrite]: lim (h(|lim := l|)) = l ⟨proof⟩

lemma *Array-alloc-def′* [rewrite]:
Array.alloc xs h = (let l = lim h; r = Array l in (r, (Array.set r xs (h(|lim := l + 1|)))))
⟨proof⟩

⟨ML⟩

lemma *refs-on-Array-set* [rewrite]: refs (Array.set p xs h) t i = refs h t i
⟨proof⟩

lemma *arrays-on-Ref-set* [*rewrite*]: $arrays\ (Ref.set\ p\ x\ h)\ t\ i = arrays\ h\ t\ i$
  ⟨*proof*⟩

lemma *refs-on-Array-alloc* [*rewrite*]: $refs\ (snd\ (Array.alloc\ xs\ h))\ t\ i = refs\ h\ t\ i$
  ⟨*proof*⟩

lemma *arrays-on-Ref-alloc* [*rewrite*]: $arrays\ (snd\ (Ref.alloc\ x\ h))\ t\ i = arrays\ h\ t\ i$
  ⟨*proof*⟩

lemma *arrays-on-Array-alloc* [*rewrite*]: $i < lim\ h \Longrightarrow arrays\ (snd\ (Array.alloc\ xs\ h))\ t\ i = arrays\ h\ t\ i$
  ⟨*proof*⟩

lemma *refs-on-Ref-alloc* [*rewrite*]: $i < lim\ h \Longrightarrow refs\ (snd\ (Ref.alloc\ x\ h))\ t\ i = refs\ h\ t\ i$
  ⟨*proof*⟩

⟨*ML*⟩
lemma *new-rule*:
  $<emp>\ Array.new\ n\ x\ <\lambda r.\ r \mapsto_a replicate\ n\ x>$ ⟨*proof*⟩

⟨*ML*⟩
lemma *of-list-rule*:
  $<emp>\ Array.of\text{-}list\ xs\ <\lambda r.\ r \mapsto_a xs>$ ⟨*proof*⟩

⟨*ML*⟩
lemma *ref-rule*:
  $<emp>\ ref\ x\ <\lambda r.\ r \mapsto_r x>$ ⟨*proof*⟩

⟨*ML*⟩

16.6 Definition of procedures

ASCII abbreviations for ML files.

abbreviation (*input*) *ex-assn-ascii* :: $('a \Rightarrow assn) \Rightarrow assn$ (**binder** ‹*EXA*› *11*)
  where $ex\text{-}assn\text{-}ascii \equiv ex\text{-}assn$

abbreviation (*input*) *models-ascii* :: $pheap \Rightarrow assn \Rightarrow bool$ (**infix** ‹\|=› *50*)
  where $h \mid= P \equiv h \models P$

⟨*ML*⟩

Some simple tests

theorem $<emp>\ ref\ x\ <\lambda r.\ r \mapsto_r x>$ ⟨*proof*⟩
theorem $<a \mapsto_r x>\ ref\ x\ <\lambda r.\ a \mapsto_r x * r \mapsto_r x>$ ⟨*proof*⟩
theorem $<a \mapsto_r x>\ (!a)\ <\lambda r.\ a \mapsto_r x * \uparrow(r = x)>$ ⟨*proof*⟩
theorem $<a \mapsto_r x * b \mapsto_r y>\ (!a)\ <\lambda r.\ a \mapsto_r x * b \mapsto_r y * \uparrow(r = x)>$ ⟨*proof*⟩

theorem $<a \mapsto_r x * b \mapsto_r y>\ (!b)\ <\lambda r.\ a \mapsto_r x * b \mapsto_r y * \uparrow(r = y)>$ ⟨*proof*⟩
theorem $<a \mapsto_r x>\ do\ \{\ a := y;\ !a\ \}\ <\lambda r.\ a \mapsto_r y * \uparrow(r = y)>$ ⟨*proof*⟩
theorem $<a \mapsto_r x>\ do\ \{\ a := y;\ a := z;\ !a\ \}\ <\lambda r.\ a \mapsto_r z * \uparrow(r = z)>$ ⟨*proof*⟩
theorem $<a \mapsto_r x>\ do\ \{\ y \leftarrow !a;\ ref\ y\}\ <\lambda r.\ a \mapsto_r x * r \mapsto_r x>$ ⟨*proof*⟩
theorem $<emp>\ return\ x\ <\lambda r.\ \uparrow(r = x)>$ ⟨*proof*⟩

end

theory *GCD-Impl*
  imports *SepAuto*
begin

A tutorial example for computation of GCD.

Turn on auto2's trace

declare [[*print-trace*]]

Property of gcd that justifies the recursive computation. Add as a right-to-left rewrite rule.

⟨*ML*⟩

Functional version of gcd.

fun *gcd-fun* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  gcd-fun a b = (*if b* = *0 then a else gcd-fun b* (*a mod b*))

The fun package automatically generates induction rule upon showing termination. This adds the induction rule for the @fun_induct command.

⟨*ML*⟩

lemma *gcd-fun-correct*:
  gcd-fun a b = *gcd a b*
⟨*proof*⟩

Imperative version of gcd.

partial-function (*heap*) *gcd-impl* :: *nat* ⇒ *nat* ⇒ *nat Heap* **where**
  gcd-impl a b = (
    if b = *0 then return a*
    else do {
      c ← *return* (*a mod b*);
      r ← *gcd-impl b c*;
      return r
    })

The program is sufficiently simple that we can prove the Hoare triple directly (without going through the functional program).

theorem *gcd-impl-correct*:
  $<emp>\ gcd\text{-}impl\ a\ b\ <\lambda r.\ \uparrow(r = gcd\ a\ b)>$

⟨*proof*⟩

Turn off trace.

declare [[*print-trace* = *false*]]

end

17 Implementation of linked list

theory *LinkedList*
imports *SepAuto*
begin

Examples in linked lists. Definitions and some of the examples are based on List_Seg and Open_List theories in [5] by Lammich and Meis.

17.1 List Assertion

datatype *'a node* = *Node* (*val*: *'a*) (*nxt*: *'a node ref option*)
⟨*ML*⟩

fun *node-encode* :: *'a::heap node* ⇒ *nat* **where**
node-encode (*Node x r*) = *to-nat* (*x*, *r*)

instance *node* :: (*heap*) *heap*
⟨*proof*⟩

fun *os-list* :: *'a::heap list* ⇒ *'a node ref option* ⇒ *assn* **where**
os-list [] *p* = ↑(*p* = *None*)
| *os-list* (*x* # *l*) (*Some p*) = ($\exists_A$*q*. *p* $\mapsto_r$ *Node x q* ∗ *os-list l q*)
| *os-list* (*x* # *l*) *None* = *false*
⟨*ML*⟩

lemma *os-list-empty* [*forward-ent*]:
os-list [] *p* $\Longrightarrow_A$ ↑(*p* = *None*) ⟨*proof*⟩

lemma *os-list-Cons* [*forward-ent*]:
os-list (*x* # *l*) *p* $\Longrightarrow_A$ ($\exists_A$*q*. *the p* $\mapsto_r$ *Node x q* ∗ *os-list l q* ∗ ↑(*p* ≠ *None*))
⟨*proof*⟩

lemma *os-list-none*: *emp* $\Longrightarrow_A$ *os-list* [] *None* ⟨*proof*⟩

lemma *os-list-constr-ent*:
p $\mapsto_r$ *Node x q* ∗ *os-list l q* $\Longrightarrow_A$ *os-list* (*x* # *l*) (*Some p*) ⟨*proof*⟩

⟨*ML*⟩

type-synonym *'a os-list* = *'a node ref option*

17.2 Basic operations

definition *os-empty* :: 'a::heap os-list Heap **where**
  os-empty = return None

lemma *os-empty-rule* [*hoare-triple*]:
  <emp> os-empty <os-list []> ⟨*proof*⟩

definition *os-is-empty* :: 'a::heap os-list ⇒ bool Heap **where**
  os-is-empty b = return (b = None)

lemma *os-is-empty-rule* [*hoare-triple*]:
  <os-list xs b> os-is-empty b <λr. os-list xs b ∗ ↑(r ⟷ xs = [])>
⟨*proof*⟩

definition *os-prepend* :: 'a ⇒ 'a::heap os-list ⇒ 'a os-list Heap **where**
  os-prepend a n = do { p ← ref (Node a n); return (Some p) }

lemma *os-prepend-rule* [*hoare-triple*]:
  <os-list xs n> os-prepend x n <os-list (x # xs)> ⟨*proof*⟩

definition *os-pop* :: 'a::heap os-list ⇒ ('a × 'a os-list) Heap **where**
  os-pop r = (case r of
    None ⇒ raise STR ''Empty Os-list'' |
    Some p ⇒ do {m ← !p; return (val m, nxt m)})

lemma *os-pop-rule* [*hoare-triple*]:
  <os-list xs (Some p)>
   os-pop (Some p)
  <λ(x,r'). os-list (tl xs) r' ∗ p $\mapsto_r$ (Node x r') ∗ ↑(x = hd xs)>
⟨*proof*⟩

17.3 Reverse

partial-function (*heap*) *os-reverse-aux* :: 'a::heap os-list ⇒ 'a os-list ⇒ 'a os-list Heap **where**
  os-reverse-aux q p = (case p of
    None ⇒ return q |
    Some r ⇒ do {
      v ← !r;
      r := Node (val v) q;
      os-reverse-aux p (nxt v) })

lemma *os-reverse-aux-rule* [*hoare-triple*]:
  <os-list xs p ∗ os-list ys q>
    os-reverse-aux q p
  <os-list ((rev xs) @ ys)>
⟨*proof*⟩

definition *os-reverse* :: 'a::heap os-list ⇒ 'a os-list Heap **where**

os-reverse p = *os-reverse-aux None p*

lemma *os-reverse-rule*:
<*os-list xs p*> *os-reverse p* <*os-list* (*rev xs*)> ⟨*proof*⟩

17.4 Remove

⟨*ML*⟩

partial-function (*heap*) *os-rem* :: *'a::heap* ⇒ *'a node ref option* ⇒ *'a node ref option Heap* **where**
os-rem x b = (*case b of*
None ⇒ *return None* |
Some p ⇒ *do* {
n ← !*p*;
q ← *os-rem x* (*nxt n*);
(*if* (*val n* = *x*)
then return q
else do {
p := *Node* (*val n*) *q*;
return (*Some p*) }) })

lemma *os-rem-rule* [*hoare-triple*]:
<*os-list xs b*> *os-rem x b* <λ*r*. *os-list* (*removeAll x xs*) *r*>$_t$
⟨*proof*⟩

17.5 Extract list

partial-function (*heap*) *extract-list* :: *'a::heap os-list* ⇒ *'a list Heap* **where**
extract-list p = (*case p of*
None ⇒ *return* []
| *Some pp* ⇒ *do* {
v ← !*pp*;
ls ← *extract-list* (*nxt v*);
return (*val v* # *ls*)
})

lemma *extract-list-rule* [*hoare-triple*]:
<*os-list l p*> *extract-list p* <λ*r*. *os-list l p* ∗ ↑(*r* = *l*)>
⟨*proof*⟩

17.6 Ordered insert

fun *list-insert* :: *'a::ord* ⇒ *'a list* ⇒ *'a list* **where**
list-insert x [] = [*x*]
| *list-insert x* (*y* # *ys*) = (
if x ≤ *y then x* # (*y* # *ys*) *else y* # *list-insert x ys*)
⟨*ML*⟩

lemma *list-insert-length*:

length (*list-insert x xs*) = *length xs* + *1*
⟨*proof*⟩
⟨*ML*⟩

lemma *list-insert-mset* [*rewrite*]:
mset (*list-insert x xs*) = {#*x*#} + *mset xs*
⟨*proof*⟩

lemma *list-insert-set* [*rewrite*]:
set (*list-insert x xs*) = {*x*} ∪ *set xs*
⟨*proof*⟩

lemma *list-insert-sorted* [*forward*]:
sorted xs ⟹ *sorted* (*list-insert x xs*)
⟨*proof*⟩

partial-function (*heap*) *os-insert* :: *'a*::{*ord*,*heap*} ⇒ *'a os-list* ⇒ *'a os-list Heap*
where
os-insert x b = (*case b of*
None ⇒ *os-prepend x None*
| *Some p* ⇒ *do* {
v ← !*p*;
(*if x* ≤ *val v then os-prepend x b*
else do {
q ← *os-insert x* (*nxt v*);
p := *Node* (*val v*) *q*;
return (*Some p*) }) })

lemma *os-insert-to-fun* [*hoare-triple*]:
<*os-list xs b*> *os-insert x b* <*os-list* (*list-insert x xs*)>
⟨*proof*⟩

17.7 Insertion sort

fun *insert-sort* :: *'a*::*ord list* ⇒ *'a list* **where**
insert-sort [] = []
| *insert-sort* (*x* # *xs*) = *list-insert x* (*insert-sort xs*)
⟨*ML*⟩

lemma *insert-sort-mset* [*rewrite*]:
mset (*insert-sort xs*) = *mset xs*
⟨*proof*⟩

lemma *insert-sort-sorted* [*forward*]:
sorted (*insert-sort xs*)
⟨*proof*⟩

lemma *insert-sort-is-sort* [*rewrite*]:
insert-sort xs = *sort xs* ⟨*proof*⟩

fun *os-insert-sort-aux* :: *'a*::{*ord*,*heap*} *list* ⇒ *'a os-list Heap* **where**
  os-insert-sort-aux [] = (*return None*)
| *os-insert-sort-aux* (*x* # *xs*) = *do* {
      b ← *os-insert-sort-aux xs*;
      b' ← *os-insert x b*;
      return b'
   }

lemma *os-insert-sort-aux-correct* [*hoare-triple*]:
  <*emp*> *os-insert-sort-aux xs* <*os-list* (*insert-sort xs*)>
⟨*proof*⟩

definition *os-insert-sort* :: *'a*::{*ord*,*heap*} *list* ⇒ *'a list Heap* **where**
  os-insert-sort xs = *do* {
    p ← *os-insert-sort-aux xs*;
    l ← *extract-list p*;
    return l
  }

lemma *insertion-sort-rule* [*hoare-triple*]:
  <*emp*> *os-insert-sort xs* <λ*ys*. ↑(*ys* = *sort xs*)>$_t$ ⟨*proof*⟩

17.8 Merging two lists

fun *merge-list* :: (*'a*::*ord*) *list* ⇒ *'a list* ⇒ *'a list* **where**
  merge-list xs [] = *xs*
| *merge-list* [] *ys* = *ys*
| *merge-list* (*x* # *xs*) (*y* # *ys*) = (
    if x ≤ *y then x* # (*merge-list xs* (*y* # *ys*))
    else y # (*merge-list* (*x* # *xs*) *ys*))
⟨*ML*⟩

lemma *merge-list-correct* [*rewrite*]:
  set (*merge-list xs ys*) = *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *merge-list-sorted* [*forward*]:
  sorted xs ⟹ *sorted ys* ⟹ *sorted* (*merge-list xs ys*)
⟨*proof*⟩

partial-function (*heap*) *merge-os-list* :: (*'a*::{*heap*, *ord*}) *os-list* ⇒ *'a os-list* ⇒ *'a os-list Heap* **where**
  merge-os-list p q = (
    if p = *None then return q*
    else if q = *None then return p*
    else do {
      np ← !(*the p*); *nq* ← !(*the q*);
      if val np ≤ *val nq then*

```
      do { npq ← merge-os-list (nxt np) q;
           (the p) := Node (val np) npq;
           return p }
    else
      do { pnq ← merge-os-list p (nxt nq);
           (the q) := Node (val nq) pnq;
           return q } })
```

```
lemma merge-os-list-to-fun [hoare-triple]:
  <os-list xs p ∗ os-list ys q>
  merge-os-list p q
  <λr. os-list (merge-list xs ys) r>
⟨proof⟩
```

17.9 List copy

```
partial-function (heap) copy-os-list :: 'a::heap os-list ⇒ 'a os-list Heap where
  copy-os-list b = (case b of
      None ⇒ return None
    | Some p ⇒ do {
        v ← !p;
        q ← copy-os-list (nxt v);
        os-prepend (val v) q })
```

```
lemma copy-os-list-rule [hoare-triple]:
  <os-list xs b> copy-os-list b <λr. os-list xs b ∗ os-list xs r>
⟨proof⟩
```

17.10 Higher-order functions

```
partial-function (heap) map-os-list :: ('a::heap ⇒ 'a) ⇒ 'a os-list ⇒ 'a os-list
Heap where
  map-os-list f b = (case b of
      None ⇒ return None
    | Some p ⇒ do {
        v ← !p;
        q ← map-os-list f (nxt v);
        p := Node (f (val v)) q;
        return (Some p) })
```

```
lemma map-os-list-rule [hoare-triple]:
  <os-list xs b> map-os-list f b <os-list (map f xs)>
⟨proof⟩
```

```
partial-function (heap) filter-os-list :: ('a::heap ⇒ bool) ⇒ 'a os-list ⇒ 'a os-list
Heap where
  filter-os-list f b = (case b of
      None ⇒ return None
    | Some p ⇒ do {
        v ← !p;
```

```
        q ← filter-os-list f (nxt v);
        (if (f (val v)) then do {
           p := Node (val v) q;
           return (Some p) }
         else return q) })
```

```
lemma filter-os-list-rule [hoare-triple]:
  <os-list xs b> filter-os-list f b <λr. os-list (filter f xs) r ∗ true>
⟨proof⟩
```

```
partial-function (heap) filter-os-list2 :: ('a::heap ⇒ bool) ⇒ 'a os-list ⇒ 'a os-list
Heap where
  filter-os-list2 f b = (case b of
      None ⇒ return None
    | Some p ⇒ do {
        v ← !p;
        q ← filter-os-list2 f (nxt v);
        (if (f (val v)) then os-prepend (val v) q
         else return q) })
```

```
lemma filter-os-list2-rule [hoare-triple]:
  <os-list xs b> filter-os-list2 f b <λr. os-list xs b ∗ os-list (filter f xs) r>
⟨proof⟩
```

```
⟨ML⟩
```

```
partial-function (heap) fold-os-list :: ('a::heap ⇒ 'b ⇒ 'b) ⇒ 'a os-list ⇒ 'b ⇒
'b Heap where
  fold-os-list f b x = (case b of
      None ⇒ return x
    | Some p ⇒ do {
       v ← !p;
       r ← fold-os-list f (nxt v) (f (val v) x);
       return r})
```

```
lemma fold-os-list-rule [hoare-triple]:
  <os-list xs b> fold-os-list f b x <λr. os-list xs b ∗ ↑(r = fold f xs x)>
⟨proof⟩
```

end

18 Implementation of binary search tree

```
theory BST-Impl
  imports SepAuto ../Functional/BST
begin
```

Imperative version of binary search trees.

18.1 Tree nodes

datatype (*'a*, *'b*) *node* =
Node (*lsub*: (*'a*, *'b*) *node ref option*) (*key*: *'a*) (*val*: *'b*) (*rsub*: (*'a*, *'b*) *node ref option*)
⟨*ML*⟩

fun *node-encode* :: (*'a*::*heap*, *'b*::*heap*) *node* ⇒ *nat* **where**
node-encode (*Node l k v r*) = *to-nat* (*l*, *k*, *v*, *r*)

instance *node* :: (*heap*, *heap*) *heap*
⟨*proof*⟩

fun *btree* :: (*'a*::*heap*, *'b*::*heap*) *tree* ⇒ (*'a*, *'b*) *node ref option* ⇒ *assn* **where**
btree Tip p = ↑(*p* = *None*)
| *btree* (*tree.Node lt k v rt*) (*Some p*) = ($\exists_A$*lp rp*. *p* $\mapsto_r$ *Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp*)
| *btree* (*tree.Node lt k v rt*) *None* = *false*
⟨*ML*⟩

lemma *btree-Tip* [*forward-ent*]: *btree Tip p* $\Longrightarrow_A$ ↑(*p* = *None*) ⟨*proof*⟩

lemma *btree-Node* [*forward-ent*]:
btree (*tree.Node lt k v rt*) *p* $\Longrightarrow_A$ ($\exists_A$*lp rp*. *the p* $\mapsto_r$ *Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ∗ ↑(*p* ≠ *None*))
⟨*proof*⟩

lemma *btree-none*: *emp* $\Longrightarrow_A$ *btree tree.Tip None* ⟨*proof*⟩

lemma *btree-constr-ent*:
p $\mapsto_r$ *Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp* $\Longrightarrow_A$ *btree* (*tree.Node lt k v rt*) (*Some p*) ⟨*proof*⟩

⟨*ML*⟩

type-synonym (*'a*, *'b*) *btree* = (*'a*, *'b*) *node ref option*

18.2 Operations

18.2.1 Basic operations

definition *tree-empty* :: (*'a*, *'b*) *btree Heap* **where**
tree-empty = *return None*

lemma *tree-empty-rule* [*hoare-triple*]:
<*emp*> *tree-empty* <*btree Tip*> ⟨*proof*⟩

definition *tree-is-empty* :: (*'a*, *'b*) *btree* ⇒ *bool Heap* **where**
tree-is-empty b = *return* (*b* = *None*)

lemma *tree-is-empty-rule*:
<btree t b> tree-is-empty b <λr. btree t b ∗ ↑(r ⟷ t = Tip)> ⟨proof⟩

definition *btree-constr* ::
('a::heap, 'b::heap) btree ⇒ 'a ⇒ 'b ⇒ ('a, 'b) btree ⇒ ('a, 'b) btree Heap **where**
btree-constr lp k v rp = do { p ← ref (Node lp k v rp); return (Some p) }

lemma *btree-constr-rule* [*hoare-triple*]:
<btree lt lp ∗ btree rt rp> btree-constr lp k v rp <btree (tree.Node lt k v rt)>
⟨proof⟩

18.2.2 Insertion

partial-function (*heap*) *btree-insert* ::
'a::{heap,linorder} ⇒ 'b::heap ⇒ ('a, 'b) btree ⇒ ('a, 'b) btree Heap **where**
btree-insert k v b = (case b of
None ⇒ btree-constr None k v None
| Some p ⇒ do {
t ← !p;
(if k = key t then do {
p := Node (lsub t) k v (rsub t);
return (Some p) }
else if k < key t then do {
q ← btree-insert k v (lsub t);
p := Node q (key t) (val t) (rsub t);
return (Some p) }
else do {
q ← btree-insert k v (rsub t);
p := Node (lsub t) (key t) (val t) q;
return (Some p)}) })

lemma *btree-insert-to-fun* [*hoare-triple*]:
<btree t b>
btree-insert k v b
<btree (tree-insert k v t)>
⟨proof⟩

18.2.3 Deletion

partial-function (*heap*) *btree-del-min* :: ('a::heap, 'b::heap) btree ⇒ (('a × 'b) × ('a, 'b) btree) Heap **where**
btree-del-min b = (case b of
None ⇒ raise STR ''del-min: empty tree''
| Some p ⇒ do {
t ← !p;
(if lsub t = None then
return ((key t, val t), rsub t)
else do {
r ← btree-del-min (lsub t);
p := Node (snd r) (key t) (val t) (rsub t);

```
      return (fst r, Some p) }) })

lemma btree-del-min-to-fun [hoare-triple]:
  <btree t b * ↑(b ≠ None)>
   btree-del-min b
   <λ(r,p). btree (snd (del-min t)) p * ↑(r = fst (del-min t))>_t
⟨proof⟩

definition btree-del-elt :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-del-elt b = (case b of
     None ⇒ raise STR ''del-elt: empty tree''
   | Some p ⇒ do {
       t ← !p;
       (if lsub t = None then return (rsub t)
        else if rsub t = None then return (lsub t)
        else do {
          r ← btree-del-min (rsub t);
          p := Node (lsub t) (fst (fst r)) (snd (fst r)) (snd r);
          return (Some p) }) })

lemma btree-del-elt-to-fun [hoare-triple]:
  <btree (tree.Node lt x v rt) b>
   btree-del-elt b
   <btree (delete-elt-tree (tree.Node lt x v rt))>_t ⟨proof⟩

partial-function (heap) btree-delete ::
  'a::{heap,linorder} ⇒ ('a, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-delete x b = (case b of
     None ⇒ return None
   | Some p ⇒ do {
      t ← !p;
      (if x = key t then do {
         r ← btree-del-elt b;
         return r }
       else if x < key t then do {
         q ← btree-delete x (lsub t);
         p := Node q (key t) (val t) (rsub t);
         return (Some p) }
       else do {
         q ← btree-delete x (rsub t);
         p := Node (lsub t) (key t) (val t) q;
         return (Some p)}) })

lemma btree-delete-to-fun [hoare-triple]:
  <btree t b>
   btree-delete x b
   <btree (tree-delete x t)>_t
⟨proof⟩
```

18.2.4 Search

partial-function (*heap*) *btree-search* ::
  'a::{*heap*,*linorder*} ⇒ (*'a*, *'b*::*heap*) *btree* ⇒ *'b option Heap* **where**
  btree-search x b = (*case b of*
    None ⇒ *return None*
  | *Some p* ⇒ *do* {
      t ← !*p*;
      (*if x* = *key t then return* (*Some* (*val t*))
       else if x < *key t then btree-search x* (*lsub t*)
       else btree-search x (*rsub t*)) })

lemma *btree-search-correct* [*hoare-triple*]:
  <*btree t b* ∗ ↑(*tree-sorted t*)>
   btree-search x b
  <λ*r*. *btree t b* ∗ ↑(*r* = *tree-search t x*)>
⟨*proof*⟩

18.3 Outer interface

Express Hoare triples for operations on binary search tree in terms of the mapping represented by the tree.

definition *btree-map* :: (*'a*, *'b*) *map* ⇒ (*'a*::{*heap*,*linorder*}, *'b*::*heap*) *node ref option* ⇒ *assn* **where**
  btree-map M p = ($\exists_A$ *t*. *btree t p* ∗ ↑(*tree-sorted t*) ∗ ↑(*M* = *tree-map t*))
⟨*ML*⟩

theorem *btree-empty-rule-map* [*hoare-triple*]:
  <*emp*> *tree-empty* <*btree-map empty-map*> ⟨*proof*⟩

theorem *btree-insert-rule-map* [*hoare-triple*]:
  <*btree-map M b*> *btree-insert k v b* <*btree-map* (*M* {*k* → *v*})> ⟨*proof*⟩

theorem *btree-delete-rule-map* [*hoare-triple*]:
  <*btree-map M b*> *btree-delete x b* <*btree-map* (*delete-map x M*)>$_t$ ⟨*proof*⟩

theorem *btree-search-rule-map* [*hoare-triple*]:
  <*btree-map M b*> *btree-search x b* <λ*r*. *btree-map M b* ∗ ↑(*r* = *M*⟨*x*⟩)> ⟨*proof*⟩

end

19 Implementation of red-black tree

theory *RBTree-Impl*
  imports *SepAuto ../Functional/RBTree*
begin

Verification of imperative red-black trees.

19.1 Tree nodes

datatype ('a, 'b) *rbt-node* =
  Node (*lsub*: ('a, 'b) *rbt-node ref option*) (*cl*: *color*) (*key*: 'a) (*val*: 'b) (*rsub*: ('a, 'b) *rbt-node ref option*)
⟨*ML*⟩

fun *color-encode* :: *color* ⇒ *nat* **where**
  color-encode B = *0*
| *color-encode R* = *1*

instance *color* :: *heap*
  ⟨*proof*⟩

fun *rbt-node-encode* :: ('a::*heap*, 'b::*heap*) *rbt-node* ⇒ *nat* **where**
  rbt-node-encode (*Node l c k v r*) = *to-nat* (*l*, *c*, *k*, *v*, *r*)

instance *rbt-node* :: (*heap*, *heap*) *heap*
  ⟨*proof*⟩

fun *btree* :: ('a::*heap*, 'b::*heap*) *rbt* ⇒ ('a, 'b) *rbt-node ref option* ⇒ *assn* **where**
  btree Leaf p = ↑(*p* = *None*)
| *btree* (*rbt.Node lt c k v rt*) (*Some p*) = ($\exists_A$*lp rp*. *p* $\mapsto_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp*)
| *btree* (*rbt.Node lt c k v rt*) *None* = *false*
⟨*ML*⟩

lemma *btree-Leaf* [*forward-ent*]: *btree Leaf p* $\Longrightarrow_A$ ↑(*p* = *None*) ⟨*proof*⟩

lemma *btree-Node* [*forward-ent*]:
  btree (*rbt.Node lt c k v rt*) *p* $\Longrightarrow_A$ ($\exists_A$*lp rp*. *the p* $\mapsto_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ∗ ↑(*p* ≠ *None*))
⟨*proof*⟩

lemma *btree-none*: *emp* $\Longrightarrow_A$ *btree Leaf None* ⟨*proof*⟩

lemma *btree-constr-ent*:
  p $\mapsto_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp* $\Longrightarrow_A$ *btree* (*rbt.Node lt c k v rt*) (*Some p*) ⟨*proof*⟩

⟨*ML*⟩

type-synonym ('a, 'b) *btree* = ('a, 'b) *rbt-node ref option*

19.2 Operations

19.2.1 Basic operations

definition *tree-empty* :: ('a, 'b) *btree Heap* **where**
  tree-empty = *return None*

lemma *tree-empty-rule* [*hoare-triple*]:
<*emp*> *tree-empty* <*btree Leaf*> ⟨*proof*⟩

definition *tree-is-empty* :: ('*a*, '*b*) *btree* ⇒ *bool Heap* **where**
tree-is-empty b = *return* (*b* = *None*)

lemma *tree-is-empty-rule*:
<*btree t b*> *tree-is-empty b* <λ*r*. *btree t b* ∗ ↑(*r* ⟷ *t* = *Leaf*)> ⟨*proof*⟩

definition *btree-constr* ::
('*a*::*heap*, '*b*::*heap*) *btree* ⇒ *color* ⇒ '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *btree* ⇒ ('*a*, '*b*) *btree Heap*
where
btree-constr lp c k v rp = *do* { *p* ← *ref* (*Node lp c k v rp*); *return* (*Some p*) }

lemma *btree-constr-rule* [*hoare-triple*]:
<*btree lt lp* ∗ *btree rt rp*>
btree-constr lp c k v rp
<*btree* (*rbt.Node lt c k v rt*)> ⟨*proof*⟩

definition *set-color* :: *color* ⇒ ('*a*::*heap*, '*b*::*heap*) *btree* ⇒ *unit Heap* **where**
set-color c p = (*case p of*
None ⇒ *raise STR ''set-color''*
| *Some pp* ⇒ *do* {
t ← !*pp*;
pp := *Node* (*lsub t*) *c* (*key t*) (*val t*) (*rsub t*)
})

lemma *set-color-rule* [*hoare-triple*]:
<*btree* (*rbt.Node a c x v b*) *p*>
set-color c' p
<λ-. *btree* (*rbt.Node a c' x v b*) *p*> ⟨*proof*⟩

definition *get-color* :: ('*a*::*heap*, '*b*::*heap*) *btree* ⇒ *color Heap* **where**
get-color p = (*case p of*
None ⇒ *return B*
| *Some pp* ⇒ *do* {
t ← !*pp*;
return (*cl t*)
})

lemma *get-color-rule* [*hoare-triple*]:
<*btree t p*> *get-color p* <λ*r*. *btree t p* ∗ ↑(*r* = *rbt.cl t*)>
⟨*proof*⟩

definition *paint* :: *color* ⇒ ('*a*::*heap*, '*b*::*heap*) *btree* ⇒ *unit Heap* **where**
paint c p = (*case p of*
None ⇒ *return* ()
| *Some pp* ⇒ *do* {

```
    t ← !pp;
    pp := Node (lsub t) c (key t) (val t) (rsub t)
  })
```

```
lemma paint-rule [hoare-triple]:
  <btree t p>
  paint c p
  <λ-. btree (RBTree.paint c t) p>
⟨proof⟩
```

19.2.2 Rotation

```
definition btree-rotate-l :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-rotate-l p = (case p of
    None ⇒ raise STR ''Empty btree''
  | Some pp ⇒ do {
     t ← !pp;
     (case rsub t of
        None ⇒ raise STR ''Empty rsub''
      | Some rp ⇒ do {
          rt ← !rp;
          pp := Node (lsub t) (cl t) (key t) (val t) (lsub rt);
          rp := Node p (cl rt) (key rt) (val rt) (rsub rt);
          return (rsub t) })})
```

```
lemma btree-rotate-l-rule [hoare-triple]:
  <btree (rbt.Node a c1 x v (rbt.Node b c2 y w c)) p>
   btree-rotate-l p
   <btree (rbt.Node (rbt.Node a c1 x v b) c2 y w c)> ⟨proof⟩
```

```
definition btree-rotate-r :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-rotate-r p = (case p of
    None ⇒ raise STR ''Empty btree''
  | Some pp ⇒ do {
     t ← !pp;
     (case lsub t of
        None ⇒ raise STR ''Empty lsub''
      | Some lp ⇒ do {
          lt ← !lp;
          pp := Node (rsub lt) (cl t) (key t) (val t) (rsub t);
          lp := Node (lsub lt) (cl lt) (key lt) (val lt) p;
          return (lsub t) })})
```

```
lemma btree-rotate-r-rule [hoare-triple]:
  <btree (rbt.Node (rbt.Node a c1 x v b) c2 y w c) p>
   btree-rotate-r p
   <btree (rbt.Node a c1 x v (rbt.Node b c2 y w c))> ⟨proof⟩
```

19.2.3 Balance

```
definition btree-balanceR :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-balanceR p = (case p of None ⇒ return None | Some pp ⇒ do {
    t ← !pp;
    cl-r ← get-color (rsub t);
    if cl-r = R then do {
      rt ← !(the (rsub t));
      cl-lr ← get-color (lsub rt);
      cl-rr ← get-color (rsub rt);
      if cl-lr = R then do {
        rp' ← btree-rotate-r (rsub t);
        pp := Node (lsub t) (cl t) (key t) (val t) rp';
        p' ← btree-rotate-l p;
        t' ← !(the p');
        set-color B (rsub t');
        return p'
      } else if cl-rr = R then do {
        p' ← btree-rotate-l p;
        t' ← !(the p');
        set-color B (rsub t');
        return p'
       } else return p }
    else return p})
```

```
lemma balanceR-to-fun [hoare-triple]:
  <btree (rbt.Node l B k v r) p>
   btree-balanceR p
  <btree (balanceR l k v r)>
⟨proof⟩
```

```
definition btree-balance :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-balance p = (case p of None ⇒ return None | Some pp ⇒ do {
    t ← !pp;
    cl-l ← get-color (lsub t);
    if cl-l = R then do {
      lt ← !(the (lsub t));
      cl-rl ← get-color (rsub lt);
      cl-ll ← get-color (lsub lt);
      if cl-ll = R then do {
        p' ← btree-rotate-r p;
        t' ← !(the p');
        set-color B (lsub t');
        return p' }
      else if cl-rl = R then do {
        lp' ← btree-rotate-l (lsub t);
        pp := Node lp' (cl t) (key t) (val t) (rsub t);
        p' ← btree-rotate-r p;
        t' ← !(the p');
        set-color B (lsub t');
```

```
      return p′
    } else btree-balanceR p }
  else do {
    p′ ← btree-balanceR p;
    return p′}})
```

lemma *balance-to-fun* [*hoare-triple*]:

```
  <btree (rbt.Node l B k v r) p>
   btree-balance p
   <btree (balance l k v r)>
⟨proof⟩
```

19.2.4 Insertion

partial-function (*heap*) *rbt-ins* ::

```
  ′a::{heap,ord} ⇒ ′b::heap ⇒ (′a, ′b) btree ⇒ (′a, ′b) btree Heap where
  rbt-ins k v p = (case p of
     None ⇒ btree-constr None R k v None
   | Some pp ⇒ do {
      t ← !pp;
      (if cl t = B then
        (if k = key t then do {
           pp := Node (lsub t) (cl t) k v (rsub t);
           return (Some pp) }
         else if k < key t then do {
           q ← rbt-ins k v (lsub t);
           pp := Node q (cl t) (key t) (val t) (rsub t);
           btree-balance p }
         else do {
           q ← rbt-ins k v (rsub t);
           pp := Node (lsub t) (cl t) (key t) (val t) q;
           btree-balance p })
       else
        (if k = key t then do {
           pp := Node (lsub t) (cl t) k v (rsub t);
           return (Some pp) }
         else if k < key t then do {
           q ← rbt-ins k v (lsub t);
           pp := Node q (cl t) (key t) (val t) (rsub t);
           return (Some pp) }
         else do {
           q ← rbt-ins k v (rsub t);
           pp := Node (lsub t) (cl t) (key t) (val t) q;
           return (Some pp) }))})
```

lemma *rbt-ins-to-fun* [*hoare-triple*]:

```
  <btree t p>
   rbt-ins k v p
   <btree (ins k v t)>
```

```
⟨proof⟩

definition rbt-insert ::
  'a::{heap,ord} ⇒ 'b::heap ⇒ ('a, 'b) btree ⇒ ('a, 'b) btree Heap where
  rbt-insert k v p = do {
    p' ← rbt-ins k v p;
    paint B p';
    return p' }

lemma rbt-insert-to-fun [hoare-triple]:
  <btree t p>
   rbt-insert k v p
   <btree (RBTree.rbt-insert k v t)> ⟨proof⟩
```

19.2.5 Search

```
partial-function (heap) rbt-search ::
  'a::{heap,linorder} ⇒ ('a, 'b::heap) btree ⇒ 'b option Heap where
  rbt-search x b = (case b of
     None ⇒ return None
   | Some p ⇒ do {
      t ← !p;
      (if x = key t then return (Some (val t))
       else if x < key t then rbt-search x (lsub t)
       else rbt-search x (rsub t)) })

lemma btree-search-correct [hoare-triple]:
  <btree t b ∗ ↑(rbt-sorted t)>
   rbt-search x b
   <λr. btree t b ∗ ↑(r = RBTree.rbt-search t x)>
⟨proof⟩
```

19.2.6 Delete

```
definition btree-balL :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  btree-balL p = (case p of
     None ⇒ return None
   | Some pp ⇒ do {
      t ← !pp;
      cl-l ← get-color (lsub t);
      if cl-l = R then do {
        set-color B (lsub t);  — Case 1
        return p}
      else case rsub t of
        None ⇒ return p  — Case 2
      | Some rp ⇒ do {
         rt ← !rp;
         if cl rt = B then do {
           set-color R (rsub t);  — Case 3
           set-color B p;
```

```
        btree-balance p}
      else case lsub rt of
        None ⇒ return p  — Case 4
      | Some lrp ⇒ do {
          lrt ← !lrp;
          if cl lrt = B then do {
            set-color R (lsub rt);  — Case 5
            paint R (rsub rt);
            set-color B (rsub t);
            rp′ ← btree-rotate-r (rsub t);
            pp := Node (lsub t) (cl t) (key t) (val t) rp′;
            p′ ← btree-rotate-l p;
            t′ ← !(the p′);
            set-color B (lsub t′);
            rp′′ ← btree-balance (rsub t′);
            the p′ := Node (lsub t′) (cl t′) (key t′) (val t′) rp′′;
            return p′}
          else return p}}})
```

lemma *balL-to-fun* [*hoare-triple*]:

```
  <btree (rbt.Node l R k v r) p>
   btree-balL p
   <btree (balL l k v r)>
⟨proof⟩
```

definition *btree-balR* :: ('a::heap, 'b::heap) btree ⇒ ('a, 'b) btree Heap **where**

```
  btree-balR p = (case p of
     None ⇒ return None
   | Some pp ⇒ do {
      t ← !pp;
      cl-r ← get-color (rsub t);
      if cl-r = R then do {
        set-color B (rsub t);  — Case 1
        return p}
      else case lsub t of
        None ⇒ return p  — Case 2
      | Some lp ⇒ do {
         lt ← !lp;
         if cl lt = B then do {
           set-color R (lsub t);  — Case 3
           set-color B p;
           btree-balance p}
         else case rsub lt of
           None ⇒ return p  — Case 4
         | Some rlp ⇒ do {
            rlt ← !rlp;
            if cl rlt = B then do {
              set-color R (rsub lt);  — Case 5
              paint R (lsub lt);
```

```
            set-color B (lsub t);
            lp′ ← btree-rotate-l (lsub t);
            pp := Node lp′ (cl t) (key t) (val t) (rsub t);
            p′ ← btree-rotate-r p;
            t′ ← !(the p′);
            set-color B (rsub t′);
            lp′′ ← btree-balance (lsub t′);
            the p′ := Node lp′′ (cl t′) (key t′) (val t′) (rsub t′);
            return p′}
          else return p}}})

lemma balR-to-fun [hoare-triple]:
  <btree (rbt.Node l R k v r) p>
   btree-balR p
   <btree (balR l k v r)>
⟨proof⟩

partial-function (heap) btree-combine ::
  (′a::heap, ′b::heap) btree ⇒ (′a, ′b) btree ⇒ (′a, ′b) btree Heap where
  btree-combine lp rp =
   (if lp = None then return rp
    else if rp = None then return lp
    else do {
      lt ← !(the lp);
      rt ← !(the rp);
      if cl lt = R then
        if cl rt = R then do {
          tmp ← btree-combine (rsub lt) (lsub rt);
          cl-tm ← get-color tmp;
          if cl-tm = R then do {
            tmt ← !(the tmp);
            the lp := Node (lsub lt) R (key lt) (val lt) (lsub tmt);
            the rp := Node (rsub tmt) R (key rt) (val rt) (rsub rt);
            the tmp := Node lp R (key tmt) (val tmt) rp;
            return tmp}
          else do {
            the rp := Node tmp R (key rt) (val rt) (rsub rt);
            the lp := Node (lsub lt) R (key lt) (val lt) rp;
            return lp}}
        else do {
          tmp ← btree-combine (rsub lt) rp;
          the lp := Node (lsub lt) R (key lt) (val lt) tmp;
          return lp}
      else if cl rt = B then do {
        tmp ← btree-combine (rsub lt) (lsub rt);
        cl-tm ← get-color tmp;
        if cl-tm = R then do {
          tmt ← !(the tmp);
          the lp := Node (lsub lt) B (key lt) (val lt) (lsub tmt);
```

```
        the rp := Node (rsub tmt) B (key rt) (val rt) (rsub rt);
        the tmp := Node lp R (key tmt) (val tmt) rp;
        return tmp}
      else do {
        the rp := Node tmp B (key rt) (val rt) (rsub rt);
        the lp := Node (lsub lt) R (key lt) (val lt) rp;
        btree-balL lp}}
    else do {
      tmp ← btree-combine lp (lsub rt);
      the rp := Node tmp R (key rt) (val rt) (rsub rt);
      return rp}})
```

lemma *combine-to-fun* [*hoare-triple*]:

```
  <btree lt lp * btree rt rp>
   btree-combine lp rp
   <btree (combine lt rt)>
⟨proof⟩
```

partial-function (*heap*) *rbt-del* ::

```
  'a::{heap,linorder} ⇒ ('a, 'b::heap) btree ⇒ ('a, 'b) btree Heap where
  rbt-del x p = (case p of
     None ⇒ return None
   | Some pp ⇒ do {
      t ← !pp;
      (if x = key t then btree-combine (lsub t) (rsub t)
       else if x < key t then case lsub t of
         None ⇒ do {
           set-color R p;
           return p}
       | Some lp ⇒ do {
           lt ← !lp;
           if cl lt = B then do {
             q ← rbt-del x (lsub t);
             pp := Node q R (key t) (val t) (rsub t);
             btree-balL p }
           else do {
             q ← rbt-del x (lsub t);
             pp := Node q R (key t) (val t) (rsub t);
             return p }}
       else case rsub t of
         None ⇒ do {
           set-color R p;
           return p}
       | Some rp ⇒ do {
           rt ← !rp;
           if cl rt = B then do {
             q ← rbt-del x (rsub t);
             pp := Node (lsub t) R (key t) (val t) q;
             btree-balR p }
```

else do {
  q ← *rbt-del x* (*rsub t*);
  pp := *Node* (*lsub t*) *R* (*key t*) (*val t*) *q*;
  return p }})})

lemma *rbt-del-to-fun* [*hoare-triple*]:
  <*btree t p*>
  rbt-del x p
  <*btree* (*del x t*)>$_t$
⟨*proof*⟩

definition *rbt-delete* ::
  'a::{*heap*,*linorder*} ⇒ (*'a*, *'b*::*heap*) *btree* ⇒ (*'a*, *'b*) *btree Heap* **where**
  rbt-delete k p = *do* {
    p' ← *rbt-del k p*;
    paint B p';
    return p'}

lemma *rbt-delete-to-fun* [*hoare-triple*]:
  <*btree t p*>
  rbt-delete k p
  <*btree* (*RBTree.delete k t*)>$_t$ ⟨*proof*⟩

19.3 Outer interface

Express Hoare triples for operations on red-black tree in terms of the mapping represented by the tree.

definition *rbt-map-assn* :: (*'a*, *'b*) *map* ⇒ (*'a*::{*heap*,*linorder*}, *'b*::*heap*) *rbt-node ref option* ⇒ *assn* **where**
  rbt-map-assn M p = (∃$_A$*t*. *btree t p* ∗ ↑(*is-rbt t*) ∗ ↑(*rbt-sorted t*) ∗ ↑(*M* = *rbt-map t*))
⟨*ML*⟩

theorem *rbt-empty-rule* [*hoare-triple*]:
  <*emp*> *tree-empty* <*rbt-map-assn empty-map*> ⟨*proof*⟩

theorem *rbt-insert-rule* [*hoare-triple*]:
  <*rbt-map-assn M b*> *rbt-insert k v b* <*rbt-map-assn* (*M* {*k* → *v*})> ⟨*proof*⟩

theorem *rbt-search* [*hoare-triple*]:
  <*rbt-map-assn M b*> *rbt-search x b* <λ*r*. *rbt-map-assn M b* ∗ ↑(*r* = *M*⟨*x*⟩)> ⟨*proof*⟩

theorem *rbt-delete-rule* [*hoare-triple*]:
  <*rbt-map-assn M b*> *rbt-delete k b* <*rbt-map-assn* (*delete-map k M*)>$_t$ ⟨*proof*⟩

end

20 Implementation of arrays

```
theory Arrays-Impl
  imports SepAuto ../Functional/Arrays-Ex
begin
```

Imperative implementations of common array operations.

Imperative reverse on arrays is also verified in theory Imperative_Reverse in Imperative_HOL/ex in the Isabelle library.

20.1 Array copy

```
fun array-copy :: 'a::heap array ⇒ 'a array ⇒ nat ⇒ unit Heap where
  array-copy a b 0 = (return ())
| array-copy a b (Suc n) = do {
      array-copy a b n;
      x ← Array.nth a n;
      Array.upd n x b;
      return () }

lemma array-copy-rule [hoare-triple]:
  n ≤ length as ⟹ n ≤ length bs ⟹
   <a ↦a as ∗ b ↦a bs>
    array-copy a b n
   <λ-. a ↦a as ∗ b ↦a Arrays-Ex.array-copy as bs n>
⟨proof⟩
```

20.2 Swap

```
definition swap :: 'a::heap array ⇒ nat ⇒ nat ⇒ unit Heap where
  swap a i j = do {
     x ← Array.nth a i;
     y ← Array.nth a j;
     Array.upd i y a;
     Array.upd j x a;
     return ()
   }

lemma swap-rule [hoare-triple]:
  i < length xs ⟹ j < length xs ⟹
   <p ↦a xs>
   swap p i j
   <λ-. p ↦a list-swap xs i j> ⟨proof⟩
```

20.3 Reverse

```
fun rev :: 'a::heap array ⇒ nat ⇒ nat ⇒ unit Heap where
  rev a i j = (if i < j then do {
     swap a i j;
```

```
    rev a (i + 1) (j − 1)
  }
  else return ())
```

```
lemma rev-to-fun [hoare-triple]:
  j < length xs ⟹
   <p ↦a xs>
   rev p i j
   <λ-. p ↦a rev-swap xs i j>
⟨proof⟩
```

Correctness of imperative reverse.

```
theorem rev-is-rev [hoare-triple]:
  xs ≠ [] ⟹
   <p ↦a xs>
   rev p 0 (length xs − 1)
   <λ-. p ↦a List.rev xs> ⟨proof⟩
```

end

21 Implementation of quicksort

```
theory Quicksort-Impl
  imports Arrays-Impl ../Functional/Quicksort
begin
```

Imperative implementation of quicksort. Also verified in theory Imperative_Quicksort in HOL/Imperative_HOL/ex in the Isabelle library.

```
partial-function (heap) part1 :: 'a::{heap,linorder} array ⇒ nat ⇒ nat ⇒ 'a ⇒
nat Heap where
  part1 a l r p = (
     if r ≤ l then return r
     else do {
       v ← Array.nth a l;
       if v ≤ p then
         part1 a (l + 1) r p
       else do {
         swap a l r;
         part1 a l (r − 1) p }})
```

```
lemma part1-to-fun [hoare-triple]:
  r < length xs ⟹ <p ↦a xs>
   part1 p l r a
   <λrs. p ↦a snd (Quicksort.part1 xs l r a) ∗ ↑(rs = fst (Quicksort.part1 xs l r
a))>
⟨proof⟩
```

Partition function

definition *partition* :: *'a*::{*heap*,*linorder*} *array* ⇒ *nat* ⇒ *nat* ⇒ *nat Heap* **where**
  partition a l r = *do* {
    p ← *Array.nth a r*;
    m ← *part1 a l* (*r* − *1*) *p*;
    v ← *Array.nth a m*;
    m' ← *return* (*if v* ≤ *p then m* + *1 else m*);
    swap a m' r;
    return m'
  }

lemma *partition-to-fun* [*hoare-triple*]:
  l < *r* ⟹ *r* < *length xs* ⟹ <*a* $\mapsto_a$ *xs*>
  partition a l r
  <λ*rs. a* $\mapsto_a$ *snd* (*Quicksort.partition xs l r*) ∗ ↑(*rs* = *fst* (*Quicksort.partition xs l r*))>
⟨*proof*⟩

Quicksort function

partial-function (*heap*) *quicksort* :: *'a*::{*heap*,*linorder*} *array* ⇒ *nat* ⇒ *nat* ⇒ *unit Heap* **where**
  quicksort a l r = *do* {
    len ← *Array.len a*;
    if l ≥ *r then return* ()
    else if r < *len then do* {
      p ← *partition a l r*;
      quicksort a l (*p* − *1*);
      quicksort a (*p* + *1*) *r*
    }
    else return ()
  }

lemma *quicksort-to-fun* [*hoare-triple*]:
  r < *length xs* ⟹ <*a* $\mapsto_a$ *xs*>
  quicksort a l r
  <λ-. *a* $\mapsto_a$ *Quicksort.quicksort xs l r*>
⟨*proof*⟩

definition *quicksort-all* :: (*'a*::{*heap*,*linorder*}) *array* ⇒ *unit Heap* **where**
  quicksort-all a = *do* {
    n ← *Array.len a*;
    if n = *0 then return* ()
    else quicksort a 0 (*n* − *1*)
  }

Correctness of quicksort.

theorem *quicksort-sorts-basic* [*hoare-triple*]:
  <*a* $\mapsto_a$ *xs*>
  quicksort-all a
  <λ-. *a* $\mapsto_a$ *sort xs*> ⟨*proof*⟩

end

22 Implementation of union find

```
theory Union-Find-Impl
  imports SepAuto ../Functional/Union-Find
begin
```

Development follows theory Union_Find in [5] by Lammich and Meis.

```
type-synonym uf = nat array × nat array

definition is-uf :: nat ⇒ (nat×nat) set ⇒ uf ⇒ assn where [rewrite-ent]:
  is-uf n R u = (∃A l szl. snd u ↦a l ∗ fst u ↦a szl ∗
      ↑(ufa-invar l) ∗ ↑(ufa-α l = R) ∗ ↑(length l = n) ∗ ↑(length szl = n))

definition uf-init :: nat ⇒ uf Heap where
  uf-init n = do {
    l ← Array.of-list [0..<n];
    szl ← Array.new n (1::nat);
    return (szl, l)
  }
```

Correctness of uf_init.

```
theorem uf-init-rule [hoare-triple]:
  <emp> uf-init n <is-uf n (uf-init-rel n)> ⟨proof⟩

partial-function (heap) uf-rep-of :: nat array ⇒ nat ⇒ nat Heap where
  uf-rep-of p i = do {
    n ← Array.nth p i;
    if n = i then return i else uf-rep-of p n
  }

lemma uf-rep-of-rule [hoare-triple]:
  ufa-invar l ⟹ i < length l ⟹
   <p ↦a l>
   uf-rep-of p i
   <λr. p ↦a l ∗ ↑(r = rep-of l i)>
⟨proof⟩

partial-function (heap) uf-compress :: nat ⇒ nat ⇒ nat array ⇒ unit Heap
where
  uf-compress i ci p = (
    if i = ci then return ()
    else do {
      ni ← Array.nth p i;
      uf-compress ni ci p;
      Array.upd i ci p;
```

```
    return ()
  })
```

```
lemma uf-compress-rule [hoare-triple]:
  ufa-invar l ⟹ i < length l ⟹
   <p ↦_a l>
   uf-compress i (rep-of l i) p
   <λ-. ∃_A l'. p ↦_a l' * ↑(ufa-invar l' ∧ length l' = length l ∧
            (∀ i<length l. rep-of l' i = rep-of l i))>
⟨proof⟩
```

```
definition uf-rep-of-c :: nat array ⇒ nat ⇒ nat Heap where
  uf-rep-of-c p i = do {
    ci ← uf-rep-of p i;
    uf-compress i ci p;
    return ci
  }
```

```
lemma uf-rep-of-c-rule [hoare-triple]:
  ufa-invar l ⟹ i < length l ⟹
   <p ↦_a l>
   uf-rep-of-c p i
   <λr. ∃_A l'. p ↦_a l' * ↑(r = rep-of l i ∧ ufa-invar l' ∧ length l' = length l ∧
                        (∀ i<length l. rep-of l' i = rep-of l i))>
  ⟨proof⟩
```

```
definition uf-cmp :: uf ⇒ nat ⇒ nat ⇒ bool Heap where
  uf-cmp u i j = do {
    n ← Array.len (snd u);
    if (i≥n ∨ j≥n) then return False
    else do {
      ci ← uf-rep-of-c (snd u) i;
      cj ← uf-rep-of-c (snd u) j;
      return (ci = cj)
    }
  }
```

Correctness of compare.

```
theorem uf-cmp-rule [hoare-triple]:
  <is-uf n R u>
   uf-cmp u i j
   <λr. is-uf n R u * ↑(r ⟷ (i,j)∈R)> ⟨proof⟩
```

```
definition uf-union :: uf ⇒ nat ⇒ nat ⇒ uf Heap where
  uf-union u i j = do {
    ci ← uf-rep-of (snd u) i;
    cj ← uf-rep-of (snd u) j;
    if (ci = cj) then return u
    else do {
```

```
      si ← Array.nth (fst u) ci;
      sj ← Array.nth (fst u) cj;
      if si < sj then do {
        Array.upd ci cj (snd u);
        Array.upd cj (si+sj) (fst u);
        return u
      } else do {
        Array.upd cj ci (snd u);
        Array.upd ci (si+sj) (fst u);
        return u
      }
    }
  }
```

Correctness of union.

```
theorem uf-union-rule [hoare-triple]:
  i < n ⟹ j < n ⟹
   <is-uf n R u>
   uf-union u i j
   <is-uf n (per-union R i j)> ⟨proof⟩

⟨ML⟩

end
```

23 Implementation of connectivity on graphs

```
theory Connectivity-Impl
  imports Union-Find-Impl ../Functional/Connectivity
begin
```

Imperative version of graph-connectivity example.

23.1 Constructing the connected relation

```
fun connected-rel-imp :: nat ⇒ (nat × nat) list ⇒ nat ⇒ uf Heap where
  connected-rel-imp n es 0 = do { p ← uf-init n; return p }
| connected-rel-imp n es (Suc k) = do {
    p ← connected-rel-imp n es k;
    p' ← uf-union p (fst (es ! k)) (snd (es ! k));
    return p' }

lemma connected-rel-imp-to-fun [hoare-triple]:
  is-valid-graph n (set es) ⟹ k ≤ length es ⟹
   <emp>
   connected-rel-imp n es k
   <is-uf n (connected-rel-ind n es k)>
⟨proof⟩
```

lemma *connected-rel-imp-correct* [*hoare-triple*]:
is-valid-graph n (*set es*) ⟹
<*emp*>
connected-rel-imp n es (*length es*)
<*is-uf n* (*connected-rel n* (*set es*))> ⟨*proof*⟩

23.2 Connectedness tests

Correctness of the algorithm for detecting connectivity.

theorem *uf-cmp-correct* [*hoare-triple*]:
<*is-uf n* (*connected-rel n S*) *p*>
uf-cmp p i j
<λ*r*. *is-uf n* (*connected-rel n S*) *p* ∗ ↑(*r* ⟷ *has-path n S i j*)> ⟨*proof*⟩

end

24 Implementation of dynamic arrays

theory *DynamicArray*
imports *Arrays-Impl*
begin

Dynamically allocated arrays.

datatype *′a dynamic-array* = *Dyn-Array* (*alen*: *nat*) (*aref*: *′a array*)
⟨*ML*⟩

24.1 Raw assertion

fun *dyn-array-raw* :: *′a*::*heap list* × *nat* ⇒ *′a dynamic-array* ⇒ *assn* **where**
dyn-array-raw (*xs*, *n*) (*Dyn-Array m a*) = (*a* $\mapsto_a$ *xs* ∗ ↑(*m* = *n*))
⟨*ML*⟩

definition *dyn-array-new* :: *′a*::*heap dynamic-array Heap* **where**
dyn-array-new = *do* {
p ← *Array.new 5 undefined*;
return (*Dyn-Array 0 p*)
}

lemma *dyn-array-new-rule′* [*hoare-triple*]:
<*emp*>
dyn-array-new
<*dyn-array-raw* (*replicate 5 undefined*, *0*)> ⟨*proof*⟩

fun *double-length* :: *′a*::*heap dynamic-array* ⇒ *′a dynamic-array Heap* **where**
double-length (*Dyn-Array al ar*) = *do* {
am ← *Array.len ar*;
p ← *Array.new* (*2* ∗ *am* + *1*) *undefined*;

array-copy ar p am;
return (*Dyn-Array am p*)
}

fun *double-length-fun* :: 'a::heap list × nat ⇒ 'a list × nat **where**
double-length-fun (*xs*, *n*) =
(*Arrays-Ex.array-copy xs* (*replicate* (2 ∗ n + 1) *undefined*) *n*, *n*)
⟨*ML*⟩

lemma *double-length-rule'* [*hoare-triple*]:
length xs = *n* ⟹
<*dyn-array-raw* (*xs*, *n*) *p*>
double-length p
<*dyn-array-raw* (*double-length-fun* (*xs*, *n*))>$_t$ ⟨*proof*⟩

fun *push-array-basic* :: 'a ⇒ 'a::heap dynamic-array ⇒ 'a dynamic-array Heap
where
push-array-basic x (*Dyn-Array al ar*) = *do* {
Array.upd al x ar;
return (*Dyn-Array* (*al* + 1) *ar*)
}

fun *push-array-basic-fun* :: 'a ⇒ 'a::heap list × nat ⇒ 'a list × nat **where**
push-array-basic-fun x (*xs*, *n*) = (*list-update xs n x*, *n* + 1)
⟨*ML*⟩

lemma *push-array-basic-rule'* [*hoare-triple*]:
n < *length xs* ⟹
<*dyn-array-raw* (*xs*, *n*) *p*>
push-array-basic x p
<*dyn-array-raw* (*push-array-basic-fun x* (*xs*, *n*))> ⟨*proof*⟩

definition *array-length* :: 'a dynamic-array ⇒ nat Heap **where**
array-length d = *return* (*alen d*)

lemma *array-length-rule'* [*hoare-triple*]:
<*dyn-array-raw* (*xs*, *n*) *p*>
array-length p
<λ*r*. *dyn-array-raw* (*xs*, *n*) *p* ∗ ↑(*r* = *n*)> ⟨*proof*⟩

definition *array-max* :: 'a::heap dynamic-array ⇒ nat Heap **where**
array-max d = *Array.len* (*aref d*)

lemma *array-max-rule'* [*hoare-triple*]:
<*dyn-array-raw* (*xs*, *n*) *p*>
array-max p
<λ*r*. *dyn-array-raw* (*xs*, *n*) *p* ∗ ↑(*r* = *length xs*)> ⟨*proof*⟩

definition *array-nth* :: 'a::heap dynamic-array ⇒ nat ⇒ 'a Heap **where**

array-nth d i = Array.nth (aref d) i

lemma *array-nth-rule′* [*hoare-triple*]:
i < n ⟹ n ≤ length xs ⟹
<dyn-array-raw (xs, n) p>
array-nth p i
<λr. dyn-array-raw (xs, n) p ∗ ↑(r = xs ! i)> ⟨*proof*⟩

definition *array-upd* :: *nat ⇒ ′a ⇒ ′a::heap dynamic-array ⇒ unit Heap* **where**
array-upd i x d = do { Array.upd i x (aref d); return () }

lemma *array-upd-rule′* [*hoare-triple*]:
i < n ⟹ n ≤ length xs ⟹
<dyn-array-raw (xs, n) p>
array-upd i x p
<λ-. dyn-array-raw (list-update xs i x, n) p> ⟨*proof*⟩

definition *push-array* :: *′a ⇒ ′a::heap dynamic-array ⇒ ′a dynamic-array Heap*
where
push-array x p = do {
m ← array-max p;
l ← array-length p;
if l < m then push-array-basic x p
else do {
u ← double-length p;
push-array-basic x u
}
}

definition *pop-array* :: *′a::heap dynamic-array ⇒ (′a × ′a dynamic-array) Heap*
where
pop-array d = do {
x ← Array.nth (aref d) (alen d − 1);
return (x, Dyn-Array (alen d − 1) (aref d))
}

lemma *pop-array-rule′* [*hoare-triple*]:
n > 0 ⟹ n ≤ length xs ⟹
<dyn-array-raw (xs, n) p>
pop-array p
<λ(x, r). dyn-array-raw (xs, n − 1) r ∗ ↑(x = xs ! (n − 1))> ⟨*proof*⟩

⟨*ML*⟩

fun *push-array-fun* :: *′a ⇒ ′a::heap list × nat ⇒ ′a list × nat* **where**
push-array-fun x (xs, n) = (
if n < length xs then push-array-basic-fun x (xs, n)
else push-array-basic-fun x (double-length-fun (xs, n)))
⟨*ML*⟩

lemma *push-array-rule'* [*hoare-triple*]:
n ≤ *length xs* ⟹
<*dyn-array-raw* (*xs*, *n*) *p*>
push-array x p
<*dyn-array-raw* (*push-array-fun x* (*xs*, *n*))>$_t$ ⟨*proof*⟩

24.2 Abstract assertion

fun *abs-array* :: *'a::heap list* × *nat* ⇒ *'a list* **where**
abs-array (*xs*, *n*) = *take n xs*
⟨*ML*⟩

lemma *double-length-abs* [*rewrite*]:
length xs = *n* ⟹ *abs-array* (*double-length-fun* (*xs*, *n*)) = *abs-array* (*xs*, *n*) ⟨*proof*⟩

lemma *push-array-basic-abs* [*rewrite*]:
n < *length xs* ⟹ *abs-array* (*push-array-basic-fun x* (*xs*, *n*)) = *abs-array* (*xs*, *n*) @ [*x*]
⟨*proof*⟩

lemma *push-array-fun-abs* [*rewrite*]:
n ≤ *length xs* ⟹ *abs-array* (*push-array-fun x* (*xs*, *n*)) = *abs-array* (*xs*, *n*) @ [*x*]
⟨*proof*⟩

definition *dyn-array* :: *'a::heap list* ⇒ *'a dynamic-array* ⇒ *assn* **where** [*rewrite-ent*]:
dyn-array xs a = (∃$_A$*p*. *dyn-array-raw p a* ∗ ↑(*xs* = *abs-array p*) ∗ ↑(*snd p* ≤ *length* (*fst p*)))

lemma *dyn-array-new-rule* [*hoare-triple*]:
<*emp*> *dyn-array-new* <*dyn-array* []> ⟨*proof*⟩

lemma *array-length-rule* [*hoare-triple*]:
<*dyn-array xs p*>
array-length p
<λ*r*. *dyn-array xs p* ∗ ↑(*r* = *length xs*)> ⟨*proof*⟩

lemma *array-nth-rule* [*hoare-triple*]:
i < *length xs* ⟹
<*dyn-array xs p*>
array-nth p i
<λ*r*. *dyn-array xs p* ∗ ↑(*r* = *xs* ! *i*)> ⟨*proof*⟩

lemma *array-upd-rule* [*hoare-triple*]:
i < *length xs* ⟹
<*dyn-array xs p*>
array-upd i x p
<λ-. *dyn-array* (*list-update xs i x*) *p*> ⟨*proof*⟩

lemma *push-array-rule* [*hoare-triple*]:
<*dyn-array xs p*>
push-array x p
<*dyn-array* (*xs* @ [*x*])>$_t$ ⟨*proof*⟩

lemma *pop-array-rule* [*hoare-triple*]:
xs ≠ [] ⟹
<*dyn-array xs p*>
pop-array p
<λ(*x*, *r*). *dyn-array* (*butlast xs*) *r* ∗ ↑(*x* = *last xs*)>
⟨*proof*⟩

⟨*ML*⟩

24.3 Derived operations

definition *array-swap* :: *'a*::*heap dynamic-array* ⇒ *nat* ⇒ *nat* ⇒ *unit Heap* **where**
array-swap d i j = *do* {
x ← *array-nth d i*;
y ← *array-nth d j*;
array-upd i y d;
array-upd j x d;
return ()
}

lemma *array-swap-rule* [*hoare-triple*]:
i < *length xs* ⟹ *j* < *length xs* ⟹
<*dyn-array xs p*>
array-swap p i j
<λ-. *dyn-array* (*list-swap xs i j*) *p*> ⟨*proof*⟩

end

25 Implementation of the indexed priority queue

theory *Indexed-PQueue-Impl*
imports *DynamicArray ../Functional/Indexed-PQueue*
begin

Imperative implementation of indexed priority queue. The data structure is also verified in [4] by Peter Lammich.

datatype *'a indexed-pqueue* =
Indexed-PQueue (*pqueue*: (*nat* × *'a*) *dynamic-array*) (*index*: *nat option array*)
⟨*ML*⟩

fun *idx-pqueue* :: *'a*::*heap idx-pqueue* ⇒ *'a indexed-pqueue* ⇒ *assn* **where**
idx-pqueue (*xs*, *m*) (*Indexed-PQueue pq idx*) = (*dyn-array xs pq* ∗ *idx* $\mapsto_a$ *m*)
⟨*ML*⟩

25.1 Basic operations

definition *idx-pqueue-empty* :: *nat* ⇒ *'a::heap indexed-pqueue Heap* **where**
idx-pqueue-empty k = *do* {
pq ← *dyn-array-new*;
idx ← *Array.new k None*;
return (*Indexed-PQueue pq idx*) }

lemma *idx-pqueue-empty-rule* [*hoare-triple*]:
<*emp*>
idx-pqueue-empty n
<*idx-pqueue* ([], *replicate n None*)> ⟨*proof*⟩

definition *idx-pqueue-nth* :: *'a::heap indexed-pqueue* ⇒ *nat* ⇒ (*nat* × *'a*) *Heap*
where
idx-pqueue-nth p i = *array-nth* (*pqueue p*) *i*

lemma *idx-pqueue-nth-rule* [*hoare-triple*]:
<*idx-pqueue* (*xs*, *m*) *p* ∗ ↑(*i* < *length xs*)>
idx-pqueue-nth p i
<λ*r*. *idx-pqueue* (*xs*, *m*) *p* ∗ ↑(*r* = *xs* ! *i*)> ⟨*proof*⟩

definition *idx-nth* :: *'a::heap indexed-pqueue* ⇒ *nat* ⇒ *nat option Heap* **where**
idx-nth p i = *Array.nth* (*index p*) *i*

lemma *idx-nth-rule* [*hoare-triple*]:
<*idx-pqueue* (*xs*, *m*) *p* ∗ ↑(*i* < *length m*)>
idx-nth p i
<λ*r*. *idx-pqueue* (*xs*, *m*) *p* ∗ ↑(*r* = *m* ! *i*)> ⟨*proof*⟩

definition *idx-pqueue-length* :: *'a indexed-pqueue* ⇒ *nat Heap* **where**
idx-pqueue-length a = *array-length* (*pqueue a*)

lemma *idx-pqueue-length-rule* [*hoare-triple*]:
<*idx-pqueue* (*xs*, *m*) *p*>
idx-pqueue-length p
<λ*r*. *idx-pqueue* (*xs*, *m*) *p* ∗ ↑(*r* = *length xs*)> ⟨*proof*⟩

definition *idx-pqueue-swap* ::
'a::{heap,linorder} indexed-pqueue ⇒ *nat* ⇒ *nat* ⇒ *unit Heap* **where**
idx-pqueue-swap p i j = *do* {
pr-i ← *array-nth* (*pqueue p*) *i*;
pr-j ← *array-nth* (*pqueue p*) *j*;
Array.upd (*fst pr-i*) (*Some j*) (*index p*);
Array.upd (*fst pr-j*) (*Some i*) (*index p*);
array-swap (*pqueue p*) *i j*
}

lemma *idx-pqueue-swap-rule* [*hoare-triple*]:
i < *length xs* ⟹ *j* < *length xs* ⟹ *index-of-pqueue* (*xs*, *m*) ⟹

<idx-pqueue (xs, m) p>
idx-pqueue-swap p i j
<λ-. idx-pqueue (idx-pqueue-swap-fun (xs, m) i j) p>
⟨proof⟩

definition idx-pqueue-push :: nat ⇒ 'a::heap ⇒ 'a indexed-pqueue ⇒ 'a indexed-pqueue Heap **where**
idx-pqueue-push k v p = do {
len ← array-length (pqueue p);
d' ← push-array (k, v) (pqueue p);
Array.upd k (Some len) (index p);
return (Indexed-PQueue d' (index p))
}

lemma idx-pqueue-push-rule [hoare-triple]:
k < length m ⟹ ¬has-key-alist xs k ⟹
<idx-pqueue (xs, m) p>
idx-pqueue-push k v p
<idx-pqueue (idx-pqueue-push-fun k v (xs, m))>$_t$
⟨proof⟩

definition idx-pqueue-pop :: 'a::heap indexed-pqueue ⇒ ((nat × 'a) × 'a indexed-pqueue) Heap **where**
idx-pqueue-pop p = do {
(x, d') ← pop-array (pqueue p);
Array.upd (fst x) None (index p);
return (x, Indexed-PQueue d' (index p))
}

lemma idx-pqueue-pop-rule [hoare-triple]:
xs ≠ [] ⟹ index-of-pqueue (xs, m) ⟹
<idx-pqueue (xs, m) p>
idx-pqueue-pop p
<λ(x, r). idx-pqueue (idx-pqueue-pop-fun (xs, m)) r ∗ ↑(x = last xs)>
⟨proof⟩

definition idx-pqueue-array-upd :: nat ⇒ 'a ⇒ 'a::heap dynamic-array ⇒ unit Heap **where**
idx-pqueue-array-upd i x d = array-upd i x d

lemma array-upd-idx-pqueue-rule [hoare-triple]:
i < length xs ⟹ k = fst (xs ! i) ⟹
<idx-pqueue (xs, m) p>
idx-pqueue-array-upd i (k, v) (pqueue p)
<λ-. idx-pqueue (list-update xs i (k, v), m) p> ⟨proof⟩

definition has-key-idx-pqueue :: nat ⇒ 'a::{heap,linorder} indexed-pqueue ⇒ bool Heap **where**
has-key-idx-pqueue k p = do {

```
    i-opt ← Array.nth (index p) k;
    return (i-opt ≠ None) }
```

lemma *has-key-idx-pqueue-rule* [*hoare-triple*]:

```
  k < length m ⟹ index-of-pqueue (xs, m) ⟹
    <idx-pqueue (xs, m) p>
    has-key-idx-pqueue k p
    <λr. idx-pqueue (xs, m) p ∗ ↑(r ⟷ has-key-alist xs k)> ⟨proof⟩
```

⟨*ML*⟩

25.2 Bubble up and down

partial-function (*heap*) *idx-bubble-down* :: 'a::{*heap*,*linorder*} *indexed-pqueue* ⇒ *nat* ⇒ *unit Heap* **where**

```
  idx-bubble-down a k = do {
    len ← idx-pqueue-length a;
    (if s2 k < len then do {
      vk ← idx-pqueue-nth a k;
      vs1k ← idx-pqueue-nth a (s1 k);
      vs2k ← idx-pqueue-nth a (s2 k);
      (if snd vs1k ≤ snd vs2k then
         if snd vk > snd vs1k then
           do { idx-pqueue-swap a k (s1 k); idx-bubble-down a (s1 k) }
         else return ()
       else
         if snd vk > snd vs2k then
           do { idx-pqueue-swap a k (s2 k); idx-bubble-down a (s2 k) }
         else return ()) }
     else if s1 k < len then do {
       vk ← idx-pqueue-nth a k;
       vs1k ← idx-pqueue-nth a (s1 k);
       (if snd vk > snd vs1k then
          do { idx-pqueue-swap a k (s1 k); idx-bubble-down a (s1 k) }
        else return ()) }
     else return ()) }
```

lemma *idx-bubble-down-rule* [*hoare-triple*]:

```
  index-of-pqueue x ⟹
    <idx-pqueue x a>
    idx-bubble-down a k
    <λ-. idx-pqueue (idx-bubble-down-fun x k) a>
⟨proof⟩
```

partial-function (*heap*) *idx-bubble-up* :: 'a::{*heap*,*linorder*} *indexed-pqueue* ⇒ *nat* ⇒ *unit Heap* **where**

```
  idx-bubble-up a k =
    (if k = 0 then return () else do {
       len ← idx-pqueue-length a;
```

```
    (if k < len then do {
      vk ← idx-pqueue-nth a k;
      vpk ← idx-pqueue-nth a (par k);
      (if snd vk < snd vpk then
         do { idx-pqueue-swap a k (par k); idx-bubble-up a (par k) }
       else return ()) }
     else return ())})
```

```
lemma idx-bubble-up-rule [hoare-triple]:
  index-of-pqueue x ⟹
   <idx-pqueue x a>
   idx-bubble-up a k
   <λ-. idx-pqueue (idx-bubble-up-fun x k) a>
⟨proof⟩
```

25.3 Main operations

```
definition delete-min-idx-pqueue :: 'a::{heap,linorder} indexed-pqueue ⇒ ((nat ×
'a) × 'a indexed-pqueue) Heap where
  delete-min-idx-pqueue p = do {
     len ← idx-pqueue-length p;
     if len = 0 then raise STR ''delete-min''
     else do {
       idx-pqueue-swap p 0 (len − 1);
       (x', r) ← idx-pqueue-pop p;
       idx-bubble-down r 0;
       return (x', r)
     }
   }
```

```
lemma delete-min-idx-pqueue-rule [hoare-triple]:
  xs ≠ [] ⟹ index-of-pqueue (xs, m) ⟹
   <idx-pqueue (xs, m) p>
   delete-min-idx-pqueue p
   <λ(x, r). idx-pqueue (snd (delete-min-idx-pqueue-fun (xs, m))) r ∗
       ↑(x = fst (delete-min-idx-pqueue-fun (xs, m)))>
⟨proof⟩
```

```
definition insert-idx-pqueue :: nat ⇒ 'a::{heap,linorder} ⇒ 'a indexed-pqueue ⇒
'a indexed-pqueue Heap where
  insert-idx-pqueue k v p = do {
     p' ← idx-pqueue-push k v p;
     len ← idx-pqueue-length p';
     idx-bubble-up p' (len − 1);
     return p'
   }
```

```
lemma insert-idx-pqueue-rule [hoare-triple]:
  k < length m ⟹ ¬has-key-alist xs k ⟹ index-of-pqueue (xs, m) ⟹
```

<idx-pqueue (xs, m) p>
insert-idx-pqueue k v p
<idx-pqueue (insert-idx-pqueue-fun k v (xs, m))>$_t$
⟨proof⟩

definition *update-idx-pqueue* ::
nat ⇒ 'a::{heap,linorder} ⇒ 'a indexed-pqueue ⇒ 'a indexed-pqueue Heap **where**
update-idx-pqueue k v p = do {
i-opt ← idx-nth p k;
case i-opt of
None ⇒ insert-idx-pqueue k v p
| Some i ⇒ do {
x ← idx-pqueue-nth p i;
idx-pqueue-array-upd i (k, v) (pqueue p);
(if snd x ≤ v then do {idx-bubble-down p i; return p}
else do {idx-bubble-up p i; return p}) }}

lemma *update-idx-pqueue-rule* [*hoare-triple*]:
k < length m ⟹ index-of-pqueue (xs, m) ⟹
<idx-pqueue (xs, m) p>
update-idx-pqueue k v p
<idx-pqueue (update-idx-pqueue-fun k v (xs, m))>$_t$
⟨proof⟩

25.4 Outer interface

Express Hoare triples for indexed priority queue operations in terms of the mapping represented by the queue.

definition *idx-pqueue-map* :: (*nat*, *'a*::{*heap*,*linorder*}) *map* ⇒ *nat* ⇒ *'a indexed-pqueue* ⇒ *assn* **where**
idx-pqueue-map M n p = ($\exists_A$*xs m*. *idx-pqueue* (*xs*, *m*) *p* ∗
↑(*index-of-pqueue* (*xs*, *m*)) ∗ ↑(*is-heap xs*) ∗ ↑(*M* = *map-of-alist xs*) ∗ ↑(*n* = *length m*))
⟨ML⟩

lemma *heap-implies-hd-min2* [*resolve*]:
is-heap xs ⟹ xs ≠ [] ⟹ (map-of-alist xs)⟨k⟩ = Some v ⟹ snd (hd xs) ≤ v
⟨proof⟩

theorem *idx-pqueue-empty-map* [*hoare-triple*]:
<emp>
idx-pqueue-empty n
<idx-pqueue-map empty-map n> ⟨proof⟩

theorem *delete-min-idx-pqueue-map* [*hoare-triple*]:
<idx-pqueue-map M n p ∗ ↑(M ≠ empty-map)>
delete-min-idx-pqueue p
<λ(x, r). idx-pqueue-map (delete-map (fst x) M) n r ∗ ↑(fst x < n) ∗
↑(is-heap-min (fst x) M) ∗ ↑(M⟨fst x⟩ = Some (snd x))> ⟨proof⟩

theorem *insert-idx-pqueue-map* [*hoare-triple*]:
  $k < n \Longrightarrow k \notin$ *keys-of* $M \Longrightarrow$
    <*idx-pqueue-map M n p*>
    insert-idx-pqueue k v p
    <*idx-pqueue-map* (*M* {$k \rightarrow v$}) *n*>$_t$ ⟨*proof*⟩

theorem *has-key-idx-pqueue-map* [*hoare-triple*]:
  $k < n \Longrightarrow$
    <*idx-pqueue-map M n p*>
    has-key-idx-pqueue k p
    <λ*r. idx-pqueue-map M n p* ∗ ↑(*r* ⟷ *k* ∈ *keys-of M*)> ⟨*proof*⟩

theorem *update-idx-pqueue-map* [*hoare-triple*]:
  $k < n \Longrightarrow$
    <*idx-pqueue-map M n p*>
    update-idx-pqueue k v p
    <*idx-pqueue-map* (*M* {$k \rightarrow v$}) *n*>$_t$ ⟨*proof*⟩

⟨*ML*⟩

end

26 Implementation of Dijkstra's algorithm

theory *Dijkstra-Impl*
  imports *Indexed-PQueue-Impl ../Functional/Dijkstra*
begin

Imperative implementation of Dijkstra's shortest path algorithm. The algorithm is also verified by Nordhoff and Lammich in [8].

datatype *dijkstra-state* = *Dijkstra-State* (*est-a*: *nat array*) (*heap-pq*: *nat indexed-pqueue*)
⟨*ML*⟩

fun *dstate* :: *state* ⇒ *dijkstra-state* ⇒ *assn* **where**
  dstate (*State e M*) (*Dijkstra-State a pq*) = $a \mapsto_a e$ ∗ *idx-pqueue-map M* (*length e*) *pq*
⟨*ML*⟩

26.1 Basic operations

fun *dstate-pq-init* :: *graph* ⇒ *nat* ⇒ *nat indexed-pqueue Heap* **where**
  dstate-pq-init G 0 = *idx-pqueue-empty* (*size G*)
| *dstate-pq-init G* (*Suc k*) = *do* {
    p ← *dstate-pq-init G k*;
    if k > *0 then update-idx-pqueue k* (*weight G 0 k*) *p*
    else return p }

lemma *dstate-pq-init-to-fun* [*hoare-triple*]:

k ≤ size G ⟹
<emp>
dstate-pq-init G k
<idx-pqueue-map (map-constr (λi. i > 0) (λi. weight G 0 i) k) (size G)>$_t$
⟨*proof*⟩

definition *dstate-init* :: *graph ⇒ dijkstra-state Heap* **where**
dstate-init G = do {
a ← Array.of-list (list (λi. if i = 0 then 0 else weight G 0 i) (size G));
pq ← dstate-pq-init G (size G);
return (Dijkstra-State a pq)
}

lemma *dstate-init-to-fun* [*hoare-triple*]:
<emp>
dstate-init G
<dstate (dijkstra-start-state G)>$_t$ ⟨*proof*⟩

fun *dstate-update-est* :: *graph ⇒ nat ⇒ nat ⇒ nat indexed-pqueue ⇒ nat array ⇒ nat array Heap* **where**
dstate-update-est G m 0 pq a = (return a)
| *dstate-update-est G m (Suc k) pq a = do {*
b ← has-key-idx-pqueue k pq;
if b then do {
ek ← Array.nth a k;
em ← Array.nth a m;
a′ ← dstate-update-est G m k pq a;
a′′ ← Array.upd k (min (em + weight G m k) ek) a′;
return a′′ }
else dstate-update-est G m k pq a }

lemma *dstate-update-est-ind* [*hoare-triple*]:
k ≤ length e ⟹ m < length e ⟹
<a $\mapsto_a$ *e ∗ idx-pqueue-map M (length e) pq>*
dstate-update-est G m k pq a
<λr. dstate (State (list-update-set-impl (λi. i ∈ keys-of M)
(λi. min (e ! m + weight G m i) (e ! i)) e k) M) (Dijkstra-State r pq)>$_t$
⟨*proof*⟩

lemma *dstate-update-est-to-fun* [*hoare-triple*]:
<dstate (State e M) (Dijkstra-State a pq) ∗ ↑(m < length e)>
dstate-update-est G m (length e) pq a
<λr. dstate (State (list-update-set (λi. i ∈ keys-of M)
(λi. min (e ! m + weight G m i) (e ! i)) e) M) (Dijkstra-State r pq)>$_t$
⟨*proof*⟩

fun *dstate-update-heap* ::
graph ⇒ nat ⇒ nat ⇒ nat array ⇒ nat indexed-pqueue ⇒ nat indexed-pqueue

Heap **where**
  dstate-update-heap G m 0 a pq = return pq
| *dstate-update-heap G m (Suc k) a pq = do {*
    b ← has-key-idx-pqueue k pq;
    if b then do {
      ek ← Array.nth a k;
      pq′ ← dstate-update-heap G m k a pq;
      update-idx-pqueue k ek pq′ }
    else dstate-update-heap G m k a pq }

lemma *dstate-update-heap-ind* [*hoare-triple*]:
  k ≤ length e ⟹ m < length e ⟹
  <*a* $\mapsto_a$ *e * idx-pqueue-map M (length e) pq*>
  dstate-update-heap G m k a pq
  <*λr. dstate (State e (map-update-all-impl (λi. e ! i) M k)) (Dijkstra-State a r)*>$_t$
⟨*proof*⟩

lemma *dstate-update-heap-to-fun* [*hoare-triple*]:
  m < length e ⟹
  ∀ i∈keys-of M. i < length e ⟹
  <*dstate (State e M) (Dijkstra-State a pq)*>
  dstate-update-heap G m (length e) a pq
  <*λr. dstate (State e (map-update-all (λi. e ! i) M)) (Dijkstra-State a r)*>$_t$
⟨*proof*⟩

fun *dijkstra-extract-min* :: *dijkstra-state ⇒ (nat × dijkstra-state) Heap* **where**
  dijkstra-extract-min (Dijkstra-State a pq) = do {
    (x, pq′) ← delete-min-idx-pqueue pq;
    return (fst x, Dijkstra-State a pq′) }

lemma *dijkstra-extract-min-rule* [*hoare-triple*]:
  M ≠ empty-map ⟹
  <*dstate (State e M) (Dijkstra-State a pq)*>
  dijkstra-extract-min (Dijkstra-State a pq)
  <*λ(m, r). dstate (State e (delete-map m M)) r * ↑(m < length e) * ↑(is-heap-min m M)*>$_t$ ⟨*proof*⟩

⟨*ML*⟩

26.2 Main operations

fun *dijkstra-step-impl* :: *graph ⇒ dijkstra-state ⇒ dijkstra-state Heap* **where**
  dijkstra-step-impl G (Dijkstra-State a pq) = do {
    (x, S′) ← dijkstra-extract-min (Dijkstra-State a pq);
    a′ ← dstate-update-est G x (size G) (heap-pq S′) (est-a S′);
    pq′′ ← dstate-update-heap G x (size G) a′ (heap-pq S′);
    return (Dijkstra-State a′ pq′′) }

lemma *dijkstra-step-impl-to-fun* [*hoare-triple*]:
  heap S $\neq$ *empty-map* $\Longrightarrow$ *inv G S* $\Longrightarrow$
   <*dstate S* (*Dijkstra-State a pq*)>
   dijkstra-step-impl G (*Dijkstra-State a pq*)
   <$\lambda r.\ \exists_A S'.$ *dstate* S' *r* $*$ $\uparrow$(*is-dijkstra-step G S* S')>$_t$ ⟨*proof*⟩

lemma *dijkstra-step-impl-correct* [*hoare-triple*]:
  heap S $\neq$ *empty-map* $\Longrightarrow$ *inv G S* $\Longrightarrow$
   <*dstate S p*>
   dijkstra-step-impl G p
  <$\lambda r.\ \exists_A S'.$ *dstate* S' *r* $*$ $\uparrow$(*inv G* S') $*$ $\uparrow$(*card* (*unknown-set* S') = *card* (*unknown-set S*) $-$ *1*)>$_t$ ⟨*proof*⟩

fun *dijkstra-loop* :: *graph* $\Rightarrow$ *nat* $\Rightarrow$ *dijkstra-state* $\Rightarrow$ *dijkstra-state Heap* **where**
  dijkstra-loop G 0 p = (*return p*)
| *dijkstra-loop G* (*Suc k*) *p* = *do* {
    p' $\leftarrow$ *dijkstra-step-impl G p*;
    p'' $\leftarrow$ *dijkstra-loop G k* p';
    return p'' }

lemma *dijkstra-loop-correct* [*hoare-triple*]:
  <*dstate S p* $*$ $\uparrow$(*inv G S*) $*$ $\uparrow$(*n* $\leq$ *card* (*unknown-set S*))>
   dijkstra-loop G n p
  <$\lambda r.\ \exists_A S'.$ *dstate* S' *r* $*$ $\uparrow$(*inv G* S') $*$ $\uparrow$(*card* (*unknown-set* S') = *card* (*unknown-set S*) $-$ *n*)>$_t$
⟨*proof*⟩

definition *dijkstra* :: *graph* $\Rightarrow$ *dijkstra-state Heap* **where**
  dijkstra G = *do* {
    p $\leftarrow$ *dstate-init G*;
    dijkstra-loop G (*size G* $-$ *1*) *p* }

Correctness of Dijkstra's algorithm.

theorem *dijkstra-correct* [*hoare-triple*]:
  size G > *0* $\Longrightarrow$
   <*emp*>
   dijkstra G
   <$\lambda r.\ \exists_A S.$ *dstate S r* $*$ $\uparrow$(*inv G S*) $*$ $\uparrow$(*unknown-set S* = {}) $*$
     $\uparrow$($\forall$ *i*$\in$*verts G*. *has-dist G 0 i* $\wedge$ *est S* ! *i* = *dist G 0 i*)>$_t$ ⟨*proof*⟩

end

27 Implementation of interval tree

theory *IntervalTree-Impl*
  imports *SepAuto ../Functional/Interval-Tree*
begin

Imperative version of interval tree.

27.1 Interval and IdxInterval

fun *interval-encode* :: ('*a*::*heap*) *interval* ⇒ *nat* **where**
interval-encode (*Interval l h*) = *to-nat* (*l*, *h*)

instance *interval* :: (*heap*) *heap*
⟨*proof*⟩

fun *idx-interval-encode* :: ('*a*::*heap*) *idx-interval* ⇒ *nat* **where**
idx-interval-encode (*IdxInterval it i*) = *to-nat* (*it*, *i*)

instance *idx-interval* :: (*heap*) *heap*
⟨*proof*⟩

27.2 Tree nodes

datatype '*a node* =
Node (*lsub*: '*a node ref option*) (*val*: '*a idx-interval*) (*tmax*: *nat*) (*rsub*: '*a node ref option*)
⟨*ML*⟩

fun *node-encode* :: ('*a*::*heap*) *node* ⇒ *nat* **where**
node-encode (*Node l v m r*) = *to-nat* (*l*, *v*, *m*, *r*)

instance *node* :: (*heap*) *heap*
⟨*proof*⟩

fun *int-tree* :: *interval-tree* ⇒ *nat node ref option* ⇒ *assn* **where**
int-tree Tip p = ↑(*p* = *None*)
| *int-tree* (*interval-tree.Node lt v m rt*) (*Some p*) = ($\exists_A$*lp rp*. *p* $\mapsto_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp*)
| *int-tree* (*interval-tree.Node lt v m rt*) *None* = *false*
⟨*ML*⟩

lemma *int-tree-Tip* [*forward-ent*]: *int-tree Tip p* $\Longrightarrow_A$ ↑(*p* = *None*) ⟨*proof*⟩

lemma *int-tree-Node* [*forward-ent*]:
int-tree (*interval-tree.Node lt v m rt*) *p* $\Longrightarrow_A$ ($\exists_A$*lp rp*. *the p* $\mapsto_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp* ∗ ↑(*p* ≠ *None*))
⟨*proof*⟩

lemma *int-tree-none*: *emp* $\Longrightarrow_A$ *int-tree interval-tree.Tip None* ⟨*proof*⟩

lemma *int-tree-constr-ent*:
p $\mapsto_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp* $\Longrightarrow_A$ *int-tree* (*interval-tree.Node lt v m rt*) (*Some p*) ⟨*proof*⟩

⟨*ML*⟩

type-synonym *int-tree* = *nat node ref option*

27.3 Operations

27.3.1 Basic operation

definition *int-tree-empty* :: *int-tree Heap* **where**
  int-tree-empty = *return None*

lemma *int-tree-empty-to-fun* [*hoare-triple*]:
  <*emp*> *int-tree-empty* <*int-tree Tip*> ⟨*proof*⟩

definition *int-tree-is-empty* :: *int-tree* ⇒ *bool Heap* **where**
  int-tree-is-empty b = *return* (*b* = *None*)

lemma *int-tree-is-empty-rule* [*hoare-triple*]:
  <*int-tree t b*>
   int-tree-is-empty b
   <λ*r*. *int-tree t b* ∗ ↑(*r* ⟷ *t* = *Tip*)> ⟨*proof*⟩

definition *get-tmax* :: *int-tree* ⇒ *nat Heap* **where**
  get-tmax b = (*case b of*
     None ⇒ *return 0*
   | *Some p* ⇒ *do* {
      t ← !*p*;
      return (*tmax t*) })

lemma *get-tmax-rule* [*hoare-triple*]:
  <*int-tree t b*> *get-tmax b* <λ*r*. *int-tree t b* ∗ ↑(*r* = *interval-tree.tmax t*)>
⟨*proof*⟩

definition *compute-tmax* :: *nat idx-interval* ⇒ *int-tree* ⇒ *int-tree* ⇒ *nat Heap*
where
  compute-tmax it l r = *do* {
    lm ← *get-tmax l*;
    rm ← *get-tmax r*;
    return (*max3 it lm rm*)
  }

lemma *compute-tmax-rule* [*hoare-triple*]:
  <*int-tree t1 b1* ∗ *int-tree t2 b2*>
   compute-tmax it b1 b2
    <λ*r*. *int-tree t1 b1* ∗ *int-tree t2 b2* ∗ ↑(*r* = *max3 it* (*interval-tree.tmax t1*)
(*interval-tree.tmax t2*))>
  ⟨*proof*⟩

definition *int-tree-constr* :: *int-tree* ⇒ *nat idx-interval* ⇒ *int-tree* ⇒ *int-tree Heap*
where
  int-tree-constr lp v rp = *do* {
    m ← *compute-tmax v lp rp*;
    p ← *ref* (*Node lp v m rp*);
    return (*Some p*) }

```
lemma int-tree-constr-rule [hoare-triple]:
  <int-tree lt lp * int-tree rt rp>
   int-tree-constr lp v rp
  <int-tree (interval-tree.Node lt v (max3 v (interval-tree.tmax lt) (interval-tree.tmax
rt)) rt)>
  ⟨proof⟩
```

27.3.2 Insertion

```
partial-function (heap) insert-impl :: nat idx-interval ⇒ int-tree ⇒ int-tree Heap
where
  insert-impl v b = (case b of
    None ⇒ int-tree-constr None v None
  | Some p ⇒ do {
    t ← !p;
    (if v = val t then do {
       return (Some p) }
     else if v < val t then do {
       q ← insert-impl v (lsub t);
       m ← compute-tmax (val t) q (rsub t);
       p := Node q (val t) m (rsub t);
       return (Some p) }
     else do {
       q ← insert-impl v (rsub t);
       m ← compute-tmax (val t) (lsub t) q;
       p := Node (lsub t) (val t) m q;
       return (Some p) })})
```

```
lemma int-tree-insert-to-fun [hoare-triple]:
  <int-tree t b>
    insert-impl v b
   <int-tree (insert v t)>
⟨proof⟩
```

27.3.3 Deletion

```
partial-function (heap) int-tree-del-min :: int-tree ⇒ (nat idx-interval × int-tree)
Heap where
  int-tree-del-min b = (case b of
     None ⇒ raise STR ''del-min: empty tree''
   | Some p ⇒ do {
      t ← !p;
      (if lsub t = None then
         return (val t, rsub t)
       else do {
         r ← int-tree-del-min (lsub t);
         m ← compute-tmax (val t) (snd r) (rsub t);
         p := Node (snd r) (val t) m (rsub t);
         return (fst r, Some p) })})
```

```
lemma int-tree-del-min-to-fun [hoare-triple]:
  <int-tree t b * ↑(b ≠ None)>
   int-tree-del-min b
   <λr. int-tree (snd (del-min t)) (snd r) * ↑(fst(r) = fst (del-min t))>t
⟨proof⟩
```

```
definition int-tree-del-elt :: int-tree ⇒ int-tree Heap where
  int-tree-del-elt b = (case b of
     None ⇒ raise STR ''del-elt: empty tree''
   | Some p ⇒ do {
       t ← !p;
       (if lsub t = None then return (rsub t)
        else if rsub t = None then return (lsub t)
        else do {
          r ← int-tree-del-min (rsub t);
          m ← compute-tmax (fst r) (lsub t) (snd r);
          p := Node (lsub t) (fst r) m (snd r);
          return (Some p) }) })
```

```
lemma int-tree-del-elt-to-fun [hoare-triple]:
  <int-tree (interval-tree.Node lt v m rt) b>
   int-tree-del-elt b
   <int-tree (delete-elt-tree (interval-tree.Node lt v m rt))>t ⟨proof⟩
```

```
partial-function (heap) delete-impl :: nat idx-interval ⇒ int-tree ⇒ int-tree Heap
where
  delete-impl x b = (case b of
     None ⇒ return None
   | Some p ⇒ do {
      t ← !p;
      (if x = val t then do {
         r ← int-tree-del-elt b;
         return r }
       else if x < val t then do {
         q ← delete-impl x (lsub t);
         m ← compute-tmax (val t) q (rsub t);
         p := Node q (val t) m (rsub t);
         return (Some p) }
       else do {
         q ← delete-impl x (rsub t);
         m ← compute-tmax (val t) (lsub t) q;
         p := Node (lsub t) (val t) m q;
         return (Some p) })})
```

```
lemma int-tree-delete-to-fun [hoare-triple]:
  <int-tree t b>
    delete-impl x b
   <int-tree (delete x t)>t
```

⟨proof⟩

27.3.4 Search

```
partial-function (heap) search-impl :: nat interval ⇒ int-tree ⇒ bool Heap where
  search-impl x b = (case b of
     None ⇒ return False
   | Some p ⇒ do {
      t ← !p;
      (if is-overlap (int (val t)) x then return True
       else case lsub t of
           None ⇒ do { b ← search-impl x (rsub t); return b }
         | Some lp ⇒ do {
            lt ← !lp;
            if tmax lt ≥ low x then
              do { b ← search-impl x (lsub t); return b }
            else
              do { b ← search-impl x (rsub t); return b }})})

lemma search-impl-correct [hoare-triple]:
  <int-tree t b>
   search-impl x b
  <λr. int-tree t b ∗ ↑(r ⟷ search t x)>
⟨proof⟩
```

27.4 Outer interface

Express Hoare triples for operations on interval tree in terms of the set of intervals represented by the tree.

```
definition int-tree-set :: nat idx-interval set ⇒ int-tree ⇒ assn where
  int-tree-set S p = (∃A t. int-tree t p ∗ ↑(is-interval-tree t) ∗ ↑(S = tree-set t))
⟨ML⟩

theorem int-tree-empty-rule [hoare-triple]:
  <emp> int-tree-empty <int-tree-set {}> ⟨proof⟩

theorem int-tree-insert-rule [hoare-triple]:
  <int-tree-set S b ∗ ↑(is-interval (int x))>
   insert-impl x b
  <int-tree-set (S ∪ {x})> ⟨proof⟩

theorem int-tree-delete-rule [hoare-triple]:
  <int-tree-set S b ∗ ↑(is-interval (int x))>
   delete-impl x b
  <int-tree-set (S − {x})>t ⟨proof⟩

theorem int-tree-search-rule [hoare-triple]:
  <int-tree-set S b ∗ ↑(is-interval x)>
   search-impl x b
```

<λr. *int-tree-set S b* ∗ ↑(*r* ⟷ *has-overlap S x*)> ⟨*proof*⟩

⟨*ML*⟩

end

28 Implementation of rectangle intersection

theory *Rect-Intersect-Impl*
imports *../Functional/Rect-Intersect IntervalTree-Impl Quicksort-Impl*
begin

Imperative version of rectangle-intersection algorithm.

28.1 Operations

fun *operation-encode* :: (*'a*::*heap*) *operation* ⇒ *nat* **where**
operation-encode oper =
(*case oper of INS p i n* ⇒ *to-nat* (*is-INS oper*, *p*, *i*, *n*)
| *DEL p i n* ⇒ *to-nat* (*is-INS oper*, *p*, *i*, *n*))

instance *operation* :: (*heap*) *heap*
⟨*proof*⟩

28.2 Initial state

definition *rect-inter-init* :: *nat rectangle list* ⇒ *nat operation array Heap* **where**
rect-inter-init rects = *do* {
p ← *Array.of-list* (*ins-ops rects* @ *del-ops rects*);
quicksort-all p;
return p }

⟨*ML*⟩
lemma *rect-inter-init-rule* [*hoare-triple*]:
<*emp*> *rect-inter-init rects* <λp. $p \mapsto_a$ *all-ops rects*> ⟨*proof*⟩
⟨*ML*⟩

definition *rect-inter-next* :: *nat operation array* ⇒ *int-tree* ⇒ *nat* ⇒ *int-tree Heap*
where
rect-inter-next a b k = *do* {
oper ← *Array.nth a k*;
if is-INS oper then
IntervalTree-Impl.insert-impl (*IdxInterval* (*op-int oper*) (*op-idx oper*)) *b*
else
IntervalTree-Impl.delete-impl (*IdxInterval* (*op-int oper*) (*op-idx oper*)) *b* }

lemma *op-int-is-interval*:
is-rect-list rects ⟹ *ops* = *all-ops rects* ⟹ *k* < *length ops* ⟹
is-interval (*op-int* (*ops* ! *k*))

⟨*proof*⟩
⟨*ML*⟩

lemma *rect-inter-next-rule* [*hoare-triple*]:
is-rect-list rects ⟹ *k* < *length* (*all-ops rects*) ⟹
<*a* $\mapsto_a$ *all-ops rects* ∗ *int-tree-set S b*>
rect-inter-next a b k
<λ*r*. *a* $\mapsto_a$ *all-ops rects* ∗ *int-tree-set* (*apply-ops-k-next rects S k*) *r*>$_t$ ⟨*proof*⟩

partial-function (*heap*) *rect-inter-impl* ::
nat operation array ⇒ *int-tree* ⇒ *nat* ⇒ *bool Heap* **where**
rect-inter-impl a b k = *do* {
  n ← *Array.len a*;
  (*if k* ≥ *n then return False*
   else do {
    oper ← *Array.nth a k*;
    (*if is-INS oper then do* {
      overlap ← *IntervalTree-Impl.search-impl* (*op-int oper*) *b*;
      if overlap then return True
      else if k = *n* − *1 then return False*
      else do {
        b′ ← *rect-inter-next a b k*;
        rect-inter-impl a b′ (*k* + *1*)}}
     else
      if k = *n* − *1 then return False*
      else do {
        b′ ← *rect-inter-next a b k*;
        rect-inter-impl a b′ (*k* + *1*)}})})}

lemma *rect-inter-to-fun-ind* [*hoare-triple*]:
is-rect-list rects ⟹ *k* < *length* (*all-ops rects*) ⟹
<*a* $\mapsto_a$ *all-ops rects* ∗ *int-tree-set S b*>
rect-inter-impl a b k
<λ*r*. *a* $\mapsto_a$ *all-ops rects* ∗ ↑(*r* ⟷ *rect-inter rects S k*)>$_t$
⟨*proof*⟩

definition *rect-inter-all* :: *nat rectangle list* ⇒ *bool Heap* **where**
rect-inter-all rects =
  (*if rects* = [] *then return False*
   else do {
    a ← *rect-inter-init rects*;
    b ← *int-tree-empty*;
    rect-inter-impl a b 0 })

Correctness of rectangle intersection algorithm.

theorem *rect-inter-all-correct*:
is-rect-list rects ⟹
<*emp*>
rect-inter-all rects

$<\lambda r.\ \uparrow(r = \mathit{has\text{-}rect\text{-}overlap}\ \mathit{rects})>_t$ ⟨*proof*⟩

end

References

[1] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with isabelle/hol. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 134–149, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms third edition. 2009.

[3] P. Lammich. Collections framework. *Archive of Formal Proofs*, Nov. 2009. http://isa-afp.org/entries/Collections.html, Formal proof development.

[4] P. Lammich. The imperative refinement framework. *Archive of Formal Proofs*, Aug. 2016. http://isa-afp.org/entries/Refine_Imperative_HOL.html, Formal proof development.

[5] P. Lammich and R. Meis. A separation logic framework for imperative hol. *Archive of Formal Proofs*, Nov. 2012. http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

[6] T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, pages 307–322, Cham, 2016. Springer International Publishing.

[7] T. Nipkow. Programming and proving in isabelle/hol. 2018.

[8] B. Nordhoff and P. Lammich. Dijkstra's shortest path algorithm. *Archive of Formal Proofs*, Jan. 2012. http://isa-afp.org/entries/Dijkstra_Shortest_Path.html, Formal proof development.

[9] B. Zhan. Efficient verification of imperative programs using auto2. In D. Beyer and M. Huisman, editors, *TACAS 2018*, pages 23–40, 2018.