Verifying Imperative Programs using Auto2

Bohua Zhan

May 26, 2024

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* $'a \Rightarrow 'b$ *option*

fun *meval* :: $('a, 'b)$ *map* $\Rightarrow 'a \Rightarrow 'b$ *option* (-⟨-⟩ [*90*]) **where**
  $(Map\ f)\ \langle h \rangle = f\ h$
setup ‹*add-rewrite-rule* @{*thm meval.simps*}›

lemma *meval-ext*: $\forall x.\ M\langle x \rangle = N\langle x \rangle \Longrightarrow M = N$
  apply (*cases M*) **apply** (*cases N*) **by** *auto*
setup ‹*add-backward-prfstep-cond* @{*thm meval-ext*} [*with-filt* (*order-filter M N*)]›

definition *empty-map* :: $('a, 'b)$ *map* **where**
  empty-map $=$ *Map* $(\lambda x.$ *None*$)$
setup ‹*add-rewrite-rule* @{*thm empty-map-def*}›

definition *update-map* :: $('a, 'b)$ *map* $\Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *map* (- { - → - } [*89*,*90*,*90*] *90*) **where**
  $M\ \{k \rightarrow v\} = Map\ (\lambda x.$ *if* $x = k$ *then Some* v *else* $M\langle x \rangle)$
setup ‹*add-rewrite-rule* @{*thm update-map-def*}›

definition *delete-map* :: $'a \Rightarrow ('a, 'b)$ *map* $\Rightarrow ('a, 'b)$ *map* **where**
  delete-map $k\ M = Map\ (\lambda x.$ *if* $x = k$ *then None else* $M\langle x \rangle)$
setup ‹*add-rewrite-rule* @{*thm delete-map-def*}›

2.1 Map from an AList

fun *map-of-alist* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *map* **where**
  map-of-alist [] = *empty-map*
| *map-of-alist* (*x* # *xs*) = (*map-of-alist xs*) {*fst x* → *snd x*}
setup ‹*fold add-rewrite-rule* @{*thms map-of-alist.simps*}›

definition *has-key-alist* :: ('*a* × '*b*) *list* ⇒ '*a* ⇒ *bool* **where** [*rewrite*]:
  has-key-alist xs a ⟷ (∃ *p*∈*set xs*. *fst p* = *a*)

lemma *map-of-alist-nil* [*rewrite-back*]:
  has-key-alist ys x ⟷ (*map-of-alist ys*)⟨*x*⟩ ≠ *None*
@**proof** @**induct** *ys* @**qed**
setup ‹*add-rewrite-rule-cond* @{*thm map-of-alist-nil*} [*with-term* (*map-of-alist ?ys*)⟨*?x*⟩]›

lemma *map-of-alist-some* [*forward*]:
  (*map-of-alist xs*)⟨*k*⟩ = *Some v* ⟹ (*k*, *v*) ∈ *set xs*
@**proof** @**induct** *xs* @**qed**

lemma *map-of-alist-nil'*:
  x ∈ *set* (*map fst ys*) ⟷ (*map-of-alist ys*)⟨*x*⟩ ≠ *None*
@**proof** @**induct** *ys* @**qed**
setup ‹*add-rewrite-rule-cond* @{*thm map-of-alist-nil'*} [*with-term* (*map-of-alist ?ys*)⟨*?x*⟩]›

2.2 Mapping defined by a set of key-value pairs

definition *unique-keys-set* :: ('*a* × '*b*) *set* ⇒ *bool* **where** [*rewrite*]:
  unique-keys-set S = (∀ *i x y*. (*i*, *x*) ∈ *S* ⟶ (*i*, *y*) ∈ *S* ⟶ *x* = *y*)

lemma *unique-keys-setD* [*forward*]: *unique-keys-set S* ⟹ (*i*, *x*) ∈ *S* ⟹ (*i*, *y*) ∈ *S* ⟹ *x* = *y* **by** *auto2*
setup ‹*del-prfstep-thm-eqforward* @{*thm unique-keys-set-def*}›

definition *map-of-aset* :: ('*a* × '*b*) *set* ⇒ ('*a*, '*b*) *map* **where**
  map-of-aset S = *Map* (λ*a*. *if* ∃ *b*. (*a*, *b*) ∈ *S then Some* (*THE b*. (*a*, *b*) ∈ *S*) *else None*)
setup ‹*add-rewrite-rule* @{*thm map-of-aset-def*}›
setup ‹*add-prfstep-check-req* (*map-of-aset S*, *unique-keys-set S*)›

lemma *map-of-asetI1* [*rewrite*]: *unique-keys-set S* ⟹ (*a*, *b*) ∈ *S* ⟹ (*map-of-aset S*)⟨*a*⟩ = *Some b*
@**proof** @**have** ∃ *b*. (*a*, *b*) ∈ *S* @**have** ∃!*b*. (*a*, *b*) ∈ *S* @**qed**

lemma *map-of-asetI2* [*rewrite*]: ∀ *b*. (*a*, *b*) ∉ *S* ⟹ (*map-of-aset S*)⟨*a*⟩ = *None* **by** *auto2*

lemma *map-of-asetD1* [*forward*]: (*map-of-aset S*)⟨*a*⟩ = *None* ⟹ ∀ *b*. (*a*, *b*) ∉ *S* **by** *auto2*

lemma *map-of-asetD2* [*forward*]:

unique-keys-set $S \Longrightarrow$ (*map-of-aset* S)$\langle a \rangle$ = *Some* $b \Longrightarrow (a, b) \in S$ **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm map-of-aset-def*}›

lemma *map-of-aset-insert* [*rewrite*]:
unique-keys-set ($S \cup \{(k, v)\}$) $\Longrightarrow$ *map-of-aset* ($S \cup \{(k, v)\}$) = (*map-of-aset* S) $\{k \to v\}$
@**proof**
@**let** M = *map-of-aset* S N = *map-of-aset* ($S \cup \{(k, v)\}$)
@**have** (@*rule*) $\forall x.\ N\langle x \rangle = (M\ \{k \to v\})\ \langle x \rangle$ @*with* @**case** $M\langle x \rangle$ = *None* @**end**
@**qed**

lemma *map-of-alist-to-aset* [*rewrite*]:
unique-keys-set (*set xs*) $\Longrightarrow$ *map-of-aset* (*set xs*) = *map-of-alist xs*
@**proof** @**induct** *xs* @*with*
@**subgoal** $xs = x \# xs'$
@**have** *set* ($x \# xs'$) = *set* $xs' \cup \{x\}$
@**endgoal** @**end**
@**qed**

lemma *map-of-aset-delete* [*rewrite*]:
unique-keys-set $S \Longrightarrow (k, v) \in S \Longrightarrow$ *map-of-aset* ($S - \{(k, v)\}$) = *delete-map* k (*map-of-aset* S)
@**proof**
@**let** $T = S - \{(k, v)\}$
@**let** M = *map-of-aset* S N = *map-of-aset* T
@**have** (@*rule*) $\forall x.\ N\langle x \rangle = (\textit{delete-map}\ k\ M)\ \langle x \rangle$ @*with*
@**case** $M\langle x \rangle$ = *None* @**case** $x = k$
@**obtain** y **where** $M\langle x \rangle$ = *Some* y @**have** $(x, y) \in T$
@**end**
@**qed**

lemma *map-of-aset-update* [*rewrite*]:
unique-keys-set $S \Longrightarrow (k, v) \in S \Longrightarrow$
map-of-aset ($S - \{(k, v)\} \cup \{(k, v')\}$) = (*map-of-aset* S) $\{k \to v'\}$ **by** *auto2*

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

lemma *map-of-alist-insert* [*rewrite*]:
set xs' = *set* $xs \cup \{x\} \Longrightarrow$ *unique-keys-set* (*set* xs') $\Longrightarrow$
map-of-alist xs' = (*map-of-alist xs*) $\{fst\ x \to snd\ x\}$
@**proof** @**have** *map-of-alist* xs' = *map-of-aset* (*set* xs') @**qed**

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

@**proof** @**have** *map-of-alist xs′ = map-of-aset (set xs′)* @**qed**

2.3 Set of keys of a mapping

definition *keys-of* :: (*′a*, *′b*) *map* ⇒ *′a set* **where** [*rewrite*]:
keys-of M = {*x*. *M*⟨*x*⟩ ≠ *None*}

lemma *keys-of-iff* [*rewrite-bidir*]: *x* ∈ *keys-of M* ⟷ *M*⟨*x*⟩ ≠ *None* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm keys-of-def*}›

lemma *keys-of-empty* [*rewrite*]: *keys-of empty-map* = {} **by** *auto2*

lemma *keys-of-delete* [*rewrite*]:
keys-of (*delete-map x M*) = *keys-of M* − {*x*} **by** *auto2*

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

definition *is-heap-min* :: *′a* ⇒ (*′a*, *′b*::*linorder*) *map* ⇒ *bool* **where** [*rewrite*]:
is-heap-min x M ⟷ *x* ∈ *keys-of M* ∧ (∀ *k*∈*keys-of M*. *the* (*M*⟨*x*⟩) ≤ *the* (*M*⟨*k*⟩))

2.5 General construction and update of maps

fun *map-constr* :: (*nat* ⇒ *bool*) ⇒ (*nat* ⇒ *′a*) ⇒ *nat* ⇒ (*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* → *f k*} *else M*)
setup ‹*fold add-rewrite-rule* @{*thms map-constr.simps*}›

lemma *map-constr-eval* [*rewrite*]:
map-constr S f n = *Map* (λ*i*. *if i* < *n then if S i then Some* (*f i*) *else None else None*)
@**proof** @**induct** *n* @**qed**

lemma *keys-of-map-constr* [*rewrite*]:
i ∈ *keys-of* (*map-constr S f n*) ⟷ (*S i* ∧ *i* < *n*) **by** *auto2*

definition *map-update-all* :: (*nat* ⇒ *′a*) ⇒ (*nat*, *′a*) *map* ⇒ (*nat*, *′a*) *map* **where** [*rewrite*]:
map-update-all f M = *Map* (λ*i*. *if i* ∈ *keys-of M then Some* (*f i*) *else M*⟨*i*⟩)

fun *map-update-all-impl* :: (*nat* ⇒ *′a*) ⇒ (*nat*, *′a*) *map* ⇒ *nat* ⇒ (*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* ∈ *keys-of M then M′* {*k* → *f k*} *else M′*)
setup ‹*fold add-rewrite-rule* @{*thms map-update-all-impl.simps*}›

lemma *map-update-all-impl-ind* [*rewrite*]:

map-update-all-impl f M n = Map (λi. if i < n then if i ∈ keys-of M then Some (f i) else None else M⟨i⟩)
@proof @induct *n* **@qed**

lemma *map-update-all-impl-correct* [*rewrite*]:
∀ i∈keys-of M. i < n ⟹ map-update-all-impl f M n = map-update-all f M **by** *auto2*

lemma *keys-of-map-update-all* [*rewrite*]:
keys-of (map-update-all f M) = keys-of M **by** *auto2*

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)*
setup ‹*fold add-rewrite-rule* @{*thms itrev.simps*}›

lemma *itrev-eq-rev*: *itrev x [] = rev x*
@proof
@induct *x* **for** *∀ y. itrev x y = rev x @ y* **arbitrary** *y*
@qed

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)*
setup ‹*fold add-rewrite-rule* @{*thms strict-sorted.simps*}›

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

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

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

lemma *strict-sorted-distinct* [*forward*]: *strict-sorted l* ⟹ *distinct l*
@**proof** @**induct** *l* @**qed**

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*)
setup ‹*fold add-rewrite-rule* @{*thms ordered-insert.simps*}›

lemma *ordered-insert-set* [*rewrite*]:
set (*ordered-insert x ys*) = {*x*} ∪ *set ys*
@**proof** @**induct** *ys* @**qed**

lemma *ordered-insert-sorted* [*forward*]:
strict-sorted ys ⟹ *strict-sorted* (*ordered-insert x ys*)
@**proof** @**induct** *ys* @**qed**

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** @**induct** *xs* @**qed**

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*)
setup ‹*fold add-rewrite-rule* @{*thms remove-elt-list.simps*}›

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

lemma *remove-elt-list-sorted* [*forward*]:

strict-sorted ys $\Longrightarrow$ *strict-sorted* (*remove-elt-list x ys*)
@proof @induct *ys* **@qed**

lemma *remove-elt-idem* [*rewrite*]:
x $\notin$ *set ys* $\Longrightarrow$ *remove-elt-list x ys* = *ys*
@proof @induct *ys* **@qed**

lemma *remove-elt-list-binary* [*rewrite*]:
strict-sorted (*xs* @ *a* # *ys*) $\Longrightarrow$ *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 @induct *xs* **@***with*
@subgoal *xs* = []
@case *x* < *a* **@***with* **@have** *x* $\notin$ *set ys* **@end**
@endgoal @end
@qed

3.5 Ordered insertion into list of pairs

fun *ordered-insert-pairs* :: *'a*::*ord* $\Rightarrow$ *'b* $\Rightarrow$ (*'a* $\times$ *'b*) *list* $\Rightarrow$ (*'a* $\times$ *'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*)
setup ‹*fold add-rewrite-rule* @{*thms ordered-insert-pairs.simps*}›

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

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

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

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 @induct *xs* **@qed**

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*)
setup ‹*fold add-rewrite-rule* @{*thms remove-elt-pairs.simps*}›

lemma *remove-elt-pairs-map* [*rewrite*]:
strict-sorted (*map fst ys*) ⟹ *map-of-alist* (*remove-elt-pairs x ys*) = *delete-map x* (*map-of-alist ys*)
@**proof** @**induct** *ys* @*with*
@**subgoal** *ys* = *y* # *ys*′
@**case** *fst y* = *x* @*with* @**have** *x* ∉ *set* (*map fst ys*′) @**end**
@**endgoal** @**end**
@**qed**

lemma *remove-elt-pairs-on-set* [*rewrite*]:
strict-sorted (*map fst ys*) ⟹ *set* (*map fst* (*remove-elt-pairs x ys*)) = *set* (*map fst ys*) − {*x*}
@**proof** @**induct** *ys* @**qed**

lemma *remove-elt-pairs-sorted* [*backward*]:
strict-sorted (*map fst ys*) ⟹ *strict-sorted* (*map fst* (*remove-elt-pairs x ys*))
@**proof** @**induct** *ys* @**qed**

lemma *remove-elt-pairs-idem* [*rewrite*]:
x ∉ *set* (*map fst ys*) ⟹ *remove-elt-pairs x ys* = *ys*
@**proof** @**induct** *ys* @**qed**

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** @**induct** *xs* @*with*
@**subgoal** *xs* = []
@**case** *x* < *fst a* @*with* @**have** *x* ∉ *set* (*map fst ys*) @**end**
@**endgoal** @**end**
@**qed**

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** @**induct** *xs* @*with*
@**subgoal** *xs* = []
@**case** *x* ∉ *set* (*map fst ys*)
@**endgoal** @**end**
@**qed**

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*)

setup ‹*add-resolve-prfstep* @{*thm tree.distinct*(*1*)}›
setup ‹*fold add-rewrite-rule* @{*thms tree.sel*}›
setup ‹*add-forward-prfstep* @{*thm tree.collapse*}›
setup ‹*add-var-induct-rule* @{*thm tree.induct*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms in-traverse.simps*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms tree-set.simps*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms in-traverse-pairs.simps*}›

lemma *in-traverse-fst* [*rewrite*]:
  map fst (*in-traverse-pairs t*) = *in-traverse t*
@**proof** @**induct** *t* @**qed**

definition *tree-map* :: (*'a*, *'b*) *tree* ⇒ (*'a*, *'b*) *map* **where**
  tree-map t = *map-of-alist* (*in-traverse-pairs t*)
setup ‹*add-rewrite-rule* @{*thm tree-map-def*}›

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*)

$\wedge$ *tree-sorted l* $\wedge$ *tree-sorted r*)
setup ‹*fold add-rewrite-rule* @{*thms tree-sorted.simps*}›

lemma *tree-sorted-lr* [*forward*]:
tree-sorted (*Node l k v r*) $\Longrightarrow$ *tree-sorted l* $\wedge$ *tree-sorted r* **by** *auto2*

lemma *inorder-preserve-set* [*rewrite*]:
tree-set t = *set* (*in-traverse t*)
@**proof** @**induct** *t* @**qed**

lemma *inorder-pairs-sorted* [*rewrite*]:
tree-sorted t $\longleftrightarrow$ *strict-sorted* (*map fst* (*in-traverse-pairs t*))
@**proof** @**induct** *t* @**qed**

Use definition in terms of in_traverse from now on.

setup ‹*fold del-prfstep-thm* (@{*thms tree-set.simps*} @ @{*thms tree-sorted.simps*})›

4.4 Rotation on trees

definition *rotateL* :: ('*a*, '*b*) *tree* $\Rightarrow$ ('*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* $\Rightarrow$ ('*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* **by** *auto2*
lemma *rotateR-in-trav* [*rewrite*]: *in-traverse* (*rotateR t*) = *in-traverse t* **by** *auto2*

lemma *rotateL-sorted* [*forward*]: *tree-sorted t* $\Longrightarrow$ *tree-sorted* (*rotateL t*) **by** *auto2*
lemma *rotateR-sorted* [*forward*]: *tree-sorted t* $\Longrightarrow$ *tree-sorted* (*rotateR t*) **by** *auto2*

4.5 Insertion on trees

fun *tree-insert* :: '*a*::*ord* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *tree* $\Rightarrow$ ('*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*))
setup ‹*fold add-rewrite-rule* @{*thms tree-insert.simps*}›

lemma *insert-in-traverse-pairs* [*rewrite*]:
tree-sorted t $\Longrightarrow$ *in-traverse-pairs* (*tree-insert x v t*) = *ordered-insert-pairs x v*
(*in-traverse-pairs t*)
@**proof** @**induct** *t* @**qed**

Correctness results for insertion.

theorem *insert-sorted* [*forward*]:
tree-sorted t $\Longrightarrow$ *tree-sorted* (*tree-insert x v t*) **by** *auto2*

theorem *insert-on-map*:
tree-sorted t $\Longrightarrow$ *tree-map* (*tree-insert x v t*) = (*tree-map t*) $\{x \rightarrow v\}$ **by** *auto2*

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*))
setup ‹*add-rewrite-rule* @{*thm del-min.simps*(*2*)}›

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** @**induct** *t* @**qed**

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*)))
setup ‹*add-rewrite-rule* @{*thm delete-elt-tree.simps*(*2*)}›

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* **by** *auto2*

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*))
setup ‹*fold add-rewrite-rule* @{*thms tree-delete.simps*}›

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** @**induct** *t* @**qed**

Correctness results for deletion.

theorem *tree-delete-sorted* [*forward*]:
tree-sorted t ⟹ *tree-sorted* (*tree-delete x t*) **by** *auto2*

theorem *tree-delete-map* [*rewrite*]:
tree-sorted t $\Longrightarrow$ *tree-map* (*tree-delete x t*) = *delete-map x* (*tree-map t*) **by** *auto2*

4.7 Search on sorted trees

fun *tree-search* :: ('*a*::*ord*, '*b*) *tree* $\Rightarrow$ '*a* $\Rightarrow$ '*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)
setup ‹*fold add-rewrite-rule* @{*thms tree-search.simps*}›

Correctness of search.

theorem *tree-search-correct* [*rewrite*]:
tree-sorted t $\Longrightarrow$ *tree-search t x* = (*tree-map t*)⟨*x*⟩
@**proof** @**induct** *t* @**qed**

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* $\Rightarrow$ *bool* **where** [*rewrite*]:
part-equiv R $\longleftrightarrow$ *sym R* ∧ *trans R*

lemma *part-equivI* [*forward*]: *sym R* $\Longrightarrow$ *trans R* $\Longrightarrow$ *part-equiv R* **by** *auto2*
lemma *part-equivD1* [*forward*]: *part-equiv R* $\Longrightarrow$ *sym R* **by** *auto2*
lemma *part-equivD2* [*forward*]: *part-equiv R* $\Longrightarrow$ *trans R* **by** *auto2*
setup ‹*del-prfstep-thm-eqforward* @{*thm part-equiv-def*}›

5.1 Combining two elements in a partial equivalence relation

definition *per-union* :: ('*a* × '*a*) *set* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ ('*a* × '*a*) *set* **where** [*rewrite*]:
per-union R a b = *R* ∪ { (*x*,*y*). (*x*,*a*)∈*R* ∧ (*b*,*y*)∈*R* } ∪ { (*x*,*y*). (*x*,*b*)∈*R* ∧ (*a*,*y*)∈*R* }

lemma *per-union-memI1* [*backward*]:
(*x*, *y*) ∈ *R* $\Longrightarrow$ (*x*, *y*) ∈ *per-union R a b* **by** (*simp add*: *per-union-def*)
setup ‹*add-forward-prfstep-cond* @{*thm per-union-memI1*} [*with-term per-union ?R ?a ?b*]›

lemma *per-union-memI2* [*backward*]:

$(x,\ a) \in R \Longrightarrow (b,\ y) \in R \Longrightarrow (x,\ y) \in \textit{per-union}\ R\ a\ b$ **by** (*simp add*: *per-union-def*)

lemma *per-union-memI3* [*backward*]:
$(x,\ b) \in R \Longrightarrow (a,\ y) \in R \Longrightarrow (x,\ y) \in \textit{per-union}\ R\ a\ b$ **by** (*simp add*: *per-union-def*)

lemma *per-union-memD*:
$(x,\ y) \in \textit{per-union}\ R\ a\ b \Longrightarrow (x,\ y) \in R \vee ((x,\ a) \in R \wedge (b,\ y) \in R) \vee ((x,\ b) \in R \wedge (a,\ y) \in R)$
by (*simp add*: *per-union-def*)
setup ‹*add-forward-prfstep-cond* @{*thm per-union-memD*} [*with-cond* *?x* ≠ *?y*, *with-filt* (*order-filter x y*)]›
setup ‹*del-prfstep-thm* @{*thm per-union-def*}›

lemma *per-union-is-trans* [*forward*]:
trans R ⟹ *trans* (*per-union R a b*) **by** *auto2*

lemma *per-union-is-part-equiv* [*forward*]:
part-equiv R ⟹ *part-equiv* (*per-union R a b*) **by** *auto2*

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*)) **by** *auto*

setup ‹*register-wellform-data* (*rep-of l i*, [*i* < *length l*])›
setup ‹*add-backward-prfstep* @{*thm rep-of.domintros*}›
setup ‹*add-rewrite-rule* @{*thm rep-of.psimps*}›
setup ‹*add-prop-induct-rule* @{*thm rep-of.pinduct*}›

definition *ufa-invar* :: *nat list* ⇒ *bool* **where** [*rewrite*]:
ufa-invar l = (∀ *i*<*length l*. *rep-of-dom* (*l*, *i*) ∧ *l* ! *i* < *length l*)

lemma *ufa-invarD*:
ufa-invar l ⟹ *i* < *length l* ⟹ *rep-of-dom* (*l*, *i*) ∧ *l* ! *i* < *length l* **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm ufa-invarD*} [*with-term* *?l* ! *?i*]›
setup ‹*del-prfstep-thm-eqforward* @{*thm ufa-invar-def*}›

lemma *rep-of-id* [*rewrite*]: *ufa-invar l* $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *l* ! *i* = *i* $\Longrightarrow$ *rep-of l i* = *i* **by** *auto2*

lemma *rep-of-iff* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *rep-of l i* = (*if l* ! *i* = *i then i else rep-of l* (*l* ! *i*)) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm rep-of.psimps*}›

lemma *rep-of-min* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *l* ! (*rep-of l i*) = *rep-of l i*
@**proof** @**prop-induct** *rep-of-dom* (*l*, *i*) @**qed**

lemma *rep-of-induct*:
ufa-invar l $\wedge$ *i* < *length l* $\Longrightarrow$
$\forall$ *i*<*length l*. *l* ! *i* = *i* $\longrightarrow$ *P l i* $\Longrightarrow$
$\forall$ *i*<*length l*. *l* ! *i* $\neq$ *i* $\longrightarrow$ *P l* (*l* ! *i*) $\longrightarrow$ *P l i* $\Longrightarrow$ *P l i*
@**proof** @**prop-induct** *rep-of-dom* (*l*, *i*) @**qed**
setup ‹*add-prop-induct-rule* @{*thm rep-of-induct*}›

lemma *rep-of-bound* [*forward-arg1*]:
ufa-invar l $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *rep-of l i* < *length l*
@**proof** @**prop-induct** *ufa-invar l* $\wedge$ *i* < *length l* @**qed**

lemma *rep-of-idem* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *rep-of l* (*rep-of l i*) = *rep-of l i* **by** *auto2*

lemma *rep-of-idx* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *i* < *length l* $\Longrightarrow$ *rep-of l* (*l* ! *i*) = *rep-of l i* **by** *auto2*

definition *ufa-*α :: *nat list* $\Rightarrow$ (*nat* $\times$ *nat*) *set* **where** [*rewrite*]:
ufa-α *l* = {(*x*, *y*). *x* < *length l* $\wedge$ *y* < *length l* $\wedge$ *rep-of l x* = *rep-of l y*}

lemma *ufa-*α*-memI* [*backward*, *forward-arg*]:
x < *length l* $\Longrightarrow$ *y* < *length l* $\Longrightarrow$ *rep-of l x* = *rep-of l y* $\Longrightarrow$ (*x*, *y*) $\in$ *ufa-*α *l*
by (*simp add*: *ufa-*α*-def*)

lemma *ufa-*α*-memD* [*forward*]:
(*x*, *y*) $\in$ *ufa-*α *l* $\Longrightarrow$ *x* < *length l* $\wedge$ *y* < *length l* $\wedge$ *rep-of l x* = *rep-of l y*
by (*simp add*: *ufa-*α*-def*)
setup ‹*del-prfstep-thm* @{*thm ufa-*α*-def*}›

lemma *ufa-*α*-equiv* [*forward*]: *part-equiv* (*ufa-*α *l*) **by** *auto2*

lemma *ufa-*α*-refl* [*rewrite*]: (*i*, *i*) $\in$ *ufa-*α *l* $\longleftrightarrow$ *i* < *length l* **by** *auto2*

6.2 Operations on rep_of array

definition *uf-init-rel* :: *nat* $\Rightarrow$ (*nat* $\times$ *nat*) *set* **where** [*rewrite*]:

uf-init-rel n = ufa-α [0..<n]

lemma *ufa-init-invar* [*resolve*]: *ufa-invar [0..<n]* **by** *auto2*

lemma *ufa-init-correct* [*rewrite*]:
(x, y) ∈ uf-init-rel n ⟷ (x = y ∧ x < n)
@**proof** @**have** *ufa-invar [0..<n]* @**qed**

abbreviation *ufa-union* :: *nat list ⇒ nat ⇒ nat ⇒ nat list* **where**
ufa-union l x y ≡ l[rep-of l x := rep-of l y]

lemma *ufa-union-invar* [*forward-arg*]:
ufa-invar l ⟹ x < length l ⟹ y < length l ⟹ l' = ufa-union l x y ⟹ ufa-invar l'
@**proof**
@**have** *∀ i<length l'. rep-of-dom (l', i) ∧ l' ! i < length l'* @*with*
@**prop-induct** *ufa-invar l ∧ i < length l*
@**end**
@**qed**

lemma *ufa-union-aux* [*rewrite*]:
ufa-invar l ⟹ x < length l ⟹ y < length l ⟹ l' = ufa-union l x y ⟹
i < length l' ⟹ rep-of l' i = (if rep-of l i = rep-of l x then rep-of l y else rep-of l i)
@**proof** @**prop-induct** *ufa-invar l ∧ i < length l* @**qed**

Correctness of union operation.

theorem *ufa-union-correct* [*rewrite*]:
ufa-invar l ⟹ x < length l ⟹ y < length l ⟹ l' = ufa-union l x y ⟹
ufa-α l' = per-union (ufa-α l) x y
@**proof**
@**have** *∀ a b. (a,b) ∈ ufa-α l' ⟷ (a,b) ∈ per-union (ufa-α l) x y* @*with*
@**case** *(a,b) ∈ ufa-α l'* @*with*
@**case** *rep-of l a = rep-of l x*
@**case** *rep-of l a = rep-of l y*
@**end**
@**end**
@**qed**

abbreviation *ufa-compress* :: *nat list ⇒ nat ⇒ nat list* **where**
ufa-compress l x ≡ l[x := rep-of l x]

lemma *ufa-compress-invar* [*forward-arg*]:
ufa-invar l ⟹ x < length l ⟹ l' = ufa-compress l x ⟹ ufa-invar l'
@**proof**
@**have** *∀ i<length l'. rep-of-dom (l', i) ∧ l' ! i < length l'* @*with*
@**prop-induct** *ufa-invar l ∧ i < length l*
@**end**
@**qed**

lemma *ufa-compress-aux* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *x* < *length l* $\Longrightarrow$ *l'* = *ufa-compress l x* $\Longrightarrow$ *i* < *length l'* $\Longrightarrow$
rep-of l' i = *rep-of l i*
@**proof** @**prop-induct** *ufa-invar l* $\wedge$ *i* < *length l* @**qed**

Correctness of compress operation.

theorem *ufa-compress-correct* [*rewrite*]:
ufa-invar l $\Longrightarrow$ *x* < *length l* $\Longrightarrow$ *ufa-α* (*ufa-compress l x*) = *ufa-α l* **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm rep-of-iff*}›

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*)
setup ‹*fold add-rewrite-rule* @{*thms is-path.simps*}›

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* $\Longrightarrow$ *p* ≠ [] **by** *auto2*
lemma *nonempty-is-not-path* [*resolve*]: ¬*is-path n S* [] **by** *auto2*

lemma *is-path-extend* [*forward*]:
is-path n S p $\Longrightarrow$ *S* ⊆ *T* $\Longrightarrow$ *is-path n T p*
@**proof** @**induct** *p* @**qed**

lemma *has-path-extend* [*forward*]:
has-path n S i j $\Longrightarrow$ *S* ⊆ *T* $\Longrightarrow$ *has-path n T i j* **by** *auto2*

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*
setup ‹*register-wellform-data* (*path-join p q*, [*joinable p q*])›
setup ‹*add-prfstep-check-req* (*path-join p q*, *joinable p q*)›

lemma *path-join-hd* [*rewrite*]: $p \neq [] \Longrightarrow hd\ (path\text{-}join\ p\ q) = hd\ p$ **by** *auto2*

lemma *path-join-last* [*rewrite*]: $joinable\ p\ q \Longrightarrow q \neq [] \Longrightarrow last\ (path\text{-}join\ p\ q) = last\ q$
@**proof** @**have** $q = hd\ q\ \#\ tl\ q$ @**case** $tl\ q = []$ @**qed**

lemma *path-join-is-path* [*backward*]:
$joinable\ p\ q \Longrightarrow is\text{-}path\ n\ S\ p \Longrightarrow is\text{-}path\ n\ S\ q \Longrightarrow is\text{-}path\ n\ S\ (path\text{-}join\ p\ q)$
@**proof** @**induct** p @**qed**

lemma *has-path-trans* [*forward*]:
$has\text{-}path\ n\ S\ i\ j \Longrightarrow has\text{-}path\ n\ S\ j\ k \Longrightarrow has\text{-}path\ n\ S\ i\ k$
@**proof**
@**obtain** p **where** $is\text{-}path\ n\ S\ p\ hd\ p = i\ last\ p = j$
@**obtain** q **where** $is\text{-}path\ n\ S\ q\ hd\ q = j\ last\ q = k$
@**have** $is\text{-}path\ n\ S\ (path\text{-}join\ p\ q)$
@**qed**

definition *is-valid-graph* :: $nat \Rightarrow (nat \times nat)\ set \Rightarrow bool$ **where** [*rewrite*]:
$is\text{-}valid\text{-}graph\ n\ S \longleftrightarrow (\forall\, p{\in}S.\ fst\ p < n \wedge snd\ p < n)$

lemma *has-path-single1* [*backward1*]:
$is\text{-}valid\text{-}graph\ n\ S \Longrightarrow (a,\ b) \in S \Longrightarrow has\text{-}path\ n\ S\ a\ b$
@**proof** @**have** $is\text{-}path\ n\ S\ [a,\ b]$ @**qed**

lemma *has-path-single2* [*backward1*]:
$is\text{-}valid\text{-}graph\ n\ S \Longrightarrow (a,\ b) \in S \Longrightarrow has\text{-}path\ n\ S\ b\ a$
@**proof** @**have** $is\text{-}path\ n\ S\ [b,\ a]$ @**qed**

lemma *has-path-refl* [*backward2*]:
$is\text{-}valid\text{-}graph\ n\ S \Longrightarrow a < n \Longrightarrow has\text{-}path\ n\ S\ a\ a$
@**proof** @**have** $is\text{-}path\ n\ S\ [a]$ @**qed**

definition *connected-rel* :: $nat \Rightarrow (nat \times nat)\ set \Rightarrow (nat \times nat)\ set$ **where**
$connected\text{-}rel\ n\ S = \{(a,b).\ has\text{-}path\ n\ S\ a\ b\}$

lemma *connected-rel-iff* [*rewrite*]:
$(a,\ b) \in connected\text{-}rel\ n\ S \longleftrightarrow has\text{-}path\ n\ S\ a\ b$ **using** *connected-rel-def* **by** *simp*

lemma *connected-rel-trans* [*forward*]:
$trans\ (connected\text{-}rel\ n\ S)$ **by** *auto2*

lemma *connected-rel-refl* [*backward2*]:
$is\text{-}valid\text{-}graph\ n\ S \Longrightarrow a < n \Longrightarrow (a,\ a) \in connected\text{-}rel\ n\ S$ **by** *auto2*

lemma *is-path-per-union* [*rewrite*]:
$is\text{-}valid\text{-}graph\ n\ (S \cup \{(a,\ b)\}) \Longrightarrow$
$has\text{-}path\ n\ (S \cup \{(a,\ b)\})\ i\ j \longleftrightarrow (i,\ j) \in per\text{-}union\ (connected\text{-}rel\ n\ S)\ a\ b$
@**proof**

```
  @let R = connected-rel n S
  @let S' = S ∪ {(a, b)} @have S ⊆ S'
  @case (i, j) ∈ per-union R a b @with
    @case (i, a) ∈ R ∧ (b, j) ∈ R @with
      @have has-path n S' i a @have has-path n S' a b @have has-path n S' b j
    @end
    @case (i, b) ∈ R ∧ (a, j) ∈ R @with
      @have has-path n S' i b @have has-path n S' b a @have has-path n S' a j
    @end
  @end
  @case has-path n S' i j @with
    @have (@rule) ∀ p. is-path n S' p ⟶ (hd p, last p) ∈ per-union R a b @with
      @induct p @with
      @subgoal p = x # xs @case xs = []
        @have (x, hd xs) ∈ per-union R a b @with
          @have is-valid-graph n S
          @case (x, hd xs) ∈ S' @with @case (x, hd xs) ∈ S @end
          @case (hd xs, x) ∈ S' @with @case (hd xs, x) ∈ S @end
        @end
      @endgoal @end
    @end
    @obtain p where is-path n S' p hd p = i last p = j
  @end
@qed

lemma connected-rel-union [rewrite]:
  is-valid-graph n (S ∪ {(a, b)}) ⟹
   connected-rel n (S ∪ {(a, b)}) = per-union (connected-rel n S) a b by auto2

lemma connected-rel-init [rewrite]:
  connected-rel n {} = uf-init-rel n
@proof
  @have is-valid-graph n {}
  @have ∀ i j. has-path n {} i j ⟷ (i, j) ∈ uf-init-rel n @with
    @case has-path n {} i j @with
      @obtain p where is-path n {} p hd p = i last p = j
      @have p = hd p # tl p
    @end
  @end
@qed

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))
setup ‹fold add-rewrite-rule @{thms connected-rel-ind.simps}›

lemma connected-rel-ind-rule [rewrite]:
```

is-valid-graph n (set es) $\Longrightarrow$ $k \leq$ *length es* $\Longrightarrow$
connected-rel-ind n es k = connected-rel n (set (take k es))
@**proof** @**induct** *k* @*with*
@**subgoal** *k = Suc m*
@**have** *is-valid-graph n (set (take (Suc m) es))*
@**endgoal** @**end**
@**qed**

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) **by** *auto2*

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* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a list* **where** [*rewrite*]:
list-swap xs i j = xs[*i* := *xs* ! *j*, *j* := *xs* ! *i*]
setup ‹*register-wellform-data* (*list-swap xs i j*, [*i* < *length xs*, *j* < *length xs*])›
setup ‹*add-prfstep-check-req* (*list-swap xs i j*, *i* < *length xs* ∧ *j* < *length xs*)›

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*)
by *auto2*
setup ‹*add-rewrite-rule-cond* @{*thm list-swap-eval*} [*with-cond ?k* ≠ *?i*, *with-cond ?k* ≠ *?j*]›

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* **by** *auto2+*

lemma *length-list-swap* [*rewrite-arg*]:
length (*list-swap xs i j*) = *length xs* **by** *auto2*

lemma *mset-list-swap* [*rewrite*]:
i < *length xs* $\Longrightarrow$ *j* < *length xs* $\Longrightarrow$ *mset* (*list-swap xs i j*) = *mset xs* **by** *auto2*

lemma *set-list-swap* [*rewrite*]:

i < length xs ⟹ j < length xs ⟹ set (list-swap xs i j) = set xs **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm list-swap-def*}›
setup ‹*add-rewrite-rule-back* @{*thm list-swap-def*}›

8.2 Reverse

lemma *rev-nth* [*rewrite*]:
n < length xs ⟹ rev xs ! n = xs ! (length xs − 1 − n)
@**proof** @**induct** *xs* @**qed**

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)
setup ‹*register-wellform-data* (*rev-swap xs i j*, [*j < length xs*])›
setup ‹*add-prfstep-check-req* (*rev-swap xs i j*, *j < length xs*)›

lemma *rev-swap-length* [*rewrite-arg*]:
j < length xs ⟹ length (rev-swap xs i j) = length xs
@**proof** @**fun-induct** *rev-swap xs i j* @**unfold** *rev-swap xs i j* @**qed**

lemma *rev-swap-eval* [*rewrite*]:
j < length xs ⟹ (rev-swap xs i j) ! k =
(if k < i then xs ! k else if k > j then xs ! k else xs ! (j − (k − i)))
@**proof** @**fun-induct** *rev-swap xs i j* @**unfold** *rev-swap xs i j*
@**case** *i < j* @*with*
@**case** *k < i* @**case** *k > j* @**have** *j − (k − i) = j − k + i*
@**end**
@**qed**

lemma *rev-swap-is-rev* [*rewrite*]:
length xs ≥ 1 ⟹ rev-swap xs 0 (length xs − 1) = rev xs **by** *auto2*

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)*
setup ‹*fold add-rewrite-rule* @{*thms array-copy.simps*}›
setup ‹*register-wellform-data* (*array-copy xs xs' n*, [*n ≤ length xs*, *n ≤ length xs'*])›
setup ‹*add-prfstep-check-req* (*array-copy xs xs' n*, *n ≤ length xs ∧ n ≤ length xs'*)›

lemma *array-copy-length* [*rewrite-arg*]:
n ≤ length xs ⟹ n ≤ length xs' ⟹ length (array-copy xs xs' n) = length xs'
@**proof** @**induct** *n* @**qed**

lemma *array-copy-ind* [*rewrite*]:
n ≤ length xs ⟹ n ≤ length xs' ⟹ k < n ⟹ (array-copy xs xs' n) ! k = xs ! k
@**proof** @**induct** *n* @**qed**

lemma *array-copy-correct* [*rewrite*]:
n ≤ *length xs* ⟹ *n* ≤ *length xs'* ⟹ *take n* (*array-copy xs xs' n*) = *take n xs*
by *auto2*

8.4 Sublist

definition *sublist* :: *nat* ⇒ *nat* ⇒ *'a list* ⇒ *'a list* **where** [*rewrite*]:
sublist l r xs = *drop l* (*take r xs*)
setup ‹*register-wellform-data* (*sublist l r xs*, [*l* ≤ *r*, *r* ≤ *length xs*])›
setup ‹*add-prfstep-check-req* (*sublist l r xs*, *l* ≤ *r* ∧ *r* ≤ *length xs*)›

lemma *length-sublist* [*rewrite-arg*]:
r ≤ *length xs* ⟹ *length* (*sublist l r xs*) = *r* − *l* **by** *auto2*

lemma *nth-sublist* [*rewrite*]:
r ≤ *length xs* ⟹ *xs'* = *sublist l r xs* ⟹ *i* < *length xs'* ⟹ *xs'* ! *i* = *xs* ! (*i* + *l*) **by** *auto2*

lemma *sublist-nil* [*rewrite*]:
r ≤ *length xs* ⟹ *r* ≤ *l* ⟹ *sublist l r xs* = [] **by** *auto2*

lemma *sublist-0* [*rewrite*]:
sublist 0 l xs = *take l xs* **by** *auto2*

lemma *sublist-drop* [*rewrite*]:
sublist l r (*drop n xs*) = *sublist* (*l* + *n*) (*r* + *n*) *xs* **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm sublist-def*}›

lemma *sublist-single* [*rewrite*]:
l + *1* ≤ *length xs* ⟹ *sublist l* (*l* + *1*) *xs* = [*xs* ! *l*]
@**proof** @**have** *length* [*xs* ! *l*] = *1* @**qed**

lemma *sublist-append* [*rewrite*]:
l ≤ *m* ⟹ *m* ≤ *r* ⟹ *r* ≤ *length xs* ⟹ *sublist l m xs* @ *sublist m r xs* = *sublist l r xs*
@**proof**
@**let** *xs1* = *sublist l r xs xs2* = *sublist l m xs xs3* = *sublist m r xs*
@**have** *length* (*xs2* @ *xs3*) = (*r* − *m*) + (*m* − *l*)
@**have** ∀ *i*<*length xs1*. *xs1* ! *i* = (*xs2* @ *xs3*) ! *i* @*with*
@**case** *i* < *length xs2*
@**have** *i* − *length xs2* < *length xs3*
@**end**
@**qed**

lemma *sublist-Cons* [*rewrite*]:
r ≤ *length xs* ⟹ *l* < *r* ⟹ *xs* ! *l* # *sublist* (*l* + *1*) *r xs* = *sublist l r xs*
@**proof**
@**have** *sublist l r xs* = *sublist l* (*l* + *1*) *xs* @ *sublist* (*l* + *1*) *r xs*

@qed

lemma *sublist-equalityI*:
i ≤ *j* ⟹ *j* ≤ *length xs* ⟹ *length xs* = *length ys* ⟹
∀ *k*. *i* ≤ *k* ⟶ *k* < *j* ⟶ *xs* ! *k* = *ys* ! *k* ⟹ *sublist i j xs* = *sublist i j ys* **by** *auto2*
setup ‹*add-backward2-prfstep-cond* @{*thm sublist-equalityI*} [*with-filt* (*order-filter xs ys*)]›

lemma *set-sublist* [*resolve*]:
j ≤ *length xs* ⟹ *x* ∈ *set* (*sublist i j xs*) ⟹ ∃ *k*. *k* ≥ *i* ∧ *k* < *j* ∧ *x* = *xs* ! *k*
@proof
@let *xs'* = *sublist i j xs*
@obtain *l* **where** *l* < *length xs'* *xs'* ! *l* = *x*
@qed

lemma *list-take-sublist-drop-eq* [*rewrite*]:
l ≤ *r* ⟹ *r* ≤ *length xs* ⟹ *take l xs* @ *sublist l r xs* @ *drop r xs* = *xs*
@proof
@have *take l xs* = *sublist 0 l xs*
@have *drop r xs* = *sublist r* (*length xs*) *xs*
@qed

8.5 Updating a set of elements in an array

definition *list-update-set* :: (*nat* ⇒ *bool*) ⇒ (*nat* ⇒ *'a*) ⇒ *'a list* ⇒ *'a list* **where** [*rewrite*]:
list-update-set S f xs = *list* (λ*i*. *if S i then f i else xs* ! *i*) (*length xs*)

lemma *list-update-set-length* [*rewrite-arg*]:
length (*list-update-set S f xs*) = *length xs* **by** *auto2*

lemma *list-update-set-nth* [*rewrite*]:
xs' = *list-update-set S f xs* ⟹ *i* < *length xs'* ⟹ *xs'* ! *i* = (*if S i then f i else xs* ! *i*) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm list-update-set-def*}›

fun *list-update-set-impl* :: (*nat* ⇒ *bool*) ⇒ (*nat* ⇒ *'a*) ⇒ *'a list* ⇒ *nat* ⇒ *'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'*)
setup ‹*fold add-rewrite-rule* @{*thms list-update-set-impl.simps*}›
setup ‹*register-wellform-data* (*list-update-set-impl S f xs n*, [*n* ≤ *length xs*])›

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

@proof @induct *n* **arbitrary** *xs* **@qed**

lemma *list-update-set-impl-correct* [*rewrite*]:
list-update-set-impl S f xs (*length xs*) = *list-update-set S f xs* **by** *auto2*

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*)
setup ‹*add-rewrite-rule* @{*thm valid-graph.simps*}›

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* ∈ *verts G* ⟷ *i* < *size G* **by** (*simp add*: *verts-def*)
lemma *card-verts* [*rewrite*]: *card* (*verts G*) = *size G* **using** *verts-def* **by** *auto*
lemma *finite-verts* [*forward*]: *finite* (*verts G*) **using** *verts-def* **by** *auto*

definition *is-path* :: *graph* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:
is-path G p ⟷ *p* ≠ [] ∧ *set p* ⊆ *verts G*

lemma *is-path-to-in-verts* [*forward*]: *is-path G p* ⟹ *hd p* ∈ *verts G* ∧ *last p* ∈ *verts G*
@proof @have *last p* ∈ *set p* **@qed**

definition *joinable* :: *graph* ⇒ *nat list* ⇒ *nat list* ⇒ *bool* **where** [*rewrite*]:

joinable G p q $\longleftrightarrow$ (*is-path G p* $\wedge$ *is-path G q* $\wedge$ *last p* = *hd q*)

definition *path-join* :: *graph* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list* **where** [*rewrite*]:
path-join G p q = *p* @ *tl q*
setup ‹*register-wellform-data* (*path-join G p q*, [*joinable G p q*])›
setup ‹*add-prfstep-check-req* (*path-join G p q*, *joinable G p q*)›

lemma *path-join-is-path*:
joinable G p q $\Longrightarrow$ *is-path G* (*path-join G p q*)
@**proof** @**have** *q* = *hd q* # *tl q* @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm path-join-is-path*} [*with-term path-join ?G ?p ?q*]›

fun *path-weight* :: *graph* $\Rightarrow$ *nat list* $\Rightarrow$ *nat* **where**
path-weight G [] = *0*
| *path-weight G* (*x* # *xs*) = (*if xs* = [] *then 0 else weight G x* (*hd xs*) + *path-weight G xs*)
setup ‹*fold add-rewrite-rule* @{*thms path-weight.simps*}›

lemma *path-weight-singleton* [*rewrite*]: *path-weight G* [*x*] = *0* **by** *auto2*
lemma *path-weight-doubleton* [*rewrite*]: *path-weight G* [*m*, *n*] = *weight G m n* **by** *auto2*

lemma *path-weight-sum* [*rewrite*]:
joinable G p q $\Longrightarrow$ *path-weight G* (*path-join G p q*) = *path-weight G p* + *path-weight G q*
@**proof** @**induct** *p* @**qed**

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

lemma *path-set-mem* [*rewrite*]:
p $\in$ *path-set G m n* $\longleftrightarrow$ *is-path G p* $\wedge$ *hd p* = *m* $\wedge$ *last p* = *n* **by** *simp*

lemma *path-join-set*: *joinable G p q* $\Longrightarrow$ *path-join G p q* $\in$ *path-set G* (*hd p*) (*last q*)
@**proof** @**have** *q* = *hd q* # *tl q* @**case** *tl q* = [] @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm path-join-set*} [*with-term path-join ?G ?p ?q*]›

9.3 Shortest paths

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

lemma *is-shortest-pathD1* [*forward*]:

is-shortest-path G m n p $\Longrightarrow$ *p* $\in$ *path-set G m n* **by** *auto2*

lemma *is-shortest-pathD2* [*forward*]:
is-shortest-path G m n p $\Longrightarrow$ *p'* $\in$ *path-set G m n* $\Longrightarrow$ *path-weight G p'* $\geq$ *path-weight G p* **by** *auto2*
setup ‹*del-prfstep-thm-eqforward* @{*thm is-shortest-path-def*}›

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

lemma *has-distI* [*forward*]: *is-shortest-path G m n p* $\Longrightarrow$ *has-dist G m n* **by** *auto2*
lemma *has-distD* [*resolve*]: *has-dist G m n* $\Longrightarrow$ $\exists$ *p. is-shortest-path G m n p* **by** *auto2*
lemma *has-dist-to-in-verts* [*forward*]: *has-dist G u v* $\Longrightarrow$ *u* $\in$ *verts G* $\wedge$ *v* $\in$ *verts G* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm has-dist-def*}›

definition *dist* :: *graph* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where** [*rewrite*]:
dist G m n = *path-weight G* (*SOME p. is-shortest-path G m n p*)
setup ‹*register-wellform-data* (*dist G m n*, [*has-dist G m n*])›

lemma *dist-eq* [*rewrite*]:
is-shortest-path G m n p $\Longrightarrow$ *dist G m n* = *path-weight G p* **by** *auto2*

lemma *distD* [*forward*]:
has-dist G m n $\Longrightarrow$ *p* $\in$ *path-set G m n* $\Longrightarrow$ *path-weight G p* $\geq$ *dist G m n* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm dist-def*}›

lemma *shortest-init* [*resolve*]: *n* $\in$ *verts G* $\Longrightarrow$ *is-shortest-path G n n* [*n*] **by** *auto2*

9.4 Interior points

List of interior points

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

lemma *int-pts-singleton* [*rewrite*]: *int-pts* [*x*] = {} **by** *auto2*
lemma *int-pts-doubleton* [*rewrite*]: *int-pts* [*x*, *y*] = {*x*} **by** *auto2*

definition *path-set-on* :: *graph* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat set* $\Rightarrow$ *nat list set* **where**
path-set-on G m n V = {*p*. *p* $\in$ *path-set G m n* $\wedge$ *int-pts p* $\subseteq$ *V*}

lemma *path-set-on-mem* [*rewrite*]:
p $\in$ *path-set-on G m n V* $\longleftrightarrow$ *p* $\in$ *path-set G m n* $\wedge$ *int-pts p* $\subseteq$ *V* **by** (*simp add*: *path-set-on-def*)

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-shortest-path-on G m n p V $\longleftrightarrow$
($p \in$ *path-set-on G m n V* $\wedge$ ($\forall p' \in$*path-set-on G m n V*. *path-weight G p'* $\geq$ *path-weight G p*))

lemma *is-shortest-path-onD1* [*forward*]:
is-shortest-path-on G m n p V $\Longrightarrow$ $p \in$ *path-set-on G m n V* **by** *auto2*

lemma *is-shortest-path-onD2* [*forward*]:
is-shortest-path-on G m n p V $\Longrightarrow$ $p' \in$ *path-set-on G m n V* $\Longrightarrow$ *path-weight G p'* $\geq$ *path-weight G p* **by** *auto2*
setup ‹*del-prfstep-thm-eqforward* @{*thm is-shortest-path-on-def*}›

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

lemma *has-dist-onI* [*forward*]: *is-shortest-path-on G m n p V* $\Longrightarrow$ *has-dist-on G m n V* **by** *auto2*
lemma *has-dist-onD* [*resolve*]: *has-dist-on G m n V* $\Longrightarrow$ $\exists p$. *is-shortest-path-on G m n p V* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm has-dist-on-def*}›

definition *dist-on* :: *graph* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat set* $\Rightarrow$ *nat* **where** [*rewrite*]:
dist-on G m n V = *path-weight G* (*SOME p. is-shortest-path-on G m n p V*)
setup ‹*register-wellform-data* (*dist-on G m n V*, [*has-dist-on G m n V*])›

lemma *dist-on-eq* [*rewrite*]:
is-shortest-path-on G m n p V $\Longrightarrow$ *dist-on G m n V* = *path-weight G p* **by** *auto2*

lemma *dist-onD* [*forward*]:
has-dist-on G m n V $\Longrightarrow$ $p \in$ *path-set-on G m n V* $\Longrightarrow$ *path-weight G p* $\geq$ *dist-on G m n V* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm dist-on-def*}›

9.5 Two splitting lemmas

lemma *path-split1* [*backward*]: *is-path G p* $\Longrightarrow$ *hd p* $\in$ *V* $\Longrightarrow$ *last p* $\notin$ *V* $\Longrightarrow$
$\exists$ *p1 p2*. *joinable G p1 p2* $\wedge$ *p* = *path-join G p1 p2* $\wedge$ *int-pts p1* $\subseteq$ *V* $\wedge$ *hd p2* $\notin$ *V*
@**proof** @**induct** *p* @*with*
@**subgoal** *p* = *a* # *p'*
@**let** *p* = *a* # *p'*
@**case** *p'* = []
@**case** *hd p'* $\notin$ *V* @*with* @**have** *p* = *path-join G* [*a*, *hd p'*] *p'* @**end**
@**obtain** *p1 p2* **where** *joinable G p1 p2 p'* = *path-join G p1 p2 int-pts p1* $\subseteq$ *V hd p2* $\notin$ *V*
@**have** *p* = *path-join G* (*a* # *p1*) *p2*
@**endgoal** @**end**
@**qed**

lemma *path-split2* [*backward*]: *is-path G p* $\Longrightarrow$ *hd p* $\neq$ *last p* $\Longrightarrow$
  $\exists$ *q n. joinable G q* [*n*, *last p*] $\wedge$ *p* = *path-join G q* [*n*, *last p*]
@proof
  @have *p* = *butlast p* @ [*last p*]
  @have *butlast p* $\neq$ []
  @let *n* = *last* (*butlast p*)
  @have *p* = *path-join G* (*butlast p*) [*n*, *last p*]
@qed

9.6 Deriving has_dist and has_dist_on

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

lemma *derive-dist* [*backward2*]:
  known-dists G V $\Longrightarrow$
   m $\in$ *verts G* $-$ *V* $\Longrightarrow$
   $\forall$ *i*$\in$*verts G* $-$ *V. dist-on G 0 i V* $\geq$ *dist-on G 0 m V* $\Longrightarrow$
   has-dist G 0 m $\wedge$ *dist G 0 m* = *dist-on G 0 m V*
@proof
  @obtain *p* **where** *is-shortest-path-on G 0 m p V*
  @have *is-shortest-path G 0 m p @with*
    @have *p* $\in$ *path-set G 0 m*
    @have $\forall$ *p'*$\in$*path-set G 0 m. path-weight G p'* $\geq$ *path-weight G p @with*
      @obtain *p1 p2* **where** *joinable G p1 p2 p'* = *path-join G p1 p2*
                    int-pts p1 $\subseteq$ *V hd p2* $\notin$ *V*
      @let *x* = *last p1*
      @have *dist-on G 0 x V* $\geq$ *dist-on G 0 m V*
      @have *p1* $\in$ *path-set-on G 0 x V*
      @have *path-weight G p1* $\geq$ *dist-on G 0 x V*
      @have *path-weight G p'* $\geq$ *dist-on G 0 m V* + *path-weight G p2*
    @end
  @end
@qed

lemma *join-def'* [*resolve*]: *joinable G p q* $\Longrightarrow$ *path-join G p q* = *butlast p* @ *q*
@proof
  @have *p* = *butlast p* @ [*last p*]
  @have *path-join G p q* = *butlast p* @ [*last p*] @ *tl q*
@qed

lemma *int-pts-join* [*rewrite*]:
  joinable G p q $\Longrightarrow$ *int-pts* (*path-join G p q*) = *int-pts p* $\cup$ *int-pts q*
@proof **@have** *path-join G p q* = *butlast p* @ *q* **@qed**

lemma *dist-on-triangle-ineq* [*backward*]:

has-dist-on G k m V $\Longrightarrow$ *has-dist-on G k n V* $\Longrightarrow$ $V \subseteq$ *verts G* $\Longrightarrow$ $n \in$ *verts G* $\Longrightarrow$ $m \in V \Longrightarrow$
dist-on G k m V + weight G m n $\geq$ *dist-on G k n V*
@proof
@obtain *p* **where** *is-shortest-path-on G k m p V*
@let *pq = path-join G p [m, n]*
@have $V \cup \{m\} = V$
@have *pq* $\in$ *path-set-on G k n V*
@qed

lemma *derive-dist-on* [*backward2*]:
known-dists G V $\Longrightarrow$
$m \in$ *verts G* $-$ *V* $\Longrightarrow$
$\forall\, i \in$ *verts G* $-$ *V*. *dist-on G 0 i V* $\geq$ *dist-on G 0 m V* $\Longrightarrow$
$V' = V \cup \{m\} \Longrightarrow$
$n \in$ *verts G* $-$ $V' \Longrightarrow$
has-dist-on G 0 n V' $\wedge$ *dist-on G 0 n V' = min* (*dist-on G 0 n V*) (*dist-on G 0 m V + weight G m n*)
@proof
@have *has-dist G 0 m* $\wedge$ *dist G 0 m = dist-on G 0 m V*
@let *M = min* (*dist-on G 0 n V*) (*dist-on G 0 m V + weight G m n*)
@have $\forall\, p \in$ *path-set-on G 0 n V'*. *path-weight G p* $\geq$ *M* **@***with*
@obtain *q n'* **where** *joinable G q* [*n'*, *n*] *p = path-join G q* [*n'*, *n*]
@have *q* $\in$ *path-set G 0 n'*
@have $n' \in V'$
@case $n' \in V$ **@***with*
@have *dist-on G 0 n' V = dist G 0 n'*
@have *path-weight G q* $\geq$ *dist-on G 0 n' V*
@have *path-weight G p* $\geq$ *dist-on G 0 n' V + weight G n' n*
@have *dist-on G 0 n' V + weight G n' n* $\geq$ *dist-on G 0 n V*
@end
@have $n' = m$
@have *path-weight G q* $\geq$ *dist G 0 m*
@have *path-weight G p* $\geq$ *dist G 0 m + weight G m n*
@end
@case *dist-on G 0 m V + weight G m n* $\geq$ *dist-on G 0 n V* **@***with*
@obtain *p* **where** *is-shortest-path-on G 0 n p V*
@have *is-shortest-path-on G 0 n p V'* **@***with*
@have *p* $\in$ *path-set-on G 0 n V'* **@***with* **@have** $V \subseteq V'$ **@end**
@end
@end
@have *M = dist-on G 0 m V + weight G m n*
@obtain *pm* **where** *is-shortest-path-on G 0 m pm V*
@have *path-weight G pm = dist G 0 m*
@let *p = path-join G pm* [*m*, *n*]
@have *joinable G pm* [*m*, *n*]
@have *path-weight G p = path-weight G pm + weight G m n*
@have *is-shortest-path-on G 0 n p V'*
@qed

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*)
setup ‹*add-simple-datatype state*›

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

definition *known-set* :: *state* ⇒ *nat set* **where** [*rewrite*]:
known-set S = {..<*length* (*est S*)} − *unknown-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*]:
inv G S ⟷ (*let V* = *known-set S*; *W* = *unknown-set S*; *M* = *heap S in*
(*length* (*est S*) = *size G* ∧ *known-dists G V* ∧
keys-of M ⊆ *verts G* ∧
(∀ *i*∈*W*. *M*⟨*i*⟩ = *Some* (*est S* ! *i*)) ∧
(∀ *i*∈*V*. *est S* ! *i* = *dist G 0 i*) ∧
(∀ *i*∈*verts G*. *est S* ! *i* = *dist-on G 0 i V*)))

lemma *invE1* [*forward*]: *inv G S* ⟹ *length* (*est S*) = *size G* ∧ *known-dists G* (*known-set S*) ∧ *unknown-set S* ⊆ *verts G* **by** *auto2*
lemma *invE2* [*forward*]: *inv G S* ⟹ *i* ∈ *known-set S* ⟹ *est S* ! *i* = *dist G 0 i* **by** *auto2*
lemma *invE3* [*forward*]: *inv G S* ⟹ *i* ∈ *verts G* ⟹ *est S* ! *i* = *dist-on G 0 i* (*known-set S*) **by** *auto2*
lemma *invE4* [*rewrite*]: *inv G S* ⟹ *i* ∈ *unknown-set S* ⟹ (*heap S*)⟨*i*⟩ = *Some* (*est S* ! *i*) **by** *auto2*
setup ‹*del-prfstep-thm-str* @*eqforward* @{*thm inv-def*}›

lemma *inv-unknown-set* [*rewrite*]:
inv G S ⟹ *unknown-set S* = *verts G* − *known-set S* **by** *auto2*

lemma *dijkstra-end-inv* [*forward*]:
inv G S ⟹ *unknown-set S* = {} ⟹ ∀ *i*∈*verts G*. *has-dist G 0 i* ∧ *est S* ! *i* = *dist G 0 i* **by** *auto2*

9.8 Starting state

definition *dijkstra-start-state* :: *graph* ⇒ *state* **where** [*rewrite*]:
dijkstra-start-state G =
State (*list* (λ*i*. *if i* = 0 *then 0 else weight G 0 i*) (*size G*))
(*map-constr* (λ*i*. *i* > 0) (λ*i*. *weight G 0 i*) (*size G*))
setup ‹*register-wellform-data* (*dijkstra-start-state G*, [*size G* > 0])›

lemma *dijkstra-start-known-set* [*rewrite*]:

size G > 0 ⟹ known-set (dijkstra-start-state G) = {0} **by** *auto2*

lemma *dijkstra-start-unknown-set* [*rewrite*]:
size G > 0 ⟹ unknown-set (dijkstra-start-state G) = verts G − {0} **by** *auto2*

lemma *card-start-state* [*rewrite*]:
size G > 0 ⟹ card (unknown-set (dijkstra-start-state G)) = size G − 1
@**proof** @**have** *0 ∈ verts G* @**qed**

Starting start of Dijkstra's algorithm satisfies the invariant.

theorem *dijkstra-start-inv* [*backward*]:
size G > 0 ⟹ inv G (dijkstra-start-state G)
@**proof**
@**let** *V = {0::nat}*
@**have** *has-dist G 0 0 ∧ dist G 0 0 = 0 @with*
@**have** *is-shortest-path G 0 0 [0]* @**end**
@**have** *has-dist-on G 0 0 V ∧ dist-on G 0 0 V = 0 @with*
@**have** *is-shortest-path-on G 0 0 [0] V* @**end**
@**have** *V ⊆ verts G ∧ 0 ∈ V*
@**have** (@*rule*) *∀ i∈verts G. i ≠ 0 ⟶ has-dist-on G 0 i V ∧ dist-on G 0 i V = weight G 0 i @with*
@**let** *p = [0, i]*
@**have** *is-shortest-path-on G 0 i p V @with*
@**have** *p ∈ path-set-on G 0 i V*
@**have** *∀ p'∈path-set-on G 0 i V. path-weight G p' ≥ weight G 0 i @with*
@**obtain** *q n* **where** *joinable G q [n, last p'] p' = path-join G q [n, last p']*
@**have** *n ∈ V* @**have** *n = 0*
@**have** *path-weight G p' = path-weight G q + weight G 0 i*
@**end**
@**end**
@**end**
@**qed**

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'')
setup ‹*add-rewrite-rule* @{*thm dijkstra-step.simps*}›
setup ‹*register-wellform-data* (*dijkstra-step G m S*, [*inv G S*, *m ∈ unknown-set S*])›

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**
  @**obtain** *p* **where** *is-shortest-path-on G m n p V*
  @**let** $V' = V \cup \{x\}$
  @**have** *p* ∈ *path-set-on G m n V'* @*with* @**have** $V \subseteq V'$ @**end**
  @**have** *is-shortest-path-on G m n p V'*
@**qed**

lemma *dijkstra-step-unknown-set* [*rewrite*]:
  inv G S ⟹ *m* ∈ *unknown-set S* ⟹ *unknown-set* (*dijkstra-step G m S*) = *unknown-set S* − {*m*} **by** *auto2*

lemma *dijkstra-step-known-set* [*rewrite*]:
  inv G S ⟹ *m* ∈ *unknown-set S* ⟹ *known-set* (*dijkstra-step G m S*) = *known-set S* ∪ {*m*} **by** *auto2*

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**
  @**let** *V* = *known-set S V'* = *V* ∪ {*m*}
  @**have** (@*rule*) ∀ *i*∈*V*. *has-dist G 0 i* ∧ *has-dist-on G 0 i V'* ∧ *dist-on G 0 i V'* = *dist G 0 i*
  @**have** *has-dist G 0 m* ∧ *dist G 0 m* = *dist-on G 0 m V*
  @**have** *has-dist-on G 0 m V'* ∧ *dist-on G 0 m V'* = *dist G 0 m*
  @**have** (@*rule*) ∀ *i*∈*verts G* − *V'*. *has-dist-on G 0 i V'* ∧ *dist-on G 0 i V'* = *min* (*dist-on G 0 i V*) (*dist-on G 0 m V* + *weight G m i*)
  @**let** *S'* = *dijkstra-step G m S*
  @**have** *known-dists G V'*
  @**have** ∀ *i*∈*V'*. *est S'* ! *i* = *dist G 0 i*
  @**have** ∀ *i*∈*verts G*. *est S'* ! *i* = *dist-on G 0 i V'* @*with* @**case** *i* ∈ *V'* @**end**
@**qed**

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'* **by** *auto2*

lemma *is-dijkstra-stepD1* [*forward*]:
  inv G S ⟹ *is-dijkstra-step G S S'* ⟹ *inv G S'* **by** *auto2*

lemma *is-dijkstra-stepD2* [*forward*]:
  inv G S ⟹ *is-dijkstra-step G S S'* ⟹ *card* (*unknown-set S'*) = *card* (*unknown-set S*) − *1* **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm is-dijkstra-step-def*}›

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*)
setup ‹*add-simple-datatype interval*›

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
  fix *x y z* :: *'a interval*
  show *a*: $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    using *int-less int-less-eq* **by** *force*
  show *b*: $x \leq x$
    by (*simp add*: *int-less-eq*)
  show *c*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*smt int-less-eq dual-order.trans less-trans*)
  show *d*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    using *int-less-eq a interval.expand int-less* **by** *fastforce*
  show *e*: $x \leq y \vee y \leq x$
    by (*meson int-less-eq leI not-less-iff-gr-or-eq*)
qed 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*)
setup ‹*add-simple-datatype idx-interval*›

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
  fix *x y z* :: *'a idx-interval*
  show *a*: $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    using *iint-less iint-less-eq* **by** *force*

show *b*: $x \leq x$
by (*simp add*: *iint-less-eq*)
show *c*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
by (*smt iint-less-eq dual-order.trans less-trans*)
show *d*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
using *a idx-interval.expand iint-less iint-less-eq* **by** *auto*
show *e*: $x \leq y \vee y \leq x$
by (*meson iint-less-eq leI not-less-iff-gr-or-eq*)
qed end

lemma *interval-less-to-le-low* [*forward*]:
$(a::('a::linorder\ idx\text{-}interval)) < b \Longrightarrow low\ (int\ a) \leq low\ (int\ b)$
by (*metis eq-iff iint-less int-less less-imp-le*)

10.3 Overlapping intervals

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

definition *has-overlap* :: $('a::linorder)\ idx\text{-}interval\ set \Rightarrow 'a\ interval \Rightarrow bool$ **where** [*rewrite*]:
$has\text{-}overlap\ xs\ y \longleftrightarrow (\exists\, x{\in}xs.\ is\text{-}overlap\ (int\ x)\ y)$

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
$tmax\ Tip = 0$

setup ‹*add-resolve-prfstep* @{*thm interval-tree.distinct*(*1*)}›
setup ‹*fold add-rewrite-rule* @{*thms interval-tree.sel*}›
setup ‹*add-forward-prfstep* @{*thm interval-tree.collapse*}›
setup ‹*add-var-induct-rule* @{*thm interval-tree.induct*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms in-traverse.simps*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms tree-set.simps*}›

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*)
setup ‹*fold add-rewrite-rule* @{*thms tree-sorted.simps*}›

lemma *tree-sorted-lr* [*forward*]:
  tree-sorted (*Node l it m r*) ⟹ *tree-sorted l* ∧ *tree-sorted r* **by** *auto2*

lemma *tree-sortedD1* [*forward*]:
  tree-sorted (*Node l it m r*) ⟹ *x* ∈ *tree-set l* ⟹ *x* < *it* **by** *auto2*

lemma *tree-sortedD2* [*forward*]:
  tree-sorted (*Node l it m r*) ⟹ *x* ∈ *tree-set r* ⟹ *x* > *it* **by** *auto2*

lemma *inorder-preserve-set* [*rewrite*]:
  tree-set t = *set* (*in-traverse t*)
@**proof** @**induct** *t* @**qed**

lemma *inorder-sorted* [*rewrite*]:
  tree-sorted t ⟷ *strict-sorted* (*in-traverse t*)
@**proof** @**induct** *t* @**qed**

Use definition in terms of in_traverse from now on.

setup ‹*fold del-prfstep-thm* (@{*thms tree-set.simps*} @ @{*thms tree-sorted.simps*})›

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*) ⟷ (*tree-max-inv l* ∧ *tree-max-inv r* ∧ *m* = *max3 it* (*tmax l*) (*tmax r*))
setup ‹*fold add-rewrite-rule* @{*thms tree-max-inv.simps*}›

lemma *tree-max-is-max* [*resolve*]:

tree-max-inv t $\Longrightarrow$ *it* $\in$ *tree-set t* $\Longrightarrow$ *high* (*int it*) $\leq$ *tmax t*
@proof @induct *t* **@qed**

lemma *tmax-exists* [*backward*]:
tree-max-inv t $\Longrightarrow$ *t* $\neq$ *Tip* $\Longrightarrow$ $\exists$ *p*$\in$*tree-set t*. *high* (*int p*) = *tmax t*
@proof @induct *t* *@with*
@subgoal *t* = *Node l it m r*
@case *l* = *Tip* *@with* **@case** *r* = *Tip* **@end**
@case *r* = *Tip*
@endgoal @end
@qed

For insertion

lemma *max3-insert* [*rewrite*]: *max3 it 0 0* = *high* (*int it*) **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm max3-def*}›

11.4 Condition on the values

definition *tree-interval-inv* :: *interval-tree* $\Rightarrow$ *bool* **where** [*rewrite*]:
tree-interval-inv t $\longleftrightarrow$ ($\forall$ *p*$\in$*tree-set t*. *is-interval* (*int p*))

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

lemma *is-interval-tree-lr* [*forward*]:
is-interval-tree (*Node l x m r*) $\Longrightarrow$ *is-interval-tree l* $\wedge$ *is-interval-tree r* **by** *auto2*

11.5 Insertion on trees

fun *insert* :: *nat idx-interval* $\Rightarrow$ *interval-tree* $\Rightarrow$ *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′*)
setup ‹*fold add-rewrite-rule* @{*thms insert.simps*}›

lemma *tree-insert-in-traverse* [*rewrite*]:
tree-sorted t $\Longrightarrow$ *in-traverse* (*insert x t*) = *ordered-insert x* (*in-traverse t*)
@proof @induct *t* **@qed**

lemma *tree-insert-max-inv* [*forward*]:
tree-max-inv t $\Longrightarrow$ *tree-max-inv* (*insert x t*)
@proof @induct *t* **@qed**

Correctness of insertion.

theorem *tree-insert-all-inv* [*forward*]:
is-interval-tree t ⟹ *is-interval* (*int it*) ⟹ *is-interval-tree* (*insert it t*) **by** *auto2*

theorem *tree-insert-on-set* [*rewrite*]:
tree-sorted t ⟹ *tree-set* (*insert it t*) = {*it*} ∪ *tree-set t* **by** *auto2*

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*))
setup ‹*add-rewrite-rule* @{*thm del-min.simps*(*2*)}›
setup ‹*register-wellform-data* (*del-min t*, [*t* ≠ *Tip*])›

lemma *delete-min-del-hd*:
t ≠ *Tip* ⟹ *fst* (*del-min t*) # *in-traverse* (*snd* (*del-min t*)) = *in-traverse t*
@**proof** @**induct** *t* @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm delete-min-del-hd*} [*with-term in-traverse* (*snd* (*del-min ?t*))]›

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

lemma *delete-min-on-set*:
t ≠ *Tip* ⟹ {*fst* (*del-min t*)} ∪ *tree-set* (*snd* (*del-min t*)) = *tree-set t* **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm delete-min-on-set*} [*with-term tree-set* (*snd* (*del-min ?t*))]›

lemma *delete-min-interval-inv* [*forward-arg*]:
tree-interval-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-interval-inv* (*snd* (*del-min t*)) **by** *auto2*

lemma *delete-min-all-inv* [*forward-arg*]:
is-interval-tree t ⟹ *t* ≠ *Tip* ⟹ *is-interval-tree* (*snd* (*del-min t*)) **by** *auto2*

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′*)
setup ‹*add-rewrite-rule* @{*thm delete-elt-tree.simps*(*2*)}›

lemma *delete-elt-in-traverse* [*rewrite*]:
in-traverse (*delete-elt-tree* (*Node lt x m rt*)) = *in-traverse lt* @ *in-traverse rt* **by** *auto2*

lemma *delete-elt-max-inv* [*forward-arg*]:
tree-max-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-max-inv* (*delete-elt-tree t*) **by** *auto2*

lemma *delete-elt-on-set* [*rewrite*]:
t ≠ *Tip* ⟹ *tree-set* (*delete-elt-tree* (*Node lt x m rt*)) = *tree-set lt* ∪ *tree-set rt* **by** *auto2*

lemma *delete-elt-interval-inv* [*forward-arg*]:
tree-interval-inv t ⟹ *t* ≠ *Tip* ⟹ *tree-interval-inv* (*delete-elt-tree t*) **by** *auto2*

lemma *delete-elt-all-inv* [*forward-arg*]:
is-interval-tree t ⟹ *t* ≠ *Tip* ⟹ *is-interval-tree* (*delete-elt-tree t*) **by** *auto2*

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′*)
setup ‹*fold add-rewrite-rule* @{*thms delete.simps*}›

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

lemma *tree-delete-max-inv* [*forward*]:
tree-max-inv t ⟹ *tree-max-inv* (*delete x t*)
@**proof** @**induct** *t* @**qed**

Correctness of deletion.

theorem *tree-delete-all-inv* [*forward*]:
is-interval-tree t ⟹ *is-interval-tree* (*delete x t*)
@**proof** @**have** *tree-set* (*delete x t*) ⊆ *tree-set t* @**qed**

theorem *tree-delete-on-set* [*rewrite*]:
tree-sorted t ⟹ *tree-set* (*delete x t*) = *tree-set t* − {*x*} **by** *auto2*

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)
setup ‹fold add-rewrite-rule @{thms search.simps}›
```

Correctness of search

```
theorem search-correct [rewrite]:
  is-interval-tree t ⟹ is-interval x ⟹ search t x ⟷ has-overlap (tree-set t) x
@proof
  @induct t @with
    @subgoal t = Node l y m r
      @let t = Node l y m r
      @case is-overlap (int y) x
      @case l ≠ Tip ∧ tmax l ≥ low x @with
        @obtain p∈tree-set l where high (int p) = tmax l
        @case is-overlap (int p) x
      @end
      @case l = Tip
    @endgoal
  @end
@qed

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' by auto2

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 by auto2+
```

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′* **by** *auto2+*
setup ‹*del-prfstep-thm-eqforward* @{*thm outer-remains-def*}›

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*)
by *auto*
termination by (*relation measure* (λ(-,*l*,*r*,-). *r* − *l*)) *auto*
setup ‹*register-wellform-data* (*part1 xs l r a*, [*r* < *length xs*])›
setup ‹*add-prfstep-check-req* (*part1 xs l r a*, *r* < *length xs*)›

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** @**fun-induct** *part1 xs l r a* @**unfold** *part1 xs l r a* @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm part1-basic*} [*with-term part1 ?xs ?l ?r ?a*]›

lemma *part1-partitions1* [*backward*]:
r < *length xs* ⟹ (*rs*, *xs′*) = *part1 xs l r a* ⟹ *l* ≤ *i* ⟹ *i* < *rs* ⟹ *xs′* ! *i* ≤ *a*
@**proof** @**fun-induct** *part1 xs l r a* @**unfold** *part1 xs l r a* @**qed**

lemma *part1-partitions2* [*backward*]:
r < *length xs* ⟹ (*rs*, *xs′*) = *part1 xs l r a* ⟹ *rs* < *i* ⟹ *i* ≤ *r* ⟹ *xs′* ! *i* ≥ *a*
@**proof** @**fun-induct** *part1 xs l r a* @**unfold** *part1 xs l r a* @**qed**

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*))
setup ‹*register-wellform-data* (*partition xs l r*, [*l* < *r*, *r* < *length xs*])›

lemma *partition-basic*:
l < *r* ⟹ *r* < *length xs* ⟹ (*rs*, *xs′*) = *partition xs l r* ⟹

outer-remains xs xs' l r ∧ *mset xs'* = *mset xs* ∧ *l* ≤ *rs* ∧ *rs* ≤ *r* **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm partition-basic*} [*with-term partition ?xs ?l ?r*]›

lemma *partition-partitions1* [*forward*]:
l < *r* ⟹ *r* < *length xs* ⟹ (*rs*, *xs'*) = *partition xs l r* ⟹
x ∈ *set* (*sublist l rs xs'*) ⟹ *x* ≤ *xs'* ! *rs*
@**proof** @**obtain** *i* **where** *i* ≥ *l i* < *rs x* = *xs'* ! *i* @**qed**

lemma *partition-partitions2* [*forward*]:
l < *r* ⟹ *r* < *length xs* ⟹ (*rs*, *xs''*) = *partition xs l r* ⟹
x ∈ *set* (*sublist* (*rs* + *1*) (*r* + *1*) *xs''*) ⟹ *x* ≥ *xs''* ! *rs*
@**proof**
@**obtain** *i* **where** *i* ≥ *rs* + *1 i* < *r* + *1 x* = *xs''* ! *i*
@**let** *p* = *xs* ! *r*
@**let** *m* = *fst* (*part1 xs l* (*r* − *1*) *p*)
@**let** *xs'* = *snd* (*part1 xs l* (*r* − *1*) *p*)
@**case** *xs'* ! *m* ≤ *p*
@**qed**
setup ‹*del-prfstep-thm* @{*thm partition-def*}›

lemma *quicksort-term1*:
¬*r* ≤ *l* ⟹ ¬ *length xs* ≤ *r* ⟹ *x* = *partition xs l r* ⟹ (*p*, *xs1*) = *x* ⟹ *p* − *Suc l* < *r* − *l*
@**proof** @**have** *fst* (*partition xs l r*) − *l* − *1* < *r* − *l* @**qed**

lemma *quicksort-term2*:
¬*r* ≤ *l* ⟹ ¬ *length xs* ≤ *r* ⟹ *x* = *partition xs l r* ⟹ (*p*, *xs2*) = *x* ⟹ *r* − *Suc p* < *r* − *l*
@**proof** @**have** *r* − *fst* (*partition xs l r*) − *1* < *r* − *l* @**qed**

12.4 Quicksort function

function *quicksort* :: ('*a*::*linorder*) *list* ⇒ *nat* ⇒ *nat* ⇒ '*a list* **where**
quicksort xs l r = (
if l ≥ *r then xs*
else if r ≥ *length xs then xs*
else let
(*p*, *xs1*) = *partition xs l r*;
xs2 = *quicksort xs1 l* (*p* − *1*)
in
quicksort xs2 (*p* + *1*) *r*)
by *auto* **termination apply** (*relation measure* (λ(*a*, *l*, *r*). (*r* − *l*)))
by (*auto simp add*: *quicksort-term1 quicksort-term2*)

lemma *quicksort-basic* [*rewrite-arg*]:
mset (*quicksort xs l r*) = *mset xs* ∧ *outer-remains xs* (*quicksort xs l r*) *l r*
@**proof** @**fun-induct** *quicksort xs l r* @**unfold** *quicksort xs l r* @**qed**

lemma *quicksort-trivial1* [*rewrite*]:
  $l \geq r \Longrightarrow$ *quicksort xs l r* = *xs*
@**proof** @**unfold** *quicksort xs l r* @**qed**

lemma *quicksort-trivial2* [*rewrite*]:
  $r \geq$ *length xs* $\Longrightarrow$ *quicksort xs l r* = *xs*
@**proof** @**unfold** *quicksort xs l r* @**qed**

lemma *quicksort-permutes* [*resolve*]:
  $xs' =$ *quicksort xs l r* $\Longrightarrow$ *set* (*sublist l* (*r* + *1*) *xs'*) = *set* (*sublist l* (*r* + *1*) *xs*)
@**proof**
  @**case** $l \geq r$ @**case** $r \geq$ *length xs*
  @**have** *xs* = *take l xs* @ *sublist l* (*r* + *1*) *xs* @ *drop* (*r* + *1*) *xs*
  @**have** $xs' =$ *take l xs'* @ *sublist l* (*r* + *1*) xs' @ *drop* (*r* + *1*) xs'
  @**have** *take l xs* = *take l* xs'
  @**have** *drop* (*r* + *1*) *xs* = *drop* (*r* + *1*) xs'
@**qed**

lemma *quicksort-sorts* [*forward-arg*]:
  r < *length xs* $\Longrightarrow$ *sorted* (*sublist l* (*r* + *1*) (*quicksort xs l r*))
@**proof** @**fun-induct** *quicksort xs l r*
  @**case** $l \geq r$ @*with* @**case** *l* = *r* @**end**
  @**case** $r \geq$ *length xs*
  @**let** *p* = *fst* (*partition xs l r*)
  @**let** *xs1* = *snd* (*partition xs l r*)
  @**let** *xs2* = *quicksort xs1 l* (*p* − *1*)
  @**let** *xs3* = *quicksort xs2* (*p* + *1*) *r*
  @**have** *sorted* (*sublist l* (*r* + *1*) *xs3*) @*with*
    @**have** $l \leq p$ @**have** $p + 1 \leq r + 1$ @**have** $r + 1 \leq$ *length xs3*
    @**have** *sublist l p xs2* = *sublist l p xs3*
    @**have** *set* (*sublist l p xs1*) = *set* (*sublist l p xs2*)
    @**have** *sublist* (*p* + *1*) (*r* + *1*) *xs1* = *sublist* (*p* + *1*) (*r* + *1*) *xs2*
    @**have** *set* (*sublist* (*p* + *1*) (*r* + *1*) *xs2*) = *set* (*sublist* (*p* + *1*) (*r* + *1*) *xs3*)
    @**have** $\forall\, x \in$ *set* (*sublist l p xs3*). $x \leq$ *xs3* ! *p*
    @**have** $\forall\, x \in$ *set* (*sublist* (*p* + *1*) (*r* + *1*) *xs3*). $x \geq$ *xs3* ! *p*
    @**have** *sorted* (*sublist l p xs3*)
    @**have** *sorted* (*sublist* (*p* + *1*) (*r* + *1*) *xs3*)
    @**have** *sublist l* (*r* + *1*) *xs3* = *sublist l p xs3* @ (*xs3* ! *p*) # *sublist* (*p* + *1*) (*r* + *1*) *xs3*
  @**end**
  @**unfold** *quicksort xs l r*
@**qed**

Main result: correctness of functional quicksort.

theorem *quicksort-sorts-all* [*rewrite*]:
  xs ≠ [] $\Longrightarrow$ *quicksort xs 0* (*length xs* − *1*) = *sort xs*
@**proof**
  @**let** $xs' =$ *quicksort xs 0* (*length xs* − *1*)
  @**have** *sublist 0* (*length xs* − *1* + *1*) $xs' = xs'$

@qed

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* ⇒ *nat* **where** *s1 m* = *2* ∗ *m* + *1*
fun *s2* :: *nat* ⇒ *nat* **where** *s2 m* = *2* ∗ *m* + *2*

lemma *s-inj* [*forward*]:
s1 m = *s1 m'* ⟹ *m* = *m'* *s2 m* = *s2 m'* ⟹ *m* = *m'* **by** *auto*
lemma *s-neq* [*resolve*]:
s1 m ≠ *s2 m'* *s1 m* > *m* *s2 m* > *m* *s2 m* > *s1 m* **using** *s1.simps s2.simps* **by** *presburger+*
setup ‹*add-forward-prfstep-cond* @{*thm s-neq*(*2*)} [*with-term s1 ?m*]›
setup ‹*add-forward-prfstep-cond* @{*thm s-neq*(*3*)} [*with-term s2 ?m*]›
setup ‹*add-forward-prfstep-cond* @{*thm s-neq*(*4*)} [*with-term s2 ?m*, *with-term s1 ?m*]›

inductive *eq-pred* :: *nat* ⇒ *nat* ⇒ *bool* **where**
eq-pred n n
| *eq-pred n m* ⟹ *eq-pred n* (*s1 m*)
| *eq-pred n m* ⟹ *eq-pred n* (*s2 m*)
setup ‹*add-case-induct-rule* @{*thm eq-pred.cases*}›
setup ‹*add-prop-induct-rule* @{*thm eq-pred.induct*}›
setup ‹*add-resolve-prfstep* @{*thm eq-pred.intros*(*1*)}›
setup ‹*fold add-backward-prfstep* @{*thms eq-pred.intros*(*2*,*3*)}›

lemma *eq-pred-parent1* [*forward*]:
eq-pred i (*s1 k*) ⟹ *i* ≠ *s1 k* ⟹ *eq-pred i k*
@proof **@let** *v* = *s1 k* **@prop-induct** *eq-pred i v* **@qed**

lemma *eq-pred-parent2* [*forward*]:
eq-pred i (*s2 k*) ⟹ *i* ≠ *s2 k* ⟹ *eq-pred i k*
@proof **@let** *v* = *s2 k* **@prop-induct** *eq-pred i v* **@qed**

lemma *eq-pred-cases*:
eq-pred i j ⟹ *eq-pred* (*s1 i*) *j* ∨ *eq-pred* (*s2 i*) *j* ∨ *j* = *i* ∨ *j* = *s1 i* ∨ *j* = *s2 i*
@proof **@prop-induct** *eq-pred i j* **@qed**

setup ‹*add-forward-prfstep-cond* @{*thm eq-pred-cases*} [*with-cond ?i ≠ s1 ?k*, *with-cond ?i ≠ s2 ?k*]›

lemma *eq-pred-le* [*forward*]: *eq-pred i j* ⟹ *i* ≤ *j*
@**proof** @**prop-induct** *eq-pred i j* @**qed**

13.2 Heap property

The corresponding tree is a heap

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

lemma *is-heapD*:
is-heap xs ⟹ *j* < *length xs* ⟹ *eq-pred i j* ⟹ *snd* (*xs* ! *i*) ≤ *snd* (*xs* ! *j*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm is-heapD*} [*with-term ?xs* ! *?j*]›
setup ‹*del-prfstep-thm-eqforward* @{*thm is-heap-def*}›

13.3 Bubble-down

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

definition *is-heap-partial1* :: (*'a* × *'b*::*linorder*) *list* ⇒ *nat* ⇒ *bool* **where** [*rewrite*]:
is-heap-partial1 xs k = (∀ *i j*. *eq-pred i j* ⟶ *i* ≠ *k* ⟶ *j* < *length xs* ⟶ *snd* (*xs* ! *i*) ≤ *snd* (*xs* ! *j*))

Two cases of switching with s1 k.

lemma *bubble-down1*:
s1 k < *length xs* ⟹ *is-heap-partial1 xs k* ⟹ *snd* (*xs* ! *k*) > *snd* (*xs* ! *s1 k*) ⟹
snd (*xs* ! *s1 k*) ≤ *snd* (*xs* ! *s2 k*) ⟹ *is-heap-partial1* (*list-swap xs k* (*s1 k*)) (*s1 k*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm bubble-down1*} [*with-term list-swap ?xs ?k* (*s1 ?k*)]›

lemma *bubble-down2*:
s1 k < *length xs* ⟹ *is-heap-partial1 xs k* ⟹ *snd* (*xs* ! *k*) > *snd* (*xs* ! *s1 k*) ⟹
s2 k ≥ *length xs* ⟹ *is-heap-partial1* (*list-swap xs k* (*s1 k*)) (*s1 k*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm bubble-down2*} [*with-term list-swap ?xs ?k* (*s1 ?k*)]›

One case of switching with s2 k.

lemma *bubble-down3*:
s2 k < *length xs* ⟹ *is-heap-partial1 xs k* ⟹ *snd* (*xs* ! *s1 k*) > *snd* (*xs* ! *s2 k*) ⟹
snd (*xs* ! *k*) > *snd* (*xs* ! *s2 k*) ⟹ *xs'* = *list-swap xs k* (*s2 k*) ⟹ *is-heap-partial1 xs'* (*s2 k*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm bubble-down3*} [*with-term ?xs'*]›

13.4 Bubble-up

fun *par* :: *nat* ⇒ *nat* **where**
  par m = (*m* − *1*) *div 2*
setup ‹*register-wellform-data* (*par m*, [*m* ≠ *0*])›

lemma *ps-inverse* [*rewrite*]: *par* (*s1 k*) = *k par* (*s2 k*) = *k* **by** *simp+*

lemma *p-basic*: *m* ≠ *0* ⟹ *par m* < *m* **by** *auto*
setup ‹*add-forward-prfstep-cond* @{*thm p-basic*} [*with-term par ?m*]›

lemma *p-cases*: *m* ≠ *0* ⟹ *m* = *s1* (*par m*) ∨ *m* = *s2* (*par m*) **by** *auto*
setup ‹*add-forward-prfstep-cond* @{*thm p-cases*} [*with-term par ?m*]›

lemma *eq-pred-p-next*:
  i ≠ *0* ⟹ *eq-pred i j* ⟹ *eq-pred* (*par i*) *j*
@**proof** @**prop-induct** *eq-pred i j* @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm eq-pred-p-next*} [*with-term par ?i*]›

lemma *heap-implies-hd-min* [*resolve*]:
  is-heap xs ⟹ *i* < *length xs* ⟹ *xs* ≠ [] ⟹ *snd* (*hd xs*) ≤ *snd* (*xs* ! *i*)
@**proof**
  @**strong-induct** *i*
  @**case** *i* = *0* @**apply-induct-hyp** *par i*
  @**have** *eq-pred* (*par i*) *i*
@**qed**

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*) **by** *auto2*

lemma *bubble-up2* [*forward*]:
  k < *length xs* ⟹ *is-heap-partial2 xs k* ⟹ *snd* (*xs* ! *k*) ≥ *snd* (*xs* ! *par k*) ⟹ *k* ≠ *0* ⟹
    is-heap xs **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm p-cases*}›

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*) = (

(∀ i<*length xs*. *fst* (*xs* ! *i*) < *length m* ∧ *m* ! (*fst* (*xs* ! *i*)) = *Some i*) ∧
(∀ *i*. ∀ *k*<*length m*. *m* ! *k* = *Some i* ⟶ *i* < *length xs* ∧ *fst* (*xs* ! *i*) = *k*))
setup ‹*add-rewrite-rule* @{*thm index-of-pqueue.simps*}›

lemma *index-of-pqueueD1*:
i < *length xs* ⟹ *index-of-pqueue* (*xs*, *m*) ⟹
fst (*xs* ! *i*) < *length m* ∧ *m* ! (*fst* (*xs* ! *i*)) = *Some i* **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm index-of-pqueueD1*} [*with-term ?xs* ! *?i*]›

lemma *index-of-pqueueD2* [*forward*]:
k < *length m* ⟹ *index-of-pqueue* (*xs*, *m*) ⟹
m ! *k* = *Some i* ⟹ *i* < *length xs* ∧ *fst* (*xs* ! *i*) = *k* **by** *auto2*

lemma *index-of-pqueueD3* [*forward*]:
index-of-pqueue (*xs*, *m*) ⟹ *p* ∈ *set xs* ⟹ *fst p* < *length m*
@**proof** @**obtain** *i* **where** *i* < *length xs xs* ! *i* = *p* @**qed**
setup ‹*del-prfstep-thm-eqforward* @{*thm index-of-pqueue.simps*}›

lemma *has-index-unique-key* [*forward*]:
index-of-pqueue (*xs*, *m*) ⟹ *unique-keys-set* (*set xs*)
@**proof**
@**have** ∀ *a x y*. (*a*, *x*) ∈ *set xs* ⟶ (*a*, *y*) ∈ *set xs* ⟶ *x* = *y* @*with*
@**obtain** *i* **where** *i* < *length xs xs* ! *i* = (*a*, *x*)
@**obtain** *j* **where** *j* < *length xs xs* ! *j* = (*a*, *y*)
@**end**
@**qed**

lemma *has-index-keys-of* [*rewrite*]:
index-of-pqueue (*xs*, *m*) ⟹ *has-key-alist xs k* ⟷ (*k* < *length m* ∧ *m* ! *k* ≠ *None*)
@**proof**
@**case** *has-key-alist xs k* @*with*
@**obtain** *v'* **where** (*k*, *v'*) ∈ *set xs*
@**obtain** *i* **where** *i* < *length xs* ∧ *xs* ! *i* = (*k*, *v'*)
@**end**
@**qed**

lemma *has-index-distinct* [*forward*]:
index-of-pqueue (*xs*, *m*) ⟹ *distinct xs*
@**proof**
@**have** ∀ *i*<*length xs*. ∀ *j*<*length xs*. *i* ≠ *j* ⟶ *xs* ! *i* ≠ *xs* ! *j*
@**qed**

13.6 Basic operations on indexed_queue

fun *idx-pqueue-swap-fun* :: (*nat* × *'a*) *list* × *nat option list* ⇒ *nat* ⇒ *nat* ⇒ (*nat* × *'a*) *list* × *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* ⟹ *j* < *length xs* ⟹ *index-of-pqueue* (*xs*, *m*) ⟹
   index-of-pqueue (*idx-pqueue-swap-fun* (*xs*, *m*) *i j*)
@**proof** @**unfold** *idx-pqueue-swap-fun* (*xs*, *m*) *i j* @**qed**

lemma *fst-idx-pqueue-swap* [*rewrite*]:
  fst (*idx-pqueue-swap-fun* (*xs*, *m*) *i j*) = *list-swap xs i j*
@**proof** @**unfold** *idx-pqueue-swap-fun* (*xs*, *m*) *i j* @**qed**

lemma *snd-idx-pqueue-swap* [*rewrite*]:
  length (*snd* (*idx-pqueue-swap-fun* (*xs*, *m*) *i j*)) = *length m*
@**proof** @**unfold** *idx-pqueue-swap-fun* (*xs*, *m*) *i j* @**qed**

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** @**unfold** *idx-pqueue-push-fun k v* (*xs*, *m*) @**qed**

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**
  @**unfold** *idx-pqueue-pop-fun* (*xs*, *m*)
  @**have** *length xs* = *length* (*butlast xs*) + *1*
  @**have** *fst* (*xs* ! *length* (*butlast xs*)) < *length m*
@**qed**

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))
  by pat-completeness auto
  termination by (relation measure (λ((xs,-),k). (length xs − k))) (simp-all,
auto2+)

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 @fun-induct idx-bubble-down-fun x k @with
  @subgoal (x = (xs, m), k = k)
  @unfold idx-bubble-down-fun (xs, m) k
  @case s2 k < length xs @with
    @case snd (xs ! s1 k) ≤ snd (xs ! s2 k)
  @end
  @case s1 k < length xs @end
@qed
setup ‹add-forward-prfstep-cond @{thm idx-bubble-down-fun-correct} [with-term
?r]›

lemma idx-bubble-down-fun-correct2 [forward]:
  index-of-pqueue x ⟹ index-of-pqueue (idx-bubble-down-fun x k)
@proof @fun-induct idx-bubble-down-fun x k @with
  @subgoal (x = (xs, m), k = k)
  @unfold idx-bubble-down-fun (xs, m) k
  @case s2 k < length xs @with
    @case snd (xs ! s1 k) ≤ snd (xs ! s2 k)
  @end
  @case s1 k < length xs @end
@qed

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 ⟹ is-heap-partial2 (fst x) k ⟹
   is-heap (fst r) ∧ mset (fst r) = mset (fst x) ∧ length (snd r) = length (snd x)
@proof @fun-induct idx-bubble-up-fun x k @with
  @subgoal (x = (xs, m), k = k)
```

@**unfold** *idx-bubble-up-fun* (*xs*, *m*) *k* @**end**
@**qed**
setup ‹*add-forward-prfstep-cond* @{*thm idx-bubble-up-fun-correct*} [*with-term ?r*]›

lemma *idx-bubble-up-fun-correct2* [*forward*]:
index-of-pqueue x ⟹ *index-of-pqueue* (*idx-bubble-up-fun x k*)
@**proof** @**fun-induct** *idx-bubble-up-fun x k* @*with*
@**subgoal** (*x* = (*xs*, *m*), *k* = *k*)
@**unfold** *idx-bubble-up-fun* (*xs*, *m*) *k* @**end**
@**qed**

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*) ⟹ *xs* ≠ [] ⟹ *res* = *delete-min-idx-pqueue-fun* (*xs*, *m*)
⟹
index-of-pqueue (*snd res*)
@**proof** @**unfold** *delete-min-idx-pqueue-fun* (*xs*, *m*) @**qed**
setup ‹*add-forward-prfstep-cond* @{*thm delete-min-idx-pqueue-correct*} [*with-term ?res*]›

lemma *hd-last-swap-eval-last* [*rewrite*]:
xs ≠ [] ⟹ *last* (*list-swap xs 0* (*length xs* − *1*)) = *hd xs*
@**proof**
@**let** *xs'* = *list-swap xs 0* (*length xs* − *1*)
@**have** *last xs'* = *xs'* ! (*length xs* − *1*)
@**qed**

Correctness of delete-min.

theorem *delete-min-idx-pqueue-correct2*:
is-heap xs ⟹ *xs* ≠ [] ⟹ *res* = *delete-min-idx-pqueue-fun* (*xs*, *m*) ⟹ *index-of-pqueue* (*xs*, *m*) ⟹
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** @**unfold** *delete-min-idx-pqueue-fun* (*xs*, *m*)
@**let** *xs'* = *list-swap xs 0* (*length xs* − *1*)
@**have** *is-heap-partial1* (*butlast xs'*) *0*
@**qed**
setup ‹*add-forward-prfstep-cond* @{*thm delete-min-idx-pqueue-correct2*} [*with-term ?res*]›

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** @**unfold** *insert-idx-pqueue-fun k v* (*xs*, *m*) @**qed**

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** @**unfold** *insert-idx-pqueue-fun k v* (*xs*, *m*)
@**have** *is-heap-partial2* (*xs* @ [(*k*, *v*)]) (*length xs*)
@**qed**
setup ‹*add-forward-prfstep-cond* @{*thm insert-idx-pqueue-correct2*} [*with-term ?r*]›

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** @**unfold** *update-idx-pqueue-fun k v* (*xs*, *m*)
@**let** *i′* = *the* (*m* ! *k*)
@**let** *xs′* = *list-update xs i′* (*k*, *v*)
@**case** *m* ! *k* = *None*
@**have** *index-of-pqueue* (*xs′*, *m*)
@**qed**

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 **@unfold** *update-idx-pqueue-fun k v* (*xs*, *m*)
@let *i* = *the* (*m* ! *k*)
@let *xs′* = *list-update xs i* (*k*, *v*)
@have *xs′* = *fst* (*xs′*, *m*)
@case *m* ! *k* = *None*
@case *snd* (*xs* ! *the* (*m* ! *k*)) ≤ *v* **@with**
@have *is-heap-partial1 xs′ i*
@end
@have *is-heap-partial2 xs′ i*
@qed
setup ‹*add-forward-prfstep-cond* @{*thm update-idx-pqueue-correct2*} [*with-term ?r*]›

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*

setup ‹*add-resolve-prfstep* @{*thm color.distinct*(*1*)}›
setup ‹*add-resolve-prfstep* @{*thm rbt.distinct*(*1*)}›
setup ‹*fold add-rewrite-rule* @{*thms rbt.sel*}›
setup ‹*add-forward-prfstep* @{*thm rbt.collapse*}›
setup ‹*add-var-induct-rule* @{*thm rbt.induct*}›

lemma *not-R* [*forward*]: *c* ≠ *R* ⟹ *c* = *B* **using** *color.exhaust* **by** *blast*
lemma *not-B* [*forward*]: *c* ≠ *B* ⟹ *c* = *R* **using** *color.exhaust* **by** *blast*
lemma *red-not-leaf* [*forward*]: *cl t* = *R* ⟹ *t* ≠ *Leaf* **by** *auto*

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*
setup ‹*fold add-rewrite-rule* @{*thms black-depth.simps*}›

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*)
setup ‹*fold add-rewrite-rule* @{*thms cl-inv.simps*}›

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*)
setup ‹*fold add-rewrite-rule* @{*thms bd-inv.simps*}›

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*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm cl-invI*} [*with-term Node ?l B ?k ?v ?r*]›

lemma *bd-invI*: *bd-inv l* ⟹ *bd-inv r* ⟹ *black-depth l* = *black-depth r* ⟹ *bd-inv* (*Node l c k v r*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm bd-invI*} [*with-term Node ?l ?c ?k ?v ?r*]›

lemma *is-rbt-rec* [*forward*]: *is-rbt* (*Node l c k v r*) ⟹ *is-rbt l* ∧ *is-rbt r*
@**proof** @**case** *c* = *R* @**qed**

14.3 Balancedness of RBT

lemma *two-distrib* [*rewrite*]: (*2*::*nat*) ∗ (*a* + *1*) = *2* ∗ *a* + *2* **by** *simp*

fun *min-depth* :: (*'a*, *'b*) *rbt* ⇒ *nat* **where**
  min-depth Leaf = *0*
| *min-depth* (*Node l c k v r*) = *min* (*min-depth l*) (*min-depth r*) + *1*
setup ‹*fold add-rewrite-rule* @{*thms min-depth.simps*}›

fun *max-depth* :: (*'a*, *'b*) *rbt* ⇒ *nat* **where**
  max-depth Leaf = *0*
| *max-depth* (*Node l c k v r*) = *max* (*max-depth l*) (*max-depth r*) + *1*
setup ‹*fold add-rewrite-rule* @{*thms max-depth.simps*}›

Balancedness of red-black trees.

theorem *rbt-balanced*: *is-rbt t* ⟹ *max-depth t* ≤ *2* ∗ *min-depth t* + *1*
@**proof**
  @**induct** *t* **for** *is-rbt t* ⟶ *black-depth t* ≤ *min-depth t* @*with*
    @**subgoal** *t* = *Node l c k v r* @**case** *c* = *R* @**endgoal**
  @**end**
  @**induct** *t* **for** *is-rbt t* ⟶ (*if cl t* = *R then max-depth t* ≤ *2* ∗ *black-depth t* + *1*
                    else max-depth t ≤ *2* ∗ *black-depth t*) @*with*
    @**subgoal** *t* = *Node l c k v r* @**case** *c* = *R* @**endgoal**
  @**end**

@**have** *max-depth t* ≤ *2* ∗ *black-depth t* + *1*
@**qed**

14.4 Definition and basic properties of cl_inv'

fun *cl-inv'* :: (*'a*, *'b*) *rbt* ⇒ *bool* **where**
cl-inv' Leaf = *True*
| *cl-inv'* (*Node l c k v r*) = (*cl-inv l* ∧ *cl-inv r*)
setup ‹*fold add-rewrite-rule* @{*thms cl-inv'.simps*}›

lemma *cl-inv'B* [*forward*, *backward1*]:
cl-inv' t ⟹ *cl t* = *B* ⟹ *cl-inv t*
@**proof** @**case** *t* = *Leaf* @**qed**

lemma *cl-inv'R* [*forward*]:
cl-inv' (*Node l R k v r*) ⟹ *cl l* = *B* ⟹ *cl r* = *B* ⟹ *cl-inv* (*Node l R k v r*)
by *auto2*

lemma *cl-inv-to-cl-inv'* [*forward*]: *cl-inv t* ⟹ *cl-inv' t*
@**proof** @**case** *t* = *Leaf* @**case** *cl t* = *R* @**qed**

lemma *cl-inv'I* [*forward-arg*]:
cl-inv l ⟹ *cl-inv r* ⟹ *cl-inv'* (*Node l c k v r*) **by** *auto*

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*
setup ‹*fold add-rewrite-rule* @{*thms rbt-in-traverse.simps*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms rbt-set.simps*}›

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*
setup ‹*fold add-rewrite-rule* @{*thms rbt-in-traverse-pairs.simps*}›

lemma *rbt-in-traverse-fst* [*rewrite*]: *map fst* (*rbt-in-traverse-pairs t*) = *rbt-in-traverse t*
@**proof** @**induct** *t* @**qed**

definition *rbt-map* :: (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *map* **where**
rbt-map t = *map-of-alist* (*rbt-in-traverse-pairs t*)
setup ‹*add-rewrite-rule* @{*thm rbt-map-def*}›

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*)
setup ‹*fold add-rewrite-rule* @{*thms rbt-sorted.simps*}›

lemma *rbt-sorted-lr* [*forward*]:
  rbt-sorted (*Node l c k v r*) ⟹ *rbt-sorted l* ∧ *rbt-sorted r* **by** *auto2*

lemma *rbt-inorder-preserve-set* [*rewrite*]:
  rbt-set t = *set* (*rbt-in-traverse t*)
@**proof** @**induct** *t* @**qed**

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

setup ‹*fold del-prfstep-thm* (@{*thms rbt-set.simps*} @ @{*thms rbt-sorted.simps*})›

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)
setup ‹*register-wellform-data* (*balance l k v r*, [*black-depth l* = *black-depth r*])›
setup ‹*add-prfstep-check-req* (*balance l k v r*, *black-depth l* = *black-depth r*)›

lemma *balance-non-Leaf* [*resolve*]: *balance l k v r* ≠ *Leaf* **by** *auto2*

lemma *balance-bdinv* [*forward-arg*]:
  bd-inv l $\Longrightarrow$ *bd-inv r* $\Longrightarrow$ *black-depth l* = *black-depth r* $\Longrightarrow$ *bd-inv* (*balance l k v r*)
@proof **@have** *bd-inv* (*balanceR l k v r*) **@qed**

lemma *balance-bd* [*rewrite*]:
  bd-inv l $\Longrightarrow$ *bd-inv r* $\Longrightarrow$ *black-depth l* = *black-depth r* $\Longrightarrow$
   black-depth (*balance l k v r*) = *black-depth l* + *1*
@proof **@have** *black-depth* (*balanceR l k v r*) = *black-depth l* + *1* **@qed**

lemma *balance-cl1* [*forward*]:
  cl-inv′ l $\Longrightarrow$ *cl-inv r* $\Longrightarrow$ *cl-inv* (*balance l k v r*) **by** *auto2*

lemma *balance-cl2* [*forward*]:
  cl-inv l $\Longrightarrow$ *cl-inv′ r* $\Longrightarrow$ *cl-inv* (*balance l k v r*) **by** *auto2*

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* **by** *auto2*

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* **by** *auto2*

setup ‹*fold del-prfstep-thm* [@{*thm balanceR-def*}, @{*thm balance-def*}]›

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*)))
setup ‹*fold add-rewrite-rule* @{*thms ins.simps*}›

lemma *ins-non-Leaf* [*resolve*]: *ins x v t* ≠ *Leaf*
@proof **@case** *t* = *Leaf* **@qed**

lemma *cl-inv-ins* [*forward*]:
  cl-inv t $\Longrightarrow$ *cl-inv′* (*ins x v t*)
@proof
  @induct *t* **for** *cl-inv t* ⟶ (*if cl t* = *B then cl-inv* (*ins x v t*) *else cl-inv′* (*ins x v t*))

@qed

lemma *bd-inv-ins*:
bd-inv t $\Longrightarrow$ *bd-inv* (*ins x v t*) $\wedge$ *black-depth t* = *black-depth* (*ins x v t*)
@proof @induct *t* **@qed**
setup ‹*add-forward-prfstep-cond* (*conj-left-th* @{*thm bd-inv-ins*}) [*with-term ins ?x ?v ?t*]›

lemma *ins-inorder-pairs* [*rewrite*]:
rbt-sorted t $\Longrightarrow$ *rbt-in-traverse-pairs* (*ins x v t*) = *ordered-insert-pairs x v* (*rbt-in-traverse-pairs t*)
@proof @induct *t* **@qed**

14.8 Paint function

fun *paint* :: *color* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
paint c Leaf = *Leaf*
| *paint c* (*Node l c' x v r*) = *Node l c x v r*
setup ‹*fold add-rewrite-rule* @{*thms paint.simps*}›
setup ‹*register-wellform-data* (*paint c t*, [*t* $\neq$ *Leaf*])›
setup ‹*add-prfstep-check-req* (*paint c t*, *t* $\neq$ *Leaf*)›

lemma *paint-cl-inv'* [*forward*]: *cl-inv' t* $\Longrightarrow$ *cl-inv'* (*paint c t*) **by** *auto2*

lemma *paint-bd-inv* [*forward*]: *bd-inv t* $\Longrightarrow$ *bd-inv* (*paint c t*) **by** *auto2*

lemma *paint-bd* [*rewrite*]:
bd-inv t $\Longrightarrow$ *t* $\neq$ *Leaf* $\Longrightarrow$ *cl t* = *B* $\Longrightarrow$ *black-depth* (*paint R t*) = *black-depth t* − *1* **by** *auto2*

lemma *paint-in-traverse-pairs* [*rewrite*]:
rbt-in-traverse-pairs (*paint c t*) = *rbt-in-traverse-pairs t* **by** *auto2*

14.9 Insert function

definition *rbt-insert* :: '*a*::*order* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*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 $\Longrightarrow$ *is-rbt* (*rbt-insert x v t*) **by** *auto2*

theorem *insert-sorted* [*forward*]:
rbt-sorted t $\Longrightarrow$ *rbt-sorted* (*rbt-insert x v t*) **by** *auto2*

theorem *insert-rbt-map* [*rewrite*]:
rbt-sorted t $\Longrightarrow$ *rbt-map* (*rbt-insert x v t*) = (*rbt-map t*) {*x* $\rightarrow$ *v*} **by** *auto2*

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)
setup ‹*fold add-rewrite-rule* @{*thms rbt-search.simps*}›

Correctness of search

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

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)
setup ‹*register-wellform-data* (*balL l k v r*, [*black-depth l* + *1* = *black-depth r*]›
setup ‹*add-prfstep-check-req* (*balL l k v r*, *black-depth l* + *1* = *black-depth r*)›

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)
setup ‹*register-wellform-data* (*balR l k v r*, [*black-depth l* = *black-depth r* + *1*]›
setup ‹*add-prfstep-check-req* (*balR l k v r*, *black-depth l* = *black-depth r* + *1*)›

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* **by** *auto2*

lemma *balL-bd'* [*forward-arg*]:

bd-inv l $\Longrightarrow$ *bd-inv r* $\Longrightarrow$ *cl-inv r* $\Longrightarrow$ *black-depth l + 1 = black-depth r* $\Longrightarrow$
bd-inv (*balL l k v r*) $\wedge$ *black-depth* (*balL l k v r*) = *black-depth l + 1* **by** *auto2*

lemma *balL-cl* [*forward-arg*]:
cl-inv' *l* $\Longrightarrow$ *cl-inv r* $\Longrightarrow$ *cl r = B* $\Longrightarrow$ *cl-inv* (*balL l k v r*) **by** *auto2*

lemma *balL-cl'* [*forward*]:
cl-inv' *l* $\Longrightarrow$ *cl-inv r* $\Longrightarrow$ *cl-inv'* (*balL l k v r*) **by** *auto2*

lemma *balR-bd* [*forward-arg*]:
bd-inv l $\Longrightarrow$ *bd-inv r* $\Longrightarrow$ *cl-inv l* $\Longrightarrow$ *black-depth l = black-depth r + 1* $\Longrightarrow$
bd-inv (*balR l k v r*) $\wedge$ *black-depth* (*balR l k v r*) = *black-depth l* **by** *auto2*

lemma *balR-cl* [*forward-arg*]:
cl-inv l $\Longrightarrow$ *cl-inv'* *r* $\Longrightarrow$ *cl l = B* $\Longrightarrow$ *cl-inv* (*balR l k v r*) **by** *auto2*

lemma *balR-cl'* [*forward*]:
cl-inv l $\Longrightarrow$ *cl-inv'* *r* $\Longrightarrow$ *cl-inv'* (*balR l k v r*) **by** *auto2*

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* **by** *auto2*

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* **by** *auto2*

setup ‹*fold del-prfstep-thm* [@{*thm balL-def*}, @{*thm balR-def*}]›

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)
setup ‹fold add-rewrite-rule @{thms combine.simps(1,2)}›

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 @fun-induct combine lt rt @with
  @subgoal (lt = Node l1 c1 k1 v1 r1, rt = Node l2 c2 k2 v2 r2)
    @unfold combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2 r2)
     @case c1 = B @with @case c2 = B @with @case cl (combine r1 l2) = B
@with
      @have cl (Node (combine r1 l2) B k2 v2 r2) = B @end @end @end
  @endgoal @end
@qed

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 @fun-induct combine lt rt @with
  @subgoal (lt = Node l1 c1 k1 v1 r1, rt = Node l2 c2 k2 v2 r2)
    @unfold combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2 r2)
     @case c1 = B @with @case c2 = B @with @case cl (combine r1 l2) = B
@with
      @have cl (Node (combine r1 l2) B k2 v2 r2) = B @end @end @end
  @endgoal @end
@qed
setup ‹add-forward-prfstep-cond @{thm combine-cl} [with-term combine ?lt ?rt]›

lemma combine-in-traverse-pairs [rewrite]:
 rbt-in-traverse-pairs (combine lt rt) = rbt-in-traverse-pairs lt @ rbt-in-traverse-pairs
rt
@proof @fun-induct combine lt rt @with
  @subgoal (lt = Node l1 c1 k1 v1 r1, rt = Node l2 c2 k2 v2 r2)
    @unfold combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2 r2)
     @case c1 = R @with @case c2 = R @with @case cl (combine r1 l2) = R
@with
      @have rbt-in-traverse-pairs (combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2
r2)) =
              rbt-in-traverse-pairs l1 @ (k1, v1) # rbt-in-traverse-pairs (combine r1
l2) @ (k2, v2) # rbt-in-traverse-pairs r2
     @end @end @end
     @case c1 = B @with @case c2 = B @with @case cl (combine r1 l2) = R
@with
      @have rbt-in-traverse-pairs (combine (Node l1 c1 k1 v1 r1) (Node l2 c2 k2 v2
```

r2)) =
        rbt-in-traverse-pairs l1 @ (*k1*, *v1*) # *rbt-in-traverse-pairs* (*combine r1 l2*) @ (*k2*, *v2*) # *rbt-in-traverse-pairs r2*
  @end @end @end
 @endgoal @end
@qed

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*))
setup ‹*add-rewrite-rule* @{*thm del.simps*(*1*)}›

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 @induct *t* @*with*
 @subgoal *t* = *Node l c k v r*
  @unfold *del x* (*Node l c k v r*)
  @case *x* = *k* **@case** *x* < *k* @*with*
   @case *l* = *Leaf* **@case** *cl l* = *B* **@end**
  @case *x* > *k* @*with*
   @case *r* = *Leaf* **@case** *cl r* = *B* **@end**
 @endgoal @end
@qed

lemma *del-cl*:
 cl-inv t ⟹ *if cl t* = *R then cl-inv* (*del x t*) *else cl-inv'* (*del x t*)
@proof @induct *t* @*with*
 @subgoal *t* = *Node l c k v r*
  @unfold *del x* (*Node l c k v r*)
  @case *x* = *k* **@case** *x* < *k*
 @endgoal @end
@qed
setup ‹*add-forward-prfstep-cond* @{*thm del-cl*} [*with-term del ?x ?t*]›

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** @**induct** *t* @*with*
  @**subgoal** *t* = *Node l c k v r*
    @**unfold** *del x* (*Node l c k v r*)
  @**endgoal** @**end**
@**qed**

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*) **by** *auto2*

theorem *delete-sorted* [*forward*]:
  rbt-sorted t ⟹ *rbt-sorted* (*delete x t*) **by** *auto2*

theorem *delete-rbt-map* [*rewrite*]:
  rbt-sorted t ⟹ *rbt-map* (*delete x t*) = *delete-map x* (*rbt-map t*) **by** *auto2*

setup ‹*del-prfstep RBTree.balance-case*›
setup ‹*del-prfstep RBTree.balL-case*›
setup ‹*del-prfstep RBTree.balR-case*›
setup ‹*del-prfstep RBTree.paint-case*›

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*)
setup ‹*add-simple-datatype rectangle*›

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*) **by** *auto2*

setup ‹*add-forward-prfstep-cond* @{*thm is-rect-listD*} [*with-term ?rects* ! *?i*]›

setup ‹*del-prfstep-thm-eqforward* @{*thm is-rect-list-def*}›

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*)
setup ‹*fold add-rewrite-rule-back* @{*thms operation.collapse*}›
setup ‹*fold add-rewrite-rule* @{*thms operation.sel*}›
setup ‹*fold add-rewrite-rule* @{*thms operation.case*}›
setup ‹*add-resolve-prfstep* @{*thm operation.distinct*(*1*)}›
setup ‹*add-forward-prfstep-cond* @{*thm operation.disc*(*1*)} [*with-term INS ?x11.0 ?x12.0 ?x13.0*]›
setup ‹*add-forward-prfstep-cond* @{*thm operation.disc*(*2*)} [*with-term DEL ?x21.0 ?x22.0 ?x23.0*]›

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
fix *x y z* :: '*a operation*
show *a*: (*x* < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)
by (*smt Rect-Intersect.less Rect-Intersect.less-eq leD le-cases3 not-less-iff-gr-or-eq*)
show *b*: *x* ≤ *x*
by (*simp add*: *local.less-eq*)
show *c*: *x* ≤ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z*
by (*smt Rect-Intersect.less Rect-Intersect.less-eq a dual-order.trans less-trans*)
show *d*: *x* ≤ *y* ⟹ *y* ≤ *x* ⟹ *x* = *y*
by (*metis Rect-Intersect.less Rect-Intersect.less-eq a le-imp-less-or-eq operation.expand*)

show *e*: $x \leq y \vee y \leq x$
using *local.less-eq* **by** *fastforce*
qed end

setup ‹*fold add-rewrite-rule* [@{*thm less-eq*}, @{*thm less*}]›

lemma *operation-leD* [*forward*]:
$(a::('a::linorder\ operation)) \leq b \Longrightarrow pos\ a \leq pos\ b$ **by** *auto2*

lemma *operation-lessI* [*backward*]:
$p1 \leq p2 \Longrightarrow INS\ p1\ n1\ i1 < DEL\ p2\ n2\ i2$
@**proof**
@**have** $is\text{-}INS\ (INS\ p1\ n1\ i1) = True$
@**have** $is\text{-}INS\ (DEL\ p2\ n2\ i2) = False$
@**qed**

setup ‹*fold del-prfstep-thm* [@{*thm less-eq*}, @{*thm less*}]›

15.3 Set of operations corresponding to a list of rectangles

fun *ins-op* :: $'a\ rectangle\ list \Rightarrow nat \Rightarrow ('a::linorder)\ operation$ **where**
$ins\text{-}op\ rects\ i = INS\ (low\ (yint\ (rects\ !\ i)))\ i\ (xint\ (rects\ !\ i))$
setup ‹*add-rewrite-rule* @{*thm ins-op.simps*}›

fun *del-op* :: $'a\ rectangle\ list \Rightarrow nat \Rightarrow ('a::linorder)\ operation$ **where**
$del\text{-}op\ rects\ i = DEL\ (high\ (yint\ (rects\ !\ i)))\ i\ (xint\ (rects\ !\ i))$
setup ‹*add-rewrite-rule* @{*thm del-op.simps*}›

definition *ins-ops* :: $'a\ rectangle\ list \Rightarrow ('a::linorder)\ operation\ list$ **where** [*rewrite*]:
$ins\text{-}ops\ rects = list\ (\lambda i.\ ins\text{-}op\ rects\ i)\ (length\ rects)$

definition *del-ops* :: $'a\ rectangle\ list \Rightarrow ('a::linorder)\ operation\ list$ **where** [*rewrite*]:
$del\text{-}ops\ rects = list\ (\lambda i.\ del\text{-}op\ rects\ i)\ (length\ rects)$

lemma *ins-ops-distinct* [*forward*]: $distinct\ (ins\text{-}ops\ rects)$
@**proof**
@**let** $xs = ins\text{-}ops\ rects$
@**have** $\forall i{<}length\ xs.\ \forall j{<}length\ xs.\ i \neq j \longrightarrow xs\ !\ i \neq xs\ !\ j$
@**qed**

lemma *del-ops-distinct* [*forward*]: $distinct\ (del\text{-}ops\ rects)$
@**proof**
@**let** $xs = del\text{-}ops\ rects$
@**have** $\forall i{<}length\ xs.\ \forall j{<}length\ xs.\ i \neq j \longrightarrow xs\ !\ i \neq xs\ !\ j$
@**qed**

lemma *set-ins-ops* [*rewrite*]:
$oper \in set\ (ins\text{-}ops\ rects) \longleftrightarrow op\text{-}idx\ oper < length\ rects \wedge oper = ins\text{-}op\ rects\ (op\text{-}idx\ oper)$

@**proof**
  @**case** *oper* ∈ *set* (*ins-ops rects*) @*with*
    @**obtain** *i* **where** *i* < *length rects ins-ops rects* ! *i* = *oper* @**end**
  @**case** *op-idx oper* < *length rects* ∧ *oper* = *ins-op rects* (*op-idx oper*) @*with*
    @**have** *oper* = (*ins-ops rects*) ! (*op-idx oper*) @**end**
@**qed**

lemma *set-del-ops* [*rewrite*]:
  oper ∈ *set* (*del-ops rects*) ⟷ *op-idx oper* < *length rects* ∧ *oper* = *del-op rects* (*op-idx oper*)
@**proof**
  @**case** *oper* ∈ *set* (*del-ops rects*) @*with*
    @**obtain** *i* **where** *i* < *length rects del-ops rects* ! *i* = *oper* @**end**
  @**case** *op-idx oper* < *length rects* ∧ *oper* = *del-op rects* (*op-idx oper*) @*with*
    @**have** *oper* = (*del-ops rects*) ! (*op-idx oper*) @**end**
@**qed**

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** @**have** *distinct* (*ins-ops rects* @ *del-ops rects*) @**qed**

lemma *set-all-ops-idx* [*forward*]:
  oper ∈ *set* (*all-ops rects*) ⟹ *op-idx oper* < *length rects* **by** *auto2*

lemma *set-all-ops-ins* [*forward*]:
  INS p n i ∈ *set* (*all-ops rects*) ⟹ *INS p n i* = *ins-op rects n* **by** *auto2*

lemma *set-all-ops-del* [*forward*]:
  DEL p n i ∈ *set* (*all-ops rects*) ⟹ *DEL p n i* = *del-op rects n* **by** *auto2*

lemma *ins-in-set-all-ops*:
  i < *length rects* ⟹ *ins-op rects i* ∈ *set* (*all-ops rects*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm ins-in-set-all-ops*} [*with-term ins-op ?rects ?i*]›

lemma *del-in-set-all-ops*:
  i < *length rects* ⟹ *del-op rects i* ∈ *set* (*all-ops rects*) **by** *auto2*
setup ‹*add-forward-prfstep-cond* @{*thm del-in-set-all-ops*} [*with-term del-op ?rects ?i*]›

lemma *all-ops-sorted* [*forward*]: *sorted* (*all-ops rects*) **by** *auto2*

lemma *all-ops-nonempty* [*backward*]: *rects* ≠ [] ⟹ *all-ops rects* ≠ []
@**proof** @**have** *length* (*all-ops rects*) > *0* @**qed**

setup ‹*del-prfstep-thm* @{*thm all-ops-def*}›

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* ∧ (∃ *j*<*k*. *ins-op rects i* = *ops* ! *j*) ∧ ¬(∃ *j*<*k*. *del-op rects i* = *ops* ! *j*)})
setup ‹*register-wellform-data* (*apply-ops-k rects k*, [*k* < *length* (*all-ops rects*)])›

lemma *apply-ops-set-mem* [*rewrite*]:
ops = *all-ops rects* ⟹
i ∈ *apply-ops-k rects k* ⟷ (*i* < *length rects* ∧ (∃ *j*<*k*. *ins-op rects i* = *ops* ! *j*) ∧ ¬(∃ *j*<*k*. *del-op rects i* = *ops* ! *j*))
by *auto2*
setup ‹*del-prfstep-thm* @{*thm apply-ops-k-def*}›

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 ∈ *xints-of rect is* ⟷ (*i* ∈ *is* ∧ *xint* (*rect* ! *i*) = *it*) **using** *xints-of-def* **by** *auto*

lemma *xints-diff* [*rewrite*]:
xints-of rects (*A* − *B*) = *xints-of rects A* − *xints-of rects B*
@**proof** @**have** *inj* (λ*i*. *IdxInterval* (*xint* (*rects* ! *i*)) *i*) @**qed**

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*)))
setup ‹*register-wellform-data* (*has-overlap-at-k rects k*, [*k* < *length* (*all-ops rects*)])›

lemma *has-overlap-at-k-equiv* [*forward*]:
is-rect-list rects ⟹ *ops* = *all-ops rects* ⟹ *k* < *length ops* ⟹
has-overlap-at-k rects k ⟹ *has-rect-overlap rects*
@**proof**
@**let** *S* = *apply-ops-k rects k*
@**have** *has-overlap* (*xints-of rects S*) (*op-int* (*ops* ! *k*))
@**obtain** *xs'* ∈ *xints-of rects S* **where** *is-overlap* (*int xs'*) (*op-int* (*ops* ! *k*))
@**let** *xs* = *int xs' i* = *idx xs'*
@**let** *j* = *op-idx* (*ops* ! *k*)
@**have** *ops* ! *k* = *ins-op rects j*
@**have** *i* ≠ *j* @*with* @**contradiction**
@**obtain** *k'* **where** *k'* < *k ops* ! *k'* = *ins-op rects i*
@**have** *ops* ! *k* = *ops* ! *k'*
@**end**
@**have** *low* (*yint* (*rects* ! *i*)) ≤ *pos* (*ops* ! *k*) @*with*

```
    @obtain k' where k' < k ops ! k' = ins-op rects i
    @have ops ! k' ≤ ops ! k
  @end
  @have high (yint (rects ! i)) ≥ pos (ops ! k) @with
    @obtain k' where k' < length ops ops ! k' = del-op rects i
    @have ops ! k' ≥ ops ! k
  @end
  @have is-rect-overlap (rects ! i) (rects ! j)
@qed

lemma has-overlap-at-k-equiv2 [resolve]:
  is-rect-list rects ⟹ ops = all-ops rects ⟹ has-rect-overlap rects ⟹
   ∃k<length ops. has-overlap-at-k rects k
@proof
  @obtain i j where i < length rects j < length rects i ≠ j
                   is-rect-overlap (rects ! i) (rects ! j)
  @have is-rect-overlap (rects ! j) (rects ! i)
  @obtain i1 where i1 < length ops ops ! i1 = ins-op rects i
  @obtain j1 where j1 < length ops ops ! j1 = ins-op rects j
  @obtain i2 where i2 < length ops ops ! i2 = del-op rects i
  @obtain j2 where j2 < length ops ops ! j2 = del-op rects j
  @case ins-op rects i < ins-op rects j @with
    @have i1 < j1
    @have j1 < i2 @with @have ops ! j1 < ops ! i2 @end
    @have is-overlap (int (IdxInterval (xint (rects ! i)) i)) (xint (rects ! j))
    @have has-overlap-at-k rects j1
  @end
  @case ins-op rects j < ins-op rects i @with
    @have j1 < i1
    @have i1 < j2 @with @have ops ! i1 < ops ! j2 @end
    @have is-overlap (int (IdxInterval (xint (rects ! j)) j)) (xint (rects ! i))
    @have has-overlap-at-k rects i1
  @end
@qed

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 ⟹ has-overlap-lst rects ⟷ has-rect-overlap rects by auto2
```

15.5 Implementation of apply_ops_k

```
lemma apply-ops-k-next1 [rewrite]:
  is-rect-list rects ⟹ ops = all-ops rects ⟹ n < length ops ⟹ is-INS (ops ! n)
⟹
   apply-ops-k rects (n + 1) = apply-ops-k rects n ∪ {op-idx (ops ! n)}
@proof
```

```
  @have ∀ i. i∈apply-ops-k rects (n + 1) ⟷ i∈apply-ops-k rects n ∪ {op-idx (ops
! n)} @with
    @case i ∈ apply-ops-k rects n ∪ {op-idx (ops ! n)} @with
      @case i = op-idx (ops ! n) @with
        @have ins-op rects i < del-op rects i
      @end
    @end
  @end
@qed

lemma apply-ops-k-next2 [rewrite]:
  is-rect-list rects ⟹ ops = all-ops rects ⟹ n < length ops ⟹ ¬is-INS (ops !
n) ⟹
   apply-ops-k rects (n + 1) = apply-ops-k rects n − {op-idx (ops ! n)} by auto2

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}))
setup ‹add-rewrite-rule @{thm apply-ops-k-next-def}›

lemma apply-ops-k-next-is-correct [rewrite]:
  is-rect-list rects ⟹ ops = all-ops rects ⟹ n < length ops ⟹
   S = xints-of rects (apply-ops-k rects n) ⟹
   xints-of rects (apply-ops-k rects (n + 1)) = apply-ops-k-next rects S n
@proof @case is-INS (ops ! n) @qed

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))
  by auto
 termination by (relation measure (λ(rects,S,k). length (all-ops rects) − k)) auto

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
  @let ints = xints-of rects (apply-ops-k rects n)
  @fun-induct rect-inter rects ints n
  @unfold rect-inter rects ints n
```

@**case** $n \geq$ *length ops*
@**case** *is-INS* (*ops* ! *n*) $\wedge$ *has-overlap ints* (*op-int* (*ops* ! *n*))
@**case** *n* = *length ops* − *1*
@**qed**

Correctness of functional algorithm.

theorem *rect-inter-correct* [*rewrite*]:
is-rect-list rects $\Longrightarrow$ *rect-inter rects* {} *0* $\longleftrightarrow$ *has-rect-overlap rects*
@**proof**
@**have** {} = *xints-of rects* (*apply-ops-k rects 0*)
@**have** *rect-inter rects* {} *0* = *has-overlap-lst rects* @*with*
@**unfold** *rect-inter rects* {} *0*
@**end**
@**qed**

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-file *sep-util-base.ML*
ML-file *assn-matcher.ML*
ML-file *sep-steps.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*)
setup ‹*add-simple-datatype pheap*›

fun *in-range* :: (*heap* × *addr set*) ⇒ *bool* **where**
in-range (*h*,*as*) $\longleftrightarrow$ ($\forall$ *a*∈*as*. *a* < *lim h*)
setup ‹*add-rewrite-rule* @{*thm in-range.simps*}›

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*) **by** *auto2*

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* **by** *auto2*+
setup ‹*del-prfstep-thm-eqforward* @{*thm relH-def*}›

lemma *relH-dist-union* [*forward*]:
  relH (*as* ∪ *as′*) *h h′* ⟹ *relH as h h′* ∧ *relH as′ h h′* **by** *auto2*

lemma *relH-ref* [*rewrite*]:
  relH as h h′ ⟹ *addr-of-ref r* ∈ *as* ⟹ *Ref.get h r* = *Ref.get h′ r*
  by (*auto intro*: *relH-D2 arg-cong simp*: *Ref.get-def*)

lemma *relH-array* [*rewrite*]:
  relH as h h′ ⟹ *addr-of-array r* ∈ *as* ⟹ *Array.get h r* = *Array.get h′ r*
  by (*auto intro*: *relH-D2 arg-cong simp*: *Array.get-def*)

lemma *relH-set-ref* [*resolve*]:
  relH {*a*. *a* < *lim h* ∧ *a* ∉ {*addr-of-ref r*}} *h* (*Ref.set r x h*)
  by (*simp add*: *Ref.set-def relH-def*)

lemma *relH-set-array* [*resolve*]:
  relH {*a*. *a* < *lim h* ∧ *a* ∉ {*addr-of-array r*}} *h* (*Array.set r x h*)
  by (*simp add*: *Array.set-def relH-def*)

16.2 Assertions

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

fun *aseval* :: *assn-raw* ⇒ *pheap* ⇒ *bool* **where**
  aseval (*Assn f*) *h* = *f h*
setup ‹*add-rewrite-rule* @{*thm aseval.simps*}›

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*)
setup ‹*add-rewrite-rule* @{*thm in-range-assn.simps*}›

typedef *assn* = *Collect proper*
@**proof** @**have** *Assn in-range-assn* ∈ *Collect proper* @**qed**

setup ‹*add-rewrite-rule* @{*thm Rep-assn-inject*}›
setup ‹*register-wellform-data* (*Abs-assn P*, [*proper P*])›
setup ‹*add-prfstep-check-req* (*Abs-assn P*, *proper P*)›

lemma *Abs-assn-inverse'* [*rewrite*]: *proper y* ⟹ *Rep-assn* (*Abs-assn y*) = *y*
by (*simp add*: *Abs-assn-inverse*)

lemma *proper-Rep-assn* [*forward*]: *proper* (*Rep-assn P*) **using** *Rep-assn* **by** *auto*

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*) **by** *auto2*

lemma *mod-relH* [*forward*]: *relH as h h'* ⟹ *pHeap h as* ⊨ *P* ⟹ *pHeap h' as* ⊨ *P* **by** *auto2*

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

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

lemma *one-assn-rule* [*rewrite*]: *h* ⊨ *emp* ⟷ *addrOf h* = {} **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm one-assn-def*}›

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 .. **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*) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm times-assn-def*}›

lemma *aseval-ext* [*backward*]: ∀ *h*. *aseval P h* = *aseval P' h* ⟹ *P* = *P'*
apply (*cases P*) **apply** (*cases P'*) **by** *auto*

lemma *assn-ext*: ∀ *h as*. *pHeap h as* ⊨ *P* ⟷ *pHeap h as* ⊨ *Q* ⟹ *P* = *Q*
@**proof** @**have** *Rep-assn P* = *Rep-assn Q* @**qed**

setup ‹*add-backward-prfstep-cond* @{*thm assn-ext*} [*with-filt* (*order-filter P Q*)]›

setup ‹*del-prfstep-thm* @{*thm aseval-ext*}›

lemma *assn-one-left*: *1 * P = (P::assn)*
@proof
@have *∀h as. pHeap h as ⊨ P ⟷ pHeap h as ⊨ 1 * P @with*
@have *as = {} ∪ as*
@end
@qed

lemma *assn-times-comm*: *P * Q = Q * (P::assn)*
@proof
@have *∀h as. pHeap h as ⊨ P * Q ⟷ pHeap h as ⊨ Q * P @with*
@case *pHeap h as ⊨ P * Q @with*
@obtain *as1 as2* **where** *as = as1 ∪ as2 as1 ∩ as2 = {} pHeap h as1 ⊨ P pHeap h as2 ⊨ Q*
@have *as = as2 ∪ as1*
@end
@case *pHeap h as ⊨ Q * P @with*
@obtain *as1 as2* **where** *as = as1 ∪ as2 as1 ∩ as2 = {} pHeap h as1 ⊨ Q pHeap h as2 ⊨ P*
@have *as = as2 ∪ as1*
@end
@end
@qed

lemma *assn-times-assoc*: *(P * Q) * R = P * (Q * (R::assn))*
@proof
@have *∀h as. pHeap h as ⊨ (P * Q) * R ⟷ pHeap h as ⊨ P * (Q * R) @with*
@case *pHeap h as ⊨ (P * Q) * R @with*
@obtain *as1 as2* **where** *as = as1 ∪ as2 as1 ∩ as2 = {} pHeap h as1 ⊨ P * Q pHeap h as2 ⊨ R*
@obtain *as11 as12* **where** *as1 = as11 ∪ as12 as11 ∩ as12 = {} pHeap h as11 ⊨ P pHeap h as12 ⊨ Q*
@have *as = as11 ∪ (as12 ∪ as2)*
@end
@case *pHeap h as ⊨ P * (Q * R) @with*
@obtain *as1 as2* **where** *as = as1 ∪ as2 as1 ∩ as2 = {} pHeap h as1 ⊨ P pHeap h as2 ⊨ Q * R*
@obtain *as21 as22* **where** *as2 = as21 ∪ as22 as21 ∩ as22 = {} pHeap h as21 ⊨ Q pHeap h as22 ⊨ R*
@have *as = (as1 ∪ as21) ∪ as22*
@end
@end
@qed

instantiation *assn* :: *comm-monoid-mult* **begin**
instance apply *standard*

apply (*rule assn-times-assoc*) **apply** (*rule assn-times-comm*) **by** (*rule assn-one-left*)
end

16.2.1 Existential Quantification

definition *ex-assn* :: ('*a* ⇒ *assn*) ⇒ *assn* (**binder** ∃$_A$ *11*) **where** [*rewrite*]:
(∃$_A$*x*. *P x*) = *Abs-assn* (*Assn* (λ*h*. ∃ *x*. *h* ⊨ *P x*))

lemma *mod-ex-dist* [*rewrite*]: (*h* ⊨ (∃$_A$*x*. *P x*)) ⟷ (∃ *x*. *h* ⊨ *P x*) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm ex-assn-def*}›

lemma *ex-distrib-star*: (∃$_A$*x*. *P x* ∗ *Q*) = (∃$_A$*x*. *P x*) ∗ *Q*
@**proof**
@**have** ∀ *h as*. *pHeap h as* ⊨ (∃$_A$*x*. *P x*) ∗ *Q* ⟷ *pHeap h as* ⊨ (∃$_A$*x*. *P x* ∗ *Q*) @*with*
@**case** *pHeap h as* ⊨ (∃$_A$*x*. *P x*) ∗ *Q* @*with*
@**obtain** *as1 as2* **where** *as* = *as1* ∪ *as2 as1* ∩ *as2* = {} *pHeap h as1* ⊨ (∃$_A$*x*. *P x*) *pHeap h as2* ⊨ *Q*
@**obtain** *x* **where** *pHeap h as1* ⊨ *P x*
@**have** *pHeap h as* ⊨ *P x* ∗ *Q*
@**end**
@**end**
@**qed**

16.2.2 Pointers

definition *sngr-assn* :: '*a*::*heap ref* ⇒ '*a* ⇒ *assn* (**infix** ↦$_r$ *82*) **where** [*rewrite*]:
r ↦$_r$ *x* = *Abs-assn* (*Assn* (
λ*h*. *Ref.get* (*heapOf h*) *r* = *x* ∧ *addrOf h* = {*addr-of-ref r*} ∧ *addr-of-ref r* < *lim* (*heapOf h*)))

lemma *sngr-assn-rule* [*rewrite*]:
pHeap h as ⊨ *r* ↦$_r$ *x* ⟷ (*Ref.get h r* = *x* ∧ *as* = {*addr-of-ref r*} ∧ *addr-of-ref r* < *lim h*) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm sngr-assn-def*}›

definition *snga-assn* :: '*a*::*heap array* ⇒ '*a list* ⇒ *assn* (**infix** ↦$_a$ *82*) **where** [*rewrite*]:
r ↦$_a$ *x* = *Abs-assn* (*Assn* (
λ*h*. *Array.get* (*heapOf h*) *r* = *x* ∧ *addrOf h* = {*addr-of-array r*} ∧ *addr-of-array r* < *lim* (*heapOf h*)))

lemma *snga-assn-rule* [*rewrite*]:
pHeap h as ⊨ *r* ↦$_a$ *x* ⟷ (*Array.get h r* = *x* ∧ *as* = {*addr-of-array r*} ∧ *addr-of-array r* < *lim h*) **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm snga-assn-def*}›

16.2.3 Pure Assertions

definition *pure-assn* :: *bool* ⇒ *assn* (↑) **where** [*rewrite*]:

$\uparrow b = Abs\text{-}assn\ (Assn\ (\lambda h.\ addrOf\ h = \{\} \wedge b))$

lemma *pure-assn-rule* [*rewrite*]: $h \models \uparrow b \longleftrightarrow (addrOf\ h = \{\} \wedge b)$ **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm pure-assn-def*}›

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

lemma *top-assn-rule* [*rewrite*]: $pHeap\ h\ as \models true \longleftrightarrow in\text{-}range\ (h, as)$ **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm top-assn-def*}›

setup ‹*del-prfstep-thm* @{*thm models-def*}›

16.2.4 Properties of assertions

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

lemma *top-assn-reduce*: $true * true = true$
@**proof**
@**have** $\forall h.\ h \models true \longleftrightarrow h \models true * true$ @*with*
@**have** $addrOf\ h = addrOf\ h \cup \{\}$
@**end**
@**qed**

lemma *mod-pure-star-dist* [*rewrite*]:
$h \models P * \uparrow b \longleftrightarrow (h \models P \wedge b)$
@**proof**
@**case** $h \models P \wedge b$ @*with*
@**have** $addrOf\ h = addrOf\ h \cup \{\}$
@**end**
@**qed**

lemma *pure-conj*: $\uparrow(P \wedge Q) = \uparrow P * \uparrow Q$ **by** *auto2*

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*: $A \Longrightarrow_A A$ **by** *auto2*
lemma *entails-true*: $A \Longrightarrow_A true$ **by** *auto2*
lemma *entails-frame* [*backward*]: $P \Longrightarrow_A Q \Longrightarrow P * R \Longrightarrow_A Q * R$ **by** *auto2*
lemma *entails-frame'*: $\neg (A * F \Longrightarrow_A Q) \Longrightarrow A \Longrightarrow_A B \Longrightarrow \neg (B * F \Longrightarrow_A Q)$
by *auto2*
lemma *entails-frame''*: $\neg (P \Longrightarrow_A B * F) \Longrightarrow A \Longrightarrow_A B \Longrightarrow \neg (P \Longrightarrow_A A * F)$
by *auto2*
lemma *entails-equiv-forward*: $P = Q \Longrightarrow P \Longrightarrow_A Q$ **by** *auto2*
lemma *entails-equiv-backward*: $P = Q \Longrightarrow Q \Longrightarrow_A P$ **by** *auto2*
lemma *entailsD* [*forward*]: $P \Longrightarrow_A Q \Longrightarrow h \models P \Longrightarrow h \models Q$ **by** *auto2*
lemma *entails-trans2*: $A \Longrightarrow_A D * B \Longrightarrow B \Longrightarrow_A C \Longrightarrow A \Longrightarrow_A D * C$ **by** *auto2*

lemma *entails-pure'*: $\neg(\uparrow b \Longrightarrow_A Q) \longleftrightarrow (\neg(emp \Longrightarrow_A Q) \wedge b)$ **by** *auto2*
lemma *entails-pure*: $\neg(P * \uparrow b \Longrightarrow_A Q) \longleftrightarrow (\neg(P \Longrightarrow_A Q) \wedge b)$ **by** *auto2*
lemma *entails-ex*: $\neg((\exists_A x.\ P\ x) \Longrightarrow_A Q) \longleftrightarrow (\exists x.\ \neg(P\ x \Longrightarrow_A Q))$ **by** *auto2*
lemma *entails-ex-post*: $\neg(P \Longrightarrow_A (\exists_A x.\ Q\ x)) \Longrightarrow \forall x.\ \neg(P \Longrightarrow_A Q\ x)$ **by** *auto2*
lemma *entails-pure-post*: $\neg(P \Longrightarrow_A Q * \uparrow b) \Longrightarrow P \Longrightarrow_A Q \Longrightarrow \neg b$ **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm entails-def*}›

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*
setup ‹*add-case-induct-rule* @{*thm run.cases*}›
setup ‹*fold add-resolve-prfstep* @{*thms run.intros*(*1*,*2*)}›
setup ‹*add-forward-prfstep* @{*thm run.intros*(*3*)}›

lemma *run-complete* [*resolve*]:
  $\exists \sigma'\ r.$ *run c σ σ' (r::'a)*
@**proof**
  @**obtain** *r::'a* **where** *r = r*
  @**case** *σ = None* @*with* @**have** *run c None None r* @**end**
  @**case** *execute c (the σ) = None* @*with* @**have** *run c σ None r* @**end**
@**qed**

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** @**case-induct** *run c (Some h) σ' r* @**qed**

setup ‹*add-rewrite-rule* @{*thm execute-bind*(*1*)}›
lemma *runE* [*forward*]:
  run f (Some h) (Some h') r' ⟹ *run (f ⤜ g) (Some h) σ r* ⟹ *run (g r') (Some h') σ r* **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm Array.get-alloc*}›
setup ‹*add-rewrite-rule* @{*thm Ref.get-alloc*}›
setup ‹*add-rewrite-rule-bidir* @{*thm Array.length-def*}›

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* ⟶

$(\sigma \neq None \wedge pHeap\ (the\ \sigma)\ (new\text{-}addrs\ h\ as\ (the\ \sigma)) \models Q\ r \wedge relH\ \{a\ .\ a < lim\ h \wedge a \notin as\}\ h\ (the\ \sigma) \wedge$
$lim\ h \leq lim\ (the\ \sigma)))$

lemma *hoare-tripleD* [*forward*]:
$<P>\ c\ <Q> \Longrightarrow run\ c\ (Some\ h)\ \sigma\ r \Longrightarrow \forall as.\ pHeap\ h\ as \models P \longrightarrow$
$(\sigma \neq None \wedge pHeap\ (the\ \sigma)\ (new\text{-}addrs\ h\ as\ (the\ \sigma)) \models Q\ r \wedge relH\ \{a\ .\ a < lim\ h \wedge a \notin as\}\ h\ (the\ \sigma) \wedge$
$lim\ h \leq lim\ (the\ \sigma))$
by *auto2*
setup ‹*del-prfstep-thm-eqforward* @{*thm hoare-triple-def*}›

abbreviation *hoare-triple'* :: *assn* ⇒ *'r Heap* ⇒ (*'r* ⇒ *assn*) ⇒ *bool* (<-> - <->$_t$)
where
$<P>\ c\ <Q>_t \equiv <P>\ c\ <\lambda r.\ Q\ r * true>$

theorem *frame-rule* [*backward*]:
$<P>\ c\ <Q> \Longrightarrow <P * R>\ c\ <\lambda x.\ Q\ x * R>$
@**proof**
@**have** $\forall h\ as\ \sigma\ r.\ pHeap\ h\ as \models P * R \longrightarrow run\ c\ (Some\ h)\ \sigma\ r \longrightarrow$
$(\sigma \neq None \wedge pHeap\ (the\ \sigma)\ (new\text{-}addrs\ h\ as\ (the\ \sigma)) \models Q\ r * R \wedge$
$relH\ \{a\ .\ a < lim\ h \wedge a \notin as\}\ h\ (the\ \sigma) \wedge lim\ h \leq lim\ (the\ \sigma))$
@*with*
@**obtain** *as1 as2* **where** $as = as1 \cup as2\ as1 \cap as2 = \{\}\ pHeap\ h\ as1 \models P \wedge pHeap\ h\ as2 \models R$
@**have** *relH as2 h* (*the* σ)
@**have** *new-addrs h as* (*the* σ) = *new-addrs h as1* (*the* σ) ∪ *as2*
@**end**
@**qed**

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

setup ‹*del-prfstep-thm* @{*thm mod-star-conv*}›

theorem *bind-rule*:
$<P>\ f\ <Q> \Longrightarrow \forall x.\ <Q\ x>\ g\ x\ <R> \Longrightarrow <P>\ f \gg= g\ <R>$
@**proof**
@**have** $\forall h\ as\ \sigma\ r.\ pHeap\ h\ as \models P \longrightarrow run\ (f \gg= g)\ (Some\ h)\ \sigma\ r \longrightarrow$
$(\sigma \neq None \wedge pHeap\ (the\ \sigma)\ (new\text{-}addrs\ h\ as\ (the\ \sigma)) \models R\ r \wedge$
$relH\ \{a\ .\ a < lim\ h \wedge a \notin as\}\ h\ (the\ \sigma) \wedge lim\ h \leq lim\ (the\ \sigma))$
@*with*
— First step from h to h'
@**obtain** $\sigma'\ r'$ **where** *run f* (*Some h*) $\sigma'\ r'$
@**obtain** h' **where** $\sigma' = Some\ h'$
@**let** $as' = new\text{-}addrs\ h\ as\ h'$
@**have** *pHeap* $h'\ as' \models Q\ r'$

— Second step from h' to h"
@**have** *run* (*g r'*) (*Some h'*) σ *r*
@**obtain** h'' **where** $\sigma = Some\ h''$

@**let** $as'' = new\text{-}addrs\ h'\ as'\ h''$
@**have** $pHeap\ h''\ as'' \models R\ r$
@**have** $as'' = new\text{-}addrs\ h\ as\ h''$
@**end**
@**qed**

Actual statement used:

lemma *bind-rule'*:
$<P>\ f\ <Q> \Longrightarrow \neg\ <P>\ f \gg= g\ <R> \Longrightarrow \exists x.\ \neg\ <Q\ x>\ g\ x\ <R>$ **using** *bind-rule* **by** *blast*

lemma *pre-rule'*:
$\neg\ <P * R>\ f\ <Q> \Longrightarrow P \Longrightarrow_A P' \Longrightarrow \neg\ <P' * R>\ f\ <Q>$
@**proof** @**have** $P * R \Longrightarrow_A P' * R$ @**qed**

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

lemma *pre-ex-rule*:
$\neg\ <\exists_A x.\ P\ x>\ f\ <Q> \longleftrightarrow (\exists x.\ \neg\ <P\ x>\ f\ <Q>)$ **by** *auto2*

lemma *pre-pure-rule*:
$\neg\ <P * \uparrow b>\ f\ <Q> \longleftrightarrow \neg\ <P>\ f\ <Q> \wedge b$ **by** *auto2*

lemma *pre-pure-rule'*:
$\neg\ <\uparrow b>\ f\ <Q> \longleftrightarrow \neg\ <emp>\ f\ <Q> \wedge b$ **by** *auto2*

lemma *post-rule*:
$<P>\ f\ <Q> \Longrightarrow \forall x.\ Q\ x \Longrightarrow_A R\ x \Longrightarrow <P>\ f\ <R>$ **by** *auto2*

setup ‹*fold del-prfstep-thm* [@{*thm entailsD*}, @{*thm entails-frame*}, @{*thm frame-rule*}]›

Actual statement used:

lemma *post-rule'*:
$<P>\ f\ <Q> \Longrightarrow \neg\ <P>\ f\ <R> \Longrightarrow \exists x.\ \neg\ (Q\ x \Longrightarrow_A R\ x)$ **using** *post-rule* **by** *blast*

lemma *norm-pre-pure-iff*: $<P * \uparrow b>\ c\ <Q> \longleftrightarrow (b \longrightarrow <P>\ c\ <Q>)$ **by** *auto2*
lemma *norm-pre-pure-iff2*: $<\uparrow b>\ c\ <Q> \longleftrightarrow (b \longrightarrow <emp>\ c\ <Q>)$ **by** *auto2*

16.5 Hoare triples for atomic commands

First, those that do not modify the heap.

setup ‹*add-rewrite-rule* @{*thm execute-assert*(*1*)}›
lemma *assert-rule*:
$<\uparrow(R\ x)>\ assert\ R\ x\ <\lambda r.\ \uparrow(r = x)>$ **by** *auto2*

lemma *execute-return'* [*rewrite*]: *execute* (*return x*) *h* = *Some* (*x*, *h*) **by** (*metis comp-eq-dest-lhs execute-return*)
lemma *return-rule*:
<*emp*> *return x* <λ*r*. ↑(*r* = *x*)> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-nth*(*1*)}›
lemma *nth-rule*:
<*a* ↦$_a$ *xs* ∗ ↑(*i* < *length xs*)> *Array.nth a i* <λ*r*. *a* ↦$_a$ *xs* ∗ ↑(*r* = *xs* ! *i*)> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-len*}›
lemma *length-rule*:
<*a* ↦$_a$ *xs*> *Array.len a* <λ*r*. *a* ↦$_a$ *xs* ∗ ↑(*r* = *length xs*)> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-lookup*}›
lemma *lookup-rule*:
<*p* ↦$_r$ *x*> !*p* <λ*r*. *p* ↦$_r$ *x* ∗ ↑(*r* = *x*)> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-freeze*}›
lemma *freeze-rule*:
<*a* ↦$_a$ *xs*> *Array.freeze a* <λ*r*. *a* ↦$_a$ *xs* ∗ ↑(*r* = *xs*)> **by** *auto2*

Next, the update rules.

setup ‹*add-rewrite-rule* @{*thm Ref.lim-set*}›
lemma *Array-lim-set* [*rewrite*]: *lim* (*Array.set p xs h*) = *lim h* **by** (*simp add*: *Array.set-def*)

setup ‹*fold add-rewrite-rule* [@{*thm Ref.get-set-eq*}, @{*thm Array.get-set-eq*}]›
setup ‹*add-rewrite-rule* @{*thm Array.update-def*}›

setup ‹*add-rewrite-rule* @{*thm execute-upd*(*1*)}›
lemma *upd-rule*:
<*a* ↦$_a$ *xs* ∗ ↑(*i* < *length xs*)> *Array.upd i x a* <λ*r*. *a* ↦$_a$ *list-update xs i x* ∗ ↑(*r* = *a*)> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-update*}›
lemma *update-rule*:
<*p* ↦$_r$ *y*> *p* := *x* <λ*r*. *p* ↦$_r$ *x*> **by** *auto2*

Finally, the allocation rules.

lemma *lim-set-gen* [*rewrite*]: *lim* (*h*(|*lim* := *l*|)) = *l* **by** *simp*

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*|)))))
by (*simp add*: *Array.alloc-def*)

setup ‹*fold add-rewrite-rule* [
@{*thm addr-of-array.simps*}, @{*thm addr-of-ref.simps*}, @{*thm Ref.alloc-def*}]›

lemma *refs-on-Array-set* [*rewrite*]: *refs* (*Array.set p xs h*) *t i* = *refs h t i*
by (*simp add*: *Array.set-def*)

lemma *arrays-on-Ref-set* [*rewrite*]: *arrays* (*Ref.set p x h*) *t i* = *arrays h t i*
by (*simp add*: *Ref.set-def*)

lemma *refs-on-Array-alloc* [*rewrite*]: *refs* (*snd* (*Array.alloc xs h*)) *t i* = *refs h t i*
by (*metis* (*no-types*, *lifting*) *Array.alloc-def refs-on-Array-set select-convs*(*2*) *snd-conv surjective update-convs*(*3*))

lemma *arrays-on-Ref-alloc* [*rewrite*]: *arrays* (*snd* (*Ref.alloc x h*)) *t i* = *arrays h t i*
by (*metis* (*no-types*, *lifting*) *Ref.alloc-def arrays-on-Ref-set select-convs*(*1*) *sndI surjective update-convs*(*3*))

lemma *arrays-on-Array-alloc* [*rewrite*]: *i* < *lim h* ⟹ *arrays* (*snd* (*Array.alloc xs h*)) *t i* = *arrays h t i*
by (*smt Array.alloc-def Array.set-def addr-of-array.simps fun-upd-apply less-or-eq-imp-le linorder-not-less simps*(*1*) *snd-conv surjective update-convs*(*1*) *update-convs*(*3*))

lemma *refs-on-Ref-alloc* [*rewrite*]: *i* < *lim h* ⟹ *refs* (*snd* (*Ref.alloc x h*)) *t i* = *refs h t i*
by (*smt Ref.alloc-def Ref.set-def addr-of-ref.simps fun-upd-apply less-or-eq-imp-le linorder-not-less select-convs*(*2*) *simps*(*6*) *snd-conv surjective update-convs*(*3*))

setup ‹*add-rewrite-rule* @{*thm execute-new*}›
lemma *new-rule*:
<*emp*> *Array.new n x* <λ*r*. *r* ↦$_a$ *replicate n x*> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-of-list*}›
lemma *of-list-rule*:
<*emp*> *Array.of-list xs* <λ*r*. *r* ↦$_a$ *xs*> **by** *auto2*

setup ‹*add-rewrite-rule* @{*thm execute-ref*}›
lemma *ref-rule*:
<*emp*> *ref x* <λ*r*. *r* ↦$_r$ *x*> **by** *auto2*

setup ‹*fold del-prfstep-thm* [
@{*thm sngr-assn-rule*}, @{*thm snga-assn-rule*}, @{*thm pure-assn-rule*}, @{*thm top-assn-rule*},
@{*thm mod-pure-star-dist*}, @{*thm one-assn-rule*}, @{*thm hoare-triple-def*}, @{*thm mod-ex-dist*}]›
setup ‹*del-simple-datatype pheap*›

16.6 Definition of procedures

ASCII abbreviations for ML files.

abbreviation (*input*) *ex-assn-ascii* :: ('*a* ⇒ *assn*) ⇒ *assn* (**binder** *EXA 11*)

where *ex-assn-ascii* ≡ *ex-assn*

abbreviation (*input*) *models-ascii* :: *pheap* ⇒ *assn* ⇒ *bool* (**infix** |= *50*)
where *h* |= *P* ≡ *h* ⊨ *P*

ML-file *sep-util.ML*

ML ‹
structure AssnMatcher = AssnMatcher(SepUtil)
structure SepLogic = SepLogic(SepUtil)
val add-assn-matcher = AssnMatcher.add-assn-matcher
val add-entail-matcher = AssnMatcher.add-entail-matcher
val add-forward-ent-prfstep = SepLogic.add-forward-ent-prfstep
val add-rewrite-ent-rule = SepLogic.add-rewrite-ent-rule
val add-hoare-triple-prfstep = SepLogic.add-hoare-triple-prfstep
›
setup ‹*AssnMatcher.add-assn-matcher-proofsteps*›
setup ‹*SepLogic.add-sep-logic-proofsteps*›

ML-file *sep-steps-test.ML*

attribute-setup *forward-ent* = ‹*setup-attrib add-forward-ent-prfstep*›
attribute-setup *rewrite-ent* = ‹*setup-attrib add-rewrite-ent-rule*›
attribute-setup *hoare-triple* = ‹*setup-attrib add-hoare-triple-prfstep*›

setup ‹*fold add-hoare-triple-prfstep* [
@{*thm assert-rule*}, @{*thm update-rule*}, @{*thm nth-rule*}, @{*thm upd-rule*},
@{*thm return-rule*}, @{*thm ref-rule*}, @{*thm lookup-rule*}, @{*thm new-rule*},
@{*thm of-list-rule*}, @{*thm length-rule*}, @{*thm freeze-rule*}]›

Some simple tests

theorem <*emp*> *ref x* <λ*r*. *r* ↦$_r$ *x*> **by** *auto2*
theorem <*a* ↦$_r$ *x*> *ref x* <λ*r*. *a* ↦$_r$ *x* ∗ *r* ↦$_r$ *x*> **by** *auto2*
theorem <*a* ↦$_r$ *x*> (!*a*) <λ*r*. *a* ↦$_r$ *x* ∗ ↑(*r* = *x*)> **by** *auto2*
theorem <*a* ↦$_r$ *x* ∗ *b* ↦$_r$ *y*> (!*a*) <λ*r*. *a* ↦$_r$ *x* ∗ *b* ↦$_r$ *y* ∗ ↑(*r* = *x*)> **by** *auto2*
theorem <*a* ↦$_r$ *x* ∗ *b* ↦$_r$ *y*> (!*b*) <λ*r*. *a* ↦$_r$ *x* ∗ *b* ↦$_r$ *y* ∗ ↑(*r* = *y*)> **by** *auto2*
theorem <*a* ↦$_r$ *x*> *do* { *a* := *y*; !*a* } <λ*r*. *a* ↦$_r$ *y* ∗ ↑(*r* = *y*)> **by** *auto2*
theorem <*a* ↦$_r$ *x*> *do* { *a* := *y*; *a* := *z*; !*a* } <λ*r*. *a* ↦$_r$ *z* ∗ ↑(*r* = *z*)> **by** *auto2*
theorem <*a* ↦$_r$ *x*> *do* { *y* ← !*a*; *ref y*} <λ*r*. *a* ↦$_r$ *x* ∗ *r* ↦$_r$ *x*> **by** *auto2*
theorem <*emp*> *return x* <λ*r*. ↑(*r* = *x*)> **by** *auto2*

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.

setup ‹*add-rewrite-rule-back* @{*thm gcd-red-nat*}›

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.

setup ‹*add-fun-induct-rule* (@{*term gcd-fun*}, @{*thm gcd-fun.induct*})›

lemma *gcd-fun-correct*:
  gcd-fun a b = *gcd a b*
@**proof**
  @**fun-induct** *gcd-fun a b*
  @**unfold** *gcd-fun a b*
@**qed**

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-impl a b* <λ*r*. ↑(*r* = *gcd a b*)>
@**proof**
  @**fun-induct** *gcd-fun a b*
@**qed**

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*)
setup ‹*fold add-rewrite-rule* @{*thms node.sel*}›

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

instance *node* :: (*heap*) *heap*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of node-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

fun *os-list* :: *'a*::*heap list* ⇒ *'a node ref option* ⇒ *assn* **where**
  os-list [] *p* = ↑(*p* = *None*)
| *os-list* (*x* # *l*) (*Some p*) = (∃$_A$*q*. *p* ↦$_r$ *Node x q* ∗ *os-list l q*)
| *os-list* (*x* # *l*) *None* = *false*
setup ‹*fold add-rewrite-ent-rule* @{*thms os-list.simps*}›

lemma *os-list-empty* [*forward-ent*]:
  os-list [] *p* ⟹$_A$ ↑(*p* = *None*) **by** *auto2*

lemma *os-list-Cons* [*forward-ent*]:
  os-list (*x* # *l*) *p* ⟹$_A$ (∃$_A$*q*. *the p* ↦$_r$ *Node x q* ∗ *os-list l q* ∗ ↑(*p* ≠ *None*))
@**proof** @**case** *p* = *None* @**qed**

lemma *os-list-none*: *emp* ⟹$_A$ *os-list* [] *None* **by** *auto2*

lemma *os-list-constr-ent*:
  p ↦$_r$ *Node x q* ∗ *os-list l q* ⟹$_A$ *os-list* (*x* # *l*) (*Some p*) **by** *auto2*

setup ‹*fold add-entail-matcher* [@{*thm os-list-none*}, @{*thm os-list-constr-ent*}]›
setup ‹*fold del-prfstep-thm* @{*thms os-list.simps*}›

ML-file *list-matcher-test.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 []> **by** *auto2*

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 @case *xs = []* **@have** *xs = hd xs # tl xs* **@qed**

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)> **by** *auto2*

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 ↦ᵣ (Node x r') ∗ ↑(x = hd xs)>
@proof @case *xs = []* **@have** *xs = hd xs # tl xs* **@qed**

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 @induct *xs* **arbitrary** *p q ys* **@qed**

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)> **by** *auto2*

17.4 Remove

setup ‹*fold add-rewrite-rule* @{*thms removeAll.simps*}›

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 <λr. os-list (*removeAll x xs*) *r>*$_t$
@**proof** @**induct** *xs* **arbitrary** *b* @**qed**

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 <λr. os-list l p ∗ ↑(*r* = *l*)*>*
@**proof** @**induct** *l* **arbitrary** *p* @**qed**

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*)
setup ‹*fold add-rewrite-rule* @{*thms list-insert.simps*}›

lemma *list-insert-length*:

```
  length (list-insert x xs) = length xs + 1
@proof @induct xs @qed
setup ‹add-forward-prfstep-cond @{thm list-insert-length} [with-term list-insert ?x ?xs]›

lemma list-insert-mset [rewrite]:
  mset (list-insert x xs) = {#x#} + mset xs
@proof @induct xs @qed

lemma list-insert-set [rewrite]:
  set (list-insert x xs) = {x} ∪ set xs
@proof @induct xs @qed

lemma list-insert-sorted [forward]:
  sorted xs ⟹ sorted (list-insert x xs)
@proof @induct xs @qed

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 @induct xs arbitrary b @qed
```

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)
setup ‹fold add-rewrite-rule @{thms insert-sort.simps}›

lemma insert-sort-mset [rewrite]:
  mset (insert-sort xs) = mset xs
@proof @induct xs @qed

lemma insert-sort-sorted [forward]:
  sorted (insert-sort xs)
@proof @induct xs @qed

lemma insert-sort-is-sort [rewrite]:
```

```
  insert-sort xs = sort xs by auto2

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 @induct xs @qed

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 <λys. ↑(ys = sort xs)>t by auto2
```

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))
setup ‹fold add-rewrite-rule @{thms merge-list.simps}›

lemma merge-list-correct [rewrite]:
  set (merge-list xs ys) = set xs ∪ set ys
@proof @fun-induct merge-list xs ys @qed

lemma merge-list-sorted [forward]:
  sorted xs ⟹ sorted ys ⟹ sorted (merge-list xs ys)
@proof @fun-induct merge-list xs ys @qed

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 @fun-induct merge-list xs ys arbitrary p q @qed
```

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 @induct xs arbitrary b @qed
```

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 @induct xs arbitrary b @qed

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 @induct xs arbitrary b @qed

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 @induct xs arbitrary b @qed

setup ‹fold add-rewrite-rule @{thms List.fold-simps}›

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 @induct xs arbitrary b x @qed

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*)
setup ‹*fold add-rewrite-rule* @{*thms node.sel*}›

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*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of node-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

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*) = (∃*ₐlp rp*. *p* ↦*ᵣ Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp*)
| *btree* (*tree.Node lt k v rt*) *None* = *false*
setup ‹*fold add-rewrite-ent-rule* @{*thms btree.simps*}›

lemma *btree-Tip* [*forward-ent*]: *btree Tip p* ⟹*ₐ* ↑(*p* = *None*) **by** *auto2*

lemma *btree-Node* [*forward-ent*]:
  btree (*tree.Node lt k v rt*) *p* ⟹*ₐ* (∃*ₐlp rp*. *the p* ↦*ᵣ Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ∗ ↑(*p* ≠ *None*))
@**proof** @**case** *p* = *None* @**qed**

lemma *btree-none*: *emp* ⟹*ₐ btree tree.Tip None* **by** *auto2*

lemma *btree-constr-ent*:
  p ↦*ᵣ Node lp k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ⟹*ₐ btree* (*tree.Node lt k v rt*) (*Some p*) **by** *auto2*

setup ‹*fold add-entail-matcher* [@{*thm btree-none*}, @{*thm btree-constr-ent*}]›
setup ‹*fold del-prfstep-thm* @{*thms btree.simps*}›

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*> **by** *auto2*

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* = *Tip*)> **by** *auto2*

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*)> **by** *auto2*

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** @**induct** *t* **arbitrary** *b* @**qed**

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 @induct t arbitrary b @qed

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 by auto2

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 @induct *t* **arbitrary** *b* **@qed**

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 @induct *t* **arbitrary** *b* **@qed**

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 = (∃$_A$*t*. *btree t p* ∗ ↑(*tree-sorted t*) ∗ ↑(*M* = *tree-map t*))
setup ‹*add-rewrite-ent-rule* @{*thm btree-map-def*}›

theorem *btree-empty-rule-map* [*hoare-triple*]:
<emp> tree-empty <btree-map empty-map> **by** *auto2*

theorem *btree-insert-rule-map* [*hoare-triple*]:
<btree-map M b> btree-insert k v b <btree-map (*M* {*k* → *v*})*>* **by** *auto2*

theorem *btree-delete-rule-map* [*hoare-triple*]:
<btree-map M b> btree-delete x b <btree-map (*delete-map x M*)*>*$_t$ **by** *auto2*

theorem *btree-search-rule-map* [*hoare-triple*]:
*<btree-map M b> btree-search x b <*λ*r. btree-map M b* ∗ ↑(*r* = *M*⟨*x*⟩)*>* **by** *auto2*

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*)
setup ‹*fold add-rewrite-rule* @{*thms rbt-node.sel*}›

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

instance *color* :: *heap*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of color-encode*])
  apply (*metis color-encode.simps*(*1*) *color-encode.simps*(*2*) *not-B zero-neq-one*)
  ..

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*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of rbt-node-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

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*) = (∃$_A$*lp rp*. *p* ↦$_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp*)
| *btree* (*rbt.Node lt c k v rt*) *None* = *false*
setup ‹*fold add-rewrite-ent-rule* @{*thms btree.simps*}›

lemma *btree-Leaf* [*forward-ent*]: *btree Leaf p* ⟹$_A$ ↑(*p* = *None*) **by** *auto2*

lemma *btree-Node* [*forward-ent*]:
  btree (*rbt.Node lt c k v rt*) *p* ⟹$_A$ (∃$_A$*lp rp*. *the p* ↦$_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ∗ ↑(*p* ≠ *None*))
@**proof** @**case** *p* = *None* @**qed**

lemma *btree-none*: *emp* ⟹$_A$ *btree Leaf None* **by** *auto2*

lemma *btree-constr-ent*:
  p ↦$_r$ *Node lp c k v rp* ∗ *btree lt lp* ∗ *btree rt rp* ⟹$_A$ *btree* (*rbt.Node lt c k v rt*)

(*Some p*) **by** *auto2*

setup ‹*fold add-entail-matcher* [@{*thm btree-none*}, @{*thm btree-constr-ent*}]›
setup ‹*fold del-prfstep-thm* @{*thms btree.simps*}›

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*> **by** *auto2*

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*)> **by** *auto2*

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*)> **by** *auto2*

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*> **by** *auto2*

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 <λr. btree t p ∗ ↑(r = rbt.cl t)>
@proof @case t = Leaf @qed

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 @case t = Leaf @qed
```

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)> by auto2

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))> by auto2
```

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 @unfold balanceR l k v r @qed
```

```
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 @unfold balance l k v r @qed
```

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 @induct *t* **arbitrary** *p* **@qed**

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*)> **by** *auto2*

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
   <λ*r*. *btree t b* ∗ ↑(*r* = *RBTree.rbt-search t x*)>
@proof @induct *t* **arbitrary** *b* **@qed**

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 @unfold balL l k v r @qed

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 @unfold balR l k v r @qed

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 @fun-induct combine lt rt arbitrary lp rp @with
  @subgoal (lt = rbt.Node l1 c1 k1 v1 r1, rt = rbt.Node l2 c2 k2 v2 r2)
    @unfold combine (rbt.Node l1 c1 k1 v1 r1) (rbt.Node l2 c2 k2 v2 r2)
  @endgoal @end
@qed

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 @induct t arbitrary p @with
  @subgoal t = rbt.Node l c k v r
    @unfold del x (rbt.Node l c k v r)
  @endgoal @end
@qed
```

```
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 by auto2
```

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*))
setup ‹*add-rewrite-ent-rule* @{*thm rbt-map-assn-def*}›

theorem *rbt-empty-rule* [*hoare-triple*]:
<*emp*> *tree-empty* <*rbt-map-assn empty-map*> **by** *auto2*

theorem *rbt-insert-rule* [*hoare-triple*]:
<*rbt-map-assn M b*> *rbt-insert k v b* <*rbt-map-assn* (*M* {*k* → *v*})> **by** *auto2*

theorem *rbt-search* [*hoare-triple*]:
<*rbt-map-assn M b*> *rbt-search x b* <λ*r*. *rbt-map-assn M b* ∗ ↑(*r* = *M*⟨*x*⟩)> **by** *auto2*

theorem *rbt-delete-rule* [*hoare-triple*]:
<*rbt-map-assn M b*> *rbt-delete k b* <*rbt-map-assn* (*delete-map k M*)>$_t$ **by** *auto2*

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** @**induct** *n* @**qed**

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> by auto2
```

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 @fun-induct rev-swap xs i j @unfold rev-swap xs i j @qed
```

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> by auto2

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 @fun-induct *Quicksort.part1 xs l r a* **@unfold** *Quicksort.part1 xs l r a* **@qed**

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 ↦a xs>
partition a l r
<λrs. a ↦a snd (Quicksort.partition xs l r) ∗ ↑(rs = fst (Quicksort.partition xs l r))>
@proof @unfold *Quicksort.partition xs l r* **@qed**

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*]:
  $r < length\ xs \Longrightarrow <a \mapsto_a xs>$
   quicksort a l r
   $<\lambda\text{-}.\ a \mapsto_a Quicksort.quicksort\ xs\ l\ r>$
@**proof** @**fun-induct** *Quicksort.quicksort xs l r* @**unfold** *Quicksort.quicksort xs l r* @**qed**

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
   $<\lambda\text{-}.\ a \mapsto_a sort\ xs>$ **by** *auto2*

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* $\mapsto_a$ *l* ∗ *fst u* $\mapsto_a$ *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*)> **by** *auto2*

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 @prop-induct ufa-invar l ∧ i < length l @qed

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 @prop-induct ufa-invar l ∧ i < length l @qed

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))>
  by auto2

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)> by auto2
```

```
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)> by auto2
```

```
setup ‹del-prfstep-thm @{thm is-uf-def}›
```

```
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** @**induct** *k* @**qed**

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*))> **by** *auto2*

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*)> **by** *auto2*

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*)
setup ‹*add-simple-datatype dynamic-array*›

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* ↦$_a$ *xs* ∗ ↑(*m* = *n*))
setup ‹*add-rewrite-ent-rule* @{*thm dyn-array-raw.simps*}›

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*)> **by** *auto2*

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*)
setup ‹*add-rewrite-rule* @{*thm double-length-fun.simps*}›

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*))>$_t$ **by** *auto2*

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*)
setup ‹*add-rewrite-rule* @{*thm push-array-basic-fun.simps*}›

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))> by auto2

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
   <λr. dyn-array-raw (xs, n) p ∗ ↑(r = n)> by auto2

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
   <λr. dyn-array-raw (xs, n) p ∗ ↑(r = length xs)> by auto2

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)> by auto2

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> by auto2

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*))> **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm dyn-array-raw.simps*}›
setup ‹*del-simple-datatype dynamic-array*›

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*)))
setup ‹*add-rewrite-rule* @{*thm push-array-fun.simps*}›

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*))>$_t$ **by** *auto2*

24.2 Abstract assertion

fun *abs-array* :: *'a::heap list* × *nat* ⇒ *'a list* **where**
abs-array (*xs*, *n*) = *take n xs*
setup ‹*add-rewrite-rule* @{*thm abs-array.simps*}›

lemma *double-length-abs* [*rewrite*]:
length xs = *n* ⟹ *abs-array* (*double-length-fun* (*xs*, *n*)) = *abs-array* (*xs*, *n*) **by** *auto2*

lemma *push-array-basic-abs* [*rewrite*]:
n < *length xs* ⟹ *abs-array* (*push-array-basic-fun x* (*xs*, *n*)) = *abs-array* (*xs*, *n*) @ [*x*]
@**proof** @**have** *length* (*take n xs* @ [*x*]) = *n* + *1* @**qed**

lemma *push-array-fun-abs* [*rewrite*]:
n ≤ *length xs* ⟹ *abs-array* (*push-array-fun x* (*xs*, *n*)) = *abs-array* (*xs*, *n*) @ [*x*] **by** *auto2*

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* []> **by** *auto2*

lemma *array-length-rule* [*hoare-triple*]:
<*dyn-array xs p*>
array-length p
<λ*r*. *dyn-array xs p* ∗ ↑(*r* = *length xs*)> **by** *auto2*

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*)> **by** *auto2*

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*> **by** *auto2*

lemma *push-array-rule* [*hoare-triple*]:
<*dyn-array xs p*>
push-array x p
<*dyn-array* (*xs* @ [*x*])>$_t$ **by** *auto2*

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** @**have** *last xs* = *xs* ! (*length xs* − *1*) @**qed**

setup ‹*del-prfstep-thm* @{*thm dyn-array-def*}›

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*> **by** *auto2*

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*)
setup ‹*add-simple-datatype indexed-pqueue*›

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* ↦$_a$ *m*)
setup ‹*add-rewrite-ent-rule* @{*thm idx-pqueue.simps*}›

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*)> **by** *auto2*

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*)> **by** *auto2*

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
  <λr. idx-pqueue (xs, m) p * ↑(r = m ! i)> by auto2

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
   <λr. idx-pqueue (xs, m) p * ↑(r = length xs)> by auto2

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 @unfold idx-pqueue-swap-fun (xs, m) i j @qed

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 @unfold idx-pqueue-push-fun k v (xs, m) @qed

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 @unfold idx-pqueue-pop-fun (xs, m) @qed

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> by auto2

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)> by auto2

setup ‹del-prfstep-thm @{thm idx-pqueue.simps}›
setup ‹del-simple-datatype indexed-pqueue›
```

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 @fun-induct idx-bubble-down-fun x k @with
  @subgoal (x = (xs, m), k = k)
  @unfold idx-bubble-down-fun (xs, m) k
  @case s2 k < length xs @with
    @case snd (xs ! s1 k) ≤ snd (xs ! s2 k)
  @end
  @case s1 k < length xs @end
@qed

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 @fun-induct idx-bubble-up-fun x k @with
  @subgoal (x = (xs, m), k = k)
  @unfold idx-bubble-up-fun (xs, m) k @end
@qed
```

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** @**unfold** *delete-min-idx-pqueue-fun* (*xs*, *m*) @**qed**

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** @**unfold** *insert-idx-pqueue-fun k v* (*xs*, *m*) @**qed**

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*]:
  $k < \textit{length}\ m \Longrightarrow \textit{index-of-pqueue}\ (xs,\ m) \Longrightarrow$
   <*idx-pqueue* $(xs,\ m)\ p$>
   update-idx-pqueue $k\ v\ p$
   <*idx-pqueue* (*update-idx-pqueue-fun* $k\ v\ (xs,\ m)$)>$_t$
@**proof** @**unfold** *update-idx-pqueue-fun* $k\ v\ (xs,\ m)$ @**qed**

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 $M\ n\ p = (\exists_A xs\ m.\ \textit{idx-pqueue}\ (xs,\ m)\ p\ *$
    $\uparrow(\textit{index-of-pqueue}\ (xs,\ m)) * \uparrow(\textit{is-heap}\ xs) * \uparrow(M = \textit{map-of-alist}\ xs) * \uparrow(n = \textit{length}\ m))$
setup ‹*add-rewrite-ent-rule* @{*thm idx-pqueue-map-def*}›

lemma *heap-implies-hd-min2* [*resolve*]:
  $\textit{is-heap}\ xs \Longrightarrow xs \neq [] \Longrightarrow (\textit{map-of-alist}\ xs)\langle k\rangle = \textit{Some}\ v \Longrightarrow \textit{snd}\ (\textit{hd}\ xs) \leq v$
@**proof**
  @**obtain** i **where** $i < \textit{length}\ xs\ xs\ !\ i = (k,\ v)$
  @**have** $\textit{snd}\ (\textit{hd}\ xs) \leq \textit{snd}\ (xs\ !\ i)$
@**qed**

theorem *idx-pqueue-empty-map* [*hoare-triple*]:
  <*emp*>
   idx-pqueue-empty n
   <*idx-pqueue-map empty-map* n> **by** *auto2*

theorem *delete-min-idx-pqueue-map* [*hoare-triple*]:
  <*idx-pqueue-map* $M\ n\ p * \uparrow(M \neq \textit{empty-map})$>
   delete-min-idx-pqueue p
   <$\lambda(x,\ r).$ *idx-pqueue-map* $(\textit{delete-map}\ (\textit{fst}\ x)\ M)\ n\ r * \uparrow(\textit{fst}\ x < n) *$
            $\uparrow(\textit{is-heap-min}\ (\textit{fst}\ x)\ M) * \uparrow(M\langle \textit{fst}\ x\rangle = \textit{Some}\ (\textit{snd}\ x))$> **by** *auto2*

theorem *insert-idx-pqueue-map* [*hoare-triple*]:
  $k < n \Longrightarrow k \notin \textit{keys-of}\ M \Longrightarrow$
   <*idx-pqueue-map* $M\ n\ p$>
   insert-idx-pqueue $k\ v\ p$
   <*idx-pqueue-map* $(M\ \{k \rightarrow v\})\ n$>$_t$ **by** *auto2*

theorem *has-key-idx-pqueue-map* [*hoare-triple*]:
  $k < n \Longrightarrow$
   <*idx-pqueue-map* $M\ n\ p$>
   has-key-idx-pqueue $k\ p$
   <$\lambda r.$ *idx-pqueue-map* $M\ n\ p * \uparrow(r \longleftrightarrow k \in \textit{keys-of}\ M)$> **by** *auto2*

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* $(M\ \{k \to v\})$ *n*>$_t$ **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm idx-pqueue-map-def*}›

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*)
setup ‹*add-simple-datatype dijkstra-state*›

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*
setup ‹*add-rewrite-ent-rule* @{*thm dstate.simps*}›

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*]:
  $k \leq$ *size G* $\Longrightarrow$
   <*emp*>
   dstate-pq-init G k
   <*idx-pqueue-map* (*map-constr* (λ*i*. *i* > *0*) (λ*i*. *weight G 0 i*) *k*) (*size G*)>$_t$
@**proof** @**induct** *k* @**qed**

definition *dstate-init* :: *graph* ⇒ *dijkstra-state Heap* **where**
  dstate-init G = *do* {
    a ← *Array.of-list* (*list* (λ*i*. *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
   $<dstate\ (dijkstra\text{-}start\text{-}state\ G)>_t$ **by** *auto2*

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* ⟹
   $<a \mapsto_a e * idx\text{-}pqueue\text{-}map\ M\ (length\ e)\ pq>$
   dstate-update-est G m k pq a
   $<\lambda r.\ dstate\ (State\ (list\text{-}update\text{-}set\text{-}impl\ (\lambda i.\ i \in keys\text{-}of\ M)$
                  $(\lambda i.\ min\ (e\ !\ m + weight\ G\ m\ i)\ (e\ !\ i))\ e\ k)\ M)\ (Dijkstra\text{-}State\ r\ pq)>_t$
@**proof** @**induct** *k* @**qed**

lemma *dstate-update-est-to-fun* [*hoare-triple*]:
  $<dstate\ (State\ e\ M)\ (Dijkstra\text{-}State\ a\ pq) * \uparrow(m < length\ e)>$
   dstate-update-est G m (*length e*) *pq a*
   $<\lambda r.\ dstate\ (State\ (list\text{-}update\text{-}set\ (\lambda i.\ i \in keys\text{-}of\ M)$
            $(\lambda i.\ min\ (e\ !\ m + weight\ G\ m\ i)\ (e\ !\ i))\ e)\ M)\ (Dijkstra\text{-}State\ r\ pq)>_t$
by *auto2*

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* ⟹
   $<a \mapsto_a e * idx\text{-}pqueue\text{-}map\ M\ (length\ e)\ pq>$

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 @induct *k* **@qed**

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)>$_t$ **by** *auto2*

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)>$_t$ **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm dstate.simps*}›

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 ≠ empty-map ⟹ inv G S ⟹
<dstate S (Dijkstra-State a pq)>
dijkstra-step-impl G (Dijkstra-State a pq)
<λr. ∃$_A$S′. dstate S′ r ∗ ↑(is-dijkstra-step G S S′)>$_t$ **by** *auto2*

lemma *dijkstra-step-impl-correct* [*hoare-triple*]:
heap S ≠ empty-map ⟹ inv G S ⟹
<dstate S p>
dijkstra-step-impl G p
<λr. ∃$_A$S′. dstate S′ r ∗ ↑(inv G S′) ∗ ↑(card (unknown-set S′) = card (unknown-set S) − 1)>$_t$ **by** *auto2*

fun *dijkstra-loop* :: *graph* ⇒ *nat* ⇒ *dijkstra-state* ⇒ *dijkstra-state Heap* **where**
  dijkstra-loop G 0 p = (*return p*)
| *dijkstra-loop G* (*Suc k*) *p* = *do* {
    p' ← *dijkstra-step-impl G p*;
    p'' ← *dijkstra-loop G k p'*;
    return p'' }

lemma *dijkstra-loop-correct* [*hoare-triple*]:
  <*dstate S p* ∗ ↑(*inv G S*) ∗ ↑(*n* ≤ *card* (*unknown-set S*))>
   dijkstra-loop G n p
  <λ*r*. ∃ *A*S'. *dstate S' r* ∗ ↑(*inv G S'*) ∗ ↑(*card* (*unknown-set S'*) = *card* (*unknown-set S*) − *n*)>*t*
@**proof** @**induct** *n* **arbitrary** *S p* @**qed**

definition *dijkstra* :: *graph* ⇒ *dijkstra-state Heap* **where**
  dijkstra G = *do* {
    p ← *dstate-init G*;
    dijkstra-loop G (*size G* − *1*) *p* }

Correctness of Dijkstra's algorithm.

theorem *dijkstra-correct* [*hoare-triple*]:
  size G > *0* ⟹
   <*emp*>
   dijkstra G
   <λ*r*. ∃ *A*S. *dstate S r* ∗ ↑(*inv G S*) ∗ ↑(*unknown-set S* = {}) ∗
       ↑(∀ *i*∈*verts G*. *has-dist G 0 i* ∧ *est S* ! *i* = *dist G 0 i*)>*t* **by** *auto2*

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*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of interval-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

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*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of idx-interval-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

27.2 Tree nodes

datatype *'a node* =
  Node (*lsub*: *'a node ref option*) (*val*: *'a idx-interval*) (*tmax*: *nat*) (*rsub*: *'a node ref option*)
setup ‹*fold add-rewrite-rule* @{*thms node.sel*}›

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*
  apply (*rule heap-class.intro*)
  apply (*rule countable-classI* [*of node-encode*])
  apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
  ..

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*) = (∃$_A$*lp rp*. *p* ↦$_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp*)
| *int-tree* (*interval-tree.Node lt v m rt*) *None* = *false*
setup ‹*fold add-rewrite-ent-rule* @{*thms int-tree.simps*}›

lemma *int-tree-Tip* [*forward-ent*]: *int-tree Tip p* ⟹$_A$ ↑(*p* = *None*) **by** *auto2*

lemma *int-tree-Node* [*forward-ent*]:
  int-tree (*interval-tree.Node lt v m rt*) *p* ⟹$_A$ (∃$_A$*lp rp*. *the p* ↦$_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp* ∗ ↑(*p* ≠ *None*))
@**proof** @**case** *p* = *None* @**qed**

lemma *int-tree-none*: *emp* ⟹$_A$ *int-tree interval-tree.Tip None* **by** *auto2*

lemma *int-tree-constr-ent*:
  p ↦$_r$ *Node lp v m rp* ∗ *int-tree lt lp* ∗ *int-tree rt rp* ⟹$_A$ *int-tree* (*interval-tree.Node lt v m rt*) (*Some p*) **by** *auto2*

setup ‹*fold add-entail-matcher* [@{*thm int-tree-none*}, @{*thm int-tree-constr-ent*}]›
setup ‹*fold del-prfstep-thm* @{*thms int-tree.simps*}›

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*> **by** *auto2*

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*)> **by** *auto2*

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** @**case** *t* = *Tip* @**qed**

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*))>
by *auto2*

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*)>
  by *auto2*

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** @**induct** *t* **arbitrary** *b* @**qed**

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 @induct t arbitrary b @qed

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 by auto2

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 @induct *t* **arbitrary** *b* **@qed**

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
<λ*r*. *int-tree t b* ∗ ↑(*r* ⟷ *search t x*)>
@proof @induct *t* **arbitrary** *b* *@with*
@subgoal *t* = *interval-tree.Node l v m r*
@case *is-overlap* (*int v*) *x*
@case *l* ≠ *Tip* ∧ *interval-tree.tmax l* ≥ *low x*
@endgoal @end
@qed

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*))
setup ‹*add-rewrite-ent-rule* @{*thm int-tree-set-def*}›

theorem *int-tree-empty-rule* [*hoare-triple*]:
<*emp*> *int-tree-empty* <*int-tree-set* {}> **by** *auto2*

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*})> **by** *auto2*

theorem *int-tree-delete-rule* [*hoare-triple*]:
<*int-tree-set S b* ∗ ↑(*is-interval* (*int x*))>
delete-impl x b

$<$*int-tree-set* $(S - \{x\})>_t$ **by** *auto2*

theorem *int-tree-search-rule* [*hoare-triple*]:
$<$*int-tree-set* $S\ b * \uparrow($*is-interval* $x)>$
search-impl $x\ b$
$<\lambda r.$ *int-tree-set* $S\ b * \uparrow(r \longleftrightarrow$ *has-overlap* $S\ x)>$ **by** *auto2*

setup ‹*del-prfstep-thm* @{*thm int-tree-set-def*}›

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*
apply (*rule heap-class.intro*)
apply (*rule countable-classI* [*of operation-encode*])
apply (*case-tac x*, *simp-all*, *case-tac y*, *simp-all*)
apply (*simp add*: *operation.case-eq-if*)
..

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 }

setup ‹*add-rewrite-rule* @{*thm all-ops-def*}›
lemma *rect-inter-init-rule* [*hoare-triple*]:
$<$*emp*$>$ *rect-inter-init rects* $<\lambda p.\ p \mapsto_a$ *all-ops rects*$>$ **by** *auto2*
setup ‹*del-prfstep-thm* @{*thm all-ops-def*}›

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 @have ops ! k ∈ set ops @case is-INS (ops ! k) @qed
setup ‹add-forward-prfstep-cond @{thm op-int-is-interval} [with-term op-int (?ops
! ?k)]›

lemma rect-inter-next-rule [hoare-triple]:
  is-rect-list rects ⟹ k < length (all-ops rects) ⟹
   <a ↦a all-ops rects * int-tree-set S b>
   rect-inter-next a b k
   <λr. a ↦a all-ops rects * int-tree-set (apply-ops-k-next rects S k) r>t by auto2

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 ↦a all-ops rects * int-tree-set S b>
   rect-inter-impl a b k
   <λr. a ↦a all-ops rects * ↑(r ⟷ rect-inter rects S k)>t
@proof
  @let d = length (all-ops rects) − k
  @strong-induct d arbitrary k S b
  @case k ≥ length (all-ops rects)
  @unfold rect-inter rects S k
```

```
  @case is-INS (all-ops rects ! k) @with
    @case has-overlap S (op-int (all-ops rects ! k))
    @case k = length (all-ops rects) − 1
    @apply-induct-hyp length (all-ops rects) − (k + 1) k + 1
    @have length (all-ops rects) − (k + 1) < d
  @end
  @case k = length (all-ops rects) − 1
  @apply-induct-hyp length (all-ops rects) − (k + 1) k + 1
  @have length (all-ops rects) − (k + 1) < d
@qed

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
   <λr. ↑(r = has-rect-overlap rects)>t by auto2

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.