

IMP2 Binary Heap

Simon Griebel

March 17, 2025

Abstract

In this submission array-based binary minimum heaps are formalized. The correctness of the following heap operations is proven: insert, get-min, delete-min and make-heap. These are then used to verify an in-place heapsort. The formalization is based on IMP2, an imperative program verification framework implemented in Isabelle/HOL. The verified heap functions are iterative versions of the partly recursive functions found in “Algorithms and Data Structures – The Basic Toolbox” by K. Mehlhorn and P. Sanders and “Introduction to Algorithms” by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.

Contents

1	Introduction	3
2	Heap Related Definitions and Theorems	3
2.1	Array Bounds	3
2.2	Parent and Children	3
2.2.1	Definitions	3
2.2.2	Lemmas	4
2.3	Heap Invariants	5
2.3.1	Definitions	5
2.3.2	Lemmas	6
2.3.3	First Heap Element	8
3	General Lemmas on Arrays	10
3.1	Lemmas on <i>mset-ran</i>	10
3.2	Lemmas on <i>swap</i> and <i>eq-on</i>	11
4	Imperative Heap Implementation	13
4.1	Simple Functions	13
4.1.1	Parent, Children and Swap	13
4.1.2	<i>get-min</i>	13
4.2	Modifying Functions	14

4.2.1	<i>sift-up</i> and <i>insert</i>	14
4.2.2	<i>sift-down</i> , <i>del-min</i> and <i>make-heap</i>	17
4.3	Heapsort Implementation	20
4.3.1	Auxiliary Lemmas	21
4.3.2	Implementation	23

```

theory IMP2-Binary-Heap
  imports IMP2.IMP2 IMP2.IMP2-Aux-Lemmas
begin

```

1 Introduction

In this submission imperative versions of the following array-based binary minimum heap functions are implemented and verified: insert, get-min, delete-min, make-heap. The latter three are then used to prove the correctness of an in-place heapsort, which sorts an array in descending order. To do that in Isabelle/HOL, the proof framework IMP2 [2] is used. Here arrays are modeled by $int \Rightarrow int$ functions. The imperative implementations are iterative versions of the partly recursive algorithms described in [3] and [1].

This submission starts with the basic definitions and lemmas, which are needed for array-based binary heaps. These definitions and lemmas are parameterised with an arbitrary (transitive) comparison function (where such a function is needed), so they are not only applicable to minimum heaps. After some more general, useful lemmas on arrays, the imperative minimum heap functions and the heapsort are implemented and verified.

2 Heap Related Definitions and Theorems

2.1 Array Bounds

A small helper function is used to define valid array indices. Note that the lower index bound l is arbitrary and not fixed to 0 or 1. The upper index bound r is not a valid index itself, so that the empty array can be denoted by having $l = r$.

abbreviation *bounded* :: $int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
bounded $l\ r\ x \equiv l \leq x \wedge x < r$

2.2 Parent and Children

2.2.1 Definitions

For the notion of an array-based binary heap, the parent and child relations on the array indices need to be defined.

definition *parent* :: $int \Rightarrow int \Rightarrow int$ **where**
parent $l\ c = l + (c - l - 1) \text{ div } 2$

definition *l-child* :: $int \Rightarrow int \Rightarrow int$ **where**
l-child $l\ p = 2 * p - l + 1$

definition *r-child* :: *int* \Rightarrow *int* \Rightarrow *int* **where**
 r-child *l p* = 2 * *p* - *l* + 2

2.2.2 Lemmas

lemma *parent-upper-bound*: *parent l c* < *c* \longleftrightarrow *l* \leq *c*
 unfolding *parent-def* **by** *auto*

lemma *parent-upper-bound-alt*: *l* \leq *parent l c* \implies *parent l c* < *c*
 unfolding *parent-def* **by** *simp*

lemma *parent-lower-bound*: *l* \leq *parent l c* \longleftrightarrow *l* < *c*
 unfolding *parent-def* **by** *linarith*

lemma *grand-parent-upper-bound*: *parent l (parent l c)* < *c* \longleftrightarrow *l* \leq *c*
 unfolding *parent-def* **by** *linarith*

corollary *parent-bounds*: *l* < *x* \implies *x* < *r* \implies *bounded l r (parent l x)*
 using *parent-lower-bound* *parent-upper-bound-alt* **by** *fastforce*

lemma *l-child-lower-bound*: *p* < *l-child l p* \longleftrightarrow *l* \leq *p*
 unfolding *l-child-def* **by** *simp*

corollary *l-child-lower-bound-alt*: *l* \leq *x* \implies *x* \leq *p* \implies *x* < *l-child l p*
 using *l-child-lower-bound[of p l]* **by** *linarith*

lemma *parent-l-child[simp]*: *parent l (l-child l n)* = *n*
 unfolding *parent-def* *l-child-def* **by** *simp*

lemma *r-child-lower-bound*: *l* \leq *p* \implies *p* < *r-child l p*
 unfolding *r-child-def* **by** *simp*

corollary *r-child-lower-bound-alt*: *l* \leq *x* \implies *x* \leq *p* \implies *x* < *r-child l p*
 using *r-child-lower-bound[of l p]* **by** *linarith*

lemma *parent-r-child[simp]*: *parent l (r-child l n)* = *n*
 unfolding *parent-def* *r-child-def* **by** *simp*

lemma *smaller-l-child*: *l-child l x* < *r-child l x*
 unfolding *l-child-def* *r-child-def* **by** *simp*

lemma *parent-two-children*:

$$(c = l\text{-child } l \ p \vee c = r\text{-child } l \ p) \longleftrightarrow \text{parent } l \ c = p$$

unfolding *parent-def l-child-def r-child-def* **by** *linarith*

2.3 Heap Invariants

2.3.1 Definitions

The following heap invariants and the following lemmas are parameterised with an arbitrary (transitive) comparison function. For the concrete function implementations at the end of this submission \leq on ints is used.

For the *make-heap* function, which transforms an unordered array into a valid heap, the notion of a partial heap is needed. Here the heap invariant only holds for array indices between a certain valid array index m and r . The standard heap invariant is then simply the special case where $m = l$.

definition *is-partial-heap*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\textit{is-partial-heap cmp heap } l \ m \ r = (\forall x. \textit{bounded } m \ r \ x \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ x) \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \ x)) \ (\textit{heap } x))$$

abbreviation *is-heap*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\textit{is-heap cmp heap } l \ r \equiv \textit{is-partial-heap cmp heap } l \ l \ r$$

During all of the modifying heap functions the heap invariant is temporarily violated at a single index i and it is then gradually restored by either *sift-down* or *sift-up*. The following definitions formalize these weakened invariants.

The second part of the conjunction in the following definitions states, that the comparison between the parent of i and each of the children of i evaluates to *True* without explicitly using the child relations.

definition *is-partial-heap-except-down*

$$:: ('a::\text{order} \Rightarrow 'a::\text{order} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow 'a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \textbf{ where}$$

$$\begin{aligned} \textit{is-partial-heap-except-down cmp heap } l \ m \ r \ i = & (\forall x. \textit{bounded } m \ r \ x \longrightarrow \\ & ((\textit{parent } l \ x \neq i \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ x) \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \\ & x)) \ (\textit{heap } x)) \wedge \\ & (\textit{parent } l \ x = i \longrightarrow \textit{bounded } m \ r \ (\textit{parent } l \ (\textit{parent } l \ x)) \\ & \longrightarrow \textit{cmp } (\textit{heap } (\textit{parent } l \ (\textit{parent } l \ x))) \ (\textit{heap } x)))) \end{aligned}$$

abbreviation *is-heap-except-down*

$:: ('a::order \Rightarrow 'a::order \Rightarrow bool) \Rightarrow (int \Rightarrow 'a::order) \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
is-heap-except-down *cmp heap l r i* \equiv *is-partial-heap-except-down* *cmp heap l l r i*

As mentioned the notion of a partial heap is only needed for *make-heap*, which only uses *sift-down* internally, so there doesn't need to be an additional definition for the partial heap version of the *sift-up* invariant.

definition *is-heap-except-up*

$:: ('a::order \Rightarrow 'a::order \Rightarrow bool) \Rightarrow (int \Rightarrow 'a::order) \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
is-heap-except-up *cmp heap l r i* $= (\forall x. \text{bounded } l r x \longrightarrow ((x \neq i \longrightarrow \text{bounded } l r (\text{parent } l x) \longrightarrow \text{cmp } (\text{heap } (\text{parent } l x)) (\text{heap } x)) \wedge (\text{parent } l x = i \longrightarrow \text{bounded } l r (\text{parent } l (\text{parent } l x)) \longrightarrow \text{cmp } (\text{heap } (\text{parent } l (\text{parent } l x))) (\text{heap } x))))$

2.3.2 Lemmas

lemma *empty-partial-heap[simp]*: *is-partial-heap* *cmp heap l r r*
unfolding *is-partial-heap-def* **by** *linarith*

lemma *is-partial-heap-smaller-back*:

is-partial-heap *cmp heap l m r* $\implies r' \leq r \implies$ *is-partial-heap* *cmp heap l m r'*
unfolding *is-partial-heap-def* **by** *simp*

lemma *is-partial-heap-smaller-front*:

is-partial-heap *cmp heap l m r* $\implies m \leq m' \implies$ *is-partial-heap* *cmp heap l m' r*
unfolding *is-partial-heap-def* **by** *simp*

The second half of each array is a partial binary heap, since it contains only leaves, which are all trivial binary heaps.

lemma *snd-half-is-partial-heap*:

$(l + r) \text{ div } 2 \leq m \implies$ *is-partial-heap* *cmp heap l m r*
unfolding *is-partial-heap-def* *parent-def* **by** *linarith*

lemma *modify-outside-partial-heap*:

assumes
heap = heap' *on* $\{m..<r\}$
is-partial-heap *cmp heap l m r*
shows *is-partial-heap* *cmp heap' l m r*

using *assms eq-onD unfolding is-partial-heap-def by fastforce*

The next few lemmas formalize how the heap invariant is weakened, when the heap is modified in a certain way.

This lemma is used by *make-heap*.

lemma *partial-heap-added-first-el:*

assumes

$l \leq m \ m \leq r$

is-partial-heap cmp heap l (m + 1) r

shows *is-partial-heap-except-down cmp heap l m r m*

unfolding *is-partial-heap-except-down-def*

proof

fix *x*

let *?p-x = parent l x*

let *?gp-x = parent l ?p-x*

show *bounded m r x \longrightarrow*

(?p-x \neq m \longrightarrow bounded m r ?p-x \longrightarrow cmp (heap ?p-x) (heap x)) \wedge

(?p-x = m \longrightarrow bounded m r ?gp-x \longrightarrow cmp (heap ?gp-x) (heap x))

proof

assume *x-bound: bounded m r x*

have *p-x-lower: ?p-x \neq m \longrightarrow bounded m r ?p-x \longrightarrow ?p-x \geq m + 1*

by *simp*

hence *?p-x \neq m \longrightarrow bounded m r ?p-x \longrightarrow x \geq m + 1*

using *parent-upper-bound[of l x] x-bound assms(1) by linarith*

hence *p-invariant: (?p-x \neq m \longrightarrow bounded m r ?p-x \longrightarrow cmp (heap ?p-x) (heap x))*

using *assms(3) is-partial-heap-def p-x-lower x-bound by blast*

have *gp-up-bound: (?p-x = m \longrightarrow ?gp-x < m)*

by *(simp add: assms(1) parent-upper-bound)*

show *(?p-x \neq m \longrightarrow bounded m r ?p-x \longrightarrow cmp (heap ?p-x) (heap x))*

\wedge

(?p-x = m \longrightarrow bounded m r ?gp-x \longrightarrow cmp (heap ?gp-x) (heap x))

using *gp-up-bound p-invariant by linarith*

qed

qed

This lemma is used by *del-min*.

lemma *heap-changed-first-el:*

assumes *is-heap cmp heap l r l \leq r*

shows *is-heap-except-down cmp (heap(l := b)) l r l*

proof –

have *is-partial-heap cmp heap l (l + 1) r*

```

    using assms(1) is-partial-heap-smaller-front by fastforce
  hence is-partial-heap cmp (heap(l := b)) l (l + 1) r
    using modify-outside-partial-heap[of heap] by simp
  thus ?thesis
    by (simp add: assms(2) partial-heap-added-first-el)
qed

```

This lemma is used by *insert*.

```

lemma heap-appended-el:
  assumes
    is-heap cmp heap l r
    heap = heap' on {l..<r}
  shows is-heap-except-up cmp heap' l (r+1) r
proof –
  have is-heap cmp heap' l r
    using assms(1,2) modify-outside-partial-heap by blast
  thus ?thesis unfolding is-partial-heap-def is-heap-except-up-def
    by (metis not-less-iff-gr-or-eq parent-upper-bound zless-add1-eq)
qed

```

2.3.3 First Heap Element

The next step is to show that the first element of the heap is always the “smallest” according to the given comparison function. For the proof a rule for strong induction on lower bounded integers is needed. Its proof is based on the proof of strong induction on natural numbers found in [4].

```

lemma strong-int-gr-induct-helper:
  assumes  $k < (i::int) (\bigwedge i. k < i \implies (\bigwedge j. k < j \implies j < i \implies P j) \implies P i)$ 
  shows  $\bigwedge j. k < j \implies j < i \implies P j$ 
  using assms
proof(induction i rule: int-gr-induct)
  case base
  then show ?case by linarith
next
  case (step i)
  then show ?case
  proof(cases j = i)
    case True
    then show ?thesis using step.IH step.prem(1,3) by blast
  next
  case False
  hence  $j < i$  using step.prem(2) by simp
  then show ?thesis using step.IH step.prem(1,3) by blast

```


qed
qed

theorem *strong-int-gr-induct*:

assumes

$k < (i::int)$

$(\wedge i. k < i \implies (\wedge j. k < j \implies j < i \implies P j) \implies P i)$

shows $P i$

using *assms less-induct strong-int-gr-induct-helper* **by** *blast*

Now the main theorem, that the first heap element is the “smallest” according to the given comparison function, can be proven.

theorem *heap-first-el*:

assumes

is-heap cmp heap l r

transp cmp

$l < x \ x < r$

shows *cmp (heap l) (heap x)*

using *assms unfolding is-partial-heap-def*

proof(*induction x rule: strong-int-gr-induct[of l]*)

case *1*

then show *?case* **using** *assms(3)* **by** *simp*

next

case (*2 i*)

have *cmp-pi-i: cmp (heap (parent l i)) (heap i)*

using *2.hyps 2.prem(1,4) parent-bounds* **by** *simp*

then show *?case*

proof(*cases*)

assume *parent l i > l*

then have *cmp (heap l) (heap (parent l i))*

using *2.IH 2.prem(1,2,4) parent-upper-bound-alt* **by** *simp*

then show *?thesis*

using *2.prem(2) cmp-pi-i transpE* **by** *metis*

next

assume \neg *parent l i > l*

then have *parent l i = l*

using *2.hyps dual-order.order-iff-strict parent-lower-bound* **by** *metis*

then show *?thesis*

using *cmp-pi-i* **by** *simp*

qed

qed

3 General Lemmas on Arrays

Some additional lemmas on *mset-ran*, *swap* and *eq-on* are needed for the final proofs.

3.1 Lemmas on *mset-ran*

abbreviation $arr\text{-}mset :: (int \Rightarrow 'a) \Rightarrow int \Rightarrow int \Rightarrow 'a$ *multiset* **where**
 $arr\text{-}mset\ arr\ l\ r \equiv mset\text{-}ran\ arr\ \{l..<r\}$

lemma *in-mset-imp-in-array*:

$x \in \# (arr\text{-}mset\ arr\ l\ r) \longleftrightarrow (\exists i. \text{bounded}\ l\ r\ i \wedge arr\ i = x)$

unfolding *mset-ran-def* **by** *fastforce*

lemma *arr-mset-remove-last*:

$l \leq r \implies arr\text{-}mset\ arr\ l\ r = arr\text{-}mset\ arr\ l\ (r + 1) - \{\#arr\ r\# \}$

by (*simp add: intvs-upper-decr mset-ran-def*)

lemma *arr-mset-append*:

$l \leq r \implies arr\text{-}mset\ arr\ l\ (r + 1) = arr\text{-}mset\ arr\ l\ r + \{\#arr\ r\# \}$

using *arr-mset-remove-last*[of *l r arr*] **by** *simp*

corollary *arr-mset-append-alt*:

$l \leq r \implies arr\text{-}mset\ (arr(r := b))\ l\ (r + 1) = arr\text{-}mset\ arr\ l\ r + \{\#b\# \}$

by (*simp add: arr-mset-append mset-ran-subst-outside*)

lemma *arr-mset-remove-first*:

$i \leq r \implies arr\text{-}mset\ arr\ (i - 1)\ r = arr\text{-}mset\ arr\ i\ r + \{\#arr\ (i - 1)\# \}$

by(*induction r rule: int-ge-induct*) (*auto simp add: arr-mset-append*)

lemma *arr-mset-split*:

assumes $l \leq m\ m \leq r$

shows $arr\text{-}mset\ arr\ l\ r = arr\text{-}mset\ arr\ l\ m + arr\text{-}mset\ arr\ m\ r$

using *assms*

proof(*induction m rule: int-ge-induct*[of *l*])

case (*step i*)

have *add-last*: $arr\text{-}mset\ arr\ l\ (i + 1) = arr\text{-}mset\ arr\ l\ i + \{\#arr\ i\# \}$

using *step arr-mset-append* **by** *blast*

have *rem-first*: $arr\text{-}mset\ arr\ (i+1)\ r = arr\text{-}mset\ arr\ i\ r - \{\#arr\ i\# \}$

by (*metis step.prem1 arr-mset-remove-first add-diff-cancel-right*)

show *?case*

using *step add-last rem-first* **by** *fastforce*

qed (*simp*)

That the first element in a heap is the “smallest”, can now be expressed using multisets.

corollary *heap-first-el-alt*:

assumes

transp cmp

is-heap cmp heap l r

$x \in \# (\text{arr-mset heap } l \ r)$

heap l \neq x

shows *cmp (heap l) x*

by (*metis assms heap-first-el in-mset-imp-in-array le-less*)

3.2 Lemmas on *swap* and *eq-on*

lemma *eq-on-subset*:

$\text{arr1} = \text{arr2 on } R \implies S \subseteq R \implies \text{arr1} = \text{arr2 on } S$

by (*simp add: eq-on-def set-mp*)

lemma *swap-swaps*:

$\text{arr}' = \text{swap arr } x \ y \implies \text{arr}' y = \text{arr } x \wedge \text{arr}' x = \text{arr } y$

unfolding *swap-def* **by** *simp*

lemma *swap-only-swaps*:

$\text{arr}' = \text{swap arr } x \ y \implies z \neq x \implies z \neq y \implies \text{arr}' z = \text{arr } z$

unfolding *swap-def* **by** *simp*

lemma *swap-commute*: $\text{swap arr } x \ y = \text{swap arr } y \ x$

unfolding *swap-def* **by** *fastforce*

lemma *swap-eq-on*:

$\text{arr1} = \text{arr2 on } S \implies x \notin S \implies y \notin S \implies \text{arr1} = \text{swap arr2 } x \ y \text{ on } S$

unfolding *swap-def* **by** *simp*

corollary *swap-parent-eq-on*:

assumes

$\text{arr1} = \text{arr2 on } - \{l..<r\}$

$l < c < r$

shows $\text{arr1} = \text{swap arr2 (parent } l \ c) \ c \text{ on } - \{l..<r\}$

using *parent-bounds swap-eq-on assms* **by** *fastforce*

corollary *swap-child-eq-on*:

assumes

$\text{arr1} = \text{arr2 on } - \{l..<r\}$

$c = l\text{-child } l \ p \vee c = r\text{-child } l \ p$

$l \leq p < r$

shows $arr1 = swap\ arr2\ p\ c\ on - \{l..<r\}$
by (*metis assms parent-lower-bound parent-two-children swap-parent-eq-on*)

lemma *swap-child-mset*:

assumes

$arr\text{-}mset\ arr1\ l\ r = arr\text{-}mset\ arr2\ l\ r$

$c = l\text{-}child\ l\ p \vee c = r\text{-}child\ l\ p$

$l \leq p < c < r$

shows $arr\text{-}mset\ arr1\ l\ r = arr\text{-}mset\ (swap\ arr2\ p\ c)\ l\ r$

proof –

have *child-bounded*: $l < c \wedge c < r$

by (*metis assms(2-4) parent-lower-bound parent-two-children*)

have *parent-bounded*: $bounded\ l\ r\ p$

by (*metis assms(2-4) dual-order.strict-trans parent-two-children parent-upper-bound-alt*)

thus *?thesis*

using *assms(1) child-bounded mset-ran-swap[of p {l..<r} c arr2] atLeast-LessThan-iff*

by *simp*

qed

The following lemma shows, which propositions have to hold on the pre-swap array, so that a comparison between two elements holds on the post-swap array. This is useful for the proofs of the loop invariants of *sift-up* and *sift-down*. The lemma is kept quite general (except for the argument order) and could probably be more closely related to the parent relation for more concise proofs.

lemma *cmp-swapI*:

fixes $arr::'a::order \Rightarrow 'a::order$

assumes

$m < n \wedge x < y$

$m < n \wedge x < y \Longrightarrow x = m \Longrightarrow y = n \Longrightarrow P\ (arr\ n)\ (arr\ m)$

$m < n \wedge x < y \Longrightarrow x \neq m \Longrightarrow x \neq n \Longrightarrow y \neq m \Longrightarrow y \neq n \Longrightarrow P\ (arr\ m)\ (arr\ n)$

$m < n \wedge x < y \Longrightarrow x = m \Longrightarrow y \neq m \Longrightarrow y \neq n \Longrightarrow P\ (arr\ y)\ (arr\ n)$

$m < n \wedge x < y \Longrightarrow x = n \Longrightarrow y \neq m \Longrightarrow y \neq n \Longrightarrow P\ (arr\ m)\ (arr\ y)$

$m < n \wedge x < y \Longrightarrow x \neq m \Longrightarrow x \neq n \Longrightarrow y = n \Longrightarrow P\ (arr\ m)\ (arr\ x)$

$m < n \wedge x < y \Longrightarrow x \neq m \Longrightarrow x \neq n \Longrightarrow y = m \Longrightarrow P\ (arr\ x)\ (arr\ n)$

shows $P\ (swap\ arr\ x\ y\ m)\ (swap\ arr\ x\ y\ n)$

by (*metis assms order.asym swap-only-swaps swap-swaps*)

4 Imperative Heap Implementation

The following imperative heap functions are based on [3] and [1]. All functions, that are recursive in these books, are iterative in the following implementations. The function definitions are done with IMP2 [2]. From now on the heaps only contain ints and only use \leq as comparison function. The auxiliary lemmas used from now on are heavily modeled after the proof goals, that are generated by the vcg tool (also part of IMP2).

4.1 Simple Functions

4.1.1 Parent, Children and Swap

In this section the parent and children relations are expressed as IMP2 procedures. Additionally a simple procedure, that swaps two array elements, is defined.

```
procedure-spec prnt (l, x) returns p  
  assumes True  
  ensures  $p = \text{parent } l_0 \ x_0$   
  defines  $\langle p = ((x - l - 1) / 2 + l) \rangle$   
  by vcg (simp add: parent-def)
```

```
procedure-spec left-child (l, x) returns lc  
  assumes True  
  ensures  $lc = \text{l-child } l_0 \ x_0$   
  defines  $\langle lc = 2 * x - l + 1 \rangle$   
  by vcg (simp add: l-child-def)
```

```
procedure-spec right-child (l, x) returns rc  
  assumes True  
  ensures  $rc = \text{r-child } l_0 \ x_0$   
  defines  $\langle rc = 2 * x - l + 2 \rangle$   
  by vcg (simp add: r-child-def)
```

```
procedure-spec swp (heap, x, y) returns heap  
  assumes True  
  ensures  $\text{heap} = \text{swap } \text{heap}_0 \ x_0 \ y_0$   
  defines  $\langle \text{tmp} = \text{heap}[x]; \text{heap}[x] = \text{heap}[y]; \text{heap}[y] = \text{tmp} \rangle$   
  by vcg (simp add: swap-def)
```

4.1.2 *get-min*

In this section *get-min* is defined, which simply returns the first element (the minimum) of the heap. For this definition an additional theorem is proven,

which enables the use of *Min-mset* in the postcondition.

theorem *heap-minimum*:

assumes

$l < r$

is-heap (\leq) *heap* $l\ r$

shows *heap* $l = \text{Min-mset} (\text{arr-mset } \text{heap } l\ r)$

proof –

have $(\forall x \in \# (\text{arr-mset } \text{heap } l\ r). (\text{heap } l) \leq x)$

using *assms*(2) *heap-first-el-alt transp-on-le* **by** *blast*

thus *?thesis*

by (*simp add: assms*(1) *dual-order.antisym*)

qed

procedure-spec *get-min* (*heap*, l , r) **returns** *min*

assumes $l < r \wedge \text{is-heap } (\leq) \text{ heap } l\ r$

ensures $\text{min} = \text{Min-mset} (\text{arr-mset } \text{heap}_0\ l_0\ r_0)$

for *heap*[] $l\ r$

defines $\langle \text{min} = \text{heap}[l] \rangle$

by *vcg* (*simp add: heap-minimum*)

4.2 Modifying Functions

4.2.1 *sift-up* and *insert*

The next heap function is *insert*, which internally uses *sift-up*. In the beginning of this section *sift-up-step* is proven, which states that each *sift-up* loop iteration correctly transforms the weakened heap invariant. For its proof two additional auxiliary lemmas are used. After *sift-up-step* *sift-up* and then *insert* are verified.

sift-up-step can be proven directly by the smt-solver without auxiliary lemmas, but they were introduced to show the proof details. The analogous proofs for *sift-down* were just solved with smt, since the proof structure should be very similar, even though the *sift-down* proof goals are slightly more complex.

lemma *sift-up-step-aux1*:

fixes *heap*::*int* \Rightarrow *int*

assumes

is-heap-except-up (\leq) *heap* $l\ r\ x$

parent $l\ x \geq l$

$(\text{heap } x) \leq (\text{heap } (\text{parent } l\ x))$

bounded $l\ r\ k$

$k \neq (\text{parent } l\ x)$

bounded $l\ r (\text{parent } l\ k)$

shows $(\text{swap heap } (\text{parent } l \ x) \ x \ (\text{parent } l \ k)) \leq (\text{swap heap } (\text{parent } l \ x) \ x \ k)$
apply $(\text{intro cmp-swapI}[\text{of } (\text{parent } l \ k) \ k \ (\text{parent } l \ x) \ x \ (\leq) \ \text{heap}]])$
subgoal using $\text{assms}(2,6)$ **parent-upper-bound-alt** **by** *blast*
subgoal using $\text{assms}(3)$ **by** *blast*
subgoal using $\text{assms}(1,4,6)$ **unfolding** *is-heap-except-up-def* **by** *blast*
subgoal using $\text{assms}(1,3,4,6)$ **unfolding** *is-heap-except-up-def* **by** *fast-force*
subgoal using $\text{assms}(5)$ **by** *blast*
subgoal by *blast*
subgoal using $\text{assms}(1,2,4)$ **unfolding** *is-heap-except-up-def* **by** *simp*
done

lemma *sift-up-step-aux2*:

fixes $\text{heap}::\text{int} \Rightarrow \text{int}$

assumes

is-heap-except-up (\leq) $\text{heap } l \ r \ x$

$\text{parent } l \ x \geq l$

$\text{heap } x \leq (\text{heap } (\text{parent } l \ x))$

bounded $l \ r \ k$

$\text{parent } l \ k = \text{parent } l \ x$

bounded $l \ r \ (\text{parent } l \ (\text{parent } l \ k))$

shows

$\text{swap heap } (\text{parent } l \ x) \ x \ (\text{parent } l \ (\text{parent } l \ k)) \leq \text{swap heap } (\text{parent } l \ x) \ x \ k$

using assms **unfolding** *is-heap-except-up-def*

proof–

let $?gp-k = \text{parent } l \ (\text{parent } l \ k)$

let $?gp-x = \text{parent } l \ (\text{parent } l \ x)$

have $gp-k\text{-eq-gp-x}$: $\text{swap heap } (\text{parent } l \ x) \ x \ ?gp-k = \text{heap } ?gp-x$

by $(\text{metis } \text{assms}(2,5) \ \text{grand-parent-upper-bound} \ \text{less-irrefl} \ \text{swap-only-swaps})$

show $?thesis$

using assms **unfolding** *is-heap-except-up-def*

proof(*cases*)

assume $k\text{-eq-x}$: $k = x$

have $\text{swap heap } (\text{parent } l \ x) \ x \ k = \text{heap } (\text{parent } l \ x)$

by $(\text{metis } k\text{-eq-x} \ \text{swap-swaps})$

then show $?thesis$

using $\text{assms}(1,2,4,6)$ **unfolding** *is-heap-except-up-def*

by $(\text{metis } gp-k\text{-eq-gp-x} \ k\text{-eq-x} \ \text{parent-bounds} \ \text{parent-lower-bound})$

next

assume $k\text{-neq-x}$: $k \neq x$

have $\text{swap heap } (\text{parent } l \ x) \ x \ k = \text{heap } k$

by $(\text{metis } \text{assms}(5) \ gp-k\text{-eq-gp-x} \ k\text{-neq-x} \ \text{swap-only-swaps})$

then show *?thesis* **using** *assms unfolding is-heap-except-up-def*
by (*metis gp-k-eq-gp-x k-neq-x order-trans parent-bounds parent-lower-bound*)
qed
qed

lemma *sift-up-step*:

fixes *heap::int* \Rightarrow *int*

assumes

is-heap-except-up (\leq) *heap l r x*

parent l x $\geq l$

$(\text{heap } x) \leq (\text{heap } (\text{parent } l \ x))$

shows *is-heap-except-up* (\leq) (*swap heap (parent l x) x*) *l r (parent l x)*

using *assms sift-up-step-aux1 sift-up-step-aux2*

unfolding *is-heap-except-up-def* **by** *blast*

sift-up restores the heap invariant, that is only violated at the current position, by iteratively swapping the current element with its parent until the beginning of the array is reached or the current element is bigger than its parent.

procedure-spec *sift-up (heap, l, r, x)* **returns** *heap*

assumes *is-heap-except-up* (\leq) *heap l r x* \wedge *bounded l r x*

ensures *is-heap* (\leq) *heap l₀ r₀* \wedge

arr-mset heap₀ l₀ r₀ = *arr-mset heap l r* \wedge

heap₀ = *heap on - {l₀..0}*

for *heap[] l x r*

defines \langle

p = *prnt(l, x)*;

while (*x* > *l* \wedge *heap[x]* \leq *heap[p]*)

$\text{@variant } \langle x - l \rangle$

$\text{@invariant } \langle \text{is-heap-except-up } (\leq) \text{ heap } l \ r \ x \wedge p = \text{parent } l \ x \wedge$

bounded l r x \wedge *arr-mset heap₀ l₀ r₀* = *arr-mset heap l r* \wedge

heap₀ = *heap on - {l₀..0}* \rangle

{

heap = *swp(heap, p, x)*;

x = *p*;

p = *prnt(l, x)*

} \rangle

apply *vcg-cs*

apply(*intro conjI*)

subgoal using *parent-lower-bound sift-up-step* **by** *blast*

subgoal using *parent-lower-bound* **by** *blast*

subgoal using *parent-bounds* **by** *blast*

subgoal using *parent-bounds* **by** (*simp add: mset-ran-swap*)

subgoal using *swap-parent-eq-on* **by** *blast*


```

subgoal using parent-upper-bound by simp
subgoal unfolding is-heap-except-up-def is-partial-heap-def
  by (metis le-less not-less parent-lower-bound)
done

```

insert inserts an element into a heap by appending it to the heap and restoring the heap invariant with *sift-up*.

```

procedure-spec insert (heap, l, r, el) returns (heap, l, r)
  assumes is-heap ( $\leq$ ) heap l r  $\wedge$   $l \leq r$ 
  ensures is-heap ( $\leq$ ) heap l r  $\wedge$ 
     $arr\text{-}mset\ heap\ l\ r = arr\text{-}mset\ heap_0\ l_0\ r_0 + \{\#el_0\# \} \wedge$ 
     $l = l_0 \wedge r = r_0 + 1 \wedge heap_0 = heap\ on - \{l..<r\}$ 
  for heap l r el
  defines  $\langle$ 
    heap[r] = el;
    x = r;
    r = r + 1;
    heap = sift-up(heap, l, r, x)
   $\rangle$ 
  apply vcg-cs
  subgoal by (simp add: heap-appended-el)
  subgoal by (metis arr-mset-append-alt add-mset-add-single)
  done

```

4.2.2 *sift-down*, *del-min* and *make-heap*

The next heap functions are *del-min* and *make-heap*, which both use *sift-down* to restore/establish the heap invariant. *sift-down* is proven first (this time without additional auxiliary lemmas) followed by *del-min* and *make-heap*.

sift-down restores the heap invariant, that is only violated at the current position, by iteratively swapping the current element with its smallest child until the end of the array is reached or the current element is smaller than its children.

```

procedure-spec sift-down(heap, l, r, x) returns heap
  assumes is-partial-heap-except-down ( $\leq$ ) heap l x r  $\wedge$   $l \leq x \wedge x \leq r$ 
  ensures is-partial-heap ( $\leq$ ) heap l_0 x_0 r_0  $\wedge$ 
     $arr\text{-}mset\ heap_0\ l_0\ r_0 = arr\text{-}mset\ heap\ l_0\ r_0 \wedge$ 
     $heap_0 = heap\ on - \{l_0..<r_0\}$ 
  defines  $\langle$ 
    lc = left-child(l, x);
    rc = right-child(l, x);
    while ( $lc < r \wedge (heap[lc] < heap[x] \vee (rc < r \wedge heap[rc] < heap[x]))$ )
      @variant  $\langle r - x \rangle$ 
   $\rangle$ 

```

```

    @invariant ⟨is-partial-heap-except-down (≤) heap l x0 r x ∧
                x0 ≤ x ∧ x ≤ r ∧ lc = l-child l x ∧ rc = r-child l x ∧
                arr-mset heap0 l r = arr-mset heap l r ∧
                heap0 = heap on - {l..<r}⟩
  {
    smallest = lc;
    if (rc < r ∧ heap[rc] < heap[lc]) {
      smallest = rc
    };
    heap = swp(heap, x, smallest);
    x = smallest;
    lc = left-child(l, x);
    rc = right-child(l, x)
  }
apply vcg-cs
subgoal
  apply(intro conjI)
  subgoal unfolding is-partial-heap-except-down-def
    by (smt parent-two-children swap-swaps swap-only-swaps
         swap-commute parent-upper-bound-alt)
  subgoal using r-child-lower-bound-alt by fastforce
  subgoal using swap-child-mset order-trans by blast
  subgoal using swap-child-eq-on by fastforce
  done
subgoal
  by (meson less-le-trans not-le order.asym r-child-lower-bound)
subgoal
  apply(intro conjI)
  subgoal unfolding is-partial-heap-except-down-def
    by (smt parent-two-children swap-swaps swap-only-swaps
         swap-commute parent-upper-bound-alt)
  subgoal using l-child-lower-bound-alt by fastforce
  subgoal using swap-child-mset order-trans by blast
  subgoal using swap-child-eq-on by fastforce
  done
subgoal
  by (meson less-le-trans not-le order.asym l-child-lower-bound)
subgoal unfolding is-partial-heap-except-down-def is-partial-heap-def
  by (metis dual-order.strict-trans not-less parent-two-children smaller-l-child)
done

```

del-min needs an additional lemma which shows, that it actually removes (only) the minimum from the heap.

lemma *del-min-mset*:

```

fixes heap::int  $\Rightarrow$  int
assumes
  l < r
  is-heap ( $\leq$ ) heap l r
  mod-heap = heap(l := heap (r - 1))
  arr-mset mod-heap l (r - 1) = arr-mset new-heap l (r - 1)
shows
  arr-mset new-heap l (r - 1) = arr-mset heap l r - {#Min-mset (arr-mset
heap l r)#}
proof -
  let ?heap-mset = arr-mset heap l r
  have l-is-min: heap l = Min-mset ?heap-mset
    using assms(1,2) heap-minimum by blast
  have (arr-mset mod-heap l r) = ?heap-mset + {#heap (r-1)#} - {#heap
l#}
    by (simp add: assms(1,3) mset-ran-subst-inside)
  hence (arr-mset mod-heap l (r - 1)) = ?heap-mset - {#heap l#}
    by (simp add: assms(1,3) arr-mset-remove-last)
  thus ?thesis
    using assms(4) l-is-min by simp
qed

```

del-min removes the minimum element from the heap by replacing the first element with the last element, shrinking the array by one and subsequently restoring the heap invariant with *sift-down*.

```

procedure-spec del-min (heap, l, r) returns (heap, l, r)
assumes l < r  $\wedge$  is-heap ( $\leq$ ) heap l r
ensures is-heap ( $\leq$ ) heap l r  $\wedge$ 
  arr-mset heap l r = arr-mset heap0 l0 r0 - {#Min-mset (arr-mset
heap0 l0 r0)#}  $\wedge$ 
  l = l0  $\wedge$  r = r0 - 1  $\wedge$ 
  heap0 = heap on - {l0..0r0}
for heap l r
defines  $\langle$ 
  r = r - 1;
  heap[l] = heap[r];
  heap = sift-down(heap, l, r, l)
 $\rangle$ 
apply vcg-cs
subgoal by (simp add: heap-changed-first-el is-partial-heap-smaller-back)
subgoal
  apply(rule conjI)
  subgoal using del-min-mset by blast
  subgoal by (simp add: eq-on-def intvs-incdec(3) intvs-lower-incr)

```

done
done

make-heap transforms an arbitrary array into a heap by iterating through all array positions from the middle of the array up to the beginning of the array and calling *sift-down* for each one.

procedure-spec *make-heap* (*heap*, *l*, *r*) **returns** *heap*
assumes $l \leq r$
ensures *is-heap* (\leq) *heap* l_0 $r_0 \wedge$
 $arr\text{-}mset\ heap\ l_0\ r_0 = arr\text{-}mset\ heap_0\ l_0\ r_0 \wedge$
 $heap_0 = heap\ on\ -\ \{l_0..<r_0\}$
for *heap*[] *l* *r*
defines \langle
 $y = (r + l)/2 - 1;$
 $while\ (y \geq l)$
 $\quad @variant\ \langle y - l + 1 \rangle$
 $\quad @invariant\ \langle is\text{-}partial\text{-}heap\ (\leq)\ heap\ l\ (y + 1)\ r \wedge$
 $\quad \quad arr\text{-}mset\ heap\ l\ r = arr\text{-}mset\ heap_0\ l_0\ r_0 \wedge$
 $\quad \quad l - 1 \leq y \wedge y < r \wedge heap_0 = heap\ on\ -\ \{l..<r\} \rangle$
 $\{$
 $\quad heap = sift\text{-}down(heap, l, r, y);$
 $\quad y = y - 1$
 $\}$
apply(*vcg-cs*)
subgoal
apply(*rule conjI*)
subgoal by (*simp add: snd-half-is-partial-heap add.commute*)
subgoal by *linarith*
done
subgoal using *partial-heap-added-first-el le-less* **by** *blast*
subgoal using *eq-on-trans* **by** *blast*
subgoal using *dual-order.antisym* **by** *fastforce*
done

4.3 Heapsort Implementation

The final part of this submission is the implementation of the in-place heapsort. Firstly it builds the \leq -heap and then it iteratively removes the minimum of the heap, which is put at the now vacant end of the shrinking heap. This is done until the heap is empty, which leaves the array sorted in descending order.

4.3.1 Auxiliary Lemmas

Firstly the notion of a sorted array is needed. This is more or less the same as *ran-sorted* generalized for arbitrary comparison functions.

definition *array-is-sorted* :: (int ⇒ int ⇒ bool) ⇒ (int ⇒ int) ⇒ int ⇒ int ⇒ bool **where**

array-is-sorted *cmp* *a* *l* *r* ≡ ∀ *i*. ∀ *j*. bounded *l* *r* *i* → bounded *l* *r* *j* → *i* < *j* → *cmp* (*a* *i*) (*a* *j*)

This lemma states, that the heapsort doesn't change the elements contained in the array during the loop iterations.

lemma *heap-sort-mset-step*:

fixes *arr*::int ⇒ int

assumes

$l < m \leq r$

arr-mset *arr'* *l* (*m* - 1) = *arr-mset* *arr* *l* *m* - {#*Min-mset* (*arr-mset* *arr* *l* *m*)#}

arr = *arr'* *on* - {*l*..*m*}

mod-arr = *arr'*(*m* - 1 := *Min-mset* (*arr-mset* *arr* *l* *m*))

shows *arr-mset* *arr* *l* *r* = *arr-mset* *mod-arr* *l* *r*

proof -

let *?min* = {#*Min-mset* (*arr-mset* *arr* *l* *m*)#}

let *?new-arr-mset* = *arr-mset* *mod-arr*

have *middle*: *?new-arr-mset* (*m* - 1) *m* = *?min*

by (*simp* *add*: *assms*(5))

have *first-half*: *?new-arr-mset* *l* (*m* - 1) = *arr-mset* *arr* *l* *m* - *?min*

by (*simp* *add*: *assms*(3,5) *mset-ran-subst-outside*)

hence *?new-arr-mset* *l* *m* = *?new-arr-mset* *l* (*m* - 1) + *?new-arr-mset* (*m* - 1) *m*

by (*metis* *assms*(1,3,5) *diff-add-cancel* *middle* *arr-mset-append-alt* *zle-diff1-eq*)

hence *first-half-middle*: *?new-arr-mset* *l* *m* = *arr-mset* *arr* *l* *m*

using *middle* *first-half* *assms*(1) **by** *simp*

hence *mod-arr* = *arr* *on* - {*l*..*m*}

using *assms*(1,4,5) *eq-on-sym* *eq-on-trans* **by** *auto*

then have *second-half*: *arr-mset* *arr* *m* *r* = *arr-mset* *mod-arr* *m* *r*

by (*simp* *add*: *eq-on-def* *mset-ran-cong*)

then show *?thesis*

by (*metis* *arr-mset-split* *assms*(1,2) *first-half-middle* *le-less*)

qed

This lemma states, that each loop iteration leaves the growing second half of the array sorted in descending order.

lemma *heap-sort-second-half-sorted-step*:

fixes $arr::int \Rightarrow int$

assumes

$l_0 < m \ m \leq r_0$

$arr = arr' \text{ on } - \{l_0..<m\}$

$\forall i. \forall j. \text{bounded } m \ r_0 \ i \longrightarrow \text{bounded } m \ r_0 \ j \longrightarrow i < j \longrightarrow arr \ j \leq arr \ i$

$\forall x \in \#arr\text{-mset } arr \ l_0 \ m. \forall y \in \#arr\text{-mset } arr \ m \ r_0. \neg x < y$

$\text{bounded } (m - 1) \ r_0 \ i$

$\text{bounded } (m - 1) \ r_0 \ j$

$i < j$

$\text{mod-arr} = (arr'(m - 1 := \text{Min-mset } (arr\text{-mset } arr \ l_0 \ m)))$

shows $\text{mod-arr } j \leq \text{mod-arr } i$

proof –

have *second-half-eq*: $\text{mod-arr} = arr \text{ on } \{m..<r_0\}$

using *assms(3, 9)* **unfolding** *eq-on-def* **by** *simp*

have *j-stricter-bound*: $\text{bounded } m \ r_0 \ j$

using *assms(6-8)* **by** *simp*

then have *el-at-j*: $\text{mod-arr } j \in \# \text{arr-mset } arr \ m \ r_0$

using *eq-onD second-half-eq* **by** *fastforce*

then show *?thesis*

proof(*cases*)

assume $i = (m - 1)$

then have *mod-arr i* $\in \# \text{arr-mset } arr \ l_0 \ m$

by (*simp add: assms(1, 9)*)

then show *?thesis*

using *assms(5) el-at-j not-less* **by** *blast*

next

assume $i \neq (m - 1)$

then have *bounded m r0 i*

using *assms(6)* **by** *simp*

then show *?thesis*

using *assms(4, 8) eq-on-def j-stricter-bound second-half-eq* **by** *force*

qed

qed

The following lemma shows that all elements in the first part of the array (the binary heap) are bigger than the elements in the second part (the sorted part) after every iteration. This lemma and the invariant of the *heap-sort* loop use $\neg x < y$ instead of $x \geq y$ since *vcg-cs* doesn't terminate in the latter case.

lemma *heap-sort-fst-part-bigger-snd-part-step*:

fixes $arr::int \Rightarrow int$

assumes

$l_0 < m$

```

    m ≤ r₀
    arr-mset arr' l₀ (m - 1) = arr-mset arr l₀ m - {#Min-mset (arr-mset
arr l₀ m)#}
    arr = arr' on - {l₀..<m}
    ∀ x ∈ #arr-mset arr l₀ m. ∀ y ∈ #arr-mset arr m r₀. ¬ x < y
    mod-arr = arr'(m - 1 := Min-mset (arr-mset arr l₀ m))
    x ∈ #arr-mset mod-arr l₀ (m - 1)
    y ∈ #arr-mset mod-arr (m - 1) r₀
shows ¬ x < y
proof -
have {m..<r₀} ⊆ - {l₀..<m}
  by auto
hence arr' = arr on {m..<r₀}
  using assms(4) eq-on-sym eq-on-subset by blast
hence arr-eq-on: mod-arr = arr on {m..<r₀}
  by (simp add: assms(6))
hence same-mset: arr-mset mod-arr m r₀ = arr-mset arr m r₀
  using mset-ran-cong by blast
have x ∈ # arr-mset arr l₀ m using same-mset assms
  by (metis assms(3,6,7) add-mset-remove-trivial-eq lran-upd-outside(2)
mset-lran cancel-ab-semigroup-add-class.diff-right-commute
diff-single-trivial multi-self-add-other-not-self order-refl)
then have x-bigger-min: x ≥ Min-mset (arr-mset arr l₀ m)
  using Min-le by blast
have y-smaller-min: y ≤ Min-mset (arr-mset arr l₀ m)
proof(cases y = mod-arr (m - 1))
  case False
    hence y ∈ #arr-mset mod-arr (m - 1) r₀ - {#mod-arr (m - 1)#}
    by (metis assms(8) diff-single-trivial insert-DiffM insert-noteq-member)
    then have y ∈ #arr-mset arr m r₀
    by (simp add: assms(2) intvs-decr-l mset-ran-insert same-mset)
    then show ?thesis
    using assms(1) assms(5) by fastforce
  qed (simp add: assms(6))
then show ?thesis
  using x-bigger-min by linarith
qed

```

4.3.2 Implementation

Now finally the correctness of the *heap-sort* is shown. As mentioned, it starts by transforming the array into a minimum heap using *make-heap*. Then in each iteration it removes the first element from the heap with *del-min* after its value was retrieved with *get-min*. This value is then put at the position

freed by *del-min*.

program-spec *heap-sort*

assumes $l \leq r$

ensures $array\text{-is-sorted} (\geq) arr\ l_0\ r_0 \wedge$
 $arr\text{-mset}\ arr_0\ l_0\ r_0 = arr\text{-mset}\ arr\ l_0\ r_0 \wedge$
 $arr_0 = arr\ on - \{l_0 ..<r_0\} \wedge l = l_0 \wedge r = r_0$

for $l\ r\ arr[]$

defines \langle

$arr = make\text{-heap}(arr, l, r);$

$m = r;$

$while\ (m > l)$

$\ @variant\ \langle m - l + 1 \rangle$

$\ @invariant\ \langle is\text{-heap} (\leq) arr\ l\ m \wedge$

$array\text{-is-sorted} (\geq) arr\ m\ r_0 \wedge$

$(\forall x \in \# arr\text{-mset}\ arr\ l_0\ m. \forall y \in \# arr\text{-mset}\ arr\ m\ r_0. \neg x < y) \wedge$

$arr\text{-mset}\ arr_0\ l_0\ r_0 = arr\text{-mset}\ arr\ l_0\ r_0 \wedge$

$l \leq m \wedge m \leq r_0 \wedge l = l_0 \wedge arr_0 = arr\ on - \{l_0 ..<r_0\} \rangle$

$\{$

$min = get\text{-min}(arr, l, m);$

$(arr, l, m) = del\text{-min}(arr, l, m);$

$arr[m] = min$

$\}$

\rangle

apply *vcg-cs*

subgoal unfolding *array-is-sorted-def* **by** *simp*

subgoal

apply(*intro conjI*)

subgoal unfolding *is-partial-heap-def* **by** *simp*

subgoal unfolding *array-is-sorted-def* **using** *heap-sort-second-half-sorted-step*

by *blast*

subgoal using *heap-sort-fst-part-bigger-snd-part-step* **by** *blast*

subgoal using *heap-sort-mset-step* **by** *blast*

subgoal unfolding *eq-on-def*

by (*metis ComplD ComplI atLeastLessThan-iff less-le-trans*)

done

done

end

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. 2009.

- [2] P. Lammich and S. Wimmer. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs*, Jan. 2019. <http://isa-afp.org/entries/IMP2.html>, Formal proof development.
- [3] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. 2007.
- [4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, page 184. 2018. <http://isabelle.in.tum.de/dist/Isabelle2018/doc/tutorial.pdf>.