IMP2 Binary Heap

Simon Griebel

June 13, 2019

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 mset-ran ......................... 10
   3.2 Lemmas on swap and eq-on .................. 11

4 Imperative Heap Implementation 13
   4.1 Simple Functions ......................... 13
      4.1.1 Parent, Children and Swap ............... 13
      4.1.2 get-min ............................... 13
   4.2 Modifying Functions ....................... 14
4.2.1 sift-up and insert ................................. 14
4.2.2 sift-down, del-min and make-heap ............... 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 \textit{int} \Rightarrow \textit{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 \( l = r \).

\textbf{abbreviation} \texttt{bounded} :: \textit{int} \Rightarrow \textit{int} \Rightarrow \textit{int} \Rightarrow \textit{bool} \ where \\
\text{bounded} \ l \ r \ x \equiv l \leq x \land 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.

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

\textbf{definition} \texttt{l-child} :: \textit{int} \Rightarrow \textit{int} \Rightarrow \textit{int} \ where \\
l-child \ l \ p = 2 \ast p - l + 1
definition r-child :: int ⇒ int ⇒ int where
r-child l p = 2 * p − l + 2

2.2.2 Lemmas

lemma parent-upper-bound: parent l c < c ←→ l ≤ c
unfolding parent-def by auto

lemma parent-upper-bound-alt: l ≤ parent l c ⇒ parent l c < c
unfolding parent-def by simp

lemma parent-lower-bound: l ≤ parent l c ←→ l < c
unfolding parent-def by linarith

lemma grand-parent-upper-bound: parent l (parent l c) < c ←→ l ≤ c
unfolding parent-def by linarith

corollary parent-bounds: l < x ⇒ x < r ⇒ 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 ←→ l ≤ p
unfolding l-child-def by simp

corollary l-child-lower-bound-alt: l ≤ x ⇒ x ≤ p ⇒ x < l-child l p
using l-child-lower-bound[of l p] 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 ≤ p ⇒ p < r-child l p
unfolding r-child-def by simp

corollary r-child-lower-bound-alt: l ≤ x ⇒ x ≤ p ⇒ 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:
\[(c = \text{l-child } l \ p \lor c = \text{r-child } l \ p) \iff \text{parent } l \ c = p\]

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
\[
\text{is-partial-heap} \ cmp \ \text{heap} \ l \ m \ r \ i = \\
\forall x. \text{bounded } m \ r \ x \implies \\
\text{bounded } m \ r \ (\text{parent } l \ x) \implies \text{cmp } (\text{heap } (\text{parent } l \ x)) (\text{heap } x)
\]

Abstraction is-heap
\[
is-heap \ cmp \ \text{heap} \ l \ r \equiv \text{is-partial-heap} \ cmp \ \text{heap} \ l \ l \ r
\]

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
\[
is-partial-heap-except-down \ cmp \ \text{heap} \ l \ m \ r \ i = \\
((\text{parent } l \ x \neq i \implies \text{bounded } m \ r \ x) \implies \\
\text{cmp } (\text{heap } (\text{parent } l \ x)) (\text{heap } x)) \land \\
\text{cmp } (\text{heap } (\text{parent } l \ (\text{parent } l \ x))) (\text{heap } x))
\]
abbreviation is-heap-except-down
:: ('a::order ⇒ 'a::order ⇒ bool) ⇒ (int ⇒ 'a::order) ⇒ int ⇒ int ⇒ int
⇒ bool
where
is-heap-except-down cmp heap l r i ≡ 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 ⇒ 'a::order ⇒ bool) ⇒ (int ⇒ 'a::order) ⇒ int ⇒ int ⇒ int
⇒ bool
where
is-heap-except-up cmp heap l r i = (∀ x. bounded l r x →
((x ≠ i → bounded l r (parent l x) → cmp (heap (parent l x)) (heap x)) ∧
  (parent l x = i → bounded l r (parent l (parent l x))) →
   cmp (heap (parent l (parent l x))) (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 ⇒ r′ ≤ r ⇒ 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 ⇒ m ≤ m′ ⇒ is-partial-heap cmp heap
  l m′ r
  unfolding is-partial-heap-def by simp

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

lemma snd-half-is-partial-heap:
  (l + r) div 2 ≤ m ⇒ 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**

\[ l \leq m \leq r \]

\( \text{is-partial-heap} \ cmp \ heap \ l \ (m + 1) \ r \)

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

**proof**

fix \( x \)

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

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

show bounded m r x \[\Rightarrow\]

\((?p-x \neq m \Rightarrow \text{bounded} \ m \ r \ ?p-x \Rightarrow \text{cmp} (\text{heap} \ ?p-x) (\text{heap} \ x)) \land \]

\((?p-x = m \Rightarrow \text{bounded} \ m \ r \ ?gp-x \Rightarrow \text{cmp} (\text{heap} \ ?gp-x) (\text{heap} \ x))\)

**proof**

assume x-bound: bounded m r x

have p-x-lower: \(?p-x \neq m \Rightarrow \text{bounded} \ m \ r \ ?p-x \Rightarrow \ ?p-x \geq m + 1\)

by simp

hence \(?p-x \neq m \Rightarrow \text{bounded} \ m \ r \ ?p-x \Rightarrow \ ?p-x \geq m + 1\)

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

hence p-invariant: \(?p-x \neq m \Rightarrow \text{bounded} \ m \ r \ ?p-x \Rightarrow \text{cmp} (\text{heap} \ ?p-x) (\text{heap} \ x)\)

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

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

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

show \(?p-x \neq m \Rightarrow \text{bounded} \ m \ r \ ?p-x \Rightarrow \text{cmp} (\text{heap} \ ?p-x) (\text{heap} \ x)\) \land \?

\(?p-x = m \Rightarrow \text{bounded} \ m \ r \ ?gp-x \Rightarrow \text{cmp} (\text{heap} \ ?gp-x) (\text{heap} \ x)\)

using gp-up-bound p-invariant by linarith

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) \( \forall i. \forall j. k < j \Rightarrow P j \Rightarrow P i \) \( \Rightarrow P i \)
  shows \( \forall j. k < j \Rightarrow j < i \Rightarrow P j \)
  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.prems(1,3) by blast
  next
    case False
    hence j < i using step.prems(2) by simp
    then show ?thesis using step.IH step.prems(1,3) by blast
next
theorem strong-int-gr-induct:
assumes \( k < (i::\text{int}) \)
\((\forall i. k < i \Rightarrow (\forall j. k < j \Rightarrow j < i \Rightarrow P j) \Rightarrow P i)\)
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
\( \text{is-heap \ cmp \ heap\ l\ r} \)
\( \text{transp \ cmp} \)
\( l < x \ x < r \)
shows \( \text{cmp \ (heap \ l) \ (heap \ x)} \)
using assms unfolding is-partial-heap-def
proof(induction \( x \) rule: strong-int-gr-induct[of \( l \)])
case 1
then show \( ?\text{case} \) using assms(3) by simp
next
case \( (2\ i) \)
have \( \text{cmp-pi-i: \ cmp \ (heap \ (parent \ l \ i)) \ (heap \ i)} \)
using 2.hyps 2.prems(1,4) parent-bounds by simp
then show \( ?\text{case} \)
proof(cases)
assume parent \( l \ i > l \)
then have \( \text{cmp \ (heap \ l) \ (heap \ (parent \ l \ i))} \)
using 2.IH 2.prems(1,2,4) parent-upper-bound-alt by simp
then show \( ?\text{thesis} \)
using 2.prems(2) cmp-pi-i transpE by metis
next
assume \( \neg \text{parent \ l \ i} = l \)
then have parent \( l \ i = l \)
using 2.hyps dual-order.order-iff-strict parent-lower-bound by metis
then show \( ?\text{thesis} \)
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-mset :: (int ⇒ 'a ⇒ int) ⇒ int ⇒ int ⇒ 'a multiset where
arr-mset arr l r ≡ mset-ran arr {l..<r}

lemma in-mset-imp-in-array:
  x ∈# arr-mset arr l r ←→ (∃ i. bounded l r i ∧ arr i = x)
  unfolding mset-ran-def by fastforce

lemma arr-mset-remove-last:
  l ≤ r =⇒ arr-mset arr l r = arr-mset arr l (r + 1) − {#arr r#}
  by (simp add: intvs-upper-decr mset-ran-def)

lemma arr-mset-append:
  l ≤ r =⇒ arr-mset arr l (r + 1) = arr-mset arr l r + {#arr r#}
  using arr-mset-remove-last[of l r arr] by simp

corollary arr-mset-append-alt:
  l ≤ r =⇒ arr-mset (arr(r := b)) l (r + 1) = arr-mset arr l r + {#b#}
  by (simp add: arr-mset-append mset-ran-subst-outside)

lemma arr-mset-remove-first:
  i ≤ r =⇒ arr-mset arr (i − 1) r = arr-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 ≤ m m ≤ r
  shows arr-mset arr l r = arr-mset arr l m + arr-mset arr m r
  using assms
proof (induction m rule: int-ge-induct[of l])
  case (step i)
  have add-last: arr-mset arr l (i + 1) = arr-mset arr l i + {#arr i#}
    using step arr-mset-append by blast
  have rem-first: arr-mset arr (i+1) r = arr-mset arr i r − {#arr i#}
    by (metis step.prems arr-mset-remove-first add-diff-cancel-right')
  show ?case
    using step add-last rem-first by fastforce
qed (simp)

10
That the first element in a heap is the “smallest”, can now be expressed using multisets.

**corollary** heap-first-el-alt:

**assumes**

\[
\begin{align*}
& \text{transp } \text{cmp} \\
& \text{is-heap } \text{cmp } \text{heap } l \ r \\
& x \in \# (\text{arr-mset } \text{heap } l \ r) \\
& \text{heap } l \neq x
\end{align*}
\]

**shows** \( \text{cmp } (\text{heap } l) \ x \)

by (metis asms 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:

\[
\text{arr'} = \text{swap arr } x \ y \implies \text{arr'} y = \text{arr } x \land \text{arr'} x = \text{arr } y
\]

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: swap arr x y = 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**

\[
\begin{align*}
& \text{arr1 } = \text{arr2 on } - \{l..<r\} \\
& l < c \ c < r
\end{align*}
\]

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

using parent-bounds swap-eq-on asms by fastforce

**corollary** swap-child-eq-on:

**assumes**

\[
\begin{align*}
& \text{arr1 } = \text{arr2 on } - \{l..<r\} \\
& c = l\text{-child } l \ p \lor c = r\text{-child } l \ p \\
& l \leq p \ c < r
\end{align*}
\]
shows \( arr_1 = \text{swap} \ arr_2 \ p \ c \ on \ − \{l..<r\} \)
by \((\text{metis asms parent-lower-bound parent-two-children swap-parent-eq-on})\)

**Lemma swap-child-mset:**

**Assumes**
- \( \text{arr-mset} \ arr_1 \ l \ r = \text{arr-mset} \ arr_2 \ l \ r \)
- \( c = \text{l-child} \ l \ p \lor c = \text{r-child} \ l \ p \)
- \( l \leq p \ c < r \)

**Shows** \( \text{arr-mset} \ arr_1 \ l \ r = \text{arr-mset} (\text{swap} \ arr_2 \ p \ c) \ l \ r \)

**Proof** –
- have child-bounded: \( l < c \land c < r \)
  by \((\text{metis asms } (2−4) \text{ parent-lower-bound parent-two-children})\)
- have parent-bounded: \( \text{bounded} \ l \ r \ p \)
  by \((\text{metis asms } (2−4) \text{ dual-order.strict-trans parent-two-children parent-upper-bound-alt})\)
- thus \(?\text{thesis}\)
  using asms \( (1) \text{ child-bounded mset-ran-swap[of } p \ \{l..<r\} \ c \ arr_2 \} \text{ atLeastLessThan-iff} \)
  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** \( \text{arr} : 'a::order \Rightarrow 'a::order \)

**Assumes**
- \( m < n \land x < y \)
- \( m < n \land x < y \Rightarrow x = m \Rightarrow y = n \Rightarrow P (arr \ n) (arr \ m) \)
- \( m < n \land x < y \Rightarrow x \neq m \Rightarrow y \neq n \Rightarrow y \neq m \Rightarrow y \neq n \Rightarrow P (arr \ m) (arr \ n) \)
- \( m < n \land x < y \Rightarrow x = n \Rightarrow y \neq m \Rightarrow y \neq n \Rightarrow P (arr \ m) (arr \ y) \)
- \( m < n \land x < y \Rightarrow x \neq m \Rightarrow x \neq n \Rightarrow y = n \Rightarrow P (arr \ m) (arr \ x) \)
- \( m < n \land x < y \Rightarrow x \neq m \Rightarrow x \neq n \Rightarrow y = m \Rightarrow P (arr \ x) (arr \ n) \)

**Shows** \( P (\text{swap} \ arr \ x \ y \ m) (\text{swap} \ arr \ x \ y \ n) \)
by \((\text{metis asms order.asym swap-only-swaps swap-swaps})\)
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 ≤ 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.

\textbf{procedure-spec} \textit{prnt} \((l, x)\) \textbf{returns} \(p\)
\textbf{assumes} True
\textbf{ensures} \(p = \text{parent} ((l - 1) / 2 + l)\)
\textbf{by} \(\text{vcg simp add: parent-def}\)

\textbf{procedure-spec} \textit{left-child} \((l, x)\) \textbf{returns} \(lc\)
\textbf{assumes} True
\textbf{ensures} \(lc = l-child (2 * x - l + 1)\)
\textbf{by} \(\text{vcg simp add: l-child-def}\)

\textbf{procedure-spec} \textit{right-child} \((l, x)\) \textbf{returns} \(rc\)
\textbf{assumes} True
\textbf{ensures} \(rc = r-child (2 * x - l + 2)\)
\textbf{by} \(\text{vcg simp add: r-child-def}\)

\textbf{procedure-spec} \textit{swp} \((\text{heap}, x, y)\) \textbf{returns} \(\text{heap}\)
\textbf{assumes} True
\textbf{ensures} \(\text{heap = swap} \text{heap0} x y\)
\textbf{by} \(\text{vcg simp add: swap-def}\)

4.1.2 get-min

In this section \textit{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 \(l = \text{Min-mset} \ (\text{arr-mset heap} \ l \ r)\)

**Proof**

- Have \((\forall x \in \# (\text{arr-mset heap} \ l \ r). (\text{heap} \ l) \leq x)\)
- Using assms(2) heap-first-el-alt transp-le by blast
- Thus \(\text{thesis}\)
- By \((\text{simp add: assms(1) dual-order.antisym})\)

**Qed**

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

**Assumes** \( l < r \land \text{is-heap} \ (\leq) \ \text{heap} \ l \ r\)

**Ensures** min = Min-mset \((\text{arr-mset heap} \ l_0 \ l_0 \ r_0)\)

**For** heap[] \(l r\)

**Defines** \((\text{min} = \text{heap}\[l\])\)

**By** vcg \((\text{simp add: heap-minimum})\)

### 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-axxx1:**

**Fixes** heap::int => int

**Assumes**

- \(\text{is-heap-except-up} \ (\leq) \ \text{heap} \ l \ r \ x\)
- 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 (intro cmp-swapI[of \((\text{parent } l \; k) \; \text{ k } \; (\text{parent } l \; x) \; \text{ x } (\leq) \; \text{ heap})\])
subgoal using assms(2,6) parent-upper-bound-alt by blast
subgoal using assms(3) by blast
subgoal using assms(1,4,6) unfolding is-heap-except-up-def by blast
subgoal using assms(1,3,4,6) unfolding is-heap-except-up-def by fastforce
subgoal by blast
subgoal using assms(1,2,4) unfolding is-heap-except-up-def by simp

done

lemma sift-up-step-aux2:
fixes heap :: int \Rightarrow int
assumes
  is-heap-except-up \((\leq)\) heap l r x
  parent l x \geq l
  heap x \leq (heap (parent l x))
  bounded l r k
  parent l k = parent l x
  bounded l r (parent l (parent l k))
shows
  \((\text{swap heap } (\text{parent } l \; x) \; x \; \text{ k}) \leq (\text{swap heap } (\text{parent } l \; x) \; x \; k)\) 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-eq-gp-x: \((\text{swap heap } (\text{parent } l \; x) \; x \; \text{ ?gp-k = heap } \; \text{ ?gp-x})\)
    by (metis assms(2,5) grand-parent-upper-bound less-irrefl swap-only-swaps)
  show \?thesis
    using assms unfolding is-heap-except-up-def
    by (metis gp-k-eq-gp-x k-eq-x parent-bounds parent-lower-bound)
next
  assume k-neq-x: \text{ k } \neq x
  have \((\text{swap heap } (\text{parent } l \; x) \; x \; \text{ k = heap } \; \text{ k})\)
    by (metis assms(5) gp-k-eq-gp-x k-neq-x 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 ⇒ int
  assumes
   is-heap-except-up (≤) heap l r x
   parent l x ≥ l
   (heap x) ≤ (heap (parent l x))
  shows is-heap-except-up (≤) (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 (≤) heap l r x ∧ bounded l r x
  ensures is-heap (≤) heap l 0 r 0 ∧
   arr-mset heap0 l 0 r 0 = arr-mset heap l 0 r 0 ∧
   heap0 = heap on − {l..<r0}
for heap[ ] l x r
defines (p = prnt(l, x);
  while (x > l ∧ heap[x] ≤ heap[p])
    @variant (x − l)
    @invariant (is-heap-except-up (≤) heap l r x ∧ p = parent l x ∧
      bounded l r x ∧ arr-mset heap0 l 0 r 0 = arr-mset heap l r ∧
      heap0 = heap on − {l..<r})
    {
      heap = swap(heap, p, x);
      x = p;
      p = prnt(l, x)
    })
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 (≤) heap l r ∧ l ≤ r  
**ensures** is-heap (≤) heap l r ∧  
arr-mset heap l r = arr-mset heap l0 r0 + {#el#} ∧  
l = l0 ∧ r = r0 + 1 ∧ heap0 = heap on − {l..<r}

for heap l r el  
defines (  
heap[r] = el;  
x = r;  
r = r + 1;  
heap = sift-up(heap, l, r, x)
)

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 (≤) heap l x r x ∧ l ≤ x ∧ x ≤ r  
**ensures** is-partial-heap (≤) heap l0 x0 r0 ∧  
arr-mset heap0 l0 r0 = arr-mset heap l0 r0 ∧  
heap0 = heap on − {l0..<r0}

defines (  
lc = left-child(l, x);  
rc = right-child(l, x);  
while (lc < r ∧ (heap[lc] < heap[x] ∨ rc < r ∧ heap[rc] < heap[x])))  
@variant (r - x)  
@invariant (is-partial-heap-except-down (≤) heap l x0 r x ∧
\[ x_0 \leq x \land x \leq r \land lc = l\text{-child } l \land rc = r\text{-child } l \land \]

\[ \text{arr-mset } \text{heap}_0 \ l \ r = \text{arr-mset } \text{heap } l \ r \land \]

\[ \text{heap}_0 = \text{heap on } - \{l..<r\}; \]

\[
\{
\begin{align*}
\text{smallest} &= lc; \\
\text{if } (rc < r \land \text{heap}[rc] < \text{heap}[lc]) \{ \\
\text{smallest} &= rc \\
\};
\text{heap} &= \text{swp}(\text{heap}, x, \text{smallest}); \\
\text{x} &= \text{smallest}; \\
\text{lc} &= \text{left-child}(l, x); \\
\text{rc} &= \text{right-child}(l, x)
\end{align*}
\}
\]

apply vecg-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
\[ l < r \]
\[ \text{is-heap } (\leq) \text{ heap } l r \]
\[ \text{mod-heap } = \text{heap}(l := \text{heap } (r - 1)) \]
\[ \text{arr-mset } \text{mod-heap } l (r - 1) = \text{arr-mset } \text{new-heap } l (r - 1) \]

shows
\[ \text{arr-mset } \text{new-heap } l (r - 1) = \text{arr-mset } \text{heap } l r - \{ \# \text{Min-mset } (\text{arr-mset } \text{heap } l r) \# \} \]

proof —
let \( ?\text{heap-mset} = \text{arr-mset } \text{heap } l r \)
have \( l\text{-is-min}: \text{heap } l = \text{Min-mset } ?\text{heap-mset} \)
using \( \text{assms}(1,2) \) \( \text{heap-minimum}\) by blast
have \( (\text{arr-mset } \text{mod-heap } l r) = ?\text{heap-mset} + \{ \# \text{heap } (r - 1) \# \} - \{ \# \text{heap } l \# \} \)
by \( (\text{simp add: assms}(1,3) \) \( \text{mset-ran-subst-inside}) \)
hence \( (\text{arr-mset } \text{mod-heap } l (r - 1)) = ?\text{heap-mset} - \{ \# \text{heap } l \# \} \)
by \( (\text{simp add: assms}(1,3) \) \( \text{arr-mset-removal-last}) \)
thus ?thesis
using \( \text{assms}(4) \) \( l\text{-is-min}\) 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 \( \text{sift-down}\).

procedure-spec del-min \((\text{heap}, l, r)\) returns \((\text{heap}, l, r)\)
assumes \( l < r \land \text{is-heap } (\leq) \text{ heap } l r \)
ensures \( \text{is-heap } (\leq) \text{ heap } l r \land \)
\( \text{arr-mset } \text{heap } l r = \text{arr-mset } \text{heap } l_0 r_0 - \{ \# \text{Min-mset } \text{arr-mset } \text{heap } l_0 r_0 \# \} \land \)
\( l = l_0 \land r = r_0 - 1 \land \)
\( \text{heap}_0 = \text{heap on } - \{ l_0..<r_0 \} \)
for \( \text{heap } l r \)
defines (\)
\( r = r - 1; \)
\( \text{heap}[l] = \text{heap}[r]; \)
\( \text{heap} = \text{sift-down}(\text{heap}, l, r, l) \)
)
apply vcg-cs
subgoal by \( (\text{simp add: heap-changed-first-el is-partial-heap-smaller-back}) \)
subgoal
apply \( \text{rule conjI} \)
subgoal using del-min-mset by blast
subgoal by \( (\text{simp add: eq-on-def intvs-incdec}(3) \) \( \text{intvs-lower-incr}) \)
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 ≤ r
ensures is-heap (≤) heap l0 r0 ∧
arr-mset heap l0 r0 = arr-mset heap0 l0 r0 ∧
heap0 = heap on − {l0..< r0}
for heap[] l r
defines (y = (r + l)/2 − 1;
while (y ≥ l)
  @variant (y − l + 1)
  @invariant (is-partial-heap (≤) heap l (y + 1) r ∧
  arr-mset heap l r = arr-mset heap0 l0 r0 ∧
  l − 1 ≤ y ∧ y < r ∧ heap0 = heap on − {l..<r})
  {heap = sift-down(heap, l, r, y);
y = y − 1}
); apply(vcgs)
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 ≤-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 run-sorted generalized for arbitrary comparison functions.

**Definition**

\[
\text{array-is-sorted} :: (\text{int} \Rightarrow \text{int} \Rightarrow \text{bool}) \Rightarrow (\text{int} \Rightarrow \text{int}) \Rightarrow \text{int} \Rightarrow \text{bool}
\]

\[
\text{array-is-sorted} \ cmp \ a \ l \ r \equiv \forall \ i. \ \forall \ j. \ \text{bounded} \ l \ r \ i \rightarrow \text{bounded} \ l \ r \ j \rightarrow i < j \rightarrow \text{cmp} \ (a \ i) \ (a \ j)
\]

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**

\[
\text{arr} :: \text{int} \Rightarrow \text{int}
\]

**Assumes**

\[
l < m \ m \leq r
\]

\[
\text{arr-mset} \ \text{arr} \ l \ (m - 1) = \text{arr-mset} \ \text{arr} \ l \ m \ - \ \#\text{Min-mset} \ (\text{arr-mset} \ \text{arr} \ l \ m)\#
\]

\[
\text{arr} = \text{arr}' \ \text{on} \ - \ {l..<m}
\]

\[
\text{mod-arr} = \text{arr}'(m - 1 := \text{Min-mset} \ (\text{arr-mset} \ \text{arr} \ l \ m))
\]

**Shows**

\[
\text{arr-mset} \ \text{arr} \ l \ r = \text{arr-mset} \ \text{mod-arr} \ l \ r
\]

**Proof**

- Let \(?\min = \#\text{Min-mset} \ (\text{arr-mset} \ \text{arr} \ l \ m)\#\)
- Let \(?\new-arr-mset = \text{arr-mset} \ \text{mod-arr}\)
- Have middle: \(?\new-arr-mset \ (m - 1) \ m = \?\min\)
  - By (simp add: assms(5))
- Have first-half: \(?\new-arr-mset \ l \ (m - 1) = \text{arr-mset} \ \text{arr} \ l \ m \ - \ ?\min\)
  - 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: \(?\new-arr-mset \ l \ m = \text{arr-mset} \ \text{arr} \ l \ m\)
  - Using middle first-half assms(1) by simp
- Hence \mod-arr = \text{arr} \ \text{on} \ - \ {l..<m}\)
  - Using assms(1,4,5) eq-on-sym eq-on-trans by auto
- Then have second-half: \text{arr-mset} \ \text{arr} \ m \ r = \text{arr-mset} \ \text{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 ⇒ int
assumes
  \( l_0 < m \leq r_0 \)
  \( \text{arr} = \text{arr}' \) on \( \{ l_0..<m \} \)
  \( \forall i. \forall j. \text{bounded m } r_0 \ i \rightarrow \text{bounded m } r_0 \ j \rightarrow i < j \rightarrow \text{arr } j \leq \text{arr } i \)
  \( \forall x \in \# \text{arr-mset arr } l_0 \ m. \forall y \in \# \text{arr-mset arr } m \ r_0. \neg x < y \)
  \( \text{bounded } (m - 1) \ r_0 \ i \)
  \( \text{bounded } (m - 1) \ r_0 \ j \)
  \( i < j \)
  \( \text{mod-arr} = (\text{arr}'(m - 1 := \text{Min-mset } (\text{arr-mset arr } l_0 \ m))) \)
shows \( \text{mod-arr } j \leq \text{mod-arr } i \)
proof
  have second-half-eq: \( \text{mod-arr} = \text{arr} \) 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 \( m \ r_0 \ i \)
  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 ⇒ int
assumes
\[ l_0 < m \]
\[ m \leq r_0 \]
\[ \text{arr-mset } \text{arr}' \ l_0 \ (m - 1) = \text{arr-mset } \text{arr} \ l_0 \ m - \{ \#\text{Min-mset } (\text{arr-mset } \text{arr} \ l_0) \# \} \]
\[ \text{arr} = \text{arr}' \ on - \{ l_0..<m \} \]
\[ \forall x \in \#\text{arr-mset } \text{arr} \ l_0 \ m. \ \forall y \in \#\text{arr-mset } \text{arr} \ m \ r_0. \ \neg \ x < y \]
\[ \text{mod-arr} = \text{arr}'(m - 1 := \text{Min-mset } (\text{arr-mset } \text{arr} \ l_0 \ m)) \]
\[ x \in \#\text{arr-mset } \text{mod-arr} \ l_0 \ (m - 1) \]
\[ y \in \#\text{arr-mset } \text{mod-arr} \ (m - 1) \ r_0 \]
\[ \text{shows} \ \neg \ x < y \]

**proof**

- **have** \{m..<r_0\} \(\subseteq\) \(-\{l_0..<m\}\)
  - by auto
- **hence** \text{arr}' = \text{arr} on \{m..<r_0\}
  - using assms(4) eq-on-sym eq-on-subset by blast
- **hence** \text{arr}-eq-on: \text{mod-arr} = \text{arr} on \{m..<r_0\}
  - by (simp add: assms(6))
- **hence** same-mset: \text{arr-mset } \text{mod-arr} \ m \ r_0 = \text{arr-mset } \text{arr} \ m \ r_0
  - using mset-ran-cong by blast
- **have** \(x \in \#\text{arr-mset } \text{arr} \ l_0 \ m\)
  - by (metis assms(3,6,7) add-mset-remove-trivial-eq ran-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 \text{x-bigger-min}: \(x \geq \text{Min-mset } (\text{arr-mset } \text{arr} \ l_0 \ m)\)
    - using Min-le by blast
  - **have** y-smaller-min: \(y \leq \text{Min-mset } (\text{arr-mset } \text{arr} \ l_0 \ m)\)
    - **proof** (cases \(y = \text{mod-arr} (m - 1)\))
      - **case** False
      - **hence** \(y \in \#\text{arr-mset } \text{mod-arr} (m - 1) \ r_0 - \{ \#\text{mod-arr} (m - 1)\# \}\)
        - by (metis assms(8) diff-single-trivial insert-DiffM insert-noteq-member)
      - **then have** \(y \in \#\text{arr-mset } \text{arr} \ m \ r_0\)
        - by (simp add: assms(2) intvs-decr-l mset-ran-insert same-mset)
      - **then show** ?thesis
        - using assms(1) assms(5) by fastforce
  - **then show** ?thesis
    - using 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 \textit{get-min}. This value is then put at the position freed by \textit{del-min}.

\textbf{program-spec heap-sort}
\begin{itemize}
  \item \textbf{assumes} $l \leq r$
  \item \textbf{ensures} $\text{array-is-sorted}(\geq) \, arr \, l_0 \, r_0 \wedge$
    \begin{align*}
      & 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
    \end{align*}
\end{itemize}
\begin{itemize}
  \item \textbf{for} $l \, r \, arr[]$
  \item \textbf{defines}:
    \begin{itemize}
      \item $arr = \text{make-heap}(arr, \, l, \, r)$;
      \item $m = r$;
      \item \textbf{while} ($m > l$)
        \begin{itemize}
          \item \textbf{variant} ($m - l + 1$)
          \item \textbf{invariant} ($\text{is-heap} \, \leq \, arr \, m$)
            \begin{itemize}
              \item $\text{array-is-sorted}(\geq) \, arr \, m \, r_0 \wedge$
              \item $(\forall \, x \, \in \# \, arr\text{-mset} \, arr_0 \, m \, r_0. \, \forall \, y \, \in \# \, arr\text{-mset} \, arr \, m \, r_0. \, \neg \, x < y) \wedge$
              \item $arr\text{-mset} \, arr_0 \, l_0 \, r_0 = arr\text{-mset} \, arr \, l_0 \, r_0 \wedge$
              \item $l \leq m \wedge m \leq r_0 \wedge l = l_0 \wedge arr_0 = arr \, on - \{ l_0 \, .. < r_0 \}$
            \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item \textbf{apply} \emph{vcg-cs}
  \item \textbf{subgoal unfolding} \emph{array-is-sorted-def} \textbf{by} \emph{simp}
  \item \textbf{subgoal}
    \begin{itemize}
      \item \textbf{apply}(\emph{intro conjI})
      \item \textbf{subgoal unfolding} \emph{is-partial-heap-def} \textbf{by} \emph{simp}
      \item \textbf{subgoal unfolding} \emph{array-is-sorted-def} \textbf{using} \emph{heap-sort-second-half-sorted-step}
        \textbf{by} \emph{blast}
      \item \textbf{subgoal using} \emph{heap-sort-fst-part-bigger-snd-part-step} \textbf{by} \emph{blast}
      \item \textbf{subgoal using} \emph{heap-sort-mset-step} \textbf{by} \emph{blast}
      \item \textbf{subgoal unfolding} \emph{eq-on-def}
        \textbf{by} \textbf{(metis \emph{ComplD ComplI atLeastLessThan-iff less-le-trans})}
    \end{itemize}
  \item \textbf{done}
  \item \textbf{done}
\end{itemize}
\textbf{end}

\textbf{References}

