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 ............................. 7

3 General Lemmas on Arrays 8
  3.1 Lemmas on \textit{mset-ran} .......................... 8
  3.2 Lemmas on \textit{swap} and \textit{eq-on} ................. 9

4 Imperative Heap Implementation 10
  4.1 Simple Functions .................................... 11
    4.1.1 Parent, Children and Swap .................... 11
    4.1.2 get-min .......................................... 11
  4.2 Modifying Functions ................................. 12
4.2.1  sift-up and insert ........................................ 12
4.2.2  sift-down, del-min and make-heap .................... 14
4.3  Heapsort Implementation ................................. 16
  4.3.1  Auxiliary Lemmas ...................................... 16
  4.3.2  Implementation ....................................... 18
theory IMP2-Binary-Heap
imports 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 \texttt{int ⇒ 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 \( l = r \).

\begin{verbatim}
abbreviation bounded :: \texttt{int ⇒ int ⇒ int ⇒ bool} where
    bounded \( l \ r \ x \equiv l \leq x \land x < r \)
\end{verbatim}

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.

\begin{verbatim}
definition parent :: \texttt{int ⇒ int ⇒ int where}
    parent \( l \ c \equiv l + (c - l - 1) \ div 2 \)

definition l-child :: \texttt{int ⇒ int ⇒ int where}
    l-child \( l \ p \equiv 2 \ast p - l + 1 \)
\end{verbatim}
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
⟨proof⟩

lemma parent-upper-bound-alt: l ≤ parent l c ⇒ parent l c < c
⟨proof⟩

lemma parent-lower-bound: l ≤ parent l c ←→ l < c
⟨proof⟩

lemma grand-parent-upper-bound: parent l (parent l c) < c ←→ l ≤ c
⟨proof⟩

corollary parent-bounds: l < x ⇒ x < r ⇒ bounded l r (parent l x)
⟨proof⟩

lemma l-child-lower-bound: p < l-child l p ←→ l ≤ p
⟨proof⟩

corollary l-child-lower-bound-alt: l ≤ x ⇒ x ≤ p ⇒ x < l-child l p
⟨proof⟩

lemma parent-l-child[simp]: parent l (l-child l n) = n
⟨proof⟩

lemma r-child-lower-bound: l ≤ p ⇒ p < r-child l p
⟨proof⟩

corollary r-child-lower-bound-alt: l ≤ x ⇒ x ≤ p ⇒ x < r-child l p
⟨proof⟩

lemma parent-r-child[simp]: parent l (r-child l n) = n
⟨proof⟩

lemma smaller-l-child: l-child l x < r-child l x
⟨proof⟩
Lemma \( \text{parent-two-children}: \)
\[
(c = \text{l-child \ l \ p} \lor c = \text{r-child \ l \ p}) \leftrightarrow \text{parent \ l \ c} = \text{p}
\]
\( \text{⟨proof} \rangle \)

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 \text{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 \text{is-partial-heap}
\[
:: \ (\forall a::\text{order} \Rightarrow \forall a::\text{order} \Rightarrow \text{bool}) \Rightarrow \ (\text{int} \Rightarrow \forall a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool}
\]
where
\[
is\text{-partial-heap} \ cmp \ heap \ l \ m \ r \ i = (\forall x. \text{bounded} \ m \ r \ x \rightarrow \\
\text{bounded} \ m \ r \ (\text{parent} \ l \ x) \rightarrow \text{cmp} \ (\text{heap} \ (\text{parent} \ l \ x)) \ (\text{heap} \ x))
\]

Abbreviation \text{is-heap}
\[
:: \ (\forall a::\text{order} \Rightarrow \forall a::\text{order} \Rightarrow \text{bool}) \Rightarrow \ (\text{int} \Rightarrow \forall a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \\
\text{bool} \ where
\text{is-heap} \ cmp \ heap \ l \ r = \text{is-partial-heap} \ cmp \ heap \ l \ l \ r
\]

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 \text{sift-down} or \text{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 \( \text{True} \) without explicitly using the child relations.

Definition \text{is-partial-heap-except-down}
\[
:: \ (\forall a::\text{order} \Rightarrow \forall a::\text{order} \Rightarrow \text{bool}) \Rightarrow \ (\text{int} \Rightarrow \forall a::\text{order}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \\
\text{int} \Rightarrow \text{int} \Rightarrow \text{bool} \ where
\text{is-partial-heap-except-down} \ cmp \ heap \ l \ m \ r \ i = (\forall x. \text{bounded} \ m \ r \ x \rightarrow \\
((\text{parent} \ l \ x \neq i \rightarrow \text{bounded} \ m \ r \ (\text{parent} \ l \ x) \rightarrow \text{cmp} \ (\text{heap} \ (\text{parent} \ l \ x))) \ (\text{heap} \ x)) \land \\
(\text{parent} \ l \ x = i \rightarrow \text{bounded} \ m \ r \ (\text{parent} \ l \ (\text{parent} \ l \ x)) \\
\rightarrow \text{cmp} \ (\text{heap} \ (\text{parent} \ l \ (\text{parent} \ l \ x))) \ (\text{heap} \ x))))
\]
**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`:  
`is-partial-heap cmp heap l r r`  
(proof)

**lemma** `is-partial-heap-smaller-back`:  
`is-partial-heap cmp heap l m r → r' ≤ r → is-partial-heap cmp heap l m r'`  
(proof)

**lemma** `is-partial-heap-smaller-front`:  
`is-partial-heap cmp heap l m r → m ≤ m' → is-partial-heap cmp heap l m' r`  
(proof)

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) mod 2 ≤ m → is-partial-heap cmp heap l m r`  
(proof)

**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`
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**
  \[ l \leq m \leq r \]
  is-partial-heap cmp heap \(l\) \((m+1)\) \(r\)
- **Shows**
  is-partial-heap-except-down cmp heap \(l\) \(m\) \(r\) \(m\)

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(\(l := b\))) \(l\) \(r\) \(l\)

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’ \(l\) \((r+1)\) \(r\)

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**
  \(k < (i::int)\) \((\forall i. k < i \implies (\forall j. k < j \implies j < i \implies P j)) \implies P i)\)
- **Shows**
  \((\forall j. k < j \implies j < i \implies P j) \implies P j\)

**Theorem** strong-int-gr-induct:
- **Assumes**
  \(k < (i::int)\)
  \((\forall i. k < i \implies (\forall j. k < j \implies j < i \implies P j) \implies P i)\)
- **Shows**
  \(P i\)
proof

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)`

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 ⇒ '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)`  

**lemma** `arr-mset-remove-last`:

- `l ≤ r ⟹ arr-mset arr l r = arr-mset arr l (r + 1) - {#arr r#}`

**lemma** `arr-mset-append`:

- `l ≤ r ⟹ arr-mset arr l (r + 1) = arr-mset arr l r + {#arr r#}`

**corollary** `arr-mset-append-alt`:

- `l ≤ r ⟹ arr-mset (arr(r := b)) l (r + 1) = arr-mset arr l r + {#b#}`

**lemma** `arr-mset-remove-first`:

- `i ≤ r ⟹ arr-mset arr (i - 1) r = arr-mset arr i r + {#arr (i - 1)#}`

**lemma** `arr-mset-split`:
assumes  \( l \leq m \leq r \)
shows  \( \text{arr-mset} \ arr \ l \ r = \text{arr-mset} \ arr \ l \ m + \text{arr-mset} \ arr \ m \ r \)
⟨proof⟩

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} & \not= x
\end{align*}
shows  \( \text{cmp} \ (\text{heap} \ l) \ x \)
⟨proof⟩

3.2 Lemmas on swap and eq-on

lemma eq-on-subset:
\( \text{arr}1 = \text{arr}2 \, \text{on} \ R \implies S \subseteq R \implies \text{arr}1 = \text{arr}2 \, \text{on} \ S \)
⟨proof⟩

lemma swap-swaps:
\( \text{arr}’ = \text{swap} \ \text{arr} \ x \ y \implies \text{arr}’ \ y = \text{arr} \ x \land \text{arr}’ \ x = \text{arr} \ y \)
⟨proof⟩

lemma swap-only-swaps:
\( \text{arr}’ = \text{swap} \ \text{arr} \ x \ y \implies z \neq x \implies z \neq y \implies \text{arr}’ \ z = \text{arr} \ z \)
⟨proof⟩

lemma swap-commute: \( \text{swap} \ \text{arr} \ x \ y = \text{swap} \ \text{arr} \ y \ x \)
⟨proof⟩

lemma swap-eq-on:
\( \text{arr}1 = \text{arr}2 \, \text{on} \ S \implies x \notin S \implies y \notin S \implies \text{arr}1 = \text{swap} \ \text{arr}2 \ x \ y \, \text{on} \ S \)
⟨proof⟩

corollary swap-parent-eq-on:
assumes  
\begin{align*}
\text{arr}1 & = \text{arr}2 \, \text{on} \ - \{l..<r\} \\
l & < c \ c & < r
\end{align*}
shows  \( \text{arr}1 = \text{swap} \ \text{arr}2 \ (\text{parent} \ l \ c) \ c \, \text{on} \ - \{l..<r\} \)
⟨proof⟩

corollary swap-child-eq-on:
assumes

\( \text{arr1} = \text{arr2 on} - \{l..<r\} \)
\( c = l\text{-child} \ l \ p \lor c = r\text{-child} \ l \ p \)
\( l \leq p < c \)

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

\langle \text{proof} \rangle

lemma \text{swap-child-mset}:

assumes

\( \text{arr-mset arr1} \ l \ r = \text{arr-mset arr2} \ l \ r \)
\( c = l\text{-child} \ l \ p \lor c = r\text{-child} \ l \ p \)
\( l \leq p < c \)

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

\langle \text{proof} \rangle

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 \text{sift-up} and \text{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 \text{cmp-swapI}:

fixes \text{arr} :: 'a::order => 'a::order

assumes

\( m < n \land x < y \)
\( m < n \land x < y \Rightarrow x = m \Rightarrow y = n \Rightarrow P (\text{arr} \ n) (\text{arr} \ m) \)
\( m < n \land x < y \Rightarrow x \neq m \Rightarrow x \neq n \Rightarrow y \neq m \Rightarrow y \neq n \Rightarrow P (\text{arr} \ m) (\text{arr} \ n) \)
\( m < n \land x < y \Rightarrow x = n \Rightarrow y \neq m \Rightarrow y \neq n \Rightarrow P (\text{arr} \ m) (\text{arr} \ y) \)
\( m < n \land x < y \Rightarrow x \neq m \Rightarrow x \neq n \Rightarrow y = n \Rightarrow P (\text{arr} \ m) (\text{arr} \ x) \)
\( m < n \land x < y \Rightarrow x \neq m \Rightarrow x \neq n \Rightarrow y = m \Rightarrow P (\text{arr} \ x) (\text{arr} \ n) \)

shows \( P (\text{swap arr} \ x \ y \ m) (\text{swap arr} \ x \ y \ n) \)

\langle \text{proof} \rangle

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 imple-
mentations. 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.

```plaintext
procedure-spec prnt (l, x) returns p
  assumes True
  ensures p = parent l 0 x 0
  defines \langle p = ((x - l - 1) / 2 + l) \rangle
  \langle proof \rangle

procedure-spec left-child (l, x) returns lc
  assumes True
  ensures lc = l-child l 0 x 0
  defines \langle lc = 2 * x - l + 1 \rangle
  \langle proof \rangle

procedure-spec right-child (l, x) returns rc
  assumes True
  ensures rc = r-child l 0 x 0
  defines \langle rc = 2 * x - l + 2 \rangle
  \langle proof \rangle

procedure-spec swp (heap, x, y) returns heap
  assumes True
  ensures heap = swap heap 0 x 0 y 0
  defines \langle tmp = heap[x]; heap[x] = heap[y]; heap[y] = tmp \rangle
  \langle proof \rangle

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
\[ \text{l < r} \]
\[ \text{is-heap (\leq) heap l r} \]
\[ \text{shows heap l = Min-mset (arr-mset heap l r)} \]
\[ \langle \text{proof} \rangle \]

**procedure-spec** get-min (heap, l, r) **returns** min
\[ \text{assumes l < r \land is-heap (\leq) heap l r} \]
\[ \text{ensures min = Min-mset (arr-mset heap0 l0 r0)} \]
\[ \text{for heap[]} l r \]
\[ \text{defines (min = heap[l])} \]
\[ \langle \text{proof} \rangle \]

### 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:
\[ \text{fixes heap::int \Rightarrow int} \]
\[ \text{assumes} \]
\[ \text{is-heap-except-up (\leq) heap l r x} \]
\[ \text{parent l x \geq l} \]
\[ (heap x) \leq (heap (parent l x)) \]
\[ \text{bounded l r k} \]
\[ k \neq (\text{parent l x}) \]
\[ \text{bounded l r (parent l k)} \]
\[ \text{shows (swap heap (parent l x) x (parent l k)) \leq (swap heap (parent l x) x k)} \]
\[ \langle \text{proof} \rangle \]

**lemma** sift-up-step-aux2:
\[ \text{fixes heap::int \Rightarrow int} \]
\[ \text{assumes} \]
\[ \text{is-heap-except-up (\leq) heap l r x} \]
\[
\begin{align*}
\text{parent } l x & \geq l \\
\text{heap } x & \leq (\text{heap } (\text{parent } l x)) \\
\text{bounded } l r k \\
\text{parent } l k & = \text{parent } l x \\
\text{bounded } l r (\text{parent } l (\text{parent } l k))
\end{align*}
\]

shows
\[
\text{swap heap } (\text{parent } l x) x (\text{parent } l (\text{parent } l k)) \leq \text{swap heap } (\text{parent } l x) x k
\]

\langle \text{proof} \rangle

\textbf{lemma sift-up-step:}
\textbf{fixes heap::int }\Rightarrow\text{ int}
\textbf{assumes}
\text{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))
\textbf{shows is-heap-except-up } (\leq) \text{ (swap heap } (\text{parent } l x) x) l r (\text{parent } l x)
\langle \text{proof} \rangle

\text{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.}

\textbf{procedure-spec sift-up } (\text{heap, } l, r, x) \textbf{ returns heap}
\textbf{assumes is-heap-except-up } (\leq) \text{ heap } l r x \land \text{bounded } l r x
\textbf{ensures is-heap } (\leq) \text{ heap } l_0 r_0 \\
\text{arr-mset heap}_0 l_0 r_0 = \text{arr-mset heap } l_0 r_0 \\
\text{heap}_0 = \text{heap on } - \{l_0..<r_0\}
\textbf{for heap}[l x r]
\textbf{defines (}
\text{p} = \text{prnt}(l, x);
\text{while } (x > l \land \text{heap}[x] \leq \text{heap}[p])
\text{@variant } (x - l)
\text{@invariant is-heap-except-up } (\leq) \text{ heap } l r x \land p = \text{parent } l x \\
\text{bounded } l r x \land \text{arr-mset heap}_0 l_0 r_0 = \text{arr-mset heap } l r \\
\text{heap}_0 = \text{heap on } - \{l..<r\}
\{ \\
\text{heap} = \text{swp}(\text{heap}, p, x); \\
x = p; \\
p = \text{prnt}(l, x)
\}
\text{)}
\langle \text{proof} \rangle

\textit{insert} inserts an element into a heap by appending it to the heap and restor-
ing the heap invariant with sift-up.

**procedure-spec** `insert (heap, l, r, el) returns (heap, l, r)`

**assumes** `is-heap (≤) heap l r` \( l \leq r \)

**ensures** `is-heap (≤) heap l r`

\[ l = l_0 \land r = r_0 + 1 \land heap_0 = heap_{on - \{l..<r\}} \]

for `heap l r el`

**defines**

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

**proof**

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` \( l \leq x \land x \leq r \)

**ensures** `is-partial-heap (≤) heap l x r` \( l_0 x_0 r_0 \land \)

\[ arr-mset heap_0 l_0 r_0 = arr-mset heap l_0 r_0 \land \]

\[ heap_0 = heap_{on - \{l_0..<r_0\}} \]

**defines**

- `lc = left-child(l, x);`
- `rc = right-child(l, x);`

while `(l < r \land (heap[lc] < heap[x] \lor (rc < r \land heap[rc] < heap[x])))`

@variant `(r - x)`

@invariant `is-partial-heap-except-down (≤) heap l x r x` \( x_0 \leq x \land x \leq r \land lc = l\text{-child } l x \land rc = r\text{-child } l x \land \)

\[ arr-mset heap_0 l r = arr-mset heap l r \land \]

\[ heap_0 = heap_{on - \{l..<r\}} \]

{ `smallest = lc;`
  if `(rc < r \land heap[rc] < heap[lc])` { `smallest = rc` }
};
heap = swp(heap, x, smallest);
x = smallest;
lc = left-child(l, x);
rc = right-child(l, x)
}
⟨proof⟩
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 ⇒ 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⟩
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 ∧ is-heap (\leq) heap l r
ensures is-heap (\leq) heap l r ∧
arr-mset heap l r = arr-mset heap0 l0 r0 - {#Min-mset (arr-mset heap0 l0 r0)#} ∧
  l = l0 ∧ r = r0 - 1 ∧
  heap0 = heap on - {l0..<r0}
for heap l r
defines
  r = r - 1;
  heap[l] = heap[r];
  heap = sift-down(heap, l, r, l)
⟩
⟨proof⟩

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 l_0 r_0 ∧
arr-mset heap l_0 r_0 = arr-mset heap_0 l_0 r_0 ∧
heap_0 = heap on − {l_0..< r_0}
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 heap_0 l_0 r_0 ∧
    l − 1 ≤ y ∧ y < r ∧ heap_0 = heap on − {1..<r})
  { heap = sift-down(heap, l, r, y);
    y = y − 1
  }
⟨proof⟩

4.3 Heapsort Implementation

The final part of this submission is the implementation of the in-place heap-
sort. Firstly it builds the ≤-heap and then it iteratively removes the min-
imum 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 m ≤ r
arr-mset arr’ l (m − 1) = arr-mset arr l m − {#Min-mset (arr-mset
arr l m)#}
arr = arr’ on − {l..<m}

16
\begin{align*}
\text{mod-arr} &= \text{arr}'(m - 1 := \text{Min-mset } (\text{arr-mset } \text{arr } l \ m)) \\
\text{shows} \quad &\text{arr-mset } \text{arr } l \ r = \text{arr-mset } \text{mod-arr } l \ r \\
\langle \text{proof} \rangle
\end{align*}

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** \( \text{arr} :\Rightarrow \text{int} \)

**assumes**

\[ l_0 < m \ m \leq r_0 \]
\[ \text{arr} = \text{arr}' \text{ on } - \{l_0..<m\} \]
\[ \forall i. \forall j. \text{bounded } m \ r_0 \ i \rightarrow \text{bounded } m \ r_0 \ j \rightarrow \ i < j \rightarrow \text{arr } j \leq \text{arr } i \]
\[ \forall x \in \#\text{arr-mset } \text{arr } l_0 \ m. \forall y \in \#\text{arr-mset } \text{arr } m \ r_0. \neg \ x < y \]
\[ \text{bounded } (m - 1) \ r_0 \ i \]
\[ \text{bounded } (m - 1) \ r_0 \ j \]
\[ i < j \]
\[ \text{mod-arr} = (\text{arr}'(m - 1 := \text{Min-mset } (\text{arr-mset } \text{arr } l_0 \ m))) \]

**shows** \( \text{mod-arr } j \leq \text{mod-arr } i \)

\( \langle \text{proof} \rangle \)

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 \texttt{vcs-cs} doesn’t terminate in the latter case.

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

**fixes** \( \text{arr} :\text{int } \Rightarrow \text{int} \)

**assumes**

\[ l_0 < m \]
\[ m \leq r_0 \]
\[ \text{arr-mset } \text{arr}' \ l_0 \ (m - 1) = \text{arr-mset } \text{arr } l_0 \ m \ - \ \{\#\text{Min-mset } (\text{arr-mset } \text{arr } l_0 \ m)\}\} \]
\[ \text{arr} = \text{arr}' \text{ on } - \{l_0..<m\} \]
\[ \forall x \in \#\text{arr-mset } \text{arr } l_0 \ m. \forall y \in \#\text{arr-mset } \text{arr } m \ r_0. \neg \ x < y \]
\[ \text{mod-arr} = \text{arr}'(m - 1 := \text{Min-mset } (\text{arr-mset } \text{arr } l_0 \ m)) \]
\[ x \in \#\text{arr-mset } \text{mod-arr } l_0 \ (m - 1) \]
\[ y \in \#\text{arr-mset } \text{mod-arr } (m - 1) \ r_0 \]

**shows** \( \neg \ x < y \)

\( \langle \text{proof} \rangle \)
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 ≤ r
  ensures array-is-sorted (≥) arr l0 r0 ∧
    arr-mset arr0 l0 r0 = arr-mset arr l0 r0 ∧
    arr0 = arr on − {l0..<r0} ∧ l = l0 ∧ r = r0
  for l r arr[]
  defines (m = r;
    while (m > l)
      @variant (m − l + 1)
      @invariant (is-heap (≤) arr l m ∧
        array-is-sorted (≥) arr m r0 ∧
        (∀x ∈ # arr-mset arr l0 m. ∀y ∈ # arr-mset arr m r0. ¬x < y) ∧
        arr-mset arr0 l0 r0 = arr-mset arr l0 r0 ∧
        l ≤ m ∧ m ≤ r0 ∧ l = l0 ∧ arr0 = arr on − {l0..<r0};
      { min = get-min(arr, l, m);
        (arr, l, m) = del-min(arr, l, m);
        arr[m] = min
      }
    )
  )
⟨proof⟩
end

References

