Verification of Selection and Heap Sort Using Locales

Danijela Petrović

September 19, 2015

Abstract

Stepwise program refinement techniques can be used to simplify program verification. Programs are better understood since their main properties are clearly stated, and verification of rather complex algorithms is reduced to proving simple statements connecting successive program specifications. Additionally, it is easy to analyze similar algorithms and to compare their properties within a single formalization. Usually, formal analysis is not done in educational setting due to complexity of verification and a lack of tools and procedures to make comparison easy. Verification of an algorithm should not only give correctness proof, but also better understanding of an algorithm. If the verification is based on small step program refinement, it can become simple enough to be demonstrated within the university-level computer science curriculum. In this paper we demonstrate this and give a formal analysis of two well known algorithms (Selection Sort and Heap Sort) using proof assistant Isabelle/HOL and program refinement techniques.

Contents

1 Introduction 2
2 Locale Sort 4
3 Defining data structure and key function remove_max 5
  3.1 Describing data structure ............................... 5
  3.2 Function remove_max ................................. 6
4 Verification of functional Selection Sort 10
  4.1 Defining data structure ............................. 10
  4.2 Defining function remove_max ..................... 11
1 Introduction

Using program verification within computer science education. Program verification is usually considered to be too hard and long process that acquires good mathematical background. A verification of a program is performed using mathematical logic. Having the specification of an algorithm inside the logic, its correctness can be proved again by using the standard mathematical apparatus (mainly induction and equational reasoning). These proofs are commonly complex and the reader must have some knowledge about mathematical logic. The reader must be familiar with notions such as satisfiability, validity, logical consequence, etc. Any misunderstanding leads into a loss of accuracy of the verification. These formalizations have common disadvantage, they are too complex to be understood by students, and this discourage students most of the time. Therefore, programmers and their educators rather use traditional (usually trial-and-error) methods.

However, many authors claim that nowadays education lacks the formal approach and it is clear why many advocate in using proof assistants[?]. This is also the case with computer science education. Students are presented many algorithms, but without formal analysis, often omitting to mention when algorithm would not work properly. Frequently, the center of a study is implementation of an algorithm whereas understanding of its structure and its properties is put aside. Software verification can bring more formal approach into teaching of algorithms and can have some advantages over traditional teaching methods.

• Verification helps to point out what are the requirements and conditions that an algorithm satisfies (pre-conditions, post-conditions and invariant conditions) and then to apply this knowledge during programming. This would help both students and educators to better understand input and output specification and the relations between them.

• Though program works in general case, it can happen that it does not work for some inputs and students must be able to detect these
situations and to create software that works properly for all inputs.

- It is suitable to separate abstract algorithm from its specific implementation. Students can compare properties of different implementations of the same algorithms, to see the benefits of one approach or another. Also, it is possible to compare different algorithms for same purpose (for example, for searching element, sorting, etc.) and this could help in overall understanding of algorithm construction techniques.

Therefore, lessons learned from formal verification of an algorithm can improve someone's style of programming.

**Modularity and refinement.** The most used languages today are those who can easily be compiled into efficient code. Using heuristics and different data types makes code more complex and seems to novices like perplex mixture of many new notions, definitions, concepts. These techniques and methods in programming makes programs more efficient but are rather hard to be intuitively understood. On the other hand highly accepted principle in nowadays programming is modularity. Adhering to this principle enables programmer to easily maintain the code.

The best way to apply modularity on program verification and to make verification flexible enough to add new capabilities to the program keeping current verification intact is program refinement. Program refinement is the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. It starts from the abstract level, describing only the requirements for input and output. Implementation is obtained at the end of the verification process (often by means of code generation [?]). Stepwise refinement allows this process to be done in stages. There are many benefits of using refinement techniques in verification:

- It gives a better understanding of programs that are verified.
- The algorithm can be analyzed and understood on different level of abstraction.
- It is possible to verify different implementations for some part of the program, discussing the benefits of one approach or another.
- It can be easily proved that these different implementation share some same properties which are proved before splitting into two directions.
- It is easy to maintain the code and the verification. Usually, whenever the implementation of the program changes, the correctness proofs must be adapted to these changes, and if refinement is used, it is not necessary to rewrite entire verification, just add or change small part of it.
Using refinement approach makes algorithm suitable for a case study in teaching. Properties and specifications of the program are clearly stated and it helps teachers and students better to teach or understand them.

We claim that the full potential of refinement comes only when it is applied stepwise, and in many small steps. If the program is refined in many steps, and data structures and algorithms are introduced one-by-one, then proving the correctness between the successive specifications becomes easy. Abstracting and separating each algorithmic idea and each data-structure that is used to give an efficient implementation of an algorithm is very important task in programmer education.

As an example of using small step refinement, in this paper we analyze two widely known algorithms, Selection Sort and Heap Sort. There are many reasons why we decided to use them.

- They are largely studied in different contexts and they are studied in almost all computer science curricula.
- They belong to the same family of algorithms and they are good example for illustrating the refinement techniques. They are a nice example of how one can improve on a same idea by introducing more efficient underlying data-structures and more efficient algorithms.
- Their implementation uses different programming constructs: loops (or recursion), arrays (or lists), trees, etc. We show how to analyze all these constructs in a formal setting.

There are many formalizations of sorting algorithms that are done both automatically or interactively and they undoubtedly proved that these algorithms are correct. In this paper we are giving a new approach in their verification, that insists on formally analyzing connections between them, instead of only proving their correctness (which has been well established many times). Our central motivation is that these connections contribute to deeper algorithm understanding much more than separate verification of each algorithm.

## 2 Locale Sort

```plaintext
theory Sort
imports Main
  ~/src/HOL/Library/Permutation
begin
```

First, we start from the definition of sorting algorithm. What are the basic properties that any sorting algorithm must satisfy? There are two basic features any sorting algorithm must satisfy:
The elements of sorted array must be in some order, e.g. ascending or descending order. In this paper we are sorting in ascending order.

\[ \text{sorted} (\text{sort } l) \]

The algorithm does not change or delete elements of the given array, e.g. the sorted array is the permutation of the input array.

\[ \text{sort } l <\sim\sim> l \]

code

locale \textit{Sort} =
fixes sort :: 'a::linorder list ⇒ 'a list
assumes sorted: sorted (sort l)
assumes permutation: sort l <\sim\sim> l
end

3 Defining data structure and key function remove_max

theory \textit{RemoveMax}
imports \textit{Sort}
begin

3.1 Describing data structure

We have already said that we are going to formalize heap and selection sort and to show connections between these two sorts. However, one can immediately notice that selection sort is using list and heap sort is using heap during its work. It would be very difficult to show equivalency between these two sorts if it is continued straightforward and independently proved that they satisfy conditions of locale \textit{Sort}. They work with different objects. Much better thing to do is to stay on the abstract level and to add the new locale, one that describes characteristics of both list and heap.

locale \textit{Collection} =
fixes empty :: 'b
— Represents empty element of the object (for example, for list it is [])
fixes is-empty :: 'b ⇒ bool
— Function that checks weather the object is empty or not
fixes of-list :: 'a list ⇒ 'b
— Function transforms given list to desired object (for example, for heap sort, function of-list transforms list to heap)
fixes multiset :: 'b ⇒ 'a multiset
— Function makes a multiset from the given object. A multiset is a collection without order.
assumes is-empty-inj: is-empty e ⇒ e = empty
It must be assured that the empty element is empty.

**assumes** is-empty-empty: is-empty empty

Must be satisfied that function is_empty returns true for element empty.

**assumes** multiset-empty: multiset empty = \{#\}

Multiset of an empty object is empty multiset.

**assumes** multiset-of-list: multiset (of-list i) = mset i

Multiset of an object gained by applying function of_list must be the same as the multiset of the list. This, practically, means that function of_list does not delete or change elements of the starting list.

**begin**

**lemma** is-empty-as-list: is-empty e \implies multiset e = \{#\}

**using** is-empty-inj multiset-empty

**by** auto

**definition** set :: 'b \Rightarrow a set

\[\text{simp}]: \text{set l = set-mset (multiset l)}\]

**end**

### 3.2 Function remove_max

We wanted to emphasize that algorithms are same. Due to the complexity of the implementation it usually happens that simple properties are omitted, such as the connection between these two sorting algorithms. This is a key feature that should be presented to students in order to understand these algorithms. It is not unknown that students usually prefer selection sort for its simplicity whereas avoid heap sort for its complexity. However, if we can present them as the algorithms that are same they may hesitate less in using the heap sort. This is why the refinement is important. Using this technique we were able to notice these characteristics. Separate verification would not bring anything new. Being on the abstract level does not only simplify the verifications, but also helps us to notice and to show students important features. Even further, we can prove them formally and completely justify our observation.

**locale** RemoveMax = Collection empty is-empty of-list multiset for

**empty** :: 'b

**is-empty** :: 'b \Rightarrow bool \text{ and}

**of-list** :: 'a::linorder list \Rightarrow 'b \text{ and}

**multiset** :: 'b \Rightarrow 'a::linorder multiset +

**fixes** remove-max :: 'b \Rightarrow 'a \times 'b

Function that removes maximum element from the object of type 'b. It returns maximum element and the object without that maximum element.

**fixes** inv :: 'b \Rightarrow bool

It checks weather the object is in required condition. For example, if we expect to work with heap it checks weather the object is heap. This is called invariant condition.

**assumes** of-list-inv: inv (of-list x)

This condition assures that function of_list made a object with desired
property.

assumes remove-max-max:
\[
\neg \text{is-empty } l ; \text{inv } l ; (m, l') = \text{remove-max } l \implies m = \text{Max } \{ \text{set } l \}
\]
— — First parameter of the return value of the function remove-max is the maximum element.

assumes remove-max-multiset:
\[
\neg \text{is-empty } l ; \text{inv } l ; (m, l') = \text{remove-max } l \implies \text{multiset } l' + \{ \#m\} = \text{multiset } l
\]
— — Condition for multiset, ensures that nothing new is added or nothing is lost after applying remove-max function.

assumes remove-max-inv:
\[
\neg \text{is-empty } l ; \text{inv } l ; (m, l') = \text{remove-max } l \implies \text{inv } l'
\]
— — Ensures that invariant condition is true after removing maximum element. Invariant condition must be true in each step of sorting algorithm, for example if we are sorting using heap than in each iteration we must have heap and function remove-max must not change that.

begin

lemma remove-max-multiset-size:
\[
\neg \text{is-empty } l ; \text{inv } l ; (m, l') = \text{remove-max } l \implies 
\text{size } \{ \text{multiset } l \} > \text{size } \{ \text{multiset } l' \}
\]
using remove-max-multiset[of l m l']
by (metis mset-less-size multi-psub-of-add-self)

lemma remove-max-set:
\[
\neg \text{is-empty } l ; \text{inv } l ; (m, l') = \text{remove-max } l \implies 
\text{set } l' \cup \{ \#m\} = \text{set } l
\]
using remove-max-multiset[of l m l']
by (metis set-def set-mset-single set-mset-union)

As it is said before in each iteration invariant condition must be satisfied, so the inv l is always true, e.g. before and after execution of any function. This is also the reason why sort function must be defined as partial. This function parameters stay the same in each step of iteration – list stays list, and heap stays heap. As we said before, in Isabelle/HOL we can only define total function, but there is a mechanism that enables total function to appear as partial one:

partial-function (tailrec) ssort' where

ssort' l sl =
( if is-empty l then
  sl
else
  let
    (m, l') = remove-max l
  in
    ssort' l' (m \# sl))

declare ssort'..simps[code]
definition ssort :: 'a list \Rightarrow 'a list where
ssort l = ssort' (of-list l) []

inductive ssort'-dom where
  step: [∀m l', [¬ is-empty l; (m, l') = remove-max l] \implies
           ssort'-dom (l', m \# sl)] \implies ssort'-dom (l, sl)

lemma ssort'-termination:
  assumes inv (fst p)
  shows ssort'-dom p
using assms
proof (induct p rule: wf-induct [of measure (λ(l, sl). size (multiset l))])
  let ?r = measure (λ(l, sl). size (multiset l))
  fix p :: 'b × 'a list
  assume inv (fst p) and *:
  ∀y. (y, p) ∈ ?r \implies inv (fst y) \implies ssort'-dom y
  obtain l sl where p = (l, sl)
  by (cases p) auto
  show ssort'-dom p
  proof (rule *)
    assume ¬ is-empty l (m, l') = remove-max l
  show ssort'-dom (l', m\#sl)
  proof (rule ssort'-dom.step)
    fix l'
    assume ¬ is-empty l (m, l') = remove-max l
    show ssort'-dom (l', m\#sl)
    proof (rule ssort'-dom.step)
      show ((l', m\#sl), p) ∈ ?r inv (fst (l', m\#sl))
        using (p = (l, sl)); (inv (fst p)): ¬ is-empty l
        using (m, l') = remove-max l
        using remove-max-inv[of l m l']
      using remove-max-multiset-size[of l m l']
      by auto
    qed
  qed
  qed simp

lemma ssort'Induct:
  assumes inv l P l sl
  \( l sl m l'\)
  [¬ is-empty l; inv l; (m, l') = remove-max l; P l sl] \implies P l' (m \# sl)
  shows P empty (ssort' l sl)
proof
  from (inv l) have ssort'-dom (l, sl)
    using ssort'-termination
    by auto
  thus ?thesis
  using assms
proof (induct (l, sl) arbitrary: l sl rule: ssort'-dom.induct)
  case (step l sl)
  show ?case
  proof (cases is-empty l)
    case True
    thus ?thesis
  qed
next
  case False
  let ?p = remove-max l
  let ?m = fst ?p and ?l' = snd ?p
  show ?thesis
  using False step(2)[of ?m ?l'] step(3)
  using step(4) step(5)[of l ?m ?l' sl] step(5)
  using remove-max-inv[of l ?m ?l']
  using ssort'.simps[of l sl]
  by (cases ?p) auto
qed
qed
qed

lemma mset-ssort':
  assumes inv l
  shows mset (ssort' l sl) = multiset l + mset sl
  using assms
  proof -
    have multiset empty + mset (ssort' l sl) = multiset l + mset sl
    using assms
    proof (rule ssort'Induct)
      fix l sl1 m l'
      assume ¬ is-empty l1
      inv l1
      (m, l') = remove-max l1
      multiset l1 + mset sl1 = multiset l + mset sl
      thus multiset l' + mset (m # sl1) = multiset l + mset sl
      using remove-max-multiset[of l1 m l']
      by (auto simp add: union-commute union-lcomm)
    qed simp
    thus ?thesis
    using multiset-empty
    by simp
  qed

lemma sorted-ssort':
  assumes inv l sorted sl ∧ (∀ x ∈ set l. (∀ y ∈ List.set sl. x ≤ y))
  shows sorted (ssort' l sl)
  using assms
  proof -
    have sorted (ssort' l sl) ∧
      (∀ x ∈ set empty. (∀ y ∈ List.set (ssort' l sl). x ≤ y))
    using assms
    proof (rule ssort'Induct)
      fix l sl m l'
      assume ¬ is-empty l
      ...
\[
\text{inv } l
\]
\[
(m, l') = \text{remove-max } l
\]
\[
\text{sorted } sl \land (\forall x \in \text{set } l, \forall y \in \text{List set } sl, \ x \leq y)
\]
\[
\text{thus } \text{sorted } (m \# sl) \land (\forall x \in \text{set } l', \forall y \in \text{List set } (m \# sl), \ x \leq y)
\]
\[
\text{using } \text{remove-max-set}[\text{of } l m l'] \text{ remove-max-max}[\text{of } l m l']
\]
\[
\text{apply } (\text{auto simp add: sorted-Cons})
\]
\[
\text{by } (\text{metis Max-ge finite-set-mset insert-iff mem-set-mset-iff})
\]
\[
\text{qed}
\]
\[
\text{thus } \text{thesis}
\]
\[
\text{by simp}
\]
\[
\text{qed}
\]

**lemma** sorted-ssort: sorted (ssort i)

**unfolding** ssort-def

**using** sorted-ssort"[of of-list i []] of-list-inv

**by** auto

**lemma** permutation-ssort: ssort l <~~> l

**proof** (subst mset-eq-perm[symmetric])

**show** mset (ssort l) = mset l

**unfolding** ssort-def

**using** mset-ssort"[of of-list l []]

**using** multiset-of-list of-list-inv

**by** simp

**qed**

**end**

Using assumptions given in the definitions of the locales *Collection* and *RemoveMax* for the functions *multiset*, *is_empty*, *of_list* and *remove_max* it is no difficulty to show:

**sublocale** RemoveMax < Sort ssort

**by** (unfold-locales) (auto simp add: sorted-ssort permutation-ssort)

**end**

### 4 Verification of functional Selection Sort

**theory** SelectionSort-Functional

**imports** RemoveMax

**begin**

#### 4.1 Defining data structure

Selection sort works with list and that is the reason why *Collection* should be interpreted as list.

**interpretation** Collection [] \( \lambda l. \ l = [] \) id mset

**by** (unfold-locales, auto)
4.2 Defining function remove_max

The following is definition of remove_max function. The idea is very well known – assume that the maximum element is the first one and then compare with each element of the list. Function \( f \) is one step in iteration, it compares current maximum \( m \) with one element \( x \), if it is bigger then \( m \) stays current maximum and \( x \) is added in the resulting list, otherwise \( x \) is current maximum and \( m \) is added in the resulting list.

\[ \text{fun } f \text{ where } f (m, l) x = (\text{if } x \geq m \text{ then } (x, m#l) \text{ else } (m, x#l)) \]

\[ \text{definition remove-max where } \]
\[ \text{remove-max } l = \text{foldl } f (\text{hd } l, []) (\text{tl } l) \]

\[ \text{lemma max-Max-commute: } \]
\[ \text{finite } A \rightarrow \text{ max (Max (insert } m \text{ A)) } x = \text{ max m (Max (insert } x \text{ A))} \]
\[ \text{apply } \text{ (cases } A = \{\}, \text{ simp) by } (\text{metis Max-insert max.commute max.left-commute}) \]

The function really returned the maximum value.

\[ \text{lemma remove-max-max-lemma: } \]
\[ \text{shows } \text{fst (foldl } f (m, t) l) = \text{ Max (set } m \text{ # l)} \]
\[ \text{using assms proof } (\text{induct } l \text{ arbitrary: } m \text{ t rule: rev-induct) case (snoc } x \text{ xs)} \]
\[ \text{let } ?a = \text{foldl } f (m, t) \text{ xs) let } ?m' = \text{fst } ?a \text{ and } ?t' = \text{snd } ?a \]
\[ \text{have } \text{fst (foldl } f (m, t) (\text{xs } \text{@[}x]) = \text{ max } m' \text{ x) by } (\text{cases } ?a) \text{ (auto simp add: max-def) thus } ?case } \]
\[ \text{using snoc by } (\text{simp add: max-Max-commute}) qed simp \]

\[ \text{lemma remove-max-max: } \]
\[ \text{assumes } l \neq [] (m, l') = \text{remove-max } l \]
\[ \text{shows } m = \text{Max (set } l) \]
\[ \text{using assms unfolding remove-max-def using remove-max-max-lemma[of } hd \text{ } l [] \text{ tl } l \]
\[ \text{using } \text{fst-cone[of } m \text{ } l' ] by simp} \]

Nothing new is added in the list and noting is deleted from the list except the maximum element.

\[ \text{lemma remove-max-mset-lemma: } \]
\[ \text{assumes } (m, l') = \text{foldl } f (m', t') \text{ l} \]
\[ \text{shows } \text{mset } (m \text{ } \# \text{ } l') = \text{mset } (m' \text{ } \# \text{ } t' \@ \text{ } l) \]
\[ \text{using assms} \]
proof (induct l arbitrary; l' m m' t' rule: rev-induct)
  case (snoc x xs)
  let ?a = foldl f (m', t') xs
  let ?m' = fst ?a and ?t' = snd ?a
  have mset (?m' # ?t') = mset (m' # t' @ xs)
    using snoc(1) [of ?m' ?t' m']
    by simp
  thus ?case
    using snoc(2)
    apply (cases ?a)
    apply (auto split: split-if-asn, (simp add: union-lcomm union-commute)+)
    by (metis union-assoc)
qed simp

lemma remove-max-mset:
  assumes l ≠ [] (m, l') = remove-max l
  shows mset l' + {#m#} = mset l
using assms
unfolding remove-max-def
using remove-max-mset-lemma[of m l' hd l tl l]
by auto

definition ssf-ssort' where
  [simp, code del]: ssf-ssort' = RemoveMax.ssort' (λ l. l = []) remove-max

definition ssf-ssort where
  [simp, code del]: ssf-ssort = RemoveMax.ssort (λ l. l = []) id remove-max

interpretation SSRemoveMax:
RemoveMax [] λ l. l = [] id mset remove-max λ -. True
where
RemoveMax.ssort' (λ l. l = []) remove-max = ssf-ssort'
and
RemoveMax.ssort (λ l. l = []) id remove-max = ssf-ssort
using remove-max-max
by (unfold-locales, auto simp add: remove-max-mset)

end

5 Verification of Heap Sort

theory Heap
imports RemoveMax
begin

5.1 Defining tree and properties of heap

datatype 'a Tree = E | T 'a 'a Tree 'a Tree
With $E$ is represented empty tree and with $T \quad 'a \quad 'a \quad Tree \quad 'a \quad Tree$ is represented a node whose root element is of type \textquote SingleQuote 'a and its left and right branch is also a tree of type \textquote SingleQuote 'a.

\begin{verbatim}
primrec size :: \textquotenote SingleQuote 'a Tree \Rightarrow \textquotenote SingleQuote nat
where
  size E = 0
| size (T v l r) = 1 + size l + size r
\end{verbatim}

Definition of the function that makes a multiset from the given tree:

\begin{verbatim}
primrec multiset where
  multiset E = {#}
| multiset (T v l r) = multiset l + {#v#} + multiset r
\end{verbatim}

\begin{verbatim}
primrec val where
  val (T v - -) = v
\end{verbatim}

Definition of the function that has the value \textquote SingleQuote True if the tree is heap, otherwise it is \textquote SingleQuote False:

\begin{verbatim}
fun is-heap :: \textquotenote SingleQuote 'a::linorder Tree \Rightarrow \textquotenote SingleQuote bool
where
  is-heap E = True
| is-heap (T v E E) = True
| is-heap (T v E r) = (v \geq val r \land is-heap r)
| is-heap (T v l E) = (v \geq val l \land is-heap l)
| is-heap (T v l r) = (v \geq val r \land is-heap r \land v \geq val l \land is-heap l)
\end{verbatim}

\begin{verbatim}
lemma heap-top-geq:
  assumes a \in \# multiset t is-heap t
  shows val t \geq a
using assms
by (induct t rule: is-heap.induct) (auto split: split-if-asm)
\end{verbatim}

\begin{verbatim}
lemma heap-top-max:
  assumes t \neq E is-heap t
  shows val t = Max (set-mset (multiset t))
proof (rule Max-eqI[symmetric])
  fix y
  assume y \in set-mset (multiset t)
  thus y \leq val t
    using heap-top-geq[of t y] (is-heap t)
    by simp
next
  show val t \in set-mset (multiset t)
    using (t \neq E)
    by (cases t) auto
qed simp
\end{verbatim}

The next step is to define function \textquote SingleQuote remove_max, but the question is whether implementation of \textquote SingleQuote remove_max depends on implementation of the functions \textquote SingleQuote is_heap and \textquote SingleQuote multiset. The answer is negative. This suggests that another
step of refinement could be added before definition of function \texttt{remove\_max}. Additionally, there are other reasons why this should be done, for example, function \texttt{remove\_max} could be implemented in functional or in imperative manner.

\textbf{locale} \texttt{Heap} = Collection empty is-empty of-list multiset for
\begin{itemize}
\item \texttt{empty} :: \texttt{'b and}
\item \texttt{is-empty} :: \texttt{'b \Rightarrow bool and}
\item \texttt{of-list} :: \texttt{'a::linorder list \Rightarrow \texttt{'b and}}
\item \texttt{multiset} :: \texttt{'b \Rightarrow \texttt{'a::linorder multiset +}}
\end{itemize}
\texttt{fixes} as-tree :: \texttt{'b \Rightarrow \texttt{'a::linorder Tree}}
   
   — This function is not very important, but it is needed in order to avoid problems with types and to detect that observed object is a tree.

\texttt{fixes} remove-max :: \texttt{'b \Rightarrow \texttt{'a \times 'b}}
\texttt{assumes} multiset: multiset \texttt{l} = \texttt{Heap\_multiset (as-tree \texttt{l})}
\texttt{assumes} is-heap-of-list: is-heap (as-tree (of-list \texttt{i}))
\texttt{assumes} as-tree-empty: as-tree \texttt{t} = \texttt{E \leftrightarrow is-empty \texttt{t}}
\texttt{assumes} remove-max-multiset:
\begin{itemize}
\item [\neg is-empty \texttt{l}; (\texttt{m, l'}) = remove-max \texttt{l}] \Rightarrow multiset \texttt{l' + \{\#m\#\} = multiset \texttt{l}}
\end{itemize}
\texttt{assumes} remove-max-is-heap:
\begin{itemize}
\item [\neg is-empty \texttt{l}; is-heap (as-tree \texttt{l}); (\texttt{m, l'}) = remove-max \texttt{l}] \Rightarrow
\item is-heap (as-tree \texttt{l'})
\end{itemize}
\texttt{assumes} remove-max-val:
\begin{itemize}
\item [\neg is-empty \texttt{t}; (\texttt{m, t'}) = remove-max \texttt{t}] \Rightarrow \texttt{m = val (as-tree \texttt{t})}
\end{itemize}

It is very easy to prove that locale \texttt{Heap} is sublocale of locale \texttt{RemoveMax}

\textbf{sublocale} \texttt{Heap < RemoveMax empty is-empty of-list multiset remove-max \lambda \texttt{t. is-heap (as-tree \texttt{t})}}
\texttt{proof}
\begin{itemize}
\item fix \texttt{x}
\item show is-heap (as-tree (of-list \texttt{x}))
\item by (rule is-heap-of-list)
\end{itemize}
\texttt{next}
\begin{itemize}
\item fix \texttt{l m l'}
\item assume \neg is-empty \texttt{l} (\texttt{m, l'}) = remove-max \texttt{l}
\item thus multiset \texttt{l' + \{\#m\#\} = multiset \texttt{l}}
\item by (rule remove-max-multiset')
\end{itemize}
\texttt{next}
\begin{itemize}
\item fix \texttt{l m l'}
\item assume \neg is-empty \texttt{l} is-heap (as-tree \texttt{l}) (\texttt{m, l'}) = remove-max \texttt{l}
\item thus is-heap (as-tree \texttt{l'})
\item by (rule remove-max-is-heap)
\end{itemize}
\texttt{next}
\begin{itemize}
\item fix \texttt{l m l'}
\item assume \neg is-empty \texttt{l} is-heap (as-tree \texttt{l}) (\texttt{m, l'}) = remove-max \texttt{l}
\item thus \texttt{m = Max (set \texttt{l})}
\item unfolding set-def
\item using heap-top-max[of as-tree \texttt{l}] remove-max-val[of \texttt{l m l'}]
\item using multiset is-empty-inj as-tree-empty
\item by auto
\end{itemize}
primrec in-tree where
  in-tree v E = False
| in-tree v (T v' l r) ←→ v = v' ∨ in-tree v l ∨ in-tree v r

lemma is-heap-max:
  assumes in-tree v t is-heap t
  shows val t ≥ v
using assms
apply (induct t rule:is-heap.induct)
by auto
end

6 Verification of Functional Heap Sort

theory HeapFunctional
imports Heap
begin

As we said before, maximum element of the heap is its root. So, finding
maximum element is not difficulty. But, this element should also be removed
and remainder after deleting this element is two trees, left and right branch
of original heap. Those branches are also heaps by the definition of the
heap. To maintain consistancy, branches should be combined into one tree
that satisfies heap condition:

function merge :: 'a::linorder Tree ⇒ 'a Tree ⇒ 'a Tree where
  merge t1 E = t1
| merge E t2 = t2
| merge (T v1 l1 r1) (T v2 l2 r2) =
  (if v1 ≥ v2 then T v1 (merge l1 (T v2 l2 r2)) r1
   else T v2 (merge l2 (T v1 l1 r1)) r2)
by (pat-completeness) auto
termination
proof (relation measure (λ (t1, t2). size t1 + size t2))
  fix v1 l1 r1 v2 l2 r2
  assume v2 ≤ v1
  thus ((l1, T v2 l2 r2), T v1 l1 r1, T v2 l2 r2) ∈
    measure (λ(t1, t2). Heap.size t1 + Heap.size t2)
    by auto
next
  fix v1 l1 r1 v2 l2 r2
  assume ¬ v2 ≤ v1
  thus ((l2, T v1 l1 r1), T v1 l1 r1, T v2 l2 r2) ∈
    measure (λ(t1, t2). Heap.size t1 + Heap.size t2)
    by auto
qed simp
lemma merge-val:
val(merge l r) = val l ∨ val(merge l r) = val r
proof(induct l r rule:merge.induct)
case (1 l)
  thus ?case
    by auto
next
case (2 r)
  thus ?case
    by auto
next
case (3 v1 l1 r1 v2 l2 r2)
  thus ?case
  proof(cases v2 ≤ v1)
    case True
    hence val (merge (T v1 l1 r1) (T v2 l2 r2)) = val (T v1 l1 r1)
      by auto
    thus ?thesis
      by auto
    next
    case False
    hence val (merge (T v1 l1 r1) (T v2 l2 r2)) = val (T v2 l2 r2)
      by auto
    thus ?thesis
      by auto
  qed
qed

Function merge merges two heaps into one:

lemma merge-heap-is-heap:
  assumes is-heap l is-heap r
  shows is-heap (merge l r)
using assms
proof(induct l r rule:merge.induct)
case (1 l)
  thus ?case
    by auto
next
case (2 r)
  thus ?case
    by auto
next
case (3 v1 l1 r1 v2 l2 r2)
  thus ?case
  proof(cases v2 ≤ v1)
    case True
    have is-heap l1
      using (is-heap (T v1 l1 r1):
by (metis Tree.exhaust is-heap.simps(1) is-heap.simps(4) is-heap.simps(5))
hence is-heap (merge l1 (T v2 l2 r2))
using True is-heap (T v2 l2 r2) : 3
by auto
have val (merge l1 (T v2 l2 r2)) = val l1 ∨ val (merge l1 (T v2 l2 r2)) = v2
using merge-val [af l1 T v2 l2 r2]
by auto
show ?thesis
proof (cases r1 = E)
case True
show ?thesis
proof (cases l1 = E)
case True
hence merge (T v1 l1 r1) (T v2 l2 r2) = T v1 (T v2 l2 r2) E
using (r1 = E) (v2 ≤ v1)
by auto
thus ?thesis
using 3
using (v2 ≤ v1)
by auto
next
case False
hence v1 ≥ val l1
using 3(3)
by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
thus ?thesis
using (r1 = E) (v1 ≥ v2)
using (val (merge l1 (T v2 l2 r2)) = val l1
\lor (val (merge l1 (T v2 l2 r2)) = v2)
using (is-heap (merge l1 (T v2 l2 r2)))
by (metis False Tree.exhaust is-heap.simps(2)
is-heap.simps(4) merge.simps(3) val.simps)
qed
next
case False
hence v1 ≥ val r1
using 3(3)
by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
show ?thesis
proof (cases l1 = E)
case True
hence merge (T v1 l1 r1) (T v2 l2 r2) = T v1 (T v2 l2 r2) r1
using (v2 ≤ v1)
by auto
thus ?thesis
using 3 (v1 ≥ val r1)
using (v2 ≤ v1)
by (metis False Tree.exhaust Tree.inject Tree.simps(3)
True is-heap.simps(3) is-heap.simps(6) merge.simps(2))
merge.simps(3) order-eq-iff val.simps

next
case False
hence \( v1 \geq \text{val} \ l1 \)
  using 3(3)
  by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
have \( \text{merge} \ l1 \ (T \ v2 \ l2 \ r2) \neq E \)
  using False
  by (metis Tree.exhaust Tree.simps(2) merge.simps(3))
have is-heap r1
  using 3(3)
  by (metis False Tree.exhaust \( \langle r1 \neq E \rangle \) is-heap.simps(5))
obtain l1 lr1 lv1 where \( r1 = T \ lv1 \ ll1 \ lr1 \)
  using \( \langle r1 \neq E \rangle \)
  by (metis Tree.exhaust)
  let \( r1 = T \ lv1 \ ll1 \ lr1 \)
  using \( \langle \text{is-heap} \ r1 \rangle \)
  using \( \langle \text{is-heap} \ (\text{merge} \ l1 \ (T \ v2 \ l2 \ r2)) \rangle , (v1 \geq \text{val} \ r1) \)
  by auto
hence is-heap (T v1 (merge l1 (T v2 l2 r2)) r1)
  using is-heap.simps(5)[of \( v1 \leq l1 \ ll1 \ lr1 \) \( r1 \neq E \) \( r1 \neq E \)]
  using \( \langle r1 \neq E \rangle \)
  using \( \langle \text{merge} \ l1 \ (T \ v2 \ l2 \ r2) = T \ rv1 \ rl1 \ rr1 \rangle \)
  using \( \langle \text{is-heap} \ r1 \rangle \)
  using \( \langle \text{is-heap} \ (\text{merge} \ l1 \ (T \ v2 \ l2 \ r2)) \rangle , (v1 \geq \text{val} \ r1) \)
  by auto
thus \?thesis
  using \( \langle v2 \leq v1 \rangle \)
  by auto
qed

qed

next
case False
have is-heap l2
  using 3(4)
  by (metis Tree.exhaust is-heap.simps(1)
    is-heap.simps(4) is-heap.simps(5))
hence is-heap (merge l2 (T v1 ll r1))
  using False \( \langle \text{is-heap} \ (T \ v1 \ ll \ r1) \rangle \)
    3
  by auto
have \( \text{val} \ (\text{merge} \ l2 \ (T \ v1 \ ll \ r1)) = \text{val} \ l2 \lor \text{val}(\text{merge} \ l2 \ (T \ v1 \ ll \ r1)) = v1 \)
  using merge-val[of \( l2 \ T \ v1 \ ll \ r1 \)]
  by auto
show \?thesis
proof(cases \( r2 = E \))
case True
show $\theta$thesis
proof (cases $l_2 = E$)
  case True
    hence merge ($T v_1 l_1 r_1$) ($T v_2 l_2 r_2$) = $T v_2$ ($T v_1 l_1 r_1$) $E$
    using ($r_2 = E$) ($\lnot v_2 \leq v_1$)
    by auto
  thus $\theta$thesis

next
  case False
  hence $v_2 \geq val l_2$
  using 3
  by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  thus $\theta$thesis

next
  case False
  hence $v_2 \geq val r_2$
  using 3
  by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  show $\theta$thesis
proof (cases $l_2 = E$)
  case True
    hence merge ($T v_1 l_1 r_1$) ($T v_2 l_2 r_2$) = $T v_2$ ($T v_1 l_1 r_1$) $r_2$
    using ($\lnot v_2 \leq v_1$)
    by auto
  thus $\theta$thesis

next
  case False
  hence $v_2 \geq val l_2$
  using 3
  by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  have merge $l_2$ ($T v_1 l_1 r_1$) $\neq E$
    using False
    by (metis Tree.exhaust Tree.simps(2) merge.simps(3))
  have is-heap $r_2$
    using 3(4)
by (metis False Tree.exhaust \(r_2 \neq E\) is-heap.simps(5))

obtain \(ll_1\) \(lr_1\) \(lv_1\) where \(r_2 = T\) \(lv_1\) \(ll_1\) \(lr_1\)
using \(r_2 \neq E\)
by (metis Tree.exhaust)

obtain \(r_1\) \(rr_1\) \(rv_1\) where \(\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\)) = \(T\) \(rv_1\) \(rr_1\)
using \(\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\)) \(\neq E\)
by (metis Tree.exhaust)

have \(\text{val}\) \((\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\))) \(\leq\) \(v_2\)
using \(\text{val}\) \((\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\))) = \(\text{val}\) \(l_2\) \(\lor\)
\(\text{val}\)\((\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\))) = \(v_1\)
by auto

hence is-heap \((T\) \(v_2\) \((\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\))) \(r_2\))
using is-heap.simps(5)[of \(v_1\) \(lv_1\) \(ll_1\) \(lr_1\) \(rv_1\) \(rl_1\) \(rr_1\)]
using is-heap \((T\) \(v_2\) \((\text{merge}\) \(l_2\) \((T\) \(v_1\) \(l_1\) \(r_1\))) \(\neq E\)) \(r_2\)
by auto

thus ?thesis
using \(\neg\) \(v_1\) \(\leq\) \(v_2\)
by auto

qed

primrec hs-of-list where
\hs-of-list [] = E
| \hs-of-list \((\text{v} \# \text{l})\) = insert \(\text{v}\) \((\text{hs-of-list}\) \text{l})

definition hs-is-empty where
\hs-is-empty \text{t} \longleftrightarrow \text{t} = E

Definition of function \text{remove-max}:

fun \hs-remove-max:: \('\text{a}\!::\text{linorder}\) Tree \Rightarrow 'a \times 'a Tree where
\hs-remove-max \((T\) \(v\) \(l\) \(r\)) = \((\text{v},\) \(\text{merge}\) \(l\) \(r\))

lemma merge-multiset:
\multiset \(l\) + \multiset \(g\) = \multiset \((\text{merge}\) \(l\) \(g\))

proof(induct \(l\) \(g\) rule:merge.induct)
case \((1\ell)\)
thus ?case
by auto
next
case \((2\ g)\)
thus ?case
by auto

20
next
case (3 v1 l1 r1 v2 l2 r2)
thus ?case
proof(cases v2 ≤ v1)
case True
hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
{#v1#} + multiset (merge l1 (T v2 l2 r2)) + multiset r1
by auto (metis union-commute)
hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
{#v1#} + multiset l1 + multiset (T v2 l2 r2) + multiset r1
using 3 True
by (metis union-commute union-assoc)
hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
{#v1#} + multiset l1 + multiset r1 + multiset (T v2 l2 r2)
by (metis union-commute union-lcomm)
thus ?thesis
by auto (metis union-commute)
next
case False
hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
{#v2#} + multiset (merge l2 (T v1 l1 r1)) + multiset r2
by auto (metis union-commute)
hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
{#v2#} + multiset l2 + multiset r2 + multiset (T v1 l1 r1)
using 3 False
by (metis union-commute union-lcomm)
thus ?thesis
by (metis multiset.simps(2) union-commute)
qed
qed

Proof that defined functions are interpretation of abstract functions from locale Collection:

interpretation HS: Collection E hs-is-empty hs-of-list multiset

proof
fix t
assume hs-is-empty t
thus t = E
by auto
next
show hs-is-empty E
by auto
next
show multiset E = {#}
by auto
next
fix l
show multiset (hs-of-list l) = mset l
proof(induct l)
Proof that defined functions are interpretation of abstract functions from locale Heap:

\textbf{interpretation} Heap $E$ hs-is-empty hs-of-list multiset id hs-remove-max

\textbf{proof}

\begin{itemize}
\item fix $l$
\item show $\text{multiset } l = \text{Heap.multiset } (\text{id } l)$
\item by auto
\end{itemize}

\textbf{next}

\begin{itemize}
\item fix $l$
\item show $\text{is-heap } (\text{id } (\text{hs-of-list } l))$
\item proof (induct $l$)
\item case Nil
\item thus $\hat{?}$case
\item by auto
\item next
\item case ($\text{Cons } a \ l$)
\item have $\text{hs-of-list } (a \ # \ l) = \text{merge } (\text{hs-of-list } l) (T a E E)$
\item apply auto
\item unfolding insert-def
\item by auto
\item have $\text{is-heap } (T a E E)$
\item by auto
\item hence $\text{is-heap } (\text{merge } (\text{hs-of-list } l) (T a E E))$
\item using $\text{Cons merge-is-heap } (\text{of } \text{hs-of-list } l T a E E)$
\item by auto
\item thus $\hat{?}$case
\item using $\text{hs-of-list } (a \ # \ l) = \text{merge } (\text{hs-of-list } l) (T a E E)$
\item by auto
\item qed
\item next
\item fix $t$
\item show $(\text{id } t = E) = \text{hs-is-empty } t$
\item by auto
\end{itemize}
next
fix $t \; m \; t'$
assume $\neg \text{hs-is-empty } t \left( m, \; t' \right) = \text{hs-remove-max} \; t$
then obtain $l \; r$ where $t = T \; m \; l \; r$
   by (metis Pair-inject Tree.exhaust hs-is-empty-def hs-remove-max.simps)
thus $\text{multiset } t' + \{ \#m\# \} = \text{multiset } t$
   using $\text{merge-multiset}[\text{of } l \; r]$
   using $(m, \; t') = \text{hs-remove-max} \; t$
   by (metis Pair-eq multiset.simps(2) hs-remove-max.simps
        union-assoc union-commute)

next
fix $t \; m \; t'$
assume $\neg \text{hs-is-empty } t \text{ is-heap } (id \; t) \left( m, \; t' \right) = \text{hs-remove-max} \; t$
then obtain $v \; l \; r$ where $t = T \; v \; l \; r$
   by (metis Tree.exhaust hs-is-empty-def)
hence $t' = \text{merge } l \; r$
   using $(m, \; t') = \text{hs-remove-max} \; t$
   by auto
have $\text{is-heap } l \land \text{is-heap } r$
   using $\text{is-heap } (id \; t) ;$
   using $(t = T \; v \; l \; r)$
   by (metis Tree.exhaust id-apply is-heap.simps(1)
        is-heap.simps(3) is-heap.simps(4) is-heap.simps(5))
thus $\text{is-heap } (id \; t')$
   using $(t' = \text{merge } l \; r)$
   using $\text{merge-heaps-is-heap}$
   by auto

next
fix $t \; m \; t'$
assume $\neg \text{hs-is-empty } t \left( m, \; t' \right) = \text{hs-remove-max} \; t$
thus $m = \text{val } (id \; t)$
   by (metis Pair-inject Tree.exhaust hs-is-empty-def
        hs-remove-max.simps id-apply val.simps)
qed
end

7 Verification of Imperative Heap Sort

theory HeapImperative
imports Heap
begin

primrec left :: 'a Tree ⇒ 'a Tree where
  left $(T \; v \; l \; r) = l$

abbreviation left-val :: 'a Tree ⇒ 'a where
  left-val $t ≡ \text{val } (left \; t)$
primrec right :: 'a Tree ⇒ 'a Tree where
geright (T v l r) = r

abbreviation right-val :: 'a Tree ⇒ 'a where
right-val t ≡ val (right t)

abbreviation set-val :: 'a Tree ⇒ 'a ⇒ 'a Tree where
set-val t x ≡ T x (left t) (right t)

The first step is to implement function siftDown. If some node does not satisfy heap property, this function moves it down the heap until it does. For a node is checked weather it satisfies heap property or not. If it does nothing is changed. If it does not, value of the root node becomes a value of the larger child and the value of that child becomes the value of the root node. This is the reason this function is called siftDown – value of the node is places down in the heap. Now, the problem is that the child node may not satisfy the heap property and that is the reason why function siftDown is recursively applied.

fun siftDown :: 'a::linorder Tree ⇒ 'a Tree where
siftDown E = E
| siftDown (T v E E) = T v E E
| siftDown (T v l E) =
  (if v ≥ val l then T v l E else T (val l) (siftDown (set-val l v)) E)
| siftDown (T v E r) =
  (if v ≥ val r then T v E r else T (val r) E (siftDown (set-val r v)))
| siftDown (T v l r) =
  (if val l ≥ val r then
    if v ≥ val l then T v l r else T (val l) (siftDown (set-val l v)) r
  else
    if v ≥ val r then T v l r else T (val r) l (siftDown (set-val r v)))

lemma siftDown-Node:
assumes t = T v l r
shows ∃ l' v' r'. siftDown t = T v' l' r' ∧ v' ≥ v
using assms
apply(induct t rule:siftDown.induct)
by auto

lemma siftDown-in-tree:
assumes t ≠ E
shows in-tree (val (siftDown t)) t
using assms
apply(induct t rule:siftDown.induct)
by auto

lemma siftDown-in-tree-set:
shows in-tree v t ←→ in-tree v (siftDown t)
proof
assume \( \text{in-tree } v \ t \)
thus \( \text{in-tree } v \ (\text{siftDown } t) \)
   apply (induct t rule:siftDown.induct)
   by auto
next
assume \( \text{in-tree } v \ (\text{siftDown } t) \)
thus \( \text{in-tree } v \ t \)
proof (induct t rule:siftDown.induct)
   case 1
   thus \(?case\)
   by auto
next
   case (2 v1)
   thus \(?case\)
   by auto
next
   case (3 v2 v1 l1 r1)
   show \(?case\)
   proof (cases v2 ≥ v1)
   case True
   thus \(?thesis\)
   using 3
   by auto
   next
   case False
   show \(?thesis\)
   proof (cases v1 = v)
   case True
   thus \(?thesis\)
   using 3 False
   by auto
   next
   case False
   hence \( \text{in-tree } v \ (\text{siftDown } \ (\text{set-val } (T \ v1 l1 r1) \ v2)) \)
   using (\( \neg \ v2 \geq v1 \) 3(2)
   by auto
   hence \( \text{in-tree } v \ (T \ v2 l1 r1) \)
   using 3(1) (\( \neg \ v2 \geq v1 \)
   by auto
   thus \(?thesis\)
   proof (cases v2 = v)
   case True
   thus \(?thesis\)
   by auto
next
   case False
   hence \( \text{in-tree } v \ (T \ v1 l1 r1) \)
   using (\( \text{in-tree } v \ (T \ v2 l1 r1) \))
   by auto
thus \(?thesis\) 
 by auto
qed
qed
qed
next
case (4 \(v2\ \ v1\ l1\ r1\) )
show \(?case\)
proof (cases \(v2\ \geq\ v1\))
case \(True\)
thus \(?thesis\)
 using 4
by auto
next
case \(False\)
show \(?thesis\)
proof (cases \(v1\ =\ v\) )
case \(True\)
thus \(?thesis\)
 using 4 \(False\)
by auto
next
case \(False\)
hence in-tree \(v\) \(\left(\text{siftDown}\ \left(\text{set-val}\ (T\ v1\ l1\ r1)\ v2\right)\right)\)
 using \(\left(\neg\ v2\ \geq\ v1\right)\ 4\(2)\)
by auto
hence in-tree \(v\) \(\left(T\ v2\ l1\ r1\right)\)
 using 4 \(1)\ \(\left(\neg\ v2\ \geq\ v1\right)\)
by auto
thus \(?thesis\)
proof (cases \(v2\ =\ v\) )
case \(True\)
thus \(?thesis\)
 by auto
next
case \(False\)
hence in-tree \(v\) \(\left(T\ v1\ l1\ r1\right)\)
 using \(\left(\text{in-tree v}\ \left(T\ v2\ l1\ r1\right)\right)\)
by auto
thus \(?thesis\)
 by auto
qed
qed
qed
next
case (5-1 \(v'\ \ v1'\ l1\ r1\ v2\ l2\ r2\) )
show \(?case\)
proof (cases \(v\ =\ v'\ \lor\ v\ =\ v1\ \lor\ v\ =\ v2\) )
case \(True\)
thus \( \text{thesis} \)
by auto
next

\textbf{case} False
\textbf{show} \( \text{thesis} \)
\textbf{proof}\((\text{cases } v' \geq v1)\)
\textbf{case} True
\textbf{thus} \( \text{thesis} \)
\textbf{using} \( \langle v1 \geq v2 \rangle \)
by auto

\textbf{next}
\textbf{case} False
\textbf{thus} \( \text{thesis} \)
\textbf{proof}\((\text{cases in-tree } v \ (T \ v2 \ l2 \ r2))\)
\textbf{case} True
\textbf{thus} \( \text{thesis} \)
by auto

\textbf{next}
\textbf{case} False
\textbf{hence} in-tree v \( (\text{siftDown} \ (\text{set-val} \ (T \ v1 \ l1 \ r1) \ v')) \)
\textbf{using} \( 5-1(3) \)
\textbf{by auto}

\textbf{hence} in-tree v \( (T \ v' \ l2 \ r1) \)
\textbf{using} \( 5-1(1) \)
\textbf{by auto}

\textbf{hence} in-tree v \( (T \ v2 \ l2 \ r2) \)
\textbf{by auto}

\textbf{hence} \( \langle \neg (v = v' \lor v = v1 \lor v = v2) \rangle \)
\textbf{by auto}
\textbf{thus} \( \text{thesis} \)
\textbf{by auto}
\textbf{qed}
\textbf{qed}

\textbf{next}
\textbf{case} False
\textbf{show} \( \text{thesis} \)
\textbf{proof}\((\text{cases } v' \geq v2)\)
\textbf{case} True
\textbf{thus} \( \text{thesis} \)
\textbf{using} \( \langle \neg v1 \geq v2 \rangle \)
\textbf{by auto}

\textbf{next}
\textbf{case} False
\textbf{thus} \( \text{thesis} \)
\textbf{proof}\((\text{cases in-tree } v \ (T \ v1 \ l1 \ r1))\)
\textbf{case} True
thus \textit{?thesis}
  by auto
next
case False
hence in-tree v (siftDown (set-val (T v2 l2 r2) v'))
  using 5-1(3) (\neg in-tree v (T v1 l1 r1)) (\neg v1 \geq v2) (\neg v' \geq v2)
  by auto
hence in-tree v (T v' l2 r2)
  using 5-1(2) (\neg v1 \geq v2) (\neg v' \geq v2)
  by auto
hence in-tree v (T v2 l2 r2)
  thus \textit{?thesis}
  by auto
qed
qed
qed
qed
next
case (5-2 v' v1 l1 r1 v2 l2 r2)
show \textit{?case}
proof(cases v = v' \lor v = v1 \lor v = v2)
  case True
  thus \textit{?thesis}
  by auto
next
case False
show \textit{?thesis}
proof(cases v1 \geq v2)
  case True
  show \textit{?thesis}
  proof(cases v' \geq v1)
    case True
    thus \textit{?thesis}
    using \langle v1 \geq v2 \rangle 5-2
    by auto
  next
case False
  thus \textit{?thesis}
  proof(cases in-tree v (T v2 l2 r2))
    case True
    thus \textit{?thesis}
    by auto
next
case False
hence in-tree v (siftDown (set-val (T v1 l1 r1) v'))
  using 5-2(3) (\neg in-tree v (T v2 l2 r2)) (\neg v1 \geq v2) (\neg v' \geq v1)
using \( \neg (v = v' \lor v = v \lor v = v2) \)
by auto

hence in-tree \( v (T v' l1 r1) \)
using \( 5-2(1) (v1 \geq v2) (\neg v' \geq v1) \)
by auto

hence in-tree \( v (T v1 l1 r1) \)
using \( (v = v' \lor v = v \lor v = v2) \)
by auto

thus \(?thesis\)
by auto

qed
qed

next

case False

show \(?thesis\)

proof(cases \( v' \geq v2 \))

case True

thus \(?thesis\)

using \( \neg v1 \geq v2 \) 5-2
by auto

next

case False

thus \(?thesis\)

proof(cases in-tree \( v (T v1 l1 r1) \))

case True

thus \(?thesis\)
by auto

next

case False

hence in-tree \( v (siftDown (set-val (T v2 l2 r2) v')) \)
using \( 5-2(3) (\neg in-tree v (T v1 l1 r1)) (\neg v1 \geq v2) (\neg v' \geq v2) \)
using \( (\neg (v = v' \lor v = v1 \lor v = v2)) \)
by auto

hence in-tree \( v (T v' l2 r2) \)
using \( 5-2(2) (\neg v1 \geq v2) (\neg v' \geq v2) \)
by auto

hence in-tree \( v (T v2 l2 r2) \)
using \( (v = v' \lor v = v1 \lor v = v2) \)
by auto

thus \(?thesis\)
by auto

qed
qed

qed

qed

lemma siftDown-heap-is-heap:
assumes \textit{is-heap} \(l\) \textit{is-heap} \(r\) \(t\) = \(T\) \(v\) \(l\) \(r\)

shows \textit{is-heap} \((\text{siftDown} \ t)\)

using \textit{assms}

proof (induct \(t\) arbitrary: \(v\) \(l\) \(r\) rule:\textit{siftDown.induct})
  case \(1\)
  thus \(?\)case
    by simp
  next
  case \((2 \ v')\)
  show \(?\)case
    by simp
  next
  case \((3 \ v2 \ v1 \ l1 \ r1)\)
  show \(?\)case
  proof (cases \(v2 \geq v1\))
    case True
    thus \(?\)thesis
      using \(3(2) \ 3(4)\)
      by auto
    next
    case False
    show \(?\)thesis
      proof (cases \(v' = v2\))
        case True
        thus \(?\)thesis
          using False \(\langle\ \text{is-heap} \ ?t \rangle \ast\)
          by auto
        next
        case False
        have in-tree \(v' \ ?t\)
          using *
          using siftDown-in-tree[of \(\ ?t\)\]
          by simp
  end
hence in-tree \( v' \) (\( T v2 l1 r1 \))
using siftDown-in-tree-set[symmetric, of \( v' \) \( T v2 l1 r1 \)]
by auto

hence in-tree \( v' \) (\( T v1 l1 r1 \))
using False
by simp

hence \( v1 \geq v' \)
using 3
using is-heap-max[of \( v' \) \( T v1 l1 r1 \)]
by auto

thus ?thesis
using (is-heap \( ?t \)) * (\( \neg v2 \geq v1 \))
by auto

qed

qed

qed

next
case (4 \( v2 v1 l1 r1 \))
show ?case
proof(cases \( v2 \geq v1 \))
case True
thus ?thesis
using 4(2-4)
by auto

next
case False
let \( ?t = \text{siftDown} (T v2 l1 r1) \)
obtain \( v' l' r' \) where *: ?t = \( T v' l' r' v' \geq v2 \)
using siftDown-Node[of \( T v2 l1 r1 v2 l1 r1 \)]
by auto

have \( r = T v1 l1 r1 \)
using 4(4)
by auto

hence is-heap \( l1 \) is-heap \( r1 \)
using 4(3)
apply (induct \( r \) rule:is-heap.induct)
by auto

hence is-heap \( ?t \)
using False 4(1)[of l1 r1 v2]
by auto

show ?thesis
proof(cases \( v' = v2 \))
case True
thus ?thesis
using * (is-heap \( ?t \)) False
by auto

next
case False
have in-tree \( v' \) \( ?t \)
using *
using siftDown-in-tree[of ?t]
by auto

hence in-tree v' (T v2 l1 r1)
  using * siftDown-in-tree-set[of v' T v2 l1 r1]
  by auto

hence in-tree v' (T v1 l1 r1)
  using False
  by auto

hence v1 ≥ v'
  using is-heap-max[of v' T v1 l1 r1] 4
  by auto

thus ?thesis
  using (is-heap ?t) False *
  by auto

qed

next

next case (5-1 v1 v2 l2 r2 v3 l3 r3)
show ?case

proof(cases v2 ≥ v3)
  case True
  show ?thesis
    proof(cases v1 ≥ v2)
      case True
        thus ?thesis
          using ⟨v2 ≥ v3⟩ 5-1
        by auto
    next
    case False
    let ?t = siftDown (T v1 l2 r2)
    obtain l' v' r' where *: ?t = T v' l' r' v' ≥ v1
      using siftDown-Node
      by blast
    have is-heap l2 is-heap r2
      using 5-1 (3, 5)
    apply(induct l rule:is-heap.induct)
    by auto

    hence is-heap ?t
      using 5-1 (1) [of l2 r2 v1] (v2 ≥ v3) False
      by auto
    have v2 ≥ v'
      proof(cases v' = v1)
        case True
        thus ?thesis
          using False
        by auto
      next
      case False
      32
have in-tree $v'$ ?t
  using * siftDown-in-tree
  by auto

hence in-tree $v'$ (T $v1$ l2 r2)
  using siftDown-in-tree-set[of $v'$ T $v1$ l2 r2]
  by auto

hence in-tree $v'$ (T $v2$ l2 r2)
  using False
  by auto

thus ?thesis
  using is-heap-max[of $v'$ T $v2$ l2 r2] 5-1
  by auto

qed
thus ?thesis
  using (is-heap ?t) ($v2 \geq v3$) * False 5-1
  by auto

qed

next

case False
show ?thesis
proof (cases $v1 \geq v3$)
  case True
  thus ?thesis
    using ($\neg v2 \geq v3$) 5-1
    by auto

next

case False
let ?t = siftDown (T $v1$ l3 r3)
obtain $l'$ $v'$ $r'$ where *: ?t = T $v'$ $l'$ $r'$ $v'$ \geq v1
  using siftDown-Node
  by blast
have is-heap l3 is-heap r3
  using 5-1(4, 5)
  apply (induct r rule: is-heap.induct)
  by auto

hence is-heap ?t
  using 5-1(2)[of l3 r3 v1] ($\neg v2 \geq v3$) False
  by auto

have $v3 \geq v'$
proof (cases $v' = v1$)
  case True
  thus ?thesis
    using False
    by auto

next

case False
have in-tree $v'$ ?t
  using * siftDown-in-tree
  by auto
hence in-tree \( v' \) \((T \ v1 \ l3 \ r3)\)
  using siftDown-in-tree-set[\( v' \) \( T \ v1 \ l3 \ r3 \)]
  by auto

hence in-tree \( v' \) \((T \ v3 \ l3 \ r3)\)
  using False
  by auto

thus ?thesis
  using is-heap-max[\( v' \) \( T \ v3 \ l3 \ r3 \)] 5-1
  by auto

qed

thus ?thesis
  using \( \langle \text{is-heap} \ ?t \rangle \ \langle \neg \ v2 \ge v3 \rangle \ * \ False \) 5-1
  by auto

qed

next
case \((5-2 \ v1 \ v2 \ l2 \ r2 \ v3 \ l3 \ r3)\)

show ?case

proof\((\text{cases} \ v2 \ge v3)\)

  case True

  show ?thesis
  proof\((\text{cases} \ v1 \ge v2)\)

    case True
    thus ?thesis
        using \( \langle v2 \ge v3 \rangle \) 5-2
        by auto

    next
case False

  let \(?t = \text{siftDown} \ (T \ v1 \ l2 \ r2)\)

  obtain \( l' \ v' \ r' \) where *: \(?t = T \ v' \ l' \ r' \ v1 \le v' \)

    using siftDown-Node
    by blast

  have is-heap \( l2 \) is-heap \( r2 \)
    using 5-2(3, 5)
    apply\((\text{induct} \ l \ \text{rule:is-heap.induct})\)
    by auto

  hence is-heap \(?t\)
    using 5-2(1)[of \( l2 \ r2 \ v1 \)] \( v2 \ge v3 \); False
    by auto

  have \( v2 \ge v' \)
  proof\((\text{cases} \ v' = v1)\)

    case True
    thus ?thesis
        using False
        by auto

    next
case False

  have in-tree \( v' \) \(?t\)
    using * siftDown-in-tree
by auto

hence in-tree \( v' (T v1 l2 r2) \)
  using siftDown-in-tree-set[of \( v' \) T v1 l2 r2]
  by auto

hence in-tree \( v' (T v2 l2 r2) \)
  using False
  by auto

thus \( \neg \)thesis
  using is-heap-max[of \( v' \) T v2 l2 r2] 5-2
  by auto

qed

thus \( \neg \)thesis
  using \( \langle \neg v2 \geq v3 \rangle \) * False 5-2
  by auto

qed

next
case False
show \( \neg \)thesis

proof(cases \( v1 \geq v3 \))
case True
  thus \( \neg \)thesis
    using \( \langle \neg v2 \geq v3 \rangle \) 5-2
    by auto

next
case False
let \( \neg t = \text{siftDown} (T v1 l3 r3) \)
obtain \( \langle \neg t = T v' l' r' v' \geq v1 \rangle \)
  using siftDown-Node
  by blast

have is-heap l3 is-heap r3
  using 5-2(4, 5)
  apply(induct r rule:is-heap.induct)
  by auto

hence is-heap \( \neg t \)
  using 5-2(2)[of l3 r3 v1] (\( \neg v2 \geq v3 \)) False
  by auto
have \( v3 \geq v' \)

proof(cases \( v' = v1 \))
case True
  using False
  by auto

next
case False
have in-tree \( v' \) \( \neg t \)
  using * siftDown-in-tree
  by auto

hence in-tree \( v' (T v1 l3 r3) \)
  using siftDown-in-tree-set[of \( v' \) T v1 l3 r3]
by auto
hence in-tree \( v' \) \((T \ v3 \ l3 \ r3)\)
using False
by auto
thus \( ?thesis \)
using is-heap-max[of \( v' \) \( T \ v3 \ l3 \ r3 \)] 5-2
by auto
qed
thus \( ?thesis \)
using \( \langle \text{is-heap} \ ?t \rangle \langle \neg \ v2 \geq \ v3 \rangle * \text{False} \) 5-2
by auto
qed
qed

Definition of the function \textit{heapify} which makes a heap from any given binary tree.

\textbf{primrec} \textit{heapify} \textbf{where}

\begin{align*}
\text{heapify} \ E & = E \\
\text{heapify} \ (T \ v \ l \ r) & = \text{siftDown} \ (T \ v \ (\text{heapify} \ l) \ (\text{heapify} \ r))
\end{align*}

\textbf{lemma} \textit{heapify-heap-is-heap}:

\text{is-heap} \ (\text{heapify} \ t)

\textbf{proof}(\text{induct} \ t)

\begin{enumerate}
\item \textbf{case} \( E \)
\item \textbf{thus} \( ?case \)
\item \textbf{by} auto
\end{enumerate}

\textbf{next}

\begin{enumerate}
\item \textbf{case} \( (T \ v \ l \ r) \)
\item \textbf{thus} \( ?case \)
\item \textbf{using} siftDown-heap-is-heap[of \( \text{heapify} \ l \ \text{heapify} \ r \) \( T \ v \ (\text{heapify} \ l) \ (\text{heapify} \ r) \) \( v \)]
\item \textbf{by} auto
\end{enumerate}

\textbf{qed}

Definition of \textit{removeLeaf} function. Function returns two values. The first one is the value of removed leaf element. The second returned value is tree without that leaf.

\textbf{fun} \textit{removeLeaf}:: \( 'a::\text{linorder} \ Tree \Rightarrow \ 'a \times 'a \ Tree \ where \)

\begin{align*}
\text{removeLeaf} \ (T \ v \ E \ E) & = (v, \ E) \\
\text{removeLeaf} \ (T \ v \ l \ E) & = (\text{fst} \ (\text{removeLeaf} \ l), \ T \ v \ (\text{snd} \ (\text{removeLeaf} \ l)) \ E) \\
\text{removeLeaf} \ (T \ v \ E \ r) & = (\text{fst} \ (\text{removeLeaf} \ r), \ T \ v \ E \ (\text{snd} \ (\text{removeLeaf} \ r))) \\
\text{removeLeaf} \ (T \ v \ l \ r) & = (\text{fst} \ (\text{removeLeaf} \ l), \ T \ v \ (\text{snd} \ (\text{removeLeaf} \ l)) \ r)
\end{align*}

Function \textit{of-list-tree} makes a binary tree from any given list.

\textbf{primrec} \textit{of-list-tree}:: \( 'a::\text{linorder} \ list \Rightarrow \ 'a \ Tree \ where \)

\begin{align*}
of-list-tree \ [] & = E \\
of-list-tree \ (v \ # \ \text{tail}) & = T \ v \ (\text{of-list-tree} \ \text{tail}) \ E
\end{align*}

By applying \textit{heapify} binary tree is transformed into heap.
**Definition hs-of-list where**

hs-of-list l = heapify (of-list-tree l)

Definition of function \textit{hs\_remove\_max}. As it is already well established, finding maximum is not a problem, since it is in the root element of the heap. The root element is replaced with leaf of the heap and that leaf is erased from its previous position. However, now the new root element may not satisfy heap property and that is the reason to apply function \textit{siftDown}.

**Definition hs\_remove\_max :: 'a\::linorder Tree \Rightarrow 'a \times 'a Tree where**

hs\_remove\_max t ≡ (let v' = fst (removeLeaf t);
                t' = snd (removeLeaf t) in
                (if t' = E then (val t, E)
                 else (val t, siftDown (set-val t' v'))))

**Definition hs\_is\_empty where**

\[\text{simp}]: hs\_is\_empty t \leftrightarrow t = E

**Lemma siftDown\_multiset:**

\[\text{multiset (siftDown t)} = \text{multiset t}\]

**Proof** (induct t rule: siftDown.induct)

1. **case 1**
   - thus ?case
     - by simp
2. **next**
   - **case (2 v)**
     - thus ?case
     - by simp
3. **next**
   - **case (3 v1 v l r)**
     - thus ?case
     - proof (cases v ≤ v1)
       - **case True**
         - thus ?thesis
           - by auto
     - **next**
       - **case False**
         - hence \[\text{multiset (siftDown (T v1 (T v l r) E))} = \text{multiset l} + \{#v1#\} + \text{multiset r} + \{#v#\}\]
         - using 3
         - by auto
     - **moreover**
       - have \[\text{multiset (T v1 (T v l r) E)} = \text{multiset l} + \{#v#\} + \text{multiset r} + \{#v1#\}\]
         - by auto
       - **moreover**
         - have \[\text{multiset l} + \{#v1#\} + \text{multiset r} + \{#v#\} = \text{multiset l} + \{#v#\} + \text{multiset r} + \{#v1#\}\]
         - by (metis union-commute union-lcomm)
ultimately
  show ?thesis
  by auto
qed

next
  case (4 v1 v l r)
  thus ?case
  proof (cases \(v \leq v1\))
    case True
    thus ?thesis
    by auto
  next
    case False
    have multisets (set-val (T v l r) v1) =
      multisets l + \(#v1\#\) + multisets r
    by auto
    hence multisets (siftDown (T v1 E (T v l r))) =
      \(#v\#\) + multisets (set-val (T v l r) v1)
    using 4 False
    by auto
    hence multisets (siftDown (T v1 E (T v l r))) =
      \(#v\#\) + multisets l + \(#v1\#\) + multisets r
    using multisets (set-val (T v l r) v1) =
    multisets l + \(#v1\#\) + multisets r
    by (metis union-commute union-lcomm)
  moreover
  have multisets (T v1 E (T v l r)) =
    \(#v1\#\) + multisets l + \(#v\#\) + multisets r
  by (metis calculation monoid-add-class.add.left-neutral
       multisets.simps(1) multisets.simps(2) union-commute union-lcomm)
  moreover
  have \(#v\#\) + multisets l + \(#v1\#\) + multisets r =
    \(#v1\#\) + multisets l + \(#v\#\) + multisets r
  by (metis union-commute union-lcomm)
  ultimately
  show ?thesis
  by auto
qed

next
  case (5-1 v v1 l1 r1 v2 l2 r2)
  thus ?case
  proof (cases \(v1 \geq v2\))
    case True
    thus ?thesis
    proof (cases \(v \geq v1\))
      case True
      thus ?thesis
      using \(v1 \geq v2\)
      by auto
next

\textbf{case} \texttt{False}

\textbf{hence} \quad \text{multiset} \left( \text{siftDown} \left( \text{T} \ v \ \left( \text{T} \ v_1 \ l_1 \ r_1 \right) \left( \text{T} \ v_2 \ l_2 \ r_2 \right) \right) \right) =

\quad \text{multiset} \ l_1 + \{#v\#\} + \text{multiset} \ r_1 + \{#v1\#\} + 

\quad \text{multiset} \left( \text{T} \ v_2 \ l_2 \ r_2 \right)

\textbf{using} \ \langle \neg \ v1 \geq v2 \rangle \ 5-1(1)

\textbf{by} \ \texttt{auto}

\textbf{moreover}

\textbf{have} \quad \text{multiset} \left( \text{T} \ v \ \left( \text{T} \ v_1 \ l_1 \ r_1 \right) \left( \text{T} \ v_2 \ l_2 \ r_2 \right) \right) =

\quad \text{multiset} \ l_1 + \{#v1\#\} + \text{multiset} \ r_1 + \{#v\#\} + 

\quad \text{multiset} \left( \text{T} \ v_2 \ l_2 \ r_2 \right)

\textbf{by} \ \texttt{auto}

\textbf{moreover}

\textbf{have} \quad \text{multiset} \ l_1 + \{#v1\#\} + \text{multiset} \ r_1 + \{#v\#\} + 

\quad \text{multiset}(T v_2 l_2 r_2) =

\quad \text{multiset} \ l_1 + \{#v\#\} + \text{multiset} \ r_1 + \{#v1\#\} + 

\quad \text{multiset} \left( \text{T} \ v_2 \ l_2 \ r_2 \right)

\textbf{by} \ (\text{metis union-commute union-lcomm})

\textbf{ultimately}

\textbf{show} \ \langle \neg \ v1 \geq v2 \rangle\ \textbf{by} \ \texttt{auto}

\textbf{qed}

next

\textbf{case} \texttt{False}

\textbf{show} \ \langle \neg \ v1 \geq v2 \rangle

\textbf{proof} (cases \ \langle \neg \ v1 \geq v2 \rangle)

\textbf{case} \texttt{True}

\textbf{thus} \ \langle \neg \ v1 \geq v2 \rangle

\textbf{using} \ \langle \neg \ v1 \geq v2 \rangle \ 5-1(2)

\textbf{by} \ (\text{simp add: ac-simps})

\textbf{moreover}

\textbf{have} \quad \text{multiset} \left( \text{T} \ v \ \left( \text{T} \ v_1 \ l_1 \ r_1 \right) \left( \text{T} \ v_2 \ l_2 \ r_2 \right) \right) =

\quad \text{multiset} \ l_2 + \{#v2\#\} + \text{multiset} r_2

\textbf{by} \ (\text{metis hide-lams, no-types multiset.simps(2)

\quad \text{union-assoc union-commute union-lcomm})

\textbf{moreover}

\textbf{have} \quad \text{multiset} \left( \text{T} \ v \ \left( \text{T} \ v_1 \ l_1 \ r_1 \right) \left( \text{T} \ v_2 \ l_2 \ r_2 \right) \right) =

\quad \text{multiset} \ l_2 + \{#v2\#\} + \text{multiset} r_2

\textbf{by} \ (\text{metis hide-lams, no-types multiset.simps(2)

\quad \text{union-assoc union-commute union-lcomm})
{#v#} + multiset r2
by (metis union-commute union-lcomm)
ultimately
show ?thesis
by auto
qed
qed
next
case (5-2 v v1 l1 r1 v2 l2 r2)
thus ?case
proof (cases v1 ≥ v2)
case True
thus ?thesis
proof (cases v ≥ v1)
case True
thus ?thesis
using ⟨v1 ≥ v2⟩
by auto
next
case False
hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
  multiset l1 + {#v#} + multiset r1 + {#v1#} +
  multiset (T v2 l2 r2)
using ⟨v1 ≥ v2; 5-2(1)⟩
by auto
moreover
have multiset (T v (T v1 l1 r1) (T v2 l2 r2)) =
  multiset l1 + {#v1#} + multiset r1 +
  {#v#} + multiset (T v2 l2 r2)
by auto
moreover
have multiset l1 + {#v1#} + multiset r1 + {#v#} +
  multiset (T v2 l2 r2) =
  multiset l1 + {#v#} + multiset r1 + {#v1#} +
  multiset (T v2 l2 r2)
by (metis union-commute union-lcomm)
ultimately
show ?thesis
by auto
qed
next
case False
show ?thesis
proof (cases v ≥ v2)
case True
thus ?thesis
using False
by auto
next
case False
hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
    multiset (T v1 l1 r1) + {#v2#} + multiset l2 + {#v#} +
    multiset r2
using (¬ v1 ≥ v2) 5-2(2)
by (simp add: ac-simps)
moreover
have multiset (T v1 l1 r1) (T v2 l2 r2)) =
    multiset (T v1 l1 r1) + {#v#} + multiset l2 + {#v2#} +
    multiset r2
by (metis (hide-lams, no-types) multiset.simps(2)
    union-assoc union-commute union-lcomm)
moreover
have multiset (T v1 l1 r1) + {#v#} + multiset l2 + {#v2#} +
    multiset r2 =
    multiset (T v1 l1 r1) + {#v2#} + multiset l2 + {#v#} +
    multiset r2
by (metis union-commute union-lcomm)
ultimately
show ?thesis
by auto
qed
qed

lemma mset-list-tree:
multiset (of-list-tree l) = mset l
proof (induct l)
case Nil
  thus ?case
  by auto
next
case (Cons v tail)
hence multiset (of-list-tree (v # tail)) = mset tail + {#v#}
  by auto
also have ... = mset (v # tail)
  by auto
finally show multiset (of-list-tree (v # tail)) = mset (v # tail)
  by auto
qed

lemma multiset-heapify:
multiset (heapify t) = multiset t
proof (induct t)
case E
  thus ?case
  by auto
next

41
case (T v l r)
hence multiset (heapify (T v l r)) = multiset l + {#v#} + multiset r
  using siftDown-multiset[of T v (heapify l) (heapify r)]
  by auto
thus ?case
  by auto
qed

lemma multiset-heapify-of-list-tree:
multiset (heapify (of-list-tree l)) = mset l
using multiset-heapify[of of-list-tree l]
using mset-list-tree[of l]
by auto

lemma removeLeaf-val-val:
  assumes snd (removeLeaf t) ≠ E t ≠ E
  shows val t = val (snd (removeLeaf t))
using assms
apply (induct t rule:removeLeaf.induct)
by auto

lemma removeLeaf-heap-is-heap:
  assumes is-heap t t ≠ E
  shows is-heap (snd (removeLeaf t))
using assms
proof (induct t rule:removeLeaf.induct)
case (1 v)
  thus ?case
    by auto
next
case (2 v v1 l1 r1)
  have is-heap (T v1 l1 r1)
    using 2(3)
    by auto
  hence is-heap (snd (removeLeaf (T v1 l1 r1)))
    using 2(1)
    by auto
  let ?t = (snd (removeLeaf (T v1 l1 r1)))
  show ?case
proof (cases ?t = E)
case True
  thus ?thesis
    by auto
next
case False
  have v ≥ v1
    using 2(3)
    by auto
hence \( v \geq \text{val} \ ?t \)
  using False removeLeaf-val-val[of T v l1 r1]
  by auto
hence \( \text{is-heap} \ (T \ v \ (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1)))) \ E \)
  using (\text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1))))
  by (metis Tree.exhaust is-heap.simps(2) is-heap.simps(4))
thus \?thesis
  using 2
  by auto
qed

next

  case (3 v v1 l1 r1)
  have \( \text{is-heap} \ (T \ v1 l1 r1) \)
    using 3(3)
    by auto
  hence \( \text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1))) \)
    using 3(1)
    by auto
  let \( \ ?t = (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1))) \)
  show \?case
    proof (cases \?t = E)
      case True
      thus \?thesis
        by auto
    next
      case False
      have \( v \geq v1 \)
        using 3(3)
        by auto
      hence \( v \geq \text{val} \ ?t \)
        using False removeLeaf-val-val[of T v l1 r1]
        by auto
      hence \( \text{is-heap} \ (T \ v \ E \ (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1)))) \)
        using (\text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1))))
        by (metis False Tree.exhaust is-heap.simps(3))
      thus \?thesis
        using 3
        by auto
    qed
  next
    case (4-1 v v1 l1 r1 v2 l2 r2)
    have \( \text{is-heap} \ (T \ v1 l1 r1) \ \text{is-heap} \ (T \ v2 l2 r2) \ v \geq v1 \ v \geq v2 \)
      using 4-1(3)
      by (simp add:is-heap.simps(5))+
    hence \( \text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 l1 r1))) \)
      using 4-1(1)
      by auto
    let \( \ ?t = (\text{snd} \ (\text{removeLeaf} \ (T \ v l1 r1))) \)
    show \?case

43
proof (cases \( \bar{t} = E \))
  case True
  thus \(?thesis\)
    using (is-heap \((T v2 l2 r2)\); \(v \geq v2\))
    by auto
next
  case False
  then obtain \(v1' l1' r1'\) where \(\bar{t} = T v1' l1' r1'\)
    by (metis Tree.exhaust)
  hence is-heap \((T v1' l1' r1')\)
    using (is-heap (snd (removeLeaf \((T v1 l1 r1)\))))
    by auto
  have \(v \geq v1\)
    using 4-1(3)
    by auto
  hence \(v \geq v1\)
    using \(\bar{t} = T v1' l1' r1'\)
    by auto
  hence is-heap \((T v (T v1' l1' r1') (T v2 l2 r2))\)
    using (simp add: is-heap.simps(5))
  thus \(?thesis\)
    using 4-1 \(\bar{t} = T v1' l1' r1'\)
    by auto
qed
next
  case (4-2 v v1 l1 r1 v2 l2 r2)
  have is-heap \((T v1 l1 r1)\) is-heap \((T v2 l2 r2)\) \(v \geq v1\) \(v \geq v2\)
    using 4-2(3)
    by (simp add:is-heap.simps(5))+
  hence is-heap (snd (removeLeaf \((T v1 l1 r1)\)))
    using 4-2(1)
    by auto
  let \(?t = (snd (removeLeaf \((T v1 l1 r1)\)))\)
  show \(?case\)
proof (cases \(?t = E\))
  case True
  thus \(?thesis\)
    using (is-heap \((T v2 l2 r2)\); \(v \geq v2\))
    by auto
next
  case False
  then obtain \(v1' l1' r1'\) where \(\bar{t} = T v1' l1' r1'\)
    by (metis Tree.exhaust)
  hence is-heap \((T v1' l1' r1')\)
using (is-heap (snd (removeLeaf (T v1 l1 r1))))
by auto
have v ≥ v1
using 4-2(3)
by auto
hence v ≥ val ?t
using False removeLeaf-val-val[of T v1 l1 r1]
by auto
hence v ≥ v1'
using (?t = T v1' l1' r1')
by auto
hence is-heap (T v (T v1' l1' r1') (T v2 l2 r2))
using (is-heap (T v1' l1' r1'))
using (is-heap (T v2 l2 r2)) (v ≥ v2)
by (simp add: is-heap.simps(5))
thus ?thesis
using 4-2 (?t = T v1' l1' r1')
by auto
qed
next
case 5
thus ?case
by auto
qed

Defined functions satisfy conditions of locale Collection and thus represent interpretation of this locale.

interpretation HS: Collection E hs-is-empty hs-of-list multiset

proof
fix t
assume hs-is-empty t
thus t = E
by auto
next
show hs-is-empty E
by auto
next
show multiset E = {#}
by auto
next
fix l
show multiset (hs-of-list l) = mset l
unfolding hs-of-list-def
using multiset-heapify-of-list-tree[of l]
by auto
qed

lemma removeLeaf-multiset:
assumes (v', t') = removeLeaf t t ≠ E
shows \{\#v'\#\} + multiset t' = multiset t
using assms

proof (induct t arbitrary: v' t' rule: removeLeaf.induct)
case 1
thus ?case
  by auto

next
case (2 v v1 l1 r1)
have t' = T v (snd (removeLeaf (T v1 l1 r1))) E
  using 2(3)
  by auto
have v' = fst (removeLeaf (T v1 l1 r1))
  using 2(3)
  by auto
hence \{\#v'\#\} + multiset t' =
  \{\#fst (removeLeaf (T v1 l1 r1))\#\} +
  multiset (snd (removeLeaf (T v1 l1 r1))) +
  \{\#v\#\}
  using (t' = T v (snd (removeLeaf (T v1 l1 r1))) E):
by (simp add: ac-simps)
have \{\#fst (removeLeaf (T v1 l1 r1))\#\} +
  multiset (snd (removeLeaf (T v1 l1 r1))) =
  multiset (T v1 l1 r1)
  using 2(1)
  by auto
hence \{\#v'\#\} + multiset t' = multiset (T v1 l1 r1) + \{\#v\#\}
  using \{\#v'\#\} + multiset t' =
  \{\#fst (removeLeaf (T v1 l1 r1))\#\} +
  multiset (snd (removeLeaf (T v1 l1 r1))) + \{\#v\#\};
by auto
thus ?case
  by auto

next
case (3 v v1 l1 r1)
have t' = T v E (snd (removeLeaf (T v1 l1 r1)))
  using 3(3)
  by auto
have v' = fst (removeLeaf (T v1 l1 r1))
  using 3(3)
  by auto
hence \{\#v'\#\} + multiset t' =
  \{\#fst (removeLeaf (T v1 l1 r1))\#\} +
  multiset (snd (removeLeaf (T v1 l1 r1))) +
  \{\#v\#\}
  using (t' = T v E (snd (removeLeaf (T v1 l1 r1)))):
by (simp add: ac-simps)
have \{\#fst (removeLeaf (T v1 l1 r1))\#\} +
  multiset (snd (removeLeaf (T v1 l1 r1))) =
  multiset (T v1 l1 r1)
using 3(1)
by auto

hence \( \{v'\} + \text{multiset } t' = \text{multiset } (T v1 l1 r1) + \{v\} \)
using \( \{v'\} + \text{multiset } t' = \)
\( \{\text{fst } (\text{removeLeaf } (T v1 l1 r1))\} + \)
\( \text{multiset } (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) + \{v\} \)
by auto

thus \?case
by \( \text{metis monoid-add-class.add_right-neutral} \)
\( \text{multiset.simps(1) multiset.simps(2) union-commute} \)

next
case \( 4-1 \ v v1 l1 r1 v2 l2 r2 \)
have \( t' = T v (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) (T v2 l2 r2) \)
using \( 4-1(3) \)
by auto
have \( v' = \text{fst } (\text{removeLeaf } (T v1 l1 r1)) \)
using \( 4-1(3) \)
by auto
hence \( \{v'\} + \text{multiset } t' = \)
\( \{\text{fst } (\text{removeLeaf } (T v1 l1 r1))\} + \)
\( \text{multiset } (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) + \{v\} \)
using \( t' = T v (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) (T v2 l2 r2) \)
by \( \text{metis multiset.simps(2) union-assoc} \)
have \( \{\text{fst } (\text{removeLeaf } (T v1 l1 r1))\} + \)
\( \text{multiset } (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) = \)
\( \text{multiset } (T v1 l1 r1) \)
using \( 4-1(1) \)
by auto

hence \( \{v'\} + \text{multiset } t' = \)
\( \text{multiset } (T v1 l1 r1) + \{v\} + \text{multiset } (T v2 l2 r2) \)
using \( \{v'\} + \text{multiset } t' = \)
\( \{\text{fst } (\text{removeLeaf } (T v1 l1 r1))\} + \)
\( \text{multiset } (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) + \{v\} \)
+ \text{multiset } (T v2 l2 r2) \)
by auto
thus \?case
by auto

next
case \( 4-2 \ v v1 l1 r1 v2 l2 r2 \)
have \( t' = T v (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) (T v2 l2 r2) \)
using \( 4-2(3) \)
by auto
have \( v' = \text{fst } (\text{removeLeaf } (T v1 l1 r1)) \)
using \( 4-2(3) \)
by auto
hence \( \{v'\} + \text{multiset } t' = \)
\( \{\text{fst } (\text{removeLeaf } (T v1 l1 r1))\} + \)
\( \text{multiset } (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) + \)
\{\#v\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2) \\
\text{using } \langle t' = T \ v \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))) \ (T \ v2 \ l2 \ r2) \rangle \\
\text{by } \langle \text{metis multiset.simps(2) union-assoc} \rangle \\
\text{have } \{\#\text{fst} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))\#\} + \\
\text{multiset} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))) = \\
\text{multiset} \ (T \ v1 \ l1 \ r1) \\
\text{using } 4-2(1) \\
\text{by } \text{auto} \\
\text{hence } \{\#v'\#\} + \text{multiset} \ t' = \\
\text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2) \\
\text{using } \langle \{\#v'\#\} + \text{multiset} \ t' = \\
\{\#\text{fst} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))\#\} + \\
\text{multiset} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))) + \\
\{\#v\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2) \rangle \\
\text{by } \text{auto} \\
\text{thus } \text{?case} \\
\text{by } \text{auto} \\
\text{next} \\
\text{case 5} \\
\text{thus } \text{?case} \\
\text{by } \text{auto} \\
\text{qed} \\

\text{lemma } \text{set-val-multiset}: \\
\text{assumes } t \neq E \\
\text{shows } \text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \{\#v'\#\} + \text{multiset} \ t \\
\text{proof} - \\
\text{obtain } v \ l \ r \ \text{where } t = T \ v \ l \ r \\
\text{using } \text{assms} \\
\text{by } \langle \text{metis Tree.exhaust} \rangle \\
\text{hence } \text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \\
\text{by } \text{auto} \\
\text{have } \{\#v'\#\} + \text{multiset} \ t = \\
\{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r \\
\text{using } \langle t = T \ v \ l \ r \rangle \\
\text{by } \langle \text{metis multiset.simps(2) union-assoc} \rangle \\
\text{have } \{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \\
\text{by } \langle \text{metis union-commute union-lcomm} \rangle \\
\text{thus } \text{?thesis} \\
\text{using } \langle \text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \rangle \\
\text{using } \langle \{\#v'\#\} + \text{multiset} \ t = \\
\{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r \rangle \\
\text{by } \text{auto} \\
\text{qed} \\

\text{lemma } \text{hs-remove-max-multiset}: \\
\text{proof} - \\
\text{obtain } v \ l \ r \ \text{where } t = T \ v \ l \ r \\
\text{using } \text{assms} \\
\text{by } \langle \text{metis Tree.exhaust} \rangle \\
\text{hence } \text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \\
\text{by } \text{auto} \\
\text{have } \{\#v'\#\} + \text{multiset} \ t = \\
\{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r \\
\text{using } \langle t = T \ v \ l \ r \rangle \\
\text{by } \langle \text{metis multiset.simps(2) union-assoc} \rangle \\
\text{have } \{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \\
\text{by } \langle \text{metis union-commute union-lcomm} \rangle \\
\text{thus } \text{?thesis} \\
\text{using } \langle \text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \\
\text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\} \rangle \\
\text{using } \langle \{\#v'\#\} + \text{multiset} \ t = \\
\{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r \rangle \\
\text{by } \text{auto} \\
\text{qed}
assumes \((m, t') = \mathit{hs-remove-max} t t \neq E\)
shows \(\{m\#\} + \mathit{multiset} t' = \mathit{multiset} t\)

**proof**
- let \(?v1 = \mathit{fst} (\mathit{removeLeaf} t)\)
- let \(?t1 = \mathit{snd} (\mathit{removeLeaf} t)\)
- show \(?\mathit{thesis}\)
  **proof** (cases \(?t1 = E\))
  - case \(\mathit{True}\)
    hence \(\{m\#\} + \mathit{multiset} t' = \{m\#\}\)
      using assms
      unfolding \(\mathit{hs-remove-max-def}\)
      by auto
    have \(?v1 = \mathit{val} t\)
      using \(\mathit{True} \ \mathit{assms}(2)\)
      apply (induct \(t\) rule: removeLeaf.induct)
      by auto
    hence \(?v1 = m\)
      using \(\mathit{assms}(1) \ \mathit{True}\)
      unfolding \(\mathit{hs-remove-max-def}\)
      by auto
    hence \(\mathit{multiset} t = \{m\#\}\)
      using \(\mathit{removeLeaf-multiset}[of \ ?v1 \ ?t1 \ t] \ \mathit{True} \ \mathit{assms}(2)\)
      by (metis \(\mathit{empty-neutral}(2)\) \multiset.simps(1) pair-collapse)
    thus \(?\mathit{thesis}\)
      using \(\{m\#\} + \mathit{multiset} t' = \{m\#\}\)
      by auto
  - next
    case \(\mathit{False}\)
    hence \(t' = \mathit{siftDown} (\mathit{set-val} \ ?t1 \ ?v1)\)
      using \(\mathit{assms}(1)\)
      by (auto simp add: \(\mathit{hs-remove-max-def}\) (metis \(\mathit{prod.inject}\))
    hence \(\mathit{multiset} t' + \{\mathit{val} ?t1\#\} = \mathit{multiset} t\)
      using \(\mathit{siftDown-multiset}[of \ \mathit{set-val} ?t1 \ ?v1]\)
      using \(\mathit{set-val-multiset}[of \ ?t1 \ ?v1] \ False\)
      using \(\mathit{removeLeaf-multiset}[of \ ?v1 \ ?t1 \ t] \ \mathit{assms}(2)\)
      by auto
    have \(\mathit{val} ?t1 = \mathit{val} t\)
      using \(\mathit{False} \ \mathit{assms}(2)\)
      apply (induct \(t\) rule: removeLeaf.induct)
      by auto
    have \(\mathit{val} t = m\)
      using \(\mathit{assms}(1) \ \mathit{False}\)
      using \(t' = \mathit{siftDown} (\mathit{set-val} \ ?t1 \ ?v1)\)
      unfolding \(\mathit{hs-remove-max-def}\)
      by (metis (full-types) \mathit{fst-conv} removeLeaf.simps(1))
    hence \(\mathit{val} ?t1 = m\)
      using \(\mathit{val} ?t1 = \mathit{val} t\)
      by auto
    hence \(\mathit{multiset} t' + \{m\#\} = \mathit{multiset} t\)
using (multiset t' + {#val ?t1 #} = multiset t)
by metis
thus ?thesis
by (metis union-commute)
qed

Difined functions satisfy conditions of locale Heap and thus represent interpretation of this locale.

interpretation Heap E hs-is-empty hs-of-list multiset id hs-remove-max
proof
fix t
show multiset t = multiset (id t)
by auto
next
fix t
show is-heap (id (hs-of-list t))
unfolding hs-of-list-def
using heapify-heap-is-heap[of of-list-tree t]
by auto
next
fix t
show (id t = E) = hs-is-empty t
by auto
next
fix t m t'
assume ¬ hs-is-empty t (m, t') = hs-remove-max t
thus multiset t' + {#m#} = multiset t
using hs-remove-max-multiset[of m t' t]
by (auto, metis union-commute)
next
fix t v t'
assume ¬ hs-is-empty t is-heap (id t) (v', t') = hs-remove-max t
let ?v1 = fst (removeLeaf t)
let ?t1 = snd (removeLeaf t)
have is-heap ?t1
using (¬ hs-is-empty t) (is-heap (id t))
using removeLeaf-heap-is-heap[of]
by auto
show is-heap (id t')
proof(cases ?t1 = E)
case True
hence t' = E
using ((v', t') = hs-remove-max t)
unfolding hs-remove-max-def
by auto
thus ?thesis
by auto
next
case False
then obtain v-t1 l-t1 r-t1 where ?t1 = T v-t1 l-t1 r-t1
by (metis Tree.exhaust)
hence is-heap l-t1 is-heap r-t1
using (is-heap ?t1)
by (auto, metis (full-types) Tree.exhaust
is-heap.simps(1) is-heap.simps(4) is-heap.simps(5))
(metais (full-types) Tree.exhaust
is-heap.simps(1) is-heap.simps(3) is-heap.simps(5))

have set-val ?t1 ?v1 = T ?v1 l-t1 r-t1
using (?t1 = T v-t1 l-t1 r-t1)
by auto
hence is-heap (siftDown (set-val ?t1 ?v1))
using (is-heap l-t1) is-heap r-t1)
using siftDown-heap-is-heap[of l-t1 r-t1 set-val ?t1 ?v1 ?v1]
by auto

have t' = siftDown (set-val ?t1 ?v1)
using ((v', t') = hs-remove-max t False
by (auto simp add: hs-remove-max-def) (metis prod.inject)
thus ?thesis
using (is-heap (siftDown (set-val ?t1 ?v1)));
by auto
qed

next

fix t m t'
let ?t1 = snd (removeLeaf t)
assume ¬ hs-is-empty t (m, t') = hs-remove-max t
hence m = val t
apply (simp add: hs-remove-max-def)
apply (cases ?t1 = E)
by (auto, metis prod.inject)
thus m = val (id t)
by auto
qed

end

8 Related work

To study sorting algorithms from a top down was proposed in [?]. All
sorting algorithms are based on divide-and-conquer algorithm and all sorts
are divided into two groups: hard_split/easy_join and easy_split/hard_join.
Fallowing this idea in [?], authors described sorting algorithms using object-
oriented approach. They suggested that this approach could be used in
computer science education and that presenting sorting algorithms from top
down will help students to understand them better.

The paper [?] represent different recursion patterns — catamorphism, anamor-
phism, hylomorphism and paramorphisms. Selection, bubble, merge, heap
and quick sort are expressed using these patterns of recursion and it is shown
that there is a little freedom left in implementation level. Also, connection
between different patterns are given and thus a conclusion about connection
between sorting algorithms can be easily conducted. Furthermore, in the
paper are generalized tree data types – list, binary trees and binary leaf
trees.

Satisfiability procedures for working with arrays was proposed in paper
“What is decidable about arrays?”[?]. This procedure is called $SAT_A$ and
can give an answer if two arrays are equal or if array is sorted and so on.
Completeness and soundness for procedures are proved. There are, though,
several cases when procedures are unsatisfiable. They also studied theory
of maps. One of the application for these procedures is verification of sort-
ing algorithms and they gave an example that insertion sort returns sorted
array.

Tools for program verification are developed by different groups and with
different results. Some of them are automated and some are half-automated.
Ralph-Johan Back and Johannes Eriksson [?] developed SOCOS, tool for
program verification based on invariant diagrams. SOCOS environment
supports interactive and non-interactive checking of program correctness.
For each program tree types of verification conditions are generated: consist-
tency, completeness and termination conditions. They described invariant-
based programming in SOCOS. In [?] this tool was used to verify heap sort
algorithm.

There are many tools for Java program developers maid to automatically
prove program correctness. Krakatoa Modeling Language (KML) is de-
scribed in [?] with example of sorting algorithms. Refinement is not sup-
ported in KML and any refinement property could not automatically be
proved. The language KML is also not formally verified, but some parts are
proved by Alt-Ergo, Simplify and Yices. The paper proposed some improve-
ments for working with permutation and arrays in KML. Why/Krakatoa/Caduceus[?]
is a tool for deductive program verification for Java and C. The approach
is to use Krakatoa and Caduceus to translate Java/C programs into Why
program. This language is suitable for program verification. The idea is to
generate verification conditions based on weakest precondition calculus.
9 Conclusions and Further Work

In this paper we illustrated a proof management technology. The methodology that we use in this paper for the formalization is refinement: the formalization begins with a most basic specification, which is then refined by introducing more advanced techniques, while preserving the correctness. This incremental approach proves to be a very natural approach in formalizing complex software systems. It simplifies understanding of the system and reduces the overall verification effort.

Modularity is very popular in nowadays imperative languages. This approach could be used for software verification and Isabelle/HOL locales provide means for modular reasoning. They support multiple inheritance and this means that locales can imitate connections between functions, procedures or objects. It is possible to establish some general properties of an algorithm or to compare these properties. So, it is possible to compare programs. And this is a great advantage in program verification, something that is not done very often. This could help in better understanding of an algorithm which is essential for computer science education. So apart from being able to formalize verification in easier manner, this approach gives us opportunity to compare different programs. This was showed on Selection and Heap sort example and the connection between these two sorts was easy to comprehend. The value of this approach is not so much in obtaining a nice implementation of some algorithm, but in unraveling its structure. This is very important for computer science education and this can help in better teaching and understanding of an algorithms.

Using experience from this formalization, we came to conclusion that the general principle for refinement in program verification should be: divide program into small modules (functions, classes) and verify each modulo separately in order that corresponds to the order in entire program implementation. Someone may argue that this principle was not followed in each step of formalization, for example when we implemented Selection sort or when we defined is_heap and multiset in one step, but we feel that those function were simple and deviations in their implementations are minimal.

The next step is to formally verify all sorting algorithms and using refinement method to formally analyze and compare different sorting algorithms.