Verification of Selection and Heap Sort Using Locales

Danijela Petrović

May 28, 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 someones 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

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.

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

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

3 Defining data structure and
key function remove_max

theory \(\text{RemoveMax}\)
imports \(\text{Sort}\)
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 \(\text{Sort}\). 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 \(\text{Collection} =
\) fixes \(\text{empty} :: \text{'}b\)
— Represents empty element of the object (for example, for list it is [])
fixes \(\text{is-empty} :: \text{'}b \Rightarrow \text{bool}\)
— Function that checks weather the object is empty or not
fixes \(\text{of-list} :: \text{'}a \: \text{list} \Rightarrow \text{'}b\)
— Function transforms given list to desired object (for example, for heap sort,
function \(\text{of-list}\) transforms list to heap)
fixes \(\text{multiset} :: \text{'}b \Rightarrow \text{'}a \: \text{multiset}\)
— Function makes a multiset from the given object. A multiset is a collection
without order.
assumes \(\text{is-empty-inj}: \text{is-empty} \: e \implies e = \text{empty}\)
— 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) = multiset-of 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 ⇒ multiset e = {#}

using is-empty-inj multiset-empty

by auto

definition set :: 'b ⇒ 'a set where

[simp]: set l = set-of (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 and

is-empty :: 'b ⇒ bool and

of-list :: 'a::linorder list ⇒ 'b and

multiset :: 'b ⇒ 'a::linorder multiset +

fixes remove-max :: 'b ⇒ 'a × '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 ⇒ 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:

\[
\neg \text{is-empty } l \land \text{inv } l; (m, l') = \text{remove-max } l \implies m = \text{Max } (\text{set } l)
\]

— — First parameter of the return value of the function \text{remove-max} is the maximum element.

**assumes** remove-max-multiset:

\[
\neg \text{is-empty } l \land \text{inv } l; (m, l') = \text{remove-max } l \implies \text{multiset } l' + \{ \#m\# \} = \text{multiset } l
\]

— — Condition for multiset, ensures that nothing new is added or nothing is lost after applying \text{remove-max} function.

**assumes** remove-max-inv:

\[
\neg \text{is-empty } l \land \text{inv } l; (m, l') = \text{remove-max } l \implies \text{inv } l'
\]

— — 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 \text{remove-max} must not change that.

**begin**

**lemma** remove-max-multiset-size:

\[
\neg \text{is-empty } l \land \text{inv } l; (m, l') = \text{remove-max } l \implies \text{size } (\text{multiset } l) > \text{size } (\text{multiset } l')
\]

**using** remove-max-multiset[of \(l\) \(m\) \(l'\)]

**by** (metis mset-less-size multi-psub-of-add-self)

**lemma** remove-max-set:

\[
\neg \text{is-empty } l \land \text{inv } l; (m, l') = \text{remove-max } l \implies \text{set } l' \cup \{ m \} = \text{set } l
\]

**using** remove-max-multiset[of \(l\) \(m\) \(l'\)]

**by** (metis set-def set-of-single set-of-union)

As it is said before in each iteration invariant condition must be satisfied, so the \text{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) \text{ssort'} where

\[
\text{ssort'} \ l \ sl =
\]

\[
\begin{cases}
\text{if is-empty } l \text{ then } sl \\
\text{else}
\end{cases}
\]

**declare** ssort'.simps[code]

**definition** \text{ssort} :: \(' \ a \ list \Rightarrow \ ' \ a \ list\ where
\[\text{ssort } l = \text{ssort}' \text{ (of-list } l) \]\\

**inductive ssort'\text{-dom where}**\\
\[
\text{step: } [\forall m' l'. \neg \text{is-empty } l; (m, l') = \text{remove-max } l] \implies ssort'\text{-dom } (l', m \# sl)] \implies ssort'\text{-dom } (l, sl)
\]

**lemma ssort'\text{-termination}:**\\
\[\text{assumes } \neg \text{is-empty } l; (m, l') = \text{remove-max } l; \text{ssort}'\text{-dom } (l, sl) \implies P l' (m \# sl)]\]

**proof** (\[\text{induct } l \text{ rule: wf-induct[of measure } (\lambda (l, sl). \text{size (multiset } l))\])\\
\[\text{let } ?r = \text{measure } (\lambda (l, sl). \text{size (multiset } l))\]
\[\text{fix } m' l'. \neg \text{is-empty } l; (m, l') = \text{remove-max } l; \text{ssort}'\text{-dom } (l, sl)\]
\[\text{by (cases } p)\]
\[\text{show } \neg \text{is-empty } l; (m, l') = \text{remove-max } l; \text{ssort}'\text{-dom } (l, sl)\]
\[\text{by auto}\]
\[\text{qed}\]
\[\text{qed simp}\]

**lemma ssort'Induct:**\\
\[\text{assumes } \neg \text{is-empty } l; (m, l') = \text{remove-max } l; \text{ssort}'\text{-dom } (l, sl)\]
\[\text{shows } P l' (m \# sl)]\]

**proof** (\[\text{induct } (l, sl) \text{ arbitrary: } l sl \text{ rule: ssort}'\text{-dom.induct}\])\\
\[\text{case } (\text{step } l sl)\]
\[\text{show } \neg \text{is-empty } l\]
\[\text{case True}\]
\[\text{thus } \neg \text{is-empty } l\]
\[\text{using } \neg \text{is-empty } l\]
\[\text{by auto}\]
\[\text{thus } \neg \text{is-empty } l\]
\[\text{using } \neg \text{is-empty } l\]
\[\text{by auto}\]
\[\text{thus } \neg \text{is-empty } l\]
\[\text{using } \neg \text{is-empty } l\]
\[\text{by auto}\]
\[\text{thus } \neg \text{is-empty } l\]

8
using step(4) step(5) ssort'.simps[of l sl] is-empty-inv[of l]
by simp
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 multiset-of-ssort':
  assumes inv l
  shows multiset-of (ssort' l sl) = multiset l + multiset-of sl
  using assms
proof -
  have multiset empty + multiset-of (ssort' l sl) = multiset l + multiset-of sl
    using assms
proof (rule ssort'Induct)
fix l sl m l'
assumption
  inv l l
  (m, l') = remove-max l l
  multiset l l + multiset-of sl l = multiset l + multiset-of sl
thus multiset l' + multiset-of (m # sl) = multiset l + multiset-of sl
using remove-max-multiset[of l l 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'
assumption
assume ¬ is-empty l

\[
\text{inv } l = \text{remove-max } l \\
(m, l') = \text{remove-max } l \\
\text{sorted } sl \land (\forall x \in \text{set } l, \forall y \in \text{List } sl, x \leq y)
\]

thus \( \text{sorted } (m \# sl) \land (\forall x \in \text{set } l', \forall y \in \text{List } sl \# m, x \leq y) \)

using \( \text{remove-max-set}[of } l \# m \land \text{remove-max-max}[of } l \# m \land \text{of-list-inv} \)

apply \( (\text{auto simp add: sorted-Cons}) \)

by \( (\text{metis Max-ge finite-set-of insert-iff mem-set-of-iff}) \)

qed

thus \( ?\text{thesis} \)

by \( \text{simp} \)

qed

lemma \( \text{sorted-ssort}: \text{sorted } (\text{ssort } i) \)

unfolding \( \text{ssort-def} \)

using \( \text{sorted-ssort}\'[of } \text{of-list } i \land \text{of-list-inv} \)

by \( \text{auto} \)

lemma \( \text{permutation-ssort}: \text{ssort } l <\sim\sim> l \)

proof \( (\text{subst multiset-of-eq-perm}[\text{symmetric}]) \)

show \( \text{multiset-of } (\text{ssort } l) = \text{multiset-of } l \)

unfolding \( \text{ssort-def} \)

using \( \text{multiset-of-ssort}\'[of } \text{of-list } l \land \text{of-list-inv} \)

by \( \text{simp} \)

qed

Using assumptions given in the definitions of the locales \( \text{Collection} \) and \( \text{RemoveMax} \) for the functions \( \text{multiset}, \text{is_empty}, \text{of_list} \) and \( \text{remove_max} \) it is no difficulty to show:

sublocale \( \text{RemoveMax} < \text{Sort } \text{ssort} \)

by \( (\text{unfold-locales} \ (\text{auto simp add: sorted-ssort permutation-ssort})) \)

end

4 Verification of functional Selection Sort

theory \( \text{SelectionSort-Functional} \)

imports \( \text{RemoveMax} \)

begin

4.1 Defining data structure

Selection sort works with list and that is the reason why \( \text{Collection} \) should be interpreted as list.

interpretation \( \text{Collection } \lambda l. l = \lambda id \text{multiset-of} \)

by \( (\text{unfold-locales}, \text{auto}) \)
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.

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

\[
\text{definition } \text{remove-max } \text{where}
\]

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

\[
\text{lemma } \text{max-Max-commute:}
\]

\[
\text{finite } A \implies \text{max} \ (\text{Max} \ (\text{insert} \ m \ A)) \ x = \text{max} \ m \ (\text{Max} \ (\text{insert} \ x \ A))
\]

\[
\text{apply (cases } A = \{\}, \text{ simp)}
\]

\[
\text{by (metis Max-insert max.commute max.left-commute)}
\]

The function really returned the maximum value.

\[
\text{lemma } \text{remove-max-max-lemma:}
\]

\[
\text{shows } \text{fst} \ (\text{foldl } f \ (m, t) \ l) = \text{Max} \ (\text{set} \ (m \# l))
\]

\[
\text{using assms}
\]

\[
\text{proof (induct } l \text{ arbitrary; } m \ t \text{ rule: rev-induct)}
\]

\[
\text{case (snoc } x \ xs\}
\]

\[
\text{let } ?a = \text{foldl } f \ (m, t) \ xs
\]

\[
\text{let } ?m' = \text{fst } ?a \text{ and } ?t' = \text{snd } ?a
\]

\[
\text{have } \text{fst} \ (\text{foldl } f \ (m, t) \ (xs \ @ [x])) = \text{max} \ ?m' \ x
\]

\[
\text{by (cases } ?a \text{) (auto simp add: max-def)}
\]

\[
\text{thus } ?\text{case}
\]

\[
\text{using snoc}
\]

\[
\text{by (simp add: max-Max-commute)}
\]

\[
\text{qed simp}
\]

\[
\text{lemma } \text{remove-max-max:}
\]

\[
\text{assumes } l \neq [] \ (m, l') = \text{remove-max } l
\]

\[
\text{shows } m = \text{Max} \ (\text{set } l)
\]

\[
\text{using assms}
\]

\[
\text{unfolding remove-max-def}
\]

\[
\text{using remove-max-max-lemma[of } \text{hd } l \ [] \ \text{tl } l\]
\]

\[
\text{using } \text{fst-cone[of } m \ l'\]

\[
\text{by simp}
\]

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

\[
\text{lemma } \text{remove-max-multiset-of-lemma:}
\]

\[
\text{assumes } (m, l') = \text{foldl } f \ (m', t') \ l
\]

\[
\text{shows } \text{multiset-of} \ (m \# l') = \text{multiset-of} \ (m' \# t' @ l)
\]

\[
\text{using assms}
\]
proof (induct \( l \) arbitrary; \( l' \) \( m \) \( m' \) \( t' \) rule: rev-induct)

\begin{verbatim}
case (snoc \( x \) \( xs \))
let \(?a = \text{foldl } f \ (m', \ t') \ xs\) 
let \(?m' = \text{fst } \?a \text{ and } \?t' = \text{snd } \?a\)

have multiset-of \(?m' \# \?t'\) = multiset-of \((m' \# t' @ xs))
  using snoc(1)[of \(?m' \?t' \m' \t']
  by simp
thus \(?case\)
  using snoc(2)
  apply (cases \(?a\))
  apply (auto split: split-if-asm, (simp add: union-lcomm union-commute)+)
  by (metis union-assoc)
\end{verbatim}

qed simp

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

definition ssf-ssort' where
  \[\text{simp, code del: } \text{ssf-ssort'} = \text{RemoveMax.ssort'} (\lambda \ l. \ l = []) \text{ remove-max}\]

definition ssf-ssort where
  \[\text{simp, code del: } \text{ssf-ssort} = \text{RemoveMax.ssort} (\lambda \ l. \ l = []) \text{ id } \text{remove-max}\]

interpretation SSSRemoveMax:
  \[\text{RemoveMax [] \lambda } l. \ l = [] \text{ id } \text{multiset-of } \text{remove-max } \lambda -. \text{ True}\]
  where
  \[\text{RemoveMax.ssort'} (\lambda \ l. \ l = []) \text{ remove-max } = \text{ssf-ssort'} \text{ and}\]
  \[\text{RemoveMax.ssort} (\lambda \ l. \ l = []) \text{ id } \text{remove-max } = \text{ssf-ssort} \text{ using}\]
  \text{remove-max-max}
  by (unfold-locales, auto simp add: remove-max-multiset-of)

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 $T$ a Tree a Tree is represented a node whose root element is of type 'a and its left and right branch is also a tree of type 'a.

```haskell
primrec size :: 'a Tree ⇒ nat where
  size E = 0
  | size (T v l r) = 1 + size l + size r
```

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

```haskell
primrec multiset where
  multiset E = {#}
  | multiset (T v l r) = multiset l + {#v#} + multiset r
```

```haskell
primrec val where
  val (T v - -) = v
```

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

```haskell
fun is-heap :: 'a::linorder Tree ⇒ bool where
  is-heap E = True
  | is-heap (T v E E) = True
  | is-heap (T v E r) = (v ≥ val r ∧ is-heap r)
  | is-heap (T v l E) = (v ≥ val l ∧ is-heap l)
  | is-heap (T v l r) = (v ≥ val r ∧ is-heap r ∧ v ≥ val l ∧ is-heap l)
```

```haskell
lemma heap-top-geq:
  assumes a ∈ # multiset t is-heap t
  shows val t ≥ a
  using assms
  by (induct t rule: is-heap.induct) (auto split: split-if-asm)
```

```haskell
lemma heap-top-max:
  assumes t ≠ E is-heap t
  shows val t = Max (set-of (multiset t))
  proof (rule Max-eqI[ symmetric])
    fix y
    assume y ∈ set-of (multiset t)
    thus y ≤ val t
      using heap-top-geq[of t y] (is-heap t)
      by simp
    next
    show val t ∈ set-of (multiset t)
      using (t ≠ E)
      by (cases t) auto
  qed simp
```

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

locale Heap = Collection empty is-empty of-list multiset for
empty :: 'b and
is-empty :: 'b ⇒ bool and
of-list :: 'a::linorder list ⇒ 'b and
multiset :: 'b ⇒ 'a::linorder multiset +
fixes as-tree :: 'b ⇒ 'a::linorder Tree
— This function is not very important, but it is needed in order to avoide problems with types and to detect that observed object is a tree.
fixes remove-max :: 'b ⇒ 'a × 'b
assumes multiset: multiset l = Heap.multiset (as-tree l)
assumes is-heap-of-list: is-heap (as-tree (of-list i))
assumes as-tree-empty: as-tree t = E ⟷ is-empty t
assumes remove-max-multiset':
[ ¬ is-empty l; (m, l') = remove-max l ] ⇒ multiset l' + {#m#} = multiset l
assumes remove-max-is-heap:
[ ¬ is-empty l; is-heap (as-tree l); (m, l') = remove-max l ] ⇒
is-heap (as-tree l')
assumes remove-max-val:
[ ¬ is-empty t; (m, t') = remove-max t ] ⇒ m = val (as-tree t)

It is very easy to prove that locale Heap is sublocale of locale RemoveMax

subst locale Heap <
RemoveMax empty is-empty of-list multiset remove-max λ t. is-heap (as-tree t)
proof
fix x
show is-heap (as-tree (of-list x))
  by (rule is-heap-of-list)
next
fix l m l'
assume ¬ is-empty l (m, l') = remove-max l
thus multiset l' + {#m#} = multiset l
  by (rule remove-max-multiset')
next
fix l m l'
assume ¬ is-empty l is-heap (as-tree l) (m, l') = remove-max l
thus is-heap (as-tree l')
  by (rule remove-max-is-heap)
next
fix l m l'
assume ¬ is-empty l is-heap (as-tree l) (m, l') = remove-max l
thus m = Max (set l)
  unfolding set-def
  using heap-top-max[of as-tree l] remove-max-val[of l m l']
  using multiset is-empty-inj as-tree-empty
  by auto
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 consistency, 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:
\[ \text{val}(\text{merge } l \ r) = \text{val } l \lor \text{val}(\text{merge } l \ r) = \text{val } r \]
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 \( v_1 \ l_1 \ r_1 \ v_2 \ l_2 \ r_2 \))
  thus \(?case\)
    proof (cases \( v_2 \leq v_1 \))
      case True
      hence \[ \text{val}(\text{merge } (T \ v_1 \ l_1 \ r_1) (T \ v_2 \ l_2 \ r_2)) = \text{val } (T \ v_1 \ l_1 \ r_1) \]
        by auto
      thus \(?thesis\)
        by auto
    next
    case False
    hence \[ \text{val}(\text{merge } (T \ v_1 \ l_1 \ r_1) (T \ v_2 \ l_2 \ r_2)) = \text{val } (T \ v_2 \ l_2 \ r_2) \]
      by auto
    thus \(?thesis\)
      by auto
  qed
qed

Function \text{merge} merges two heaps into one:

lemma merge-heap-is-heap:
\[ \text{assumes } \text{is-heap } l \ \text{is-heap } r \]
\[ \text{shows } \text{is-heap } (\text{merge } l \ r) \]
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 \( v_1 \ l_1 \ r_1 \ v_2 \ l_2 \ r_2 \))
  thus \(?case\)
    proof (cases \( v_2 \leq v_1 \))
      case True
      have \( \text{is-heap } l_1 \)
        using \( \text{is-heap } (T \ v_1 \ l_1 \ r_1) \).
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[of l1 T v2 l2 r2]
by auto
show ?thesis
proof(cases r1 = E)
case True
  hence is-heap (merge l1 (T v1 l1 r1) (T v2 l2 r2))
    using (T v1 l1 r1) (T v2 l2 r2) = T v1 (T v2 l2 r2) E
    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
    ∨ 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-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 \( r1 \neq E \) is-heap.simps(5))
obtain \( ll1 \ lr1 \ lv1 \) where \( r1 = T \ le1 \ ll1 \ lr1 \)
  using \( r1 \neq E \)
  by (metis Tree.exhaust)
obtain \( rr1 \ rv1 \) where \( \text{merge } l1 \ (T \ v2 \ l2 \ r2) = T \ \text{rv1} \ \text{rl1} \ \text{rr1} \)
  using (merge \( l1 \ (T \ v2 \ l2 \ r2) = E \))
  by (metis Tree.exhaust)
have \( \text{val} \ (\text{merge } l1 \ (T \ v2 \ l2 \ r2)) \leq v1 \)
  using (\( \text{val} \ (\text{merge } l1 \ (T \ v2 \ l2 \ r2)) = \text{val } l1 \) \( \land \)
  \( \text{val}(\text{merge } l1 \ (T \ v2 \ l2 \ r2)) = v2 \))
  using (\( v1 \geq v2 \) \( \land \) \( v1 \geq \text{val } l1 \))
  by auto
hence is-heap \( T \ v1 \ (\text{merge } l1 \ (T \ v2 \ l2 \ r2)) \ r1 \)
  using is-heap.simps(5)[of \( v1 \ le1 \ ll1 \ lr1 \ \text{rv1} \ \text{rl1} \ \text{rr1} \)]
  is-heap.simps(4) is-heap.simps(5)
  by auto
the **thesis**
  using (\( v2 \leq v1 \))
  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 \( \text{merge } l2 \ (T \ v1 \ l1 \ r1) \)
  using False is-heap \( (T \ v1 \ l1 \ r1) \ r1 \)
  by auto
have \( \text{val} \ (\text{merge } l2 \ (T \ v1 \ l1 \ r1)) = \text{val } l2 \lor \)
  \( \text{val}(\text{merge } l2 \ (T \ v1 \ l1 \ r1)) = v1 \)
  using merge-val[of \( l2 \ T \ v1 \ l1 \ r1 \)]
  by auto
show **thesis**
proof(cases \( r2 = E \))
case True
show ?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 \(\langle r_2 = E \rangle \ (\neg v_2 \leq v_1)\)
      by auto
    thus ?thesis
      using 3
      using \(\langle \neg v_2 \leq v_1 \rangle\)
      by auto
next
  case False
    hence \(v_2 \geq \text{val} \ l_2\)
      using 3
      (4)
    by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  qed
next
  case False
    hence \(v_2 \geq \text{val} \ r_2\)
      using 3
      (4)
    by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  qed
by (metis False Tree.exhaust (r2 ≠ E) is-heap.simps(5))
obtain ll1 lr1 lv1 where r2 = T lv1 ll1 lr1
  using (r2 ≠ E)
  by (metis Tree.exhaust)
obtain rl1 rr1 rv1 where merge l2 (T v1 l1 r1) = T rv1 rl1 rr1
  using (merge l2 (T v1 l1 r1) ≠ E)
  by (metis Tree.exhaust)
have val (merge l2 (T v1 l1 r1)) ≤ v2
  using (val (merge l2 (T v1 l1 r1)) = val l2 ∨
    val (merge l2 (T v1 l1 r1)) = v1)
  by auto
hence is-heap (T v2 (merge l2 (T v1 l1 r1)) r2)
  using is-heap.simps(5)[of v1 lv1 ll1 lr1 rv1 rl1 rr1]
  using (r2 = T lv1 ll1 lr1) (merge l2 (T v1 l1 r1) = T rv1 rl1 rr1)
  using (is-heap r2) (is-heap (merge l2 (T v1 l1 r1)) (v2 ≥ val r2)
  by auto
thus ?thesis
  using (∀ v1 ≥ v2. v2 ≥ val l2)
  by auto
qed
qed
qed

definition insert :: 'a::linorder ⇒ 'a Tree ⇒ 'a Tree where
insert v t = merge t (T v E E)

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

definition hs-is-empty where
hs-is-empty t ≡ t = E

Definition of function remove_max:

fun hs-remove-max :: 'a::linorder Tree ⇒ 'a × 'a Tree where
hs-remove-max (T v l r) = (v, merge l r)

lemma merge-multiset:
  multiset l + multiset g = multiset (merge l g)
proof
  (induct l g rule:merge.induct)
next
  case (1 l)
  thus ?case
    by auto
next
  case (2 g)
  thus ?case
    by auto
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-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) = multiset-of l
  proof (induct l)
Proof that defined functions are interpretation of abstract functions from locale Heap:

interpretation Heap E hs-is-empty hs-of-list multiset id hs-remove-max
proof
fix l
show multiset l = Heap.multiset (id l)
by auto
next
fix l
show is-heap (id (hs-of-list l))
proof (induct l)
case Nil
thus ?case
by auto
next
case (Cons a l)
have hs-of-list (a # l) = merge (hs-of-list l) (T a E E)
apply auto
unfolding insert-def
by auto
have is-heap (T a E E)
by auto
hence is-heap (merge (hs-of-list l) (T a E E))
using Cons merge-heaps-is-heap[of hs-of-list l T a E E]
by auto
thus ?case
using (hs-of-list (a # l) = merge (hs-of-list l) (T a E E))
by auto
qed
next
fix t
show (id t = E) = hs-is-empty t
by auto
next
fix t m t'
assume \( \neg \text{hs-is-empty} \ t \ (m, t') = \text{hs-remove-max} \ t \)
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 merge-multiset[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 \( \neg \text{hs-is-empty} \ t \ (m, t') = \text{hs-remove-max} \ t \)
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 merge-heap-is-heap
   by auto

next
fix t m t'
assume \( \neg \text{hs-is-empty} \ t \ (m, t') = \text{hs-remove-max} \ t \)
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 \( t \) ≡ \( \text{val} \ (\text{left} \ t) \)
primrec right :: 'a Tree ⇒ 'a Tree where
  right (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 \( v2 \geq v1 \))
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 \(v\) \((\text{siftDown} (\text{set-val} (T \ v1 \ l1 \ r1) \ v2))\)
using \(\neg v2 \geq v1\) 4(2)
by auto
hence in-tree \(v\) \((T \ v2 \ l1 \ r1)\)
using 4(1) \(\neg v2 \geq v1\)
by auto
thus \(?thesis\
proof(cases \(v2 = v)\
case True
thus \(?thesis\
by auto
next
case False
hence 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 \((5-1 \ v' \ v1 \ l1 \ r1 \ v2 \ l2 \ r2)\)
show \(?case\
proof(cases \(v = v' \lor v = v1 \lor v = v2\))
case True

26
thus \(?\)thesis
by auto

next
case False
show \(?\)thesis
proof(cases v1 ≥ v2)
case True
show \(?\)thesis
proof(cases v' ≥ v1)
case True
thus \(?\)thesis
using \((v1 ≥ v2) \ 5-1\)
by auto

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

next
case False
hence in-tree v (siftDown (set-val (T v1 l1 r1) v'))
using 5-1(3) \((¬ \text{ in-tree } v (T v2 l2 r2) \ (v1 ≥ v2) \ (¬ v' ≥ v1))\)
using \((v = v' \vee v = v1 \vee v = v2)\)
by auto
hence in-tree v (T v' l1 r1)
using 5-1(1) \((v1 ≥ v2) \ (¬ v' ≥ v1)\)
by auto
hence in-tree v (T v1 l1 r1)
using \((¬ (v = v' \vee v = v1 \vee v = v2))\)
by auto
thus \(?\)thesis
by auto
qed
qed

next
case False
show \(?\)thesis
proof(cases v' ≥ v2)
case True
thus \(?\)thesis
using \((¬ v1 ≥ v2) \ 5-1\)
by auto

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

next
case (5-2 v' v1 l1 r1 v2 l2 r2)
  show \( ?\text{case} \)
  proof (cases \( v = v' \lor v = v1 \lor v = v2 \))
    case True
    thus \( \text{thesis} \)
    by auto
next
case False
  show \( ?\text{thesis} \)
  proof (cases \( v1 \geq v2 \))
    case True
    show \( ?\text{thesis} \)
    proof (cases \( v' \geq v1 \))
      case True
      thus \( ?\text{thesis} \)
      using \( v1 \geq v2 \) 5-2
      by auto
    next
case False
      thus \( ?\text{thesis} \)
      proof (cases in-tree v (T v2 l2 r2))
        case True
        thus \( ?\text{thesis} \)
        by auto
    next
case False
    hence in-tree v (siftDown (set-val (T v1 l1 r1) v'))
    using 5-2(3) \( \neg \text{in-tree v (T v2 l2 r2)} \) \( \neg a1 \geq v2 \) \( \neg v' \geq v1 \)
using $\lnot (v = v' \lor v = v1 \lor v = v2)$
by auto
hence in-tree $v$ ($T v1 l1 r1$)
using $\text{5-2(1)} \langle v1 \geq v2 \rangle (\lnot v' \geq v1)$
by auto
hence in-tree $v$ ($T v1 l1 r1$)
using $\langle \lnot (v = v' \lor v = v1 \lor v = v2) \rangle$
by auto
thus $\text{?thesis}$
by auto
qed
qed
next
case $\text{False}$
show $\text{?thesis}$
proof(cases $v' \geq v2$)
  case $\text{True}$
  thus $\text{?thesis}$
  using $\langle \lnot v1 \geq v2 \rangle \text{ 5-2}$
  by auto
next
case $\text{False}$
thus $\text{?thesis}$
proof(cases in-tree $v$ ($T v1 l1 r1$))
  case $\text{True}$
  thus $\text{?thesis}$
  by auto
next
case $\text{False}$
  hence in-tree $v$ ($\text{siftDown (set-val (T v2 l2 r2) v')} \rangle$
  using $\text{5-2(3)} \langle \lnot \text{in-tree v (T v1 l1 r1) \langle v1 \geq v2 \rangle (\lnot v' \geq v2) \rangle$
  using $\langle \lnot (v = v' \lor v = v1 \lor v = v2) \rangle$
  by auto
  hence in-tree $v$ ($T v' l2 r2$)
  using $\text{5-2(2)} \langle v1 \geq v2 \rangle (\lnot v' \geq v2)$
  by auto
  hence in-tree $v$ ($T v2 l2 r2$)
  using $\langle \lnot (v = v' \lor v = v1 \lor v = v2) \rangle$
  by auto
  thus $\text{?thesis}$
  by auto
  qed
  qed
  qed
  qed
  qed
  qed
lemma $\text{siftDown-heap-is-heap:}$
assumes \( \text{is-heap} \ l \ \text{is-heap} \ r \ t = T \ v \ l \ r \)
shows \( \text{is-heap} \ (\text{siftDown} \ t) \)
using \( \text{assms} \)
proof (induct \( t \) arbitrary: \( v \ l \ r \) rule: \( \text{siftDown}.\text{induct} \))
\begin{itemize}
\item case 1
  \begin{itemize}
  \item thus \( \exists \) case
    \begin{itemize}
    \item by simp
    \end{itemize}
  \end{itemize}
\item next
\item case \( (2 \ v') \)
  \begin{itemize}
  \item show \( \exists \) case
    \begin{itemize}
    \item by simp
    \end{itemize}
  \end{itemize}
\item next
\item case \( (3 \ v2 \ v1 \ l1 \ r1) \)
  \begin{itemize}
  \item show \( \exists \) case
    \begin{itemize}
    \item proof (cases \( v2 \geq v1 \))
    \begin{itemize}
    \item case True
      \begin{itemize}
      \item thus \( \exists \) thesis
        \begin{itemize}
        \item using \( 3(2) \ 3(4) \)
        \item by auto
        \end{itemize}
      \end{itemize}
    \item next
    \begin{itemize}
    \item case False
      \begin{itemize}
      \item show \( \exists \) thesis
        \begin{itemize}
        \item proof (cases \( v' = v2 \))
        \begin{itemize}
        \item case True
          \begin{itemize}
          \item thus \( \exists \) thesis
            \begin{itemize}
            \item using False (is-heap \( ?t \) \* 
            \item by auto
            \end{itemize}
          \end{itemize}
        \item next
        \begin{itemize}
        \item case False
          \begin{itemize}
          \item have in-tree \( v' \ ?t \)
            \begin{itemize}
            \item using *
            \item using siftDown-in-tree[of \( ?t \)]
            \item by simp
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
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
      by auto
  next
case False
  let ?t = 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
  by auto
hence is-heap $l1$ is-heap $r1$
  using 4
  apply (induct $r$ rule:is-heap.induct)
  by auto
hence is-heap ?t
  using False 4[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]
  by auto
thus ?thesis
  using (is-heap ?t) False *
  by auto
qed
qed

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

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

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

thus \(?thesis\)
using \(\text{is-heap-max}[\text{of } v' T v2 \ l2 \ r2] \ 5-1\)
by auto

qed

thus \(?thesis\)
using \((\text{is-heap } ?t) \ (v2 \geq v3) \ast \text{False} \ 5-1\)
by auto

qed

next

case False
show \(?thesis\)
proof(cases \( v1 \geq v3 \))

case True
thus \(?thesis\)
using \(\lnot \ v2 \geq v3) \ 5-1\)
by auto

next

case False
let \(?t = \text{siftDown} (T \ v1 \ l3 \ r3)\)
obtain \(l' \ v' \ r' \ where \ast: ?t = T \ v' \ l' \ r' \ v' \geq v1\)
using siftDown-Node
by blast

have \(\text{is-heap } l3 \ \text{is-heap } r3\)
using \(5-1(4, \ 5)\)
apply(induct \( r \ \text{rule: is-heap.induct}\))
by auto

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

have \(v3 \geq v'\)
proof(cases \( v' = v1 \))

case True
thus \(?thesis\)
using False
by auto

next

case False
have \(\text{in-tree } v' \ ?t\)
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 \( \exists \)thesis
  using is-heap-max[of \( v' \) \( T v3 l3 r3 \)] 5-1
  by auto

qed

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

qed

qed

next

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

show \( \exists \)case

proof((cases \( v2 \geq v3 \))
  case True
    show \( \exists \)thesis
      proof((cases \( v1 \geq v2 \))
        case True
        thus \( \exists \)thesis
          using \( \langle v2 \geq v3 \rangle \) 5-2
          by auto
        next
        case False
        let \( ?t = \text{siftDown} \ (T v1 l2 r2) \)
        obtain \( l' \ v' r' \) where \( *: \ ?t = T v' l' r' v1 \leq v' \)
          using siftDown-Node
          by blast
        have is-heap \( l2 \) is-heap \( r2 \)
          using 5-2(3, 5)
          apply(induct \( l \) rule:is-heap.induct)
          by auto
        hence is-heap \( ?t \)
          using 5-2(1)[of \( l2 \) \( r2 \) \( v1 \)] \( \langle v2 \geq v3 \rangle * \) False
          by auto
        have \( v2 \geq v' \)
        proof((cases \( v' = v1 \))
          case True
          thus \( \exists \)thesis
            using False
            by auto
          next
          case False
          have in-tree \( v' \) \( ?t \)
            using * siftDown-in-tree

34
by auto
hence \( \text{in-tree } v' (T v1 l2 r2) \)
  using siftDown-in-tree-set[of \( v' \) \( T v1 l2 r2 \)]
  by auto
hence \( \text{in-tree } v' (T v2 l2 r2) \)
  using False
  by auto
thus \(?\text{thesis}\)
  using is-heap-max[of \( v' \) \( T v2 l2 r2 \)] 5-2
  by auto
qed
thus \(?\text{thesis}\)
  using \( \langle \text{is-heap } ?t \rangle \langle v2 \geq v3 \rangle \ast \text{False } 5-2 \)
  by auto
qed
next
case False
show \(?\text{thesis}\)
proof(cases \( v1 \geq v3 \))
case True
  thus \(?\text{thesis}\)
    using \( \neg v2 \geq v3 \) 5-2
    by auto
next
case False
let \(?t = \text{siftDown} (T v1 l3 r3)\)
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 5-2(4, 5)
  apply(induct r rule:is-heap.induct)
  by auto
hence is-heap \(?t\)
  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 \( \text{in-tree } v' ?t\)
  using * siftDown-in-tree
  by auto
hence \( \text{in-tree } v' (T v1 l3 r3) \)
  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 $\neg v2 \geq v3$ * False

by auto

qed

Definition of the function heapify which makes a heap from any given binary tree.

primrec heapify where

heapify $E = E$

| heapify ($T v l r$) = siftDown ($T v$ (heapify $l$) (heapify $r$))

lemma heapify-heap-is-heap:

is-heap (heapify $t$)

proof (induct $t$)

| case $E$

| thus $\neg v2 \geq v3$ * False

by auto

next

| case ($T v l r$)

| thus $\neg v2 \geq v3$ * False

| using siftDown-heap-is-heap[of heapify $l$ heapify $r$ $T v$ (heapify $l$) (heapify $r$) $v$]

by auto

qed

Definition of 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.

fun removeLeaf:: 'a::linorder Tree $\Rightarrow$ 'a $\times$ 'a Tree where

removeLeaf ($T v E E$) = ($v$, $E$)

| removeLeaf ($T v l E$) = (fst (removeLeaf $l$), $T v$ (snd (removeLeaf $l$)) $E$)

| removeLeaf ($T v E r$) = (fst (removeLeaf $r$), $T v E$ (snd (removeLeaf $r$)))

| removeLeaf ($T v l r$) = (fst (removeLeaf $l$), $T v$ (snd (removeLeaf $l$)) $r$)

Function of_list_tree makes a binary tree from any given list.

primrec of_list_tree:: 'a::linorder list $\Rightarrow$ 'a Tree where

of_list_tree $[] = E$

| of_list_tree ($v$ # tail) = $T v$ (of_list_tree tail) $E$

By applying heapify binary tree is transformed into heap.
definition hs-of-list where
hs-of-list l = heapify (of-list-tree l)

Definition of function 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 siftDown.

definition hs-remove-max :: 'a::linorder Tree ⇒ 'a × '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
[simp]: hs-is-empty t ≜ t = E

lemma siftDown-multiset:
multiset (siftDown t) = multiset t
proof(induct t rule:siftDown.induct)
case 1
  thus ?case
  by simp
next
case (2 v)
  thus ?case
  by simp
next
case (3 v1 v l r)
  thus ?case
  proof(cases v ≤ v1)
    case True
    thus ?thesis
    by auto
  next
  case False
  hence multiset (siftDown (T v1 (T v l r) E)) =
    multiset l + {#v1#} + multiset r + {#v#}
  using 3
  by auto
next
have multiset (T v1 (T v l r) E) =
  multiset l + {#v#} + multiset r + {#v1#}
  by auto
moreover
have multiset l + {#v1#} + multiset r + {#v#} =
  multiset l + {#v#} + multiset r + {#v1#}
  by (metis union-commute union-lcomm)
ultimately
show ?thesis
by auto
qed
next
case (4 v1 v l r)
thus ?thesis
proof(cases v ≤ v1)
  case True
  thus ?thesis
  by auto
next
case False
have multiset (set-val (T v1 E (T v l r)) v1) =
  multiset l + {#v1#} + multiset r
by auto
hence multiset (siftDown (T v1 E (T v l r))) =
  {#v#} + multiset (set-val (T v l r) v1)
using 4 False
by auto
hence multiset (siftDown (T v1 E (T v l r))) =
  {#v#} + multiset l + {#v1#} + multiset r
using multiset (set-val (T v l r) v1) =
  multiset l + {#v1#} + multiset r
by (metis union-commute union-lcomm)
moreover
have multiset (T v1 E (T v l r)) =
  {#v#} + multiset l + {#v#} + multiset r
by (metis calculation monoid-add-class.add.left-neutral
  multiset.simps(1) multiset.simps(2) union-commute union-lcomm)
moreover
have {#v#} + multiset l + {#v1#} + multiset r =
  {#v1#} + multiset l + {#v#} + multiset r
by (metis union-commute union-lcomm)
ultimately
show ?thesis
by auto
qed
next
case (5-1 v1 v l1 r1 v2 l2 r2)
thus ?thesis
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 + \{#v\} +
          multiset (T v2 l2 r2)
          using \(v1 \geq v2\); 5-1(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 \(v \geq v2\))
    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) + \{#v\} + multiset l2 +
          \{#v2\} + multiset r2
          using \(\neg v1 \geq v2\); 5-1(2)
          by (simp add: ac-simps)
  moreover
  have
          multiset (T v (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 +
\[
\{\#v\#\} + \text{multiset } r2
\]
by \text{metis union-commute union-lcomm}
ultimately
show \(\text{thesis}\)
by auto
qed
qed

next
case \((5 \cdot 2 v v1 l1 r1 v2 l2 r2)\)
thus \(\text{thesis}\)
proof\((\text{cases } v1 \geq v2)\)
case \(\text{True}\)
thus \(\text{thesis}\)
proof\((\text{cases } v \geq v1)\)
case \(\text{True}\)
thus \(\text{thesis}\)
using \(\langle v1 \geq v2 \rangle\)
by auto
next
case \(\text{False}\)
show \(\text{thesis}\)
proof\((\text{cases } v \geq v2)\)
case \(\text{True}\)
thus \(\text{thesis}\)
using \(\langle \text{False} \rangle\)
by auto

moreover
have \(\text{multiset } (\text{siftDown } (T v (T v1 l1 r1) (T v2 l2 r2))) =\)
\(\text{multiset } l1 + \{\#v\#\} + \text{multiset } r1 + \{\#v1\#\} +\)
\(\text{multiset } (T v2 l2 r2)\)
using \(\langle v1 \geq v2, 5 \cdot 2(1) \rangle\)
by auto
moreover
have \(\text{multiset } (T v (T v1 l1 r1) (T v2 l2 r2)) =\)
\(\text{multiset } l1 + \{\#v1\#\} + \text{multiset } r1 + \{\#v\#\} + \text{multiset } (T v2 l2 r2)\)
by auto
moreover
have \(\text{multiset } l1 + \{\#v1\#\} + \text{multiset } r1 + \{\#v\#\} + \text{multiset } (T v2 l2 r2) =\)
\(\text{multiset } l1 + \{\#v\#\} + \text{multiset } r1 + \{\#v1\#\} + \text{multiset } (T v2 l2 r2)\)
by \text{metis union-commute union-lcomm}
ultimately
show \(\text{thesis}\)
by auto
qed

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

40
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 v (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 multiset-of-list-tree:
multiset (of-list-tree l) = multiset-of l
proof (induct l)
case Nil
  thus ?case
    by auto
next
case (Cons v tail)
hence multiset (of-list-tree (v # tail)) = multiset-of tail + {#v#}
  by auto
also have ... = multiset-of (v # tail)
  by auto
finally show multiset (of-list-tree (v # tail)) = multiset-of (v # tail)
  by auto
qed

lemma multiset-heapify:
multiset (heapify t) = multiset t
proof (induct t)
case E
  thus ?case
    by auto
next
case \((T \ v \ l \ r)\)

hence \(\text{multiset} \left( \text{heapify} \ (T \ v \ l \ r) \right) = \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r\)

using \(\text{siftDown-multiset[of } T \ v \ (\text{heapify} \ l) \ (\text{heapify} \ r)\] \)

by auto

thus \(?\text{case}\)

by auto

qed

lemma \(\text{multiset-heapify-of-list-tree}:\)

\(\text{multiset} \ (\text{heapify} \ (\text{of-list-tree} \ l)) = \text{multiset-of} \ l\)

using \(\text{multiset-heapify[of of-list-tree} \ l]\)

using \(\text{multiset-of-list-tree[of} \ l]\)

by auto

lemma \(\text{removeLeaf-val-val}:\)

assumes \(\text{snd} \ (\text{removeLeaf} \ t) \neq E \ t \neq E\)

shows \(\text{val} \ t = \text{val} \ (\text{snd} \ (\text{removeLeaf} \ t))\)

using \(\text{assms}\)

apply \(\text{(induct} \ t \ \text{rule:removeLeaf.induct)}\)

by auto

lemma \(\text{removeLeaf-heap-is-heap}:\)

assumes \(\text{is-heap} \ t \ t \neq E\)

shows \(\text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ t))\)

using \(\text{assms}\)

proof \(\text{(induct} \ t \ \text{rule:removeLeaf.induct)}\)

case \((1 \ v)\)

thus \(?\text{case}\)

by auto

next

case \((2 \ v \ v1 \ l1 \ r1)\)

have \(\text{is-heap} \ (T \ v1 \ l1 \ r1)\)

using \(2(3)\)

by auto

hence \(\text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1)))\)

using \(2(1)\)

by auto

let \(?t = (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1)))\)

show \(?\text{case}\)

proof \((\text{cases} \ ?t = E)\)

case \(\text{True}\)

thus \(?\text{thesis}\)

by auto

next

case \(\text{False}\)

have \(v \geq v1\)

using \(2(3)\)

by auto
hence $v \geq \text{val } ?t$
  using False removeLeaf-val-val[of $T v \ll r1$]
  by auto

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

using \textit{assms}

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

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

next
  case (3 v v1 l1 r1)
  have \(t' = T \cdot v \cdot E \cdot (\text{snd} (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1)))\)
    using 3(3)
    by auto
  have \(v' = \text{fst} \ (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1))\)
    using 3(3)
    by auto
  hence \(\{\#v'\}\ + \textit{multiset} t' =\)
    \(\{\#\text{fst} \ (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1))\}\ + \textit{multiset} (\text{snd} (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1)))\ + \{\#v'\}\)
    using \(t' = T \cdot v \cdot E \cdot (\text{snd} (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1)))\) \(E\);
    by (simp add: \textit{ac-simps})
  have \(\{\#\text{fst} \ (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1))\}\ + \textit{multiset} (\text{snd} (\text{removeLeaf} \ (T \cdot v1 \cdot l1 \cdot r1))) =\)
    \textit{multiset} (T \cdot v1 \cdot l1 \cdot 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 (metis monoid-add-class.add_right-neutral
\multiset.\text{simps}(1) \multiset.\text{simps}(2) \text{union-commute})

next

\text{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\} + \text{multiset } (T v2 l2 r2)
using (t' = T v (\text{snd } (\text{removeLeaf } (T v1 l1 r1))) (T v2 l2 r2))
by (metis \multiset.\text{simps}(2) \text{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

\text{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))) +

47
\[
\{\#v\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2)
\]
using \(t' = T \ v \ (\text{snd \ (removeLeaf \ (T \ v1 \ l1 \ r1)))} \ (T \ v2 \ l2 \ r2)\)
by \(\text{metis \ multiset.simps(2) \ union-assoc}\)

have \(\{\#\text{fst \ (removeLeaf \ (T \ v1 \ l1 \ r1))}\#\} + \text{multiset} \ (\text{snd \ (removeLeaf \ (T \ v1 \ l1 \ r1)))} = \text{multiset} \ (T \ v1 \ l1 \ r1)\)
using 4-2(1)
by auto

hence \(\{\#v'\#\} + \text{multiset} \ t' = \text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2)\)

proof
- obtain \(v \ l \ r\) where \(t = T \ v \ l \ r\)
  using asms
  by \(\text{metis \ Tree.exhaust}\)
  hence \(\text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\}\)
  by auto
  have \(\{\#v'\#\} + \text{multiset} \ t = \text{multiset} \ (T \ v \ l \ r)\)
  by \(\text{metis \ multiset.simps(2) \ union-assoc}\)
  have \(\{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r = \text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\}\)
  by \(\text{metis \ union-commute \ union-lcomm}\)
  thus \(\text{thesis}\)
  using \(\text{multiset} \ (\text{set-val} \ t \ v') + \{\#\text{val} \ t\#\} = \text{multiset} \ l + \{\#v'\#\} + \text{multiset} \ r + \{\#v\#\}\);
  using \(\{\#v'\#\} + \text{multiset} \ t = \{\#v'\#\} + \text{multiset} \ l + \{\#v\#\} + \text{multiset} \ r\)
  by auto

qed

lemma \text{hs-remove-max-multiset}:
assumes \( (m, t') = \text{hs-remove-max } t \neq E \)
shows \( \{#m\} + \text{multiset } t' = \text{multiset } t \)

proof

let \(?v1 = \text{fst (removeLeaf } t\)
let \(?t1 = \text{snd (removeLeaf } t\)
show \(?thesis\)
proof (cases \(?t1 = E\))

  case \(\text{True}\)
  hence \(\{#m\} + \text{multiset } t' = \{#m\}\)
    using \(\text{assms}\)
    unfolding \(\text{hs-remove-max-def}\)
    by \(\text{auto}\)
  have \(?v1 = \text{val } t\)
    using \(\text{True assms(2)}\)
    apply \((\text{induct } t \text{ rule:removeLeaf.induct})\)
    by \(\text{auto}\)
  hence \(?v1 = m\)
    using \(\text{assms(1)}\) \(\text{True}\)
    unfolding \(\text{hs-remove-max-def}\)
    by \(\text{auto}\)
  hence \(\text{multiset } t = \{#m\}\)
    using \(\text{removeLeaf-multiset[of } \text{?v1 } ?t1 } t\) \(\text{True assms(2)}\)
    by \((\text{metis empty-neutral(2) multiset.simps(1) pair-collapse})\)
  thus \(?thesis\)
    using \(\{#m\} + \text{multiset } t' = \{#m\}\)
    by \(\text{auto}\)

next

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

Difined functions satisfy conditions of locale \textit{Heap} and thus represent interpretation of this locale.

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

\textbf{proof}

\begin{itemize}
\item \textbf{fix} \textit{t}
  \begin{itemize}
  \item \textbf{show} \textit{multiset }\textit{t} = \textit{multiset }\textit{(id }\textit{t})
    \begin{itemize}
    \item \textbf{by} \textit{auto}
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{next}
  \begin{itemize}
  \item \textbf{fix} \textit{t}
    \begin{itemize}
    \item \textbf{show} \textit{is-heap }\textit{(id }\textit{(hs-of-list }\textit{t}))
      \begin{itemize}
      \item \textbf{unfolding} \textit{hs-of-list-def}
      \item \textbf{using} \textit{heapify-heap-is-heap[of of-list-tree }\textit{t]}
      \item \textbf{by} \textit{auto}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{next}
  \begin{itemize}
  \item \textbf{fix} \textit{t m t'}
    \begin{itemize}
    \item \textbf{assume} \textit{¬} \textit{hs-is-empty }\textit{t (m, t')} = \textit{hs-remove-max }\textit{t}
    \item \textbf{thus} \textit{multiset }\textit{t'} + \textit{\{#m\#\}} = \textit{multiset }\textit{t}
      \begin{itemize}
      \item \textbf{using} \textit{hs-remove-max-multiset[of m t']}
      \item \textbf{by} \textit{(auto, metis union-commute)}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{next}
  \begin{itemize}
  \item \textbf{fix} \textit{t v t'}
    \begin{itemize}
    \item \textbf{assume} \textit{¬} \textit{hs-is-empty }\textit{t is-heap }\textit{(id }\textit{t)} \textit{(v', t')} = \textit{hs-remove-max }\textit{t}
    \item \textbf{let} \textit{?v1} = \textit{fst }\textit{(removeLeaf }\textit{t)}
    \item \textbf{let} \textit{?t1} = \textit{snd }\textit{(removeLeaf }\textit{t)}
    \item \textbf{have} \textit{is-heap }\textit{?t1}
      \begin{itemize}
      \item \textbf{using} \textit{(¬} \textit{hs-is-empty }\textit{t) (is-heap }\textit{id }\textit{t));}
      \item \textbf{using} \textit{removeLeaf-heap-is-heap[of ]}
      \item \textbf{by} \textit{auto}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

\textbf{show} \textit{is-heap }\textit{(id }\textit{t')}

\textbf{proof}(\textit{cases }\textit{?t1 = E})

\begin{itemize}
\item \textbf{case} \textit{True}
  \begin{itemize}
  \item \textbf{hence} \textit{t'} = \textit{E}
    \begin{itemize}
    \item \textbf{using} \textit{(v', t') = hs-remove-max }\textit{t}
      \begin{itemize}
      \item \textbf{unfolding} \textit{hs-remove-max-def}
      \item \textbf{by} \textit{auto}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{thus} \textit{?thesis}
  \begin{itemize}
  \item \textbf{by} \textit{auto}
  \end{itemize}
\end{itemize}

\textbf{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))
(metis (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: consis-
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.