Verification of Selection and Heap Sort Using Locales

Danijela Petrović

April 20, 2020

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

3 Defining data structure and
key function \textit{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 :: \text{'}b
  — – Represents empty element of the object (for example, for list it is [])
  fixes is-empty :: \text{'}b ⇒ \text{'}bool
  — – Function that checks weather the object is empty or not
  fixes of-list :: \text{'}a list ⇒ \text{'}b
  — – Function transforms given list to desired object (for example, for heap sort,
  function of-list transforms list to heap)
  fixes multiset :: \text{'}b ⇒ \text{'}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 = \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) = 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 where

[simp]: 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 and

is-empty :: 'b \Rightarrow bool and

of-list :: 'a::linorder list \Rightarrow 'b 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:

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

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

assumes remove-max-multiset:

\[ \lnot \text{is-empty } l; \ inv \ l; (m, l') = \text{remove-max } l \implies \text{add-mset } m \ (\text{multiset } l') = \text{multiset } l \]

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

assumes remove-max-inv:

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

— — 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:

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

using remove-max-multiset[of \ l \ m \ l']
by (metis mset-subset-size multi-psub-of-add-self)

lemma remove-max-set:

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

using remove-max-multiset[of \ l \ m \ l']
by (metis Un-insert-right local.set-def set-mset-add-mset-insert sup-bot-right)

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) ssort' where

\[ \text{sssort'} \ l \ s l = \]

\( (\text{if is-empty } l \text{ then } s l \]
else
let
\( (m, l') = \text{remove-max } l \]
in
\( \text{sssort'} \ l' (m \# s l) )

declare ssort'.simp[simp]
definition ssort :: 'a list ⇒ 'a list where
\[ \text{ssort } l = \text{ssort}' (\text{of-list } l) \]  

**inductive** \text{ssort}'-\text{dom} \text{ where}

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

**lemma** \text{ssort}'-termination:

\[ \text{assumes inv } (\text{fst } p) \]

\[ \text{shows ssort}'-\text{dom } p \]

**using** \text{assms}

**proof** (induct \( p \) rule: af-induct[of measure (\( \lambda (l, sl). \; \text{size } (\text{multiset } l)\))]  

**let** \( \exists r = \text{measure } (\lambda (l, sl). \; \text{size } (\text{multiset } l)) \)

**fix** \( p :: 'b \times 'a \text{ list} \)

**assume inv \( (\text{fst } p) \) and \( *: \)

\[ \forall y. \; (y, p) \in \exists r \implies \text{inv } (\text{fst } y) \implies \text{ssort}'-\text{dom } y \]

**obtain \( l \) \text{ sl} \text{ where } p = (l, sl) \]

**by** \text{(cases } p \text{) auto} 

**show ssort}'-\text{dom } p \]

**proof** (subst \( p = (l, sl) \), rule ssort}'-\text{dom.step})

**fix** \( m l' \)

**assume \( \neg \text{is-empty } l \; (m, l') = \text{remove-max } l \)

**show ssort}'-\text{dom } (l', m \# sl) \]

**proof** (rule *\text{rule-format}) 

**show \((l', m \# sl), p) \in \exists r \; \text{inv } (\text{fst } (l', m \# sl)) \]

**using** \( p = (l, sl); \; \text{inv } (\text{fst } p); \; (\neg \text{is-empty } l) \)

**using** \( (m, l') = \text{remove-max } l \)

**using** \text{remove-max-inv[of } l \text{ m } l'] \]

**using** \text{remove-max-multiset-size[of } l \text{ m } l'] \]

**by auto**

**qed**

**qed**

**simp**

**lemma** \text{ssort}'\text{Induct}:

\[ \text{assumes inv } l \; P \; l \; sl \]

\[ \land \; l \; sl \; m \; l'. \]

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

**shows** \( P \; \text{empty } (\text{ssort}' \; l \; sl) \)

**proof**

**from** \( \text{inv } l \) \text{ have ssort}'-\text{dom } (l, sl) \]

**using** \text{ssort}'-termination

**by** \text{auto}

**thus \( ?\text{thesis} \]

**using** \text{assms}

**proof** (induct \( (l, sl) \) arbitrary: \( l \; sl \) \text{ rule: ssort}'-\text{dom.induct})

**case** \( \text{step } l \; sl \)

**show \( ?\text{case} \]

**proof** (\text{cases is-empty } l)

**case** \text{True}

**thus \( ?\text{thesis} \]

8
using step(4) step(5) ssort'.simps[of l sl] is-empty-inj[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

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 l' m l''
    assume ¬ is-empty l
    inv l
    (m, l') = remove-max l
    multiset l + mset l' = multiset l + mset sl
  thus multiset l'' + mset (m # l'' sl) = multiset l + mset sl
  using remove-max-multiset[of l l m l'' sl]
  by (metis union-mset-add-mset-left union-mset-add-mset-right mset.simps(2))
  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
  qed


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

thus \[ \text{sorted } (m \# sl) \land (\forall x \in \text{set } l'. \forall y \in \text{List.set } (m \# sl). x \leq y) \]
using \[ \text{remove-max-set[of } l \text{ m l'] remove-max-max[of l m l']} \]
by (auto intro: Max-ge)

qed
thus \?thesis
by simp

qed

lemma \text{sorted-ssort: } \text{sorted (ssort } i) \]
unfolding \text{ssort-def}
using \text{sorted-ssort'[of of-list } i \text{ [] of-list-inv}
by auto

lemma \text{permutation-ssort: } \text{ssort } l \prec \sim \prec \sim l \]
proof (subst \text{mset-eq-perm}[\text{symmetric}])
show \text{mset (ssort } l) = \text{mset } l \]
unfolding \text{ssort-def}
using \text{mset-ssort'[of of-list } l \text{ []]}
using \text{multiset-of-list of-list-inv}
by simp

qed

end

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

sublocale \text{RemoveMax} < \text{Sort ssort}
by (unfold-locales) (auto simp add: \text{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 = [] \ \text{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.

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

definition remove max where remove max l = foldl f (hd l, []) (tl l)

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

The function really returned the maximum value.

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

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

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

lemma remove-max-mset-lemma:
assumes \( m, l' = \operatorname{foldl} f (m', t') l \)
shows \( \operatorname{mset} (m \# l') = \operatorname{mset} (m' \# t' @ l) \)
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) if ?m' ?t' m' t'
  by simp
thus ?case
  using snoc(2)
  apply (cases ?a)
  by (auto split: if-split-asm)
qed simp

lemma remove-max-mset:
  assumes l ≠ [] (m, l') = remove-max l
  shows add-mset m (mset l') = 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
  rewrites
  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
import 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 '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: if-split-asm)
```

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

The next step is to define function remove_max, but the question is weather implementation of remove_max depends on implementation of the functions is_heap and 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 +
fixed as-tree : 'b ⇒ 'a::linorder Tree
— This function is not very important, but it is needed in order to avoid problems with types and to detect that an observed object is a tree.

fixed 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] ⇒ add-mset m (multiset l') = 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

sublocale Heap <
RemoveMax empty is-empty of-list multiset remove-max 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 add-mset m (multiset l') = 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
qed
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:
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): (metis Tree.exhaust is-heap.simps(1) is-heap.simps(4) is-heap.simps(5)))
      hence is-heap (merge l1 (T v2 l2 r2))

qed
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
  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 r1
    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-eq-iff val.simps)
  next
next
case False
hence v1 ≥ val l1
using 3(3)
  by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
have ll (T v2 l2 r2) ≠ 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 ≠ E⟩ is-heap.simps(5))
obtain l1 lr1 lv1 where r1 = T lv1 ll1 lr1
  using ⟨r1 ≠ E⟩
  by (metis Tree.exhaust)
obtain r1 rr1 rv1 where merge l1 (T v2 l2 r2) = T rv1 rl1 rr1
  using ⟨merge l1 (T v2 l2 r2) ≠ E⟩
  by (metis Tree.exhaust)
have val (merge l1 (T v2 l2 r2)) ≤ v1
  using ⟨merge l1 (T v2 l2 r2) = val l1 ∨ val(merge l1 (T v2 l2 r2)) = v2⟩
  using ⟨v1 ≥ v2⟩ ⟨v1 ≥ val l1⟩
  by auto
hence is-heap (T v1 (merge l1 (T v2 l2 r2)) r1)
  using is-heap.simps(5)[of v1 lv1 ll1 lr1 rv1 rl1 rr1]
  using ⟨r1 = T lv1 ll1 lr1⟩ ⟨merge l1 (T v2 l2 r2) = T rv1 rl1 rr1⟩
  using ⟨is-heap r1⟩ ⟨is-heap (merge l1 (T v2 l2 r2))⟩ ⟨v1 ≥ val r1⟩
  by auto
thus ?thesis
  using ⟨v2 ≤ 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 (merge l2 (T v1 ll r1))
  using False ⟨is-heap (T v1 ll r1)⟩ 3
  by auto
have val (merge l2 (T v1 ll r1)) = val l2 ∨
  val(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 ?thesis
  proof(cases l2 = E)
case True
  hence merge \((T \, v_1 \, l_1 \, r_1) \, (T \, v_2 \, l_2 \, r_2)\) = \(T \, v_2 \, (T \, v_1 \, l_1 \, r_1)\) \(\text{E}\)
    using \((r_2 = \text{E}\) \((\neg \, v_2 \leq v_1))\)
    by auto
  thus \?thesis
    using 3
    using \((\neg \, v_2 \leq v_1))
    by auto
next
  case False
  hence \(v_2 \geq \text{val} \, l_2\)
    using 3
    by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  thus \?thesis
    using \((r_2 = \text{E}\) \((\neg \, v_2 \leq v_1))\)
    by (metis False Tree.exhaust is-heap.simps(2)
        is-heap.simps(4) linorder-linear merge.simps(3) val.simps)
  qed
next
  case False
  hence \(v_2 \geq \text{val} \, r_2\)
    using 3
    by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  show \?thesis
    proof (cases \(l_2 = \text{E}\))
      case True
      hence merge \((T \, v_1 \, l_1 \, r_1) \, (T \, v_2 \, l_2 \, r_2)\) = \(T \, v_2 \, (T \, v_1 \, l_1 \, r_1)\) \(\text{r_2}\)
        using \((\neg \, v_2 \leq v_1))
        by auto
      thus \?thesis
        using 3 \((v_2 \geq \text{val} \, r_2)\)
        using \((\neg \, v_2 \leq v_1))
        by (metis False Tree.exhaust Tree.simps(3) is-heap.simps(3)
            is-heap.simps(5) linorder-linear val.simps)
    next
      case False
      hence \(v_2 \geq \text{val} \, l_2\)
        using 3
        by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
      have merge \(l_2 \, (T \, v_1 \, l_1 \, r_1)\) \(\neq \text{E}\)
        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 \((r_2 \neq \text{E})\) is-heap.simps(5))
      obtain \(l_1 \, b_1 \, l_1 \, b_1\) \(\text{where} \, \text{r_2} = T \, l_1 \, l_1 \, b_1\)
using $\langle r_2 \neq E \rangle$

by (metis Tree.exhaust)

obtain rl1 rr1 rv1 where merge l2 $(T v_1 l_1 r_1) = T rv1 rl1 rr1$

using $\langle$merge l2 $(T v_1 l_1 r_1) \neq E \rangle$

by (metis Tree.exhaust)

have $\text{val} (\text{merge} l_2 (T v_1 l_1 r_1)) \leq v_2$

using $\langle\text{val} (\text{merge} l_2 (T v_1 l_1 r_1)) = \text{val} l_2 \lor$

val(merge l2 (T v_1 l_1 r_1)) = v_1\rangle$

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 l_1 l_1 l_1 r_1 r_1 r_1]

using $\langle r_2 = T lv1 ll1 lr1 \rangle$ (merge l2 $(T v_1 l_1 r_1) = T rv1 rl1 rr1$)

using (is-heap r2) (is-heap (merge l2 (T v_1 l_1 r_1)) $\langle v_2 \geq \text{val} r_2 \rangle$

by auto

thus ?thesis

using $\langle \neg v_2 \leq v_1 \rangle$

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
[simp]: hs-is-empty t $\iff t = E$

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)
case (1 l)
thus $?case$
by auto

next
case (2 g)
thus $?case$
by auto

next
case (3 v_1 l_1 r_1 v_2 l_2 r_2)
thus ?case
proof (cases v2 \leq v1)
  case True
  hence \( \text{multiset} (\text{merge} \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2)) = \)
    \{\#v1\#\} + \text{multiset} (\text{merge} \ l1 \ (T \ v2 \ l2 \ r2)) + \text{multiset} \ r1
    by auto
  hence \( \text{multiset} (\text{merge} \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2)) = \)
    \{\#v1\#\} + \text{multiset} \ l1 + \text{multiset} \ (T \ v2 \ l2 \ r2) + \text{multiset} \ r1
    using 3 True
    by (metis union-assoc)
  hence \( \text{multiset} (\text{merge} \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2)) = \)
    \{\#v1\#\} + \text{multiset} \ l1 + \text{multiset} \ r1 + \text{multiset} \ (T \ v2 \ l2 \ r2)
    by (metis union-commute union-lcomm)
  thus \?thesis
    by auto
next
  case False
  hence \( \text{multiset} (\text{merge} \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2)) = \)
    \{\#v2\#\} + \text{multiset} \ l2 + \text{multiset} \ r2 + \text{multiset} \ (T \ v1 \ l1 \ r1)
    using 3 False
    by (metis union-commute union-lcomm)
  hence \?thesis
    by (metis multiset.simps(2) union-commute)
qed
qed

Proof that defined functions are interpretation of abstract functions from locale \textit{Collection}:

\textbf{interpretation HS:} Collection \( E \) \hs-is-empty \hs-of-list multiset

\textbf{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 \( l \))
      case Nil
      thus \?case
by auto
next
case (Cons a l)
  have multiset (hs-of-list (a ≠ l)) = multiset (hs-of-list l) + { #a# }
  using merge-multiset[of hs-of-list l T a E E]
  apply auto
  unfolding insert-def
  by auto
  thus ?case
  using Cons
  by auto
qed
qed

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-heap-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'

22
7 Verification of Imperative Heap Sort

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

hence in-tree v (siftDown (set-val (T v1 l1 r1) v1)) using 5-1(3) (¬ in-tree v (T v2 l2 r2), (¬ v1 ≥ v2), (¬ v' ≥ v2)) by auto

hence in-tree v (T v' l2 r2) using 5-1(2) (¬ v1 ≥ v2), (¬ v' ≥ v2) by auto

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

thus ?thesis by auto

qed

qed

qed

qed

next
case (5-2 v' v1 l1 r1 v2 l2 r2)
show ?case

proof (cases v = v' ∨ v = v1 ∨ v = v2)

case True

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) 5-2 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) v1)) using 5-2(3) (¬ in-tree v (T v2 l2 r2), (v1 ≥ v2), (¬ v' ≥ v1)) using (¬ (v = v' ∨ v = v1 ∨ v = v2)) by auto

hence in-tree v (T v' l1 r1)
using 5-2(1) \langle v_1 \geq v_2 \rangle \langle \neg v' \geq v_1 \rangle
by auto

hence in-tree v (T v_1 l_1 r_1)
using \langle \neg (v = v' \lor v = v_1 \lor v = v_2) \rangle
by auto

thus \?thesis
by auto

qed
qed

next
case False
show \?thesis
proof(cases v' \geq v_2)
case True
thus \?thesis
using \langle \neg v_1 \geq v_2 \rangle 5-2
by auto

next
case False
thus \?thesis
proof(cases in-tree v (T v_1 l_1 r_1))
case True
thus \?thesis
by auto

next
case False
hence in-tree v (siftDown (set-val (T v_2 l_2 r_2) v'))
using 5-2(3) \langle \neg in-tree v (T v_1 l_1 r_1) \rangle \langle \neg v_1 \geq v_2 \rangle \langle \neg v' \geq v_2 \rangle
using \langle \neg (v = v' \lor v = v_1 \lor v = v_2) \rangle
by auto

hence in-tree v (T v' l_2 r_2)
using 5-2(2) \langle \neg v_1 \geq v_2 \rangle \langle \neg v' \geq v_2 \rangle
by auto

hence in-tree v (T v_2 l_2 r_2)
using \langle \neg (v = v' \lor v = v_1 \lor v = v_2) \rangle
by auto

thus \?thesis
by auto

qed
qed
qed
qed
qed
qed

lemma siftDown-heap-is-heap:
assumes is-heap l is-heap r t = T v l r
shows is-heap (siftDown t)
using assms
proof (induct t arbitrary: v l r rule: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 ≥ v1)
      case True
      thus ?thesis
        using 3(2) 3(4)
        by auto
      next
      case False
      show ?thesis
        proof
          let ?t = siftDown (T v2 l1 r1)
          obtain l' v' r' where *: ?t = T v' l' r' v' ≥ v2
            using siftDown-Node[of T v2 l1 r1 v2 l1 r1]
            by auto
          have l = T v1 l1 r1
            using 3(4)
            by auto
          hence is-heap l1 is-heap r1
            using 3(2)
            apply (induct l rule:is-heap.induct)
            by auto
          hence is-heap ?t
            using 3(1)[of l1 r1 v2] False 3
            by auto
          show ?thesis
          proof (cases v' = v2)
            case True
            thus ?thesis
              using False (is-heap ?t) *
              by auto
          next
          case False
          have in-tree v' ?t
            using *
            using siftDown-in-tree[of ?t]
            by simp
          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 = 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 $\ast$ 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 $v1 \geq v'$ using is-heap-max[of $v'$ $T v1 l1 r1$] 4 by auto
thus ?thesis using (is-heap $?t$) False $\ast$
by auto
qed

due

next
case (5-1 $v1$ $v2$ $l2$ $r2$ $v3$ $l3$ $r3$)
show ?case
proof(cases $v2 \geq v3$)
case True
show ?thesis
proof(cases $v1 \geq v2$)
case True
thus ?thesis using ($v2 \geq v3$; 5-1)
by auto
next
case False
let $?t = siftDown (T v1 l2 r2)$
obtain $l' v' r' where *: $?t = T v' l' r' v' \geq v1$
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 $l2$ $r2$ $v1$] ($v2 \geq v3$) False
by auto
have $v2 \geq v'$
proof(cases $v' = v1$)
case True
thus ?thesis
using False
by auto
next
case False
have in-tree $v'$ $?t$
using $\ast$ siftDown-in-tree
by auto
hence in-tree \( v' (T v1 l2 r2) \)
  using siftDown-in-tree-set[\( v' T v1 l2 r2 \)]
  by auto
hence in-tree \( v' (T v2 l2 r2) \)
  using False
  by auto
thus \(?thesis\)
  using is-heap-max[\( v' T v2 l2 r2 \)] 5-1
  by auto
qed
thus \(?thesis\)
  using \( \langle \text{is-heap } \neg v2 \geq v3 \rangle \ast False \) 5-1
  by auto
qed
next
case False
show \(?thesis\)
proof(cases \( v1 \geq v3 \))
case True
thus \(?thesis\)
  using \( \langle \neg v2 \geq v3 \rangle \ast 5-1 \)
  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 \( \text{is-heap } l3 \) \( \text{is-heap } r3 \)
  using 5-1(4, 5)
  apply(induct r rule:is-heap.induct)
  by auto
hence \( \text{is-heap } ?t \)
  using 5-1(2)[\( l3 r3 v1 \)] \( \langle \neg v2 \geq v3 \rangle \ast False \)
  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 \( \ast \) 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 \textit{in-tree} \(v'(T v3 l3 r3)\)

using \textit{False}

by \textit{auto}

thus \(\textit{thesis}\)

using \textit{is-heap-max}[of \(v'(T v3 l3 r3)]\textit{5-1}\)

by \textit{auto}

qed

thus \(\textit{thesis}\)

using \textit{⟨is-heap \(\langle\neg \ v2 \geq v3\rangle\) * False 5-1}\)

by \textit{auto}

qed

qed

next

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

show \textit{?case}\n

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

case \textit{True}\n

show \textit{?thesis}\n

proof\((\text{cases} \ v1 \geq v2)\)

case \textit{True}\n

thus \(\textit{thesis}\)

using \(\langle v2 \geq v3\rangle\) \textit{5-2}\n

by \textit{auto}\n

next\n
case \textit{False}\n

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

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

using \textit{siftDown-Node}

by \textit{blast}\n

have \textit{is-heap} \textit{l2} \textit{is-heap} \textit{r2}

using \textit{5-2}(3, 5)

apply\((\text{induct} \ l \ \text{rule:is-heap.induct})\)

by \textit{auto}\n

hence \textit{is-heap} \(\ ?t\)

using \textit{5-2}(1)[of \textit{l2} \textit{r2} \textit{v1}] (\textit{v2} \geq \textit{v3}) \textit{False}\n
by \textit{auto}\n

have \textit{v2} \geq \textit{v'}\n

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

case \textit{True}\n

thus \(\textit{thesis}\)

using \textit{False}\n

by \textit{auto}\n

next\n
case \textit{False}\n

have \textit{in-tree} \(v' ?t\)

using \textit{siftDown-in-tree}\n

by \textit{auto}\n

hence \textit{in-tree} \(v'(T v1 l2 r2)\)

using \textit{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] 5-2
  by auto
qed
thus ?thesis
  using ⟨is-heap ?t⟩ ⟨v2 ≥ v3⟩ * False 5-2
  by auto
qed
next
case False
show ?thesis
proof(cases v1 ≥ v3)
case True
thus ?thesis
  using ⟨¬ v2 ≥ v3⟩ 5-2
  by auto
next
case False
let ?t = siftDown (T v1 l3 r3)
obtain l’ v’ r’ where *: ?t = T v’ l’ r’ v’ ≥ 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] ⟨¬ v2 ≥ v3⟩ False
  by auto
have v3 ≥ 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[of v’ T v1 l3 r3]
  by auto
hence in-tree v’ (T v3 l3 r3)
  using False

35
by auto
thus ?thesis
  using is-heap-max[of v' T v3 l3 r3] 5-2
by auto
qed
thus ?thesis
  using ⟨is-heap ⟨t⟩ ⟨¬ v2 ≥ v3⟩ * False 5-2
by auto
qed
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 ?case
  by auto
next
case (T v l r)
  thus ?case
  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 ⇒ 'a × 'a Tree where
  removeLeaf 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 ⇒ '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 \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}.

\begin{verbatim}
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'))))
\end{verbatim}

\begin{verbatim}
definition hs-is-empty where
[simp]: hs-is-empty t ←→ t = E
\end{verbatim}

\begin{verbatim}
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
moreover
  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-comm)
ultimately
  show ?thesis
\end{verbatim}
by auto

qed

case (4 v1 v l r)

proof (cases v ≤ v1)

  case True
  thus thesis
  by auto

next

case False

have multiset (set-val (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 + {#v1#} + 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

case (5-1 v v1 l1 r1 v2 l2 r2)

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 \( \text{multiset} \ (\text{siftDown} \ (T \ v \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2))) = \)
\( \text{multiset} \ l1 + \{\#v\#\} + \text{multiset} \ r1 + \{\#v1\#\} + \text{multiset} \ (T \ v2 \ l2 \ r2) \)
\( \text{using } \langle v1 \geq v2 \rangle \ 5\text{-1}(1) \)
by \text{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 \text{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 \text{auto} 
qed 
next 
case False 
show \(?\text{thesis}\)
proof \(\text{(cases } v \geq v2)\)
  case True 
  thus \(?\text{thesis}\)
  using False 
  by \text{auto} 
next 
case False 
  hence \( \text{multiset} \ (\text{siftDown} \ (T \ v \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2))) = \)
\( \text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v1\#\} + \text{multiset} \ l2 + \{\#v\#\} + \text{multiset} \ r2 \)
\( \text{using } \langle \neg v1 \geq v2 \rangle \ 5\text{-1}(2) \)
by \text{(simp add: ac-simps)}
moreover 
have 
\( \text{multiset} \ (T \ v \ (T \ v1 \ l1 \ r1) \ (T \ v2 \ l2 \ r2)) = \)
\( \text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v\#\} + \text{multiset} \ l2 + \{\#v2\#\} + \text{multiset} \ r2 \)
by \text{simp} 
moreover 
have 
\( \text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v\#\} + \text{multiset} \ l2 + \{\#v2\#\} + \text{multiset} \ r2 = \)
\( \text{multiset} \ (T \ v1 \ l1 \ r1) + \{\#v2\#\} + \text{multiset} \ l2 + \{\#v\#\} + \text{multiset} \ r2 \)
by \text{(metis union-commute union-lcomm)}
ultimately
show \( ?\text{thesis} \)
  
  by auto

qed

qed

next

case (5-2 \( v \) \( v1 \) \( l1 \) \( r1 \) \( v2 \) \( l2 \) \( r2 \))

thus \( ?\text{thesis} \)

proof(cases \( v1 \geq v2 \))

  case True

  thus \( ?\text{thesis} \)

  proof(cases \( v \geq v1 \))

    case True

    thus \( ?\text{thesis} \)

    using \( \langle v1 \geq v2 \rangle \)

    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 \( \langle v1 \geq v2 \rangle \)

by auto

moreover

have multiset (T \( v \) (T \( v1 \) \( l1 \) \( r1 \)) (T \( v2 \) \( l2 \) \( r2 \))) =

  multiset \( l1 \) + \( \{\#v\#\} \) + 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 \( ?\text{thesis} \)

by auto

qed

next

case False

show \( ?\text{thesis} \)

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

  case True

  thus \( ?\text{thesis} \)

  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 \( l2 \) + \( \{\#v\#\} \) +
using \( \langle \neg v_1 \geq v_2 \rangle \cdot 5 - 2(2) \)
by (simp add: ac-simps)
moreover
have multiset \( (T \, v \, (T \, v_1 \, l_1 \, r_1) \, (T \, v_2 \, l_2 \, r_2)) = \)
multiset \( (T \, v_1 \, l_1 \, r_1) + \{\#v\#\} + \operatorname{multiset} l_2 + \{\#v_2\#\} + \) 
multiset \( r_2 \)
by simp
moreover
have multiset \( (T \, v_1 \, l_1 \, r_1) + \{\#v\#\} + \operatorname{multiset} l_2 + \{\#v_2\#\} + \) 
multiset \( r_2 = \)
multiset \( (T \, v_1 \, l_1 \, r_1) + \{\#v_2\#\} + \operatorname{multiset} l_2 + \{\#v\#\} + \) 
multiset \( r_2 \)
by (metis union-commute union-comm)
ultimately
show \( \text{thesis} \)
by auto
qed

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

lemma multiset-heapify:
multiset \( (\text{heapify} \, t) = \text{multiset} \, t \)
proof (induct \( t \))
case E
thus \( ?\text{case} \)
by auto
next
case (T \( v \, l \, r \))
hence multiset \( (\text{heapify} \, (T \, v \, l \, r)) = \text{multiset} \, l + \{\#v\#\} + \text{multiset} \, r \)
using siftDown-multiset[of T \( v \) (heapify \( l \)) (heapify \( r \))]
by auto
thus \( ?\text{case} \)
by auto
qed

lemma multiset-heapify-of-list-tree:
  \( \text{multiset} \left( \text{heapify} \left( \text{of-list-tree} \ l \right) \right) = \text{mset} \ l \)
using multiset-heapify[of of-list-tree \ l]
using mset-list-tree[of \ l]
by auto

lemma 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 assms
apply (induct \ t \ rule:removeLeaf.induct)
by auto

lemma removeLeaf-heap-is-heap:
  assumes \( \text{is-heap} \ t \ t \neq E \)
  shows \( \text{is-heap} \ (\text{snd} \ (\text{removeLeaf} \ t)) \)
using assms
proof (induct \ t \ 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 (cases \( ?t = E \))
  case True
  thus \( ?\text{thesis} \)
  by auto
next
case False
have \( v \geq v1 \)
  using 2(3)
  by auto
hence \( v \geq \text{val} \ ?t \)
  using False removeLeaf-val-val[of \ T \ v1 \ l1 \ r1]
  by auto
hence \( \text{is-heap} \ (T \ v \ (\text{snd} \ (\text{removeLeaf} \ (T \ v1 \ l1 \ r1))) \ E) \)
using (is-heap (snd (removeLeaf (T v1 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 is-heap (T v1 l1 r1)
  using 3(3)
  by auto
hence is-heap (snd (removeLeaf (T v1 l1 r1)))
  using 3(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 3(2)
  by auto
hence v ≥ val ?t
  using False removeLeaf-val-val[of T v1 l1 r1]
  by auto
hence is-heap (T v E (snd (removeLeaf (T v1 l1 r1))))
  using (is-heap (snd (removeLeaf (T v1 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 is-heap (T v1 l1 r1) is-heap (T v2 l2 r2) v ≥ v1 v ≥ v2
  using 4-1(3)
  by (simp add:is-heap.simps(5))+
hence is-heap (snd (removeLeaf (T v1 l1 r1)))
  using 4-1(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 ≥ v2)
by auto
next
case False
then obtain \( v' \ l' \ r' \) where \( ?t = T v' \ l' \ r' \)
by (metis Tree.exhaust)
hence is-heap \( (T v' \ l' \ r') \)
using is-heap \( (\text{snd (removeLeaf (T v l r)))} \)
by auto
have \( v \geq v_1 \)
using 4-1(3)
by auto
hence \( v \geq v_1 \)
using False removeLeaf-val-val[of \( T v l r \)]
by auto
hence \( v \geq v_1 \)
using \( ?t = T v' \ l' \ r' \)
by auto
hence is-heap \( (T v (T v' \ l' \ r') (T v_2 l_2 r_2)) \)
using is-heap \( (T v' \ l' \ r') \)
using is-heap \( (T v_2 l_2 r_2) \) \( (v \geq v_2) \)
by (simp add: is-heap.simps(5))
thus \( \text{thesis} \)
using 4-1 \( (?t = T v' \ l' \ r') \)
by auto
qed
next
case \( (4-2 v v_1 l_1 r_1 v_2 l_2 r_2) \)
have is-heap \( (T v_1 l_1 r_1) \) is-heap \( (T v_2 l_2 r_2) \) \( v \geq v_1 \) \( v \geq v_2 \)
using 4-2(3)
by (simp add:is-heap.simps(5))
next
case True
thus \( \text{thesis} \)
using is-heap \( (T v_2 l_2 r_2) \) \( (v \geq v_2) \)
by auto
next
case False
then obtain \( v' \ l' \ r' \) where \( ?t = T v' \ l' \ r' \)
by (metis Tree.exhaust)
hence is-heap \( (T v' \ l' \ r') \)
using is-heap \( (\text{snd (removeLeaf (T v l r)))}\)
by auto
have \( v \geq v_1 \)
using 4-2(3)
by auto

hence $v \geq \text{val } \gamma$
  using False removeLeaf-val-val[of $T v l l1 r1$]
  by auto

hence $v \geq v1'$
  using $(\gamma t = T v1' l1' r1')$
  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 $(T v2 l2 r2)$ $(v \geq v2)$)
  by (simp add: is-heap.simps(5))

thus $\gamma$thesis
  using 4-2 $(\gamma t = T v1' l1' r1')$
  by auto

qed

next
case 5

thus $\gamma$case
  by auto

qed

Defined functions satisfy conditions of locale \textit{Collection} and thus represent
interpretation of this locale.

\textbf{interpretation HS:} Collection \textit{E} hs-is-empty hs-of-list multiset

\textbf{proof}

fix $t$

assume hs-is-empty $t$

thus $t = \text{E}$
  by auto

next

show hs-is-empty $\text{E}$
  by auto

next

show multiset $E = \{\#\}$
  by auto

next

fix $l$

show multiset $(hs-of-list \ l) = \text{mset} \ l$
  unfolding hs-of-list-def
  using multiset-heapify-of-list-tree[of $l$]
  by auto

qed

\textbf{lemma removeLeaf-multiset:}

\textbf{assumes} $(v', t') = \text{removeLeaf } t \ t \neq E$

\textbf{shows} $\{\#v'\#\} + \text{multiset } t' = \text{multiset } t$

\textbf{using} \text{assms}

\textbf{proof}(induct $t$ arbitrary: $v'$ $t'$ rule:removeLeaf.induct)

case 1

45
thus ?case
    by auto
next
  case (2 v vl l1 r1)
  have \( t' = T v \) \((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) \( E \)
      using 2(3)
      by auto
  have \( v' = fst \ (removeLeaf \ (T \ v l1 l1 \ r1)) \)
      using 2(3)
      by auto
  hence \( \#v'\# \) + \multiset t' =
      \{\#fst \ (removeLeaf \ (T \ v l1 l1 \ r1))\#\} +
      \multiset \ ((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) +
      \{\#v\#\}
      using \((t' = T v \) \((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) \( E)\)
      by (simp add: ac-simps)
  have \( \#fst \ (removeLeaf \ (T \ v l1 l1 \ r1))\#\) +
      \multiset \ ((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) =
      \multiset \ (T \ v1 l1 l1 \ r1)
      using 2(1)
      by auto
  hence \( \#v'\# \) + \multiset t' = \multiset \ (T \ v1 l1 l1 \ r1) + \{\#v\#\}
      by auto
  thus ?case
      by auto
next
  case (3 v vl l1 r1)
  have \( t' = T v \) \((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\)
      using 3(3)
      by auto
  have \( v' = fst \ (removeLeaf \ (T \ v l1 l1 \ r1)) \)
      using 3(3)
      by auto
  hence \( \#v'\# \) + \multiset t' =
      \{\#fst \ (removeLeaf \ (T \ v l1 l1 \ r1))\#\} +
      \multiset \ ((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) +
      \{\#v\#\}
      using \((t' = T v \) \((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) \( E)\)
      by (simp add: ac-simps)
  have \( \#fst \ (removeLeaf \ (T \ v l1 l1 \ r1))\#\) +
      \multiset \ ((snd \ (removeLeaf \ (T \ v l1 l1 \ r1)))\) =
      \multiset \ (T \ v1 l1 l1 \ r1)
      using 3(1)
      by auto
  hence \( \#v'\# \) + \multiset t' = \multiset \ (T \ v1 l1 l1 \ r1) + \{\#v\#\}
      by auto
  hence \( \#v'\# \) + \multiset t' = \multiset \ (T \ v1 l1 l1 \ r1) + \{\#v\#\}
      using \{(\#v'\#\) + \multiset t' =

\[
\{ \# \text{fst} (\text{removeLeaf} (T \text{ v1 l1 r1})) \# \} + \\
\text{multiset} (\text{snd} (\text{removeLeaf} (T \text{ v1 l1 r1}))) + \{ \# v \# \};
\]

by auto

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

next

case (4-1 v v1 l1 r1 v2 l2 r2)

have \( t' = T \ v \ (\text{snd} (\text{removeLeaf} (T \text{ v1 l1 r1}))) \ (T \text{ v2 l2 r2}) \)
using 4-1(3)
by auto

have \( v' = \text{fst} (\text{removeLeaf} (T \text{ v1 l1 r1})) \)
using 4-1(3)
by auto

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

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

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

thus ?case 
by auto

next

case (4-2 v v1 l1 r1 v2 l2 r2)

have \( t' = T \ v \ (\text{snd} (\text{removeLeaf} (T \text{ v1 l1 r1}))) \ (T \text{ v2 l2 r2}) \)
using 4-2(3)
by auto

have \( v' = \text{fst} (\text{removeLeaf} (T \text{ v1 l1 r1})) \)
using 4-2(3)
by auto

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

have \( \{ \# \text{fst} (\text{removeLeaf} (T \text{ v1 l1 r1})) \# \} + \\
\} \)
\[
\text{multiset (snd (removeLeaf (T v1 l1 r1))) = multiset (T v1 l1 r1)}
\]
using 4-2(1)
by auto
hence \{#v'\#\} + multiset t' =
\[
\text{multiset (T v1 l1 r1) + \{#v\#\} + multiset (T v2 l2 r2)}
\]
using \{#v'\#\} + multiset t' =
\[
\{\text{fst (removeLeaf (T v1 l1 r1))}\#\} + \text{multiset (snd (removeLeaf (T v1 l1 r1)))} + \{#v\#\} + \text{multiset (T v2 l2 r2)}
\]
by auto
thus ?case
by auto
next
case 5
thus ?case
by auto
qed

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

lemma hs-remove-max-multiset:
assumes \((m, t') = hs\text{-remove-max} t t \neq E\)
shows \{#m\#\} + multiset t' = multiset t
proof-
let \(?v1 = \text{fst (removeLeaf} t)\)
let \( ?t1 = \text{snd} \ (\text{removeLeaf} \ t) \)

show \( \text{thesis} \)

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

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[hs-of-list-tree t]
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

qed
qed
using \((\text{is-heap}\ ?t1)\)

by \((\text{auto, metis (full-types)}\ \text{Tree.exhaust})\)
\(\text{is-heap.simps(1) is-heap.simps(4) is-heap.simps(5)}\)
\((\text{metis (full-types)}\ \text{Tree.exhaust})\)
\(\text{is-heap.simps(1) is-heap.simps(3) is-heap.simps(5)}\)

have \(\text{set-val}\ ?t1 \ ?v1 = T \ ?v1\ l-t1\ r-t1\)
using \((?t1 = T\ v-t1\ l-t1\ r-t1)\)
by \(\text{auto}\)

hence \(\text{is-heap}\ (\text{siftDown}\ (\text{set-val}\ ?t1\ ?v1))\)
using \((\text{is-heap}\ l-t1)\ :\ \text{is-heap}\ r-t1\)
using \(\text{siftDown-heap-is-heap[of}\ l-t1\ r-t1\ \text{set-val}\ ?t1\ ?v1\ ?v1\)
by \(\text{auto}\)

have \(t' = \text{siftDown}\ (\text{set-val}\ ?t1\ ?v1)\)
using \((\langle v', t' \rangle = \text{hs-remove-max} t) = \text{False}\)
by \(\text{(auto simp add: hs-remove-max-def)}\ (\text{metis prod.inject})\)

thus \(?\text{thesis}\)
using \((\text{is-heap}\ (\text{siftDown}\ (\text{set-val}\ ?t1\ ?v1)))\)
by \(\text{auto}\)

qed

down

next

fix \(t\ m\ t'\)

let \(?t1 = \text{snd}\ (\text{removeLeaf}\ t)\)

assume \(\neg\ \text{hs-is-empty}\ t\ (m, t') = \text{hs-remove-max} t\)

hence \(m = \text{val}\ t\)

apply \((\text{simp add: hs-remove-max-def})\)
apply \((\text{cases} ?t1 = E)\)
by \(\text{(auto, metis prod.inject)}\)

thus \(m = \text{val}\ (\text{id}\ t)\)
by \(\text{auto}\)

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. Following 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, anamorphism, hylomorphism and paramorphisms. Selection, buble, 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 sorting 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: consistency, 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 described in [?] with example of sorting algorithms. Refinement is not supported 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 improvements 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 formal-
izing 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.