Verification of Selection and Heap Sort Using Locales

Danijela Petrović

April 17, 2016

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

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.

$$sorted\ (sort\ \ l)$$

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

$$sort\ l\ \ <\sim\sim>\ \ l$$

locale *Sort* =
  fixes *sort* :: *'a*::*linorder list* ⇒ *'a list*
  assumes *sorted*: *sorted* (*sort l*)
  assumes *permutation*: *sort l* <~~> *l*

end

3 Defining data structure and key function remove_max

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

— – It must be assured that the empty element is *empty*
assumes *is-empty-empty*: *is-empty empty*
— – Must be satisfied that function *is_empty* returns true for element *empty*
assumes *multiset-empty*: *multiset empty* = {#}
— – Multiset of an empty object is empty multiset.
assumes *multiset-of-list*: *multiset* (*of-list i*) = *mset i*
— – Multiset of an object gained by applying function *of_list* must be the same as the multiset of the list. This, practically, means that function *of_list* does not delete or change elements of the starting list.
begin
lemma *is-empty-as-list*: *is-empty e* ⟹ *multiset e* = {#}
using *is-empty-inj multiset-empty*
by *auto*

definition *set* :: 'b ⇒ '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 ⇒ *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*:
$[\![\neg$ *is-empty* l; *inv* l; $(m,\ l') =$ *remove-max* $l]\!] \Longrightarrow m = Max\ (set\ l)$
— — First parameter of the return value of the function *remove_max* is the maximum element
assumes *remove-max-multiset*:
$[\![\neg$ *is-empty* l; *inv* l; $(m,\ l') =$ *remove-max* $l]\!] \Longrightarrow$
multiset $l' + \{\#m\#\} =$ *multiset* l
— — Condition for multiset, ensures that nothing new is added or nothing is lost after applying *remove_max* function.
assumes *remove-max-inv*:
$[\![\neg$ *is-empty* l; *inv* l; $(m,\ l') =$ *remove-max* $l]\!] \Longrightarrow$ *inv* l'
— — Ensures that invariant condition is true after removing maximum element. Invariant condition must be true in each step of sorting algorithm, for example if we are sorting using heap than in each iteration we must have heap and function *remove_max* must not change that.
begin

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

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

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

partial-function (*tailrec*) *ssort'* **where**
ssort' l sl =
(*if is-empty l then*
sl
else
let
$(m,\ l') =$ *remove-max* l
in
ssort' l' $(m\ \#\ sl)$)
declare *ssort'.simps*[*code*]
definition *ssort* :: *'a list* ⇒ *'a list* **where**

```
  ssort l = ssort′ (of-list l) []

inductive ssort′-dom where
   step: ⟦⋀m l′. ⟦¬ is-empty l; (m, l′) = remove-max l⟧ ⟹
                 ssort′-dom (l′, m # sl)⟧ ⟹ ssort′-dom (l, sl)
lemma ssort′-termination:
  assumes inv (fst p)
  shows ssort′-dom p
using assms
proof (induct p rule: wf-induct[of measure (λ(l, sl). size (multiset l))])
  let ?r = measure (λ(l, sl). size (multiset l))
  fix p :: ′b × ′a list
  assume inv (fst p) and ∗:
         ∀ y. (y, p) ∈ ?r ⟶ inv (fst y) ⟶ ssort′-dom y
  obtain l sl where p = (l, sl)
    by (cases p) auto
  show ssort′-dom p
  proof (subst ‹p = (l, sl)›, rule ssort′-dom.step)
    fix m l′
    assume ¬ is-empty l (m, l′) = remove-max l
    show ssort′-dom (l′, m#sl)
    proof (rule ∗[rule-format])
      show ((l′, m#sl), p) ∈ ?r inv (fst (l′, m#sl))
        using ‹p = (l, sl)› ‹inv (fst p)› ‹¬ is-empty l›
        using ‹(m, l′) = remove-max l›
        using remove-max-inv[of l m l′]
        using remove-max-multiset-size[of l m l′]
        by auto
    qed
  qed
qed simp

lemma ssort′Induct:
  assumes inv l P l sl
   ⋀ l sl m l′.
   ⟦¬ is-empty l; inv l; (m, l′) = remove-max l; P l sl⟧ ⟹ P l′ (m # sl)
  shows P empty (ssort′ l sl)
proof−
  from ‹inv l› have ssort′-dom (l, sl)
    using ssort′-termination
    by auto
  thus ?thesis
    using assms
  proof (induct (l, sl) arbitrary: l sl rule: ssort′-dom.induct)
    case (step l sl)
    show ?case
    proof (cases is-empty l)
      case True
      thus ?thesis
```

```
      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
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 l1 sl1 m l'
    assume ¬ is-empty l1
           inv l1
           (m, l') = remove-max l1
           multiset l1 + mset sl1 = multiset l + mset sl
    thus multiset l' + mset (m # sl1) = multiset l + mset sl
      using remove-max-multiset[of l1 m l']
      by (auto simp add: union-commute union-lcomm)
  qed simp
  thus ?thesis
    using multiset-empty
    by simp
qed

lemma sorted-ssort':
 assumes inv l sorted sl ∧ (∀ x ∈ set l. (∀ y ∈ List.set sl. x ≤ y))
 shows sorted (ssort' l sl)
using assms
proof−
 have sorted (ssort' l sl) ∧
      (∀ x ∈ set empty. (∀ y ∈ List.set (ssort' l sl). x ≤ y))
   using assms
 proof (rule ssort'Induct)
   fix l sl m l'
   assume ¬ is-empty l
```

```
      inv l
      (m, l') = remove-max l
      sorted sl ∧ (∀ x∈set l. ∀ y∈List.set sl. x ≤ y)
  thus sorted (m # sl) ∧ (∀ x∈set l'. ∀ y∈List.set (m # sl). x ≤ y)
   using remove-max-set[of l m l'] remove-max-max[of l m l']
   by (auto simp add: sorted-Cons intro: Max-ge)
 qed
 thus ?thesis
  by simp
qed

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

lemma permutation-ssort: ssort l <~~> l
proof (subst mset-eq-perm[symmetric])
 show mset (ssort l) = mset l
  unfolding ssort-def
  using mset-ssort'[of of-list l []]
  using multiset-of-list of-list-inv
  by simp
qed
end
```

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

```
sublocale RemoveMax < Sort ssort
by (unfold-locales) (auto simp add: sorted-ssort permutation-ssort)

end
```

4 Verification of functional Selection Sort

```
theory SelectionSort-Functional
imports RemoveMax
begin
```

4.1 Defining data structure

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

```
interpretation Collection [] λ l. l = [] id mset
by (unfold-locales, auto)
```

4.2 Defining function remove_max

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

fun *f* **where** $f\ (m,\ l)\ x = (if\ x \geq m\ then\ (x,\ m\#l)\ else\ (m,\ x\#l))$

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

lemma *max-Max-commute*:
  $finite\ A \Longrightarrow max\ (Max\ (insert\ m\ A))\ x = max\ m\ (Max\ (insert\ x\ A))$
  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 *fst (foldl f (m, t) l) = Max (set (m # l))*
using *assms*
proof (*induct l arbitrary*: *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 *fst (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 ≠ [] (m, l′) = remove-max l*
  shows *m = Max (set l)*
using *assms*
unfolding *remove-max-def*
using *remove-max-max-lemma[of hd l [] tl l]*
using *fst-conv[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′) = foldl f (m′, t′) l*
  shows *mset (m # l′) = 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*)[*of ?m' ?t' m' t'*]
    by *simp*
  thus *?case*
    using *snoc*(*2*)
    apply (*cases ?a*)
    apply (*auto split*: *if-split-asm*, (*simp add*: *union-lcomm union-commute*)+)
    by (*metis union-assoc*)
qed *simp*

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

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

interpretation *SSRemoveMax*:
  RemoveMax [] λ *l*. *l* = [] *id mset remove-max* λ -. *True*
  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*
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 '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*.

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:

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

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*:

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*)

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*)

lemma *heap-top-max*:
assumes *t* ≠ *E is-heap t*
shows *val t* = *Max* (*set-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 :: $'b \Rightarrow$ *bool* **and**
of-list :: $'a$::*linorder list* $\Rightarrow$ $'b$ **and**
multiset :: $'b \Rightarrow$ $'a$::*linorder multiset* +
fixes *as-tree* :: $'b \Rightarrow$ $'a$::*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* :: $'b \Rightarrow 'a \times 'b$
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* $\longleftrightarrow$ *is-empty t*
assumes *remove-max-multiset'*:
$[\![\neg$ *is-empty l*; (m, l') = *remove-max l*$]\!] \Longrightarrow$ *multiset* l' + {#m#} = *multiset l*
assumes *remove-max-is-heap*:
$[\![\neg$ *is-empty l*; *is-heap* (*as-tree l*); (m, l') = *remove-max l*$]\!] \Longrightarrow$
is-heap (*as-tree* l')
assumes *remove-max-val*:
$[\![\neg$ *is-empty t*; (m, t') = *remove-max t*$]\!] \Longrightarrow$ 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 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 $\neg$ *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 $\neg$ *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 $\neg$ *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)›
```

```
  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
  show ?thesis
  proof(cases l1 = E)
    case True
    hence merge (T v1 l1 r1) (T v2 l2 r2) = T v1 (T v2 l2 r2) E
      using ‹r1 = E› ‹v2 ≤ v1›
      by auto
    thus ?thesis
      using 3
      using ‹v2 ≤ v1›
      by auto
  next
    case False
    hence v1 ≥ val l1
      using 3(3)
      by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
    thus ?thesis
      using ‹r1 = E› ‹v1 ≥ v2›
      using ‹val (merge l1 (T v2 l2 r2)) = val l1
                ∨ val(merge l1 (T v2 l2 r2)) = v2›
      using ‹is-heap (merge l1 (T v2 l2 r2))›
      by (metis False Tree.exhaust is-heap.simps(2)
          is-heap.simps(4) merge.simps(3) val.simps)
  qed
next
  case False
  hence v1 ≥ val r1
    using 3(3)
    by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
  show ?thesis
  proof(cases l1 = E)
    case True
    hence merge (T v1 l1 r1) (T v2 l2 r2) = T v1 (T v2 l2 r2) r1
      using ‹v2 ≤ v1›
      by auto
    thus ?thesis
      using 3 ‹v1 ≥ val r1›
      using ‹v2 ≤ v1›
      by (metis False Tree.exhaust Tree.inject Tree.simps(3)
          True is-heap.simps(3) is-heap.simps(6) merge.simps(2)
```

```
        merge.simps(3) order-eq-iff val.simps)
    next
      case False
      hence v1 ≥ val l1
        using 3(3)
        by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
      have merge l1 (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 ll1 lr1 lv1 where r1 = T lv1 ll1 lr1
        using ‹r1 ≠ E›
        by (metis Tree.exhaust)
      obtain rl1 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 ‹val (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 l1 r1))
    using False ‹is-heap (T v1 l1 r1)› 3
    by auto
  have val (merge l2 (T v1 l1 r1)) = val l2 ∨
      val(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 l2 = E)
      case True
      hence merge (T v1 l1 r1) (T v2 l2 r2) = T v2 (T v1 l1 r1) E
        using ‹r2 = E› ‹¬ v2 ≤ v1›
        by auto
      thus ?thesis
        using 3
        using ‹¬ v2 ≤ v1›
        by auto
    next
      case False
      hence v2 ≥ val l2
        using 3(4)
        by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
      thus ?thesis
        using ‹r2 = E› ‹¬ v1 ≥ v2›
        using ‹is-heap (merge l2 (T v1 l1 r1))›
        using ‹val (merge l2 (T v1 l1 r1)) = val l2 ∨
               val(merge l2 (T v1 l1 r1)) = v1›
        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 v2 ≥ val r2
      using 3(4)
      by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
    show ?thesis
    proof(cases l2 = E)
      case True
      hence merge (T v1 l1 r1) (T v2 l2 r2) = T v2 (T v1 l1 r1) r2
        using ‹¬ v2 ≤ v1›
        by auto
      thus ?thesis
        using 3 ‹v2 ≥ val r2›
        using ‹¬ v2 ≤ v1›
        by (metis False Tree.exhaust Tree.simps(3) is-heap.simps(3)
            is-heap.simps(5) linorder-linear val.simps)
    next
      case False
      hence v2 ≥ val l2
        using 3(4)
        by (metis Tree.exhaust in-tree.simps(2) is-heap-max val.simps)
      have merge l2 (T v1 l1 r1) ≠ E
        using False
        by (metis Tree.exhaust Tree.simps(2) merge.simps(3))
      have is-heap r2
        using 3(4)
```

```
        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›
        using ‹¬ v1 ≥ v2› ‹v2 ≥ val l2›
        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 ‹¬ v2 ≤ v1›
        by auto
    qed
  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 ⟷  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 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-assoc)
    hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
          {#v1#} + multiset l1 + multiset r1 + multiset (T v2 l2 r2)
      by (metis union-commute union-lcomm)
    thus ?thesis
      by auto (metis union-commute)
  next
    case False
    hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
          {#v2#} + multiset (merge l2 (T v1 l1 r1)) + multiset r2
      by auto (metis union-commute)
    hence multiset (merge (T v1 l1 r1) (T v2 l2 r2)) =
          {#v2#} + multiset l2 + multiset r2 + multiset (T v1 l1 r1)
      using 3 False
      by (metis union-commute union-lcomm)
    thus ?thesis
      by (metis multiset.simps(2) union-commute)
  qed
qed
```

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

```
interpretation HS: Collection E hs-is-empty hs-of-list multiset
proof
   fix t
  assume hs-is-empty t
  thus t = E
    by auto
next
  show hs-is-empty E
    by auto
next
  show multiset E = {#}
    by auto
next
  fix l
  show multiset (hs-of-list l) = mset l
  proof(induct l)
```

```
    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'
  assume ¬ hs-is-empty t (m, t') = 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 multiset t' + {#m#} = multiset t
    using merge-multiset[of l r]
    using ‹(m, t') = hs-remove-max t›
    by (metis prod.inject multiset.simps(2) hs-remove-max.simps
        union-assoc union-commute)
next
  fix t m t'
  assume ¬ hs-is-empty t is-heap (id t) (m, t') = 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' = merge l r
    using ‹(m, t') = hs-remove-max t›
    by auto
  have is-heap l ∧ is-heap r
    using ‹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 is-heap (id t')
    using ‹t' = merge l r›
    using merge-heap-is-heap
    by auto
next
  fix t m t'
  assume ¬ hs-is-empty t (m, t') = hs-remove-max t
  thus m = 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 ≡ val (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 in-tree v t
  thus in-tree v (siftDown t)
    apply (induct t rule:siftDown.induct)
    by auto
next
  assume  in-tree v (siftDown t)
  thus in-tree v t
  proof (induct t rule:siftDown.induct)
    case 1
    thus ?case
      by auto
  next
    case (2 v1)
    thus ?case
      by auto
  next
    case (3 v2 v1 l1 r1)
    show ?case
    proof(cases v2 ≥ v1)
      case True
      thus ?thesis
        using 3
        by auto
    next
      case False
      show ?thesis
      proof(cases v1 = v)
        case True
        thus ?thesis
          using 3 False
          by auto
      next
        case False
        hence in-tree v (siftDown (set-val (T v1 l1 r1) v2))
          using ‹¬ v2 ≥ v1› 3(2)
          by auto
        hence in-tree v (T v2 l1 r1)
          using 3(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
      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
      qed
    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
      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) ‹¬ 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 v2 l2 r2) v′))
            using 5-1(3) ‹¬ in-tree v (T v1 l1 r1)› ‹¬ v1 ≥ v2› ‹¬ v′ ≥ v2›
            using ‹ ¬ (v = v′ ∨ v = v1 ∨ 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) v′))
            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) ‹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› 5-2
        by auto
    next
      case False
      thus ?thesis
      proof(cases in-tree v (T v1 l1 r1))
        case True
        thus ?thesis
          by auto
      next
        case False
        hence in-tree v (siftDown (set-val (T v2 l2 r2) v'))
          using 5-2(3) ‹¬ in-tree v (T v1 l1 r1)› ‹¬ v1 ≥ v2› ‹¬ v' ≥ v2›
          using ‹ ¬ (v = v' ∨ v = v1 ∨ v = v2)›
          by auto
        hence in-tree v (T v' l2 r2)
          using 5-2(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
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 ≥ v'
          using 3
          using is-heap-max[of v' T v1 l1 r1]
          by auto
        thus ?thesis
          using ‹is-heap ?t› ∗ ‹¬ v2 ≥ v1›
          by auto
      qed
    qed
  qed
next
  case (4 v2 v1 l1 r1)
  show ?case
  proof(cases v2 ≥ 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' ≥ v2
      using siftDown-Node[of T v2 l1 r1 v2 l1 r1]
      by auto
    have r = T v1 l1 r1
      using 4(4)
      by auto
    hence is-heap l1 is-heap r1
      using 4(3)
      apply (induct r rule:is-heap.induct)
      by auto
    hence is-heap ?t
      using False 4(1)[of l1 r1 v2]
      by auto
    show ?thesis
    proof(cases v' = v2)
      case True
      thus ?thesis
        using ∗ ‹is-heap ?t› False
        by auto
    next
      case False
      have in-tree v' ?t
```

```
        using ∗
        using siftDown-in-tree[of ?t]
        by auto
      hence in-tree v′ (T v2 l1 r1)
        using ∗ siftDown-in-tree-set[of v′ T v2 l1 r1]
        by auto
      hence in-tree v′ (T v1 l1 r1)
        using False
        by auto
      hence v1 ≥ v′
        using is-heap-max[of v′ T v1 l1 r1] 4
        by auto
      thus ?thesis
        using ‹is-heap ?t› False ∗
        by auto
    qed
  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 in-tree v′ ?t
        using ∗ siftDown-in-tree
        by auto
      hence in-tree v′ (T v1 l2 r2)
        using siftDown-in-tree-set[of v′ T v1 l2 r2]
        by auto
      hence in-tree v′ (T v2 l2 r2)
        using False
        by auto
      thus ?thesis
        using is-heap-max[of v′ T v2 l2 r2] 5-1
        by auto
    qed
    thus ?thesis
      using ‹is-heap ?t› ‹v2 ≥ v3› ∗ False 5-1
      by auto
  qed
next
  case False
  show ?thesis
  proof(cases v1 ≥ v3)
    case True
    thus ?thesis
      using ‹¬ v2 ≥ v3› 5-1
      by auto
  next
    case False
    let ?t = siftDown (T v1 l3 r3)
    obtain l′ v′ r′ where ∗: ?t = T v′ l′ r′ v′ ≥ v1
      using siftDown-Node
      by blast
    have is-heap l3 is-heap r3
      using 5-1(4, 5)
      apply(induct r rule:is-heap.induct)
      by auto
    hence is-heap ?t
      using 5-1(2)[of l3 r3 v1] ‹¬ 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*
by *auto*
thus *?thesis*
using *is-heap-max*[*of* v' *T v3 l3 r3*] *5-1*
by *auto*
qed
thus *?thesis*
using ‹*is-heap ?t*› ‹¬ *v2* ≥ *v3*› ∗ *False 5-1*
by *auto*
qed
qed
next
case (*5-2 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*› *5-2*
by *auto*
next
case *False*
let *?t* = *siftDown* (*T v1 l2 r2*)
obtain l' v' r' **where** ∗: *?t* = *T* v' l' r' *v1* ≤ 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*] ‹*v2* ≥ *v3*› *False*
by *auto*
have *v2* ≥ 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 l2 r2)
        using siftDown-in-tree-set[of v′ T v1 l2 r2]
        by auto
      hence in-tree v′ (T v2 l2 r2)
        using False
        by auto
      thus ?thesis
        using is-heap-max[of v′ T v2 l2 r2] 5-2
        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
        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
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 romoved leaf element. The second returned value is tree without that leaf.

```
fun removeLeaf:: 'a::linorder Tree ⇒ 'a × '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 ⇒ '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
    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-lcomm)
```

```
    ultimately
    show ?thesis
      by auto
  qed
next
  case (4 v1 v l r)
  thus ?case
  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)) =
         {#v1#} + 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 v v1 l1 r1 v2 l2 r2)
  thus ?case
  proof(cases v1 ≥ v2)
    case True
    thus ?thesis
    proof(cases v ≥ v1)
      case True
      thus ?thesis
        using ‹v1 ≥ v2›
        by auto
```

```
  next
    case False
    hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
          multiset l1 + {#v#} + multiset r1 + {#v1#} +
          multiset (T v2 l2 r2)
      using ‹v1 ≥ v2› 5-1(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 ≥ v2)
    case True
    thus ?thesis
      using False
      by auto
  next
    case False
    hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
          multiset (T v1 l1 r1) + {#v2#} +
          multiset l2 + {#v#} + multiset r2
      using ‹¬ v1 ≥ v2› 5-1(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#} + multiset r2
        by (metis union-commute union-lcomm)
      ultimately
      show ?thesis
        by auto
    qed
  qed
next
  case (5-2 v v1 l1 r1 v2 l2 r2)
  thus ?case
  proof(cases v1 ≥ v2)
    case True
    thus ?thesis
    proof(cases v ≥ v1)
      case True
      thus ?thesis
        using ‹v1 ≥ v2›
        by auto
    next
      case False
      hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
               multiset l1 + {#v#} + multiset r1 + {#v1#} +
               multiset (T v2 l2 r2)
        using ‹v1 ≥ v2› 5-2(1)
        by auto
      moreover
      have multiset (T v (T v1 l1 r1) (T v2 l2 r2)) =
              multiset l1 + {#v1#} + multiset r1 +
              {#v#} + multiset(T v2 l2 r2)
        by auto
      moreover
      have multiset l1 + {#v1#} + multiset r1 + {#v#} +
            multiset(T v2 l2 r2) =
              multiset l1 + {#v#} + multiset r1 + {#v1#} +
              multiset (T v2 l2 r2)
        by (metis union-commute union-lcomm)
      ultimately
      show ?thesis
        by auto
    qed
  next
    case False
    show ?thesis
    proof(cases v ≥ v2)
      case True
      thus ?thesis
        using False
        by auto
    next
```

```
      case False
      hence multiset (siftDown (T v (T v1 l1 r1) (T v2 l2 r2))) =
              multiset (T v1 l1 r1) + {#v2#} + multiset l2 + {#v#} +
              multiset r2
        using ‹¬ v1 ≥ v2› 5-2(2)
        by (simp add: ac-simps)
      moreover
      have multiset (T 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
qed

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


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

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

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

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

lemma *removeLeaf-heap-is-heap*:
assumes *is-heap t t* ≠ *E*
shows *is-heap* (*snd* (*removeLeaf t*))
using *assms*
proof(*induct t rule*:*removeLeaf.induct*)
case (*1 v*)
thus *?case*
by *auto*
next
case (*2 v v1 l1 r1*)
have *is-heap* (*T v1 l1 r1*)
using *2*(*3*)
by *auto*
hence *is-heap* (*snd* (*removeLeaf* (*T v1 l1 r1*)))
using *2*(*1*)
by *auto*
let *?t* = (*snd* (*removeLeaf* (*T v1 l1 r1*)))
show *?case*
proof(*cases ?t* = *E*)
case *True*
thus *?thesis*
by *auto*
next
case *False*
have *v* ≥ *v1*
using *2*(*3*)
by *auto*

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

```
      using ‹is-heap (snd (removeLeaf (T v1 l1 r1)))›
      by auto
    have v ≥ v1
      using 4-2(3)
      by auto
    hence v ≥ val ?t
      using False removeLeaf-val-val[of T v1 l1 r1]
      by auto
    hence v ≥ v1'
      using ‹?t = T v1' l1' r1'›
      by auto
    hence is-heap (T v (T v1' l1' r1') (T v2 l2 r2))
      using ‹is-heap (T v1' l1' r1')›
      using ‹is-heap (T v2 l2 r2)› ‹v ≥ v2›
      by (simp add: is-heap.simps(5))
    thus ?thesis
      using 4-2 ‹?t = T v1' l1' r1'›
      by auto
  qed
next
  case 5
  thus ?case
    by auto
qed
```

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

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

lemma removeLeaf-multiset:
  assumes (v', t') = removeLeaf t t ≠ E
```

```
  shows {#v'#} + multiset t' = multiset t
using assms
proof(induct t arbitrary: v' t' rule:removeLeaf.induct)
  case 1
  thus ?case
    by auto
next
  case (2 v v1 l1 r1)
  have t' = T v (snd (removeLeaf (T v1 l1 r1))) E
    using 2(3)
    by auto
  have v' = fst (removeLeaf (T v1 l1 r1))
    using 2(3)
    by auto
  hence {#v'#} + multiset t' =
          {#fst (removeLeaf (T v1 l1 r1))#} +
          multiset (snd (removeLeaf (T v1 l1 r1))) +
          {#v#}
    using ‹t' = T v (snd (removeLeaf (T v1 l1 r1))) E›
    by (simp add: ac-simps)
  have {#fst (removeLeaf (T v1 l1 r1))#} +
        multiset (snd (removeLeaf (T v1 l1 r1))) =
          multiset (T v1 l1 r1)
    using 2(1)
    by auto
  hence {#v'#} + multiset t' = multiset (T v1 l1 r1) + {#v#}
    using ‹{#v'#} + multiset t' =
          {#fst (removeLeaf (T v1 l1 r1))#} +
          multiset (snd (removeLeaf (T v1 l1 r1))) + {#v#}›
    by auto
  thus ?case
    by auto
next
  case (3 v v1 l1 r1)
  have t' = T v E (snd (removeLeaf (T v1 l1 r1)))
    using 3(3)
    by auto
  have v' = fst (removeLeaf (T v1 l1 r1))
    using 3(3)
    by auto
  hence {#v'#} + multiset t' =
          {#fst (removeLeaf (T v1 l1 r1))#} +
          multiset (snd (removeLeaf (T v1 l1 r1))) +
          {#v#}
    using ‹t' = T v E (snd (removeLeaf (T v1 l1 r1)))›
    by (simp add: ac-simps)
  have {#fst (removeLeaf (T v1 l1 r1))#} +
        multiset (snd (removeLeaf (T v1 l1 r1))) =
          multiset (T v1 l1 r1)
```

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

```
        {#v#} + multiset (T v2 l2 r2)
    using ‹t' = T v (snd (removeLeaf (T v1 l1 r1))) (T v2 l2 r2)›
    by (metis multiset.simps(2) union-assoc)
  have {#fst (removeLeaf (T v1 l1 r1))#} +
       multiset (snd (removeLeaf (T v1 l1 r1))) =
         multiset (T v1 l1 r1)
    using 4-2(1)
    by auto
  hence {#v'#} + multiset t' =
        multiset (T v1 l1 r1) + {#v#} + multiset (T v2 l2 r2)
    using ‹{#v'#} + multiset t' =
          {#fst (removeLeaf (T v1 l1 r1))#} +
          multiset (snd (removeLeaf (T v1 l1 r1))) +
          {#v#} + 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 ≠ E
  shows multiset (set-val t v') +  {#val t#} = {#v'#} + multiset t
proof−
  obtain v l r where t = T v l r
    using assms
    by (metis Tree.exhaust)
  hence multiset (set-val t v') + {#val t#} =
        multiset l + {#v'#} + multiset r + {#v#}
    by auto
  have {#v'#} + multiset t =
       {#v'#} + multiset l + {#v#} + multiset r
    using ‹t = T v l r›
    by (metis multiset.simps(2) union-assoc)
  have {#v'#} + multiset l + {#v#} + multiset r =
       multiset l + {#v'#} + multiset r + {#v#}
    by (metis union-commute union-lcomm)
  thus ?thesis
    using ‹multiset (set-val t v') + {#val t#} =
          multiset l + {#v'#} + multiset r + {#v#}›
    using ‹{#v'#} + multiset t =
          {#v'#} + multiset l + {#v#} + multiset r›
    by auto
qed

lemma hs-remove-max-multiset:
```

```
  assumes (m, t′) = hs-remove-max t t ≠ E
  shows {#m#} + multiset t′ = multiset t
proof−
  let ?v1 = fst (removeLeaf t)
  let ?t1 = snd (removeLeaf t)
  show ?thesis
  proof(cases ?t1 = E)
    case True
    hence {#m#} + multiset t′ = {#m#}
      using assms
      unfolding hs-remove-max-def
      by auto
    have ?v1 = val t
      using True assms(2)
      apply (induct t rule:removeLeaf.induct)
      by auto
    hence ?v1 = m
      using assms(1) True
      unfolding hs-remove-max-def
      by auto
    hence multiset t = {#m#}
      using removeLeaf-multiset[of ?v1 ?t1 t] True assms(2)
      by (metis empty-neutral(2) multiset.simps(1) prod.collapse)
    thus ?thesis
      using ‹{#m#} + multiset t′ = {#m#}›
      by auto
  next
    case False
    hence t′ = siftDown (set-val ?t1 ?v1)
      using assms(1)
      by (auto simp add: hs-remove-max-def) (metis prod.inject)
    hence multiset t′ + {#val ?t1#} = multiset t
      using siftDown-multiset[of set-val ?t1 ?v1]
      using set-val-multiset[of ?t1 ?v1] False
      using removeLeaf-multiset[of ?v1 ?t1 t] assms(2)
      by auto
    have val ?t1 = val t
      using False assms(2)
      apply (induct t rule:removeLeaf.induct)
      by auto
    have val t = m
      using assms(1) False
      using ‹t′ = siftDown (set-val ?t1 ?v1)›
      unfolding hs-remove-max-def
      by (metis (full-types) fst-conv removeLeaf.simps(1))
    hence val ?t1 = m
      using ‹val ?t1 = val t›
      by auto
    hence multiset t′ + {#m#} = multiset t
```

```
      using ‹multiset t' + {#val ?t1#} = multiset t›
      by metis
    thus ?thesis
      by (metis union-commute)
  qed
qed
```

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

```
interpretation Heap E hs-is-empty hs-of-list multiset id hs-remove-max
proof
  fix t
  show multiset t = multiset (id t)
    by auto
next
  fix t
  show  is-heap (id (hs-of-list t))
    unfolding hs-of-list-def
    using heapify-heap-is-heap[of of-list-tree t]
    by auto
next
  fix t
  show (id t = E) = hs-is-empty t
    by auto
next
  fix t m t'
  assume ¬ hs-is-empty t (m, t') = hs-remove-max t
  thus multiset t' + {#m#} = multiset t
    using hs-remove-max-multiset[of m t' t]
    by (auto, metis union-commute)
next
  fix t v' t'
  assume ¬ hs-is-empty t is-heap (id t) (v', t') = hs-remove-max t
  let ?v1 = fst (removeLeaf t)
  let ?t1 = snd (removeLeaf t)
  have is-heap ?t1
    using ‹¬ hs-is-empty t› ‹is-heap (id t)›
    using removeLeaf-heap-is-heap[of 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
      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, 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 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.