Verification of Selection and Heap Sort Using Locales

Danijela Petrović

February 23, 2021

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*
  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* = {#}
⟨*proof*⟩

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*:
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow m = \textit{Max}\ (\textit{set}\ l)$
— — First parameter of the return value of the function *remove_max* is the maximum element
assumes *remove-max-multiset*:
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow$
$\textit{add-mset}\ m\ (\textit{multiset}\ l') = \textit{multiset}\ l$
— — Condition for multiset, ensures that nothing new is added or nothing is lost after applying *remove_max* function.
assumes *remove-max-inv*:
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow \textit{inv}\ l'$
— — 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*:
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow$
$\textit{size}\ (\textit{multiset}\ l) > \textit{size}\ (\textit{multiset}\ l')$
$\langle \textit{proof} \rangle$

lemma *remove-max-set*:
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow$
$\textit{set}\ l' \cup \{m\} = \textit{set}\ l$
$\langle \textit{proof} \rangle$

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
$(m,\ l') = \textit{remove-max}\ l$
in
$\textit{ssort}'\ l'\ (m\ \#\ \textit{sl}))$
declare *ssort'.simps*[*code*]
definition $\textit{ssort} :: {}'a\ \textit{list} \Rightarrow {}'a\ \textit{list}$ **where**
$\textit{ssort}\ l = \textit{ssort}'\ (\textit{of-list}\ l)\ []$

inductive *ssort'-dom* **where**

step: $[\![\bigwedge m\ l'.\ [\![\neg\ \textit{is-empty}\ l;\ (m,\ l') = \textit{remove-max}\ l]\!] \Longrightarrow$
$\textit{ssort}'\textit{-dom}\ (l',\ m\ \#\ sl)]\!] \Longrightarrow \textit{ssort}'\textit{-dom}\ (l,\ sl)$

lemma *ssort′-termination*:
assumes *inv* (*fst p*)
shows *ssort′-dom p*
⟨*proof*⟩

lemma *ssort′Induct*:
assumes *inv l P l sl*
$\bigwedge\ l\ sl\ m\ l'.$
$[\![\neg\ \textit{is-empty}\ l;\ \textit{inv}\ l;\ (m,\ l') = \textit{remove-max}\ l;\ P\ l\ sl]\!] \Longrightarrow P\ l'\ (m\ \#\ sl)$
shows *P empty* (*ssort′ l sl*)
⟨*proof*⟩

lemma *mset-ssort′*:
assumes *inv l*
shows *mset* (*ssort′ l sl*) = *multiset l* + *mset sl*
⟨*proof*⟩

lemma *sorted-ssort′*:
assumes $\textit{inv}\ l\ \textit{sorted}\ sl \wedge (\forall\ x \in \textit{set}\ l.\ (\forall\ y \in \textit{List.set}\ sl.\ x \leq y))$
shows *sorted* (*ssort′ l sl*)
⟨*proof*⟩

lemma *sorted-ssort*: *sorted* (*ssort i*)
⟨*proof*⟩

lemma *permutation-ssort*: *ssort l* <~~> *l*
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ≥ *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 ⟹ *max* (*Max* (*insert m A*)) *x* = *max m* (*Max* (*insert x A*))
  ⟨*proof*⟩

The function really returned the maximum value.

lemma *remove-max-max-lemma*:
  shows *fst* (*foldl f* (*m*, *t*) *l*) = *Max* (*set* (*m* # *l*))
⟨*proof*⟩

lemma *remove-max-max*:
  assumes *l* ≠ [] (*m*, *l*′) = *remove-max l*
  shows *m* = *Max* (*set l*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *remove-max-mset*:
  assumes *l* ≠ [] (*m*, *l*′) = *remove-max l*
  shows *add-mset m* (*mset l*′) = *mset l*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) = ($v \geq$ *val r* $\wedge$ *is-heap r*)
| *is-heap* (*T v l E*) = ($v \geq$ *val l* $\wedge$ *is-heap l*)
| *is-heap* (*T v l r*) = ($v \geq$ *val r* $\wedge$ *is-heap r* $\wedge$ $v \geq$ *val l* $\wedge$ *is-heap l*)

lemma *heap-top-geq*:
  assumes $a \in\# \ multiset\ t\ is\text{-}heap\ t$
  shows $val\ t \geq a$
$\langle proof \rangle$

lemma *heap-top-max*:
  assumes $t \neq E\ is\text{-}heap\ t$
  shows $val\ t = Max\text{-}mset\ (multiset\ t)$
$\langle proof \rangle$

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**
  $is\text{-}empty :: {'b} \Rightarrow bool$ **and**
  $of\text{-}list :: {'a}::linorder\ list \Rightarrow {'b}$ **and**
  $multiset :: {'b} \Rightarrow {'a}::linorder\ multiset$ +
  fixes $as\text{-}tree :: {'b} \Rightarrow {'a}::linorder\ Tree$
— 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 $remove\text{-}max :: {'b} \Rightarrow {'a} \times {'b}$
  assumes *multiset*: $multiset\ l = Heap.multiset\ (as\text{-}tree\ l)$
  assumes *is-heap-of-list*: $is\text{-}heap\ (as\text{-}tree\ (of\text{-}list\ i))$
  assumes *as-tree-empty*: $as\text{-}tree\ t = E \longleftrightarrow is\text{-}empty\ t$
  assumes *remove-max-multiset'*:
  $[\![\neg\ is\text{-}empty\ l;\ (m,\ l') = remove\text{-}max\ l]\!] \Longrightarrow add\text{-}mset\ m\ (multiset\ l') = multiset\ l$
  assumes *remove-max-is-heap*:
  $[\![\neg\ is\text{-}empty\ l;\ is\text{-}heap\ (as\text{-}tree\ l);\ (m,\ l') = remove\text{-}max\ l]\!] \Longrightarrow$
  $is\text{-}heap\ (as\text{-}tree\ l')$
  assumes *remove-max-val*:
  $[\![\ \neg\ is\text{-}empty\ t;\ (m,\ t') = remove\text{-}max\ t]\!] \Longrightarrow m = val\ (as\text{-}tree\ t)$

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

sublocale *Heap* <
  $RemoveMax\ empty\ is\text{-}empty\ of\text{-}list\ multiset\ remove\text{-}max\ \lambda\ t.\ is\text{-}heap\ (as\text{-}tree\ t)$
$\langle proof \rangle$

primrec *in-tree* **where**
  $in\text{-}tree\ v\ E = False$
| $in\text{-}tree\ v\ (T\ v'\ l\ r) \longleftrightarrow v = v' \lor in\text{-}tree\ v\ l \lor in\text{-}tree\ v\ r$

lemma *is-heap-max*:

assumes *in-tree v t is-heap t*
shows *val t* $\geq$ *v*
$\langle$*proof*$\rangle$

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* $\Rightarrow$ $'a$ *Tree* $\Rightarrow$ $'a$ *Tree* **where**
merge t1 E = *t1*
| *merge E t2* = *t2*
| *merge* (*T v1 l1 r1*) (*T v2 l2 r2*) =
(*if v1* $\geq$ *v2 then T v1* (*merge l1* (*T v2 l2 r2*)) *r1*
else T v2 (*merge l2* (*T v1 l1 r1*)) *r2*)
$\langle$*proof*$\rangle$
termination
$\langle$*proof*$\rangle$

lemma *merge-val*:
val(*merge l r*) = *val l* $\vee$ *val*(*merge l r*) = *val r*
$\langle$*proof*$\rangle$

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*)
$\langle$*proof*$\rangle$

definition *insert* :: $'a$::*linorder* $\Rightarrow$ $'a$ *Tree* $\Rightarrow$ $'a$ *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* $\longleftrightarrow$ *t* = *E*

Definition of function *remove_max*:

fun *hs-remove-max*:: $'a$::*linorder Tree* $\Rightarrow$ $'a \times 'a$ *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*⟩

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

interpretation *HS*: *Collection E hs-is-empty hs-of-list multiset*
⟨*proof*⟩

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*⟩

end

7 Verification of Imperative Heap Sort

theory *HeapImperative*
imports *Heap*
begin

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

abbreviation *left-val* :: $'a$ *Tree* $\Rightarrow$ $'a$ **where**
left-val $t \equiv$ *val* (*left* t)

primrec *right* :: $'a$ *Tree* $\Rightarrow$ $'a$ *Tree* **where**
right $(T\ v\ l\ r) = r$

abbreviation *right-val* :: $'a$ *Tree* $\Rightarrow$ $'a$ **where**
right-val $t \equiv$ *val* (*right* t)

abbreviation *set-val* :: $'a$ *Tree* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *Tree* **where**
set-val $t\ x \equiv T\ x$ (*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* $\Rightarrow$ $'a$ *Tree* **where**
$\quad$ *siftDown* $E = E$
$|$ *siftDown* $(T\ v\ E\ E) = T\ v\ E\ E$
$|$ *siftDown* $(T\ v\ l\ E) =$
$\qquad$ (*if* $v \geq$ *val* l *then* $T\ v\ l\ E$ *else* T (*val* l) (*siftDown* (*set-val* $l\ v$)) E)
$|$ *siftDown* $(T\ v\ E\ r) =$
$\qquad$ (*if* $v \geq$ *val* r *then* $T\ v\ E\ r$ *else* T (*val* r) E (*siftDown* (*set-val* $r\ v$)))
$|$ *siftDown* $(T\ v\ l\ r) =$
$\qquad$ (*if* *val* $l \geq$ *val* r *then*
$\qquad\quad$ *if* $v \geq$ *val* l *then* $T\ v\ l\ r$ *else* T (*val* l) (*siftDown* (*set-val* $l\ v$)) r
$\qquad$ *else*
$\qquad\quad$ *if* $v \geq$ *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 $\exists\ l'\ v'\ r'.$ *siftDown* $t = T\ v'\ l'\ r' \wedge v' \geq v$
$\langle$*proof*$\rangle$

lemma *siftDown-in-tree*:
assumes $t \neq E$
shows *in-tree* (*val* (*siftDown* t)) t
$\langle$*proof*$\rangle$

lemma *siftDown-in-tree-set*:
shows *in-tree* $v\ t \longleftrightarrow$ *in-tree* v (*siftDown* t)
$\langle$*proof*$\rangle$

lemma *siftDown-heap-is-heap*:
assumes *is-heap* l *is-heap* r $t = T\ v\ l\ r$
shows *is-heap* (*siftDown* t)
$\langle$*proof*$\rangle$

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

primrec *heapify* **where**
$\quad$ *heapify* $E = E$
$|$ *heapify* $(T\ v\ l\ r) =$ *siftDown* $(T\ v$ (*heapify* l) (*heapify* r))

lemma *heapify-heap-is-heap*:
is-heap (*heapify* t)
$\langle$*proof*$\rangle$

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* $\Rightarrow$ $'a \times 'a$ *Tree* **where**
removeLeaf $(T\ v\ E\ E) = (v,\ E)$
| *removeLeaf* $(T\ v\ l\ E) = (fst\ (removeLeaf\ l),\ T\ v\ (snd\ (removeLeaf\ l))\ E)$
| *removeLeaf* $(T\ v\ E\ r) = (fst\ (removeLeaf\ r),\ T\ v\ E\ (snd\ (removeLeaf\ r)))$
| *removeLeaf* $(T\ v\ l\ r) = (fst\ (removeLeaf\ l),\ T\ v\ (snd\ (removeLeaf\ l))\ r)$

Function *of_list_tree* makes a binary tree from any given list.

primrec *of-list-tree*:: $'a$::*linorder list* $\Rightarrow$ $'a$ *Tree* **where**
of-list-tree [] = E
| *of-list-tree* $(v\ \#\ tail) = T\ v\ (of\text{-}list\text{-}tree\ tail)\ E$

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* $\Rightarrow$ $'a \times 'a$ *Tree* **where**
hs-remove-max $t \equiv$
  (*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* $t \longleftrightarrow t = E$

lemma *siftDown-multiset*:
multiset (*siftDown* t) = *multiset* t
⟨*proof*⟩

lemma *mset-list-tree*:
multiset (*of-list-tree* l) = *mset* l
⟨*proof*⟩

lemma *multiset-heapify*:
multiset (*heapify* t) = *multiset* t
⟨*proof*⟩

lemma *multiset-heapify-of-list-tree*:
multiset (*heapify* (*of-list-tree* l)) = *mset* l
⟨*proof*⟩

lemma *removeLeaf-val-val*:
  assumes $snd\ (removeLeaf\ t) \neq E\ t \neq E$
  shows $val\ t = val\ (snd\ (removeLeaf\ t))$
⟨*proof*⟩

lemma *removeLeaf-heap-is-heap*:
  assumes $is\text{-}heap\ t\ t \neq E$
  shows $is\text{-}heap\ (snd\ (removeLeaf\ t))$
⟨*proof*⟩

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*⟩

lemma *removeLeaf-multiset*:
  assumes $(v', t') = removeLeaf\ t\ t \neq E$
  shows $\{\#v'\#\} + multiset\ t' = multiset\ t$
⟨*proof*⟩

lemma *set-val-multiset*:
  assumes $t \neq E$
  shows $multiset\ (set\text{-}val\ t\ v') + \{\#val\ t\#\} = \{\#v'\#\} + multiset\ t$
⟨*proof*⟩

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

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*⟩

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.