JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Lucas Payr

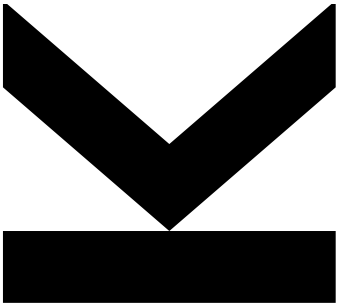
Submitted at
RISC
Research Institute for Symbolic Computation

Supervisor
Assoc.Univ.-Prof. DI Dr. Wolfgang Windsteiger

Co-Supervisor
Assoc.Univ.-Prof. DI Dr. Wolfgang Schreiner

December 21, 2018

Formalization and Validation of Fundamental Sequence Algorithms by Computer-assisted Checking of Finite Models



Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Technische Mathematik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

ABSTRACT

While common textbooks go into great details when it comes to the analysis of sequence algorithms, they lack in proper proofs and moreover formal specifications. The most essential part, the loop invariant is either described very vaguely or is completely missing. This thesis gives those missing specifications as well as so called verification conditions upon which one can fully prove an algorithm. Normally such a process is very difficult and it is easy to wrongly specify an algorithm. With the help of the RISC Algorithm Language (RISCAL), developed at the Research Institute for Symbolic Computation (RISC), this process is simpler as this system provides additional checks that help noticing early if a specification is wrong. It uses finite model checking, which makes it possible to check the adequacy of specifications even if they include general quantifiers, which would be difficult infinite models.

The result of the thesis is a collection of specifications for various searching and sorting algorithms. The algorithms are implemented for different data types (array, recursive lists, pointer-linked lists) to serve as an addition to common textbooks.

CONTENTS

1 INTRODUCTION

1.1 INTRODUCTION AND BACKGROUND

As more and more parts in our society get automated, we rely on sophisticated and complex algorithms that need to work perfectly. Even a small bug could have major consequences, as we start introducing more autonomous machines that should act on our behalves, such as self driving cars, personal assistance's and smart homes. Currently we still rely on humans to find bugs. But in future we might have programming languages and tools that can prove correctness of a piece of code. Generally one might use the Hoare Calculus [Hoa69] for proving correctness, as it brings a calculus for algorithm correctness that is based on predicate logic. But using it for a manual proof of a bigger algorithm is quite a tedious task. Therefore, motivated by the theory of automated verification of algorithms as well as automated proving, RISC (Research Institute for Symbolic Computation) is currently working in this field and has developed a few notable tools:

- *Theorema*: Theorema is a Mathematica-based framework for computer assisted theorem proving and theory exploration [Buc+16].
- *RISC ProgramExplorer*: The RISC ProgramExplorer is an integrated environment supporting formal reasoning about computer programs and computing systems using a variant of Hoare Calculus. It uses the RISC ProofNavigator as an interactive proving assistant [Sch09].
- *RISC Algorithmic Language*: RISC Algorithmic Language, RISCAL for short, is a system (still in development) for formally specifying algorithms and validating their correctness by checking finite instances [Sch17; SBF18; Sch18].

RISCAL is primarily for educational use, as it gives students a tool to validate their algorithms with respect to formal specifications. The system combines a mathematical modeling language with an algorithmic descriptive language. Model instances are finite and their properties decidable; the system implements a corresponding model checker [Cla+18]. This checker iterates over all inputs of a finite domain and validates the algorithm for each input individually. However, for the validation over an infinite domain we need so called loop invariants [FMV14] to derive logic verification conditions [Cor+09] whose validity implies the correctness of the algorithm. But not every loop invariant is sharp enough to yield valid verification

conditions [DF08]. RISCAL can also support the process of finding suitable invariants not only by checking the invariants during the execution of the algorithm but also by checking the verification conditions over a finite domain; subsequently we may use a proof assistant such as the RISC ProgramNavigator to verify the conditions also over infinite domains.

1.2 GOALS OF THE THESIS

In this thesis we will investigate the formal specification and validation of various algorithms operating over the data type "finite sequence" in different concrete representations. For this we will use the already mentioned RISCAL system.

In more detail we will look at the following three concrete implementations of finite sequences:

- **Array**: a finite sequence where one can access any element by an index in constant time.
- **Recursive List**: a recursively defined sequence, which is either empty or a pair of an element and a remainder list. Such lists are naturally processed by recursive algorithms.
- **Pointer-Linked List**: a sequence represented by a numerical address which may be either a special constant (indicating the empty list) or the index of a memory cell holding an element and another pointer pointing towards the remainder of the list. This representation also allows low-level algorithms that update a list in place.

The implementations of different algorithms always depend on the fundamental data structure used. During the thesis we will look at two problems of finite sequences:

- **Searching:** the basic idea is to iterate over all elements until one finds the given key. This can be accomplished in linear time, therefore it is known as linear search [SW16]. For a sorted list one can look at the middle-element, throw away one side, then recursively search in the other side. This is called a binary search [SW16].
- **Sorting:** as discussed above, an already sorted sequence is easier to search. That is why one might consider sorting a sequence once, to then be afterwards faster in searching.

  In the thesis we will look at one of the more basic sorting algorithms: InsertionSort. How ever there are also many other more advanced sorting algorithms [SW16]; for this thesis we will focus on QuickSort, MergeSort and HeapSort. All three have an underlying fundamental algorithm that is responsible for their efficiency:

  - "Partitioning" for QuickSort.
  - "Merging" for MergeSort.
  - "Heapifying" for HeapSort.

We will look at the linear search algorithm for all three data types. For the binary search, only an implementation using arrays makes sense. For all the sorting algorithms we will discuss the implementation for arrays and recursive linked lists. Additionally we will implement the QuickSort and therefore also the partitioning algorithm using pointer linked lists.

Another aspect for InsertionSort and MergeSort is to check whether the algorithm is stable, i.e. if the order of two elements with the same value gets changed during the sorting algorithm or not.

1.3 ACHIEVED RESULTS

In this thesis we not only have derived a collection of correct implementation but also all necessary pieces of information that are needed to not only use the algorithm but to let its correctness be validated by a computer. This information contains the following:

- **pre-conditions**: statements that must be valid before the algorithm is started (i.e. the requirements for executing the algorithm).
- **post-conditions**: statements that must be valid after the algorithm has terminated (i.e. the guarantees established by the algorithm).
- **invariants**: Statements that are valid before and after every loop iteration (i.e. they are valid before the first iteration as well as after the last one).
- **termination terms**: measurements for loops that assure their termination.
- **verification conditions**: conditions that are derived from the invariants such that their validity shows the correctness of the algorithm (i.e. requirements for a general proof).

First we have found suitable pre- and post-conditions as well as termination terms for all loops. Then to prove that an invariant is suitable, we have first checked it over a finite domain to see if it is not too strong. Then we have derived the following verification conditions:

- the invariant holds before the loop.
- the invariant holds after every iteration.
- after termination the invariant implies the postcondition.

For the last chapter of this thesis we have derived the verification conditions automatically. The actual proving over infinite domains was not done in this thesis. This could be done by feeding all the pieces of information listed above into a suitable automated prover or interactive proving assistant.

This thesis is organized as follows: In Chapter 2, we fist look at the state of the art and explain how to formally specify, formalize and validate algorithms. Then we give an overview of the tools that can be used for the task and finally we describe the RISCAL system in more detail. In Chapter 3, we take a look at searching algorithms and explain in great detail the differences between validating procedural and functional algorithms. In Chapter 4, we inspect the InsertionSort as well as the underlying theory for checking the stability of sorting algorithms. In the Chapters 5, 6, 7 we focus on QuickSort, MergeSort and HeapSort. We always implement the underlying subalgorithm first and then build the sorting algorithm upon it. Each subchapter is split by the datatype used, and the corresponding modified algorithm. The conclusion of the thesis is discussed in Chapter 8.

Work on this thesis was supported by the Johannes Kepler University, Linz, Institute of Technology(LIT) project LOGTECHEDU.

2 STATE OF THE ART

2.1 SEQUENCE ALGORITHMS

The algorithms introduced in this paper are known since the 1960s. Since then there have been numerous analyses [Knu98; SW16; Cor+09] and sketches of proofs [Cor+09] regarding their correctness.

As an example we investigate three collections of algorithms ("The Art of Computer Programming" by Donald E. Knuth [Knu98], "Introduction to Algorithms" by Thomas H. Cormen et al. [Cor+09] and "Algorithms" by Robert Sedgewick [SW16]).

Knuth uses records for his algorithms. The algorithms are first written descriptively and then implemented in an assembly language. He explains in details the validity of the presented algorithms and then analyzes the amount of computations needed. There are neither termination terms nor invariants in his book. This is not surprising, as at the time when it was written invariants where not of importance.

Cormen explores algorithms for arrays. He uses pseudo code to formalize his algorithms. He explains all loop invariant in a very detailed way but not formally and demonstrates their correctness using vivid arguments instead of a formal proof.

Sedgewick explains the algorithm using arrays but also takes modified algorithms for linked-lists into account. He analyzes the efficiency of every given algorithm but does not prove its correctness. That said he does provide the termination terms as well as the invariants but only as a side remark.

2.2 FORMAL SPECIFICATION, PROVING, MODEL CHECKING

The idea of using computers for proofs goes almost as far as the algorithms presented in this thesis themselves. 1969 Hoare and Floyed introduced Hoare Logic [Hoa69] with which one can formally annotate and prove algorithms. A program in Hoare logic can be described as a triple

$$\{P\}\ C\ \{Q\}$$

where P is the precondition, C is a command and Q is the post-condition. While P and Q are written as formulas in predicate logic, C is a command, i.e., a composition of assignments, while-loops, skips and if-else conditions.

For a valid Hoare-triple

$$\{P\}\ \mathit{while}\ B\ \mathtt{do}\ S\ \{\neg B \wedge P\}$$

we call P the loop invariant. To prove the validity of the triple, we need to show the validity of

$$\{P \wedge B\}\ S\ \{P\}.$$

Therefore it is of importance to know the loop invariants of an algorithm in order to prove it is correctness [Hat+12].

1981 Clarke, Emerson and Sifakis proposed a different method for the verification of programs: model checking [Cla+18]. The idea is that (the formal model of) all possible executions of a program are checked to satisfy specific properties such as safety (a generalization of partial correctness) or liveliness (a generalization of termination). However, this requires a model that can be efficiently checked in a finite number of steps, in particular a model with a finite input-domain respectively a model that allows only finitely many states in which a program can be.

In the following subsections, we discuss the formal specification of programs and their verification by proof respectively by model checking in more detail.

2.2.1 *Formal Specifications*

A formal specification differs from a normal specification in that it is written in clearly defined mathematically meaningful statements. A language that allows to formulate an algorithm and its specification in a mathematically precise way is called a specification language. As from that point onward the algorithm and its specification are clearly defined, the correctness of the algorithm with respect to its specification can be verified with mathematical rigor.

As an example consider the following algorithm:

Reversing an array
Input: array a with length n
Output: array b with length n where the elements are reversed

Loop $n/2$ times where in iteration i
    swap the i-th element with the i-th last element

Even though this algorithm is very intuitive, in practice it lacks some details, for example, if the array starts with 1 or 0. Therefore we need to formally specify the array:

Array
$elem = arbitrary\ set$
$nat = \mathbb{N}$
$index = \{x \in \mathbb{N} : x \leqslant n\}$

$Create : elem^n \to array$
$Set : array \times index \times elem \to array$
$Get : array \times index \to elem$
$Length : array \to nat$
$Create(a_0, ..., a_{n-1}) =< a_0, ..., a_{n-1} >$
$Set(< a_0, ..., a_{n-1} >, i, x) =< a_0, ..., a_{i-1}, x, a_{i+1}, ..., a_{n-1} >$
$Get(< a_0, ..., a_{n-1} >, i) = a_i$
$Length(< a_0, ..., a_{n-1} >) = n$

This specification in combination with Hoare-logic gives us a very basic specification language in which we can formalize our algorithm.

```
1 ReversingAnArray(a:array, n:ℕ, out: b:array)
2 {Length(a) = n}
3 b := a;
4 i := 0;
5 While i < ⌊n/2⌋ do
6   b := Set(b,i,Get(a,n-1-i));
7   b := Set(b,i,Get(a,i));
8   i := i + 1
9 {∀i∈{0,...,n-1}: Get(a,i) = Get(b,n-1-i)}
```

In order to prove the correctness of the algorithm we still need to find a termination term as well as a loop invariant.

The termination term must satisfy two conditions:

1. The termination term is always greater or equal 0.
2. The termination term gets smaller in each iteration.

We can derive our termination term from the loop condition:

$$T(i) := n - i$$

One can easily see that this term satisfies the needed conditions.

As of our loop invariant, it must satisfy the following conditions:

1. The loop invariant must hold before the first iteration.
2. The loop invariant is preserved by every iteration.
3. When the loop terminates, the loop invariant implies the post-condition.

To derive the loop invariant we can rewrite the post-condition in a way, that it is dependent on i [FMV14].

$$I(a,b,i) :<=> 0 \leqslant i < \lfloor n/2 \rfloor + 1 \wedge$$
$$\forall j \in \{0,...,n-1\} : \begin{cases} Get(a,j) = Get(b,n-1-j) & \text{if } j < i \vee j > n-1-j \\ Get(a,j) = Get(b,j) & \text{else} \end{cases}$$

For now, we will assume this is a valid loop invariant, we will prove its correctness in the next chapter.

2.2.2 *Proving*

Hoare provides a calculus (a set of inference rules and axioms) [Hoa69] with which one may validate Hoare-Triples. Among them are the following rules:

- **(D0) Axiom of Assignment**:
  $\vdash \{P_0\}\ x := y\ \{P\}$
  Where P_0 denotes the statement obtained from P by replacing all free occurrences of x with y.
- **(D1) Rules of Consequence**:
  if $\vdash \{P\}\ Q\ \{R\}$ and $\vdash (R \Rightarrow S)$ then $\vdash \{P\}\ Q\ \{S\}$
  if $\vdash \{P\}\ Q\ \{R\}$ and $\vdash (S \Rightarrow P)$ then $\vdash \{S\}\ Q\ \{R\}$
- **(D2) Rule of Composition**:
  if $\vdash \{P\}\ S\ \{Q\}$ and $\vdash \{Q\}\ T\ \{R\}$ then $\vdash \{P\}\ S;T\ \{R\}$
- **(D3) Rule of Iteration:**
  if $\vdash \{P \wedge B\}\ S\ \{P\}$ then $\vdash \{P\}$ while B do $S\ \{\neg B \wedge P\}$

We can now sketch a proof of our previous example.

Proof. For this sketch we will denote Pre as the precondition, $Post$ as the postcondition and S as the body of the while-loop. Aso for easier readability we will define $a[i] := Get(a,i)$ and $a_{[i]=x} := Set(a,i,x)$.
We can use (D2) to prove each command separately.

$$\begin{array}{l|l} \{Pre\}\ b{:=}a\ \{Pre \wedge b = a\} & (D0) \\ \{Pre \wedge b = a\}\ i := 0\ \{Pre \wedge b = a \wedge i = 0\} & (D0) \\ \{Pre \wedge b = a \wedge i = 0\}\ While\ i < \lfloor n/2 \rfloor\ do\ S\ \{Post\} & \end{array}$$

To prove the while-loop one must first prove

$$Pre \wedge b = a \wedge i = 0 \Rightarrow I(a,b,i)$$

After that we can use (D1) leaving us with

$$\{I(a,b,i)\}\ \mathit{While}\ i < \lfloor n/2 \rfloor \text{ do } S\ \{\text{Post}\}$$

Next one needs to prove that

$$I(a,b,i) \wedge\ i < \lfloor n/2 \rfloor \Rightarrow I(a, b_{[i]=a[n-i-1],[n-i-1]=a[i]}, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor$$

Applying (D1) to the pre-condition of the loop we can again use (D2)

$$\left.\begin{array}{l}\{I(a, b_{[i]=a[n-i-1],[n-i-1]=a[i]}, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor\} \\ b := \mathit{Set}(b, i, \mathit{Get}(a, n-1-i)) \\ \{I(a, b_{[n-i-1]=a[i]}, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor\}\end{array}\right\} (D0)$$

$$\left.\begin{array}{l}\{I(a, b_{[n-i-1]=a[i]}, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor\} \\ b := \mathit{Set}(b, n-1-i, \mathit{Get}(a, i)) \\ \{I(a, b, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor\}\end{array}\right\} (D0)$$

$$\left.\begin{array}{l}\{I(a, b, i+1) \wedge\ i+1 < \lfloor n/2 \rfloor\} \\ i := i+1 \\ \{I(a, b, i) \wedge\ i < \lfloor n/2 \rfloor\}\end{array}\right\} (D0)$$

We can now apply (D3) and conclude by proving

$$\neg(i < \lfloor n/2 \rfloor) \wedge I(a,b,i) \Rightarrow \text{Post}$$

and applying (D1). □

This method can be subsumed as finding a set of (verification) conditions whose validity implies the correctness of the algorithm. In case of a finite input domain these conditions can be checked via brute force. For an infinite input domains we can use automated reasoners (e.g. Satisfiability modulo theories solvers, short SMT solvers) to validate the conditions.

Dijkstra introduced 1975 the predicate transformer calculus [Dij75] that automates the process of finding these verification conditions. It is based on Hoare-Logic and uses the weakest precondition function. As the name may imply this function finds the weakest precondition for a given algorithm and post-condition.

2.2.3 *Model Checking*

The basic idea of model checking is that if we create a specification language that is in its core checkable, we can verify the algorithm automatically, by investigating all possible executions of a system. This can be done

by first translating the algorithm into a finite state machine and then investigating every reachable state. A finite state machine can always be represented as a directed graph (a so called Kripke structure). Paths on the graph may then be checked against different properties. There are in general two properties that interest us: safety and liveness. To imply safety in its simplest form we check if every dead end implies the post-condition. To imply liveness in its simplest form we check if every path leads to a dead end.

General properties of Kripke structures can be specified in temporal logic with two main variants, LTL and CTL. Linear Temporal Logic (LTL) considers a graph as a set of linear paths and can formulate statements about these paths. Computation Tree Logic (CTL) considers a graph as an infinite tree and can formulate statements about the various branches of that tree [MSS99].

2.2.3.1 *Explicit State Model Checking*

The original model proposed by Carke and Emerson [Cla+18] explicitly represented the Kripke structure by a directed graph. Since there are only finitely many possible paths, a computer can check every path for the safety and liveness properties and can give a counter example if one of the properties does not hold.

This model works for a small amount of states but as the number of variables grows the number of states grows exponentially. This problem is called the state explosion problem. One way to deal with this is to limit the domains of all variables such that the number of states still grows but at a manageable rate.

2.2.3.2 *Symbolic Model Checking*

Another way to model a programming language is to represent different sets of states by boolean functions. There are different ways how this can be used for checking.

The original method uses Binary Decision Diagrams (BDDs) to describe in a compact form ("symbolically") the set of states from which a system satisfies a specification; this set is iteratively expanded until no more states can be added. If than all initial states are in the set, the system satisfies the specification.

A different approach is called Bounded Model Checking (BMC). The idea is to search for counterexamples on paths that have at most k transitions. If no counterexample was found, k is increased. This process terminates either if a given upper bound for k is reached or all paths have been traversed [Bie+03].

Another method is Counterexample-guided Abstraction Refinement (CEGAR). This Method uses an abstract model derived from the original model

by only considering the values of a fixed set of predicates in every state. If a specification is true in the more abstract model it is also true for the original model. If a counterexample was found, this example might not necessarily be present for the original model. If this happens, the abstraction must be refined in such a way that the counterexample is eliminated [Cla+00].

2.3 LANGUAGES AND TOOLS

The following list presents different verification tools and associated reasoners. We will look at each tool and the according programs to describe the process of specifying, formalizing and verifying algorithms.

- **Dafny** [Lei17] is a programming language that supports formal specification as well as verification. It uses an intermediate verification language called Boogie to generate verification conditions from programs. It verifies using the SMT Solver Z3. Because this is a proper programming language, this means that one can only validate an implementation of an algorithm, not the algorithm in its most general case.

- **Java Modeling Language (JML)** [TGK09] is a specification language where specifications are embedded in valid Java code. Again this means that one can only validate specific implementations. But different to Dafny, Java is broadly used in the industry, making JML quite useful for industry-purposes. There are various tools that work with JML, e.g:
  - **ESC/Java** [Fla+13] is a tool that tries to find run-time errors at compile-time by a limited form of proving. Is has an own theorem prover, but later versions also support different third-party provers. ESC/Java is the predecessor of OpenJML.
  - **OpenJML** [Cok14] is a program verification tool that can be used either for static or runtime checking. It can be used with many common SMT Solvers like Z3, CVC4 and Yices.
  - **KeY** [Ahr+16] is a system used for formal verification of algorithms specified in JML. It may be used in combination with an SMT-Solver or as a proof assistant.

- **Krakatoa Verification Tool** [TGK09] is a front-end for the verification platform Why. It has its own specification language in the spirit of JML but works for Java as well as C. It generates verification conditions which then can be used by provers like Coq, PVS, Isabelle/HOL and many more.

- **Rodin Platform** [Abr+10] is an Eclipse-based IDE for reasoning in the formal method Event-B, a method that provides specification, formalization and generation of verification conditions. Event-B uses the abstract machine notation (AMN) as its specification language. Rodin uses a proof assistant instead of an SMT Solver, where the proof must be done manually with assistance of a computer.

- **Vienna Development Method (VDM)** [Bjø79] is a formal method that uses its own specification language capable of defining procedural programs as well as object orientated systems. It works with the external developed Overture Tool, an open source IDE that is build on top of the Eclipse-IDE.

- **PlusCal/TLA+** [LM09] TLA+ is a temporal logic, meaning a logic that supports time-related statements. PlusCal is a specification language that trans-complies into TLA+. Properties of finite state models specified in TLA+ can be verified with the TLA+ model checker (TLC).

- **Alloy** [Jac06] Is a specification language that defines relations between objects. It is used with the formal method Alloy Analyzer that operates on finite domains and uses explicit model checking as well as a build-in SAT Solver (similar to SMT but for finite domains). Different to the previous tools, algorithms formalized in Alloy must be of a relation-based nature. While it is Turing complete, it was not designed to work with function-oriented algorithms.

Daniela Ritirc looked at some of these tools in heir master thesis [Rit16] and noticed that there is no perfect software for verifying mathematical algorithms. For example, she noticed that Event-B had increasing difficulties proving larger invariants, VDM could not always deliver the verification conditions and PlusCal/TLA+ does not allow recursive algorithms.

2.4 RISCAL

RISCAL [Figure 1] is being developed by Wolfgang Schreiner at the RISC Institute [SBF18]. It is a so called "algorithm language", meaning it can be used to specify and verify abstract programs ("algorithms"). It uses a model with finite input domains and validates them using brute force. Starting with version 2.0 it is now possible to automatically generate verification conditions. Before that version one needed to find them manually using the invariant. Once these verification conditions are derived one can in the future use a proof assistant such as the RISCProofNavigator, also developed by Wolfgang Schreiner, to verify them. The language itself supports Unicode-symbols making a model look very similar to a mathematical algorithm.

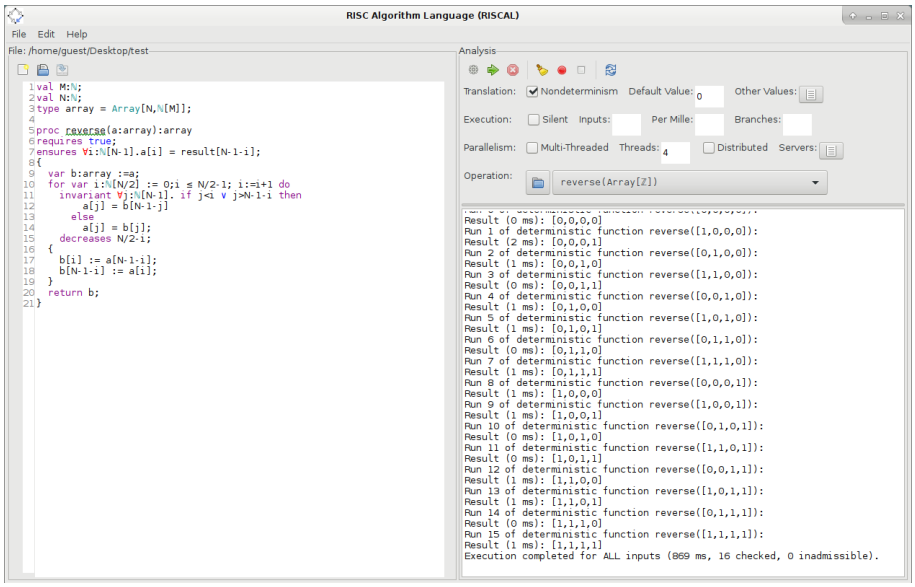


Figure 1: The RISCAL interface

Its mathematical model consists of several parts: types, predicates, functions, theorems and procedures. These can be used not only to implement an algorithm but also so formulate theorems around the algorithms like said verification conditions, but also various statement regarding attributes of pre-conditon, post-conditon, invariant as well as termination. Additionally, it can generate conditions that may be checked to validate the meaningfulness of specifications [Figure 2]. The system can check all statements and even provide a counter example if a statement does not hold. With this method the formalization of an algorithm can be assisted by the RISCAL software.

Finally, we will formalize our example using the RISCAL language:

```
1  val M:ℕ;
2  val N:ℕ;
3  type array = Array[N,ℕ[M]];
4
5  proc reverse(a:array):array
6  requires true;
7  ensures ∀i:ℕ[N-1].a[i] = result[N-1-i];
8  {
9    var b:array :=a;
10   for var i:ℕ[N/2] := 0;i ⩽ N/2-1; i:=i+1 do
11     invariant ∀j:ℕ[N-1]. if j<i ∨ j>N-1-i then
12         a[j] = b[N-1-j]
13       else
14         a[j] = b[j];
15     decreases N/2-i;
16   {
```

```
17      b[i] := a[N-1-i];
18      b[N-1-i] := a[i];
19    }
20    return b;
21  }
```

As we can see, the program is still quite intuitively readable. Pre-condition, post-condition, invariant and termination-term are all part of the formalization, so that they are considered when RISCAL validates the code. Also, RISCAL supports the use of quantifiers which, because of the finiteness of the variable domains, are checkable. For this thesis, we will use RISCAL to formalize algorithms.

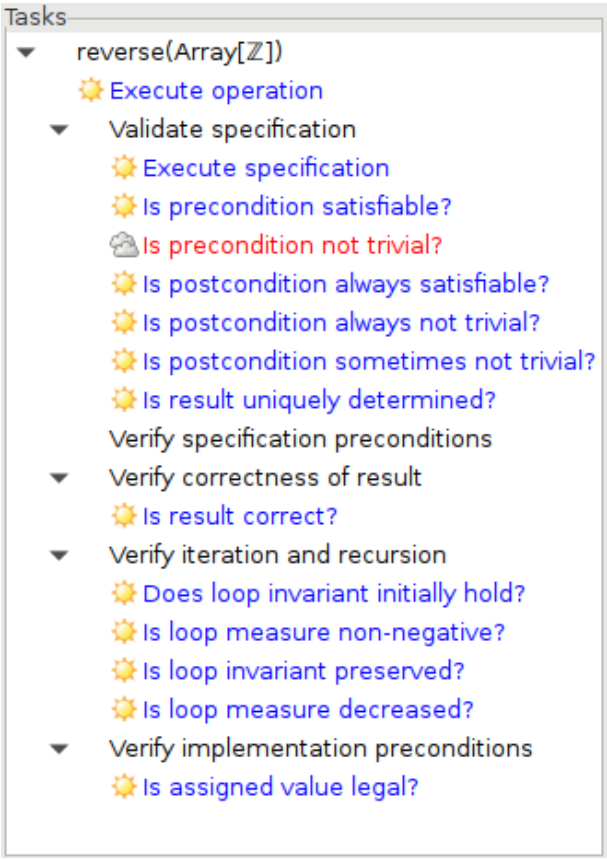


Figure 2: The task bar containing tools to generate the verification conditions. Once the task gets successfully executed, the cloud turns into a sun, indicating that the check is valid.

3 SEARCH ALGORITHMS

The most basic problem for a list a is to determine if an element x is an element of the list and if so, to return an index i so that $a[i] = x$. This problem can be specified in predicate logic with the help of the following function:

$$contains(x, a) = \begin{cases} -1 & \text{if } \forall i : a[i] \neq x \\ \text{choose } i : a[i] = x & \text{else} \end{cases}$$

This will be the main element of the post-condition for our searching algorithms. To sharpen the condition, we will often demand the output to be uniquely defined as the minimum index i with $a[i] = x$. In this case, there will be for any given input only one possible output that serves the post-condition. As this notion depends on the algorithm, we will need to formulate the post-condition for every algorithm individually.

3.1 BASIC ALGORITHMS

The most basic algorithm solving our problem is the linear search algorithm:

Linear Search
Input: an array a and an element x
Output: index i so that $a[i] = x$ respectively -1 if such an index does not exist.

Loop over every element where in iteration i
    Check if $a[i] = x$ and if so return i
Return -1

This algorithm works for all arrays, i.e. only the trivial precondition "true" is required. From the analysis of the algorithm we know that it needs $\mathcal{O}(n)$ steps. To ensure the uniqueness of the result the postcondition will also need to state that the resulting element is the first occurrence of x in a.

A more efficient algorithm is the binary search algorithm, as it only takes $\mathcal{O}(\log(n))$ amount of time [SW16]. To achieve this, it requires a sorted array as an input. The basic idea is the following:

Binary Search
Input: sorted array a with length n, element x
Output: index i so that a[i]=x respectively -1 if such an index does not exist.

Let $left := 0, right := n$
While $left < right$ and x is not found
    If the $middle := left + (right - left)/2$ element is not equal to x
        If $a[middle] > x$ set $right := middle - 1$
        else set $left := middle + 1$
    else return middle

This algorithm does not necessarily return the first occurrence of x. This can be simply archived by subsequently decreasing the found index, if x also appears at a smaller index.

3.2 DATA TYPES

We are now going to investigate three possible datatypes representations for sequences: Arrays, Recursive Lists and Pointer-Linked Lists.

ARRAYS Arrays are a primitive data type in RISCAL:

```
1   val M:ℕ;
2   val N:ℕ;
3   type nat = ℕ[M];
4   type array = Array[N,nat];
```

In our case we are using arrays over a bound subset of the natural numbers.

RECURSIVE LISTS In RISCAL a list is no primitive data type and can be implemented using a recursive type-declaration:

```
5   rectype(N) list = nil | cons(nat,list);
```

For such recursive types, also algorithms are naturally defined in a recursive style. While in RISCAL arrays have fixed length N, lists have a *maximal* length N. Therefore, we will need a function that calculates the length of a given list:

```
6  //listLength(a:list)
7  //  returns the length of a  list
8  fun listLengthRec(a:list,out:ℕ[N]):ℕ[N]
9    decreases N-out;
10   = match a with
```

```
11   {
12     nil -> out;
13     cons(elem:nat,rem:list) -> listLengthRec(rem,out+1);
14   };
15 fun listLength(a:list):ℕ[N] = listLengthRec(a,0);
```

As we can see, we need to match the pattern of a list before its elements can be used.

Another useful function is a get-function to allow random-access on lists. We will use it in the post-condition of our search algorithm to specify that $get(a, i) = x$.

```
16 //get(a:list,i:index):nat
17 //  returns the i-th element
18 fun get(a:list,i:index):nat
19   requires i < listLength(a);
20   decreases i;
21   = match a with
22   {
23     cons(elem:nat,rem:list) ->
24       if i = 0 then
25         elem
26       else
27         get(rem,i-1);
28   };
```

POINTER-LINKED LISTS Especially in low-level applications a list is better represented as a pointer, i.e. as an array of "cells" each of which is a tuple of an element and another pointer. We call such an array a "store".

```
29 type index = ℕ[N-1];
30 type cell = Record[head:nat,tail:pointer];
31 type store = Array[N,cell];
32
33 type pointer = ℕ[N];
34 val null:pointer = N;
```

We call the element of a cell its "head" and the pointer its "tail". In this way a pointer may represent a recursive list where the head is the first element and the tail points to another list. In order to consider a pointer as a list, the sequence of cells reachable from that pointer must eventually terminate. This path can be obtained by recursively getting the cell corresponding to the tail of the previous cell. An empty list is represented as a special pointer called "null". For a list to terminate the sequence of cells will eventually lead to a null pointer. Therefore we describe a list by a predicate $leadsTo$ that holds if one pointer leads to another via a sequence of intermediate

cells. The definition can be either recursive or use a list of all pointers on the path and check whether the specific pointer is an element of the list.

```
39 //leadsTo(a:pointer,b:pointer,s:store,counter:ℕ[N+1])
40 //  recursivly returns if the cell pointed by b is part
41 //  of the list a
42 pred leadsToRec(a:pointer,b:pointer,s:store,counter:ℕ[N+1])
43   decreases counter;
44 ⇔ counter > 0
45   ∧ (
46     a ≠ b
47     ⇒ a ≠ null
48     ∧ leadsToRec(s[a].tail,b,s,counter-1)
49   );
50 pred leadsTo(a:pointer,b:pointer,s:store)
51 ⇔ leadsToRec(a,b,s,N+1);
52
53 //leadsTo2(a:pointer,b:pointer,s:store)
54 //  returns if the cell pointed by b is part of the list a
55 pred leadsTo2(a:pointer,b:pointer,s:store)
56 ⇔ ∃ps:Array[N+1,pointer].ps[0] = a
57   ∧ (
58       ∃j:ℕ[N].ps[j] = b
59       ∧ ∀i:ℕ[N-1].i<j ⇒
60       ps[i+1] = if ps[i] ≠ null then
61         s[ps[i]].tail
62       else
63         null
64     );
```

We also claim the equivalence of these two implementations with respect to our model.

```
39 //leadsToAndLeadsTo2AreEquivalent()
40 //  leadTo and LeadTo2 are Equivalent
41 theorem leadsToAndLeadsTo2AreEquivalent()
42 ⇔ ∀a:pointer.∀b:pointer.∀s:store.
43   leadsTo(a,b,s) ⇔ leadsTo2(a,b,s);
```

With this we now claim that a well-defined list has no cycles:

```
39 theorem DefListIsWellDefined()
40 ⇔ ∀s:store.∀a:pointer.(
41     ∀b:pointer.b ≠ null ∧ leadsTo(a,b,s)
42     ⇒ ¬leadsTo(s[b].tail,b,s)
43   )
44   ⇔ leadsTo(a,N,s);
```

3.3 LINEAR SEARCH

In this subsection we will discussed different implementations for the linear search algorithm.

3.3.1 *Implementation for Arrays*

For arrays we have the following implementation of "linear search":

```
108 proc linearSearch(a:array, x:nat):ret
109   requires pre(a,x);
110   ensures post(a,x,result);
111 {
112   var out:ret := -1;
113   for var i:ℕ[N-1+1] := 0;i < N ∧ out = -1; i := i+1 do
114     invariant inv(a,x,out,i);
115     decreases termin(i);
116   {
117     if a[i] = x then
118       out := i;
119   }
120   return out;
121 }
```

This algorithm is specified/annotated with the help of the following predicates:

```
30 //pre-Condition
31 //  (*) trivial
32 pred pre(a:array,x:nat) ⇔ true;
33
34 //post-Condition
35 //  (*) result is negative if a does not contain x
36 //  (*) else its the index of x
37 //  (*) and result is the first appearance of x
38 pred post(a:array,x:nat,result:ret)
39 ⇔ ((∃i:index.a[i] = x) ⇔ result ⩾ 0)
40   ∧ (
41     result ⩾ 0
42     ⇒ a[result] = x
43     ∧ isFirstOccurrence(a,x,result)
44   );
45
46 //invariant
47 //  (*) if out is -1, then in the array up to i, x has
48 //      not appeared
49 //  (*) else out is the last iteration
```

```
50 //  (*) and its the index of x
51 //  (*) and its the first appearance of x
52 pred inv(a:array,x:nat,out:ret,i:ℕ[N-1+1])
53 ⇔ if out = -1 then
54     ∀j:index.j < i ⇒ a[j] ≠ x
55   else
56     out = i-1
57     ∧ a[out] = x
58     ∧ isFirstOccurrence(a,x,out);
59
60 //termination term
61 //  it terminates after at most N iterations
62 fun termin(i:ℕ[N-1+1]):ℕ[N] = N-i;
```

We define the post-condition with the help of an auxiliary predicate that states what it means to be the first occurrence of a value in an array:

```
22 //isFirstOccurrence(a:array,x:nat,i:index)
23 //  holds if the first occurrence of x in a is i
24 pred isFirstOccurrence(a:array,x:nat,i:index)
25 ⇔  ∀j:index.a[j] = x ⇒ i ⩽ j;
```

Next we need to formulate an invariant for the loop contained in the algorithm. Note that the upper bound of i is increased by 1 to ensure that $i < N$ can fail. To denote this we write $\mathbb{N}[N-1+1]$, which should be read as $index \cup \{N\}$. The invariant can be derived from the post-condition by formulating the condition respect to the all variables that are used in the loop (in our case i and out). This new condition should imply the post-condition if the loop-condition is invalid (in our case: $i \geqslant N \vee out \neq -1$). This can be achieved by replacing $result$ with out and by only considering the all element in a from 0 to $i-1$. The termination term is $N-i$. This is the equivalent to saying that the loop will run for at most N times. If not explicitly described, from now on, if not explicitly stated, we will use $upperBound(i)-i$, as the termination term.

3.3.2 *Implementation for Recursive Lists*

For recursive lists the recursive implementation is as follows.

```
157 fun linearSearch(a:list, x:nat):ret
158   requires pre(a,x);
159   ensures post(a,x,result);
160   decreases termin(a,x);
161 = match a with
162   {
163     nil -> -1;
```

```
164     cons(elem:nat,rem:list) ->
165       if elem = x then
166         0
167       else if linearSearch(rem,x) ⩾ 0 then
168         linearSearch(rem,x) + 1
169       else
170         -1;
171   };
```

This algorithm is specified/annotated with the help of the following predicates:

```
84  //pre-Condition
85  //  (*) trivial
86  pred pre(a:list,x:nat) ⇔ true;
87
88  //post-Condition
89  //  (*) result is negative iff a contains x
90  //  (*) else its the index of x
91  //  (*) and its the first appearance of x
92  pred post(a:list,x:nat,result:ret)
93  ⇔ (result ⩾ 0 ⇔ contains(a,x))
94    ∧ (
95      result ⩾ 0
96      ⇒ result < listLength(a)
97      ∧ get(a,result) = x
98      ∧ isFirstOccurrence(a,x,result)
99    );
100
101 //termination term
102 //  each recursion the array-size gets smaller
103 fun termin(a:list,x:nat):ℕ[N] = listLength(a);
```

The function $\mathrm{FirstOccurrence}(a, x, i)$ is defined recursively.

```
64  //isFirstOccurrence(a:list,x:nat,i:index)
65  //  returns if x appears first at i-th position
66  pred isFirstOccurrence(a:list,x:nat,i:index)
67    requires i < listLength(a) ∧ get(a,i) = x;
68    decreases listLength(a);
69  ⇔ match a with
70    {
71      nil -> false;
72      cons(elem:nat,rem:list) ->
73        if i = 0 then
74          true
75        else if elem = x then
```

```
76          false
77        else
78          isFirstOccurrence(rem,x,i-1);
79    };
```

We will also need to implement a function $contains$ that plays the counterpart of the condition $\exists i : index.a[i] = x$ for arrays.

```
64  //contains(a:list,x:nat)
65  //  states if x is in a
66  pred contains(a:list,x:nat)
67    decreases listLength(a);
68  ⇔ match a with
69    {
70      nil -> false;
71      cons(elem:nat,rem:list) ->
72        if elem = x then
73          true
74        else
75          contains(rem,x);
76    };
```

Consequently we can formalize the post-condition in the same way as for arrays. Looking at the formalization of the algorithm we notice that in every iteration the list gets smaller. This motivates the termination term.

```
101 //termination term
102 //  each recursion the array-size gets smaller
103 fun termin(a:list,x:nat):ℕ[N] = listLength(a);
```

From now on we will also expect the termination term for an algorithm with lists to be the length of the list.

3.3.3 *Implementation for Pointer Lists*

For pointer list we use a imperative algorithm.

```
210 proc linearSearch(a:pointer,x:nat, s:store):pointer
211   requires pre(a,x,s);
212   ensures post(a,x,s,result);
213 {
214   var out:pointer := null;
215   var b:pointer := a;
216   while b ≠ null ∧ out = null do
217     invariant inv(a,b,x,s,out);
218     decreases termin(b,out,s);
219   {
```

```
220     if s[b].head = x then
221       out := b;
222     else
223       b := s[b].tail;
224   }
225   return out;
226 }
```

This algorithm is specified/annotated with the help of the following predicates:

```
111 //pre-condition
112 //  (*) s is well formed
113 pred pre(a:pointer,x:nat, s:store)
114 ⇔ leadsTo(a,null,s);
115
116 //post-condition
117 //  (*) result is negative is a does not contain x
118 //  (*) if result is not negative, its the index of the first
119 //      occurrence of x in a
120 pred post(a:pointer,x:nat,s:store,result:pointer)
121 ⇔ (result ≠ null ⇔ contains(a,s,x))
122   ∧ (
123     result ≠ null
124     ⇒ leadsTo(a,result,s)
125     ∧ s[result].head = x
126     ∧ isFirstOccurrence(a,s,x,result)
127   );
128
129 //invariant
130 //  (*) if out is N, then x only appears in a if it also
131 //      appears in b
132 //  (*) and if b is not N, its part of the list a
133 //  (*) if b is N, and out is N, then x is not contained
134 //      in a
135 //  (*) if out is not N, i must return the first occurence
136 //      of x in a
137 pred inv(a:pointer,b:pointer,x:nat,s:store,out:pointer)
138 ⇔ if out = null then(
139     (
140       ∀c:pointer.c ≠ null ∧ s[c].head = x ∧ leadsTo(a,c,s)
141       ⇒ leadsTo(b,c,s)
142     )
143     ∧ if b ≠ null then
144       leadsTo(a,b,s)
145     else
146       ¬contains(a,s,x)
```

```
147   )
148   else (
149     out = b
150     ∧ leadsTo(a,b,s)
151     ∧ s[b].head = x
152     ∧ isFirstOccurrence(a,s,x,out)
153   );
154
155 //termin(i:ℕ[N-1+1]):ℕ[N]
156 //  the length of the array, but if an element was found,
157 //  return 0
158 fun termin(b:pointer,out:pointer,s:store):ℕ[N]
159 = listLength(b,s)-listLength(out,s);
```

As a pre-condition we require that the list is well-defined. In order to randomly access a cell in the pointer list, we need to ensure that the pointer is actually part of the array. This means instead of $a[result]$ we need to write $leadsTo(a, result, s) \wedge s[result].head = x$. With this in mind we can again use the post-condition for arrays to derive the new conditions. This requires the result to be a pointer instead of an index. In line 221, b does not get reduced. This means the termination term can not be the length of b. We know out is $null$ until x was found and therefore the length of out is mostly zero. This motivates the termination term. For the invariant the case that x was found can be taken from the post-condition. For the case that x was not yet found we need do define what that means: as we only have the two lists that are accessed though the two pointers a and b, we can define "x is not yet found" as "if x is in a, it must be in b" as b is the list of all elements that still need to be checked. This also means that a should lead to b.

3.4 BINARY SEARCH

Binary Search works only for sorted arrays. For the implementation we verify both the imperative and recursive version.

```
173 proc binarySearch(a:array, x:nat):ret
174   requires pre(a,x);
175   ensures post(a,x,result);
176 {
177   var out:ret := -1;
178   var left:ℕ[N-1+1] := 0;
179   var right:ℤ[-1,N-1] := N-1;
180   while out = -1 ∧ left ⩽ right do
181     invariant inv(a,x,left,right,out);
182     decreases termin(left,right,out);
```

```
183   {
184     var i:index := left + (right-left)/2;
185     if a[i] = x then
186       out := i;
187     else if a[i] > x then
188       right := i-1;
189     else
190       left := i+1;
191   }
192   return out;
193 }
194
195 //Recursive implementaion
196 fun binarySearchRec(a:array, x:nat, left:ℕ[N-1+1], right:ℤ[-1,N-1]):ret
197   requires preRec(a,x,left,right);
198   ensures post(a,x,result);
199   decreases terminRec(left,right);
200 = if left > right then
201     -1
202   else if a[i(left,right)] = x then
203     i(left,right)
204   else if a[i(left,right)] > x then
205     binarySearchRec(a,x,left,i(left,right)-1)
206   else
207     binarySearchRec(a,x,i(left,right)+1,right);
208 fun binarySearchRecursive(a:array, x:nat):ret
209   requires pre(a,x);
210   ensures post(a,x,result);
211 = binarySearchRec(a,x,0,N-1);
```

This algorithm is specified/annotated with the help of the following predicates:

```
37 //pre-Condition
38 //  (*) a is sorted
39 pred pre(a:array,x:nat)
40 ⇔ isSorted(a,0,N-1);
41
42 //post-Condition
43 //  (*) result is negative if a does not contain x
44 //  (*) else its the index of x
45 pred post(a:array,x:nat,result:ret)
46 ⇔ (result = -1 ⇔ ∀i:index.a[i] ≠ x)
47   ∧ (result ⩾ 0 ⇒ a[result] = x);
48
49 //invariant
50 //  (*) if out is not negative, a[out] = x
```

```
51 //  (*) if out is negative and or left is smaller then
52 //      right, then x is found
53 //  (*) else x does not occur left of left or right of right
54 pred inv(a:array,x:nat,left:ℕ[N-1+1],right:ℤ[-1,N-1],out:ret)
55 ⇔ -1 ⩽ right-left
56   ∧ (out ⩾ 0 ⇒ a[out] = x)
57   ∧ (
58     out = -1 ∨ left ⩽ right
59     ⇒ ∀i:index.(0 ⩽ i ∧ i < left) ∨ (right < i ∧ i ⩽ N-1)
60     ⇒ a[i] ≠ x
61   );
62
63 //pre-Condition for recursive implementation
64 //  (*) normal pre-condition must hold
65 //  (*) invariant must hold
66 pred preRec(a:array,x:nat,left:ℕ[N-1+1],right:ℤ[-1,N-1])
67 ⇔ pre(a,x) ∧ inv(a,x,left,right,-1);
68
69 //termination term
70 //  the indexes left and right are moving closer to each
71 //  other.
72 fun termin(left:ℕ[N-1+1],right:ℤ[-1,N-1],out:ret):ℕ[2*N+1]
73 = if out = -1 then right-left+1 else 0;
74
75 //termination term for recursive implementation
76 //  the indexes left and right are moving closer to each
77 //  other.
78 fun terminRec(left:ℕ[N-1+1],right:ℤ[-1,N-1]):ℕ[N]
79 = right-left+1;
```

We start with the imperative version. First of all, we denote when a array is sorted.

```
29 //isSorted(a:array,m:index,n:index)
30 //  states if all elements from m to n are sorted
31 pred isSorted(a:array,m:index,n:index)
32 ⇔ ∀i:index.m ⩽ i ∧ i < n ⇒ a[i] ⩽ a[i+1];
```

The pre-condition just states that the array is sorted. The post-condition is the same as for linear search with array without the result being the first occurrence. For the invariant we need to rewrite the post-condition using left,right and out. The interval [left, right] contains all unchecked indexes. Therefore, we will replace a with the complement of [left, right]. We also replace result with out. If left is bigger than right, than only at most by one. Left being bigger than right denotes that the element does not exist in the array and that out $= -1$. For the termination term right $-$ left is a strong candidate, but again if we consider line 186 in the implementation,

we see that we need to use out as well. A termination term that works is as followed.

```
69 //termination term
70 //  the indexes left and right are moving closer to each
71 //  other.
72 fun termin(left:ℕ[N-1+1],right:ℤ[-1,N-1],out:ret):ℕ[2*N+1]
73 = if out = -1 then right-left+1 else 0;
```

For the recursive version we need to formulate the post-condition such that it contains the relation between left, right and a. This version should reuse as much as possible from the imperative version. We therefore use the invariant to state the relation of left and right.

```
63 //pre-Condition for recursive implementation
64 //  (*) normal pre-condition must hold
65 //  (*) invariant must hold
66 pred preRec(a:array,x:nat,left:ℕ[N-1+1],right:ℤ[-1,N-1])
67 ⇔ pre(a,x) ∧ inv(a,x,left,right,-1);
```

We can now look at the formalization. Here i(left, right) denotes the same variable as introduced in the imperative version.

```
22 //i(left:index,right:index)
23 //  returns the middle index between left and right
24 fun i(left:index,right:index):ret = left+(right-left)/2;
```

The termination term only needs to be considered for the recursive parts of the algorithm. We can use the original candidate:

```
75 //termination term for recursive implementation
76 //  the indexes left and right are moving closer to each
77 //  other.
78 fun terminRec(left:ℕ[N-1+1],right:ℤ[-1,N-1]):ℕ[N]
79 = right-left+1;
```

Note that left can be at most 1 bigger than right.

3.5 CHECKING THE MODEL

Before we check the conditions, we may decide to further validate our pre- and post-conditions.

1. To ensure that the pre-condition is not trivial, there should be at least one case where the pre-condition does not hold.
2. For the post-condition we can check whether for every input there exists exactly one possible output that serves the post-condition.

RISCAL has these checks built in, but they are not always necessary. Some times, additional modifications are required, that might slow down the algorithm.

To actually check the claims that we made in the previous section we need to validate them. This will be done by:

1. Checking that all annotations of the algorithm (specifications and invariants) hold for all possible inputs.

2. Checking that the (manually derived) verification conditions hold. These checks are performed for restricted models, which does not prove correctness of all models but increases our confidence in correctness.

For these checks we use N = 4, M = 1. This means we only consider arrays and lists of size 4 and with elements in $\{0, 1\}$ (because the checking complexities grows exponentially, $M = 2$ already takes considerable time).

3.5.1 *Validating Specifications for Imperative Algorithms*

Even if a specifications holds for an algorithm, this does not make the algorithm valid. The specification could be wrong or prohibits inputs that we actually would like to allow. Therefore, we need to check manually if the algorithm works, not only for regular cases but in particular for possible special cases. This can be done by letting RISCAL execute all possible inputs and then checking if the particular special case was considered. Next we use the checking capacities of RISCAL to execute the algorithm for all inputs (allowed by the pre-condition). If this process runs without any errors, it gives a few insights:

1. The algorithm terminates and the termination terms are decreasing in each iteration.

2. If the pre-conditions holds for any case and the algorithm ensures the post-condition, we can say that the pre-condition is not too strong.

3. The post-condition is not too strong. This means that all results satisfy the post-condition.

4. The loop invariants are not too strong. This means that for each iteration the loop invariant holds.

In order for the loop invariant to be not too weak we need to define verification conditions. These conditions may then be used for proving the correctness of the algorithm. This is why the validation of the termination term will again be part of this verification condition. These prerequisites are as follows:

1. Given the pre-condition is true, the invariant holds before the loop is started.
2. Given the pre-condition is true, the termination term is never negative.
3. Given the pre-condition is true, the invariant is preserved by every loop iteration and the termination term decreases for every iteration.
4. Given the pre-condition is true, on termination the invariant implies the post-condition.

Except of the third one, these conditions are very generic and look very similar. The following verification conditions are for the linear search algorithm for arrays.

```
71  //Verification Condition 1
72  //   given the precondition,
73  //   (*) invariant holds before the loop started
74  theorem VC_beginning(a:array, x:nat, out:ret, i:ℕ[N-1+1])
75    requires pre(a,x);
76  ⇔ i = 0 ∧ out = -1 ⇒ inv(a, x, out, i);
77
78  //Verification Condition 2
79  //  given the precondition,
80  //  (*) termination term is never negative
81  theorem VC_termination(a:array, x:nat, out:ret, i:ℕ[N-1+1])
82    requires pre(a,x);
83  ⇔ inv(a, x, out, i) ⇒ termin(i)⩾0;
84
85  //Verification Condition 3
86  //  given the precondition,
87  //  (*) invariant is preserved by every loop iteration
88  //  (*) termination term decreases at every iteration
89  theorem VC_iteration(a:array, x:nat, out:ret, i:ℕ[N-1+1])
90    requires pre(a,x);
91  ⇔ inv(a, x, out, i) ∧ loop_cond(out,i)
92    ⇒ if a[i] = x then
93        inv(a,x, i,i+1)
94      else
95        inv(a,x,out,i+1)
96    ∧ termin(i+1) < termin(i);
97
98  //Verification Condition 4
99  //  given the precondition,
100 //  (*) on termination invariant implies postcondition
101 theorem VC_end(a:array, x:nat, out:ret, i:ℕ[N-1+1])
```

```
102   requires pre(a,x);
103 ⇔ inv(a, x, out, i) ∧ ¬loop_cond(out,i) ⇒ post(a,x,out);
```

For the third condition we take the loop invariant and specify how the inputs for the invariant have changed after an iteration. Verification conditions for each algorithm can be found in the appendix.

3.5.2 *Validating Specifications for Recursive Algorithms*

For recursive algorithms most of the validation procedure stays the same. Because any recursive function may run indefinitely, a termination term must be defined for each one of them. Recursive algorithms do not have any loops but instead nested functions, therefore it satisfies to define function specifications instead of verification conditions. These specifications are as follows:

1. Given the precondition holds, all preconditions of the sub-functions hold.
2. Given the precondition holds, the postcondition holds given that all sub-functions can be defined by their postconditions.
3. Given the precondition holds, the termination term is always non negative and each recursive function application reduces the termination term.

The following function specifications are for the linear search algorithm for recursive lists.

```
108 //Function Specification 1
109 //   given the precondition,
110 //   (*) all preconditions of the subfunctions hold.
111 theorem FS_pre(a:list,x:nat)
112   requires pre(a,x);
113 ⇔ match a with
114   {
115     nil -> true;
116     cons(elem:nat,rem:list) ->
117       elem ≠ x ⇒ pre(rem,x);
118   };
119
120 //Function Specification 2
121 //  given the precondition,
122 //  (*) the postcondition holds given that all subfunctions
123 //      can be defined by there postconditions.
124 theorem FS_post(a:list,x:nat)
125   requires pre(a,x);
```

```
126 ⇔ match a with
127   {
128     nil -> post(a,x,-1);
129     cons(elem:nat,rem:list) ->
130       if elem = x then
131         post(a,x,0)
132       else
133         ∀b:ret.post(rem,x,b)
134         ⇒ if b ⩾ 0 then
135           post(a,x,b+1)
136         else
137           post(a,x,b);
138   };
139
140 //Function Specification 3
141 //  given the precondition,
142 //  (*) the termination term always positive or zero
143 //  (*) each recursion reduces the termination term
144 theorem FS_termination(a:list,x:nat)
145   requires pre(a,x);
146 ⇔ termin(a,x) ⩾ 0
147   ∧ match a with
148   {
149     nil -> true;
150     cons(elem:nat,rem:list) ->
151       elem ≠ x ⇒ termin(rem,x) < termin(a,x);
152   };
```

3.5.3 *Outcome of the Checks*

For the linear search algorithm we have checked that the post-condition is unique. After that, we have checked if the defined pre- and post-condition as well as the invariant is not too strong by using RISCAL to check every input. Finally, the verification conditions were derived and again checked by again using RISCAL to check all cases. All checks where evaluated successfully, therefore we are now confident that the verification conditions can be used to prove the correctness with respect to the specifications.

For the binary search algorithm we have checked that the pre-condition is not trivial. Next we have checked that the specifications (pre-condition, post-condition and invariant) are not too strong by letting RISCAL check all cases. Finally, we have derived and validated the verification conditions. The verification conditions may now be further used to fully prove the correctness.

3.6 CONCLUSION

The searching algorithms are a nice introduction to the world of formal checking.

The linear search algorithm demonstrated nicely the core differences in the different programming styles: Imperative algorithms prefers arrays and use verification conditions for its loops. Recursive algorithms prefer recursive lists and use function specifications. Pointer lists can be used with both, but are not always equally efficient. In the case of linear search, an imperative algorithm turned out to be the right choice.

The binary search algorithm used a specific part of the array that can be anywhere within the array. Therefore the use of recursive lists is impractical as it relies on random access. Nevertheless we have validated both an imperative as well as an recursive implementation.

4 INSERTIONSORT

In the following chapters we will focus on sorting algorithms. The core of the postcondition of these algorithms is represented by the predicate $\mathit{isSorted}$ which we have already used in the pre-condition of binary search. However, we also need to ensure that the resulting array is a permutation of the original one. For this we will specify a corresponding predicate:

```
1 //isPermutationOf(a:array,b:array)
2 //  states if a is a permutation of the input array
3 pred isPermutationOf(a:array,b:array)
4 ⇔ ∃p:Array[N,index].
5     (∀i:index,j:index. i ≠ j ⇒ p[i] ≠ p[j])
6     ∧(∀i:index. a[i] = b[p[i]]);
```

With this we can now specify the problem for an array a.

$$\mathit{sort}(a) = \text{find array } b : \mathit{isPermutationOf}(a, b) \wedge \mathit{isSorted}(b)$$

The resulting array b is unique, therefore we can demand the output to be unique as well. For imperative (non recursive) algorithms we can use this post-condition as is. For (recursive) functional algorithms we will need to add additional conditions depending on the individual algorithm.

4.1 INSERTIONSORT

A very basic algorithm is the so called InsertionSort. This algorithm is most commonly used by humans when ordering playing cards.

InsertionSort
Input : An array a
Output : An array b so that its the sorted equivalent to a.

Loop over every element where in iteration i
    Loop over every element down from i to 0 with index j
        If $a[j] < a[i]$ then
            Shift the elements $[j+1, i-1]$ up
            Replace $a[j+1]$ with the original $a[i]$

4.1.1 *Implementation for Arrays*

For arrays, we have implemented the algorithm the following way:

```
176 proc insertionSort(a:array): array
177   requires pre(a);
178   ensures post(a,result);
179 {
180   var b:array := a;
181   for var i:Int[1,N-1+1] := 1; i ⩽ N-1; i := i+1 do
182     invariant inv_outer(a,b,i);
183     decreases termin_outer(i);
184   {
185     var x:nat := b[i];
186     for var j:Int[-1,N-1-1] := i-1; j ⩾ 0 ∧ b[j] > x; j := j-1 do
187       invariant inv_inner(a,b,j,i,x);
188       decreases termin_inner(j);
189     {
190       b[j+1] := b[j];
191       b[j] := x;
192     }
193   }
194   return b;
195 }
```

This algorithm is annotated with the help of the following predicates:

```
41 //pre-Condition
42 //  (*) trivial
43 pred pre(a:array) ⇔ true;
44
45 //post-Condition of the outer loop
46 //  (*) resulting array is sorted
47 //  (*) a is a permutation of the input array
48 pred post(a:array,result:array)
49 ⇔ isSorted(result,0,N-1)
50   ∧ isPermutationOf(a,result);
51
52 //invariant of the outer loop
53 //  (*) elements to the left of i are sorted
54 //  (*) elements right of i are unchanced
55 //  (*) b is a permutation of a
56 pred inv_outer(a:array,b:array,i:Int[1,N-1+1])
57 ⇔ isSorted(b,0,i-1)
58   ∧ (i<N-1 ⇒ isPartEqual(a,b,i+1,N-1))
59   ∧ isPermutationOf(a,b);
60
61 //termination term of the outer loop
62 //  it terminates after at most N-1 iterations
63 fun termin_outer(i:Int[1,N-1+1]):index = N-i;
64
```

```
65  //loop condition of the outer loop
66  pred loop_cond_outer(i:Int[1,N-1+1]) ⇔ i ⩽ N-1;
67
68  //post-Condition of the inner loop
69  //  (*) elements left of i are sorted
70  //  (*) elements right of i are unchanced
71  //  (*) b is a permutation of a
72  pred post_inner(a:array,b:array,i:Int[1,N-1+1])
73  ⇔ isSorted(b,0,i)
74    ∧ (i+1 < N-1 ⇒ isPartEqual(a,b,i+1,N-1))
75    ∧ isPermutationOf(a,b);
76
77  //invariant of the inner loop
78  //  (*) elements left of j+1 are sorted
79  //  (*) elements right of j+1 are sorted
80  //  (*) j+1 is smaller as its right neighbour
81  //  (*) the array without j+1 is sorted.
82  //  (*) elements right of i are unchanced
83  //  (*) b is a permutation of a
84  pred inv_inner(a:array,b:array,j:Int[-1,N-2],i:Int[1,N-1],
85  x:nat)
86  ⇔ j ⩽ i-1
87    ∧ x = b[j+1]
88    ∧ (j ⩾ 0 ⇒ isSorted(b,0,j))
89    ∧ (j+2 < i ⇒ isSorted(b,j+2,i))
90    ∧ (j+2 ⩽ i ⇒ x < b[j+2])
91    ∧ (j+2 ⩽ i ∧ j ⩾ 0 ⇒ b[j] ⩽ b[j+2])
92    ∧ (i+1 < N-1 ⇒ isPartEqual(a,b,i+1,N-1))
93    ∧ isPermutationOf(a,b);
94
95  //termination term of the inner loop
96  //  it terminates after at most N-1 iterations
97  fun termin_inner(j:Int[-1,N-1-1]):index = j+1;
98
99  //loop condition of the inner loop
100 pred loop_cond_inner(j:Int[-1,N-1-1],b:array,x:nat)
101 ⇔ j ⩾ 0 ∧ b[j] > x;
```

The main specifications are the pre- and post-condition of the outer loop. For the inner loop we need to specify its own pre- and post-conditions. We will need to derive separate verification conditions for the inner and the outer loop.

To specify the invariant of the outer loop we can use the specification of the post-condition and state that the array b is sorted up to $i-1$. This gives us the additional condition that all other elements should remain the same. We denote this fact as the predicate $\mathtt{isPartEqual}$.

```
33 //isPartEqual(a:array,b:array,m:index,n:index)
34 //  states if a and b from m to n are equal
35 pred isPartEqual(a:array,b:array,m:index,n:index)
36 ⇔ (∀i:index. (m ⩽ i ∧ i ⩽ n) ⇒ a[i] = b[i]);
```

The pre-condition of the inner loop is the invariant of the outer loop. The post-condition of the inner loop is the invariant of the next iteration (the invariant for $i+1$).

For the invariant of the inner loop we include all statements from the invariant of the outer loop. Instead of saying the interval $\{0,...,i\}$ is sorted, we now say that $\{0,...,i\}\backslash\{j+1\}$ is sorted. The other statements do not change though.

The termination term of both loops are straightforward: The outer loop needs exactly N iterations. The inner loop needs at most i iterations.

4.1.2 *Implementation for Recursive Lists*

For recursive lists we can use the following implementation:

```
255 //insert(x:nat,a:list):list
256 //  takes a element and inserts it into
257 //  a sorted list, so that the resulting
258 //  list is again sorted.
259 fun insert(x:nat,a:list):list
260   requires preInsert(x,a);
261   ensures postInsert(x,a,result);
262   decreases listLength(a);
263 = match a with
264   {
265     nil -> list!cons(x,a);
266     cons(elem:nat,rem:list) ->
267       if elem ⩾ x then
268         list!cons(x,a)
269       else
270         list!cons(elem,insert(x,rem));
271   };
272
273 //insertionSort(a:list):list
274 //  sorts a list
275 fun insertionSort(a:list):list
276   requires pre(a);
277   ensures post(a,result);
278   decreases termin(a);
279 = match a with
280   {
281     nil -> list!nil;
```

```
282     cons(elem:nat,rem:list) ->
283       insert(elem,insertionSort(rem));
284   };
```

This algorithm is specified/annotated with the help of the following predicates:

```
113 //pre-Condition for insertionSort()
114 //  (*) trivial
115 pred pre(a:list) ⇔ true;
116
117 //post-Condition for insertionSort()
118 //  (*) result is sorted
119 //  (*) result is a permutation of a
120 pred post(a:list,result:list)
121 ⇔ listLength(a) = listLength(result)
122   ∧ isSorted(result)
123   ∧ isPermutationOf(toArray(a),toArray(result));
124
125 //termination term for insertionSort()
126 //  each recursion the array-size gets smaller
127 fun termin(a:list):ℕ[N] = listLength(a);
128
129 //pre-Condition for insert()
130 //  (*) a is sorted
131 pred preInsert(x:nat,a:list)
132 ⇔ listLength(a)<N
133   ∧ isSorted(a);
134
135 //post-Condition for insert()
136 //  (*) result is sorted
137 //  (*) result is a permutation of a
138 pred postInsert(x:nat,a:list,result:list)
139 ⇔ listLength(a)+1 = listLength(result)
140   ∧ isSorted(result)
141   ∧ isPermutationOf(toArray(list!cons(x,a)),toArray(result));
142
143 //termination term for insert()
144 //  each recursion the array-size gets smaller
145 fun terminInsert(x:nat,a:list):ℕ[N] = listLength(a);
```

Similar to the imperative variant, we have two functions that need to be validated: *insertionSort* and *insert*. The pre- and post-condition are the same as for the array-variant. In order to implement the *isPermutationOf* predicate, we can only use an array and not a list. Therefore, we will formulate a *toArray* function that takes a list of any length and returns an equivalent array with a fixed length. In order to be sure that this function

is well-defined we also introduce the counterpart toList that transforms an array and a length into a list. We use the RISCAL system and a new theorem to validate the identity of the two functions:

```
33 //toArray(a:list):array
34 //  returns an array representation of a list
35 //  empty slots get filled with 0.
36 proc toArray(a:list):array
37 {
38   var out:array := Array[N,nat](0);
39   var b:list := a;
40   for var i:ℤ[0,N-1+1] := 0; i < N; i := i+1 do
41   {
42     match b with
43     {
44       nil -> ;
45       cons(elem:nat,rem:list) ->
46       {
47         out[i] := elem;
48         b := rem;
49       }
50     }
51   }
52   return out;
53 }
54
55 //toList(a:array,n:ℕ[N]):list
56 //  returns a list created by the last n elements of a
57 proc toList(a:array,n:ℕ[N]):list
58 {
59   var out:list = list!nil;
60   for var i:ℤ[-1,N-1] := n-1; i ⩾ 0; i := i-1 do
61   {
62     out := list!cons(a[i],out);
63   }
64   return out;
65 }
```

```
102 theorem ListsAndArraysAreIsomorph()
103 ⇔ (∀a:list. a = toList(toArray(a),listLength(a)))
104   ∧ (∀a:array.a = toArray(toList(a,N)))
105   ∧ (
106     ∀a:list.isSorted(a)
107     ⇔ isSortedArray(toArray(a),listLength(a))
108   );
```

We lose the length of the array when using the $\mathtt{toArray}$ function, therefore the predicate $\mathtt{isPermutationOf}$ does no ensure that the input list and the resulting list have the same length; to solve this problem, we add the condition $\mathtt{ListLength}(a) = \mathtt{ListLength}(b)$. For the pre-condition of the insert-function we require the array to be sorted and the length to be smaller than max-length. The post-condition is again the same as the invariant of $i + 1$ for the array-variant.

4.2 STABILITY OF SORTING ALGORITHMS

Additionally to the typical correctness conditions, for sorting algorithms we can also require that the order of elements with the same sorting key is preserved. An algorithm with this property is called "stable".

In order to check this property we define the elements of an array as the tuple of a key and an identifier.

```
1  type array = Array[N,Tuple[nat,nat]];
2
3  //pre-Condition
4  //  (*) elements are unique
5  pred pre(a:array)
6  ⇔ ∀i:index.∀j:index.
7    a[i] = a[j] ⇒ i = j;
```

The uniqueness of the elements will be required in the pre-condition.

An algorithm can be checked for stability by checking if all results of the algorithm have the following property.

```
1  //isStable(a:array,result:array)
2  //  checks if a algorithm is stable.
3  pred isStable(a:array,result:array)
4  ⇔ ∀i1:index.∀j1:index.i1 < j1 ∧ a[i1].1 = a[j1].1
5    ⇒ ∀i2:index.∀j2:index.(
6      a[i1] = result[i2]
7      ∧ a[j1] = result[j2]
8      ⇒ i2 < j2
9    );
```

Once we know that the algorithm can ensure stability, we include it in the post-conditions and derive verification conditions.

4.2.1 *Stable Check of InsertionSort*

We use a modified version of the variant for arrays.

```
195 //proc insertionSort(a:array): array
196 //  sorts an array
197 proc insertionSort(a:array): array
198   requires pre(a);
199   ensures post(a,result);
200 {
201   var b:array := a;
202   for var i:Int[1,N-1+1] := 1; i ⩽ N-1; i := i+1 do
203     invariant inv_outer(a,b,i);
204     decreases termin_outer(i);
205   {
206     var x:Tuple[nat,nat] := b[i];
207     for var j:Int[-1,N-1-1] := i-1; j ⩾ 0 ∧ b[j].1 > x.1; j := j-1 do
208       invariant inv_inner(a,b,j,i,x);
209       decreases termin_inner(j);
210     {
211       b[j+1] := b[j];
212       b[j] := x;
213     }
214   }
215   return b;
216 }
```

As expected, the original verification conditions are still valid. We can then check that the insertion sort is stable, by testing if the $isStable$ predicate holds for the results of the implementation. This in fact the case. Therefore, we can include the $isStable$ predicate to the post-condition of the outer loop. The resulting specifications are the following:

```
51 //pre-Condition
52 //  (*) elements are unique
53 pred pre(a:array)
54 ⇔ ∀i:index.∀j:index.
55   a[i] = a[j] ⇒ i = j;
56
57 //post-Condition of the inner loop
58 //  (*) resulting array is sorted
59 //  (*) a is a permutation of the input array
60 //  (*) result is stable
61 pred post(a:array,result:array)
62 ⇔ isSorted(result,0,N-1)
63   ∧ isPermutationOf(a,result)
64   ∧ isStable(a,result);
65
66 //invariant of the inner loop
67 //  (*) elements to the left of i are sorted
68 //  (*) elements right of i are unchanced
```

```
69  //  (*) b is a permutation of a
70  //  (*) b is stable
71  pred inv_outer(a:array,b:array,i:Int[1,N-1+1])
72  ⇔ isSorted(b,0,i-1)
73    ∧ (i < N-1 ⇒ isPartEqual(a,b,i+1,N-1))
74    ∧ isPermutationOf(a,b)
75    ∧ isStable(a,b);
76
77  //termination term of the outer loop
78  //  it terminates after at most N-1 iterations
79  fun termin_outer(i:Int[1,N-1+1]):index = N-i;
80
81  //loop condition of the outer loop
82  pred loop_cond_outer(i:Int[1,N-1+1])
83  ⇔ i ⩽ N-1;
84
85  //post-Condition of the inner loop
86  //  (*) elements left of i are sorted
87  //  (*) elements right of i are unchanced
88  //  (*) b is a permutation of a
89  pred post_inner(a:array,b:array,i:Int[1,N-1+1])
90  ⇔ isSorted(b,0,i)
91    ∧ (i+1 < N-1 ⇒ isPartEqual(a,b,i+1,N-1))
92    ∧ isPermutationOf(a,b)
93    ∧ isStable(a,b);
94
95  //invariant of the inner loop
96  //  (*) elements left of j+1 are sorted
97  //  (*) elements right of j+1 are sorted
98  //  (*) j+1 is smaller as its right neighbour
99  //  (*) the array without j+1 is sorted.
100 //  (*) elements right of i are unchanced
101 //  (*) b is a permutation of a
102 pred inv_inner(a:array,b:array,j:Int[-1,N-2],i:Int[1,N-1],
103   x:Tuple[nat,index])
104 ⇔ j ⩽ i-1
105   ∧ x = b[j+1]
106   ∧ (j ⩾ 0 ⇒ isSorted(b,0,j))
107   ∧ (j+2 < i ⇒ isSorted(b,j+2,i))
108   ∧ (j+2 ⩽ i ⇒ x.1 < b[j+2].1)
109   ∧ (j+2 ⩽ i ∧ j ⩾ 0 ⇒ b[j].1 ⩽ b[j+2].1)
110   ∧ (i+1<N-1 ⇒ isPartEqual(a,b,i+1,N-1))
111   ∧ isPermutationOf(a,b);
112
113 //termination term of the inner loop
114 //  it terminates after at most N-1 iterations
115 fun termin_inner(j:Int[-1,N-1-1]):index = j+1;
```

```
116
117 //loop condition of the inner loop
118 pred loop_cond_inner(j:Int[-1,N-1-1],b:array,
119 x:Tuple[nat,nat])
120 ⇔ j ⩾ 0 ∧ b[j].1 > x.1;
```

These specifications are mostly the same as those of the array-variant. We have only added the *isStable* check to the invariant and the post-condition of the inner-loop.

4.3 OUTCOME OF THE CHECKS

For the array- and list-variants we have checked that the post-condition is unique. Next we have validated that the specifications are not too strong by checking all input using RISCAL. Finally, we have derived and validated the verification conditions. They can now be used to fully prove the correctness with respect to the specifications.

For the stable check we have verified that the pre-condition is not trivial and that the post-condition is unique. Again we have checked whether the specifications hold for all inputs and have derived verification conditions that can further be used to prove the correctness.

4.4 CONCLUSION

The InsertionSort is the first algorithm that consists of two nested loops. For imperative algorithms we used two separate verification conditions. The invariant of the outer loop acted as the pre-condition of the inner loop we have therefore shown the relation between the two loops. For functional algorithms we use functions instead of loops. The relation between the functions is stated in the function specification of the outer function.

5 MERGESORT

Another stable sorting algorithm is called MergeSort. It utilizes the principle of dividing and conquering as follows:

1. Divide the array into sub-arrays of half size and sort each sub-array recursively.
2. Merge the two sorted sub-arrays into a totally sorted array.

Before we validate this algorithm we will first look at the specification and validation of the merge algorithm.

5.1 MERGE ALGORITHM

The merge algorithm serves the following specification: Given two sorted arrays, return the sorted array that contains all elements of both inputs (and only these elements). In order for the algorithm to be useful when using it in the MergeSort algorithm, we will need to implement it for arrays with arbitrary length.

Merge Algorithm
Input : Two arrays a, b and two natural numbers $size_a, size_b$
Output : An array c of size $size_a + size_b$ so that its the sorted equivalent to the array containing all elements of a and b.

let $key_a := 0, key_b := 0$
Loop from 0 to $size_a + size_b - 1$ with index i
    if $a[key_a] > b[key_b]$ then
        $c[i] := b[key_b]$
        $key_b := key_b + 1$
    else
        $c[i] := b[key_a]$
        $key_a := key_a + 1$

5.1.1 *Implementation for Arrays*

RISCAL only supports arrays with a fixed length, therefore we describe dynamic arrays by an array of fixed length and a natural number less than equal this length that denotes the number of elements considered in that array; all subsequent elements are zero.

For arrays we implement the algorithm with a for-loop:

```
160 proc merge(a:array,b:array,size_a:ℤ[1,N],size_b:ℤ[1,N]):array
161   requires pre(a,b,size_a,size_b);
162   ensures post(a,b,size_a,size_b,result);
163 {
164   var key_a:ℕ[N-1+1] := 0;
165   var key_b:ℕ[N-1+1] := 0;
166   var out:array := Array[N,nat](0);
167   for var i:ℤ[0,N-1+1] := 0; i<size_a+size_b; i := i+1 do
168     decreases termin(i,size_a,size_b);
169     invariant inv(a,b,size_a,size_b,key_a,key_b,i,out);
170   {
171     if
172       key_a < size_a
173       ⇒ (key_b < size_b ∧ a[key_a] > b[key_b])
174     then
175     {
176       out[i] := b[key_b];
177       key_b := key_b+1;
178     }
179     else
180     {
181       out[i] := a[key_a];
182       key_a := key_a+1;
183     }
184   }
185   return out;
186 }
```

Its specification uses the following predicates:

```
52 //pre-Condition
53 //  (*) a and b are sorted
54 pred pre(a:array,b:array,size_a:ℤ[1,N],size_b:ℤ[1,N])
55 ⇔ size_a+size_b ⩽ N
56   ∧ hasSize(a,size_a)
57   ∧ hasSize(b,size_b)
58   ∧ isSorted(a,0,size_a-1)
59   ∧ isSorted(b,0,size_b-1);
60
61 //post-Condition
62 //  (*) result is a permutation of the joined array a+b
63 //  (*) result is sorted
64 //  (*) result is unique -> unused elements are 0
65 pred post(a:array,b:array,size_a:ℤ[1,N],size_b:ℤ[1,N],
66   result:array)
67 ⇔ (
```

```
68      ∃c:array.
69      isJoinedArray(a,b,size_a,size_b,c)
70      ∧ isPermutationOf(result,c)
71    )
72    ∧ isSorted(result,0,size_a+size_b-1)
73    ∧ hasSize(result,size_a+size_b);
74
75  //invariant
76  //  (*)i is the sum of key_a and key_b
77  //  (*)out is the perumtation of the joined parts a
78  //     (0 to key_a)+b(0 to key_b)
79  //  (*)out is (part) sorted
80  //  (*)elements that are not yet merged are not smaller then
81  //     elements that are already merged
82  pred inv(a:array,b:array,size_a:ℤ[1,N],size_b:ℤ[1,N],
83    key_a:ℕ[N-1+1],key_b:ℕ[N-1+1],i:ℕ[N-1+1],out:array)
84  ⇔ i ⩽ size_a+size_b ∧ key_a ⩽ size_a ∧ key_b ⩽ size_b
85    ∧ i = key_a+key_b
86    ∧ (
87        ∃c:array.
88        isJoinedArray(a,b,key_a,key_b,c)
89        ∧ isPermutationOf(out,c)
90    )
91    ∧ (i > 0 => isSorted(out,0,i-1))
92    ∧ ∀j:index.if j ⩾ i then
93        out[j] = 0
94      else
95      (
96        (∀k:ℕ[N-1].k < size_a ∧ k ⩾ key_a ⇒ out[j] ⩽ a[k])
97        ∧ (∀k:ℕ[N-1].k < size_b ∧ k ⩾ key_b ⇒ out[j] ⩽ b[k])
98      );
99
100 //loop condition
101 pred loop_cond(i:ℤ[0,N-1+1],size_a:ℤ[1,N],size_b:ℤ[1,N])
102 ⇔ i < size_a+size_b;
103
104 //termination term
105 // lenth of the array
106 fun termin(i:ℕ[N-1+1],size_a:ℤ[1,N],size_b:ℤ[1,N]):ℕ[N+1]
107   = size_a+size_b-i;
```

The pre-condition states that both input arrays a, b are sorted. We already know that the result will be a combined array, therefore it requires $size_a + size_b \leqslant N$. To avoid additional computational complexity we also state that elements outside of the dynamic array are equal to 0 by using the $hasSize$ predicate:

```
47 //hasSize(a:array,size:ℕ[N])
48 //  state that all elements outside of the dynamic array are 0
49 pred hasSize(a:array,size:ℕ[N])
50 ⇔ ∀i:index.i ⩾ size ⇒ a[i]=0;
```

The post condition is quite straightforward: We use the following predicate to describe a joined array:

```
28 //isJoinedArray(a:array,b:array,size_a:ℕ[N],size_b:ℕ[N]
29   ,out:array)
30 //  states if out is the array obtains by joining a and b.
31 pred isJoinedArray(a:array,b:array,size_a:ℕ[N],size_b:ℕ[N],
32   out:array)
33 ⇔ ∀i:index.if i < size_a then
34     out[i] = a[i]
35   else if i-size_a < size_b then
36     out[i] = b[i-size_a]
37   else
38     out[i] = 0;
```

The result must be a sorted permutation of an array c that satisfies $isJoinedArray(a, b, size_a, size_b, c)$.

For the invariant we need to specify the relation between i, key_a and key_b as $i = key_a + key_b$. We can model our resulting array as a dynamic array, this gives us the second predicate: out is a permutation of the array c that satisfies $isJoinedArray(a, b, key_a, key_b, c)$. Finally, we can specify that out is sorted and that those elements that are not yet elements of out are greater or equal to the elements in out.

The termination term is $size_a + size_b - i$, the number of elements that have not yet been merged.

5.1.2 *Implementation for Recursive Lists*

For recursive lists we replace the loop by the following recursive function (which does not use the variables $left, right$):

```
187 fun merge(a:list,b:list):list
188   requires pre(a,b);
189   ensures post(a,b,result);
190   decreases termin(a,b);
191 = match a with
192   {
193     nil -> b;
194     cons(elem_a:nat,rem_a:list) ->
195       match b with
196       {
```

```
197         nil -> a;
198         cons(elem_b:nat,rem_b:list) ->
199           if elem_a < elem_b then
200             list!cons(elem_a,merge(rem_a,b))
201           else
202             list!cons(elem_b,merge(a,rem_b));
203       };
204   };
```

This algorithm has the following specification:

```
95  //pre-Condition
96  //  (*) a and b are sorted
97  //  (*) the length of the combined list a+b is not bigger N
98  pred pre(a:list,b:list)
99  ⇔ isSorted(a)
100   ∧ isSorted(b)
101   ∧ listLength(a)+listLength(b) ⩽ N;
102
103 //post-Condition
104 //  (*) result is a permutation of the list (a,b)
105 //  (*) result is sorted
106 pred post(a:list,b:list,result:list)
107 ⇔ listLength(result) = listLength(a) + listLength(b)
108   ∧ isPermutationOf(toArray(result),toArray(append(a,b)))
109   ∧ isSorted(result);
110
111 //termination term
112 // lenth of the list
113 fun termin(a:list,b:list):ℕ[N+1]
114 = listLength(a) + listLength(b);
```

The pre-condition is in its essence the same as for arrays: $size_a, size_b$ are now $listLength(a), listLength(b)$. The post-condition is also the same: we specify that the resulting length should be the sum of the lengths of original lists. Instead of $isJoinedArray$ we use the function $append$ that concatenates one lists with another.

```
55  //append(a:list,b:list)
56  //  appends b to a
57  fun append(a:list,b:list):list
58    decreases listLength(a);
59  = match a with
60    {
61      nil -> b;
62      cons(elem:nat,rem:list) ->
```

```
63        list!cons(elem,append(rem,b));
64    };
```

This function is typically implemented in languages that have lists as a basic type. We know the function will iterate over every element, therefore the termination term is $\mathrm{listLength}(a) + \mathrm{listLength}(b)$.

5.2 MERGESORT

The MergeSort can be implemented either by starting with the full array (recursive "Top-Down") or with dynamic arrays of length 1 (iterative "Bottom-Up").

5.2.1 *Implementation for Arrays*

The two implementations are the following:

```
189 //  implementation of the Top-Down variant
190 fun mergeSortRec(a:array,begin:index,end:ℕ[N-1+1]):array
191   requires pre(a,begin,end);
192   ensures post(a,begin,end,result);
193   decreases termin(a,begin,end);
194 = if end = begin then
195     Array[N,nat](0)
196   else if (end-begin = 1) then
197     mergeSortRec(a,begin,begin) with [0] = a[begin]
198   else
199     merge(
200       mergeSortRec(a,begin,(begin+end)/2),
201       mergeSortRec(a,(begin+end)/2,end),
202       (begin+end)/2-begin,
203       end-(begin+end)/2
204     );
205 fun mergeSort(a:array):array
206 = mergeSortRec(a,0,N);
```

We can now specify the algorithm.

```
87 //pre-Condition
88 //  (*) trivial
89 pred pre(a:array,begin:index,end:ℕ[N-1+1])
90 ⇔ begin ⩽ end;
91
92 //post-Condition
93 //  (*) resulting array is sorted
94 //  (*) result is a permutation of a
```

```
 95 pred post(a:array,begin:index,end:ℕ[N-1+1],result:array)
 96 ⇔ hasSize(result,end-begin)
 97   ∧ (
 98     end > begin ⇒
 99     (
100       isSorted(result,0,end-begin-1)
101       ∧ isPartlyPermutationOf(a,result,begin,0,end-begin)
102     )
103   );
104
105 //termination term of the outer loop
106 //  the difference between begin and end gets reduced each
107 //  iteration
108 fun termin(a:array,begin:index,end:ℕ[N-1+1]):ℕ[N-1+1]
109 = end-begin;
```

Here we describe a sub-array as a dynamic array with a *begin* and an *end* index. If begin is zero and end is the length of the array, this indicates that the array has full size. This way all sub-arrays can be stored in one array (so called "in-place" sorting). For the post-conditions we have the general specification for a sorting algorithm. The termination term is the length of the array.

Additionally we have implemented the "bottom-up" variant.

```
208 //mergeSortBottomUp(a:array):array
209 //  implementation of the Bottom-Up variant
210 proc mergeSortBottomUp(a:array):array
211 {
212   var arrays:Array[N,array] := Array[N,array](Array[N,nat](0));
213   for var i:ℕ[N-1+1] := 0; i < N;i := i+1 do
214    arrays[i][0] := a[i];
215   //length of arrays in var arrays
216   var sizes:Array[N,ℕ[N]] := Array[N,ℕ[N]](1);
217   var counter:ℕ[N] := N; //length of Arrays
218
219   while counter > 1 do
220   {
221     val old_counter := counter;
222     for var i:ℕ[N-1+1] := 0; i < old_counter/2; i := i+1 do
223     {
224       arrays[i] := merge(
225         arrays[i*2],arrays[i*2+1],sizes[i*2],sizes[i*2+1]
226       );
227       sizes[i] := sizes[i*2]+sizes[i*2+1];
228       counter := counter-1;
229     }
230     if old_counter%2 = 1 then
```

```
231     {
232       arrays[counter-1] := arrays[old_counter-1];
233       sizes[counter-1] := sizes[old_counter-1];
234     }
235   }
236   return arrays[0];
237 }
```

This variant uses a counter in order to check if there are uneven numbers of sub-arrays: If so the last sub-array gets added. This makes it very inefficient. We will not validate the algorithm but only show that its equivalent to the previous implementation:

```
239 //mergeSortEquivalence()
240 //  BottomUp and TopDown are equivalent
241 theorem mergeSortEquivalence()
242 ⇔ ∀a:array.mergeSortBottomUp(a) = mergeSort(a);
```

5.2.2 *Implementation for Recursive Lists*

The implementation for recursive list is as follows:

```
240 fun mergeSort(a:list):list
241   requires pre(a);
242   ensures post(a,result);
243   decreases termin(a);
244 = match a with
245   {
246     nil -> a;
247     cons(elem:nat,rem:list) ->
248       match rem with
249       {
250         nil -> a;
251         cons(elem2:nat,rem2:list) ->
252           merge(
253             mergeSort(split(a).1),
254             mergeSort(split(a).2)
255           );
256       };
257   };
```

For recursive lists we have the following specification:

```
134 //pre-Condition
135 //  (*) true
136 pred pre(a:list) ⇔ true;
137
```

```
138 //post-Condition
139 //  (*) result is sorted
140 //  (*) result is a permutation of a
141 pred post(a:list,result:list)
142 ⇔ listLength(a) = listLength(result)
143   ∧ isSorted(result)
144   ∧ isPermutationOf(toArray(a),toArray(result));
145
146 //termination term for the merge sort algorithm
147 // lenth of the list
148 fun termin(a:list):ℕ[N+1] = listLength(a);
```

The conditions remain the same.

5.3 OUTCOME OF THE CHECKS

First we have used RISCAL to check that the specification of the merge algorithms are not too strong and that the "result" is uniquely defined. Then we have formulated and checked the verification conditions which can be subsequently used to prove the correctness of the algorithm.

For the Merge algorithm, the pre-condition for the array-version is not trivial. Storing all sub-arrays in one array requires that $begin$ and end actually are at the beginning and the end of the sub-array. The variant for recursive lists is not an "in-place" sorting algorithm, as this would not be feasible for recursive lists. This provides us in return a trivial pre-condition.

Next we have checked that the sorting algorithms satisfy their specifications and return unique results. We have also formulated and checked the verification conditions that imply the correctness of the algorithm. All checks where done for lists of size $N = 4$ or smaller. Checking the correctness of the specification and the algorithm were done with an upper bound of 2 for all elements. All checked needed less than a minute.

Finally we have also checked that our implementations are stable. We have added the predicate of unique values in the pre-condition and the predicate of stability in the post-condition. The modified specification was then validated the same way as before and the new verification conditions have been derived.

5.4 CONCLUSION

The use of dynamic arrays has proven to be essential, not only for the implementation in RISCAL but also for the in-place variant (the top-down variant for arrays) of the MergeSort. The specifications needed to also include predicates about the dynamic arrays. This shows that trivial seaming details, like $begin \leqslant end$, are essential to the validity of a specification.

6 QUICKSORT

The QuickSort algorithm is, as the name suggests, faster than the Merge Sort algorithm. Both have a an symbolic time complexity of $\mathcal{O}(n \log n)$, but the QuickSort algorithm beats the MergeSort by the leading constant [SW16]. It also utilizes the "divide and conquer"-principle but optimizes the dividing process instead of the conquering/merging part.

QuickSort Algorithm
Input : an arbitrary array.
Output : a sorted permutation of the input.

- Pick a pivot element (any element in the array e.g. the last one)
- Partition the array into two sub-arrays: One for all elements smaller or equal and one for all elements bigger than the pivot element.
- Apply the algorithm to both sub-arrays and join them together.

Same as for the MergeSort algorithm we will first validate the auxiliary algorithm (partitioning).

6.1 PARTITIONING ALGORITHM

The variant for arrays will be again an in-place implementation where both sub-arrays are part of the same array, thus we can later skip the joining.

Partitioning Algorithm
Input : An array a with indices $begin$ and end
Output : An array b which is a permutation of a and an index $pivot$, so that all elements of b with index in $[begin, pivot]$ are smaller or equal $b[pivot]$, elements in $(pivot, end_a)$ are bigger then $b[pivot]$ and $a[end-1] = b[pivot]$; all elements of b with indices outside of $[begin, end]$ are the same as in a.

let $b := a$
let $pivot := end - 1$
Loop while $i < pivot$ with $i := begin$
    if $b[i] > b[pivot]$ then
        rotate: $b[i] \to b[pivot] \to b[pivot-1] \to b[i]$
        pivot := pivot -1

```
        else
            i := i+1
    return ⟨b, pivot⟩
```

6.1.1 *Implementation for Arrays*

For arrays, we can implement the algorithm as follows:

```
147 proc partitioning(a:array,begin:index,end:ℕ[N-1+1]):ret
148   requires pre(a,begin,end);
149   ensures post(a,begin,end,result);
150 {
151   var out:ret := ⟨array:a,pivot:end-1⟩;
152   var i:index := begin;
153   while i < out.pivot do
154     decreases termin(i,out.pivot);
155     invariant inv(a,begin,end,out,i);
156   {
157     if out.array[i] > a[end-1] then
158     {
159       out.array[out.pivot] := out.array[i];
160       out.pivot := out.pivot-1;
161       out.array[i] := out.array[out.pivot];
162       out.array[out.pivot] := a[end-1];
163     }
164     else
165       i := i+1;
166   }
167   return out;
168 }
```

This implementation satisfies the following specifications:

```
32 //pre-Condition
33 //  (*) trivial
34 pred pre(a:array,begin:index,end:ℕ[N-1+1])
35 ⇔ begin < end;
36
37 //post-Condition
38 //  (*) result.1 is a permutation of a
39 //  (*) outside [begin,end) result and a are the same
40 //  (*) in the interval, smaller and equal elements are left
41 //      of result.2,higher elements are right of result.2
42 pred post(a:array,begin:index,end:ℕ[N-1+1],result:ret)
43 ⇔ isPermutationOf(a,result.array)
44   ∧ result.array[result.pivot] = a[end-1]
45   ∧ begin ⩽ result.pivot ∧ result.pivot < end
```

```
46    ∧ ∀i:index.if i < begin then
47        result.array[i] = a[i]
48      else if i < result.pivot then
49        result.array[i] ⩽ result.array[result.pivot]
50      else if i = result.pivot then
51        true
52      else if i < end then
53        result.array[i] > result.array[result.pivot]
54      else
55        result.array[i] = a[i];
56
57  //invariant
58  //  (*) out.1 is a permutation of a
59  //  (*) elements between out.2 and end are higher then out.2
60  //  (*) elements between begin and k are smaller then out.2
61  //  (*) all other elements stay the same.
62  pred inv(a:array,begin:index,end:ℕ[N-1+1],out:ret,i:index)
63  ⇔ i ⩽ out.pivot
64    ∧ out.pivot < end
65    ∧ begin ⩽ i
66    ∧ isPermutationOf(a,out.array)
67    ∧ (
68      ∀k:index.if k > end-1 then
69        out.array[k] = a[k]
70      else if k > out.pivot then
71        out.array[k] > out.array[out.pivot]
72      else if k = out.pivot then
73        out.array[k]=a[end-1]
74      else if k > i then
75        out.array[k] = a[k]
76      else if k = i then
77        true
78      else if k ⩾ begin then
79        out.array[k] ⩽ out.array[out.pivot]
80      else
81        out.array[k] = a[k]
82    );
83
84  //loop condition
85  pred loop_cond(i:index,out:ret)
86  ⇔ i < out.pivot;
87
88  //termination term of the outer loop
89  //  the difference between i and j:=out.2,the pivot element
90  fun termin(i:index,j:index):ℕ[N-1+1] = j-i+1;
```

The post-conditions includes conditions that we have already discussed: Permutation, elements left of $pivot$ are smaller or equal, elements to the right are bigger. We also need so specify the relation between $pivot$ and $begin, end$ namely: $begin \leqslant pivot \leqslant pivot < end$ as well as that all elements outside $[begin, end)$ have not changed.

The invariant has the same conditions but also states that elements in $(i, pivot)$ have not changed. The termination term is the difference between i and the index of the pivot element. From the implementation we see that either i increases or $pivot$ decreases. At the point where the loop terminates, i equals $pivot$.

6.1.2 *Implementation for Recursive Lists*

For recursive lists the value for which we partition the lists must be given. Typically, this will be the first element of the original list. Instead of rotating the elements as we have done in the in-place variant, we will now use two lists and append the element to the corresponding lists. This will give use a trivial pre-condition as the in-place variant requires the relation between $begin$ and end. The resulting algorithm is the following:

```
175 fun partitioning(a:list, pivot:nat):ret
176   requires pre(a,pivot);
177   ensures post(a,pivot,result);
178   decreases termin(a,pivot);
179 = match a with
180   {
181     nil -> ⟨a,a⟩;
182     cons(elem:nat,rem:list) ->
183       let
184         lists = partitioning(rem,pivot)
185       in
186       if elem ⩽ pivot then
187         ⟨lists.1,list!cons(elem,lists.2)⟩
188       else
189         ⟨list!cons(elem,lists.1),lists.2⟩;
190   };
```

The corresponding specifications are as follows:

```
103 //pre-Condition
104 //  (*) trivial
105 pred pre(a:list, pivot:nat)
106 ⇔ true;
107
108 //post-Condition
109 //  (*) a is a permutation of result.1+2
```

```
110 //  (*) elements in result.1 are smaller or equal then the
111 //      pivot element
112 //  (*) elements in result.2 are bigger then the pivot
113 //      element
114 pred post(a:list, pivot:nat, result:ret)
115 ⇔ listLength(a) = listLength(result.1) + listLength(result.2)
116   ∧ isPermutationOf(
117       toArray(a),toArray(append(result.1,result.2))
118   )
119   ∧ elementsAreSmallerEqual(result.2,pivot)
120   ∧ elementsAreLarger(result.1,pivot);
121
122 //termination term
123 //  each recursion the array-size gets smaller
124 fun termin(a:list, pivot:nat):ℕ[N] = listLength(a);
```

In the specification i used a function called *append*:

```
34 //append(a:list,b:list)
35 //  appends b to a
36 fun append(a:list,b:list):list
37   decreases listLength(a);
38 = match a with
39   {
40     nil -> b;
41     cons(elem:nat,rem:list) -> list!cons(elem,append(rem,b));
42   };
```

This predicate is typically already implemented in any language that has lists as a basic type.

6.1.3 *Implementation for Linked Lists*

The implementation of the linked list variant was done after most of the other implementations in this chapter. At that time the version 2 of RISCAL was just released. We can now write the implementation and the specification in one function and have the verification condition automatically generated.

```
93 fun partitioning(rem:pointer,end:pointer,pivot:pointer,
94   s:store):ret
95   requires
96     leadsTo(end,N,s)
97     ∧ leadsTo(rem,end,s)
98     ∧ leadsTo(pivot,rem,s)
99     ∧ ( ∀c:pointer.c < null
100             ∧ leadsTo(pivot,c,s)
```

```
101             ∧ leadsTo(c,rem,s)
102             ∧ c ≠ pivot ∧ c ≠ rem
103             ⇒ s[c].head > s[pivot].head
104           )
105     ∧ (pivot = end ⇒ rem = end);
106   ensures
107     isStorePermutationOf(s,result.store)
108     ∧ leadsTo(pivot,end,result.store)
109     ∧ (
110       ∀c:pointer.c < null ∧ s[c] ≠ result.store[c]
111       ⇒ leadsTo(pivot,c,s) ∧ leadsTo(c,end,s)
112     )
113     ∧ isPermutationOf(
114         toArray(pivot,end,s),toArray(pivot,end,result.store)
115       )
116     ∧ listLength(pivot,end,s)
117       = listLength(pivot,end,result.store)
118     ∧ (
119       if pivot ≠ end then
120         result.pivot ≠ null
121         ∧ pivot ≠ null
122         ∧ result.store[result.pivot].head = s[pivot].head
123       else
124         result.pivot = end
125     )
126     ∧ (
127       pivot ≠ null
128       ⇒ ( ∀c:pointer.c < null
129         ∧ leadsTo(pivot,c,result.store)
130         ∧ leadsTo(result.store[c].tail,end,result.store)
131         ⇒ if leadsTo(c,result.pivot,result.store) then
132           result.store[c].head ⩽ s[pivot].head
133         else
134           result.store[c].head > s[pivot].head
135       )
136     );
137   decreases
138     listLength(rem,end,s);
139 = if rem = end then
140     ⟨store:s,pivot:pivot⟩
141   else if rem ≠ pivot ⇒ s[rem].head > s[pivot].head then
142     partitioning(s[rem].tail,end,pivot,s)
143   else
144     partitioning(
145       s[rem].tail,end,s[pivot].tail,
146       s
147         with [rem]
```

```
148           = ⟨head:s[s[pivot].tail].head,tail:s[rem].tail⟩
149         with [s[pivot].tail]
150           = ⟨head:s[pivot].head,tail:s[s[pivot].tail].tail⟩
151         with [pivot]
152           = ⟨head:s[rem].head,tail:s[pivot].tail⟩
153     );
```

This implementation is again sorting in place but its also recursive. We note that the variable i from the array-version is equivalent to $s[rem].head$. We can also note that elements left of $pivot$ will not chance anymore. Because we are using a list, our pivot element will be the first element in the list. This leads to the recursive variant as seen above.

For the pre-condition we first give the relation between $pivot, rem$ and end: $pivot$ leads to rem, which leads to end. Next we state that element between $pivot$ and rem are bigger than $s[pivot].head$.

For the post-condition we first state the store and the list at the original pointer are both permutations of the originals. This is equivalent to showing that a recursive list is a permutation of another. Same as for a recursive list we need to also show that the lengths of the lists are the same. The predicate $isStorePermutationOf$ will only check if the heads are the same:

```
72  //isStorePermutationOf(a:store,b:store)
73  //  states if a is a permutation of b
74  pred isStorePermutationOf(a:store,b:store)
75  ⇔ ∃p:Array[N,index].
76      (∀i:index,j:index. i ≠ j ⇒ p[i] ≠ p[j])
77      ∧ (
78        ∀i:index. a[i].head = b[p[i]].head
79        ∧ a[i].tail = b[i].tail
80      );
```

Finally, the post-condition, which states that the head of the pointer $pivot$ does not chance and that elements "left" of $pivot$ are smaller or equal and elements "right" are bigger.

The implemented function is a tail-recursive function, making it possible to implement the same algorithm imperatively.

```
72  //*************************************
73  //  IMPLEMENTATION
74  //
75  //  Preconditions
76  //  (*) a is a linked list and end is in a.
77  //
78  //  Postconditions
79  //  (*) resulting store is a permutation
80  //  (*) pivot is still a linked lists in the new store
```

```
81  //  (*) all chances happen in the linked list, between pivot
82  //      and end
83  //  (*) resulting pivot is a permutation
84  //  (*) the head of the pivot has not chanced (if there was
85  //      a head to begin with)
86  //
87  //  Invariant
88  //  (*) resulting store is a permutation
89  //  (*) pivot is still a linked lists in the new store
90  //  (*) all chances happen in the linked list, between pivot
91  //      and end
92  //  (*) resulting list is a permutation
93  //  (*) the head of the pivot has not chanced (if there was
94  //      a head to begin with)
95  //
96  //  Termination Term
97  //  the length of the list from pointer rem to pointer end
98  //**************************************
99  proc partitioningLoop(a:pointer,end:pointer,s:store):ret
100   requires
101     leadsTo(end,null,s)
102     ∧ leadsTo(a,end,s);
103   ensures
104     isStorePermutationOf(s,result.store)
105     ∧ leadsTo(a,end,result.store)
106     ∧ (
107       ∀c:pointer.c < null ∧ s[c] ≠ result.store[c]
108       ⇒ leadsTo(a,c,s) ∧ leadsTo(c,end,s)
109     )
110     ∧ isPermutationOf(
111         toArray(a,end,s),toArray(a,end,result.store)
112       )
113     ∧ listLength(a,end,s)
114       = listLength(a,end,result.store)
115     ∧ (
116       if a ≠ end then
117         result.pivot ≠ null
118         ∧ result.store[result.pivot].head = s[a].head
119       else
120         result.pivot = end
121     )
122     ∧ (
123       a ≠ null
124       ⇒ ( ∀c:pointer.c < null
125         ∧ leadsTo(a,c,result.store)
126         ∧ leadsTo(result.store[c].tail,end,result.store)
127         ⇒ if leadsTo(c,result.pivot,result.store) then
```

```
128           result.store[c].head ⩽ s[a].head
129         else
130           result.store[c].head > s[a].head
131       )
132     );
133 {
134   var out:ret := ⟨store:s,pivot:end⟩;
135   if a ≠ end then
136   {
137     var i:pointer := s[a].tail;
138     out.pivot := a;
139     out.store := s;
140     while i ≠ end do
141       invariant
142         isStorePermutationOf(s,out.store)
143         ∧ leadsTo(end,null,out.store)
144         ∧ leadsTo(i,end,out.store)
145         ∧ leadsTo(out.pivot,i,out.store)
146         ∧ leadsTo(a,out.pivot,out.store)
147         ∧ out.pivot ≠ end
148         ∧ (
149           ∀c:pointer.c < null ∧ s[c] ≠ out.store[c]
150           ⇒ leadsTo(a,c,s) ∧ leadsTo(c,end,s)
151         )
152         ∧ isPermutationOf(
153           toArray(a,end,s),toArray(a,end,out.store)
154         )
155         ∧ listLength(a,end,s)
156           = listLength(a,end,out.store)
157         ∧ (out.store[out.pivot].head = s[a].head
158         )
159         ∧ (out.store[out.pivot].tail ≠ end ⇒ i ≠ out.pivot)
160         ∧ (
161           a ≠ null
162           ⇒ ( ∀c:pointer.c < null
163             ∧ leadsTo(a,c,out.store)
164             ∧ leadsTo(out.store[c].tail,i,out.store)
165             ⇒ if leadsTo(c,out.pivot,out.store) then
166               out.store[c].head ⩽ s[a].head
167             else
168               out.store[c].head > s[a].head
169           )
170         );
171       decreases
172         listLength(i,end,s);
173     {
174       if
```

```
175         i ≠ out.pivot ∧ out.store[i].head
176         ⩽ out.store[out.pivot].head
177       then
178       {
179         var temp:nat := out.store[i].head;
180         out.store[i].head
181           := out.store[out.store[out.pivot].tail].head;
182         out.store[out.store[out.pivot].tail].head
183           := out.store[out.pivot].head;
184         out.store[out.pivot].head := temp;
185         out.pivot := out.store[out.pivot].tail;
186       };
187       i := out.store[i].tail;
188     };
189   };
190   return out;
191 }
```

For the invariant we first of all specify that the resulting store is a permutation of the original and that a is a list containing $pivot$, i as well as end. Next we specify that as long as the loop is running, the list starting with $pivot$ and ending with end is not empty. Next we specify that all chances in the store are found in the list a. To narrow down the allowed chances we also require the resulting list a to be a permutation of the original. Finally, we add that the value of pivot is defined as the first element of the original list a and that in the resulting list elements that have already been considered are smaller or equal to $pivot$ if they are in front of $pivot$ and larger if not.

6.2 QUICKSORT ALGORITHM

We can now use the partitioning algorithm to formulate our algorithm. The QuickSort algorithm has trivial pre-conditions if we ignore the requirements for a well formed linked list. Same as for all previous sorting algorithms the post-condition is unique.

6.2.1 *Implementation for Arrays*

For arrays we implement the "in-place" variant.

```
168 fun quickSortRec(a:array,begin:ℕ[N-1+1],end:ℤ[-1,N-1+1]):array
169   requires pre(a,begin,end);
170   ensures post(a,begin,end,result);
171   decreases termin(a,begin,end);
172 = if begin < end then
```

```
173     let
174       part = partitioning(a,begin,end),
175       b = quickSortRec(part.array,part.pivot+1,end),
176       c = quickSortRec(b,begin,part.pivot)
177     in
178     c
179   else
180     a;
181 fun quickSort(a:array):array = quickSortRec(a,0,N);
```

We can also formulate the specification of the algorithm.

```
59 //pre-Condition
60 //  (*) trivial
61 pred pre(a:array,begin:ℕ[N-1+1],end:ℤ[-1,N-1+1])
62 ⇔ true;
63
64 //post-Condition
65 //  (*) resulting array is sorted
66 //  (*) result is a permutation of a
67 pred post(a:array,begin:ℕ[N-1+1],end:ℤ[-1,N-1+1],
68   result:array)
69 ⇔ isPermutationOf(a,result)
70   ∧ (
71     begin < N ∧ end ⩾ 0 ∧ begin < end
72     ⇒ isSorted(result,begin,end-1)
73   )
74   ∧ (
75     ∀i:index. (i < begin ⇒ a[i] = result[i])
76     ∧ (i ⩾ end ⇒ a[i] = result[i])
77   );
78
79 //termination term of the outer loop
80 //  the difference between i and j:=out.2,the pivot element
81 fun termin(a:array,begin:ℕ[N-1+1],end:ℤ[-1,N-1+1]):ℕ[N-1+2]
82 = if begin < end then end-begin+1 else 0;
```

For the termination term we use the difference between $begin$ and end. In the case that $begin$ is not smaller than end, the termination term would not decrease. To avoid this special case we define the termination term as 0 if $begin = end$.

6.2.2 *Implementation for Recursive Lists*

For lists, we have a way to optimize the algorithm such that we do not need to define the $append$ function. This can be done by giving the function one

more argument and to then carry that argument recursively to the end of the list, where it gets added. We define this variant as "qSort" and write separate specifications for it.

```
292 fun quickSort(a:list):list
293   requires pre(a);
294   ensures post(a,result);
295   decreases termin(a);
296 = match a with
297   {
298     nil -> a;
299     cons(elem:nat,rem:list) ->
300       let
301         lists = partitioning(rem,elem)
302       in
303       append(
304         quickSort(lists.2),
305         list!cons(elem,quickSort(lists.1))
306       );
307   };
308
309 fun qSort(a:list,b:list):list
310   requires pre2(a,b);
311   ensures post2(a,b,result);
312   decreases termin(a);
313 = match a with
314   {
315     nil -> b;
316     cons(elem:nat,rem:list) ->
317       let
318         lists = partitioning(rem,elem)
319       in
320       qSort(
321         lists.2,
322         list!cons(elem,qSort(lists.1,b))
323       );
324   };
```

We can validate that these algorithms are equivalent with respect to the model.

```
167 //quickSortEquivalence()
168 //  quickSort and qSort are equivalent
169 theorem quickSortEquivalence()
170 ⇔ ∀a:list.quickSort(a) = qSort(a,list!nil);
```

The specification of the "qSort"-variant is an extension of the original:

```
136 //pre-Condition
137 //  (*) true
138 pred pre(a:list) ⇔ true;
139
140 //post-Condition
141 //  (*) result is sorted
142 //  (*) result is a permutation of a
143 pred post(a:list,result:list)
144 ⇔ listLength(a) = listLength(result)
145   ∧ isSorted(result)
146   ∧ isPermutationOf(toArray(a),toArray(result));
147
148 //termination term
149 //  each recursion the array-size gets smaller
150 fun termin(a:list):ℕ[N] = listLength(a);
151
152 //pre-Condition for the qSort-Variant
153 //  (*) b is sorted
154 //  (*) elements in a are smaller or equal then the
155 //      smallest element in b
156 pred pre2(a:list,b:list)
157 ⇔ pre(a)
158   ∧ listLength(a)+listLength(b) ⩽ N
159   ∧ isSorted(b)
160   ∧ match b with
161   {
162     nil -> true;
163     cons(elem:nat,rem:list) ->
164       elementsAreSmallerEqual(a,elem);
165   };
166
167 //post-Condition for the qSort-Variant
168 //  (*) result is sorted
169 //  (*) result is a permutation of a+b
170 pred post2(a:list,b:list,result:list)
171 ⇔ post(append(a,b),result);
```

The pre-condition of the "qSort" variant states that the length of both lists together should not exceed the maximal length N given by RISCAL and the second argument should be a sorted list. For the post-condition of the "qSort" variant we consider the original post condition for the joined list. Regarding the other specifications they are the same as the ones we defined for a general sorting algorithm.

6.2.3 *Implementation for Linked Lists*

For linked Lists as well as for all following algorithms we use RISCAL2.

```
138 fun quickSortRec(rem:pointer,end:pointer,s:store):store
139   requires
140     leadsTo(end,N,s)
141     ∧ leadsTo(rem,end,s);
142   ensures
143     isStorePermutationOf(s,result)
144     ∧ (
145       ∀c:pointer.c<null ∧ s[c] ≠ result[c]
146       => leadsTo(rem,c,s)∧leadsTo(c,end,s)
147     )
148     ∧ leadsTo(end,N,result)
149     ∧ leadsTo(rem,end,result)
150     ∧ isPermutationOf(
151       toArray(rem,end,s),toArray(rem,end,result)
152     )
153     ∧ listLength(rem,end,s) = listLength(rem,end,result)
154     ∧ isSorted(rem,end,result);
155   decreases
156     listLength(rem,end,s);
157 = if rem = end then
158         s
159   else
160           let
161             pivot = partitioning(rem,end,rem,s).pivot,
162             newS = partitioning(rem,end,rem,s).store
163           in
164     quickSortRec(
165       rem,
166       pivot,
167       quickSortRec(newS[pivot].tail,end,newS)
168     );
169 fun quickSort(a:pointer,s:store):store
170   requires
171     leadsTo(a,N,s);
172 = quickSortRec(a,N,s);
```

The implementation is done the same way as for arrays with the difference that we do not need *begin* and that *end* is not an index anymore but a pointer. The pre-condition states that *end* is an element in the list with the pointer *rem*. Most noticeable in the post-condition is the statement that all chances in the store are happening in the list. All the other conditions are essentially the same as the onces from the implementation for arrays. The

definition of isSorted as well as listLength have chanced, so that they work for linked lists as well:

```
107 //isSorted(a:list,end:pointer,s:store)
108 //  returns if all elements of a are sorted
109 pred isSorted(a:pointer,end:pointer,s:store)
110   decreases listLength(a,N,s);
111 ⇔ if a = end then
112     true
113   else if s[a].tail = end then
114           true
115   else
116           s[s[a].tail].head ⩾ s[a].head
117           ∧ isSorted(s[a].tail,end,s);
```

```
26 //listLengthOfList(a:pointer,s:store):ℕ[N]
27 //  returns the length of the pointer list a
28 fun lengthOfLinkedListRec(a:pointer,s:store,out:ℕ[N]):ℕ[N]
29   decreases N-out;
30 = if a = null then
31     out
32   else
33     lengthOfLinkedListRec(s[a].tail,s,out+1);
34 fun lengthOfLinkedList(a:pointer,s:store):ℕ[N]
35 = lengthOfLinkedListRec(a,s,0);
36
37 fun listLength(a:pointer,end:pointer,s:store):ℕ[N]
38 = lengthOfLinkedList(a,s)-lengthOfLinkedList(end,s);
```

6.3 OUTCOME OF THE CHECKS

We specified the different variants of the partitioning algorithm, validated them for the model and finally formulated the verification conditions (these are now generateable with the second version of RISCAL, which we have already used for the implementation for linked lists).

Likewise we specified the various variant of the sorting algorithm and checked them by using RISCAL. We successfully derived the verification conditions (as in the case of linked lists, we used the second version of RISCAL which derived them automatically). These can now be used to formally prove the algorithm.

For the checks of the implementations for arrays and recursive lists, we used a maximal length $N = 4$ and an upper bound of $M = 2$ for the elements. Checks regarding arrays and recursive lists were all executed within a minute. For linked lists we checked if the specifications hold for a store

of size $N = 4$ and then started validating them using the built-in feature of RISCAL. For some parts like, checking the correctness of the result, the process took too long, and we reduced N to 3 to decrease the time to about two minutes.

6.4 CONCLUSION

The specification of the partitioning algorithm for arrays and recursive lists turned out to be quite straight forward whereas the specification for linked lists was the exact opposite. The main problem was to understand how to model the store as a mutable object. This meant that the information of the list was passed with the store and the pointers were useless (or invalid) with a wrong or missing store. Another obstacle was the role of the pivot-pointer. Although in an array-representation *pivot* symbolized the divide between smaller/equal elements and larger elements, now the pointer actually symbolized the start of the list. This meant that one could not specify anything regarding the elements that were smaller or equal than *pivot* and already sorted out. By negating the statement and saying "all changes are located in the resulting list and the list with the pointer *pivot* contains exclusively all elements that are larger" the algorithm can still be specified. This second obstacle turned out to be trivial for the imperative variant.

7 HEAPSORT

For our last sorting algorithm we look at HeapSort and its sub-function Heapify. The general idea is to use a Max-Heap data structure. A Max-Heap is a binary tree, where every node is greater or equal than its children. We can then take the root node (equal to the highest value in the tree) and replace it with an arbitrary one (which invalidates the Max-Heap definition). Once we have derived a new Max-Heap from this new tree, we can repeat the process. The function that takes a binary tree, where only the top layer is not a Max-Heap and returns a full Max-Heap is called a Heapify-function. We do not need to introduce a new datatype: binary trees can be represented as single arrays.

HeapSort Algorithm
Input : an arbitrary array.
Output : a sorted permutation of the input

Create a Max-Heap from the array
Loop while the heap is not empty
    swap the first element with the last one
    remove the last element
    heapify the remaining array

We will use a dynamic array for the implementation. This way removing the last element is equivalent to reducing the size of the array by one. For a given element at position i in our array the children of that element are now defined as the elements at positions $2 \cdot i + 1$ and $2 \cdot i + 2$.

7.1 HEAPIFY

The Heapify-algorithm requires that the two subtrees of the root are already Max-Heaps. The algorithm swaps the top element with the greater one of its children and then goes recursively down the corresponding subtree. If the top element is greater than both children, the algorithm stops.

Heapify Algorithm
Input : a tree a where the subtrees are Max-Heaps
Output : a Max-Heap permutation of the input

let $largest = max(root, leftchild, rightchild)$
if $(largest \neq root)$
    swap $largest$ with $root$
    apply the algorithm to the tree starting at $root$

We can now implement that algorithm using RISCAL.

```
78  fun heapify(a:array, size:ℕ[N], root:index):array
79  requires
80    root < size
81    ∧ (2*root+1 < size ⇒ isMaxHeap(a,size,2*root+1))
82    ∧ (2*root+2 < size ⇒ isMaxHeap(a,size,2*root+2));
83  ensures
84    isMaxHeap(result,size,root)
85    ∧ isPermutationOf(a,result)
86    ∧ ∀i:index.i ⩾ size ⇒ a[i] = result[i]
87    ∧ (∀j:index. a[j] ≠ result[j] ⇒ leadsTo(root,j));
88  decreases
89    N-root;
90  =
91    let
92      left_node = 2*root+1,
93      right_node = 2*root+2,
94      largest_part =
95        if
96          left_node < size ∧ a[left_node] > a[root]
97        then
98          left_node
99        else
100         root,
101     largest =
102       if
103         right_node < size ∧ a[right_node] > a[largest_part]
104       then
105         right_node
106       else
107         largest_part
108   in
109   if largest ≠ root then
110     heapify(
111       a with [root] = a[largest] with [largest] = a[root],
112       size,largest
113     )
114   else
115     a;
```

We use the predicate $isMaxHeap$ to specify whether the tree, described by an array, is a Max-Heap.

```
24 pred isMaxHeap(a:array,size:ℕ[N],root:index)
25 ⇔ let
26     left_node = 2*root + 1,
27     right_node = 2*root + 2
28   in
29     ( left_node < size
30       ⇒ a[root] ⩾ a[left_node]
31       ∧ isMaxHeap(a,size,left_node)
32     )
33     ∧ ( right_node < size
34       ⇒ a[root] ⩾ a[right_node]
35       ∧ isMaxHeap(a,size,right_node)
36     );
```

For the post-condition we need to verify that only element in the trees starting at $root$ are changed. This is done by first checking that element can only be changed in the interval between $root$ and $size$ and second verifying that the root element leads to the change. This last attribute is specified by the predicate $leadsTo$.

```
45 //leadsTo(from:index,to:index)
46 //  states if "from" leads to "to"
47 //  by recursively going down each branch
48 pred leadsTo(from:index,to:index)
49 ⇔ from = to
50   ∨ (2*from+1 < N ∧ leadsTo(2*from+1,to))
51   ∨ (2*from+2 < N ∧ leadsTo(2*from+2,to));
52
53 //leadsTo2(from:index,to:index)
54 //  checks if "from" leads to "to"
55 //  using the formular
56 //    to \in from*2^n+[2^n-1,2^(n+1)-2]
57 pred leadsTo2(from:index,to:index)
58 ⇔ (from ≠ to)
59   ⇒ ∃n:index. (0<n ∧ 2^n-1<N)
60   ∧ ∃i:index.(2^n-1⩽i ∧ i ⩽ 2^(n+1)-2 ∧ i < N)
61     ∧ from*2^n + i = to;
62
63 theorem leadsToEquivalence()
64 ⇔ ∀i:index.∀j:index.leadsTo(i,j) ⇔ leadsTo2(i,j);
```

The first version is defined recursively: Go through the nodes until it either reaches the end or the searched node. The second version uses the location of nodes in binary-trees: $to = 2 \cdot from + i$ for $i \in \{1, 2\}$ to derive a general

formula for the relation between two nodes. A short proof per induction validates the new formula:

Proof. We want to show:

$$a \text{ leads to } b \Leftrightarrow \exists n \in \mathbb{N} : \exists j \in \{2^n - 1, ..., 2^{n+1} - 2\} : b = 2^n a + j$$

We already know that if b is the child of a, then $b = 2a + i$ for $i \in \{1, 2\}$. Given that $b = 2^n a + j$ for $j \in \{2^n - 1, ..., 2^{n+1} - 2\}$ for some $n \in \mathbb{N}$, we can state for a given $i \in \{1, 2\}$:

$$2b + i = 2(2^n a + j) + i = 2^{n+1} a + (2j + i)$$

We introduce a new variable $k := 2j + i$. By the definition of i and j we know that

$$k \in \{2(2^n - 1) + 1, 2(2^{n+1} - 2) + 2\} = \{2^{n+1} - 1, 2^{n+2} - 2\}$$

and therefore we conclude the premise

$$2^{n+1} a + k \text{ for } k \in \{2^{n+1} - 1, 2^{n+2} - 2\}$$

□

After validating that both versions are equivalent we use the first version for all the specifications, as it is the faster one using RISCAL. Note that in theory the second version could be implemented faster if one would define the upper bound 2^n from $\leqslant$ to for n. This way one could stop searching for another n as soon as the upper bound was reached. If this optimization for the second version was in place it would supersede the first.

As the algorithm is written in a tail-recursive style, we may also implement it by a loop:

```
134 proc heapifyLoop(a:array, size:ℕ[N], root:index):array
135 requires
136   root < size
137   ∧ (2*root+1 < size ⇒ isMaxHeap(a,size,2*root+1))
138   ∧ (2*root+2 < size ⇒ isMaxHeap(a,size,2*root+2));
139 ensures
140   isMaxHeap(result,size,root)
141   ∧ isPermutationOf(a,result)
142   ∧ ∀i:index.(root > i ∨ i ⩾ size) ⇒ a[i] = result[i]
143   ∧ (∀j:index. a[j] ≠ result[j] ⇒ leadsTo(root,j));
144 {
145   var i:index := root;
146   var b:array := a;
147   var largest:index := root;
148
```

```
149   do
150     decreases
151       if largest  ≠ i then N-largest else 0;
152     invariant
153       largest ⩾ root ∧ i ⩾ root ∧ i < size ∧ largest < size
154       ∧ (largest = i ∨ largest = 2*i+1 ∨ largest = 2*i+2)
155       ∧ b[largest] ⩾ b[i]
156       ∧ (b[largest] = b[i] ⇒ largest = i)
157       ∧ ( ∀j:index.j ⩾ root ∧ j ≠ i
158         ⇒ (j*2+1 < size ⇒ b[j] ⩾ b[2*j+1])
159         ∧ (j*2+2 < size ⇒ b[j] ⩾ b[2*j+2])
160       )
161       ∧ (∀j:index. a[j] ≠ b[j] ⇒ leadsTo(root,j))
162       ∧ ( ∀j:index.j ⩾ root ∧ leadsTo(j,i) ∧ j ≠ i
163         ⇒ b[j] ⩾ b[largest]
164       )
165       ∧ (2*largest+1 < size ⇒ isMaxHeap(b,size,2*largest+1))
166       ∧ (2*i+1 < size ⇒ b[largest] ⩾ b[2*i+1])
167       ∧ ( 2*largest+2 < size ⇒ isMaxHeap(b,size,2*largest+2))
168       ∧ (2*i+2 < size ⇒ b[largest] ⩾ b[2*i+2])
169       ∧ isPermutationOf(a,b)
170       ∧ (∀j:index.(root > j ∨ j ⩾ size) ⇒ a[j] = b[j]);
171   {
172     var temp:nat := b[i];
173     b[i] := b[largest];
174     b[largest] := temp;
175     i := largest;
176     var left_node:ℕ[2*N] := 2*i+1;
177     var right_node:ℕ[2*N] := 2*i+2;
178
179     if
180       left_node < size ∧ b[left_node] > b[i]
181     then
182       largest := left_node;
183     if
184       right_node < size ∧ b[right_node] > b[largest]
185     then
186       largest := right_node;
187   }
188   while largest ≠ i;
189   return b;
190  }
```

A lot of the conditions in the invariant are trivial facts about the bounds of the particular variables. These are fundamental for the invariant to be persistent during the loop.

An important part of the invariant is that it exactly specifies what elements are mutable in the next iteration:

$$\begin{aligned} &\forall j : index.a[j] \neq b[j] \Rightarrow leadsTo(root, j) \\ \wedge \forall j : index.j \geqslant root \wedge j \neq i \Rightarrow &(j * 2 + 1 < size \Rightarrow b[j] \geqslant b[2 * j + 1]) \\ &\wedge (j * 2 + 2 < size \Rightarrow b[j] \geqslant b[2 * j + 2]) \end{aligned}$$

The first condition states that, if root is not the highest node of the tree, mutations can only occur in the sub-tree starting at root. The second condition states that the node at i may be the only node that is not greater than its children. Therefore only i and its children can change next iteration.

Another noteworthy condition states that the highest computed value largest is smaller or equal than all parents of i.

7.2 HEAPSORT

We will use the recursive version of heapify to implement the HeapSort algorithm.

```
45 proc HeapSort(a:array):array
46 ensures
47   isSorted(result,0,N-1)
48   ∧ isPermutationOf(a,result);
49 {
50   var b:array := a;
51   for var i:ℤ[-1,N-1] =  N / 2 - 1; i ⩾ 0; i=i-1 do
52   invariant
53     (∀j:index.j > i ⇒ isMaxHeap(b,N,j))
54     ∧ isPermutationOf(a,b);
55   decreases
56     i+1;
57   { b = heapify(b,N,i); }
58
59   for var i:ℤ[-1,N-1] = N-1; i ⩾ 0; i=i-1 do
60   invariant
61     ( i<N-1
62       ⇒ isSorted(b,i+1,N-1)
63       ∧ b[0] ⩽ b[i+1]
64     )
65     ∧ isMaxHeap(b,i+1,0)
66     ∧ isPermutationOf(a,b);
67   decreases
68     i+1;
69   {
70     var temp:nat := b[0];
```

```
71      b[0] = b[i];
72      b[i] = temp;
73      b = heapify(b,i,0);
74    }
75    return b;
76  }
```

To build the initial Max-Heap we use *heapify* and apply it from the bottom upwards. The invariants for both loops are also very obvious: generally we want the result to be a permutation of the original input and we can also specify that we have the required conditions for the Heapify algorithm.

7.3 OUTCOME OF THE CHECKS

The results of the Heapify algorithm is not unique and the preconditions are not trivial. We specified and validated both the imperative and functional version, which includes checking the verification conditions provided by RISCAL.

The HeapSort has a trivial pre-condition and a unique result. There were no difficulties during the specification or validation process.

Both versions where checked for sequences of length $N = 4$ and an upper bound of $M = 2$. All checks where completed within a minute.

7.4 CONCLUSION

Using an array to represent a binary tree was quite an abstraction task as the array was handled like a tree-structure while keeping the random access property. The invariant needed not only the properties of the array but also of the tree structure, resulting in the introduction of the *leadsTo* predicate.

8 CONCLUSIONS AND SUMMARY

In the future, methods to produce safe and trustworthy software will be increasingly important. But in the current state we are still far from sufficiently proving a real production code. The work done in this bachelor thesis is not ground breaking for sure, but it gives a glimpse into the steps necessary to produce verified and therefore hopefully bug free software.

Basic sequence algorithms, as those presented in this thesis, are complex enough to get a feeling as for how the verification process of more complex algorithms might look like. RISCAL from version 2 onwards can generate usable verification conditions, making some of the earlier work of this thesis (the manual construction of the verification conditions) obsolete. Nevertheless the exact specification of the various methods and of annotations (invariants) that are strong enough to prove the correctness of the algorithms is an outcome of the thesis that remains relevant of the future. But these verification conditions still have to be verified by a human using a proof assistant. Some verification conditions might even be automatically verifiable using an SMT-Solver. Maybe a future RISCAL version might have such a feature implemented. We know that because of the state explosion problem it will never be possible to fully automate this process. Still, there is room for improvement.

The question as to how far one might be able to automate the verification process becomes useless if a problem is not correctly specified. Part of this thesis was also to specify the algorithms and to refine the specification, such that "unwanted" side-effects for correct implementations appear as little as possible. As time goes on, automated validating will move forward, while specification will stay in the realm of humans. We will never be able to automate the specification process and therefore we can only rely on the reasoning and the wit of humans to avoid dangerous errors. If there is one thing one should take away from this thesis, it is to be sure to know what one asks for and what in includes.

BIBLIOGRAPHY

[Abr+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: An Open Toolset for Modelling and Reasoning in Event-B". In: *International Journal on Software Tools for Technology Transfer* 12.6 (2010), pp. 447–466.

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: https://doi.org/10.1007/978-3-319-49812-6.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2. URL: https://doi.org/10.1016/S0065-2458(03)58003-2.

[Bjø79] Dines Bjørner. "The Vienna Development Method (VDM): Software Specification &Amp; Program Synthesis". In: *Proceedings of the International Conference on Mathematical Studies of Information Processing*. London, UK, UK: Springer-Verlag, 1979, pp. 326–359. ISBN: 3-540-09541-1. URL: http://dl.acm.org/citation.cfm?id=648090.747626.

[Buc+16] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. "Theorema 2.0: Computer-Assisted Natural-Style Mathematics". In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 149–185. ISSN: 1972-5787. DOI: 10.6092/issn.1972-5787/4568. URL: https://jfr.unibo.it/article/view/4568.

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. 2000, pp. 154–169. DOI: 10.1007/10722167_15. URL: https://doi.org/10.1007/10722167_15.

[Cla+18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018.

ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8. URL: https://doi.org/10.1007/978-3-319-10575-8.

[Cok14] David R. Cok. "OpenJML: Software Verification for Java 7 Using JML, OpenJDK, and Eclipse". In: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*. 2014, pp. 79–92. DOI: 10.4204/EPTCS.149.8. URL: https://doi.org/10.4204/EPTCS.149.8.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[DF08] Ewen Denney and Bernd Fischer. "Explaining Verification Conditions". In: *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*. 2008, pp. 145–159. DOI: 10.1007/978-3-540-79980-1_12. URL: https://doi.org/10.1007/978-3-540-79980-1_12.

[Dij75] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (1975), pp. 453–457. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

[Fla+13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "PLDI 2002: Extended static checking for Java". In: *SIGPLAN Notices* 48.4S (2013), pp. 22–33. DOI: 10.1145/2502508.2502520. URL: http://doi.acm.org/10.1145/2502508.2502520.

[FMV14] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. "Loop invariants: Analysis, classification, and examples". In: *ACM Comput. Surv.* 46.3 (2014), 34:1–34:51. DOI: 10.1145/2506375. URL: http://doi.acm.org/10.1145/2506375.

[Hat+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. "Behavioral interface specification languages". In: *ACM Comput. Surv.* 44.3 (2012), 16:1–16:58. DOI: 10.1145/2187671.2187678. URL: http://doi.acm.org/10.1145/2187671.2187678.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 978-0-262-10114-1. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN: 0201896850. URL: http://www.worldcat.org/oclc/312994415.

[Lei17] K. Rustan M. Leino. "Accessible Software Verification with Dafny". In: *IEEE Software* 34.6 (2017), pp. 94–97. DOI: 10.1109/MS.2017.4121212. URL: https://doi.org/10.1109/MS.2017.4121212.

[LM09] Martin Leucker and Carroll Morgan, eds. *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*. Vol. 5684. Lecture Notes in Computer Science. Springer, 2009. ISBN: 978-3-642-03465-7. DOI: 10.1007/978-3-642-03466-4. URL: https://doi.org/10.1007/978-3-642-03466-4.

[MSS99] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. "Model-Checking: A Tutorial Introduction". In: *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*. 1999, pp. 330–354. DOI: 10.1007/3-540-48294-6_22. URL: https://doi.org/10.1007/3-540-48294-6_22.

[Rit16] Daniela Ritirc. "Formally Modeling and Analyzing Mathematical Algorithms with Software Specification Languages & Tools". MA thesis. Research Institute for Symbolic Computation (RISC); Johannes Kepler University; Linz; Austria, 2016.

[Sch09] Wolfgang Schreiner. "The RISC ProofNavigator: a proving assistant for program verification in the classroom". In: *Formal Asp. Comput.* 21.3 (2009), pp. 277–291. DOI: 10.1007/s00165-008-0069-4. URL: https://doi.org/10.1007/s00165-008-0069-4.

[Sch17] Wolfgang Schreiner. *The RISC Algorithm Language - Tutorial and Reference Manual*. Tech. rep. Research Institute for Symbolic Computation (RISC); Johannes Kepler University; Linz; Austria, 2017. URL: https://www.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf.

[Sch18] Wolfgang Schreiner. "Validating Mathematical Theories and Algorithms with RISCAL". In: *Intelligent Computer Mathematics*. Ed. by F. Rabe, W. Farmer, G. Passmore, and A. Youssef. Vol. 11006. Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence. The final authenticated version is available online at Springer. Berlin: Springer, 2018, pp. 248–254. URL: https://doi.org/10.1007/978-3-319-96812-4_21.

[SBF18] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. "Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models". In: *Post-Proceedings ThEdu'17*. Ed. by Pedro Quaresma and Walther Neuper. Vol. 267. Electronic Proceedings in Theoretical Computer Science (EPTCS). Open Publishing Association, 2018, pp. 120–139. URL: http://dx.doi.org/10.4204/EPTCS.267.8.

[SW16] Robert Sedgewick and Kevin Wayne. *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016. ISBN: 978-0-1343-8468-9.

[TGK09] Elena Tushkanova, Alain Giorgetti, and Olga Kouchnarenko. *Specifying and Proving a Sorting Algorithm*. 2009. URL: https://hal.archives-ouvertes.fr/hal-00429040/document.

A RISCAL SPECIFICATIONS

The content of this appendix (all RISCAL specifications developed in this thesis) is available from the RISCAL Website:
https://www.risc.jku.at/research/formal/software/RISCAL

EIDESTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, December 21, 2018 Lucas Payr