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

Lucas Payr

Bachelor Thesis Report

Supervisors: Wolfgang Schreiner & Wolfgang Windsteiger

25. April 2018

Understanding the Title

- **Sequences** - generalization of arrays, recursive lists and pointer lists
- **Sequence algorithms** - includes searching and sorting.
- **Formalization** - to mathematically specify the problem and define an algorithm solving the problem
- **Validation** - to increase the confidence that the algorithm satisfies the specification
- **Finite model** - model of algorithms and specifications using variables over finite domains.
- **Model checking** - to check if the algorithm meets given specification in the finite model

Tool: The RISC Algorithm Language (RISCAL)

Goal of the Thesis

Motivation:

- Algorithm text books usually do not give formal correctness proofs
- Do not give formal specifications and annotations (loop invariants) required for such proofs

Goal of the Thesis

Goals:

1. Give formal problem specifications
2. Define algorithms solving the problems
3. Give additional information required for proofs (esp. loop invariants)
4. Validate algorithms, specifications, annotations by checking
5. Define verification conditions that imply correctness of algorithms with respect to specification and check their validity

Goal of the Thesis

- **Algorithms**
  - Linear search, Binary search
  - Insertion sort
  - Quick-sort $\rightarrow$ subalgorithm "Partitioning"
  - Merge-sort $\rightarrow$ subalgorithm "Merge"
  - Heap-sort $\rightarrow$ subalgorithm "Heapify"
- **Data Types**
  - Arrays
  - Recursive lists $\rightarrow$ recursive algorithms
  - Pointer linked lists

RISC Algorithmic Language

- Algorithmic language and associated software system for checking algorithms, specifications, annotations and theorems
- Supports unicode-symbols
- Uses variables over finite domains
- All statements can be executed
- All formulas can be evaluated (checkable via brute force)

Annotations

Specifications

- **Pre-condition** - what the algorithm requires from its arguments.
- **Post-condition** - what the algorithm ensures for its results.

Extra Information

- **Loop Invariant** - condition that must be satisfied before and after each iteration.
- **Termination term** - integer term whose value gets smaller after each iteration, but does not become negative!

Annotations

```
val M:ℕ;
val N:ℕ;
type array = Array[N,ℕ[M]];

proc reverse(a:array):array
requires true;
ensures ∀i:ℕ[N-1].a[i] = result[N-1-i];
{
  var b:array :=a;
  for var i:ℕ[N/2] := 0;i ≤ N/2-1; i:=i+1 do
    invariant ∀j:ℕ[N-1]. if j<i ∨ j>N-1-i then
        a[j] = b[N-1-j]
      else
        a[j] = b[j];
    decreases N/2-i;
  {
    b[i] := a[N-1-i];
    b[N-1-i] := a[i];
  }
  return b;
}
```

Validation Process

Checking the Algorithms

- Algorithm "works as intended"
- Pre-condition is not to strong (holds for expected inputs)
- Post-condition is not to strong (is ensured by algorithm)
- Invariants are not too strong
- Termination terms are adequate

Validating the Specification

- Pre-condition is not too weak (e.g. is not trivial)
- Post-condition is not too weak (e.g. defines result uniquely)

Validate the Invariants

- Formulate verification conditions $\rightarrow$ invariant is not too weak

Validation Conditions

- **VC1:** The invariant holds before the loop starts
- **VC2:** The termination term never becomes negative.
- **VC3:** Every loop iteration preserves the invariant and decreases the termination term
- **VC4:** On termination of the loop, the invariant implies the postcondition

Validation Conditions

```
theorem VC_beginning(a:array,b:array,i:ℕ[N/2])
  requires preconditon(a);
<=> b = a ∧ i = 0 => loop_invariant(a,b,i);

theorem VC_termination(a:array,b:array,i:ℕ[N/2])
  requires preconditon(a);
<=> loop_invariant(a,b,i) => termination_term(i) ≥0;

theorem VC_iteration(a:array,b:array,i:ℕ[N/2])
  requires preconditon(a);
<=> loop_invariant(a,b,i) ∧ i ≤ N/2-1
  => loop_invariant(a,b with [i]=a[N-1-i] with [N-1-i]=a[i],i+1)
  ∧ termination_term(i+1) < termination_term(i);

theorem VC_end(a:array,b:array,i:ℕ[N/2])
  requires preconditon(a);
<=> loop_invariant(a,b,i) ∧ i > N/2-1 => postcondition(a,b);
```

Insertion Sort for arrays

Idea: "Loop over every element and insert it at the right position."

Algorithm with Two Loops

- Create verification conditions for each loop.
- Invariant of the outer loop = pre-condition of inner loop
- Post-condition of inner loop $\Rightarrow$ invariant of outer loop

$\Rightarrow$ **Live demonstration**

Insertion Sort for Linked Lists

Recursive Functions

- Must terminate $\rightarrow$ termination term is decreased in each recursion step

Algorithms with Two Functions

- No loop invariant $\Rightarrow$ function specifications state relations between the functions
- Pre-condition holds $\Rightarrow$ pre-conditions of subfunctions hold.
- Post-condition of all subfunctions hold $\Rightarrow$ post-condition holds

$\Rightarrow$ **Live demonstration**

Current Work

Started in March 2018

- **Finished** insertion sort for arrays and linked lists
- **Finished** linear search for arrays, linked lists and pointer lists
- **Finished** binary search for arrays
- **Working on** merge algorithm for arrays

Expected completion in August 2018

Thanks for your attention!