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

Lucas Payr

2nd Bachelor Thesis Report

Supervisors: Wolfgang Schreiner & Wolfgang Windsteiger

27. Juni 2018

Goal of the Thesis

Goal:
Present formal specifications and verification conditions for common searching and sorting algorithms.

Searching Algorithms:
Linear Search, Binary Search

Sorting Algorithms:
Insertion Sort, Quick-Sort, Merge-Sort, Heap-Sort

Data Types:
Arrays, Recursive lists and Pointer linked lists

Table of Content

In the Previous Report

- Explained basic concept of validating an algorithm with respect to its specifications
- Gave short introduction into RISC Algorithm Language.
- Showed an example of an imperative and a recursive algorithm and the corresponding validation method.

In this Report

- Short refreshment of the topic
- Validation of binary Search
- Validation of Merge Sort

RISC Algorithm Language

Algorithms are formalized and validated using the **RISC Algorithm Language**.

- Used for formalization, and checking of the specifications. (pre- and post-conditions, termination terms, loop invariants)
- Theorems and predicates (verification conditions) are checkable against the model.

Validation Process

1. Refine pre- and post-condition if possible
    - Pre-condition is not trivial.
    - Post-condition ensures unique results.
2. Validate the specification
    - Let RISCAL check all possible cases.
3. Formulate and validate the verification 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.

Binary Search Algorithm

Input: sorted array, element
Output: *index* if the element was found, else -1

1. If the middle element is equal to the searched element, return the index.
2. Else repeat the previous step for the elements left/right of the middle element.

```
proc binarySearch(a:array, x:nat):ret
  requires pre(a,x);
  ensures post(a,x,result);
{
  var out:ret := -1;
  var left:ℕ[N-1+1] := 0;
  var right:ℤ[-1,N-1] := N-1;
  while out = -1 ∧ left ≤ right do
    invariant inv(a,x,left,right,out);
    decreases termin(left,right,out);
  {
    var i:index := left + (right-left)/2;
    if a[i] = x then
      out := i;
    else if a[i] > x then
      right := i-1;
    else
      left := i+1;
  }
  return out;
}
```

Binary Search Algorithm

Pre-condition:

Array is sorted.

Post-condition:

A index corresponding to the searched element or -1.

Invariant:

Out = -1 if the element is not found outside [left,right] else out = index.

Termination term:

If out = -1 then right-left, else 0.

```
//pre-Condition
pred pre(a:array,x:nat)
⇔ isSorted(a,0,N-1);

//post-Condition
pred post(a:array,x:nat,result:ret)
⇔ (result = -1 ⇔ ∀i:index.a[i] ≠ x)
  ∧ (result ≥ 0 ⇒ a[result] = x);

//invariant
pred inv(a:array,x:nat,left:ℕ[N-1+1],right:ℤ[-1,N-1],
  out:ret)
⇔ -1 ≤ right-left
  ∧ (out ≥ 0 ⇒ a[out] = x)
  ∧ (
    out = -1 ∨ left ≤ right
    ⇒ ∀i:index.
      (0 ≤ i ∧ i < left) ∨ (right < i ∧ i ≤ N-1)
      ⇒ a[i] ≠ x
  );

//termination term
fun terminRec(left:ℕ[N-1+1],right:ℤ[-1,N-1]):ℕ[N]
= right-left+1;
```

Validation of the Binary Search Algorithm

Validation of the imperative variant of binary Search:

1. **Formulate and refine pre- and post-condition.**
   Pre-condition
   - Array is sorted.
     $\rightarrow$ Supporting statement: *isSorted(a:array,from:index,to:index)*
   - Check the pre-condition is not trivial.

   Post-condition
   - A index corresponding to the searched element or -1.
   - Result is not uniquely defined.

Validation of the Binary Search Algorithm

2. **Validate the specification**
   Invariant
   - Main-condition: out = -1 if the element is not found outside [left,right] else out = index.
   - Additional conditions: to ensure that invariant implies post-condition.
3. **Formulate and validate the verification conditions.**

$\Rightarrow$ **Live demonstration of the 3 steps.**

Merge Sort Algorithm

Input: Arbitrary List
Output: Sorted List

1. Split the array into two parts.
2. Recursively apply the algorithm to the parts.
3. Merge the parts.

```
fun mergeSort(a:list):list
  requires true;
  ensures
    listLength(a) = listLength(result)
    ∧ isSorted(result)
    ∧ isPermutationOf(toArray(a),toArray(result));
  decreases listLength(a)+1;
= match a with
  {
    nil -> a;
    cons(elem:nat,rem:list) ->
      match rem with
      {
        nil -> a;
        cons(elem2:nat,rem2:list) ->
          merge(
         mergeSort(split(a).1),
            mergeSort(split(a).2)
          );
      };
  };
```

Merge Algorithm

Input: Two sorted Lists
Output: Combined sorted List

```
//pre-Condition
//  (*) a and b are sorted
//  (*) The length of the combined list a+b
        is not bigger N
pred pre(a:list,b:list) <=>
  isSorted(a)
  ∧ isSorted(b)
  ∧ listLength(a)+listLength(b) ≤ N;

//post-Condition
//  (*) result is a permutation of the list (a,b)
//  (*) result is sorted
pred post(a:list,b:list,result:list) <=>
  listLength(result) = listLength(a)+listLength(b)
  ∧ isPermutationOf(
    toArray(result),
    toArray(append(a,b))
  )
  ∧ isSorted(result);
```

```
fun merge(a:list,b:list):list
  requires pre(a,b);
  ensures post(a,b,result);
  decreases termin(a,b);
  = match a with
  {
    nil -> b;
    cons(elem_a:nat,rem_a:list) ->
      match b with
      {
        nil -> a;
        cons(elem_b:nat,rem_b:list) ->
          if elem_a > elem_b then
            list!cons(elem_a,merge(rem_a,b))
          else
            list!cons(elem_b,merge(a,rem_b));
      };
  };
```

Validation of the Merge Sort Algorithm

1. **Formulate and refine pre- and post-condition.**
   - Pre-condition: trivial (true)
   - Post-condition: list is sorted.
     $\rightarrow$ Check if result is unique.
2. **Validate the specification**
   Termination term
   - The length of the list gets smaller each iteration.

Validation of the Merge Sort Algorithm

3. **Formulate and validate the function specifications.**
    - All preconditions of the subfunctions hold.
        i The pre-condition holds for the split parts.
        ii Iterate over all possible resulting parts that fulfill the post-condition.
        iii The pre-condition of the merge algorithm holds for the two resulting parts.

Validation of the Merge Sort Algorithm

- The post-condition holds given that all sub-functions can be defined by their post-conditions.
  i Iterate over all possible resulting parts that fulfill the post-condition for the split parts.
  ii Iterate over all possible results that fulfill the post-condition of the merge algorithm for the two resulting parts.
  iii The post-condition holds for the result.
- The termination term is always positive or zero and each recursion reduces the termination term.

$\Rightarrow$ **Live demonstration of the 3 steps**

Stable Check of the Merge Sort Algorithm

1. Check if the resulting array of the algorithm is stable.
2. Include the stable-condition in the post-conditions.
3. Formulate and validate the verification conditions.

$\rightarrow$ **The merge sort is stable**

Current Work

Started in March 2018

- **Finished** linear Search, binary search.
- **Finished** insertion sort.
- **Finished** merge algorithm, merge sort.
- **Finished** partitioning algorithm for arrays and recursive lists.
- **Finished** quick sort algorithm for arrays and recursive lists.
- **Working on** partitioning and quick sort for linked lists.
- **Next** heapify and heap sort algorithm

Expected completion in October 2018

Thanks for your attention!