Capita Selecta: Formal verification quicksort

W.R.M. Schols (0975945)

January 12, 2020

1 Introduction

In this report we will prove correctness of quicksort. Quicksort is a sorting algorithm developed by Tony Hoare [1]. Quicksort sorts arrays by selecting a pivot. The pivot is an arbitrary element in the array. After this the quicksort algorithm recurses twice. Quicksort recurses once on the array containing all elements which are smaller or equal to the pivot. The resulting array is placed in front of the pivot. After this quicksort recurses on the array containing all elements larger then the pivot. The resulting array is placed behind the pivot. This results in a sorted array.

Even though Quicksort has a running time of $O(n^2)$ it is still is commonly used in practise. Quicksort is popular since it has an expected running time of $O(n\ log(n))$ with a low overhead. Furthermore quicksort is an in-place algorithm and it can elegantly be defined as a function.

In this paper we will prove correctness of the quicksort algorithm using Coq. There are two major challenges which we have to overcome. We have to implement quicksort in Coq and we have to prove correctness of the algorithm. To prove correctness we will prove that quicksort only returns sorted arrays and we will prove that the sorted array will contain the same elements as the input. The Coq code used in this paper can be found here.

2 Defining quicksort

We want to define quicksort in Coq. In functional programming language quicksort can be defined as:

```
quickSort:: list nat -> list nat
quickSort  [ ] := [ ]
quickSort x:xs := (filter <=  x xs) ++ x ++ (filter > x xs)
```

We could try to specify quicksort like this in Coq:

```
Fixpoint quicksort (l: list nat) :=
    match l with
            | nil => nil
            | h :: t => (quicksort (filter_lte h t)) ++ h :: (quicksort (filter_gt h t))
    end.
```

While this is a correct definition of quicksort Coq doesn't accept it. Coq can only work with finite fixpoints. No infinite recursion is allowed. This is usually not a problem since Coq can identify termination of most fixpoints. Coq notices that if we have (a set of) strictly decreasing parameters in a recursive call then the recursion terminates. In the definition shown above we recurse on `(quicksort (filter_lte h t))`. Trivially `(filter_lte h t)` is strictly smaller then `l`, however Coq does not know this. Coq therefore does not accept our definition. We can solve this by adding a strictly decreasing parameter of type `qs_tree` as proposed by Gabe Dijkstra [2]:

```
Inductive qs_tree : list nat -> Type :=
| qs_leaf : qs_concat nil
| qs_node : forall ( x: nat) (xs: list nat),
  qs_tree (filter (fun y => leb y  x) (xs)) ->
```

```
  qs_tree (filter (fun y => negb (leb y x)) (xs)) ->
    qs_tree (cons x xs).
```

While this type looks very impressive it can be seen as the recursion tree of quicksort. We can look as the nodes in the tree `qs_node x:xs` as a recursive call *quicksort x:xs*. From the definition of quicksort we learn that *quicksort x:xs* makes two recursive calls. Once on all elements smaller or equal to the pivot x and once on all elements larger then x. So in our recursion tree a call *quicksort x:xs* will be represented in a node `qs_node x:xs`. This node then has two children, the first node child is `qs_tree (filter (fun y => leb y x) (xs))` and the second child node is `qs_tree (filter (fun y => negb (leb y x)) (xs))`. The quicksort algorithm terminates once it is called on an empty array. This corresponds to the leaf *(quicksort nil)*.

We can now define a quicksort helper function with one more parameter:

```
Fixpoint qs_helper (l : list nat) (q : qs_tree l) {struct q} : list nat :=
  match q with
  | qs_leaf => nil
  | qs_node x xs q0 q1 =>
      (qs_helper (filter (fun y : nat => leb y x) xs) q0) ++ (x ::
      (qs_helper (filter (fun y : nat => negb (leb y x)) xs) q1))
  end.
```

By definition of `qs_tree l` Coq sees that `q` is a strictly decreasing parameter. From this Coq can conclude that the recursion terminates. However this definition is not powerful enough to prove correctness of quicksort. If we call `qs_helper` we need to provide an argument `q` of type `qs_tree l`. When quicksort recurses we know that `q0` and `q1` are of type `qs_tree l'` for some `l'` however we don't know what `l'` is. Because of this we cannot effectively use induction on the recursion tree. There are two ways to provide the extra information needed to get the value of `q0,q1` and use induction. In this paper we will define an executable lemma `create_qs_tree` which converts an array `l : list nat` into a recursion tree `q : qs_tree l`. Using this lemma we can give a new definition of quicksort. Because the lemma is executable Coq can derive the complete type of `q1,q0` and we can use an induction. Alternatively we can provide two invariants as proposed by Dijkstra [2]. Using these invariants dijkstra can add the required information about `q0,q1` to `qs_helper`. We will discuss both approaches.

2.1 Approach create_qs_tree

We want to define a lemma which create a recursion tree `qs_tree l` for any input l: list nat.

```
Lemma create_qs_tree: forall (l : list nat), (qs_tree l).
Proof.
...
Defined.
```

Note that we need to make this lemma executable. So we need to conclude the proof with `Defined` and we cannot use lemmas from the standard Coq libraries. Using `create_qs_tree` we can then simply define quicksort as.

```
Definition quickSort (l :list nat) := qs_helper l (create_qs_tree l).
```

An advantage of this approach is that this definition of quickSort allows us to give very clean proofs. Lemmas don't need to consider the parameter `qs_tree`, all goals and assumptions are easily readable and using recursion is simple. Coq can derive a lot of facts without having to use any invariants. However there are two problems. The first problem is that the lemma `create_qs_tree` is hard to prove. The second problem is more complex. Coq knows `quickSort l = qs_helper l q`. If `l` is not empty then `qs_helper l q` is:

```
(qs_helper (filter (fun y :  nat => leb y x) xs) q0) ++
(x ::  (qs_helper (filter (fun y :  nat => negb (leb y x)) xs) q1)).
```

Coq cannot fold this back to:

```
(quickSort (filter (fun y :  nat => leb y x) xs)) ++
(x ::  (quickSort (filter (fun y :  nat => negb (leb y x)) xs))).
```

This is unfortunate since this means that we cannot reuse lemmas which we have proven for `quickSort` and Coq cannot derive anything from qs_helper again. So we can only use induction on the recursion tree once. These downsides are not important to prove correctness for quicksort however if these downside can be significant if the approaches are used to prove correctness of different recursive algorithms.

2.2 Approach Dijkstra

Dijkstra [2] proposes an alternative approach. Dijkstra proposes a new definition of `qs_helper`. In this definition Dijkstra uses two invariants. The invariants are used to guide the recursion. We need to prove invariants:

```
Lemma qs_acc_inv_1_0'' : forall (l:list nat)(x: nat)(xs : list nat),
qs_tree l -> l = cons x xs -> qs_tree (filter (fun y => leb y  x) xs).
Proof.
...
Defined.

Lemma qs_acc_inv_1_1'' : forall (l:list nat)(x: nat)(xs : list nat),
qs_tree l -> l = cons x xs -> qs_tree (filter (fun y => negb (leb y x)) xs).
Proof.
...
Defined.
```

Note that both invariants need to be executable. We can now define quicksort as:

```
Fixpoint qss (xl : list nat) (q : qs_tree xl) {struct conc}:=
match xl as _y0 return (xl = _y0) -> list nat with
   | nil       => fun  _h0 => nil
   | cons x xs => fun _h0 =>
      (qss (filter (fun y => leb y  x) xs)(qs_acc_inv_1_0'' _ _ _ q _h0))
      ++ (cons x
      (qss (filter (fun y => negb (leb y x)) xs)(qs_acc_inv_1_1'' _ _ _ q _h0)))
  end (refl_equal xl).
```

This definition has two advantage over our approach. First of all the invariants are a lot easier to prove. The second advantage is that we don't need to unfold any definition. We never lose information so we can always reuse lemmas and reuse induction. A disadvantage of dijkstras approach is that the definitions are more cumbersome to work with. Coq does not handle the invariants automatically and lemmas always need reason about arbitrary recursion trees `q :  qs_tree xl`. We can also not evaluate this definition of quicksort without first generating a recursion tree.

3 Lemmas quickSort

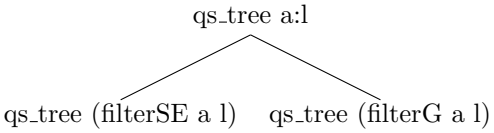
In this section we outline the proofs of the lemmas which are used to implement quicksort and we will prove correctness of quicksort. First we will discuss the lemma `create_qs_tree` which states that we can create a recursion tree `q:  qs_tree l` for any array `l :  list nat`. After this we prove the invariants of Dijkstra. Lastly we prove correctness of quicksort by proving the lemma `qs_In_equiv` which states that quicksort doesn't add or remove elements and the lemma `qs_sorted` which states that quicksort only returns sorted arrays.

3.1 Create_qs_tree

We want to prove:

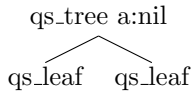
```
Lemma create_qs_tree: forall (l : list nat), (qs_tree l).
```

This is the easily most complicated proof of this report. We want to prove that `qs_tree l` exists for any `l`. We start by induction on `l`. Trivially if `l` is empty then `qs_leaf` is a valid `qs_tree l`. The step states that `qs_tree a:l` exists given that `qs_tree l` exists. Using the constructor of `qs_tree a:l` we need to prove its children `qs_tree filterSE a l` and `qs_tree filterG a l` exist.



So we have two new proof goals:
1: `qs_tree (filterSE a l)`
2: `qs_tree (filterG a l)`

We use structural induction on our induction hypothesis `qs_tree l`. If `qs_tree l = qs_leaf` then `l` is empty. Proof goals 1 trivially hold since the filter of an empty list is empty.

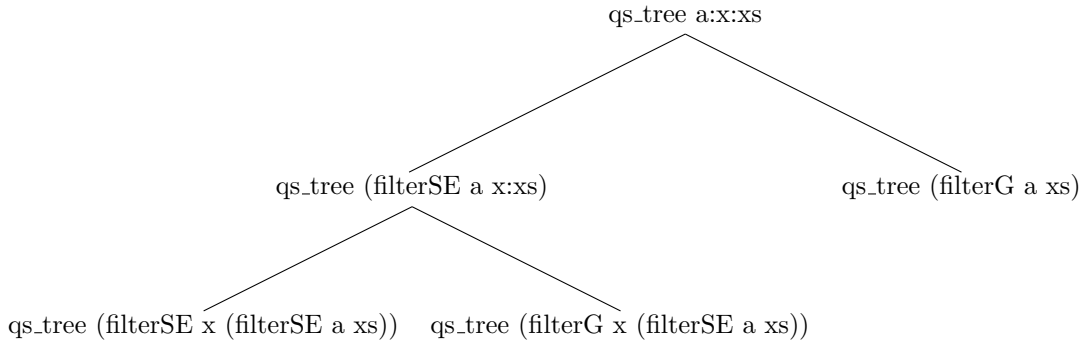


Next we handle the step `qs_tree l = qs_tree x:xs`. We need to prove prove goals:
1': `qs_tree (filterSE a x:xs)`
2': `qs_tree (filterG a x:xs)`

We will first prove proof goal 1'. We have two induction hypothesis:
`IH1 = qs_tree (filterSE a (filterSE x xs))`
`IH2 = qs_tree (filterSE a (filterG x xs))`.

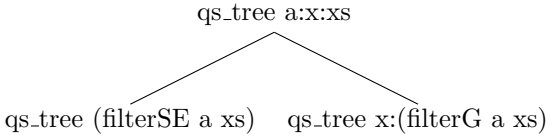
We will prove goal 1' using a case distinction. We distinguish two cases $x \leq a$ and $x > a$.

In the first case $x \leq a$ we know that `(filterSE a x:xs) = x:filterSE xs` so we can apply the constructor in goal 1' to obtain proof goals 1a and 1b.
1a : `qs_tree (filterSE x (filterSE a xs))`
1b : `qs_tree (filterG x (filterSE a xs))`.



Trivially `filter1 (filter2 l) = filter2 (filter1 l)` so both proof goals hold by $IH1, IH2$.

The second case is $x > a$ then we need to prove: 1': `qs_tree (filterSE a x:xs)` exists. Since $x > a$ this is equivalent to: 1": `qs_tree (filterSE a xs)`



Since $x > a$ We know that for any number n we have $(x \leq n) \Rightarrow (a \leq n)$ so we can remove one filter from `IH1`. This gives us IH1' = `qs_tree (filterSE a xs)` which is what we want to prove. This concludes our proof of proof obligation 1. The proof of proof obligation 2 is similar and will therefore be skipped.

3.2 Invariants Dijkstra

As described above Dijkstra defines quickSort using two invariants. We state that an advantages of his approach is that these invariants are easier to prove then the lemma `create_qs_tree`. The invariants can indeed easily be derived using the `inversion` tactic.

```
Lemma qs_acc_inv_1_0'' : forall (l:list nat)(x: nat)(xs : list nat),
qs_tree l -> l = cons x xs -> qs_tree (filter (fun y => leb y  x) xs).
intros l x xs H.
inversion H.
congruence.
intro H3.
inversion H3.
rewrite H5 in H0.
rewrite H6 in H0.
exact H0.
Defined.

Lemma qs_acc_inv_1_1'' : forall (l:list nat)(x: nat)(xs : list nat),
qs_tree l -> l = cons x xs -> qs_tree (filter (fun y => negb (leb y x)) xs).
intros l x xs H.
inversion H.
congruence.
intro H3.
inversion H3.
rewrite H5 in H1.
rewrite H6 in H1.
exact H1.
Defined.
```

3.3 Quicksort maintains elements

We have proven the lemmas needed to define quicksort. Next we will prove correctness. We use the definition of quicksort using `create_qs_tree`. First we want to prove the lemma:

```
Lemma qs_In_equiv : forall (xs : list nat) (x : nat),
  (In x (quickSort xs)) <-> ( In x xs).
```

To prove this we have to prove the lemmas:

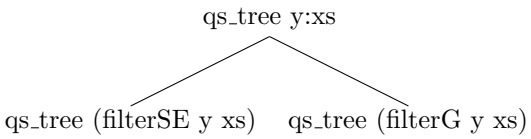
```
Lemma qs_contains : forall (xs : list nat) (x : nat),
  (In x xs) -> (In x (quickSort xs)).

Lemma qs_maintains : forall (xs : list nat) (x : nat),
  (In x (quickSort xs)) -> (In x xs)  .
```

We first prove `qs_contains`. We start by unfolding the definition of `quickSort`. This gives us access to the recursion tree. Using induction on this recursion tree proof is straightforward. If the recursion tree is empty then the list is empty and the proof trivially holds. If the recursion tree is not empty then we have to prove:

```
    In x (qs_helper (filter (fun y : nat => y <=? x0) xs) q1 ++
        y :: qs_helper (filter (fun y : nat => negb (y <=? x0)) xs) q2)
```

Where we have induction hypothesises:

```
IHq1 : In x (filter (fun y : nat => y <=? x0) xs) ->
   In x (qs_helper (filter (fun y : nat => y <=? x0) xs) q1)
IHq2 : In x (filter (fun y : nat => negb (y <=? x0)) xs) ->
   In x (qs_helper (filter (fun y : nat => negb (y <=? x0)) xs) q2)}
```



Since we know `In x x0:xs` we can use a case distinction `x=x0, In x xs`. Both these cases are trivial. The proof the other for `qs_maintains` is similar.

3.4 Quicksort sorted

Lastly we will prove the quicksort returns only sorted arrays. We want to prove the lemma:

```
qs_sorted : forall (xs: list nat),
  Sorted le (quickSort xs).
```

Here sorted is defined in the Coq standard library. Once again we unfold quicksort and we apply the induction hypothesis on the recursion tree. The base case immediately follows since an empty array is always sorted.

The step requires us to prove `Sorted le (qs_helper (x ::xs) (qs_tree_step x xs q1 q2))`. Which is:

```
Sorted le
  (qs_helper (filter (fun y : nat => y <=? x) xs) q1 ++
   x :: qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)
```

Furthermore we have induction hypothesis:

```
IHq1 : Sorted le (qs_helper (filter (fun y : nat => y <=? x) xs) q1)
IHq2 : Sorted le (qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)
```

Here `q1` is of type `qs_tree (filter (fun y : nat => y <=? x) xs)` and `q2` is of type `qs_tree (filter (fun y : nat => negb (y <=? x)) xs)`. Note that we only know the type of `q1,q2` we don't know the exact values. As mentioned before this is a weakness of our definition of quicksort. Our lemma create_qs_tree can create elements of type `q1 : qs_helper l` for any list `l`, however we don't know that if an element of type `qs_helper l` is equal to `create_qs_tree l`. Because of this we cannot fold our definition of quicksort back after using simple. We cannot rewrite our goal as:

```
Sorted le
  (quicksort (filter (fun y : nat => y <=? x) xs)) ++
   x :: (quicksort (filter (fun y : nat => negb (y <=? x)) xs))
```

This is unfortunate since we cannot use facts we have proven for quicksort in our proof. However we can use facts we prove about `qs_helper`.

To continue with our proof we first prove an helper lemma:

```
Lemma sorted_comb : forall (xls: list nat) (xrs: list nat),
  (Sorted le xls) -> (Sorted le xrs) -> (forall (x:nat), (In x xls) ->
    (HdRel le x xrs)) -> Sorted le (xls ++ xrs).
```

This lemma states that if we have two sorted arrays `xls, xrs` and we know that `xrs` is not empty implies that any element in `xls` is smaller then the head of `xrs` then we can conclude that `xls++xrs` is sorted. This proof follows directly from the definition of sorted if we use induction on `xls`. We will not explain this proof in more detail.

Now we can continue to prove our goal. We apply the lemma `sorted_comb` with
`xls = (quicksort (filter (fun y : nat => y <=? x) xs))` and

`xrs = x ::(quicksort (filter (fun y : nat => negb (y <=? x)) xs))`. If the preconditions of the lemma `sorted_comb` hold then we have finished our proof. We obtain 3 proof obligations:
1: `Sorted le (qs_helper (filter (fun y : nat => y <=? x) xs) q1)`
2: `Sorted le (x :: qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)`
3:
```
forall x0 : nat,
  In x0 (qs_helper (filter (fun y : nat => y <=? x) xs) q1) ->
  HdRel le x0 (x :: qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)
```

Proof obligation 1 holds directly by IH1. Proof obligation 2 is more complex. Using the constructor of sorted we gain 2 new proof obligations:
2a : `Sorted le (qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)`
2b : `HdRel le x (qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)`

Proof obligation 2a holds by IH2 which leaves proof obligation 2b. Note that `HdRel le x ys` is defined in the standard library and `HdRel` means that `ys` has a head `y` implies `le x y`. We can easily prove that the following lemma holds:

```
Lemma HdRel_redef : forall (xs : list nat) (a : nat),
  (forall (x:nat), (In x xs) -> (le a x)) -> (HdRel le a xs).
```

This lemma is useful since it is easier to reason about all elements an array then to reason about a head which might or might not exist. Using `HdRel_redef` on our goal `2b` we gain the hypothesis:
H : `In x0 (qs_helper (filter (fun y : nat => negb (y <=? x)) xs) q2)`
and proof obligation:
2b' : `x <= x0`

If we could apply the lemma `qs_In_equiv` on hypothesis H then by the definition of `filter` the proof would follow naturally. However as mentioned above we cannot use the fact that
`q2 = create_qs_tree (filter (fun y : nat => negb (y <=? x))` so we will need to redo the proof of `qs_In_equiv` for `quicksort_helper`. We redo a slightly weaker version `qs_In_equiv`. We will prove:

```
Lemma qs_helper_maintains : forall (xs : list nat) (x : nat) (conc : qs_tree xs),
  (In x (qs_helper xs conc))-> (In x xs).
```

The proof itself doesn't change at all from the proof of `qs_maintains`. After we have proven `qs_maintains` proof obligation 2b' quickly follows. The proof for prove obligation 3 is the same as the proof of proof obligation 2b and will be skipped. This covers all proof obligations and concludes our proof.

4 Conclusion

In this report we have implemented and validated the quicksort algorithm using Coq. We have discussed the problems which are encountered when implementing quicksort and their solutions. Furthermore we have discussed two approaches to implement quicksort in Coq each with their own up and downsides. These approaches can also be used to validate other structural recursive algorithms. The approach used in our definition of quicksort is useful if one wants to gain more insight in validating structural recursive algorithms using Coq. However if one wants to validate more difficult algorithms this approach will be cumbersome. Dijkstras approach offers an alternative which feels less natural but makes the validation easier. The Coq code used in this report can be found in the following git repository. The Coq code has been written in version 8.9.0 of Coq.

References

[1] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[2] Gabe Dijkstra. Experimentation project report: Translating haskell programs to coq programs. 2012.