15-414: Bug Catching: Automated Program Verification

Lecture Notes on Data Structures

Ruben Martins[*]

Carnegie Mellon University

Lecture 3
January 23, 2024

1 Introduction

Our study of logical contracts so far has focused on *control structures*: How do functions, assignments, and loops give rise to verification conditions that entail the correctness of the code? In the next example we will introduce two elementary data structures (lists and queues) and verify an implementation of queues using two lists. We then deepen our investigation into how to reason about data structures. A key concept that we need to capture is that of a *data structure invariant*. With executable contracts, such invariants are checked by functions. With logical contracts, they are expressed as logical properties of the data structures. We can classify data structures as *persistent* (also called *immutable*) and *ephemeral* (also called *mutable*). Persistent data structures are prevalent in functional programming, while ephemeral data structure are more common in imperative programming. Since WhyML covers a spectrum of functional and imperative programming, we will consider both and identify some commonalities and differences.

Learning goals. After this lecture, you should be able to:

- Formulate and verify non-executable specifications, while understanding some of their pitfalls
- Write logical contracts for small functional programs over simple immutable data types such as lists

[*]Based on notes written by Frank Pfenning in Spring 2022. This lecture was given by Tony Yu.

- Verify simple code over (inductive) data structure
- Learn to formulate data structure invariants, and how work with them in Why3

2 Nonexecutable Specifications

The specification we used in our verification of the mystery function was *executable*:

```
let rec function fib (n:int) : int =
requires { n >= 0 }
variant { n }
if n = 0 then 0
else if n = 1 then 1
else fib (n-2) + fib (n-1)
```

We draw your attention here to three keywords: `let rec` means that we are defining and executable, recursive function fib. The additional keyword `function` means that we can use fib in contracts to reason about code. This requires the fib function to be *pure* (have no effects, just return a value) and *terminating*. Purity is established simply by traversing the code, while termination comes from proving the verification condition for `variant { n }`.

In practice it is mostly the case that our specifications are *not executable* but *purely logical formulas*. That's because they are intended to express what the function accomplishes, but abstract away from how. We will see a number of ways to express these logical specifications. As a first example, we can express the defining property of the function fib rather than giving its explicit definition.

We start with declaring fib to be a function from integers to integers and then describe three properties as *axioms*.

```
function fib (n:int) : int
axiom fib0 : fib 0 = 0
axiom fib1 : fib 1 = 1
axiom fib2 : forall n:int. fib (n+2) = fib (n+1) + fib n
```

Describing the desired properties by axioms is very elegant, and can be used directly for logical reasoning about the code. For example, if we swap out the recursive definition of fib with the above, our mystery function f will still be verified just as easily (and possibly more easily). You can experiment yourself with the file mystery.mlw.

On the other hand, it also has a very serious danger: if we aren't careful, our axioms could be *inconsistent* by which we mean they derive a contradiction. If we have a contradiction we can conclude anything from that (because we are in an impossible situation), including any verification condition. As an example, let's say we made a typo and wrote `fib 0 = 1` in the second line, which would imply the contradictory `0 = 1`. We also change the postcondition to wrongly claim `result = fib (n-1)`.

```
function fib (n:int) : int
axiom fib0 : fib 0 = 0
axiom fib1 : fib 0 = 1 (* bug here!! *)
axiom fib2 : forall n:int. fib (n+2) = fib (n+1) + fib n
```

```

let f (n:int) : int =
requires { n >= 0 }
ensures { result = fib (n-1) } (* bug here!! *)
let ref i = 0 in
let ref a = 0 in
let ref b = 1 in
while (i < n) do
  invariant { 0 <= i <= n }
  invariant { a = fib i /\ b = fib (i+1) }
  variant { n-i }
  b <- a + b ;
  a <- b - a ;
  i <- i + 1
done ;
assert { i = n /\ a = fib i } ;
a
```

Because the axioms are inconsistent, the verification still succeeds!

```
% why3 prove -P alt-ergo mystery.mlw
File mystery.mlw:
Verification condition f'vc.
Prover result is: Valid (0.00s, 2 steps).
```

We can also prove it through the IDE and then replay the session:

```
% why3 ide mystery.mlw
# (prove and save session)
% why3 replay mystery
 1/1 (replay OK)
%
```

As a reminder, our contracts are blatantly incorrect! Because this is a serious problem, Why3 offers the possibility to search for inconsistencies among the axioms using the `--smoke-detector` option to the `replay` command.

```
% why3 replay --smoke-detector mystery
1/1 (replay failed)
goal 'f'vc.0', prover 'CVC4 1.7': result is: Valid
(0.02s, 1949 steps) -> Smoke detected!
Replay failed.
```

When reviewing your assignments, we will run the smoke detector, and you should do the same on your session files before you hand in homework assignments. While in this particular example the error was obvious, when specifications become more complex it is quite easy to introduce subtle errors into the specification. You also need to keep in mind that the smoke detector may not always work (the inconsistency is too hard to prove), and that errors in your axioms may render them incorrect without being inconsistent. You should always be careful when using axioms since inconsistent axioms will let you verify any result. Therefore, we recommend using axioms only when necessary.

3 Verifying Properties of Data Structures

For the moment, we'll stay in the functional world, although in WhyML we use data structures similarly also in imperative programs. There is a clever implementation of queues in a functional language using two stacks (usually directly represented by lists), sometimes called *functional queues*. If the queue is used in a single-threaded way (that is, we don't dequeue from a queue in the same state more than once) then amortized analysis shows that both enqueue and dequeue operations have constant amortized cost.

The basic algorithmic idea is as follows. The queue is represented by two lists, the front and the back. Initially both are empty. When we enqueue, we add elements to the back, and when we dequeue, we take them from the front. If the front happens to be empty when a dequeue request comes in we *reverse* the back to become the new front. For example, after enqueuing 1, 2, 3 in that order, the front is still empty and the back will be the list $[3, 2, 1]$. If we now dequeue, the front will become $[1, 2, 3]$ (the reverse of the back) and we remove 1 from the front, leaving it $[2, 3]$ with the back empty.

To represent the queue, we see a couple of new constructs. One of them is *polymorphism* because we would like queues to be generic in the types of the elements. In particular, the type `queue 'a` will be a queue with elements of type `'a` (usually pronounced *alpha*). We also need lists, which we can find in the Why3 Standard Library. Among many other things, we find:

```
type list 'a = Nil | Cons 'a (list 'a)

let rec function (++) (l1 l2: list 'a) : list 'a =
  match l1 with
  | Nil          -> l2
  | Cons x1 r1 -> Cons x1 (r1 ++ l2)

let rec function reverse (l: list 'a) : list 'a =
  match l with
  | Nil       -> Nil
  | Cons x r -> reverse r ++ Cons x Nil
  end
```

These functions are declared with `let rec function` which means they can be used in contracts as well as computationally. We do note that, computationally, we may want to use a different function that this particular `reverse` because its complexity is $O(n^2)$ for a list of length n. We'll note that, but we won't worry about it in today's lecture.

We see that lists have constructors `Nil` and `Cons` and that we discriminate between lists using the expression `match ... with ... end`. We use a *record* of two elements, the front and the back, as our representation of queues.

```
type queue 'a = { front : list 'a ; back : list 'a }
```

We would like to define the following functions

```
empty () : queue 'a
enq (x : 'a) (q : queue 'a) : queue 'a
deq (q : queue a) : option ('a , queue 'a)
```

A queue might be empty, so deq returns and optional pair consisting of the first element and the remainder of the queue. For this we need the option library:

```
type option 'a = None | Some 'a
```

Before we write code, we should decide on the specifications. The key idea is that we use a single list to represent the queue, with the first element in the queue at the front of the list. In other words, we use another data structure (a list) in the specification of the behavior of the queue. Of course, we do not want to use such a list as an *implementation* because the cost of an enqueue operation would be linear in the size of the queue (rather than have amortized constant cost). Therefore we define the function sequence that represents the state of a queue in the proper sequence. Recall that because we add the elements to the back, to represent the proper state of the queue we have to *reverse* the back.

```
function sequence (q : queue 'a) : list 'a = q.front ++ reverse q.back
```

Now let's write the postconditions for all of the functions. They have no precondition since any state of the queue is valid. The ones for empty and enq are fairly straightforward. For the enqueue operation we add the new element to the end of the sequence.

```
let empty () : queue 'a =
ensures { sequence result = Nil }
...

let enq (x : 'a) (q : queue 'a) : queue 'a =
ensures { sequence result = sequence q ++ Cons x Nil }
...

let deq (q : queue 'a) : option ('a , queue 'a) =
```

For the dequeue operation, we have to return Nil if the queue is empty and Some (x, r) if x is at the front of the queue and r is the remainder. Writing this out logically, we use an existential quantifier.

```
let deq (q : queue 'a) : option ('a , queue 'a) =
ensures { (result = None /\ sequence q = Nil)
          \/ ( exists x:'a. exists r:queue 'a. result = Some(x,r)
               /\ sequence q = Cons x (sequence r)) }
```

At this point the code itself is not too difficult, just for the case of enqueue we nest two matches because if the front is empty the queue is empty only if the back is also empty. We show the code here; the live-coded version which may be different in minor details can be found in queue.mlw.

```
module Queue

  use int.Int
  use list.List
  use list.Append
  use list.Reverse
  use option.Option

  (* type list 'a = Nil | Cons 'a (list 'a) *)
  (* type option 'a = None | Some 'a *)

  type queue 'a = { front : list 'a ; back : list 'a }

  function sequence (q : queue 'a) : list 'a = q.front ++ reverse q.
      back

  let empty () : queue 'a =
  ensures { sequence result = Nil }
  { front = Nil ; back = Nil }

  let enq (x : 'a) (q : queue 'a) : queue 'a =
  ensures { sequence result = sequence q ++ (Cons x Nil) }
  { front = q.front ; back = Cons x q.back }

  let deq (q : queue 'a) : option ('a , queue 'a) =
  ensures { (sequence q = Nil /\ result = None)
            \/ (exists x:'a, q':queue 'a. sequence q = Cons x (
                sequence q') /\ result = Some (x,q')) }
  match q.front with
  | Nil -> match reverse q.back with
           | Nil -> None
           | Cons y ys -> Some (y, { front = ys ; back = Nil })
           end
  | Cons x xs -> Some (x, { front = xs; back = q.back })
  end

end
```

While the code and specifications seem correct, we cannot be 100% confident that the prover will be able to verify it. In particular, the reasoning depends on the lemmas in the libraries pertaining to the properties of append ('++') and reverse. Fortunately, in this case it succeeds. Just for variety, we tried it with the CVC4 prover.

```
% why3 prove -P cvc4 queue.mlw
queue.mlw Queue empty'vc: Valid (0.04s, 6867 steps)
queue.mlw Queue enq'vc: Valid (0.05s, 7687 steps)
queue.mlw Queue deq'vc: Valid (0.08s, 13374 steps)
%
```

4 Data Structure Invariants

As a simple example of data structure invariants we reconsider our implementation of queues. We would like to extend the interface with a function `qsize` that returns the number of elements in a queue. The obvious way to compute this would be to compute the lengths of the front and back and add them, but the complexity would be linear in the size of the queue. To answer this query in constant time we add a `size` field to the queue and maintain it as we enqueue or dequeue elements. The data structure invariant here is that the size field always contains the sum of the lengths of the front and back.

```
type queue 'a = { front : list 'a ;
                  back : list 'a ;
                  size : int }
invariant { size = length front + length back }
```

In the logic, the verifier assumes the invariant when reasoning about queues. This could create a logical inconsistency (*everything is provable!*) if the invariant is unsatisfiable. For example, the trivial invariant *false* means any program in the scope of this declaration could now be verified. To avoid this, Why3 will try to prove that there exists an instance of the data structure for which the invariant is satisfied. For complex invariants this can be difficult, so WhyML gives you a way to specify an instance of the data structure satisfying the invariant with a '`by`' clause. In this particular case it would be unnecessary, but it is good practice to always supply it.

```
type queue 'a = { front : list 'a ;
                  back : list 'a ;
                  size : int }
invariant { size = length front + length back }
by { front = Nil ; back = Nil ; size = 0 }
```

In this example it is relatively easy to update the code to maintain the size field and Why3 will prove that it is always correct. In constructing the verification condition we may think of the invariant as being *assumed* at the beginning of a function (like a precondition) and *proved* at the end of a function (like a postcondition). In between, the invariant may be violated, although this possibility does not come into play here. It is necessary because you may build or modify an element of the data structure incrementally and only at the end does the invariant hold.

The `qsize` function just returns the size field. In addition, the postcondition certifies that it is indeed the length of the queue (when viewed as a single sequence of element, which represents the client's perspective.

```
let qsize (q : queue 'a) : int =
ensures { result = length (sequence q) }
q.size
```