15-414: Bug Catching: Automated Program Verification

Lecture Notes on Data Structures

Matt Fredrikson*

Carnegie Mellon University
Lecture 3
January 24, 2023

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.

We will also take a look at testing, why it is still important, and briefly consider how to use the Why3 IDE during code development.

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

- Formulate and verify non-executable specifications, while understanding some of their pitfalls

*Closely adapted from notes written by Frank Pfenning in Spring 2022

- Write logical contracts for small functional programs over simple immutable data types such as lists
- Verify simple code over (inductive) data structure

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!! *)
```

```
4   axiom fib2 : forall n:int. fib (n+2) = fib (n+1) + fib n
5
6   let f (n:int) : int =
7   requires { n >= 0 }
8   ensures { result = fib (n-1) } (* bug here!! *)
9   let ref i = 0 in
10  let ref a = 0 in
11  let ref b = 1 in
12  while (i < n) do
13    invariant { 0 <= i <= n }
14    invariant { a = fib i /\ b = fib (i+1) }
15    variant { n-i }
16    b <- a + b ;
17    a <- b - a ;
18    i <- i + 1
19  done ;
20  assert { i = n /\ a = fib i } ;
21  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.
```

In the autograder we routinely 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.

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 Testing and Implicit Preconditions

In 1977 Donald Knuth famously wrote "*Beware of bugs in the above code; I have only proved it correct, not tried it.*" in a 5-page memo *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion.* You might think that if he had proved it using Why3 then he wouldn't have to be worried. I think he still would have been, and he should have been! Here are some of the things that can still go wrong even if Why3 says "Verified!".

- Your preconditions could be prohibitively strict, even to the point where *no client could possibly call your functions*.
- Your postconditions could be prohibitively lax, to the point where *the client obtains no information at all about the computation of your function.*
- Your definitions and axioms could be incorrect in the sense that they do not capture the property you were trying to prove. In the extreme case they could be *vacuous* (equivalent to true and therefore not saying anything) or *inconsistent* (equivalent to false and therefore implying everything whatsoever).
- There could be a bug in Why3 or one or more of the back-end provers, extracting an incorrect verification condition or proving one that isn't valid. In fact, it is almost certain that Why3 and all the back-end provers have bugs, so it is just a matter of probabilities whether you trip any of them.

You may think these are unlikely, but you should be prepared that almost certainly at least *some of these things will happen to you*.

When grading your homework, we combat these issues by combining manual inspection with replaying the Why3 sessions.

When you develop your code, you most likely will be in the *opposite* situation for a while: your code can not be verified. Then you have to look for the exact opposite of the points raised above, plus a very real first possibility:

- Your code is incorrect!
- Your preconditions could be prohibitively lax, to the point where they are too weak to imply loop invariants or preconditions for operations or function calls, or the postcondition at the end of the function. Of course, this may be the case even if your code is correct!
- Your postconditions could be prohibitively strict, to the point where they simply do not follow from what you know at the end of the function. Again, this may be the case even with correct code.
- Your definitions and axioms do not properly capture the property you wish to prove (and are convinced is true).

- The back-end provers in Why3 are not strong enough to prove the verification condition *even though it is true* and, on top of it, *your code is correct*.

To mitigate all these issues, it is sensible to combine testing with verification even during the development process. Among other things:

- It may help you to determine whether your code is correct and, if not, have some counterexamples. Unfortunately, today's technology is such that it is difficult to obtain counterexamples from failing provers. A reasonable set of successful test cases may point you towards other kinds of issues you might have.
- If you run Why3 on your program including the testing code it may help you uncover situations where the preconditions are too strict. That could mean your test function will fail to verify, even if Why3 assumes all the pre- and post-conditions in the rest of your program.
- It may help you to think about the code by writing out explicitly how it should behave on specific examples.

Even though our code has been verified, let's write a little test function. By convention, a test function should take unit as an argument, because the `why3 execute` command will pass the unit element to the specified function.

```
let test () : (int, int) =
let q0 = empty () in
let q1 = enq 1 q0 in
let q2 = enq 2 q1 in
match deq q2 with
| Some (x, q3) -> match deq q3 with
| Some (y, q4) -> match deq q4 with
| None -> (x,y)
end end end
```

We should be able to execute this with the command

```
% why3 execute queue.mlw --use=Queue 'test ()'
```

but unfortunately due to limitation (or bug) in the implementation of the WhyML interpreter related to polymorphism this fails. It is nevertheless interesting because in the first two match expressions above we only supply the branch for `Some`. When we dequeue from `q2` it is impossible for the queue to be empty, so the value returned cannot be `None`. When Why3 encounters a `match` expression that does not cover all the possible cases based only on the type information, it generates a verification condition to *prove* that the omitted cases are impossible. In this example, by tracking the postconditions, the verifier should know precisely that the sequence at the point where `deq q2` is called is 1, 2 and therefore the queue nonempty.

We can actually test this: let's add the postcondition that the answer *must be* `(1,2)` and run the alt-ergo theorem prover and the file with the additional, strict function `test`.

```
% why3 prove -P alt-ergo queue.mlw
queue.mlw Queue empty'vc: Valid (0.01s, 26 steps)
queue.mlw Queue enq'vc: Valid (0.02s, 123 steps)
queue.mlw Queue deq'vc: Valid (0.28s, 908 steps)
queue.mlw Queue qsize'vc: Valid (0.00s, 10 steps)
queue.mlw Queue test'vc: Timeout (5.00s)
%
```

Hmmm, it doesn't seem to be able to prove our test function. Suspecting that the code and contracts are correct and the theorem prover is too weak, let's try CVC4.

```
% why3 prove -P cvc4 queue.mlw
queue.mlw Queue queue'vc: Valid (0.02s, 4244 steps)
queue.mlw Queue empty'vc: Valid (0.04s, 6706 steps)
queue.mlw Queue enq'vc: Valid (0.06s, 9124 steps)
queue.mlw Queue deq'vc: Valid (0.19s, 25682 steps)
queue.mlw Queue qsize'vc: Timeout (5.00s, 2064194 steps)
queue.mlw Queue test'vc: Valid (0.43s, 57994 steps)
%
```

Although it is somewhat slow and take a lot of steps, CVC4 manages to verify that `test ()` must return `(1,2)`! However, for some strange reason it fails to verify the very simple `qsize` function which was no problem at all for alt-ergo.

At this point we can invoke

```
% why3 ide queue.mlw
```

and launch strategy `Auto level 2` (or keyboard shortcut 2) which tries different provers on different subgoals and quickly verifies the code. If we save the session under the file menu (or keyboard shortcut Control-S) we can examine the session statistics.

```
% why3 session info --stats queue
== Number of root goals ==
  total: 6  proved: 6

== Number of sub goals ==
  total: 0  proved: 0

== Goals not proved ==

== Goals proved by only one prover ==
  +-- file [../queue.mlw]
    +-- theory Queue
      +-- goal queue'vc: CVC4 1.7
      +-- goal empty'vc: CVC4 1.7
      +-- goal enq'vc: CVC4 1.7
      +-- goal deq'vc: CVC4 1.7
      +-- goal qsize'vc: Alt-Ergo 2.3.1
      +-- goal test'vc: CVC4 1.7
```

```
== Statistics per prover: number of proofs, time (minimum/maximum/average) in seconds ==
  Alt-Ergo 2.3.1      :   1   0.00   0.00   0.00
  CVC4 1.7            :   5   0.02   0.46   0.16

%
```

One lesson from this example is that certain constructs have *implicit preconditions*. This includes pattern matches against data types (they should exhaustive), calls to functions such as `div` or `mod` (the second argument should not be zero), array accesses (the index should be in bounds; not yet discussed), and functions passing data structures endowed with invariants (they should hold).