15-414: Bug Catching: Automated Program Verification

Lecture Notes on Data Structures

Frank Pfenning

Carnegie Mellon University
Lecture 3
February 9, 2021

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 last example we introduced two elementary data structures (lists and queues) and verified an implementation of queues using two lists. In this lecture we 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:

- Exploit representation invariants in verification;
- Combine verification with testing;
- Use the Why3 IDE;
- Verify code using mutable data structures such as arrays.

2 Data Structure Invariants

As a simple example of data structure invariants we reconsider our implementation of queues.

```
module Queue

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

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

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

  let empty () =
  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 { (result = None /\ sequence q = Nil)
            \/ ( exists x:'a. exists r:queue 'a. result = Some(x,r)
                  /\ sequence q = Cons x (sequence r)) }
  match q.front with
  | Nil -> match reverse q.back with
           | Nil -> None
           | Cons x b -> Some (x, { front = b ; back = Nil })
           end
  | Cons x f -> Some (x, { front = f ; back = q.back })
  end

end
```

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
```

You can find the other functions in the file queue.mlw.

3 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 () =
let q0 = empty () in
let q1 = enq 1 q0 in
let q2 = enq 2 q1 in
let q3 = enq 3 q2 in
match deq q3 with Some (x1, q4) ->
match deq q4 with Some (x2, q5) ->
match deq q5 with Some (x3, q6) ->
match deq q6 with None -> (x1, x2, x3)
end end end end
```

and we get (note the necessary module qualifier `Queue.`):

```
% why3 execute queue.mlw Queue.test
Execution of Queue.test ():
     type: (int, int, int)
   result: (1, 2, 3)
  globals:
%
```

Fortunately, this is the expected answer because we enqueue 1, 2, 3 and then dequeue three elements and return a tuple consisting of them. An interesting aspect of the function `test` is that the patterns it matches against are not exhaustive. When we dequeue from `q3` 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 q3` is called is 1, 2, 3 and therefore the queue nonempty.

We can actually test this: let's add the postcondition that the answer *must be* `(1,2,3)` 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 queue'vc: Valid (0.00s, 0 steps)
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 `(1,2,3)`! 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 Ctrl-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 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).

4 Arrays

Because of their efficiency arrays are a common data structure in imperative programs. Reasoning about arrays requires a number of techniques due to their inherent mutability and range requirements for array access.

We consider a key/value store in a simple array, usually not the preferred choice due to its fixed size. We use integers as keys and data are of arbitrary type. Because integers represent both keys and array indices, we introduce a new type alias `key`. A key/value pair is just a record with a `key` field and a `data` field. The complete live code for this example can be found in the file search.mlw.

```
module SearchArray
  use int.Int
  use array.Array
  use array.ArrayEq

  type key = int  (* transparent type alias *)
  type keyval 'a = { key : key ; data : 'a }

  ...
end
```

We start with a search function that does not modify the array and returns an index of the entry matching the given key, or -1 if no such entry exists. We start:

```
  let search (k : key) (a : array (keyval 'a)) : int =
  ensures { (result = -1 /\ not (defined k a))
            \/ a[result].key = k }
  ensures { a = old a }
```

The second postcondition uses the keyword `old` to indicate that the state of a (which, as an array is mutable) at the end of the function is the same as the state of a at the beginning of the function (expressed as `old a`).

In the first postcondition, it remains implicit that $0 \leq \mathsf{result} < a.\mathsf{length}$, because if result were not in the range we definitely would not be able prove that $a[\mathsf{result}].key = k$. Generally, we might prefer to make this explicit. Second, we note the predicate defined k a which should be true if key k is defined *somewhere* in the array a. So we specify before the function `search`:

```
  predicate defined (k : key) (a : array (keyval 'a)) =
  exists i:int. 0 <= i < a.length /\ a[i].key = k
```

It is important to think *logically* rather than *operationally* when defining predicates, to be used only in contracts. In particular, we use quantifiers rather than iteration or recursion.

Next we write the body of the function, at first without loop invariants. An array is actually a record, where `.length` returns its length (which is an immutable field). We use n as an abbreviation for $a.\mathsf{length}$ and then iterate through the array until either we reach the end or we find an element matching the given key.

```
  let search (k : key) (a : array (keyval 'a)) : int =
```

```
  ensures { (result = -1 /\ not (defined k a))
            \/ a[result].key = k }
  ensures { a = old a }
  let n = a.length in
  let ref i = 0 in
  while i < n && a[i].key <> k do
    i <- i + 1
  done ;
  if i = n then -1 else i
```

Note that in a computational context we use `&&` for conjunction, which is short-circuiting. That's important here because if we reach the end of the array we have $i = n$ so the access $a[i]$ would otherwise be out of bounds.

The next questions are the loop invariants and the loop variant. In fact, the variant is easy: we increment i which is bounded by n above, so $n - i$ is the variant. We also record $0 \leq i \leq n$ as an invariant, which is mechanical for this kind of loop.

```
let search (k : key) (a : array (keyval 'a)) : int =
ensures { (result = -1 /\ not (defined k a))
          \/ a[result].key = k }
ensures { a = old a }
let n = a.length in
let ref i = 0 in
while i < n && a[i].key <> k do
  invariant { 0 <= i <= n }
  invariant { ... }
  variant { n - i}
  i <- i + 1
done ;
if i = n then -1 else i
```

We have left room for a second invariant which is central for the correctness of this function. We have to express that all the elements we have already scanned have a key different from k. One way to express this would be to generalize the predicate `defined` to take an upper bound. Instead, we just express it here in line using quantification.

```
let search (k : key) (a : array (keyval 'a)) : int =
ensures { (result = -1 /\ not (defined k a))
          \/ a[result].key = k }
ensures { a = old a }
let n = a.length in
let ref i = 0 in
while i < n && a[i].key <> k do
  invariant { 0 <= i <= n }
  invariant { forall j. 0 <= j < i -> a[j].key <> k } }
  variant { n - i}
  i <- i + 1
done ;
if i = n then -1 else i
```

This function can now be verified because at the end of the loop either $i = n$ (in which case the second loop invariant tells us that the key k is not in the array) or $i < n$, in which case $a[i].\mathsf{key} = k$ and we can return i.

5 Mutating Arrays

Searching through an array has the special property that we do not modify it. When we modify an array as we traverse it the loop invariant generally has to be more complicated. This is because with any assignment to an array element inside a loop, we lose all information about what *any* element in the array is. Therefore, we generally need to specify the entries of the array we change and how, and in addition that the remaining entries do not change.

As an example, we consider a function to negate every element in a key/value array of integers. In order to verify this, we define a predicate

```
predicate negated (a : array (keyval int)) (b : array (keyval int))
    (i : int) =
```

which is true if the elements of $a[0..i)$ are the negated versions of the corresponding elements in b. In its definition we just need to be careful that the arrays a and b have the same length and that the index i is in bounds.

```
predicate negated (a : array (keyval int)) (b : array (keyval int))
    (i : int) =
a.length = b.length /\ 0 <= i <= a.length
/\ forall j. 0 <= j < i -> a[j].key = b[j].key
                         /\ a[j].data = -b[j].data
```

The postcondition for out negation function now expresses that the state of the array upon the return is equal to the old value of the array all the way up to the last element.

```
let negate (a : array (keyval int)) : unit =
ensures { negated a (old a) a.length }
for i = 0 to a.length-1 do
   a[i] <- { key = a[i].key ; data = -a[i].data }
done ;
()
```

This code has two additional new constructs. We use the type `unit` (whose only element is `()`) to express that the function returns no interesting value. This usually implies that it mutates some of its arguments, in this case the array a.

We also use a `for` loop, of the form for $i = lower$ to $upper$ do ... done for which we give *inclusive bounds*. It generates suitable loop invariants for the index $lower \leq i \leq upper + 1$ and a variant of $upper + 1 - lower$. There is an analogous form for $i = upper$ downto $lower$ do ... done.

What remains is to state the invariants regarding the array. Intuitively, at iteration i we have negated all the elements up to i while the elements at indices greater or equal to i have remained unchanged. To specify this we find in the useful `array.ArrayEq` module the function

```
array_eq_sub (a : array 'a) (b : array 'a) (lower : int) (upper : int)
```

which is true if for all $lower \leq i < upper$ we have $a[i] = b[i]$. We use this where b is the original version of a.

```
let negate (a : array (keyval int)) : unit =
ensures { negated a (old a) a.length }
for i = 0 to a.length-1 do
   invariant { negated a (old a) i }
   invariant { array_eq_sub a (old a) i a.length }
   a[i] <- { key = a[i].key ; data = -a[i].data }
done ;
()
```

It is not necessary to explicitly return the unit element (since the for-loop already returns the unit element), but we write it out for emphasis.

The code now verifies, but true to our methodology we can also try it out.

```
let mk (k : key) (x : 'a) : keyval 'a = { key = k ; data = x }

let test () =
let a = Array.make 3 (mk 0 "") in
( a[0] <- mk 3 "c" ; a[1] <- mk 2 "b" ; a[2] <- mk 1 "a" ) ;
( search 2 a , search 4 a , search 1 a )
```

```
% why3 execute search.mlw SearchArray.test
Execution of SearchArray.test ():
     type: (int, int, int)
   result: (1, (-1), 2)
  globals:
%
```

This time, we would not be able to prove the expected outcome because the prover does not have enough information about the behavior of the function. Expressing such information in the form of *models* of the data structure will be part of the next lecture.