15-414: Bug Catching: Automated Program Verification

Lecture Notes on Arrays and Ghosts

Carnegie Mellon University
Lecture 4
September 4, 2025

1 Introduction

At this point we have experimented with simple imperative programs over integers using loops, recursive functions, and immutable data structures. Today, we start by looking at mutable data structures, specifically arrays. How do we specify and verify simple loops that operator on arrays? The key constructs we have (loop invariants and variants) are sufficient but become more complex because they have to express not only properties of what we *change* in an array, but also about the parts that *do not change*. It is this additional requirement that makes reasoning about ephemeral (mutable) data structures in many cases more difficult than reasoning about persistent (immutable) ones.

After we understand the basics, we consider a particularly important technique to tame the basics, namely simple *models* of complex data structures. Consider, for example, a red/black tree implementation of a map. To control its complexity we have introduced the concept of a data structure invariant. In the red/black tree example, this would include the *ordering invariant* (keys in left subtrees are all smaller and keys in the right subtree are all larger than the key in a node), the *color invariant* (there are no two adjacent red nodes in the tree), and the *black height invariant* (the number of black nodes on any path from a leaf to the root is the same). These are all *internal invariants* in the sense that the implementation of the data structure must maintain them but the client should only care that red/black trees provide a correct and efficient implementation of a map from keys to values. In this case, we say the map provides a *model* of the intended behavior of the data structure. We would like the model to be *logical* and as high-level as possible to support reasoning by the client. Maps and sets are common

models. In today's lecture we exemplify sets as a model of an ephemeral (mutable) implementation of bit vectors as arrays.

When implementing a data structure we have to maintain the correspondence between the low-level implementation and the high-level model. The purpose of the model is to *reason* about a data structure, but not to *compute* with it. So we would like to *erase* the code that maintains the model: actually computing it would negate all the advantages of the efficient implementation! This is the primary purpose of *ghosts*. They are pieces of code or data that exist solely for the purpose of verification and do not contribute to the outcome of the computation. This has to be checked by the verification engine. Ghost variables, or ghost fields of records, can only be used in other ghost computations. Otherwise, erasing them before the program is run would lead to incorrect code. This condition is related to the fact that *executable contracts* in C0 can not have any externally observable effects: running the program with or without executable contracts should yield the same answer (as long as all contracts are satisfied, of course).

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

- Verify simple imperative programs computing over arrays
- Model data structures using ghosts

2 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.

For the sake of simplicity, we consider a fixed, unordered array holding a finite set of integers. The complete live code for this example can be found in the file intset.mlw.

```
module IntSet

use int.Int
use ref.Ref
use array.Array
use option.Option

let search (x:int) (a : array int) : option int =
...

end (* module IntSet *)
```

The search function should not modify the array and returns the index of the entry matching the given key, or `None` if no such entry exists. We start:

```
predicate mem (x:int) (a:array int) (lower:int) (upper:int) =
  0 <= lower <= upper <= a.length /\
  exists i:int. lower <= i < upper /\ a[i] = x
```

```
4
5  let search (x:int) (a : array int) : option int =
6  ensures {
7    match result with
8    | None   -> not (mem x a 0 a.length)
9    | Some i -> 0 <= i < a.length /\ a[i] = x
10   end
11 }
12 ensures { a = (old a) }
```

The second postcondition uses the keyword `old` to indicate that the state of a (which 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`). If we do not say this the search function could internally modify the array and a client couldn't call this function and reason about its result.

In the first postcondition, we make the result of the function explicit: the result is an `option int` that is `Some i` if a matching index exists, or `None` if the value does not occur in the array. Note that we need to describe the bounds on `i`, as it is not possible to reason about whether `a[i] = x` unless `i` is a valid index into `a`.

We then iterate through the array until either we reach the end or we find an element matching the given key. If $i = a.\text{length}$ after the loop it means we have tried all elements in the array and not found x.

```
1  let search (x:int) (a : array int) : option int =
2  ensures {
3    match result with
4    | None   -> not (mem x a 0 a.length)
5    | Some i -> 0 <= i < a.length /\ a[i] = x
6    end
7  }
8  ensures { a = (old a) }
9  let ref i = 0 in
10 while i < a.length && a[i] <> x do
11   i <- i + 1
12 done ;
13 if i = a.length then None else Some 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 $i = a.\text{length}$ 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 $a.\text{length}$ above, so $a.\text{length} - i$ is the variant. We also record $0 \leq i \leq a.\text{length}$ as an invariant, which is mechanical for this kind of loop. Furthermore, we want to express that the array a does not change in the loop by stating that $a = (\text{old } a)$.

```
1  let search (x:int) (a : array int) : option int =
2  ensures {
3    match result with
4    | None   -> not (mem x a 0 a.length)
5    | Some i -> 0 <= i < a.length /\ a[i] = x
6    end
```

```
}
ensures { a = (old a) }
let ref i = 0 in
while i < a.length && a[i] <> x do
  variant { a.length - i }
  invariant { 0 <= i <= a.length }
  invariant { not (mem x a 0 i) }
  invariant { a = (old a) }
  i <- i + 1
done ;
assert { i = a.length \/ a[i] = x } ;
if i = a.length then None else Some i
```

This function can now be verified because at the end of the loop either $i = a.\text{length}$ (in which case the third loop invariant tells us that the value x is not in the array) or $i < a.\text{length}$, in which case $a[i] = x$ and we can return `Some i`.

3 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 may be. 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 the set. In order to verify this, we define a predicate negated. As is common (and perhaps we should have done this for the mem predicate) test the property for a slice of the array in the interval $[\text{lower}, \text{upper})$ (inclusive the lower bound and exclusive the upper bound).

```
predicate negated (a : array int) (b : array int) (lower : int) (upper
    : int) =
  a.length = b.length /\ 0 <= lower <= upper <= a.length /\
  forall j:int. lower <= j < upper -> a[j] = -b[j]
```

We also ensure that the arrays a and b have the same length.

The postcondition for our negation function expresses that the state of the array upon the return is equal to the negated initial values of the array all the way up to the last element.

```
let negate (a: array int) : unit =
ensures { negated a (old a) 0 a.length }
for i = 0 to a.length - 1 do
  (* no variant, or invariant on i needed *)
  invariant { ... }
  a[i] <- -a[i]
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 int) : unit =
ensures { negated a (old a) 0 a.length }
for i = 0 to a.length - 1 do
  (* no variant, or invariant on i needed *)
  invariant { negated a (old a) 0 i }
  invariant { array_eq_sub a (old a) i a.length }
  a[i] <- -a[i]
done ; ()
```

Observe how the invariants in this form express concisely that the values in the interval $[0, i)$ are negated, while those in the interval $[i, a.\mathsf{length})$ are still the same as in the original array. It is generally easier to understand and express the invariant in this form than using complicated quantified formulas in-line.

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 test () =
let a = Array.make 4 0 in
( a[0] <- 1 ; a[1] <- 27 ; a[2] <- 4 ; a[3] <- 3 ;
  (search 1 a , search 2 a , search 27 a ) )
```

We obtain the expected answer.

```
% why3 execute intset.mlw --use="IntSet" 'test ()'
result: (option int, option int, option int) = (Some 0, None, Some 1)
globals:
%
```

Instead of executing it, we can also *prove* that this must be the outcome of the function.

```
let test () =
ensures { result = (Some 0, None, Some 1) }
let a = Array.make 4 0 in
( a[0] <- 1 ; a[1] <- 27 ; a[2] <- 4 ; a[3] <- 3 ;
  (search 1 a , search 2 a , search 27 a ) )
```

```
% why3 prove -P alt-ergo intset.mlw
File intset-test.mlw:
Verification condition search'vc.
Prover result is: Valid (0.01s, 94 steps).

File intset-test.mlw:
Verification condition negate'vc.
Prover result is: Valid (0.03s, 383 steps).

File intset-test.mlw:
Verification condition test'vc.
Prover result is: Valid (0.03s, 160 steps).
```

4 Diagnosis of Failing Goals

We provide here a brief illustration how we can use the Why3 IDE to isolate failures of verification. We change the source so that the middle invariant

```
1   invariant { forall j. 0 <= j < i -> a[j] <> x }
```

has an off-by-one error

```
1   invariant { forall j. 0 <= j <= i -> a[j] <> x }
```

We start the Why3 IDE with

```
why3 ide intset.mlw
```

select the outermost goal and hit '2' for a relatively advanced strategy. It cannot prove the verification condition but splits it into several parts. We can see that many subgoals are checked as green, but two of them still have question marks. Goals depending on them higher up in the goal/subgoal tree will then be similarly marked. The unproven subgoals are labeled `[loop invariant init]` and `[loop invariant preservation]` which indicates that there is a loop invariant that cannot be established initially, nor can it be shown to be preserved. To see which one we select the second one in the pane on the left and examine the program in the pane on the right. The IDE will highlight in green the assumptions it uses and in yellow the proof goal it is trying to prove. Here we see it is `a[j] <> x` in the middle invariant. Even though it does not occur here, the IDE will highlight formulas in your program as red if it uses its negation in the proof attempt. This occurs for the else-branch of conditionals and loop guards when reasoning about the state after the loop is exited.

Highlighting the other unprovable subgoal (labeled `[loop invariant init]`) leads to the same culprit. Some further forensics and thought about this will hopefully reveal the bug at this point. Fortunately, it is not in the program but in the invariant.

5 Ghosts and Models

Before we introduce ghosts, recall how we specified queues in the previous lecture. There, the queue was implemented by two lists ('front' and 'back'), but we reasoned about its behavior using a *logical view* of the queue as a single list given by a `sequence` function:

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

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

Postconditions for operations such as `empty`, `enq`, and `deq` were phrased in terms of `sequence q`, not in terms of the two-list representation. This separation is the key idea: the client reasons about the *abstract* behavior (the sequence of elements), while the implementation is free to use an efficient representation.

This is what we mean by a *model*: a high-level, logical description of the data structure's state that we use in specifications. The model is not meant to be executed, and doing so would in most cases be very inefficient. The model exists to make reasoning easy and modular for the client of our code. For purely functional structures like the queue in the previous lecture, this is often straightforward. For mutable structures, we must relate the changing concrete state to a stable abstraction across updates.

To maintain such a model without having to actually compute it, we use *ghosts*: specification-only fields and computations that that allow us to refer to the model in contracts, and establish its consistency with the computational data structure. When the program is compiled into executable code, all of the ghost fields and computations are erased.

As an example, we use the standard set library to model a bit vector implementation of bounded finite sets. Here is an excerpt of the finite set module.

```
module Fset
  type fset 'a
  predicate mem (x: 'a) (s: fset 'a)
  predicate is_empty (s: fset 'a) = forall x: 'a. not (mem x s)
  constant empty: fset 'a
  function add (x: 'a) (s: fset 'a) : fset 'a
  axiom add_def: forall x: 'a, s: fset 'a, y: 'a.
      mem y (add x s) <-> (mem y s \/ y = x)
  function remove (x: 'a) (s: fset 'a) : fset 'a
  axiom remove_def: forall x: 'a, s: fset 'a, y: 'a.
      mem y (remove x s) <-> (mem y s /\ y <> x)
  ...
end
```

We start by defining the `Bitset` module by defining a `bset` as a record consisting of an array `a`, a `bound` and a ghost field called `model` containing a finite set of integers. Because bitsets are mutable (for example, we actually *change* a bset by adding an element to it), the `model` field must also be mutable.

```
type bset = { a : array bool ;
              bound : int ;
```

```
3                mutable ghost model : Fset.fset int }
4 invariant { 0 <= bound <= a.length
5             /\ forall j. 0 <= j < a.length -> a[j] <-> Fset.mem j
                  model }
6 by { a = Array.make 0 false ; bound = 0 ; model = Fset.empty }
```

The invariant states that element $a[i]$ of the array is true if and only if the number i is in the model set. We witness the existence of such a bset with the empty array and empty model. Note that the fields `a` and `bound` are immutable, although the elements in the array are mutable.

To create an empty bset we need a bound on the elements we may add to the set, which will be the length of the array.

```
1 let empty_bset (bound : int) : bset =
2 requires { bound >= 0 }
3 ensures { Fset.is_empty result.model /\ result.bound = bound }
4 { a = Array.make bound false ; bound = bound ; model = Fset.empty }
```

The model is just the empty set. Note that the postcondition states that `empty_bset` models the empty finite set.

To add an element i to a bset we just set the corresponding array element to `true` (whether it was already true or not). This requires the precondition that the i is in the permissible range. Because this operation is destructive, modifying the given bset, the postcondition needs to state the model *after* the update is equal to the model *before* the update, plus the element i. For this purpose we use again the `old` keyword to refer to the state of the model at the time the function is called.

```
1 let add_bset (x : int) (s : bset) : unit =
2 requires { 0 <= x < s.bound }
3 ensures { s.model = Fset.add x (old s).model }
4 s.a[x] <- true ;
5 ghost (s.model <- Fset.add x s.model) ;
6 ()
```

The assignment to `s.model` is enclosed in `ghost (...)` to be explicit that this update of the model will not be carried out if the program is executed. Instead it is there to maintain the data structure invariant which must be restored before the function `add_bset` exits. Just before this line, by the way, the data structure invariant is false because we have updated the array but not the model. We can see the significance of checking the data structure invariance exactly at function boundaries.

Our postcondition will allow the client to reason about the effects of it add operations. Note that the pre- and post-conditions *do not reference the array*, only properties of the model and the bound. The client can reason about the behavior of these functions without knowing the representation, using only the model.

The remove operation is entirely analogous.

```
1 let remove_bset (i : int) (s : bset) : unit =
2 requires { 0 <= i < s.bound }
3 ensures { s.model = Fset.remove i (old s).model }
4 s.a[i] <- false ;
5 ghost (s.model <- Fset.remove i s.model) ;
```

```
6 ()
```

We did not implement any more complex operations such as union or intersection, even though this would certainly be possible. You can find the live-code `Bitset` module in the file bitset.mlw.