15-414: Bug Catching: Automated Program Verification

Lecture Notes on Arrays and Ghosts

Ruben Martins[*]

Carnegie Mellon University

Lecture 4
January 25, 2024

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

[*]Extended from notes written by Frank Pfenning in Spring 2022

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
- Diagnosis of failing goal
- 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 array.Array

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

end (* module IntSet *)
```

The search function should not modify the array and returns the index of the entry matching the given key, or -1 if no such entry exists. We start:

```
let search (x:int) (a : array int) : int =
ensures { (0 <= result < a.length /\ a[result] = x)
          \/ (result = -1 /\ not (mem x a)) }
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. Note that this postcondition does not allow the array to be modified even if the contents remain unchanged. If we wanted to allow the array to be modified but to have the same contents then we should use the `array_eq` function in the module `array.ArrayEq` to compare the contents of both arrays as follows:

```
ensures { array_eq a (old a) }
```

In the first postcondition, we see that we can access the length of an array with $a.\mathsf{length}$ because and array is really a record but has special syntax to access its values. Second, we note the predicate $\mathsf{mem}\ x\ a$ which should be true if the value x is *somewhere* in the array a. So we specify before the function `search`:

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

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.

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. If $i < n$ after the loop it means we have x and we return i. If $i = n$ then we have tried all elements in the array and not found x.

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

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. Furthermore, we want to express that the array a does not change in the loop by stating that $a = (\mathsf{old}\ a)$.

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

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

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

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 value x is not in the array) or $i < n$, in which case $a[i] = x$ and we can return 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 [lower, 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 <= a.length /\ 0 <= 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 (old a) a 0 a.length}
let n = a.length in
for i = 0 to n-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 (old a) a 0 a.length}
let n = a.length in
for i = 0 to n-1 do
  (* no variant, or invariant on i needed *)
  invariant { negated a (old a) 0 i }
  invariant { array_eq_sub a (old a) i n }
  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 $[i, n)$ 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.

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, int) =
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: (int, int, int) = (0, (-1), 1)
globals:
%
```

5 Diagnosis of Failing Goals

The question you will find yourself asking most often while using Why3 is the following: *why didn't this goal prove ?*. There are three possible answers to this question:

1. The goal you are attempting to prove is *false*, which means there is an error in either your implementation or your specification.

2. The goal you are attempting to prove is *unprovable* because you missed an invariant or because some part of your implementation is underspecified. In the latter case, this means that you are missing a `requires` in the current function or that you are making a call to a function whose behavior is underconstrained (some `ensures` are missing). You have to keep in mind that, when looking at a function call, the provers have no access to this function's body and only see its specification.

3. The goal you are attempting to prove is *true* but the provers are not smart enough to figure it out. You will need to annotate your code more.

Here is a list of what you should do when one of your goals does not check:

1. Always start by splitting your goal and launch **all** provers, i.e., Alt-Ergo, CVC4, and Z3 on every generated subgoal. Look at what exact subgoals fail to be proved and what part of the code they correspond to (using the *Source* tab of the Why3 IDE).

2. If a subgoal G fails to be proved automatically, think of a proof of G yourself. Then, write down each argument or intermediate step in proving G as an assertion in the code and see what assertions fail to check.

3. If the subgoal that fails to be proved is small and simple enough, you can look at the *Task* tab of Why3 IDE to see the exact proof obligation that has been sent to the provers. A red flag indicating that it may be unprovable is when the conclusion features a variable that is almost unconstrained in the hypotheses.

4. If you manage to decompose your reasoning in many small steps using assertions, you should eventually reach a point where it becomes clear that either:
   a) the main goal is indeed wrong: you should fix your implementation or your specification.
   b) the main goal is unprovable: you should add some `invariant`, `requires` or `ensures` annotations.
   c) the provers are missing some piece of subtle reasoning and you should help them by providing external lemmas. Note that unless specified, we were able to solve every Why3 assignment without running into this.

In our experience though, when a goal does not check and it does not feature some crazy mathematical content, you are more likely to have missed something than the provers!

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

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

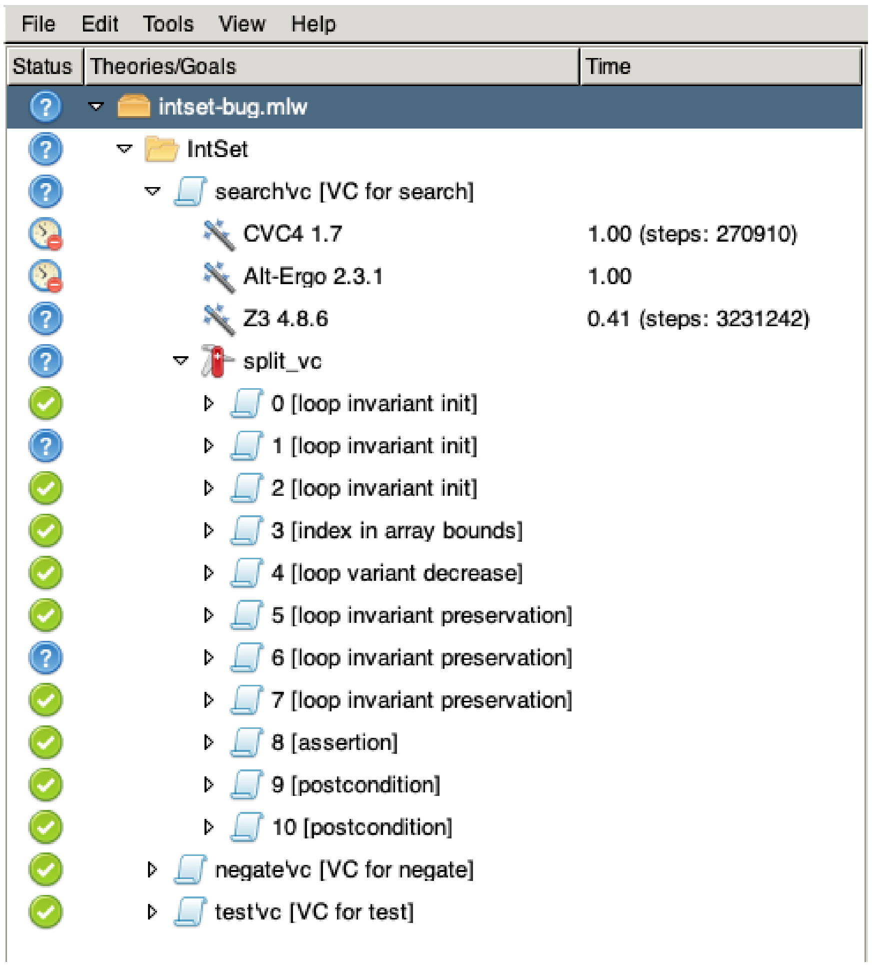
has an off-by-one error

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

We start the Why3 IDE with

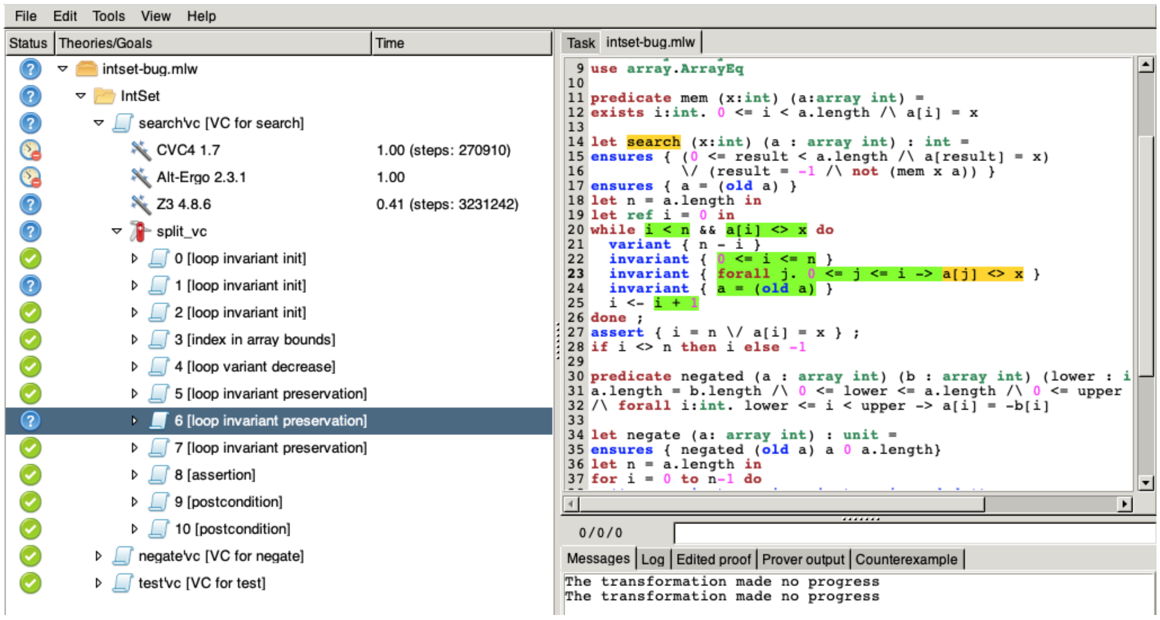
```
why3 ide intset.mlw
```

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.



Status	Theories/Goals	Time
?	intset-bug.mlw	
?	IntSet	
?	search'vc [VC for search]	
	CVC4 1.7	1.00 (steps: 270910)
	Alt-Ergo 2.3.1	1.00
?	Z3 4.8.6	0.41 (steps: 3231242)
?	split_vc	
✓	0 [loop invariant init]	
?	1 [loop invariant init]	
✓	2 [loop invariant init]	
✓	3 [index in array bounds]	
✓	4 [loop variant decrease]	
✓	5 [loop invariant preservation]	
?	6 [loop invariant preservation]	
✓	7 [loop invariant preservation]	
✓	8 [assertion]	
✓	9 [postcondition]	
✓	10 [postcondition]	
✓	negate'vc [VC for negate]	
✓	test'vc [VC for test]	

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.



```
intset-bug.mlw
  IntSet
    search'vc [VC for search]
      CVC4 1.7                              1.00 (steps: 270910)
      Alt-Ergo 2.3.1                        1.00
      Z3 4.8.6                              0.41 (steps: 3231242)
      split_vc
        0 [loop invariant init]
        1 [loop invariant init]
        2 [loop invariant init]
        3 [index in array bounds]
        4 [loop variant decrease]
        5 [loop invariant preservation]
        6 [loop invariant preservation]
        7 [loop invariant preservation]
        8 [assertion]
        9 [postcondition]
        10 [postcondition]
    negate'vc [VC for negate]
    test'vc [VC for test]
```

```
use array.ArrayEq

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

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

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

let negate (a: array int) : unit =
ensures { negated (old a) a 0 a.length}
let n = a.length in
for i = 0 to n-1 do
```

```
The transformation made no progress
The transformation made no progress
```

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.

6 Loop Variants

Which postconditions arise from *loop variants*? The short description "*it has to be a non-negative quantity that strictly decreases in the loop*" is somewhat imprecise, so let's look at the details. Why3 will check two conditions

1. If the loop guard is true, the variant v has to be nonnegative $v \geq 0$.

2. If we go around the loop once, the variant v strictly decreases.

The first point guarantees that if the variant is negative $v < 0$, the loop body will not be executed. In other words, the loop must exit if the variant becomes negative.

The first point also guarantees that the variant is nonnegative $v \geq 0$ when entering the loop, and the second point check that it strictly decreases. Eventually, it will therefore have to become negative, at which point the loop must terminate.

As an example, consider the simple loop

```
let ref i = 0 in                    (* value here is irrelevant! *)
while (i < n) do
  variant { n - i }
  i <- i + 1                        (* increment has to be positive *)
done
```

Here is the verification condition

```
goal f'vc :
  forall n:int.
   forall i:int.
    i < n ->
    (forall i1:int. i1 = (i + 1) -> 0 <= (n - i) /\ (n - i1) < (n - i))
```

The assumption $i < n$ means that loop guard is true. The part $0 \leq (n-i)$ guarantees that the variant $n - i$ is nonnegative (which follows from $i < n$). The part $(n - i1) < (n - i)$ verifies that the variant strictly decreases, because $i1 = i + 1$ is the value of i after one iteration of the loop. We wrote this as i' when analyzing our first mystery function.

7 Ghosts and Models

As an example of a model 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 ;
              mutable ghost model : Fset.fset int }
invariant { 0 <= bound <= a.length
            /\ forall j. 0 <= j < a.length -> a[j] <-> Fset.mem j
               model }
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.

```
let empty_bset (bound : int) : bset =
requires { bound >= 0 }
ensures { Fset.is_empty result.model /\ result.bound = bound }
{ 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.

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

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.

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

We did not implement any more complex operations such as union or intersection, even though this would certainly be possible. Note that the data structure invariant could also be strengthen as we will see in Assignment 1. You can find the code for the `Bitset` module in the file bitset.mlw.