Lab 1: 4 Why you get to know Why3

15-414: Automated program verification

Lab goals

This lab is designed to help you getting started with Why3 by specifying and implementing simple algorithms. This handout features commented examples. You should read them carefully as they demonstrate pieces of Why3's syntax and semantics that you will need for completing the exercises.

Lab instructions

First download the template file `lab1.mlw` on the class website. It is divided in five sections corresponding to five different exercises. Each exercise has a template in comments with holes (* "..." *). You should uncomment those lines (by removing the lines "(*" and "*)") and fill the holes ("...") in order to complete the homework. Here are some general remarks before we give detailed instructions for each exercise.

Remarks

- It is possible to complete every exercise without importing additional modules and theories from the Why3 standard library but you are free to do otherwise.
- You are *encouraged* to extend the template we give you with additional predicates and auxiliary functions, especially when it helps clarity.
- If you try to launch the Why3 IDE on a file that is not syntactically correct, it will print an error and fail. Therefore, we recommend you to uncomment and solve each exercise one by one.

Example 1: loops and mutable references

Our first example is taken from the live coding session and can be found Figure 1. It should teach you how to work with `while` loops and mutable references.

```
use int.Int
use ref.Ref

let square (x:int) : int =
  requires { 0 <= x }
  ensures { result = x * x }
    let s = ref 0 in
    let i = ref 0 in
    while !i < x do
      invariant { !s = !i * !i }
      invariant { !i <= x }
      variant { x - !i}
        s := !s + 2 * !i + 1;
        i := !i + 1
    done;
    !s
```

Figure 1: Find the square of a number by using only addition

Mutable references Remember that in ML, variables are immutable by default. In order to have mutable memory cells, you should use references (from `ref.Ref`). References are created using the "`ref`" keyword, accessed using the "`!`" operator and modified using ":=".

Proving correctness with loop invariants In order to prove the correctness of functions featuring loops, it is critical to declare *loop invariants*. A loop invariant is a formula P for which we can prove that:

1. **Invariant initiation:** P is true before the loop is entered.
2. **Invariant preservation:** whenever P is true before executing the loop body, it remains true after.

When the two properties above are true, P is also true **after** the loop execution. The provers only have access to the following information about the state of a program after a loop: [1]

- The loop guard is false.
- Loop invariants are true.
- Variables that appear to be never modified by the loop body have the same value before and after the loop.

In particular, any variable that is modified in the loop body should be mentioned in an invariant. Otherwise, the provers will have no information about its value after the loop.

Proving termination with loop variants To guarantee termination, a `while` loop must be attached a *variant declaration* that provides an integer expression that can be proven to be nonnegative and strictly decreasing at every loop iteration.

Mentioning the result of a function in its specification The `result` keyword can be used in specifications to denote the result of the function being specified.

[1] If you are unsure whether or not some formula can be deduced from these informations, add an assertion after the loop body and see whether or not it checks.

Exercise 1: computing the n^{th} Fibonacci number

The goal of this exercise is implement a verified function `compute_fib` that computes the n^{th} Fibonacci number in time linear with n. Because the Fibonacci sequence is not defined in the Why3 standard library, we have to axiomatize it ourselves. This is done in the template file as follows:

```
function fib int : int
axiom fib_0: fib 0 = 0
axiom fib_1: fib 1 = 1
axiom fib_n: forall n:int. n >= 2 -> fib n = fib (n-1) + fib (n-2)
```

This code introduces an abstract function symbol `fib` and provides three defining axioms for it.

Instructions

1. Fill in the body of `compute_fib` (the specification is already provided in the template file). Your implementation must be non-recursive and have linear time complexity.

2. Make sure that the generated verification conditions for `compute_fib` are successfully checked by an automated prover using the Why3 IDE. As Alt-Ergo is bad at dealing with arithmetic, we advise that you use Z3 instead. [2]

[2] Even though Alt-Ergo is bad at dealing with arithmetic, we were able to use it to check the verification conditions of our solution implementation.

Example 2: lists and recursive functions

Our second example is taken from Lab 0 and can be found Figure 2. It should teach you how to work with lists and define recursive functions.

```
use int.Int
use list.List
use list.Mem (* Theory of list membership *)

let rec remove (a : int) (l : list int) : list int =
  ensures { not (mem a result) }
  ensures { forall x:int. x <> a -> (mem x result <-> mem x l) }
  variant { l }
  match l with
    | Nil -> Nil
    | Cons x xs ->
      if x = a then remove a xs
      else Cons x (remove a xs)
  end
```

Figure 2: Removing every instance of an element from a list

Working with lists Lists in WhyML work exactly like in SML, and they are defined as follows in the standard library [3] (in module `list.List`):

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

. Lists are built using the `Nil` and `Cons` constructors [4] and examined through pattern matching. Because lists are defined as a recursive datatype, functions manipulating them are usually recursive.

[3] The documentation of the standard library is available at `http://why3.lri.fr/stdlib/`.

[4] There is no "`[]`" and "`::`" syntactic sugar for lists in WhyML.

Writing variants In order to guarantee termination, every definition of a recursive function should be accompanied by a variant declaration. A variant can be either:

- The name of a list argument l such that every recursive call is made on a sublist of l (this is the case in the example of Figure 2).
- An integer expression for which we can prove that it is nonnegative and it strictly decreases at each recursive call.

Proving recursive functions correct You may find it surprising that checking the correctness of `remove` in Figure 2 does not require adding any code annotation in addition to the specification. Indeed, verifying functions featuring loops usually requires manually writing loop invariants. Why would recursive functions require no additional annotations, especially since loops can be emulated using recursion ? The answer is that in the case of recursive functions, the specification (`requires` and `ensures` statements) also serves as a recursive invariant. For example, in Figure 2, provers can use the hypothesis that the postcondition of `remove` holds for every recursive subcalls when proving that it holds after its main body. You are encouraged to see this by yourself using the *Task* tab of the Why3 IDE.

Other details you should pay attention to:

- How pattern matching is used on lists using the "`match ... with`" construct.
- The keyword `rec` has to be used with `let` when defining recursive functions.
- The predicate `mem` is imported from the standard library (`list.Mem`) and refers to list membership.

Exercise 2: generating a list of all integers in an interval

The goal of this exercise is to specify and implement a recursive function `range` that takes two integers a and b with $a \leq b$ and returns a list containing every integer in the $[a, b)$ interval.

Instructions

1. Write a specification for `range` by filling in the holes within the `requires` and `ensures` statements. The specification should use the `mem` list membership predicate in `list.Mem` and state something like: an integer belongs to the resulting list if and only if it belongs to the $[a, b)$ interval. Note that your specification only needs to cover the existence of integers in the $[a, b)$ interval and does not need to enforce any restrictions on duplicated or unordered integers.

2. Write the implementation of `range`.

3. Provide a variant for `range`

4. Make sure that the generated verification conditions for `range` are successfully checked by an automated prover using the Why3 IDE.

Example 3: arrays, loops and and exceptions

Our third and final example is taken from Lab 0 and is displayed in Figures 3 and 4. It should teach you how to work with arrays, `for` loops and exceptions. It consists in two different implementations of a function that computes the index of the first positive number in an array of integers when it exists and returns `None` otherwise.

Option type Option types are defined as follows in `option.Option`, in a similar way than in SML:

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

In Figure 3, the `find_first_pos_idx` function returns a value of type `option int` instead of an `int` to account for the case where the array given in argument contains no positive value.

Arrays Arrays are defined in the `array.Array` module from the standard library, whose documentation is available at:

```
http://why3.lri.fr/stdlib/array.html
```

The expression "`make n v`" creates an array of `n` cells that are initialized to value `v`. Besides, "`length t`" denotes the length of array `t`. Arrays can be accessed using the "`t[i]`" notation and modified using the "`t[i] <- v`" notation. Every time an array is modified or accessed at a given index, Why3 generates a proof obligation to ensure that this index is not out of range. Finally, you should know that the equality operator "`=`" does not work for arrays. In order to express the fact that two arrays `t` and `t'` are equal, you should write:

```
length t = length t' /\ forall i:int. 0 <= i < length t -> t[i] = t'[i]
```

or use the predicate `array_eq` that is defined in `array.ArrayEq`.

Initial values One important thing to notice about these examples is the last postcondition:

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

Within a postcondition, `old t` refers to the value of `t` in the prestate, i.e., the state in which the precondition is evaluated. This part of the specification documents the fact that the array given to the procedure remains unchanged throughout its execution, to the point of return.

Because arrays are mutable, the rest of the specification could easily be satisfied by an incorrect implementation if we did not include this part of the spec. It is good to get into the habit of documenting the effect that a procedure will have on mutable state such as arrays, even in the simplest cases where that state remains unchanged.

Exceptions As there are no `break` or `return` statements in ML, it is idiomatic to use exceptions for escaping loops or functions. Exceptions can be declared using the keyword `exception`, raised using `raise` and caught within "`try ... catch`" blocks.

For loops Unsurprisingly, `for` loops are very similar to `while` loops, except that it is not required to provide a variant for them.

```
use array.Array
use array.ArrayEq
use ref.Ref
use option.Option

predicate none_pos_sub (t : array int) (a b : int) =
  forall i:int. a <= i < b -> t[i] <= 0

predicate none_pos (t : array int) = none_pos_sub t 0 (length t)

predicate first_pos_at (t : array int) (ix : int) =
  0 <= ix < length t /\ t[ix] > 0 /\ none_pos_sub t 0 ix

let find_first_pos_idx (t : array int) : option int =
  ensures { result = None -> none_pos t }
  ensures { forall ix:int. result = Some ix -> first_pos_at t ix }
  ensures { array_eq t (old t) }

  let v = ref None in
  let i = ref 0 in
  while !i < length t && is_none !v do
    variant { length t - !i }
    invariant { 0 <= !i }
    invariant { forall ix:int. !v = Some ix -> first_pos_at t ix }
    invariant { !v = None -> none_pos_sub t 0 !i }
    if t[!i] > 0 then v := Some !i ;
    i := !i + 1
  done ;
  !v
```

Figure 3: Finding the index of the first positive element in an array of integers

```
exception Found int

let find_first_pos_idx' (t : array int) : option int =
  ensures { result = None -> none_pos t }
  ensures { forall ix:int. result = Some ix -> first_pos_at t ix }
  ensures { array_eq t (old t) }

  try
    for i = 0 to length t - 1 do
      invariant { none_pos_sub t 0 i }
      if t[i] > 0 then raise (Found i)
    done ;
    None
  with Found ix -> Some ix end
```

Figure 4: An alternative implementation using a `for` loop that raises an exception

Exercise 3: finding the maximum element in an array of integers

In this exercise, we want you to specify and implement a function `max_array` that finds the maximum element in an array of integers. It should raise the exception `Empty` when given an empty array as an argument.

Instructions

1. Write a specification for `max_array`. It has to be complete in the sense that it should rule out any incorrect implementation.
2. Fill in the implementation of `max_array` and make sure that the generated verification conditions are successfully checked by an automated prover using the Why3 IDE.

Note: The template file features the declaration

```
raises { Empty -> length t = 0 }
```

as part of the specification for `max_array`. Indeed, every function that may raise an uncaught exception must declare a post-condition for it. In our case, the declaration above tells Why3 that when a call to "`max_array t`" raises exception `Empty`, it has to be true that `t` is the empty array.

Exercise 4: finding the maximum element in a list of integers

In this exercise, we want you to specify and implement a function `max_list` that finds the maximum element in a list of integers. It should raise the exception `Empty` when given an empty list as an argument.

Instructions

1. Write a specification for `max_list`. It has to be complete in the sense that it should rule out any incorrect implementation.
2. Fill in the implementation of `max_list` and make sure that the generated verification conditions are successfully checked by an automated prover using the Why3 IDE.

Exercise 5: finding the maximum element in a list of integers

In this exercise, we want you to specify and implement a function `smallest_greater_than` that:

- takes as an argument an array of integers t that is sorted in increasing order, along with an integer b
- returns the smallest element in t that is strictly greater than b (and `None` if no such element exists).

Instructions

1. Write a specification for `smallest_greater_than`. It has to be complete in the sense that it should rule out any incorrect implementation.
2. Write an implementation for `smallest_greater_than` and make sure that the generated verification conditions are successfully checked by an automated prover using the Why3 IDE. For **bonus points**, your implementation should have **logarithmic time complexity**.

What to hand back

First make sure that every goal is handled successfully by the provers in your completed version of `lab1.mlw`. Labs should be submitted on the course Canvas website, under the Lab1 assignment. Upload a zip file named AndrewID_lab1.zip, using your Andrew ID. The zip archive should contain the following content:

1. The completed `lab1.mlw` file
2. The session folder generated by Why3 IDE
3. An ASCII text file with extension `.txt` or a .pdf file containing some comments you may want to share with us (optional).

In order to make sure your archive is valid, you can uncompress it and run the following two commands (we will do the same):

```
why3 replay lab1  # should print that everything replayed OK
why3 session info --stats lab1  # prints the list of unproved goals
```

Note Having all your proof goals checked does **not** necessarily mean that you will get a perfect grade on your homework. Indeed, you also have to make sure that your specifications are correct and complete. For example, it doesn't help if everything proves but you assumed precondition *false* everywhere.

Some tricks and advice on using Why3

The question you will find yourself asking most often while using Why3 is the following: *why the hell 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 **both** Alt-Ergo 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 we were able to solve every Why3 lab without running into this. [5]

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!

[5] with one exception in the last lab that we will discuss later.