Assignment 1
Variations on a Theme

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Thursday, February 3, 2022
70 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~15414/assignments.html`.

Working With Why3

Before you begin this assignment, you will need to install Why3 and the relevant provers. To do so, please follow the installation instructions on the course website (`https://www.cs.cmu.edu/~15414/misc/installation.pdf`).

To help you out with Why3, we've provided some useful commands below:

- To verify using the command line, run `why3 prove -P <prover> <filename>.mlw`. This is useful for simple programs where more fine-grained control over the provers is unnecessary, as well as for intermediate checking. However, your final submission should include proof sessions as created by the IDE.
- To open the Why3 IDE, run `why3 ide <filename>.mlw`.
  - When you attempt to prove the goals in a file `filename.mlw` using the IDE, a folder called `filename` will be created, containing a *proof session*. Make sure that you always save the current proof session when you exit the IDE. To check your session after the fact, you can run the following two commands:

    ```
    why3 replay filename   # should print that everything replayed OK
    why3 session info --stats filename   # prints a summary of the goals
    ```

  - Although it's not possible to modify code directly from the IDE, if you make changes in a different editor (VSCode, Atom, etc.), you can refresh the IDE session with Ctrl+R.

What To Hand In

You should hand in the file `asst1.zip`, which you can generate by running `make`. This will include all of the raw `mlw` files, as well as the proof sessions created by the IDE.

1 Negative Rabbits (18 pts)

The Fibonacci sequence may be extended to negative numbers simply by applying the definition $\mathsf{fib}(0) = 0$, $\mathsf{fib}(1) = 1$, and $\mathsf{fib}(i) + \mathsf{fib}(i+1) = \mathsf{fib}(i+2)$ to all integers. In this problem we ask you to rewrite the specification and implementations of the imperative and functional Fibonacci functions to cover all integers. Your code should be similar in its efficiency to the implementation we developed and proved in Lectures 1 and 2 which you can find in the file `fib.mlw`.

Task 1 (9 pts). Provide a verified *imperative* implementation of Fibonacci numbers `fib_loop`.

Task 2 (9 pts). Provide a verified *functional* implementation of Fibonacci numbers `fib_pure`.

Both of these functions should be in the file `fib.mlw`.

Note! While the provided implementation f has a precondition:

```
  requires { n >= 0 }
```

Your implementations of `fib_loop` and `fib_pure` should NOT have this contract since they should extend the Fibonacci sequence to negative numbers.

2 The Fine Print (8 pts)

Unlike software license agreements that nobody ever reads (I agree!), program contracts should be studied carefully because they might not mean what you think at first and you may be left holding the bag. The following is an *incorrect* attempt to implement an iterative factorial function (which you can find in the file `fact.mlw`).

```
module Factorial

  use int.Int

  function factorial (n:int):int
  axiom factorial0: factorial 0 = 1
  axiom factorialn: forall n. n > 0 -> factorial n = n * factorial (n - 1)

  let fact(n:int) : int =
  requires { n >= 0 }
  ensures { result = factorial n }
  let ref i = 0 in
  let ref r = 1 in
  while i < n do
    invariant { 0 <= i <= n }
    invariant { r = factorial i }
    variant { n-i }
    r <- r * i ;
    i <- i + 1 ;
  done ;
  r

end
```

Task 3 (8 pts). In each of the following sub-tasks you should change the contracts, *and only the contracts* (except in part 5) of the above incorrect implementation, so that the command

```
why3 prove -P alt-ergo fact.mlw
```

succeeds in verifying the code.

1. You may remove two lines.
2. You may add conjunction `/\` and falsehoold `false`, as many copies as you wish.
3. You may add disjunction `\/` and truth `true`, as many copies as you wish.
4. You may add comparison `<` between variables and implication `->`, as many copies as you wish.

   Note! You may NOT use comparators that are different from `<` (e.g. not allowed to use `>`). Also, the comparison `<` can only be between two variables, and not a variable and a constant.
5. You may swap any two lines (not restricted to contracts).

Name your functions `fact_i` for $1 \leq i \leq 5$ and place them in the file `fact.mlw`.

3 Queue Up (18 pts)

In this problem we ask you to refactor the implementation of queues in `queue.mlw` by using the sequence representation as a *model* of the queue state.

Task 4 (18 pts). Provide a verified implementation of queues (with `empty`, `enq`, and `deq` operations) where the sequence represented by the queue is carried as a model of the data structure. That is, use the type

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

where `q.model` is the state of the queue represented as a list. This property should be captured as a data structure invariant. Make sure there are no *redundant* pre- or post-conditions in your code.

The `ghost` annotation here means that the `model` field of the record can only be used in contracts and other `ghost` fields and variables. It is for verification only and can be safely erased when the program is compiled. Ghosts are discussed in more detail in Lecture 4.

Note! Ghost models should make it easier to reason about your code in contracts. While the `sequence` function is used in the initial contracts of the `queue.mlw` file, when you have an appropriate model and invariant, the `sequence` function becomes unnecessary and should not appear in the new contracts you write. Instead, you should enforce properties about the model, which should be guaranteed by an invariant to be an accurate representation of the full queue.

Place your implementation in the file `queue.mlw`.

4 Differentiate Discretely (26 pts)

Discrete differentiation is an operation that replaces a sequence such as $2, 5, 10, 17, 26$ by the differences between consecutive elements, $3, 5, 7, 9$, in this case. Iterating the process once more give us $2, 2, 2$. Even though we are not pursuing it in this problem, it is possible to determine a polynomial representation of the sequence from the iterated finite differences (here: $x^2 + 2x + 2$).

Task 5 (13 pts). Write a verified function `diffs (a : array int) : array int` that returns a new array of differences between the elements of a, starting with $a[1]-a[0]$, $a[2]-a[1]$, etc. Your function should not modify a itself, i.e. a at the end of the function should be equal to a at the beginning. The length of the output array should be one less than the length of the input array.

Task 6 (13 pts). Write a verified function `diffs_in_place (a : array int) : unit` that replaces each element in the array by the difference to the next one, without allocating a new array. The last element can be arbitrary.

[Hint: for working with mutable arrays we found the alt-ergo and Z3 provers to be generally more effective than CVC4. Also, the `array.ArrayEq` standard library may be helpful for concise specifications.]

Place your implementations in the file `diff.mlw`.

Note! Be careful to ensure that your contracts cover ALL of the parts of the functions' specifications from the task descriptions.