Assignment 1
Twists on a Theme

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Friday, September 12, 2025
65 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 The Fine Print (15 pts)

Unlike software license agreements that nobody ever reads, 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 fact (n : int) : int
  axiom fact0: fact 0 = 1
  axiom factn: forall n. n > 0 -> fact n = n * fact (n - 1)

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

end
```

Task 1 (15 pts). In each of the following sub-tasks you should change the contracts, *and only the contracts* (except in part 4) 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 disjunction `\/` and truth `true`, as many copies as you wish.
3. You may add comparison `<` between variables and implication `->`, as many copies as you wish.
4. You may swap any two lines (not restricted to contracts), and add at most two contracts. Your proof in this case ***must be correct***.

Name your functions `fact_i` for $1 \leq i \leq 4$ and place them in the file `fact.mlw`.

2 Set It Straight (20 pts)

In this problem you are given a *partial implementation* of a data set finite set of integers in `intset.mlw`. This is a generaliztion of the `bitset` data structure that we implemented in Lecture 3 that does not place a bound on the size of the elements in the set, although it does place a bound on the number of elements in the set.

```
type intset = {
  slots: array int ;
  used: array bool ;
  capacity: int ;
  mutable full: bool ;
  ghost mutable model: S.fset int
}
```

At the end of the file, there are a set of fully specified and implemented operations on the data structure: `empty`, `mem`, `add`, and `remove`.

Task 2 (20 pts). Your task is to supply a set of invariants (and, if helpful, auxiliary spcification such as predicates or functions) so that all of the provided operations verify completely.

Rules and constraints:

- You may only add logical content before the "do not modify" line: e.g., invariants over `intset`, predicates, and functions.
- **Do not change the data structure itself.**
- **Do not change the code of the specifications of** of `empty`, `mem`, `add`, or `remove`.
- While you are allowed to define functions using axioms, you should avoid doing so. This task can be completed without introducing axioms.

You will find it helpful to look at the contracts and proof (loop invariants, assertions, etc.) of the provided operations to help you write your invariants. We do not require that you write any particular invariant, aside from any that you need to verify all of the operations.

Do not modify any code or contracts below the marked line in `intset.mlw`; add only invariants and ghost logic above it.

3 Prefix Sums (30 pts)

Let a be an array of integers. The prefix sums of a are defined by $b[i] = \sum_{j=0}^{i} a[j]$ for each valid index i.

Task 3 (5 pts). Define a predicate `is_prefix_sums (a : array int) (b : array int)` that holds exactly when b contains the prefix sums of a. Your definition should capture the required length relation and, for each index i, the value of $b[i]$ in terms of $a[0], a[1], \ldots, a[i]$.

Task 4 (10 pts). Write a verified function `prefix_sums (a : array int) : array int` that returns a new array b such that $\text{is\_prefix\_sums}(a, b)$ holds. Your function should not modify a itself.

Task 5 (15 pts). Write a verified function `prefix_sums_in_place (a : array int) : unit` that overwrites a so that $\text{is\_prefix\_sums}(\text{old } a, a)$ holds. Do not allocate a new array.

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

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