Assignment 1
Variations on a Theme

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Friday, February 2, 2024
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 with version 1.4.0 of Why3:

    ```
    why3 replay <filename>                # should indicate that session is okay
    why3 session info --stats <filename>  # prints a summary of the goals
    ```

    If you are using version 1.7.0, then you should run the following two commands:

    ```
    why3 replay <filename>                # should indicate that session is okay
    # prints proof statistics for each given session
    why3 session info --session-stats <filename>
    ```

  - Although it's not possible to modify code directly from the IDE, if you make changes in a different editor (VSCode, Emacs, 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 Arrays and Maximum Element (10 pts)

In this exercise, we want you to specify, implement, and verify a function `max_array` that finds the maximum element in an array of integers. You can write preconditions that assume that the array is not empty.

```
module MaxArray

  use int.Int
  use array.Array

  let max_array (t : array int) : int =

end
```

Task 1 (10 pts). We provide an initial file `maxarray.mlw` with the function declaration. You should modify this file and prove in Why3 that your implementation satisfies your specification. Your specification should be complete, i.e., it should rule out any incorrect implementations of this function. Note that if you prefer you can also change the function declaration to be recursive. You can also use any of the functions defined in the standard library of Why3 for arrays.

2 Lists and Maximum Element (10 pts)

In this exercise, we want you to specify, implement, and verify a function `max_list` that finds the maximum element in a list of integers. This is similar to the previous task but using lists instead of arrays. Again, you can write preconditions that assume that the list is not empty.

```
module MaxList

  use int.Int
  use list.List

  let max_list (t : list int) : int =

end
```

Task 2 (10 pts). We provide an initial file `maxlist.mlw` with the function declaration. You should modify this file and prove in Why3 that your implementation satisfies your specification. Your specification should be complete, i.e., it should rule out any incorrect implementations of this function. Note that if you prefer you can also change the function declaration to be recursive. You can also use any of the functions and predicates defined in the standard library of Why3 for lists. For instance, the predicate `list.Mem` may be useful to check membership of an element in a list.

3 Relaxed Requirements (20 pts)

In this problem, we ask you to extend and modify the implementation of integer sets using bitvectors that we briefly covered in lecture (and include in `bitset.mlw`).

Recall that a `bset` is a record consisting of an array `a`, a `bound` and a ghost field called `model` containing a finite set of integers. If an integer i is in the bitset then `a[i]=true`. The bitset has a bound on the number of elements we can add and it uses the ghost field to have an abstract representation of the content of the array. Ghost variables, or ghost fields of records, can only be used in other ghost computations and exist solely for the purpose of the verification. The data structure invariants ensure that the relationship between the ghost field and the contents of the array is consistent. Since these data structure invariants must be preserved every time we use bitsets, we can write our contracts using the ghost model instead of referring to the underlying implementation. More details can be found on the lecture notes on Arrays and Ghosts.

Task 3 (20 pts). Implement and verify a function `test` that checks if a given number x is in the bitset s. This function should return *false* if the number is not in the bitset and *true* if the number is in the bitset. Note that if a number x is not within bounds, then the function should return *false*. The function `test` has the following function declaration and specification:

```
1    let test (x : int) (s : bset) : bool =
2        ensures { result <-> Fset.mem x s.model }
3        ensures { s.model == (old s.model) }
```

You must use this specification for the `test` function, i.e., you should not add or modify any additional posconditions or preconditions. Note that this functions does not assume that the x is within bounds, i.e., it does not `requires { 0 <= x < s.bound }`. In order to verify this function, you will need to extend the data structures invariants so that Why3 is able to verify the `Bitset` module.

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

4 Differentiate Discretely (30 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: $x^2 + 2x + 2$).

```
module Diff

  use int.Int
  use array.Array
  use array.ArrayEq

  let diffs (a : array int) : array int =
  let diffs_in_place (a : array int) : unit =
```

Task 4 (15 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 5 (15 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. Note that to compare array contents, you should use `array.ArrayEq` from the standard library. There may also be other functions or predicates that may be helpful for concise specifications in the standard library for arrays.]

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.