CSCI 2041: Advanced Programming Principles
Spring 2041, University of Minnesota
Homework 5

Posted: Oct 27, 2017
Due: Nov 10, 2017 by 23:59

Submission Protocol

Some requirements that you must adhere to in submitting your work for grade for this homework:

- All of your work must be submitted in a file named *hw5.ml*. This file must be **pushed** to your personal git repository in the organization at the UMN github for this course, i.e. in a repository that has the name *umn-csci-2041F17d/repo-abcde001* where *abcde001* is your username. This file should be placed at the top level in this directory.
- Include in the file a comment of the form (* Solution for Problem X *), where X is the number of the problem, before the solution to each problem.
- Most of the problems have subparts. Include a comment of the form (* Problem X, Part Y *) prior to the solution you provide for each of these parts.
- The programming problems are specific about the names of the types, functions and the types of the functions you must define. As usual, you **must** stick to the requirements spelled out in each case.
- When the problem requires you to provide a descriptive answer such as a proof (as is the case for the last three problems), include this in the form of a comment in your OCaml file.

Note that general requirements for homework submissions are described on the homeworks page and have also been discussed in class. In addition, use all the requirements for good programming that you would have learnt in preceding programming courses, doing so can have an impact on the grade.

Before submitting the final version of your homework, you should ensure it is well-formed and type-correct, i.e., contain no syntax or type errors. To check you could use the following command on a cselabs machine in a directory containing the file:

```
ocamlbuild hw5.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
```

If you see:

```
---INCORRECT SUBMISSION---
```

it means that your file does not compile and hence is not yet ready to be submitted.

Problem 1 (2 + 2 + 4 + 2 points)

In this problem, we will assume a language with statements like in Java or C but in which we can represent lists and use functions on them like in OCaml. In this language, we can write the following program fragment to reverse a list

```
rev = [];
while not (l = []) {
  rev = (List.hd l) :: rev;
  l = List.tl l;
}
```

The task in this problem is to using the ideas we discussed in class and that you used in Lab 9 to eventually write a let declaration that binds the identifier `revprog` to the encoding of this program in OCaml. However, you should do it in the following steps:

1. Explain (in a comment in the `hw5.ml` file) what the representation of a state would be for this program fragment. Then provide a type declaration that defines the type `state` based on the representation you have chosen. Assume for this part that all our lists are integer lists.

2. Write the `get` and `put` functions for each of the variables used in this program based on your encoding of the state.

3. Write a let declaration that binds the identifier `revprog` to the encoding in OCaml of the program fragment shown. You may use the definitions of `seq`, `ifstat`, and `whilestat` provided to you in the file `impprog.ml` that you will find in the `code` subdirectory of the public repository for this course.

4. Using `revprog`, define a function `revlist : (int list) -> (int list)` that takes an integer list as input and returns its reverse as output. An example interaction using this function would be the following:

   ```
   # revlist [1;2;3];;
   - : int list = [3; 2; 1]
   #
   ```

   Your definition of `revlist` **must** use `revprog` to get credit, simply defining a function for reversing integer lists will not be sufficient.

Problem 2 (5 + (1 + 1 + 3) + 2 points)

1. Recall the `do-while` construct from C. The syntax for it is the following

   ```
   do <statement> while <condition>;
   ```

   where `<statement>` can be any statement in C and `<condition>` is any boolean-valued (actually, integer-valued, but for simplicity we will use booleans here) expression in C. The meaning of this construct is that we should execute the statement, then repeatedly execute the loop so long as the condition remains true.

   Assuming the model for encoding statements in C that we considered in class and in Lab 9, provide a definition for the identifier `dostat` so that it encodes this `do-while` construct.

2. Using the encoding of `do-while` described above, provide an encoding of the following program fragment from C that calculates the sum of the numbers up to a given positive number `n`:

```
i = 0;
sum = 0;
do { i = i + 1; sum = sum + i; }
while (i < n);
```

   You should do this through the following steps:

   a. Explain in a comment how you will represent state as relevant to this program fragment and then define a type `state` corresponding to this representation.

   b. Based on your representation of state write the `get` and `put` functions for the variables used in this program fragment.

   c. Using a let declaration, bind the identifier `sumup` to an expression in OCaml that represents the shown program fragment.

3. Using the definition of `sumup` from the previous part, define the function `sumToN : int -> int` that takes a positive number and sums the natural numbers up to that number. An example interaction using this function would be the following:

```
# sumToN 7;;
- : int = 28
#
```

   You **must** use `sumup` from the previous part in defining this function. Simply writing a function to sum up the natural numbers up to a given positive number will not be sufficient.

Problem 3 (3 + 5 + 2 points)

First, let us recall the definition of the Fibonnaci numbers: the first and second Fibonnaci numbers are both 1; the n^{th} Fibonnaci number for n greater than 2 is the sum of the $(n-1)^{th}$ and $(n-2)^{nd}$ Fibonnaci numbers.

Now consider the following function in OCaml that calculates the n^{th} Fibonnaci number

```
let fib n =
  let rec fib' n m f s =
    if (n = m) then f
    else fib' n (m+1) s (f+s)
  in fib' n 1 1 1
```

Your task in this problem is to prove that this function is correct, i.e., you need to show

For `n` greater than or equal to 1, `(fib n)` evaluates to the n^{th} Fibonnaci number.

The steps through which you should do this are the following:

1. Identify a property of `fib'` that will be useful in showing this; recall that this kind of property of auxiliary functions is what we have been calling a "lemma" about them.

2. Using induction, prove that `fib'` has this property.

3. Using the property you have shown `fib'` to possess, prove that `fib` has the property that we want to show for it.

Be very careful in picking the property you want to show for `fib'`. As we saw in the case of `tr_fact` in class, you need to pick a property that satisfies two different criteria: you must be able to show that `fib'` possesses it *and* it must be useful in showing that `fib` is correct.

As mentioned earlier, whatever you write for this part should be included in comments in the `hw5.ml` file.

Problem 4 ((2 + 4 + 2) + (2 + 4 + 2) points)

1. Consider the tail-recursive version of the `reverse` function we defined in class a while ago.

```
let reverse lst =
   let rec rev lst1 lst2 =
      match lst1 with
      | [] -> lst2
      | (h::t) -> rev t (h::lst2)
   in rev lst []
```

   Let us represent the reverse of a list *l* in mathematical notation by writing *l^R*. Further, let us use the infix operator + to represent the appending of lists at a mathematical level, i.e. $l_1 + l_2$ represents the result of appending the two lists l_1 and l_2. Using this notation, the correctness of *reverse* can be expressed as the following property:

   > For all lists *l*, *(reverse l)* evaluates to *l^R*.

   The task in this part is to prove that *reverse* satisfies this property. You are to do this in the following steps:

   a. Articulate a property of *rev* that you can both prove and that would be helpful in proving that *reverse* satisfies the property required of it.

   b. Prove the property of *rev* that you have stated using induction on the structure of lists.

   c. Use the property that you have proved for *rev* to prove that *reverse* satisfies the property required of it.

   In writing your arguments, you can use simple facts about lists at the mathematical level, such as the following

   > for all lists l_1 and l_2 and all *x* that can be elements of such lists, $(x::l_1)^R + l_2 = l_1^R + (x:: l_2)$.

   Of course, such properties must be obvious once you write them down, otherwise you must prove them.

2. Consider now the following function that computes the length of a list:

```
let rec length lst =
   match lst with
   | [] -> 0
   | (h::t) -> (length t) + 1
```

   The task in this part is to show that *reverse* preserves the lengths of lists, i.e. that

   > *for all lists l, (length (reverse l))* evaluates to the same value as *(length l)*

Follow the same sequence of steps as in the first part, i.e.

a. Articulate a property based on *length* for *rev*.

b. Prove the property about *rev* and *length* using induction on lists.

c. Use the property you have proved for *rev* to conclude that *reverse* preserves the lengths of lists.

Include solutions to this problem in comment form in the homework file.

Problem 5 (8 points)

Consider the following code that defines a representation for binary trees and then defines functions for inserting items into the tree and for looking for items in the tree.

```
type 'a btree =
  | Empty
  | Node of 'a * 'a btree * 'a btree

let rec insert t i =
   match t with
   | Empty -> Node (i,Empty,Empty)
   | Node (i',l,r) ->
        if (i < i') then Node (i',insert l i,r)
        else Node (i',l,insert r i)

let rec find t i =
   match t with
   | Empty -> false
   | Node (i',l,r) ->
       if (i = i') then true
       else if (i < i') then find l i
            else find r i
```

To keep things simple, we have followed the somewhat naive (and "broken") idea of using = and < directly on the items stored in the tree, even though we now know how to do this better.

There are a few different properties that we would want to prove of our *insert* function in order to be able to claim it is correct. We will look at some of these in class. For this problem, your task is to prove the following:

> For any given tree *t* and for any given data item *x* suppose that *(insert t x)* evaluates to the tree *t'*. Then *(find t' x')* evaluates to true for any data item *x'* of the type stored in *t* only if either *x'* is *x* or *(find t x')* evaluates to true.

Use induction on the structure of the tree *t* to prove this.

A curious thing about the property you are going to prove: it is true regardless of whether or not our trees respect the invariant of binary search trees, even though the *insert* and *find* functions are structured based on this invariant.

Include solutions to this problem in comment form in the homework file.
