15-150 Fall 2013
Homework 06

Out: Wednesday, 2 October 2013
Due: Tuesday, 8 October 2013 at 23:59 EST

1 Introduction

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/06` directory should contain a file named exactly `hw06.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/06` directory (that contains a `code` folder and a file `hw06.pdf`). This should produce a file `hw06.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw06.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

Your `hw06.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 8 October 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 Higher-Order Functions

Now that you are getting comfortable with higher-order functions, you can explore some different combinations and applications. In the next problems, you will identify the types and meanings of expressions containing higher-order functions.

2.1 `map` and `filter`

Recall : the `map` function is defined as follows:

```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)
```

and `filter` is defined as follows:

```
(* filter: ('a -> bool) -> 'a list -> 'a list *)
fun filter f [] = []
  | filter f (x::xs) = if f x then x :: (filter f xs)
                       else filter f xs
```

For each of the following expressions, determine if it has a type.

- If it is not well typed, say so and explain why.
- If it is well-typed, state its type, the value it will evaluate to, and if the expression has a function type, in in a sentence or two what the function will do if given an appropriate argument.

Task 2.1 (3%). `filter (fn (x,y) => x+y = x*3) [(2,5),(8,16),(0,0),(40,2)]`

Task 2.2 (3%). `map (filter (fn x => x^"k"))`

Task 2.3 (3%). `filter (fn x => (map (fn y => 12) x) = x)`

Task 2.4 (4%). `map (fn x => map x [4,5,6])`

3 Radix Sort

3.1 Background

Radix sort is a sorting algorithm that is based on the fact that, for two integers a, b,

$$a < b \iff (a \texttt{ div } 2 < b \texttt{ div } 2) \vee ((a \texttt{ div } 2 = b \texttt{ div } 2) \wedge (a \texttt{ mod } 2 < b \texttt{ mod } 2)).$$

That is, a is less than b if either the most significant bits (all bits but the least significant bits) of a are less than the most significant bits of b, or the most significant bits of a and b are the same, and the least significant bit of a is less than the least significant bit of b.

(Note that the most significant bits of a number are not the same thing as the most significant bit. For example, given the number 12, which has binary representation 1100, the least significant bit is the rightmost 0, and the most significant bits are the left three bits – 110.)

In this section, you will write several functions that will lead up to writing a function that radixsorts a list of integers.

3.2 Representation

For this problem, we provide you with the function `toInt2 : int list -> int`, which takes some value `L : int list` such that for each `x` in `L`, either `x = 0` or `x = 1`, and evaluates to the integer whose bit representation is `L`, with the most significant bit at the leftmost end.

Examples:

- `toInt2 [] = 0`.
- `toInt2 [1] = 1`.
- `toInt2 [1,1] = 3`.
- `toInt2 [0,1,1] = 3`, as leading zeros do not change the value of a bit string.
- `toInt2 [1,0,1] = 5`.
- `toInt2 [1,0,0,0] = 8`.

For this problem, we will represent an integer as an `int * (int list)`, such that `(x, ds)` represents the integer $x * 2^{\texttt{length } ds} + (\texttt{toInt2 } ds)$.

For example, the valid ways of representing the number eleven are the following:

- (11,[])
- (5,[1]), as $5 * 2 + (1) = 11$.
- (2,[1,1]), as $2 * (2^2) + (3) = 11$.
- (1,[0,1,1]), as $1 * (2^3) + (3) = 11$.
- (0,[1,0,1,1]), as $0 * (2^4) + (11) = 11$.

The first and last of these are special cases, in that they contain all of the information about the integer in one element of the pair.

In the first case, any `x:int` can be represented as `(x,[])`, and in the last, and `x:int` can be represented by `(0,ds)` for some `ds:int list`. This `ds` will be the binary representation of `x`, as written with the most significant bit at the left-hand side of the list.
The first is useful in that it is easy to generate from a list of integers, and so it is what we will consider the starting point of our sorting function.

Task 3.1 (4%). Write the function `rep : (int * int list) -> int` that takes in some `(x,ds)` and evaluates to $\texttt{x} * 2^{\texttt{length ds}} + (\texttt{toInt2 ds})$.

Task 3.2 (3%). Now, write the function

```
divmod:(int * int list) -> (int * int list)
```

such that for all values `(x : int,ds : int list)`,

```
divmod (x,ds) = (x div 2,(x mod 2)::ds)
```

3.3 Partitioning

Next, you will write the function `partition` such that for all types `t`, total functions `p : t -> bool`, and values `L : t list`

```
partition p L = (L1,L2),
```

such that

- `L1 @ L2` is a permutation of `L`.
- For each `x` in `L1`, `p x = true`.
- For each `x` in `L2`, `p x = false`.
- For `x,y` in `L` such that `x` is before `y` in `L`, if `p x = p y`, then `x` is before `y` in `L1 @ L2`.

For example,

- `partition (fn x => true) [] = ([], [])`
- `partition (fn x => x mod 2 = 0) [0,1,2,3,4,5] = ([0,2,4], [1,3,5])`
- `partition (fn x => x = []) [[],[1],[],[2],[3,4],[]] = ([[],[],[]],[[1],[2],[3,4]])`

Task 3.3 (8%). Write the function `partition`.

3.4 The sort

For this section, we also provide the function

```
allZeros :  (int * int list) list -> bool,
```

such that for all values `L : (int * int list) list`,

```
allZeros L = b,
```

where `b = true` if and only if for all `(x,ds)` in `L`, `x = 0`, and `b = false` otherwise.

Using this function, and the other functions that you have written for the previous parts of task 3, write the function `rad`, described below.

`rad` should be a function such that for all values

```
L = [(x1,ds1),...,(xk,dsk)] :  (int * (int list)) list
```

such that for all $i, j \in \mathbb{N}, 1 \leq i, j \leq k$, `length dsi = length dsj`, and for each `d` in `dsi`, either `d = 0` or `d = 1`.

```
rad [(x1,ds1),...,(xk,dsk)] = [(0,bits_1),...,(0,bits_k)] = L',
```

such that

- `map rep L'` is a sorted permutation of `map rep L` using the comparison `fn (x,y) => Int.compare (rep x, rep y)`
- For each `(x,ds)` in `L'`, `x = 0`.

Further, for any initial call to `rad` (that is, a call that is not a recursive call inside `rad` itself), for each element `(x,ds)` of the input list, `ds` must be equal to `[]`.

You may use `@`, but should not write any helper functions other than the ones that you have written in the earlier tasks in this section.

Examples:

- `rad [] = []`
- `rad [(5,[])] = [(0,[1,0,1])]`
- `rad [(5,[]),(7,[]),(6,[])] = [(0,[1,0,1]),(0,[1,1,0]),(0,[1,1,1])]`
- `rad [(3,[]),(5,[]),(4,[])] = [(0,[0,1,1]),(0,[1,0,0]),(0,[1,0,1]]`

Task 3.4 (9%). Write the function `rad`. This does not need to take into account negative numbers.

Now, write the function `radixsort` that takes some `L : int list` and evaluates to `L' : int list` such that `L'` is a sorted permutation of `L`, using the functions defined above (In particular, the actual sorting procedure should be done by `rad`.)
Examples:

- `radixsort [] = []`
- `radixsort [1,2,3] = [1,2,3]`
- `radixsort [1,3,42,9001,314,217,54] = [1,3,42,54,217,314,9001]`

Task 3.5 (3%). Finally, write the function `radixsort`.

4 Insertion and `map`

4.1 Insertion

You have seen the function `ins` before in lecture, as used in insertion sort. Its code is provided here for use in the next problem.

```
(* ins : ('a * 'a -> order) -> ('a * 'a list) -> 'a list
 * REQUIRES: L is a cmp-sorted list
 * ENSURES: ins cmp (x,L) = a cmp-sorted list consisting of
 * x and all of the elements of L.
 *)
fun ins cmp (x, []) = [x]
  | ins cmp (x,y::R) = case cmp (x,y) of
                         GREATER => y::(ins cmp (x,R))
                       | _ => x::(y::R)
```

4.2 Proof

Now, using this, you will prove the the following theorem:
Theorem 1: *Let `t1`, `t2` be types, let `cmp1` and `cmp2` be comparison functions for `t1` and `t2`, respectively, and let `f` be a total function of type `t1 -> t2` such that for all values `x,y : t1`, `cmp1(x,y) = cmp2(f x, f y)`.*

Then, for all values `x : t1` and `L : t1 list` such that `L` is cmp-sorted,

```
map f (ins cmp1 (x,L)) = ins cmp2 (f x, map f L).
```

State clearly in your proof if and where you use the assumption that `f` is total.

You may assume these obvious lemmas about `map`, which follow from its definition:

Lemma 1: For all total functions `f`, `map f [] = []`.

Lemma 2: For all types `t1, t2`, total functions `f : t1 -> t2`,
and values `x::xs : t1 list`, `map f (x::xs) = (f x)::(map f xs)`.

Task 4.1 (20%). Prove **Theorem 1**.

5 Continuations

5.1 Background

Continuations are a variant of accumulator variables that happen to be functions.

5.2 An Example

The following code gives three different recursive implementations of a function to find the product of a list, short-circuiting when a 0 is reached. The first is a straightforward recursive function, the second uses tail recursion with an accumulator, and the third uses a continuation to accumulate information.

```
(* Standard recursion *)
fun prod ([] : int list) : int = 1
  | prod (0::xs) = 0
  | prod (x::xs) = x * (prod xs)


(* Tail recursion with accumulator a *)
fun prodthelp ([] : int list, a : int) : int = a
  | prodthelp (0::xs, a) = 0
  | prodthelp (x::xs, a) = prodthelp (xs, x * a)
fun prodt L = prodthelp (L,1)


(* Tail recursion with a continuation k *)
fun prodchelp ([] : int list, k : int -> int) : int = k 1
  | prodchelp (0::xs, k) = k 0
  | prodchelp (x::xs, k) = prodchelp (xs,(fn y => k (x * y)))
fun prodc L = prodchelp (L,(fn x => x))
```

For the next task, you will step through the code for `prodc`, to help you to better understand how continuations work.

Task 5.1 (10%). Show using evaluational reasoning that `prodc [1,0,3]` $\Rightarrow^*$ 0. Include enough detail to make sure that the order in which subexpressions and function calls are evaluated is very clear.

6 Combination Continuation

In class this week, you learned a bit about continuations, functions that are used like accumulators for control flow inside a function (as well as other uses you will see later). In this question, you will be using continuations to determine if a target number can be produced by combining a source list of other numbers using only + and *.

There are a few special conditions for this combination.
First, for convenience, we will consider only nonzero integers as possible targets or elements in the source list.
Second, every element of the source list must be used in the combination.
Third, the precedence of operations is strictly right-precedence. For example, for the target number 12 and the source list [4,1,2], we can use 4*(1+2) to produce 12, but for target 12 and source [5,7,8], no combination exists. Additionally, 1+2+3*4 = 1+(2+(3*4)) = 15.

6.1 All of the Things

One very inefficient way to discover if the target number can be produced is to make a list of all possible combinations of the source list and see if the target is contained within. The following code for `combos` does precisely this.

```
fun combos ([] : int list) : int list = []
  | combos [x] = [x]
  | combos (x::R) =
    let
      val C = combos R
    in
      (map (fn c => x + c) C) @ (map (fn c => x*c) C)
    end
```

For example, `combos [1,2,3]` will give `[6,5,7,6]`. Not only does this version give every possible combined number, it can give the same number multiple times! There must be a better way to do this.

6.2 Optionally, Some of the Things

In addition to generating an entire list when we are only looking for a single element, the previous solution does not give us any indication as to which operations we used to make our target number. With the power of option types, we can either return `NONE` if no combination exists, or `SOME(x)`, where x is a list of operations, if it does.

Task 6.1 (10%). Write a function `make: int -> int list -> string list option` such that for all values `x : int`, `L : int list`,

`make x L = s`,

where `s = NONE` if no combination (as described above) of `L` can produce `x`, and `s = SOME(y)` otherwise, where `y` is a list of "*" and "+" that indicate how to combine the integers of the `L` to produce the `x`.

For example, `make 7 [1,2,3]` should evaluate to `SOME["+","*"]`, as $1 + (2 * 3) = 7$.

6.3 Succeeding (And Failing) at the Things

Finally, there is one more way to find out if a combination exists (and give a solution if it does). We can use continuations! In this final method, we will make use of two continuations: one that we will use if the combination we're currently trying is working, and one if it is not.

Task 6.2 (15%). Write

```
make_C: int -> int list -> (string list -> 'a ) -> (unit -> 'a) -> 'a
```

such that for all types `t`, values `x : int`, `L : int list`, and total functions `f : string list -> t` and `k : unit -> 'a`,

`make_C x L f k = y`,

such that if there is some combination of `L` that produces `x`,
then `y` will be equivalent to `f` applied to the string list of operations (As in 6.3).
If no such combination exists, then `y = k ()`.

`f` is called the success continuation, and `k` the failure continuation.

Task 6.3 (5%). Give an example of a top-level call to `make_C` and what this call will evaluate to.