15-150 Spring 2016
Lab 6

18 February 2015

1 Introduction

The goal for the this lab is to make you more familiar with polymorphic types and option types. It is also to introduce you to the idea that **functions are values** and to prepare you for the material that will be covered in lecture tomorrow.

Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this week's lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You must use the five step methodology for every function you write on this assignment. In particular, every function you write should have a REQUIRES and ENSURES clause and tests.

2 Types

Task 2.1 For each of the following expressions, what is its most general type? Recall that `map` has type `('a -> 'b) -> 'a list -> 'b list`. If you think the expression is not well-typed, say so.

(a) `(fn x => x+1.0)`

(b) `map (fn x => x ^ "Hello")`

(c) `map (fn x => x + 1) [41]`

(d) `map map`

(e) `foldl (op o)`

(f) `[]::[]`

(g) `map foldr`

(h) `foldr map`

3 Higher-Order Functions on Lists

The `foldr` function was defined in class. Here is its type and definition:

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b


fun foldr g z []     = z
  | foldr g z (x::L) = g(x, foldr g z L);
```

`foldr` can be used in place of recursive functions. For instance, take a look at the function `sum` that takes an int list and computes the sum of its elements:

```
fun sum (L : int list) : int =
  case L of
    [] => 0
  | x::xs => x + (sum xs)
```

We can rewrite this function using `foldr` without recursion as:

```
fun sum' (xs : int list) : int = foldr (fn (x,y) => x + y) 0 xs
```

3.1 Proving with Higher-Order-Functions

Task 3.1 Prove that the two implementations of `sum` are equivalent. That is, prove the theorem

Theorem: For any `L : int list`, `sum L` $\cong$ `sum' L`.

Your proof should use structural induction and equational reasoning.
You may assume the following lemma:
Lemma 1: If `f` is total, then `foldr f b L` evaluates to a value for all values `b` and `L`.

3.2 Quantifiers

Task 3.2 Using `foldr`, write

```
exists : ('a -> bool) -> 'a list -> bool
forall : ('a -> bool) -> 'a list -> bool
```

such that when `p` is a total function of type `t -> bool`, and `L` is a list of type `t list`:

- `exists p L` $\Longrightarrow$ `true` if there is an `x` in `L` such that `p x` $\cong$ `true`;
  `exists p L` $\Longrightarrow$ `false` otherwise
- `forall p L` $\Longrightarrow$ `true` if `p x` $\cong$ `true` for every item `x` in `L`;
  `forall p L` $\Longrightarrow$ `false` otherwise.

Hint: If you're having trouble writing these functions with `foldr`, try writing them recursively first and then changing them to use `foldr`, like we did with `sum`.

4 Higher-Order Functions on Trees

Recall our definition of binary trees:

```
datatype 'a tree = Empty
                 | Node of 'a tree * 'a * 'a tree
```

4.1 Implementation

We will be working with some higher-order functions on these trees.

Task 4.1 Define a recursive ML function

```
treeExists : ('a -> bool) -> 'a tree -> 'a option
```

such that `treeExists p t` evaluates to `SOME e` where `e` is any element of `t` that satisfies `p` and `NONE` if no such element exists.

Task 4.2 Define a recursive ML function

```
treeAll : ('a -> bool) -> 'a tree -> bool
```

such that `treeAll p t` evaluates to `true` if and only if every element of `t` satisfies `p`. Please do not use `treeExists`.

Task 4.3 Now let's try a more general higher-order function. A common pattern is to take a tree and combine all the elements two at a time until you end up with a single value. This operation is called `reduce` and is similar to `foldl` on lists (but notice the different types).

Define a recursive ML function

```
treeReduce :  ('a *'a -> 'a) -> 'a -> 'a tree -> 'a
```

such that `treeReduce f b t` uses the function f to combine the values of t and returns the base case b wherever it sees an empty tree. In the node case you will need to combine 3 values - you can combine them with f in whichever order you like (assume f is associative).

Task 4.4 Define a recursive ML function

```
treeFilter :  ('a -> bool) -> 'a tree -> 'a tree
```

such that `treeFilter f t` uses the function f to filter the elements of t. This means `treeFilter f t` should evaluate to a tree with all the elements in t that satisfy f.

5 Staging

Recall from yesterday's lecture the concept of *staging* - sometimes a function can do useful work before it gets all of its arguments. For example, say we have an unsorted list and want to know the i'th-smallest element. It would be wasteful to sort the list every time we ask for an element, so instead we can write a staged function `nthSmallest` that accepts a list, sorts it, then waits for an index before returning the i'th element.

Task 5.1 Define a function

```
nthSmallest : int list -> int -> int
```

that behaves as described above. You should use the provided function `quicksort` to sort the list and the built-in function `List.nth : 'a list * int -> 'a` to look up elements.