15-150 Fall 2014
Lab 07

Thursday 16th October, 2014

This goal of this lab is to give you familiarity with higher-order functions and their applications.

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

1 Introduction

1.1 Getting Labs

We will be distributing the text and any starter code for the labs using Autolab. Each week's lab will start being available at the beginning of class.

On the Autolab page for the course, the current lab is the last entry under `Lab`. Say it is called "`labnn`". Click on this link. There, two links matter

- View writeup: this is the text of the lab in PDF format.
- Download handout: this is the starter code, if any, for the lab. It will always be distributed as a compressed archive (in `.tgz` format). Uncompressing it will create the following directories:

`lab`*`nn`*`/`	Directory for lab *nn*
`code/`	Code directory for lab *nn*
`*.sml`	Starter files for lab *nn*
`handout.pdf`	Copy of writeup for lab *nn*

1.2 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need

to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

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

1. In the first line of comments, write a call template 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 (include type annotations for the arguments and result of the function)
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  factorial (n) ==> res
 *  REQUIRES:  n >= 0
 *  ENSURES: res is  n!
 *)
fun factorial (0: int): int = 1
  | factorial n = n * factorial (n-1)

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

2 Quantifiers

For this task, we will use the following definition for trees.

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

Recall that we defined the function `reduce` on `tree`'s as follows:

```
fun reduce (f: 'a * 'b * 'a -> 'a) (e: 'a) (Empty: 'b tree): 'a = e
  | reduce f e (Node(tL,x,tR)) = f (reduce f e tL, x, reduce f e tR)
```

Task 2.1 Write a function `Exists: ('a -> bool) -> 'a tree -> bool` using `reduce` such that, when `p` is a total function of type `'a -> bool` and `T` is a tree of type `'a tree`, we have that:

- `Exists p T` $\cong$ `true` if there is an `x` in `T` such that `p x` $\cong$ `true`,
- `Exists p T` $\cong$ `false` otherwise.

Task 2.2 Write a function `Forall: ('a -> bool) -> 'a tree -> bool` using `reduce` such that, when `p` is a total function of type `'a -> bool` and `T` is a tree of type `'a tree`, we have that:

- `Forall p T` $\cong$ `true` if `p x` $\cong$ `true` for every item `x` in `T`,
- `Forall p T` $\cong$ `false` otherwise.

Task 2.3 Write the functions `exists: ('a -> bool) -> 'a list -> bool` and `forall: ('a -> bool) -> 'a list -> bool`, which have analogous specs to the previous two tasks, except they operate on lists. Use the predefined function `foldr` for this task (remember, that's the list version of `reduce`).[1]

[1] These functions are predefined as `List.exists` and `List.forall`, but of course you should not use them!

3 Tabulate

In this section, we will write and use several new higher-order functions on lists.

Task 3.1 Define the SML function

```
tabulate: (int -> 'a) -> int -> 'a list
```

such that `tabulate f n` returns a list of `n` elements such that each element `x` of the list, starting at index 0, has value `f x`, up to `n-1`.[2]

Task 3.2 Using `tabulate`, write the function `evens: int -> int list` such that `evens n` returns a list of the first n natural numbers.

Task 3.3 Using `tabulate` and other higher-order functions on lists, write `fact: int -> int` such that `fact n` is the factorial of `n`.

[2] A similar function is predefined as `List.tabulate`, but of course you should not use it!

4 Map-Reduce

In this section, we will write and use several new higher-order functions on trees.

Task 4.1 For this task, we will use the definition of trees from the previous section. Write the function

```
mapreduce: ('a -> 'b) -> ('c * 'b * 'c -> 'c) -> 'c -> 'a tree -> 'c
```

such that `mapreduce f g e t` maps the function `f` to the elements of `t` and then reduces the results using `g` and the base case `e`. You may do this recursively or using the higher order functions on trees we have discussed in lecture (namely, `map` and `reduce`).

Task 4.2 Using `mapreduce`, define the SML function

```
totalLength: string tree -> int
```

that, given a tree `t` with strings as data, return the sum of the length of all the strings in `t`. You may use the predefined function `String.size: string -> int` that returns the length of a string.

Checkout point!

Completing everything up to here in the lab assignment will guarantee credit for this lab.

Click here or go to the class schedule and click on a `Check me in` button.

5 Higher-Order Complexity

In this section, we analyze the complexity of `map` and `reduce` on trees. For all tasks, you may assume the trees are balanced. Additionally, assume that all functions passed to `map` and `reduce` are $O(1)$.

For reference, here is the code for `map` and `reduce`.

```
fun map f Empty = Empty
  | map f (Node(tL,x,tR)) = Node(map f tL, f x, map f tR)

fun reduce f e Empty = e
  | reduce f e (Node(tLl,x,tR)) = f (reduce f e tL, x, reduce f e tR)
```

Task 5.1 Write a recurrence for the *work* of `map`, then solve for its complexity in big-O notation.

Task 5.2 Write a recurrence for the *span* of `map`, then solve for its complexity in big-O notation.

Task 5.3 Write a recurrence for the *work* of `reduce`, then solve for its complexity in big-O notation.

Task 5.4 Write a recurrence for the *span* of `reduce`, then solve for its complexity in big-O notation.

