15-150 Fall 2013
Lab 6

2 Oct 2013

1 Introduction

The goal for this lab is to make you more familiar with higher-order functions, polymorphism, and currying in Standard ML.

Please take advantage of this opportunity to practice writing functions and 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.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks 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 writing functions for every function you write on this assignment. In particular, every function you write should have REQUIRES and ENSURES clauses and tests.

2 Higher Order Functions with Polymorphism

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 (fn l => map (fn x => x) l)`

(e) `map map`

Solution 2.1

(a) `real -> real`

(b) `string list -> string list`

(c) `int list`

(d) `'a list list -> 'a list list`

(e) `('a -> 'b) list -> ('a list -> 'b list) list`.

3 Folding a List

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 = sum' L`.

Your proof should use structural induction and equational reasoning.

Solution 3.1 **Theorem:** For any `L : int list`, `sum L = sum' L`.

Proof:

Base Case: `L = []`

Need to Show: `sum [] = sum' []`

Let us first step the left side (`sum []`) :

`sum [] = 0` *step*

Let us also step the right side (`sum' []`) :

`sum' []`	=	`foldr (fn (x,y) => x + y) 0 []`	*step*
	=	`0`	*step*

By transitivity of =, this implies `sum [] = 0 = sum' []`. So by the transitivity `sum [] = sum' []` and the theorem holds in the base case.

Inductive Hypothesis: Assume for some `L : int list` that `sum R = sum' R`.

Indutive Case: `L = x::R`
Need to Show : `sum (x::R) = sum' (x::R)`

Let us first step the left side (`sum (x::R)`):

`sum x::R` = `x + sum R` *step*

Let us also step the right side (`sum' (x::R)`) :

`sum' x::R`	=	`foldr (fn (x,y) => x + y) 0 (x::R)`	*step*
	=	`(fn (x,y) => x + y) (x, foldr (fn (x,y) => x + y) 0 R)`	*step*

Notice that with one step of evaluation `sum' R = foldr (fn (x,y) => x + y) 0 R`. So by referential transparency we can substitute in the above to get

`sum' x::R`	=	`(fn (x,y) => x + y) (x, sum' R)`	*referential trans.*
	=	`x + sum' R`	*step*
	=	`x + sum R`	*Inductive Hypothesis*

By transitivity of =, `sum x::R = sum' x::R`, so the inductive case holds.

By structural induction on `L`, `sum L = sum' L` for any `L : int list`.

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` $=$ `true`;
  `exists p L` $\Longrightarrow^*$ `false` otherwise
- `forall p L` $\Longrightarrow^*$ `true` if `p x` $=$ `true` for every item `x` in `L`;
  `forall p L` $\Longrightarrow^*$ `false` otherwise.

Hint: Write these functions recursively at first, and then convert them to use `foldr` as was done with `sum`.

$\boxed{\textbf{Solution 3.2}}$ See solutions in `lab06-sol.sml`.

4 Higher Order 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

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

such that `treeFilter p t` keeps tree elements that satisfy `p` by wrapping them in `SOME` while replacing those elements that fail with `NONE`.

Solution 4.1 See solutions in `lab06-sol.sml`.

Task 4.2 Define a recursive ML function

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

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

Solution 4.2 See solutions in `lab06-sol.sml`.

Task 4.3 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 `treexists`.

Solution 4.3 See solutions in `lab06-sol.sml`.

Task 4.4 Define an ML function

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

that is non-recursive but works identically to `treeAll`. You may use `treexists`.

Solution 4.4 See solutions in `lab06-sol.sml`.

4.2 Polymorphism

Task 4.5

(a) What is the most general type of the following function?

```
fun foo t = treeFilter (fn [] => false | x::L => true) t
```

(b) What does it do?

Solution 4.5

(a) The most general type is `'a list tree -> 'a list option tree`.

(b) This takes a tree of lists and keeps only nonempty lists (by wrapping them in `SOME`).

4.3 Trees on trees

Task 4.6 Please define an ML function

```
onlyEvenTrees : (int tree) tree => (int tree option) tree
```

such that `onlyEvenTrees t` evaluates to a tree that has `NONE` wherever `t` had a tree containing any odd number and `SOME e` wherever `t` had a tree `e` containing no odd numbers.

Solution 4.6 See solutions in `lab06-sol.sml`.

4.4 Do the safetree dance

Task 4.7 We were perfectly happy with our tree implementation until some `c hax0rs` mutated our trees. Please write a non-recursive function:

```
safetree : int tree -> int option tree
```

which transforms each `Leaf(n)` to `Leaf(NONE)` if `n = 0`, and `Leaf(SOME n)` otherwise.

Solution 4.7 See solutions in `lab06-sol.sml`.

5 More Trees

Oftentimes we want a tree with more than two branches at any node. For example, the B-trees used to implement your filesystem have arbitrary branching factor (post-lab reading for the curious: en.wikipedia.org/wiki/B-tree)!

We can extend our definition of binary trees to trees with an arbitrary branching factor with the following datatype:

```
datatype 'a narytree = Emp
                     |  Bud of 'a
                     |  Branch of 'a narytree list
```

Notice how we already have an `Emp` type in the tree – to avoid being redundant, we will impose an invariant requiring the `'a narytree list` in `Branch` to be non-empty.

Task 5.1 Remember making "full" `rtrees` with `geometricTree` on HW 4? Note that it can only make a tree with 2^n leaves for a given n - it's pretty boring. Define an ML function

```
fuller : (int * int) -> int narytree
```

that for some non negative `n` and positive `a`, returns an n-ary tree of depth `a`, with n^a leaves each containing your favorite number.

Solution 5.1 See solutions in `lab06-sol.sml`.

5.1 Higher-Order Functions (again)

If you haven't already guessed, we can extend the same higher order functions on binary trees to n-ary trees!

Task 5.2 Define an ML function

```
narytreemap : ('a -> 'b) -> ('a narytree -> 'b narytree)
```

for applying a function to every leaf in an n-ary tree.

Solution 5.2 See solutions in `lab06-sol.sml`.

Task 5.3 Define an ML function

```
narytreereduce : ('a * 'a -> 'a) -> 'a -> 'a narytree -> 'a
```

for combining the items at the leaves of an n-ary tree with some base value.

Solution 5.3 See solutions in `lab06-sol.sml`.

Task 5.4 Define an ML function

```
narytreemapreduce : ('a -> 'b) -> ('b * 'b -> 'b) -> 'b -> 'b narytree -> 'b
```

for mapping a function to every leaf in an n-ary tree and then combining the resulting leaves with a base value. (This should be non-recursive.)

Note: There is at least one implementation of mapreduce which stores a tree-sized intermediate result. You should probably try this first! Challenge: can we do without this intermediary?

$\boxed{\textbf{Solution 5.4}}$ See solutions in `lab06-sol.sml`.