15-150 Fall 2013
Lab 5

25 September 2013

1 Introduction

The goal for the this lab is to make you more familiar with polymorphic types and option types, as well as higher order functions. Higher order functions are strongly related to the idea that **functions are values** because higher order functions either take in values that are of function type, or they produce values that are of function type. This practice will also 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.

Remember to follow the methodology for writing functions- specifications and tests are part of your code! When writing tests for functions that can raise exceptions, you currently only need to write tests for the cases that don't raise exceptions.

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 a REQUIRES and ENSURES clause and tests.

2 Warmup

Task 2.1 Give the most general (possibly polymorphic) type for the following functions and expressions, or state that the function/expression is ill-typed.

1. `fn x => x + 3.0`

   Solution 2.1 `real -> real`

2. ```
   let
       fun fact x = "functions are " ^ x
   in
       fact "values "
   end
   ```

   Solution 2.1 `string`

3. `fn (f,(x,y)) => f(x,y)`
   Hint : Because of the part of the function after `=>`*, we know that* `f : 'a * 'b -> 'c` *. In English, we know that* `f` *is a function that takes in a tuple whose elements have types that we don't know, and we also don't have any constraints on the type that* `f` *evaluates to either.*

   Solution 2.1 `('a * 'b -> 'c) * ('a * 'b) -> 'c`

4. `fun f (x,y,z) = (f(x,y),z)`

   Solution 2.1 This function is ill-typed.
   To see this, assume that the function was not ill-typed. Then, it must have type `'a*'b*'c -> 'd`. But, looking at the body of the function, `f` only takes `x,y` which will cause `tycon mistmatch`.

5. ```
   fun zip ([], _) = []
     | zip (_, []) = []
     | zip (x::xs, y::ys) = (x,y) :: zip (xs,ys)
   ```

   Solution 2.1 `'a list * 'b list -> ('a * 'b) list`

6. (Bonus)

   Work on this only after you're done with most of the lab!

   ```
   (fn y => (fn (f,x) => f(x)) ((fn x => y), 150))
   ```

Solution 2.1 `'a * 'a`

Task 2.2 How many different total functions of polymorphic type `'a -> 'a` can there be? (By "different" we mean "not extensionally equivalent".) List all of them.

Solution 2.2 One, the identity function.

3 Higher Order Functions

Anonymous functions are function literals that are not explicitly bound to a name. They provide a useful way to define functions inline and will be used extensively with curried and higher-order functions both of which will be discussed in lecture tomorrow. Anonymous functions have the following syntax:

`(fn (x,y) => x + y)` is an anonymous function that adds two integers.

Since functions are values, the same function can also be defined as follows:

```
val add = (fn (x,y) => x + y)
```

Task 3.1 Implement the following as anonymous functions and then bind each to the given variable name.

1. `val double: int -> int` that doubles an integer value
2. `val quadruple: int -> int` that multiplies an integer by four, without using arithmetic operations

Task 3.2 Implement function `val thenAddOne : ((int -> int) * int) -> int` that either doubles, quadruples, some other trasformation to a number and then adds one to it.

Hints:

- The first line of your function will look like

  ```
  fun thenAddOne ((f:int -> int), (x:int)) : int =
  ```

  This means that `thenAddOne` is a function such that if you give it the function `f` and an integer x, then it will produce an integer.

- You can (and should) use `f` inside of `thenAddOne`.

[the lab continues on the next page]

Consider the following functions that take an integer list as argument and map the list such that a function is applied to each element:

```
fun doubleList ([ ] : int list) : int list = [ ]
  | doubleList (x::xs : int list) : int list =
     (double x)::(doubleList xs)

fun quadrupleList ([ ] : int list) : int list = [ ]
  | quadrupleList (x :: xs: int list) : int list =
     (quadruple x)::(quadrupleList xs)
```

Both these functions follow a similar pattern and only differ in the function applied to each element of the `int list`. Could we, then, generalize the function such that it takes in an `int list` and a function as arguments and maps the function to each element of the list?

Indeed, this is possible, since functions are values that can be passed to, and returned from other functions. Functions that accept other functions as arguments or return them as results are called **Higher-Order Functions**. Higher-Order Functions are very powerful and allow functional languages to do amazing things in beautifully short lines of code!

Task 3.3 Write a higher-order function `mapList` that generalizes the functions `doubleList` and `quadrupleList` above. It will have the following type:

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

Task 3.4 Write a higher-order function `mapList'` that also maps a function across a list, but has a slightly different type.

```
mapList':  ('a -> 'b) -> ('a list -> 'b list)
```

Note that this function differs from the previous one in that it returns a function of type `'a list -> 'b list`. This concept, called **currying**, is especially useful when dealing with higher-order functions because it lets you apply partial parameters to a function and still evaluate to a value!

Task 3.5 Test `mapList` and `mapList'`using the functions you wrote in Task 3.1. You can also test it by writing anonymous functions inline within the test case. Try both to get used to the syntax!

Have the TAs check your code for `mapList` and `mapList'` before proceeding!

4 Options

Recall that SML has a built in datatype `'a option` defined as follows:

```
datatype 'a option = NONE
  | SOME of 'a
```

Intuitively, options let us deal with computations that may not have a result. As such, the `SOME(x)` case represents the succesful computation of a value and the `NONE` case represents a failure to compute the value. Note that a failure here is not necesarily a bad thing, it's just a case that we have to handle. Using options lets us avoid many nasty bugs like null pointer dereferencing.

Task 4.1 Define the function

```
findOdd: int list -> int option
```

such that `findOdd L` returns `SOME x` where x is the first odd number in L or `NONE` if L contains only even numbers. Note you may find the built-in SML function `mod` useful here.

Subset Sum Revisited

Recall from homework 3 the function `subsetSumCert`. For convenience, the source code is reproduced below:

```
fun subsetSumCert(nil : int list, s : int) : bool * int list = (s=0,nil)
  | subsetSumCert(x::xs, s) =
      case subsetSumCert(xs, s - x) of
        (true, l1) => (true, x::l1)
      | (false, _) => subsetSumCert(xs,s)
```

Observe that when `subsetSumCert(L,s)` returns false, there is no subset of L that sums to `s`. Still, because of the type definition, we're required to return `(false, l)` where l is some arbitrary list. This is rather silly as we're ignoring the list all together in the false case. We can clean this code up by using option types!

Task 4.2 Define a new function

```
subsetSumOption : int list * int -> int list option
```

such that `subsetSumOption(L,s)` evaluates to `SOME L'` if there exists a subset L' of L whose elements sum to s and `NONE` if no such subset exists.

Task 4.3 Prove the following theorem:

```
For all s:int, L:int list, if subsetSumOption(L,s) == SOME(L')
then sum(L') == s.
```

You may assume that `sum` is defined as follows:

```
fun sum [] = 0
  | sum x::xs = x + sum xs
```

Case for `nil`
To show:

Case for `x::xs`
Inductive Hypothesis

To Show:

Solution 4.3

Case for `nil`
To show:
∀`s:int` if `subsetSumOption(nil,s) == SOME(L')` then `sum(L') == s`

We proceed by casing on `s`.

1. `s` = 0 Then by stepping we have:

```
     subsetSumOption([],0)
==>  SOME []                    (by first clause)
```

   Since we have `L' = []` and since `sum [] ==> 0` we have shown the claim.

2. `s` ≠ 0 Then by stepping we have:

```
     subsetSumOption([],s)
==>  NONE                       (by second clause)
```

   Since we have `subsetSumOption([],s) == NONE`, the implication is trivially true.

Case for `x::xs`
Inductive Hypothesis:
∀`s:int` if `subsetSumOption(xs,s) == SOME(L) then sum(L) == s`

To Show:
∀`s':int` if `subsetSumOption(x::xs,s') == SOME(L')` then `sum(L') == s'`

We proceed by stepping

```
     subsetSumOption(x::xs,s')
==>  case subsetSumOption(xs,s'-x) of
         SOME(L) => SOME(x::L)
       | NONE => subsetSumOption(xs, s')
```

Analyzing each case:

1. `subsetSumOption(xs,s'-x) == SOME(L)`
   By the induction hypothesis (taking `s` = `s'-x`, `L` = `L`) we have `sum(L) == s'-x`.

By stepping, we see `sum(x::L) ==> x + sum L == x + s' - x == s'`. It follows that `subsetSumOption(x::xs,s') == SOME(L')` (where `L' == x::L` here) and `sum(L') == s'` and we are done.

2. `subsetSumOption(xs,s'-x) == NONE`
   Then we have `subsetSumOption(x::xs,s') == subsetSumOption(xs,s')` (by the second case).
   If this is NONE then we are done (since we have nothing to prove for the NONE case). Consider the non-trivial case where `subsetSumOption(xs,s') == SOME(L')`. By the **IH** (taking `s = s',L = L'`) we have `sum(L') == s'`. Thus, `subsetSumOption(x::xs,s') == SOME(L')` and `sum(L') == s'` and so we are done.

Since the theorem holds in both cases, by the principles of induction we are done.