Higher Order Functions – Part 2
Functions as Results

COMP 302

Some Useful High Order Functions

There are several high order functions that turn out to be very useful in problem solving at a more abstract level

They can be defined over recursively defined datatypes

We look at a some of them as defined over lists

Map

An extremely useful function, map, allows us to apply a function to each element in a recursive datatype

For example, we can map a function over all values in a list:

```
map : ('a -> 'b) * 'a list -> 'b list
fun map (f lst) =
  case lst
     []  -> []
   | h::tail -> (f h)::(map f tail)
```

Examples of map

```
val x = map (increment, [4,8,12,16])


val y = map (hd, [[1,2],[3,4],[5,6,7]])
```

A brief glimpse at Closures

A very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope.

For example we can get the incr function we defined earlier by:

```
fun incr (lst , c) = map (fn x => x + c , lst)
```

The function `fn x => x + c` adds c to its argument, where c bound as the argument to incr

In the following evaluation by substitution the c is set to 2 and the function

```
fn x => x + 2
```

is applied to each element of [1,2,3]

A brief glimpse at Closures

```
incr ( [1,2,3] , 2)
|-> map (fn x => x + 2 , [1,2,3])
|-> (fn x => x + 2) 1 :: map (fn x => x + 2 , [2,3])
|-> 1 + 2 :: map (fn x => x + 2 , [2,3])
|-> 3 :: map (fn x => x + 2 , [2,3])
|-> 3 :: (fn x => x + 2) 2 :: map (fn x => x + 2 , [3])
== 3 :: 4 :: map (fn x => x + 2 , [3])
== 3 :: 4 :: (fn x => x + 2) 3 :: map (fn x => x + 2 , [])
== 3 :: 4 :: 5 :: map (fn x => x + 2 , [])
== 3 :: 4 :: 5 :: []
```

A brief glimpse at Closures

The important fact, which takes some getting used to, is that the function `fn x => x + 2`

Is dynamically generated at run-time.

It is a new function, which does not appear anywhere in the program's source code

A brief glimpse at Closures

Consider

```
let val x = 3
    val f = fn y => y + x
    val x = 5
in
    f 10
end
```

A brief glimpse at Closures

Within the let, evaluate the declarations in order substituting as you go

```
let val x = 3
    val f = fn y => y + 3
    val x = 5
in
    f 10
end
```

This code is tricky but the substitution model allows us to determine the meaning in spite of the fact that the first binding of x is later shadowed.

Filter

Filter allows us to extract values that satisfy some Boolean relation

```
filter : ('a -> bool) * 'a list -> 'a list

fun filter (p : 'a -> bool,  lst : 'a list) =
  case lst of
    []    => []
  | x::xs =>  if p x then x::filter p l
                     else filter p l
```

Example of Filter

Define a function to extract the strictly positive integers from an int list

```
fun pos (lst) = filter (fn n => n>0, lst)
fun is_even v =
    (v mod 2 = 0)

fun get_all_even lst =
    filter(is_even,lst)
```

Alternatively:

```
val get_all_even =
    fn lst => filter((fn (s,v) => v mod 2 = 0), lst)
```

Functions over datatypes

We can also define functions over user-defined datatypes

For example the following datatype defines simple arithmetic expressions

```
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp
```

Functions over datatypes

This function determines if all constants in the expression satisfy some property

```
fun true_of_all_constants(f,e) =
    case e of
        Constant i => f i
      | Negate e1 => true_of_all_constants(f,e1)
      | Add(e1,e2) => true_of_all_constants(f,e1)
                      andalso true_of_all_constants(f,e2)
      | Multiply(e1,e2) => true_of_all_constants(f,e1)
                           andalso true_of_all_constants(f,e2)

fun all_even e = true_of_all_constants(is_even,e)
```

Returning Functions

Not only can we send functions as arguments to other functions, we are also able to generate new functions

Given a function on two arguments, `f(x,y)`, there is a function of a single argument f'(x) which is a function that can be applied to an argument `(f'(x) (y)) =f(x,y)`

For example

```
fun add (x, y) = x + y
```

Has the alternate form

```
fun add_c x = fn (y) => x + y
```

This is known as a Curried version of add

Currying

Currying is the concept that a function with two arguments can be viewed as a function that takes one argument and returns a function as its result

It was known in the 19th century that when studying functions it suffices to consider only functions of a single argument.

In a language such as SML that supports higher order functions, we can translate a function f of type (‘a * ‘b) -> ‘c into a function that takes a single argument of type ‘a and returns a function of type ‘b -> ‘c which can then be applied to an argument of type ‘b to give a result of type ‘c.

This technique is called currying after the logician Haskell Curry (who also has a programming language named after him). It was actually described by Shoenfinkel earlier and developed further by Curry, but it’s easier to say “currying” than “shoenfineling”

Why Curry?

It seems currying is just a trivial syntactic observation. Why bother?

There are several advantages of curried functions that we shall see:

- It is possible to apply one argument to a curried function and obtain a new function that can be passed as an argument to another function or composed (coming up soon) with other functions. This concept is known as partial application.
- The are concepts such as staged computation that allows you to do real computations between the passing of the arguments, something that can’t be done otherwise.

Curried function example

```
(* Computes x ^ y. Assumes y >= 0. *)
fun pow (x, y) : int * int -> int =
  case y of
     0 => 1
  |  _ = x * pow (x, (y - 1))
```

Curried version:

```
(* Computes x ^ y. Assumes y >= 0. *)
fun pow x : int -> int -> int =
  fn (y) => case y of
                 0 => 1
              |  _ = x * pow (x, (y - 1))
```

SML syntax

SML provides special syntactic support for curried functions

You can write multiple arguments to a function (separated by spaces) and this is taken to mean the same as the previous example

```
fun pow x y =
  case y of
     0 => 1
  |  _ = x * pow (x, (y - 1))
```

Giving this function a single argument returns a function

```
val powerOfTwo = pow 2
```

This is a function that, when given an argument a, computes 2^a

That new *partial application* function can be useful to pass to map, filter, etc

Map – Previous Version

```
map : ('a -> 'b) * 'a list -> 'b list
fun map (f lst) =
  case lst
     []  -> []
   | h::tail -> (f h)::(map f tail)
```

Map – Curried Version

```
map : ('a -> 'b) -> 'a list -> 'b list
fun map f lst =
  case lst
     []  -> []
   | h::tail -> f h :: map f tail
```

Which is the short SML syntactic version of : ...

Map – Curried Version

```
map : ('a -> 'b) -> 'a list -> 'b list
fun map f : ('a -> 'b) : 'a list -> 'b list =
  fn lst =>
    case lst
       []  -> []
      | h::tail -> f h :: map f tail
```

Powers of 2 function

```
val powersOfTwo = map powerOfTwo
```

Is a function which takes a list of integers and produces a list of 2 raised to the power of the list

```
powersof2 [2, 0, 3, 1] == [4, 1, 8, 2]
```