Higher Order Functions Part 3

COMP 302

11 HOF – PART 3

More HOF's

Now that we have been exposed to higher order functions, we will spend some time exploring them further

It is often useful to create new functions as combinations of other functions

In Mathematics, we can compose two functions to produce a new function

In SML we can define:

```
fun compose (f, g)
  = fn x => f (g x)
```

The type of compose is:

```
compose : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c
```

Composition

Using some SML library functions:

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

We can use composition :

```
fun sqrt_of_abs i = compose (Math.sqrt,compose(Real.fromInt, abs) i
```

Composition

Composition is actually defined as an infix operator in SML denoted as o (the lower case letter) so we can write:

```
fun sqrt_of_abs_v2 i = (Math.sqrt o Real.fromInt o abs) i
```

In this form it is clear that we can use composition to define new functions:

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

Pipelining

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

This definition uses the Mathematical convention that computation goes from right to left.

Many programmers prefer thinking from left to right. We can define another operator, similar to composition but “backwards”. This is often called pipelining

fun pipeline (f, g) = g f

Infix Pipeline Operator

It is possible to define the pipeline function as an infix operator |>

(Aside: this operator can cause confusion in Emacs, so we will use a variant, !> instead)

```
infix !>

fun x !> f = f x

fun sqrt_of_abs i = i !> abs !> Real.fromInt !> Math.sqrt
```

This operator is popular in some languages (e.g. F#) but it is just syntactic sugar and not very complicated or interesting

Infix Operators

When we defined our own list datatype we could have used :: in place of Cons:

```
datatype 'a list = Nil | :: of 'a * 'a list
```

To construct a list, we would write `(2, :: (5, :: (3 :: Nil)`

However, ML allows us to define :: as an infix operator and write h::t

```
infix ::
```

We can define any function of two arguments to be infix

```
infix plus
val plus = fn (x,y) => x+y
val eleven = 6 plus 5
```

Currying

We can convert between functions in curried form and in the form where they take tuples as arguments very easily

```
fun curry f = fn x => fn y => f(x,y)
curry : (('a * 'b) -> 'c) -> ('a -> 'b -> 'c)
```

The syntactic shortcuts allow us to write

```
fun curry f = fn x y => f(x,y)
```

Going one step further:

```
fun curry f x y = f (x,y)
```

Currying and Uncurrying

We can convert between functions in curried form and in the form where they take tuples as arguments very easily

```
fun curry f x y = f (x,y)
curry : (('a * 'b) -> 'c) -> ('a -> 'b -> 'c)


fun uncurry f (x,y) = f x y
uncurry :  ('a -> 'b -> 'c) -> (('a  'b) -> 'c)
```

Left or Right Associative?

At first glance there is some possible confusion about function types.

Type 'a -> 'b -> 'c, is equivalent to 'a -> ('b -> 'c): function types are right associative

On the other hand, writing f arg1 arg2 is equivalent to writing (f arg1) arg2: function application is left associative

If we look closely this duality makes sense.

If function f has type 'a -> 'b -> 'c, arg1 has type 'a.

Therefore f arg1 must be a function which has type 'b -> 'c.

Applying this function to arg2, which must be of type ‘b will return a result of type ‘c

The map-reduce Paradigm

Map operates on a data structure (such as a list) to produce a new structure in which some computation has been applied to each value in the structure

There are various reduce operations (we'll see soon) that take a data structure (such as a list) and collapses the values to a single value (or smaller number of values)

Using these operations can produce short elegant programs for problems involving collections of data

This model has been used for large-scale highly parallel processing of large datasets

Google has presented results using this paradigm for designing methodologies for processing large datasets over clusters of machines (e. g. for counting the number of occurrences of a word in each document across a collection of documents)

Reduce (aka fold, aka inject)

Back to summing a list of integers:

```
fun sum lst =
  case lst of
    []    => 0
  | h::t => h + sum t
```

How about concatenating a list of strings?

```
fun concat lst =
  case lst of
    []    => ""
  | h::t => h ^ concat t
```

Tail Recursive Versions

We will soon discuss tail recursion

Tail recursive functions are functions in which the last operation performed is the recursive step

In the previous version of sum, after the recursive call is completed, there is still an addition step to perform

In many cases (as we shall see) tail recursive functions are more efficient to implement

At the implementation level they function like loops in imperative languages

We will look at a methodology for converting functions into tail recursive form using accumulators but for now, here are examples

Tail Recursive Sum

A tail recursive sum:

```
fun sum_tr lst =
  let
    fun f(acc, lst) =
      case lst of
        []   => acc
      | h::t => f(h+acc,t)
  in
    f(0, lst)
  end
```

Tail Recursive Concat

A tail recursive concatenation of strings:

```
fun concat_tr lst =
  let
    fun f(acc, lst) =
      case lst of
        []   => acc
      | h::t => f(h^acc, t)
  in
    f("", lst)
  end
```

Abstracting Sum

The two helper functions are identical except for one character.

We can rewrite the functions to make that character (the operator) an argument

```
fun sum_tr lst =
  let
    fun f(combiner, acc, lst) =
      case lst of
        []   => acc
      | h::t => f(combiner, combiner(h,acc), t)
  in
    f(op +, 0, lst)
  end
```

Abstracting Concat

```
fun concat_tr lst =
  let
    fun f(combiner, acc, lst) =
      case lst of
        []   => acc
      | h::t => f(combiner, combiner(h,acc), t)
  in
    f(fn(x,y)=>y^x, 0, lst)
  end
```

(We don’t use op ^ because we have to switch the order of the operands)

Factoring out the common code

```
fun f(combiner, acc, lst) =
  case lst of
    []   => acc
  | h::t => f(combiner, combiner(h,acc), t)

fun sum_tr lst = f(op +, 0, lst)
fun concat_tr lst = f(fn(x,y)=>y^x, "", lst)
```

Fold

This common code that was factored out is an implementation of the function fold

Fold is a reduce operation that iterates over a list to combine the elements

This version is called foldl since it combines the list elements from left to right

```
foldl (f, acc, [x1,x2,x3,…,xn]) =
      f(xn … f(x3, (f(x2, (f(x1, acc)))…)
```

Foldl

```
fun foldl(f, acc, lst) =
  case lst of
    []   => acc
  | h::t => foldl(f, f(h,acc), t)

Example:

length lst = foldl op+ 0 (map (fn x => 1) lst)
```

Foldr

Another version of fold is called foldr since it combines the list elements from right to left

```
foldr (f, acc, [x1,x2,x3,…,xn]) =
     f(x1, (f(x2, (f(x3, … f(xn, acc)…)))
```

```
fun foldr(f, acc, lst) =
  case lst of
    []   => acc
  | h::t => f(h, foldr(f, acc, t))
```

Foldr

Foldl is tail recursive

Foldr is not tail recursive so (as we see later) its performance is not as good

However, it can be useful in cases like our concatenation of strings example

If we process the list from the right, there is no need to change the order of operands to the string concatenation operator

```
fun concat lst = foldl(fn(x,y)=>y^x, "", lst)
fun concat lst = foldr(op ^, "", lst)
```

Applications of fold

Consider the curried version of the fold operations

What familiar operations are the following functions equivalent to?

```
foldl (op ::) []
```

Applications of fold

Consider the curried version of the fold operations

What familiar operations are the following functions equivalent to?

```
foldl (fn (_,a) => a+1) 0
```

Applications of fold

Consider the curried version of the fold operations

What familiar operations are the following functions equivalent to?

```
foldr (fn (x,a) => (f x)::a) []
```

Applications of fold

Consider the curried version of the fold operations

What familiar operations are the following functions equivalent to?

```
foldr (fn(x,a) =>( if f x then x::a else a) []
```