Higher Order Functions Part 1

COMP 302

08 HIGHER ORDER FUNCTIONS

Functional Programming

SML is an example of a functional programming language so it's only proper that we begin looking at what it means to program by manipulating functions

Higher order functions allow us to abstract over common functionality and support the development of modular, reusable programs

Where similar functions are carried out by different pieces of code, it is helpful to combine them and abstract out the parts where they vary

HOFs help us deal with lazy evaluation and handling infinite data as well as modelling closures and objects

First Class Functions

A value is called “first class” if it can be passed as an argument to a function, returned as the result of a function and bound to a variable.

Values such as numbers are typically first class but in many languages, such as java, functions cannot be directly passed as argument, returned as results or assigned to a variable.

However, in ML functions are first class. They can be bound to variables, carried by datatypes, put into lists.

A function that takes or returns a function is called a higher order function

Root Finding by Bisection

Given a continuous function, f

It there are two points a and b where f(a) and f(b) have different signs, there is a root of f between a and b (by the intermediate value theorem of Calculus)

We can search for a root in the interval [a, b], by dividing it in half and maintaining the invariant that the sign of the function differs at the two endpoints of the interval

This leads us to the following function definition

Root Finding by Bisection Method

```
func bisection (f, a, b) =
  let mid = (a + b) / 2.0   in
    if  Math.abs (a - b) < 0.00001 then mid
    else
      if (f mid)*(f a) < 0.0
        then bisection (f, a, mid)
        else bisection (f, mid, b)
```

Some basic patterns

We often see similar patterns in programs:

```
fun sumInts (a, b) =
  if (a>b) then 0 else a + sumInts (a+1, b)
```

This computes $\sum_{i=a}^{i=b} i$.

How about $\sum_{i=a}^{i=b} i^2$

```
fun square (a) = a * a
fun sumSquares (a, b) =
  if (a>b) then 0 else square(a) + sumSquares (a+1, b)
```

More similar patterns

How about $\sum_{i=a}^{i=b} i^3$ or $\sum_{i=a}^{i=b} 2^i$

```
fun cube (a) = a * a * a

fun sumCubes (a, b) =
  if (a>b) then 0 else cube(a) + sumCubes (a+1, b)


fun exp (b,n) = if n = 0 then 1 else b * exp(b, n-1)

fun sumExp (a, b) =
  if (a>b) then 0 else exp(2,a) + sumExp (a+1, b)
```

Generalizing

All these definitions look very similar

We can abstract out the functions (square, cube, exp, etc.) that occur in this pattern to get a general sum function

```
fun sum (f, a, b) : (int -> int) * int *int -> int =
  if (a>b) then 0
           else (f a) + sum (f, a+1, b)
```

Type of sum:

```
sum: (int -> int) -> int * int -> int
```

Applying the general function

Then

```
fun sumSquares (a, b) = sum (square, a, b)

fun sumCubes (a, b) = sum (cube, a, b)
```

How about

```
sumInts (a, b) = sum (???, a, b)

sumExp (a, b) = sum (???, a, b)
```

Anonymous Functions

Do we really have to define the function square just to use it to sum the squares?

How can we define sumExp without having to define a function to compue 2^x?

The expression `fn (x:t) => e` creates an anonymous function

The return type if not declared but is inferred by the type inference system

The function declaration

```
fun square (a) = a * a
```

is actually syntactic sugar for

```
val square : int -> int = fn(a) => a * a
```

Anonymous Functions

To evaluate an anonymous function applied to an argument, you plug the value of the argument in for the variable. E.g.

```
(fn x => 2 * x) 3 |-> 2 * 3
```

Functions are values. The value of

```
fn x => x + (1 + 1)
```

is

```
fn x => x + (1 + 1)
```

You don't evaluate the body until you apply the function.

Sum with anonymous functions

```
fun sumSquares (a, b) = sum (fn (x:int) => x*x , a, b)
```

We often drop the type declaration and rely on type inference

```
fun sumSquares (a, b) = sum (fn x => x*x , a, b)
```

```
fun sumExp (a, b) = sum (fn x => exp(2,x), a, b)
```

How about our original function

```
fun sumInts (a, b) =
  if (a>b) then 0 else a + sumInts (a+1, b)
```

Replace it with

```
fun sumExp (a, b) = sum (fn x ???, a, b)
```

Sum with anonymous functions

How about our original function?

```
fun sumInts (a, b) =
  if (a>b) then 0 else a + sumInts (a+1, b)
```

Replace it with the identity function (which often comes in useful)

```
fun sumInts (a, b) = sum (fn x => x, a, b)
```

SumExp

```
fun exp (b,n) = if n = 0 then 1 else b * exp(b, n-1)
fun sumExp (a, b) =
  if (a>b) then 0 else exp(2,a) + sumExp (a+1, b)
```

Can be considered a special case of the generalized sum function

```
fun sumExp (a, b) = sum (fn x => exp(2, x), a, b)
```

Apply n times

Here is another example of a function that acts on other functions:

```
(* return f(f(...(f(x)))), where f is applied n times  *)
fun apply_n_times(f,n,x) =
  if n = 0
  then x
  else f(apply_n_times(f,n-1,x))
```

Having a bit of fun

```
fun add (n,x) = apply_n_times(fn x => x+1,n,x)
fun exp n = apply_n_times(fn x => 2*x,n,1)
```

apply-n-times is polymorphic

```
fun nth(n, x) = apply_n_times(hd, n, x)
```

(Not a very good idea. hd [] raises an exception)

Sum of odd numbers

```
sumOdd: int -> int -> int

 fun sumOdd (a, b) =
   let fun sum (a,b) =
       if a > b then 0
       else a + sum (a+2, b)
   in
     if (a mod 2) = 1 then
       sum (a,b)
     else
       sum (a+1, b)
   end
```

Generalizing Increments

To handle cases like the previous example, we can generalize the increment

Rather than just incrementing by 1 we can provide a general increment function.

```
fun sum (f, a, b, inc) =
    if (a > b) then 0
    else (f a) + sum (f, inc(a) ,b inc)


sum : (int -> int) * int * int * (int -> int) -> int
```

Generalizing Increments

```
fun sumOdds (a,b) =
  if (a mod 2) = 1 then
    sum (fn x => x, a, b, fn x => x + 2)
  else
    sum (fn x => x, a+1, b, fn x => x + 2)
```

Approximating pi

There is an infinite series approximation of π attributed to Leibniz

$$\frac{\pi}{8} = \frac{1}{1.3} + \frac{1}{5.7} + \frac{1}{9.11} + \ldots$$

This series converges very slowly so it is not of much practical use but we can compute it using the generalized summation

```
fun piBy8 = sum (fn x => 1.0 / x * (x + 2.0)),
                a, b, fn x => x + 4.0)
```

Numerical Integration

The integral of f(x) is the area below the curve f(x) in the interval [a, b].

We will use midpoint method to approximate the integral of f(x) in the interval [a,b],

This method sums a series of small rectangles

using midpoint approximation.

Integrate a function using the midpoint method:

```
fun integral (f, a, b, dx) =
  dx * sum (f, a+dx/2.0, b, (fn x => (x+dx))
```

Are products different than sums?

```
fun product (f, lo, hi, inc) =
  if (lo > hi) then ??
  else (f lo) * product (f, inc(lo), hi, inc)
```

Using 0 again (as in sum) would be a problem

Are products different than sums?

```
fun product (f, lo, hi, inc) =
  if (lo > hi) then 1
  else (f lo) * product (f, inc(lo), hi, inc)
```

We use the multiplicative identity in the base case.

Using product to define factorial.:

```
fun fact n = product (fn x => x, 1, n, fn x = x + 1)
```

Generalizing even more

```
fun series (operator, f, lo, hi, inc, identity) =
  if (lo > hi) then identity
  else operator((f lo) , series (operator,f, inc(lo), hi, inc, identity))
```

Then

```
fun sumSeries(f, a, b, inc) = series (fun x y -> x + y, f, a, b, inc, 0)
fun prodSeries(f, a, b, inc) = series (fun x y -> x * y, f, a, b, inc, 1)
```

Or, using an SML abreviation

```
fun sumSeries(f, a, b, inc) = series (op +, f, a, b, inc, 0)
fun prodSeries(f, a, b, inc) = series (op *, f, a, b, inc, 1)
```