CS 457/557 Functional Programming

Lecture 9

More on Higher-order Functions

Currying ("eta reduction")

Recall (from Ch. 1) the function: `simple n a b = n * (a+b)`

Note that:

`simple n a b` is really

`(((simple n) a) b)` in fully parenthesized notation

```
simple :: Float -> Float -> Float -> Float
simple n ::  Float -> Float -> Float
(simple n) a :: Float -> Float
((simple n) a) b :: Float
```

Therefore:

`multSumByFive a b = simple 5 a b` is the same as

`multSumByFive = simple 5`

Use of Eta Reduction

```
listSum, listProd :: [Integer] -> Integer
listSum  xs       = foldr (+) 0 xs
listProd xs       = foldr (*) 1 xs
```

```
listSum           = foldr (+) 0
listProd          = foldr (*) 1
```

```
and, or :: [Bool] -> Bool
and xs   = foldr (&&) True  xs
or  xs   = foldr (||) False xs
```

```
and      = foldr (&&) True
or       = foldr (||) False
```

Be Careful Though …

Consider:

- `f x = g (x+2) y x`

This is not equal to:

- `f = g (x+2) y`

because we lose the binding for `x`.

In general:

- `f x = e x`

is equal to

- `f = e`

only if `x` *does not appear free in* `e`.

Simplify Definitions

Recall:

```
reverse xs = foldl revOp [] xs
  where revOp acc x = x : acc
```

In the prelude we have: `flip f x y = f y x`. Thus:

```
revOp acc x = flip (:) acc x
```

or even better:

```
revOp = flip (:)
```

And thus:

```
reverse xs = foldl (flip (:)) [] xs
```

or even better:

```
reverse = foldl (flip (:)) []
```

Anonymous Functions

- So far, all of our functions have been defined using an *equation*, such as the function **`succ`** defined by:

```
succ x = x+1
```

- This raises the question: Is it possible to define a *value* that behaves just like **`succ`**, but has no name? Much in the same way that **`3.14159`** is a value that behaves like **`pi`**?

- The answer is *yes*, and it is written **`\x -> x+1`**. Indeed, we could rewrite the previous definition of **`succ`** as:

```
succ = \x -> x+1.
```

- The backslash (**`\`**) is meant to look like (and is read as) the Greek letter "lambda." Anonymous functions figure prominently in the "lambda calculus," an important foundational formalism for computation.

Sections

- Sections are like currying for infix operators. For example:

```
(+5) = \x -> x + 5
(4-) = \y -> 4 - y
```

  So in fact `succ x = x + 1` can be written more simply as `(+1)`!

- Sections also permit specifying the **right-hand** argument to an operator.

- Although convenient, sections are less expressive than anonymous functions. For example, it's hard to represent `\x -> (x+1)/2` as a section.

- You can also pattern match using an anonymous function, as in `\(x:xs) -> x`, which is the `head` function.

Function Composition

- Very often we would like to combine the effects of one function with that of another. *Function composition* accomplishes this for us, and is simply defined as the infix operator `(.)`:

```
(f . g) x = f (g x)
```

- So `f.g` is the same as `\x -> f (g x)`.

- Function composition can be used to simplify previous definitions:

```
totalSquareArea sides
  = sumList (map squareArea sides)
  = (sumList . map squareArea) sides
```

Combining this with eta reduction yields:

```
totalSquareArea = sumList . map squareArea
```