CS 457/557 Functional Programming

Lecture 9
More on Higher-order Functions
Currying ("eta reduction")

Recall (from Ch. 1) the function: \[ \text{simple } n \ a \ b = n * (a+b) \]

Note that:
\[ \text{simple } n \ a \ b \]
is really
\[ (((\text{simple } n) \ a) \ b) \]
in fully parenthesized notation

\[ \text{simple} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \]
\[ \text{simple } n :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \]
\[ ((\text{simple } n) \ a :: \text{Float} \rightarrow \text{Float} \]
\[ (((\text{simple } n) \ a) \ b :: \text{Float} \]

Therefore:
\[ \text{multSumByFive } a \ b = \text{simple } 5 \ a \ b \] is the same as
\[ \text{multSumByFive} = \text{simple } 5 \]
Use of Eta Reduction

\[\text{listSum}, \text{listProd} :: [\text{Integer}] -> \text{Integer}\]

\[\text{listSum } xs = \text{foldr } (+) 0 xs\]
\[\text{listProd } xs = \text{foldr } (*) 1 xs\]

\[\text{listSum} = \text{foldr } (+) 0\]
\[\text{listProd} = \text{foldr } (*) 1\]

\[\text{and, or} :: [\text{Bool}] -> \text{Bool}\]

\[\text{and } xs = \text{foldr } (\&\&) \text{ True } xs\]
\[\text{or } xs = \text{foldr } (||) \text{ False } xs\]

\[\text{and} = \text{foldr } (\&\&) \text{ True}\]
\[\text{or} = \text{foldr } (||) \text{ False}\]
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:

\[
\text{reverse } xs = \text{foldl } \text{revOp } [] \text{ xs}
\]

where \( \text{revOp acc x = x : acc} \)

In the prelude we have: \( \text{flip f x y = f y x} \). Thus:

\[
\text{revOp acc x = flip (:) acc x}
\]

or even better:

\[
\text{revOp = flip (:
}\]

And thus:

\[
\text{reverse } xs = \text{foldl (flip (:) ) [] xs}
\]

or even better:

\[
\text{reverse = foldl (flip (:) ) []}
\]
Anonymous Functions

• So far, all of our functions have been defined using an equation, such as the function \texttt{succ} defined by:
  \[
  \texttt{succ} \; x = x + 1
  \]

• This raises the question: Is it possible to define a value that behaves just like \texttt{succ}, but has no name? Much in the same way that \texttt{3.14159} is a value that behaves like \texttt{pi}?

• The answer is yes, and it is written \(\lambda x \rightarrow x + 1\). Indeed, we could rewrite the previous definition of \texttt{succ} as:
  \[
  \texttt{succ} = \lambda x \rightarrow x + 1.
  \]

• The backslash (\(\lambda\)) 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:

\( (+5) = \lambda x \rightarrow x + 5 \)
\( (4-) = \lambda y \rightarrow 4 - y \)

So in fact \( \text{succ } x = x + 1 \) 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 \( \lambda x \rightarrow (x+1)/2 \) as a section.

• You can also pattern match using an anonymous function, as in \( \lambda (x:xs) \rightarrow x \), 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 (.):

\[(f \cdot g) \, x = f \, (g \, x)\]

• So \(f \cdot g\) is the same as \(\lambda x \rightarrow f \, (g \, x)\).

• Function composition can be used to simplify previous definitions:

\[
\text{totalSquareArea sides}
\]
\[
= \text{sumList} \, (\text{map squareArea sides})
\]
\[
= (\text{sumList} \, . \, \text{map squareArea}) \, \text{sides}
\]

Combining this with eta reduction yields:

\[
\text{totalSquareArea} = \text{sumList} \, . \, \text{map squareArea}
\]