CSc 520

Principles of Programming Languages

17: Haskell — Composing Functions

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
We want to discover frequently occurring patterns of computation. These patterns are then made into (often higher-order) functions which can be specialized and combined. \texttt{map \ f \ L} and \texttt{filter \ f \ L} can be specialized and combined:

\begin{verbatim}
double :: [Int] -> [Int]
double xs = map ((*) 2) xs

positive :: [Int] -> [Int]
positive xs = filter ((<) 0) xs

doublePos xs = map ((*) 2) (filter ((<) 0) xs)
\end{verbatim}

\texttt{doublePos [2,3,0,-1,5]}

\texttt{[4, 6, 10]}
Functional composition is a kind of “glue” that is used to “stick” simple functions together to make more powerful ones.

In mathematics the ring symbol (\( \circ \)) is used to compose functions:

\[ (f \circ g)(x) = f(g(x)) \]

In Haskell we use the dot ("." ) symbol:

```haskell
define operator infixr 9 .
( . ) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g)(x) = f(g(x))
```
Composing Functions.

\[
(\cdot) :: (b\to c) \to (a\to b) \to (a\to c)
\]

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

"\cdot" takes two functions \(f\) and \(g\) as arguments, and returns a new function \(h\) as result.

- \(g\) is a function of type \(a\to b\).
- \(f\) is a function of type \(b\to c\).
- \(h\) is a function of type \(a\to c\).
- \((f \cdot g)(x)\) is the same as \(z=g(x)\) followed by \(f(z)\).
Composing Functions...

We use functional composition to write functions more concisely. These definitions are equivalent:

\[
\text{doit } x = f_1 \ (f_2 \ (f_3 \ (f_4 \ x)))
\]
\[
\text{doit } x = (f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4) \ x
\]

\[
\text{doit} = f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4
\]

The last form of \text{doit} is preferred. \text{doit}’s arguments are implicit; it has the same parameters as the composition.

\text{doit} can be used in higher-order functions (the second form is preferred):

\[
? \ \text{map} \ (\text{doit}) \ xs
\]
\[
? \ \text{map} \ (f_1 \ . \ f_2 \ . \ f_3 \ . \ f_4) \ xs
\]
Example: Splitting Lines

Assume that we have a function \texttt{fill} that splits a string into filled lines:

\begin{verbatim}
fill :: string \rightarrow [string]
fill s = splitLines (splitWords s)
\end{verbatim}

\texttt{fill} first splits the string into words (using \texttt{splitWords}) and then into lines:

\begin{verbatim}
splitWords :: string \rightarrow [word]
splitLines :: [word] \rightarrow [line]
\end{verbatim}

We can rewrite \texttt{fill} using function composition:

\begin{verbatim}
fill = splitLines \ . \ splitWords
\end{verbatim}
1. "." is right associative. I.e.
   \[ f \cdot g \cdot h \cdot i \cdot j = f \cdot (g \cdot (h \cdot (i \cdot j))) \]
2. "." has higher precedence (binding power) than any other operator, except function application:
   \[ 5 + f \cdot g \ 6 = 5 + (f \cdot (g \ 6)) \]
3. "." is associative:
   \[ f \cdot (g \cdot h) = (f \cdot g) \cdot h \]
4. "id" is "."’s identity element, i.e 
   \[ \text{id} \cdot f = f = f \]
   \[ \text{id} :: a -> a \]
   \[ \text{id} \ x = x \]
The **count** Function

Define a function `count` which counts the number of lists of length `n` in a list `L`:

\[
\text{count 2 } [[1],[],[2,3],[4,5],[]] \Rightarrow 2
\]

**Using recursion:**

\[
\text{count :: Int -> [[a]] -> Int}
\]

\[
\text{count _ [] = 0}
\]

\[
\text{count n (x:xs)}
\]

\[
\begin{align*}
\text{length x == n} & \quad = 1 + \text{count n xs} \\
\text{otherwise} & \quad = \text{count n xs}
\end{align*}
\]

**Using functional composition:**

\[
\text{count’ n = length \ . \ filter (==n) \ . \ map length}
\]
The `count` Function...

\[
\text{count'} \ n \ = \ \text{length} . \ \text{filter (==n)} . \ \text{map length}
\]

- **What does \text{count'} do?**

\[
[[1], [], [2,3], [4,5], []] \\
\downarrow \text{map length} \\
[1,0,2,2,0] \\
\downarrow \text{filter (==2)} \\
[2,2] \\
\downarrow \text{length} \\
2
\]

- **Note that**

\[
\text{count'} \ n \ \text{xs} = \ \text{length (filter (==n) (map length \text{xs}))}
\]
The \textit{init} \& \textit{last} Functions

- \textit{last} returns the last element of a list.
- \textit{init} returns everything but the last element of a list.

\textbf{Definitions:}

\begin{align*}
\text{last} &= \text{head} \ . \ \text{reverse} \\
\text{init} &= \text{reverse} \ . \ \text{tail} \ . \ \text{reverse}
\end{align*}

\textbf{Simulations:}

\begin{align*}
[1,2,3] \overset{\text{reverse}}{\rightarrow} [3,2,1] \overset{\text{head}}{\rightarrow} 3 \\
[1,2,3] \overset{\text{reverse}}{\rightarrow} [3,2,1] \overset{\text{tail}}{\rightarrow} [2,1] \overset{\text{reverse}}{\rightarrow} [1,2]
\end{align*}
The \texttt{any} Function

\textbullet \quad \texttt{any \ p \ xs} \ \textbf{returns} \ True \ \textbf{if} \ p \ x = \ True \ \textbf{for some} \ x \ \textbf{in} \ xs:

\texttt{any \ ((==)0) \ [1,2,3,0,5]} \Rightarrow \textbf{True}
\texttt{any \ ((==)0) \ [1,2,3,4]} \Rightarrow \textbf{False}

\underline{Using recursion:}
\texttt{any :: (a \to \textbf{Bool}) \to [a] \to \textbf{Bool}}
\texttt{any _ [] = False}
\texttt{any p (x:xs) = | \ p x = True}
\texttt{| \ otherwise = any p xs}

\underline{Using composition:}
\texttt{any p = or . map p}
\texttt{[1,0,3]^\text{map \ ((==)0)} [False,True,False]^\text{or}True}
commaint Revisited...

Let’s have another look at one simple (!) function, commaint.

commaint works on strings, which are simply lists of characters.

You are not now supposed to understand this!

From the commaint documentation:

[commaint] takes a single string argument containing a sequence of digits, and outputs the same sequence with commas inserted after every group of three digits, …
commaint Revisited...

Sample interaction:

? commaint "1234567"
1,234,567

commaint in Haskell:

commaint = reverse . foldr1 (\x y->x++","++y) .
group 3 . reverse
  where group n = takeWhile (not.null) .
    map (take n).iterate (drop n)
commaint Revisited...

```
"1234567"
\downarrow reverse

"7654321"
\downarrow iterate (drop 3)

["7654321","4321","1","","",...
\downarrow map (take 3)

["765","432","1","","",...
\downarrow takeWhile (not.null)

["765","432","1"]
\downarrow foldr1 (\(x\ y\rightarrow x++","++y)

"765,432,1"
\downarrow reverse

"1,234,567"
```
commaint Revisited...

commaint = reverse . foldr1 (\x y->x++","++y) .
group 3 . reverse
where group n = takeWhile (not.null) .
map (take n).iterate (drop n)

iterate (drop 3) s returns the infinite list of strings

[s, drop 3 s, drop 3 (drop 3 s),
 drop 3 (drop 3 (drop 3 s)), ...]

map (take n) xss shortens the lists in xss to n elements.
commaint Revisited...

commaint = reverse . foldr1 (\x y->x++","++y) .
group 3 . reverse
where group n = takeWhile (not.null) . map (take n).iterate (drop n)

- **takeWhile (not.null)** removes all empty strings from a list of strings.
- **foldr1 (\x y->x++","++y)** s takes a list of strings s as input. It appends the strings together, inserting a comma in between each pair of strings.
Lambda Expressions

- $(\lambda x\ y \to x++",\"++y)$ is called a lambda expression.
- Lambda expressions are simply a way of writing (short) functions inline. Syntax:

\[ \text{\textbackslash arguments} \to \text{expression} \]

Thus, `comment` could just as well have been written as

```haskell
comment = \n \cdot foldr1 (\n \cdot insert

where group n = \n
  insert x y = x++","++y
```

Examples:

```haskell
squareAll xs = map (\x \to x * x) xs
length = foldl' (\n \to n+1) 0
```
The built-in operator " . " (pronounced “compose”) takes two functions \( f \) and \( g \) as argument, and returns a new function \( h \) as result.

The new function \( h = f . g \) combines the behavior of \( f \) and \( g \): applying \( h \) to an argument \( a \) is the same as first applying \( g \) to \( a \), and then applying \( f \) to this result.

Operators can, of course, also be composed: \( ((+2) . (*3)) 3 \) will return \( 2 + (3 * 3) = 11 \).
Write a function `mid xs` which returns the list `xs` without its first and last element.

1. use recursion
2. use `init`, `tail`, and functional composition.
3. use `reverse`, `tail`, and functional composition.

? mid [1,2,3,4,5] ⇒ [2,3,4]
? mid [] ⇒ ERROR
? mid [1] ⇒ ERROR
? mid [1,3] ⇒ []