CSc 372 — Comparative Programming Languages

12 : Haskell — Composing Functions

Christian Collberg

Department of Computer Science
University of Arizona
collberg+372@gmail.com

September 16, 2005

1 Composing Functions

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. `map f L` and `filter f L` can be specialized and combined:

```
double ::  [Int] -> [Int]
double xs = map ((*) 2) xs

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

doublePos xs = map ((*) 2) (filter ((<) 0) xs)
?  doublePos [2,3,0,-1,5]
[4, 6, 10]
```

2 Composing Functions. . .

- Functional composition is a kind of "glue" that is used to "stick" simple functions together to make more powerful ones.

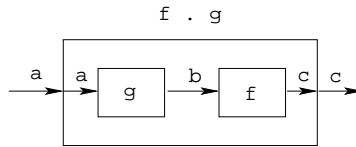- In mathematics the ring symbol (∘) is used to compose functions:

$$(f \circ g)(x) = f(g(x))$$

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

```
infixr 9 .
(.)  ::  (b->c) -> (a->b) -> (a->c)
(f .  g)(x) = f(g(x))
```

3 Composing Functions...

```
(.)   ::   (b->c) -> (a->b) -> (a->c)
(f .  g)(x) = f(g(x))
```


f . g

- "." takes two functions f and g as arguments, and returns a new function h as result.

- g is a function of type a->b.

- f is a function of type b->c.

- h is a function of type a->c.

- (f.g)(x) is the same as z=g(x) followed by f(z).

4 Composing Functions...

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

```
doit x = f1 (f2 (f3 (f4 x)))
doit x = (f1 .  f2 .  f3 .  f4) x
doit = f1 .  f2 .  f3 .  f4
```

- The last form of doit is preferred. doit's arguments are implicit; it has the same parameters as the composition.

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

```
?  map (doit) xs
?  map (f1 .  f2 .  f3 .  f4) xs
```

5 Example: Splitting Lines

- Assume that we have a function fill that splits a string into filled lines:

```
fill ::  string -> [string]
fill s = splitLines (splitWords s)
```

- fill first splits the string into words (using splitWords) and then into lines:

```
splitWords ::  string -> [word]
splitLines ::  [word] -> [line]
```

- We can rewrite fill using function composition:

```
fill = splitLines .  splitWords
```

2

6 Precedence & Associativity

1. "." is right associative. I.e.

   ```
   f.g.h.i.j = f.(g.(h.(i.j)))
   ```

2. "." has higher precedence (binding power) than any other operator, except function application:

   ```
   5 + f.g 6 = 5 + (f.  (g 6))
   ```

3. "." is associative:

   ```
   f .  (g .  h) = (f .  g) .  h
   ```

4. "id" is "."'s identity element, i.e `id .  f = f = f .  id`:

   ```
   id ::  a -> a
   id x = x
   ```

7 The count Function

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

   ```
   count 2 [[1],[],[2,3],[4,5],[]] ⇒ 2
   ```

<u>Using recursion:</u>

```
count ::  Int -> [[a]] -> Int
count _ [] = 0
count n (x:xs)
   | length x == n   = 1 + count n xs
   | otherwise       = count n xs
```

<u>Using functional composition:</u>

```
count' n = length .  filter (==n) .  map length
```

8 The count Function...

```
count' n = length .  filter (==n) .  map length
```

- What does `count'` do?

```
            [[1],[],[2,3],[4,5],[]]
                     |
                     ↓         map length
               [1,0,2,2,0]
                     |
                     ↓           filter (==2)
                  [2,2]
                     |
                     ↓           length
                     2
```

- Note that

   ```
   count' n xs = length (filter (==n) (map length xs))
   ```

3

9 The `init` & `last` Functions

- `last` returns the last element of a list.

- `init` returns everything but the last element of a list.

```
last = head .  reverse

init = reverse .  tail .  reverse
```

$[1,2,3] \overset{\mathtt{reverse}}{\Longrightarrow} [3,2,1] \overset{\mathtt{head}}{\Longrightarrow} 3$

$[1,2,3] \overset{\mathtt{reverse}}{\Longrightarrow} [3,2,1] \overset{\mathtt{tail}}{\Longrightarrow} [2,1] \overset{\mathtt{reverse}}{\Longrightarrow} [1,2]$

10 The `any` Function

- `any p xs` returns `True` if `p x == True` for some `x` in `xs`:

```
any ((==)0) [1,2,3,0,5] ⇒ True
any ((==)0) [1,2,3,4] ⇒ False
```

```
any ::  (a -> Bool) -> [a] -> Bool
any _ [] = False
any p (x:xs) = | p x = True
               | otherwise = any p xs
```

```
any p = or .  map p
```
$[1,0,3] \overset{\mathtt{map\ ((==)0)}}{\Longrightarrow} [\mathtt{False,True,False}] \overset{\mathtt{or}}{\Longrightarrow} \mathtt{True}$

11 `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!

[`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, · · ·
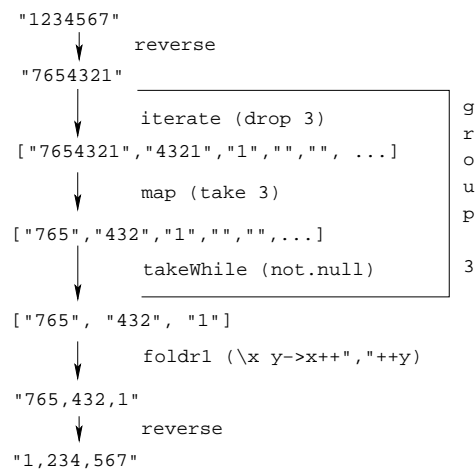
12   `commaint` Revisited. . .

```
?  commaint "1234567"
   1,234,567
```

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

13   `commaint` Revisited. . .

```
"1234567"
   │         reverse
   ↓
"7654321"
                 ┌──────────────────────────┐
   │               iterate (drop 3)         │ g
   ↓                                        │ r
["7654321","4321","1","","", ...]           │ o
                                            │ u
   │               map (take 3)             │ p
   ↓                                        │
["765","432","1","","",...]                 │
                                            │ 3
   │               takeWhile (not.null)     │
   ↓                                        │
                 └──────────────────────────┘
["765", "432", "1"]
   │               foldr1 (\x y->x++",".++y)
   ↓
"765,432,1"
   │         reverse
   ↓
"1,234,567"
```

14   `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.

15   `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.

16 Lambda Expressions

- `(\x y->x++","++y)` is called a lambda expression.

- Lambda expressions are simply a way of writing (short) functions inline. Syntax:

      \ arguments -> expression

- Thus, `commaint` could just as well have been written as

```
commaint = ··· .  foldr1 insert .  ···
   where group n = ···
         insert x y = x++","++y
```

<div align="center">Examples:</div>

```
squareAll xs = map (\ x -> x * x) xs
length = foldl' (\n _ -> n+1) 0
```

17 Summary

- 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`.

18 Homework

- 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]  ⇒  []
```