CSc 372

Comparative Programming Languages

12 : Haskell — Composing Functions

Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

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]
```

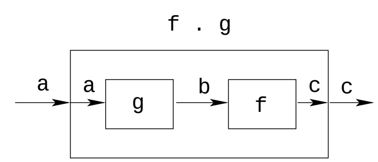
Composing Functions. . .

- 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 ($\circ$) 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))
```

Composing Functions. . .

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



- "." 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)`.

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
```

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
```

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
   ```

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`

Using recursion:

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

Using functional composition:

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

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))`

The init & last Functions

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

Definitions:

```
last = head . reverse

init = reverse . tail . reverse
```

Simulations:

$\texttt{[1,2,3]} \stackrel{\texttt{reverse}}{\Longrightarrow} \texttt{[3,2,1]} \stackrel{\texttt{head}}{\Longrightarrow} \texttt{3}$

$\texttt{[1,2,3]} \stackrel{\texttt{reverse}}{\Longrightarrow} \texttt{[3,2,1]} \stackrel{\texttt{tail}}{\Longrightarrow} \texttt{[2,1]} \stackrel{\texttt{reverse}}{\Longrightarrow} \texttt{[1,2]}$

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
```

Using recursion:

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

Using composition:

```
any p = or . map p
```

$\texttt{[1,0,3]} \stackrel{\texttt{map ((==)0)}}{\Longrightarrow} \texttt{[False,True,False]} \stackrel{\texttt{or}}{\Longrightarrow} \texttt{True}$

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. . .

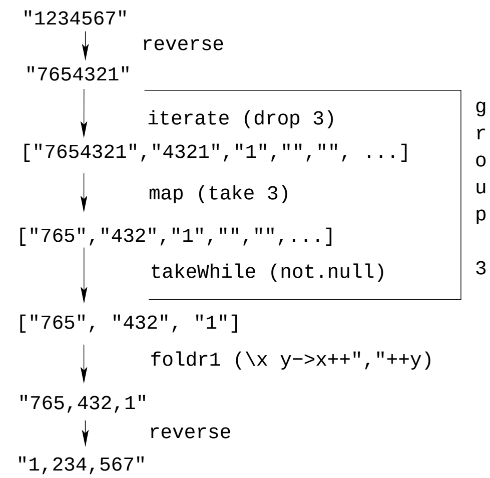
Sample interaction:

```
?  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. . .



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

- `(\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
```

Examples:

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

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`.

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] ⇒ []
```