University of Auckland

415.360

Functional Programming

Christian Collberg
March 22, 1996

Functional Composition

Composing Functions I

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` are examples of such functions. They can be specialized:

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

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

and they can be combined:

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

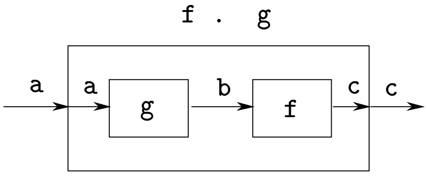

Composing Functions II

- 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:

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

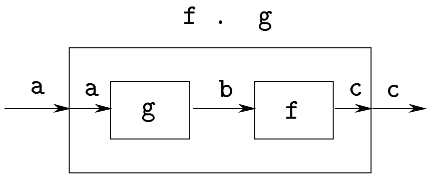
- In Gofer we use the dot (".") symbol:

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



Composing Functions III

```
(.)  ::  (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 first computing `z=g(x)` and then `f(z)`.

Composing Functions IV

- We use functional composition to write functions more concisely:

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

is the same as:

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

which is the same as:

```
doit = f1 .  f2 .  f3 .  f4
```

- The last form of doit is the preferred one. Note that the arguments are **implicit**. `doit` has the same parameters as the composition.
- We can now use `doit` in higher-order functions (the second form is often preferred):

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

Slide 9–4

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
```

Slide 9–5

Precedence & Associativity

Note that

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**:

   `id .  f = f = f .  id`

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

Slide 9–6

The `count` Function I

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

Examples:

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
```

Slide 9–7

The `count` Function II

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

- What does `count'` do?

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

- Note that

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

Slide 9–8

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:

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

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

Slide 9–9

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
```

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

Slide 9–10

`commaint` Revisited I

- 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, . . .

Slide 9–11

`commaint` Revisited II

Sample interaction:

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

`commaint` simulation:

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

Slide 9–12

`commaint` Revisited III

```
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.

Slide 9–13

`commaint` Revisited IV

```
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.

Slide 9–14

Lambda Expressions

- `(\x y->x++","++y)` is called a **lambda expression**.
- Lambda expressions are simply a way of writing (short) functions inline.
- Thus, `commaint` could just as well have been written as

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

Lambda expression syntax:

```
\ arguments -> expression
```

Examples:

```
squareAll xs = map (\ x -> x * x) xs

length = foldl' (\n _ -> n+1) 0
```

Slide 9–15

Summary

- B&W reading list:
  1. Function composition: 38
- 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`.

Slide 9–16

Homework I

- 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.

Examples:

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

Slide 9–17