Funções como valores

Márcio Lopes Cornélio

Centro de Informática - UFPE

A função de composição

Definição

```
(f . g) x = f (g x)
```

A função de composição

```
(.) :: (u -> v) -> (t -> u) -> 1(t -> v)

(.) f g x = f (g x)
==
f g = \x-> f (g x)

:type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Composição de funções

Estruturar programas compondo funções

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

splitWords :: String -> [Word]
splitLines :: [Word] -> [Line]

fill = splitLines . splitWords
```

Funções como valores e resultados

```
twice :: (t -> t) -> (t -> t)
twice f = f . f
```

```
(twice succ) 12
= (succ . succ) 12
= succ (succ 12)
= 14
```

Funções como valores e resultados

```
iter :: Int -> (t -> t) -> (t -> t)
iter 0 f = id
iter n f = (iter (n-1) f).f

(iter 10 double) 3
```

Expressões que definem funções

```
addNum :: Int -> (Int -> Int)
addNum n = h
  where
  h m = n + m
```

Notação lambda

```
\m -> 3+m
addNum n = (\m -> n+m)
```

Composição de função

```
\x y -> g (f x) (f y)

comp2 :: (t -> u) -> (u -> u -> v) ->  (t -> t -> v)
-- qual o tipo da funcao abaixo?
comp2 f g = (\x y -> g (f x) (f y))
```

Exercício

Dada uma função f do tipo `t -> u -> v`, defina uma expressão da forma `(\... -> ...)` para uma função do tipo `u -> t -> v` que se comporta como f mas recebe seus argumentos na ordem inversa

Aplicações parciais

```haskell
multiply :: Int -> Int -> Int
multiply a b = a*b

doubleList :: [Int] -> [Int]
doubleList = map (multiply 2)

(multiply 2) :: Int -> Int
map (multiply 2) :: [Int] -> [Int]
```

Aplicações parciais

```
whiteSpace = "␣"

elem :: Char -> [Char] -> Bool
elem ch whiteSpace

\ch -> elem ch whiteSpace

filter (\ch -> not(elem ch whitespace))
```

Qual o tipo da funçãoo acima?

Associatividade

```
f a b = (f a) b

f a b ≠ f (a b)

t -> u -> v = t -> (u -> v)

t -> u -> v ≠ (t -> u) -> v

g :: (Int -> Int) -> Int
g h = h 0 + h 1
```

A função de composição revisitada

```
(.) :: (u -> v) -> (t -> u) -> (t -> v)

(.) f g x = f (g x)
==
(.) f g = \x->f (g x)


:type (.)
(.)::(b->c)->(a->b)->a->c
```

Revisitando iter

```
iter 10 double 3 ==
iter 10 ((*) 2) 3 ==
iter 10 (2 *) 3 ==
iter 10 (* 2) 3 ?

iter 10 (/ 2) 2000 ==
iter 10 ((/) 2) 2000 ?
```

Quantos argumentos uma função tem?

```haskell
multiply :: Int -> Int -> Int
multiply :: Int -> (Int -> Int)

multiply 4
(multiply 4) 5
```

Mais aplicações parciais

```
dropSpace = dropWhile (member whitespace)
dropWord = dropWhile (not.(member whitespace))
getWord = takeWhile (not . member whitespace)

member ::[t]->t->Bool
member st x = elem x st
```

Mais exmplos

```
(+2)
(>2)
(3:)
(++ "\n")
filter (>0).map (+1)
double = map (*2)
dificil = map.filter
maisdificil = map.foldr
maisainda = foldr.map
```

Bibliografia

Slides elaborados a partir de originais por André Santos e Fernando Castor

[1] Simon Thompson.
Haskell: the craft of functional programming.
Addison-Wesley, terceira edition, Julho 2011.