How to compose point-free functions

utashih

Warm-up: eta-reduction

- The following two functions are equivalent under eta-conversion:

```
\x -> f x
```

and

```
f
```

Warm-up: eta-reduction

- Haskell's *partially applying* functionality enables us to remove the last (rightmost) parameter of *curried* functions

```
sum xs = foldr (+) 0 xs
```

By applying eta-reduction

```
sum = foldr (+) 0
```

Point-free style

- also called *tacit programming*
- *(from Wikipedia)* Function definitions do not identify the arguments (points) on which they operate; they merely compose other functions
- considered unnecessarily obscure
- …and we’re going into this obscurity!

Examples from H-99

```haskell
last = foldr1 (const id)
reverse = foldl (flip (:)) []

elementAt :: [a] -> Int -> a
elementAt = (last .) . flip take

isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome = (==) <*> reverse
```

Composition of functions

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

- will be used ubiquitously

Example from Codewars

- WeIrD StRiNg CaSe

```
toWeirdCase :: String -> String
toWeirdCase =
    unwords . map (zipWith ($) weird) . words
    where weird = cycle [toUpper, toLower]
```

Digression: functor

- types that can be mapped over (with restrictions)

```
class Functor f where
    fmap :: (a -> b) ~> f a -> f b
```

Hom functor

- For each type r, the (covariant) hom functor $Hom(r, -)$ maps
  - each type a to type $r \to a$
  - each function of type $a \to b$ to a function of type $(r \ \to \ a) \to (r \ \to \ b)$

```
fmap :: (a -> b) ~> (r -> a) -> (r -> b)
```

Hom functor

- a more familiar name would be the Reader functor

```
instance Functor ((->) r) where
    fmap = (.)
```

Example: concatMap

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap = (concat .) . map
```

- Or equivalently

```
concatMap f xs = concat (map f xs)
```

Example: concatMap

- To validate the equivalence

```
concatMap f xs = concat (map f xs)
concatMap f xs = concat $ map f xs
concatMap f xs = concat . map f $ xs
concatMap f    = concat . map f
concatMap f    = (.) concat (map f)
concatMap f    = (.) concat $ map f
concatMap f    = (.) concat . map $ f
concatMap      = (.) concat . map
concatMap      = (concat .) . map
```

Example: concatMap

- The key idea is to find the rightmost parameter by changing (nested) function applications into single composed function applied to single argument

```
concatMap f xs = concat (map f xs)
concatMap f xs = concat $ map f xs
concatMap f xs = concat . map f $ xs
concatMap f    = concat . map f
```

Example: concatMap

- Note that it is incorrect to take the second step like

```
concatMap f xs = concat $ map f xs
concatMap f xs = concat . map $ f xs
```

- The latter formula corresponds to

```
concat (map (f xs))
```

- Function application associates to the left while `($)` associates to the right; thus the outermost application is `(map f) xs` instead of `map (f xs)`

Example: elementAt

```
elementAt :: [a] -> Int -> a
elementAt = (last .) . flip take

(last .) :: (r -> [a]) -> (r -> a)
flip take :: [a] -> (Int -> [a])
```

…so we have more dots!

- Composing function with 2 arguments

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

- Thus we have

```
concatMap = concat .: map
```

- This operator is defined in Data.Composition in the *composition* package

Back to the hom functor

- Of course, it is an applicative functor/monad as well
- Some more definitions:

```
instance Applicative ((->) r) where
    pure = const
    f <*> g = \r -> f r (g r)

instance Monad ((->) r) where
    f >>= g = \r -> g (f r) r
```

Example from H-99

```
isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome = (==) <*> reverse

isPalindrome = \x -> x == reverse x
             = \x -> (==) x (reverse x)
             = \x -> (==) <*> reverse $ x
             = (==) <*> reverse
```

Example from Codewars

- Equal Sides of An Array

```
findEvenIndex :: [Int] -> Int
findEvenIndex = fromMaybe (-1) . elmIndex True .
    (zipWith (==) <$> scanl1 (+) <*> scanr1 (+))
```

- Note that `fmap`, `(<$>)` and `(.)` serve the same purpose

Example from Codewars

- Sort the Odd

```
sortArray :: [Int] -> [Int]
sortArray = replaceOdd <$> id <*> sort .
    filter odd

replaceOdd :: [Int] -> [Int] -> [Int]
```

Example from Codewars

- Are they the “same”

```
comp :: [Integer] -> [Integer] -> Bool
comp = (. sort) . (==) . sort . map (^2)

comp xs ys = sort (map (^2) xs) == sort ys
```

Example: map filter

```
mp :: (b -> Bool) -> (a -> b) -> [a] -> [b]
mp p f xs = filter p (map f xs)
mp p f    = filter p . map f
mp        = (. map) . (.) . filter
```

Partial applying to `(.)`

- Putting it off

```
f :: (a -> b)
(f .) :: (r -> a) -> (r -> b)
```

- Bringing it forward

```
g :: (a -> b)
(. g) :: (b -> s) -> (a -> s)
```

In all: understanding `(.)`

- Composition of functions
- Functor map of hom functor
- Continuation-like

“Boilerplate” pattern

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

func x = f (g x) (h x)
func   = f <$> g <*> h
       = f . g <*> h

func x y = f (g x) (h y)
func     = (. h) . f . g
```

Try it out

```
agreeLen :: (Eq a) => [a] -> [a] -> Int
agreeLen x y = length $
    takeWhile (\(a, b) -> a == b) (zip x y)
```

- Check your answer at http://pointfree.io/

Typical misuse

```
mapWithIndex :: (Int -> a -> b) -> [a] -> [b]
mapWithIndex =
    (snd .) . (`mapAccumL` 0) . ((.) . (,) . (+ 1) <*>)
```

- This should be

```
mapWithIndex f xs =
    snd $ mapAccumL (\i x -> (i+1, f i x)) 0 xs
```

```
mapAccumL :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
```

Q&A