

GTD in Haskell

Composition

Susan Potter

2019-10-27

Composition

What is composition?

- compose functions

```
compose :: (b -> c) -- g
        -> (a -> b) -- f
        -> (a -> c)
compose g f = \a -> g (f a)
```

- compose independent effects
- compose dependent effects
- ... much more
- compose multiple of the above

Intuition of composition & examples

Break down a problem into constituent parts.

Examples:

- pipe functions together (e.g., `.`, `&`) to build a new function
- pipe data with simple effects together (e.g., `<$>`, `<&>`, `<*>`)
- pipe programs together (e.g., `>>=`, `=<<`, `>>`)
- **traverse** structures applying a given action just once
- fold F-algebra over Functor (e.g., **cata**)
- unfold codata from a coalgebra (e.g., **ana**)
- refold given a coalgebra and F-algebra (e.g., **hylo**)

Why composition?

- Decompose problem into simpler parts
- Implement each part in a pure testable way
- Combine ("glue") parts using composition *operators*

Goal: No magic, everything just works (and tested)~!

~ according to some specification of "works".

Function composition (after aka (.))

```
after :: (b -> c) -> (a -> b) -> (a -> c)
after g f = \a -> g (f a)
(.) = after -- alias
```

```
double x      = x * 2
increment x    = x + 1
```

```
-- >>> foo 3      => 7
foo = increment `after` double
-- foo = increment . double
```

```
-- >>> bar 3      => 8
bar = double `after` increment
-- bar = double . increment
```

Reverse compose (before aka (&))

```
before :: (a -> b) -> (b -> c) -> (a -> c)
before f g = a -> g (f a)
(&) = before -- alias
```

```
-- >>> foo' 3      => 7
foo' = double & increment
-- foo' = double `before` increment
```

```
-- >>> bar' 3      => 8
bar' = increment & double
-- bar' = increment `before` double
```


Compose independent effects (ap aka <*>)

```
(<*>) :: Applicative f
      -- defined for each Applicative instance
      => f (a -> b) -> f a -> f b
```

```
data User = MkUser String Date (Maybe String)
getNick   :: WebForm -> Maybe String
getDOB    :: WebForm -> Maybe Date
getBio    :: WebForm -> Maybe (Maybe String)
```

```
-- >>> getUser formWithNoDOB => Nothing
-- >>> getUser validFormData => Just (...)
getUser f      -- :: WebForm -> Maybe User
  = pure MkUser <*> getNick f
                  <*> getDOB f
                  <*> getBio f
```

Compose dependent actions (>>=, >>)

```
readFile      :: Path -> IO String
getArgs       :: IO [String]
parseArgs     :: [String] -> Path
readPaths     :: Path -> IO [Path]
getFile       :: Path -> IO (Path, String)
printFile     :: (Path, String) -> IO ()

main
  = getArgs
    >>= pure . parseArgs
    >>= readPaths
    >>= \ps -> sequence (getFile <$> ps)
    >>= \fs -> sequence (printFile <$> fs)
    >> pure ()
```

Composing Arrows ((&&&), (***))

```
type Count = Const (Sum Int)
count = Const <<< Sum <<< length

-- decomposition into small parts
countL = count <<< lines
countW = count <<< words
countC = count <<< id

-- avoid traversing twice by composing
-- the functions packaged up as Arrows
-- into one traversal
countAll
    = foldMap (countL &&& countW &&& countC)
```

Comparison to wc C (tiny snippet)

```
311 else if (!count_chars && !count_complicated)
312 {
313     /* Use a separate loop when counting only
314        ↪ lines or lines and bytes --
315        but not chars or words. */
316     bool long_lines = false;
317     while ((bytes_read = safe_read (fd, buf,
318        ↪ BUFFER_SIZE)) > 0)
319     {
320         if (bytes_read == SAFE_READ_ERROR)
321         {
322             error (0, errno, "%s", quotef
323                 ↪ (file));
324             ok = false;
325             break;
326         }
327     }
328 }
```

Comparison to wc metrics

Concern	LOC
All	895
arg parsing	37
compute width	26
get file status	20
count words	429

<https://github.com/coreutils/coreutils/blob/master/src/wc.c>