GTD in Haskell

Transforms, filters, folds (simple data)

Susan Potter

2019-11-11

+STARTUP: beamer

Review

Kinds of data

- Simple: no nested or recursive structures, e.g. lists, maps.
- Nested: data structures with nested structures to traverse, e.g. maps of lists, list of lists.
- Recursive: data structures with recursive structure, e.g. abstract syntax trees.

Simple data structures (non-nested, non-recursive)

Useful types of operations include ability to:

- transform each element
- filter elements
- fold/reduce a structure down to one value

Non-nested, non-recursive data structures are the focus of this material.

Transforms

Patterns of Transforms

- Scaling or converting
  Typically `a -> b` (including `a -> a` aka ”scaling”)
- Structuring
  Typically `a -> F b`. See `Kleisli` arrows.
- Flattening
  Typically `F a -> b`. See `CoKleisli` arrows.
- Composition of above :)

Thinking in terms of type patterns gives us more structured thought with the goal of building more composable software, which IME allows us to test and reason about our software easily leading to codebases you don’t dread maintaining.

Extensions and imports

With no implicit prelude:

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE InstanceSigs      #-}

import Data.Functor  (fmap, void, (<$>),
 ↪  (<$), (<&>))
import Data.Function ((&), ($))
```

`Functor`'s primary interface

Review:

```
fmap  :: Functor f => (a -> b) -> f a -> f b
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<&>) :: Functor f => f a -> (a -> b) -> f b
```

Operator	Usage	Example	Result
`fmap`	prefix	`fmap incr [1..5]`	`[2,3,4,5,6]`
`(<$>)`	infix	`incr <$> [1..5]`	`[2,3,4,5,6]`
`(<&>)`	infix	`[1..5] <&> incr`	`[2,3,4,5,6]`

`Functor` operators: ”applies” and fmap

Remember:

```
($)   ::                  (a -> b) -> a   -> b
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Let’s say:

```
incr n = n + 1
```

Then `incr` *applies to result of* `(1+5)` like so:

```
>>> incr $ 1 + 5 -- incr (1+5) --> ((1+5)+1)
7
```

Or `incr` *applies to each element in* `1 thru 5`:

```
>>> incr <$> [1..5]
[2,3,4,5,6]
```

`Functor`: *"passed to"* aka flipped fmap `(<&>)`

Remember that:

```
(&)   ::                  a   -> (a -> b) -> b
(<&>) :: Functor f => f a -> (a -> b) -> f b
```

Then `(1+5)` *is passed to* `incr` like so:

```
>>> 1 + 5 & incr
7
```

Or *1 thru 5 is passed to* `incr` (each):

```
>>> [1..5] <&> incr
[2,3,4,5,6]
```

`Functor`: *"inject into"* (`(<$)`)

Given:

```
>>> :t (<$)
(<$) :: Functor f => a -> f b -> f a
```

We can *inject* `42` *into* `Just "Meaning of life"` like so:

```
>>> 42 <$ Just "Meaning of life"
Just 42
```

Useful when data construtors of context type not exported.

`Functor`: voiding

Voiding the wrapped value:

```
>>> :t void
void :: Functor f => f a -> f ()

>>> void (Just 42)
Just ()
```

- Useful at end of effectful pipeline to avoid leaking data to caller or to match types.
- Typically used as prefix function, e.g., `void $ uppercaseAndSave <*> openFile`

Story data type

If we are building a multi-tenant story publishing platform we might start with this data type definition:

```haskell
data Story
  = MkStory { storyId       :: Int
            , storyAuthor   :: Author
            , storyTitle    :: Text
            , storySubtitle :: Maybe Text
            , storyContent  :: Text
            -- ^ we fixed the content as Text
            }
```

Support different representations

Maybe now we need to support multiple formats or representations of the story's content.

```
data StoryContent
  = TopNList Text [(Image, Text)]
  -- ^ E.g. "7 places climate change..."
  | Obituary Image Text
  -- ^ For when a politician or actor dies
  | Opinion Image Text Text
  -- ^ OpEd
  | TweetStorm [(TweetId, Text)]
  -- ^ Tweet-by-tweet analysis of a thread
  | ElectionReport Election [Text]
  -- ^ Append-only election result analysis
```

Poke a hole for the content type

```
data Story content
  = MkStory { storyId        :: Int
            , storyAuthor    :: Author
            , storyTitle     :: Text
            , storySubtitle  :: Maybe Text
            , storyContent   :: content
            -- ^ can now vary content type
            }
```

The `Story` data type became a type constructor (requiring a type argument for content type).

```
>>> :kind Story
Story :: * -> *
```

Story examples

```
>>> let obit
  = Obituary photo obitText

>>> let obitStory
  = MkStory 123 author title Nothing obit

>>> let freeformStory
  = MkStory 124 me "free form story" "woot!"

>>> :t obitStory
Story StoryContent

>>> :t freeformStory
Story Text
```

Let's define our `Functor` instance

```
instance Functor Story where
  fmap :: (a -> b) -> Story a -> Story b
  fmap f (MkStory id auth title sub content)
    = MkStory id auth title sub (f content)

-- countWords :: ToText t => t -> Int
>>> countWords <$> myStory

-- sanitizeHTML :: HTML -> HTML
>>> sanitizeHTML <$> otherStory

-- countOcc :: ToText t => t -> t -> Int
>>> :t countOcc "Trump"
countOcc "Trump" :: ToText t => t -> Int
>>> countOcc "Trump" <$> whitehouseStory
```

Now make the compiler do it!

`Story` has only one type argument which supplies the type of a field in a record (`content` type); we say it is in *"positive position"* and thus can derive a `Functor` automatically like so:

```
{-# LANGUAGE DeriveFunctor #-}

data Story content
  = MkStory { storyId       :: Int
            , storyAuthor   :: Author
            , storyTitle    :: Text
            , storySubtitle :: Maybe Text
            , storyContent  :: content
            } deriving (Functor)
            -- ^^^^ Look Ma, no hands!
```

What does ”positive position” mean?

Basic intuition:

- when a type is only produced it is in *”positive position”*
- when a type must be consumed it is in *”negative position”*

Type	Polarity of `t`
`t`	`+`
`Int -> t`	`+`
`t -> Bool`	`-`
`(Int -> t) -> Bool`	`-`
`(t -> Int) -> Bool`	`+`
`(Int -> t) -> t`	both
`(t -> t) -> Int`	both

Variance: Why does anyone care?

"positive position" $\Longrightarrow$ covariant

"negative position" $\Longrightarrow$ contravariant

none or mixed $\Longrightarrow$ invariant

- covariance is *"varying together in the same direction"*
- contravariance is *"varying in opposite direction"*
- invariance is *"not varying in any direction"*

Helps us choose appropriate abstraction

Variance	Position	Core Abstraction	Package
Covariance	`+`	`Functor`	base
Contravariance	`-`	`Contravariant`	base (GHC >=8.6)
Invariant	none	`Invariant`	invariant

Examples of scaling functions

```haskell
incr, double :: Num a => a -> a
incr = (+1)
-- incr n = n + 1
double = (2*)
-- double n = 2 * n

percentage :: Fractional a => a -> a
percentage = (/100)
-- percentage n = n / 100

celsius, fahrenheit :: Fractional a => a -> a
celsius f = (f - 32) * (5/9)
fahrenheit c = c * (9/5) + 32
```

Using scaling functions on [a]

```
>>> let temperaturesInF = [32, 41, 55, 98]

>>> fmap celsius temperaturesInF
[0.0,5.0,12.777777777777779,36.66666666666666]

>>> celsius <$> temperaturesInF
[0.0,5.0,12.777777777777779,36.66666666666666]

>>> temperaturesInF <&> celsius
[0.0,5.0,12.777777777777779,36.66666666666666]
```

Using ”scaling” functions on `Story a`

Assume we have:

```
redact       :: Text -> Text -> Text
sanitizeHTML :: HTML -> HTML
```

Then we can use these scaling functions (`a -> a`) as so:

```
>>> let story = MkStory 1 me "First story"
 ↪  Nothing "<marquee>I love NixOS!</marquee>"

>>> redact "love" <$> story
MkStory 1 me "First story" Nothing
 ↪  "<marquee>I l*** NixOS!</marquee>"
>>> sanitizeHTML <$> story
MkStory 1 me "First story" Nothing "I love
 ↪  NixOS!"
>>> "I love sleep!" <$ story
```

Converting Story content

Remember the following from prior section:

```
redact        :: Text -> Text -> Text
sanitizeHTML :: HTML -> HTML
```

And assume we add the following:

```
countOcc :: ToText t => Text -> t -> Int
```

```
>>> countOcc "NixOS" <$> story
1
>>> countOcc "NixOS" . redact "NixOS" <$>
 ↪  story
0
```

Structuring Story content

```
title = "This is news!"
subtitle = Just "(not really)"
body = "<b>Content</b> is <crap>here</crap>"

story = MkStory 55 me title subtitle body
badStory = MkStory 56 me title subtitle "<b"

-- Right (Div (B "Content" <> Raw " is here"))
safeStory = sanitize . htmlize <$> story

-- Left "Could not parse into HTML"
safeBadStory = sanitize . htmlize <$> badStory
```

Flatten Story content (render)

```haskell
-- flattenHTML :: Story HTML -> Story Text

render :: Story HTML -> Text
render story = toText $ flattenHTML <$> story
```

When to use what

Scaling functions (`a -> a`) are used to:

- sanitizing/stripping data

Converting functions (`a -> b`) are used to:

- adapt for third party APIs or libraries

Structuring functions (`a -> f b`) are used to:

- add context to pipeline with possibility of failure
- parse into structure for further analysis
- adapt data constructor into applicative pipeline

Flattening functions (`f a -> b`) are used to:

- render

Composing transformations into useful pipelines

```
-- TODO
```

Filters

Folds