Rockin' in a free world

Alessandro Candolini

November 21, 2022

Agenda

1. Free monoids
2. Free monads
3. Polysemy
4. Free applicatives and optparse

Free monoids

Definition (monoid, set-theoretical)

A "monoid" is a tuple (A, φ, e) where

- A is a set[1]
- $\varphi : A \times A \to A$ is a binary *associative* operation on A
- $e \in A$ is a "neutral" element, ie, for every $a \in A$, $\varphi(a, e) = \varphi(e, a) = a$

[1] For all practical purposes, it's convenient to restrict the definition to non-empty sets.

```haskell
class Monoid a where
   (<>) :: a -> a -> a -- binary operation
   mempty :: a  -- neutral element
```

```haskell
class Semigroup a where
  (<>) :: a -> a -> a -- binary operation

class Semigroup a => Monoid a where
  mempty :: a -- neutral element
```

Monoids are everywhere.

Classic examples:

- $(\mathbb{N}, +, 0)$ is a (commutative) monoid
- $(\mathbb{N}, \times, 1)$ is a (commutative) monoid
- String concatenation is a (non-commutative) monoid, with empty string as neutral element
- Singly-linked list concatenation is a (non-commutative) monoid, with empty list as neutral element
- etc

```haskell
{-# LANGUAGE DerivingVia #-}

newtype AdditiveInteger = AdditiveInteger Integer
  deriving (Eq,Show)
  deriving Num via Integer

instance Monoid AdditiveInteger where
   (<>) = (+)
   mempty = 0
```

```haskell
{-# LANGUAGE DerivingVia #-}

newtype MultiplicativeInteger = MultiplicativeInteger
  deriving (Eq,Show)
  deriving Num via Integer

instance Monoid MultiplicativeInteger where
   (<>) = (*)
   mempty = 1
```

```haskell
{-# LANGUAGE FlexibleInstances #-}

instance Monoid String where
  (<>) = (++)
  mempty = ""
```

[1] https://github.com/ghc-proposals/ghc-proposals/pull/279

"Homeworks":

- prove whether (`Double` , $+$, 0) is a Monoid in Scala or not
- prove whether (`BigDecimal` , $+$, 0) is a Monoid in Scala or not
- prove whether sorters on a list of sortable elements can be equipped with a Monoid instance or not.
- what about filter predicates?

```
data Erratum = BannedHashtag
   | PriceLowerThanShippingCost
   | TooManyHashtags
   | .. deriving (Eq, Show)
```

```
instance Monoid [a] where
   (<>) = (++)
   mempty = []
```

`[Erratum]` forms a monoid

(((()()()(()))(())()()((())()()()((

$$A = \{a, b, e\}$$

$$\varphi(a, e) = \varphi(e, a) = a$$

$$\varphi(b, e) = \varphi(e, b) = a$$

$$\varphi(a, b) = e$$

$$\varphi(a, b) \neq e$$

```
data Bicyclic = Bicyclic Int Int
      deriving (Eq, Show)

instance Semigroup Bicyclic where
  (<>) (Bicyclic a b) (Bicyclic c d)
     | b <= c = Bicyclic (a + c -b ) d
     | otherwise = Bicyclic a (d + b - c)

instance Monoid Bicyclic where
  mempty = Balance 0 0
```

```
transform :: Char -> Bicyclic
transform '(' = Bicyclic 0 1
transform ')' = Bicyclic 1 0
transform _ = mempty

balance :: String -> Bicyclic
balance = foldMap transform

balanced = (mempty ==) . balance
```

```haskell
import Test.Hspec

spec :: Spec
spec = describe "Monoidal␣parsing" $ do
 it "()" $
  balance "()" ‘shouldBe‘ Balance 0 0
 it ")(" $
  balance ")(" ‘shouldBe‘ Balance 1 1
 it "))" $
  balance "))" ‘shouldBe‘ Balance 2 0
 it "((" $
  balance "((" ‘shouldBe‘ Balance 0 2S

 it "()((())()(()))(())" $
  balance "()((())()(()))(())" ‘shouldBe‘
     Balance 0 0
```

Free monads

```
import Data.Time

data Todo = Todo String (Maybe UTCTime)
  deriving (Eq,Show)
```

```haskell
program :: Todo -> IO ()
program t@(Todo _ Nothing) = persist t
program t@(Todo _ (Just due))=
  getCurrentTime >>= \ now ->
  if now >= due then
     persist t
  else
     pure () -- ignore errors

-- somewhere
persist :: Todo -> IO()
persist = undefined
```

```haskell
{-# LANGUAGE DeriveFunctor #-}

 data Program a =
   Persist Todo (Program a) |
   Done a deriving (Eq,Show, Functor)
```

```
program' :: Todo -> Program ()
program' t = Persist t (pure ())
```

```haskell
data Program a =
  GetCurrentTime (UTCTime -> Program a) |
  Persist Todo (Program a) |
  Done a deriving (Functor)
```

```
program' :: Todo -> Program ()
program' t@(Todo _ Nothing) = Persist t (pure ())
program' t@(Todo _ (Just due))=
  GetCurrentTime (\ now ->
  if now >= due then
     Persist t (pure ())
  else
     pure () -- ignore errors
     )
```

```haskell
data Free f a
  = Pure a
  | Free (f (Free f a)) deriving (Functor)
```

```haskell
liftF :: Functor f => f a -> Free f a
liftF = Free . fmap Pure
```

```
instance Functor f => Monad (Free f) where
  return = Pure
  Pure x  >>= g  =  g x
  Free fx >>= g  =  Free ((>>= g) <$> fx)
```

```
data ProgramF a =
  GetCurrentTime (UTCTime -> Program a) |
  Persist Todo (Program a) |
  Done a deriving (Functor)
```

```
type Program = Free ProgramF
```

Advantages:

- Managing capabilities precisely
- Better separation of business logic and effectful interpreters

```
interface Clock {
   def now : Instant
}
```

Boilerplate:

- lifting of operations (can be absorbed with metaprogramming)
- Multiple algebras / capabilities
- Copy paste code in almost similar interpreters
- test interpreters (pure ones, eg, State)

Polysemy

Free applicatives and optparse

Questions?