Applicative DSLs

Franz Thoma

BOBKonf 2019, Berlin, 2019-03-22

TNG TECHNOLOGY CONSULTING

Monadic vs. Applicative Effects

The Applicative Class

$$\begin{tikzcd}
\boxed{\text{Functor}} \arrow[r, Rightarrow] & \boxed{\text{Applicative}} \arrow[r, Rightarrow] & \boxed{\text{Monad}} \\
\texttt{fmap} \\ \texttt{(aka map)} & \texttt{pure, liftA2} & \texttt{>>=} \\ \texttt{(aka flatMap)}
\end{tikzcd}$$

```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Monadic Effects

```haskell
>>= :: Monad m
    => m a
    -> (a -> m b)
    -> m b
```

```haskell
ma >>= f = case ma of
    Just a  -> f a
    Nothing -> Nothing
```

Applicative Effects

```haskell
liftA2 :: Applicative m
       ⇒ (a → b → c)
       → m a
       → m b
       → m c
```

```haskell
liftA2 f ma mb = case (ma, mb) of
    (Just a, Just b) → Just (f a b)
    _                → Nothing
```

```haskell
fmap :: Functor m                       liftA2 :: Applicative m
     => (a -> b)                               => (a -> b -> c)
     -> m a                                    -> m a
     -> m b                                    -> m b
                                               -> m c
```

Applicative derived from Monad

```
liftA2 f ma mb = do
    a ← ma
    b ← mb
    pure (f a b)
```

```
liftA2 f ma mb =
    ma >>= ( \a →
        mb >>= ( \b →
            pure (f a b) ) )
```

Syntactic Sugar: `ApplicativeDo`

```
myLiftA2 f ma mb = do
    a ← ma
    b ← mb
    pure (f a b)
```

```
myLiftA2 :: Monad m ⇒ (a → b → c) → m a → m b → m c
myLiftA2 f ma mb =
    ma ≫= (\a →
        mb a ≫= (\b →
            pure (f a b) ) ) )
```

```haskell
{-# LANGUAGE ApplicativeDo #-}
myLiftA2 f ma mb = do
    a <- ma
    b <- mb
    pure (f a b)
```

```haskell
myLiftA2 :: Applicative m => (a -> b -> c) -> m a -> m b -> m c
myLiftA2 f ma mb = liftA2 f ma mb
```

Leveraging `Applicative`

Concurrently

```haskell
liftA2 :: Applicative m ⇒ (a → b → c) → m a → m b → m c
```

```haskell
newtype Concurrently a = Concurrently { runConcurrently :: IO a }

instance Functor Concurrently where …
instance Applicative Concurrently where …
instance Alternative Concurrently where …
```

```haskell
{-# LANGUAGE ApplicativeDo #-}
runConcurrently $ do
    customerMasterData ← Concurrently $ fetchCustomerMasterData cid
    savedShoppingCart ← Concurrently $ fetchShoppingCartForCustomer cid
    pure CustomerProfile
        { shoppingCart = savedShoppingCart
        , name = name customerMasterData
        , age = age customerMasterData }
```

No `Monad` for `Concurrently`

```
do
    a ← ma                                         ma >>= ( \a →
    b ← mb                                             mb >>= ( \b →
    pure (f a b)                                           pure (f a b) ) )
```

Applicative Parsers

`Monad` — Context-Sensitive Languages
`Applicative` — Context-Free Languages

optparse-applicative

```haskell
{-# LANGUAGE ApplicativeDo #-}

data Args = Args { verbose :: Bool, config :: Maybe String, file :: String }
    deriving (Show)

argsP :: Parser Args
argsP = do
    verbose ← switch
        ( long "verbose"
        <> short 'v'
        <> help "Enable verbose output" )
    config ← optional $ strOption
        ( long "config"
        <> short 'c'
        <> metavar "CONFIG_FILE"
        <> help "Override config file" )
    file ← argument str
        ( metavar "INPUT_FILE"
        <> help "The input file" )
    pure Args { verbose = verbose, config = config, file = file }
```

```haskell
main :: IO ()
main = do
    args ← customExecParser (prefs showHelpOnError) (info argsP fullDesc)
    print args
```

```
> example foo.txt
Args {verbose = False, config = Nothing, file = "foo.txt"}
```

```
> example foo.txt -v
Args {verbose = True, config = Nothing, file = "foo.txt"}
```

```
> example foo.txt --config /etc/myconfig
Args {verbose = False, config = Just "/etc/myconfig", file = "foo.txt"}
```

```
> example foo.txt bar.txt
Invalid argument `bar.txt'

Usage: example [-v|--verbose] [-c|--config CONFIG_FILE] INPUT_FILE

Available options:
  -v,--verbose             Enable verbose output
  -c,--config CONFIG_FILE  Override config file
  INPUT_FILE               The input file
```

Validation

```haskell
data Either a b     = Left a    | Right b   deriving (Functor)
data Validation a b = Failure a | Success b deriving (Functor)
```

```haskell
instance Applicative (Either a) where
    pure = Right
    liftA2 f (Left a)  _         = Left a                             -- First error wins
    liftA2 f (Right a) (Left b)  = Left b
    liftA2 f (Right a) (Right b) = Right (f a b)

instance Monoid a ⇒ Applicative (Validation a b) where
    pure = Success
    liftA2 f (Failure a) (Failure b) = Failure (a <> b)          -- Failures are accumulated
    liftA2 f (Failure a) (Success _) = Failure a
    liftA2 f (Success _) (Failure b) = Failure b
    liftA2 f (Success a) (Success b) = Success (f a b)
```

```haskell
instance Monad (Either a) where
    Left a >>= f = Left a
    Right b >>= f = f b

-- instance Monad (Validation a) is not possible
```

```haskell
validCustomer :: String -> String -> Validation [Error] Customer
validCustomer firstName lastName =
    liftA2 Customer
        (validFirstName firstName)
        (validLastName  lastName)
```

Composition

```haskell
newtype Compose f g a = Compose f (g a)

instance (Functor f, Functor g) ⇒ Functor (Compose f g)
instance (Applicative f, Applicative g) ⇒ Applicative (Compose f g)
```

Look ma, no `transformers`

```haskell
type LoggingParser w a = Compose Parser (Writer w) a

parseWithLogging :: LoggingParser w a -> String -> Maybe (a, w)
parseWithLogging (Compose pw) input = fmap runWriter (parse pw input)

anytoken' :: Show a => LoggingParser [String] a
anytoken' = Compose (fmap (\a -> tell ["anytoken: " ++ show a] *> pure a) anytoken)
```

Creating an Applicative DSL

Test Data Generator

- Self-documenting, easy-to-read DSL
- Extensible
- Easily parseable from a config file, e.g. YAML

Structure

```haskell
{-# LANGUAGE RankNTypes #-}

import qualified System.Random as R

newtype Gen a = Gen { runGen :: forall g. R.RandomGen g => g -> a }

instance Functor Gen where
    fmap f (Gen gen) = Gen (f . gen)

instance Applicative Gen where
    pure a = Gen (const a)
    liftA2 f (Gen gen1) (Gen gen2) = Gen $ \g ->
        let (g1, g2) = R.split g
        in  f (gen1 g1) (gen2 g2)
```

Combinators

```haskell
constant :: a -> Gen a
constant = pure

random :: R.Random a => Gen a
random = Gen (fst . R.random)
-- R.random :: (R.RandomGen g, R.Random a) => (a, g)

bounded :: R.Random a => (a, a) -> Gen a
bounded (lo, hi) = Gen (fst . R.randomR (lo, hi))
-- R.randomR :: (R.RandomGen g, R.Random a) => (a, a) -> (a, g)

choose :: [Gen a] -> Gen a
choose gens = Gen $ \g ->
    let (g1, g2) = split g
        ix = fst (R.randomR (0, length gens - 1) g1)
    in  runGen (gens !! ix) g2

pick :: [a] -> Gen a
pick = choose . fmap constant
```

```haskell
optional :: Gen a → Gen (Maybe a)
optional gen = choose [fmap Just gen, pure Nothing]

either :: Gen a → Gen b → Gen (Either a b)
either lgen rgen = choose [fmap Left lgen, fmap Right rgen]

randomString :: Int → Gen String
randomString len = replicateM len (pick latinLetters)
  where
    latinLetters = ['a'..'z'] ++ ['A'..'Z']

csv :: [Gen String] → Gen String
csv columns = fmap (intercalate ",") (sequenceA columns)
```

Applications

```haskell
{-# LANGUAGE ApplicativeDo #-}
data Person = Person { name :: String, age :: Int }

somePerson :: Gen Person
somePerson = do
    randomName ← randomString 10
    randomAge ← bounded (18, 30)
    pure Person { name = randomName, age = randomAge }
```

```yaml
!Person
name: !randomString
  length: 10
age: !bounded
  min: 18
  max: 30
```

Thank you!

Questions?

Thank you!

Slides on Github: fmthoma/applicative-dsls-slides

fmthoma on Github

fmthoma on keybase.io

franz.thoma@tngtech.com

Papers:

Conor McBride, Ross Paterson: Applicative Programming with Effects (2008)

Paolo Capriotti, Ambrus Kaposi: Free Applicative Functors (2014)