Functional Programming

Monad Transformers

Dr. Gabriel Radanne

Albert-Ludwigs-Universität Freiburg, Germany

WS 2017-2018

Reminder: Monad

Definition of a Monad – Lecture 7

- abstract datatype for instructions that produce values
- built-in combination >>=
- abstracts over different interpretations (computations)

Monad definition

The type class Monad

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail :: String -> m a
```

with the following laws:

- `return x >>= f == f x`
- `m >>= return == m`
- `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

What about Composition?

- Applicatives compose (as seen in the lecture on parsing).

What about Composition?

- Applicatives compose (as seen in the lecture on parsing).
- Monads do not necessarily compose.

What about Composition?

- Applicatives compose (as seen in the lecture on parsing).
- Monads do not necessarily compose.
- We sometimes want to use multiples monads at once!

Why combine monads

Lecture 10: Monadic interpreters.

Interpreters can have many features:

- Failure (**`Maybe`**).
- Keeping some state (`State`).
- Reading from the environment (`Reader`).
- ...

To implement an interpreter, we need to combine all these monads!

Let's combine Monads! – State alone

The State monad

```
data ST s a = ST (s -> (a, s))
runST (ST sas) = sas

instance Monad (ST s) where
  return a = ST (\s -> (a, s))
  m >>= f = ST (\s ->
                let (a, s') = runST m s in
                runST (f a) s')
```

Let's combine Monads! – Maybe+State

The MaybeState monad

```haskell
data MaybeState s a = MS { runMS :: s -> Maybe (a, s) }

....

instance Monad (MST s) where
  return a = MST (\s -> Just (a, s))
  ms >>= f = MST (\s -> case runMST ms s of
                     Nothing -> Nothing
                     Just (a,s') -> runMST (f a) s')
```

Let's combine Monads! – Maybe+State

The MaybeState monad

```haskell
data MaybeState s a = MS { runMS :: s -> Maybe (a, s) }

....

instance Monad (MST s) where
  return a = MST (\s -> Just (a, s))
  ms >>= f = MST (\s -> case runMST ms s of
                    Nothing -> Nothing
                    Just (a,s') -> runMST (f a) s')
```

We would have to write this again for each combination!

Alternative solution: Monad transformers

Monad transformers offer a better solution:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

A monad transformer `t` takes a monad `m` and yield a new monad `(t m)`.
`lift` allows to lift a computation from the underlying monad to the new monad.

MaybeT

Definition

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
  return = MaybeT . return . Just
  (MaybeT mmx) >>= f = MaybeT $ do
    mx <- mmx
    case mx of
      Nothing -> return Nothing
      Just x -> runMaybeT (f x)

instance MonadTrans MaybeT where
  lift mx = MaybeT $ do { x <- mx ; return $ Just x }
```

A simple usage of MaybeT

We can recover the "normal" monad by applying to `Identity`.

```
type MaybeLike = MaybeT Identity
```

StateT

Definition

```haskell
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

instance (Monad m) => Monad (StateT m) where
  return a = StateT $ \s -> return (a, s)
  m >>= f = StateT $ \s -> do
      (a, s') <- runStateT m s
      runStateT (f a) s'

instance MonadTrans StateT where
  lift ma = StateT $ \s -> do { a <- ma ; return (a, s) }
```

Let's combine Monads with transformers!

Demo!

ReaderT

Definition

```haskell
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

ask :: (Monad m) => ReaderT r m r
ask = ReaderT return

instance Monad m => Monad (ReaderT r m) where
   return = lift . return
   m >>= k = ReaderT $ \r -> do
             a <- runReaderT m r
             runReaderT (k a) r

instance MonadTrans (ReaderT r) where
   lift m = ReaderT (const m)
```

Back to interpreters

During lecture 10, a monadic interpreter for:

```haskell
data Term = Con Integer
          | Bin Term Op Term
            deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
            deriving (Eq, Show)
```

Back to interpreters

During lecture 10, a monadic interpreter for:

```haskell
data Term = Con Integer
          | Bin Term Op Term
           deriving (Eq, Show)

data Op = Add | Sub | Mul | Div
           deriving (Eq, Show)
```

Different interpreters with various features:

- Failure
- Counting instructions
- Traces

Key points

- Monads do not always compose ...

Key points

- Monads do not always compose ...
- But monad transformers help.

Key points

- Monads do not always compose ...
- But monad transformers help.
- Order is important!

Key points

- Monads do not always compose ...
- But monad transformers help.
- Order is important!
- You should not overdo it.

Key points

- Monads do not always compose ...
- But monad transformers help.
- Order is important!
- You should not overdo it.
- It's all in the `mtl` library.