Functional Programming
Monad Transformers

Dr. Gabriel Radanne

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Reminder: Monad

Definition of a Monad – Previous lecture

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

Monad definition

The type class Monad

```
1 class Monad m where
2   (>>=) :: m a -> (a -> m b) -> m b
3   return :: a -> m a
4   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?

- What does it even mean?

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

- Questions

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

- Questions
  - If f and g are functors, what about Comp f g?

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

- Questions
  - If f and g are functors, what about Comp f g?
  - If f and g are applicatives, what about Comp f g?

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

- Questions
  - If f and g are functors, what about Comp f g?
  - If f and g are applicatives, what about Comp f g?
  - If f and g are monads, what about Comp f g?

What about Composition?

- What does it even mean?
- Given two **functors** f and g, is f . g also a functor?
  I.e., the type constructor that first applies g and then f?
- Can make that more formal

```
1 newtype Comp f g a = Comp (f (g a))
```

- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:

```
1 Comp :: (* -> *) -> (* -> *) -> (* -> *)
```

- Questions
  - If f and g are functors, what about Comp f g?
  - If f and g are applicatives, what about Comp f g?
  - If f and g are monads, what about Comp f g?
- We sometimes want to use multiple functors, applicatives, monads at once!

Why combine functors, applicatives, monads

Lecture 12: 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 start by combining functors!

To show that `Comp f g` is a functor ...

- Implement `fmap` (i.e., give an instance of the **Functor** class)
- Show that the functor laws hold
  - The identity function gets mapped to the identity function.
  - Functor composition commutes with function composition.

Let's combine applicatives!

To show that `Comp f g` is an applicative ...
- Implement pure and (`<*>`) (i.e., give an instance of the `Applicative` class)
- Show that the applicative laws hold ...

Let's combine Monads! – State alone

The State monad

```
1  data ST s a = ST (s -> (a, s))
2  runST (ST sas) = sas
3
4  instance Monad (ST s) where
5    return a = ST (\s -> (a, s))
6    m >>= f = ST (\s ->
7                  let (a, s') = runST m s in
8                  runST (f a) s')
```

Let's combine Monads! – Maybe+State

The MaybeState monad

- Purpose: propagate state and signaling of errors
- Attention: the state is lost

```
1 data MaybeState s a = MST { runMST :: s -> Maybe (a, s) }
2
3 ....
4
5 instance Monad (MST s) where
6   return a = MST (\s -> Just (a, s))
7   ms >>= f = MST (\s -> case runMST ms s of
8                    Nothing -> Nothing
9                    Just (a,s') -> runMST (f a) s')
```

Let's combine Monads! – Maybe+State

The MaybeState monad

- Purpose: propagate state and signaling of errors
- Attention: the state is lost

```haskell
data MaybeState s a = MST { runMST :: 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:

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

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

Alternative solution: Monad transformers

Monad transformers offer a better solution:

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

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

Intermezzo

Alternative solution: Monad transformers

Monad transformers offer a better solution:

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

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

Intermezzo

- What's the kind of t in MonadTrans?

Alternative solution: Monad transformers

Monad transformers offer a better solution:

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

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

Intermezzo

- What's the kind of t in MonadTrans?
- Answer: t :: (* -> *) -> (* -> *)

The MaybeT monad transformer

Definition

```
1 newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
2
3 instance (Monad m) => Monad (MaybeT m) where
4   return = MaybeT . return . Just
5   (MaybeT mmx) >>= f = MaybeT $ do
6     mx <- mmx
7     case mx of
8       Nothing -> return Nothing
9       Just x -> runMaybeT (f x)
10
11 instance MonadTrans MaybeT where
12   lift mx = MaybeT $ mx >>= (return . Just)
```

A simple use of MaybeT

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

```
1 type MaybeLike = MaybeT Identity
```

The StateT monad transformer

Definition

```
1 newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
2
3 instance (Monad m) => Monad (StateT m) where
4   return a = StateT $ \s -> return (a, s)
5   m >>= f = StateT $ \s -> do
6      (a, s') <- runStateT m s
7      runStateT (f a) s'
8
9 instance MonadTrans StateT where
10   lift ma = StateT $ \s -> do { a <- ma ; return (a, s) }
```

Let's combine Monads with transformers!

Demo!

The ReaderT monad transformer

Definition

```
1  newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
2
3  ask :: (Monad m) => ReaderT r m r
4  ask = ReaderT return
5
6  instance Monad m => Monad (ReaderT r m) where
7      return = lift . return
8      m >>= k = ReaderT $ \r -> do
9                a <- runReaderT m r
10               runReaderT (k a) r
11
12 instance MonadTrans (ReaderT r) where
13     lift m = ReaderT (const m)
```

Back to interpreters

During lecture 12, a monadic interpreter for:

```
1 data Term = Con Integer
2          | Bin Term Op Term
3            deriving (Eq, Show)
4
5 data Op = Add | Sub | Mul | Div
6            deriving (Eq, Show)
```

Back to interpreters

During lecture 12, a monadic interpreter for:

```haskell
1 data Term = Con Integer
2          | Bin Term Op Term
3           deriving (Eq, Show)
4
5 data Op = Add | Sub | Mul | Div
6          deriving (Eq, Show)
```

Different interpreters with various features:

- Failure ($\Rightarrow$ exception/Maybe monad)
- Counting instructions ($\Rightarrow$ state monad)
- Traces ($\Rightarrow$ writer monad)

Key points

- Monads do not always compose:
  if m1 and m2 are monads, there is no general definition that makes Comp m1 m2 a monad

Key points

- Monads do not always compose:
  if m1 and m2 are monads, there is no general definition that makes Comp m1 m2 a monad
- But monad transformers help.

Key points

- Monads do not always compose:
  if m1 and m2 are monads, there is no general definition that makes Comp m1 m2 a monad
- But monad transformers help.
- Order is important:
  StateT s **Maybe** $\neq$ MaybeT (ST s)

Key points

- Monads do not always compose:
  if m1 and m2 are monads, there is no general definition that makes `Comp m1 m2` a monad
- But monad transformers help.
- Order is important:
  `StateT s` **Maybe** $\neq$ `MaybeT (ST s)`
- You should not overdo it.

Key points

- Monads do not always compose:
  if m1 and m2 are monads, there is no general definition that makes Comp m1 m2 a monad
- But monad transformers help.
- Order is important:
  StateT s **Maybe** $\neq$ MaybeT (ST s)
- You should not overdo it.
- It's all in the mtl library.