Functional Programming

Monad Transformers

Dr. Gabriel Radanne   Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

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

```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?

- 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

```
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

```
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:

```
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

```
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:

```
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

```
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:

```
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

```
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:

```
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

```
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:

```
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

```
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:

```
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

```
data ST s a = ST { runST :: s -> (s, a) }

instance Functor (ST s) where
  fmap h sg = ST (fmap h . runST sg)

instance Applicative (ST s) where
  pure a = ST (, a)
  ST fab <*> ST fa = ST $ \s -> let (s', f) = fab s in
                                fmap f $ fa s'

instance Monad (ST s) where
  ST fa >>= h = ST $ \s -> let (s', a) = fa s in
                           runST (h a) s'
```

Let's combine Monads! - Maybe+State

Consider `Comp (ST s) Maybe`

- Corresponds to `s -> (s, Maybe a)`, a stateful computation that may fail
- **`return`** and >>= are easy to define

Let's combine Monads! - Maybe+State

Consider `Comp (ST s) Maybe`

- Corresponds to `s -> (s, Maybe a)`, a stateful computation that may fail
- **return** and >>= are easy to define

Consider `Comp Maybe (ST s)`

- Corresponds to `Maybe (s -> (s, a))`
- `return a = return_Maybe (return_ST a)`
- But there's not way to write the bind function:

```
Nothing >>= f = Nothing
Just sta >>= f = ???
```

Let's combine Monads! - Maybe+State

Consider `Comp (ST s) Maybe`

- Corresponds to `s -> (s, Maybe a)`, a stateful computation that may fail
- **return** and >>= are easy to define

Consider `Comp Maybe (ST s)`

- Corresponds to `Maybe (s -> (s, a))`
- `return a = return_Maybe (return_ST a)`
- But there's not way to write the bind function:

```
Nothing >>= f = Nothing
Just sta >>= f = ???
```

Lessons

- Monads do not compose, in general
- Monad composition is not commutative!

Let's combine Monads! – Maybe+State

A different construction: The MaybeState monad

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

```
data MaybeState s a = MST { runMST :: s -> Maybe (s, a) }

....

instance Monad (MST s) where
  return a = MST (\s -> return (s, a))
  ms >>= f = MST (\s -> runMST ms s >>= \(s', a) -> runMST (f a) s')
```

Let's combine Monads! – Maybe+State

A different construction: The MaybeState monad

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

```
data MaybeState s a = MST { runMST :: s -> Maybe (s, a) }

....

instance Monad (MST s) where
  return a = MST (\s -> return (s, a))
  ms >>= f = MST (\s -> runMST ms s >>= \(s', a) -> runMST (f a) s')
```

- Interestingly, the implementation does not depend on **Maybe** at all!

Let's combine Monads! – Maybe+State

A different construction: The MaybeState monad

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

```
data MaybeState s a = MST { runMST :: s -> Maybe (s, a) }

....

instance Monad (MST s) where
  return a = MST (\s -> return (s, a))
  ms >>= f = MST (\s -> runMST ms s >>= \(s', a) -> runMST (f a) s')
```

- Interestingly, the implementation does not depend on **Maybe** at all!
- We don't have to write this definition again for other combinations!

Alternative solution: Monad transformers

We've seen a particular instance of a *monad transformer*:

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

A monad transformer `t` takes a monad `m` and yields a new monad `(t m)`.
Function `lift` moves a computation from the underlying monad to the new monad.

Alternative solution: Monad transformers

We've seen a particular instance of a *monad transformer*:

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

A monad transformer `t` takes a monad `m` and yields a new monad `(t m)`.
Function `lift` moves a computation from the underlying monad to the new monad.

Intermezzo

Alternative solution: Monad transformers

We've seen a particular instance of a *monad transformer*:

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

A monad transformer `t` takes a monad `m` and yields a new monad `(t m)`.
Function `lift` moves a computation from the underlying monad to the new monad.

Intermezzo

- What's the kind of `t` in `MonadTrans`?

Alternative solution: Monad transformers

We've seen a particular instance of a *monad transformer*:

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

A monad transformer `t` takes a monad `m` and yields a new monad `(t m)`.
Function `lift` moves 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

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

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

instance MonadTrans MaybeT where
  lift mx = MaybeT $ mx >>= (return . Just)
```

A simple use of MaybeT

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

```
type MaybeLike = MaybeT Identity
```

The StateT monad transformer

Definition

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

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

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

Let's combine Monads with transformers!

Demo!

The ReaderT monad transformer

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

Earlier we had 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

Earlier we had 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 ($\Rightarrow$ 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.