

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

```
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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ 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 $f \circ g$ also a functor?
I.e., the type constructor that first applies g and then f ?
- Can make that more formal
 - 1 **newtype** $\text{Comp } f \ g \ a = \text{Comp } (f \ (g \ a))$
- The type constructor Comp has an interesting kinding that corresponds to the type of function composition:
 - 1 $\text{Comp} :: (* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow (* \rightarrow *)$
- Questions
 - ▶ If f and g are functors, what about $\text{Comp } f \ g$?
 - ▶ If f and g are applicatives, what about $\text{Comp } f \ g$?
 - ▶ If f and g are monads, what about $\text{Comp } f \ g$?
- 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 $\text{Comp } f \ g$ 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 $\text{Comp } f \ g$ 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

Consider $\text{Comp } (ST \ s) \ \text{Maybe}$

- Corresponds to $s \rightarrow (\text{Maybe } a, s)$, a stateful computation that may fail
- **return** and $\gg=$ are easy to define

Let's combine Monads! - Maybe+State

Consider `Comp (ST s) Maybe`

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

Consider `Comp Maybe (ST s)`

- Corresponds to `Maybe (s -> (a, s))`
- `return a = return_Maybe (return_ST a)`
- But there's not way to write the bind function:
 - 1 `Nothing >=> f = Nothing`
 - 2 `Just sta >=> f = ???`

Let's combine Monads! - Maybe+State

Consider `Comp (ST s) Maybe`

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

Consider `Comp Maybe (ST s)`

- Corresponds to **Maybe** `(s -> (a, s))`
- **return** `a = return_Maybe (return_ST a)`
- But there's not way to write the bind function:
 - 1 **Nothing** `>=> f = Nothing`
 - 2 **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

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

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

```
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 -> return (a, s))  
7   ms >>= f = MST (\s -> runMST ms s >>= \(a, s') -> 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*:

```
1 class MonadTrans t where  
2   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*:

```
1 class MonadTrans t where  
2   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*:

```
1 class MonadTrans t where  
2   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*:

```
1 class MonadTrans t where  
2   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

```
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   mmx >=> f = MaybeT $ do
6     mx <- runMaybeT 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

Earlier we had 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

Earlier we had 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)
```

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 m_1 and m_2 are monads, there is no general definition that makes $\text{Comp } m_1 \ m_2$ a monad

Key points

- Monads do not always compose:
if m_1 and m_2 are monads, there is no general definition that makes $\text{Comp } m_1 \ m_2$ 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 $\text{Comp } m1 \ m2$ a monad
- But monad transformers help.
- Order is important:
 $\text{StateT } s \ \mathbf{Maybe} \neq \text{MaybeT } (ST \ s)$

Key points

- Monads do not always compose:
if m_1 and m_2 are monads, there is no general definition that makes $\text{Comp } m_1 \ m_2$ a monad
- But monad transformers help.
- Order is important:
 $\text{StateT } s \ \mathbf{Maybe} \neq \text{MaybeT } (ST \ s)$
- 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 $\text{Comp } m1 \ m2$ a monad
- But monad transformers help.
- Order is important:
 $\text{StateT } s \ \mathbf{Maybe} \neq \text{MaybeT } (ST \ s)$
- You should not overdo it.
- It's all in the `mtl` library.