

# Monads and More

Advanced Haskell

Andres Löh

10–11 October 2013 — Copyright © 2013 Well-Typed LLP



Monads again

# The `Monad` class

```
class Monad m where  
  return :: a → m a  
  (≫=) :: m a → (a → m b) → m b
```

There's also `fail`, but let's ignore that.

# The `Monad` class

```
class Monad m where
```

```
  return :: a → m a
```

```
  (≫=) :: m a → (a → m b) → m b
```

There's also `fail`, but let's ignore that.

Monads help to abstract from actions

- ▶ that have some sort of effect;
- ▶ that support a notion of embedding,
- ▶ and sequencing.

# Many, many monads

	a	-- some type, no effect
IO	a	-- IO, exceptions, randomness, concurrency, ...
Gen	a	-- random numbers only
ST s	a	-- local mutable variables only
STM	a	-- transaction log variables only
State s	a	-- (persistent) state only
Maybe	a	-- failure only
Error	a	-- exceptions only
Signal	a	-- time-changing value
Eval	a	-- parallel computation
Par	a	-- parallel computation
...		

Note that many of these are completely independent of **IO**, and **IO** is in a way the “worst case” of all of these.

# Advantages of the abstraction

- ▶ Smaller vocabulary, name reuse.
- ▶ **do** notation.
- ▶ A large library of monadic operations such as **mapM**, **replicateM** and many more.

# Monad laws

All monad instances are supposed to adhere to the following laws:

```
return x >>= f  ≡ f x           -- left unit
m >>= return    ≡ m             -- right unit
(m >>= f) >>= g ≡ m >>= (λx → f x >>= g) -- associativity
```

# Monad laws

All monad instances are supposed to adhere to the following laws:

```
return x >>= f  ≡ f x           -- left unit
m >>= return    ≡ m             -- right unit
(m >>= f) >>= g ≡ m >>= (λx → f x >>= g) -- associativity
```

These laws imply that several transformations that seem natural – also on **do** expressions – are valid.



# Implications of monad laws

Intermediate returns have no effect:

```
do  
  comp1  
  return 42  
  comp2
```

is the same as

```
do  
  comp1  
  comp2
```

# Implications of monad laws (contd.)

Returning the result of a final action is superfluous:

```
do
  comp1
  x ← comp2
  return x
```

is the same as

```
do
  comp1
  comp2
```

# Implications of monad laws (contd.)

Inlining computations is valid:

```
phaseA = do { comp1; comp2 }  
phaseB = do { comp3; comp4 }  
all      = do { phaseA; phaseB }
```

is the same as

```
all = do  
  comp1  
  comp2  
  comp3  
  comp4
```

ST

# Local destructive updates

Some computations need access to mutable variables, but nothing else.

From `Control.Monad.ST` and `Data.STRef` which are part of base:

```
data ST s a  -- abstract
data STRef s a  -- abstract
newSTRef  :: a → ST s (STRef s a)
readSTRef :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s ()
```

Very similar to `IO` and `IORef` – but what is the `s` about?

# Regions

- ▶ Because we can run **ST** operations and forget all about their dependencies on mutable variables, we have to make sure that no references to variables can escape.
- ▶ Consider having an **ST** operation return an **STRef** and using it in a different **ST** operation which may run completely independently – the state of the reference would be unclear at that point.

# Regions

- ▶ Because we can run **ST** operations and forget all about their dependencies on mutable variables, we have to make sure that no references to variables can escape.
- ▶ Consider having an **ST** operation return an **STRef** and using it in a different **ST** operation which may run completely independently – the state of the reference would be unclear at that point.
- ▶ The **s** parameter represents the **region** the **ST** computation runs in.
- ▶ Note that we don't mention regions by name, we only introduce constraints: **STRef** actions must have the same region parameter as their surrounding computations.

$\text{runST} :: (\forall s. \text{ST } s \ a) \rightarrow a$

- ▶ This is a locally quantified type (and needs the RankNTypes extension). The argument to **ST** itself must be polymorphic.
- ▶ In particular, an **ST** computations must make no concrete assumptions on where it will run. It's the run-time system making that decision.
- ▶ The important yet somewhat invisible part of the type is that **a** cannot depend on **s**. This ensures that no reference escapes.
- ▶ Both the use of region parameters and of higher-rank types are independently useful in Haskell.



State

# A fresh look at `State`

We can think of `State` as defined like this:

```
newtype State s a = State { runState :: s → (a, s) }
```

# A fresh look at `State`

We can think of `State` as defined like this:

```
newtype State s a = State {runState :: s → (a, s)}
```

Note that this is just record notation and extracts the component.

# A fresh look at `State`

We can think of `State` as defined like this:

```
newtype State s a = State { runState :: s → (a, s) }
```

The kind of `State` is  $* \rightarrow * \rightarrow *$ , as we have two arguments, `s` and `a`, both of kind  $*$ .

# A fresh look at `State`

We can think of `State` as defined like this:

```
newtype State s a = State { runState :: s → (a, s) }
```

The type is in essence a function type taking some state of type `s` and producing a result of type `a` plus a new state of type `s`.

# The monad instance for `State`

```
instance Monad (State s) where  
  return x = State (\s → (x, s))  
  m >>= f = State (\s → case runState m s of  
                      (x, s') → runState (f x) s')
```

## The monad instance for `State`

```
instance Monad (State s) where  
  return x = State (\s → (x, s))  
  m >>= f = State (\s → case runState m s of  
                      (x, s') → runState (f x) s')
```

One of the nice aspects of Haskell's simple evaluation model is that one can easily **prove** that `State` adheres to the monad laws as desired.

# The monad instance for `State`

```
instance Monad (State s) where  
  return x = State (\s → (x, s))  
  m >>= f = State (\s → case runState m s of  
                      (x, s') → runState (f x) s')
```

One of the nice aspects of Haskell's simple evaluation model is that one can easily **prove** that `State` adheres to the monad laws as desired.

Simply convert the left hand side of each law into the right hand side by applying meaning-preserving transformations.



# Proving the left unit law for **State**

$\text{return } x \gg= f$

$\equiv$  { Definition of  $(\gg=)$  }

**State**  $(\lambda s \rightarrow \text{case runState (return } x) \text{ s of}$   
 $(x, s') \rightarrow \text{runState (f } x) \text{ s'})$

$\equiv$  { Definition of **return** }

**State**  $(\lambda s \rightarrow \text{case runState (State } (\lambda s \rightarrow (x, s))) \text{ s of}$   
 $(x, s') \rightarrow \text{runState (f } x) \text{ s'})$

$\equiv$  { Definition of **runState** }

**State**  $(\lambda s \rightarrow \text{case } (\lambda s \rightarrow (x, s)) \text{ s of}$   
 $(x, s') \rightarrow \text{runState (f } x) \text{ s'})$

$\equiv$  { Reducing the lambda }

**State**  $(\lambda s \rightarrow \text{case } (x, s) \text{ of}$   
 $(x, s') \rightarrow \text{runState (f } x) \text{ s'})$

## Proving the left unit law for **State** (contd.)

**State**  $(\lambda s \rightarrow \text{case } (x, s) \text{ of}$   
 $(x, s') \rightarrow \text{runState } (f\ x) \ s')$

$\equiv$  { Reducing the **case** }

**State**  $(\lambda s \rightarrow \text{runState } (f\ x) \ s)$

$\equiv$  { “eta reduction” }

**State**  $(\text{runState } (f\ x))$

$\equiv$  { **State** and **runState** are inverses for a **newtype** }

$f\ x$

# Accessing the state

```
get :: State s s  
get = State $ \s → (s, s)  
put :: s → State s ()  
put s = State $ \_ → ((), s)
```

# Accessing the state

```
get :: State s s
get = State $ \s → (s, s)
put :: s → State s ()
put s = State $ \_ → ((), s)
```

From now on, we can treat `State` as an abstract type:

```
modify :: (s → s) → State s ()
modify f = do
  s ← get f
  put (f s)
```

An interpreter for an expression language

# A datatype of expressions

```
data Expr = Num Int
          | Add Expr Expr
          | Var Name
type Name = String
```

# A datatype of expressions

```
data Expr  = Num Int
           | Add Expr Expr
           | Var Name
type Name = String
```

We'd like to write a function like:

```
eval :: Expr → Int
```

Is this possible?

# A datatype of expressions

```
data Expr  = Num Int
           | Add Expr Expr
           | Var Name
type Name = String
```

We'd like to write a function like:

```
eval :: Expr → Int
```

Is this possible?

We'll need something to deal with the variables.



# Without variables, it's easy

```
data Expr = Num Int  
          | Add Expr Expr
```

```
eval :: Expr → Int  
eval (Num n)      = n  
eval (Add e1 e2) = eval e1 + eval e2
```

# Introducing an environment

```
type Env = Map Name Int
```

# Introducing an environment

```
type Env = Map Name Int
```

```
data Expr = Num Int  
          | Add Expr Expr  
          | Var Name
```

```
type Name = String
```

# Introducing an environment

```
type Env = Map Name Int
```

```
data Expr = Num Int  
         | Add Expr Expr  
         | Var Name
```

```
type Name = String
```

```
eval :: Expr → Env → Int  
eval (Num n)      env = n  
eval (Add e1 e2) env = eval env e1 + eval env e2  
eval (Var x)      env = env ! x
```

Follows the standard design principle for type **Expr** !

# Looking at `eval` closely

```
eval :: Expr → Env → Int
eval (Num n)      env = n
eval (Add e1 e2) env = eval env e1 + eval env e2
eval (Var x)      env = env ! x
```

Question: what does

```
eval (Var "x") empty
```

evaluate to?

# Using Maybe

```
eval :: Expr → Env → Maybe Value
eval (Num n)      _ = Just n
eval (Add e1 e2) env =
  case eval e1 env of
    Nothing → Nothing
    Just r1  → case eval e2 env of
      Nothing → Nothing
      Just r2  → Just (r1 + r2)
eval (Var x)      env = lookup x env
```

This version is safer, but a bit ugly.

# The evolution of the case for **Add**

$\text{eval } (\text{Add } e1 \ e2) = \text{eval } e1 + \text{eval } e2$

$\text{eval } (\text{Add } e1 \ e2) \ \text{env} = \text{eval } e1 \ \text{env} + \text{eval } e2 \ \text{env}$

$\text{eval } (\text{Add } e1 \ e2) \ \text{env} =$

**case**  $\text{eval } e1 \ \text{env}$  **of**

Nothing  $\rightarrow$  Nothing

Just  $r1 \rightarrow$  **case**  $\text{eval } e2 \ \text{env}$  **of**

Nothing  $\rightarrow$  Nothing

Just  $r2 \rightarrow$  Just  $(r1 + r2)$

# The evolution of the case for **Add**

```
eval (Add e1 e2)      = eval e1 + eval e2
eval (Add e1 e2) env = eval e1 env + eval e2 env
eval (Add e1 e2) env =
  case eval e1 env of
    Nothing → Nothing
    Just r1  → case eval e2 env of
      Nothing → Nothing
      Just r2  → Just (r1 + r2)
```

The essential part is adding the two numbers. The rest is clutter:

- ▶ propagating an unchanged environment,
- ▶ propagating errors.



# The evolution of the case for **Add**

```
eval (Add e1 e2)      = eval e1 + eval e2
eval (Add e1 e2) env = eval e1 env + eval e2 env
eval (Add e1 e2) env =
  case eval e1 env of
    Nothing → Nothing
    Just r1  → case eval e2 env of
      Nothing → Nothing
      Just r2  → Just (r1 + r2)
```

The essential part is adding the two numbers. The rest is clutter:

- ▶ propagating an unchanged environment,
- ▶ propagating errors.

Clearly, we should try to use monads to abstract.

# Using State

```
eval :: Expr → State Env Int
eval (Num n)      = return n
eval (Add e1 e2) = do
  r1 ← eval e1
  r2 ← eval e2
  return (r1 + r2)
eval (Var x)      = do
  env ← get
  return (env ! x)
```

This is the monadic version passing the environment, but not treating errors.

We can actually rewrite this in a nicer way ...

# Using State

```
eval :: Expr → State Env Int  
eval (Num n)      = return n  
eval (Add e1 e2) = liftM2 (+) (eval e1) (eval e2)  
eval (Var x)      = liftM (! x) get
```

... or in an even nicer way ...

# Using State

```
eval :: Expr → State Env Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> get
```

...based on the `applicative functor` instance for `State`.

# Every monad is an applicative functor

```
class Functor f  $\Rightarrow$  Applicative f where
```

```
  pure   :: a  $\rightarrow$  f a
```

```
  (<*>) :: f (a  $\rightarrow$  b)  $\rightarrow$  f a  $\rightarrow$  f b
```

```
instance Functor State where
```

```
  fmap = liftM
```

```
instance Applicative State where
```

```
  pure  = return
```

```
  (<*>) = ap
```

# Every monad is an applicative functor

```
class Functor f  $\Rightarrow$  Applicative f where
```

```
  pure   :: a  $\rightarrow$  f a
```

```
  (<*>) :: f (a  $\rightarrow$  b)  $\rightarrow$  f a  $\rightarrow$  f b
```

```
instance Functor State where
```

```
  fmap = liftM
```

```
instance Applicative State where
```

```
  pure  = return
```

```
  (<*>) = ap
```

```
ap mf mx = mf  $\gg=$   $\lambda f \rightarrow$  mx  $\gg=$   $\lambda x \rightarrow$  return (f x)
```

```
liftM f x  = return f 'ap' x
```

# Is every applicative functor also a monad?

No. For example, `ZipList` isn't a monad.

```
newtype ZipList a = ZipList { getZipList :: [a] }  
instance Applicative ZipList where  
  pure x = repeat x  
  fs <*> xs = zipWith ($) fs xs
```

Here, we're using a `newtype` wrapper in order to give another list instance for applicative – the standard one being the monad-induced one.

## Some advice on `Monad` and `Applicative`

- ▶ Applicative notation is very functional in nature and often less verbose than using `do`. You can use it for monads in cases where the rest of the computation does not depend on earlier results.
- ▶ This dependence that `(>>=)` offers is the key difference: monads are more powerful, but it also makes monadic computations less easy to analyze.
- ▶ A motivation for library authors to use applicative functors instead of monads is thus usually that they want to perform more static analysis. `Arrows` are inspired by the same motivation.



## Back to `eval` – do we really need `State`?

```
eval :: Expr → State Env Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> get
```

## Back to `eval` – do we really need `State`?

```
eval :: Expr → State Env Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> get
```

- ▶ The environment never changes – it's just distributed.
- ▶ We were able to define the function with type `Expr → Env → Int` rather than `Expr → Env → (Int, Env)`.
- ▶ It turns out that this is a common pattern which is also a monad.

# The Reader monad

```
newtype Reader r a = Reader {runReader :: r → a}
```

```
instance Monad (Reader r) where
```

```
  return x = Reader (λ_ → x)
```

```
  m >>= f = Reader (λr → runReader (f (runReader m r)) r)
```

# The Reader monad

```
newtype Reader r a = Reader {runReader :: r → a}  
instance Monad (Reader r) where  
  return x = Reader (λ_ → x)  
  m >>= f = Reader (λr → runReader (f (runReader m r)) r)
```

Accessing the state:

```
ask :: Reader r r  
ask = Reader (λr → r)
```

# Locally modifying the state

Perhaps surprisingly, it is still possible to change the state – but only for a local subcomputation:

```
local :: (r → r) → Reader r a → Reader r a  
local f m = Reader (λr → runReader m (f r))
```

## Back to `eval` again

```
eval :: Expr → Reader Env Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> ask
```

Great, almost as before ...

## Back to **eval** again

```
eval :: Expr → Reader Env Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> ask
```

Great, almost as before ...

... but this is still the variant that crashes on unknown variables.  
How to add in **Maybe** ?

# What we could do . . .

Notice that the type we had before was

```
eval :: Expr → Env → Maybe Int
```

so perhaps we could define

```
newtype MaybeReader r a = MR {runMR :: r → Maybe a}
```

and try to make that an instance of monad.



# What we could do ...

Notice that the type we had before was

```
eval :: Expr → Env → Maybe Int
```

so perhaps we could define

```
newtype MaybeReader r a = MR {runMR :: r → Maybe a}
```

and try to make that an instance of monad.

This actually works, but:

- ▶ it's not very modular;
- ▶ we cannot reuse the `Maybe` and `Reader` instances;
- ▶ we also have to redefine `ask`, `local`, and possibly other functions.

# Monad transformers

# The idea

Instead of creating lots of monolithic monads,

- ▶ let's build a toolkit of reusable components,
- ▶ by starting with a very simple monad as basis,
- ▶ and then trying to explain how to add one new aspect to an already existing monad while maintaining the aspects that are already there.

# The basic monad: Identity

```
newtype Identity a = Identity {runIdentity :: a}  
instance Monad Identity where  
  return x = Identity x  
  m >>= f = Identity (runIdentity f (runIdentity m))
```

## Adding a Reader aspect

```
newtype ReaderT r m a = ReaderT {runReaderT :: r → m a}
```

# Adding a Reader aspect

```
newtype ReaderT r m a = ReaderT {runReaderT :: r → m a}
```

Compare with the old

```
newtype Reader r a = Reader {runReader :: r → a}
```

and note that

```
ReaderT r Identity a ≈ Reader r a
```

# Adding a Reader aspect

```
newtype ReaderT r m a = ReaderT {runReaderT :: r → m a}
```

Compare with the old

```
newtype Reader r a = Reader {runReader :: r → a}
```

and note that

```
ReaderT r Identity a ≈ Reader r a
```

Question: What's the kind of ReaderT ?

# Adding a Reader aspect

```
newtype ReaderT r m a = ReaderT {runReaderT :: r → m a}
```

Compare with the old

```
newtype Reader r a = Reader {runReader :: r → a}
```

and note that

$$\text{ReaderT } r \text{ Identity } a \approx \text{Reader } r \ a$$

Question: What's the kind of ReaderT ?

$$\text{ReaderT} :: * \rightarrow (* \rightarrow *) \rightarrow (* \rightarrow *)$$



# ReaderT is really a monad transformer

```
instance Monad m  $\Rightarrow$  Monad (ReaderT r m) where  
  return x = ReaderT ( $\lambda\_ \rightarrow$  return x)  
  m  $\gg=$  f = ReaderT ( $\lambda r \rightarrow$  do  
    a  $\leftarrow$  runReaderT m r  
    runReaderT (f a) r)
```

# ReaderT is really a monad transformer

```
instance Monad m  $\Rightarrow$  Monad (ReaderT r m) where
```

```
  return x = ReaderT ( $\lambda\_ \rightarrow$  return x)
```

```
  m  $\gg=$  f = ReaderT ( $\lambda r \rightarrow$  do  
    a  $\leftarrow$  runReaderT m r  
    runReaderT (f a) r)
```

Contrast this with the old, direct instance for Reader :

```
instance Monad (Reader r) where
```

```
  return x = Reader ( $\lambda\_ \rightarrow$  x)
```

```
  m  $\gg=$  f = Reader ( $\lambda r \rightarrow$  runReader (f (runReader m r)) r)
```

# What about `ask` and `local`?

We can redefine these for `ReaderT r m` rather than `Reader r`, too:

```
ask :: Monad m => ReaderT r m r
```

```
ask = ReaderT (\r -> return r)
```

```
local :: Monad m => (r -> r) -> ReaderT r m a -> ReaderT r m a
```

```
local f m = ReaderT (\r -> runReaderT m (f r))
```

## Back to eval

The library `Control.Monad.Reader` in package `mtl` actually defines:

```
type Reader r = ReaderT r Identity
```

## Back to `eval`

The library `Control.Monad.Reader` in package `mtl` actually defines:

```
type Reader r = ReaderT r Identity
```

The definition of `eval` is not affected at all:

```
eval :: Expr → Reader Env Int  
eval (Num n)      = return n  
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2  
eval (Var x)      = (! x) <$> ask
```

## Back to `eval`

The library `Control.Monad.Reader` in package `mtl` actually defines:

```
type Reader r = ReaderT r Identity
```

The definition of `eval` is not affected at all:

```
eval :: Expr → Reader Env Int  
eval (Num n)      = return n  
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2  
eval (Var x)      = (! x) <$> ask
```

Can we now add the error aspect, too?

# The `ErrorT` monad transformer

```
newtype ErrorT e m a = ErrorT { runErrorT :: m (Either e a) }  
class Error a where  
  noMsg :: a  
  strMsg :: String → a  
instance (Monad m) ⇒ Monad (ErrorT e m) where  
  return x = ErrorT (return (Right x))  
  m >>= f = ErrorT (do  
    ea ← runErrorT m  
    case ea of  
      Left err → return (Left err)  
      Right a → runErrorT (f a))
```

# Using the **ErrorT** monad transformer

We note that

$$\text{ErrorT String (Reader Env) } a \approx \text{Env} \rightarrow (\text{Either String } a)$$



# Using the `ErrorT` monad transformer

We note that

$$\text{ErrorT String (Reader Env) } a \approx \text{Env} \rightarrow (\text{Either String } a)$$

and define:

```
eval :: Expr → ErrorT String (Reader Env) Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = ...
```

# Using the **ErrorT** monad transformer

We note that

$$\text{ErrorT String (Reader Env) } a \approx \text{Env} \rightarrow (\text{Either String } a)$$

and define:

```
eval :: Expr → ErrorT String (Reader Env) Int
eval (Num n)      = return n
eval (Add e1 e2)  = (+) <$> eval e1 <*> eval e2
eval (Var x)      = ...
```

We'd like to call **ask** at this point, but that's a type error.

# Abstracting from `ask` and `local`

- ▶ We now have several different types – all monad transformer stacks involving `ReaderT` – that should support `ask`.
- ▶ Fortunately, we can define a type class for this purpose ... or can we?

# Abstracting from `ask` and `local`

- ▶ We now have several different types – all monad transformer stacks involving `ReaderT` – that should support `ask`.
- ▶ Fortunately, we can define a type class for this purpose ... or can we?

```
class Monad m  $\Rightarrow$  MonadReader m where  
  ask :: m ...
```

# Abstracting from `ask` and `local`

- ▶ We now have several different types – all monad transformer stacks involving `ReaderT` – that should support `ask`.
- ▶ Fortunately, we can define a type class for this purpose ... or can we?

```
class Monad m  $\Rightarrow$  MonadReader m where  
  ask :: m ...
```

We have the next problem – we cannot put `r` here, we have to gain access to the type of the state being distributed.

# Multi-parameter type classes

# Classes are relations

- ▶ Type classes can be seen as predicates (or unary relations) on types.
- ▶ In addition, any type that is in the relation must support the type class methods.

# Classes are relations

- ▶ Type classes can be seen as predicates (or unary relations) on types.
- ▶ In addition, any type that is in the relation must support the type class methods.
- ▶ It seems natural to generalize this to n-ary relations and allow type classes with multiple type parameters.
- ▶ We will need the `MultiParamTypeClasses` and `FlexibleInstances` extensions.



## Another attempt

```
class Monad m  $\Rightarrow$  MonadReader r m where
```

```
ask  :: m r
```

```
local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```

```
instance Monad m  $\Rightarrow$  MonadReader r (ReaderT r m) where
```

```
-- as before:
```

```
ask      = ReaderT ( $\lambda$ r  $\rightarrow$  return r)
```

```
local f m = ReaderT ( $\lambda$ r  $\rightarrow$  runReaderT m (f r))
```

## Another attempt

```
class Monad m  $\Rightarrow$  MonadReader r m where
  ask  :: m r
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
instance Monad m  $\Rightarrow$  MonadReader r (ReaderT r m) where
  -- as before:
  ask      = ReaderT (\r  $\rightarrow$  return r)
  local f m = ReaderT (\r  $\rightarrow$  runReaderT m (f r))
```

It is in the instance that we establish the correspondence between the type of the state and the monad type.

## Another attempt

```
class Monad m  $\Rightarrow$  MonadReader r m where
  ask  :: m r
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
instance Monad m  $\Rightarrow$  MonadReader r (ReaderT r m) where
  -- as before:
  ask      = ReaderT ( $\lambda$ r  $\rightarrow$  return r)
  local f m = ReaderT ( $\lambda$ r  $\rightarrow$  runReaderT m (f r))
```

It is in the instance that we establish the correspondence between the type of the state and the monad type.

However, there are still problems.

# Unresolved overloading, revisited

Even with single-parameter type classes, there are situations where GHC fails to resolve overloading without further type annotations:

```
strange :: Read a ⇒ String → String -- incorrect  
strange x = show (read x)
```

The intermediate type is ambiguous, and may affect the result.

# Unresolved overloading, revisited

Even with single-parameter type classes, there are situations where GHC fails to resolve overloading without further type annotations:

```
strange :: Read a => String -> String  -- incorrect  
strange x = show (read x)
```

The intermediate type is ambiguous, and may affect the result.

Unfortunately, in the presence of multi-parameter type classes, this problem occurs much more frequently!

# Example

```
example :: Reader Int Bool  -- incorrect  
example = ask >>= \s → return (s == 0)
```

This might seem like a reasonable definition at first.

# Example

```
example :: (Num r, Eq r, MonadReader r (Reader Int)) =>  
          Reader Int Bool  -- incorrect  
example = ask >>= \s -> return (s == 0)
```

But the code gives rise to these constraints.

There are no further hints to say that `r` should be `Int`, and while there is our instance matching

```
instance MonadReader Int (Reader Int)
```

there's nothing that would prevent users from defining other instances such as

```
instance MonadReader Char (Reader Int)
```

as well.

# Functional dependencies

The solution lies in using another language extension  
FunctionalDependencies:

```
class Monad m  $\Rightarrow$  MonadReader r m | m  $\rightarrow$  r where  
  ask  :: m r  
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```



# Functional dependencies

The solution lies in using another language extension  
FunctionalDependencies:

```
class Monad m  $\Rightarrow$  MonadReader r m | m  $\rightarrow$  r where  
  ask  :: m r  
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```

- ▶ The **functional dependency** tells GHC that for all combinations of **m** and **r** in the relation, knowledge of **m** must be sufficient to uniquely determine **r**.

# Functional dependencies

The solution lies in using another language extension  
FunctionalDependencies:

```
class Monad m  $\Rightarrow$  MonadReader r m | m  $\rightarrow$  r where  
  ask  :: m r  
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```

- ▶ The **functional dependency** tells GHC that for all combinations of **m** and **r** in the relation, knowledge of **m** must be sufficient to uniquely determine **r**.
- ▶ So given that there is an instance for **MonadReader r (ReaderT r m)**, there must be no other instances involving **ReaderT** as the second argument.
- ▶ In compensation for this restriction, GHC can now resolve the ambiguity in our **example** automatically.

Associated types, type families

# A second look at the functional dependency

```
class Monad m  $\Rightarrow$  MonadReader r m | m  $\rightarrow$  r where  
  ask  :: m r  
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```

What this is stating is that the type class instance implicitly define a **function on the type level** from certain monads to their reader state.

# A second look at the functional dependency

```
class Monad m  $\Rightarrow$  MonadReader r m | m  $\rightarrow$  r where  
  ask  :: m r  
  local :: (r  $\rightarrow$  r)  $\rightarrow$  m a  $\rightarrow$  m a
```

What this is stating is that the type class instance implicitly define a **function on the type level** from certain monads to their reader state.

There's another way we are allowed to express this function, using the `TypeFamilies` extension:

```
class Monad m  $\Rightarrow$  MonadReader m where  
  type EnvType m  
  ask  :: m (EnvType m)  
  local :: (EnvType m  $\rightarrow$  EnvType m)  $\rightarrow$  m a  $\rightarrow$  m a
```

# Associated types

```
class Monad m  $\Rightarrow$  MonadReader m where  
  type EnvType m  
  ask  :: m (EnvType m)  
  local :: (EnvType m  $\rightarrow$  EnvType m)  $\rightarrow$  m a  $\rightarrow$  m a
```

The type synonym is also overloaded, and called an [associated type](#). In every instance, we can provide a definition for the type synonym.

```
instance Monad m  $\Rightarrow$  MonadReader (ReaderT r m) where  
  type EnvType (ReaderT r m) = r  
  ...    -- rest as before
```

# Type families

More or less equivalently, the associated type can also be lifted out of the class:

```
type family EnvType m
class Monad m  $\Rightarrow$  MonadReader m where
  ask  :: m (EnvType m)
  local :: (EnvType m  $\rightarrow$  EnvType m)  $\rightarrow$  m a  $\rightarrow$  m a
type instance EnvType (ReaderT r m) = r
instance Monad m  $\Rightarrow$  MonadReader (ReaderT r m) where
  ...    -- rest as before
```

# Type families

More or less equivalently, the associated type can also be lifted out of the class:

```
type family EnvType m
class Monad m  $\Rightarrow$  MonadReader m where
  ask  :: m (EnvType m)
  local :: (EnvType m  $\rightarrow$  EnvType m)  $\rightarrow$  m a  $\rightarrow$  m a
type instance EnvType (ReaderT r m) = r
instance Monad m  $\Rightarrow$  MonadReader (ReaderT r m) where
  ...    -- rest as before
```

This syntax is mainly advantageous in cases where a type function is useful independently of one single type class.



# Type families vs. monad transformers

- ▶ Monad transformers and type families are incredibly powerful, and can be used to perform [type-level programming](#) – defining limited computations on the type level, all evaluated at compile time.
- ▶ Type families have been introduced much more recently as a more functional alternative to the very relational functional dependencies.
- ▶ Both are supported in GHC for the time being, with type families now perhaps a bit closer to making it into the standard.
- ▶ For mainly historical reasons, the default monad transformer library `mtl` uses functional dependencies – but there are replacements using type families such as `monads-tf`.

# Lifting monad interfaces

# Back to the original problem

```
eval :: Expr → ErrorT String (Reader Env) Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = ...
```

The real question is: can we make an error-transformed reader monad into an instance of `MonadReader` ?

# Lifting a monad through a transformer

```
instance (Error e, MonadReader r m)  
    ⇒ MonadReader r (ErrorT e m) where  
    ask      = ErrorT (liftM Right ask)  
    local f m = ErrorT (local f (runErrorT m))
```

# Lifting a monad through a transformer

```
instance (Error e, MonadReader r m)
    ⇒ MonadReader r (ErrorT e m) where
  ask      = ErrorT (liftM Right ask)
  local f m = ErrorT (local f (runErrorT m))
```

- ▶ This instance requires the `UndecidableInstances` extension.
- ▶ In similar ways, we can lift several monad-specific interfaces (such as `MonadReader`) through all sorts of other monads.

## Back where we once were ...

```
eval :: Expr → ErrorT String (Reader Env) Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = (! x) <$> ask
```

Now, we still have to actually make use of `ErrorT` .

# The `MonadError` class

```
class Error a where
```

```
  noMsg :: a
```

```
  strMsg :: String → a
```

```
class Monad m ⇒ MonadError e m | m → e where
```

```
  throwError :: e → m a
```

```
  catchError :: m a → (e → m a) → m a
```

# Triggering an error

```
eval :: Expr → ErrorT String (Reader Env) Int
eval (Num n)      = return n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Var x)      = do
  env ← ask
  case lookup x env of
    Nothing → throwError "unknown variable"
    Just v  → return v
```



# What have we achieved?

- ▶ We introduced a lot of machinery, but using it is not hard.
- ▶ We can combine different effects (error, pieces of state) by stacking monad transformers on top of each other.
- ▶ Even if we change the monad, nearly all code can remain unchanged – only parts related to the new functionality have to be adapted.

10

# Monad transformers and IO

- ▶ Due to the special nature of IO, there is no way of defining an IO monad transformer.
- ▶ But IO can still be combined with other monads. In such cases, IO replaces Identity as the base of the transformer stack.
- ▶ Unlike State or Reader or Error, the IO monad has a rather large “interface” of IO-specific operations, and we need a way to lift these.

# Lifting IO operations

```
class Monad m  $\Rightarrow$  MonadIO m where  
  liftIO :: IO a  $\rightarrow$  m a
```

# Lifting IO operations

```
class Monad m  $\Rightarrow$  MonadIO m where  
  liftIO :: IO a  $\rightarrow$  m a
```

Example instances:

```
instance MonadIO IO where  
  liftIO m = m
```

```
instance (MonadIO m)  $\Rightarrow$  MonadIO (ReaderT r m) where  
  liftIO m = ReaderT ( $\lambda$ r  $\rightarrow$  liftIO m)
```

## More monad transformers

# State

For completeness, let's look at **State** once more:

```
newtype StateT s m a = StateT { runStateT :: s → m (a, s) }  
type State s = StateT s Identity  
class Monad m ⇒ MonadState s m | m → s where  
  get :: m s  
  put :: s → m ()
```

A **Writer** monad is another “partial” state monad, suitable for example for logging purposes:

```
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}  
type Writer w = WriterT w Identity  
class (Monoid w, Monad m)  
    => MonadWriter w m | m -> w where  
    tell    :: w -> m ()  
    listen  :: m a -> m (a, w)  
    pass    :: m (a, w -> w) -> m a
```



# Monoids

Monoids are algebraic structures (defined in `Data.Monoid`) with a neutral element and an associative binary operation:

```
class Monoid a where
  mempty  :: a
  mappend :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr mappend mempty

(◇) :: Monoid a ⇒ a → a → a
(◇) = mappend

instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

There are many (potential) instances of monoid, and often several for one type (via `newtype` wrappers).

# Lessons

- ▶ You can design complex monads by stacking together some monad transformers.
- ▶ Independent libraries can offer particular aspects as monad transformers.
- ▶ As a result of combining such aspects, one stack can involve several occurrences of a particular transformer internally.
- ▶ Most code only changes if it's actually affected by a new aspect.
- ▶ The most important monad transformers are state variants and error variants. There are also list-based monad transformers and continuation monad transformers, and some more.