Monads and More

Advanced Haskell

Andres Löh

10–11 October 2013 —

Well-Typed

The Haskell Consultants

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:

$$
\begin{array}{lll}
\mathsf{return}\ x \gg\!= f & \equiv f\ x & \text{-- left unit} \\
m \gg\!= \mathsf{return} & \equiv m & \text{-- right unit} \\
(m \gg\!= f) \gg\!= g & \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g) & \text{-- associativity}
\end{array}
$$

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.

Running ST

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

- 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:

$$\textbf{newtype}\ \mathsf{State}\ \mathsf{s}\ \mathsf{a} = \mathsf{State}\ \{\mathsf{runState} :: \mathsf{s} \rightarrow (\mathsf{a}, \mathsf{s})\}$$

A fresh look at State

We can think of State as defined like this:

$\textbf{newtype}\ \mathsf{State\ s\ a} = \mathsf{State}\ \{\boxed{\mathsf{runState} ::}\ \mathsf{s} \rightarrow (\mathsf{a}, \mathsf{s})\}$

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:

$$\textbf{newtype}\ \mathsf{State}\ \boxed{\mathsf{s\ a}} = \mathsf{State}\ \{\mathsf{runState} :: \mathsf{s} \to (\mathsf{a}, \mathsf{s})\}$$

The kind of State is $* \to * \to *$, 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:

$$\textbf{newtype}\ \mathsf{State}\ s\ a = \mathsf{State}\ \{\mathsf{runState} :: \boxed{s \to (a, s)}\}$$

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 (λs → (x, s))
  m >>= f  = State (λs → case runState m s of
                            (x, s') → runState (f x) s')
```

The monad instance for State

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ s)\ \textbf{where} \\
\quad \mathsf{return}\ x = \mathsf{State}\ (\lambda s \to (x, s)) \\
\quad m \gg\!= f \ \ = \mathsf{State}\ (\lambda s \to \textbf{case}\ \mathsf{runState}\ m\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (x, s') \to \mathsf{runState}\ (f\ x)\ s')
\end{array}
$$

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

$$
\begin{aligned}
&\textbf{instance}\ \text{Monad}\ (\text{State}\ s)\ \textbf{where} \\
&\quad \text{return}\ x = \text{State}\ (\lambda s \to (x, s)) \\
&\quad m \gg\!= f\ \ = \text{State}\ (\lambda s \to \textbf{case}\ \text{runState}\ m\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (x, s') \to \text{runState}\ (f\ x)\ s')
\end{aligned}
$$

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

$$
\begin{array}{ll}
& \mathsf{return}\ x \gg\!= f \\
\equiv & \{\ \text{Definition of}\ (\gg\!=)\ \} \\
& \mathsf{State}\ (\lambda s \rightarrow \mathbf{case}\ \mathsf{runState}\ (\mathsf{return}\ x)\ s\ \mathbf{of} \\
& \qquad\qquad\qquad (x, s') \rightarrow \mathsf{runState}\ (f\ x)\ s') \\
\equiv & \{\ \text{Definition of}\ \mathsf{return}\ \} \\
& \mathsf{State}\ (\lambda s \rightarrow \mathbf{case}\ \mathsf{runState}\ (\mathsf{State}\ (\lambda s \rightarrow (x, s)))\ s\ \mathbf{of} \\
& \qquad\qquad\qquad (x, s') \rightarrow \mathsf{runState}\ (f\ x)\ s') \\
\equiv & \{\ \text{Definition of}\ \mathsf{runState}\ \} \\
& \mathsf{State}\ (\lambda s \rightarrow \mathbf{case}\ (\lambda s \rightarrow (x, s))\ s\ \mathbf{of} \\
& \qquad\qquad\qquad (x, s') \rightarrow \mathsf{runState}\ (f\ x)\ s') \\
\equiv & \{\ \text{Reducing the lambda}\ \} \\
& \mathsf{State}\ (\lambda s \rightarrow \mathbf{case}\ (x, s)\ \mathbf{of} \\
& \qquad\qquad\qquad (x, s') \rightarrow \mathsf{runState}\ (f\ x)\ s')
\end{array}
$$

Proving the left unit law for State (contd.)

$$
\begin{array}{ll}
 & \mathsf{State}\ (\lambda s \rightarrow \mathbf{case}\ (x, s)\ \mathbf{of} \\
 & \qquad\qquad (x, s') \rightarrow \mathsf{runState}\ (f\ x)\ s') \\
\equiv & \quad \{\ \text{Reducing the } \mathbf{case}\ \} \\
 & \mathsf{State}\ (\lambda s \rightarrow \mathsf{runState}\ (f\ x)\ s) \\
\equiv & \quad \{\ \text{“eta reduction”}\ \} \\
 & \mathsf{State}\ (\mathsf{runState}\ (f\ x)) \\
\equiv & \quad \{\ \mathsf{State} \text{ and } \mathsf{runState} \text{ are inverses for a } \mathbf{newtype}\ \} \\
 & f\ x
\end{array}
$$

Accessing the state

$$
\begin{aligned}
&\mathsf{get} :: \mathsf{State}\ \mathsf{s}\ \mathsf{s} \\
&\mathsf{get} = \mathsf{State}\ \$\ \lambda \mathsf{s} \rightarrow (\mathsf{s}, \mathsf{s}) \\
&\mathsf{put} :: \mathsf{s} \rightarrow \mathsf{State}\ \mathsf{s}\ () \\
&\mathsf{put}\ \mathsf{s} = \mathsf{State}\ \$\ \lambda\_ \rightarrow ((), \mathsf{s})
\end{aligned}
$$

Accessing the state

```
get :: State s s
get = State $ λs → (s, s)

put :: s → State s ()
put s = State $ λ_ → ((), 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

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr} &&= \mathsf{Num}\ \mathsf{Int}\\
& &&\mid\ \mathsf{Add}\ \mathsf{Expr}\ \mathsf{Expr}\\
& &&\mid\ \mathsf{Var}\ \mathsf{Name}\\
&\textbf{type}\ \mathsf{Name} &&= \mathsf{String}
\end{aligned}
$$

We'd like to write a function like:

$$\mathsf{eval} :: \mathsf{Expr} \rightarrow \mathsf{Int}$$

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

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

```haskell
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

```
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 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 ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → 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 $\gg\!=$ λf $\rightarrow$ mx $\gg\!=$ λx $\rightarrow$ 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 that 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 $(\gg=)$ 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?

$$
\begin{array}{ll}
\text{eval} :: \text{Expr} \rightarrow \text{State Env Int} & \\
\text{eval (Num n)} & = \text{return n} \\
\text{eval (Add e1 e2)} & = (+) <\$> \text{eval e1} <*> \text{eval e2} \\
\text{eval (Var x)} & = (!\ \text{x}) <\$> \text{get}
\end{array}
$$

Back to eval – do we really need State?

$$
\begin{array}{lcl}
\mathsf{eval} :: \mathsf{Expr} \to \mathsf{State\ Env\ Int} & & \\
\mathsf{eval\ (Num\ n)} & = & \mathsf{return\ n} \\
\mathsf{eval\ (Add\ e1\ e2)} & = & (+) <\$> \mathsf{eval\ e1} <*> \mathsf{eval\ e2} \\
\mathsf{eval\ (Var\ x)} & = & (!\ \mathsf{x}) <\$> \mathsf{get}
\end{array}
$$

- The environment never changes – it's just distributed.
- We were able to define the function with type $\mathsf{Expr} \to \mathsf{Env} \to \mathsf{Int}$ rather than $\mathsf{Expr} \to \mathsf{Env} \to (\mathsf{Int}, \mathsf{Env})$.
- 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

$$\textbf{newtype}\ \text{Reader r a} = \text{Reader}\ \{\text{runReader} :: \text{r} \rightarrow \text{a}\}$$
$$\textbf{instance}\ \text{Monad (Reader r)}\ \textbf{where}$$
$$\quad \text{return x} = \text{Reader}\ (\lambda\_ \rightarrow \text{x})$$
$$\quad \text{m} \gg\!= \text{f} \ = \text{Reader}\ (\lambda \text{r} \rightarrow \text{runReader (f (runReader m r)) r})$$

Accessing the state:

$$\text{ask} :: \text{Reader r r}$$
$$\text{ask} = \text{Reader}\ (\lambda \text{r} \rightarrow \text{r})$$

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

$$\text{eval} :: \text{Expr} \to \text{Env} \to \text{Maybe Int}$$

so perhaps we could define

$$\textbf{newtype}\ \text{MaybeReader r a} = \text{MR}\ \{\text{runMR} :: \text{r} \to \text{Maybe a}\}$$

and try to make that an instance of monad.

What we could do . . .

Notice that the type we had before was

$$\mathsf{eval} :: \mathsf{Expr} \to \mathsf{Env} \to \mathsf{Maybe\ Int}$$

so perhaps we could define

$$\textbf{newtype}\ \mathsf{MaybeReader\ r\ a} = \mathsf{MR}\ \{\mathsf{runMR} :: \mathsf{r} \to \mathsf{Maybe\ a}\}$$

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

$$\textbf{newtype}\ \mathsf{ReaderT}\ r\ m\ a = \mathsf{ReaderT}\ \{\mathsf{runReaderT} :: r \to m\ a\}$$

Adding a Reader aspect

$$\textbf{newtype}\ \text{ReaderT r m a} = \text{ReaderT}\ \{\text{runReaderT} :: \text{r} \to \text{m a}\}$$

Compare with the old

$$\textbf{newtype}\ \text{Reader r a} = \text{Reader}\ \{\text{runReader} :: \text{r} \to \text{a}\}$$

and note that

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

Adding a Reader aspect

$$\textbf{newtype}\ \mathsf{ReaderT}\ \mathsf{r}\ \mathsf{m}\ \mathsf{a} = \mathsf{ReaderT}\ \{\mathsf{runReaderT} :: \mathsf{r} \to \mathsf{m}\ \mathsf{a}\}$$

Compare with the old

$$\textbf{newtype}\ \mathsf{Reader}\ \mathsf{r}\ \mathsf{a} = \mathsf{Reader}\ \{\mathsf{runReader} :: \mathsf{r} \to \mathsf{a}\}$$

and note that

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

Question: What's the kind of ReaderT ?

Adding a Reader aspect

$$\textbf{newtype}\ \mathsf{ReaderT}\ r\ m\ a = \mathsf{ReaderT}\ \{\mathsf{runReaderT} :: r \to m\ a\}$$

Compare with the old

$$\textbf{newtype}\ \mathsf{Reader}\ r\ a = \mathsf{Reader}\ \{\mathsf{runReader} :: r \to a\}$$

and note that

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

Question: What's the kind of ReaderT ?

$$\mathsf{ReaderT} :: * \to (* \to *) \to (* \to *)$$

ReaderT is really a monad transformer

```
instance Monad m => Monad (ReaderT r m) where
  return x = ReaderT (\_ -> return x)
  m >>= f  = ReaderT (\r -> do
                a <- runReaderT m r
                runReaderT (f a) r)
```

ReaderT is really a monad transformer

```haskell
instance Monad m => Monad (ReaderT r m) where
  return x = ReaderT (\_ -> return x)
  m >>= f  = ReaderT (\r -> do
                 a <- runReaderT m r
                 runReaderT (f a) r)
```

Contrast this with the old, direct instance for Reader:

```haskell
instance Monad (Reader r) where
  return x = Reader (\_ -> x)
  m >>= f  = Reader (\r -> runReader (f (runReader m r)) r)
```

What about ask and local?

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

$$
\begin{aligned}
&\mathsf{ask} :: \mathsf{Monad}\ \mathsf{m} \Rightarrow \mathsf{ReaderT}\ \mathsf{r}\ \mathsf{m}\ \mathsf{r} \\
&\mathsf{ask} = \mathsf{ReaderT}\ (\lambda \mathsf{r} \to \mathsf{return}\ \mathsf{r}) \\
&\mathsf{local} :: \mathsf{Monad}\ \mathsf{m} \Rightarrow (\mathsf{r} \to \mathsf{r}) \to \mathsf{ReaderT}\ \mathsf{r}\ \mathsf{m}\ \mathsf{a} \to \mathsf{ReaderT}\ \mathsf{r}\ \mathsf{m}\ \mathsf{a} \\
&\mathsf{local}\ \mathsf{f}\ \mathsf{m} = \mathsf{ReaderT}\ (\lambda \mathsf{r} \to \mathsf{runReaderT}\ \mathsf{m}\ (\mathsf{f}\ \mathsf{r}))
\end{aligned}
$$

Back to eval

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

$$\textbf{type}\ \mathrm{Reader}\ r = \mathrm{ReaderT}\ r\ \mathrm{Identity}$$

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

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

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

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

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 ⇒ MonadReader m where
  ask :: m ...
```

Well-Typed

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 ⇒ 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 ⇒ MonadReader r m where
  ask   ::  m r
  local ::  (r → r) → m a → m a
instance Monad m ⇒ MonadReader r (ReaderT r m) where
    -- as before:
  ask       = ReaderT (λr → return r)
  local f m = ReaderT (λr → runReaderT m (f r))
```

Another attempt

```
class Monad m ⇒ MonadReader r m where
  ask   ::  m r
  local ::  (r → r) → m a → m a
instance Monad m ⇒ MonadReader r (ReaderT r m) where
    -- as before:
  ask       = ReaderT (λr → return r)
  local f m = ReaderT (λr → 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 ⇒ MonadReader r m where
  ask   ::  m r
  local ::  (r → r) → m a → m a
instance Monad m ⇒ MonadReader r (ReaderT r m) where
    -- as before:
  ask       = ReaderT (λr → return r)
  local f m = ReaderT (λr → 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:

$$\begin{array}{l} \text{strange} :: \boxed{\text{Read a} \Rightarrow} \text{String} \rightarrow \text{String} \quad \text{-- incorrect} \\ \text{strange x} = \text{show (read x)} \end{array}$$

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

$$\text{example} :: \text{Reader Int Bool} \quad \text{-- incorrect}$$
$$\text{example} = \text{ask} \ggg\!= \lambda s \rightarrow \text{return}\ (s == 0)$$

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 >>= λs → 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 ⇒ MonadReader r m | m → r where
  ask   :: m r
  local :: (r → r) → m a → m a
```

Functional dependencies

The solution lies in using another language extension `FunctionalDependencies`:

```
class Monad m ⇒ MonadReader r m | m → r where
  ask   :: m r
  local :: (r → r) → m a → 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 ⇒ MonadReader r m | m → r where
  ask   :: m r
  local :: (r → r) → m a → 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 ambiguitity in our `example` automatically.

Associated types, type families

A second look at the functional dependency

$\textbf{class}$ Monad m $\Rightarrow$ MonadReader r m $|$ m $\rightarrow$ r $\textbf{where}$
  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 ⇒ MonadReader r m | m → r where
  ask   :: m r
  local :: (r → r) → m a → 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 ⇒ MonadReader m where
  type EnvType m
  ask   :: m (EnvType m)
  local :: (EnvType m → EnvType m) → m a → m a
```

Associated types

```
class Monad m ⇒ MonadReader m where
  type EnvType m
  ask   :: m (EnvType m)
  local :: (EnvType m → EnvType m) → m a → 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 ⇒ 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 ⇒ MonadReader m where
  ask   :: m (EnvType m)
  local :: (EnvType m → EnvType m) → m a → m a
type instance EnvType (ReaderT r m) = r
instance Monad m ⇒ 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 ⇒ MonadReader m where
  ask   :: m (EnvType m)
  local :: (EnvType m → EnvType m) → m a → m a
type instance EnvType (ReaderT r m) = r
instance Monad m ⇒ 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

$$
\begin{array}{ll}
\mathsf{eval} :: \mathsf{Expr} \to \mathsf{ErrorT\ String\ (Reader\ Env)\ Int} & \\
\mathsf{eval\ (Num\ n)} & = \mathsf{return\ n} \\
\mathsf{eval\ (Add\ e1\ e2)} & = (+) <\$> \mathsf{eval\ e1} <*> \mathsf{eval\ e2} \\
\mathsf{eval\ (Var\ x)} & = \ \ldots
\end{array}
$$

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.

IO

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 ⇒ MonadIO m where
  liftIO :: IO a → m a
```

Lifting IO operations

```
class Monad m ⇒ MonadIO m where
  liftIO :: IO a → m a
```

Example instances:

```
instance MonadIO IO where
  liftIO m = m
instance (MonadIO m) ⇒ MonadIO (ReaderT r m) where
  liftIO m = ReaderT (λr → 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 ()
```

Writer

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.