Applied Functional Programming

USCS 2011

Utrecht University

Andres Löh

Well-Typed

August 17, 2011

C3. Monads

In this lecture

- A number of useful programming patterns.
- We will see a similarity between seemingly different concepts.

Maybe

The Maybe type

```
data Maybe a = Nothing
             | Just a
```

The `Maybe` datatype is often used to encode failure or an exceptional value:

```
lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe b
find   :: (a → Bool) → [a] → Maybe a
```

Encoding exceptions using Maybe

Assume that we have a (Zipper-like) data structure with the following operations:

$$\begin{aligned} &\mathsf{up}, \mathsf{down}, \mathsf{right} :: \mathsf{Loc} \to \mathsf{Maybe\ Loc} \\ &\mathsf{update} \qquad\qquad :: (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Loc} \to \mathsf{Loc} \end{aligned}$$

Given a location l_1, we want to move up, right, down, and update the resulting position with using $\mathsf{update}\ (+1)$ ...
Each of the steps can fail.

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2   → case right l2 of
                Nothing → Nothing
                Just l3   → case down l3 of
                              Nothing → Nothing
                              Just l4   → Just (update (+1) l4)
```

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2   → case right l2 of
                Nothing → Nothing
                Just l3   → case down l3 of
                              Nothing → Nothing
                              Just l4   → Just (update (+1) l4)
```

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2   → case right l2 of
                Nothing → Nothing
                Just l3   → case down l3 of
                              Nothing → Nothing
                              Just l4   → Just (update (+1) l4)
```

In essence, we need

- a way to sequence function calls and use their results if successful
- a way to modify or produce successful results.

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2   → case right l2 of
                Nothing → Nothing
                Just l3   → case down l3 of
                              Nothing → Nothing
                              Just l4   → Just (update (+1) l4)
```

Sequencing:

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
f >>= g = case f of
            Nothing → Nothing
            Just x    → g x
```

Encoding exceptions using Maybe (contd.)

```
up l1 >>=

  λ l2        → case right l2 of
                  Nothing → Nothing
                  Just l3   → case down l3 of
                                Nothing → Nothing
                                Just l4   → Just (update (+1) l4)
```

Sequencing:

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
f >>= g = case f of
            Nothing → Nothing
            Just x    → g x
```

Encoding exceptions using Maybe (contd.)

$$
\begin{array}{l}
\text{up } l_1 \ \gg\!= \\
\quad \lambda\, l_2 \quad \rightarrow \ \text{right } l_2 \ \gg\!= \\
\qquad\qquad \lambda\, l_3 \quad \rightarrow \ \textbf{case} \text{ down } l_3 \textbf{ of} \\
\qquad\qquad\qquad\qquad \text{Nothing} \rightarrow \text{Nothing} \\
\qquad\qquad\qquad\qquad \text{Just } l_4 \ \rightarrow \text{Just (update } (+1)\ l_4)
\end{array}
$$

Sequencing:

$$
\begin{array}{l}
(\gg\!=) :: \text{Maybe a} \rightarrow (\text{a} \rightarrow \text{Maybe b}) \rightarrow \text{Maybe b} \\
\text{f} \gg\!= \text{g} = \textbf{case} \text{ f } \textbf{of} \\
\qquad\qquad \text{Nothing} \rightarrow \text{Nothing} \\
\qquad\qquad \text{Just x} \ \rightarrow \text{g x}
\end{array}
$$

Encoding exceptions using Maybe (contd.)

```
up l1 >>=
  \ l2 -> right l2 >>=
            \ l3 -> down l3 >>=
                      \ l4 -> Just (update (+1) l4)
```

Sequencing:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
            Nothing -> Nothing
            Just x  -> g x
```

Sequencing and embedding

$$
\begin{aligned}
&\text{up } l_1 \gg\!= \\
&\quad \lambda l_2 \to \text{right } l_2 \gg\!= \\
&\qquad\qquad \lambda l_3 \to \text{down } l_3 \gg\!= \\
&\qquad\qquad\qquad\qquad \lambda l_4 \to \text{Just (update } (+1)\ l_4)
\end{aligned}
$$

Sequencing and embedding

```
up l1 >>=
  \l2 -> right l2 >>=
            \l3 -> down l3 >>=
                     \l4 -> return (update (+1) l4)
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g   = case f of
              Nothing -> Nothing
              Just x  -> g x

return :: a -> Maybe a
return x = Just x
```

Sequencing and embedding

$$
\begin{aligned}
&\text{up } l_1 \gg\!= \\
&\quad \lambda l_2 \to \text{right } l_2 \gg\!= \\
&\qquad\qquad \lambda l_3 \to \text{down } l_3 \gg\!= \\
&\qquad\qquad\qquad \lambda l_4 \to \boxed{\text{return (update } (+1)\ l_4)}
\end{aligned}
$$

$$
\begin{aligned}
&(\gg\!=) :: \text{Maybe a} \to (\text{a} \to \text{Maybe b}) \to \text{Maybe b} \\
&\text{f} \gg\!= \text{g} \quad = \textbf{case } \text{f } \textbf{of} \\
&\qquad\qquad \text{Nothing} \to \text{Nothing} \\
&\qquad\qquad \text{Just x} \quad \to \text{g x} \\
&\text{return} :: \text{a} \to \text{Maybe a} \\
&\text{return x} = \text{Just x}
\end{aligned}
$$

$$(\text{up } l_1) \gg\!= \text{right} \gg\!= \text{down} \gg\!= \text{return} \circ \text{update } (+1)$$

Observation

Code looks a bit like imperative code. Compare:

$$
\begin{array}{ll}
\text{up } l_1 & \gg\!= \lambda l_2 \rightarrow \\
\text{right } l_2 & \gg\!= \lambda l_3 \rightarrow \\
\text{down } l_3 & \gg\!= \lambda l_4 \rightarrow \\
\text{return (update } (+1)\ l_4) &
\end{array}
$$

$$
\begin{array}{l}
l_2 := \text{up } l_1; \\
l_3 := \text{right } l_2; \\
l_4 := \text{down } l_3; \\
\textbf{return} \text{ update } (+1)\ l_4
\end{array}
$$

- In the imperative language, the occurrence of possible exceptions is a side effect.
- Haskell is more explicit because we use the `Maybe` type and the appropriate sequencing operation.

A variation: Either

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

A variation: Either

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

The datatype `Maybe` can encode exceptional function results (i.e., failure), but no information can be associated with `Nothing`. We cannot dinstinguish different kinds of errors.

A variation: Either

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

The datatype `Maybe` can encode exceptional function results (i.e., failure), but no information can be associated with `Nothing`. We cannot dinstinguish different kinds of errors.

Using `Either`, we can use `Left` to encode errors, and `Right` to encode successful results.

Example

```
type Error = String

fac :: Int → Either Error Int
fac 0                 = Right 1
fac n | n > 0         = case fac (n − 1) of
                          Left  e → Left e
                          Right r → Right (n ∗ r)
      | otherwise = Left "fac: negative argument"
```

Structure of sequencing looks similar to the sequencing for `Maybe`.

Sequencing and returning for `Either`

We can define variations of the operatons for `Maybe`:

```
(>>=) :: Either Error a → (a → Either Error b) → Either Error b
f >>= g = case f of
            Left  e → Left e
            Right x → g x

return :: a → Either Error a
return x = Right x
```

Sequencing and returning for Either

We can define variations of the operatons for Maybe :

```
(>>=) :: Either Error a → (a → Either Error b) → Either Error b
f >>= g = case f of
            Left  e → Left e
            Right x → g x

return :: a → Either Error a
return x = Right x
```

The function can now be written as:

```
fac :: Int → Either Error Int
fac 0                 = return 1
fac n | n > 0         = fac (n − 1) >>= λr → return (n ∗ r)
      | otherwise     = Left "fac: negative argument"
```

Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: Error → Either Error a
throwError e = Left e
```

Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: Error → Either Error a
throwError e = Left e

catchError :: Either Error a          →    -- computation
   (e → Either Error a)               →    -- handler
   Either Error a
catchError f handler = case f of
                          Left  e → handler e
                          Right x → Right x
```

State

Maintaining state explicitly

- We pass state to a function as an argument.
- The function modifies the state and produces it as a result.
- If the function does anything except modifying the state, we must return a tuple (or a special-purpose datatype with multiple fields).

This motivates the following type definition:

```
type State s a = s → (a, s)
```

Using state

There are many situations where maintaining state is useful:

- using a random number generator

```
type Random a = State StdGen a
```

- using a counter to generate unique labels

```
type Counter a = State Int a
```

- maintaining the complete current configuration of an application (an interpreter, a game, ...) using a user-defined datatype

```
data ProgramState = ...
type Program a = State ProgramState a
```

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \rightarrow \textbf{let}\ & (\mathsf{lvl}\ , s_2) = \mathsf{generateLevel}\ \ s_1 \\
& (\mathsf{lvl}', s_3) = \mathsf{generateStairs}\ \mathsf{lvl}\ \ s_2 \\
& (\mathsf{ms}, s_4) = \mathsf{placeMonsters}\ \mathsf{lvl}'\ s_3 \\
\textbf{in}\ & (\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}, s_4)
\end{aligned}
$$

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \to \textbf{let}\ & (\mathsf{lvl}\ , s_2) = \mathsf{generateLevel}\ \ s_1 \\
& (\mathsf{lvl}', s_3) = \mathsf{generateStairs}\ \mathsf{lvl}\ \ s_2 \\
& (\mathsf{ms}, s_4) = \mathsf{placeMonsters}\ \mathsf{lvl}'\ s_3 \\
\textbf{in}\ & (\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}, s_4)
\end{aligned}
$$

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \to \textbf{let}\ & (\mathsf{lvl}\ , s_2) = \mathsf{generateLevel}\ \ s_1 \\
& (\mathsf{lvl}', s_3) = \mathsf{generateStairs}\ \mathsf{lvl}\ \ s_2 \\
& (\mathsf{ms}, s_4) = \mathsf{placeMonsters}\ \mathsf{lvl}'\ s_3 \\
\textbf{in}\ & (\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}, s_4)
\end{aligned}
$$

Again, we need

- a way to sequence function calls and use their results
- a way to modify or produce successful results.

Bind and return for state

$$
\begin{aligned}
\lambda s_1 \to \textbf{let}\ & (\mathsf{lvl}\ , s_2) = \mathsf{generateLevel}\ \ \ \ \ \ \ \ s_1 \\
& (\mathsf{lvl}', s_3) = \mathsf{generateStairs}\ \mathsf{lvl}\ \ s_2 \\
& (\mathsf{ms}, s_4) = \mathsf{placeMonsters}\ \mathsf{lvl}'\ s_3 \\
\textbf{in}\ & (\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}, s_4)
\end{aligned}
$$

$$
\begin{aligned}
& (\gg\!=) :: \mathsf{State}\ s\ a \to (a \to \mathsf{State}\ s\ b) \to \mathsf{State}\ s\ b \\
& f \gg\!= g \quad = \lambda s \to \textbf{let}\ (x, s') = f\ s\ \textbf{in}\ g\ x\ s' \\
& \mathsf{return} :: a \to \mathsf{State}\ s\ a \\
& \mathsf{return}\ x = \lambda s \to (x, s)
\end{aligned}
$$

Bind and return for state

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad \text{generateLevel} \quad \gg\!= \lambda \text{lvl} \rightarrow \\
& \lambda s_2 \rightarrow \textbf{let}\ (\text{lvl}', s_3) = \text{generateStairs lvl}\ \ s_2 \\
& \qquad\qquad\quad (\text{ms}, s_4) = \text{placeMonsters lvl}'\ s_3 \\
& \qquad\quad \textbf{in}\ (\text{combine lvl}'\ \text{ms}, s_4)
\end{aligned}
$$

$$
\begin{aligned}
& (\gg\!=) :: \text{State s a} \rightarrow (\text{a} \rightarrow \text{State s b}) \rightarrow \text{State s b} \\
& \text{f} \gg\!= \text{g} \quad = \lambda \text{s} \rightarrow \textbf{let}\ (\text{x}, \text{s}') = \text{f s}\ \textbf{in}\ \text{g x s}' \\
& \text{return} :: \text{a} \rightarrow \text{State s a} \\
& \text{return x} = \lambda \text{s} \rightarrow (\text{x}, \text{s})
\end{aligned}
$$

Bind and return for state

$$
\begin{aligned}
& \text{generateLevel} && \gg\!= \lambda \text{lvl} \rightarrow \\
& \text{generateStairs lvl} && \gg\!= \lambda \text{lvl}' \rightarrow \\
\lambda s_3 \rightarrow\ & \textbf{let}\ (\text{ms}, s_4) = \text{placeMonsters lvl}'\ s_3 \\
& \textbf{in}\ (\text{combine lvl}'\ \text{ms}, s_4)
\end{aligned}
$$

$$
\begin{aligned}
& (\gg\!=) :: \text{State s a} \rightarrow (\text{a} \rightarrow \text{State s b}) \rightarrow \text{State s b} \\
& \text{f} \gg\!= \text{g} \quad = \lambda \text{s} \rightarrow \textbf{let}\ (\text{x}, \text{s}') = \text{f s}\ \textbf{in}\ \text{g x s}' \\
& \text{return} :: \text{a} \rightarrow \text{State s a} \\
& \text{return x} = \lambda \text{s} \rightarrow (\text{x}, \text{s})
\end{aligned}
$$

Bind and return for state

$$
\begin{aligned}
& \text{generateLevel} && \gg\!= \lambda \text{lvl} \rightarrow \\
& \text{generateStairs lvl} && \gg\!= \lambda \text{lvl}' \rightarrow \\
& \text{placeMonsters lvl}' && \gg\!= \lambda \text{ms} \rightarrow \\
\lambda s_4 \rightarrow \quad & (\text{combine lvl}' \text{ ms}, s_4)
\end{aligned}
$$

$$
\begin{aligned}
& (\gg\!=) :: \text{State s a} \rightarrow (\text{a} \rightarrow \text{State s b}) \rightarrow \text{State s b} \\
& \text{f} \gg\!= \text{g} \quad = \lambda \text{s} \rightarrow \textbf{let } (\text{x}, \text{s}') = \text{f s } \textbf{in } \text{g x s}' \\
& \text{return} :: \text{a} \rightarrow \text{State s a} \\
& \text{return x} = \lambda \text{s} \rightarrow (\text{x}, \text{s})
\end{aligned}
$$

Bind and return for state

$$
\begin{aligned}
\text{generateLevel} &\gg\!= \lambda \text{lvl} \rightarrow \\
\text{generateStairs lvl} &\gg\!= \lambda \text{lvl}' \rightarrow \\
\text{placeMonsters lvl}' &\gg\!= \lambda \text{ms} \rightarrow \\
\text{return (combine lvl}' \text{ ms)}
\end{aligned}
$$

$$
\begin{aligned}
&(\gg\!=) :: \text{State s a} \rightarrow (\text{a} \rightarrow \text{State s b}) \rightarrow \text{State s b} \\
&\text{f} \gg\!= \text{g} \quad = \lambda \text{s} \rightarrow \textbf{let } (\text{x}, \text{s}') = \text{f s } \textbf{in } \text{g x s}' \\
&\text{return} :: \text{a} \rightarrow \text{State s a} \\
&\text{return x} = \lambda \text{s} \rightarrow (\text{x}, \text{s})
\end{aligned}
$$

Observation

Again, the code looks a bit like imperative code. Compare:

$$
\begin{array}{ll}
\text{generateLevel} & \gg\!= \lambda \text{lvl} \rightarrow \\
\text{generateStairs lvl} & \gg\!= \lambda \text{lvl}' \rightarrow \\
\text{placeMonsters lvl}' & \gg\!= \lambda \text{ms} \rightarrow \\
\text{return (combine lvl}' \text{ ms)} &
\end{array}
$$

$$
\begin{array}{l}
\text{lvl} := \text{generateLevel}; \\
\text{lvl}' := \text{generateStairs lvl}; \\
\text{ms} := \text{placeMonsters lvl}'; \\
\textbf{return}\ \text{combine lvl}' \text{ ms}
\end{array}
$$

- In the imperative language, the occurrence of memory updates (random numbers) is a side effect.
- Haskell is more explicit because we use the `State` type and the appropriate sequencing operation.

“Primitive” operations for state handling

We can completely hide the implementation of `State` if we provide the following two operations as an interface:

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

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

```
inc :: State Int ()
inc =
  get >>= λs → put (s + 1)
```

“Primitive” operations for state handling

We can completely hide the implementation of `State` if we provide the following two operations as an interface:

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

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

```
inc :: State Int ()
inc =
  get >>= λs → put (s + 1)
```

Actually, Haskell’s `Control.Monad.State` module uses a slightly different implementation:

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

List

Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
concat (map length (map words (concat (map lines txts))))
```

Easier to understand with a list comprehension:

```
[length w | t ← txts, l ← lines t, w ← words l]
```

We can also define sequencing and embedding, i.e., `(>>=)` and `return`:

```
(>>=) :: [a] → (a → [b]) → [b]
xs >>= f = concat (map f xs)

return :: a → [a]
return x = [x]
```

Using bind and return for lists

```
concat (map length (map words (concat (map lines txts))))
```

$$
\begin{array}{ll}
\text{txts} & \gg\!= \lambda \text{t} \rightarrow \\
\text{lines t} & \gg\!= \lambda \text{l} \rightarrow \\
\text{words l} & \gg\!= \lambda \text{w} \rightarrow \\
\text{return (length w)} &
\end{array}
$$

Using bind and return for lists

```
concat (map length (map words (concat (map lines txts))))
```

$$
\begin{aligned}
&\text{txts} && \gg\!= \lambda \text{t} \rightarrow \\
&\text{lines t} && \gg\!= \lambda \text{l} \rightarrow \\
&\text{words l} && \gg\!= \lambda \text{w} \rightarrow \\
&\text{return (length w)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{t} && := \text{txts} \\
&\text{l} && := \text{lines t} \\
&\text{w} && := \text{words w} \\
&\textbf{return}\ \text{length w}
\end{aligned}
$$

Using bind and return for lists

```
concat (map length (map words (concat (map lines txts))))
```

```
txts      >>= λt →
lines t   >>= λl →
words l   >>= λw →
return (length w)
```

```
t  := txts
l  := lines t
w  := words w
return length w
```

- Again, we have a similarity to imperative code.
- In the imperative language, we have implicit nondeterminism (one or all of the options are chosen).
- In Haskell, we are explicit by using the list datatype and explicit sequencing using `(>>=)`.

Intermediate Summary

At least three types with `(>>=)` and `return`:

- for `Maybe`, `(>>=)` sequences operations that may trigger exceptions and shortcuts evaluation once an exception is encountered; `return` embeds a function that never throws an exception;
- for `State`, `(>>=)` sequences operations that may modify some state and threads the state through the operations; `return` embeds a function that never modifies the state;
- for `[]`, `(>>=)` sequences operations that may have multiple results and executes subsequent operations for each of the previous results; `return` embeds a function that only ever has one result.

There is a common interface here!

The `Monad` class

Monad class

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

- The name “monad” is borrowed from category theory.
- A monad is an algebraic structure similar to a monoid.
- Monads have been popularized in functional programming via the work of Moggi and Wadler.

Instances

```
instance Monad Maybe where
    ...
instance (Error e) ⇒ Monad (Either e) where
    ...
instance Monad [] where
    ...
newtype State s a = State {runState :: s → (a, s)}
instance Monad (State s) where
    ...
```

do notation

Like list comprehensions, **do** notation is a form of syntactic sugar. Unlike list comprehensions, **do** notation is not restricted to a single datatype, but applicable to all monads.

```
generateLevel        >>= \lvl ->
generateStairs lvl   >>= \lvl' ->
placeMonsters lvl'   >>= \ms ->
return (combine lvl' ms)
```

```
do
  lvl  <- generateLevel
  lvl' <- generateStairs lvl
  ms <- placeMonsters lvl'
  return (combine lvl' ms)
```

do notation – contd.

$$
\begin{aligned}
&\text{up } l_1 && \gg\!= \lambda l_2 \rightarrow \\
&\text{right } l_2 && \gg\!= \lambda l_3 \rightarrow \\
&\text{down } l_3 && \gg\!= \lambda l_4 \rightarrow \\
&\text{return (update } (+1)\ l_4) &&
\end{aligned}
$$

do

$l_2 \leftarrow \text{up } l_1$

$l_3 \leftarrow \text{right } l_2$

$l_4 \leftarrow \text{down } l_3$

$\text{return (update } (+1)\ l_4)$

More on `do` notation

- Use it, it is usually more concise.
- Never forget that it is just syntactic sugar. Use `(>>=)` and `(>>)` directly when it is more convenient.

More on `do` notation

- Use it, it is usually more concise.
- Never forget that it is just syntactic sugar. Use `(>>=)` and `(>>)` directly when it is more convenient.
- Remember that `return` is just a normal function:
  - Not every **do**-block ends with a `return`.
  - `return` can be used in the middle of a **do**-block, and it doesn't "jump" anywhere.

More on `do` notation

- Use it, it is usually more concise.
- Never forget that it is just syntactic sugar. Use `(>>=)` and `(>>)` directly when it is more convenient.
- Remember that `return` is just a normal function:
  - Not every **do**-block ends with a `return`.
  - `return` can be used in the middle of a **do**-block, and it doesn't "jump" anywhere.
- Not every monad computation has to be in a **do**-block. In particular **do** e is the same as e.
- On the other hand, you may have to "repeat" the **do** in some places, for instance in the branches of an **if**.

IO is a monad

The IO monad

Another type with actions that require sequencing.

The IO monad

Another type with actions that require sequencing.

The IO monad is special in several ways:

- IO is a primitive type, and $(\gg\!=)$ and return for IO are primitive functions,
- there is no (politically correct) function runIO :: IO a → a, whereas for most other monads there is a corresponding function,
- values of IO a denote side-effecting programs that can be executed by the run-time system.

The IO monad

Another type with actions that require sequencing.

The IO monad is special in several ways:

- IO is a primitive type, and $(\gg\!=)$ and return for IO are primitive functions,
- there is no (politically correct) function runIO :: IO a → a, whereas for most other monads there is a corresponding function,
- values of IO a denote side-effecting programs that can be executed by the run-time system.

Note that the specialty of IO has really not much to do with being a monad.

IO, internally

```
Main⟩ :i IO
newtype IO a
  = GHC.IOBase.IO (GHC.Prim.State # GHC.Prim.RealWorld
      → (# GHC.Prim.State # GHC.Prim.RealWorld, a #))
      -- Defined in GHC.Types
Main⟩ :i GHC.Prim.RealWorld
data GHC.Prim.RealWorld   -- Defined in GHC.Prim
```

Internally, GHC models `IO` as a state monad having the “real world” as state!

The role of `IO` in Haskell

More and more features have been integrated into `IO`, for instance:

- classic file and terminal IO

  `putStr, hPutStr`

- references

  `newIORef, readIORef, writeIORef`

- access to the system

  `getArgs, getEnvironment, getClockTime`

- exceptions

  `throwIO, catch`

- concurrency

  `forkIO`

The role of `IO` in Haskell (contd.)

- Because of its special status, the `IO` monad provides a safe and convenient way to express all these constructs in Haskell. Haskell's purity (referential transparency) is not compromised, and equational reasoning can be used to reason about `IO` programs.

The role of `IO` in Haskell (contd.)

- Because of its special status, the `IO` monad provides a safe and convenient way to express all these constructs in Haskell. Haskell's purity (referential transparency) is not compromised, and equational reasoning can be used to reason about `IO` programs.
- A program that involves `IO` in its type can do everything. The absence of `IO` tells us a lot, but its presence does not allow us to judge what kind of `IO` is performed.

The role of `IO` in Haskell (contd.)

- Because of its special status, the `IO` monad provides a safe and convenient way to express all these constructs in Haskell. Haskell's purity (referential transparency) is not compromised, and equational reasoning can be used to reason about `IO` programs.
- A program that involves `IO` in its type can do everything. The absence of `IO` tells us a lot, but its presence does not allow us to judge what kind of `IO` is performed.
- It would be nice to have more fine-grained control on the effects a program performs.
- For some, but not all effects in `IO`, we can use or build specialized monads.

Lifting functions to monads

```
liftM  :: (Monad m) ⇒ (a → b)       → m a → m b
liftM2 :: (Monad m) ⇒ (a → b → c) → m a → m b → m c
...
liftM  f x   = return f `ap` x
liftM2 f x y = return f `ap` x `ap` y
...
```

Question

What is liftM $(+1)$ $[1 \mathinner{..} 5]$?

Lifting functions to monads

```
liftM  :: (Monad m) ⇒ (a → b)       → m a → m b
liftM2 :: (Monad m) ⇒ (a → b → c) → m a → m b → m c
...
liftM  f x   = return f `ap` x
liftM2 f x y = return f `ap` x `ap` y
...
```

Question

What is liftM $(+1)$ $[1\,..\,5]$?

Answer

Same as map $(+1)$ $[1\,..\,5]$. The function liftM generalizes map to arbitrary monads.

Monadic map

```
mapM  :: (Monad m) ⇒ (a → m b) → [a] → m [b]
mapM_ :: (Monad m) ⇒ (a → m b) → [a] → m ()

mapM f []       = return []
mapM f (x : xs) = liftM2 (:) (f x) (mapM f xs)

mapM_ f []       = return ()
mapM_ f (x : xs) = f x ≫ mapM_ f xs
```

Sequencing monadic actions

```
sequence  :: (Monad m) ⇒ [m a] → m [a]
sequence_ :: (Monad m) ⇒ [m a] → m ()

sequence  = foldr (liftM2 (:)) (return [])
sequence_ = foldr (≫) (retun ())
```

Monadic fold

```
foldM :: (Monad m) ⇒ (a → b → m a) → a → [b] → m a
foldM op e []       = return e
foldM op e (x : xs) = do r ← op e x
                         foldM f r xs
```

More monadic operations

Browse `Control.Monad`:

```
filterM      :: (Monad m) ⇒ (a → m Bool) → [a] → m [a]
replicateM   :: (Monad m) ⇒ Int → m a → m [a]
replicateM_  :: (Monad m) ⇒ Int → m a → m ()
join         :: (Monad m) ⇒ m (m a) → m a
when         :: (Monad m) ⇒ Bool → m () → m ()
unless       :: (Monad m) ⇒ Bool → m () → m ()
forever      :: (Monad m) ⇒ m a → m ()
```

...and more!