Monads

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 —

Well-Typed
The Haskell Consultants

The plan

Let us look at a number of datatypes and typical programming problems involving these types . . .

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 data structure with the following operations:

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

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.)

$$
\begin{array}{l}
\textbf{case}\ \boxed{\mathsf{up}\ l_1}\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\quad \mathsf{Just}\ \boxed{l_2} \rightarrow \textbf{case}\ \boxed{\mathsf{right}\ l_2}\ \textbf{of} \\
\qquad\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad\qquad \mathsf{Just}\ \boxed{l_3} \rightarrow \textbf{case}\ \boxed{\mathsf{down}\ l_3}\ \textbf{of} \\
\qquad\qquad\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad\qquad\qquad \mathsf{Just}\ \boxed{l_4} \rightarrow \mathsf{Just}\ \boxed{(\mathsf{update}\ (+\ 1)\ l_4)}
\end{array}
$$

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.)

$$
\begin{array}{l}
\textbf{case}\ \mathsf{up}\ \mathsf{l}_1\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\quad \mathsf{Just}\ \mathsf{l}_2 \quad \rightarrow \textbf{case}\ \mathsf{right}\ \mathsf{l}_2\ \textbf{of} \\
\qquad\qquad\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad\qquad\qquad \mathsf{Just}\ \mathsf{l}_3 \quad \rightarrow \textbf{case}\ \mathsf{down}\ \mathsf{l}_3\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Just}\ \mathsf{l}_4 \quad \rightarrow \mathsf{Just}\ (\mathsf{update}\ (+\ 1)\ \mathsf{l}_4)
\end{array}
$$

Sequencing:

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

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.)

$$\begin{array}{l}
\text{up}\ \mathsf{l}_1 \ggg \\
\quad \lambda\,\mathsf{l}_2 \rightarrow \text{right}\ \mathsf{l}_2 \ggg \\
\qquad\qquad \lambda\,\mathsf{l}_3 \rightarrow \textbf{case}\ \text{down}\ \mathsf{l}_3\ \textbf{of} \\
\qquad\qquad\qquad \text{Nothing} \rightarrow \text{Nothing} \\
\qquad\qquad\qquad \text{Just}\ \mathsf{l}_4 \rightarrow \text{Just}\ (\text{update}\ (+\ 1)\ \mathsf{l}_4)
\end{array}$$

Sequencing:

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

Encoding exceptions using Maybe (contd.)

$$
\begin{array}{l}
\mathsf{up}\ \mathsf{l}_1 \ggg= \\
\quad \lambda\, \mathsf{l}_2 \rightarrow \mathsf{right}\ \mathsf{l}_2 \ggg= \\
\qquad \lambda\, \mathsf{l}_3 \rightarrow \mathsf{down}\ \mathsf{l}_3 \ggg= \\
\qquad\quad \lambda\, \mathsf{l}_4 \rightarrow \mathsf{Just}\ (\mathsf{update}\ (+\ 1)\ \mathsf{l}_4)
\end{array}
$$

Sequencing:

$$
\begin{array}{l}
(\ggg=) :: \mathsf{Maybe}\ \mathsf{a} \rightarrow (\mathsf{a} \rightarrow \mathsf{Maybe}\ \mathsf{b}) \rightarrow \mathsf{Maybe}\ \mathsf{b} \\
\mathsf{f} \ggg= \mathsf{g} = \mathbf{case}\ \mathsf{f}\ \mathbf{of} \\
\qquad\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad\qquad \mathsf{Just}\ \mathsf{x} \quad \rightarrow \mathsf{g}\ \mathsf{x}
\end{array}
$$

Sequencing and embedding

$$
\begin{array}{l}
\mathsf{up}\ \mathsf{l}_1 \gg= \\
\quad \lambda \mathsf{l}_2 \rightarrow \mathsf{right}\ \mathsf{l}_2 \gg= \\
\qquad\qquad \lambda \mathsf{l}_3 \rightarrow \mathsf{down}\ \mathsf{l}_3 \gg= \\
\qquad\qquad\qquad\qquad \lambda \mathsf{l}_4 \rightarrow \mathsf{Just}\ (\mathsf{update}\ (+\ 1)\ \mathsf{l}_4)
\end{array}
$$

Sequencing and embedding

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

$$
\begin{array}{l}
(\gg\!=) :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\
f \gg\!= g \quad = \mathbf{case}\ f\ \mathbf{of} \\
\qquad\qquad\quad \mathsf{Nothing} \to \mathsf{Nothing} \\
\qquad\qquad\quad \mathsf{Just}\ x \quad\ \to g\ x \\
\mathsf{return} :: a \to \mathsf{Maybe}\ a \\
\mathsf{return}\ x = \mathsf{Just}\ x
\end{array}
$$

Sequencing and embedding

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

$$
\begin{array}{l}
(\gg\!=) :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\
f \gg\!= g \quad = \mathbf{case}\ f\ \mathbf{of} \\
\qquad\qquad\qquad \mathsf{Nothing} \to \mathsf{Nothing} \\
\qquad\qquad\qquad \mathsf{Just}\ x \quad\ \to g\ x \\
\mathsf{return} :: a \to \mathsf{Maybe}\ a \\
\mathsf{return}\ x = \mathsf{Just}\ x
\end{array}
$$

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

Skills Matter

Well-Typed

Observation

Code looks a bit like imperative code. Compare:

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

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

- 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
```

Skills Matter

Well-Typed

A variation: Either

Compare the datatypes

$$
\begin{aligned}
&\textbf{data}\ \text{Either a b} = \text{Left a} \mid \text{Right b} \\
&\textbf{data}\ \text{Maybe a} \ \ = \text{Nothing} \mid \text{Just a}
\end{aligned}
$$

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:

$$
\begin{aligned}
&(\gg\!=) :: \text{Either Error } a \rightarrow (a \rightarrow \text{Either Error } b) \rightarrow \text{Either Error } b \\
&f \gg\!= g = \textbf{case } f \textbf{ of} \\
&\qquad\qquad \text{Left } e \rightarrow \text{Left } e \\
&\qquad\qquad \text{Right } x \rightarrow g\ x \\
&\text{return} :: a \rightarrow \text{Either Error } a \\
&\text{return } x = \text{Right } x
\end{aligned}
$$

The function can now be written as:

$$
\begin{aligned}
&\text{fac} :: \text{Int} \rightarrow \text{Either Error Int} \\
&\text{fac } 0 \qquad\qquad\quad = \text{return } 1 \\
&\text{fac } n \mid n > 0 \qquad = \text{fac } (n - 1) \gg\!= \lambda r \rightarrow \text{return } (n * r) \\
&\qquad\ \ \mid \text{otherwise} = \text{Left } \texttt{"fac: negative argument"}
\end{aligned}
$$

Well-Typed

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
   (Error → 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
```

Example: labelling the leaves of a tree

$$\textbf{data}\ \mathsf{Tree\ a} = \mathsf{Leaf\ a} \mid \mathsf{Node\ (Tree\ a)\ (Tree\ a)}$$

$$\begin{array}{lll}
\mathsf{labelTree} :: \mathsf{Tree\ a} \to \mathsf{State\ Int\ (Tree\ (a, Int))} & & \\
\mathsf{labelTree\ (Leaf\ x)} \quad \mathsf{c} & = & (\mathsf{Leaf}\ (\mathsf{x}, \mathsf{c}), \mathsf{c} + 1) \\
\mathsf{labelTree\ (Node\ l\ r)}\ \mathsf{c}_1 & = & \textbf{let}\ (\mathsf{ll}, \mathsf{c}_2) = \mathsf{labelTree\ l}\ \mathsf{c}_1 \\
 & & \phantom{\textbf{let}\ } (\mathsf{lr}, \mathsf{c}_3) = \mathsf{labelTree\ r}\ \mathsf{c}_2 \\
 & & \textbf{in}\ (\mathsf{Node\ ll\ lr}, \mathsf{c}_3)
\end{array}$$

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

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

Encoding state passing

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

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

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

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

$$
\begin{aligned}
&\text{return} :: a \to \text{State}\ s\ a \\
&\text{return}\ x = \lambda s \to (x, s)
\end{aligned}
$$

Well-Typed

Bind and return for state

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad \mathsf{generateLevel} \qquad \gg\!= \lambda \mathsf{lvl} \rightarrow \\
\lambda s_2 \rightarrow \mathbf{let}\ (\mathsf{lvl}', s_3) = \mathsf{generateStairs\ lvl}\ \ s_2 \\
\qquad\qquad\quad (\mathsf{ms}, s_4) = \mathsf{placeMonsters\ lvl}'\ s_3 \\
\qquad\quad \mathbf{in}\ (\mathsf{combine\ lvl}'\ \mathsf{ms}, s_4)
\end{array}
$$

$$
\begin{array}{l}
(\gg\!=) :: \mathsf{State\ s\ a} \rightarrow (\mathsf{a} \rightarrow \mathsf{State\ s\ b}) \rightarrow \mathsf{State\ s\ b} \\
\mathsf{f} \gg\!= \mathsf{g} \quad = \lambda \mathsf{s} \rightarrow \mathbf{let}\ (\mathsf{x}, \mathsf{s}') = \mathsf{f\ s}\ \mathbf{in}\ \mathsf{g\ x\ s}' \\
\mathsf{return} :: \mathsf{a} \rightarrow \mathsf{State\ s\ a} \\
\mathsf{return\ x} = \lambda \mathsf{s} \rightarrow (\mathsf{x}, \mathsf{s})
\end{array}
$$

Bind and return for state

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

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

Bind and return for state

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

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

Bind and return for state

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

```
(>>=) :: State s a -> (a -> State s b) -> State s b
f >>= g   = \s -> let (x, s') = f s in g x s'

return :: a -> State s a
return x = \s -> (x, s)
```

Observation

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

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

$$
\begin{array}{ll}
\mathsf{lvl} & := \mathsf{generateLevel}; \\
\mathsf{lvl}' & := \mathsf{generateStairs\ lvl}; \\
\mathsf{ms} & := \mathsf{placeMonsters\ lvl}'; \\
\multicolumn{2}{l}{\textbf{return}\ \mathsf{combine\ lvl'\ ms}}
\end{array}
$$

- 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:

$$\mathsf{get} :: \mathsf{State}\ \mathsf{s}\ \mathsf{s}$$
$$\mathsf{get} = \lambda \mathsf{s} \to (\mathsf{s}, \mathsf{s})$$

$$\mathsf{put} :: \mathsf{s} \to \mathsf{State}\ \mathsf{s}\ ()$$
$$\mathsf{put}\ \mathsf{s} = \lambda\_ \to ((), \mathsf{s})$$

$$\mathsf{inc} :: \mathsf{State}\ \mathsf{Int}\ ()$$
$$\mathsf{inc} = \mathsf{get} \gg\!= \lambda \mathsf{s} \to \mathsf{put}\ (\mathsf{s} + 1)$$

Labelling a tree, revisited

$$\textbf{data}\ \mathrm{Tree}\ a = \mathrm{Leaf}\ a \mid \mathrm{Node}\ (\mathrm{Tree}\ a)\ (\mathrm{Tree}\ a)$$

$$\begin{aligned}
&\mathrm{labelTree} :: \mathrm{Tree}\ a \to \mathrm{State}\ \mathrm{Int}\ (\mathrm{Tree}\ (a, \mathrm{Int}))\\
&\mathrm{labelTree}\ (\mathrm{Leaf}\ x)\quad c \ = (\mathrm{Leaf}\ (x, c), c + 1)\\
&\mathrm{labelTree}\ (\mathrm{Node}\ l\ r)\ c_1 = \textbf{let}\ (ll, c_2) = \mathrm{labelTree}\ l\ c_1\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ (lr, c_3) = \mathrm{labelTree}\ r\ c_2\\
&\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ (\mathrm{Node}\ ll\ lr, c_3)
\end{aligned}$$

The old version, with tedious explicit threading of the state.

Labelling a tree, revisited

$$\textbf{data}\ \mathsf{Tree\ a} = \mathsf{Leaf\ a} \mid \mathsf{Node\ (Tree\ a)\ (Tree\ a)}$$

$$\begin{aligned}
&\mathsf{labelTree} :: \mathsf{Tree\ a} \to \mathsf{State\ Int\ (Tree\ (a, Int))} \\
&\mathsf{labelTree\ (Leaf\ x)} \quad = \mathsf{get} \ggg\!= \lambda \mathsf{c} \to \mathsf{inc} \gg \mathsf{return\ (Leaf\ (x, c))} \\
&\mathsf{labelTree\ (Node\ l\ r)} = \mathsf{labelTree\ l} \ggg\!= \lambda \mathsf{ll} \to \\
&\qquad\qquad\qquad\qquad\quad \mathsf{labelTree\ r} \ggg\!= \lambda \mathsf{lr} \to \\
&\qquad\qquad\qquad\qquad\quad \mathsf{return\ (Node\ ll\ lr)}
\end{aligned}$$

New version, with implicit state passing, yet explicit sequencing.

$$\begin{aligned}
&(\gg) :: \mathsf{State\ s\ a} \to \mathsf{State\ s\ b} \to \mathsf{State\ s\ b} \\
&\mathsf{x} \gg \mathsf{y} = \mathsf{x} \ggg\!= \lambda_ \to \mathsf{y}
\end{aligned}$$

(The same definition as for IO . . .)

List

Encoding multiple results and nondeterminism

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

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

Encoding multiple results and nondeterminism

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

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

What is a notion of embedding and sequencing for computations with many results (nondeterministic computations)?

Encoding multiple results and nondeterminism

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

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

What is a notion of embedding and sequencing for computations with many results (nondeterministic computations)?

- Embedding a normal computation into a nondeterminstic one can work by saying the computation has exactly one result.
- Sequencing operations can work by performing the second computation on all possible results of the first one.

Defining bind and return for lists

$$(\gg\!=) :: [a] \to (a \to [b]) \to [b]$$
$$xs \gg\!= f = \text{concat}\ (\text{map}\ f\ xs)$$

$$\text{return} :: a \to [a]$$
$$\text{return}\ x = [x]$$

Note that we have to use concat in $(\gg\!=)$ to flatten the list of lists.

Using bind and return for lists

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

$$
\begin{array}{ll}
\text{txts} & \gg\!= \lambda \text{t} \rightarrow \\
\text{lines t} & \gg\!= \lambda \text{l} \rightarrow \\
\text{words l} & \gg\!= \lambda \text{w} \rightarrow \\
\multicolumn{2}{l}{\text{return (length w)}}
\end{array}
$$

Using bind and return for lists

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

$$
\begin{array}{ll}
\text{txts} & \gg\!= \lambda \text{t} \rightarrow \\
\text{lines t} & \gg\!= \lambda \text{l} \rightarrow \\
\text{words l} & \gg\!= \lambda \text{w} \rightarrow \\
\multicolumn{2}{l}{\text{return (length w)}}
\end{array}
$$

$$
\begin{array}{ll}
\text{t} & := \text{txts} \\
\text{l} & := \text{lines t} \\
\text{w} & := \text{words w} \\
\multicolumn{2}{l}{\textbf{return}\ \text{length w}}
\end{array}
$$

Using bind and return for lists

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

$$\begin{array}{ll} \text{txts} & \gg\!= \lambda \text{t} \rightarrow \\ \text{lines t} & \gg\!= \lambda \text{l} \rightarrow \\ \text{words l} & \gg\!= \lambda \text{w} \rightarrow \\ \multicolumn{2}{l}{\text{return (length w)}} \end{array}$$

$$\begin{array}{ll} \text{t} & := \text{txts} \\ \text{l} & := \text{lines t} \\ \text{w} & := \text{words w} \\ \multicolumn{2}{l}{\textbf{return} \text{ length w}} \end{array}$$

- 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 $(\gg\!=)$.

Intermediate Summary

At least four types with $(\gg=)$ and return:

- for Maybe, $(\gg=)$ sequences operations that may fail and shortcuts evaluation once failure occurs; return embeds a function that never fails;
- for State, $(\gg=)$ sequences operations that may modify some state and threads the state through the operations; return embeds a function that never modifies the state;
- for [], $(\gg=)$ 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.
- for IO, $(\gg=)$ sequences the side effects to the outside world, and return embeds a function without any side effects.

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

$$\textbf{instance}\ \text{Monad Maybe}\ \textbf{where}$$
$$\quad \ldots$$
$$\textbf{instance}\ (\text{Error e}) \Rightarrow \text{Monad (Either e)}\ \textbf{where}$$
$$\quad \ldots$$
$$\textbf{instance}\ \text{Monad []}\ \textbf{where}$$
$$\quad \ldots$$
$$\textbf{newtype}\ \text{State s a} = \text{State}\ \{\text{runState} :: \text{s} \rightarrow (\text{a}, \text{s})\}$$
$$\textbf{instance}\ \text{Monad (State s)}\ \textbf{where}$$
$$\quad \ldots$$

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
    ...
```

The **newtype** for State is required because Haskell does not allow us to directly make a type $s \to (a, s)$ an instance of Monad. (Question: why not?)

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

In fact, we have already seen one more! Which one?

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

In fact, we have already seen one more! Which one?

The generators `Gen` from QuickCheck form a monad. You can see it as an abstract state monad, allowing access to the state of a random number generator.

Additional monad operations

Class Monad contains two additional methods, but with default methods:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Monad}\ \mathsf{m}\ \textbf{where} \\
\quad \ldots \\
\quad (\gg) :: \mathsf{m\ a} \to \mathsf{m\ b} \to \mathsf{m\ b} \\
\quad \mathsf{m} \gg \mathsf{n} = \mathsf{m} \gg\!= \lambda\_ \to \mathsf{n} \\
\quad \mathsf{fail} :: \mathsf{String} \to \mathsf{m\ a} \\
\quad \mathsf{fail\ s} = \mathsf{error\ s}
\end{array}
$$

While the presence of $(\gg)$ can be justified for efficiency reasons, the presence of fail is often considered to be a design mistake.

do notation

The **do** notation we have introduced when discussing IO is available for all monads:

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

$$
\begin{array}{l}
\mathbf{do} \\
\quad \mathsf{lvl} \;\; \leftarrow \mathsf{generateLevel} \\
\quad \mathsf{lvl}' \;\, \leftarrow \mathsf{generateStairs\ lvl} \\
\quad \mathsf{ms} \leftarrow \mathsf{placeMonsters\ lvl}' \\
\quad \mathsf{return\ (combine\ lvl'\ ms)}
\end{array}
$$

do notation – contd.

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

$$
\begin{array}{l}
\textbf{do} \\
\quad l_2 \leftarrow \mathsf{up}\ l_1 \\
\quad l_3 \leftarrow \mathsf{right}\ l_2 \\
\quad l_4 \leftarrow \mathsf{down}\ l_3 \\
\quad \mathsf{return}\ (\mathsf{update}\ (+\ 1)\ l_4)
\end{array}
$$

Tree labelling, revisited once more

Using Control.Monad.State and **do** notation:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a → State Int (Tree (a, Int))
labelTree (Leaf x)   = do
  c ← get
  put (c + 1)   -- or  modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = do
  ll ← labelTree l
  lr ← labelTree r
  return (Node ll lr)
```

How to get at the final tree?

Running a stateful computation

```
evalState :: State s a → s → a
```

Running a stateful computation

$$\mathsf{evalState} :: \mathsf{State}\ \mathsf{s}\ \mathsf{a} \to \mathsf{s} \to \mathsf{a}$$

$$\mathsf{labelTreeFrom0} :: \mathsf{Tree}\ \mathsf{a} \to \mathsf{Tree}\ (\mathsf{a}, \mathsf{Int})$$
$$\mathsf{labelTreeFrom0}\ \mathsf{t} = \mathsf{evalState}\ (\mathsf{labelTree}\ \mathsf{t})\ 0$$

Running a stateful computation

```
evalState :: State s a → s → a
```

```
labelTreeFrom0 :: Tree a → Tree (a, Int)
labelTreeFrom0 t = evalState (labelTree t) 0
```

There's also

```
runState :: State s a → s → (a, s)
```

(which is just unpacking `State`'s `newtype` wrapper).

List comprehensions

For list computations

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

we can use **do** notation

```
do
  t ← txts
  l ← lines t
  w ← words l
  return (length w)
```

but also list comprehensions:

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

More on `do` notation (and list comprehensions)

- Use it, the special syntax is usually more concise.
- Never forget that it is just syntactic sugar. Use $(\gg\!=)$ and $(\gg)$ directly when it is more convenient.

And some things I've already said about `IO`:

- 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 (and list comprehensions)

- Use it, the special syntax is usually more concise.
- Never forget that it is just syntactic sugar. Use $(\ggg\!=)$ and $(\gg)$ directly when it is more convenient.

And some things I've already said about `IO`:

- 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 vs. other monads

The IO monad is special

- IO is a primitive type, and $(\gg\!=)$ and return for IO are primitive functions,
- there is no (politically correct) function $\mathrm{runIO} :: \mathrm{IO}\ a \rightarrow a$, whereas for most other monads there is a corresponding function, or at least some way to get an a out of the monad;
- values of IO a denote side-effecting programs that can be executed by the run-time system.

Effectful programming

- IO being special has little to do with it being a monad;
- you can use IO an functions on IO very much ignoring the presence of the Monad class;
- IO is about allowing real side effects to occur; the other types we have seen are entirely pure as far as Haskell is concerned, even though the capture a form of effects.

IO, internally

If you ask GHCi about IO by saying `:i IO`, you get

$$\textbf{newtype}\ \mathsf{IO}\ a$$
$$= \text{GHC.Types.IO}\ (\text{GHC.Prim.State}\#\ \text{GHC.Prim.RealWorld}$$
$$\to (\#\ \text{GHC.Prim.State}\#\ \text{GHC.Prim.RealWorld}, a\ \#))$$
$$\text{-- Defined in ` GHC.Types '}$$

So internally, GHC models IO as a kind of state monad having the “real world” as state!

Monadic operations

The advantages of an abstract interface

There are several advantages to identifying the “monad” interface:

- We have to learn fewer names. We can use the same `return` and $(\gg=)$ (and **do** notation) in many different situations.
- There are all sorts of useful derived functions that only use `return` and $(\gg=)$. All these library functions become automatically available for every monad now.

The advantages of an abstract interface

There are several advantages to identifying the “monad” interface:

- We have to learn fewer names. We can use the same return and $(\gg=)$ (and **do** notation) in many different situations.
- There are all sorts of useful derived functions that only use return and $(\gg=)$. All these library functions become automatically available for every monad now.
- There are many more monads than the ones we’ve discusses so far. Monads can be combined to form new monads.
- Application-specific code often uses just the monadic interface plus a few extra functions. As such, it is easy to switch the underlying monad of a large part of a program in order to accommodate a new aspect (error handling, logging, backtracking, . . .).

Useful monad operations

```
liftM       :: (a → b) → IO a → IO b
mapM        :: (a → IO b) → [a] → IO [b]
mapM_       :: (a → IO b) → [a] → IO ()
forM        :: [a] → (a → IO b) → IO [b]
forM_       :: [a] → (a → IO b) → IO ()
sequence    :: [IO a] → IO [a]
sequence_   :: [IO a] → IO ()
forever     :: IO a → IO b
filterM     :: (a → IO Bool) → [a] → IO [a]
replicateM  :: Int → IO a → IO [a]
replicateM_ :: Int → IO a → IO ()
when        :: Bool → IO () → IO ()
unless      :: Bool → IO () → IO ()
```

We had discussed these functions in the context of IO.

Useful monad operations

```
liftM       :: Monad m ⇒ (a → b) → m a → m b
mapM        :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_       :: Monad m ⇒ (a → m b) → [a] → m ()
forM        :: Monad m ⇒ [a] → (a → m b) → m [b]
forM_       :: Monad m ⇒ [a] → (a → m b) → m ()
sequence    :: Monad m ⇒ [m a] → m [a]
sequence_   :: Monad m ⇒ [m a] → m ()
forever     :: Monad m ⇒ a → m b
filterM     :: Monad m ⇒ (a → m Bool) → [a] → m [a]
replicateM  :: Monad m ⇒ Int → m a → m [a]
replicateM_ :: Monad m ⇒ Int → m a → m ()
when        :: Monad m ⇒ Bool → m () → m ()
unless      :: Monad m ⇒ Bool → m () → m ()
```

They're actually all overloaded! Try to infer what each of these mean for `Maybe`, `State` and `[]`.

Example: labelling a rose tree

```
data Rose a = Fork a [Rose a]
```

Each node has a (possibly empty) list of subtrees.

Example: labelling a rose tree

```
data Rose a = Fork a [Rose a]
```

Each node has a (possibly empty) list of subtrees.

```
labelRose :: Rose a → State Int (Rose (a, Int))
labelRose (Fork x cs) = do
  c ← get
  put (c + 1)
  lcs ← mapM labelRose cs
  return (Fork (x, c) lcs)
```

Questions

What do you think these will evaluate to:

```
replicateM 2 [1..3]
mapM return [1..3]
sequence [[1,2],[3,4],[5,6]]
mapM (flip lookup [(1,'x'),(2,'y'),(3,'z')]) [1..3]
mapM (flip lookup [(1,'x'),(2,'y'),(3,'z')]) [1,4,3]
evalState (replicateM_ 5 (modify (+2)) >> get) 0
```

About liftM and fmap

$$\begin{aligned}
&\text{liftM} :: (\text{Monad } m) \Rightarrow (a \to b) \to m\ a \to m\ b \\
&\text{fmap} :: (\text{Functor } f) \Rightarrow (a \to b) \to f\ a \to f\ b
\end{aligned}$$

- Nearly same type as fmap, but a different class constraint.
- For historic reasons, Functor is not a superclass of Monad in Haskell.
- But every monad can be made an instance of Functor, by defining fmap to be liftM.
- In practice, nearly all Haskell monads provide a Functor instance. So you usually have liftM, fmap and (<$>) available, all doing the same.

A common pattern

Let's once again look at tree labelling:

```
labelTree :: Tree a → State Int (Tree (a, Int))
labelTree (Leaf x)   = do
  c ← get
  put (c + 1)   -- or  modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = do
  ll ← labelTree l
  lr ← labelTree r
  return (Node ll lr)
```

We are returning an application of (constructor) function Node to the results of monadic computations.

A common pattern (contd.)

$$
\begin{array}{l}
\textbf{do} \\
\quad r_1 \leftarrow comp_1 \\
\quad r_2 \leftarrow comp_2 \\
\quad \ldots \\
\quad r_n \leftarrow comp_n \\
\quad return\ (f\ r_1\ r_2 \ldots r_n)
\end{array}
$$

A common pattern (contd.)

$$
\begin{array}{l}
\textbf{do} \\
\quad r_1 \leftarrow \mathrm{comp}_1 \\
\quad r_2 \leftarrow \mathrm{comp}_2 \\
\quad \ldots \\
\quad r_n \leftarrow \mathrm{comp}_n \\
\quad \mathrm{return}\ (f\ r_1\ r_2 \ldots r_n)
\end{array}
$$

This isn't type correct:

$$
f\ \mathrm{comp}_1\ \mathrm{comp}_2 \ldots \mathrm{comp}_n
$$

A common pattern (contd.)

$$
\begin{array}{l}
\textbf{do} \\
\quad r_1 \leftarrow \mathrm{comp}_1 \\
\quad r_2 \leftarrow \mathrm{comp}_2 \\
\quad \ldots \\
\quad r_n \leftarrow \mathrm{comp}_n \\
\quad \mathrm{return}\ (f\ r_1\ r_2 \ldots r_n)
\end{array}
$$

This isn't type correct:

$$f\ \mathrm{comp}_1\ \mathrm{comp}_2 \ldots \mathrm{comp}_n$$

But we can get close:

$$f <\$> \mathrm{comp}_1 <*> \mathrm{comp}_2 \ldots <*> \mathrm{comp}_n$$

Monadic application

We need a function that's like function application, but works on monadic values:

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = do
  f ← mf
  x ← mx
  return (f x)
```

Monadic application

We need a function that's like function application, but works on monadic values:

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = do
  f ← mf
  x ← mx
  return (f x)
```

Types supporting `return` and `ap` have their own name:

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a                    -- like return
  (<*>) :: f (a → b) → f a → f b     -- like ap
```

Every monad can be made into an applicative functor using the obvious **instance** definition.

Example

```
labelTree :: Tree a → State Int (Tree (a, Int))
labelTree (Leaf x)   = do
  c ← get
  put (c + 1)   -- or  modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = Node <$> labelTree l <*> labelTree r
```

Exercise: Convince yourself that this is type correct.

Lessons

- The abstraction of monads is useful for a multitude of different types.
- Monads can be seen as tagging computations with effects.
- While `IO` is impure and cannot be defined in Haskell, the other effects we have seen can be modelled in a pure way:
  - exceptions via `Maybe` or `Either`;
  - state via `State`;
  - nondeterminism via `[]`.
- The monad interface offers a large number of useful abstractions that can all be applied to these different scenarios.
- All monads are also applicative functors and in particular functors. The `(<$>)` and `(<*>)` operations are also useful for structuring effectful code in Haskell.