Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Nov 19, 2012

4. Monads and monad transformers

Intro: some example monads

To warm up a bit, we discuss and partially recall some interesting examples of monadic structures.

4.1 Maybe

The Maybe type

$$\textbf{data}\ \text{Maybe a} = \text{Nothing} \mid \text{Just a}$$

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

$$\begin{aligned}\text{lookup} &:: (\text{Eq a}) \Rightarrow \text{a} \to [(\text{a}, \text{b})] \to \text{Maybe b}\\ \text{find} &:: (\text{a} \to \text{Bool}) \to [\text{a}] \to \text{Maybe a}\end{aligned}$$

Encoding exceptions using Maybe

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

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

Given a location l_1, we want to move up, right, down, and update the resulting position with using update $(+1)$ ...

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
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Encoding exceptions using Maybe (contd.)

```
up l₁ >>=
  λ l₂     → right l₂ >>=
               λ 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 → right l₂ >>=
    λ l₃ → down l₃ >>=
      λ 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

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

Sequencing and embedding

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

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

$$\begin{array}{l} \text{return} :: \text{a} \to \text{Maybe a} \\ \text{return x} = \text{Just x} \end{array}$$

Sequencing and embedding

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

```
(>>=) :: 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
```

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

Observation

Code looks a bit like imperative code. Compare:

$$\begin{aligned}&\mathsf{up}\ \mathsf{l}_1 \gg= \lambda \mathsf{l}_2 \rightarrow\\&\mathsf{right}\ \mathsf{l}_2 \gg= \lambda \mathsf{l}_3 \rightarrow\\&\mathsf{down}\ \mathsf{l}_3 \gg= \lambda \mathsf{l}_4 \rightarrow\\&\mathsf{return}\ (\mathsf{update}\ (+1)\ \mathsf{l}_4)\end{aligned}$$

$$\begin{aligned}&\mathsf{l}_2 := \mathsf{up}\ \mathsf{l}_1;\\&\mathsf{l}_3 := \mathsf{right}\ \mathsf{l}_2;\\&\mathsf{l}_4 := \mathsf{down}\ \mathsf{l}_3;\\&\mathbf{return}\ \mathsf{update}\ \mathsf{l}_4\end{aligned}$$

- 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.

4.2 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 computes in addition to modifying the state, we must return a tuple (or a special-purpose datatype with multiple fields).

This motivates the following type synonym 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 (or a game) using a user-defined datatype

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

Encoding state passing

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

Encoding state passing

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

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Encoding state passing

```
λs₁ → let (lvl  , s₂) = generateLevel       s₁
          (lvl′ , s₃) = generateStairs lvl  s₂
          (ms   , s₄) = placeMonsters lvl′  s₃
      in (combine lvl′ ms , s₄)
```

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

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

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

Bind and return for state

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{generateLevel} \gg\!= \lambda \mathsf{lvl} \to \\
\lambda \mathsf{s}_2 \to \mathbf{let}\ (\mathsf{lvl}', \mathsf{s}_3) = \mathsf{generateStairs\ lvl}\ \mathsf{s}_2 \\
\qquad\qquad\quad (\mathsf{ms}, \mathsf{s}_4) = \mathsf{placeMonsters\ lvl}'\ \mathsf{s}_3 \\
\qquad\quad \mathbf{in}\ (\mathsf{combine\ lvl}'\ \mathsf{ms}, \mathsf{s}_4)
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{return} :: \mathsf{a} \to \mathsf{State\ s\ a} \\
\mathsf{return\ x} = \lambda \mathsf{s} \to (\mathsf{x}, \mathsf{s})
\end{array}
$$

Bind and return for state

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \mathsf{generateLevel} \qquad\quad \gg\!= \lambda \mathsf{lvl} \rightarrow \\
\qquad\qquad\qquad\qquad \mathsf{generateStairs\ lvl} \quad \gg\!= \lambda \mathsf{lvl}' \rightarrow \\
\lambda \mathsf{s}_3 \rightarrow \mathbf{let}\ (\mathsf{ms}\ ,\ \mathsf{s}_4) = \mathsf{placeMonsters\ lvl}'\ \mathsf{s}_3 \\
\qquad\quad \mathbf{in}\ (\mathsf{combine\ lvl}'\ \mathsf{ms}, \mathsf{s}_4)
\end{array}
$$

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

Bind and return for state

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

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

$$
\begin{array}{l}
\text{return} :: \text{a} \rightarrow \text{State s a} \\
\text{return x} = \lambda \text{s} \rightarrow (\text{x}, \text{s})
\end{array}
$$

Bind and return for state

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

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

Observation

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

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

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

- 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:

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

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

4.3 List

Encoding multiple results and nondeterminism

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

$$\text{map length (concat (map words (concat (map lines txts))))}$$

Easier to understand with a list comprehension:

$$[\text{length } w \mid t \leftarrow \text{txts}, l \leftarrow \text{lines } t, w \leftarrow \text{words } l]$$

We can also define sequencing and embedding, i.e., $(\ggg=)$ and return:

$$(\ggg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$$
$$xs \ggg= f = \text{concat (map } f \; xs)$$
$$\text{return} :: a \rightarrow [a]$$
$$\text{return } x = [x]$$

Using bind and return for lists

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

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

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

- 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 three types with $(\gg=)$ and return:

- for Maybe, $(\gg=)$ 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, $(\gg=)$ 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.

There is a common interface here!

4.4 The Monad class

Monad class

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

- 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 Monad [] where
    ...
newtype State s a = State { runState :: s → (a, s) }
instance Monad (State s) where
    ...
```

Excursion: type constructors

- The class Monad ranges not over ordinary types, but over type constructors, i.e., parameterized types.
- Such classes are also called constructor classes.
- There are types of types, called kinds.

Excursion: type constructors

- The class Monad ranges not over ordinary types, but over type constructors, i.e., parameterized types.
- Such classes are also called constructor classes.
- There are types of types, called kinds.
- Types of kind $*$ are inhabited by values. Examples: Bool, Int, Char.
- Types of kind $* \rightarrow *$ have one parameter of kind $*$. The Monad class ranges over such types. Examples: $[\,]$, Maybe.
- Applying a type constructor of kind $* \rightarrow *$ to a type of kind $*$ yields a type of kind $*$. Examples: $[\mathsf{Int}]$, Maybe Char.

Excursion: type constructors

- The class Monad ranges not over ordinary types, but over type constructors, i.e., parameterized types.
- Such classes are also called constructor classes.
- There are types of types, called kinds.
- Types of kind $*$ are inhabited by values. Examples: Bool, Int, Char.
- Types of kind $* \to *$ have one parameter of kind $*$. The Monad class ranges over such types. Examples: $[\,]$, Maybe.
- Applying a type constructor of kind $* \to *$ to a type of kind $*$ yields a type of kind $*$. Examples: $[\text{Int}]$, Maybe Char.
- The kind of State is $* \to * \to *$. For any type s, State s is of kind $* \to *$ and can thus be an instance of class Monad.

Monad laws

- Every instance of the monad class should have the following properties:
- return is the unit of $(\gg=)$

$$\begin{aligned} \text{return a} \gg= \text{f} &\equiv \text{f a} \\ \text{m} \gg= \text{return} &\equiv \text{m} \end{aligned}$$

- associativity of $(\gg=)$

$$(\text{m} \gg= \text{f}) \gg= \text{g} \equiv \text{m} \gg= (\lambda \text{x} \rightarrow \text{f x} \gg= \text{g})$$

Monad laws for Maybe

```
   return a >>= f
≡    { Definition of (>>=) }
   case return a of
     Nothing → Nothing
     Just x  → f x
≡    { Definition of return }
   case Just a of
     Nothing → Nothing
     Just x  → f x
≡    { case }
   f a
```

Monad laws for Maybe (contd.)

$$
\begin{aligned}
& m \gg\!= \text{return} \\
\equiv\ & \quad \{ \text{ Definition of } (\gg\!=) \ \} \\
& \textbf{case } m \textbf{ of} \\
& \quad \text{Nothing} \to \text{Nothing} \\
& \quad \text{Just } x \quad \to \text{return } x \\
\equiv\ & \quad \{ \text{ Definition of return } \} \\
& \textbf{case } m \textbf{ of} \\
& \quad \text{Nothing} \to \text{Nothing} \\
& \quad \text{Just } x \quad \to \text{Just } x \\
\equiv\ & \quad \{ \textbf{ case } \} \\
& m
\end{aligned}
$$

Monad laws for Maybe (contd.)

Lemma

$\forall (f :: a \to \text{Maybe } b). \text{Nothing} \gg\!= f \equiv \text{Nothing}$

Proof

$$
\begin{aligned}
& \text{Nothing} \gg\!= f \\
\equiv & \quad \{ \text{ Definition of } (\gg\!=) \text{ } \} \\
& \textbf{case } \text{Nothing } \textbf{of} \\
& \quad \text{Nothing} \to \text{Nothing} \\
& \quad \text{Just } x \quad \to f\ x \\
\equiv & \quad \{ \textbf{ case } \} \\
& \text{Nothing}
\end{aligned}
$$

Monad laws for Maybe (contd.)

$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$$

Case distinction on m. Case m is Nothing:

$$
\begin{aligned}
& (\text{Nothing} \gg\!= f) \gg\!= g \\
\equiv \quad & \{\ \text{Lemma}\ \} \\
& \text{Nothing} \gg\!= g \\
\equiv \quad & \{\ \text{Lemma}\ \} \\
& \text{Nothing} \\
\equiv \quad & \{\ \text{Lemma}\ \} \\
& \text{Nothing} \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)
\end{aligned}
$$

Monad laws for Maybe (contd.)

```
  (Just y >>= f) >>= g
≡   { Definition of (>>=) }
  (case Just y of
     Nothing → Nothing
     Just x  → f x) >>= g
≡   { case }
  f y >>= g
≡   { beta-expansion }
  (λx → f x >>= g) y
≡   { case }
  case Just y of
    Nothing → Nothing
    Just x  → (λx → f x >>= g) x
≡   { definition of (>>=) }
  Just y >>= (λx → f x >>= g)
```

Additional monad operations

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

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

While the presence of $(\gg)$ can be justified for efficiency reasons, fail should really be in a different class.

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:

$$
\begin{aligned}
&\mathbf{do}\ \{e\} &&\equiv e \\
&\mathbf{do}\ \{e; stmts\} &&\equiv e \gg \mathbf{do}\ \{stmts\} \\
&\mathbf{do}\ \{p \leftarrow e; stmts\} &&\equiv \mathbf{let}\ ok\ p = \mathbf{do}\ \{stmts\} \\
& && \qquad\quad ok\ \_ = fail\ \texttt{"error"} \\
& && \quad\ \mathbf{in}\ \ e \gg\!= ok \\
&\mathbf{do}\ \{\mathbf{let}\ decls; stmts\} &&\equiv \mathbf{let}\ decls\ \mathbf{in}\ \mathbf{do}\ \{stmts\}
\end{aligned}
$$

Monadic application

$$
\begin{aligned}
&\mathsf{ap} :: (\mathsf{Monad}\ \mathsf{m}) \Rightarrow \mathsf{m}\ (\mathsf{a} \rightarrow \mathsf{b}) \rightarrow \mathsf{m}\ \mathsf{a} \rightarrow \mathsf{m}\ \mathsf{b} \\
&\mathsf{ap}\ \mathsf{f}\ \mathsf{x} = \mathbf{do} \\
&\qquad\qquad \mathsf{f}' \leftarrow \mathsf{f} \\
&\qquad\qquad \mathsf{x}' \leftarrow \mathsf{x} \\
&\qquad\qquad \mathsf{return}\ (\mathsf{f}'\ \mathsf{x}')
\end{aligned}
$$

Without **do** notation:

$$
\begin{aligned}
&\mathsf{ap}\ \mathsf{f}\ \mathsf{x} = \mathsf{f} \gg\!= \lambda \mathsf{f}' \rightarrow \\
&\qquad\quad \mathsf{x} \gg\!= \lambda \mathsf{x}' \rightarrow \\
&\qquad\quad \mathsf{return}\ (\mathsf{f}'\ \mathsf{x}')
\end{aligned}
$$

More on do notation

- Use it, it is usually more concise.
- Never forget that it is just syntactic sugar. Use ($\gg=$) and ($\gg$) 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**.

Lifting functions to monads

$$
\begin{aligned}
&\mathsf{liftM} &&:: (\mathsf{Monad}\ m) \Rightarrow (a \to b) &&\to m\ a \to m\ b\\
&\mathsf{liftM2} &&:: (\mathsf{Monad}\ m) \Rightarrow (a \to b \to c) &&\to m\ a \to m\ b \to m\ c\\
&\ldots\\
&\mathsf{liftM}\ \ f\ x &&= \mathsf{return}\ f\ \text{`ap`}\ x\\
&\mathsf{liftM2}\ f\ x\ y &&= \mathsf{return}\ f\ \text{`ap`}\ x\ \text{`ap`}\ y\\
&\ldots
\end{aligned}
$$

Question

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

Universiteit Utrecht
[Faculty of **Science**
Information and Computing Sciences]

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.

Excursion: functors

Structures that allow mapping have their own class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor Maybe
instance Functor []
```

- All containers, in particular all trees can be made an instance of functor.
- Every monad is a functor morally (liftM), but not necessarily in Haskell.
- Not all functors are monads.
- Why isn't simply map overloaded?

Monadic map

$$\begin{aligned}
&\mathsf{mapM} &&:: (\mathsf{Monad\ m}) \Rightarrow (\mathsf{a} \to \mathsf{m\ b}) \to [\mathsf{a}] \to \mathsf{m}\ [\mathsf{b}] \\
&\mathsf{mapM_} &&:: (\mathsf{Monad\ m}) \Rightarrow (\mathsf{a} \to \mathsf{m\ b}) \to [\mathsf{a}] \to \mathsf{m}\ ()
\end{aligned}$$

$$\begin{aligned}
&\mathsf{mapM\ f}\ [\,] &&= \mathsf{return}\ [\,] \\
&\mathsf{mapM\ f}\ (\mathsf{x} : \mathsf{xs}) &&= \mathsf{liftM2}\ (:)\ (\mathsf{f\ x})\ (\mathsf{mapM\ f\ xs})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{mapM_\ f}\ [\,] &&= \mathsf{return}\ () \\
&\mathsf{mapM_\ f}\ (\mathsf{x} : \mathsf{xs}) &&= \mathsf{f\ x} \gg \mathsf{mapM_\ f\ xs}
\end{aligned}$$

Question

Why not always use mapM and ignore the result?

Sequencing monadic actions

$$
\begin{aligned}
\text{sequence} &:: (\text{Monad m}) \Rightarrow [\text{m a}] \rightarrow \text{m } [\text{a}] \\
\text{sequence\_} &:: (\text{Monad m}) \Rightarrow [\text{m a}] \rightarrow \text{m } () \\
\text{sequence} &= \text{foldr } (\text{liftM2 } (:)) \ (\text{return } [\,]) \\
\text{sequence\_} &= \text{foldr } (\gg) \ (\text{return } ())
\end{aligned}
$$

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
```

Question

Is this the same as defining the second case using

```
foldM op e (x : xs) = do r ← op e x
                         s ← foldM f r xs
                         return s
```

And why is foldM_ less essential than mapM_ or sequence_?

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!

4.5 IO is a monad

The IO monad

The well-known built-in type constructor IO is another type with actions that need sequencing and ordinary functions that can be embedded.

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 $\rightarrow$ 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.IOBase
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.
- 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.

Next lecture

- Next topic: Monad transformers