COMP4075/G54RFP: Lecture 7

Introduction to Monads

Henrik Nilsson

University of Nottingham, UK

A Blessing and a Curse

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects.

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
- The ***BIG*** problem with ***pure*** functional programming is

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The ***BIG*** problem with ***pure*** functional programming is

  “everything is explicit.”

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The ***BIG*** problem with ***pure*** functional programming is

  "everything is explicit."

  Can add a lot of clutter, make it hard to maintain code

Conundrum

"Shall I be pure or impure?" (Wadler, 1992)

Conundrum

“Shall I be pure or impure?” (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.

Conundrum

"Shall I be pure or impure?" (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.
- Effects (state, exceptions, . . .) can
  - help making code concise
  - facilitate maintenance
  - improve the efficiency.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.
- Adapted by
  - Moggi for structuring denotational semantics
  - Wadler for structuring functional programs

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;
- support ***changes*** to the effect structure with minimal impact on the overall program structure;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;
- support ***changes*** to the effect structure with minimal impact on the overall program structure;
- allow integration into a pure setting of ***real*** effects such as
    - I/O
    - mutable state.

This Lecture

Pragmatic introduction to monads:

- Effectful computations
- Identifying a common pattern
- Monads as a ***design pattern***

Example 1: A Simple Evaluator

$$
\begin{aligned}
\textbf{data}\ \mathit{Exp} = {} & \mathit{Lit}\ \mathit{Integer} \\
\mid {} & \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \\
\mid {} & \mathit{Sub}\ \mathit{Exp}\ \mathit{Exp} \\
\mid {} & \mathit{Mul}\ \mathit{Exp}\ \mathit{Exp} \\
\mid {} & \mathit{Div}\ \mathit{Exp}\ \mathit{Exp}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{eval} :: \mathit{Exp} \to \mathit{Integer} \\
& \mathit{eval}\ (\mathit{Lit}\ n) && = n \\
& \mathit{eval}\ (\mathit{Add}\ e1\ e2) && = \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
& \mathit{eval}\ (\mathit{Sub}\ e1\ e2) && = \mathit{eval}\ e1 - \mathit{eval}\ e2 \\
& \mathit{eval}\ (\mathit{Mul}\ e1\ e2) && = \mathit{eval}\ e1 * \mathit{eval}\ e2 \\
& \mathit{eval}\ (\mathit{Div}\ e1\ e2) && = \mathit{eval}\ e1\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ e2
\end{aligned}
$$

Making the Evaluator Safe (1)

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

$$\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Integer} \\
\mathit{safeEval}\ (\mathit{Lit}\ n) \qquad\quad = \mathit{Just}\ n \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \textbf{case}\ \mathit{safeEval}\ e1\ \textbf{of} \\
\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad \mathit{Just}\ n1\ \to \textbf{case}\ \mathit{safeEval}\ e2\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ n2\ \to \mathit{Just}\ (n1 + n2)
\end{array}$$

Making the Evaluator Safe (2)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Sub}\ e1\ \ e2) = \\
\quad \textbf{case}\ \mathit{safeEval}\ e1\ \textbf{of} \\
\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Just}\ n1\ \ \rightarrow \textbf{case}\ \mathit{safeEval}\ e2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\quad \mathit{Just}\ n2\ \ \rightarrow \mathit{Just}\ (n1 - n2)
\end{array}
$$

Making the Evaluator Safe (3)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Mul}\ \mathit{e1}\ \mathit{e2}) = \\
\quad \mathbf{case}\ \mathit{safeEval}\ \mathit{e1}\ \mathbf{of} \\
\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Just}\ \mathit{n1}\ \rightarrow \mathbf{case}\ \mathit{safeEval}\ \mathit{e2}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{n2}\ \rightarrow \mathit{Just}\ (\mathit{n1} * \mathit{n2})
\end{array}
$$

Making the Evaluator Safe (4)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Div}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \textbf{case}\ \mathit{safeEval}\ \mathit{e1}\ \textbf{of} \\
\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Just}\ \mathit{n1}\ \ \rightarrow \textbf{case}\ \mathit{safeEval}\ \mathit{e2}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{n2}\ \ \rightarrow \\
\qquad\qquad\qquad\qquad\quad \textbf{if}\ \mathit{n2} \equiv 0 \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{Just}\ (\mathit{n1}\ `\mathit{div}`\ \mathit{n2})
\end{array}
$$

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).
- If one evaluation fails, fail overall.

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.

Sequencing Evaluations

$$
\begin{aligned}
&\mathit{evalSeq} :: \mathit{Maybe\ Integer} \\
&\qquad\qquad \to (\mathit{Integer} \to \mathit{Maybe\ Integer}) \\
&\qquad\qquad \to \mathit{Maybe\ Integer} \\
&\mathit{evalSeq}\ \mathit{ma}\ f = \mathbf{case}\ \mathit{ma}\ \mathbf{of} \\
&\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad\qquad\qquad\quad \mathit{Just}\ a \to f\ a
\end{aligned}
$$

Exercise 1: Refactoring *safeEval*

Rewrite *safeEval*, case *Add*, using *evalSeq*:

```
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)

evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Exercise 1: Solution

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \mathit{evalSeq}\ (\mathit{safeEval}\ e1) \\
\qquad\qquad (\lambda n1 \to \mathit{evalSeq}\ (\mathit{safeEval}\ e2) \\
\qquad\qquad\qquad\qquad (\lambda n2 \to \mathit{Just}\ (n1 + n2)))
\end{array}
$$

or

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ \text{`}\mathit{evalSeq}\text{`}\ \lambda n1 \to \\
\quad \mathit{safeEval}\ e2\ \text{`}\mathit{evalSeq}\text{`}\ \lambda n2 \to \\
\quad \mathit{Just}\ (n1 + n2)
\end{array}
$$

Refactored Safe Evaluator (1)

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \rightarrow \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Lit}\ n) = \mathit{Just}\ n \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ `\mathit{evalSeq}`\ \lambda n1 \rightarrow \\
\quad \mathit{safeEval}\ e2\ `\mathit{evalSeq}`\ \lambda n2 \rightarrow \\
\quad \mathit{Just}\ (n1 + n2) \\
\mathit{safeEval}\ (\mathit{Sub}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ `\mathit{evalSeq}`\ \lambda n1 \rightarrow \\
\quad \mathit{safeEval}\ e2\ `\mathit{evalSeq}`\ \lambda n2 \rightarrow \\
\quad \mathit{Just}\ (n1 - n2)
\end{array}
$$

Refactored Safe Evaluator (2)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Mul}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \mathit{safeEval}\ \mathit{e1}\ `\mathit{evalSeq}`\ \lambda \mathit{n1} \rightarrow \\
\quad \mathit{safeEval}\ \mathit{e2}\ `\mathit{evalSeq}`\ \lambda \mathit{n2} \rightarrow \\
\quad \mathit{Just}\ (\mathit{n1} * \mathit{n2}) \\
\mathit{safeEval}\ (\mathit{Div}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \mathit{safeEval}\ \mathit{e1}\ `\mathit{evalSeq}`\ \lambda \mathit{n1} \rightarrow \\
\quad \mathit{safeEval}\ \mathit{e2}\ `\mathit{evalSeq}`\ \lambda \mathit{n2} \rightarrow \\
\quad \mathbf{if}\ \mathit{n2} \equiv 0 \\
\quad \mathbf{then}\ \mathit{Nothing} \\
\quad \mathbf{else}\ \mathit{Just}\ (\mathit{n1}\ `\mathit{div}`\ \mathit{n2})
\end{array}
$$

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.
- Let's generalize and adopt names reflecting our intentions.

`Maybe` Viewed as a Computation (2)

Successful computation of a value:

$$
\begin{aligned}
&mbReturn :: a \to Maybe\ a \\
&mbReturn = Just
\end{aligned}
$$

Sequencing of possibly failing computations:

$$
\begin{aligned}
&mbSeq :: Maybe\ a \to (a \to Maybe\ b) \to Maybe\ b \\
&mbSeq\ ma\ f = \mathbf{case}\ ma\ \mathbf{of} \\
&\qquad\qquad\qquad Nothing \to Nothing \\
&\qquad\qquad\qquad Just\ a \quad \to f\ a
\end{aligned}
$$

`Maybe` Viewed as a Computation (3)

Failing computation:

$$mbFail :: Maybe\ a$$
$$mbFail = Nothing$$

The Safe Evaluator Revisited

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Integer} \\
\mathit{safeEval}\ (\mathit{Lit}\ n) = \mathit{mbReturn}\ n \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ `\mathit{mbSeq}`\ \lambda n1 \to \\
\quad \mathit{safeEval}\ e2\ `\mathit{mbSeq}`\ \lambda n2 \to \\
\quad \mathit{mbReturn}\ (n1 + n2) \\
\ldots \\
\mathit{safeEval}\ (\mathit{Div}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ `\mathit{mbSeq}`\ \lambda n1 \to \\
\quad \mathit{safeEval}\ e2\ `\mathit{mbSeq}`\ \lambda n2 \to \\
\quad \mathbf{if}\ n2 \equiv 0\ \mathbf{then}\ \mathit{mbFail}\ \mathbf{else}\ \mathit{mbReturn}\ (n1\ `\mathit{div}`\ n
\end{array}
$$

Example 2: Numbering Trees

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

$$
\begin{array}{l}
\mathit{numberTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int} \\
\mathit{numberTree}\ t = \mathit{fst}\ (\mathit{ntAux}\ t\ 0) \\
\quad \textbf{where} \\
\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to \mathit{Int} \to (\mathit{Tree}\ \mathit{Int}, \mathit{Int}) \\
\qquad \mathit{ntAux}\ (\mathit{Leaf}\ \_)\ n \qquad\quad = (\mathit{Leaf}\ n, n+1) \\
\qquad \mathit{ntAux}\ (\mathit{Node}\ t1\ t2)\ n = \\
\qquad\quad \textbf{let}\ (t1', n') = \mathit{ntAux}\ t1\ n \\
\qquad\quad \textbf{in let}\ (t2', n'') = \mathit{ntAux}\ t2\ n' \\
\qquad\qquad \textbf{in}\ (\mathit{Node}\ t1'\ t2', n'')
\end{array}
$$

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

$$\mathbf{type}\ S\ a = Int \rightarrow (a, Int)$$

(Only Int state for the sake of simplicity.)

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

$$\mathbf{type}\ S\ a = Int \to (a, Int)$$

(Only Int state for the sake of simplicity.)

- A value (function) of type $S\ a$ can now be viewed as denoting a stateful computation computing a value of type a.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. ***state updating is an effect***, implicitly affecting subsequent computations.
  (As we would expect.)

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \quad = \quad Int \to (a, Int)$):

$$sReturn :: a \to S\ a$$
$$sReturn\ a = ???$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \;\; = \;\; Int \to (a, Int)$):

$$sReturn :: a \to S\ a$$
$$sReturn\ a = \lambda n \to (a, n)$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \;=\; Int \rightarrow (a, Int)$):

$$sReturn :: a \rightarrow S\ a$$
$$sReturn\ a = \lambda n \rightarrow (a, n)$$

Sequencing of stateful computations:

$$sSeq :: S\ a \rightarrow (a \rightarrow S\ b) \rightarrow S\ b$$
$$sSeq\ sa\ f = ???$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \;=\; Int \to (a, Int)$):

$$sReturn :: a \to S\ a$$
$$sReturn\ a = \lambda n \to (a, n)$$

Sequencing of stateful computations:

$$sSeq :: S\ a \to (a \to S\ b) \to S\ b$$
$$\begin{aligned} &sSeq\ sa\ f = \lambda n \to \\ &\quad \mathbf{let}\ (a, n') = sa\ n \\ &\quad \mathbf{in}\ f\ a\ n' \end{aligned}$$

Stateful Computations (4)

Reading and incrementing the state
(For ref.: $S\ a \;\;=\;\; Int \to (a, Int)$):

$$sInc :: S\ Int$$

$$sInc = \lambda n \to (n, n+1)$$

Numbering trees revisited

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

$$\mathit{numberTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}$$

$$\mathit{numberTree}\ t = \mathit{fst}\ (\mathit{ntAux}\ t\ 0)$$

$$\begin{array}{l}
\quad \textbf{where} \\
\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to S\ (\mathit{Tree}\ \mathit{Int}) \\
\qquad \mathit{ntAux}\ (\mathit{Leaf}\ _) = \\
\qquad\quad \mathit{sInc}\ \text{`}\mathit{sSeq}\text{`}\ \lambda n \to \mathit{sReturn}\ (\mathit{Leaf}\ n) \\
\qquad \mathit{ntAux}\ (\mathit{Node}\ t1\ t2) = \\
\qquad\quad \mathit{ntAux}\ t1\ \text{`}\mathit{sSeq}\text{`}\ \lambda t1' \to \\
\qquad\quad \mathit{ntAux}\ t2\ \text{`}\mathit{sSeq}\text{`}\ \lambda t2' \to \\
\qquad\quad \mathit{sReturn}\ (\mathit{Node}\ t1'\ t2')
\end{array}$$

Observations

- The “plumbing” has been captured by the abstractions.

Observations

- The “plumbing” has been captured by the abstractions.
- In particular:
  - counter no longer manipulated directly
  - no longer any risk of “passing on” the wrong version of the counter!

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations
- In fact, both examples are instances of the general notion of a ***MONAD***.

Monads in Functional Programming

A monad is represented by:

- A type constructor

  $$M :: * \to *$$

  $M\ \ T$ represents computations of value of type T.

- A polymorphic function

  $$return :: a \to M\ \ a$$

  for lifting a value to a computation.

- A polymorphic function

  $$(\gg\!=) :: M\ \ a \to (a \to M\ \ b) \to M\ \ b$$

  for sequencing computations.

Exercise 2: *join* and *fmap*

Equivalently, the notion of a monad can be captured through the following functions:

$$\begin{aligned} &\mathit{return} :: a \to M\ a \\ &\mathit{join} \quad :: (M\ (M\ a)) \to M\ a \\ &\mathit{fmap} \quad :: (a \to b) \to M\ a \to M\ b \end{aligned}$$

join "flattens" a computation, fmap "lifts" a function to map computations to computations.

Define join and fmap in terms of $(\gg\!=)$ (and return), and $(\gg\!=)$ in terms of join and fmap.

$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$

Exercise 2: Solution

$$join :: M\ (M\ a) \to M\ a$$
$$join\ mm = mm \gg\!= id$$

$$fmap :: (a \to b) \to M\ a \to M\ b$$
$$fmap\ f\ m = m \gg\!= return \circ f$$

$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$
$$m \gg\!= f = join\ (fmap\ f\ m)$$

Monad laws

Additionally, the following ***laws*** must be satisfied:

$$
\begin{aligned}
return\ x \gg= f &= f\ x \\
m \gg= return &= m \\
(m \gg= f) \gg= g &= m \gg= (\lambda x \rightarrow f\ x \gg= g)
\end{aligned}
$$

I.e., $return$ is the right and left identity for $(\gg=)$, and $(\gg=)$ is associative.

Exercise 3: The Identity Monad

The ***Identity Monad*** can be understood as representing ***effect-free*** computations:

$$\textbf{type}\ I\ a = a$$

1. Provide suitable definitions of $return$ and $(\gg\!=)$.
2. Verify that the monad laws hold for your definitions.

Exercise 3: Solution

$$
\begin{aligned}
&return :: a \to I\ \ a \\
&return = id \\
&(\gg\!=) :: I\ \ a \to (a \to I\ \ b) \to I\ \ b \\
&m \gg\!= f = f\ \ m
\end{aligned}
$$

(Or: $(\gg\!=) = \mathit{flip}\ (\$)$)

Simple calculations verify the laws, e.g.:

$$
\begin{aligned}
return\ x \gg\!= f \quad &= \quad id\ x \gg\!= f \\
&= \quad x \gg\!= f \\
&= \quad f\ \ x
\end{aligned}
$$

Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- *All About Monads.* `http://www.haskell.org/all_about_monads`