COMP4075: Lecture 8

Introduction to Monads

Henrik Nilsson

University of Nottingham, UK

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)

- 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.
- Key idea: ***Computational types***: an object of type MA 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;
- 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

```
data Exp = Lit Integer
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```


Making the Evaluator Safe (1)

```
data Maybe a = Nothing | Just a

safeEval :: Exp -> Maybe Integer
safeEval (Lit n)     = Just n
safeEval (Add e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just n2 -> Just (n1 + n2)
```


Making the Evaluator Safe (2)

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

Making the Evaluator Safe (3)

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

Making the Evaluator Safe (4)

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

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

$$
\begin{array}{l}
evalSeq :: Maybe\ Integer \\
\quad\quad\quad \rightarrow (Integer \rightarrow Maybe\ Integer) \\
\quad\quad\quad \rightarrow Maybe\ Integer \\
evalSeq\ ma\ f = \textbf{case}\ ma\ \textbf{of} \\
\quad\quad\quad\quad\quad Nothing \rightarrow Nothing \\
\quad\quad\quad\quad\quad Just\ a \rightarrow f\ a
\end{array}
$$

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

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

or

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

Refactored Safe Evaluator (1)

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

Refactored Safe Evaluator (2)

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

`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***.
- 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:

$$mbReturn :: a \to Maybe\ a$$
$$mbReturn = Just$$

Sequencing of possibly failing computations:

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

`Maybe` Viewed as a Computation (3)

Failing computation:

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

The Safe Evaluator Revisited

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

Example 2: Numbering Trees

$$\mathbf{data}\ Tree\ a = Leaf\ a \mid Node\ (Tree\ a)\ (Tree\ a)$$
$$numberTree :: Tree\ a \to Tree\ Int$$
$$numberTree\ t = fst\ (ntAux\ t\ 0)$$
$$\quad \mathbf{where}$$
$$\quad\quad ntAux :: Tree\ a \to Int \to (Tree\ Int, Int)$$
$$\quad\quad ntAux\ (Leaf\ \_)\ n \quad = (Leaf\ n, n + 1)$$
$$\quad\quad ntAux\ (Node\ t1\ t2)\ n =$$
$$\quad\quad\quad \mathbf{let}\ (t1', n') = ntAux\ t1\ n$$
$$\quad\quad\quad \mathbf{in\ let}\ (t2', n'') = ntAux\ t2\ n'$$
$$\quad\quad\quad\quad \mathbf{in}\ (Node\ t1'\ t2', n'')$$

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.
- 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.
- 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.: $S\ a \;=\; Int \to (a, Int)$):

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

Sequencing of stateful computations:

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

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\mathit{numberTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}\\
&\mathit{numberTree}\ t = \mathit{fst}\ (\mathit{ntAux}\ t\ 0)\\
&\quad \textbf{where}\\
&\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to S\ (\mathit{Tree}\ \mathit{Int})\\
&\qquad \mathit{ntAux}\ (\mathit{Leaf}\ \_) =\\
&\qquad\quad \mathit{sInc}\ `\mathit{sSeq}`\ \lambda n \to \mathit{sReturn}\ (\mathit{Leaf}\ n)\\
&\qquad \mathit{ntAux}\ (\mathit{Node}\ t1\ t2) =\\
&\qquad\quad \mathit{ntAux}\ t1\ `\mathit{sSeq}`\ \lambda t1' \to\\
&\qquad\quad \mathit{ntAux}\ t2\ `\mathit{sSeq}`\ \lambda t2' \to\\
&\qquad\quad \mathit{sReturn}\ (\mathit{Node}\ t1'\ t2')
\end{aligned}
$$

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.
- 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

  $\mathit{return} :: a \to M\ a$

  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:

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

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

$$
\begin{aligned}
&\mathit{join} :: M\ (M\ a) \to M\ a\\
&\mathit{join}\ \mathit{mm} = \mathit{mm} \gg\!= \mathit{id}\\
&\mathit{fmap} :: (a \to b) \to M\ a \to M\ b\\
&\mathit{fmap}\ f\ m = m \gg\!= \mathit{return} \circ f\\
&(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b\\
&m \gg\!= f = \mathit{join}\ (\mathit{fmap}\ f\ m)
\end{aligned}
$$

Monad laws

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

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

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

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

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

Simple calculations verify the laws, e.g.:

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

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`