A zoology of monads for programming

Daniel Brown
dbrown@ccs.neu.edu

February 15, 2010

Abstract

Monads are an abstraction for programming with *effects*—things like state, exceptions, nondeterminism, and even continuation passing. Although monads are often thought of as a math trick that makes I/O work in Haskell, they are actually just a programming trick for writing simpler and more modular code. In this talk we will look at a variety of effectful programs, identify the redundant parts, and see how refactoring leads us to the structure of a monad.

Time permitting, we will transfer our intuitions about monads to the nearby concept of comonads and briefly see how they give us an abstraction for programming with *context dependency*.

1 Abstracting over failure

Abstraction

Consider:

```
let fin = open "grades" in
if fin = fail then
  fail
else
  let fout = open "new-grades" in
  if fout = fail then
    fail
  else
    let _ = untilFail (\_.
      let s = readline fin in
      if s = fail then
        fail
      else
        let s' = inflate s in
        writeline s' fout) in
    print "done!"
```

```
    open : string -> file ∪ fail
readline : file -> string ∪ fail
writeline : string -> file -> unit ∪ fail

untilFail : (unit -> a ∪ fail) -> list a
  inflate : string -> string
    print : string -> unit
```

Ugly! Redundant! We need a new `let`:

$$\frac{\Gamma \vdash e : s \cup \texttt{fail} \qquad \Gamma, x{:}s \vdash e' : t \cup \texttt{fail}}{\Gamma \vdash \texttt{letfail x = } e \texttt{ in } e' : t \cup \texttt{fail}}$$

$$\texttt{letfail x = } e \texttt{ in } e' = \texttt{let x = } e \texttt{ in if x = fail then fail else } e'$$

```
letfail fin  = open "grades" in
letfail fout = open "new-grades" in
let     _    = untilFail (\_.
                 letfail s  = readline fin in
                 let     s' = inflate s in
                 writeline s' fout) in
print "done!"
```

Nicer, but can we avoid interleaving `letfail` and `let`? Need to mark expressions that can't fail:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{safe } e : t \cup \texttt{fail}} \qquad \texttt{safe } e = e$$

```
letfail fin  = open "grades"
        fout = open "new-grades"
        _    = safe (untilFail (\_.
                 letfail s  = readline fin
                         s' = safe (inflate s)
                 in writeline s' fout))
in safe (print "done!")
```

Much nicer!

Correctness?

But have we changed anything? Not if we can prove:

$$\texttt{letfail x = safe } e \texttt{ in } e' = \texttt{let x = } e \texttt{ in } e'$$

which is equivalent to:

$$\texttt{letfail x = safe x in } e = e \qquad (\beta)$$

Let's calculate:

$$\begin{aligned}
&\texttt{letfail x = safe x in } e \\
=\ &\texttt{let x = safe x in if x = fail then fail else } e \\
=\ &\texttt{if safe x = fail then fail else } e \\
=\ &\texttt{if x = fail then fail else } e \\
=\ &\ldots \text{oops!}
\end{aligned}$$

We've changed the program!

Correctness

It's a good idea but we did it wrong—instead of $- \cup \texttt{fail}$ we want $- + \texttt{fail}$:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{L } e : t + u} \qquad \frac{\Gamma \vdash e : u}{\Gamma \vdash \texttt{R } e : t + u}$$

$$\frac{\Gamma \vdash e : s \texttt{ + fail} \qquad \Gamma, x{:}s \vdash e' : t \texttt{ + fail}}{\Gamma \vdash \texttt{letfail x = } e \texttt{ in } e' : t \texttt{ + fail}} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{safe } e : t \texttt{ + fail}}$$

$$\texttt{letfail x = } e \texttt{ in } e' = \texttt{match } e \texttt{ in L x -> } e' \texttt{; R fail -> R fail}$$

$$\texttt{safe } e' = \texttt{L } e'$$

The law above now holds:

$$\begin{aligned}
&\texttt{letfail x = safe x in } e \\
=\ &\texttt{match safe x in L x -> } e\texttt{; R fail -> R fail} \\
=\ &\texttt{match L x in L x -> } e\texttt{; R fail -> R fail} \\
=\ &e
\end{aligned}$$

And the program refactoring is correct.

2 Abstracting over state passing

Consider:

```
fib : nat -> map nat nat -> nat * map nat nat
fib a m = case
  member a (dom m) -> (lookup a m, m)
  a = 0 or a = 1   -> (a,m)
  true             -> let (b,m')  = fib (a-1) m
                          (c,m'') = fib (a-2) m' in
                      (b+c, insert a (b+c) m'')
```

Though concise, there are many mentions of the maps—and we have to be careful to use the right one!

If we're clever we can eliminate this bookkeeping. Consider the type of `fib`:

$$\texttt{fib} : \texttt{nat} \to \texttt{map nat nat} \to \texttt{nat} \times \texttt{map nat nat}$$

It wants to be a $\texttt{nat} \to \texttt{nat}$ but it must also *transform state*: it relies on an input map and, in case it changes it, produces and output map. The pattern above is sequencing on state transformers: pass the initial state into one and thread its output state into the next— can we come up with a replacement for `let` that will perform this threading for us?

$$\texttt{λs0. let (a,s1) = f s0 \\ (b,s2) = g s1 \\ (c,s3) = h s2 \\ ...} \quad\longrightarrow\quad \texttt{letst a = f \\ b = g \\ c = h \\ ...}$$

$$\frac{\Gamma \vdash e : s \to t \times s \qquad \Gamma, x{:}t \vdash e' : s \to u \times s}{\Gamma \vdash \texttt{letst}_s\ \texttt{x = } e\ \texttt{in } e' : s \to u \times s} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{pure}_s\ \texttt{e} : s \to t \times s}$$

$$\texttt{letst}_s\ \texttt{x = } e\ \texttt{in } e' = \texttt{λs. let (x,s') = } e\ \texttt{s in } e'\ \texttt{s'} \qquad \texttt{pure}_s\ e = \texttt{λs.(}e\texttt{,s)}$$

```
fib : nat -> map nat nat -> nat * map nat nat
fib a =
  letst m = (\m.(m,m)) in case
    member a (dom m) -> pure (lookup a m)
    a = 0 or a = 1   -> pure a
    true             -> letst b = fib (a-1)
                                c = fib (a-2)
                                _ = (\m. (*, insert a (b+c) m)) in
                        pure (b+c)
```

Operators:

$\texttt{get} : s \to s \times s$ $\quad\texttt{get} = \texttt{λs.(s,s)}$

$\texttt{modify} : (s \to s) \to s \to \texttt{unit} \times s$ $\quad\texttt{modify f} = \texttt{λs. (f s, s)}$

$\texttt{put} : s \to s \to \texttt{unit} \times s$ $\quad\texttt{put s'} = \texttt{λs. (*,s')}$

Correctness Yes...

3 Monads embody effects

Our trick is to replace let with some let_M and pure_M for some type constructor M:

$$\frac{\Gamma \vdash e : Ms \qquad \Gamma, x{:}s \vdash e' : Mt}{\Gamma \vdash \text{let}_M\, x = e \text{ in } e' : Mt} \qquad\qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{pure}_M\, e : Mt}$$

As suggested by the law we saw earlier for failure, a replacement for let should satisfy its laws:

$$\begin{array}{lcll}
\text{let}\, x = x \text{ in } e \;=\; e & & \text{let}_M\, x = \text{pure}_M\, x \text{ in } e \;=\; e & (\beta) \\
\text{let}\, x = e \text{ in } x \;=\; e & & \text{let}_M\, x = e \text{ in } \text{pure}_M\, x \;=\; e & (\eta) \\
\text{let}\, y = (\text{let}\, x = e \text{ in } e') \text{ in } e'' \;= & \rightsquigarrow & \text{let}_M\, y = (\text{let}_M\, x = e \text{ in } e') \text{ in } e'' \;= & (\text{assoc}) \\
\quad \text{let}\, x = e \text{ in } (\text{let}\, y = e' \text{ in } e'') & & \quad \text{let}_M\, x = e \text{ in } (\text{let}_M\, y = e' \text{ in } e'') &
\end{array}$$

Such a replacement $(M, \text{let}_M, \text{pure}_M)$ is called a *monad*. Easy calculations show that both replacements we've seen

$$\text{Fail}\, t = t + 1 \qquad \begin{array}{l}\text{let}_{\text{Fail}}\, x = e \text{ in } e' = \\ \quad \text{match } e \text{ in } \text{L}\, x \to e';\ \text{R}\, * \to \text{R}\, *\end{array} \qquad \text{pure}_{\text{Fail}}\, e = \text{L}\, e$$

$$\text{State}_s\, t = s \to t \times s \qquad \begin{array}{l}\text{let}_{\text{State}_s}\, x = e \text{ in } e' = \\ \quad \lambda s.\ \text{let}\, (x, s') = e\ s \text{ in } e'\ s'\end{array} \qquad \text{pure}_{\text{State}_s}\, e = \lambda s.\, (e, s)$$

satisfy all three laws and are thus monads.

A monad $(M, \text{let}_M, \text{pure}_M)$ gives us a way to program with an *implicit effect* managed by let_M:

- When we build a $t + 1$ expression using let_{Fail} and $\text{pure}_{\text{Fail}}$ we are programming with a *failure effect*: any subexpression can have a side effect of failing, in which case the whole expression fails.
- When we build an $s \to t \times s$ expression using $\text{let}_{\text{State}_s}$ and $\text{pure}_{\text{State}_s}$ we are programming with a *state effect*: the whole expression is a state transformer, $\text{let}_{\text{State}_s}$ threads the state through all the subexpressions, and any subexpression can read and update the threaded state.

So what other monads are there and what kinds of effectful programs can we write with them?

4 Zoology

$$\begin{aligned}\mathrm{Fail}\, t &= t + 1\\ \mathrm{pure}_{\mathrm{Fail}}\, e &= \mathrm{L}\, e\\ \mathrm{let}_{\mathrm{Fail}}\, x = e \text{ in } e' &= \mathrm{match}\, e \text{ in } \mathrm{L}\, x \to e';\ \mathrm{R} * \to \mathrm{R} *\end{aligned}$$

$$\begin{aligned}&\mathrm{fail} \ :\ \mathrm{Fail}\, t\\ &\mathrm{fail} = \mathrm{R} *\end{aligned}$$

$$\begin{aligned}\mathrm{Exn}_{err}\, t &= t + err\\ \mathrm{pure}_{\mathrm{Exn}_{err}}\, e &= \mathrm{L}\, e\\ \mathrm{let}_{\mathrm{Exn}_{err}}\, x = e \text{ in } e' &= \mathrm{match}\, e \text{ in } \mathrm{L}\, x \to e';\ \mathrm{R}\, y \to \mathrm{R}\, y\end{aligned}$$

$$\begin{aligned}&\mathrm{throw} \ :\ err \to \mathrm{Exn}_{err}\, t\\ &\mathrm{throw}\, x = \mathrm{R}\, x\\ &\mathrm{catch} \ :\ \mathrm{Exn}_{err}\, t \to (err \to t) \to t\\ &\mathrm{catch}\ (\mathrm{L}\, x)\ f = x\\ &\mathrm{catch}\ (\mathrm{R}\, y)\ f = f\ y\end{aligned}$$

$$\begin{aligned}\mathrm{State}_s\, t &= s \to t \times s\\ \mathrm{pure}_{\mathrm{State}_s}\, e &= \lambda s.\, (e, s)\\ \mathrm{let}_{\mathrm{State}_s}\, x = e \text{ in } e' &= \lambda s.\ \mathrm{let}\, (x, s') = e\ s \text{ in } e'\ s'\end{aligned}$$

$$\begin{aligned}&\mathrm{get} \ :\ \mathrm{State}_s\, s\\ &\mathrm{get} = \lambda s.\, (s, s)\\ &\mathrm{put} \ :\ s \to \mathrm{State}_s\, 1\\ &\mathrm{put}\, s' = \lambda s.\, (*, s')\end{aligned}$$

$$\begin{aligned}\mathrm{Env}_r\, t &= r \to t\\ \mathrm{pure}_{\mathrm{Env}_r}\, e &= \lambda \_.\, e\\ \mathrm{let}_{\mathrm{Env}_r}\, x = e \text{ in } e' &= \lambda r.\ \mathrm{let}\, x = e\ r \text{ in } e'\ r\end{aligned}$$

$$\begin{aligned}&\mathrm{ask} \ :\ \mathrm{Env}_r\, r\\ &\mathrm{ask} = \lambda r.\, r\end{aligned}$$

$$\begin{aligned}\mathrm{Writer}_m\, t &= t \times m\\ \mathrm{pure}_{\mathrm{Writer}_m}\, e &= (e, \mathrm{id})\\ \mathrm{let}_{\mathrm{Writer}_m}\, x = e \text{ in } e' &= \mathrm{let}\, (x, a) = e;\ (y, b) = e' \text{ in } (y, a \oplus b)\end{aligned}$$

$$\begin{aligned}&\mathrm{tell} \ :\ m \to \mathrm{Writer}_m\, 1\\ &\mathrm{tell}\, a = (*, a)\end{aligned}$$

$$\begin{aligned}\mathrm{List}\, t &= 1 + t \times \mathrm{List}\, t\\ \mathrm{pure}_{\mathrm{List}}\, e &= (e, \mathrm{nil})\\ \mathrm{let}_{\mathrm{List}}\, x = e \text{ in } e' &= \mathrm{concat}\ (\mathrm{map}\ (\lambda x.\, e')\ e)\end{aligned}$$

$$\begin{aligned}&\mathrm{fail} \ :\ \mathrm{List}\, t\\ &\mathrm{fail} = \mathrm{nil}\\ &\mathrm{amb} \ :\ \mathrm{List}\, t \to \mathrm{List}\, t \to \mathrm{List}\, t\\ &\mathrm{amb} = \mathrm{append}\end{aligned}$$

$$\begin{aligned}\mathrm{Dist}\, t &= \mathrm{FinMap}\, t\ [0, 1]\\ \mathrm{pure}_{\mathrm{Dist}}\, e &= [e \mapsto 1]\\ \mathrm{let}_{\mathrm{Dist}}\, x = e \text{ in } e' &= \mathrm{concat}\ (\mathrm{map}\ (\lambda x.\, e')\ e)\end{aligned}$$

$$\begin{aligned}&\mathrm{flip} \ :\ [0, 1] \to \mathrm{Dist}\, t \to \mathrm{Dist}\, t \to \mathrm{Dist}\, t\\ &\mathrm{flip}\, p\ \mu\ \nu = p\mu + (1 - p)\nu\end{aligned}$$

$$\begin{aligned}\mathrm{CPS}_r\, t &= (t \to r) \to r\\ \mathrm{pure}_{\mathrm{CPS}_r}\, e &= \lambda k.\, k\ e\\ \mathrm{let}_{\mathrm{CPS}_r}\, x = e \text{ in } e' &= \lambda k.\, e\ (\lambda a.\, e'\ a\ k)\end{aligned}$$

$$\begin{aligned}&\mathrm{callcc} \ :\ ((t \to \mathrm{CPS}_r\, u) \to \mathrm{CPS}_r\, t) \to \mathrm{CPS}_r\, t\\ &\mathrm{callcc}\, f = \lambda k.\, f\ (\lambda a k'.\, k\ a)\ k\\ &\mathrm{abort} \ :\ r \to \mathrm{CPS}_r\, t\\ &\mathrm{abort}\, r = \lambda k.\, r\end{aligned}$$

5 Slogans and intuitions

Slogans

A monad is a type you can transform just by constructing, $(X \to MY) \to MX \to MY$, and there's always a trivial construction $X \to MX$.

A comonad is a type you can transform just by observing, $(WX \to Y) \to WX \to WY$, and there's always a trivial observation $WX \to X$.

effects	monads	*construction*	$X \to MY$
context dependency	comonads	*observation*	$WX \to Y$

Definitions Three equivalent definitions (plus laws):

$$\frac{\Gamma \vdash e : Ms \qquad \Gamma, x{:}s \vdash e' : Mt}{\Gamma \vdash \mathrm{let}_M\, x = e \,\mathrm{in}\, e' : Mt} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathrm{pure}_M\, e : Mt} \tag{1}$$

$$\frac{\Gamma \vdash e : s \to Mt}{\Gamma \vdash \mathrm{extend}_M\, e : Ms \to Mt} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathrm{pure}_M\, e : Mt} \qquad \text{HO—(2)}$$

$$\frac{\Gamma \vdash e : s \to t}{\Gamma \vdash \mathrm{map}_M\, e : Ms \to Mt} \quad \frac{\Gamma \vdash e : M(Mt)}{\Gamma \vdash \mathrm{join}_M\, e : Mt} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathrm{pure}_M\, e : Mt} \tag{3}$$

Or, in a non-strict higher-order setting:

$$\begin{aligned} \mathrm{let} &: MX \to (X \to MY) \to MY \\ \mathrm{pure} &: X \to MX \end{aligned}$$

$$\begin{aligned} \mathrm{extend} &: (X \to MY) \to MX \to MY \\ \mathrm{pure} &: X \to MX \end{aligned}$$

$$\begin{aligned} \mathrm{map} &: (X \to Y) \to MX \to MY \\ \mathrm{join} &: M(MX) \to MX \\ \mathrm{pure} &: X \to MX \end{aligned}$$

6 Comonads

$$\begin{aligned}
\mathrm{Env}_r\, t &= r \times t \\
\mathrm{pure}_{\mathrm{Env}_r}\ (r, e) &= e \\
\mathrm{extend}_{\mathrm{Env}_r}\ f &= \lambda(r, x).\ (r, f\ (r, x))
\end{aligned}$$

$$\begin{aligned}
\mathrm{ask} &: \mathrm{Env}_r\, t \to r \\
\mathrm{ask}\ (r, x) &= r
\end{aligned}$$

$$\begin{aligned}
\mathrm{Line}\, t &= \mathrm{Stream}\, t \times t \times \mathrm{Stream}\, t \\
\mathrm{pure}_{\mathrm{Line}}\ (l, e, r) &= e \\
\mathrm{extend}_{\mathrm{Line}}\ f &= \lambda x.\ (\mathrm{map}\ f\ (\mathrm{iterate}\, \mathrm{left}\ x), \\
&\qquad f\ x, \\
&\qquad \mathrm{map}\ f\ (\mathrm{iterate}\, \mathrm{right}\ x))
\end{aligned}$$

$$\begin{aligned}
\mathrm{left}, \mathrm{right} &: \mathrm{Line}\, t \to \mathrm{Line}\, t \\
\mathrm{left}\ ((a, l), o, r) &= (l, a, (o, r)) \\
\mathrm{right}\ (l, o, (a, r)) &= ((o, l), a, r)
\end{aligned}$$

A Ideas

Monads: construction / effects

Failure	$A + 1$
Error	$A + E$
Nondeterminism (sets)	$\mathcal{P}_\omega A$
Probability (discrete)	$\Pi_\omega A$
Entropy	$2^\omega \to A \times 2^\omega$
State	$S \to A \times S$
Environment/reader	$S \to A$
Monoid/writer	$A \times S$
Continuation	$(A \to R) \to R$
Input	$\mu X.\, A + (I \to X) \cong I^* \to A$
Output	$\mu X.\, A + (O \times X) \cong A \times O^*$
Modalities	$\Diamond\varphi$
Syntax / substitution	
Local state?	

Comonads: observation / context dependency (overview in [UV08])

Environment/reader
Exceptions [N05]
Intensional semantics [BG92]
Replication in linear λ
Codata [K99]
Implicit parameters / dynamic scope [LLMS00]
Redecoration [UV02]
Signals [UV05]
Dataflow [UV06]
Attribute grammers [UV05]

Cellular automata: http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html
Entropy: http://hackage.haskell.org/package/comonad-random-0.1.2
Many more: http://hackage.haskell.org/package/category-extras-0.53.5