# Code “monadification” made easy

J.N. Oliveira

Notes for the MiEI/LCC degrees

University of Minho/DI, June 2010

(last update: May 2020)

# Pointwise Haskell

Starting point: we unfold function $sum = (\![zero\ , add]\!)$ into

$$sum\ [\,] = 0$$
$$sum\ (h : t) = h + sum\ t$$

noting that this could have been written as follows

$$sum\ [\,] = id\ 0$$
$$sum\ (h : t) = \textbf{let}\ x = sum\ t\ \textbf{in}\ id\ (h + x)$$

using **let** notation. Why such a "verbose" version of the starting, so simple a piece of code?

# The easy rules

The **let** … **in**… notation stresses the fact that **recursive call** happens earlier than the delivery of the result, in general:

$$(f \cdot g)\ a = \textbf{let}\ b = g\ a\ \textbf{in}\ f\ \ b$$

The *id* function signals the **exit** points of the algorithm, that is, the points where it **returns** something to the caller.

Both lead straight to the equivalent, monadic version

$$\begin{aligned} &\mathit{msum}\ [\,] = \mathit{return}\ 0 \\ &\mathit{msum}\ (h : t) = \textbf{do}\ \{x \leftarrow \mathit{msum}\ t; \mathit{return}\ (h + x)\} \end{aligned}$$

under the rules:

- *id* becomes *return*
- **let** $x =$ …**in**… becomes **do** $\{x \leftarrow \ldots; \ldots\}$

# Identity monad

In fact, in the **identity** monad this version of *sum* is equivalent to the previous two, for **let** and **do** mean the same in such a monad, as do *id* and *return*.

It turns out that the monadic version just given,

$$\mathit{msum}\ [\,] = \mathit{return}\ 0$$
$$\mathit{msum}\ (h : t) = \mathbf{do}\ \{x \leftarrow \mathit{msum}\ t; \mathit{return}\ (h + x)\}$$

is *generic* in the sense that it runs on whatever monad you like. By default, the identity monad is chosen:

```
*Main> msum [3,4,5]
12
```

Haskell automatically switches to the monad you need, for instance

```
do { a <- msum [3,4,5]; writeFile "x" (show  a) }
```

# Adding effects

Indeed, you may add effects to your code that implicitly do the switching. For instance, by adding "printouts"

$$
\begin{aligned}
&\mathit{msum}'\ [\,] = \mathit{return}\ 0\\
&\mathit{msum}'\ (h : t) =\\
&\quad \textbf{do}\ \{x \leftarrow \mathit{msum}'\ t;\\
&\qquad \mathit{print}\ (\texttt{"x= "} + \mathit{show}\ x);\\
&\qquad \mathit{return}\ (h + x)\}
\end{aligned}
$$

traces the code in the way prescribed by the *print* function:

```
*Main> msum' [3,5,1,3,4]
"x= 0"
"x= 4"
"x= 7"
"x= 8"
"x= 13"
*Main>
```

# Summary

Recall the parallel,

$$(f \cdot g)\ x = \textbf{let}\ y = (g\ x)\ \textbf{in}\ f\ y$$

compared with

$$(f \bullet g)\ x = \textbf{do}\ \{y \leftarrow g\ x; f\ y\}$$

and

$$f \cdot id = f = id \cdot f$$

compared with

$$f \bullet return = f = return \bullet f$$

In the identity monad, $f \bullet g = f \cdot g$ and $return = id$.

# Adding effects

Adding effects is not as arbitrary as it may seem from the previous examples. This can be appreciated by defining the function $getmin$ that yields the smallest element of a list:

$$getmin\ [a] = a$$
$$getmin\ (h : t) = min\ h\ (getmin\ t)$$

This is incomplete because it does not specify the meaning of $getmin\ [\,]$.

To complete the definition, we first go monadic as we did before:

$$mgetmin\ [a] = return\ a$$
$$mgetmin\ (h : t) = \mathbf{do}\ \{x \leftarrow mgetmin\ t; return\ (min\ h\ x)\}$$

# Adding effects

Then we choose a monad to express the meaning of $\mathit{getmin}\ [\,]$, for instance the $\mathit{Maybe}$ monad

$$\begin{aligned}&\mathit{mgetmin}\ [\,] = \mathit{Nothing}\\&\mathit{mgetmin}\ [a] = \mathit{return}\ a\\&\mathit{mgetmin}\ (h : t) = \mathbf{do}\ \{x \leftarrow \mathit{mgetmin}\ t; \mathit{return}\ (\mathit{min}\ h\ x)\}\end{aligned}$$

Alternatively, we might have written

$$\mathit{mgetmin}\ [\,] = \mathit{Error}\ \texttt{"Empty input"}$$

going into the $\mathit{Error}$ monad, or even the simpler (yet interesting) $\mathit{mgetmin}\ [\,] = [\,]$, which shifts the code into the list monad, yielding singleton lists in the success case, otherwise the empty list.

# Example: map goes monadic

Partial functions (such as *getmin* above) cause much interference in functional programming. Monads help us to keep this under control.

Take $map\ f = (\!|\ \mathbf{in} \cdot (id + f \times id)\ |\!)$, that is

$$
\begin{array}{l}
map\ f\ [\,] = [\,] \\
map\ f\ (h : t) = (f\ \ h) : map\ f\ \ t
\end{array}
$$

as example and suppose $f$ is a partial function. How do we cope with erring evaluations of $f\ \ h$?

Easy — first we "letify" the function as before:

$$
\begin{array}{l}
map\ f\ [\,] = id\ [\,] \\
map\ f\ (h : t) = \mathbf{let} \\
\quad b = f\ \ h \\
\quad x = map\ f\ \ t\ \mathbf{in}\ id\ (b : x)
\end{array}
$$

# Example: map goes monadic

Then we go monadic in the usual way,

$$mmap\ f\ [\,] = return\ [\,]$$
$$mmap\ f\ (h : t) = \mathbf{do}\ \{b \leftarrow f\ \ h; x \leftarrow mmap\ f\ \ t; return\ (b : x)\}$$

thus building a function of the expected type:

$$mmap :: (Monad\ \ m) \Rightarrow (a \rightarrow m\ \ b) \rightarrow [a] \rightarrow m\ [b]$$

Let us see this at work:

```
mmap mgetmin [[1,2],[3]] = Just [1,3]
mmap mgetmin [[1,2],[]] = Nothing
```

# Another example: map goes monadic

Let us see the **same code** automatically switching to another monad, this time coping with probabilistic computations, e.g.

$$f\ x = \begin{cases} x + 1 & 70\% \\ x - 1 & 30\% \end{cases}$$

Probabilistic function $f$ either increments or decrements its input, with different probabilities.

We get a **probabilistic map** without changing a single line of code, cf. e.g.

$$\begin{array}{l} *\ Main > mmap\ f\ [1,2] \\ [2,3]\ 49.0\ \% \\ [0,3]\ 21.0\ \% \\ [2,1]\ 21.0\ \% \\ [0,1]\ 9.0\ \% \end{array}$$

# Final example: $(\!|inBTree|\!)$ goes (state) monadic

Recall that, by cata-reflection, function $f = (\!|inBTree|\!)$, that is,

$$
\begin{array}{l}
f\ Empty = Empty \\
f\ (Node\ (a,(x,y))) = Node\ (a,(f\ x,f\ y))
\end{array}
$$

does nothing, since $f = id$. Let us write this monadically, using the rules as before:

$$
\begin{array}{l}
f :: (Monad\ m) \Rightarrow BTree\ a \rightarrow m\ (BTree\ a) \\
f\ Empty = return\ Empty \\
f\ (Node\ (a,(x,y))) = \mathbf{do}\ \{ \\
\quad x' \leftarrow f\ x; \\
\quad y' \leftarrow f\ y; \\
\quad return\ (Node\ (a,(x',y')))\}
\end{array}
$$

*Doing nothing* can lead to *doing something useful* provided we add effects to $f$. This time we choose the **state** monad.

# Decorating trees

Recall two basic actions of the **state** monad:

- $get = \langle id, id \rangle$ — reads the current value of the state
- $put\ x = \langle !, \underline{x} \rangle$ writes value $x$ into the state

We can add these to $f$ above so that this decorates each node of input tree with a kind of "serial number", as follows:

$$
\begin{array}{l}
f\ Empty = return\ Empty \\
f\ (Node\ (a, (x, y))) = \mathbf{do}\ \{ \\
\quad n \leftarrow get; put\ (n + 1); \\
\quad x' \leftarrow f\ x; \\
\quad y' \leftarrow f\ y; \\
\quad return\ (Node\ ((a, n), (x', y')))\}
\end{array}
$$

# Decorating trees

**St.hs** (state monad) library:

$$\textbf{data}\ St\ s\ a = St\ \{ st :: (s \to (a, s)) \}$$

where $St$ and $st$ are the **in** and **out** of this type.

Final comments:

- Mind the type of $f$:

  $$f :: (Num\ s) \Rightarrow BTree\ a \to St\ s\ (BTree\ (a, s))$$

  once you choose the version of the sate monad available from module $St.hs$.
- Don't forget that the output of $f$ is now an action of an automaton; so you need to supply an **initial state** for the automaton to "run" — see examples in $St.hs$.
- Writing monadic code is not difficult provided one is **systematic**.

# Decorating trees

Another example (**Exp.hs** library)

$$
\begin{array}{l}
deco :: Num\ n \Rightarrow Exp\ v\ o \rightarrow Exp\ (n, v)\ (n, o) \\
deco\ e = \pi_1\ (st\ (f\ e)\ 0)\ \mathbf{where} \\
\quad f\ (Var\ e) = \mathbf{do}\ \{n \leftarrow get; put\ (n+1); return\ (Var\ (n, e))\} \\
\quad f\ (Term\ o\ l) = \mathbf{do}\ \{ \\
\quad\quad n \leftarrow get; put\ (n+1); \\
\quad\quad m \leftarrow sequence\ (map\ f\ l); \\
\quad\quad return\ (Term\ (n, o)\ m) \\
\quad\quad \}
\end{array}
$$

where

$$sequence :: [m\ a] \rightarrow m\ [a]$$

# Another St example

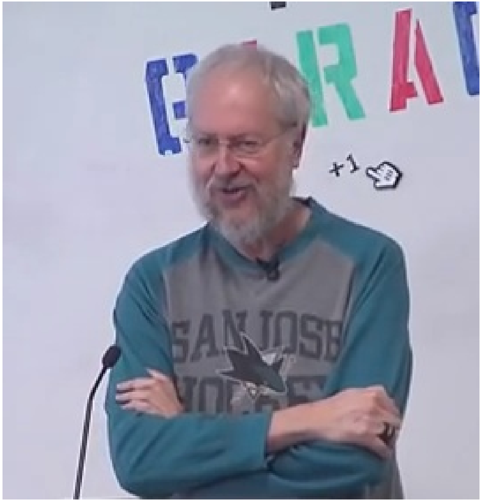
Stack automaton evaluating expression $x * (y + 2)$:

```
run x y = exec prog empty_stack
  where prog = do {   -- loading
    push (x);
    push (2);
    push (y);
      -- evaluating y + 2
    r1 ← pop ();
    r2 ← pop ();
    push (r1 + r2);
      -- evaluating x * (y + 2)
    r1 ← pop ();
    r2 ← pop ();
    push (r1 * r2);
      -- get returns current state
    query head
    }
```

# The monadic "curse" :-)

*"Monads [...] come with a curse. The monadic curse is that once someone learns what monads are and how to use them, they lose the ability to explain it to other people"*

(Douglas Crockford: *Google Tech Talk on how to express monads in JavaScript* YouTube 2013)




Douglas Crockford (2013)