“Monadification” made easy

J.N. Oliveira

CP 2009/10 - 2nd year course, LCC/LEI

University of Minho/DI, June 2010

Pointwise Haskell

Starting point: the *sum* function

$$
\begin{aligned}
&\mathit{sum}\ [\,] = 0\\
&\mathit{sum}\ (h : t) = h + \mathit{sum}\ t
\end{aligned}
$$

could have been written as follows

$$
\begin{aligned}
&\mathit{sum}\ [\,] = \mathit{id}\ 0\\
&\mathit{sum}\ (h : t) = \mathbf{let}\ x = \mathit{sum}\ t\ \mathbf{in}\ \mathit{id}\ (h + x)
\end{aligned}
$$

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

The easy rules

Comments:

- The **let** ... **in**... notation stresses the fact that recursive call happens earlier than the delivery of the result
- 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, under the rules:
  - *id* becomes *return*
  - **let** $x = \ldots$**in**... becomes **do** $\{x \leftarrow \ldots; \ldots\}$

  cf.

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

Identity monad

- In fact, there is a monad — the **identity** monad — in which 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,

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

  is *generic* in the sense that it runs on whatever monad you like, provided you add **effects** to it.
- Haskell automatically switches to the monad you want, depending on the effects you chose. (Examples follow.)

Adding effects

EXAMPLE: adding "printouts"

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

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>*

Adding effects

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

$$\begin{aligned} &getmin\ [a] = a \\ &getmin\ (h : t) = min\ h\ (getmin\ t) \end{aligned}$$

which is incomplete in the sense that it does not specify the meaning of $getmin\ [\,]$. To complete the defintion, we first go monadic, as we did before,

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

Adding effects

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

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

Alternatively, we might have written

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

going into the 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, which monads help us to keep under control.
- Take

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

  as example and suppose f is not a total function. How do we cope with erring evaluations of $f\ \ h$?
- Easy — first we "letify" the code as before:

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

Example: map goes monadic

... Then we go monadic in the usual way,

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

thus building a function of the expected type:

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

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

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

$$f\ Empty = Empty$$
$$f\ (Node\ (a, (x, y))) = Node\ (a, (f\ x, f\ y))$$

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 its "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

Final comments:

- Mind the type of f:

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

  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 no difficult provided one is systematic.

Good luck!