2.2. Reasoning

Eg universal property of *foldr*:

$$h = foldr\ f\ e \quad \Leftrightarrow \quad h\,[\,] = e \;\wedge\; h\,(x : xs) = f\ x\ (h\ xs)$$

(proof by appeal to initiality of list algebra).

Hence *fold fusion*:

$$h \circ foldr\ f\ e = foldr\ f'\ e' \quad \Leftarrow \quad h\ e = e' \;\wedge\; h\,(f\ x\ y) = f'\ x\ (h\ y)$$

Hence *fold–map fusion*:

$$foldr\ f\ e \circ map\ g = foldr\ (f \circ g)\ e$$

(Caveat calculator: assuming sets and total functions, for simplicity.)

3.2. State benefits

Subclasses of *Monad* for particular computational effects.

Eg a subclass of computations that supports mutable state:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad \mathit{get} :: m\ s \\
\quad \mathit{put} :: s \rightarrow m\ ()
\end{array}
$$

Four axioms should be satisfied:

$$
\begin{array}{lll}
\mathit{put}\ s \gg\!= \lambda() \rightarrow \mathit{put}\ s' & = \mathit{put}\ s' & \text{-- put-put} \\
\mathit{put}\ s \gg\!= \lambda() \rightarrow \mathit{get} & = \mathit{put}\ s \gg\!= \lambda() \rightarrow \mathit{return}\ s & \text{-- put-get} \\
\mathit{get} \gg\!= \mathit{put} & = \mathit{return}\ () & \text{-- get-put} \\
\mathit{get} \gg\!= \lambda s \rightarrow \mathit{get} \gg\!= k\ s & = \mathit{get} \gg\!= \lambda s \rightarrow k\ s\ s & \text{-- get-get}
\end{array}
$$

3.3. Two shorthands

Define

$$
\begin{aligned}
&skip && :: Monad\ m \Rightarrow m\ ()\\
&skip && = return\ ()\\
&(\gg) && :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b\\
&p \gg q = p \gg\!= (\lambda x \to q)
\end{aligned}
$$

Then state axioms become less noisy:

$$
\begin{aligned}
&put\ s \gg put\ s' && = put\ s' && \text{-- put-put}\\
&put\ s \gg get && = put\ s \gg return\ s && \text{-- put-get}\\
&get \gg\!= put && = skip && \text{-- get-put}\\
&get \gg\!= \lambda s \to get \gg\!= k\ s && = get \gg\!= \lambda s \to k\ s\ s && \text{-- get-get}
\end{aligned}
$$

3.5. Simulating state

Simulate mutable state via state-transforming functions—roughly,

$$\textbf{type}\ State\ s\ a = (s \to (a, s))$$

return leaves state unchanged, $\gg\!=$ chains state transformations together:

$$\begin{array}{l}\textbf{instance}\ Monad\ (State\ s)\ \textbf{where} \\ \quad return\ x = (\lambda s \to (x, s)) \\ \quad p \gg\!= k \quad = (\lambda s \to \textbf{let}\ (x, s') = p\ s\ \textbf{in}\ k\ x\ s')\end{array}$$

Of course, state-transforming functions simulate mutable state:

$$\begin{array}{l}\textbf{instance}\ MonadState\ s\ (State\ s)\ \textbf{where} \\ \quad get \quad\ = (\lambda s \to (s, s)) \\ \quad put\ s' = (\lambda s \to ((), s'))\end{array}$$

(Indeed, they're the initial model of the specification.)

4. Equational reasoning with effects

Augmenting an integer state:

$$\begin{aligned}&\mathit{add} :: \mathit{MonadState}\ \mathit{Integer}\ m \Rightarrow \mathit{Integer} \rightarrow m\ ()\\ &\mathit{add}\ n = \mathbf{do}\ \{m \leftarrow \mathit{get}\,;\mathit{put}\ (m+n)\}\end{aligned}$$

Then augmenting multiple times, via

$$\begin{aligned}&\mathit{addAll} :: \mathit{MonadState}\ \mathit{Integer}\ m \Rightarrow [\mathit{Integer}] \rightarrow m\ ()\\ &\mathit{addAll} = \mathit{sequence}\_ \circ \mathit{map}\ \mathit{add}\end{aligned}$$

is equivalent to augmenting by the sum:

$$\mathit{addAll} = \mathit{add} \circ \mathit{sum}$$

Here, $\mathit{sequence}\_$ composes a list of void-returning computations:

$$\begin{aligned}&\mathit{sequence}\_ :: \mathit{Monad}\ m \Rightarrow [m\ ()] \rightarrow m\ ()\\ &\mathit{sequence}\_ = \mathit{foldr}\ (\gg)\ \mathit{skip}\end{aligned}$$

4.1. Using fusion

Because $sequence\_$ is a fold, we can use fold–map fusion:

$$\begin{array}{l} addAll = foldr\ addThen\ skip \\ \quad \mathbf{where}\ addThen\ n\ p = \mathbf{do}\ \{add\ n\,;\ p\} \end{array}$$

And because $addAll$ and sum are both instances of $foldr$, the result

$$addAll = add \circ sum$$

then follows by fold fusion from the two simple properties

$$\begin{array}{lcl} add\ 0 & = & skip \\ add\ (n + n') & = & addThen\ n\ (add\ n') \end{array}$$

So let's prove these…

4.2. Adding zero

$$
\begin{array}{ll}
& \mathit{add}\ 0 \\
= & [\![\ \text{definition of } \mathit{add}\ ]\!] \\
& \mathbf{do}\ \{ l \leftarrow \mathit{get}\ ; \mathit{put}\ (l + 0) \} \\
= & [\![\ \text{arithmetic}\ ]\!] \\
& \mathbf{do}\ \{ l \leftarrow \mathit{get}\ ; \mathit{put}\ l \} \\
= & [\![\ \text{get-put}\ ]\!] \\
& \mathit{skip}
\end{array}
$$

4.3. Adding a sum

$$
\begin{array}{ll}
 & \mathit{addThen}\ n\ (\mathit{add}\ n') \\
= & [\![\text{ definitions of } \mathit{addThen} \text{ and } \mathit{add} \ ]\!] \\
 & \mathbf{do}\ \{ \mathbf{do}\ \{ m \leftarrow \mathit{get} \,;\, \mathit{put}\ (m+n) \} \,;\, \mathbf{do}\ \{ l \leftarrow \mathit{get} \,;\, \mathit{put}\ (l+n') \} \} \\
= & [\![\text{ associativity of composition }]\!] \\
 & \mathbf{do}\ \{ m \leftarrow \mathit{get} \,;\, \mathit{put}\ (m+n) \,;\, l \leftarrow \mathit{get} \,;\, \mathit{put}\ (l+n') \} \\
= & [\![\text{ put–get }]\!] \\
 & \mathbf{do}\ \{ m \leftarrow \mathit{get} \,;\, \mathit{put}\ (m+n) \,;\, \mathit{put}\ ((m+n)+n') \} \\
= & [\![\text{ associativity of addition }]\!] \\
 & \mathbf{do}\ \{ m \leftarrow \mathit{get} \,;\, \mathit{put}\ (m+n) \,;\, \mathit{put}\ (m+(n+n')) \} \\
= & [\![\text{ put–put }]\!] \\
 & \mathbf{do}\ \{ m \leftarrow \mathit{get} \,;\, \mathit{put}\ (m+(n+n')) \} \\
= & [\![\text{ definition of } \mathit{add} \ ]\!] \\
 & \mathit{add}\ (n+n')
\end{array}
$$

QED.