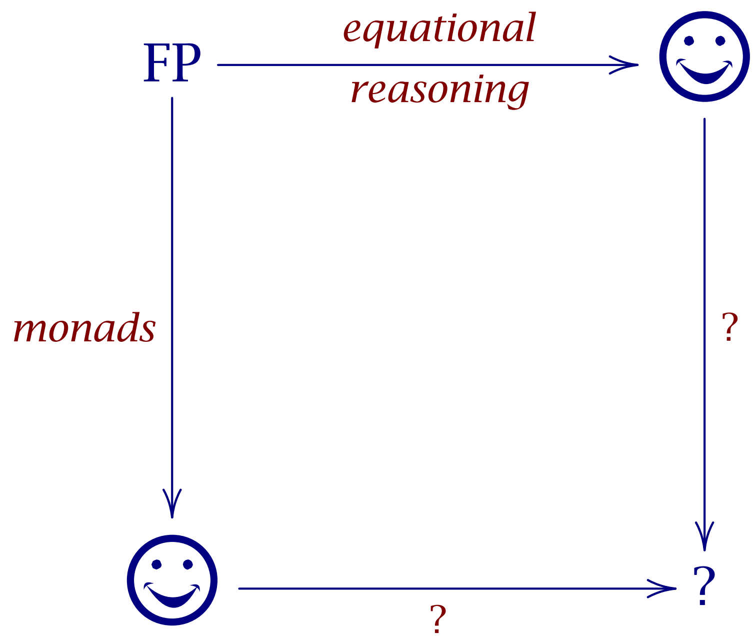
Just do It:

Simple Monadic Equational Reasoning

Jeremy Gibbons (jww Ralf Hinze)

Shonan, March 2019

1. Reasoning with effects?



2. Monads in Haskell

An interface for effectful computation:

$$\begin{array}{l} \mathbf{class}\ \mathit{Monad}\ m\ \mathbf{where} \\ \quad \mathit{return} :: a \to m\ a \\ \quad (\gg\!=) \ \ :: m\ a \to (a \to m\ b) \to m\ b \end{array}$$

Unit and associativity laws:

$$\begin{array}{lcl} \mathit{return}\ x \gg\!= k & = & k\ x \\ \mathit{mx} \gg\!= \mathit{return} & = & \mathit{mx} \\ (\mathit{mx} \gg\!= k) \gg\!= k' & = & \mathit{mx} \gg\!= (\lambda x \to k\ x \gg\!= k') \end{array}$$

Two abbreviations:

$$\begin{array}{ll} \mathit{skip} :: \mathit{Monad}\ m \Rightarrow m\ () & (\gg) \qquad :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b \\ \mathit{skip} = \mathit{return}\ () & \mathit{mx} \gg \mathit{my} = \mathit{mx} \gg\!= \mathit{const}\ \mathit{my} \end{array}$$

2.1. Imperative functional programming

$$
\begin{aligned}
\mathbf{do}\ \{e\} &= e \\
\mathbf{do}\ \{x \leftarrow e; es\} &= e \gg\!= \lambda x \rightarrow \mathbf{do}\ \{es\} \\
\mathbf{do}\ \{e; es\} &= e \gg \mathbf{do}\ \{es\} \\
\mathbf{do}\ \{\mathbf{let}\ decls; es\} &= \mathbf{let}\ decls\ \mathbf{in}\ \mathbf{do}\ \{es\}
\end{aligned}
$$

'Haskell is the world's best imperative programming language.' (SPJ)

3. A counter example

The *Monad* interface provides general-purpose plumbing.
For any particular class of effect, we need additional operations.

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadCount}\ m\ \textbf{where}$$
$$\quad \mathit{tick} :: m\ ()$$

Then, for example, Towers of Hanoi:

$$\begin{aligned}
&\mathit{hanoi} :: \mathit{MonadCount}\ m \Rightarrow \mathit{Int} \rightarrow m\ () \\
&\mathit{hanoi}\ 0 \qquad\quad\ = \textbf{do}\ \{\mathit{skip}\} \\
&\mathit{hanoi}\ (n+1) = \textbf{do}\ \{\mathit{hanoi}\ n; \mathit{tick}; \mathit{hanoi}\ n\}
\end{aligned}$$

3.1. Correctness

We claim that

$$hanoi\ n = \mathbf{do}\ \{rep\ (2^n - 1)\ tick\}$$

where

$$\begin{aligned}
&rep :: Monad\ m \Rightarrow Int \to m\ () \to m\ () \\
&rep\ 0\ \qquad\quad mx = \mathbf{do}\ \{skip\} \\
&rep\ (n+1)\ mx = \mathbf{do}\ \{mx; rep\ n\ mx\}
\end{aligned}$$

Note that

$$\begin{aligned}
&rep\ 1\ \qquad\qquad mx = \mathbf{do}\ \{mx\} \\
&rep\ (m+n)\ mx = \mathbf{do}\ \{rep\ m\ mx; rep\ n\ mx\}
\end{aligned}$$

3.2. Reasoning

Proof by induction. Base case trivial:

$$\mathit{hanoi}\ 0 = \mathbf{do}\ \{\mathit{skip}\} = \mathbf{do}\ \{\mathit{rep}\ (2^0 - 1)\ \mathit{tick}\}$$

For inductive step,

$$\begin{array}{ll}
 & \mathit{hanoi}\ (n+1) \\
= & [\![\ \text{definition of } \mathit{hanoi}\ ]\!] \\
 & \mathbf{do}\ \{\mathit{hanoi}\ n; \mathit{tick}; \mathit{hanoi}\ n\} \\
= & [\![\ \text{inductive hypothesis; } \mathit{rep}\ 1\ ]\!] \\
 & \mathbf{do}\ \{\mathit{rep}\ (2^n - 1)\ \mathit{tick}; \mathit{rep}\ 1\ \mathit{tick}; \mathit{rep}\ (2^n - 1)\ \mathit{tick}\} \\
= & [\![\ \mathit{rep} \text{ promotes through addition}\ ]\!] \\
 & \mathbf{do}\ \{\mathit{rep}\ (2^n - 1 + 1 + 2^n - 1)\ \mathit{tick}\} \\
= & [\![\ \text{arithmetic}\ ]\!] \\
 & \mathbf{do}\ \{\mathit{rep}\ (2^{n+1} - 1)\ \mathit{tick}\}
\end{array}$$

A particularly simple example, because *MonadCount* algebra is free.

4. Failure, choice and nondeterminism

A class of possibly failing computations:

$$\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFail}\ m\ \mathbf{where}$$
$$\quad \mathit{fail} :: m\ a$$

such that *fail* is a left zero of sequential composition:

$$\mathit{fail} \gg m = \mathit{fail}$$

(but not a right zero!).

Useful shorthand:

$$\mathit{guard} :: \mathit{MonadFail}\ m \Rightarrow \mathit{Bool} \rightarrow m\ ()$$
$$\mathit{guard}\ b = \mathbf{if}\ b\ \mathbf{then}\ \mathit{skip}\ \mathbf{else}\ \mathit{fail}$$

4.1. Choice

A class of computations that make choices:

$$\textbf{class}\ \mathit{MonadAlt}\ m\ \textbf{where}$$
$$\quad (\Box) :: m\ a \to m\ a \to m\ a$$

such that $\Box$ is associative, and composition distributes leftwards over it:

$$\begin{aligned}(m \Box n) \Box p &= m \Box (n \Box p)\\ (m \Box n) \gg\!= k &= (m \gg\!= k) \Box (n \gg\!= k)\end{aligned}$$

(but not rightwards!).

4.2. Nondeterminism

…as a combination of failure and choice:

$$\textbf{class}\ (\mathit{MonadFail}\ m, \mathit{MonadAlt}\ m) \Rightarrow \mathit{MonadNondet}\ m$$

No additional operations. But two additional unit laws:

$$\mathit{fail} \mathbin{\Box} \mathit{mx} = \mathit{mx} = \mathit{mx} \mathbin{\Box} \mathit{fail}$$

Finite lists, bags, and sets are instances
(the latter two adding commutativity and idempotence of $(\Box)$, respectively).

4.3. Permutations

For example,

$$
\begin{aligned}
&\mathit{perms} :: \mathit{MonadNondet}\ m \Rightarrow [\,a\,] \rightarrow m\,[\,a\,]\\
&\mathit{perms}\ [\,] = \mathbf{do}\ \{\mathit{return}\ [\,]\,\}\\
&\mathit{perms}\ \mathit{xs} = \mathbf{do}\ \{(y, \mathit{ys}) \leftarrow \mathit{select}\ \mathit{xs}; \mathit{zs} \leftarrow \mathit{perms}\ \mathit{ys}; \mathit{return}\ (y : \mathit{zs})\,\}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{select} :: \mathit{MonadNondet}\ m \Rightarrow [\,a\,] \rightarrow m\ (a, [\,a\,])\\
&\mathit{select}\ [\,] \qquad\quad = \mathbf{do}\ \{\mathit{fail}\,\}\\
&\mathit{select}\ (x : \mathit{xs}) = \mathbf{do}\ \{\mathit{return}\ (x, \mathit{xs})\,\} \ \square\\
&\qquad\qquad\qquad\quad \mathbf{do}\ \{(y, \mathit{ys}) \leftarrow \mathit{select}\ \mathit{xs}; \mathit{return}\ (y, x : \mathit{ys})\,\}
\end{aligned}
$$

5. State

A class of computations exploiting updatable state:

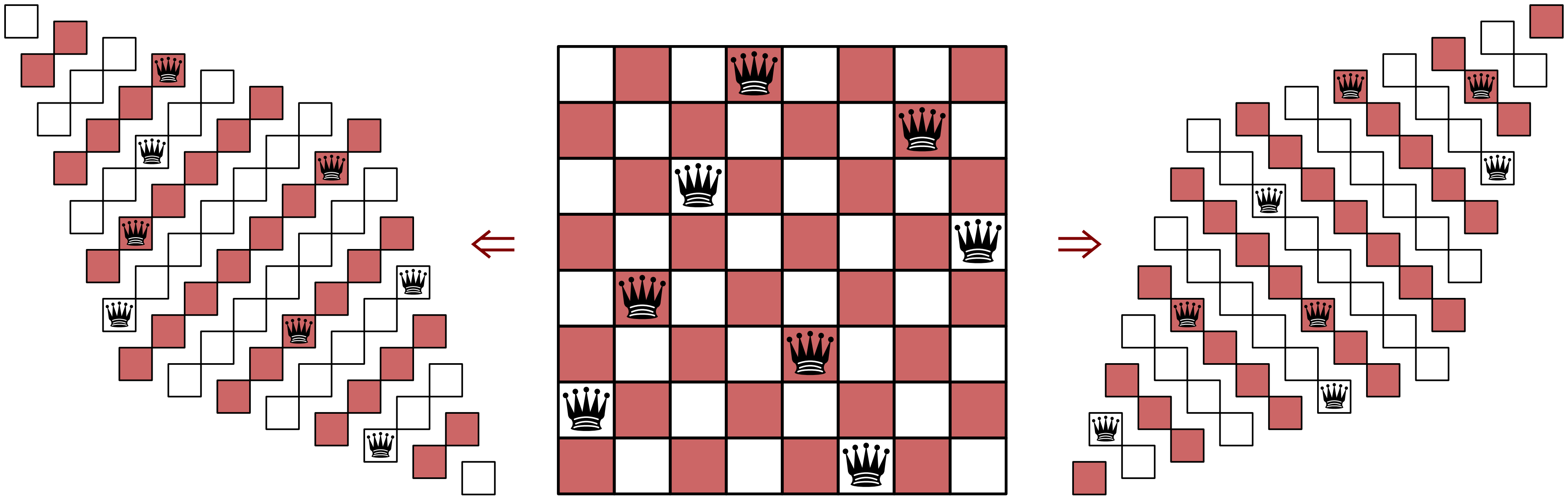
$$\begin{array}{l} \mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \mathbf{where} \\ \quad \mathit{get} :: m\ s \\ \quad \mathit{put} :: s \rightarrow m\ () \end{array}$$

with four axioms:

$$\begin{array}{ll} \mathit{put}\ s \gg \mathit{put}\ s' & = \mathit{put}\ s' \\ \mathit{put}\ s \gg \mathit{get} & = \mathit{put}\ s \gg \mathit{return}\ s \\ \mathit{get} \gg\!= \mathit{put} & = \mathit{skip} \\ \mathit{get} \gg\!= \lambda s \rightarrow \mathit{get} \gg\!= k\ s & = \mathit{get} \gg\!= \lambda s \rightarrow k\ s\ s \end{array}$$

5.1. Eight queens

Queen at (r, c) threatens up-diagonal $r-c$ and down-diagonal $r+c$:



The essence of queen safety:

$$
\begin{array}{l}
\mathit{test} :: (\mathit{Int}, \mathit{Int}) \to ([\mathit{Int}], [\mathit{Int}]) \to (\mathit{Bool}, ([\mathit{Int}], [\mathit{Int}])) \\
\mathit{test}\ (c, r)\ (\mathit{ups}, \mathit{downs}) = (u \notin \mathit{ups} \wedge d \notin \mathit{downs}, (u : \mathit{ups}, d : \mathit{downs})) \\
\quad \mathbf{where}\ (u, d) = (r-c, r+c)
\end{array}
$$

5.2. Eight queens, purely

The safety test for a candidate layout:

$$
\begin{array}{l}
\mathit{safe}_1 :: ([\,\mathit{Int}\,],[\,\mathit{Int}\,]) \to [\,(\mathit{Int},\mathit{Int})\,] \to (\mathit{Bool},([\,\mathit{Int}\,],[\,\mathit{Int}\,])) \\
\mathit{safe}_1 = \mathit{foldr}\ \mathit{step}_1 \circ \mathit{start}_1\ \mathbf{where} \\
\quad \mathit{start}_1\ \mathit{updowns} = (\mathit{True},\mathit{updowns}) \\
\quad \mathit{step}_1\ \mathit{cr}\ (\mathit{restOK},\mathit{updowns}) = (\mathit{thisOK} \wedge \mathit{restOK},\mathit{updowns}') \\
\qquad \mathbf{where}\ (\mathit{thisOK},\mathit{updowns}') = \mathit{test}\ \mathit{cr}\ \mathit{updowns}
\end{array}
$$

Then generate and test:

$$
\begin{array}{l}
\mathit{queens} :: \mathit{MonadNondet}\ m \Rightarrow \mathit{Int} \to m\,[\,\mathit{Int}\,] \\
\mathit{queens}\ n \ = \mathbf{do}\ \{\mathit{rs} \leftarrow \mathit{perms}\ [1\,..\,n]; \\
\qquad\qquad\qquad \mathit{guard}\ (\mathit{fst}\ (\mathit{safe}_1\ \mathit{empty}\ (\mathit{place}\ n\ \mathit{rs})));\mathit{return}\ \mathit{rs}\} \\
\mathit{place}\ n\ \mathit{rs} = \mathit{zip}\ [1\,..\,n]\ \mathit{rs} \\
\mathit{empty} \qquad = ([\,],[\,])
\end{array}
$$

5.3. Safety testing, statefully

Maintain the checked diagonals statefully:

$$\begin{array}{l}
\mathit{safe}_2 :: \mathit{MonadState}\ ([\mathit{Int}],[\mathit{Int}])\ m \Rightarrow [(\mathit{Int},\mathit{Int})] \rightarrow m\ \mathit{Bool} \\
\mathit{safe}_2 = \mathit{foldr}\ \mathit{step}_2\ \mathit{start}_2\ \mathbf{where} \\
\quad \mathit{start}_2 \qquad = \mathbf{do}\ \{\mathit{return}\ \mathit{True}\} \\
\quad \mathit{step}_2\ \mathit{cr}\ k = \mathbf{do}\ \{\mathit{restOK} \leftarrow k; \mathit{updowns} \leftarrow \mathit{get}; \\
\qquad\qquad\qquad\qquad \mathbf{let}\ (\mathit{thisOK}, \mathit{updowns}') = \mathit{test}\ \mathit{cr}\ \mathit{updowns}; \\
\qquad\qquad\qquad\qquad \mathit{put}\ \mathit{updowns}'; \mathit{return}\ (\mathit{thisOK} \wedge \mathit{restOK})\}
\end{array}$$

Simple proof using axioms of *get* and *put* that

$$\begin{array}{l}
\mathit{safe}_2\ \mathit{crs} \\
\quad = \mathbf{do}\ \{\mathit{updowns} \leftarrow \mathit{get}; \\
\qquad\qquad \mathbf{let}\ (\mathit{ok}, \mathit{updowns}') = \mathit{safe}_1\ \mathit{updowns}\ \mathit{crs}; \\
\qquad\qquad \mathit{put}\ \mathit{updowns}'; \mathit{return}\ \mathit{ok}\}
\end{array}$$

6. Combining effects

Nondeterminism for permutations, state for safety testing:

$$\mathbf{class}\ (\mathit{MonadState}\ s\ m, \mathit{MonadNondet}\ m) \Rightarrow \mathit{MonadStateNondet}\ s\ m \mid m \rightarrow s$$

Again, no new operations, but some additional laws—*fail* also a right zero:

$$m \gg \mathit{fail} = \mathit{fail}$$

and composition distributes also rightwards over choice:

$$m \gg\!= \lambda x \rightarrow k_1\ x \mathbin{\Box} k_2\ x = (m \gg\!= k_1) \mathbin{\Box} (m \gg\!= k_2)$$

That is, *local* or *backtrackable state*. (Each choice point entails a clean slate.)
In particular, guards commute with anything:

$$\mathit{guard}\ b \gg m = m \gg\!= \lambda x \rightarrow \mathit{guard}\ b \gg \mathit{return}\ x$$

6.1. Queens, nondeterministically and statefully

Using $get \ggg put = skip$ and commuting guards, calculate

$$
\begin{aligned}
queens\ n = {} & \mathbf{do}\ \{ rs \leftarrow perms\ [1\,..\,n]; \\
& \qquad guard\ (fst\ (safe_1\ empty\ (place\ n\ rs))); return\ rs \} \\
= {} & \mathbf{do}\ \{ s \leftarrow get; rs \leftarrow perms\ [1\,..\,n]; put\ empty; \\
& \qquad ok \leftarrow safe_2\ (place\ n\ rs); put\ s; guard\ ok; return\ rs \} \\
= {} & \mathbf{do}\ \{ s \leftarrow get; rs \leftarrow perms\ [1\,..\,n]; put\ empty; \\
& \qquad ok \leftarrow safe_2\ (place\ n\ rs); guard\ ok; put\ s; return\ rs \} \\
= {} & \mathbf{do}\ \{ s \leftarrow get; rs \leftarrow perms\ [1\,..\,n]; put\ empty; \\
& \qquad safe_3\ (place\ n\ rs); put\ s; return\ rs \}
\end{aligned}
$$

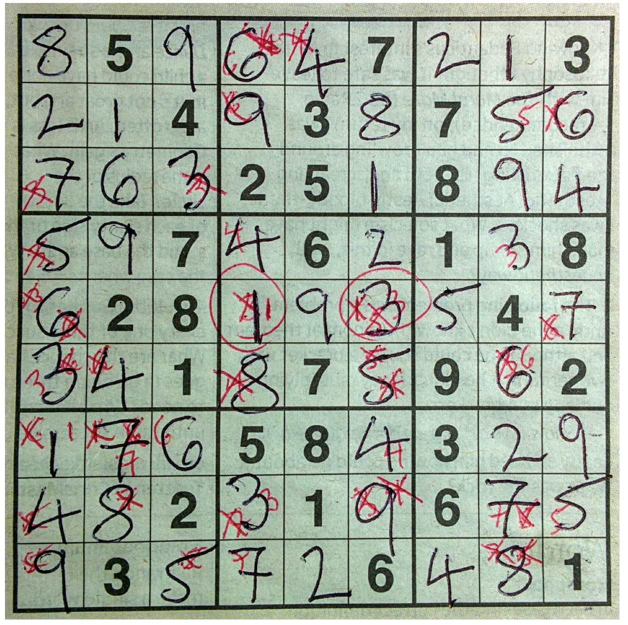
where $safe_3\ crs = safe_2\ crs \ggg guard$. Then calculate that

$$safe_3\ crs = foldr\ step_3\ start_3\ \mathbf{where}\ step_3 = \ldots; start_3 = \ldots$$

by plain ordinary equational reasoning.

7. Think locally, act globally

- state and failure combine, in two ways
- *local* state: $s \to \mathit{Maybe}\ (a, s)$
- *global* state: $s \to (\mathit{Maybe}\ a, s)$
- different interactions between the two theories
- state and nondeterminism combine nicely *locally*: $s \to [\,(a, s)\,]$
- but sometimes you want *global* state
- eg Prolog evaluator, or playing Sudoku
- however, $s \to ([\,a\,], s)$ is not a monad
- what is the equational theory? and implementation?



8. Summary

- computational effects as algebraic theories
- the axioms are important! as with type classes etc too
- theories combine—trivially, or with interaction
- *making equations great again*
- personal bugbear:
  language designers are compiler writers
- *Just do it*,
  JG and Ralf Hinze,
  ICFP 2011