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:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Unit and associativity laws:

```
return x >>= k  = k x
mx >>= return  = mx
(mx >>= k) >>= k' = mx >>= (\x -> k x >>= k')
```

Two abbreviations:

```
skip :: Monad m => m ()    (>>=) :: Monad m => m a -> m b -> m b
skip = return ()
      mx >>= my = mx >>= const my
```
2.1. Imperative functional programming

\[
\begin{align*}
\text{do } \{ e \} &= e \\
\text{do } \{ x \leftarrow e; es \} &= e \implies \lambda x \rightarrow \text{do } \{ es \} \\
\text{do } \{ e; es \} &= e \implies \text{do } \{ es \} \\
\text{do } \{ \text{let } decls; es \} &= \text{let } decls \text{ in } \text{do } \{ es \}
\end{align*}
\]

‘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.

\[
\text{class Monad } m \Rightarrow \text{MonadCount } m \text{ where}
\]
\[
tick :: m ()
\]

Then, for example, Towers of Hanoi:

\[
hanoi :: \text{MonadCount } m \Rightarrow \text{Int } \rightarrow m ()
\]
\[
hanoi 0 \quad = \text{ do } \{ \text{skip} \}
\]
\[
hanoi (n + 1) \quad = \text{ do } \{ \text{hanoi } n; \text{tick; hanoi } n \} 
\]
3.1. Correctness

We claim that

\[ hanoi \ n = \ do \ \{ \ rep \ (2^n - 1) \ tick \} \]

where

\[ rep :: \ Monad \ m \Rightarrow \ Int \rightarrow \ m () \rightarrow \ m () \]
\[ rep \ 0 \ mx = \ do \ \{ \ skip \} \]
\[ rep \ (n + 1) \ mx = \ do \ \{ \ mx; \ rep \ n \ mx \} \]

Note that

\[ rep \ 1 \ mx = \ do \ \{ \ mx \} \]
\[ rep \ (m + n) \ mx = \ do \ \{ \ rep \ m \ mx; \ rep \ n \ mx \} \]
3.2. Reasoning

Proof by induction. Base case trivial:

\[ hanoi \ 0 = \ \text{do} \ \{ \ \text{skip} \} = \ \text{do} \ \{ \ \text{rep} \ (2^0 - 1) \ \text{tick} \} \]

For inductive step,

\[
\begin{align*}
\text{hanoi} \ (n + 1) & = \ \text{[[ definition of hanoi ]]}
\text{do} \ \{ \ \text{hanoi} \ n; \ \text{tick}; \ \text{hanoi} \ n \}
\text{=} \ \text{[[ inductive hypothesis; rep 1 ]]}
\text{do} \ \{ \ \text{rep} \ (2^n - 1) \ \text{tick}; \ \text{rep} \ 1 \ \text{tick}; \ \text{rep} \ (2^n - 1) \ \text{tick} \}
\text{=} \ \text{[[ rep promotes through addition ]]}
\text{do} \ \{ \ \text{rep} \ (2^n - 1 + 1 + 2^n - 1) \ \text{tick} \}
\text{=} \ \text{[[ arithmetic ]]}
\text{do} \ \{ \ \text{rep} \ (2^{n+1} - 1) \ \text{tick} \}
\end{align*}
\]

A particularly simple example, because *MonadCount* algebra is free.
4. Failure, choice and nondeterminism

A class of possibly failing computations:

```haskell
class Monad m ⇒ MonadFail m where
  fail :: m a
```

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

```
fail >>= m = fail
```

(but not a right zero!).

Useful shorthand:

```haskell
guard :: MonadFail m ⇒ Bool → m ()
guard b = if b then skip else fail
```
4.1. Choice

A class of computations that make choices:

```haskell
class MonadAlt m where
  (□) :: m a → m a → m a
```

such that □ is associative, and composition distributes leftwards over it:

```
(m □ n) □ p = m □ (n □ p)
(m □ n) >> k = (m >> k) □ (n >> k)
```

(but not rightwards!).
4.2. Nondeterminism

...as a combination of failure and choice:

\[
\text{class } (\text{MonadFail } m, \text{MonadAlt } m) \Rightarrow \text{MonadNondet } m
\]

No additional operations. But two additional unit laws:

\[
\text{fail } \square mx = mx = mx \square \text{fail}
\]

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

For example,

\[
\text{perms} :: \text{MonadNondet} m \Rightarrow [a] \to m [a] \\
\text{perms} [ ] = \text{do} \{ \text{return} [ ] \} \\
\text{perms} xs = \text{do} \{ (y, ys) \leftarrow \text{select} \; xs; zs \leftarrow \text{perms} \; ys; \text{return} \; (y : zs) \}
\]

where

\[
\text{select} :: \text{MonadNondet} m \Rightarrow [a] \to m (a, [a]) \\
\text{select} [ ] = \text{do} \{ \text{fail} \} \\
\text{select} (x : xs) = \text{do} \{ \text{return} \; (x, xs) \} \quad \square \\
\quad \text{do} \{ (y, ys) \leftarrow \text{select} \; xs; \text{return} \; (y, x : ys) \}
\]
5. State

A class of computations exploiting updatable state:

```haskell
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

with four axioms:

- \( \text{put } s \gg \text{put } s' = \text{put } s' \)
- \( \text{put } s \gg \text{get} = \text{put } s \gg \text{return } s \)
- \( \text{get} \gg \text{put} = \text{skip} \)
- \( \text{get} \gg \lambda s \to \text{get} \gg k s = \text{get} \gg \lambda s \to k s s \)
5.1. Eight queens

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

The essence of queen safety:

\[
\text{test :: } (\text{Int}, \text{Int}) \rightarrow ([\text{Int}], [\text{Int}]) \rightarrow (\text{Bool}, ([\text{Int}], [\text{Int}]))
\]

\[
\text{test } (c, r) (\text{ups}, \text{downs}) = (u \notin \text{ups} \land d \notin \text{downs}, (u : \text{ups}, d : \text{downs}))
\]

\[
\text{where } (u, d) = (r - c, r + c)
\]
5.2. Eight queens, purely

The safety test for a candidate layout:

\[
\text{safe}_1 :: ([\text{Int}], [\text{Int}]) \rightarrow [(\text{Int}, \text{Int})] \rightarrow (\text{Bool}, ([\text{Int}], [\text{Int}]))
\]

\[
\text{safe}_1 = \text{foldr step}_1 \circ \text{start}_1 \quad \text{where}
\]

\[
\text{start}_1 \ \text{updowns} = (\text{True}, \text{updowns})
\]

\[
\text{step}_1 \ cr \ (\text{restOK}, \text{updowns}) = (\text{thisOK} \land \text{restOK}, \text{updowns}')
\]

\[
\text{where} \ (\text{thisOK}, \text{updowns}') = \text{test cr updowns}
\]

Then generate and test:

\[
\text{queens} :: \text{MonadNondet} \ m \Rightarrow \text{Int} \rightarrow m [\text{Int}]
\]

\[
\text{queens} \ n = \text{do} \{ \text{rs} \leftarrow \text{perms} [1..n]; \text{guard (fst (safe}_1 \text{ empty (place n rs)))}; \text{return rs} \}
\]

\[
\text{place} \ n \ \text{rs} = \text{zip} [1..n] \ \text{rs}
\]

\[
\text{empty} = ([], [])
\]
5.3. Safety testing, statefully

Maintain the checked diagonals statefully:

\[ \text{safe}_2 :: \text{MonadState} (\mathbb{I}nt, \mathbb{I}nt) \; m \Rightarrow \mathbb{I}nt \mathbb{I}nt \rightarrow m \text{Bool} \]

\[ \text{safe}_2 = \text{foldr} \; \text{step}_2 \; \text{start}_2 \; \text{where} \]

\[ \text{start}_2 \quad = \text{do} \{ \text{return True} \} \]

\[ \text{step}_2 \; \text{cr} \; k = \text{do} \{ \text{restOK} \leftarrow k; \text{updowns} \leftarrow \text{get}; \]

\[ \quad \text{let} \; (\text{thisOK}, \text{updowns}') = \text{test} \; \text{cr} \; \text{updowns}; \]

\[ \quad \text{put} \; \text{updowns}'; \text{return} \; (\text{thisOK} \land \text{restOK}) \} \]

Simple proof using axioms of \text{get} and \text{put} that

\[ \text{safe}_2 \; \text{crs} \]

\[ = \text{do} \{ \text{updowns} \leftarrow \text{get}; \]

\[ \quad \text{let} \; (\text{ok}, \text{updowns}') = \text{safe}_1 \; \text{updowns} \; \text{crs}; \]

\[ \quad \text{put} \; \text{updowns}'; \text{return} \; \text{ok} \} \]
6. Combining effects

Nondeterminism for permutations, state for safety testing:

```haskell
class (MonadState s m, MonadNondet m) ⇒ MonadStateNondet s m | m -> s
```

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

```haskell
m >>= fail = fail
```

and composition distributes also rightwards over choice:

```haskell
m >>= \x -> k₁ x □ k₂ x = (m >>= k₁) □ (m >>= k₂)
```

That is, local or backtrackable state. (Each choice point entails a clean slate.)

In particular, guards commute with anything:

```haskell
guard b >>= m = m >>= \x -> guard b >>= return x
```
6.1. Queens, nondeterministically and statefully

Using \( \text{get} \gg= \text{put} = \text{skip} \) and commuting guards, calculate

\[
\text{queens } n = \text{do } \{ \text{rs} \leftarrow \text{perms} [1..n]; \\
\quad \text{guard} (\text{fst} (\text{safe}_1 \text{ empty} (\text{place} n \text{ rs}))); \text{return} \text{rs} \}
\]

\[
= \text{do } \{ s \leftarrow \text{get}; \text{rs} \leftarrow \text{perms} [1..n]; \text{put empty}; \\
\quad \text{ok} \leftarrow \text{safe}_2 (\text{place} n \text{ rs}); \text{put} s; \text{guard} \text{ok}; \text{return} \text{rs} \}
\]

\[
= \text{do } \{ s \leftarrow \text{get}; \text{rs} \leftarrow \text{perms} [1..n]; \text{put empty}; \\
\quad \text{ok} \leftarrow \text{safe}_2 (\text{place} n \text{ rs}); \text{guard} \text{ok}; \text{put} s; \text{return} \text{rs} \}
\]

\[
= \text{do } \{ s \leftarrow \text{get}; \text{rs} \leftarrow \text{perms} [1..n]; \text{put empty}; \\
\quad \text{safe}_3 (\text{place} n \text{ rs}); \text{put} s; \text{return} \text{rs} \}
\]

where \( \text{safe}_3 \text{ crs} = \text{safe}_2 \text{ crs} \gg= \text{guard} \). Then calculate that

\[
\text{safe}_3 \text{ crs} = \text{foldr} \text{ step}_3 \text{ start}_3 \text{ where} \text{ step}_3 = \ldots; \text{start}_3 = \ldots
\]

by plain ordinary equational reasoning.
7. Think locally, act globally

- state and failure combine, in two ways
  
  - *local* state: \( s \rightarrow \text{Maybe} \ (a, s) \)
  
  - *global* state: \( s \rightarrow (\text{Maybe} \ a, s) \)

- different interactions between the two theories

- state and nondeterminism combine nicely *locally*: \( s \rightarrow [ \ (a, s) \ ] \)

- but sometimes you want *global* state

  - eg Prolog evaluator, or playing Sudoku

- however, \( s \rightarrow ([a], s) \) 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