1. Reasoning with effects?

FP  \[\text{equational reasoning}\]  \[\text{monads}\]

\[\text{\smiley}\]  \[?\]  \[\text{\smiley}\]  \[?\]
1.1. Seeing the wood through the trees

At TFP 2008, Hutton & Fulger discuss the ‘correctness’ of

\[
\text{relabel} :: \text{Tree } a \rightarrow \text{Tree Int}
\]

as an effectful (stateful) functional program.

I think they miss two opportunities for abstraction:

- from the specific effects (they expand the State monad to state-transforming functions), and
- from the pattern of computation (they use explicit induction on trees).

This is an attempt to address the first question. (The second is a story for another time.)
2. Monads

‘Ordinary’ monads, with the usual laws:

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

Special cases:

```haskell
skip :: Monad m => m ()
skip = return ()
(≫) :: Monad m => m a -> m b -> m b
k ≫ l = k >>= const l
```
2.1. Fallibility

Computations may fail:

```haskell
class Monad m ⇒ MonadZero m where
    mzero :: m a
```

such that

```haskell
mzero >>= k = mzero
```

(I'm curious as to why it's not like this in Haskell 98...) Often we just use

```haskell
mzero = ⊥
```
2.2. Guards

Define

\[
\text{guard} :: \text{MonadZero}~m \Rightarrow \text{Bool} \to m~() \\
guard~b = \text{if } b \text{ then } \text{skip} \text{ else } \text{mzero}
\]

We’ll write ‘\(b!\)’ for ‘\(\text{guard } b\)’.

Familiar properties:

\[
\begin{align*}
\text{True}! &= \text{skip} \\
\text{False}! &= \text{mzero} \\
(b_1 \land b_2)! &= b_1! \gg b_2! \\
b_1! \gg k \gg b_2! &= b_1! \gg k \iff b_1 \Rightarrow b_2
\end{align*}
\]
2.3. Assertions

For $k :: MonadZero ()$, write ‘$k \{ b \}$’ for

$$\text{do } \{ k; b! \} = \text{do } \{ k \} \quad (= k)$$

More generally, for $k :: MonadZero a$, define ‘$k \{ b \}$’ to be:

$$\text{do } \{ a \leftarrow k; b!; \text{return } a \} = \text{do } \{ a \leftarrow k; \text{return } a \}$$

By abuse of notation, extend to assertions about multiple statements: suppose statements $s_1; \ldots; s_n$ contain generators binding variables $\nu_1, \ldots, \nu_m$; write ‘$s_1; \ldots; s_n \{ b \}$’ for

$$\text{do } \{ s_1; \ldots; s_n; b!; \text{return } (\nu_1, \ldots, \nu_m) \} = \text{do } \{ s_1; \ldots; s_n; \text{return } (\nu_1, \ldots, \nu_m) \}$$

(A similar construction is used by Erkök and Launchbury (2000).)
2.4. Queries

A special class of monadic operations, particularly amenable to manipulation.

A *query* \( q \) has no side-effects:

\[
\text{do } \{ a \leftarrow q; k \} = \text{do } \{ k \} \quad \text{-- when } k \text{ doesn’t depend on } a
\]

and is consistent:

\[
\text{do } \{ a_1 \leftarrow q; a_2 \leftarrow q; k \ a_1 \ a_2 \} = \text{do } \{ a \leftarrow q; k \ a \ a \}
\]

(They’re not just the pure operations, ie those of the form \( \text{return } a \). Consider \( \text{get :: State } s \ s \) of the state monad.)
3. A counter example

A counting monad:

```haskell
class Monad m ⇒ MonadCount m where
  tick :: m ()
  total :: m Int
```

where `total` is a query, and

```haskell
n ← total; tick; n′ ← total {n′ = n + 1}
```

(exploiting our abuse of notation).
3.1. Towers of Hanoi—specification

Given this program:

\[
\begin{align*}
\text{hanoi} &:: \text{MonadCount} m \Rightarrow \text{Int} \rightarrow m () \\
hanoi 0 & = \text{skip} \\
hanoi (n + 1) & = \text{do} \{ \text{hanoi} n; \text{tick}; \text{hanoi} n \}
\end{align*}
\]

we claim:

\[
\begin{align*}
t &\leftarrow \text{total}; \text{hanoi} n; u &\leftarrow \text{total} \{ 2^n - 1 = u - t \}
\end{align*}
\]

Proof by induction on \( n \). The base case is immediate. Inductive step...
3.2. Reasoning

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ (n+1); \ u \leftarrow \text{total}; \ (2^{n+1} - 1 = u - t)! \}
\]

\[
= \quad [\text{definition of } \text{hanoi} \quad ]
\]

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ n; \ \text{tick}; \ hanoi \ n; \ u \leftarrow \text{total}; \ (2^{n+1} - 1 = u - t)! \}
\]

\[
= \quad [\text{inserting some queries} \quad ]
\]

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ n; \ u' \leftarrow \text{total}; \ \text{tick}; \ t' \leftarrow \text{total};
\]

\[
\quad \ hanoi \ n; \ u \leftarrow \text{total}; \ (2^{n+1} - 1 = u - t)!
\]

\[
= \quad [\text{inductive hypothesis; } \text{tick} \quad ]
\]

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ n; \ u' \leftarrow \text{total}; \ (2^n - 1 = u' - t);!; \ \text{tick}; \ t' \leftarrow \text{total};
\]

\[
\quad (t' = u' + 1);!; \ hanoi \ n; \ u \leftarrow \text{total}; \ (2^n - 1 = u - t');!; (2^{n+1} - 1 = u - t)!
\]

\[
= \quad [\text{arithmetic: } 2^{n+1} - 1 = u - t \text{ follows from other guards} \quad ]
\]

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ n; \ u' \leftarrow \text{total}; \ (2^n - 1 = u' - t);!; \ \text{tick}; \ t' \leftarrow \text{total};
\]

\[
\quad (t' = u' + 1);!; \ hanoi \ n; \ u \leftarrow \text{total}; \ (2^n - 1 = u - t')!
\]

\[
= \quad [\text{redundant guards, definition of } \text{hanoi} \quad ]
\]

\[
\text{do } \{ t \leftarrow \text{total}; \ hanoi \ (n+1); \ u \leftarrow \text{total} \} \]
4. Tree relabelling

A monad for generating fresh symbols:

```haskell
type Symbol = ...
instance Eq Symbol where ...

class Monad m ⇒ MonadGensym m where
  fresh :: m Symbol
  used :: m (Set Symbol)
```

such that `used` (only used in reasoning) is a query, and

\[
x ← used; n ← fresh; y ← used \{x ⊆ y ∧ n ∈ y − x\}
\]


4.1. Specification

Tree relabelling:

\[
\textbf{data} \quad \text{Tree } a = \text{Tip } a \mid \text{Bin (Tree } a\text{) (Tree } a\text{)}
\]

\[
\text{relabel} :: \text{MonadGensym } m \Rightarrow \text{Tree } a \to m(\text{Tree Symbol})
\]

\[
\text{relabel} (\text{Leaf } a) = \text{do}\{ n \leftarrow \text{fresh}; \text{return } (\text{Leaf } n) \}
\]

\[
\text{relabel} (\text{Bin } t u) = \text{do}\{ t' \leftarrow \text{relabel } t; u' \leftarrow \text{relabel } u; \text{return } (\text{Bin } t' u') \}
\]

(in fact, an idiomatic \textit{traverse}), satisfies

\[
x \leftarrow \text{used}; t' \leftarrow \text{relabel } t; y \leftarrow \text{used}\{\text{distinct } t' \land \text{labels } t' \subseteq y - x\}
\]

where

\[
\text{distinct} :: \text{Tree Symbol} \to \text{Bool}
\]

\[
\text{labels} :: \text{Tree Symbol} \to \text{Set Symbol}
\]

(written \(d\) and \(l\) below, for short).
4.2. Reasoning: base case

\[
\begin{align*}
&\textbf{do} \{ x \leftarrow \text{used}; \nu \leftarrow \text{relabel} \ (\text{Leaf } a); y \leftarrow \text{used}; (d \ \nu \land l \ \nu \subseteq y - x)! \} \\
= &\ [[ \text{ definition of } \text{relabel} \ ]] \\
&\textbf{do} \{ x \leftarrow \text{used}; n \leftarrow \text{fresh}; \textbf{let } \nu = \text{Leaf } n; y \leftarrow \text{used}; (d \ \nu \land l \ \nu \subseteq y - x)! \} \\
= &\ [[ \text{ definition of } d, l \ ]] \\
&\textbf{do} \{ x \leftarrow \text{used}; n \leftarrow \text{fresh}; \textbf{let } \nu = \text{Leaf } n; y \leftarrow \text{used}; (\text{True} \land \{ n \} \subseteq y - x)! \} \\
= &\ [[ \text{ axiom for fresh } \ ]] \\
&\textbf{do} \{ x \leftarrow \text{used}; n \leftarrow \text{fresh}; \textbf{let } u = \text{Leaf } n; y \leftarrow \text{used} \} \\
= &\ [[ \text{ folding definitions } \ ]] \\
&\textbf{do} \{ x \leftarrow \text{used}; \nu \leftarrow \text{relabel} \ (\text{Leaf } a); y \leftarrow \text{used} \}
\end{align*}
\]
4.3. Reasoning: inductive step

\[
\text{do } \{x \leftarrow \text{used}; v \leftarrow \text{relabel} \ (\text{Bin } t \ u); z \leftarrow \text{used}; (d \ v \land l \ v \subseteq z - x)\!\} \\
= \quad [[ \text{definition of } \text{relabel} \ ]] \\
\text{do } \{x \leftarrow \text{used}; t' \leftarrow \text{relabel } t; u' \leftarrow \text{relabel } u; \textbf{let } v = \text{Bin } t' \ u'; z \leftarrow \text{used}; \\
\quad (d \ v \land l \ v \subseteq z - x)\!\} \\
= \quad [[ \text{definition of } d, l \ ]] \\
\text{do } \{x \leftarrow \text{used}; t' \leftarrow \text{relabel } t; u' \leftarrow \text{relabel } u; \textbf{let } v = \text{Bin } t' \ u'; z \leftarrow \text{used}; \\
\quad (d \ t' \land d \ u' \land l \ t' \cap l \ u' = \emptyset \land l \ t' \cup l \ u' \subseteq z - x)\!\} \\
= \quad [[ \text{induction} \ ]] \\
\text{do } \{x \leftarrow \text{used}; t' \leftarrow \text{relabel } t; y \leftarrow \text{used}; (d \ t' \land l \ t' \subseteq y - x)\!; \\
\quad u' \leftarrow \text{relabel } u; z \leftarrow \text{used}; (d \ u' \land l \ u' \subseteq z - y)\!; \textbf{let } v = \text{Bin } t' \ u'; \\
\quad (d \ t' \land d \ u' \land l \ t' \cap l \ u' = \emptyset \land l \ t' \cup l \ u' \subseteq z - x)\!\} \\
= \quad [[ \text{queries, redundant guards, folding definitions} \ ]] \\
\text{do } \{x \leftarrow \text{used}; v \leftarrow \text{relabel} \ (\text{Bin } t \ u); z \leftarrow \text{used}\}
5. Towers of Hanoi, more directly

Hoare-style reasoning is a bit painfully long-winded: repeat the program on every line, gradually discharging guards.

Sometimes a more direct approach works. In fact,

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

where

\[ rep :: Monad\ m \Rightarrow Int \rightarrow m () \rightarrow m () \]
\[ rep\ 0 \quad ma = skip \]
\[ rep\ (n + 1)\ ma = ma \gg rep\ n\ ma \]

In particular, note that

\[ rep\ (m + n)\ ma = rep\ m\ ma \gg rep\ n\ ma \]
5.1. More direct proof

...by induction on $n$. Base case is trivial. For inductive step,

$$hanoi\ (n + 1)$$

$$= \begin{array}{l}
    \quad \text{[[ definition of } hanoi \text{ ]]} \\
    hanoi\ n \gg tick \gg hanoi\ n \\
    \quad \text{[[ inductive hypothesis ]]} \\
    rep\ (2^n - 1) \gg tick \gg rep\ (2^n - 1)\ tick \\
    \quad \text{[[ composition ]]} \\
    rep\ ((2^n - 1) + 1 + (2^n - 1))\ tick \\
    \quad \text{[[ arithmetic ]]} \\
    rep\ (2^{n+1} - 1)\ tick
\end{array}$$

But I don’t see how to do tree relabelling in this more direct style...
6. Probabilistic computations

Probability distributions form a monad (Giry, Jones, Ramsey, Erwig…).
For simplicity, only finitely-supported distributions here:

```haskell
class Monad m ⇒ MonadProb m where
  choice :: Rational → m a → m a → m a
```

where the rationals are constrained to the unit interval.
Following Hoare, let’s write ‘\(mx \triangleleft p \triangleright my\)’ for ‘\(\text{choice } p \ mx \ my\)’. 
6.1. Laws of choice

Unit, idempotence, commutativity:

\[ mx \triangleleft 0 \triangleright my = my \]
\[ mx \triangleleft 1 \triangleright my = mx \]
\[ mx \triangleleft p \triangleright mx = mx \]
\[ mx \triangleleft p \triangleright my = my \triangleleft 1 - p \triangleright mx \]

A kind of associativity:

\[ mx \triangleleft p \triangleright (my \triangleleft q \triangleright mz) = (mx \triangleleft r \triangleright my) \triangleleft s \triangleright mz \]
\[ \iff p = r s \land (1 - s) = (1 - p)(1 - q) \]

Bind distributes over choice, in both directions:

\[ mx \triangleright\triangleright \lambda a \rightarrow (k_1\ a) \triangleleft p \triangleright (k_2\ a) = (mx \triangleright\triangleright k_1) \triangleleft p \triangleright (mx \triangleright\triangleright k_2) \]
\[ mx \triangleleft p \triangleright my \triangleright\triangleright k = (mx \triangleright\triangleright k) \triangleleft p \triangleright (my \triangleright\triangleright k) \]
6.2. Normal form

Finite mappings from outcomes to probabilities (ignore order, disregard weightless entries, weights sum to one, amalgamate duplicates):

```hs
newtype Distribution a = D { unD :: [(a, Rational)] }
```

All you need to interpret a distribution is \textit{choice}:

```hs
fromDist :: MonadProb m ⇒ Distribution a → m a
fromDist d = fst (foldr1 combine [ (return a, p) | (a, p) ← unD d, p > 0 ])
  where combine (mx, p) (my, q) = (mx ◁ p/p + q ◲ my, p + q)
```

For example,

```hs
uniform :: MonadProb m ⇒ [a] → m a
uniform x = fromDist (D [(a, p) | a ← x]) where p = 1 / length x
```
6.3. Implementation

Moreover, *Distribution* itself is a fine instance of *MonadProb*:

```haskell
instance Monad Distribution where
  return a = D [(a, 1)]
  px >>= f = D [(b, p \times q) | (a, p) \leftarrow \text{unD} px, (b, q) \leftarrow \text{unD} (f a)]

instance MonadProb Distribution where
  ma \triangleright p \triangleright mb = D (\text{scale} p (\text{unD} ma) + \text{scale} (1 - p) (\text{unD} mb))
  where \text{scale} r \text{pas} = [(a, r \times p) | (a, p) \leftarrow \text{pas}]
```

(Kidd points out that *Distribution* = *WriterT* Rational (*ListT* Identity), using the writer monad from the monoid of rationals with multiplication.)
6.4. Monty Hall

```haskell
data Door = A | B | C deriving (Eq, Show)
doors = [A, B, C]

hide :: MonadProb m ⇒ m Door
hide = uniform doors

pick :: MonadProb m ⇒ m Door
pick = uniform doors

tease :: MonadProb m ⇒ Door → Door → m Door
tease h p = uniform (doors \ [h, p])

switch :: MonadProb m ⇒ Door → Door → m Door
switch p t = return (head (doors \ [p, t]))

stick :: MonadProb m ⇒ Door → Door → m Door
stick p t = return p
```
6.5. The whole story

Monty’s script:

```
play :: MonadProb m ⇒ (Door → Door → m Door) → m Bool
play strategy =
  do
    h ← hide           -- host hides the car behind door h
    p ← pick           -- you pick door p
    t ← tease h p       -- host teases you with door t (≠ h, p)
    s ← strategy p t    -- you choose, based on p and t (but not h!)
    return (s == h)     -- you win iff your choice s equals h
```
6.6. In support of Marilyn Vos Savant

It’s a straightforward proof by equational reasoning that

\[
\text{play switch} = \text{uniform} [ \text{True, True, False} ]
\]
\[
\text{play stick} = \text{uniform} [ \text{False, False, True} ]
\]

The key is that separate uniform distributions are independent:

\[
\text{do } \{ a \leftarrow \text{uniform } x; b \leftarrow \text{uniform } y; \text{return } (a, b) \} = \text{uniform } (\text{cp } x \ y)
\]

where

\[
\text{cp} :: [a] \to [b] \to [(a, b)]
\]
\[
\text{cp } x \ y = [(a, b) | a \leftarrow x, b \leftarrow y]
\]

(Ask me over a beer...)
7. Combining probability and nondeterminism

Nobody said that Monty has to play fair. He has a free choice in hiding the car, and in teasing you.

To model this, we need to combine probabilism with nondeterminism:

```
class MonadZero m ⇒ MonadPlus m where
  mplus :: m a → m a → m a
```

such that \texttt{mzero} and \texttt{mplus} form a monoid, and

\[(m \texttt{‘mplus‘ } n) \gg k = (m \gg k) \texttt{‘mplus‘ } (n \gg k)\]

Happily, although monads do not compose in general, \([\texttt{Distribution } a]\) is a monad. Moreover, it is a \texttt{MonadProb} and a \texttt{MonadPlus} too.

(So is \texttt{Distribution [a]}, but I think that doesn’t help.)

(There’s a nice tale in terms of monad transformers.)
7.1. A simple example: mixing choices

A fair coin:

\[
\text{coin} :: \text{MonadProb } m \Rightarrow m \text{ Bool} \\
\text{coin} = (\text{return True}) \triangleleft \frac{1}{2} \triangleright (\text{return False})
\]

An arbitrary choice:

\[
\text{arb} :: \text{MonadPlus } m \Rightarrow m \text{ Bool} \\
\text{arb} = \text{return True} \text{ `mplus` return False}
\]

Two combinations:

\[
\text{arbcoin, coinarb} :: (\text{MonadPlus } m, \text{MonadProb } m) \Rightarrow m \text{ Bool} \\
\text{arbcoin} = \text{do}\{ a \leftarrow \text{arb}; c \leftarrow \text{coin}; \text{return} (a :: c) \} \\
\text{coinarb} = \text{do}\{ c \leftarrow \text{coin}; a \leftarrow \text{arb}; \text{return} (a :: c) \}
\]

What do you think they do?
7.2. ... as sets of distributions

Define

\[
\text{type} \ NondetProb \ a = [\ Distribution \ a ]
\]

Then (with suitable \textit{shows}):

\begin{align*}
* \text{Main)} \ arbcoin & :: \ NondetProb \ Bool \\
& \equiv [[[ (True, \frac{1}{2}), (False, \frac{1}{2}) ]], \\
& \quad [[[ (False, \frac{1}{2}), (True, \frac{1}{2}) ]]] \\
* \text{Main)} \ coinarb & :: \ NondetProb \ Bool \\
& \equiv [[[ (True, \frac{1}{2}), (False, \frac{1}{2}) ]], \\
& \quad [[[ (True, \frac{1}{2}), (True, \frac{1}{2}) ]], \\
& \quad [[[ (False, \frac{1}{2}), (False, \frac{1}{2}) ]], \\
& \quad [[[ (False, \frac{1}{2}), (True, \frac{1}{2}) ]]]
\end{align*}
7.3. ... as expectations

```haskell
class MonadProb m ⇒ MonadExpect m where
  expect :: (Ord n, Fractional n) ⇒ m a → (a → n) → n

instance MonadExpect NondetProb where  -- morally
  expect px h = minimum (map (mean h ◦ unD) px) where
    mean h aps = sum [ p × f a | (a, p) ← aps ] / sum (map snd aps)
```

Your reward is 1 if the booleans agree, and 0 otherwise:

```haskell
reward b = if b then 1 else 0
```

Then:

```haskell
*Main> expect (arbcoin :: NondetProb Bool) reward
1/2
*Main> expect (coinarb :: NondetProb Bool) reward
0
```
7.4. Back to nondeterministic Monty...

We could define instead:

\[
\begin{align*}
hide & :: \text{MonadPlus } m \Rightarrow m \text{ Door} \\
hide &= \text{arbitrary doors} \\
tease & :: \text{MonadPlus } m \Rightarrow \text{Door} \rightarrow \text{Door} \rightarrow m \text{ Door} \\
tease \ h \ p &= \text{arbitrary } (\text{doors} \ \text{\setminus} \ [h, p])
\end{align*}
\]

where

\[
\begin{align*}
arbitrary & :: \text{MonadPlus } m \Rightarrow [a] \rightarrow m\ a \\
arbitrary &= \text{foldr mplus mzero } \circ \text{ map return}
\end{align*}
\]

I believe that the calculation carries through just as before: still

\[
\begin{align*}
\text{play switch} &= \text{uniform } [\text{True, True, False}] \\
\text{play stick} &= \text{uniform } [\text{False, False, True}]
\end{align*}
\]
8. Summary

- axiomatic approach to reasoning with effects
- simple and generic
- smacks of ‘algebraic theories of effects’ (Plotkin & Power, Lawvere) (in particular, partiality and continuations do not arise from algebraic theories)
- IO is uninteresting?
- more examples wanted!