Unifying Theories of Programming with Monads

Jeremy Gibbons

UTP 2012, Aug 2012

Unifying Theories of Programming with Monads

Jeremy Gibbons

UTP 2012, Aug 2012

Unifying Theories of Programming with Monads

Jeremy Gibbons

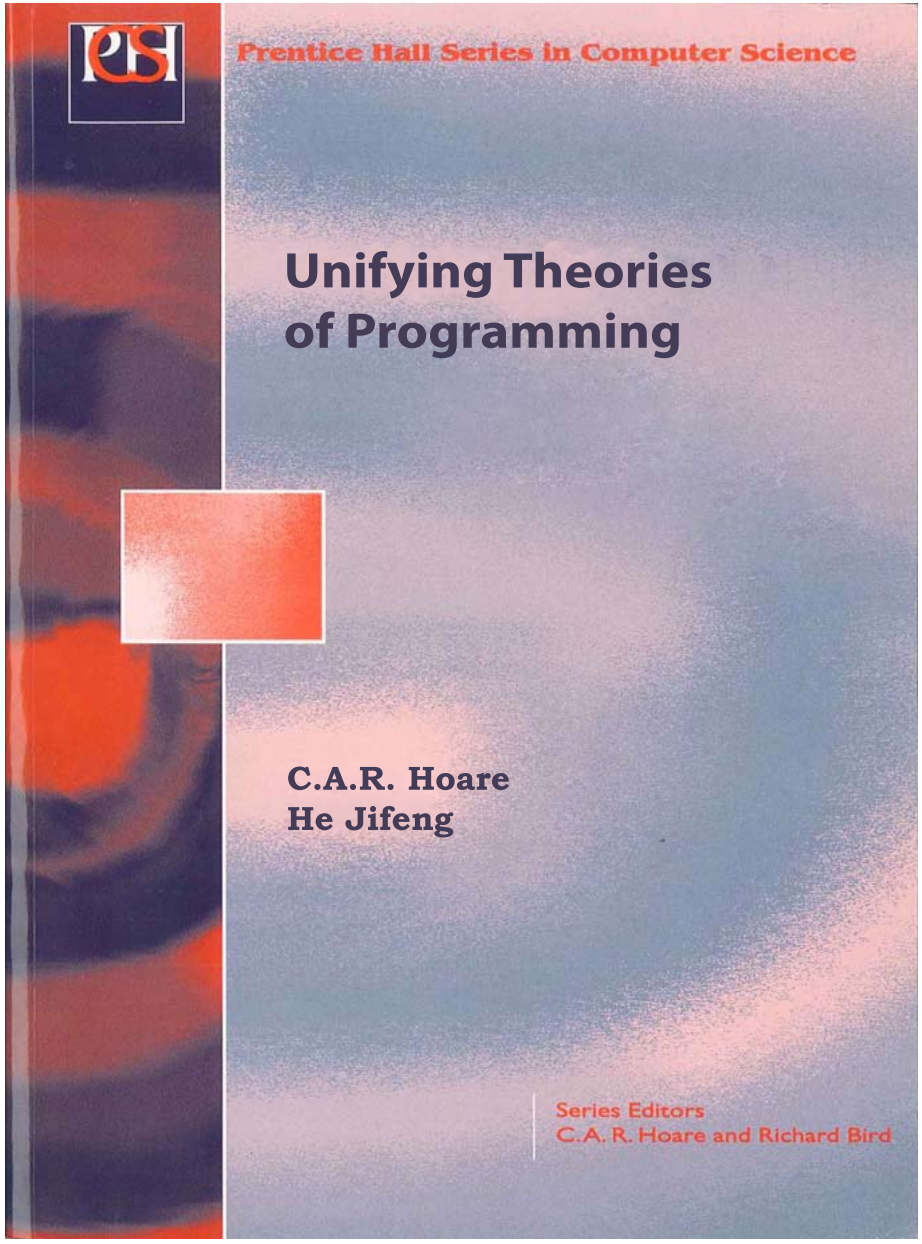
UTP 2012, Aug 2012

1. Unifying theories

Unification of paradigms: imperative, nondeterministic, concurrent, reactive, higher-order, etc.

Programs are predicates: a single domain of discourse, rather than separating syntax from semantics.

A streamlined vehicle for reasoning about different styles of computation.



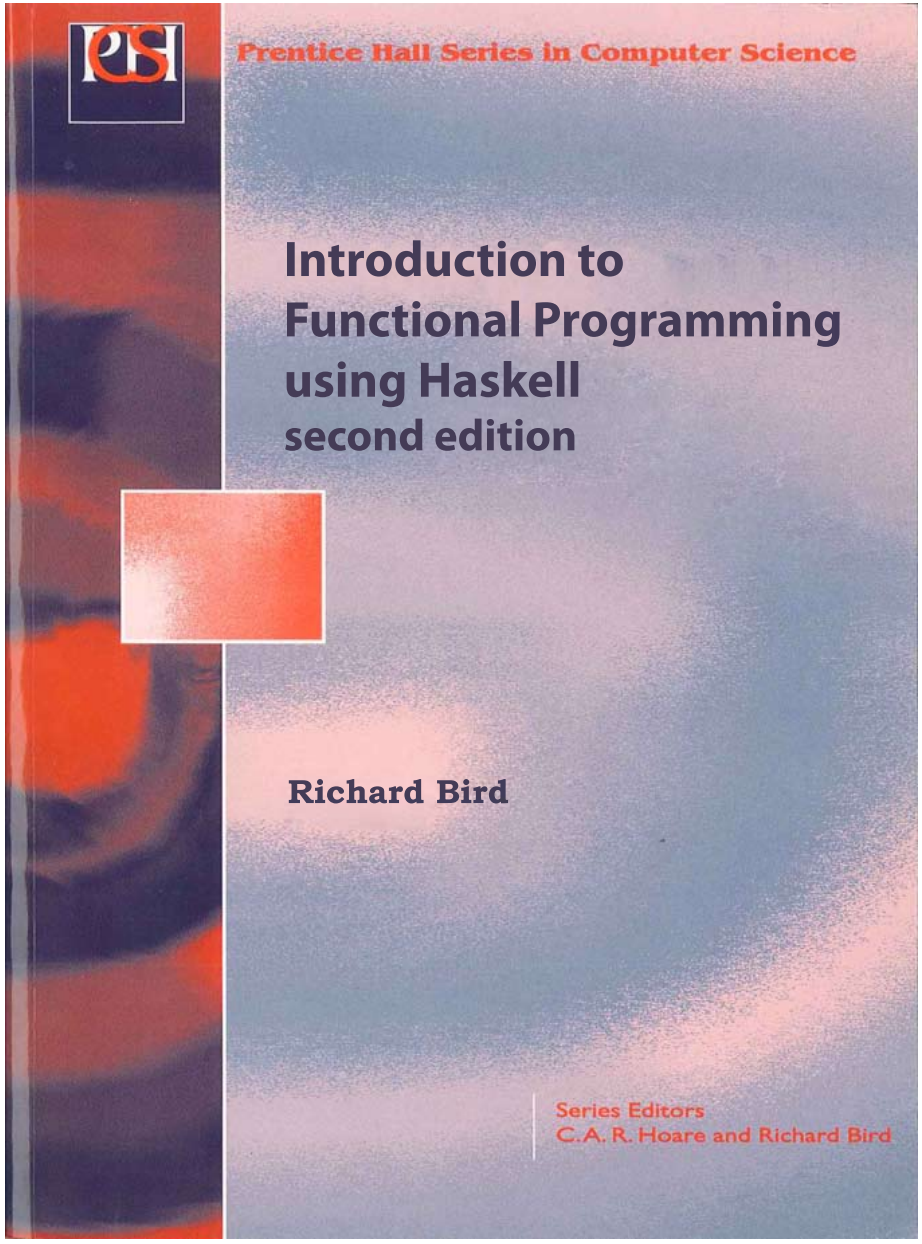
1.1. Alternatively...

Functional programming is another streamlined vehicle for reasoning about computation.

- *simple*: programming with expressions
- *natural*: substituting equals for equals
- *abstract*: machine-independent
- *powerful*: concise and expressive

Also eliding the distinction between syntax and semantics.

But programs are *functions* rather than *predicates*.



1.2. This talk

- functional programming
- equational reasoning
- computational effects via monads
- reasoning about effectful programs
- state, nondeterminism, probability...

2. Functional programming, in a nutshell

Fold pattern of computation on lists:

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ e\ [\,] \quad\ = e \\
&\mathit{foldr}\ f\ e\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)
\end{aligned}
$$

Hence, summing a list of numbers:

$$
\begin{aligned}
&\mathit{sum} :: \mathit{Num}\ a \Rightarrow [a] \to a \\
&\mathit{sum} = \mathit{foldr}\ (+)\ 0
\end{aligned}
$$

(NB: type classes for ad hoc polymorphism...)

Also, *map* operation on lists:

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to [a] \to [b] \\
&\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ ys \to f\ x : ys)\ [\,]
\end{aligned}
$$

2.1. Evaluation

$$
\begin{aligned}
& \mathit{sum}\ (1 : 2 : 3 : [\,]) \\
= \quad & [\![\ \text{definition of } \mathit{sum}\ ]\!] \\
& \mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,]) \\
= \quad & [\![\ \mathit{foldr}\text{, second clause: } \mathit{foldr}\ f\ e\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)\ ]\!] \\
& 1 + \mathit{foldr}\ (+)\ 0\ (2 : 3 : [\,]) \\
= \quad & [\![\ \ldots \text{and again}\ ]\!] \\
& 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3 : [\,])) \\
= \quad & [\![\ \ldots \text{and again}\ ]\!] \\
& 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,])) \\
= \quad & [\![\ \mathit{foldr}\text{, first clause: } \mathit{foldr}\ f\ e\ [\,] = e\ ]\!] \\
& 1 + (2 + (3 + 0)) \\
= \quad & [\![\ \text{arithmetic}\ ]\!] \\
& 6
\end{aligned}
$$

Equational reasoning, using definitions as equations.

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. The separation of Church and state

Functional programming as a model of computation:

- simple
- natural
- abstract
- expressive

Strengths derived from purity—

- referential transparency
- no mutable variables, or other side-effects
- programming language is simultaneously reasoning language

But how to combine these strengths with actual useful effects?
(state, nondeterminism, exceptions, I/O...)

3.1. Monads

An interface for effectful computation:

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\ &\quad \mathit{return} :: a \to m\ a\\ &\quad (\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

Unit and associativity laws:

$$\begin{aligned}\mathit{return}\ x \gg\!= k &= k\ x\\ p \gg\!= \mathit{return} &= p\\ (p \gg\!= k) \gg\!= k' &= p \gg\!= (\lambda x \to k\ x \gg\!= k')\end{aligned}$$

Moggi (1989), Wadler (1992).

3.2. State benefits

Subclasses of *Monad* for particular computational effects.

Eg a subclass of computations that supports mutable state:

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

Four axioms should be satisfied:

```
put s >>= λ() → put s'   = put s'                    -- put–put
put s >>= λ() → get      = put s >>= λ() → return s  -- put–get
get >>= put              = return ()                 -- get–put
get >>= λs → get >>= k s = get >>= λs → k s s        -- get–get
```

3.3. Two shorthands

Define

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

Then state axioms become less noisy:

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

3.4. Imperative functional programming

Monad *comprehensions*, via Haskell's '**do**' notation:

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

So instead of defining $incrodd :: MonadState\ Integer\ m \Rightarrow m\ Bool$ by

$$incrodd = get \gg\!= (\lambda n \rightarrow put\ (n+1) \gg return\ (odd\ n))$$

we can write

$$incrodd = \mathbf{do}\ \{n \leftarrow get\,;\, put\ (n+1)\,;\, return\ (odd\ n)\}$$

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

3.5. Simulating state

Simulate mutable state via state-transforming functions—roughly,

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

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

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

Of course, state-transforming functions simulate mutable state:

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

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

4. Equational reasoning with effects

Augmenting an integer state:

$$add :: MonadState\ Integer\ m \Rightarrow Integer \to m\ ()$$
$$add\ n = \mathbf{do}\ \{ m \leftarrow get\ ;\ put\ (m + n) \}$$

Then augmenting multiple times, via

$$addAll :: MonadState\ Integer\ m \Rightarrow [\ Integer\ ] \to m\ ()$$
$$addAll = sequence\_ \circ map\ add$$

is equivalent to augmenting by the sum:

$$addAll = add \circ sum$$

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

$$sequence\_ :: Monad\ m \Rightarrow [\ m\ ()\ ] \to m\ ()$$
$$sequence\_ = foldr\ (\gg)\ skip$$

4.1. Using fusion

Because *sequence_* is a fold, we can use fold–map fusion:

$$\begin{aligned} &addAll = foldr\ addThen\ skip \\ &\quad \textbf{where}\ addThen\ n\ p = \textbf{do}\ \{add\ n\,;\,p\} \end{aligned}$$

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

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

So let's prove these...

4.2. Adding zero

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

4.3. Adding a sum

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

QED.

5. Nondeterminism

A subclass of computations that supports choice:

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadAlt}\ m \textbf{ where}\\ &\quad (\Box) :: m\ a \to m\ a \to m\ a\end{aligned}$$

We stipulate that $\Box$ is associative, commutative, and idempotent:

$$\begin{aligned}(p \Box q) \Box r &= p \Box (q \Box r)\\ p \Box q &= q \Box p\\ p \Box p &= p\end{aligned}$$

5.1. Nondeterminism and composition

We stipulate that composition distributes *leftwards* over $\Box$:

$$(p \,\Box\, q) \gg\!= k \quad = \quad (p \gg\!= k) \,\Box\, (q \gg\!= k)$$

that is,

$$\mathbf{do}\,\{x \leftarrow (p \,\Box\, q)\,;\, k\,x\} \quad = \quad \mathbf{do}\,\{x \leftarrow p\,;\, k\,x\} \,\Box\, \mathbf{do}\,\{x \leftarrow q\,;\, k\,x\}$$

But in general, composition does not distribute *rightwards* over $\Box$:

$$p \gg\!= (\lambda x \to k_1\ x \,\Box\, k_2\ x) \quad \neq \quad (p \gg\!= k_1) \,\Box\, (p \gg\!= k_2)$$

that is,

$$\mathbf{do}\,\{x \leftarrow p\,;\, (k_1\ x \,\Box\, k_2\ x)\} \quad \neq \quad \mathbf{do}\,\{x \leftarrow p\,;\, k_1\ x\} \,\Box\, \mathbf{do}\,\{x \leftarrow p\,;\, k_2\ x\}$$

In some models (eg angelic nondeterminism), both branches are executed. Then any effects of p happen once on the left, and twice on the right.

5.2. Implementing choice

On account of the axioms, initial semantics is set-like.
We might approximate this using (non-empty) lists:

```
instance Monad [ ] where
  return a = [a]
  p >>= k  = concat (map k p)
instance MonadAlt [ ] where
  (□) = (++)
```

but consider set-equivalence (ignoring multiplicity and ordering).

(Actually, none of associativity, commutativity, idempotence is important.)

5.3. Nondeterministic subsequences

Nondeterministically select some subsequence of elements from a list:

$$
\begin{aligned}
&\mathit{subs} :: \mathit{MonadAlt}\ m \Rightarrow [a] \to m\,[a] \\
&\mathit{subs}\ [\,] \qquad = \mathit{return}\ [\,] \\
&\mathit{subs}\ (x : xs) = \mathit{fmap}\ (x:)\ \mathit{xss} \Box \mathit{xss} \quad \textbf{where}\ \mathit{xss} = \mathit{subs}\ \mathit{xs}
\end{aligned}
$$

We might wish to prove that *subs* distributes over $+\!\!+$:

$$\mathit{subs}\ (\mathit{xs} +\!\!+ \mathit{ys}) = \textbf{do}\ \{\mathit{us} \leftarrow \mathit{subs}\ \mathit{xs}\,;\, \mathit{vs} \leftarrow \mathit{subs}\ \mathit{ys}\,;\, \mathit{return}\ (\mathit{us} +\!\!+ \mathit{vs})\}$$

By induction over *xs*, using plain ordinary equational reasoning...

5.4. Base case

$$
\begin{aligned}
& \mathbf{do}\,\{us \leftarrow subs\,[\,]\,;\, vs \leftarrow subs\,ys\,;\, return\,(us +\!\!+ vs)\,\} \\
= \quad & [\![\ \text{definition of } subs\ ]\!] \\
& \mathbf{do}\,\{us \leftarrow return\,[\,]\,;\, vs \leftarrow subs\,ys\,;\, return\,(us +\!\!+ vs)\,\} \\
= \quad & [\![\ \text{left unit}\ ]\!] \\
& \mathbf{do}\,\{vs \leftarrow subs\,ys\,;\, return\,([\,] +\!\!+ vs)\,\} \\
= \quad & [\![\ \text{definition of } +\!\!+\ ]\!] \\
& \mathbf{do}\,\{vs \leftarrow subs\,ys\,;\, return\,vs\} \\
= \quad & [\![\ \text{right unit}\ ]\!] \\
& subs\,ys \\
= \quad & [\![\ \text{by assumption, } xs = [\,]\ ]\!] \\
& subs\,(xs +\!\!+ ys)
\end{aligned}
$$

5.5. Inductive step

$$
\begin{aligned}
& \mathbf{do}\ \{us \leftarrow subs\ (x : xs)\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \\
= \quad & [\![\ \text{definition of } subs\text{; let } xss = subs\ xs \quad]\!] \\
& \mathbf{do}\ \{us \leftarrow (fmap\ (x:)\ xss \,\square\, xss)\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \\
= \quad & [\![\ \text{composition distributes leftwards over } \square \quad]\!] \\
& \mathbf{do}\ \{us \leftarrow fmap\ (x:)\ xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \,\square \\
& \mathbf{do}\ \{us \leftarrow xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \\
= \quad & [\![\ fmap \text{ and } \mathbf{do} \text{ notation} \quad]\!] \\
& \mathbf{do}\ \{us' \leftarrow xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ ((x : us') +\!\!+ vs)\} \,\square \\
& \mathbf{do}\ \{us \leftarrow xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \\
= \quad & [\![\ \text{definition of } +\!\!+\text{; } \mathbf{do} \text{ notation} \quad]\!] \\
& fmap\ (x:)\ (\mathbf{do}\ \{us' \leftarrow xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us' +\!\!+ vs)\})\ \square \\
& \mathbf{do}\ \{us \leftarrow xss\ ;\ vs \leftarrow subs\ ys\ ;\ return\ (us +\!\!+ vs)\} \\
= \quad & [\![\ \ldots \text{continued} \quad]\!]
\end{aligned}
$$

5.6. Inductive step (continued)

$$
\begin{aligned}
&= \quad [\![\ \ldots \text{continued}\ \ ]\!] \\
&\quad \mathit{fmap}\ (x{:})\ (\mathbf{do}\ \{\mathit{us}' \leftarrow \mathit{xss}\,;\mathit{vs} \leftarrow \mathit{subs}\ \mathit{ys}\,;\mathit{return}\ (\mathit{us}' +\!\!+ \mathit{vs})\})\ \square \\
&\quad \mathbf{do}\ \{\mathit{us} \leftarrow \mathit{xss}\,;\mathit{vs} \leftarrow \mathit{subs}\ \mathit{ys}\,;\mathit{return}\ (\mathit{us} +\!\!+ \mathit{vs})\} \\
&= \quad [\![\ \text{by assumption, } \mathit{xss} = \mathit{subs}\ \mathit{xs}\text{; inductive hypothesis, twice}\ \ ]\!] \\
&\quad \mathit{fmap}\ (x{:})\ (\mathit{subs}\ (\mathit{xs} +\!\!+ \mathit{ys}))\ \square\ \mathit{subs}\ (\mathit{xs} +\!\!+ \mathit{ys}) \\
&= \quad [\![\ \text{definition of } \mathit{subs}\ \ ]\!] \\
&\quad \mathit{subs}\ (x : (\mathit{xs} +\!\!+ \mathit{ys})) \\
&= \quad [\![\ \text{definition of } +\!\!+\ \ ]\!] \\
&\quad \mathit{subs}\ ((x : \mathit{xs}) +\!\!+ \mathit{ys})
\end{aligned}
$$

QED.

6. Probabilistic computations

Probability distributions form a monad.

For simplicity, only finitely-supported distributions here:

$$\textbf{type}\ \mathit{Prob} = \mathit{Rational} \quad \text{-- with unit range}$$

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadProb}\ m\ \textbf{where}$$

$$(\cdot \lhd \cdot \rhd \cdot) :: (m\ a, \mathit{Prob}, m\ a) \to m\ a$$

For example,

$$\mathit{uniform} :: \mathit{MonadProb}\ m \Rightarrow [a] \to m\ a \quad \text{-- nonempty lists}$$

$$\mathit{uniform}\ [x] = \mathit{return}\ x$$

$$\mathit{uniform}\ (x : xs) = \mathit{return}\ x \lhd {}^{1}/_{\mathit{length}\ (x:xs)} \rhd \mathit{uniform}\ xs$$

Lawvere (1962), Giry (1981), Jones (1989)...

6.1. Laws of probability

Unit, idempotence, quasi-commutativity:

$$p \lhd 0 \rhd q = q$$
$$p \lhd 1 \rhd q = p$$
$$p \lhd w \rhd p = p$$
$$p \lhd w \rhd q = q \lhd 1-w \rhd p$$

Also quasi-associativity:

$$p \lhd u \rhd (q \lhd v \rhd r) = (p \lhd w \rhd q) \lhd x \rhd r$$
$$\Longleftarrow \quad u = w\,x \;\wedge\; (1-x) = (1-u)\,(1-v)$$

Composition distributes over choice, in both directions:

$$\mathbf{do}\,\{x \leftarrow (p \lhd w \rhd q)\,;\,k\,x\} = (\mathbf{do}\,\{x \leftarrow p\,;\,k\,x\}) \lhd w \rhd (\mathbf{do}\,\{x \leftarrow q\,;\,k\,x\})$$
$$\mathbf{do}\,\{x \leftarrow p\,;\,(k_1\,x \lhd w \rhd k_2\,x)\} = (\mathbf{do}\,\{x \leftarrow p\,;\,k_1\,x\}) \lhd w \rhd (\mathbf{do}\,\{x \leftarrow p\,;\,k_2\,x\})$$

6.2. Representing probability distributions

Free monad for the signature of choice:

$$\textbf{data}\ \mathit{Dist}\ a = \mathit{Return}\ a \mid \mathit{Choice}\ \mathit{Prob}\ (\mathit{Dist}\ a)\ (\mathit{Dist}\ a)$$

(modulo a suitable equivalence re associativity etc).

This being a free monad, *return* and $\gg\!=$ arise from substitution:

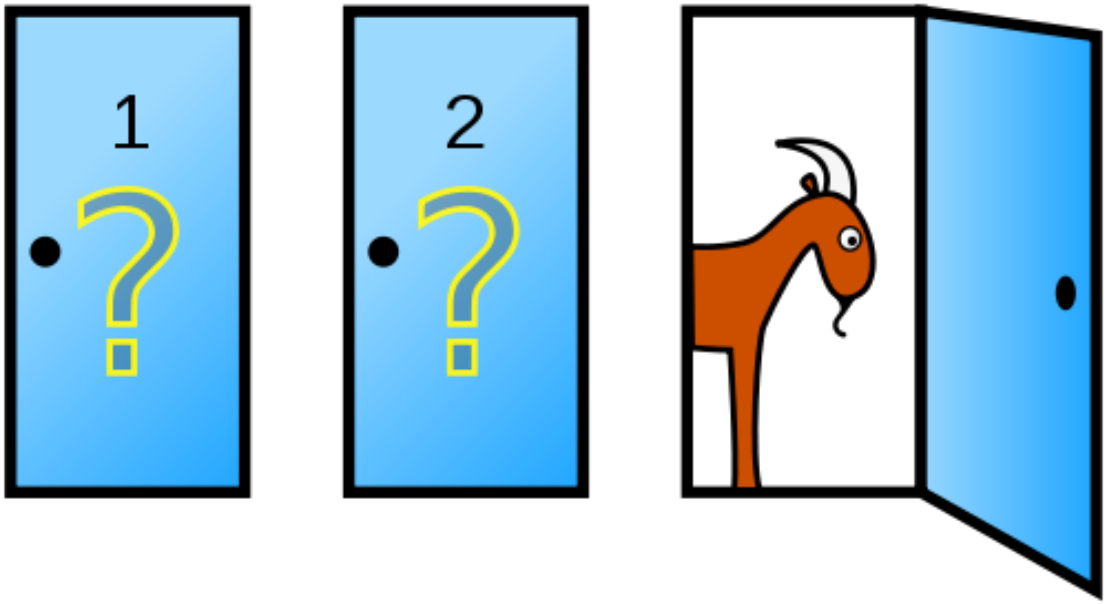
$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Dist}\ \textbf{where}\\
&\quad \mathit{return}\ x \qquad\qquad = \mathit{Return}\ x\\
&\quad \mathit{Return}\ x \gg\!= k \qquad = k\ x\\
&\quad \mathit{Choice}\ w\ p\ q \gg\!= k = \mathit{Choice}\ w\ (p \gg\!= k)\ (q \gg\!= k)
\end{aligned}$$

And by design, $\triangleleft\triangleright$ is trivial to implement:

$$\begin{aligned}
&\textbf{instance}\ \mathit{MonadProb}\ \mathit{Dist}\ \textbf{where}\\
&\quad p \triangleleft w \triangleright q = \mathit{Choice}\ w\ p\ q
\end{aligned}$$

6.3. Monty Hall

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?



6.4. The actions

```
data Door = A | B | C
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

$$\begin{aligned} &play\ switch = uniform\ [\,True, True, False\,] \\ &play\ stick\ \ = uniform\ [\,False, False, True\,] \end{aligned}$$

The key is that separate uniform distributions are independent:

$$\mathbf{do}\ \{a \leftarrow uniform\ x\,;\, b \leftarrow uniform\ y\,;\, return\ (a,b)\} = uniform\ (cp\ x\ y)$$

where

$$\begin{aligned} &cp :: [\,a\,] \to [\,b\,] \to [\,(a,b)\,] \\ &cp\ x\ y = [\,(a,b) \mid a \leftarrow x, b \leftarrow y\,] \end{aligned}$$

For example...

6.7. Equational reasoning about probability

$$
\begin{aligned}
& \mathit{play}\ \mathit{stick} \\
= \quad & [\![\ \text{definition of } \mathit{play}\text{; independent uniform choices}\ ]\!] \\
& \mathbf{do}\ \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\ ; \\
& \qquad t \leftarrow \mathit{tease}\ h\ p\ ;\ s \leftarrow \mathit{stick}\ p\ t\ ;\ \mathit{return}\ (s == h)\ \} \\
= \quad & [\![\ \mathit{stick}\ p\ t = \mathit{return}\ p\ ]\!] \\
& \mathbf{do}\ \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\ ;\ t \leftarrow \mathit{tease}\ h\ p\ ;\ \mathit{return}\ (p == h)\ \} \\
= \quad & [\![\ t \text{ unused, and } \mathit{uniform} \text{ side-effect-free}\ ]\!] \\
& \mathbf{do}\ \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\ ;\ \mathit{return}\ (p == h)\ \} \\
= \quad & [\![\ \text{definition of } \mathit{fmap}\text{; naturality of } \mathit{uniform}\ ]\!] \\
& \mathit{uniform}\ (\mathit{map}\ (\mathit{uncurry}\ (==))\ (\mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors}))) \\
= \quad & [\![\ \text{definition of } \mathit{cp}, ==, \mathit{map}\ ]\!] \\
& \mathit{uniform}\ [\mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}] \\
= \quad & [\![\ \text{matching choices: three } \mathit{True}\text{s, six } \mathit{False}\text{s}\ ]\!] \\
& \mathit{uniform}\ [\mathit{True}, \mathit{False}, \mathit{False}]
\end{aligned}
$$

7. Combining probability and nondeterminism

Nobody said that Monty has to play fair—it's his game show.
He has a free choice in hiding the car, and in teasing you.

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

$$\textbf{class}\ (\mathit{MonadAlt}\ m, \mathit{MonadProb}\ m) \Rightarrow \mathit{MonadAltProb}\ m$$

No new operations; but one additional law—
probabilistic choice distributes over nondeterministic:

$$p \lhd w \rhd (q \,\Box\, r) = (p \lhd w \rhd q) \,\Box\, (p \lhd w \rhd r)$$

7.1. A simple example: mixing choices

An arbitrary choice:

$$arb :: MonadAlt\ m \Rightarrow m\ Bool$$
$$arb = return\ True \Box return\ False$$

A fair coin:

$$coin :: MonadProb\ m \Rightarrow m\ Bool$$
$$coin = return\ True \lhd {}^{1}/_{2} \rhd return\ False$$

Two combinations:

$$arbcoin, coinarb :: MonadAltProb\ m \Rightarrow m\ Bool$$
$$arbcoin = \mathbf{do}\ \{a \leftarrow arb\,;\, c \leftarrow coin\,;\, return\ (a == c)\}$$
$$coinarb = \mathbf{do}\ \{c \leftarrow coin\,;\, a \leftarrow arb\,;\, return\ (a == c)\}$$

In *coinarb*, the nondeterministic choice can depend on the probabilistic; in *arbcoin*, it cannot. . .

7.2. Outcomes of *arbcoin*

$$
\begin{array}{ll}
 & \mathit{arbcoin} \\
= & \quad [\![\text{ definition of } \mathit{arbcoin} \text{ }]\!] \\
 & \mathbf{do}\,\{a \leftarrow \mathit{arb}\,;\, c \leftarrow \mathit{coin}\,;\, \mathit{return}\,(a == c)\} \\
= & \quad [\![\text{ definition of } \mathit{arb} \text{ }]\!] \\
 & \mathbf{do}\,\{a \leftarrow (\mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False})\,;\, c \leftarrow \mathit{coin}\,;\, \mathit{return}\,(a == c)\} \\
= & \quad [\![\text{ composition distributes leftwards over } \Box\text{; left unit }]\!] \\
 & \mathbf{do}\,\{c \leftarrow \mathit{coin}\,;\, \mathit{return}\ c\} \,\Box\, \mathbf{do}\,\{c \leftarrow \mathit{coin}\,;\, \mathit{return}\,(\neg\ c)\} \\
= & \quad [\![\text{ right unit; definition of } \mathit{coin} \text{ }]\!] \\
 & \mathit{coin} \,\Box\, \mathbf{do}\,\{\mathit{return}\ \mathit{False} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{True}\} \\
= & \quad [\![\text{ commutativity of } \lhd\rhd\text{; definition of } \mathit{coin} \text{ }]\!] \\
 & \mathit{coin} \,\Box\, \mathit{coin} \\
= & \quad [\![\text{ idempotence of } \Box \text{ }]\!] \\
 & \mathit{coin}
\end{array}
$$

7.3. Outcomes of *coinarb*

$$
\begin{aligned}
& \mathit{coinarb} \\
= \quad & [\![\text{ definitions of } \mathit{coinarb} \text{ and } \mathit{coin} \ ]\!] \\
& \mathbf{do}\ \{c \leftarrow (\mathit{return}\ \mathit{True} \lhd 1/2 \rhd \mathit{return}\ \mathit{False})\ ;\ a \leftarrow \mathit{arb}\ ;\ \mathit{return}\ (a == c)\ \} \\
= \quad & [\![\text{ composition distributes leftwards over } \lhd\rhd\text{; left unit }]\!] \\
& (\mathbf{do}\ \{a \leftarrow \mathit{arb}\ ;\ \mathit{return}\ a\}) \lhd 1/2 \rhd (\mathbf{do}\ \{a \leftarrow \mathit{arb}\ ;\ \mathit{return}\ (\neg\ a)\ \}) \\
= \quad & [\![\text{ right unit; definition of } \mathit{arb} \ ]\!] \\
& (\mathit{return}\ \mathit{True} \sqcap \mathit{return}\ \mathit{False}) \lhd 1/2 \rhd (\mathit{return}\ \mathit{False} \sqcap \mathit{return}\ \mathit{True}) \\
= \quad & [\![\text{ } \lhd\rhd \text{ distributes over } \sqcap\text{, three times }]\!] \\
& (\mathit{return}\ \mathit{True} \lhd 1/2 \rhd \mathit{return}\ \mathit{False}) \sqcap (\mathit{return}\ \mathit{False} \lhd 1/2 \rhd \mathit{return}\ \mathit{False}) \sqcap \\
& (\mathit{return}\ \mathit{True} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}) \sqcap (\mathit{return}\ \mathit{False} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}) \\
= \quad & [\![\text{ commutativity and idempotence of } \lhd\rhd\text{; definition of } \mathit{coin} \ ]\!] \\
& \mathit{coin} \sqcap \mathit{return}\ \mathit{False} \sqcap \mathit{return}\ \mathit{True} \sqcap \mathit{coin} \\
= \quad & [\![\text{ associativity, commutativity and idempotence of } \sqcap \ ]\!] \\
& \mathit{return}\ \mathit{False} \sqcap \mathit{coin} \sqcap \mathit{return}\ \mathit{True}
\end{aligned}
$$

7.4. Representing probability and nondeterminism

As ‘sets’ (nonempty lists) of distributions:

```
newtype Dists a = Ds {unDs :: [Dist a]}    -- nonempty lists
```

Given the appropriate distributive law of distributions over sets

```
swap :: Dist [a] → [Dist a]    -- nonempty lists
swap (Return xs)          = [Return x | x ← xs]
swap (Choice w ps qs) = [Choice w p q | p ← swap ps, q ← swap qs]
```

the monad instance is mechanical:

```
instance Monad Dists where
  return x       = Ds (return (return x))
  Ds xds >>= k = Ds (xds >>= map (>>=id) ∘ swap ∘ fmap (unDs ∘ k))
```

7.5. . . . continued

The *MonadAlt* and *MonadProb* instances are simply lifted:

```
instance MonadAlt Dists where
  Ds xds □ Ds yds = Ds (xds □ yds)

instance MonadProb Dists where
  (Ds xds) ◁ w ▷ (Ds yds) = Ds [xd ◁ w ▷ yd | xd ← xds, yd ← yds]
```

7.6. Convexity

Actually, sets of distributions doesn't quite work: there is no distributive law of distributions over sets (Plotkin), so the composition of the two monads isn't a monad.

Indeed, distribution of $\triangleleft\triangleright$ over $\Box$ and idempotence of $\triangleleft\triangleright$ imply a convexity property:

$$
\begin{array}{ll}
 & p \Box q \\
= & \quad [\![\text{ idempotence of } \triangleleft\triangleright \ ]\!] \\
 & (p \Box q) \triangleleft w \triangleright (p \Box q) \\
= & \quad [\![\text{ distributing } \triangleleft\triangleright \text{ over } \Box \ ]\!] \\
 & (p \triangleleft w \triangleright p) \Box (q \triangleleft w \triangleright p) \Box (p \triangleleft w \triangleright q) \Box (q \triangleleft w \triangleright q)
\end{array}
$$

This is computationally reasonable, if you consider repeated executions.

As a consequence, we should consider equivalence of sets of distributions *up to convex closure*.

7.7. Back to nondeterministic Monty...

We could define instead:

$$
\begin{aligned}
&hide :: MonadAlt\ m \Rightarrow m\ Door \\
&hide = arbitrary\ doors \\
&tease :: MonadAlt\ m \Rightarrow Door \to Door \to m\ Door \\
&tease\ h\ p = arbitrary\ (doors \setminus\setminus [h, p])
\end{aligned}
$$

where

$$
\begin{aligned}
&arbitrary :: MonadAlt\ m \Rightarrow [a] \to m\ a \quad \text{-- nonempty lists} \\
&arbitrary = foldr_1\ (\Box) \circ map\ return
\end{aligned}
$$

The calculation carries through just as before:

$$
\begin{aligned}
&play\ switch = uniform\ [True, True, False] \\
&play\ stick\ \ = uniform\ [False, False, True]
\end{aligned}
$$

8. Angels and demons

Computations with both angelic choice ($\sqcup$) and demonic choice ($\sqcap$):

$$\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadAngDem}\ m \textbf{ where}$$
$$(\sqcup), (\sqcap) :: m\ a \to m\ a \to m\ a$$

Associative, commutative, idempotent, distributive:

$$\begin{array}{ll} (p \sqcup q) \sqcup r = p \sqcup (q \sqcup r) & (p \sqcap q) \sqcap r = p \sqcap (q \sqcap r) \\ p \sqcup q = q \sqcup p & p \sqcap q = q \sqcap p \\ p \sqcup p = p & p \sqcap p = p \\ (p \sqcup q) \gg\!= k = (p \gg\!= k) \sqcup (q \gg\!= k) & (p \sqcap q) \gg\!= k = (p \gg\!= k) \sqcap (q \gg\!= k) \end{array}$$

Moreover, each distributes over the other:

$$p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r) \qquad p \sqcap (q \sqcup r) = (p \sqcup q) \sqcap (p \sqcup r)$$

(in contrast to probabilistic and nondeterministic choice...)

8.1. Normal form

Any term built from *return*, $\gg=$, $\sqcup$ and $\sqcap$ can be reduced to an angelic choice of nondeterministic choices of pure computations:

$$\bigsqcup_{i \in I} \left(\bigsqcap_{j \in J_i} \mathit{return}\ x_j \right)$$

where each index set I and J_i is finite and nonempty.

And indeed, initial model is finite nonempty sets of finite nonempty sets.

For example,

$$\mathit{return}\ 1 \sqcup (\mathit{return}\ 0 \sqcap \mathit{return}\ 2)$$

(ie angel's choice between always 1 and a demonic choice)
can be represented as the set of sets $\{\{1\}, \{0, 2\}\}$.

8.2. Up-closure

Additional healthiness condition:
if xs is an option, and $xs \subseteq ys$, then ys might as well be an option too:

$$\{\{1\}, \{0,2\}\} \quad \rightsquigarrow \quad \{xs \mid \{1\} \subseteq xs \vee \{0,2\} \subseteq xs\}$$

Analogous to convex closure for nondeterministic and probabilistic choice, and arises for same reason:

$$\begin{array}{ll} & p \sqcup q \\ = & \quad [\![\text{ idempotence of } \sqcap \ ]\!] \\ & (p \sqcup q) \sqcap (p \sqcup q) \\ = & \quad [\![\ \sqcap \text{ distributes over } \sqcup, \text{ three times } \ ]\!] \\ & (p \sqcap p) \sqcup (p \sqcap q) \sqcup (q \sqcap p) \sqcup (q \sqcap q) \\ = & \quad [\![\text{ idempotence and commutativity of } \sqcap, \text{ idempotence of } \sqcup \ ]\!] \\ & p \sqcup (p \sqcap q) \sqcup q \end{array}$$

If p, q both acceptable to angel, then allow demon to choose between them.

8.3. Representation

For simplicity, represent sets as lists; so we take equivalence modulo reordering, duplication, and up-closure—morally,

```
type MultiRel a = [[a]]
```

A monad, using the distributive-law construction:

```
swap :: [[a]] → [[a]]
swap ([xs])     = [[x] | x ← xs]
swap (xs : yss) = [x : ys | x ← xs, ys ← swap yss]
```

Angelic choice is union, demonic is cartesian product with union:

```
instance MonadAngDem MultiRel where
  xss ⊔ yss = xss ++ yss
  xss ⊓ yss = [xs ++ ys | xs ← xss, ys ← yss]
```

8.4. Illustration

Demonic choice distributes over angelic, by construction:

$$\begin{aligned} \mathit{return}\,1 \sqcap (\mathit{return}\,0 \sqcup \mathit{return}\,2) &= [[1,0],[1,2]] \\ (\mathit{return}\,1 \sqcap \mathit{return}\,0) \sqcup (\mathit{return}\,1 \sqcap \mathit{return}\,2) &= [[1,0],[1,2]] \end{aligned}$$

It's less obvious that angelic choice distributes over demonic:

$$\begin{aligned} \mathit{return}\,1 \sqcup (\mathit{return}\,0 \sqcap \mathit{return}\,2) &= [[1],[0,2]] \\ (\mathit{return}\,1 \sqcup \mathit{return}\,0) \sqcap (\mathit{return}\,1 \sqcup \mathit{return}\,2) &= [[1,1],[1,2],[0,1],[0,2]] \end{aligned}$$

—different, but with the same up-closure: $\{1\} \subseteq \{1,2\}$ and $\{1\} \subseteq \{0,1\}$.

Up-closure justified by this second distributive law; or
up-closure :: multi-relations : monotonicity :: predicate transformers; or
operationally, eg in terms of strategies for games (player chooses a set, opponent an element from that set).

9. Conclusions

This approach:

- axiomatic approach to reasoning with effects
- simple and generic
- lightweight (naive?) treatment of probability and nondeterminism
- many other classes of effect: randomization, I/O, exceptions...
- smacks of 'algebraic theories of effects' (Plotkin & Power, Lawvere)

  (but some operations require special treatment as 'handlers' in that setting)

9.1. . . . versus relations

Bird & de Moor's *Algebra of Programming* is basically FP with relations.

Their development can largely be replayed monadically. After all, $\mathbb{P}$ is a monad, and

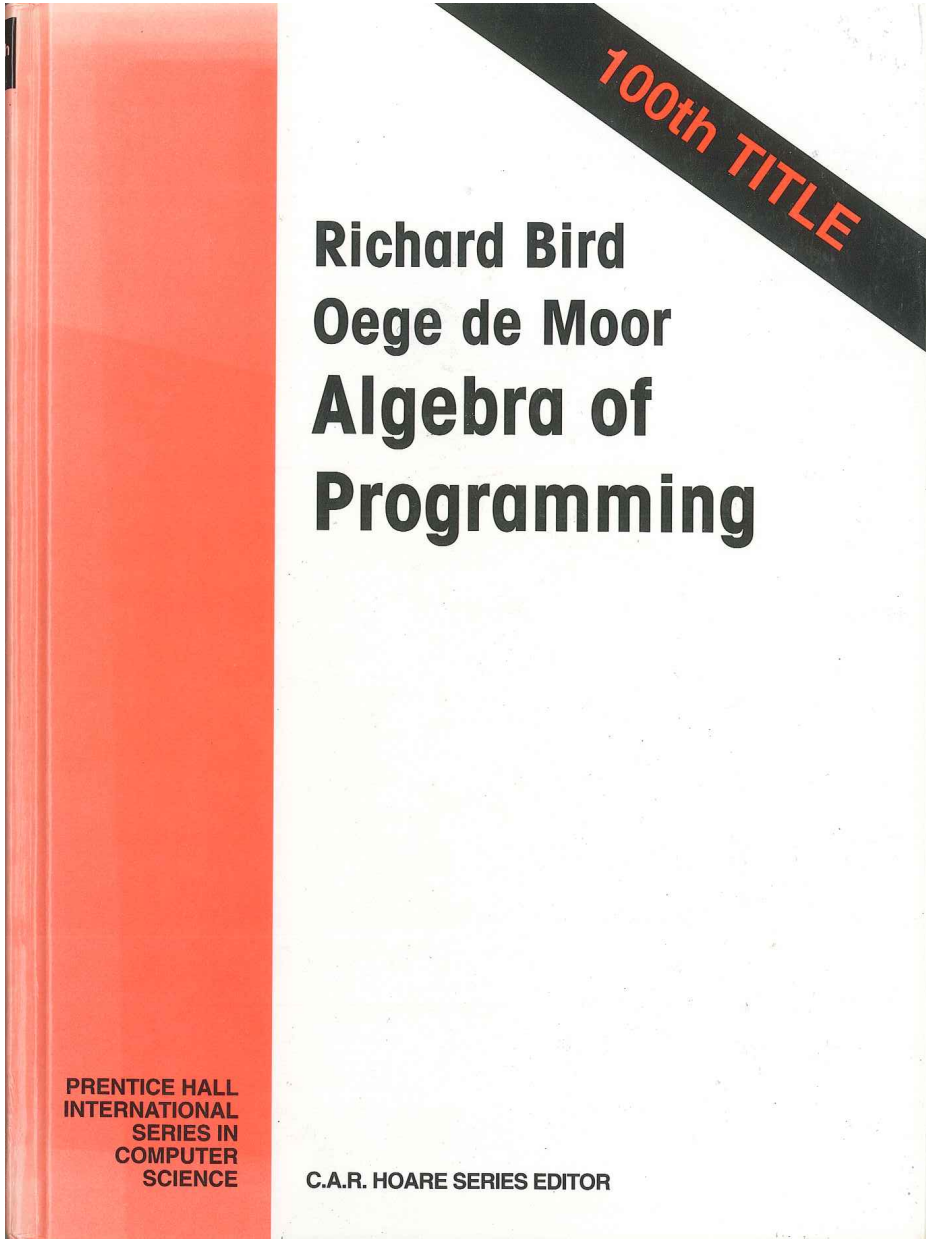
$$\mathbb{P}\ (A \times B) \quad \simeq \quad A \to \mathbb{P}\ B$$

See *Maximum Segment Sum, Monadically*, `arXiv:1109.0782v2`.

What about refinement in FP?
It's there, in the monad of nondeterminism:

$$p \sqsubseteq q \quad \hat{=} \quad (p \sqcap q = p)$$

So for example, $(p \sqcap q) \sqsubseteq (p \lhd w \rhd q)$.



9.2. UTP with monads

UTP is also based on relations, and relational programming is

> *monadic programming with one specific monad*

Why not generalize to arbitrary monads?

Then exploit the power of FP:

- powerful abstractions
- advanced datatypes
- simple reasoning
- exciting applications

9.3. MPC

Who will deny that Oxford, by her ineffable charm, keeps ever calling us nearer to the true goal of all of us, to the ideal, to perfection, – to beauty, in a word, which is only truth seen from another side? – nearer, perhaps, than all the science of Tübingen. Adorable dreamer, whose heart has been so romantic! who hast given thyself so prodigally, given thyself to sides and to heroes not mine, only never to the Philistines! *home of lost causes, and forsaken beliefs, and unpopular names, and impossible loyalties!*

— Matthew Arnold