Abstract. The combination of probabilistic and nondeterministic choice in program calculi is a notoriously tricky problem, and one with a long history. We present a simple functional programming approach to this challenge, based on algebraic theories of computational effects. We make use of the powerful abstraction facilities of modern functional languages, to introduce the choice operations as a little embedded domain-specific language rather than having to define a language extension; we rely on referential transparency, to justify straightforward equational reasoning about program behaviour.

For Carroll Morgan, on his sixtieth birthday.

1 Introduction

In the preface of his book [31] with Annabelle McIver on “rigorous reasoning about probabilistic programs and systems”, Carroll Morgan writes:

_Economy and precision of reasoning are what we have come to expect for standard programs; there is no reason we should accept less when they are probabilistic._

The book goes on to present the probabilistic Guarded Command Language pGCL, an extension of Dijkstra’s GCL [6] to include probabilistic choice as well as the usual nondeterministic choice, and its semantics in terms of expectation transformers, the generalization of predicate transformers from boolean predicates to numeric expectations.

In this paper, we present an alternative approach to rigorous reasoning about probabilistic programs and systems, also offering the desired economy and precision. Our approach is based on pure functional programming, rather than on Dijkstra’s imperative style. The main advantage of pure functional programming when it comes to manipulating programs is its referential transparency, which means that familiar _equational reasoning_ works as well for expressions denoting programs as it does for expressions denoting numbers. As a consequence, we no longer need two distinct domains of discourse—a programming notation in which
to express computations, and a logic in which to reason about them—because
the one language works for both.

A second advantage of functional programming is the facilities it provides
for defining new abstractions ‘within the language’, rather than requiring one to
step out into the meta-language in order to define a new language. Our chosen
language Haskell does not provide constructs for probabilistic choice itself, but
that is no obstacle: instead, it provides the necessary abstraction mechanisms
that allow us to define those constructs ourselves. Rather than a new language
‘probabilistic Haskell’, we can define probabilistic choice within standard Haskell;
one might characterize the result as an embedded domain-specific language for
probabilistic programming.

The apparent drawback of pure functional programming is that its very pu-
rity appears to rule out interesting computational effects, not least probabilistic
choice. After all, if $\text{coin}$ denotes the computation modelling a fair coin toss—
a 50–50 choice between heads and tails—then do not two occurrences of $\text{coin}$
denote possibly different outcomes, thereby destroying referential transparency?
The apparent problem is eliminated by distinguishing between types that repre-
sent values, such as ‘heads’, and those that represent computations, such as $\text{coin}$.
Two occurrences of $\text{coin}$ denote the same computation, and it is the execution
of these computations that may yield different outcomes. Each class of effects,
such as probabilistic choice, determines a notion of computation, in this case of
probability distributions; $\text{coin}$ denotes not a single outcome, but a distribution
of outcomes. The operations and axioms of a notion of computation can be pre-
cisely and elegantly abstracted as a monad, or equivalently an algebraic theory;
equational reasoning can be conducted within such a theory.

The paper is intended as a tutorial and a manifesto, rather than presenting
any new results. We start with a brief introduction to pure functional program-
ning and to the use of monads to capture computational effects (Section 2)—
readers familiar with functional programming in general, and Haskell in partic-
ular, may wish to skip this section. We then introduce theories of nondetermi-

#### 2 Effectful functional programming

Pure functional programming languages constitute a very appealing model of
computation: simple, due to abstraction from the details of computer architec-
ture, yet still expressive, allowing concise specification of complex constructions.
These strengths derive from referentially transparency: as far as the semantics is
concerned, the only relevant aspect of any expression is the value it denotes. In
particular, expressions have no side-effects; so any subexpression can be replaced
by any other having the same value, without affecting the surrounding context. Expressions therefore behave like ordinary high-school algebra, and reasoning about programs can be conducted using ordinary high-school equational reasoning, substituting one subexpression for another with equal value. Consequently, one’s language for programming is simultaneously one’s language for reasoning about those programs—there is no need to step outside the programming notation to a different logical domain such as predicate calculus.

2.1 Functional programming

The essence of functional programming is that programs are equations and functions are values. For example, the squaring function on integers might be defined:

\[
\text{square} :: \text{Int} \to \text{Int} \\
\text{square} x \equiv x \times x
\]

or equivalently

\[
\text{square} :: \text{Int} \to \text{Int} \\
\text{square} = \lambda x \to x \times x
\]

As well as specifying an action, namely how to compute squares, this program also serves as an equation: for any \( x \), the expression \( \text{square} x \) is equivalent to the expression \( x \times x \), and either may be replaced anywhere by the other (taking due care over bound variables); similarly, the identifier \( \text{square} \) itself may be replaced by the lambda expression \( \lambda x \to x \times x \) denoting the squaring function. Likewise, function composition \( (\circ) \) is a value, just like any other, albeit a higher-order one:

\[
(\circ) :: (b \to c) \to (a \to b) \to a \to c \\
(f \circ g) x = f (g x)
\]

Functional programmers restrict themselves to manipulating expressions, rather than statements. So in order to regain the expressivity provided by statements in imperative languages, functional programming must provide an enriched expression language. Higher-order operators like functional composition go some way towards this. Another powerful tool is to allow complex data structures to be denotable as expressions; for example, the datatype \( [a] \) of lists of elements each of type \( a \) might be defined as follows:

```
data [a] = [ ] | a : [a]
```

With this device, a data structure such as the list of three elements \( 1:(2:(3:[])) \) can be denoted as an expression; in contrast, in conventional imperative languages, complex data structures such as lists and trees can generally be constructed only via a sequence of side-effecting assignment statements.

Functions over such data structures can conveniently be defined by pattern matching. For example, here is the standard function \( \text{foldr} \) to fold a list to a value:
This is another higher-order function, since it takes a function as its first argument. One instance of \texttt{foldr} is the function \texttt{sum} that sums a list of integers:

\[
\text{sum} :: [\text{Int}] \to \text{Int} \quad \text{sum} = \text{foldr} (+) 0
\]

Another is the higher-order function \texttt{map} that applies an argument \texttt{f} to each element of a list:

\[
\text{map} :: (\text{a} \to \text{b}) \to ([\text{a}] \to [\text{b}]) \\
\text{map} g = \text{foldr} (\lambda x ys \rightarrow g x : ys) []
\]

Lists are a polymorphic datatype; the polymorphism is expressed precisely by \texttt{map}. \textit{Polymorphic functions} such as \texttt{reverse :: [a] \to [a]} are those that depend only on the structure of a datatype, oblivious to the elements; their polymorphism is expressed precisely by a corresponding \textit{naturality property}, stating that they commute with the appropriate \texttt{map} function—for example, \(\text{reverse} \circ \text{map} f = \text{map} f \circ \text{reverse}\) [52].

### 2.2 Equational reasoning

Referential transparency means that plain ordinary equational reasoning suffices for proving properties of programs. For example, one very important property of the \texttt{foldr} function is the \textit{fusion law}:

\[
\text{h} \circ \text{foldr} f e = \text{foldr} f' e' \iff h (f \; x \; y) = f' x (h \; y) \land h \; e = e'
\]

One way of proving this law is by induction over lists (which we assume here to be finite). For the base case, we have:

\[
\text{h} (\text{foldr} f e []) \\
= \quad [ \text{definition of foldr} ] \\
\text{h} e \\
= \quad [ \text{assumption} ] \\
\text{e'} \\
= \quad [ \text{definition of foldr} ] \\
\text{foldr} f' \; e' []
\]

For the inductive step, we assume that the result holds on \(xs\), and calculate for \(x : xs\) as follows:

\[
\text{h} (\text{foldr} f e (x : xs)) \\
= \quad [ \text{definition of foldr} ] \\
\text{h} (f \; x \; (\text{foldr} f e \; xs))
\]
A simple consequence of the fusion law is the fold–map fusion law, when \( h \) is itself an instance of \( \text{foldr} \), and follows a \text{map} over lists, which is another instance of \( \text{foldr} \). In this case, the fusion result

\[
\text{foldr} \ f \ e \circ \text{map} \ g = \text{foldr} \ f' \ e'
\]

follows from the fusion conditions

\[
\text{foldr} \ f \ e \left( g \ x : y s \right) = f' \ x \left( \text{foldr} \ f \ e \ y s \right) \land \text{foldr} \ f \ e \ [] = e'
\]

These in turn are satisfied if \( e' = e \) and \( f' = \lambda x \ z \rightarrow f \ (g \ x) \ z = f \circ g \); that is,

\[
\text{foldr} \ f \ e \circ \text{map} \ g = \text{foldr} \ (f \circ g) \ e
\]

For most of the paper we will assume a setting of total functions between sets. In this setting, arbitrary recursive definitions do not in general admit canonical solutions; we restrict attention to well-founded recursions such as that in \( \text{foldr} \), and correspondingly to finite data structures. We only have to relax this restriction in Section 8.

### 2.3 Effects in pure functional languages

Equational reasoning about pure computations is all very well, but to be useful, computations must have some observable effects. It may seem at first that equational reasoning must then be abandoned. After all, as soon as one allows state-mutating statements such as \( x := x + 1 \) in a programming language, the high-school algebra approach to reasoning no longer works; and similarly for other classes of effect, such as input/output, nondeterminism, probabilistic choice, exceptions, and so on.

Moggi [36] famously showed how the well-understood concept of a \textit{monad} from category theory provides exactly the right interface to an abstraction of computational effects such as mutable state, allowing the development of an elegant yet expressive \textit{computational lambda calculus} for modelling programming languages with effects. Wadler [54] popularized this approach within functional programming, and it quickly became the technology of choice for integrating effects into lazy functional programming languages such as Haskell.

With the monadic approach to computational effects, purely functional expressions are classified into two kinds: those that denote values like integers and strings, and those that denote computations with possible effects. However, both are represented as pure data—the computations are represented as pure \textit{terms}
in a certain abstract syntax, rather than some kind of impure action. When the run-time system of a language encounters the first kind of expression, it evaluates it and prints it out; when it encounters the second kind, it evaluates it, interprets the term as the effectful computation it encodes, and executes that computation. Consequently, evaluation remains pure, and any impurities are quarantined within the run-time system.

The abstract syntax needed to capture effectful computations is very simple. There is a general framework consisting of just two operators, which in a sense model the compositional structure of computations; then for each class of effect, there is an extension to the general framework to model the primitives specific to that class. (In fact, the general framework and a specific extension together represent the free term algebra for the signature corresponding to the primitives for a particular class of effects. It is no coincidence that monads turn out to be useful for modelling such term algebras, because they were developed precisely as a categorical expression of universal algebra [30]. We return to this point in Section 9.3.)

The general framework can be expressed as a type class in Haskell:

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

  p >>= k = join (fmap k p)
  join pp = pp >>= id
  fmap f p = p >>= (return . f)
```

This declaration states that the type constructor (that is, operation on types) m is in the type class Monad if we can provide suitable definitions of the four methods return, (>>=), fmap, and join, with the given types. In fact, the methods are interdefinable, and some have default definitions in terms of others; it is necessary to define return, but it suffices to define either (>>=) or both fmap and join. (We have chosen this presentation allowing alternative definitions for flexibility; it is different from but equivalent to the Monad class in the Haskell standard libraries.)

Technically, the methods should also satisfy some laws, although these cannot be stated in the Haskell type class declaration:

- `return x >>= k = k x` -- left unit
- `p >>= return = p` -- right unit
- `(p >>= k) >>= k' = p >>= (λx -> k x >>= k')` -- associativity
- `fmap id = id` -- map-identity
- `fmap (f o g) = fmap f o fmap g` -- map-composition
- `join o return = id` -- left unit
- `join o fmap return = id` -- right unit
- `join o fmap join = join o join` -- associativity
Throughout the paper, we make the following naming conventions: $p, q, r$ denote monadic terms or ‘programs’, $h, k$ denote functions yielding monadic terms, $x, y, z$ denote polymorphic variables, $a, b, c$ denote booleans, $l, m, n$ denote integers, and $u, v, w$ denote probabilities.) Informally, the type $m a$ denotes computations that may have some effect, and that yield results of type $a$. The function $\text{return}$ lifts plain values into pure computations. The operator $\gg$, pronounced ‘bind’, acts as a kind of sequential composition; the second computation may depend on the result returned by the first, and the overall result is the result of the second. The first three laws can be seen as unit and associativity properties of this form of sequential composition. The function $\text{join}$ flattens a computation that yields computations that yield results into a computation that yields results directly, and the function $\text{fmap}$ modifies each of the results of a computation; together with $\text{return}$, these two give an alternative (equivalent) perspective on sequential composition.

Two shorthands turn out to be quite convenient. We write $\text{skip}$ for the pure computation that returns the unit ():

\[
\text{skip} :: \text{Monad } m \Rightarrow m ()
\]

\[
\text{skip} = \text{return } ()
\]

and $\gg$ for the special case of $\gg$ in which the second computation is independent of the result returned by the first:

\[
(\gg) :: \text{Monad } m \Rightarrow m a \rightarrow m b \rightarrow m b
\]

\[
p \gg q = p \gg (\lambda \_ \rightarrow q)
\]

These two shorthands more obviously form analogues of the ‘skip’ and ‘sequential composition’ operators of imperative programming languages. For example, with these we can form the sequential composition of a sequence of unit-returning computations, discarding all the unit results and returning unit overall. (This is actually a type specialization of the corresponding function in the Haskell standard library, but it is sufficient for our purposes.)

\[
\text{sequence}_\_ :: \text{Monad } m \Rightarrow [m ()] \rightarrow m ()
\]

\[
\text{sequence}_\_ = \text{foldr } (\gg) \text{ skip}
\]

This function reveals one of the beauties of pure and lazy functional programming: if a useful control construct is missing from the language, it is usually possible to define it as an ordinary value rather than having to extend the syntax and the compiler. Another famous example is the conditional; if Haskell didn’t already provide the if ... then ... else... construct, something entirely equivalent (except for the concrete syntax) could be defined—the same cannot be said of a language providing only eager evaluation. And because conditional would be an ordinary value, the ordinary principles of reasoning would apply; for example, function application distributes leftwards and rightwards over conditional:

\[
f (\text{if } b \text{ then } x \text{ else } y) = \text{if } b \text{ then } f x \text{ else } f y
\]

\[
(\text{if } b \text{ then } f \text{ else } g) x = \text{if } b \text{ then } f x \text{ else } g x
\]
These laws can easily be verified by considering the two cases \( b = \text{True} \) and \( b = \text{False} \). (In fact, the first law as stated only holds in SET. Once one moves to CPO, one also needs to consider the case that \( b \) is undefined; then the first law only holds when \( f \) is strict. The second law is still unconditional, provided that \( \lambda x \to \bot = \bot \); this is the case with flat function spaces, the usual presentation in CPO, but not in fact in Haskell with the \textit{seq} operator, which distinguishes between \( \bot \) and \( \lambda x \to \bot \).) In particular, letting \( f \) be \((\Rightarrow k)\) and \((p \Rightarrow)\), we deduce from the first law that composition distributes respectively leftwards and rightwards over conditional:

\[
(\text{if } b \text{ then } p \text{ else } q) \Rightarrow k = \text{if } b \text{ then } p \Rightarrow k \text{ else } q \Rightarrow k
\]

\[
p \Rightarrow (\text{if } b \text{ then } k \text{ else } k') = \text{if } b \text{ then } p \Rightarrow k \text{ else } p \Rightarrow k'
\]

(Again, these laws hold unconditionally in SET; in CPO, they require \( \Rightarrow \) to be strict in its left and right argument, respectively.)

### 2.4 State

So much for the general framework; here is an extension to capture mutable state as a class of effects. Just two additional operations are required: \textit{get}, to read the state, and \textit{put}, to update it. We declare a subclass \textit{MonadState} of \textit{Monad}; type constructor \( m \) is a member of the class \textit{MonadState} if it is a member of \textit{Monad} and it supports the two additional methods \textit{get} and \textit{put}. (To be precise, the subclass is \textit{MonadState} \( s \) for some fixed state type \( s \), and it encompasses type constructors \( m \) that support mutable state of type \( s \); the vertical bar precedes a ‘functional dependency’, indicating that \( m \) determines \( s \).)

\[
\textit{class} \quad \textit{Monad} \quad m \Rightarrow \textit{MonadState} \quad s \quad m | m \to s \textit{ where} \\
\textit{get} :: m s \\
\textit{put} :: s \to m ()
\]

As with the two methods of the \textit{Monad} interface, it is not sufficient simply to provide implementations of \textit{get} and \textit{put} that have the right types—they should also satisfy some laws:

\[
\begin{align*}
\textit{get} \Rightarrow \lambda s \to \textit{get} \Rightarrow \lambda s' \to k s s' &= \textit{get} \Rightarrow \lambda s \to k s s \quad \text{-- get–get} \\
\textit{get} \Rightarrow \textit{put} &= \textit{skip} \quad \text{-- get–put} \\
\textit{put} s \Rightarrow \textit{put} s' &= \textit{put} s' \quad \text{-- put–put} \\
\textit{put} s \Rightarrow \textit{get} \Rightarrow \lambda s' \to k s s' &= \textit{put} s \Rightarrow k s \quad \text{-- put–get}
\end{align*}
\]

Informally: two consecutive \textit{get}s will read the same value twice; \textit{getting} a value then \textit{putting} it back is has no effect; two consecutive \textit{put}s are equivalent to just the second one; and a \textit{get} immediately after \textit{putting} a value will yield that value.

For example, here is a simple expression denoting a computation on a mutable integer state, which reads the current state, increments it, writes the new value back, and then returns the parity of the original value.
incrOdd :: MonadState Int m ⇒ m Bool
incrOdd = get >>= (λn → put (n + 1)) >>= (λ() → return (odd n))

There is an obvious simulation of mutable state in terms of state-transforming functions. A computation that acts on a state of type \( s \), and yields a result of type \( a \), can be represented as a function of type \( s → (a, s) \):

\[
\text{type State } s \ a = s → (a, s)
\]

Now, \( \text{State } s \) forms a type of computations, and so can be made an instance of the type class \( \text{Monad} \)—for \( \text{return} a \) we use the state-transforming function that yields \( x \) and leaves the state unchanged; \( \text{fmap} f \) applies \( f \) to the output value without touching the output state; and \( \text{join} \) collapses a state-transformer that yields a state-transformer by applying the output state-transformer to the output state:

\[
\begin{align*}
\text{instance Monad } (\text{State } s) \text{ where} \\
\text{return } x &= λs → (x, s) \\
\text{fmap } f \ p &= λs → \text{let } (x, s') = p \ s \text{ in } (f \ x, s') \\
\text{join } p &= λs → \text{let } (p', s') = p \ s \text{ in } p' \ s'
\end{align*}
\]

The reader may enjoy deriving from this the corresponding definition \( p ≡ k = λs → \text{let } (x, s') = p \ s \text{ in } k \ x \ s' \) of bind, which chains state transformations together. (For technical reasons, this instance declaration is not quite in Haskell syntax: rather than a type synonym, \( \text{State} \) ought to be a \texttt{newtype} or \texttt{datatype}, with a constructor and deconstructor. But what is shown is morally correct.)

Of course, \( \text{State } s \) supports the features of mutable state—\( \text{get} \) yields a copy of the state, and \( \text{put} \) overwrites it:

\[
\begin{align*}
\text{instance MonadState } s \ (\text{State } s) \text{ where} \\
\text{get} &= λs → (s, s) \\
\text{put } s' &= λs → ((), s')
\end{align*}
\]

As it happens, the datatype \( \text{State } s \) is (isomorphic to) the free term algebra on the \( \text{MonadState } s \) signature, modulo the four laws of \( \text{get} \) and \( \text{put} \) [42].

### 2.5 Imperative functional programming

Wadler also observed [53] that the methods of the \( \text{Monad} \) interface are sufficient to implement a notation based on the set comprehensions of Zermelo–Fraenkel set theory. This too has found its way into Haskell, as the ‘\texttt{do} notation’ [25], which is defined by translation into \( \text{Monad} \) methods as follows:

\[
\begin{align*}
\text{do } \{ \ p \} &= p \\
\text{do } \{ x ← p ; qs \} &= p ≡ λx → \text{do } \{ qs \} \\
\text{do } \{ p ; qs \} &= p ≡ \text{do } \{ qs \} \\
\text{do } \{ \text{let } decls ; qs \} &= \text{let } decls \text{ in } \text{do } \{ qs \}
\end{align*}
\]
In particular, instead of having to write functions (typically lambda expressions) as the second argument of >>=, with the do notation we can write a generator \( x \leftarrow p \) to bind a new variable \( x \) that is in scope in all subsequent qualifiers. Using this notation, we can rewrite the incrodd program above as follows:

\[
\text{incrodd} :: \text{MonadState Int} \Rightarrow m \text{ Bool} \\
\text{incrodd} = \text{do}\{ n \leftarrow \text{get} ; \text{put} (n + 1) ; \text{return} (\text{odd } n) \}
\]

The three monad laws appear in the do notation as follows:

\[
\text{do}\{ x \leftarrow \text{return } e ; k x \} = \text{do}\{ k e \} \\
\text{do}\{ x \leftarrow p ; \text{return } x \} = \text{do}\{ p \} \\
\text{do}\{ x \leftarrow p ; y \leftarrow h x ; k y \} = \text{do}\{ y \leftarrow \text{do}\{ x \leftarrow p ; h x \} ; k y \}
\]

(where, implicitly in the third law, \( x \) is not free in \( k \)). Distribution of composition leftwards and rightwards over conditional looks like this:

\[
\text{do}\{ x \leftarrow \text{if } b \text{ then } p \text{ else } q ; k x \} = \text{if } b \text{ then do}\{ x \leftarrow p ; k x \} \text{ else do}\{ x \leftarrow q ; k x \} \\
\text{do}\{ x \leftarrow p ; \text{if } b \text{ then } h x \text{ else } k x \} = \text{if } b \text{ then do}\{ x \leftarrow p ; h x \} \text{ else do}\{ x \leftarrow p ; k x \}
\]

(where, implicitly in the second law, \( x \) is not free in \( b \)). The four laws of state become:

\[
\text{do}\{ s \leftarrow \text{get} ; s' \leftarrow \text{get} ; k s s' \} = \text{do}\{ s \leftarrow \text{get} ; k s s \} -- \text{get-get} \\
\text{do}\{ s \leftarrow \text{get} ; \text{put} s \} = \text{do}\{ \text{skip} \} -- \text{get-put} \\
\text{do}\{ \text{put} s ; s' \leftarrow \text{get} ; k s s' \} = \text{do}\{ \text{put} s' \} -- \text{put-put} \\
\text{do}\{ \text{put} s ; s' \leftarrow \text{get} ; k s s' \} = \text{do}\{ \text{put} s ; k s \} -- \text{put-get}
\]

The do notation yields a natural imperative programming style, as we hope the rest of this paper demonstrates; indeed, it has been said that “Haskell is the world’s finest imperative programming language” [40].

### 2.6 An example of simple monadic equational reasoning

To summarize: the Monad class provides an interface for sequencing computations; one should program to that interface where appropriate, making subclasses of Monad for each specific class of effects; and the interface ought to specify laws as well as signatures for its methods. We have recently argued [10] that this perspective on monads is precisely the right one for equational reasoning about effectful programs—contrary to popular opinion, the impurities of computational effects offer no insurmountable obstacles to program calculation, at least when they are properly encapsulated. To illustrate this claim, we present a simple example of reasoning with stateful computations.

Here is a simple stateful computation to add an integer argument to an integer state:
\textit{add} :: \text{MonadState} \text{ Int} \ m \Rightarrow \text{Int} \rightarrow m ()
\text{add} \ n = \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + n) \}

We claim that adding each integer in a list in sequence to an integer state is the same as adding their sum all at once:

\text{addAll} = \text{add} \circ \text{sum}

where \text{addAll} turns each integer in a list into an integer-adding computation, then sequences this list of computations:

\text{addAll} :: \text{MonadState} \text{ Int} \ m \Rightarrow [\text{Int}] \rightarrow m ()
\text{addAll} = \text{sequence} \circ \text{map add}

Because \text{sequence} \_ \_ \_ is an instance of \text{foldr}, we can combine the two phases of \text{addAll} into one, using the fold–map fusion law:

\text{addAll} = \text{foldr} (\lambda n \ p \rightarrow \text{do} \{ \text{add} \ n ; \ p \}) \text{ skip}

Now, since \text{sum} and \text{addAll} are both instances of \text{foldr}, the claim is an instance of the standard fusion law, and follows from two simple fusion properties:

\text{add} \ 0 = \text{skip}
\text{add} \ (n + n') = \text{do} \{ \text{add} \ n ; \text{add} \ n' \}

For the first of these, we have:

\begin{align*}
\text{add} \ 0 & = \text{do} \{ \text{add} \} \\
& = \text{do} \{ l \leftarrow \text{get} ; \text{put} (l + 0) \} \\
& = \text{do} \{ l \leftarrow \text{get} ; \text{put} l \} \\
& = \text{do} \{ \text{get--put} \} \\
& = \text{skip}
\end{align*}

And for the second, starting from the more complicated right-hand side, we have:

\begin{align*}
\text{do} \{ \text{add} \ n ; \text{add} \ n' \} & = \text{do} \{ \text{add} \} \\
& = \text{do} \{ \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + n) \} \circ \text{do} \{ l \leftarrow \text{get} ; \text{put} (l + n') \} \} \\
& = \text{do} \{ \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + n) ; l \leftarrow \text{get} ; \text{put} (l + n') \} \circ \text{do} \{ m \leftarrow \text{get} ; \text{put} ((m + n) + n') \} \} \\
& = \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + n) ; \text{put} ((m + n) + n') \} \\
& = \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + n) ; \text{put} (m + (n + n')) \} \\
& = \text{do} \{ m \leftarrow \text{get} ; \text{put} (m + (n + n')) \}
\end{align*}
which completes the proof.

Of course, sum and addAll are two rather special functions, both being instances of the easily manipulated foldr pattern. However, that is incidental to our point: if we had picked an example involving a more complicated pattern of computation, then the reasoning would certainly have been more complicated too, but it would still have been plain ordinary equational reasoning—reasoning about the computational effects would pose no more of a problem.

3 An algebraic theory of nondeterministic choice

Let us now turn to a different class of effects. Nondeterministic programs are characterized by the ability to choose between multiple results. We model this as a subclass of Monad.

class Monad m ⇒ MonadAlt m where

(□) :: m a → m a → m a

We stipulate that □ is associative, commutative, and idempotent:

\[(p □ q) □ r = p □ (q □ r)\]
\[p □ q = q □ p\]
\[p □ p = p\]

and that composition distributes leftwards over it:

\[\text{do } \{ x ← (p □ q) ; k x \} = \text{do } \{ x ← p ; k x \} □ \text{do } \{ x ← q ; k x \}\]

However, we do not insist that composition distributes rightwards over choice: in general,

\[\text{do } \{ x ← p ; (h x □ k x) \} \neq \text{do } \{ x ← p ; h x \} □ \text{do } \{ x ← p ; k x \}\]

This is in order to accommodate both angelic and demonic interpretations of nondeterminism. One distinction between the two is in terms of the number of branches of a choice that an implementation might choose to follow: angelic choice will explore both branches, whereas demonic choice is free to pick either branch but will not follow both. In particular, consider the case that computation \(p\) has some non-idempotent effects in addition to nondeterminism, such as writing output. If □ is angelic, then these effects happen once on the left-hand side of the equation, and twice on the right; whereas if □ is demonic, just one branch of each choice will be picked, and the two sides of the equation are indeed equal.

On account of the associativity, commutativity, and idempotence of choice, the essential—indeed, the initial, in the categorical sense—semantics of a nondeterministic computation amounts to a finite nonempty set of alternative results.
In other words, we can simulate a computation that exploits just the effect of choice as a function that returns a nonempty set of results. A pure computation amounts to returning a singleton set, \( \text{fmap} \ f \) applies \( f \) to each element of a set, and a computation of computations can be flattened by taking the union of the resulting set of sets. (The operational behaviour of an implementation will differ, depending on the interpretation of choice: an angelic implementation will deliver the whole set of results; a demonic implementation will pick one arbitrarily. But either way, the semantics is represented as a function yielding a set.)

A convenient approximate implementation of finite nonempty sets is in terms of nonempty lists—‘approximate’ in the sense that we consider two lists to represent the same set of results if they are equal up to reordering and duplication of elements.

\[
\begin{align*}
\text{instance Monad \( [\] \) where} \\
\quad \text{return} \ a &= [a] \\
\quad \text{fmap} \ f \ p &= [f \ x \mid x \leftarrow p] \\
\quad \text{join} &= \text{concat}
\end{align*}
\]

Naturally, we implement the nondeterministic choice as concatenation:

\[
\begin{align*}
\text{instance MonadAlt \( [\] \) where} \\
\quad (\Box) &= (+ +)
\end{align*}
\]

In some other contexts, we might not want such a strong collection of laws for nondeterministic choice. For example, if we are modelling search strategies [14, 47], we might want to treat as significant the order in which results are found, and so we might want to drop the commutativity axiom.

### 3.1 Example: subsequences of a list

As an example of reasoning with nondeterministic programs, here is a rendition in terms of choice of the function \( \text{subs} \) that nondeterministically chooses a subsequence of a list. Of course, interpreted in the nonempty-list implementation of nondeterminism, \( \text{subs} \) returns the usual nonempty list of lists; but this definition supports other implementations of nondeterminism too, such as bags and sets.

\[
\begin{align*}
\text{subs} :: \text{MonadAlt} \ m \Rightarrow [a] \to m [a] \\
\text{subs} [\] &= \text{return} [\] \\
\text{subs} (x : xs) &= \text{fmap} \ (x:) \ xss \Box xss \textbf{ where } xss = \text{subs} \ xs
\end{align*}
\]

Here is a simple property that we might wish to prove—that \( \text{subs} \) distributes over list concatenation:

\[
\text{subs} \ (xs + ys) = \textbf{do} \{ us \leftarrow \text{subs} \ xs ; vs \leftarrow \text{subs} \ ys ; \text{return} \ (us + vs) \}
\]

Using the laws of nondeterminism, this property of an effectful program can be proved by induction over \( xs \), using plain ordinary equational reasoning. For the base case \( xs = [\] , we have:
For the inductive step, we assume the result for \(xs\), and calculate for \(x \cdot xs\) as follows:

\[
\begin{align*}
&\begin{array}{l}
\text{do } (us \leftarrow \text{subs } \{x \cdot xs\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (us \leftarrow \text{return } \{\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } ([] \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } vs) \\
\text{subs } ys \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{l}
\text{do } (us \leftarrow \text{subs } \{x \cdot xs\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (us \leftarrow \text{return } \{\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } vs) \\
\text{subs } ys \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{l}
\text{do } (us \leftarrow \text{subs } \{x \cdot xs\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (us \leftarrow \text{return } \{\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } vs) \\
\text{subs } ys \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{l}
\text{do } (us \leftarrow \text{subs } \{x \cdot xs\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (us \leftarrow \text{return } \{\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } vs) \\
\text{subs } ys \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{l}
\text{do } (us \leftarrow \text{subs } \{x \cdot xs\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (us \leftarrow \text{return } \{\}; vs \leftarrow \text{subs } ys; \text{return } (us \mathbin{+} vs)) \\
\text{do } (vs \leftarrow \text{subs } ys; \text{return } vs) \\
\text{subs } ys \\
\end{array} \\
\end{align*}
\]

Again, plain ordinary equational reasoning suffices, using programs as equations and the axioms of nondeterminism.

4 An algebraic theory of probabilistic choice

Here is another class of effects. Probabilistic computations are characterized by the ability to make a probabilistic choice between alternatives. We suppose a type \(\text{Prob}\) of probabilities (say, the rationals in the closed unit interval), and define a \(\text{Monad}\) subclass for computations drawing from finitely supported probability distributions, that is, distributions in which only a finite number of elements have positive probabilities:
class Monad m ⇒ MonadProb m where
choice :: Prob → m a → m a → m a

The idea is that choice w p q behaves as p with probability w and as q with probability 1 − w. From now on, we will write ‘w’ for 1 − w, and following Hoare’s and Morgan’s convention [15, 17], write choice in infix notation, ‘p ◛ w ⊥ q’, because this makes the laws more legible. We have two identity laws:

\[ p \triangleleft 0 \triangleright q = q \]
\[ p \triangleleft 1 \triangleright q = p \]

a quasi-commutativity law:

\[ p \triangleleft w \triangleright q = q \triangleleft \overline{w} \triangleright p \]

idempotence:

\[ p \triangleleft w \triangleright p = p \]

and quasi-associativity:

\[ p \triangleleft u \triangleright (q \triangleleft v \triangleright r) = (p \triangleleft w \triangleright q) \triangleleft x \triangleright r \iff u = w \times x \land x = p \times v \]

As informal justification for quasi-associativity, observe that the likelihoods of p, q, r on the left are u, \( u \times v, u \times v \), and on the right are w \times x, x \times x, x, and a little algebra shows that these are pairwise equal, given the premise.

As a final pair of laws, we stipulate that bind distributes both leftwards and rightwards over choice:

\[
\begin{align*}
do \{ x ← (p \triangleleft w \triangleright q) ; k x \} &= \ do \{ x ← p ; k x \} \triangleleft w \triangleright \ do \{ x ← q ; k x \} \\
do \{ x ← p ; (h x \triangleleft w \triangleright k x) \} &= \ do \{ x ← p ; h x \} \triangleleft w \triangleright \ do \{ x ← p ; k x \}
\end{align*}
\]

where, in the second law, x is assumed not to occur free in w. (In contrast to nondeterministic choice, we have both distributivities here. This means that, operationally, an implementation may take either branch of a probabilistic choice, but not both—like demonic choice, and unlike angelic.)

For example, a fair coin can be modelled as a 50–50 probabilistic choice between heads and tails (represented as booleans here):

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

One obvious representation to pick as an implementation of MonadProb uses probability-weighted lists of values; thus, coin might be represented as the list \([\text{((True, } \frac{1}{2})], \text{((False, } \frac{1}{2})]\).

\[
\text{type} \ \text{Dist} a = [\langle a, \text{Prob} \rangle] \quad \text{-- weights sum to 1}
\]
A pure computation is represented as a point distribution, mapping applies a function to each element, and a distribution of distributions can be flattened by taking a kind of weighted cartesian product:

\[
\text{instance } \text{Monad } \text{Dist} \text{ where }
\]
\[
\begin{align*}
\text{return } x &= [(x, 1)] \\
\text{fmap } f \ p &= [(f x, w) | (x, w) \leftrightarrow p] \\
\text{join } p &= \text{concat } [\text{scale } w x | (x, w) \leftrightarrow p]
\end{align*}
\]

where

\[
\text{scale} :: \text{Prob} \to [(a, \text{Prob})] \to [(a, \text{Prob})]
\]
\[
\text{scale } v \ p = [(x, v \times w) | (x, w) \leftrightarrow p]
\]

On the other hand, \(\triangledown\triangledown\) is a kind of weighted sum:

\[
\text{instance } \text{MonadProb } \text{Dist} \text{ where }
\]
\[
\begin{align*}
p \triangledown w \triangledown q &= \text{scale } w p + + \text{scale } w q
\end{align*}
\]

Probability-weighted lists are not quite the initial model, because the identity, idempotence, quasi-commutativity, and quasi-associativity laws of \(\triangledown\triangledown\) do not hold. In fact, the initial model of the specification consists of finite mappings from elements to probabilities, collected from these weighted lists in the obvious way—at least, for an element type in the type class \(\text{Eq}\), supporting the equality operation \(\equiv\) needed by finite maps, we can define:

\[
\text{collect} :: \text{Eq } a \Rightarrow \text{Dist } a \to (a \to \text{Prob})
\]
\[
\text{collect } p y = \text{sum } [w | (x, w) \leftrightarrow p, x \equiv y]
\]

That is, equivalences on \(\text{Dist}\) ought to be taken modulo permutations, zero-weighted elements, and repeated elements (whose weights should be added). Nevertheless, the datatype \(\text{Dist}\) itself provides a convenient approximation to the initial model.

Quasi-associativity can make the arithmetic of weights rather complicated, especially when choices are nested. Inspired by Morgan’s distribution comprehensions [38], we sometimes make use of a flat notation for nested choices. For example, instead of \((p \triangledown \frac{1}{2} \triangledown q) \triangledown \frac{1}{3} \triangledown (r \triangledown \frac{1}{4} \triangledown s)\) we allow ourselves to write \(\langle p^{\frac{1}{6}}, q^{\frac{1}{6}}, r^{\frac{1}{6}}, s^{\frac{1}{2}} \rangle\), multiplying out all the probabilities.

4.1 Example: uniform distributions

Extending the fair coin example, we might define uniform distributions

\[
\text{uniform} :: \text{MonadProb } m \Rightarrow [a] \to m a \quad \text{-- nonempty list}
\]
\[
\begin{align*}
\text{uniform } [x] &= \text{return } x \\
\text{uniform } (x : xs) &= \text{return } x \triangledown \frac{1}{\text{length } (x : xs)} \triangledown \text{uniform } xs
\end{align*}
\]

so that \(\text{coin} = \text{uniform } [\text{True}, \text{False}]\), and \(\text{uniform } [1, 2, 3] = \text{return } 1 \triangledown \frac{1}{3} \triangledown (\text{return } 2 \triangledown \frac{1}{2} \triangledown \text{return } 3)\).
Choices drawn from uniform distributions but never used are free of side-effects, and so can be discarded: it is a straightforward proof by induction over \( x \) that

\[
\text{do } \{ x \leftarrow \text{uniform } x : s ; p \} = p
\]

when \( p \) does not depend on \( x \). Similarly, \textit{uniform} distributes over concatenation:

\[
\text{uniform} (x : s + y : s) = \text{uniform} x : s \circ^{m/n} \text{uniform} y : s
\]

where \( m = \text{length } x : s \) and \( n = \text{length } y : s \). As a consequence of these properties of \textit{uniform}, we can conclude that consecutive choices drawn from uniform distributions are independent; that is, choosing consecutively from two uniform distributions is equivalent to choosing in one step from their cartesian product:

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

where

\[
\begin{align*}
\text{cp} & : [a] \to [b] \to [(a, b)] \\
\text{cp} x : s y : s &= [(x, y) | x \leftarrow x : s, y \leftarrow y : s]
\end{align*}
\]

We can prove this property by induction over \( x : s \), using equational reasoning with the laws of \textit{MonadProb}. For the base case of singleton lists, we have:

\[
\begin{align*}
\text{uniform} (cp [x] y : s) &= [ \text{definition of } cp ] \\
&= \text{uniform} [(z, y) | z \leftarrow [x], y \leftarrow y : s] \\
&= [ \text{comprehensions: } [f z | z \leftarrow [x], p] = [f x | p] ] \\
&= \text{uniform} [(x, y) | y \leftarrow y : s] \\
&= [ \text{comprehensions: } [f x | x \leftarrow x : s] = \text{map } f x : s] \\
&= \text{uniform} (\text{map } (\lambda y \to (x, y)) y : s) \\
&= [ \text{natuirlity: } \text{uniform} \circ \text{map } f = \text{fmap } f \circ \text{uniform} ] \\
\text{do } \{ y \leftarrow \text{uniform } y : s ; \text{return} (x, y) \} &= [ \text{left unit } ] \\
\text{do } \{ z \leftarrow \text{return} x : s ; y \leftarrow \text{uniform } y : s ; \text{return} (z, y) \} &= [ \text{definition of } \text{uniform} ] \\
\text{do } \{ z \leftarrow \text{uniform } [x] : s ; y \leftarrow \text{uniform } y : s ; \text{return} (z, y) \} &\]

and for the inductive step, assuming the result for \( x : s \), we have:

\[
\begin{align*}
\text{uniform} (cp (x : s) y : s) &= [ \text{definition of } cp ] \\
&= \text{uniform} [(z, y) | z \leftarrow (x : s), y \leftarrow y : s] \\
&= [ \text{comprehensions distribute over } + ] \\
&= \text{uniform} [(z, y) | z \leftarrow (x : s), y \leftarrow y : s] + [(z, y) | z \leftarrow x : s, y \leftarrow y : s] \\
&= [ \text{as above; definition of } cp ] \\
&= \text{uniform} (\text{map } (\lambda y \to (x, y)) y : s + cp x : s y : s)
\end{align*}
\]
The second step uses the property

\[
[f \lor z \leftarrow zs + zs', p] = [f \lor z \leftarrow zs, p] + [f \lor z \leftarrow zs', p]
\]

Yet again, simple equational reasoning suffices.

5 Combining algebraic theories

We have seen algebraic theories separately characterizing nondeterministic and probabilistic choice. It is relatively straightforward to combine these two separate algebraic theories into one integrated theory incorporating both nondeterministic and probabilistic choice. No new operations are required; the operations of \textit{MonadAlt} and \textit{MonadProb} together suffice:

\[
\text{class } (\text{MonadAlt } m, \text{MonadProb } m) \Rightarrow \text{MonadAltProb } m
\]

This Haskell type class declaration is complete; it has an empty collection of additional methods, beyond those inherited from the superclasses \textit{MonadAlt} and \textit{MonadProb}. Implicitly, the laws of \textit{MonadAlt} and \textit{MonadProb} are also inherited; the only effort required is to consider the behaviour of interactions between the methods of the two superclasses. We stipulate that probabilistic choice distributes over nondeterministic:

\[
p \triangleleft w \triangleright (q \sqcup r) = (p \triangleleft w \triangleright q) \sqcup (p \triangleleft w \triangleright r)
\]

(This is not an uncontentious decision—some authors [55, 39] impose the opposite distributivity, of nondeterministic choice over probabilistic; we discuss this further in Sections 5.2 and 9.1.)

It turns out that there is a simple implementation of the combined interface, as finite non-empty sets of distributions. Again, we approximate finite sets by lists, for simplicity:

\[
\text{type } \text{Dists } a = [\text{Dist } a] \quad \text{-- nonempty lists}
\]

But the justification for this implementation is a little involved. The composition as functors \(F G\) of two monads \(F; G\) does not necessarily yield a monad: it is
Abstraction and Reasoning for Probabilistic Systems

It is straightforward to provide appropriate definitions of `return` and `fmap`, but not always possible to define `join` (or, equivalently, `>>=`). However, it is a standard result [2] that the composite \( F \circ G \) does form a monad if there is a ‘distributive law of \( G \) over \( F \)—that is, a natural transformation \( \text{swap}: G F \to F G \) satisfying certain coherence conditions. Given \( \text{swap} \), it is also straightforward to define \( \text{join} : F G F G \to F F G G \); that \( \text{join} \) satisfies the monad laws then follows from the coherence conditions on \( \text{swap} \).

In programming terms, we have to provide a distributive law of distributions over lists

\[
\text{swap} :: \text{Dist} [a] \to [\text{Dist} a] \quad \text{-- nonempty lists}
\]

satisfying the following four coherence conditions:

\[
\begin{align*}
\text{swap} \circ \text{fmap}_D \text{return}_L &= \text{return}_L \\
\text{swap} \circ \text{return}_D &= \text{fmap}_L \text{return}_D \\
\text{swap} \circ \text{fmap}_D \text{join}_L &= \text{join}_L \circ \text{fmap}_L \text{swap} \circ \text{swap} \\
\text{swap} \circ \text{join}_D &= \text{fmap}_L \text{join}_D \circ \text{swap} \circ \text{fmap}_D \text{swap}
\end{align*}
\]

(where, to be explicit about typing, we have subscripted each use of \( \text{return} \), \( \text{fmap} \), and \( \text{join} \) with \( L \) or \( D \) to indicate the list and distribution instances, respectively).

Then we can declare that the composite datatype \( \text{Dists} \) forms a monad, following the standard construction [2]:

\[
\text{instance Monad \text{Dists} where}
\begin{align*}
\text{return} \ x &= \text{return} \ (\text{return} \ x) \\
\text{fmap} \ f \ p &= \text{fmap} \ (\text{fmap} \ f) \ p \\
\text{join} \ pp &= \text{fmap} \ \text{join} \ (\text{join} \ (\text{map} \ \text{swap} \ pp))
\end{align*}
\]

A suitable definition of \( \text{swap} \) is as follows:

\[
\begin{align*}
\text{swap} &= \text{foldr}_1 \ \text{pick} \circ \text{map} \ \text{split} \quad \text{where} \\
\text{split} \ (xs, w) &= [\{(x, w)\} | x \leftarrow xs] \\
\text{pick} \ xds \ yds &= [xd \uplus \ yd | xd \leftarrow xds, yd \leftarrow yds]
\end{align*}
\]

(Here, \( \text{foldr}_1 \) is a variant of \( \text{foldr} \) for non-empty lists, taking only a binary operator and no starting value.) Informally, \( \text{swap} \) takes a distribution of nondeterministic choices to a nondeterministic choice of distributions, multiplying out all the possibilities; for example,

\[
\text{swap} \ (\{(1, 2), \frac{1}{3}\}, \{(3, 4), \frac{2}{3}\}) = \{(1, \frac{1}{3}), (3, \frac{2}{3})\}, \{(1, \frac{1}{3}), (4, \frac{2}{3})\}, \{(2, \frac{1}{3}), (3, \frac{2}{3})\}, \{(2, \frac{1}{3}), (4, \frac{2}{3})\})
\]

The composite monad \( \text{Dists} \) inherits \( \text{MonadAlt} \) and \( \text{MonadProb} \) functionality straightforwardly from its two component parts:

\[
\text{instance MonadAlt \text{Dists} where}
\begin{align*}
\text{p} \Box \text{q} &= \text{p } \uplus \text{ q}
\end{align*}
\]
instance MonadProb Dists where
\[ p \triangleright w \triangleright q = [xd \triangleright w \triangleright yd \mid xd \leftarrow p, yd \leftarrow q] \]

It is therefore an instance of the integrated theory of nondeterministic and probabilistic choice:

instance MonadAltProb Dists

Of course, we should check distributivity too; we return to this point in Section 5.2 below.

5.1 Example: mixing choices

Analogous to the fair coin, here is a biased coin:

\[
\text{coin'} :: \text{MonadProb m} \Rightarrow \text{Prob} \rightarrow m \text{ Bool} \\
\text{coin'} w = \text{return True} \triangleright w \triangleright \text{return False}
\]

(we write the parameter \(w\) as a subscript) and an arbitrary nondeterministic choice between booleans:

\[
\text{arb} :: \text{MonadAlt m} \Rightarrow m \text{ Bool} \\
\text{arb} = \text{return True} \of \text{return False}
\]

And here are two programs that each make an arbitrary choice and a probabilistic choice and compare them, but do so in different orders [12, 31]:

\[
\text{arbcoint, coinarb} :: \text{MonadAltProb m} \Rightarrow \text{Prob} \rightarrow m \text{ Bool} \\
\text{arbcoint} w = \text{do} \{ a \leftarrow \text{arb} ; c \leftarrow \text{coin'} w ; \text{return} (a \equiv c) \} \\
\text{coinarb} w = \text{do} \{ c \leftarrow \text{coin'} w ; a \leftarrow \text{arb} ; \text{return} (a \equiv c) \}
\]

Intuitively, because the probabilistic choice happens ‘first’ in coinarb, the nondeterministic choice can depend on it; whereas in arbcoint, the probabilistic choice happens ‘last’, so the nondeterministic choice cannot depend on it—and moreover, the probabilistic choice cannot depend on the nondeterministic either, because it would not be probabilistic if it did so. We can capture this intuition calculationally, using the equational theory of the two kinds of choice. On the one hand, we have:

\[
\text{arbcoint} w \\
= \left[ \text{definition of arbcoint} \right] \\
= \left[ \text{definition of arb} \right] \\
= \left[ \text{composition distributes leftwards over } \of \right] \\
= \left[ \text{left unit, booleans} \right]
\]
\[
\text{do} \{ c \leftarrow \text{coin}'_w; \text{return } c \} \quad \Box \quad \text{do} \{ c \leftarrow \text{coin}'_w; \text{return } (\neg c) \}
\]
\[
\text{coin}'_w \Box \text{coin}'_w
\]

On the other hand,
\[
\text{coinarb } w = \quad \text{definition of coinarb }
\]
\[
\text{do} \{ c \leftarrow \text{coin}_w; a \leftarrow \text{arb}; \text{return } (a \odot c) \}
\quad \text{definition of coin}_w
\]
\[
\text{do} \{ c \leftarrow (\text{return } \text{True } \odot w \triangleright \text{return } \text{False}); a \leftarrow \text{arb}; \text{return } (a \odot c) \}
\quad \text{composition distributes leftwards over } \odot \triangleright \]
\[
\text{do} \{ c \leftarrow \text{return } \text{True}; a \leftarrow \text{arb}; \text{return } (a \odot c) \odot w \triangleright
\]
\[
\text{do} \{ c \leftarrow \text{return } \text{False}; a \leftarrow \text{arb}; \text{return } (a \odot c) \}
\quad \text{left unit, booleans }
\]
\[
\text{do} \{ a \leftarrow \text{arb}; \text{return } a \} \odot \triangleright \text{do} \{ a \leftarrow \text{arb}; \text{return } (\neg a) \}
\quad \text{right unit; definition of arb }
\]
\[
(p \Box q) \odot \triangleright (p \Box q)
\quad \text{idempotence of } \odot \triangleright
\]
\[
\text{return } \text{True } \Box \text{return } \text{False}
\quad \text{commutativity of } \Box \quad \text{return } \text{False}
\]
\[
\text{return } \text{True } \Box \text{return } \text{False}
\quad \text{definition of arb }
\]
\[
\text{arb}
\]

That is, the nondeterminism in \text{arbcin} can be resolved only by choosing the distribution provided by \text{coin}'_w itself, or its opposite—the nondeterministic choice happens first, and depending on whether \text{True} or \text{False} is chosen, the probabilistic choice has chance either \text{w} or \overline{\text{w}} of matching it. In particular, if \text{w} = \frac{1}{2}, then the nondeterministic choice cannot influence the final outcome. But in \text{coinarb}, the probabilistic choice happens first, and the subsequent nondeterministic choice has complete freedom to enforce any outcome.

5.2 Convex closure

At the start of Section 5, we said that collections of distributions form a model of the combined theory \text{MonadAltProb}. In fact, this is not quite right: strictly speaking, there is no distributive law of distributions over sets [49], so the composition of the two monads is not a monad. Indeed, distribution of \text{\odot } \triangleright over \Box and idempotence of \text{\odot } \triangleright together imply a convexity property:

\[
p \Box q
\quad \text{idempotence of } \odot \triangleright; \text{arbitrary } w
\]
\[
(p \Box q) \odot w \triangleright (p \Box q)
\quad \text{distributing } \odot \triangleright \text{over } \Box
\]
\[
(p \odot w \triangleright p) \Box (q \odot w \triangleright q) \Box (p \odot w \triangleright q) \Box (q \odot w \triangleright q)
\]
That is, if any two distributions \( p \) and \( q \) are possible outcomes, then so is any convex combination \( p \triangleleft w \triangleright q \) of them. As a consequence, we should consider equivalence of collections of distributions \textit{up to convex closure}. In particular, for \( \text{coinarb} \) we have:

\[
\text{coinarb} \; v
\]  
\[
\text{return True} \; \square \; \text{return False}
\]
\[
\square \; \text{return False} \; \square \; \text{return True} \; \square \; \text{return True} \; \square \; \text{definition of coin}_w'
\]
\[
\square \; \text{return False} \; \square \; \text{return True} \; \square \; \text{coin}_w'
\]

and so the possible outcomes of \( \text{coinarb} \; v \) include all convex combinations of the four basic distributions \( \text{coin}_w', \text{return False}, \text{return True}, \) and \( \text{coin}_w \), which as it happens encompasses all possible distributions of the booleans.

This convexity intuition is computationally reasonable, if one considers repeated executions of a computation such as \( \text{coin}_1/2 \; \square \; \text{coin}_1/3 \). If the nondeterminism is always resolved in favour of the fair coin, the result will be heads half the time; if the nondeterminism is always resolved in favour of the biased coin, the result will be heads one third of the time. But if resolution of the nondeterminism alternates evenly between the two, the result will be heads five-twelfths of the time. Over repeated executions, any distribution between the two extremes can be obtained by some long-term strategy for resolving the nondeterminism; but no strategy will yield a distribution outside the two extremes.

One might wonder why one distributive law (of probabilistic choice over nondeterministic) should hold, while the other (of nondeterministic choice over probabilistic) need not. It turns out that the latter does not match intuitions about behaviour; for example, it is straightforward to calculate as follows:

\[
\text{return True} \; \square \; \text{return False}
\]
\[
\square \; \text{return False} \; \square \; \text{return True}
\]
\[
\square \; \text{return False}
\]
\[
\square \; \text{return True} \; \square \; \text{coin}_w'
\]

Informally, any straight probabilistic choice is inherently polluted with some taint of nondeterministic choice too. For example, letting \( w = 1/2 \) and \( w = 1/3 \) respectively, we can conclude that
This seems quite unfortunate, and so we do not require that nondeterministic choice distributes over probabilistic.

6 Monty Hall

As an extended example, we turn to the so-called Monty Hall Problem [45], which famously caused a controversy following its discussion in Marilyn vos Savant’s column in Parade magazine in 1990 [50]. Vos Savant described the problem as follows, quoting a letter from a reader, Craig F. Whitaker:

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?

Implicit in the above statement, the car is equally likely to be behind each of the three doors, the car is the prize and the goats are booby prizes, the host always opens a door, which always differs from the one you pick and always reveals a goat, and you always get the option to switch.

We might model this as follows. There are three doors:

data Door = A | B | C

doors :: [Door]
doors = [A, B, C]

First, Monty hides the car behind one of the doors, chosen uniformly at random:

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

Second, you pick one of the doors, also uniformly at random:

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

Third, Monty teases you by opening one of the doors—not the one that hides the car, nor the one you picked—to reveal a goat, choosing randomly among the one or two doors available to him:

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

(Here, the expression xs \ ys denotes the list of those elements of xs absent from ys.) Fourth, Monty offers you the choice between two strategies—either to switch to the door that is neither your original choice nor the opened one:
\[
\text{switch} :: \text{MonadProb } m \Rightarrow \text{Door} \to \text{Door} \to m \text{ Door} \\
\text{switch} \ p \ t = \text{return} \ (\text{head} \ (\text{doors} \ \setminus \ [p, t]))
\]

or to stick with your original choice:

\[
\text{stick} :: \text{MonadProb } m \Rightarrow \text{Door} \to \text{Door} \to m \text{ Door} \\
\text{stick} \ p \ t = \text{return} \ p
\]

In either case, you know \( p \) and \( t \)—but not \( h \), of course.

Here is the whole game, parametrized by your strategy, returning whether you win the car:

\[
\text{monty} :: \text{MonadProb } m \Rightarrow (\text{Door} \to \text{Door} \to m \text{ Door}) \to m \text{ Bool} \\
\text{monty strategy} = \begin{array}{l}
\text{do} \\
\{ h \leftarrow \text{hide}; \ p \leftarrow \text{pick}; \ t \leftarrow \text{tease} \ h \ p; \ s \leftarrow \text{strategy} \ p \ t; \ h \leftarrow s \star h; \ \text{return} \ (s \star h) \}
\end{array}
\]

We will show below that the switching strategy is twice as good as the sticking strategy:

\[
\text{monty switch} = \text{coin'}_{2/3} \\
\text{monty stick} = \text{coin'}_{1/3}
\]

The key is the fact that separate uniform choices are independent:

\[
\text{do} \ \{ h \leftarrow \text{hide}; \ p \leftarrow \text{pick}; \ \text{return} \ (h, p) \} \\
= \begin{array}{l}
\text{do} \ \{ h \leftarrow \text{uniform doors}; \ p \leftarrow \text{uniform doors}; \ \text{return} \ (h, p) \}
\end{array}
\]

and so we have

\[
\text{monty strategy} = \begin{array}{l}
\text{do} \\
(h, p) \leftarrow \text{uniform (cp doors doors)} \\
t \leftarrow \text{tease} \ h \ p \\
s \leftarrow \text{strategy} \ p \ t \\
\text{return} \ (s \star h)
\end{array}
\]

Clearly, the doors \( h \) and \( p \) independently chosen at random will match one third of the time:

\[
\text{do} \ \{ (h, p) \leftarrow \text{uniform (cp doors doors)}; \ \text{return} \ (p \star h) \}
\]

= \begin{array}{l}
\text{fmap and do notation}
\end{array}
\[
\text{do } \{ \; \texttt{b } \leftarrow \texttt{fmap (uncurry ()))) (uniform (cp doors doors)) ; \texttt{return b } \; \}
= \; \texttt{[ right unit ]}
\texttt{fmap (uncurry ()))) (uniform (cp doors doors))
= \; \texttt{[ naturality of uniform ]}
\texttt{uniform (map (uncurry ()))) (cp doors doors))
= \; \texttt{[ definitions of doors, cp, ]}
\texttt{uniform [ True, False, False, False, True, False, False, True]}
= \; \texttt{[ simplifying: three True, six False ]}
\texttt{uniform [ True, False, False]}
= \; \texttt{[ definitions of uniform, coin'}_w \texttt{]}
\texttt{coin'}_1/3
\]

Therefore we calculate:

\[
\text{monty stick}
= \; \texttt{[ definition of monty, independent uniform choices ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; s } \leftarrow \texttt{stick p t ; return (s } \neq \texttt{ h)} \; \}
= \; \texttt{[ definition of stick ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; s } \leftarrow \texttt{return p ; return (s } \neq \texttt{ h)} \; \}
= \; \texttt{[ left unit ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; return (p } \bullet \texttt{ h)} \; \}
= \; \texttt{[ t unused, and uniform side-effect-free, so tease can be eliminated ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{return (p } \equiv \texttt{ h)} \; \}
= \; \texttt{[ matching choices, as above ]}
\texttt{coin'}_1/3
\]

and

\[
\text{monty switch}
= \; \texttt{[ definition of monty, independent uniform choices ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; s } \leftarrow \texttt{switch p t ; return (s } \equiv \texttt{ h)} \; \}
= \; \texttt{[ definition of switch ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; s } \leftarrow \texttt{return (head (doors } \setminus \texttt{ [p, t]})) ; \texttt{return (s } \equiv \texttt{ h)} \; \}
= \; \texttt{[ left unit ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{t } \leftarrow \texttt{tease h p ; return (h } \equiv \texttt{ head (doors } \setminus \texttt{ [p, t]})) \; \}
= \; \texttt{[ case analysis on h } \equiv \texttt{ p—see below ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{if h } \equiv \texttt{ p then return False else return True} \; \}
= \; \texttt{[ booleans ]}
\texttt{do } \{ \; \texttt{(h, p) } \leftarrow \texttt{uniform (cp doors doors)} ; \texttt{return (h } \neq \texttt{ p)} \; \}
= \; \texttt{[ analogously, mismatching choices ]}
\texttt{coin'}_2/3
\]
Now for the two branches of the case analysis. For the case \( h = p \), we have:

\[
\begin{align*}
\text{do } & \{ t \leftarrow \text{tease } h \ p \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [p, t])) \} \\
= & \quad [\text{using assumption } h = p] \\
\text{do } & \{ t \leftarrow \text{tease } h \ p \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [h, t])) \} \\
= & \quad [h \text{ is not in } \text{doors} \setminus [h, t]] \\
\text{do } & \{ t \leftarrow \text{tease } h \ p \ ; \ \text{return False} \} \\
= & \quad [t \text{ unused, and uniform side-effect-free}] \\
\text{return False}
\end{align*}
\]

And for the case \( h \neq p \), we have:

\[
\begin{align*}
\text{do } & \{ t \leftarrow \text{tease } h \ p \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [p, t])) \} \\
= & \quad [\text{definition of } \text{tease}] \\
\text{do } & \{ t \leftarrow \text{uniform } (\text{doors} \setminus [h, p]) \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [p, t])) \} \\
= & \quad [h \neq p, \text{ so } \text{doors} \setminus [h, p] \text{ is a singleton; } \text{uniform } [a] = \text{return } a] \\
\text{do } & \{ t \leftarrow \text{return } (\text{head } (\text{doors} \setminus [h, p])) \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [p, t])) \} \\
= & \quad [\text{left unit}] \\
\text{do } & \{ \text{let } t = \text{head } (\text{doors} \setminus [h, p]) \ ; \ \text{return } (h \mapsto \text{head } (\text{doors} \setminus [p, t])) \} \\
= & \quad [h \neq p, \text{ and } t \neq h, p; \text{ so } t, h, p \text{ distinct}] \\
\text{do } & \{ \text{let } t = \text{head } (\text{doors} \setminus [h, p]) \ ; \ \text{return } (h \mapsto h) \} \\
= & \quad [t \text{ unused}] \\
\text{return True}
\end{align*}
\]

So when you and Monty make uniform probabilistic choices according to the rules of the game, switching wins two thirds of the time and sticking only one third.

### 6.1 Nondeterministic Monty

Perhaps a more faithful model of the Monty Hall problem is to allow Monty to make nondeterministic rather than probabilistic choices [31]—nobody said that Monty has to play fair. That is, Monty’s two moves in the game, hiding the car and teasing you, involve a nondeterministic rather than probabilistic choice among the available alternatives:

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

where

\[
\begin{align*}
\text{arbitrary} :: \text{MonadAlt } m \Rightarrow [a] \rightarrow m a \\
\text{arbitrary} = \text{foldr}_1 (\square) \circ \text{map return}
\end{align*}
\]

Then we define the game just as before, but with Monty behaving nondeterministically:
montry₂ :: MonadAltProb m ⇒ (Door → Door → m Door) → m Bool

montry₂ strategy = do { h ← hide₂ ; p ← pick ; t ← tease₂ h p ;
  s ← strategy p t ; return (s ≡ h) }

As it happens, making this change has no effect on the outcome. The first two choices—Monty’s choice of where to hide the car, and your initial choice of door—can still be combined, because composition distributes leftwards over nondeterministic choice:

do { h ← hide₂ ; p ← pick ; return (h, p) }
  = [ let k h = do { p ← pick ; return (h, p) } ]
do { h ← hide₂ ; k h }
  = [ definition of hide₂, arbitrary ]
do { h ← (return A □ return B □ return C) ; k h }
  = [ composition distributes leftwards over □ ]
do { h ← return A ; k h □ do { h ← return B ; k h □ do { h ← return C ; k h }
  = [ left unit ]
k A □ k B □ k C
  = [ definition of k ]
do { p ← pick ; return (A, p) □ do { p ← pick ; return (B, p) □ do { p ← pick ; return (C, p) }

The remainder of the reasoning proceeds just as before. It is still the case that doors h and p will match one third of the time, even though h is now chosen nondeterministically rather than probabilistically. For brevity, let

try d = do { p ← pick ; return (d, p) }

Then we have

\[
\text{fmap } (\text{uncurry } (\equiv)) (\text{try } d)
\]

\[
= [ \text{definition of try; fmap and do notation } ]
do { p ← \text{pick} ; \text{return (d }\equiv p\text{) } ]
\]

\[
= [ \text{definition of pick } ]
do { p ← (\text{return A}^{1/3}, \text{return B}^{1/3}, \text{return C}^{1/3}) ; \text{return (d }\equiv p\text{) } ]
\]

\[
= [ \text{composition distributes leftwards over } \triangleright\triangleright; \text{right unit } ]
\]

\[
\triangleright\triangleright (\text{return (d }\equiv A)^{1/3}, \text{return (d }\equiv B)^{1/3}, \text{return (d }\equiv C)^{1/3})
\]

\[
= [ \text{d : Door, and so d is one of A, B, C } ]
\]

\[
\text{coin'}^1_{\text{w}}
\]

and therefore

\[
do { h ← \text{hide}_2 ; p ← \text{pick} ; \text{return (h }\equiv p\text{) } ]
\]

\[
= [ \text{fmap and do notation } ]
\]
\[
\text{fmap \ (uncurry \ (\equiv)) \ (\textbf{do} \ \{ \ h \leftarrow \ \text{hide}_n ; \ p \leftarrow \ \text{pick} ; \ \text{return} \ (h, p) \})} \\
= \ \left[ \ \text{combining first two choices, as above} \ \right] \\
\text{fmap \ (uncurry \ (\equiv)) \ (\text{try} \ A \ \square \ \text{try} \ B \ \square \ \text{try} \ C) } \\
= \ \left[ \ \text{naturality:} \ \text{fmap} \ f \ (p \ \square \ q) = \ \text{fmap} \ f \ p \ \square \ \text{fmap} \ f \ q \ \right] \\
\text{fmap \ (uncurry \ (\equiv)) \ (\text{try} \ A) \ \square \ \text{fmap} \ (uncurry \ (\equiv)) \ (\text{try} \ B) \ \square \ \text{fmap} \ (uncurry \ (\equiv)) \ (\text{try} \ C) } \\
= \ \left[ \ \text{matching choices, above} \ \right] \\
\text{coin}_{1/3} \ \square \ \text{coin}_{1/3} \ \square \ \text{coin}_{1/3} \\
= \ \left[ \ \text{idempotence of} \ \square \ \right] \\
\text{coin}_{1/3}
\]

and the conclusion is still that

\[
\text{monty}_n \ \text{switch} = \ \text{coin}_{2/3}^n \\
\text{monty}_n \ \text{stick} = \ \text{coin}_{1/3}^n
\]

Combining two classes of effect—here, probability and nondeterminism—did not make the reasoning any more difficult that in was with a single such class.

## 7 Failure is an option

Computations that may fail, and whose failures can be handled, are characterized by two operations for throwing and catching exceptions:

\[
\begin{align*}
\text{class} & \quad \text{Monad} \ m \Rightarrow \text{MonadExcept} \ m \ \text{where} \\
& \quad \text{throw} :: \ m \ a \\
& \quad \text{catch} :: \ m \ a \rightarrow m \ a
\end{align*}
\]

For simplicity, we suppose that there is only a single exception, just as we assumed a single updatable location for stateful computations; the model is easily extended to cover multiple exceptions. The intuition is that \text{throw} is the computation that immediately fails, and that \text{p `catch` q} represents the computation that behaves like \text{p}, except if this fails, in which case it continues as the exception handler \text{q}. (In Haskell, backquotes turn a prefix function into an infix operator; so \text{p `catch` q} = \text{catch p q}.) We stipulate that \text{throw} is a left zero of composition, so that a failure discards the subsequent part of a computation:

\[
\text{do} \ \{ \ x \leftarrow \ \text{throw} ; \ k \ x \} = \ \text{throw}
\]

We do not stipulate that \text{throw} is a right zero of composition; that would require that a failure also discards the preceding part of the computation, and so that any effects of that part would have to be rolled back—quite a strong condition. Neither do we stipulate that composition distributes over \text{catch}; in general,

\[
\text{do} \ \{ \ x \leftarrow \ (p`catch` q) ; \ k \ x \} \neq \text{do} \ \{ \ x \leftarrow \ p ; \ k \ x \} `catch` \text{do} \ \{ \ x \leftarrow \ q ; \ k \ x \}
\]

because the right-hand side brings exceptions raised by the first \text{k} under the influence of the handler \text{q}, whereas exceptions of \text{k} on the left are not handled.
We do stipulate that \textit{return} is a left zero of \textit{catch}, so that pure computations never fail:
\[
\text{return } x \cdot \text{catch} \cdot p = \text{return } x
\]

Finally, \textit{throw} and \textit{catch} form a monoid:
\[
\begin{align*}
    p \cdot \text{catch} \cdot \text{throw} &= p \\
    \text{throw} \cdot \text{catch} \cdot p &= p \\
    p \cdot \text{catch} \cdot (q \cdot \text{catch} \cdot r) &= (p \cdot \text{catch} \cdot q) \cdot \text{catch} \cdot r
\end{align*}
\]

That is: an exception handler that immediately propagates the exception has no effect; failures are indeed caught and handled; and exception handlers can be chained in sequence, with control passing along the chain as failures happen.

One obvious implementation of exceptions is via lifting:

\textbf{data} \textit{Maybe} \textit{a} = \textit{Just} \textit{a} | \textit{Nothing}

Values are lifted into pure computations via \textit{Just} and passed along by composition, whereas \textit{Nothing} forms a left zero of composition:

\textbf{instance} \textit{Monad} \textit{Maybe} \textbf{where}
\[
\begin{align*}
    \text{return } x &= \text{Just } x \\
    \text{Just } x \gg k &= k x \\
    \text{Nothing} \gg k &= \text{Nothing}
\end{align*}
\]

Of course, \textit{Nothing} represents failure; \textit{Just} and \textit{Nothing} partition computations into entirely pure ones and entirely failing ones, which form a left zero and a left unit of \textit{catch}, respectively.

\textbf{instance} \textit{MonadExcept} \textit{Maybe} \textbf{where}
\[
\begin{align*}
    \text{throw} &= \text{Nothing} \\
    \text{Just } x \cdot \text{catch} \cdot q &= \text{Just } x \\
    \text{Nothing} \cdot \text{catch} \cdot q &= q
\end{align*}
\]

The names \textit{Maybe}, \textit{Just}, and \textit{Nothing} were coined by Spivey [46], and he gives a number of examples of equational reasoning about term rewriting operations that might fail. (In ML, the datatype analogous to \textit{Maybe} \textit{a} is written \textit{a} \textit{option}.)

\subsection{7.1 Combining probability with exceptions}

Just as we did for nondeterministic and probabilistic choice, we can quite easily combine the theories of probability and exceptions. We simply combine the two interfaces, adding no new operations:

\textbf{class} (\textit{MonadExcept} \textit{m}, \textit{MonadProb} \textit{m}) \Rightarrow \textit{MonadProbExcept} \textit{m}

The laws of exceptions and of probability are inherited; the only effort required is to consider the interaction between the two theories. In this case, we have one
additional distributivity law, which intuitively states that making a probabilistic choice cannot of itself cause a failure, so \texttt{catch} distributes over it:

\[(p \triangleright w \triangleright q) \cdot \texttt{catch} \cdot r = (p \cdot \texttt{catch} \cdot r) \triangleright w \triangleright (q \cdot \texttt{catch} \cdot r)\]

The same representation works as for plain probability distributions, but now we allow the weights to sum to less than one, and the list of weighted elements to be empty—these are sometimes called \textit{subdistributions} [31] or \textit{evaluations} [20]:

\[
\text{weight} :: \text{Dist} \ a \rightarrow \text{Prob}
\]

\[
\text{weight} \ p = \text{sum} \ [w \mid (x, w) \leftarrow p]
\]

\[
\text{instance MonadExcept Dist where}
\]

\[
\text{throw} = []
\]

\[
p \cdot \texttt{catch} \cdot q = p \uplus \text{scale} \ (1 - \text{weight} \ p) \ (q)
\]

\[
\text{instance MonadProbExcept Dist}
\]

Operationally, the exceptions are represented by the ‘missing’ bits of the distribution; for example, the subdistribution \([((\text{True}, \frac{1}{2}), (\text{False}, \frac{1}{4}))\) has weight \(\frac{3}{4}\), and represents a computation that fails the remaining \(\frac{1}{4}\) of the time. The correctness of this implementation depends on the congruences we have imposed on distributions, specifically to ignore reordering and zero-weighted elements, and to coalesce duplicates.

For example, here is an attempt to simulate the biased \texttt{coin}\(_{2/3}\) using two fair coin tosses, motivated by Knuth and Yao’s trick [23] for simulating a fair die with three coins:

\[
\text{coins23} :: \text{MonadProbExcept} \ m \Rightarrow m \ \text{Bool}
\]

\[
\text{coins23} = \text{return True} \triangleright \frac{1}{2} \uplus (\text{return False} \triangleright \frac{1}{2} \triangleright \text{throw})
\]

We might illustrate the process as follows:

![Diagram](image)

This does indeed yield \texttt{True} exactly twice as often as \texttt{False}. However, a quarter of the time it will fail to yield any result at all, and throw an exception instead; so the attempt was not entirely successful. We will pick up this example again in Section 8 below.

### 7.2 Forgetful Monty

Let us return to the purely probabilistic version of the Monty Hall game, but this time suppose that Monty is becoming increasingly forgetful as he turns sixty—he can never remember where he has hidden the car [45, Chapter 3]. Therefore,
when it comes to teasing you, he uniformly at random opens one of the two
doors different from the door you picked. Of course he might accidentally reveal
the car in doing so; we treat this as a failure in the protocol:

\[
\text{tease}_f :: \text{MonadProbExcept } m \Rightarrow \text{Door} \rightarrow \text{Door} \rightarrow m \text{ Door}
\]

\[
tease_f h p = \text{do}
\begin{align*}
    & t \leftarrow \text{uniform } (\text{doors } \setminus \{ p \}) ; \\
    & \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
\text{monty}_f :: \text{MonadProbExcept } m \Rightarrow (\text{Door} \rightarrow \text{Door} \rightarrow m \text{ Door}) \rightarrow m \text{ Bool}
\]

\[
\text{monty}_f \text{ strategy} = \text{do}
\begin{align*}
    & h \leftarrow \text{hide} ; \\
    & p \leftarrow \text{pick} ; \\
    & t \leftarrow \text{tease}_f h p ; \\
    & s \leftarrow \text{strategy } p \ t ; \\
    & \text{return } (s = h)
\end{align*}
\]

Investigating \(\text{tease}_f\) in the case that \(h = p\), we have:

\[
\text{tease}_f h p
\]

\[
= \begin{bmatrix}
\text{definition of } \text{tease}_f \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{uniform } (\text{doors } \setminus \{ p \}) ; \\
& \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
= \begin{bmatrix}
h = p, \text{ so } h \text{ is not in } \text{doors } \setminus \{ p \} \text{ and hence } t \neq h \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{uniform } (\text{doors } \setminus \{ p \}) ; \\
& \text{return } t
\end{align*}
\]

\[
= \begin{bmatrix}
\text{right unit} \end{bmatrix} \begin{align*}
\text{uniform } (\text{doors } \setminus \{ p \})
\end{align*}
\]

In the case that \(h \neq p\), let \(d = \text{head } (\text{doors } \setminus \{ h, p \})\), so that \(h, p, d\) are all
distinct; then we have:

\[
\text{tease}_f h p
\]

\[
= \begin{bmatrix}
\text{definition of } \text{tease}_f \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{uniform } (\text{doors } \setminus \{ p \}) ; \\
& \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
= \begin{bmatrix}
\text{by assumption, } \text{doors } \setminus \{ p \} = \{ h, d \} \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{uniform } \{ h, d \} ; \\
& \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
= \begin{bmatrix}
\text{definition of } \text{uniform} \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{return } h \triangleleft \frac{1}{2} \triangleright \text{return } d ; \\
& \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
= \begin{bmatrix}
\text{composition distributes leftwards over } \triangleleft \end{bmatrix} \begin{align*}
\text{do } & \{ t \leftarrow \text{return } h ; \\
& \text{if } t = h \text{ then throw else return } t\} \triangleleft \frac{1}{2} \triangleright \\
\text{do } & \{ t \leftarrow \text{return } d ; \\
& \text{if } t = h \text{ then throw else return } t
\end{align*}
\]

\[
= \begin{bmatrix}
\text{left unit} \end{bmatrix} \begin{align*}
\text{(if } h = h \text{ then throw else return } h) \triangleleft \frac{1}{2} \triangleright \text{(if } d = h \text{ then throw else return } d)
\end{align*}
\]

\[
= \begin{bmatrix}
\text{by assumption, } d \neq h; \text{ conditionals} \end{bmatrix} \begin{align*}
\text{throw } \triangleleft \frac{1}{2} \triangleright \text{return } d
\end{align*}
\]

and so

\[
\text{tease}_f h p = \text{if } h = p \text{ then uniform } (\text{doors } \setminus \{ p \})
\]

\[
\text{else } (\text{throw } \triangleleft \frac{1}{2} \triangleright \text{return } (\text{head } (\text{doors } \setminus \{ h, p \})))
\]

Clearly, in the ‘else’ case, \(\text{tease}_f\) is no longer necessarily side-effect-free—even
if its result is not used, it cannot be discarded, because it might fail—and so
the calculations for the purely probabilistic version of the game do not apply. Instead, we have:

\[
\text{monty}_{f} \text{ stick} = \begin{cases} 
\text{ definition of } \text{monty}_{f} \\
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad t \leftarrow \text{tease}_{f} h \ p \ ; \ s \leftarrow \text{stick p t} \ ; \ \text{return} \ (s = h) \} 
\end{cases}
\]

\[
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad t \leftarrow \text{tease}_{f} h \ p \ ; \ \text{return} \ (p = h) \}
\]

\[
\text{case analysis in } \text{tease}_{f}, \text{ as above; let } d = \text{head (doors } \setminus \{h, p\}) 
\]

\[
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad t \leftarrow \text{if } h = p \text{ then uniform (doors } \setminus \{p\}) \text{ else } \text{throw } \frac{1}{2} \ ; \ \text{return} \ d \\
\quad \text{return} \ (p = h) \}
\]

\[
\text{first } t \text{ is unused and } \text{uniform} \text{ is side-effect-free; } p = h = \text{True} 
\]

\[
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad \text{if } h = p \text{ then } \text{return} \text{True} \\
\quad \text{else do} \{ t \leftarrow \text{throw } \frac{1}{2} \ ; \ \text{return} \ d \\
\text{return} \ (p = h) \} \}
\]

\[
\text{composition distributes over } \triangleleft \triangleright 
\]

\[
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad \text{if } h = p \text{ then return } \text{True} \\
\quad \text{else do} \{ t \leftarrow \text{throw} ; \text{return} \text{False} \} \triangleleft \frac{1}{3} \\
\quad \text{do} \{ t \leftarrow \text{return} d ; \text{return} \text{False} \} \}
\]

\[
\text{throw is a left zero of composition, and return a left unit} 
\]

\[
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad \text{if } h = p \text{ then return } \text{True} \text{ else throw } \frac{1}{2} \ ; \ \text{return} \text{False} \}
\]

\[
\text{matching choices, as in Section 6} 
\]

\[
\text{do} \{ b \leftarrow \text{coin}'\frac{1}{3} ; \text{if } b \text{ then return } \text{True} \text{ else throw } \frac{1}{2} \ ; \ \text{return} \text{False} \}
\]

\[
\text{composition distributes over } \triangleleft \triangleright 
\]

\[
\text{do} \{ b \leftarrow \text{return } \text{True} \\
\quad \text{if } b \text{ then return } \text{True} \text{ else throw } \frac{1}{2} \ ; \ \text{return} \text{False} \} \triangleleft \frac{1}{3} \\
\text{do} \{ b \leftarrow \text{return } \text{False} \\
\quad \text{if } b \text{ then return } \text{True} \text{ else throw } \frac{1}{2} \ ; \ \text{return} \text{False} \}
\]

\[
\text{left unit; conditionals} 
\]

\[
\text{return } \text{True} \triangleleft \frac{1}{3} \triangleright (\text{throw } \frac{1}{2} \ ; \ \text{return} \text{False}) 
\]

\[
\text{flattening choices} 
\]

\[
(\text{True} \triangleleft \frac{1}{3}, \text{throw} \triangleleft \frac{1}{3}, \text{False} \triangleleft \frac{1}{3})
\]

On the other hand:

\[
\text{monty}_{f} \text{ switch} = \begin{cases} 
\text{ definition of } \text{monty}_{f} \\
\text{do} \{ (h, p) \leftarrow \text{uniform (cp doors doors)} \\
\quad t \leftarrow \text{tease}_{f} h \ p ;
\end{cases}
\]
\[ s \leftarrow \text{switch } p \ t ; \ \text{return } (s \Rightarrow h) \]  
\[ = [ \text{definition of switch} ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \ t \leftarrow \text{tease} h \ p ; \]
\[ s \leftarrow \text{return } (\text{head } (\text{doors } \setminus [p, t])); \ \text{return } (s \Rightarrow h) \]
\[ = [ \text{left unit } ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \ t \leftarrow \text{tease} h \ p ; \]
\[ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \]
\[ = [ \text{case analysis in } \text{tease}, \text{as above; let } d = \text{head } (\text{doors } \setminus [h, p]) ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ t \leftarrow \text{if } h \Rightarrow p \text{ then uniform } (\text{doors } \setminus [p]) \text{ else throw } \frac{1}{2} \Rightarrow \text{return } d ; \]
\[ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \]
\[ = [ \text{composition distributes leftwards over conditional } ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then do } \{ t \leftarrow \text{uniform } (\text{doors } \setminus [p]); \]
\[ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ \text{else do } \{ t \leftarrow \text{throw } \frac{1}{2} \Rightarrow \text{return } d ; \]
\[ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ = [ \text{in then branch, } h = p, \text{ so } h \text{ is not in } \text{doors } \setminus [p, t] ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then do } \{ t \leftarrow \text{uniform } (\text{doors } \setminus [p]); \ \text{return } \text{False} \}
\[ \text{else do } \{ t \leftarrow \text{throw } \frac{1}{2} \Rightarrow \text{return } d ; \]
\[ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ = [ \text{composition distributes over } \Rightarrow ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then do } \{ t \leftarrow \text{uniform } (\text{doors } \setminus [p]); \ \text{return } \text{False} \}
\[ \text{else do } \{ t \leftarrow \text{throw } ; \ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ \text{do } \{ t \leftarrow \text{return } d ; \ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ = [ \text{in then branch, } t \text{ is unused and } \text{uniform } \text{is side-effect-free } ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then return } \text{False } \]
\[ \text{else do } \{ t \leftarrow \text{throw } ; \ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ \text{do } \{ t \leftarrow \text{return } d ; \ \text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, t])); \}
\[ = [ \text{throw is left zero, return is left unit } ] \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then return } \text{False } \]
\[ \text{else throw } \frac{1}{2} \Rightarrow (\text{return } (h \Rightarrow \text{head } (\text{doors } \setminus [p, d]))); \]
\[ \text{do } \{ (h, p) \leftarrow \text{uniform } (cp \ \text{doors } \text{doors}) ; \]
\[ \text{if } h \Rightarrow p \text{ then return } \text{False } \text{else throw } \frac{1}{2} \Rightarrow (\text{return } \text{True}) \]
\[ = [ \text{matching choices } ] \]
\[ \text{do } \{ b \leftarrow \text{coin} \frac{1}{3}; \text{if } b \text{ then return } \text{False } \text{else throw } \frac{1}{2} \Rightarrow \text{return } \text{True} \}
\[ = [ \text{composition distributes over } \Rightarrow ] \]
\[ \text{do } \{ b \leftarrow \text{return } \text{True} ; \]
\[ \text{if } b \text{ then return } \text{False } \text{else throw } \frac{1}{2} \Rightarrow \text{return } \text{True} \} \frac{1}{3} \Rightarrow \]
\[ \text{do } \{ b \leftarrow \text{return } \text{False} ; \]
if \( b \) then \( \text{return False} \)
else \( \text{throw } \frac{1}{2} \Rightarrow \text{return True} \}

return \( \frac{1}{3} \) \( \Rightarrow \) \( \text{throw } \frac{1}{2} \Rightarrow \text{return False} \)

\( \Rightarrow \) \( \text{flattening choices} \)

\( \{(\text{False}@\frac{1}{3}, \text{throw}@\frac{1}{3}, \text{True}@\frac{1}{3})\} \)

So, somewhat surprisingly, both the sticking and switching strategies are equivalent in the face of Monty’s forgetfulness: with either one, you have equal chances of winning, losing, or of the game being aborted.

\( \text{monty}\_\text{stick} = \text{monty}\_\text{switch} = (\text{True}@\frac{1}{3}, \text{False}@\frac{1}{3}, \text{throw}@\frac{1}{3}) \)

### 7.3 Conditional probability

We have so far presented the combination of probability and exceptions as modelling computations that make probabilistic choices, but that might fail. An alternative reading is in terms of conditional probability. In probability theory, the conditional probability \( P(A|B) \) is the probability of event \( A \) occurring, given that event \( B \) is known to have occurred; it is defined to be the probability of both events \( A \) and \( B \) occurring, divided by the probability of \( B \) alone:

\[
P(A | B) = \frac{P(A \land B)}{P(B)}
\]

Operationally, a subdistribution \( p \) with weight \( w > 0 \) represents the distribution of outcomes drawn from the normalized distribution \( \text{scale } (\frac{1}{w}) p \), conditioned by the outcomes outside the support of \( p \) not occurring.

\[
\text{normalize :: } \text{Dist } a \rightarrow \text{Dist } a
\]

\[
\text{normalize } p = \text{scale } (1 / \text{weight } p) (p)
\]

For example, the subdistribution

\[
\text{coins}\_23 = [(\text{True}, \frac{1}{2}), (\text{False}, \frac{1}{4})]
\]

from Section 7.1 has weight \( \frac{3}{4} \), and so represents the distribution of outcomes

\[
[(\text{True}, \frac{1}{2} \div \frac{3}{4} = \frac{2}{3}), (\text{False}, \frac{1}{4} \div \frac{3}{4} = \frac{1}{3})] = \text{coin'}\_2/3
\]

given that one does not toss two tails—so the attempt to simulate the biased \( \text{coin'}\_2/3 \) using two fair coin tosses was not so far off after all. Similarly, one could say that playing against forgetful Monty using either the switching or the sticking strategy yields a 50–50 chance of winning, assuming that Monty successfully bluff his way through his amnesia.

As a more extended example, consider the canonical ‘wet grass’ Bayesian reasoning problem [21]. Suppose that with probability \( \frac{3}{10} \) it is raining; and when it rains, with probability \( \frac{9}{10} \) it does so heavily enough to make the grass
wet. Also, there is a lawn sprinkler, operating with probability $\frac{1}{2}$; when this operates, with probability $\frac{8}{10}$ it has high enough water pressure to make the grass wet. Finally, with probability $\frac{1}{10}$ the grass is wet for some other unknown reason.

\begin{verbatim}
rain :: MonadProb m => m Bool
rain = coin' \frac{3}{10}

sprinkler :: MonadProb m => m Bool
sprinkler = coin' \frac{1}{2}

grassWet :: MonadProb m => Bool -> Bool -> m Bool
grassWet r s = do { x ← coin' \frac{9}{10} ; y ← coin' \frac{8}{10} ; z ← coin' \frac{1}{10} ;
return ((x ∧ r) ∨ (y ∧ s) ∨ z) }
\end{verbatim}

What is the probability that it is raining, given that the grass is observed to be wet?

\begin{verbatim}
experiment :: MonadProbExcept m => m Bool
experiment = do { r ← rain ; s ← sprinkler ; g ← grassWet r s ;
  if g then return r else throw }
\end{verbatim}

We simply return whether it is raining, conditioned on whether the grass is wet:

\begin{verbatim}
normalize experiment = [(False, 1610 \frac{3029}{}), (True, \frac{1419}{3029})]
\end{verbatim}

— that is, it is raining with probability $\frac{1419}{3029} \simeq 0.47$.

8 Recursion

It is very tempting to write recursive programs in the style we have shown above. For example, here is a simple Markov model of the coins23 attempt in Section 7.1:

\begin{center}
\begin{tikzpicture}
  \node (head) at (0,0) {True};
  \node (tail) at (1,0) {False};
  \draw[->] (head) -- node [above] {head} (tail);
  \draw[->] (tail) -- node [above] {head} (head);
\end{tikzpicture}
\end{center}

This depicts a process that recurses instead of failing. We can represent it symbolically with a recursive definition:

\begin{verbatim}
thirds :: MonadProbExcept m => m Bool
thirds = coins23 'catch' thirds
\end{verbatim}

In fact, we can inline the exception and its handler, on account of the various laws of $\ll$:$\gg$: for any $p$,
coins \(23\) ‘catch’ \(p\)

\[
= \begin{cases} 
\text{definition of coins}\!23 \vrule \cr
\left(\text{return } \text{True} \triangleleft 1/2 \triangleright (\text{return False} \triangleleft 1/2 \triangleright \text{throw})\right) \text{‘catch’ } p
\end{cases}
\]

\[
= \begin{cases} 
\text{catch distributes over } \triangleleft \triangleright \vrule \cr
\left(\text{return True ‘catch’ } \text{thirds} \right) \triangleleft 1/2 \triangleright
\left(\text{return False ‘catch’ } \text{thirds} \right) \triangleleft 1/2 \triangleright \left(\text{throw ‘catch’ } p\right)
\end{cases}
\]

\[
= \begin{cases} 
\text{return is left zero of } \text{catch, and } \text{throw is left unit} \vrule \cr
\text{return True } \triangleleft 1/2 \triangleright \left(\text{return False } \triangleleft 1/2 \triangleright \text{p}\right)
\end{cases}
\]

So we could have defined instead

\[
\text{thirds :: MonadProb } m \Rightarrow m \text{ Bool}
\]

\[
\text{thirds } = \text{return True } \triangleleft 1/2 \triangleright \left(\text{return False } \triangleleft 1/2 \triangleright \text{thirds}\right)
\]

Note that this definition can be given a more specific type, because it no longer depends on the ability to fail.

### 8.1 Recursive definitions

But what might the semantics of such a recursive definition be? Up until now, we have implicitly assumed a setting of sets and total functions, in which there is no guarantee that an arbitrary recursive definition has a canonical solution. And indeed, with the implementation of MonadProb in terms of finite probability-weighted lists from Section 4, the recursive equation defining thirds has no solution.

The usual response to this problem is to switch the setting from total functions between sets to continuous functions between complete partial orders; then all recursive definitions have a least solution. In this case, the least solution to the recursive definition of thirds above is the infinite probability-weighted list

\[
\left[(\text{True} , 1/2), (\text{False}, 1/4), (\text{True}, 1/8), (\text{False}, 1/16), (\text{True}, 1/32), (\text{False}, 1/64), \ldots\right]
\]

One might see this as a reasonable representation of \(\text{coin}'_{2/3}\): the weights of finite initial segments of the list converge to one; and two thirds of the weight is associated with True, one third with False. Moreover, if one samples from this distribution in the obvious way, with probability 1 one obtains an appropriate result—only in the measure-zero situation in which one tries to sample the distribution at precisely 1.0 does the computation not terminate. In fact, one can show that \(\text{coin}'_{2/3}\) is a solution to the same recursive equation as thirds:

\[
\text{return True } \triangleleft 1/2 \triangleright \left(\text{return False } \triangleleft 1/2 \triangleright \text{coin}'_{2/3}\right)
\]

\[
= \begin{cases} 
\text{definition of coin}'_{\omega} \vrule \cr
\left(\text{return True } \triangleleft 1/2 \triangleright \left(\text{return False } \triangleleft 1/2 \triangleright \text{return False}\right)\right)
\end{cases}
\]

\[
= \begin{cases} 
\text{flattening choices} \vrule \cr
\langle\text{return True}'_{\omega}/2, \text{return False}'_{\omega}/4, \text{return True}'_{\omega}/6, \text{return False}'_{\omega}/12\rangle
\end{cases}
\]

\[
= \begin{cases} 
\text{combining collisions} \vrule \cr
\langle\text{return True}'_{\omega}/3, \text{return False}'_{\omega}/3\rangle
\end{cases}
\]
However, the nice behaviour in this case is a happy accident of the rather special form of the recursive definition. Had we written the ostensibly equivalent definition

\[
\text{thirds} = (\text{thirds} \triangleright \return \text{False}) \triangleright \return \text{True}
\]

instead, the least solution would be \( \bot \), the least-defined element in the information ordering, because the definition of \( \triangleright \) on weighted lists is strict in its left argument. In fact, the codatatype of possibly-infinite, possibly-partial lists fails to satisfy all the necessary axioms: choice is not quasi-commutative when it involves undefined arguments, because lists are left-biased.

### 8.2 A free monad for choice

As we have seen, one obvious implementation of MonadProb uses probability-weighted lists of values; thus, coin is represented as \([((\text{True}, \frac{1}{2}), (\text{False}, \frac{1}{2}))]\). However, an arguably more natural representation is in terms of the free monad—‘more natural’ in the sense that it arises directly from the signature of the \( \triangleright \) operation:

\[
data \text{DistT a} = \text{Return a} | \text{Choice Prob \text{DistT a} (\text{DistT a})}
\]

This being a free monad, return and bind have simple definitions arising from substitution:

\[
\begin{align*}
\text{instance Monad DistT where} \\
\text{return } x &= \text{Return } x \\
\text{Return } x \gg k &= k x \\
\text{Choice } w p q \gg k &= \text{Choice } w (p \gg k) (q \gg k)
\end{align*}
\]

and by design, \( \triangleright \) is trivial to implement:

\[
\text{instance MonadProb DistT where} \\
p \triangleright w \triangleright q = \text{Choice } w p q
\]

Again, because this is the free monad, the monad laws necessarily hold—the left unit law holds directly by construction, and the right unit and associativity laws are easily shown by induction over the structure of the left-hand argument. This is not quite the initial model of the MonadProb specification, though, because the remaining identity, idempotence, quasi-commutativity, and quasi-associativity laws of \( \triangleright \) do not hold. Indeed, as we have already argued, the initial model of the specification consists of finite mappings from elements to probabilities, collected from the choice tree in the obvious way:
\[
\text{collectT :: } \text{Eq } a \Rightarrow \text{DistT } a \rightarrow (a \rightarrow \text{Prob}) \\
\text{collectT (Return } x \text{) } y = \text{if } x = y \text{ then } 1 \text{ else } 0 \\
\text{collectT (Choice } w \text{ } p \text{ } q \text{) } y = w \times \text{collectT } p \text{ } y + \overline{w} \times \text{collectT } q \text{ } y
\]

and equivalences on \( \text{DistT} \) ought again to be taken modulo permutations, zero-weighted elements, and repeated elements (whose weights should be added). Nevertheless, the datatype \( \text{DistT} \) itself provides another convenient approximation to the initial model.

In order to guarantee solutions to arbitrary recursive definitions, we still have to accept the CPO setting rather than SET; so really, we mean to take the codatatype interpretation of \( \text{DistT} \), including partial and infinite values as well as finite ones. (Which means that the inductive proofs of the monad laws need to be strengthened to cover these cases.)

The benefit we gain from this extra complication is that the representation is now symmetric, and so the laws of choice hold once more. Recursive definitions like those of thirds above give rise to regular choice trees—possibly infinite trees, but with only finitely many different subtrees. Moreover, \( \triangleright\triangleright \) is non-strict in both arguments, so the semantics of recursive definitions is much less sensitive to the precise form of the equation.

But of course, not all recursive equations give productive solutions. Clearly anything is a solution of the equation \( p = p \), or, thanks to the monad laws, of \( p = (p \triangleright return) \). Even some equations that completely determine proper choice trees do not define productive sampling behaviour; for example,

\[
p = p \triangleright 1 \triangleright return \text{ True}
\]

does not, and even when restricting attention to weights strictly between 0 and 1, one can try

\[
p = p \triangleright 1/2 \triangleright p
\]

We believe that for a recursive definition to define a choice tree with a productive interpretation as a sampling function, it is sufficient for recursive occurrences of the variable being defined to be guarded by a \( \triangleright\triangleright \), and for each such \( \triangleright\triangleright \) to devote positive weight to a non-recursive subcomputation. But it seems very unsatisfactory to have to consider specific implementations such as \( \text{DistT} \) of the \( \text{MonadProb} \) specification at all; in the rest of the paper, we have managed to conduct all our reasoning in terms of the algebraic theory rather than one of its models.

9 Conclusions

We have presented an approach to reasoning about effectful functional programs, based on algebraic theories. We have focussed in this paper on the effects of nondeterminism and probabilism and their combination, in honour of Carroll Morgan, but the approach works for any class of effects; our earlier paper [10] also
discusses counting, generating fresh names, mutable state, and exceptions. The
problem of reasoning about effectful programs was an open one in the functional
programming community before this point; for example, our work was inspired
by a paper concerning a relabelling operation on trees [18], which resorted to
unrolling the obvious stateful program into a pure state-transforming function
in order to conduct proofs.

One strength of functional programming is the support it provides for rea-
soning about programs: the techniques of simple high-school algebra suffice, and
one can reason directly in the language of programs, rather than having to ex-
tract verification conditions from the programs and then reason indirectly in
predicate calculus. In that respect, functional programming has a similar moti-
vation to Hoare’s ‘programs are predicates’ work [16], Hehner’s predicative pro-
gramming [13], and Morgan’s refinement calculus [37]—namely, to avoid where
possible the distinction between syntax and semantics, and to remove the layer
of interpretation that translates from the former to the latter.

The other main strength of functional programming is the tools it provides for
abstraction: for defining ‘embedded domain-specific languages’ in terms of exist-
ing language constructs such as algebraic datatypes, higher-order functions, and
lazy evaluation, rather than having to step outside the existing language to de-
fine a new one. In that respect, functional programming goes beyond predicative
programming and refinement calculus, which are not expressive enough to sup-
port such extension. (Of course, embedded DSLs have their limits. Sometimes a
desired feature cannot be defined conveniently, if at all, in terms of existing con-
structs; then the best approach is to relent and to define a completely different
language after all.)

9.1 Nondeterminism and probability in program calculi

Dijkstra [6] argued forcefully for the centrality of nondeterminism in program-
mapping calculi, particularly in order to support underspecification and program
development by stepwise refinement. That impetus led to the development of
the refinement calculus [37, 1] for imperative programming. However, although
functional programming is an excellent setting in which to calculate with pro-
grams, it does not support refinement well—for that, one has to make the step
from functions to relations [3]. Nevertheless, functional programming is very
convenient for manipulating collections of elements, and collection types such
as lists and sets form monads; so collection-oriented programming does fit the
functional view well [51, 4].

Lawvere [27, 11] pointed out that probability distributions form a monad
too; this has led to a slow but steady stream of functional programming ap-
proaches to probabilistic computation [20, 44, 9, 21]. Independently, Kozen [24]
presented a semantics for while-programs with probabilistic choice; but it was a
longstanding challenge to integrate this semantics with nondeterministic choice.
There was a flurry of work in the early 1990s addressing this issue within the
process algebra setting [55, 29, 39]. He et al. [12] used Jones’ probabilistic pow-
erdomain construction to provide a semantics for a guarded command language
with both probabilistic and nondeterministic choice; in fact, they defined two semantics—one like ours, in which, operationally speaking, demonic nondeterminism is resolved at run-time whenever a nondeterministic choice is executed, and another is which nondeterminism is resolved at compile-time, but which sacrifices idempotence of conditional. The first of He et al.’s semantics is the basis of the ‘demonic/probabilistic’ approach taken by Morgan [31]. Varacca [49], citing a personal correspondence with Gordon Plotkin, shows that although the composition of the powerset and probability distribution monads do not directly form a monad, this is fixed by taking the convex closure—giving rise to essentially the model we have in Section 5.2.

The combination of nondeterminism and probability and the selection of distributivity properties that we have presented here are not new; they are fairly well established in work on program calculi [12, 31, 32, 34]. Curiously, however, not all authors settle on the same distributivity properties; some [55, 39] have nondeterministic choice distributing over probabilistic, the opposite of the approach we take. Choosing this law sacrifices the intuitively reasonable arbcoin example; by the same line of reasoning as in Section 5.2, one can show under this alternative distributivity regime that, for arbitrary \( w \),

\[
\text{arbcoin \( v = (\text{arb} \triangleleft w \triangleright \text{return False}) \triangleleft w \triangleright (\text{return True} \triangleleft w \triangleright \text{arb})}
\]

That is, with probability \( w^2 + \pi^2 \) (independent of \( v! \)) the outcome is an arbitrary choice, and otherwise it is determined probabilistically. Worse, having nondeterministic choice distribute over probabilistic is inconsistent with idempotence—by similar reasoning again, using idempotence and commutativity of \( \Box \) and of \( \triangleright \), and distributivity of \( \Box \) over \( \triangleright \), one can show that

\[
\text{coin} = \text{coin} \triangleleft \frac{1}{2} \triangleright \text{arb}
\]

which seems a most unwelcome consequence: even a fair coin can be subverted. One might argue (as Mislove [35] does) that idempotence is a fundamental property of nondeterminism, and that this consequence for \( \text{coin} \) is untenable, and that therefore the alternative distributivity property should be avoided.

Similarly, Deng et al. [5] show that taking the perspective of testing equivalences on a probabilistic version of CSP—that is, attempting to distinguish two CSP processes by exhibiting different outcomes when they are run in parallel with a common test process—eliminates many of the otherwise reasonable equivalences; in particular, they give counterexamples to either distributivity property between nondeterministic (‘internal’) and probabilistic choice; however, distributivity of CSP’s ‘external’ choice over probabilistic choice does still survive.

However, we emphasize that our approach is agnostic as to the particular axiomatization. It is perfectly possible to impose distributivity of nondeterministic over probabilistic choice, obtaining the consequences for \( \text{arbcoin} \) and \( \text{coin} \) above; or to impose no distributivity law at all, in which case there are simply fewer program equivalences. The approach still works; whether it faithfully models intuition or reality is a separate question.
9.2 Beyond finite support

The approach we have presented here really only tells the story for finitely supported probability distributions. By exploiting recursive definitions, one might hope to be able to build distributions with infinite support; for example, with

\[
\text{naturals :: MonadProb } m \Rightarrow \text{Integer } \rightarrow m \text{ Integer}
\]

\[
\text{naturals } n = \text{return } n \triangleright \frac{1}{2} \triangleright \text{naturals } (n+1)
\]

one might hope that \text{naturals } 0 returns result \( i \) with probability \( \frac{1}{2^{i+1}} \), possibly returning any of the naturals. This works in a lazy language like Haskell, provided that the definition of \( \triangleright \) is non-strict in its right argument; for example, for an implementation based on possibly infinite weighted lists as in Section 8.1, it yields

\[
[(0, \frac{1}{2}), (1, \frac{1}{4}), (2, \frac{1}{8}), (3, \frac{1}{16}), (4, \frac{1}{32}), (5, \frac{1}{64}), \ldots]
\]

as expected. We are optimistic that the ‘free monad’ technique from Section 8.2 might be extended to give a more disciplined explanation of such cases. In related work [41], we have been using free monads to model a generic framework for tracing execution. In the case of nonterminating computations, one will in general get an infinite trace; to account for that case, one needs a coalgebraic rather than an algebraic reading of the tracing datatype. Perhaps this can all be conveniently captured in a ‘strong functional programming’ [48] or constructive type theory [33] setting, carefully distinguishing between terminating functions from algebraic datatypes and productive functions to coalgebraic codatatypes (in which case the constructions are technically no longer ‘free’ monads), or perhaps it really requires a shift from SET to CPO.

But none of these extensions will help when it comes to dealing with continuous probability distributions—say, a uniform choice among reals in the unit interval. Any particular real result will have probability zero; if represented using the free monad \( \text{DistT} \), all the leaves will have to be infinitely deep in the tree, and no useful computation can be done. There is no fundamental reason why one cannot deal with such distributions; after all, continuous probability distributions form a monad too [27, 11, 20]. But it requires an approach based on measure theory and Lebesgue integrals rather than point masses: distributions must be represented as mappings from measurable sets of outcomes to probabilities, rather than from individual outcomes. We leave this for future work.

For both of these reasons, possible nontermination and continuous distributions, the lightweight, embedded approach of unifying the ‘syntactic’ programming notation with the ‘semantic’ reasoning framework—which is one of the great appeals of functional programming—has its limitations. Morgan’s probabilistic guarded command language approach, with a clearer separation between the syntax and the semantics, pays the cost of two distinct domains of discourse, but does not suffer from the same limitation.
9.3 Lawvere theories

The axiomatic approach to reasoning about effectful functional programs that we have used here derives from our earlier paper [10]. It should come as no surprise that algebraic theories—consisting of a signature for the operations, together with laws that the operations are required to satisfy, but abstracting away from specific models of the theory—are convenient for equational reasoning about programs; after all, algebraic specifications have long been espoused as a useful technique for isolating the interface of a module from its possible implementations, separating the concerns of the module provider and consumer [7].

What did come as a surprise, at least to us, was that computational effects such as nondeterminism and probabilistic choice are amenable to algebraic specifications. But history [19] tells us that this is only to be expected; algebraic theories were introduced in Lawvere’s PhD thesis [26] as a category-theoretic formulation of universal algebra; Linton [28] showed the equivalence between such ‘Lawvere theories’ and monads (every Lawvere theory has a category of models that is isomorphic to the category of algebras of some monad, unique up to isomorphism), which arise as adjoint pairs of functors [8, 22]; and Moggi [36] and Wadler [54] showed that monads are precisely what is needed to encapsulate effects in pure functional languages. Indeed, this is precisely how impure effects are implemented in a pure functional programming language such as Haskell: pure evaluation is used to construct a term in the algebraic theory, which is subsequently interpreted—with possible side-effects—by the impure run-time system. In some ways, the algebraic theory approach to effects is more appealing than the monadic approach, since it places the additional operations and their properties front and centre; nevertheless, monads and Haskell’s do notation do provide a rather elegant programming notation.

On the other hand, not all the additional operations we have discussed technically fit into the algebraic theory framework. Specifically, the bind operator \( \gg \) should distribute over every such operation [42], as for example it does over probabilistic choice:

\[
\text{do} \{ x \leftarrow (p \triangleleft w \triangleright q) ; k x \} = \text{do} \{ x \leftarrow p ; k x \} \triangleleft w \triangleright \text{do} \{ x \leftarrow q ; k x \}
\]

But as we saw in Section 7, bind does not distribute over catch. Plotkin and Pretnar [43] call operations like catch ‘effect handlers’; they need a separate treatment.

9.4 Acknowledgements

We are grateful to members of the Algebra of Programming research group at Oxford, of IFIP Working Groups 2.1 and 2.8, and to the referees of this and the earlier paper [10], all of whom have made helpful suggestions. Nicolas Wu suggested the subsequences example in Section 3.1, and Ralf Hinze provide much useful advice. This work was supported by the UK EPSRC grant Reusability and Dependent Types.
This is a version of the paper *Unifying Theories of Programming with Monads* from the UTP Symposium 2012, specially adapted for Carroll Morgan’s 60th birthday. Happy birthday, Carroll!

References

27. Lawvere, F.W.: The category of probabilistic mappings (1962), preprint
29. Lowe, G.: Representing nondeterministic and probabilistic behaviour in reactive processes (1993), manuscript, Oxford University Computing Laboratory