Unifying Theories of Programming with Monads

Jeremy Gibbons

Oxford University Department of Computer Science
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
jeremy.gibbons@cs.ox.ac.uk

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.

1 Introduction

Hoare and He's *Unifying Theories of Programming* [17] presents a coherent model of a number of programming idioms—imperative, nondeterministic, concurrent, reactive, higher-order, and so on. The approach follows Hoare's own earlier "programs are predicates" [16] slogan: rather than separate domains of syntax and semantics, and a translation from one to the other, there is just one domain of discourse; programming notations like sequencing and choice are *defined* as operations on predicates like composition and disjunction, rather than being *interpreted* as such. The result is a simple and streamlined framework for reasoning about programs, without the clumsiness and noise imposed by ubiquitous semantic brackets.

Another streamlined vehicle for reasoning about programs is provided by pure functional programming. This too allows one to elide the distinction between syntax and semantics, on account of referential transparency: familiar *equational reasoning* works as well for expressions denoting programs as it does for expressions denoting numbers. Again, we do not 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 same language works for both.

Functional programming also conveniently allows one to discuss a variety of programming idioms within the same unifying framework. Moggi [36] showed how "notions of computation" such as mutable state, exceptions, nondeterminism, and probability can be elegantly encapsulated as *monads*, and safely embedded within an otherwise pure functional language. It may seem that purity rules out interesting computational effects, such as update, exception handling,

B. Wolff, M.-C. Gaudel, A. Feliachi (Eds.): UTP 2012, LNCS 7681, pp. 23–67, 2013.

and choice; after all, if *coin* denotes the computation modelling a fair coin toss—a 50–50 choice between heads and tails—then do not two occurrences of *coin* denote possibly different outcomes, thereby destroying referential transparency? The apparent problem is eliminated by distinguishing between types that represent *values*, such as 'true' or 'heads', and those that represent *computations*, such as *coin*. Two occurrences of *coin* denote the same computation, and it is the executions 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; *coin* denotes not a single outcome, but a distribution of outcomes. The operations and axioms of a notion of computation can be precisely and elegantly abstracted via the categorical notion of a monad. Equivalently, the operations and axioms can be captured as an *algebraic theory*, and equational reasoning can be safely conducted within such a theory.

One advantage that functional programming offers over the "programs are predicates" approach 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 feature. Our chosen language Haskell does not itself provide constructs for specific notions of computation such as probabilistic choice, 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.

We believe that the UTP and FP communities have much in common, and perhaps much to learn from each other. In this paper, we make a step towards bringing the two communities closer together, by way of unifying theories of nondeterminism and probability expressed in a functional style. 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 programming and to the use of monads to capture computational effects (Section 2)—readers familiar with functional programming in general, and Haskell in particular, may wish to skip this section. We then introduce theories of nondeterministic choice (Section 3) and probabilistic choice (Section 4) separately, and in combination (Section 5). Section 6 presents an extended example based on the infamous Monty Hall problem. In Section 7 we consider the possibility of failure and the effect of exceptions, which gives rise to conditionally probabilistic computations; and in Section 8 we look at recursive definitions. Section 9 concludes with a discussion of related work and some thoughts about future developments.

2 Effectful Functional Programming

Pure functional programming languages constitute a very appealing model of computation: *simple*, due to abstraction from the details of computer architecture, 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:

$$\begin{aligned} &\mathit{square} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{square}\ x = x \times x \end{aligned}$$

or equivalently

$$\begin{aligned} &\mathit{square} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{square} = \lambda x \to x \times x \end{aligned}$$

As well as specifying an action, namely how to compute squares, this program also serves as an equation: for any x, the expression $\mathit{square}\ x$ 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 *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:

$$\begin{aligned} &(\circ) :: (b \to c) \to (a \to b) \to a \to c \\ &(f \circ g)\ x = f\ (g\ x) \end{aligned}$$

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:

$$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

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 acting on a mutable heap.

Functions over such data structures can conveniently be defined by pattern matching. For example, here is the standard function *foldr* to fold a list to a value:

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

This is another higher-order function, since it takes a function as its first argument. One instance of *foldr* is the function *sum* that sums a list of integers:

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

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

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

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

$$\mathit{reverse} \circ \mathit{map}\ f = \mathit{map}\ f \circ \mathit{reverse}$$

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 *foldr* function is the *fusion law*:

$$h \circ \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e' \iff h\ (f\ x\ y) = f'\ x\ (h\ y) \wedge h\ e = e'$$

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

$$\begin{aligned} &\quad h\ (\mathit{foldr}\ f\ e\ [\,]) \\ &= \quad [\![\text{ definition of } \mathit{foldr} \ ]\!] \\ &\quad h\ e \\ &= \quad [\![\text{ assumption }]\!] \\ &\quad e' \\ &= \quad [\![\text{ definition of } \mathit{foldr} \ ]\!] \\ &\quad \mathit{foldr}\ f'\ e'\ [\,] \end{aligned}$$

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

$$
\begin{array}{ll}
 & h\ (\mathit{foldr}\ f\ e\ (x : xs)) \\
= & \quad [\![\text{ definition of } \mathit{foldr} \quad]\!] \\
 & h\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\
= & \quad [\![\text{ assumption }]\!] \\
 & f'\ x\ (h\ (\mathit{foldr}\ f\ e\ xs)) \\
= & \quad [\![\text{ inductive hypothesis }]\!] \\
 & f'\ x\ (\mathit{foldr}\ f'\ e'\ xs) \\
= & \quad [\![\text{ definition of } \mathit{foldr} \quad]\!] \\
 & \mathit{foldr}\ f'\ e'\ (x : xs)
\end{array}
$$

A simple consequence of the fusion law is the *fold–map fusion law*, when h is itself an instance of *foldr*, and follows a *map* over lists, which is another instance of *foldr*. In this case, the fusion result

$$\mathit{foldr}\ f\ e \circ \mathit{map}\ g = \mathit{foldr}\ f'\ e'$$

follows from the fusion conditions

$$\mathit{foldr}\ f\ e\ (g\ x : ys) = f'\ x\ (\mathit{foldr}\ f\ e\ ys) \wedge \mathit{foldr}\ f\ e\ [\,] = e'$$

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

$$\mathit{foldr}\ f\ e \circ \mathit{map}\ g = \mathit{foldr}\ (f \circ g)\ e$$

For most of the paper we will work within SET—that is, with 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 *foldr*, and correspondingly to finite data structures. We only have to relax this restriction in Section 8, moving to CPO—continuous functions between complete partial orders.

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 *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 *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 classifed 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 *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:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{return} & :: a \to m\ a \\
(\gg\!=) & :: m\ a \to (a \to m\ b) \to m\ b \\
\mathit{fmap} & :: (a \to b) \to (m\ a \to m\ b) \\
\mathit{join} & :: m\ (m\ a) \to m\ a
\end{array} \\
\quad \begin{array}{ll}
p \gg\!= k & = \mathit{join}\ (\mathit{fmap}\ k\ p) \\
\mathit{join}\ pp & = pp \gg\!= \mathit{id} \\
\mathit{fmap}\ f\ p & = p \gg\!= (\mathit{return} \circ f)
\end{array}
\end{array}
$$

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*, $(\gg\!=)$, *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 $(\gg\!=)$ 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:

$$
\begin{array}{lll}
\mathit{return}\ x \gg\!= k & = k\ x & \text{-- left unit} \\
p \gg\!= \mathit{return} & = p & \text{-- right unit}
\end{array}
$$

$$
\begin{array}{lll}
(p \gg\!= h) \gg\!= k & = p \gg\!= (\lambda x \to h\ x \gg\!= k) & \text{-- associativity} \\
\mathit{fmap}\ \mathit{id} & = \mathit{id} & \text{-- map–identity} \\
\mathit{fmap}\ (f \circ g) & = \mathit{fmap}\ f \circ \mathit{fmap}\ g & \text{-- map–composition} \\
\mathit{join} \circ \mathit{return} & = \mathit{id} & \text{-- left unit} \\
\mathit{join} \circ \mathit{fmap}\ \mathit{return} & = \mathit{id} & \text{-- right unit} \\
\mathit{join} \circ \mathit{fmap}\ \mathit{join} & = \mathit{join} \circ \mathit{join} & \text{-- associativity}
\end{array}
$$

(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 *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 *join* flattens a computation that yields computations that yield results into a computation that yields results directly, and the function *fmap* modifies each of the results of a computation; together with *return*, these two give an alternative (equivalent) perspective on sequential composition.

Two shorthands turn out to be quite convenient. We write *skip* for the pure computation that returns the sole element () of the unit type, also written ():

$$
\begin{array}{l}
\mathit{skip} :: \mathit{Monad}\ m \Rightarrow m\ () \\
\mathit{skip} = \mathit{return}\ ()
\end{array}
$$

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

$$
\begin{array}{l}
(\gg) \quad :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b \\
p \gg q = p \gg\!= (\lambda\_ \to q)
\end{array}
$$

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.)

$$
\begin{array}{l}
\mathit{sequence\_} :: \mathit{Monad}\ m \Rightarrow [m\ ()] \to m\ () \\
\mathit{sequence\_} = \mathit{foldr}\ (\gg)\ \mathit{skip}
\end{array}
$$

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:

$$f\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ f\ y$$
$$(\textbf{if}\ b\ \textbf{then}\ f\ \textbf{else}\ g)\ x = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x$$

These laws can easily be verified by considering the two cases $b = \mathit{True}$ and $b = \mathit{False}$. (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 *seq* operator, which distinguishes between $\bot$ and $\lambda x \to \bot$.) In particular, letting f be $(\gg\!= k)$ and $(p \gg\!=)$ in turn, we deduce from the first law that composition distributes respectively leftwards and rightwards over conditional:

$$(\textbf{if}\ b\ \textbf{then}\ p\ \textbf{else}\ q) \gg\!= k\ = \textbf{if}\ b\ \textbf{then}\ p \gg\!= k\ \textbf{else}\ q \gg\!= k$$
$$p \gg\!= (\textbf{if}\ b\ \textbf{then}\ k\ \textbf{else}\ k') = \textbf{if}\ b\ \textbf{then}\ p \gg\!= k\ \textbf{else}\ p \gg\!= k'$$

(Again, these laws hold unconditionally in SET; in CPO, they require $\gg\!=$ 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—for simplicity, a single mutable value—as a class of effects. Just two additional operations are required: *get*, to read the state, and *put*, to update it. We declare a subclass *MonadState* of *Monad*; type constructor m is a member of the class *MonadState* if it is a member of *Monad* and it supports the two additional methods *get* and *put*. (To be precise, the subclass is *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.)

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \textbf{where}$$
$$\quad \mathit{get} :: m\ s$$
$$\quad \mathit{put} :: s \to m\ ()$$

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

$$\mathit{get} \gg\!= \lambda s \to \mathit{get} \gg\!= \lambda s' \to k\ s\ s' = \mathit{get} \gg\!= \lambda s \to k\ s\ s \quad \text{-- get–get}$$
$$\mathit{get} \gg\!= \mathit{put} = \mathit{skip} \quad \text{-- get–put}$$

$$
\begin{array}{lll}
put\ s \gg put\ s' & = put\ s' & \text{-- put–put} \\
put\ s \gg get \gg\!= \lambda s' \to k\ s' & = put\ s \gg k\ s & \text{-- put–get}
\end{array}
$$

Informally: two consecutive *get*s will read the same value twice; *get*ting a value then *put*ting it back is has no effect; two consecutive *put*s are equivalent to just the second one; and a *get* immediately after *put*ting 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.

$$
\begin{array}{l}
incrodd :: MonadState\ Int\ m \Rightarrow m\ Bool \\
incrodd = get \gg\!= (\lambda n \to put\ (n+1) \gg\!= (\lambda() \to return\ (odd\ n)))
\end{array}
$$

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 $s \to (a, s)$:

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

Now, *State s* forms a type of computations, and so we should be able to make it an instance of the type class *Monad*. To do so, for *return a* we use the state-transforming function that yields x and leaves the state unchanged; *fmap f* applies f to the output value without touching the output state; and *join* collapses a state-transformer that yields a state-transformer by applying the output state-transformer to the output state:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ (State\ s)\ \textbf{where} \\
\quad return\ x\ = \lambda s \to (x, s) \\
\quad fmap\ f\ p = \lambda s \to \textbf{let}\ (x, s') = p\ s\ \textbf{in}\ (f\ x, s') \\
\quad join\ p\ \ \ = \lambda s \to \textbf{let}\ (p', s') = p\ s\ \textbf{in}\ p'\ s'
\end{array}
$$

The reader may enjoy deriving from this the corresponding definition

$$p \gg\!= k = \lambda s \to \textbf{let}\ (x, s') = p\ s\ \textbf{in}\ k\ x\ s'$$

of bind, which chains state transformations together. (For technical reasons, this instance declaration is not quite in Haskell syntax: rather than a type synonym, *State* ought to be a **newtype** or **data**type, with a constructor and deconstructor. But what is shown is morally correct.)

Of course, by design, *State s* supports the features of mutable state—*get* yields a copy of the state, and *put* overwrites it:

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

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

2.5 Imperative Functional Programming

Wadler also observed [53] that the methods of the *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 '**do** notation' [25], which is defined by translation into *Monad* methods as follows:

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

In particular, instead of having to write functions (typically lambda expressions) as the second argument of $\gg\!=$, 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 more elegantly as follows:

$$\begin{aligned}
&incrodd :: MonadState\ Int\ m \Rightarrow m\ Bool\\
&incrodd = \mathbf{do}\,\{\,n \leftarrow get\,;\,put\ (n+1)\,;\,return\ (odd\ n)\,\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{do}\,\{\,x \leftarrow return\ e\,;\,k\ x\,\} &&= \mathbf{do}\,\{\,k\ e\,\}\\
&\mathbf{do}\,\{\,x \leftarrow p\,;\,return\ x\,\} &&= \mathbf{do}\,\{\,p\,\}\\
&\mathbf{do}\,\{\,y \leftarrow \mathbf{do}\,\{\,x \leftarrow p\,;\,h\ x\,\}\,;\,k\ y\,\} &&= \mathbf{do}\,\{\,x \leftarrow p\,;\,y \leftarrow h\ x\,;\,k\ y\,\}
\end{aligned}$$

(where, implicitly in the third law, x is not free in k). The operators *fmap* and *join* can be expressed in **do** notation like this:

$$\begin{aligned}
&fmap\ f\ p &&= \mathbf{do}\,\{\,x \leftarrow p\,;\,return\ (f\ x)\,\}\\
&join\ pp &&= \mathbf{do}\,\{\,p \leftarrow pp\,;\,x \leftarrow p\,;\,return\ x\,\}
\end{aligned}$$

Distribution of composition leftwards and rightwards over conditional looks like this:

$$\begin{aligned}
&\mathbf{do}\,\{\,x \leftarrow \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ q\,;\,k\ x\,\} &&= \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\,\{\,x \leftarrow p\,;\,k\ x\,\}\\
&&&\qquad\mathbf{else}\ \ \mathbf{do}\,\{\,x \leftarrow q\,;\,k\ x\,\}\\
&\mathbf{do}\,\{\,x \leftarrow p\,;\mathbf{if}\ b\ \mathbf{then}\ h\ x\ \mathbf{else}\ k\ x\,\} &&= \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}\,\{\,x \leftarrow p\,;\,h\ x\,\}\\
&&&\qquad\mathbf{else}\ \ \mathbf{do}\,\{\,x \leftarrow p\,;\,k\ x\,\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{do}\,\{\,s \leftarrow get\,;\,s' \leftarrow get\,;\,k\ s\ s'\,\} &&= \mathbf{do}\,\{\,s \leftarrow get\,;\,k\ s\ s\,\} && \text{-- get–get}\\
&\mathbf{do}\,\{\,s \leftarrow get\,;\,put\ s\,\} &&= \mathbf{do}\,\{\,skip\,\} && \text{-- get–put}\\
&\mathbf{do}\,\{\,put\ s\,;\,put\ s'\,\} &&= \mathbf{do}\,\{\,put\ s'\,\} && \text{-- put–put}\\
&\mathbf{do}\,\{\,put\ s\,;\,s' \leftarrow get\,;\,k\ s'\,\} &&= \mathbf{do}\,\{\,put\ s\,;\,k\ s\,\} && \text{-- put–get}
\end{aligned}$$

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:

$$\begin{aligned} &\mathit{add} :: \mathit{MonadState}\ \mathit{Int}\ m \Rightarrow \mathit{Int} \to m\ () \\ &\mathit{add}\ n = \mathbf{do}\ \{\, m \leftarrow \mathit{get}\ ;\ \mathit{put}\ (m + n)\,\} \end{aligned}$$

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

$$\mathit{addAll} = \mathit{add} \circ \mathit{sum}$$

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

$$\begin{aligned} &\mathit{addAll} :: \mathit{MonadState}\ \mathit{Int}\ m \Rightarrow [\mathit{Int}] \to m\ () \\ &\mathit{addAll} = \mathit{sequence}\_ \circ \mathit{map}\ \mathit{add} \end{aligned}$$

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

$$\mathit{addAll} = \mathit{foldr}\ (\lambda n\ p \to \mathbf{do}\ \{\, \mathit{add}\ n\ ;\ p\,\})\ \mathit{skip}$$

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

$$\begin{aligned} \mathit{add}\ 0 \qquad &= \mathit{skip} \\ \mathit{add}\ (n + n') &= \mathbf{do}\ \{\, \mathit{add}\ n\ ;\ \mathit{add}\ n'\,\} \end{aligned}$$

For the first of these, we have:

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

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

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

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*.

$$\begin{array}{l} \mathbf{class}\ Monad\ m \Rightarrow MonadAlt\ m\ \mathbf{where} \\ \quad (\Box) :: m\ a \to m\ a \to m\ a \end{array}$$

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

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

and that composition distributes leftwards over it:

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

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

$$\mathbf{do}\,\{\,x \leftarrow p\,;\,(h\ x \Box k\ x)\,\} \neq \mathbf{do}\,\{\,x \leftarrow p\,;\,h\ x\,\} \Box \mathbf{do}\,\{\,x \leftarrow p\,;\,k\ x\,\}$$

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 $\Box$ is angelic, then these effects happen once on the left-hand side of the equation, and twice on the right; whereas if $\Box$ 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 finite nonempty set of results. A pure computation amounts to returning a singleton set, *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 set-valued function.)

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.

```
instance Monad [] where
  return a  = [a]
  fmap f p  = [f x | x ← p]
  join      = concat
```

Naturally, we implement the nondeterministic choice as concatenation:

```
instance MonadAlt [] where
  (□) = (++)
```

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], we might want to treat as significant the order in which results are found, and so we might want to drop the commutativity axiom; and to keep track of nesting depth in search trees [47], we might want to drop associativity.

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 *subs* that nondeterministically chooses a subsequence of a list. Of course, interpreted in the nonempty-list implementation of

nondeterminism, *subs* returns the usual nonempty list of lists; but this definition supports other implementations of nondeterminism too, such as bags and sets.

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

Informally, the empty list has a unique subsequence, the empty list itself; and a subsequence of a non-empty list $x : xs$ can be obtained by either prefixing x to or excluding it from a subsequence xss of xs.

Here is a simple property that we might wish to prove—that *subs* distributes over list concatenation:

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

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:

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

For the inductive step, we assume the result for xs, and calculate for $x : xs$ as follows:

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

$$
\begin{array}{ll}
& \mathit{fmap}\ (x{:})\ (\mathit{subs}\ (xs +\!\!+ ys)) \Box\ \mathit{subs}\ (xs +\!\!+ ys) \\
= & \quad [\![\text{ definition of } \mathit{subs} \ ]\!] \\
& \mathit{subs}\ (x : (xs +\!\!+ ys)) \\
= & \quad [\![\text{ definition of } +\!\!+ \ ]\!] \\
& \mathit{subs}\ ((x : xs) +\!\!+ ys)
\end{array}
$$

Again, plain ordinary equational reasoning suffices, using programs as equations together with 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 *Prob* of probabilities (say, the rationals in the closed unit interval), and define a *Monad* subclass for computations drawing from finitely supported probability distributions, that is, distributions in which only a finite number of elements have positive probabilities:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadProb}\ m\ \textbf{where} \\
\quad \mathit{choice} :: \mathit{Prob} \to m\ a \to m\ a \to m\ a
\end{array}
$$

The idea is that $\mathit{choice}\ w\ p\ q$ behaves as p with probability w and as q with probability $1-w$. From now on, we will write '$\overline{w}$' for $1-w$, and following Hoare's convention [15], write choice in infix notation, '$p \lhd w \rhd q$', because this makes the laws more legible. We have two identity laws:

$$
\begin{array}{l}
p \lhd 0 \rhd q = q \\
p \lhd 1 \rhd q = p
\end{array}
$$

a quasi-commutativity law:

$$p \lhd w \rhd q = q \lhd \overline{w} \rhd p$$

idempotence:

$$p \lhd w \rhd p = p$$

and quasi-associativity:

$$p \lhd u \rhd (q \lhd v \rhd r) = (p \lhd w \rhd q) \lhd x \rhd r \iff u = w \times x \ \wedge\ \overline{x} = \overline{u} \times \overline{v}$$

As informal justification for quasi-associativity, observe that the likelihoods of p, q, r on the left are $u, \overline{u} \times v, \overline{u} \times \overline{v}$, and on the right are $w \times x, \overline{w} \times x, \overline{x}$, 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:

$$
\begin{array}{l}
\textbf{do}\ \{x \leftarrow (p \lhd w \rhd q)\ ;\ k\ x\} = \textbf{do}\ \{x \leftarrow p\ ;\ k\ x\} \lhd w \rhd\ \textbf{do}\ \{x \leftarrow q\ ;\ k\ x\} \\
\textbf{do}\ \{x \leftarrow p\ ;\ (h\ x \lhd w \rhd k\ x) = \textbf{do}\ \{x \leftarrow p\ ;\ h\ x\} \lhd w \rhd\ \textbf{do}\ \{x \leftarrow p\ ;\ k\ x\}
\end{array}
$$

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):

$$\begin{aligned}&\mathit{coin} :: \mathit{MonadProb}\ m \Rightarrow m\ \mathit{Bool}\\&\mathit{coin} = \mathit{return}\ \mathit{True} \lhd 1/2 \rhd \mathit{return}\ \mathit{False}\end{aligned}$$

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

$$\textbf{type}\ \mathit{Dist}\ a = [(a, \mathit{Prob})] \quad \text{-- weights sum to 1}$$

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:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{Dist}\ \textbf{where}\\&\quad \mathit{return}\ x\ = [(x, 1)]\\&\quad \mathit{fmap}\ f\ p = [(f\ x, w) \mid (x, w) \leftarrow p]\\&\quad \mathit{join}\ p\ \ \ = \mathit{concat}\ [\mathit{scale}\ w\ x \mid (x, w) \leftarrow p]\end{aligned}$$

where

$$\begin{aligned}&\mathit{scale} :: \mathit{Prob} \to [(a, \mathit{Prob})] \to [(a, \mathit{Prob})]\\&\mathit{scale}\ v\ p = [(x, v \times w) \mid (x, w) \leftarrow p]\end{aligned}$$

On the other hand, $\lhd \rhd$ is a kind of weighted sum:

$$\begin{aligned}&\textbf{instance}\ \mathit{MonadProb}\ \mathit{Dist}\ \textbf{where}\\&\quad p \lhd w \rhd q = \mathit{scale}\ w\ p +\!\!+ \mathit{scale}\ \overline{w}\ q\end{aligned}$$

Probability-weighted lists are not quite the initial model, because the identity, idempotence, quasi-commutativity, and quasi-associativity laws of $\lhd \rhd$ 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 *Eq*, supporting the equality operation $==$ needed by finite maps, we can define:

$$\begin{aligned}&\mathit{collect} :: \mathit{Eq}\ a \Rightarrow \mathit{Dist}\ a \to (a \to \mathit{Prob})\\&\mathit{collect}\ p\ y = \mathit{sum}\ [w \mid (x, w) \leftarrow p, x == y]\end{aligned}$$

That is, equivalences on *Dist* ought to be taken modulo permutations, zero-weighted elements, and repeated elements (whose weights should be added). Nevertheless, the datatype *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 $(p \lhd {}^1\!/_2 \rhd q) \lhd {}^1\!/_3 \rhd (r \lhd {}^1\!/_4 \rhd s)$ we allow ourselves to write $\langle p@{}^1\!/_6, q@{}^1\!/_6, r@{}^1\!/_6, s@{}^1\!/_2\rangle$, multiplying out all the probabilities.

4.1 Example: Uniform Distributions

Extending the fair coin example, we might define uniform distributions

$$\begin{aligned}
&\mathit{uniform} :: \mathit{MonadProb}\ m \Rightarrow [a] \to m\ a \quad \text{-- nonempty list}\\
&\mathit{uniform}\ [x] \qquad = \mathit{return}\ x\\
&\mathit{uniform}\ (x : xs) = \mathit{return}\ x \lhd {}^1\!/_{\mathit{length}\ (x:xs)} \rhd \mathit{uniform}\ xs
\end{aligned}$$

so that $\mathit{coin} = \mathit{uniform}\ [\mathit{True}, \mathit{False}]$, and $\mathit{uniform}\ [1,2,3] = \mathit{return}\ 1 \lhd {}^1\!/_3 \rhd (\mathit{return}\ 2 \lhd {}^1\!/_2 \rhd \mathit{return}\ 3)$.

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 xs that

$$\mathbf{do}\ \{x \leftarrow \mathit{uniform}\ xs\ ;\ p\} = p$$

when p does not depend on x. Similarly, *uniform* distributes over concatenation:

$$\mathit{uniform}\ (xs +\!\!+\ ys) = \mathit{uniform}\ xs \lhd {}^m\!/_{m+n} \rhd \mathit{uniform}\ ys$$

where $m = \mathit{length}\ xs$ and $n = \mathit{length}\ ys$. As a consequence of these properties of *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:

$$\mathbf{do}\ \{x \leftarrow \mathit{uniform}\ xs\ ;\ y \leftarrow \mathit{uniform}\ ys\ ;\ \mathit{return}\ (x,y)\} = \mathit{uniform}\ (\mathit{cp}\ xs\ ys)$$

where

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

We can prove this property by induction over xs, using equational reasoning with the laws of *MonadProb*. For the base case of singleton lists, we have:

$$\begin{aligned}
&\quad \mathit{uniform}\ (\mathit{cp}\ [x]\ ys)\\
&= \quad [\![\ \text{definition of } \mathit{cp}\ ]\!]\\
&\quad \mathit{uniform}\ [(z,y) \mid z \leftarrow [x], y \leftarrow ys]\\
&= \quad [\![\ \text{comprehensions: } [f\ z \mid z \leftarrow [x], p] = [f\ x \mid p]\ ]\!]\\
&\quad \mathit{uniform}\ [(x,y) \mid y \leftarrow ys]\\
&= \quad [\![\ \text{comprehensions: } [f\ x \mid x \leftarrow xs] = \mathit{map}\ f\ xs\ ]\!]\\
&\quad \mathit{uniform}\ (\mathit{map}\ (\lambda y \to (x,y))\ ys)
\end{aligned}$$

$$
\begin{array}{ll}
= & [\![\text{ naturality: } \mathit{uniform} \circ \mathit{map}\ f = \mathit{fmap}\ f \circ \mathit{uniform} \ ]\!] \\
& \mathbf{do}\ \{ y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (x, y) \} \\
= & [\![\text{ left unit }]\!] \\
& \mathbf{do}\ \{ z \leftarrow \mathit{return}\ x \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \} \\
= & [\![\text{ definition of } \mathit{uniform} \ ]\!] \\
& \mathbf{do}\ \{ z \leftarrow \mathit{uniform}\ [x] \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \}
\end{array}
$$

and for the inductive step, assuming the result for xs, we have:

$$
\begin{array}{ll}
& \mathit{uniform}\ (\mathit{cp}\ (x : \mathit{xs})\ \mathit{ys}) \\
= & [\![\text{ definition of } \mathit{cp} \ ]\!] \\
& \mathit{uniform}\ [(z, y) \mid z \leftarrow x : \mathit{xs}, y \leftarrow \mathit{ys}] \\
= & [\![\text{ comprehensions distribute over } +\!\!+ \ ]\!] \\
& \mathit{uniform}\ ([(z, y) \mid z \leftarrow [x], y \leftarrow \mathit{ys}] +\!\!+ [(z, y) \mid z \leftarrow \mathit{xs}, y \leftarrow \mathit{ys}]) \\
= & [\![\text{ as above; definition of } \mathit{cp} \ ]\!] \\
& \mathit{uniform}\ (\mathit{map}\ (\lambda y \rightarrow (x, y))\ \mathit{ys} +\!\!+ \mathit{cp}\ \mathit{xs}\ \mathit{ys}) \\
= & [\![\ \mathit{uniform} \text{ distributes over } +\!\!+; \text{ let } n = \mathit{length}\ \mathit{ys},\ l = \mathit{length}\ (\mathit{cp}\ \mathit{xs}\ \mathit{ys}) \ ]\!] \\
& \mathit{uniform}\ (\mathit{map}\ (\lambda y \rightarrow (x, y))\ \mathit{ys}) \triangleleft {}^{n}\!/_{n+l} \triangleright \mathit{uniform}\ (\mathit{cp}\ \mathit{xs}\ \mathit{ys}) \\
= & [\![\text{ let } m = \mathit{length}\ \mathit{xs}, \text{ so } l = m \times n \ ]\!] \\
& \mathit{uniform}\ (\mathit{map}\ (\lambda y \rightarrow (x, y))\ \mathit{ys}) \triangleleft {}^{1}\!/_{1+m} \triangleright \mathit{uniform}\ (\mathit{cp}\ \mathit{xs}\ \mathit{ys}) \\
= & [\![\text{ base case, inductive hypothesis }]\!] \\
& \mathbf{do}\ \{ z \leftarrow \mathit{uniform}\ [x] \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \} \triangleleft {}^{1}\!/_{1+m} \triangleright \\
& \mathbf{do}\ \{ z \leftarrow \mathit{uniform}\ \mathit{xs} \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \} \\
= & [\![\text{ composition distributes leftwards over } \triangleleft\triangleright \ ]\!] \\
& \mathbf{do}\ \{ z \leftarrow \mathit{uniform}\ [x] \triangleleft {}^{1}\!/_{1+m} \triangleright \mathit{uniform}\ \mathit{xs} \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \} \\
= & [\![\text{ definition of } \mathit{uniform} \ ]\!] \\
& \mathbf{do}\ \{ z \leftarrow \mathit{uniform}\ (x : \mathit{xs}) \,;\, y \leftarrow \mathit{uniform}\ \mathit{ys} \,;\, \mathit{return}\ (z, y) \}
\end{array}
$$

The second step uses the property

$$[f\ z \mid z \leftarrow \mathit{zs} +\!\!+ \mathit{zs}', p] = [f\ z \mid z \leftarrow \mathit{zs}, p] +\!\!+ [f\ z \mid z \leftarrow \mathit{zs}', p]$$

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 *MonadAlt* and *MonadProb* together suffice:

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

This Haskell type class declaration is complete; it has an empty collection of additional methods, beyond those inherited from the superclasses *MonadAlt* and

MonadProb. Implicitly, the laws of *MonadAlt* and *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:

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

(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:

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

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

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

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

satisfying the following four coherence conditions:

$$\begin{aligned}
&\mathit{swap} \circ \mathit{fmap}_D\ \mathit{return}_L &&= \mathit{return}_L \\
&\mathit{swap} \circ \mathit{return}_D &&= \mathit{fmap}_L\ \mathit{return}_D \\
&\mathit{swap} \circ \mathit{fmap}_D\ \mathit{join}_L &&= \mathit{join}_L \circ \mathit{fmap}_L\ \mathit{swap} \circ \mathit{swap} \\
&\mathit{swap} \circ \mathit{join}_D &&= \mathit{fmap}_L\ \mathit{join}_D \circ \mathit{swap} \circ \mathit{fmap}_D\ \mathit{swap}
\end{aligned}$$

(where, to be explicit about typing, we have subscripted each use of *return*, *fmap*, and *join* with L or D to indicate the list and distribution instances, respectively). Then we can declare that the composite datatype *Dists* forms a monad, following the standard construction [2]:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{Dists}\ \textbf{where} \\
&\quad \mathit{return}\ x\ = \mathit{return}\ (\mathit{return}\ x) \\
&\quad \mathit{fmap}\ f\ p = \mathit{fmap}\ (\mathit{fmap}\ f)\ p \\
&\quad \mathit{join}\ pp\ \ = \mathit{fmap}\ \mathit{join}\ (\mathit{join}\ (\mathit{map}\ \mathit{swap}\ pp))
\end{aligned}$$

A suitable definition of *swap* is as follows:

$$
\begin{aligned}
&\mathit{swap} = \mathit{foldr}_1\ \mathit{pick} \circ \mathit{map}\ \mathit{split}\ \textbf{where}\\
&\quad \mathit{split}\ (xs, w)\ = [[(x, w)] \mid x \leftarrow xs]\\
&\quad \mathit{pick}\ \mathit{xds}\ \mathit{yds} = [xd +\!\!+ yd \mid xd \leftarrow \mathit{xds}, yd \leftarrow \mathit{yds}]
\end{aligned}
$$

(Here, foldr_1 is a variant of *foldr* for non-empty lists, taking only a binary operator and no starting value.) Informally, *swap* takes a distribution of nondeterministic choices to a nondeterministic choice of distributions, multiplying out all the possibilities; for example,

$$
\begin{aligned}
\mathit{swap}\ ([([1,2], 1/3), ([3,4], 2/3)]) = [[(1, 1/3), (3, 2/3)], [(1, 1/3), (4, 2/3)],\\
[(2, 1/3), (3, 2/3)], [(2, 1/3), (4, 2/3)]]
\end{aligned}
$$

The composite monad *Dists* inherits *MonadAlt* and *MonadProb* functionality straightforwardly from its two component parts:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonadAlt}\ \mathit{Dists}\ \textbf{where}\\
&\quad p \,\Box\, q = p +\!\!+ q\\
&\textbf{instance}\ \mathit{MonadProb}\ \mathit{Dists}\ \textbf{where}\\
&\quad p \triangleleft w \triangleright q = [xd \triangleleft w \triangleright yd \mid xd \leftarrow p, yd \leftarrow q]
\end{aligned}
$$

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

$$
\textbf{instance}\ \mathit{MonadAltProb}\ \mathit{Dists}
$$

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:

$$
\begin{aligned}
&\mathit{bcoin} :: \mathit{MonadProb}\ m \Rightarrow \mathit{Prob} \to m\ \mathit{Bool}\\
&\mathit{bcoin}_w = \mathit{return}\ \mathit{True} \triangleleft w \triangleright \mathit{return}\ \mathit{False}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{arb} :: \mathit{MonadAlt}\ m \Rightarrow m\ \mathit{Bool}\\
&\mathit{arb} = \mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False}
\end{aligned}
$$

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]:

$$
\begin{aligned}
&\mathit{arbcoin}, \mathit{coinarb} :: \mathit{MonadAltProb}\ m \Rightarrow \mathit{Prob} \to m\ \mathit{Bool}\\
&\mathit{arbcoin}\ w = \textbf{do}\ \{\, a \leftarrow \mathit{arb}\ ;\ c \leftarrow \mathit{bcoin}_w\ ;\ \mathit{return}\ (a == c)\,\}\\
&\mathit{coinarb}\ w = \textbf{do}\ \{\, c \leftarrow \mathit{bcoin}_w\ ;\ a \leftarrow \mathit{arb}\ ;\ \mathit{return}\ (a == c)\,\}
\end{aligned}
$$

Intuitively, because the probabilistic choice happens 'first' in *coinarb*, the nondeterministic choice can depend on it; whereas in *arbcoin*, the probabilistic choice happens 'last', so the nondeterministic choice cannot depend on it—and moreover, the probabilistic choice cannot be affected by the nondeterministic either, because it would not follow the distribution if it did so. We can justify this intuition calculationally, using the equational theory of the two kinds of choice. On the one hand, we have:

$$
\begin{array}{ll}
 & \mathit{arbcoin}\ w \\
= & \quad [\![\text{ definition of } \mathit{arbcoin} \ ]\!] \\
 & \mathbf{do}\,\{\, a \leftarrow \mathit{arb}\,;\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ definition of } \mathit{arb} \ ]\!] \\
 & \mathbf{do}\,\{\, a \leftarrow (\mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False})\,;\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ composition distributes leftwards over } \Box \ ]\!] \\
 & \mathbf{do}\,\{\, a \leftarrow \mathit{return}\ \mathit{True}\,;\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ (a == c)\,\} \,\Box \\
 & \mathbf{do}\,\{\, a \leftarrow \mathit{return}\ \mathit{False}\,;\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ left unit, booleans }]\!] \\
 & \mathbf{do}\,\{\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ c\,\} \,\Box\, \mathbf{do}\,\{\, c \leftarrow \mathit{bcoin}_w\,;\, \mathit{return}\ (\neg\ c)\,\} \\
= & \quad [\![\text{ right unit; definition of } \mathit{bcoin}_w \ ]\!] \\
 & \mathit{bcoin}_w \,\Box\, \mathit{bcoin}_{\overline{w}}
\end{array}
$$

On the other hand,

$$
\begin{array}{ll}
 & \mathit{coinarb}\ w \\
= & \quad [\![\text{ definition of } \mathit{coinarb} \ ]\!] \\
 & \mathbf{do}\,\{\, c \leftarrow \mathit{bcoin}_w\,;\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ definition of } \mathit{bcoin}_w \ ]\!] \\
 & \mathbf{do}\,\{\, c \leftarrow (\mathit{return}\ \mathit{True} \triangleleft w \triangleright \mathit{return}\ \mathit{False})\,;\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ composition distributes leftwards over } \triangleleft\triangleright \ ]\!] \\
 & \mathbf{do}\,\{\, c \leftarrow \mathit{return}\ \mathit{True}\,;\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ (a == c)\,\} \triangleleft w \triangleright \\
 & \mathbf{do}\,\{\, c \leftarrow \mathit{return}\ \mathit{False}\,;\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ (a == c)\,\} \\
= & \quad [\![\text{ left unit, booleans }]\!] \\
 & \mathbf{do}\,\{\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ a\,\} \triangleleft w \triangleright\ \mathbf{do}\,\{\, a \leftarrow \mathit{arb}\,;\, \mathit{return}\ (\neg\ a)\,\} \\
= & \quad [\![\text{ right unit; definition of } \mathit{arb} \ ]\!] \\
 & (\mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False}) \triangleleft w \triangleright (\mathit{return}\ \mathit{False} \,\Box\, \mathit{return}\ \mathit{True}) \\
= & \quad [\![\text{ commutativity of } \Box \ ]\!] \\
 & (\mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False}) \triangleleft w \triangleright (\mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False}) \\
= & \quad [\![\text{ idempotence of } \triangleleft\triangleright \ ]\!] \\
 & \mathit{return}\ \mathit{True} \,\Box\, \mathit{return}\ \mathit{False} \\
= & \quad [\![\text{ definition of } \mathit{arb} \ ]\!] \\
 & \mathit{arb}
\end{array}
$$

That is, the nondeterminism in *arbcoin* can be resolved only by choosing the distribution provided by bcoin_w itself, or its opposite—the nondeterministic choice happens first, and depending on whether *True* or *False* is chosen, the probabilistic choice has chance either w or $\overline{w}$ of matching it. In particular, if $w = 1/2$, then the nondeterministic choice cannot influence the final outcome.

But in *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 *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 $\lhd\rhd$ over $\Box$ and idempotence of $\lhd\rhd$ together imply a convexity property:

$$\begin{array}{ll} & p \mathbin{\Box} q \\ = & \quad [\![\text{ idempotence of } \lhd\rhd \text{; arbitrary } w \ ]\!] \\ & (p \mathbin{\Box} q) \lhd w \rhd (p \mathbin{\Box} q) \\ = & \quad [\![\text{ distributing } \lhd\rhd \text{ over } \Box \ ]\!] \\ & (p \lhd w \rhd p) \mathbin{\Box} (q \lhd w \rhd p) \mathbin{\Box} (p \lhd w \rhd q) \mathbin{\Box} (q \lhd w \rhd q) \end{array}$$

That is, if any two distributions p and q are possible outcomes, then so is any convex combination $p \lhd w \rhd q$ of them. As a consequence, we should consider equivalence of collections of distributions *up to convex closure*. In particular, for *coinarb* we have:

$$\begin{array}{ll} & \mathit{coinarb}\ v \\ = & \quad [\![\text{ calculation in previous section }]\!] \\ & \mathit{return}\ \mathit{True} \mathbin{\Box} \mathit{return}\ \mathit{False} \\ = & \quad [\![\lhd\rhd \text{ distributes over } \Box, \text{ as above; arbitrary } w \ ]\!] \\ & (\mathit{return}\ \mathit{True} \lhd w \rhd \mathit{return}\ \mathit{False}) \mathbin{\Box} (\mathit{return}\ \mathit{False} \lhd w \rhd \mathit{return}\ \mathit{False}) \mathbin{\Box} \\ & (\mathit{return}\ \mathit{True} \lhd w \rhd \mathit{return}\ \mathit{True}) \mathbin{\Box} (\mathit{return}\ \mathit{False} \lhd w \rhd \mathit{return}\ \mathit{True}) \\ = & \quad [\![\text{ commutativity and idempotence of } \lhd\rhd \text{; definition of } \mathit{bcoin}_w \ ]\!] \\ & \mathit{bcoin}_w \mathbin{\Box} \mathit{return}\ \mathit{False} \mathbin{\Box} \mathit{return}\ \mathit{True} \mathbin{\Box} \mathit{bcoin}_{\overline{w}} \end{array}$$

and so the possible outcomes of *coinarb* v include all convex combinations bcoin_w of the two extreme distributions *return False* and *return True*, 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 $\mathit{bcoin}_{1/2} \mathbin{\Box} \mathit{bcoin}_{1/3}$. 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, adopting the opposite distributive law, it is straightforward to calculate as follows:

$$
\begin{array}{ll}
& p \lhd w \rhd q \\
= & \quad [\![\text{ idempotence of } \Box \quad]\!] \\
& (p \lhd w \rhd q) \Box (p \lhd w \rhd q) \\
= & \quad [\![\text{ assuming that } \Box \text{ distributes over } \lhd \rhd \quad]\!] \\
& ((p \Box p) \lhd w \rhd (p \Box q)) \lhd w \rhd ((q \Box p) \lhd w \rhd (q \Box q)) \\
= & \quad [\![\text{ idempotence and commutativity of } \Box \quad]\!] \\
& (p \lhd w \rhd (p \Box q)) \lhd w \rhd ((p \Box q) \lhd w \rhd q) \\
= & \quad [\![\text{ flattened choices, as a distribution comprehension } \quad]\!] \\
& \langle p@w^2, (p \Box q)@2\ w\ \overline{w}, q@\overline{w}^2 \rangle \\
= & \quad [\![\text{ rearranging and renesting choices } \quad]\!] \\
& (p \lhd {}^{w^2}\!/_{w^2+\overline{w}^2} \rhd q) \lhd w^2 + \overline{w}^2 \rhd (p \Box q)
\end{array}
$$

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

$$
\begin{array}{l}
p \lhd 1/2 \rhd q = (p \lhd 1/2 \rhd q) \lhd 1/2 \rhd (p \Box q) \\
p \lhd 1/3 \rhd q = (p \lhd 1/5 \rhd q) \lhd 5/9 \rhd (p \Box q)
\end{array}
$$

This seems quite an unfortunate consequence, 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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Door} = A \mid B \mid C \\
\\
\mathit{doors} :: [\mathit{Door}] \\
\mathit{doors} = [A, B, C]
\end{array}
$$

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

$$\begin{aligned}&hide :: MonadProb\ m \Rightarrow m\ Door\\&hide = uniform\ doors\end{aligned}$$

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

$$\begin{aligned}&pick :: MonadProb\ m \Rightarrow m\ Door\\&pick = uniform\ doors\end{aligned}$$

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:

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

(Here, the expression $xs \setminus\setminus ys$ 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:

$$\begin{aligned}&switch :: MonadProb\ m \Rightarrow Door \to Door \to m\ Door\\&switch\ p\ t = return\ (head\ (doors \setminus\setminus [p, t]))\end{aligned}$$

or to stick with your original choice:

$$\begin{aligned}&stick :: MonadProb\ m \Rightarrow Door \to Door \to m\ Door\\&stick\ p\ t = return\ p\end{aligned}$$

In either case, you know p and t, but of course not h.

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

$$\begin{aligned}&monty :: MonadProb\ m \Rightarrow (Door \to Door \to m\ Door) \to m\ Bool\\&monty\ strategy\\&\quad = \mathbf{do}\ \{h \leftarrow hide\ ; && \text{-- Monty hides the car behind door } h\\&\qquad\ \ p \leftarrow pick\ ; && \text{-- you pick door } p\\&\qquad\ \ t \leftarrow tease\ h\ p\ ; && \text{-- Monty teases you with door } t\ (\neq h, p)\\&\qquad\ \ s \leftarrow strategy\ p\ t\ ; && \text{-- you choose, based on } p \text{ and } t \text{ but not } h\\&\qquad\ \ return\ (s == h) && \text{-- you win iff your choice } s \text{ equals } h\\&\qquad\}\end{aligned}$$

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

$$\begin{aligned}&monty\ switch = bcoin_{2/3}\\&monty\ stick\ \ = bcoin_{1/3}\end{aligned}$$

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

$$
\begin{array}{ll}
& \mathbf{do}\,\{h \leftarrow \mathit{hide}\,;\, p \leftarrow \mathit{pick}\,;\, \mathit{return}\,(h,p)\} \\
= & \quad [\![\text{ definitions of } \mathit{hide} \text{ and } \mathit{pick} \text{ }]\!] \\
& \mathbf{do}\,\{h \leftarrow \mathit{uniform}\ \mathit{doors}\,;\, p \leftarrow \mathit{uniform}\ \mathit{doors}\,;\, \mathit{return}\,(h,p)\} \\
= & \quad [\![\text{ independent choices }]\!] \\
& \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})
\end{array}
$$

and so we have

$$
\begin{array}{l}
\mathit{monty}\ \mathit{strategy} = \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
\qquad\qquad\qquad\qquad t \leftarrow \mathit{tease}\ h\ p\,; \\
\qquad\qquad\qquad\qquad s \leftarrow \mathit{strategy}\ p\ t\,; \\
\qquad\qquad\qquad\qquad \mathit{return}\ (s \mathbin{==} h)\}
\end{array}
$$

Naturally, the doors h and p independently chosen at random will match one third of the time:

$$
\begin{array}{ll}
& \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, \mathit{return}\,(p \mathbin{==} h)\} \\
= & \quad [\![\ \mathit{fmap} \text{ and } \mathbf{do} \text{ notation }]\!] \\
& \mathbf{do}\,\{b \leftarrow \mathit{fmap}\ (\mathit{uncurry}\ (==))\ (\mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors}))\,;\, \mathit{return}\ b\} \\
= & \quad [\![\text{ right unit }]\!] \\
& \mathit{fmap}\ (\mathit{uncurry}\ (==))\ (\mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})) \\
= & \quad [\![\text{ naturality of } \mathit{uniform} \text{ }]\!] \\
& \mathit{uniform}\ (\mathit{map}\ (\mathit{uncurry}\ (==))\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})) \\
= & \quad [\![\text{ definitions of } \mathit{doors},\ \mathit{cp},\ == \text{ }]\!] \\
& \mathit{uniform}\ [\mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}] \\
= & \quad [\![\text{ simplifying: three } \mathit{True}\text{s, six } \mathit{False}\text{s }]\!] \\
& \mathit{uniform}\ [\mathit{True}, \mathit{False}, \mathit{False}] \\
= & \quad [\![\text{ definitions of } \mathit{uniform},\ \mathit{bcoin}_w \text{ }]\!] \\
& \mathit{bcoin}_{1/3}
\end{array}
$$

Therefore we calculate:

$$
\begin{array}{ll}
& \mathit{monty}\ \mathit{stick} \\
= & \quad [\![\text{ definition of } \mathit{monty}, \text{ independent uniform choices }]\!] \\
& \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
& \qquad t \leftarrow \mathit{tease}\ h\ p\,;\, s \leftarrow \mathit{stick}\ p\ t\,;\, \mathit{return}\,(s \mathbin{==} h)\} \\
= & \quad [\![\text{ definition of } \mathit{stick} \text{ }]\!] \\
& \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
& \qquad t \leftarrow \mathit{tease}\ h\ p\,;\, s \leftarrow \mathit{return}\ p\,;\, \mathit{return}\,(s \mathbin{==} h)\} \\
= & \quad [\![\text{ left unit }]\!] \\
& \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}\ h\ p\,;\, \mathit{return}\,(p \mathbin{==} h)\} \\
= & \quad [\![\ t \text{ unused, and } \mathit{uniform} \text{ side-effect-free, so } \mathit{tease} \text{ can be eliminated }]\!] \\
& \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, \mathit{return}\,(p \mathbin{==} h)\} \\
= & \quad [\![\text{ matching choices, as above }]\!] \\
& \mathit{bcoin}_{1/3}
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathit{monty}\ \mathit{switch} \\
= & \quad [\![\text{ definition of } \mathit{monty}\text{, independent uniform choices }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}\ h\ p\,; \\
 & \qquad s \leftarrow \mathit{switch}\ p\ t\,;\, \mathit{return}\ (s == h)\} \\
= & \quad [\![\text{ definition of } \mathit{switch} \ ]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}\ h\ p\,; \\
 & \qquad s \leftarrow \mathit{return}\ (\mathit{head}\ (\mathit{doors} \setminus\setminus [p,t]))\,;\, \mathit{return}\ (s == h)\} \\
= & \quad [\![\text{ left unit }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}\ h\ p\,; \\
 & \qquad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\setminus [p,t]))\} \\
= & \quad [\![\text{ case analysis on } h = p \text{—see below }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{return}\ \mathit{False}\ \mathbf{else}\ \mathit{return}\ \mathit{True}\} \\
= & \quad [\![\text{ booleans }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, \mathit{return}\ (h \neq p)\} \\
= & \quad [\![\text{ analogously, mismatching choices }]\!] \\
 & \mathit{bcoin}_{2/3}
\end{array}
$$

Now for the two branches of the case analysis. For the case $h = p$, we have:

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

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

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

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:

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

where

$$
\begin{aligned}
&\mathit{arbitrary} :: \mathit{MonadAlt}\ m \Rightarrow [a] \to m\ a \\
&\mathit{arbitrary} = \mathit{foldr}_1\ (\Box) \circ \mathit{map}\ \mathit{return}
\end{aligned}
$$

Then we define the game just as before, but with Monty behaving nondeterministically:

$$
\begin{aligned}
&\mathit{monty}_n :: \mathit{MonadAltProb}\ m \Rightarrow (\mathit{Door} \to \mathit{Door} \to m\ \mathit{Door}) \to m\ \mathit{Bool} \\
&\mathit{monty}_n\ \mathit{strategy} = \mathbf{do}\ \{h \leftarrow \mathit{hide}_n\ ; \\
&\qquad\qquad\qquad\qquad p \leftarrow \mathit{pick}\ ; \\
&\qquad\qquad\qquad\qquad t \leftarrow \mathit{tease}_n\ h\ p\ ; \\
&\qquad\qquad\qquad\qquad s \leftarrow \mathit{strategy}\ p\ t\ ; \\
&\qquad\qquad\qquad\qquad \mathit{return}\ (s == h)\}
\end{aligned}
$$

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:

$$
\begin{aligned}
&\quad \mathbf{do}\ \{h \leftarrow \mathit{hide}_n\ ; p \leftarrow \mathit{pick}\ ; \mathit{return}\ (h, p)\} \\
&= \quad [\![\ \text{let } k\ h = \mathbf{do}\ \{p \leftarrow \mathit{pick}\ ; \mathit{return}\ (h, p)\}\ ]\!] \\
&\quad \mathbf{do}\ \{h \leftarrow \mathit{hide}_n\ ; k\ h\} \\
&= \quad [\![\ \text{definition of } \mathit{hide}_n, \mathit{arbitrary}\ ]\!] \\
&\quad \mathbf{do}\ \{h \leftarrow (\mathit{return}\ A \Box \mathit{return}\ B \Box \mathit{return}\ C)\ ; k\ h\} \\
&= \quad [\![\ \text{composition distributes leftwards over } \Box\ ]\!] \\
&\quad \mathbf{do}\ \{h \leftarrow \mathit{return}\ A\ ; k\ h\} \Box \mathbf{do}\ \{h \leftarrow \mathit{return}\ B\ ; k\ h\} \Box \\
&\quad \mathbf{do}\ \{h \leftarrow \mathit{return}\ C\ ; k\ h\} \\
&= \quad [\![\ \text{left unit}\ ]\!] \\
&\quad k\ A \Box k\ B \Box k\ C \\
&= \quad [\![\ \text{definition of } k\ ]\!] \\
&\quad \mathbf{do}\ \{p \leftarrow \mathit{pick}\ ; \mathit{return}\ (A, p)\} \Box \mathbf{do}\ \{p \leftarrow \mathit{pick}\ ; \mathit{return}\ (B, p)\} \Box \\
&\quad \mathbf{do}\ \{p \leftarrow \mathit{pick}\ ; \mathit{return}\ (C, p)\}
\end{aligned}
$$

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

$$try\ d = \mathbf{do}\ \{p \leftarrow pick\ ;\ return\ (d, p)\}$$

Then we have

$$\begin{array}{ll}
& fmap\ (uncurry\ (==))\ (try\ d) \\
= & \quad [\![\ \text{definition of } try\text{; } fmap \text{ and } \mathbf{do} \text{ notation}\ ]\!] \\
& \mathbf{do}\ \{p \leftarrow pick\ ;\ return\ (d == p)\} \\
= & \quad [\![\ \text{definition of } pick\ ]\!] \\
& \mathbf{do}\ \{p \leftarrow \langle return\ A@1/3, return\ B@1/3, return\ C@1/3\rangle\ ;\ return\ (d == p)\} \\
= & \quad [\![\ \text{composition distributes leftwards over } \triangleleft\triangleright\text{; right unit}\ ]\!] \\
& \langle return\ (d == A)@1/3, return\ (d == B)@1/3, return\ (d == C)@1/3\rangle \\
= & \quad [\![\ d :: Door\text{, and so } d \text{ is one of } A, B, C\ ]\!] \\
& \langle return\ True@1/3, return\ False@1/3, return\ False@1/3\rangle \\
= & \quad [\![\ \text{definition of } bcoin_w\ ]\!] \\
& bcoin_{1/3}
\end{array}$$

and therefore

$$\begin{array}{ll}
& \mathbf{do}\ \{h \leftarrow hide_n\ ;\ p \leftarrow pick\ ;\ return\ (h == p)\} \\
= & \quad [\![\ fmap \text{ and } \mathbf{do} \text{ notation}\ ]\!] \\
& fmap\ (uncurry\ (==))\ (\mathbf{do}\ \{h \leftarrow hide_n\ ;\ p \leftarrow pick\ ;\ return\ (h, p)\}) \\
= & \quad [\![\ \text{combining first two choices, as above}\ ]\!] \\
& fmap\ (uncurry\ (==))\ (try\ A \Box try\ B \Box try\ C) \\
= & \quad [\![\ \text{naturality: } fmap\ f\ (p \Box q) = fmap\ f\ p \Box fmap\ f\ q\ ]\!] \\
& fmap\ (uncurry\ (==))\ (try\ A) \Box fmap\ (uncurry\ (==))\ (try\ B) \Box \\
& fmap\ (uncurry\ (==))\ (try\ C) \\
= & \quad [\![\ \text{matching choices, above}\ ]\!] \\
& bcoin_{1/3} \Box bcoin_{1/3} \Box bcoin_{1/3} \\
= & \quad [\![\ \text{idempotence of } \Box\ ]\!] \\
& bcoin_{1/3}
\end{array}$$

and the conclusion is still that

$$\begin{array}{ll}
monty_n\ switch & = bcoin_{2/3} \\
monty_n\ stick & = bcoin_{1/3}
\end{array}$$

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:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadExcept}\ m\ \textbf{where}\\ \quad \mathit{throw} :: m\ a\\ \quad \mathit{catch} :: m\ a \to m\ a \to m\ a\end{array}$$

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 *throw* is the computation that immediately fails, and that $p\ `\mathit{catch}`\ q$ represents the computation that behaves like p, except if this fails, in which case it continues as the exception handler q. (In Haskell, backquotes turn a prefix function into an infix operator; so $p\ `\mathit{catch}`\ q = \mathit{catch}\ p\ q$.) We stipulate that *throw* is a left zero of composition, so that a failure discards the subsequent part of a computation:

$$\textbf{do}\ \{x \leftarrow \mathit{throw}\ ;\ k\ x\} = \mathit{throw}$$

We do not stipulate that *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 *catch*; in general,

$$\textbf{do}\ \{x \leftarrow (p\ `\mathit{catch}`\ q)\ ;\ k\ x\} \neq \textbf{do}\ \{x \leftarrow p\ ;\ k\ x\}\ `\mathit{catch}`\ \textbf{do}\ \{x \leftarrow q\ ;\ k\ x\}$$

because the right-hand side brings exceptions raised by the first k under the influence of the handler q, whereas exceptions of k on the left are not handled. We do stipulate that *return* is a left zero of *catch*, so that pure computations never fail:

$$\mathit{return}\ x\ `\mathit{catch}`\ p = \mathit{return}\ x$$

Finally, *throw* and *catch* form a monoid:

$$\begin{array}{ll}p\ `\mathit{catch}`\ \mathit{throw} & = p\\ \mathit{throw}\ `\mathit{catch}`\ p & = p\\ p\ `\mathit{catch}`\ (q\ `\mathit{catch}`\ r) & = (p\ `\mathit{catch}`\ q)\ `\mathit{catch}`\ r\end{array}$$

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:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

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

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where}\\ \quad \begin{array}{ll}\mathit{return}\ x & = \mathit{Just}\ x\\ \mathit{Just}\ x \gg\!= k & = k\ x\\ \mathit{Nothing} \gg\!= k & = \mathit{Nothing}\end{array}\end{array}$$

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

$$\begin{aligned}&\textbf{instance } \mathit{MonadExcept}\ \mathit{Maybe} \textbf{ where}\\ &\quad \mathit{throw} &&= \mathit{Nothing}\\ &\quad \mathit{Just}\ x\ \text{`}\mathit{catch}\text{`}\ q &&= \mathit{Just}\ x\\ &\quad \mathit{Nothing}\ \text{`}\mathit{catch}\text{`}\ q &&= q\end{aligned}$$

The names *Maybe*, *Just*, and *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 *Maybe a* is written *'a option*.)

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:

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

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 *catch* distributes over it:

$$(p \lhd w \rhd q)\ \text{`}\mathit{catch}\text{`}\ r = (p\ \text{`}\mathit{catch}\text{`}\ r) \lhd w \rhd (q\ \text{`}\mathit{catch}\text{`}\ r)$$

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 *subdistributions* [31] or *evaluations* [20]:

$$\begin{aligned}&\mathit{weight} :: \mathit{Dist}\ a \to \mathit{Prob}\\ &\mathit{weight}\ p = \mathit{sum}\ [w \mid (x, w) \leftarrow p]\end{aligned}$$

$$\begin{aligned}&\textbf{instance } \mathit{MonadExcept}\ \mathit{Dist} \textbf{ where}\\ &\quad \mathit{throw} &&= [\,]\\ &\quad p\ \text{`}\mathit{catch}\text{`}\ q &&= p +\!\!+ \mathit{scale}\ (1-\mathit{weight}\ p)\ (q)\end{aligned}$$

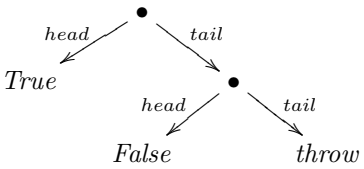
$$\textbf{instance } \mathit{MonadProbExcept}\ \mathit{Dist}$$

Operationally, the exceptions are represented by the 'missing' bits of the distribution; for example, the subdistribution $[(\mathit{True}, 1/2), (\mathit{False}, 1/4)]$ has weight $3/4$, and represents a computation that fails the remaining $1/4$ 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 $\mathit{bcoin}_{2/3}$ using two fair coin tosses, motivated by Knuth and Yao's trick [23] for simulating a fair die with three coins:

$$
\begin{aligned}
&\mathit{coins23} :: \mathit{MonadProbExcept}\ m \Rightarrow m\ \mathit{Bool}\\
&\mathit{coins23} = \mathit{return}\ \mathit{True} \triangleleft {}^1\!/_2 \triangleright (\mathit{return}\ \mathit{False} \triangleleft {}^1\!/_2 \triangleright \mathit{throw})
\end{aligned}
$$

We might illustrate the process as follows:



This does indeed yield *True* exactly twice as often as *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 in his old age—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:

$$
\begin{aligned}
&\mathit{tease}_f :: \mathit{MonadProbExcept}\ m \Rightarrow \mathit{Door} \to \mathit{Door} \to m\ \mathit{Door}\\
&\mathit{tease}_f\ h\ p = \mathbf{do}\ \{\, t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\ ;\\
&\qquad\qquad \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\,\}\\
&\mathit{monty}_f :: \mathit{MonadProbExcept}\ m \Rightarrow (\mathit{Door} \to \mathit{Door} \to m\ \mathit{Door}) \to m\ \mathit{Bool}\\
&\mathit{monty}_f\ \mathit{strategy} = \mathbf{do}\ \{\, h \leftarrow \mathit{hide}\ ;\\
&\qquad\qquad\qquad\qquad p \leftarrow \mathit{pick}\ ;\\
&\qquad\qquad\qquad\qquad t \leftarrow \mathit{tease}_f\ h\ p\ ;\\
&\qquad\qquad\qquad\qquad s \leftarrow \mathit{strategy}\ p\ t\ ;\\
&\qquad\qquad\qquad\qquad \mathit{return}\ (s == h)\,\}
\end{aligned}
$$

Investigating tease_f in the case that $h = p$, we have:

$$
\begin{aligned}
&\quad \mathit{tease}_f\ h\ p\\
&= \quad [\![\ \text{definition of } \mathit{tease}_f\ ]\!]\\
&\quad \mathbf{do}\ \{\, t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\ ;\ \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\,\}\\
&= \quad [\![\ h = p, \text{ so } h \text{ is not in } \mathit{doors} \setminus\!\setminus [p] \text{ and hence } t \neq h\ ]\!]\\
&\quad \mathbf{do}\ \{\, t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\ ;\ \mathit{return}\ t\,\}\\
&= \quad [\![\ \text{right unit}\ ]\!]\\
&\quad \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])
\end{aligned}
$$

—so if you happen to pick the car initially, Monty cannot accidentally reveal it. In the case that $h \neq p$, let $d = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [h, p])$, so that h, p, d are all distinct; then we have:

$$
\begin{array}{ll}
 & \mathit{tease}_f\ h\ p \\
= & \quad [\![\text{ definition of } \mathit{tease}_f \ ]\!] \\
 & \mathbf{do}\,\{t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\,;\, \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\} \\
= & \quad [\![\text{ by assumption, } \mathit{doors} \setminus\!\setminus [p] = [h,d] \ ]\!] \\
 & \mathbf{do}\,\{t \leftarrow \mathit{uniform}\ [h,d]\,;\, \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\} \\
= & \quad [\![\text{ definition of } \mathit{uniform} \ ]\!] \\
 & \mathbf{do}\,\{t \leftarrow \mathit{return}\ h \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;\, \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\} \\
= & \quad [\![\text{ composition distributes leftwards over } \lhd\rhd \ ]\!] \\
 & \mathbf{do}\,\{t \leftarrow \mathit{return}\ h\,;\, \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\} \lhd {}^1\!/_2 \rhd \\
 & \mathbf{do}\,\{t \leftarrow \mathit{return}\ d\,;\, \mathbf{if}\ t == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ t\} \\
= & \quad [\![\text{ left unit }]\!] \\
 & (\mathbf{if}\ h == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ h) \lhd {}^1\!/_2 \rhd (\mathbf{if}\ d == h\ \mathbf{then}\ \mathit{throw}\ \mathbf{else}\ \mathit{return}\ d) \\
= & \quad [\![\text{ by assumption, } d \neq h \text{; conditionals }]\!] \\
 & \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d
\end{array}
$$

—that is, if you initially picked a goat, Monty has a 50–50 chance of accidentally revealing the car. Putting these together, we have:

$$
\begin{array}{l}
\mathit{tease}_f\ h\ p = \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p]) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ (\mathit{head}\ (\mathit{doors} \setminus\!\setminus [h,p])))
\end{array}
$$

Clearly, in the 'else' case, 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:

$$
\begin{array}{ll}
 & \mathit{monty}_f\ \mathit{stick} \\
= & \quad [\![\text{ definition of } \mathit{monty}_f \ ]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad t \leftarrow \mathit{tease}_f\ h\ p\,;\, s \leftarrow \mathit{stick}\ p\ t\,;\, \mathit{return}\ (s == h)\} \\
= & \quad [\![\text{ definition of } \mathit{stick}\text{; left unit }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}_f\ h\ p\,;\, \mathit{return}\ (p == h)\} \\
= & \quad [\![\text{ case analysis in } \mathit{tease}_f\text{, as above; let } d = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [h,p]) \ ]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad t \leftarrow \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\ \mathbf{else}\ (\mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d)\,; \\
 & \qquad \mathit{return}\ (p == h)\} \\
= & \quad [\![\text{ conditionals }]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad \mathbf{if}\ h == p\ \mathbf{then\ do}\,\{t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\,;\, \mathit{return}\ (p == h)\} \\
 & \qquad\qquad \mathbf{else\ \ do}\,\{t \leftarrow \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;\, \mathit{return}\ (p == h)\}\} \\
= & \quad [\![\text{ first } t \text{ is unused and } \mathit{uniform} \text{ is side-effect-free; } p == h = \mathit{True} \ ]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{return}\ \mathit{True} \\
 & \qquad\qquad \mathbf{else\ \ do}\,\{t \leftarrow \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;\, \mathit{return}\ (p == h)\}\} \\
= & \quad [\![\text{ composition distributes over } \lhd\rhd \ ]\!] \\
 & \mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \qquad \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{return}\ \mathit{True}
\end{array}
$$

$$\qquad\qquad \mathbf{else}\ \ \mathbf{do}\,\{t \leftarrow \mathit{throw}\,;\mathit{return}\ \mathit{False}\} \lhd {}^1\!/_2 \rhd$$
$$\qquad\qquad\qquad \mathbf{do}\,\{t \leftarrow \mathit{return}\ d\,;\mathit{return}\ \mathit{False}\}\}$$

$= \llbracket$ *throw* is a left zero of composition, and *return* a left unit $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;$$
$$\quad \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{return}\ \mathit{True}\ \mathbf{else}\ \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False}\}$$

$= \llbracket$ matching choices, as in Section 6 $\rrbracket$

$$\mathbf{do}\,\{b \leftarrow \mathit{bcoin}_{1/3}\,;\mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ \mathit{True}\ \mathbf{else}\ \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False}\}$$

$= \llbracket$ composition distributes over $\lhd\rhd$ $\rrbracket$

$$\mathbf{do}\,\{b \leftarrow \mathit{return}\ \mathit{True}\,;$$
$$\quad \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ \mathit{True}\ \mathbf{else}\ \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False}\} \lhd {}^1\!/_3 \rhd$$
$$\mathbf{do}\,\{b \leftarrow \mathit{return}\ \mathit{False}\,;$$
$$\quad \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ \mathit{True}\ \mathbf{else}\ \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False}\}$$

$= \llbracket$ left unit; conditionals $\rrbracket$

$$\mathit{return}\ \mathit{True} \lhd {}^1\!/_3 \rhd (\mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False})$$

$= \llbracket$ flattening choices $\rrbracket$

$$\langle \mathit{True}@{}^1\!/_3, \mathit{throw}@{}^1\!/_3, \mathit{False}@{}^1\!/_3 \rangle$$

On the other hand:

$$\mathit{monty}_f\ \mathit{switch}$$

$= \llbracket$ definition of monty_f $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; t \leftarrow \mathit{tease}_f\ h\ p\,;$$
$$\quad s \leftarrow \mathit{switch}\ p\ t\,;\mathit{return}\ (s == h)\}$$

$= \llbracket$ definition of *switch* $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; t \leftarrow \mathit{tease}_f\ h\ p\,;$$
$$\quad s \leftarrow \mathit{return}\ (\mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\,;\mathit{return}\ (s == h)\}$$

$= \llbracket$ left unit $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; t \leftarrow \mathit{tease}_f\ h\ p\,;$$
$$\quad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}$$

$= \llbracket$ case analysis in tease_f, as above; let $d = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [h,p])$ $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;$$
$$\quad t \leftarrow \mathbf{if}\ h == p\ \mathbf{then}\ \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\ \mathbf{else}\ \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;$$
$$\quad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}$$

$= \llbracket$ composition distributes leftwards over conditional $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;$$
$$\quad \mathbf{if}\ h == p\ \mathbf{then}\ \mathbf{do}\,\{t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\,;$$
$$\qquad\qquad\qquad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}$$
$$\qquad\qquad \mathbf{else}\ \ \mathbf{do}\,\{t \leftarrow \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;$$
$$\qquad\qquad\qquad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}\}$$

$= \llbracket$ in **then** branch, $h = p$, so h is not in $\mathit{doors} \setminus\!\setminus [p,t]$ $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;$$
$$\quad \mathbf{if}\ h == p\ \mathbf{then}\ \mathbf{do}\,\{t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\,;\mathit{return}\ \mathit{False}\}$$
$$\qquad\qquad \mathbf{else}\ \ \mathbf{do}\,\{t \leftarrow \mathit{throw} \lhd {}^1\!/_2 \rhd \mathit{return}\ d\,;$$
$$\qquad\qquad\qquad \mathit{return}\ (h == \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}\}$$

$= \llbracket$ composition distributes over $\lhd\rhd$ $\rrbracket$

$$\mathbf{do}\,\{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;$$

$$
\begin{array}{ll}
 & \quad \textbf{if } h = p \textbf{ then do } \{t \leftarrow \mathit{uniform}\ (\mathit{doors} \setminus\!\setminus [p])\,;\ \mathit{return}\ \mathit{False}\} \\
 & \qquad \textbf{else do } \{t \leftarrow \mathit{throw}\,;\ \mathit{return}\ (h = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\} \lhd 1/2 \rhd \\
 & \qquad\quad \textbf{do } \{t \leftarrow \mathit{return}\ d\,;\ \mathit{return}\ (h = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}\} \\
= & \quad [\![\text{ in } \textbf{then} \text{ branch, } t \text{ is unused and } \mathit{uniform} \text{ is side-effect-free }]\!] \\
 & \textbf{do } \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \quad \textbf{if } h = p \textbf{ then } \mathit{return}\ \mathit{False} \\
 & \qquad \textbf{else do } \{t \leftarrow \mathit{throw}\,;\ \mathit{return}\ (h = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\} \lhd 1/2 \rhd \\
 & \qquad\quad \textbf{do } \{t \leftarrow \mathit{return}\ d\,;\ \mathit{return}\ (h = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,t]))\}\} \\
= & \quad [\![\ \mathit{throw} \text{ is left zero, } \mathit{return} \text{ is left unit }]\!] \\
 & \textbf{do } \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \quad \textbf{if } h = p \textbf{ then } \mathit{return}\ \mathit{False} \\
 & \qquad \textbf{else } \mathit{throw} \lhd 1/2 \rhd (\mathit{return}\ (h = \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,d])))\} \\
= & \quad [\![\ h,p,d \text{ are distinct, so } \mathit{head}\ (\mathit{doors} \setminus\!\setminus [p,d]) = h \ ]\!] \\
 & \textbf{do } \{(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,; \\
 & \quad \textbf{if } h = p \textbf{ then } \mathit{return}\ \mathit{False} \textbf{ else } \mathit{throw} \lhd 1/2 \rhd (\mathit{return}\ \mathit{True})\} \\
= & \quad [\![\text{ matching choices }]\!] \\
 & \textbf{do } \{b \leftarrow \mathit{bcoin}_{1/3}\,; \textbf{if } b \textbf{ then } \mathit{return}\ \mathit{False} \textbf{ else } \mathit{throw} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}\} \\
= & \quad [\![\text{ composition distributes over } \lhd\rhd \ ]\!] \\
 & \textbf{do } \{b \leftarrow \mathit{return}\ \mathit{True}\,; \\
 & \quad \textbf{if } b \textbf{ then } \mathit{return}\ \mathit{False} \textbf{ else } \mathit{throw} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}\} \lhd 1/3 \rhd \\
 & \textbf{do } \{b \leftarrow \mathit{return}\ \mathit{False}\,; \\
 & \quad \textbf{if } b \textbf{ then } \mathit{return}\ \mathit{False} \textbf{ else } \mathit{throw} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}\} \\
= & \quad [\![\text{ left unit; conditionals }]\!] \\
 & \mathit{return}\ \mathit{False} \lhd 1/3 \rhd (\mathit{throw} \lhd 1/2 \rhd \mathit{return}\ \mathit{True}) \\
= & \quad [\![\text{ flattening choices }]\!] \\
 & \langle \mathit{False}@1/3, \mathit{throw}@1/3, \mathit{True}@1/3 \rangle
\end{array}
$$

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.

$$\mathit{monty}_f\ \mathit{stick} = \mathit{monty}_f\ \mathit{switch} = \langle \mathit{True}@1/3, \mathit{False}@1/3, \mathit{throw}@1/3 \rangle$$

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 $P(A \mid B)$ 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:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

Operationally, a subdistribution p with weight $w > 0$ represents the distribution of outcomes drawn from the normalized distribution $\mathit{scale}\ (1/w)\ p$, conditioned by the non-occurrence of the outcomes outside the support of p.

$$normalize :: Dist\ a \to Dist\ a$$
$$normalize\ p = scale\ (1/_{weight\ p})\ p$$

For example, the subdistribution

$$coins23 = [(True, 1/2), (False, 1/4)]$$

from Section 7.1 has weight $3/4$, and so represents the distribution of outcomes

$$[(True, 1/2 \div 3/4 = 2/3), (False, 1/4 \div 3/4 = 1/3)] = bcoin_{2/3}$$

given that one does not toss two tails—so the attempt to simulate the biased $bcoin_{2/3}$ 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 bluffs his way through his amnesia.

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

$$rain :: MonadProb\ m \Rightarrow m\ Bool$$
$$rain = bcoin_{3/10}$$

$$sprinkler :: MonadProb\ m \Rightarrow m\ Bool$$
$$sprinkler = bcoin_{1/2}$$

$$grassWet :: MonadProb\ m \Rightarrow Bool \to Bool \to m\ Bool$$
$$grassWet\ r\ s = \mathbf{do}\ \{x \leftarrow bcoin_{9/10}\ ;\ y \leftarrow bcoin_{8/10}\ ;\ z \leftarrow bcoin_{1/10}\ ;$$
$$return\ ((x \wedge r) \vee (y \wedge s) \vee z)\}$$

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

$$experiment :: MonadProbExcept\ m \Rightarrow m\ Bool$$
$$experiment = \mathbf{do}\ \{r \leftarrow rain\ ;\ s \leftarrow sprinkler\ ;\ g \leftarrow grassWet\ r\ s\ ;$$
$$\mathbf{if}\ g\ \mathbf{then}\ return\ r\ \mathbf{else}\ throw\}$$

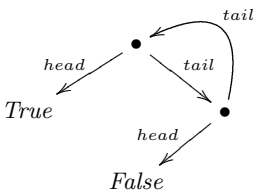
We simply return whether it is raining, conditioned on whether the grass is wet:

$$normalize\ experiment = [(False, 1610/3029), (True, 1419/3029)]$$

—that is, it is raining with probability $1419/3029 \simeq 0.47$.

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:



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

$$\begin{aligned}&\mathit{thirds} :: \mathit{MonadProbExcept}\ m \Rightarrow m\ \mathit{Bool}\\&\mathit{thirds} = \mathit{coins23}\ \text{`}\mathit{catch}\text{`}\ \mathit{thirds}\end{aligned}$$

In fact, we can inline the exception and its handler, on account of the various laws of $\triangleleft\triangleright$: for any p,

$$\begin{aligned}&\mathit{coins23}\ \text{`}\mathit{catch}\text{`}\ p\\=\quad&[\![\ \text{definition of } \mathit{coins23}\ ]\!]\\&(\mathit{return}\ \mathit{True} \triangleleft {}^1\!/_2 \triangleright (\mathit{return}\ \mathit{False} \triangleleft {}^1\!/_2 \triangleright \mathit{throw}))\ \text{`}\mathit{catch}\text{`}\ p\\=\quad&[\![\ \mathit{catch} \text{ distributes over } \triangleleft\triangleright, \text{ twice}\ ]\!]\\&(\mathit{return}\ \mathit{True}\ \text{`}\mathit{catch}\text{`}\ p) \triangleleft {}^1\!/_2 \triangleright\\&\quad((\mathit{return}\ \mathit{False}\ \text{`}\mathit{catch}\text{`}\ p) \triangleleft {}^1\!/_2 \triangleright (\mathit{throw}\ \text{`}\mathit{catch}\text{`}\ p))\\=\quad&[\![\ \mathit{return} \text{ is left zero of } \mathit{catch}, \text{ and } \mathit{throw} \text{ is left unit}\ ]\!]\\&\mathit{return}\ \mathit{True} \triangleleft {}^1\!/_2 \triangleright (\mathit{return}\ \mathit{False} \triangleleft {}^1\!/_2 \triangleright p)\end{aligned}$$

So we could have defined instead

$$\begin{aligned}&\mathit{thirds} :: \mathit{MonadProb}\ m \Rightarrow m\ \mathit{Bool}\\&\mathit{thirds} = \mathit{return}\ \mathit{True} \triangleleft {}^1\!/_2 \triangleright (\mathit{return}\ \mathit{False} \triangleleft {}^1\!/_2 \triangleright \mathit{thirds})\end{aligned}$$

Note that this definition can be given a more specific type, because it no longer exploits 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

$$[(\mathit{True}, 1/2), (\mathit{False}, 1/4), (\mathit{True}, 1/8), (\mathit{False}, 1/16), (\mathit{True}, 1/32), (\mathit{False}, 1/64), ...]$$

One might see this as a reasonable representation of $\mathit{bcoin}_{2/3}$: 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 $\mathit{bcoin}_{2/3}$ is a solution to the same recursive equation as *thirds*:

$$\begin{aligned}
& \mathit{return}\ \mathit{True} \lhd 1/2 \rhd (\mathit{return}\ \mathit{False} \lhd 1/2 \rhd \mathit{bcoin}_{2/3}) \\
= \quad & [\![\text{ definition of } \mathit{bcoin}_w \ ]\!] \\
& \mathit{return}\ \mathit{True} \lhd 1/2 \rhd (\mathit{return}\ \mathit{False} \lhd 1/2 \rhd (\mathit{return}\ \mathit{True} \lhd 2/3 \rhd \mathit{return}\ \mathit{False})) \\
= \quad & [\![\text{ flattening choices }]\!] \\
& \langle \mathit{return}\ \mathit{True} @ 1/2, \mathit{return}\ \mathit{False} @ 1/4, \mathit{return}\ \mathit{True} @ 1/6, \mathit{return}\ \mathit{False} @ 1/12 \rangle \\
= \quad & [\![\text{ combining collisions }]\!] \\
& \langle \mathit{return}\ \mathit{True} @ 2/3, \mathit{return}\ \mathit{False} @ 1/3 \rangle \\
= \quad & [\![\text{ definition of } \mathit{bcoin}_w \ ]\!] \\
& \mathit{bcoin}_{2/3}
\end{aligned}$$

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

$$\mathit{thirds} = (\mathit{thirds} \lhd 1/2 \rhd \mathit{return}\ \mathit{False}) \lhd 1/2 \rhd \mathit{return}\ \mathit{True}$$

instead, the least solution would be $\bot$, the least-defined element in the information ordering, because the definition of $\lhd\rhd$ 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 $[(\mathit{True}, 1/2), (\mathit{False}, 1/2)]$. 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 $\lhd\rhd$ operation:

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

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

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

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

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

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 $\triangleleft\triangleright$ 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:

$$\begin{aligned}&\mathit{collectT} :: \mathit{Eq}\ a \Rightarrow \mathit{DistT}\ a \to (a \to \mathit{Prob})\\ &\mathit{collectT}\ (\mathit{Return}\ x)\ y \qquad = \textbf{if } x == y \textbf{ then } 1 \textbf{ else } 0\\ &\mathit{collectT}\ (\mathit{Choice}\ w\ p\ q)\ y = w \times \mathit{collectT}\ p\ y + \overline{w} \times \mathit{collectT}\ q\ y\end{aligned}$$

and equivalences on *DistT* ought again to be taken modulo permutations, zero-weighted elements, and repeated elements (whose weights should be added). Nevertheless, the datatype *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 *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, $\triangleleft\triangleright$ is non-strict in both arguments, so the semantics of a recursive definition is much less sensitive to the precise form of the recursion.

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 $p = (p \gg\!= \mathit{return})$. Even some equations that completely determine proper choice trees do not define productive sampling behaviour; for example,

$$p = p \triangleleft 1 \triangleright \mathit{return}\ \mathit{True}$$

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

$$p = p \lhd \tfrac{1}{2} \rhd p$$

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 $\lhd\rhd$, and for each such $\lhd\rhd$ to devote positive weight to a non-recursive subcomputation. But it seems rather unsatisfactory to have to consider specific implementations such as *DistT* of the *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, together with exceptions, but the approach works for any class of effects; our earlier paper [10] also discusses counting, generating fresh names, and mutable state. 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 (not formally published) 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 *reasoning* about programs: the techniques of simple high-school algebra suffice, and one can reason directly in the language of programs, rather than having to extract verification conditions from the programs and then reason indirectly in predicate calculus. In that respect, functional programming has a similar motivation to Hoare's "programs are predicates" work [16], Hehner's predicative programming [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 existing language constructs such as algebraic datatypes, higher-order functions, and lazy evaluation, rather than having to step outside the existing language to define a new one. In that respect, functional programming goes beyond predicative programming and refinement calculus, which are not expressive enough to support such extension. (Of course, embedded DSLs have their limits. Sometimes a desired feature cannot be defined conveniently, if at all, in terms of existing constructs; then the best approach is to relent and to define a new language after all.)

9.1 Nondeterminism and Probability in Program Calculi

Dijkstra [6] argued forcefully for the centrality of nondeterminism in programming 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 programs, 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 approaches 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 powerdomain 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 in 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,

$$arbcoin\ v = (arb \lhd w \rhd return\ False) \lhd w \rhd (return\ True \lhd w \rhd arb)$$

That is, with probability $w^2 + \overline{w}^2$ (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 $\lhd\rhd$, and distributivity of $\Box$ over $\lhd\rhd$, one can show that

$$coin = coin \lhd {}^1\!/_2 \rhd arb$$

which seems a most unwelcome property: 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 *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 each 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 *arbcoin* and *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

$$\begin{aligned} &\mathit{naturals} :: \mathit{MonadProb}\ m \Rightarrow \mathit{Integer} \to m\ \mathit{Integer} \\ &\mathit{naturals}\ n = \mathit{return}\ n \lhd {}^1\!/_2 \rhd \mathit{naturals}\ (n+1) \end{aligned}$$

one might hope that *naturals* 0 returns result i with probability $1/2^{i+1}$, possibly returning any of the naturals. This works in a lazy language like Haskell, provided that the definition of $\lhd\rhd$ 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

$$[(0, {}^1\!/_2), (1, {}^1\!/_4), (2, {}^1\!/_8), (3, {}^1\!/_{16}), (4, {}^1\!/_{32}), (5, {}^1\!/_{64}), \ldots]$$

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 $DistT$, 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 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. An approach that separates syntax and 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 isloating 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:

$$\mathbf{do}\,\{x \leftarrow (p \triangleleft w \triangleright q)\,;\,k\;x\} = \mathbf{do}\,\{x \leftarrow p\,;\,k\;x\} \triangleleft w \triangleright\ \mathbf{do}\,\{x \leftarrow q\,;\,k\;x\}$$

But as we saw in Section 7, bind does not distribute over *catch*. Plotkin and Pretnar [43] call operations like *catch* 'effect handlers'; they are not 'algebraic effects', and need a different treatment.

Acknowledgements. We are grateful to members of the *Algebra of Programming* research group at Oxford and 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*.

References

1. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998), graduate Texts in Computer Science
2. Beck, J.: Distributive laws. In: Seminar on Triples and Categorical Homology Theory. Lecture Notes in Mathematics, vol. 80, pp. 119–140. Springer (1969)
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall (1997)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Operating Systems Design & Implementation, pp. 137–150. USENIX Association (2004)
5. Deng, Y., van Glabbeek, R., Hennessy, M., Morgan, C., Zhang, C.: Remarks on testing probabilistic processes. Electronic Notes in Theoretical Computer Science 172, 359–397 (2007)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall Series in Automatic Computation. Prentice-Hall (1976)
7. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification. Springer (1985)
8. Eilenberg, S., Moore, J.C.: Adjoint functors and triples. Illinois Journal of Mathematics, 381–398 (1965)
9. Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. Journal of Functional Programming 16(1), 21–34 (2006)
10. Gibbons, J., Hinze, R.: Just `do` it: Simple monadic equational reasoning. In: Danvy, O. (ed.) International Conference on Functional Programming, pp. 2–14. ACM, New York (2011)
11. Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer (1981)
12. He, J., Seidel, K., McIver, A.: Probabilistic models for the Guarded Command Language. Science of Computer Programming 28, 171–192 (1997)
13. Hehner, E.C.R.: Predicative programming, parts I and II. Communications of the ACM 27(2), 134–151 (1984)
14. Hinze, R.: Deriving backtracking monad transformers. In: International Conference on Functional Programming, pp. 186–197 (2000)
15. Hoare, C.A.R.: A couple of novelties in the propositional calculus. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 31(2), 173–178 (1985)
16. Hoare, C.A.R., Hanna, F.K.: Programs are predicates. Philosophical Transactions of the Royal Society, Part A 312(1522), 475–489 (1984)

17. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
18. Hutton, G., Fulger, D.: Reasoning about effects: Seeing the wood through the trees. In: Preproceedings of Trends in Functional Programming (May 2008)
19. Hyland, M., Power, J.: The category theoretic understanding of universal algebra: Lawvere theories and monads. Electronic Notes in Theoretical Computer Science 172, 437–458 (2007)
20. Jones, C., Plotkin, G.: A probabilistic powerdomain of evaluations. In: Logic in Computer Science, pp. 186–195 (1989)
21. Kiselyov, O., Shan, C.-c.: Embedded Probabilistic Programming. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 360–384. Springer, Heidelberg (2009)
22. Kleisli, H.: Every standard construction is induced by a pair of adjoint functors. Proceedings of the American Mathematical Society 16, 544–546 (1965)
23. Knuth, D.E., Yao, A.C.C.: The complexity of nonuniform random number generation. In: Traub, J.F. (ed.) Algorithms and Complexity: New Directions and Recent Results, pp. 357–428. Academic Press (1976); reprinted in Selected Papers on Analysis of Algorithms (CSLI 2000)
24. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22, 328–350 (1981)
25. Launchbury, J.: Lazy imperative programming. In: ACM SIGPLAN Workshop on State in Programming Languages (June 1993)
26. Lawvere, F.W.: Functorial Semantics of Algebraic Theories. Ph.D. thesis, Columbia University, also available with commentary as Theory and Applications of Categories Reprint 5 (1963),
`http://www.tac.mta.ca/tac/reprints/articles/5/tr5abs.html`
27. Lawvere, F.W.: The category of probabilistic mappings (1962) (preprint)
28. Linton, F.E.J.: Some aspects of equational theories. In: Categorical Algebra, pp. 84–95. Springer, La Jolla (1966)
29. Lowe, G.: Representing nondeterministic and probabilistic behaviour in reactive processes (1993) (manuscript) Oxford University Computing Laboratory
30. Mac Lane, S.: Categories for the Working Mathematician. Springer (1971)
31. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer (2005)
32. McIver, A.K., Weber, T.: Towards Automated Proof Support for Probabilistic Distributed Systems. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 534–548. Springer, Heidelberg (2005)
33. McKinna, J.: Why dependent types matter. In: Principles of Programming Languages, p. 1. ACM (2006), full paper available at,
`http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`
34. Meinicke, L., Solin, K.: Refinement algebra for probabilistic programs. Formal Aspects of Computing 22, 3–31 (2010)
35. Mislove, M.W.: Nondeterminism and Probabilistic Choice: Obeying the Laws. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 350–364. Springer, Heidelberg (2000)
36. Moggi, E.: Notions of computation and monads. Information and Computation 93(1) (1991)
37. Morgan, C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
38. Morgan, C.: Elementary Probability Theory in the Eindhoven Style. In: Gibbons, J., Nogueira, P. (eds.) MPC 2012. LNCS, vol. 7342, pp. 48–73. Springer, Heidelberg (2012)
39. Morgan, C., McIver, A., Seidel, K., Sanders, J.W.: Refinement-oriented probability for CSP. Formal Aspects of Computing 8(6), 617–647 (1996)

40. Peyton Jones, S.: Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Hoare, T., Broy, M., Steinbruggen, R. (eds.) Engineering Theories of Software Construction, pp. 47–96. IOS Press (2001)
41. Piróg, M., Gibbons, J.: Tracing monadic computations and representing effects. In: Mathematically Structured Functional Programming (March 2012)
42. Plotkin, G., Power, J.: Notions of Computation Determine Monads. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002)
43. Plotkin, G., Pretnar, M.: Handlers of Algebraic Effects. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 80–94. Springer, Heidelberg (2009)
44. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages, pp. 154–165 (2002)
45. Rosenhouse, J.: The Monty Hall Problem: The Remarkable Story of Math's Most Contentious Brain Teaser. Oxford University Press (2009)
46. Spivey, M.: A functional theory of exceptions. Science of Computer Programming 14, 25–42 (1990)
47. Spivey, M., Seres, S.: Combinators for logic programming. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, pp. 177–200. Cornerstones in Computing, Palgrave (2003)
48. Turner, D.A.: Total functional programming. Journal of Universal Computer Science 10(7), 751–768 (2004)
49. Varacca, D., Winskel, G.: Distributing probability over nondeterminism. Mathematical Structures in Computer Science 16, 87–113 (2006)
50. Vos Savant, M.: Ask Marilyn. Parade Magazine (September 9th, 1990), http://www.marilynvossavant.com/articles/gameshow.html
51. Wadler, P.: How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985), http://dx.doi.org/10.1007/3-540-15975-4_33
52. Wadler, P.: Theorems for free? In: Functional Programming Languages and Computer Architecture, pp. 347–359. ACM (1989), http://doi.acm.org/10.1145/99370.99404
53. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2(4), 461–493 (1992)
54. Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi: Proceedings of the Marktoberdorf Summer School (1992)
55. Yi, W., Larsen, K.G.: Testing probabilistic and nondeterministic processes. In: Linn Jr., R.J., Uyar, M.Ü. (eds.) Protocol Specification, Testing and Verification, pp. 47–61. North-Holland (1992)