Just do It: Simple Monadic Equational Reasoning

Jeremy Gibbons and Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
http://www.cs.ox.ac.uk/{jeremy.gibbons,ralf.hinze}/

Abstract

One of the appeals of pure functional programming is that it is so amenable to equational reasoning. One of the problems of pure functional programming is that it rules out computational effects. Moggi and Wadler showed how to get round this problem by using monads to encapsulate the effects, leading in essence to a phase distinction—a pure functional evaluation yielding an impure imperative computation. Still, it has not been clear how to reconcile that phase distinction with the continuing appeal of functional programming; does the impure imperative part become inaccessible to equational reasoning? We think not; and to back that up, we present a simple axiomatic approach to reasoning about programs with computational effects.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs, Programming by contract; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Assertions, Logics of programs, Pre- and post-conditions, Specification techniques; F.3.3 [*Studies of Program Constructs*]: Functional constructs.

General Terms Languages, Theory, Verification.

Keywords Monads, equational reasoning, Lawvere theories, algebraic specification.

1. Introduction

Pure functional programming languages are good for equational reasoning [36]: non-strict semantics supports algebraic manipulation through the principle of 'substitution of equals for equals'. But on the face of it, purity is very limiting: many programs of interest involve impure computational effects, such as mutable state and nondeterminism. Famously, Moggi [16] and Wadler [38] showed how *monads* can be used to neatly encapsulate these effects. A pure program can assemble a representation of an effectful computation as a plain value, which can then be executed at the top level instead of being printed out—'Haskell is the world's finest imperative programming language' [21].

ICFP'11, September 19–21, 2011, Tokyo, Japan.

Nevertheless, little work has been done on the combination of equational reasoning and programs exhibiting computational effects. Current practice involves simulating an effectful computation as a pure function, and conducting the reasoning on this pure value; for example, a recent paper by Hutton and Fulger [9] presents a proof of correctness of a stateful computation using equational reasoning in terms of pure state-transforming functions. This works, but it is unsatisfactory in a number of ways: for one thing, it breaches the abstraction boundary provided by the monadic interface, inhibiting any reuse of proof efforts across programs using different classes of effect; and for another, not all computational effects can be adequately simulated in this way.

In this paper, we present a simple axiomatic approach to equational reasoning with monadic programs, preserving the monadic abstraction. The key idea is to exploit the algebraic properties of both the generic monad interface (the 'return' and 'bind' operators, and their associativity and unit laws) and the specific operations used to support a particular class of effects (for example, the 'put' and 'get' operations of mutable state, or the 'choice' points in nondeterministic computations). In essence, this is an approach based more on the 'algebraic theory' interpretation of universal algebra within category theory [13] than on the later 'monad' interpretation [14] more familiar to functional programmers. Either can be used to model computational effects in a pure setting [10], with nearly equivalent expressiveness, but the 'algebraic theory' interpretation emphasizes the specifics of a particular class of effects, whereas the 'monad' interpretation focusses on the general notion of effectful computations and says little about specific effects.

In passing, we present (what we believe to be) a novel approach to reasoning about programs that exploit both nondeterministic and probabilistic choice. The monad of nondeterminism (that is, the list functor, or more accurately the finite powerset functor) is very familiar to functional programmers [38, 20]. Less well known in functional programming circles, but nevertheless well established in programming language semantics, is the monad of probability distributions [5, 11, 27, 3]. We show that these two monads combine neatly, providing a simple unified model of nondeterminism and probability, supporting the same simple equational reasoning as any other monad. Models for this combination of effects have been given before (see for example [15]), but we believe that none are as straightforward as ours.

The remainder of the paper is structured as follows. We warm up in Section 3 with a discussion of a simple problem involving effectful computation—counting the disc moves in the Towers of Hanoi puzzle. In Sections 4, 5, and 6 we consider some more interesting effects: nondeterminism, exceptions, and mutable state, respectively. In Section 7 we look at programs that combine two classes of effect, nondeterminism and state; and in Section 8 we look at

probabilistic computations, and their combination with nondeterminism. Finally, we return to Hutton and Fulger's tree relabelling problem in Section 9, and Section 10 concludes. But we start in Section 2 with some background on monads.

2. Background

As is well known [16, 38], the categorical notion of a *monad* precisely expresses an abstraction of sequential computations, which is necessary for controlling the consequences of computational effects in a lazy functional programming language. The 'return' and 'bind' methods model the identity and sequential composition of computations, respectively. They are captured in the following type class:

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

The two operations are required to satisfy three laws, corresponding to the monoidal properties of sequential composition:

$$\begin{array}{ll}\mathit{return}\ x \gg\!= k & = k\ x\\ mx \gg\!= \mathit{return} & = mx\\ (mx \gg\!= k) \gg\!= k' & = mx \gg\!= (\lambda x \to k\ x \gg\!= k')\end{array}$$

We will make use of two important specializations of the operations associated with a monad, which more precisely match the idea of the identity computation and sequential composition:

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

$$\begin{array}{ll}(\gg) & :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b\\ mx \gg my & = mx \gg\!= \mathit{const}\ my\end{array}$$

We will also make significant use of the derived operator

$$\begin{array}{l}\mathit{liftM} :: \mathit{Monad}\ m \Rightarrow (a \to b) \to m\ a \to m\ b\\ \mathit{liftM}\ f\ mx = mx \gg\!= \mathit{return} \circ f\end{array}$$

The reader may enjoy checking that the unit and associativity properties imply that *liftM* satisfies the map laws of a functor:

$$\begin{array}{ll}\mathit{liftM}\ \mathit{id} & = \mathit{id}\\ \mathit{liftM}\ (f \circ g) & = \mathit{liftM}\ f \circ \mathit{liftM}\ g\end{array}$$

An uncurried version of Haskell's *liftM2* can be defined as $\mathit{liftM}\ f \circ \mathit{pair}$, where *pair* sequences a pair of monadic computations:

$$\begin{array}{l}\mathit{pair} :: \mathit{Monad}\ m \Rightarrow (m\ a, m\ a) \to m\ (a,a)\\ \mathit{pair}\ (mx,my) = mx \gg\!= \lambda x \to my \gg\!= \lambda y \to \mathit{return}\ (x,y)\end{array}$$

2.1 Imperative functional programming

The return and bind operations of the *Monad* class are sufficient to support a very convenient monad comprehension or '**do**' notation for computations, offering an elegant imperative programming style [37, 23]. The body of a **do** expression consists of a non-empty sequence of *qualifiers*, which may be *expressions*, *generators*, or *local definitions*—except for the last qualifier, which must be an expression. In this paper, we restrict attention to patterns consisting of simple variables or tuples of such; then the **do** notation can be defined in terms of just $\gg\!=$ and ordinary **let** (and *return*, which is usually needed for the final expression):

$$\begin{array}{ll}\textbf{do}\ \{e\} & = e\\ \textbf{do}\ \{e; es\} & = e \gg \textbf{do}\ \{es\}\\ \textbf{do}\ \{x \leftarrow e; es\} & = e \gg\!= \lambda x \to \textbf{do}\ \{es\}\\ \textbf{do}\ \{\textbf{let}\ decls; es\} & = \textbf{let}\ decls\ \textbf{in do}\ \{es\}\end{array}$$

For example, an alternative definition of *pair* is

$$\mathit{pair}\ (mx,my) = \textbf{do}\ \{x \leftarrow mx; y \leftarrow my; \mathit{return}\ (x,y)\}$$

(More generally, the pattern match against a generated value might be refutable; then some mechanism for handling failed matches is necessary. In Haskell, the *Monad* class has an additional operation *fail* for this purpose; we will do without.)

The **do** notation makes it clearer that the monad laws are not awkward impositions, but properties essential for manipulating sequential compositions and blocks:

$$\begin{array}{ll}\textbf{do}\ \{y \leftarrow \mathit{return}\ x; k\ y\} & = \textbf{do}\ \{k\ x\}\\ \textbf{do}\ \{x \leftarrow mx; \mathit{return}\ x\} & = \textbf{do}\ \{mx\}\\ \textbf{do}\ \{x \leftarrow mx; y \leftarrow k\ x; k'\ y\} & = \textbf{do}\ \{y \leftarrow \textbf{do}\ \{x \leftarrow mx; k\ x\}; k'\ y\}\end{array}$$

where, in the third law, x is not free in k'. The first two laws state that *return* is a unit of sequential composition, and the third that nested blocks can be flattened, given appropriate care over bound variables.

3. A counter example: Towers of Hanoi

The *Monad* class specifies only the very basic general-purpose plumbing of sequential composition. To obtain any interesting computational effects, one must augment the interface with additional methods. In this paper, we will consistently do this by defining subclasses of *Monad*, rather than by declaring instances of *Monad* with additional operations; and we will studiously program and reason according to the interface, not to a particular implementation.

For example, here is a class of monads supporting a simple effect of counting:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadCount}\ m\ \textbf{where}\\ \quad \mathit{tick} :: m\ ()\end{array}$$

And here is a program for solving the Towers of Hanoi problem—it ticks the counter once for each move of a disc.

$$\begin{array}{ll}\mathit{hanoi} :: \mathit{MonadCount}\ m \Rightarrow \mathit{Int} \to m\ ()\\ \mathit{hanoi}\ 0 & = \mathit{skip}\\ \mathit{hanoi}\ (n+1) & = \mathit{hanoi}\ n \gg \mathit{tick} \gg \mathit{hanoi}\ n\end{array}$$

We claim that

$$\mathit{hanoi}\ n = \mathit{rep}\ (2^n - 1)\ \mathit{tick}$$

where *rep* repeats a unit computation a fixed number of times:

$$\begin{array}{ll}\mathit{rep} :: \mathit{Monad}\ m \Rightarrow \mathit{Int} \to m\ () \to m\ ()\\ \mathit{rep}\ 0 & mx = \mathit{skip}\\ \mathit{rep}\ (n+1)\ mx & = mx \gg \mathit{rep}\ n\ mx\end{array}$$

(This is a type specialization of the function *replicateM_* in the Haskell standard library.) Note that

$$\begin{array}{ll}\mathit{rep}\ 1 & mx = mx\\ \mathit{rep}\ (m+n)\ mx & = \mathit{rep}\ m\ mx \gg \mathit{rep}\ n\ mx\end{array}$$

The verification of *hanoi* is by induction on n. The base case is trivial; for the inductive step, we assume the result for n, and calculate:

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

This is a very simple example, because the additional algebraic structure in *MonadCount* is free—no laws were imposed on *tick*, and the only equivalences between counting programs are those induced by the monoidal structure of sequential composition. In the rest of the paper, we will look at more interesting classes of effect.

4. Nondeterministic computations

Nondeterministic programs are characterized by the ability to choose between multiple results, and (sometimes) by the possibility of returning no result at all. We express these as two separate subclasses of *Monad*.

4.1 Failure

We consider failure first, because it is the simpler feature:

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

The *fail* operation is a left zero of sequential composition:

$$fail \gg m = fail$$

but not a right zero—a computation of the form $mx \gg fail$ might yield some effects from mx before failing, and so be different from *fail* alone.

Note that this is a different design from that in Haskell standard [22], which provides a *fail* method in the *Monad* class itself (as discussed in Section 2.1), as well as an *mzero* method as part of a more extensive *MonadPlus* class (which we will discuss below)—but which is silent on the properties expected of *fail*. There is a proposal dating from 2006 on the Haskell Wiki [39] to reform the standard along the lines we have chosen, but the situation is still open.

We will make significant use of the function *guard*, which checks that a boolean condition holds, and fails if it does not:

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

and, later on, a related function that takes a predicate:

```
assert :: MonadFail m ⇒ (a → Bool) → m a → m a
assert p mx = do {x ← mx; guard (p x); return x}
```

4.2 Choice

Choice is modelled by the class *MonadAlt*:

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

subject to the axiom that $\Box$ is associative, and composition distributes leftwards over it:

$$(m \Box n) \Box p = m \Box (n \Box p)$$
$$(m \Box n) \gg\!= k = (m \gg\!= k) \Box (n \gg\!= k)$$

Following [7], and in contrast to the 'additive monads' of Goncharov *et al.* [6], we do not always require composition to distribute *rightwards* over choice: in general,

$$m \gg\!= \lambda x \to k_1\ x \Box k_2\ x \neq (m \gg\!= k_1) \Box (m \gg\!= k_2)$$

As was the case above with *fail* on the right of a composition, this is a dubious proposition when m has non-idempotent effects, such as writing output: on the left-hand side above, the effects happen once, and on the right, twice. Sometimes, though, we do want this distributivity property; Section 7 is a case in point.

4.3 Nondeterminism

We model nondeterministic programs as those using the combination of failure and choice features:

```
class (MonadFail m, MonadAlt m) ⇒ MonadNondet m where
```

There are no additional operations; the body of this class definition is empty, and the operations of *MonadNondet* are just those of *MonadFail* (namely *fail*) together with those of *MonadAlt* (namely $\Box$). However, there are additional laws, beyond those of *MonadFail* and *MonadAlt* considered in isolation: *fail* should be a unit of $\Box$, giving a monoidal structure altogether.

The obvious model of the specification (indeed, the initial one) is given by lists:

```
instance Monad [] where
  return a  = [a]
  mx >>= k = concat (map k mx)

instance MonadFail [] where
  fail = []

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

However, sometimes we do not want to treat the ordering of elements as significant, so we might sometimes impose an additional axiom that $\Box$ is commutative, in which case finite bags form the initial model. And sometimes we do not want to treat multiplicity of results as significant either, so we add the axiom that $\Box$ is idempotent; then finite sets form the initial model.

Note that *MonadNondet* is rather like the Haskell standard's *MonadPlus* class [22]. However, as we shall see in Section 5, there is an alternative interpretation of the same monoidal structure—intuitively, as exceptions rather than nondeterminism. The signatures of the operations are the same, but the laws are different, so the two interpretations should most definitely not be confused with each other. (In particular, it makes no sense at all to suppose that the *catch* operator in exception handling is commutative!)

4.4 Permutations

Here is a simple example of a nondeterministic program. The function *select* takes a non-empty list and nondeterministically chooses an element, returning that element and the remaining list; it fails on the empty list.

```
select :: MonadNondet m ⇒ [a] → m (a, [a])
select []     = fail
select (x:xs) = return (x,xs) □
                do {(y,ys) ← select xs; return (y,x:ys)}
```

Using *select*, the function *perms* nondeterministically generates a permutation of a (finite) list.

```
perms :: MonadNondet m ⇒ [a] → m [a]
perms [] = return []
perms xs = do {(y,ys) ← select xs; zs ← perms ys; return (y:zs)}
```

We will exploit the laws of nondeterminism in Section 7 and 8.

5. Exceptional computations

Before we explore the ramifications of reasoning about nondeterminism monadically, it is worth pausing to consider again the importance of the laws associated with a class. There is another interpretation of the signature of the class *MonadNondet*, namely a monad with an imposed monoidal structure; but the additional laws are quite different, and hence so of course is the computational behaviour.

We have in mind the class of effect called *exceptions*. As with nondeterminism, there is a special constant *fail*, which is a left zero of composition. There is also a binary operator, which we will call *catch*, which forms a monoid in conjunction with *fail*. We capture the latter by means of a distinct subclass *MonadExcept* of *MonadFail*:

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

The idea is that *fail* raises an exception, and that *catch m h* executes body *m*, but passes control to the handler *h* if an exception is raised during the execution of *m*. This operational intuition is expressed axiomatically by the laws—that *catch* and *fail* form a monoid:

$$\begin{array}{ll}\mathit{catch}\ \mathit{fail}\ h & = h\\ \mathit{catch}\ m\ \mathit{fail} & = m\\ \mathit{catch}\ m\ (\mathit{catch}\ h\ h') & = \mathit{catch}\ (\mathit{catch}\ m\ h)\ h'\end{array}$$

that unexceptional bodies need no handler:

$$\mathit{catch}\ (\mathit{return}\ x)\ h = \mathit{return}\ x$$

and that exceptions are left-zeros of sequential composition:

$$\mathit{fail} \gg mx = \mathit{fail}$$

No other properties are expected; in particular, we expect that in general distributivity of composition over exception handlers will fail:

$$(\mathit{catch}\ m\ h) \gg\!= k \neq \mathit{catch}\ (m \gg\!= k)\ (h \gg\!= k)$$

—exceptions raised by *k* on the left-hand side may be erroneously handled on the right-hand side. Conversely, with nondeterminism there is no requirement that pure computations are left zeros of $\Box$.

5.1 Fast product

We can illustrate reasoning with exceptions via the 'fast product' algorithm, which multiplies the elements of a list of integers, but raises an exception if it finds a zero, subsequently catching the exception and returning zero for the overall product. We define

$$\begin{array}{l}\mathit{fastprod} :: \mathit{MonadExcept}\ m \Rightarrow [\mathit{Int}] \to m\ \mathit{Int}\\ \mathit{fastprod}\ xs = \mathit{catch}\ (\mathit{work}\ xs)\ (\mathit{return}\ 0)\end{array}$$

where *work* is specified by

$$\begin{array}{l}\mathit{work} :: \mathit{MonadFail}\ m \Rightarrow [\mathit{Int}] \to m\ \mathit{Int}\\ \mathit{work}\ xs = \textbf{if}\ 0 \in xs\ \textbf{then}\ \mathit{fail}\ \textbf{else}\ \mathit{return}\ (\mathit{product}\ xs)\end{array}$$

And we calculate:

$$\begin{array}{ll} & \mathit{fastprod}\ xs\\ = & \quad [\![\ \text{definition of } \mathit{fastprod}\ ]\!]\\ & \mathit{catch}\ (\mathit{work}\ xs)\ (\mathit{return}\ 0)\\ = & \quad [\![\ \text{specification of } \mathit{work}\ ]\!]\\ & \mathit{catch}\ (\textbf{if}\ 0 \in xs\ \textbf{then}\ \mathit{fail}\ \textbf{else}\ \mathit{return}\ (\mathit{product}\ xs))\ (\mathit{return}\ 0)\\ = & \quad [\![\ \text{lift out the conditional}\ ]\!]\\ & \textbf{if}\ 0 \in xs\ \textbf{then}\ \mathit{catch}\ \mathit{fail}\ (\mathit{return}\ 0)\\ & \qquad \textbf{else}\ \ \mathit{catch}\ (\mathit{return}\ (\mathit{product}\ xs))\ (\mathit{return}\ 0)\\ = & \quad [\![\ \text{laws of } \mathit{catch}, \mathit{fail}, \text{ and } \mathit{return}\ ]\!]\\ & \textbf{if}\ 0 \in xs\ \textbf{then}\ \mathit{return}\ 0\ \textbf{else}\ \mathit{return}\ (\mathit{product}\ xs)\\ = & \quad [\![\ \text{arithmetic: } 0 \in xs \Longrightarrow \mathit{product}\ xs = 0\ ]\!]\\ & \textbf{if}\ 0 \in xs\ \textbf{then}\ \mathit{return}\ (\mathit{product}\ xs)\ \textbf{else}\ \mathit{return}\ (\mathit{product}\ xs)\\ = & \quad [\![\ \text{redundant conditional}\ ]\!]\\ & \mathit{return}\ (\mathit{product}\ xs)\end{array}$$

Thus, *fastprod* is pure, never throwing an unhandled exception. Moreover, *work* may be refined separately, to eliminate the multiple traversals; one can use the universal property of *foldr* to derive

$$\begin{array}{l}\mathit{work} = \mathit{foldr}\ \mathit{next}\ (\mathit{return}\ 1)\ \textbf{where}\\ \quad \mathit{next}\ n\ mx = \textbf{if}\ n == 0\ \textbf{then}\ \mathit{fail}\ \textbf{else}\ \mathit{liftM}\ (n\times)\ mx\end{array}$$

The two steps of the calculation concerning conditionals depend only on pure reasoning: for non-bottom boolean *b*, we have:

$$\begin{array}{l}\textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ f\ y = f\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y)\\ \textbf{if}\ b\ \textbf{then}\ x\ \ \textbf{else}\ x\ \ = x\end{array}$$

6. Stateful computations

Perhaps the foremost class of effect that springs to mind is that of stateful computations. These are captured by the interface

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

Here, the class *MonadState s m* denotes monads *m* involving functions that transform a state of type *s*; the declaration includes a functional dependency, declaring that *m* determines *s*. The operation *get* yields a copy of the current state, and *put* overwrites the state with a new value (returning unit); they are related by four axioms:

$$\begin{array}{ll}\mathit{put}\ s \gg \mathit{put}\ s' & = \mathit{put}\ s'\\ \mathit{put}\ s \gg \mathit{get} & = \mathit{put}\ s \gg \mathit{return}\ s\\ \mathit{get} \gg\!= \mathit{put} & = \mathit{skip}\\ \mathit{get} \gg\!= \lambda s \to \mathit{get} \gg\!= k\ s & = \mathit{get} \gg\!= \lambda s \to k\ s\ s\end{array}$$

6.1 Eight queens

As an example of a stateful computation, we consider a few programs for the '*n* queens' problem. First, we start with a purely functional implementation, implicitly placing one queen in each column of the board, and explicitly assigning the rows of the queens from the permutation of the column numbers, so that by construction it is only the diagonals that are potential threats. The main data structure we manipulate is a pair of lists recording the up-diagonals and down-diagonals threatened by the queens to the right of a given column, where a queen in position (c,r) threatens up-diagonal $r-c$ and down-diagonal $r+c$.

For notational brevity, we define

$$\begin{array}{l}\textbf{type}\ \mathit{Square}\ a = (a,a)\\ \\ \mathit{square} :: (a \to b) \to (\mathit{Square}\ a \to \mathit{Square}\ b)\\ \mathit{square}\ f\ (a,b) = (f\ a, f\ b)\end{array}$$

and write 'a^2' and 'f^2' for '*Square a*' and '*square f*', respectively.

The essence of the program is the following function *test*, which takes a (column, row) position for a queen, and lists of already threatened up- and down-diagonals, and returns a pair consisting of a boolean for whether this queen position is safe, and updated lists of threatened up- and down-diagonals.

$$\begin{array}{l}\mathit{test} :: \mathit{Int}^2 \to [\mathit{Int}]^2 \to (\mathit{Bool}, [\mathit{Int}]^2)\\ \mathit{test}\ (c,r)\ (\mathit{ups}, \mathit{downs}) =\\ \quad (u \notin \mathit{ups} \wedge d \notin \mathit{downs}, (u : \mathit{ups}, d : \mathit{downs}))\\ \qquad \textbf{where}\ (u,d) = (r-c, r+c)\end{array}$$

Predicate safe_1 checks a putative placement of queens for safety, using a fold from right to left over the list of column-row pairs. The carrier $(\mathit{Bool}, [\mathit{Int}]^2)$ of the fold consists of a boolean indicating whether the queens to the right, which have already been checked, are safe, and the up- and down-diagonals under threat from the queens considered so far.

$$\begin{array}{l}\mathit{safe}_1 :: [\mathit{Int}]^2 \to [\mathit{Int}^2] \to (\mathit{Bool}, [\mathit{Int}]^2)\\ \mathit{safe}_1 = \mathit{foldr}\ \mathit{step}_1 \circ \mathit{start}_1\end{array}$$

$$start_1 :: [Int]^2 \to (Bool, [Int]^2)$$
$$start_1\ updowns = (True, updowns)$$

$$step_1 :: Int^2 \to (Bool, [Int]^2) \to (Bool, [Int]^2)$$
$$step_1\ cr\ (restOK, updowns) = (thisOK \wedge restOK, updowns')$$
$$\quad \textbf{where}\ (thisOK, updowns') = test\ cr\ updowns$$

To allow convenient expression below of the relationship between $safe_1$ and some later variants, we have factored out the initial value $([],[])$ for the threatened diagonals, and have not projected away the final diagonals; so the actual test for whether a list *crs* of column-row pairs is safe is $fst\ (safe_1\ ([],[])\ crs)$. In particular, the safe arrangements of n queens can be computed in a generate-and-test fashion, nondeterministically choosing row lists as permutations and discarding the unsafe ones:

$$queens :: MonadNondet\ m \Rightarrow Int \to m\ [Int]$$
$$queens\ n$$
$$\quad = \textbf{do}\ \{rs \leftarrow perms\ [1\,..\,n]\ ;$$
$$\qquad guard\ (fst\ (safe_1\ empty\ (place\ n\ rs)))\ ;\ return\ rs\}$$

$$place\ n\ rs = zip\ [1\,..\,n]\ rs$$
$$empty \quad = ([],[])$$

6.2 Queens, statefully

Rather than do the whole safety computation as a pure function, we now consider a version that uses a stateful computation to build up the sets of up- and down-diagonals threatened by the queens considered so far. The function returns a boolean, while statefully constructing the lists of threatened diagonals.

$$safe_2 :: MonadState\ [Int]^2\ m \Rightarrow [Int^2] \to m\ Bool$$
$$safe_2 = foldr\ step_2\ start_2$$

$$start_2 :: MonadState\ [Int]^2\ m \Rightarrow m\ Bool$$
$$start_2 = return\ True$$

$$step_2 :: MonadState\ [Int]^2\ m \Rightarrow Int^2 \to m\ Bool \to m\ Bool$$
$$step_2\ cr\ k$$
$$\quad = \textbf{do}\ \{b' \leftarrow k\ ;\ uds \leftarrow get\ ;\ \textbf{let}\ (b, uds') = test\ cr\ uds\ ;$$
$$\qquad put\ uds'\ ;\ return\ (b \wedge b')\}$$

In Figure 1, we prove using the axioms of *get* and *put* that

$$safe_2\ crs$$
$$\quad = \textbf{do}\ \{uds \leftarrow get\ ;\ \textbf{let}\ (ok, uds') = safe_1\ uds\ crs\ ;$$
$$\qquad put\ uds'\ ;\ return\ ok\}$$

by showing that this **do** expression satisfies the universal property of the fold in $safe_2$. This explains how the stateful computation $safe_2$ relates to the pure function $safe_1$. However, in order to revise the definition of *queens* to use $safe_2$ instead of $safe_1$, we need to reconcile the different computational settings: *perms* is nondeterministic, and $safe_2$ is stateful. We could flatten the stateful computation to a pure function: if we assume also the existence of a reification function

$$run :: MonadState\ s\ m \Rightarrow m\ a \to (s \to (a, s))$$

then we have

$$run\ (safe_2\ crs)\ updowns = safe_1\ updowns\ crs$$

and obtain a plug-in replacement for the use of $safe_1$ in *queens*. We will not do that; instead, we will lift the two components *perms* and $safe_2$ into one unified setting, providing both nondeterministic and stateful effects.

7. Combining effects

One distinct advantage of the axiomatic approach to reasoning with effects is that it is straightforward to take reasoning conducted in one setting and reuse it in another—that is, indeed, the general benefit of 'programming to an interface'.

In the case of the n-queens problem, we have a nondeterministic program generating candidate solutions (namely, permutations of the rows of the chessboard), and a stateful program testing those candidates (namely, the safety check acting on a pair of lists of diagonals). To that end, we name the combination of effects:

$$\textbf{class}\ (MonadState\ s\ m, MonadNondet\ m) \Rightarrow$$
$$\quad MonadStateNondet\ s\ m \mid m \to s$$

There are no additional operations; *MonadStateNondet*'s operations are just those of *MonadState* (namely, *get* and *put*) together with those of *MonadNondet* (namely, *fail* and $\Box$). But, as with *MonadNondet* itself earlier, we need to specify how the two classes of effect interact. In this case, we want *backtrackable state*, with nondeterministic behaviour taking priority over stateful behaviour—a failing computation should discard any accumulated stateful effects, and choice points should be explored from a common starting state. This is captured equationally by stipulating that failure is a right zero of sequential composition:

$$m \gg fail = fail$$

and that composition distributes rightwards over choice:

$$m \gg\!= \lambda x \to k_1\ x \,\Box\, k_2\ x = (m \gg\!= k_1) \,\Box\, (m \gg\!= k_2)$$

Recall that the *MonadNondet* class already requires *fail* to be a left zero of composition, and composition to distribute leftwards over choice. It would be unreasonable to require the additional two laws in general, because not all effects are backtrackable; but this is not an issue for state transformations. The alternative is to 'keep calm and carry on'—state persists over failure, and is threaded linearly through choice points—but this is not what we want for the n-queens problem. (Failure happens in selecting from the empty list when generating a permutation, and when discovering that a putative queen position is already threatened, and in both cases we should abandon that alternative and try an alternative. Choices take place between permutations, and each permutation should be checked starting from a clean slate.) The types $s \to [(a,s)]$ and $s \to ([a],s)$ are both models of stateful and nondeterministic computations, but only the first is a model of backtrackable state.

A crucial consequence of failure being a left and right zero of composition (and *skip* being a left and right unit) is that guards commute with anything:

$$guard\ b \gg m = m \gg\!= \lambda x \to guard\ b \gg return\ x$$

Indeed, the same applies for any expression involving only the effects of *MonadFail*: if *mx* has type $MonadFail\ m \Rightarrow m\ a$ and *my* has type $MonadStateNondet\ m \Rightarrow m\ b$, then

$$\quad \textbf{do}\ \{x \leftarrow mx\ ;\ y \leftarrow my\ ;\ return\ (x,y)\}$$
$$= \textbf{do}\ \{y \leftarrow my\ ;\ x \leftarrow mx\ ;\ return\ (x,y)\}$$

7.1 Queens, statefully and nondeterministically

Now, from the axiom of state that $get \gg\!= put = skip$, we have

$$queens\ n = \textbf{do}\ \{s \leftarrow get\ ;\ put\ s\ ;\ queens\ n\}$$

and from that we can calculate using basic properties of monads, together with the fact that *perms* is independent of the state and so commutes with *put*, that

$$queens\ n$$
$$\quad = \textbf{do}\ \{s \leftarrow get\ ;\ rs \leftarrow perms\ [1\,..\,n]\ ;\ put\ empty\ ;$$
$$\qquad ok \leftarrow safe_2\ (place\ n\ rs)\ ;\ put\ s\ ;\ guard\ ok\ ;\ return\ rs\}$$

This program is a little clumsy, but it does accurately capture the relationship with the non-stateful version of *queens*: when interpreted

For the empty list, we have:

$$
\begin{array}{ll}
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds') = safe_1\ uds\ [\,]\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{definition of } safe_1\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds') = start_1\ uds\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{definition of } start_1\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds') = (True,uds)\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{lifting the } \mathbf{let};\ get \gg\!= put = return\ ()\ ]\!] \\
 & return\ True \\
= & \quad [\![\ \text{definition of } safe_2\ ]\!] \\
 & safe_2\ [\,]
\end{array}
$$

while for non-empty lists, we have:

$$
\begin{array}{ll}
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds') = safe_1\ uds\ (cr:crs)\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{definition of } safe_1 \text{ as a } foldr\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds') = step_1\ cr\ (safe_1\ uds\ crs)\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{introduce a } \mathbf{let}\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(b',uds'') = safe_1\ uds\ crs\,;\mathbf{let}\,(ok,uds') = step_1\ cr\ (b',uds'')\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{definition of } step_1\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(b',uds'') = safe_1\ uds\ crs\,;\mathbf{let}\,(b,uds''') = test\ cr\ uds''\,;\mathbf{let}\,(ok,uds') = (b \wedge b',uds''')\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ put\ s \gg put\ s' = put\ s'\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(b',uds'') = safe_1\ uds\ crs\,;\mathbf{let}\,(b,uds''') = test\ cr\ uds''\,;\mathbf{let}\,(ok,uds') = (b \wedge b',uds''')\,;put\ uds''\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{move two of the } \mathbf{let}\text{s}\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(b',uds'') = safe_1\ uds\ crs\,;put\ uds''\,;\mathbf{let}\,(b,uds''') = test\ cr\ uds''\,;\mathbf{let}\,(ok,uds') = (b \wedge b',uds''')\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ put\ s \gg get \gg\!= k = put\ s \gg k\ s\ ]\!] \\
 & \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(b',uds'') = safe_1\ uds\ crs\,;put\ uds''\,; \\
 & \quad uds'''' \leftarrow get\,;\mathbf{let}\,(b,uds''') = test\ cr\ uds''''\,;\mathbf{let}\,(ok,uds') = (b \wedge b',uds''')\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{associativity of } \gg\!=\ ]\!] \\
 & \mathbf{do}\,\{b' \leftarrow \mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds'') = safe_1\ uds\ crs\,;put\ uds''\,;return\ ok\}\,; \\
 & \quad uds'''' \leftarrow get\,;\mathbf{let}\,(b,uds''') = test\ cr\ uds''''\,;\mathbf{let}\,(ok,uds') = (b \wedge b',uds''')\,;put\ uds'\,;return\ ok\} \\
= & \quad [\![\ \text{definition of } step_2\ ]\!] \\
 & step_2\ cr\ (\mathbf{do}\,\{uds \leftarrow get\,;\mathbf{let}\,(ok,uds'') = safe_1\ uds\ crs\,;put\ uds''\,;return\ ok\})
\end{array}
$$

Figure 1. Proving the relationship between $safe_2$ and $safe_1$

in a stateful setting, it amounts to remembering the current state s, executing a stateful version of *queens*, and restoring the original state s again.

7.2 Queens, exploratively

There is a more natural generate-and-test algorithm for the n-queens problem: rather than computing a boolean ok for whether an arrangement of queens is safe, and then asserting ok, we explore the possible threats to each queen in turn, and immediately abandon this arrangement if ever two queens conflict. How does this algorithm relate to the previous one? Very straightforwardly, as it turns out.

Since guards commute with anything, in the version of the *queens* program from Section 7.1, the *put s* and *guard ok* in particular commute, and we have:

$$
\begin{array}{l}
queens\ n \\
\quad = \mathbf{do}\,\{s \leftarrow get\,;rs \leftarrow perms\,[1\,..\,n]\,;put\ empty\,; \\
\qquad\quad ok \leftarrow safe_2\ (place\ n\ rs)\,;guard\ ok\,;put\ s\,;return\ rs\}
\end{array}
$$

Let us therefore define

$$safe_3\ crs = safe_2\ crs \gg\!= guard$$

so that

$$
\begin{array}{l}
queens\ n \\
\quad = \mathbf{do}\,\{s \leftarrow get\,;rs \leftarrow perms\,[1\,..\,n]\,;put\ empty\,; \\
\qquad\quad safe_3\ (place\ n\ rs)\,;put\ s\,;return\ rs\}
\end{array}
$$

Now, let us investigate $safe_3$. Recall that $safe_2$ is a fold. Moreover, $(\gg\!= guard)$ combines with that fold using the fusion property, as we now show. The initial value is simple:

$$start_2 \gg\!= guard = guard\ True = skip$$

For the inductive step, we have:

$$
\begin{array}{ll}
 & step_2\ cr\ k \gg\!= guard \\
= & \quad [\![\ \text{definition of } step_2\ ]\!] \\
 & \mathbf{do}\,\{b' \leftarrow k\,;uds \leftarrow get\,;\mathbf{let}\,(b,uds') = test\ cr\ uds\,; \\
 & \quad put\ uds'\,;return\ (b \wedge b')\,\} \gg\!= guard \\
= & \quad [\![\ \mathbf{do}\text{-notation}\ ]\!] \\
 & \mathbf{do}\,\{b' \leftarrow k\,;uds \leftarrow get\,;\mathbf{let}\,(b,uds') = test\ cr\ uds\,; \\
 & \quad put\ uds'\,;guard\ (b \wedge b')\,\} \\
= & \quad [\![\ guard \text{ distributes over conjunction}\ ]\!] \\
 & \mathbf{do}\,\{b' \leftarrow k\,;uds \leftarrow get\,;\mathbf{let}\,(b,uds') = test\ cr\ uds\,; \\
 & \quad put\ uds'\,;guard\ b\,;guard\ b'\,\} \\
= & \quad [\![\ guard\text{s commute with anything}\ ]\!] \\
 & \mathbf{do}\,\{b' \leftarrow k\,;guard\ b'\,;uds \leftarrow get\,;\mathbf{let}\,(b,uds') = test\ cr\ uds\,; \\
 & \quad put\ uds'\,;guard\ b\,\} \\
= & \quad [\![\ \text{associativity of } \gg\!=\ ]\!] \\
 & \mathbf{do}\,\{(k \gg\!= guard)\,;uds \leftarrow get\,;\mathbf{let}\,(b,uds') = test\ cr\ uds\,; \\
 & \quad put\ uds'\,;guard\ b\,\} \\
= & \quad [\![\ \text{definition of } step_3 \text{ (see below)}\ ]\!] \\
 & step_3\ cr\ (k \gg\!= guard)
\end{array}
$$

where we define

$step_3\ cr\ m = m \gg$
  do $\{uds \leftarrow get\,;$ **let** $(b, uds') = test\ cr\ uds\,;\ put\ uds'\,;\ guard\ b\}$

Therefore, by fold fusion, we have

$safe_3\ crs = foldr\ step_3\ skip$

We have derived this 'exploratory' version of the n-queens algorithm from the merely 'stateful and nondeterministic' one using plain old-fashioned equational reasoning. Of course, it is still a rather inefficient algorithm; we have simply used the problem as a vehicle for demonstrating the modularity achievable using the axiomatic approach to specifying effects.

8. Probabilistic computations

The observation that probability distributions form a monad is fairly well known; the first published description appears to date from 1981 [5], but (as with so many things) the credit seems to go back to an observation by Lawvere twenty years earlier [12]. Jones and Plotkin [11] use this theory to construct a powerdomain of probability distributions, allowing them to give a semantics to recursive programs with probabilistic features; Ramsey and Pfeffer [27] use it to define a probabilistic lambda calculus for finitely supported distributions; Erwig and Kollmansberger [3] describe a little monadic Haskell library for programming with finitely supported distributions.

We suppose a type *Prob* of probabilities (say, the rationals in the closed unit interval), and define a type class of finitely supported probability distributions by:

class $Monad\ m \Rightarrow MonadProb\ m$ **where**
  $choice :: Prob \to m\ a \to m\ a \to m\ a$

The idea is that $choice\ p\ mx\ my$ behaves as mx with probability p and as my with probability $1-p$. From now on, we will write '$\bar{p}$' for $1-p$, and following Hoare [8], write choice in infix notation, '$mx \lhd p \rhd my$', because this makes the laws more transparent. We have two identity laws:

$$mx \lhd 0 \rhd my = my$$
$$mx \lhd 1 \rhd my = mx$$

a skewed commutativity law:

$$mx \lhd p \rhd my = my \lhd \bar{p} \rhd mx$$

idempotence:

$$mx \lhd p \rhd mx = mx$$

and quasi-associativity:

$$mx \lhd p \rhd (my \lhd q \rhd mz) = (mx \lhd r \rhd my) \lhd s \rhd mz$$
$$\Longleftarrow p = r\,s \wedge \bar{s} = \bar{p}\,\bar{q}$$

(As informal justification for the associativity law, observe that the likelihoods of mx, my, mz on the left are $p, \bar{p}\,q, \bar{p}\,\bar{q}$, and on the right are $r\,s, \bar{r}\,s, \bar{s}$, and a little algebra shows that these are pairwise equal, given the premise.) Moreover, bind distributes leftwards and rightwards over choice:

$$(mx \lhd p \rhd my) \gg\!= k \quad = (mx \gg\!= k) \lhd p \rhd (my \gg\!= k)$$
$$mx \gg\!= \lambda x \to k_1\ x \lhd p \rhd k_2\ x = (mx \gg\!= k_1) \lhd p \rhd (mx \gg\!= k_2)$$

where, in the second law, x is assumed not to occur free in p.

For example, here is a function to generate a uniform distribution from a finite list of outcomes:

$uniform :: MonadProb\ m \Rightarrow [a] \to m\ a$
$uniform\ [x] \quad = return\ x$
$uniform\ (x : xs) = return\ x \lhd {}^{1}/_{length\ (x:xs)} \rhd uniform\ xs$

Note that $uniform\ xs$ is side-effect-free ($uniform\ xs \gg m = m$), because bind distributes leftwards over choice.

8.1 The Monty Hall Problem

As an example, consider the so-called Monty Hall Problem [28], which famously caused a controversy following its discussion in Marilyn vos Savant's column in *Parade* magazine in 1990 [35]. 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?*

Implicitly, 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, and it always differs from the one you pick and always reveals a goat, and you always get the option to switch.

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

data $Door = A \mid B \mid C$ **deriving** $(Eq, Show)$

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

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

$hide :: MonadProb\ m \Rightarrow m\ Door$
$hide = uniform\ doors$

Second, you pick one of the doors, also randomly:

$pick :: MonadProb\ m \Rightarrow m\ Door$
$pick = uniform\ doors$

Third, the host 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 remaining doors:

$tease :: MonadProb\ m \Rightarrow Door \to Door \to m\ Door$
$tease\ h\ p = uniform\ (doors \setminus\setminus [h, p])$

(Here, the expression $xs \setminus\setminus ys$ denotes the list of those elements of xs absent from ys.) Fourth, the host offers you the choice between two strategies—either to switch to the door that is neither your original choice nor the opened one:

$switch :: MonadProb\ m \Rightarrow Door \to Door \to m\ Door$
$switch\ p\ t = return\ (head\ (doors \setminus\setminus [p, t]))$

or to stick with your original choice:

$stick :: MonadProb\ m \Rightarrow Door \to Door \to m\ Door$
$stick\ p\ t = return\ p$

(In either case, you know p and t but not h.) Here's the whole game, parametrized by your strategy, returning whether you win the car:

$play :: MonadProb\ m \Rightarrow (Door \to Door \to m\ Door) \to m\ Bool$
$play\ strategy =$
  do
    $h \leftarrow hide$ -- host hides the car behind door h
    $p \leftarrow pick$ -- you pick door p
    $t \leftarrow tease\ h\ p$ -- host teases you with door $t\ (\neq h, p)$
    $s \leftarrow strategy\ p\ t$ -- you choose, based on p and t
    $return\ (s == h)$ -- you win iff your choice s equals h

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

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

The key is the fact that uniform choices are independent, in the sense that choosing consecutively from two uniform distributions is equivalent to choosing simultaneously from their cartesian product:

$$\mathit{pair}\ (\mathit{uniform}\ x, \mathit{uniform}\ y) = \mathit{uniform}\ (\mathit{cp}\ x\ y)$$

where

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

We omit the straightforward proof by induction.

Expanding definitions and exploiting independence of uniform choices, we have

$$\begin{aligned}&\mathit{play\ strategy}\\&\quad = \mathbf{do}\,\{\,(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, t \leftarrow \mathit{tease}\ h\ p\,;\\&\qquad\quad s \leftarrow \mathit{strategy}\ p\ t\,;\, \mathit{return}\ (s == h)\,\}\end{aligned}$$

So we calculate:

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

and

$$\begin{aligned}&\quad \mathit{play\ switch}\\&= \quad [\![\ \text{definition of } \mathit{play}\ ]\!]\\&\quad \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 [\![\ \mathit{switch}\ p\ t = \mathit{return}\ (\mathit{the}\ (\mathit{doors} \setminus\setminus [p,t]))\text{—see below}\ ]\!]\\&\quad \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{the}\ (\mathit{doors} \setminus\setminus [p,t]))\,;\, \mathit{return}\ (s == h)\,\}\\&= \quad [\![\ \mathit{return} \text{ is left unit}\ ]\!]\\&\quad \mathbf{do}\,\{\,(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\\&\qquad t \leftarrow \mathit{tease}\ h\ p\,;\, \mathit{return}\ (h == \mathit{the}\ (\mathit{doors} \setminus\setminus [p,t]))\,\}\\&= \quad [\![\ \text{case analysis on } h == p\text{—see below}\ ]\!]\\&\quad \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{lift out conditional; booleans}\ ]\!]\\&\quad \mathbf{do}\,\{\,(h,p) \leftarrow \mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors})\,;\, \mathit{return}\ (h \not== p)\,\}\\&= \quad [\![\ \text{definition of } \mathit{liftM}\ ]\!]\\&\quad \mathit{liftM}\ (\mathit{uncurry}\ (\not==))\ (\mathit{uniform}\ (\mathit{cp}\ \mathit{doors}\ \mathit{doors}))\\&= \quad [\![\ \text{naturality of } \mathit{uniform}\text{; definition of } \mathit{cp},\ == \ ]\!]\\&\quad \mathit{uniform}\,[\mathit{False},\mathit{True},\mathit{True},\mathit{True},\mathit{False},\mathit{True},\mathit{True},\mathit{True},\mathit{False}]\\&= \quad [\![\ \text{simplifying: three } \mathit{False}\text{s, six } \mathit{True}\text{s}\ ]\!]\\&\quad \mathit{uniform}\,[\mathit{False},\mathit{True},\mathit{True}]\end{aligned}$$

For the second step above, note that t is by construction distinct from p, and so $\mathit{doors} \setminus\setminus [p,t]$ is a singleton; we therefore introduce the function *the* such that $\mathit{the}\,[a] = a$.

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

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

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

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

This concludes our proof that vos Savant was right, and that the many mathematics PhDs who wrote in to *Parade* magazine chastizing her were—at best—thinking about a different problem.

8.2 Probability and nondeterminism

One might argue that a more accurate representation of the Monty Hall scenario allows the host a *nondeterministic* rather than probabilistic choice in hiding and teasing: Monty is in charge, and nobody says that he has to play fair.

The interaction of nondeterministic and probabilistic choice is notoriously tricky [40, 15], but it turns out to be mostly straightforward to give a model in terms of monads. We combine the two classes *MonadAlt* (interpreted commutatively and idempotently) and *MonadProb* of effects:

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

As before, there are no new operations. But there is an additional law, relating the two kinds of choice—probabilistic choice should distribute over nondeterministic:

$$m \lhd p \rhd (n_1 \,\square\, n_2) = (m \lhd p \rhd n_1) \,\square\, (m \lhd p \rhd n_2)$$

Intuitively, no freedom is lost from the program on the left by making the nondeterministic choice before the probabilistic one rather than afterwards.

As a consequence of this distribution property, the semantic model is roughly as sets of distributions, as for example with the probabilistic predicate transformer work of McIver and Morgan [15]. But not quite sets; in order to retain the idempotence of probabilistic choice

$$mx \lhd p \rhd mx = mx$$

we have to live with equivalence up to *convex closure* [33]—that is, if a computation may yield any two distributions d, d', then it may also yield their convex combination $r \times d + \bar{r} \times d'$ for any r with $0 \leqslant r \leqslant 1$. This is intuitively reasonable; if d, d' are possible outcomes of an experiment, then a sequence of experiments may yield any combination of ds and d's. The alternative solution, promoted by Varacca [34], is to abandon idempotence; this is unappealing to us, because it means that unused choices, such as Monty's *tease* in the face of the *stick* strategy, cannot be discarded.

As a simple example of a computation that mixes nondeterministic and probabilistic features, consider the basic operations of a fair coin toss [15]:

$$\begin{aligned}&\mathit{coin} :: \mathit{MonadProb}\ m \Rightarrow m\ \mathit{Bool}\\&\mathit{coin} = \mathit{return}\ \mathit{True} \lhd {}^1\!/_2 \rhd \mathit{return}\ \mathit{False}\end{aligned}$$

and an arbitrary boolean choice:

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

sequentially combined in either order:

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

These two differ; informally, in *coinarb* the nondeterministic choice can depend on the result of the coin toss, whereas in *arbcoin* it cannot—and of course, the fair probabilistic choice does not depend on the arbitrary nondeterministic choice either—otherwise it wouldn't be fair. As sets of distributions (here represented as weighted lists), *arbcoin* has two possible outcomes, both being a 50–50 distribution, so really only a single possible outcome:

$$arbcoin = \{[(True, {}^1/_2), (False, {}^1/_2)], [(False, {}^1/_2), (True, {}^1/_2)]\}$$

whereas *coinarb* offers four possible outcomes—a 50–50 distribution, in two different ways, or always *False* or always *True*:

$$coinarb = \{[(True, {}^1/_2), (False, {}^1/_2)],\ [(False, {}^1/_2), (True, {}^1/_2)],$$
$$[(False, {}^1/_2), (False, {}^1/_2)], [(True, {}^1/_2), (True, {}^1/_2)]\}$$

whose convex closure is in fact the set of all boolean distributions; that is, the nondeterministic choice in *arbcoin* provides no flexibility, but the one in *coinarb* can engineer any distribution whatsoever.

Returning to the Monty Hall problem, we could allow the host to make nondeterministic rather than probabilistic choices:

$$hide :: MonadAlt\ m \Rightarrow m\ Door$$
$$hide = arbitrary\ doors$$

$$tease :: MonadAlt\ m \Rightarrow Door \to Door \to m\ Door$$
$$tease\ h\ p = arbitrary\ (doors \setminus\setminus [h,p])$$

where

$$arbitrary :: MonadAlt\ m \Rightarrow [a] \to m\ a$$
$$arbitrary = foldr_1\ (\Box) \circ map\ return$$

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

$$\begin{aligned}
&pair\ (hide, pick)\\
=\ &[\![\ \text{let } k = \lambda h \to liftM\ (h,)\ pick\ ]\!]\\
&hide \gg\!= k\\
=\ &[\![\ \text{definition of } hide\ ]\!]\\
&(return\ A \Box return\ B \Box return\ C) \gg\!= k\\
=\ &[\![\ \text{distributivity}\ ]\!]\\
&(return\ A \gg\!= k) \Box (return\ B \gg\!= k) \Box (return\ C \gg\!= k)\\
=\ &[\![\ return \text{ is left unit}\ ]\!]\\
&k\ A \Box k\ B \Box k\ C\\
=\ &[\![\ \text{definition of } k\ ]\!]\\
&liftM\ (A,)\ pick \Box liftM\ (B,)\ pick \Box liftM\ (C,)\ pick
\end{aligned}$$

(where, for brevity, we have written $(A,)$ for the function $\lambda x \to (A,x)$). The remainder of the reasoning proceeds just as before, and the conclusion is still that the strategy *switch* wins two times in three, and *stick* only one time in three.

9. Tree relabelling

Finally, we turn our attention to the question that inspired our interest in reasoning about monadic programs in the first place. Hutton and Fulger [9] present a nice problem involving effectful computation with mutable state, concerning relabelling of trees. Given is a polymorphic datatype of trees,

$$\mathbf{data}\ Tree\ a = Tip\ a \mid Bin\ (Tree\ a)^2$$

(we continue to use the $(\text{-})^2$ shorthand for pairs from Section 6.1) and a type of symbols

$$\mathbf{newtype}\ Symbol = \ldots\ \mathbf{deriving}\ (Eq)$$

with which to relabel. The problem is to prove that for some suitable definition of a function *relabel* that takes trees $t :: Tree\ a$ to relabelled trees $u :: Tree\ Symbol$, we have $distinct\ (labels\ u)$ for each such u, where

$$labels :: Tree\ Symbol \to [Symbol]$$
$$labels\ (Tip\ a) = [a]$$
$$labels\ (Bin\ (t,u)) = labels\ t +\!\!+ labels\ u$$

$$distinct :: [Symbol] \to Bool$$
$$distinct\ [\,] \quad = True$$
$$distinct\ (l : ls) = l \notin ls \wedge distinct\ ls$$

For calculational convenience, in this section we will use uncurried versions of some familiar operators:

$$add :: Int^2 \to Int$$
$$add = uncurry\ (+)$$

$$cat :: [a]^2 \to [a]$$
$$cat = uncurry\ (+\!\!+)$$

$$disjoint :: Eq\ a \Rightarrow [a]^2 \to Bool$$
$$disjoint = null \circ uncurry\ intersect$$

We will also write simply '*M*' in place of *liftM* throughout Section 9, for brevity.

9.1 Fresh symbols

We'll assume a class of monads supporting fresh symbols:

$$\mathbf{class}\ Monad\ m \Rightarrow MonadFresh\ m\ \mathbf{where}$$
$$\quad fresh :: m\ Symbol$$

The operation *fresh* is not completely unconstrained; it has to generate symbols that are indeed fresh. We specify this by asserting that any sequence of fresh symbols will be distinct. That is, given an operation to generate a given number of fresh symbols:

$$symbols :: MonadFresh\ m \Rightarrow Int \to m\ [Symbol]$$
$$symbols\ n = sequence\ (replicate\ n\ fresh)$$

we require that

$$assert\ distinct \circ symbols = symbols$$

Note that we are again combining classes of effect, this time of *MonadFresh* and *MonadFail*: the two sides of the equation specifying *fresh* have the stronger type qualification $MonadFreshFail\ m$, where

$$\mathbf{class}\ (MonadFresh\ m, MonadFail\ m) \Rightarrow MonadFreshFail\ m$$

As with the combination of state and nondeterminism, to relate the two classes of effect we add the axiom that failure is a right zero of composition; that is, we again take a backtracking interpretation, and indeed, one might think of *MonadFresh* as modelling a kind of stateful computation (where the state is the store of fresh symbols).

As it turns out, the only property we will require of the predicate *distinct* is that it is *segment-closed*. Predicate p is segment-closed if

$$p\ (x +\!\!+ y) \Longrightarrow p\ x \wedge p\ y$$

The significance of this property is that $assert\ p$ can be promoted through list concatenation: there is another predicate q on pairs of symbol lists such that

$$assert\ p \circ M\ cat \circ pair = M\ cat \circ assert\ q \circ pair \circ (assert\ p)^2$$

In particular, when $p = distinct$, then $q = disjoint$ suffices: the concatenation of two lists is all distinct if both lists are all distinct, and they are also disjoint.

For the remainder of the section, we will use only p and q, not *distinct* and *disjoint* themselves. Abstracting out the relevant properties of the predicate is helpful, because it means that the reasoning to follow can be generalized to other problems. For example, we might suppose that the *Symbol* type is not just an equality type, but an enumeration, and we want to prove that tree relabelling yields a contiguous segment of the enumeration of all symbols—informally, that no fresh symbols are wasted. Then the same condition on *symbols* suffices, but where the predicate p is 'is a contiguous segment of the enumeration'—this is segment closed, with q being the predicate on pairs 'are adjacent segments':

$$q\ (xs, ys) = null\ xs \vee null\ ys \vee (succ\ (last\ xs) == head\ ys)$$

9.2 Tree relabelling

Tree relabelling is a tree fold, using *fresh* at each tip. Given a fold combinator

$$\begin{aligned} &foldt :: (a \to b) \to (b^2 \to b) \to Tree\ a \to b \\ &foldt\ f\ g\ (Tip\ a) \quad\ = f\ a \\ &foldt\ f\ g\ (Bin\ (t,u)) = g\ (foldt\ f\ g\ t, foldt\ f\ g\ u) \end{aligned}$$

then we define

$$\begin{aligned} &relabel :: MonadFresh\ m \Rightarrow Tree\ a \to m\ (Tree\ Symbol) \\ &relabel = foldt\ (M\ Tip \circ const\ fresh)\ (M\ Bin \circ pair) \end{aligned}$$

Extracting the distinct symbol list from a tree is another fold, but this time within *MonadFail*:

$$\begin{aligned} &dlabels :: MonadFail\ m \Rightarrow Tree\ Symbol \to m\ [Symbol] \\ &dlabels = foldt\ (return \circ wrap)\ (M\ cat \circ assert\ q \circ pair) \end{aligned}$$

where $wrap\ a = [a]$ makes a singleton list.

Our claim is that the composition of *dlabels* and *relabel* never fails—the symbol list satisfies the predicate, and we always get a proper symbol list of the appropriate length:

$$dlabels \bullet relabel = symbols \circ size$$

where $m \bullet n = join \circ M\ m \circ n$ is Kleisli composition, and

$$\begin{aligned} &size :: Tree\ a \to Int \\ &size = foldt\ (const\ 1)\ add \end{aligned}$$

We justify that claim in the next section.

9.3 Verifying tree relabelling

First, we show that relabelling a tree and extracting its distinct symbols fuse to a single fold. We have

$$\begin{aligned} &(dlabels \bullet relabel) \circ Tip \\ = &\quad [\![\text{ Kleisli composition }]\!] \\ &join \circ M\ dlabels \circ relabel \circ Tip \\ = &\quad [\![\text{ definition of } relabel \text{ as a fold }]\!] \\ &join \circ M\ dlabels \circ M\ Tip \circ const\ fresh \\ = &\quad [\![\text{ functors; definition of } dlabels \text{ as a fold }]\!] \\ &join \circ M\ (return \circ wrap) \circ const\ fresh \\ = &\quad [\![\text{ functors, monads }]\!] \\ &M\ wrap \circ const\ fresh \end{aligned}$$

and

$$\begin{aligned} &(dlabels \bullet relabel) \circ Bin \\ = &\quad [\![\text{ Kleisli composition }]\!] \\ &join \circ M\ dlabels \circ relabel \circ Bin \\ = &\quad [\![\text{ definition of } relabel \text{ as a fold }]\!] \\ &join \circ M\ dlabels \circ M\ Bin \circ pair \circ relabel^2 \\ = &\quad [\![\text{ functors; definition of } dlabels \text{ as a fold }]\!] \\ &join \circ M\ (M\ cat \circ assert\ q \circ pair \circ dlabels^2) \circ pair \circ relabel^2 \\ = &\quad [\![\text{ functors; naturality of } pair \text{ and } join]\!] \\ &M\ cat \circ join \circ M\ (assert\ q \circ pair) \circ pair \circ (M\ dlabels \circ relabel)^2 \\ = &\quad [\![\text{ commutativity of assertions—see below }]\!] \\ &M\ cat \circ assert\ q \circ join \circ M\ pair \circ pair \circ (M\ dlabels \circ relabel)^2 \\ = &\quad [\![\text{ join and pairs—see below }]\!] \\ &M\ cat \circ assert\ q \circ pair \circ (join \circ M\ dlabels \circ relabel)^2 \\ = &\quad [\![\text{ Kleisli composition }]\!] \\ &M\ cat \circ assert\ q \circ pair \circ (dlabels \bullet relabel)^2 \end{aligned}$$

and so, by the universal property of fold,

$$dlabels \bullet relabel = foldt\ drTip\ drBin$$

where

$$\begin{aligned} &drTip = M\ wrap \circ const\ fresh \\ &drBin = M\ cat \circ assert\ q \circ pair \end{aligned}$$

There are two non-trivial steps. The first is 'commutativity of assertions'

$$join \circ M\ (assert\ q) = assert\ q \circ join$$

which follows from guards commuting with anything. The second is the 'join and pairs' step

$$join \circ listM\ pair \circ pair = pair \circ join^2$$

This does not hold in general. It does hold for commutative monads; but our *MonadFreshFail* is not commutative. However, it also holds on pairs (mmx, mmy) in the special case that the inner computation in the mmx has effects only from *MonadFail*, because those effects commute with those of mmy. In our case, $mmx = M\ dlabels\ (relabel\ t)$, and indeed *dlabels* introduces only the possibility of failure, not any statefulness.

Now we show that $symbols \circ size$ is the same fold. We have

$$\begin{aligned} &symbols \circ size \circ Tip \\ = &\quad [\![\text{ definition of } size \text{ as a fold }]\!] \\ &symbols \circ const\ 1 \\ = &\quad [\![\text{ property of } symbols]\!] \\ &M\ wrap \circ const\ fresh \end{aligned}$$

and

$$\begin{aligned} &symbols \circ size \circ Bin \\ = &\quad [\![\text{ specification of } symbols]\!] \\ &assert\ p \circ symbols \circ size \circ Bin \\ = &\quad [\![\text{ definition of } size]\!] \\ &assert\ p \circ symbols \circ add \circ size^2 \\ = &\quad [\![\text{ properties of } symbols\text{; functors }]\!] \\ &assert\ p \circ M\ cat \circ pair \circ (symbols \circ size)^2 \\ = &\quad [\![\text{ } assert\ p \text{ distributes over concatenation }]\!] \\ &M\ cat \circ assert\ q \circ pair \circ (assert\ p \circ symbols \circ size)^2 \\ = &\quad [\![\text{ specification of } symbols]\!] \\ &M\ cat \circ assert\ q \circ pair \circ (symbols \circ size)^2 \end{aligned}$$

and so, again by the universal property of fold,

$$symbols \circ size = foldt\ drTip\ drBin$$

also. (The two 'properties of *symbols*' referred to are that

$$\begin{aligned} &symbols \circ const\ 1 = M\ wrap \circ const\ fresh \\ &symbols \circ add \quad\ = M\ cat \circ pair \circ symbols^2 \end{aligned}$$

which follow easily from the definition of *symbols* by simple equational reasoning.)

10. Conclusions

10.1 Related work

We were inspired to write this paper by reading Hutton and Fulger on reasoning about effects [9]. We learned about the tree relabelling problem from them, although the same problem is also used as Example 4.4 in the APPSEM 2000 lecture notes on monads and effects by Benton, Hughes and Moggi [1]. That particular problem admits a number of different approaches—not only in abstracting from the specifics of the class of computational effects, as we have done here, but also in abstracting from the pattern of control, which we discuss in a related paper [4].

Hoare Type Theory [17] is a technique for introducing Hoare triples into a dependently typed language, allowing pre- and postconditions to be tracked by the type checker. It is based around a type constructor $\{P\}\ x{:}A\ \{Q\}$ of Hoare triples, denoting computations that, when run in a heap satisfying precondition P, will return a result x of type A and an updated heap that together satisfy postcondition Q. It has been embedded into the Coq proof system as an axiomatic extension called Ynot [18], and used as the basis for an implementation of separation logic [19]. Similarly, Schröder and Mossakowski [29] develop a *monad-independent Hoare logic* within the HASCASL algebraic specification language, and use this logic for reasoning about dynamic references. Either of these approaches makes a sound basis for certified development of effectful programs in the Hoare–Floyd style.

Like we do, Swierstra [31] also starts with Hutton and Fulger's tree relabelling problem, and with the observation that it is unfortunate to have to expand the definitions of return and bind for a particular monad in order to reason about a program using that monad. He uses these as a springboard for taking a lightweight approach to Hoare Type Theory, based on the state monad but not involving a heap. Roughly speaking, rather than a single monolithic type $State\ s\ a = (s \to (a,s))$ of state transformers (for fixed state type s and varying return type a), Swierstra introduces a family of types *HoareState p a q* indexed by pre- and postconditions, such that the input must be a state satisfying the precondition p, and the output will a pair establishing the postcondition q. This allows him to conduct Hoare-Floyd-style reasoning for programs in the state monad, such as the tree relabelling problem; but the approach does not seem to be directly applicable to other monads.

Peyton Jones [21] discusses the 'awkward squad' of teletype I/O, concurrency, exceptions, and interfacing to foreign functions, all of which are effects wrapped up in the Haskell *IO* monad. He gives an operational semantics for the first three, but does not discuss reasoning about programs exploiting those classes of effect. Swierstra and Altenkirch [32] provide a simple functional implementation of teletype I/O, mutable state, and concurrency, using the free monad generated by the algebra of the operations supporting each class of effect—so in fact, taking an approach more closely aligned with algebraic theories, like we do. (Swierstra's doctoral thesis [30] extends this work to software transactional memory and distributed arrays too.)

10.2 Hoare-style reasoning

As we were developing the calculational approach to reasoning about effectful programs in this paper, we explored a technique owing more to Hoare triples than to algebraic specifications. In the end, we concluded that the Hoare-style technique was less convenient than the algebraic one exhibited in the rest of the paper, and (at least for all the examples we considered) unnecessary. Nevertheless, for completeness, we outline the technique here, and explain why we think it did not work so well.

The essence of the Hoare-style technique is the use of assertions, so for this section we work within *MonadFail*. For notational brevity, we will usually write '$p!$' for '*guard p*'. We'll also write '$mx\ \{p\}$' for the assertion that monadic computation mx establishes boolean postcondition p, a shorthand for

$$\mathbf{do}\ \{mx\,;p!\} = \mathbf{do}\ \{mx\}$$

in the simple case that mx returns unit. More generally, mx may return a meaningful result; more generally still, we might make an assertion $s_1\,;\ldots;s_n\ \{p\}$ about a sequence of qualifiers. If these n statements $stmts = s_1\,;\ldots;s_n$ contain m generators with patterns $u_1,\ldots,u_m$, the returns should be of the m-tuple $pats = (u_1,\ldots,u_m)$ of those patterns; so the assertion is a shorthand for the equality:

$$\mathbf{do}\ \{stmts\,;p!\,;return\ pats\} = \mathbf{do}\ \{stmts\,;return\ pats\}$$

(We assume for simplicity that no binding shadows any other, although this doesn't actually cause a problem. When there is just one generator, the return should be the value of that pattern; when there are no generators, the return should be the empty tuple. A similar construction is used by Erkök and Launchbury [2].)

Some monadic operations are particularly amenable to manipulation, because they are discardable, copyable, and relatively commutable; we call them *queries*. These characteristic properties of queries q,q' are expressed as follows:

$$\begin{aligned}
&\mathbf{do}\ \{_ \leftarrow q\,;return\ ()\} &&= return\ ()\\
&\mathbf{do}\ \{x \leftarrow q\,;y \leftarrow q\,;return\ (x,y)\} &&= \mathbf{do}\ \{x \leftarrow q\,;return\ (x,x)\}\\
&\mathbf{do}\ \{x \leftarrow q\,;y \leftarrow q'\,;return\ (x,y)\}\\
&\quad = \mathbf{do}\ \{y \leftarrow q'\,;x \leftarrow q\,;return\ (x,y)\}
\end{aligned}$$

For example, the *get* operation of the state monad is a query. But if we were to extend the *MonadCount* class in Section 3 with an operation *reset* to reset the counter, this would be copyable but not discardable; in a similar 'latch' monad, *set* and *reset* methods would be copyable but not relatively commutable; and the probabilistic operation *coin* in Section 8.2 is discardable but not copyable. (Schröder and Mossakowski [29] call queries 'pure'; however, in general, queries are not 'pure' in the sense of being instances of *return*.)

To illustrate the assertional style, recall the Towers of Hanoi problem from Section 3. We work here with an extended version of the *MonadCount* type class, offering also an operation to yield the current value of the counter:

$$\begin{aligned}
&\mathbf{class}\ Monad\ m \Rightarrow MonadCount\ m\ \mathbf{where}\\
&\quad tick\ ::m\ ()\\
&\quad total ::m\ Int
\end{aligned}$$

Now we no longer intend the free interpretation of this interface; an implementation is constrained by the facts that *total* is a query, and that *tick* increments the total:

$$m \leftarrow total\,;tick\,;n \leftarrow total\ \{n = m+1\}$$

The operation *total* is intended a 'ghost variable', used only for reasoning and not for execution. We can therefore consider the same program as before:

$$\begin{aligned}
&hanoi :: MonadCount\ m \Rightarrow Int \to m\ ()\\
&hanoi\ 0 &&= skip\\
&hanoi\ (n+1) &&= \mathbf{do}\ \{hanoi\ n\,;tick\,;hanoi\ n\}
\end{aligned}$$

The 'correctness property' of *hanoi* is that

$$t \leftarrow total\,;hanoi\ n\,;u \leftarrow total\ \{u-t == 2^n-1\}$$

The general approach to discharging such a proof obligation is to expand out definitions until the final assertion is evidently redundant and can be eliminated. The proof is by induction on n; the base case is straightforward:

$$\begin{aligned}
&\mathbf{do}\ \{t \leftarrow total\,;hanoi\ 0\,;u \leftarrow total\,;(u-t == 2^0-1)!\,;\\
&\quad return\ (t,u)\}
\end{aligned}$$

$$
\begin{array}{ll}
= & [\![\text{ definition of } \mathit{hanoi} \ ]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{skip}\,;u \leftarrow \mathit{total}\,;(u{-}t == 2^0{-}1)!\,; \\
& \quad \mathit{return}\ (t,u)\} \\
= & [\![\ \mathit{skip} \text{ is a unit of composition }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;u \leftarrow \mathit{total}\,;(u{-}t == 2^0{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\ \mathit{total} \text{ is a query }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;(t{-}t == 2^0{-}1)!\,;\mathit{return}\ (t,t)\} \\
= & [\![\text{ arithmetic; } \mathit{True}! \text{ is just } \mathit{skip} \ ]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{return}\ (t,t)\} \\
= & [\![\text{ reversing first three steps }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ 0\,;u \leftarrow \mathit{total}\,;\mathit{return}\ (t,u)\}
\end{array}
$$

For the inductive step, we assume the statement for n, and calculate:

$$
\begin{array}{ll}
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ (n+1)\,;u \leftarrow \mathit{total}\,; \\
& \quad (u{-}t == 2^{n+1}{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\text{ definition of } \mathit{hanoi} \ ]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ n\,;\mathit{tick}\,;\mathit{hanoi}\ n\,;u \leftarrow \mathit{total}\,; \\
& \quad (u{-}t == 2^{n+1}{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\text{ inserting some queries }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ n\,;u' \leftarrow \mathit{total}\,;\mathit{tick}\,;t' \leftarrow \mathit{total}\,; \\
& \quad \mathit{hanoi}\ n\,;u \leftarrow \mathit{total}\,;(u{-}t == 2^{n+1}{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\text{ inductive hypothesis; property of } \mathit{tick} \ ]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ n\,;u' \leftarrow \mathit{total}\,;(u'{-}t = 2^n{-}1)!\,; \\
& \quad \mathit{tick}\,;t' \leftarrow \mathit{total}\,;(t' = u'+1)!\,;\mathit{hanoi}\ n\,;u \leftarrow \mathit{total}\,; \\
& \quad (u{-}t' = 2^n{-}1)!\,;(u{-}t == 2^{n+1}{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\text{ arithmetic: final guard follows from the others }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ n\,;u' \leftarrow \mathit{total}\,;(u'{-}t = 2^n{-}1)!\,;\mathit{tick}\,; \\
& \quad t' \leftarrow \mathit{total}\,;(t' = u'+1)!\,;\mathit{hanoi}\ n\,;u \leftarrow \mathit{total}\,; \\
& \quad (u{-}t' = 2^n{-}1)!\,;\mathit{return}\ (t,u)\} \\
= & [\![\text{ reversing first two steps }]\!] \\
& \mathbf{do}\ \{t \leftarrow \mathit{total}\,;\mathit{hanoi}\ (n+1)\,;u \leftarrow \mathit{total}\,;\mathit{return}\ (t,u)\}
\end{array}
$$

This proof is significantly more verbose than the one in Section 3, because it phrased indirectly. Calculations tend to have a rightwards-pointing triangular ('▷') shape—definitions are expanded, the final assertion eliminated, and definitions contracted again—and the second half of each calculation is a mirror image of the first half.

Perhaps the reader feels that the Towers of Hanoi problem is too trivial to form the basis of any verdict. We felt so too, and although we could easily see the simpler solution for that problem, it took a long time to find the corresponding solution shown in Section 9 for the tree relabelling problem. As with the Towers of Hanoi, we extended the monad with a ghost operation:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFresh}\ m\ \mathbf{where} \\
\quad \mathit{fresh} :: m\ \mathit{Symbol} \\
\quad \mathit{used} :: m\ (\mathit{Set}\ \mathit{Symbol})
\end{array}
$$

so that *used* is a query, returning the set of symbols used so far, and *fresh* returns and uses up some fresh symbols:

$$x \leftarrow \mathit{used}\,;n \leftarrow \mathit{fresh}\,;y \leftarrow \mathit{used}\ \{x \subseteq y \wedge n \in y{-}x\}$$

The problem is then to prove that for some suitable definition of a function

$$\mathit{relabel} :: \mathit{MonadFresh}\ m \Rightarrow \mathit{Tree}\ a \to m\ (\mathit{Tree}\ \mathit{Symbol})$$

we have

$$u \leftarrow \mathit{relabel}\ t\ \{\mathit{distinct}\ u\}$$

for every t, where

$$
\begin{array}{l}
\mathit{distinct} :: \mathit{Tree}\ \mathit{Symbol} \to \mathit{Bool} \\
\mathit{distinct}\ (\mathit{Tip}\ a) \quad\ = \mathit{True} \\
\mathit{distinct}\ (\mathit{Bin}\ (t,u)) = \mathit{distinct}\ t \wedge \mathit{distinct}\ u \wedge (\mathit{labels}\ t \cap \mathit{labels}\ u == \emptyset)
\end{array}
$$

and

$$
\begin{array}{l}
\mathit{labels} :: \mathit{Tree}\ \mathit{Symbol} \to \mathit{Set}\ \mathit{Symbol} \\
\mathit{labels}\ (\mathit{Tip}\ a) \quad\ = \{a\} \\
\mathit{labels}\ (\mathit{Bin}\ (t,u)) = \mathit{labels}\ t \cup \mathit{labels}\ u
\end{array}
$$

As it happens, the correctness condition is too weak to support an inductive argument, and it has to be strengthened to

$$x \leftarrow \mathit{used}\,;t' \leftarrow \mathit{relabel}\ t\,;y \leftarrow \mathit{used}\ \{\mathit{distinct}\ t' \wedge x \subseteq y \wedge \mathit{labels}\ t' \subseteq y{-}x\}$$

The resulting calculation was rather longwinded, and we much prefer the proof presented in Section 9.

As well as the verbosity of the Hoare-style approach, it seems a little odd that we used only postconditions and not preconditions. We might define the Hoare triple $\{p\}\ m\ \{q\}$ as a shorthand for the identity

$$\mathbf{do}\ \{p!\,;x \leftarrow m\,;q!\,;\mathit{return}\ x\} = \mathbf{do}\ \{p!\,;m\}$$

But note that this is equivalent to the earlier notation for assertions, since

$$\{p\}\ m\ \{q\} = p!\,;m\ \{q\}$$

and in particular

$$\{\mathit{True}\}\ m\ \{q\} = m\ \{q\}$$

Others have come to the same conclusion; for example, Schröder and Mossakowski's monad-independent Hoare logic [29] similarly defines Hoare triples in terms of 'global evaluation formulae', which are analogous to our assertions.

10.3 Discussion

We have presented an approach to effectful functional programming that treats classes of effects as an abstract datatype, with an algebraic specification capturing its operations and equations. We didn't start out this way, but it turns out that the approach to which we have been led has less to do with monads as introduced into functional programming by Moggi [16] and Wadler [38], and more to do with so-called Lawvere theories [13].

Fundamentally, monads arise from adjunctions—classically, between algebraic constructions such as term algebras in one direction, and an underlying forgetful functor in the other direction. Lawvere theories arise more directly from equational theories of operations and their laws. Both were developed as categorical formulations of universal algebra [10], but Lawvere theories start off with the associated operations and their equational properties, whereas with monads these are a secondary notion. Indeed, the adjunction classically arising from the construction of the free model of an equational theory yields precisely the monad in Moggi's sense [24]; but the additional operations and equations to support a class of effects play little part in the story that starts with monads. Nevertheless, the monadic view makes a convenient interface for programming, and of course is embedded within the design of programming languages such as Haskell; so we do not wish to abandon it. (Power [10] points out that monads are in fact slightly more general than Lawvere theories: in particular, the monads for continuations and for partiality do not arise from operations and their equations in the same way as the other monads familiar to functional programmers.)

There is an intriguing duality in the Lawvere theory approach between algebraic operations and *effect handlers* [25]. For example, the *catch* operation of *MonadExcept* is an effect handler in this sense; technically it doesn't form an 'operation' of a Lawvere theory, because it doesn't have the right distributivity properties with respect to $\gg\!=$. Operations and handlers can be seen as 'effect constructors' and 'effect deconstructors', respectively; and the handlers can be defined as folds over the initial algebra induced by the

free model of the algebraic operations [26]. Given the amenability of folds to equational reasoning, we feel that this development has consequences worthy of further investigation.

Acknowledgements

We gratefully acknowledge the many helpful comments from members of the *Algebra of Programming* research group at Oxford, especially Richard Bird, and from participants of the *IFIP WG 2.8* meeting in Marble Falls and the *European Workshop on Computational Effects* in Ljubljana, all of which have improved the presentation of this paper. This work was partially supported by UK Engineering and Physical Sciences Research Council grant EP/G034516/1 on *Reusability and Dependent Types*.

References

[1] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM 2000*, volume 2395 of *LNCS*, pages 42–122. Springer, 2002.

[2] L. Erkök and J. Launchbury. Recursive monadic bindings. In *ICFP*, pages 174–185. ACM, September 2000.

[3] M. Erwig and S. Kollmansberger. Probabilistic functional programming in Haskell. *J. Funct. Prog.*, 16(1):21–34, 2006.

[4] J. Gibbons and R. Bird. Effective reasoning about effectful traversals. Work in progress, Mar. 2011.

[5] M. Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *LNM*, pages 68–85. Springer, 1981.

[6] S. Goncharov, L. Schröder, and T. Mossakowski. Kleene monads: Handling iteration in a framework of generic effects. In *CALCO*, volume 5728 of *LNCS*, pages 18–33, 2009.

[7] R. Hinze. Prolog's control constructs in a functional setting: Axioms and implementation. *Intern. J. Found. Comput. Sci.*, 12(2):125–170, 2001.

[8] C. A. R. Hoare. A couple of novelties in the propositional calculus. *Z. Math. Logik Grundlag. Math.*, 31(2):173–178, 1985.

[9] G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *TFP*, May 2008.

[10] M. Hyland and J. Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electron. Notes Theoret. Comput. Sci.*, 172:437–458, 2007.

[11] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, pages 186–195, 1989.

[12] F. W. Lawvere. The category of probabilistic mappings. Preprint—we haven't managed to obtain a copy of this manuscript, 1962.

[13] F. W. Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963. Also available with commentary as *Theory and Applications of Categories* Reprint 5.

[14] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[15] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer Verlag, 2005.

[16] E. Moggi. Notions of computation and monads. *Inform. & Comput.*, 93(1), 1991.

[17] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, polymorphism and separation. *J. Funct. Prog.*, 18(5,6):865–911, 2008.

[18] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.

[19] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[20] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly, 2008.

[21] S. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series*, pages 47–96. IOS Press, 2001.

[22] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[23] S. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.

[24] G. Plotkin and J. Power. Notions of computation determine monads. In *FOSSACS*, volume 2303 of *LNCS*, pages 342–356, 2002.

[25] G. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Cat. Struct.*, 11(1):69–94, 2003.

[26] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*, volume 5502 of *LNCS*, pages 80–94, 2009.

[27] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

[28] J. Rosenhouse. *The Monty Hall Problem: The Remarkable Story of Math's Most Contentious Brain Teaser*. Oxford University Press, 2009.

[29] L. Schröder and T. Mossakowski. HasCasl: Integrated higher-order specification and program development. *Theoretical Comput. Sci.*, 410(12-13):1217–1260, 2009.

[30] W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.

[31] W. Swierstra. A Hoare logic for the state monad. In *TPHOLs*, volume 5674 of *LNCS*, pages 440–451. Springer-Verlag, 2009.

[32] W. Swierstra and T. Altenkirch. Beauty in the beast. In *Haskell Workshop*, pages 25–36, 2007.

[33] R. Tix. *Continuous D-Cones: Convexity and Powerdomain Constructions*. PhD thesis, Technische Universität Darmstadt, 1999.

[34] D. Varacca and G. Winskel. Distributing probability over non-determinism. *Math. Struct. Comput. Sci.*, 16(1):87–113, 2006.

[35] M. Vos Savant. Ask Marilyn. *Parade Magazine*, 9th September 1990. See also `http://www.marilynvossavant.com/articles/gameshow.html`.

[36] P. Wadler. A critique of Abelson and Sussman: Why calculating is better than scheming. *SIGPLAN Not.*, 22(3):8, 1987.

[37] P. Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992.

[38] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992.

[39] A. Yakeley, *et al.* MonadPlus reform proposal. `http://www.haskell.org/haskellwiki/MonadPlus_reform_proposal`, Jan. 2006.

[40] W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Protocol Specification, Testing and Verification XII*, pages 47–61, 1992.