Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARL

Kleisli arrows of outrageous fortune

CONOR McBRIDE
University of Strathclyde

Abstract

When we program to interact with a turbulent world, we are to some extent at its mercy. To achieve safety, we must ensure that programs act in accordance with what is known about the state of the world, as determined dynamically. Is there any hope to enforce safety policies for dynamic interaction by static typing? This paper answers with a cautious 'yes'.

Monads provide a type discipline for effectful programming, mapping value types to computation types. If we index our types by data approximating the 'state of the world', we refine our values to *witnesses* for some condition of the world. Ordinary monads for indexed types give a discipline for effectful programming contingent on state, modelling the whims of fortune in way that Atkey's indexed monads for ordinary types do not (Atkey, 2009). Arrows in the corresponding Kleisli category represent computations which a reach a given postcondition from a given precondition: their types are just specifications in a Hoare logic!

By way of an elementary introduction to this approach, I present the example of a monad for interacting with a file handle which is either 'open' or 'closed', constructed from a command interface specfied Hoare-style. An attempt to open a file results in a state which is statically unpredictable but dynamically detectable. Well typed programs behave accordingly in either case. Haskell's dependent type system, as exposed by the *Strathclyde Haskell Enhancement* preprocessor, provides a suitable basis for this simple experiment.

1 Prologue

The following C program is, alas, poor. Can you spot the problem?

```
void readFile() {
  FILE *b;
  char c;

  b = fopen("Yorick", "r");
  c = fgetc(b);
  while (!feof(b)) { putchar(c); c = fgetc(b); }
  fclose(b); }
```

If the file system knows `Yorick` then all is well, but if the file cannot be found, we attempt to read it, regardless. The program neglects to test if `fopen` returns a valid handle: `b` or not `b`, that is the question. This article considers whether 'tis nobler in the mind to suffer the slings and arrows of outrageous fortune, or to take arms against a C of troubles,

and by opposing end them. The following code, written in an obscure dialect of Haskell shortly to be elucidated, typechecks...

```
fileContents :: FilePath → (FH :* (Maybe String :≔{ Closed })) { Closed }
fileContents p = fOpen p ?>= λb → case b of
  { Closed } → (| Nothing |)
  { Open }   → (| Just readOpenFile (−fClose−) |)
```

...but the variant which bypasses the check for the outcome of fOpen does not!

```
fileContents :: FilePath → (FH :* (Maybe String :≔{ Closed })) { Closed }    -- error!
fileContents p = fOpen p ?>= λb → (| Just readOpenFile (−fClose−) |)
```

Ordinarily, it is typesafe (albeit foolish) to replace a **case**-expression by one of its alternative outcomes. Well typed programs may not go 'wrong', but they sometimes act inappropriately, given their circumstances. Ay, there's the rub—the *type* is unaware of the circumstances! The motivation for making a case distinction does not conventionally manifest itself in the types of the alternatives. *Dependent* types can, however, model notions of computation relative to circumstances; dependent case analysis allows distinctly typed alternatives to exploit distinctly specialised knowledge of those circumstances.

In this instance, I have a notion of computing with a file handle resource which models whether it is Open or Closed: the result of fOpen is a token which witnesses either one state or the other, and must be inspected before actions appropriate only to Open files are performed. To achieve this, I make use of my preprocessor—the Strathclyde Haskell Enhancement (SHE)—to desugar several new kinds of bracket, and to simulate dependent types, with type-level data witnessed by value-level singletons. I apply the standard notion of Monad to indexed data and show why the resulting abstraction suits programming in uncertain circumstances. I find in the associated notion of 'Kleisli arrow' an old friend, suggesting a standard discipline for specifying and programming within systems of state-dependent interaction. I illustrate the latter with the file handling example.

2 The braces of upward mobility

Recent versions of the Glasgow Haskell Compiler (GHC) admit datatype declarations in the style of a signature. We may declare a new type constructor, giving its *kind*. Just as the expression language has a system of types, so the type language has a system of 'kinds', with $*$ the kind of types[1]. We may then declare the associated value constructors, giving each its type. A simple type, such as Peano's natural numbers, may be given as follows.

```
data Nat :: * where
  Z ::          Nat
  S :: Nat → Nat
```

For such types, we must expect the targets of the value constructors to be uniform. However, GHC now supports the definition of type constructors at higher kinds: kinds are closed

[1] I am careful to use the word 'type' to mean only those type-level forms which can have expression-level inhabitants.

under $\rightarrow$, just as types are. The type constructor for lists, $[\,]$, has kind $* \rightarrow *$, we sometimes work with type constructor transformers of kind $(* \rightarrow *) \rightarrow (* \rightarrow *)$, and so on. Haskell offers support for abstraction at kinds other than $*$ in the *type* system rather than in the *module* system (as in current dialects of ML), and in doing so gains an effective expressive advantage.

Moreover, where type constructors take arguments, we may now declare value constructors which instantiate those arguments non-uniformly, both in recursive usages (as already supported in 'nested' types), and in the result usage. Modelled on the 'inductive families' of Martin-Löf type theories, these *generalized algebraic*[2] *datatypes* (GADTs) allow values to be constrained by and act as witnesses to type level phenomena in ways which sustain more informative testing.

SHE purports to go further, promoting every type $a :: *$ to a kind $\{a\}$, allowing types constructors to take arguments which resemble *values*. A similar syntactic convention, with 'braces of upward mobility' lifts every value constructor C to a type level constructor $\{\mathsf{C}\}$ with the correspondingly lifted argument and result kinds. We may now declare typical examples from the literature of dependently typed programming, such as the *vectors*—lists indexed by length.

$$\begin{array}{l}\textbf{data}\ \mathsf{Vec} :: * \rightarrow \{\mathsf{Nat}\} \rightarrow * \ \textbf{where}\\ \quad \mathsf{Nil} \ :: \qquad\qquad\quad \mathsf{Vec}\ a\ \{\mathsf{Z}\}\\ \quad \mathsf{Cons} :: a \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ a\ \{\mathsf{S}\ n\}\end{array}$$

In fact, SHE permits *constructor forms* in type-level braces, with the meaning that just the constructors within are lifted from the value level, leaving the variables as they stand. It serves a mnemonic purpose to write braces around type-level expressions whose kinds are also lifted, but the type of Cons can be expressed equivalently as

$$\mathsf{Cons} :: a \rightarrow \mathsf{Vec}\ a\ n \rightarrow \mathsf{Vec}\ a\ (\{\mathsf{S}\}\ n)$$

decorating only the S to be shifted between Haskell's distinct expression- and type- level namespaces. Whilst the latter variant reflects the fact that S is a first-class type-level constructor of kind $\{\mathsf{Nat}\} \rightarrow \{\mathsf{Nat}\}$, it seems somehow more awkward.

We may now specify length-related information in the types of vector operations, for example, the fact that 'map' is length-respecting.

$$\begin{array}{l}\mathsf{vmap} :: (a \rightarrow b) \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ b\ \{n\}\\ \mathsf{vmap}\ f\ \mathsf{Nil} \qquad\quad = \mathsf{Nil}\\ \mathsf{vmap}\ f\ (\mathsf{Cons}\ a\ as) = \mathsf{Cons}\ (f\ a)\ (\mathsf{vmap}\ f\ as)\end{array}$$

3 Indexed types, functions, and functors

Index-respecting functions will prove to be a significant and useful concept, so let us define some notation for them.

$$\textbf{type}\ s :\!\rightarrow t = \forall\, i\,.\, s\ \{i\} \rightarrow t\ \{i\}$$

[2] not *abstract*

So we have:

$$\mathsf{vmap} :: (a \to b) \to \mathsf{Vec}\ a :\!\!\to \mathsf{Vec}\ b$$

Digression—polymorphic kinds. The kind of $:\!\!\to$ is polymorphic—it works for any *type* of index.

$$(:\!\!\to) :: \forall\, \{a :: *\}\, .\, (\{a\} \to *) \to (\{a\} \to *) \to *$$

SHE does not introduce full polymorphism to the kind level. Kinds may only be polymorphic, as types are, in the inhabitants of a given kind, not in the choice of a kind. A happy consequence is that the variables bound by $\forall$ in a polymorphic kind may be used only within $\{..\}$, and hence the translation to standard Haskell kinds can simply erase the quantifiers and replace $\{..\}$ by $*$. **End of digression.**

We may readily check that $:\!\!\to$ is equipped with suitable notions of 'identity' and 'composition'. Indeed, the usual functional notions serve perfectly well. GHC admits that $\mathsf{id} :: t :\!\!\to t$, and that if $f :: s :\!\!\to t$ and $g :: r :\!\!\to s$ then $f\ .\ g :: r :\!\!\to t$. In other words, for each type a, we have a *category* whose objects are $\{a\}$-indexed type-formers $s, t :: \{a\} \to *$, and whose morphisms are index-respecting functions in $s :\!\!\to t$.

Correspondingly, we can now see that Vec is a functor in the categorical sense: it takes an 'element type' in $*$ to its indexed vector type-former in $\{\mathsf{Nat}\} \to *$; the associated vmap operator takes plain $*$-morphisms (functions in some $a \to b$) to $(\{\mathsf{Nat}\} \to *)$-morphisms in $\mathsf{Vec}\ a :\!\!\to \mathsf{Vec}\ b$, preserving identity and composition. Vec is not an instance of Haskell's Functor class, which concerns just those functors from $*$ to itself—the *endo*functors on $*$.[3]

We may consider, more generally, what it is to be a functor from one kind of indexed set to another. There is no need to presuppose that the input index type coincides with the output index type, only to ensure that respect for the former yields respect for the latter.

```
class IFunctor (f :: ({i} -> *) -> {o} -> *) where
  imap :: (s :-> t) -> f s :-> f t
    -- such that imap id = id and imap f . imap g = imap (f . g)
```

Note that the type signature is very much like that for *fmap*, but with $:\!\!\to$ replacing $\to$ in the types of the input and output morphisms. In our exploration of indexed programming, we shall often find famililar apparatus shifted in exactly this way. The genius of category theory is that it fosters intuitions which readily generalize.

One example of an IFunctor is the type of paths in a directed graph g on a type i, indexed by their initial and final vertex. SHE permits the elision of tuple-forming $(..)$ immediately within $\{..\}$, so we may write the following, expressing the idea that a path is a sequence of g-edges which join up domino-style.

```
data Path :: ({i,i} -> *) -> {i,i} -> * where
  Stop ::                          Path g {i,i}
  (:-:) :: g {i,j} -> Path g {j,k} -> Path g {i,k}
```

[3] We should not make the mistake of imagining that there is but one category at work in Haskell programming, and we should not foster this mistake by calling the category of types and functions **Hask**, as if it were uniquely 'the Haskell category'.

The IFunctor instance for Path captures the idea that if you can transform edges in a vertex-respecting way, then you can certainly transform whole paths, and their components will still join up properly.

```
instance IFunctor Path where
  imap f Stop       = Stop
  imap f (r :-: rs) = f r :-: imap f rs
```

The program is essentially that which we know for lists, but we work at a higher level of precision. Lists are effectively the graphs on *one* vertex. To see this, we shall benefit from a particularly useful constructor of indexed sets, capturing that idea of having an element of a given type at a particular *key* index.

```
data (:=:) :: forall (x :: *) . * -> {x} -> {x} -> * where
  V :: a -> (a :=: {k}) {k}
```

The $:=:$operator is pronounced 'at key'. $a :=: \{k\}$ is an indexed type-former which packs up an a, but only at the key index k—for other indices, the type is uninhabited. By extension,[4] an index-respecting map from $(a :=: \{k\})$ need only be defined at the key index:

$$(a :=: \{k\}) :-> t \quad \cong \quad a \to t\,\{k\}$$

We can thus describe a graph comprising a-labelled edges from one specific vertex i to another j by $a :=: \{i,j\}$. A list is a path in such a graph.

```
type List a = Path (a :=: {(), ()}) {(), ()}
```

4 Kleisli triples, Hoare triples

Having identified a suitable notion of functor between categories of indexed sets, we are at liberty to restrict our attention to *endo*functors, indexing elements and their superstructures with the same type, and ask which of them are *monads*. Let us ask Mac Lane (1998), for the concept of monad is quite independent of the category over which we work. I choose to give the *Kleisli Triple*[5] presentation, as it is more familiar to functional programmers.

```
class IFunctor m => IMonad (m :: ({i} -> *) -> {i} -> *) where
  iskip   :: p :-> m p
  iextend :: (p :-> m q) -> (m p :-> m q)
```

To explain what is going on, it may help to borrow some language from the other side of the Curry-Howard correspondence. Let us think of $p :: \{i\} \to *$ as a *predicates* on $\{i\}$, where the index set i represents some part of the 'state of the world', for example, whether a file handle is open or closed. A datum $v :: p\,\{i\}$ is a witness that p holds at state i.

A monad m is a predicate transformer which expresses a notion of *reachability*: $m\,p\,\{i\}$ asserts that from state i, we can reach some state satisfying p. The iskip operation tells us that if p holds already, then we may reach a state where it holds by doing nothing; the

[4] Category theorists can be left to fill in their own joke.
[5] not to be confused with a *tribble*, which is a warm fuzzy thing

iextend operation explains that if q is reachable from states satisfying p, then q is reachable from 'here' if p is.

As ever, such a monad induces a Kleisli category whose arrows

$$f :: p :\rightarrowtail m\ q$$

aver that a postcondition q is reachable from a precondition p. A Kleisli arrow is a *Hoare triple* (Hoare, 1969), which is why I dubbed the 'do nothing' operator iskip. Composition of Kleisli arrows corresponds exactly to Hoare's 'semicolon' for sequential composition.

$$\begin{aligned}&\mathsf{iseq} :: \mathsf{IMonad}\ m \Rightarrow (p :\rightarrowtail m\ q) \rightarrow (q :\rightarrowtail m\ r) \rightarrow p :\rightarrowtail m\ r\\ &\mathsf{iseq}\ f\ g = \mathsf{iextend}\ g\ .f\end{aligned}$$

These Kleisli arrows, as promised, express computations in a world of *outrageous fortune*, where circumstances may readily prove beyond our control. To see how, let us reconstruct the usual monadic 'bind'. We shall need it, in any case.

5 Angels, Demons and Bob

As with the $\gg\!=$ operator for monads on $*$, the 'bind' operator just flips the arguments to the Kleisli extension, untidily splitting the $m\ p :\rightarrowtail m\ q$, but allowing us to put first things first, then explain how to continue. Starting in state i, we first reach p, then, given a witness to p, we carry on to q.

$$\begin{aligned}&(?\!>\!=) :: \mathsf{IMonad}\ m \Rightarrow m\ p\ \{i\} \rightarrow (p :\rightarrowtail m\ q) \rightarrow m\ q\ \{i\}\\ &c\ ?\!>\!=\!f = \mathsf{iextend}\ f\ c\end{aligned}$$

It is informative to unpack the other $:\rightarrowtail$ and observe the pattern of quantification on states.

$$(?\!>\!=) :: \mathsf{IMonad}\ m \Rightarrow \forall\, i\,.\, m\ p\ \{i\} \rightarrow (\forall\, j\,.\, p\ \{j\} \rightarrow m\ q\ \{j\}) \rightarrow m\ q\ \{i\}$$

The initial state is governed by the outer $\forall$, which is *angelic*, letting us instantiate i as we please — typically to the state we happen to be in. However, the quantifier for j, the 'middle' state where p holds, whence we must reach q, is of the opposite, *demonic* polarity. The world chooses j with as much malice as may be mustered, given that p is to be satisfied. I call $?\!>\!=$ the 'demonic bind', with the question mark symbolising our imperfect knowledge of the state, our subjection to the whim of outrageous fortune.

We can fight back against the demon, with help from our friend $\coloneqq$, making p a predicate which specifies exactly the state j we demand. Here, then, is the 'angelic bind':

$$\begin{aligned}&(=\!\!>\!=) :: \mathsf{IMonad}\ m \Rightarrow m\ (a \coloneqq \{j\})\ \{i\} \rightarrow (a \rightarrow m\ q\ \{j\}) \rightarrow m\ q\ \{i\}\\ &c =\!\!>\!= f = c\ ?\!>\!= \lambda(\mathsf{V}\ a) \rightarrow f\ a\end{aligned}$$

By choosing the predicate $a \coloneqq \{j\}$, we ensure that the first computation finishes in state j, yielding a value in a. Thenceforth, we are once again at the mercy of the world in our quest to reach q.

We can now estalish a connection with the 'parametrized' or 'indexed' monads studied by Bob Atkey (2009), also considered under various names by Phil Wadler and Peter Thiemann (2003), Tarmo Uustalu (2003), Oleg Kiselyov and Chung-Chieh Shan (2008), and Jean-Christophe Filliâtre (1999). These extend the usual monad structure with indices standing for initial and final states in a computation.

$m :: \{i\} \to \{i\} \to * \to *$
$\mathsf{return} :: x \to m\ \{i\}\ \{i\}\ x$
$(\gg\!=) :: m\ \{i\}\ \{j\}\ a \to (a \to m\ \{j\}\ \{k\}\ b) \to m\ \{i\}\ \{k\}\ b$

It is useful to implement this signature for every IMonad, just by specializing to predicates formed by 'at key', representing a transition from state i to state j yielding a value in a:

$\mathbf{type}\ \mathsf{Atkey}\ m\ i\ j\ a = m\ (a :=\!\{j\})\ \{i\}$

We may implement a suitable 'return'

$\mathsf{ireturn} :: \mathsf{IMonad}\ m \Rightarrow a \to \mathsf{Atkey}\ m\ \{i\}\ \{i\}\ a$
$\mathsf{ireturn}\ a = \mathsf{iskip}\ (\mathsf{V}\ a)$

and note that $\Rightarrow\!\!=$, the angelic bind, is already the bind we need. An ordinary monad on indexed types induces an indexed monad on ordinary types, packaging the restricted functionality offered by the angelic bind. Operations which have an unpredictable impact on the state of the world, e.g. trying to open a file, cannot be expressed as Kleisli arrows in an indexed monad, for these must prescribe a target state. One can, of course, resort to a branching control operator with a separate continuation for each possible outcome. By allowing the free expression of pre- and postconditions, monads on indexed types can reflect that demonic choice directly as *data*.

6 From Hoare Logic Specifications to Free Monads

Let us focus on our file-handling problem. Consider a file handle as a resource with which we can interact. It will always be in one of two two possible states, given as follows.

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton
```

SHE detects the SheSingleton request and constructs the singleton GADT for the State type, which acts as if defined as follows:

```
data (::State) :: {State} -> * where
  {Open}   :: (::State) {Open}
  {Closed} :: (::State) {Closed}
```

Seen as a predicate on *static* states of kind $\{\mathsf{State}\}$, $(::\mathsf{State})\ \{i\}$ reifies the typing judgment at the value level, meaning 'i is a State known at run time'. Case analysis on an inhabitant of that type determines whether i is $\{\mathsf{Open}\}$ or $\{\mathsf{Closed}\}$. This construction allows us to promise dynamic knowledge in a postcondition, without requiring static knowledge.

Singletons and $:=$provide the basic ingredients for a language of pre- and postconditions, expressing either what we expect to find out or what we already know. Let us use these ingredients to specify operations, and let us construct our monads systematically from the operations thus specified, somewhat in the style of Wouter Swierstra (2008).

We shall need three operations, to open and close files and to get a character from a file if one is available. I *specify* these operations below, giving predicates over $\{\mathsf{State}\}$.

operation	**precondition**	**postcondition**
fOpen	FilePath $\coloneqq\{$ Closed $\}$	(::State)
fGetC	$() \coloneqq\{$ Open $\}$	Maybe Char $\coloneqq\{$ Open $\}$
fClose	$() \coloneqq\{$ Open $\}$	$() \coloneqq\{$ Closed $\}$

Crucially, the specification for fOpen demands a FilePath and that the handle is currently $\{\mathsf{Closed}\}$, but it promises only to inform us of the resulting $\{\mathsf{State}\}$, which cannot be known until run time. The other two operations make predictable state transitions.

We can be entirely systematic in constructing a monad characterizing what it is to program against this signature, using a simple predicate transformer kit. The first component expresses what it is to be reachable by executing a command with a given specification.

```
data (p :>>: q) r i = p {i} :& (q :-> r)
```

The infix constructor :& just packs up the evidence that the command's precondition p holds *now*, and a *callback* to be invoked in expectation of the result r when the command has delivered its postcondition q. It is not hard to see that postcomposition makes $p :\!\!\gg\!\!: q$ an IFunctor, weakening the result predicate:

```
instance IFunctor (p :>>: q) where
  imap h (p :& k) = p :& (h . k)
```

We can offer a *choice* of commands by closing IFunctor under sums.

```
data (f :+: g) p i = InL (f p {i}) | InR (g p {i})
instance (IFunctor f, IFunctor g) => IFunctor (f :+: g) where
  imap h (InL fp) = InL (imap h fp)
  imap h (InR gp) = InR (imap h gp)
```

Making $:\!\!\gg\!\!:$ bind more tightly than $:+:$, we may now write our signature as a single predicate transformer, offering three possible commands.

```
type FH  -- ::({State} -> *) -> {State} -> *
  =  FilePath :={ Closed } :>>: (::State)              -- fOpen
  :+: ()      :={ Open }   :>>: Maybe Char :={ Open }  -- fGetC
  :+: ()      :={ Open }   :>>: () :={ Closed }        -- fClose
```

To explain what it means to be a *strategy* for reaching some condition by executing such commands, we can just use the *free* monad construction, a kind of Kleene-star for functors.

```
data (:*) :: (({i} -> *) -> {i} -> *) -> ({i} -> *) -> {i} -> * where
  Ret ::p {i}          -> (f :* p) {i}
  Do  ::f (f :* p) {i} -> (f :* p) {i}
```

Note that this is quite standard, but for Haskell's rigid syntax, we might have declared

```
Ret ::p          :-> f :* p  -- do nothing
Do  ::f (f :* p) :-> f :* p  -- do one thing then more things
```

Witnesses to $f \mathbin{:\!*} p$ are trees with f-shaped nodes each representing a single command-response interaction, and p-witnessing leaves showing that every path succeeds. The empty tree does nothing, and we can represent sequencing by grafting a strategy tree for a later computation in place of each leaf of an earlier computation's tree. Functoriality — grafting leaf for leaf — arises as a special case.

```
instance IFunctor f ⇒ IMonad ((:*) f) where
  iskip = Ret
  iextend g (Ret p)  = g p
  iextend g (Do ffp) = Do (imap (iextend g) ffp)
instance IFunctor f ⇒ IFunctor ((:*) f) where
  imap f = iextend (iskip . f)
```

With these definitions in place, we may take $(\mathsf{FH} :\!*)$ to be the monad in which we program our interactions. We specified the interface, giving each operation its pre- and postcondition, and the monad wrote itself. In an FH strategy tree, each node represents a choice of command and the evidence for its precondition; each edge from a node represents a possible response and the evidence that the postcondition holds in the new state. In effect, we have the *interaction structures* of Peter Hancock and Anton Setzer (2000).

Of course, the generic construction results in less than readable strategy trees — anonymous mixtures of Do, InL, InR and :&. We can define the combinations which correspond to our operations, but that does not help us when we inspect strategies, as we shall in the next section. William Aitken and John Reppy (1992) proposed a technology which solves this problem: SHE adopts it, supporting *pattern synonyms* — definitions restricted to linear constructor forms, thus equally admissible left and right. We may define

```
pattern FOpen p k = Do (InL (V p :& k))
pattern FGetC   k = Do (InR (InL (V () :& k)))
pattern FClose  k = Do (InR (InR (V () :& k)))
```

and then implement our three monadic operations as one-node trees.

```
fOpen  :: FilePath → (FH :* (::State)) {Closed}
fOpen p = FOpen p Ret
fGetC  :: (FH :* (Maybe Char :={Open})) {Open}
fGetC  = FGetC Ret
fClose :: (FH :* (() :={Closed})) {Open}
fClose = FClose Ret
```

It does not take so much imagination to hope that one day we might have language support for signatures of operations with specifications, and arrive more succinctly at this outcome.

7 Interpreting Strategies

It is one thing to have a type of strategy trees for file-handling computation and quite another to get our hands on some files. To make use of our monad, we shall need to write an interpreter which reads our little commands and runs them in the big bad world of IO. This interpreter should follow one path in the tree, performing the command given at each

node in turn, and taking the edge determined by reality to be the response. We map the grafting structure of the trees to the $\gg\!=$ structure of the IO monad.

```
runFH :: (FH :* (a :=: {Closed})) {Closed} → IO a
runFH (Ret (V a)) = return a
runFH (FOpen s k) = catch
  (openFile s ReadMode >>= openFH (k {Open}))
  (λ_ → runFH (k {Closed}))
  where
    openFH :: (FH :* (a :=: {Closed})) {Open} → Handle → IO a
    openFH (FClose k) h = hClose h >> runFH (k (V ()))
    openFH (FGetC k) h = catch
      (hGetChar h >>= λc → openFH (k (V (Just c))) h)
      (λ_ → openFH (k (V Nothing)) h)
```

Exceptions arising in openFile or hGetChar are caught and rendered data. For the former, the singleton State witness $\{\mathsf{Closed}\}$ is passed to the callback k, by contrast with the $\{\mathsf{Open}\}$ witness delivered upon success. Note also that runFH supports only those processes which have the grace to leave the file handle closed.

8 The angelic Atkey applicative interface

Not only from personal predilection, but also as a matter of notational convenience, I propose also to consider how to equip our endofunctors on $\{i\} \to *$ with the interface of an *applicative functor* (McBride & Paterson, 2008). In general, demonic activity makes this structure hard to attain. Applicative functors sequence computations with no value dependency: in

```
(<*>) :: Applicative a ⇒ a (s → t) → a s → a t
```

there is no way for the *value* of the function computation to influence our *choice* of argument computation, only the way in which its value is used in turn. In our state-indexed setting, however, the result of the function computation carries the witness to the state from which the argument computation executes, so we cannot readily ignore it. We may, however, equip the angelic fragment of our state-indexed computations with the applicative interface. By ordaining the intermediate state in advance, we free ourselves from the need to depend on the intermediate value. Let us take what we can get, restricting to Atkey computations whose values yield a function and its argument, and whose states join up domino-style.

```
class IFunctor m ⇒ IApplicative (m :: ({i} → *) → {i} → *) where
  pure :: x → Atkey m i i x
  (⊛) :: Atkey m i j (s → t) → Atkey m j k s → Atkey m i k t
```

Our free monads are applicative by the standard construction which makes all monads applicative, lifted to $\{i\} \to *$ at no notational expense.

```
instance IFunctor f ⇒ IApplicative ((:*) f) where
  pure = ireturn
  mf ⊛ ms = mf >>= λf → ms >>= λs → ireturn (f s)
```

I have shadowed the usual unindexed applicative combinators in order to exploit a notational convenience. SHE provides *idiom brackets*, as previously proposed (McBride & Paterson, 2008), but with 'banana brackets' as the delimiters. In particular,

$$(|\, f\ a_1\, ..\, a_n\, |) \quad \text{expands to} \quad \mathsf{pure}\, f \circledast a_1 \circledast .. \circledast a_n$$

Moreover, 'tacks' $(-..-)$, used inside idiom brackets, delimit actions whose effects happen but whose values are not passed to the function at the head of the bracket. The effects in an idiom bracket are sequenced left-to-right; the value application structure is just as written, omitting the tacks. We shall have an example in just a moment.

9 Iron filings

We now have all the kit we need to write our fileContents program. Here it is!

```
fileContents :: FilePath → (FH :* (Maybe String :={ Closed })) { Closed }
fileContents p = fOpen p ?>= λb → case b of
  { Closed } → (| Nothing |)
  { Open }   → (| Just readOpenFile (−fClose−) |)
readOpenFile :: (FH :* (String :={ Open })) { Open }
readOpenFile = fGetC >>= λx → case x of
  Nothing → (| "" |)
  Just c  → (| (c:) readOpenFile |)
```

To get the contents of the file at path p, we try to open it with fOpen—note the demonic bind, reflecting our uncertainty as to the resulting state. Testing that state tells us whether to give up, returning Nothing in the { Closed } state, or to proceed in the { Open } state, returning Just the contents of the file, then, incidentally, closing it with the tacked $(-\mathsf{fClose}-)$. The readOpenFile, meanwhile, repeatedly calls fGetC until Nothing more is to be read—the angelic bind reflects our certainty that fGetC preserves the { Open } state.

Crucially, fileContents *must* test the state before invoking readOpenFile. If we try

```
fileContents p = fOpen p ?>= λb → (| Just readOpenFile (−fClose−) |)
```

we learn that the body of the $\lambda b \rightarrow$ inhabits $(\mathsf{FH} :^* ..)\ \{\mathsf{Open}\}$, which fails to fit the expected $(\mathsf{FH} :^* ..)\ \{i\}$, because $\{i\}$ is a *rigid* type variable, standing for a State chosen by the demon, beyond our control. We have a cast iron guarantee that we act on the file resource in accordance with what we know.

So, choose a big text file, a Shakespearean tragedy, perhaps, and invoke

```
runFH $ fileContents "Hamlet.txt"
```

and wait.

10 Codensity: free speed

My naïve free monad implementation is highly inefficient when $>\!\!=$ is heavily left-nested, resulting in repeated traversal of the strategy trees. Fortunately, the *codensity* transformation (Hutton *et al.*, 2010), replacing tree-grafting by continuation-passing, works just as usual on our $\{i\} \to *$ monads.

```
data (:^) :: ((S i -> *) -> S i -> *) -> (S i -> *) -> S i -> * where
  RET :: s i -> (m :^ s) i
  DO  :: (forall t. (s :-> (m :^ t)) -> m (m :^ t) i) -> (m :^ s) i
```

Here, DO expresses the reachability of s indirectly, as an offer to continue to any t reachable from s after at least one action. The iextend operation pastes actions into the strategy by composition.

```
instance IMonad ((:^) m) where
  iskip = RET
  iextend f (RET s) = f s
  iextend f (DO g)  = DO (\k -> g (iextend k . f))
```

We can equip $(:\!\!\wedge)$ with the same constructor 'thunk' and case analysis 'force' interface as $(:\!\!*)$, as follows. Given a command reaching a further computation, a continuation should be imapped to extend all the possible outcomes.

```
thunk :: IFunctor f => Either (f (f :^ t) (S i)) (t (S i)) -> (f :^ t) (S i)
thunk (Right t) = RET t
thunk (Left fft) = DO (\k -> imap (iextend k) fft)
```

Again, pattern synonyms make the encoding readable. We can then thunk the patterns to define fOpen, fGetC and fClose as 'smart constructors'.

```
pattern FRet a     = (Right (V a))
pattern FOpen p k  = Left (InL (V p :& k))
pattern FGetC  k   = Left (InR (InL (V () :& k))
pattern FClose k   = Left (InR (InR (V () :& k))

fOpen  :: FilePath -> (FH :^ (::State)) (S Closed)
fOpen p = thunk (FOpen p RET)
fGetC  :: (FH :^ (Maybe Char :=: S Open)) (S Open)
fGetC  = thunk (FGetC RET)
fClose :: (FH :^ (() :=: S Closed)) (S Open)
fClose = thunk (FClose RET)
```

Meanwhile, given a computation, we can tell if there is a command or not, and if the latter, we can reveal the command by passing in the trivial continuation.

```
force :: (f :^ t) (S i) -> Either (f (f :^ t) (S i)) (t (S i))
force (RET t) = Right t
force (DO k)  = Left (k RET)
```

Invoking force for each case analysis allows us to reimplement runFH for the continuation-passing version.

```
runFH :: (FH :^ (a :=: {Closed})) {Closed} → IO a
runFH c = case force c of
    FRet a      → return a
    FOpen s k → catch
      (openFile s ReadMode >>= openFH (k {Open}))
      (λ_ → runFH (k {Closed}))
  where
    openFH :: (FH :^ (a :=: {Closed})) {Open} → Handle → IO a
    openFH c h = case force c of
      FClose k → hClose h >> runFH (k (V ()))
      FGetC k → catch
        (hGetChar h >>= λc → openFH (k (V (Just c))) h)
        (λ_ → openFH (k (V Nothing)) h)
```

With this interpreter rebuilt, the fileContents we can write $(:^\wedge)$ instead of $(:^*)$ in the type but keep the program code the same.

```
fileContents :: FilePath → (FH :^ (Maybe String :=: {Closed})) {Closed}
fileContents p = fOpen p ?>= λb → case b of
  {Closed} → (| Nothing |)
  {Open}   → (| Just readOpenFile (−fClose−) |)

readOpenFile :: (FH :^ (String :=: {Open})) {Open}
readOpenFile = fGetC =>= λx → case x of
  Nothing → (| "" |)
  Just c    → (| (c:) readOpenFile |)
```

We thus achieve efficiency whilst retaining the safety of the abstraction and the modularity of the interface. Neither our specification of the interface as the FH functor, nor the code written to that interface has changed to accommodate the codensity transformation. Only the boilerplate is different.

11 Epilogue

SHE has made it convenient to work with indexed sets in Haskell, along with their appropriate notions of IFunctor and IMonad. These have just the same interface as their unindexed counterparts, given the appropriate notion of index-respecting morphism. The resulting structure is neither abstruse nor newfangled, but rather a familiar old friend, Hoare logic. Rather than following the 'Hoare Type Theory' of Aleks Nanevski and colleagues (Nanevski *et al.*, 2008), constructing a logical *superstructure* for monadic programming, I have yanked Hoare logic across the Curry-Howard correspondence and used it directly as logical *infrastructure* in Haskell's type system. Standard constructions allowed me to set up the monad I needed just by writing down a signature of commands speficied with pre- and post-conditions.

Of course, I have given but one example of computation in a dangerous world rationalised in a type-safe way, and with the least complex nontrivial state-space possible.

To scale up, we will need a compositional approach to modelling the state of just the parts of the world we locally need to consider. We shall need to design a library of type combinators, taking the rich literature of predicate transformers and refinement calculi as our guide. The Kleisli arrows of outrageous fortune explain "perchance"—it remains, however, to dream.

References

Aitken, William, & Reppy, John. (1992). *Abstract value constructors*. Tech. rept. TR 92-1290. Cornell University.

Atkey, Robert. (2009). Parameterised notions of computation. *J. funct. program.*, **19**(3-4), 335–376.

Filliâtre, J.-C. 1999 (November). *A theory of monads parameterized by effects*. Research Report 1367. LRI, Université Paris Sud.

Hancock, Peter, & Setzer, Anton. (2000). Interactive programs in dependent type theory. *Pages 317–331 of:* Clote, Peter, & Schwichtenberg, Helmut (eds), *Csl*. Lecture Notes in Computer Science, vol. 1862. Springer.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. acm*, **12**(10), 576–580.

Hutton, Graham, Jaskelioff, Mauro, & Gill, Andy. (2010). Factorising folds for faster functions. *J. funct. program.*, **20**(3-4), 353–373.

Kiselyov, Oleg, & chieh Shan, Chung. (2008). Lightweight monadic regions. *Pages 1–12 of:* Gill, Andy (ed), *Haskell*. ACM.

Mac Lane, Saunders. (1998). *Categories for the Working Mathematician*. Second edn. Graduate Texts in Mathematics, no. 5. New York: Springer-Verlag.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *J. funct. program.*, **18**(1), 1–13.

Nanevski, Aleksandar, Morrisett, J. Gregory, & Birkedal, Lars. (2008). Hoare type theory, polymorphism and separation. *J. funct. program.*, **18**(5-6), 865–911.

Swierstra, Wouter. (2008). Data types à la carte. *J. funct. program.*, **18**(4), 423–436.

Uustalu, Tarmo. (2003). Generalizing substitution. *Ita*, **37**(4), 315–336.

Wadler, Philip, & Thiemann, Peter. (2003). The marriage of effects and monads. *Acm trans. comput. log.*, **4**(1), 1–32.