How to Declare an Imperative

Philip Wadler
Bell Laboratories, Lucent Technologies

How can we integrate interaction into a purely declarative language? This tutorial describes a solution to this problem based on a *monad*. The solution has been implemented in the functional language Haskell and the declarative language Escher. Comparisons are given to other approaches to interaction based on synchronous streams, continuations, linear logic, and side effects.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Input/Output*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms: Languages, Theory

Additional Key Words and Phrases: functional programming, monad, Haskell

1. INTRODUCTION

Four centuries ago, Descartes pondered the mind-body problem: how can incorporeal minds interact with physical bodies? He posited that the solution lay in the pineal gland: here, perhaps, was the place where the senses of the body provoked the images of the mind, and where the intentions of the mind initiated the actions of the body. (For a modern take on these medieval musings, I recommend Dennett's *Consciousness Explained* [9].)

Today, computing scientists face their own version of the mind-body problem: how can virtual software interact with the real world? In the beginning, we merely wanted computers to extend our minds: to calculate trajectories, to sum finances, and to recall addresses. But as time passed, we also wanted computers to extend our bodies: to guide missiles, to link telephones, and to proffer menus.

The classic models of computation are analogous to minds without bodies. For

Accepted to *ACM Computing Surveys*, to appear. This paper is based on an invited talk given at ILPS 95. A shorter version appears in John Lloyd, editor, *International Logic Programming Symposium*, MIT Press, December 1995. This version differs in that Section 3 and some of Section 4 is new. This work was supported by the UK EPSRC projects 'Save space with linear types' and 'Declarative systems architecture: A quantitative approach' and by the EC ESPRIT working groups 'Semantique' and 'Atlantique'. Address: Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ 07974-0636; email: `wadler@research.bell-labs.com`

Turing's machine, a calculation begins with a problem on its tape, and ends with an answer there. For Church's calculus, reduction begins with a lambda term, and ends with its normal form. For Floyd's flowcharts and Hoare's triples, a program begins in a state satisfying a precondition, and ends in a state satisfying a postcondition. How the initial tape or term or state is input, and how the final one is output, are questions neither asked nor answered. These theories conform to the practice of batch computing.

Eventually, interactive models of computation emerged, analogous to minds in bodies. For Petri's nets, tokens enter and leave locations. For Kahn and MacQueen's streams, data circulates between coroutines. For Milner's and Hoare's process calculi, messages are sent and received along channels. Inputs and outputs require no special treatment, as they are represented simply as additional token sources and sinks, additional streams, or additional channels. A single input at initiation and a single output at termination is now superseded by multiple inputs and outputs distributed in time and space. These theories conform to the practice of interactive computing.

Interaction is the mind-body problem of computing. It poses a challenge to all computer scientists, but the challenge it poses to those of us interested in declarative languages is particularly acute. Although Turing machines and flowcharts are classified as imperative, the classic models of computation are essentially declarative, since a program behaves as a function from (or, if you prefer, a predicate relating) inputs to outputs. But the interactive models of computation appear inherently imperative, since the whole point of augmenting minds with bodies is to make it possible to do something.

This tutorial reviews a solution to the interaction problem that has become popular within the functional programming community. It is based on the notion of a *monad*.

Monads arose in category theory [38]. Eugenio Moggi noted that monads could be used to model a wide variety of language features, including non-termination, state, exceptions, continuations, and interaction [42; 43]. Moggi's technique of structuring a denotational semantics adapts directly for use in structuring functional programs, and my own contribution was to foster this adaptation [59; 61; 62].

This paper provides an introduction to the use of monads to add interaction to a pure functional language, as described in previous work by Simon Peyton Jones and myself [50]. Similar models have been proposed by Cupitt [7] and Gordon [14]. For a history of approaches to input-output in functional programming, I recommend Gordon's thesis [14].

These ideas have been tested in the standard lazy functional language Haskell [21]. The ideas were originally incorporated in the Glasgow Haskell compiler; subsequently added to the Chalmers and Yale Haskell compilers; and adopted for inclusion in the revised Haskell standard [49]. This paper presents a somewhat simplified version of the new Haskell standard. This style of interaction has been tested extensively, including its use in programs tens of thousands of lines long, and in a range of applications including graphical user interfaces. These ideas have also been adopted by the declarative language Escher [37], which combines functional and logic programming.

Monads have also served as a basis for adding other features to a functional language, notably state and concurrency. Little will be said about these topics here, except to give a few pointers to the relevant literature. In particular, although interaction is often associated with concurrency and non-determinism, the model pursued here will be deterministic and sequential.

The reader is assumed to have a passing familiarity with the basics of functional programming in pure languages such as Haskell [21; 49], and impure languages such as SML [40; 41]. For general background see Bird and Wadler [5] and Paulson [45]. No knowledge of category theory is assumed.

A shorter version of this paper appeared previously [65]. Material in Sections 3 and 4 is new.

The remainder of this paper is organised as follows. Section 2 introduces a monad for interaction. Section 3 relates the monad approach to other approaches to interaction. Section 4 describes related work. Section 5 sketches how monads might be incorporated into a first-order language for logic programming, and concludes.

2. A MONAD APPROACH TO INTERACTION

This section introduces, step by step, an abstract type to support interaction, called a *monad*.

2.1 Commands

The type of simple commands is written `IO ()`. Ignore the trailing `()` for now — its purpose will become apparent later.

The mind-body distinction is essential to this enterprise. A term of type `IO ()` *denotes* an action, but does not necessarily *perform* the action: it is of the realm of the mind, rather than the realm of the body.

Here is a function to print a character.

```
putc :: Char -> IO ()
```

For instance, `putc '!'` denotes the command that, *if it is ever performed*, will print an exclamation mark.

Here is a constant to do nothing.

```
done ::  IO ()
```

The term `done` doesn't actually do nothing; it just specifies the command that, *if it is ever performed*, won't do anything. Compare thinking about doing nothing to actually doing nothing: they are distinct enterprises.

Here is a function to combine commands; it is roughly the analogue of semicolon in conventional imperative languages.

```
(>>) :: IO () -> IO () -> IO ()
```

If `m` and `n` are commands, then `m >> n` denotes the command that, *if it is ever performed*, first does `m` and then does `n`. (In Haskell, `m >> n` is syntactic sugar for `(>>) m n`.)

Here is a function that takes a string and returns a command that prints the string.

```
puts        :: String -> IO ()
```

```
puts []     =  done
puts (c:s)  =  putc c >> puts s
```

If the string is empty, then the command does nothing. If the string has head `c` and tail `s`, then the command first prints character `c` and then prints string `s`. So `puts "!?"` is equivalent to

```
putc '!' >> (putc '?' >> done)
```

and both of these denote a command that, *if it is ever performed*, prints an exclamation followed by a question.

(In Haskell, a string is just a list of characters. Hence `"!?" :: String` is just an abbreviation for `['!','?'] :: [Char]`. The latter in turn is an abbreviation for `'!':('?':[])`, where `:` is pronounced "cons" and `[]` is pronounced "nil".)

By now the reader will be desperate to know *how is a command ever performed?* In other words, how does the mind connect to the body? In Haskell, this is accomplished with the distinguished top-level variable `main`, which is bound to a value that specifies the command to be performed by the program. Thus executing the program

```
main :: IO ()
main =  puts "!?"
```

prints an indicator of perplexity. Thus `main` is the link from Haskell's mind to Haskell's body — the analogue of Descartes's pineal gland.

One may be disappointed that commands can only appear at the top-level of a program. Surely such a narrow interface as `main` will prove a bottleneck? But, as we will see, our type of commands is highly expressive, and can include arbitrary blends of interaction and computation. Just as Descartes believed that a soul could infuse an entire body through the pineal gland, so can a program interact with the entire world via `main`.

2.2 Equational reasoning

Equational reasoning is a principle of such importance that it goes by many names: "referential transparency", "the rule of Leibniz", or more plainly "substituting equals for equals". Our approach to commands preserves simple equational reasoning, which an approach based on side effects does not.

To see this, let's compare the use of monads in Haskell with the use of side effects in SML. For direct comparison, we assume a primitive `putcML` such that evaluating the expression

```
putcML #"h"; putcML #"a";
putcML #"h"; putcML #"a"
```

prints "`haha`" as a side effect. (The character written `'h'` in Haskell is written `#"h"` in the most recent revision of SML [41].) If we attempt to capture the commonality in this program by writing

```
let val
  x = (putcML #"h"; putcML #"a")
in x; x end
```

then the laugh is on us: the program prints only a single "ha", at the time variable x is bound. In the presence of side effects, equational reasoning in its simplest form becomes invalid.

One can use a more complex form of equational reasoning in SML. Writing

```
let fun
  f () = (putcML #"h"; putcML #"a")
in f (); f () end
```

defines a function f with dummy argument (), and properly abstracts the interaction. Thus in SML one must abstract values and interactions differently.

In Haskell, the expression

```
putc 'h' >> putc 'a' >>
putc 'h' >> putc 'a'
```

and the expression

```
let
  x = (putc 'h' >> putc 'a')
in x >> x
```

are entirely equivalent. Thus in Haskell one may abstract values and interactions in exactly the same way. Equational reasoning is kept simple by an appropriate distinction between the roles of mind and body.

2.3 Commands that yield values

The above is adequate for output, but needs to be generalised for input. If a is a type, then IO a is the type of commands that yield a value of type a. So far, we have considered the special case IO (). (In Haskell, () is the trivial type that contains just one proper value, which is also written ().)

Here is a function to read a character.

```
getc :: IO Char
```

Performing the command getc when the input contains ABC yields the value 'A' and remaining input BC.

Generalising the command done, which does nothing and yields no value, is the command return x, which does nothing and yields value x.

```
return :: a -> IO a
```

Performing the command return 42 when the input contains ABC yields the value 42 and an unchanged input ABC. (Here a is a *type variable*, which thanks to the wonders of *polymorphism* may be instantiated to any type, in this case Int.)

Combining commands is a little tricky. One common approach is to define an operation which takes a pair of commands that yield values to a command which yields a pair of values.

```
(>>*) :: IO a -> IO b -> IO (a,b)
```

Performing the command getc >>* return 42 when the input contains ABC yields the value ('A',42) and remaining input BC.

Continuing this approach, one may also provide an operation which applies a function to the result of a command.

```
(>>@) :: IO a -> (a -> b) -> IO b
```

Performing the command `getc >>@ \c-> [c,c]` when the input contains `abc`, yields the value `"AA"` and remaining input `BC`. (In Haskell, `\x-> e` is the equivalent of the lambda expression $\lambda x.\, e$, so applying the function `\c-> [c,c]` to the character `'a'` yields the string `"aa"`.)

Here is a function to read a given number of characters.

```
gets        :: Int -> IO String
gets 0      =  return []
gets (i+1)  =  (getc >>* gets i)
               >>@ \(c,s)-> c:s
```

Performing the command `gets 2` when the input contains `ABC` yields the value `"AB"` and remaining input `C`.

The set of combinators based on `>>*` and `>>@`, which seems natural enough, leads to a style in which even as simple a function as `gets` is not especially easy to read. Fortunately, there is another set of combinators that, though it appears less natural, leads to a style in which functions are easier to read.

2.4 An analogue of `let`

The new combinator is written `>>=` and pronounced "bind".

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

If `m :: IO a` is a command yielding a value of type `a`, and `k :: a -> IO b` is a function from a value of type `a` to a command yielding a value of type `b`, then `m >>= k :: IO b` is the command that, *if it is ever performed*, behaves as follows: first perform command `m` yielding a value `x` of type `a`; then perform command `k x` yielding a value `y` of type `b`; then yield the final value `y`.

Although it may seem odd at first sight, this combinator is reassuringly similar to the familiar `let` expression. Those familiar with type inference rules will recognise the rule for `let`.

$$\frac{\vdash \mathtt{m :: a} \qquad \mathtt{x :: a} \vdash \mathtt{n :: b}}{\vdash \mathtt{let\ x{=}m\ in\ n :: b}}$$

This rule states that if term `m` has type `a`, and (assuming that variable `x` has type `a`) term `n` has type `b`, then the term `let x=m in n` has type `b`. To compute `let x=m in n`, first compute `m`, then bind `x` to the value yielded, then compute `n`.

Typically, bind is combined with lambda expressions in a way that resembles `let` expressions. Here is the corresponding type rule.

$$\frac{\vdash \mathtt{m :: IO\ a} \qquad \mathtt{x :: a} \vdash \mathtt{n :: IO\ b}}{\vdash \mathtt{m >\!>= \backslash x\text{-}> n :: IO\ b}}$$

If term `m` has type `IO a` and (assuming that variable `x` has type `a`) term `n` has type `IO b`, then the term `m >>= \x-> n` has type `IO b`. To perform `m >>= \x-> n`, first perform `m`, then bind `x` to the value yielded, then perform `n`.

Note the similarity to the SML `let` expression.

```
let val x = m in n end
```

To compute this, first compute m (and perform its side effects), then bind x to the value yielded, then compute n (and perform its side effects). The key difference is that >>= preserves equational reasoning, while the SML let with side effects does not.

Because of this similarity, one may wish to introduce a variant let expression such as

```
let x <- m in n
```

(where the equal sign has been replaced by an arrow) as equivalent to m >>= \x-> n. We'll return to this in the conclusion.

The combinator >>* may be defined in terms of return and >>=.

```
(>>*)     :: IO a -> IO b -> IO (a,b)
m >>* n   =  m >>= \x->
             n >>= \y->
             return (x,y)
```

This has a straightforward reading. To perform m >>* n, first perform m, bind its value to x, then perform n, bind its value to y, and yield the value (x,y).

Consider performing getc >>* getc when the input contains ABC. Performing the first getc yields value 'A', which is bound to x, and remaining input BC. Performing the second getc yields value 'B', which is bound to y, and remaining input C. Performing return (x,y) yields the final value ('A','B').

The combinator >>@ is also easily defined.

```
(>>@)     :: IO a -> (a -> b) -> IO b
m >>@ f   =  m >>= \x->
             return (f x)
```

To perform m >>@ f, first perform m, then bind x to the value yielded, and finally yield the value f x.

But we no longer require >>* and >>@, because it is easier to define a function like gets directly in terms of >>= and return.

```
gets         :: Int -> IO String
gets 0       =  return []
gets (i+1)   =  getc   >>= \c->
                gets i >>= \s->
                return (c:s)
```

Again, this has a straightforward reading. To get a string of length i+1, first get a character, bind it to c, then get a string of length i, bind it to s, then yield the string c:s.

The command done is a special case of return, and the combinator >> is a special case of >>=.

```
done    :: IO ()
done    =  return ()
```

```
(>>)     :: IO () -> IO () -> IO ()
m >> n   =  m >>= \()-> n
```

(Recall that `()` stands for both the trivial type and its one value.)

Several researchers, including myself, have published combinators for parsing based on operations analogous to `>>*` and `>>@` [11; 45; 58]. I now believe that `>>=` provides a far superior style. Others may have been clever enough to make the switch from `>>*` and `>>@` to `>>=` on their own, but in my case I would attribute the improvement directly to my contact with Moggi's work, and indirectly to Kleisli's abstract formulation of a monad in category theory.

To summarise, here is an interface for the input-output monad.

```
data   IO a
return  ::  a -> IO a
(>>=)   ::  IO a -> (a -> IO b) -> IO b
putc    ::  Char -> IO ()
getc    ::  IO Char
```

The first line states that `IO a` is an abstract data type. There are four operations on this type, the two combining forms, `return` and `>>=`, and the two primitives, `putc` and `getc`. Everything else, such as `done` and `>>`, can be defined in terms of these.

Operations of an abstract data type can often be characterised by the laws they satisfy, and we now turn to that question.

2.5 Monad laws

The command `done` is a left and right unit for `>>`, and `>>` is associative.

```
done >> m   =  m
m >> done   =  m
m >> (n >> o)   =  (m >> n) >> o
```

In other words, `done` and `(>>)` form a *monoid.*

Analogously, there is a sense in which `return` is a left and right unit for `>>=`, and `>>=` is associative.

```
return v >>= \x-> m   =  m[x:=v]
m >>= \x-> return x   =  m
m >>= \x-> (n >>= \y-> o)   =
          (m >>= \x-> n) >>= \y-> o
```

In the first line, variable `x` may appear free in term `m` and `m[x:=v]` stands for term `m` with each free occurrence of variable `x` replaced by term `v`. In the third line, variable `x` may appear free in term `n` but not in term `o`, and variable `y` may appear free in term `o`.

Categorists are infamous for stealing terms from philosophy, starting with the theft of *category* itself from Kant. The theft of *monad* from Leibniz to name the above structure was aided and abetted by the pun on *monoid.* (For Leibniz, monads were central to the mind-body problem, since each soul is a monad, as is God.)

In general, a monoid is a type `M` together with operators of types

```
done :: M
(>>) :: M -> M -> M
```

satisfying the first set of three laws above. The specific operators `done` and `(>>)` discussed here form a monoid, but so do many others. For instance, take `M` to be the integers, take `done` to be zero, and take `>>` to be addition.

Similarly, for functional programmers a monad is a type constructor `M`, together with operators of types

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

satisfying the second set of three laws above. Again, the specific operators `return` and `(>>=)` described here form a monad, but many others do as well. (We will discuss one other in Section 5.2.)

The three monad laws have analogues in `let` notation.

```
let x=v in m  =  m[x:=v]
let x=m in x  =  m
let y=(let x=m in n) in o  =
          let x=m in (let y=n in o)
```

These law are not merely true, they are *very* true. They hold even in a language such as SML, where the presence of side effects disables many forms of equational reasoning. For the first law to be true, `v` must be not an arbitrary term but a *value*, that is, a variable or a lambda expression but not an application. A value immediately evaluates to itself, hence its evaluation always terminates and can have no side effects. Unlike SML, Haskell distinguishes `>>=` on commands from `let` on values. While in SML one only has the above three laws for `let`, in Haskell one has a much stronger law.

```
let x=m in n  =  n[x:=m]
```

Here one may replace a variable by any term, rather than replace a variable by a value.

Using the monad laws, it is straightforward to prove some properties of programs. Write `++` for list concatenation, with the usual definition.

```
[]++s     =  s
(c:r)++s  =  c:(r++s)
```

Proposition. Two `puts` operations may be combined as follows.

```
puts r >> puts s  =  puts (r++s)
```

Proof. The proof is by induction on `r`.

Case `[]`.

```
  puts [] >> puts s
=        { definition puts }
  done >> puts s
=        { left identity >> }
  puts s
```

```
=       { definition ++ }
  puts ([]++s)
```

Case c:r.

```
  puts (c:r) >> puts s
=       { definition puts }
  (putc c >> puts r) >> puts s
=       { associativity >> }
  putc c >> (puts r >> puts s)
=       { inductive hypothesis }
  putc c >> puts (r++s)
=       { definition puts }
  puts (c:(r++s))
=       { definition ++ }
  puts ((c:r)++s)
```

□

Proposition. Two gets may be combined as follows.

```
  gets i >>= \r->
  gets j >>= \s->
  return (r++s)
=
  gets (i+j)
```

Proof. The proof is by induction on i.
Case 0.

```
  gets 0 >>= \r->
  gets j >>= \s->
  return (r++s)
=       { definition gets }
  return [] >>= \r->
  gets j >>= \s->
  return (r++s)
=       { left unit >>= }
  gets j >>= \s->
  return ([]++s)
=       { left unit ++ }
  gets j >>= \s->
  return s
=       { right unit >>= }
  gets j
=       { arithmetic }
  gets (0+j)
```

Case i+1.

```
  gets (i+1) >>=\r'->
  gets j >>=\s->
```

```
  return (r'++s)
=       { definition gets }
  (getc >>=\c->
   gets i >>=\r->
   return (c:r)) >>=\r'->
  gets j >>=\s->
  return (r'++s)
=       { associativity >>= }
  getc >>=\c->
  gets i >>=\r->
  (return (c:r) >>=\r'->
   gets j >>=\s->
   return (r'++s))
=       { left unit >>= }
  getc >>=\c->
  gets i >>=\r->
  gets j >>=\s->
  return ((c:r)++s)
=       { definition ++ }
  getc >>=\c->
  gets i >>=\r->
  gets j >>=\s->
  return (c:(r++s))
=       { left unit >>= }
  getc >>=\c->
  gets i >>=\r->
  gets j >>=\s->
  (return (r++s) >>=\t->
   return (c:t))
=       { associativity >>= }
  getc >>=\c->
  (gets i >>=\r->
   gets j >>=\s->
   return (r++s)) >>=\t->
  return (c:t)
=       { inductive hypothesis }
  getc >>=\c->
  gets (i+j) >>=\t->
  return (c:t)
=       { definition gets }
  gets ((i+j)+1)
=       { arithmetic }
  gets ((i+1)+j)
```

□

Each of these proofs is entirely straightforward, using a style common in functional programming community [5]. Here only the three monad laws are required for the proof, and we need no laws to describe the behaviour of `getc` or `putc`.

While the three monad laws are solidly established and helpful, further work is required on the best way to describe specific effects within a monad. For instance, one might want to specify that if a stream of characters is written to a file then the same stream will be read from the file, if no other program changes the file in the interim.

2.6 Monads and imperative programming

Here is a command which echoes one line of the input to the output. The newline character `'\n'` terminates the input line, but is not copied to the output.

```
echo  :: IO ()
echo  =  getc >>= \c->
         if (c == '\n') then
           done
         else
           putc c >>
           echo
```

This looks remarkably like a program in an imperative language, such as C.

```
echo () {
        int c;
loop:   c = getchar();
        if (c == '\n') {
          return
        } else {
          putchar(c);
          goto loop;
        }
}
```

Does the monadic style force one, in effect, to write a functional facsimile of an imperative program?

In one sense, the answer is yes, and rightly so. Some interactions appear most straightforward to express in an imperative style, and we should not hesitate to do so. In another sense, the answer is certainly not. For those portions of a program which are independent of interaction, all of the functional techniques that functional programmers have come to know and love still apply.

The similarity of the two programs is not in vain: the former compiles into something closely resembling the latter. This is accomplished by extensive use of equational reasoning in the Glasgow Haskell compiler. Whereas some declarative programmers only pay lip service to equational reasoning, users of functional languages exploit them every time they run a compiler, whether they notice it or not.

Combinations of imperative and functional style are possible. Here is a function that takes a list of commands that yield values to a command that yields a list of values.

```
prod         :: [IO a] -> IO [a]
```

```
prod []      =  return []
prod (m:ms)  =  m >>= \x->
                prod ms >>= \xs->
                return (x:xs)
```

Using this one may rewrite `puts` and `gets` in a higher-order style.

```
puts s  =  prod (map putc s) >>= \_->
           return ()
gets i  =  prod (take i (repeat getc))
```

(This uses Haskell library functions: `map f xs` applies function `f` to each element of list `xs`, and `take i xs` computes the first `i` elements of list `xs`, and `repeat x` computes a list consisting of `x` repeated indefinitely.)

The ability to write higher-order functions such as `prod` is a bit like the ability to define new, special-purpose constructs in an imperative language.

2.7 Calling C directly

The mechanism described above extends to integrate Haskell directly with C. The Glasgow Haskell compiler augments the language with a new form of expression

```
ccall proc e1 ... en
```

where `proc` is the name of a C procedure, and `e1` through `en` are Haskell expressions of type `Char`, `Int`, or `Float`; the expression as a whole has type `IO Char`, `IO Int`, or `IO Float`. The Haskell compiler checks that the number and type of arguments conform to the types declared in C.

Here, slightly simplified, are definitions of `getc` and `putc`.

```
putc c  =  ccall putchar c
getc    =  ccall getchar
```

The `ccall` directly invokes the corresponding C library function. A practical consequence of this approach is that most of our IO system is written directly in Haskell, with a smattering of low-level calls to C where needed.

This mechanism amounts to allowing an arbitrary set of primitives, one for each C library function that appears in `ccall`, to be added to the abstract type summarised at the end of Section 2.4.

At present, we only allow values of base type to be passed between Haskell and C. It is possible, but not especially convenient, to write special-purpose routines enabling more complex structures to pass across this narrow interface. Enabling smooth sharing of more complex data remains a challenge for the future.

3. OTHER APPROACHES TO INTERACTION

This section relates the monad approach to input-output to four other widely used approaches: synchronised streams, as used in earlier versions of Haskell; continuations, as used in Hope; linear types, as used in Clean; and side effects, as used in SML. In each case, the presentation will be streamlined to two basic operations, to read and write a single character.

Recall that the monad approach to interaction is based on the type `IO a` and four operations provided by the system.

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
putc   :: Char -> IO ()
getc   :: IO Char
```

In turn, the user must provide the value of a distinguished variable.

```
main   :: IO ()
```

This value acts as a 'pineal gland', connecting thought to action.

For each of the four other approaches, the presentation follows a similar plan. First, a set of appropriate types and operations is presented, and an appropriate distinguished variable is described. Second, the program to echo a line of input is rewritten in the new style. Third, tradeoffs between the two styles are assessed. Fourth, it is shown how to define the monad model in terms of the new model. Fifth, if possible, it is shown how to define the new model in terms of the monad model.

This provides a first step toward comparing and relating the different approaches. Similar programs of comparison, implementing various input-output models in terms of others, have been carried out by Hudak and Sundaresh [22], Gordon [14], and Peyton Jones and Wadler [50].

3.1 Interaction by synchronised stream

Like many features of functional languages, the stream model of input-output arose out of work in denotational semantics. The stream model appears in the seminal work of Landin [31], and the refinement to synchronous streams is due to Stoye [57]. Early versions of Haskell used streams for input-output [21], while later versions use monads [49]. Since streams are often used as a semantics for input-output, the definition of monads in terms of streams, presented below, may be regarded as a semantics for the input-output monad.

Review of interaction by synchronised streams. In the stream model, at the top-level a program is represented by a *dialogue*, a function that yields a stream of *requests* and accepts a stream of *responses*. In a lazy language, a stream may be represented by a list.

```
type  Dialogue  =  [Response] -> [Request]
```

This approach to input-output depends on lazy evaluation, as each request must be returned from the program before the corresponding response is generated.

A request is either of the form `Getq`, indicating a character should be read, or of the form `Putq c`, indicating that character `c` should be written. Dually, a response is either of the form `Getp c`, indicating that character `c` has been read, or of the form `Putp`, indicating that a character has been written.

```
data  Request   =  Getq | Putq Char
data  Response  =  Getp Char | Putp
```

Here we abbreviate 'request' to 'req' and thence 'q', and 'response' to 'resp' and thence 'p'.

The behaviour of the entire program is specified by a distinguished variable, here called `mainD`.

```
mainD :: Dialogue
```

Thus synchronous streams share with monads the notion of a single link, analogous to the pineal gland, between thinking and doing. Indeed, all the models of interaction that we consider will have this property, with the exception of side effects.

Here is the program which echoes a line of its input to its output, implemented with synchronous streams.

```
echoD :: Dialogue
echoD p =
  Getq :
    case p of
      Getp c : p' ->
        if (c == '\n') then
          []
        else
          Putq c :
            case p' of
              Putp : p'' -> echoD p''
```

The program first issues the request `Getq` on its request stream, indicating a character should be read, causing the response `Getp c` to appear on its response stream, indicating character `c` was read. If the character is end of line, it then terminates the request stream, indicating the end of the program. Otherwise, it next issues the request `Putq c` on its request stream, indicating character `c` should be written, causing the response `Putp` to appear on its response stream, indicating the write occurred. It then repeats the loop.

For instance, if the input begins with `"AB\n"` then the two characters preceding the newline will be echoed to the output, as specified by the following requests and responses.

```
echoD [Getp 'A', Putp,     Getp 'B', Putp,     Getp '\n']
    = [Getq,     Putq 'A', Getq,     Putq 'B', Getq     ]
```

These streams represent the entire history of the process, with all indications of causality erased.

Using the language of denotational semantics, one may express the history by a sequence of approximations converging to a fixed point. We write `bottom` to denote a stream about which nothing is known.

```
echoD (bottom)
    =  Getq    : bottom
echoD (Getp 'A': bottom)
    =  Getq    : Putq 'A': bottom
echoD (Getp 'A': Putp    : bottom)
    =  Getq    : Putq 'A': Getq    : bottom
echoD (Getp 'A': Putp    : Getp 'B': bottom)
    =  Getq    : Putq 'A': Getq    : Putq 'B': bottom
echoD (Getp 'A': Putp    : Getp 'B': Putp    : Getp '\n': bottom)
```

```
    =  Getq    : Putq 'A': Getq    : Putq 'B': Getq     : []
echoD (Getp 'A': Putp    : Getp 'B': Putp    : Getp '\n': [])
    =  Getq    : Putq 'A': Getq    : Putq 'B': Getq     : []
```

This view restores the causality: first there are no responses and the request `Getq` is issued; next the response `Getp 'A'` appears and the request `Putq 'A'` is issued, and so on.

What is the trade off between monads and streams? Synchronous streams require that you mind your `ps` and `qs`: you must take care to ensure that a request is always issued before the corresponding response is consumed. Monads hide this level of detail. Further, monads are more modular than synchronous streams. With monads, you may simply write `echo >> echo` to echo two lines from the input to the output. To support similar modularity with the synchronous stream version of echo is difficult, at best.

For these reasons, monads are now generally considered preferable to synchronous streams in practice, though the stream model remains a useful theoretical tool. In particular, the definition of the monads in terms of streams, given below, can be regarded as providing a denotational semantics for the monad model.

From streams to monads. We now consider how to define monads in terms of synchronous streams. The type `IO a` stands for a computation that generates a part of the request stream, and consumes a corresponding part of the response stream, as well as returning a value of type `a`.

```
type  IO a  =  ([Response],[Request]) ->
               (a,[Response],[Request])
```

A typical use of computation `m :: IO a` has the form

```
(x,p',q) = m (p,q')
```

where `p` is the stream of responses passed to the computation, `q` is the stream of requests returned by the computation, `p'` is the stream of responses to be consumed *after* the computation, `q'` is the stream of requests to be generated *after* the computation, and `x` is the value of type `a` returned by the computation. This depends critically on lazy evaluation, because the stream `q'` of remaining requests passed to the computation may depend on the value `x` returned by the computation.

Given this formulation, it is straightforward to define monads in terms of streams.

```
(>>=)      :: IO a -> (a -> IO b) -> IO b
m >>= k    =  \(p,q'')-> let
                           (x,p',q)   = m (p,q')
                           (y,p'',q') = k x (p',q'')
                         in
                           (y,p'',q)

return     :: a -> IO a
return x   =  \(p,q)->   (x,p,q)

putc       :: Char -> IO ()
putc c     =  \(p,q')->  let
```

```
                              q = Putq c : q'
                              Putp : p' = p
                            in
                              ((),p',q)


getc      :: IO Char
getc      =  \(p,q')->  let
                              q = Getq : q'
                              Getp c : p' = p
                            in
                              (c,p',q)
```

Note that the `let` clause defining `>>=` is mutually recursive and uses laziness in an essential way: the first line binds `p'`, which depends on `q'`, while the second line binds `q'`, which depends on `p'`.

It is equally straightforward to relate the distinguished variables for streams and monads.

```
mainD  ::  Dialogue
mainD  =   \p->  let
                    (x,p',q) = main (p,q')
                    q' = []
                  in
                    q
```

Here the stream of requests `q'` to perform after `main` is empty, and hence so is the stream of responses `p'`. And `x` must have value `()` since `main` has type `IO ()`.

(There is one subtlety here. One might naively expect that replacing `(x,p',q)` by `((),[],q)` would yield an equivalent definition, but it does not. The definition as it stands returns `q` before the values assigned to `x` and `p'` are computed, which is essential because the list of responses `p` depends on the list of requests `q`. The altered form cannot return `q` until it checks that `x` is `()` and `p'` is `[]`, and this introduces a deadlock.)

Laws. It is a straightforward exercise to show that the three monad laws are satisfied. For instance, we show

```
return x >>= k  =  k x
```

by the following calculation.

```
  (return x >>= k) (p,q'')
=       { definition >>= }
  let
    (x',p',q)  = return x (p,q')
    (y,p'',q') = k x' (p',q'')
  in
    (y,p'',q)
=       { definition return }
  let
```

```
    (x',p',q)  = (x,p,q')
    (y,p'',q') = k x' (p',q'')
  in
    (y,p'',q)
=       { simplify }
  let
    (y,p'',q') = k x (p,q'')
  in
    (y,p'',q')
=       { simplify }
  k x (p,q'')
```

The other two laws are proved similarly.

From monads to streams. One may also ask whether the stream model can be implemented in terms of the monad model. The answer is yes and no. One can write such a function, but it turns out to be incredibly inefficient. So monads are easily defined in terms of synchronous streams, but not conversely. For details of the conversion, see [50].

3.2 Interaction by continuations

The previous section showed that the naked style of synchronised streams is unappealing, but can be improved by dressing it up in monads. Continuations may also be used for this purpose, as was done in earlier versions of Haskell [21]. Further, just as monads can be treated either as the old stream type in a new package or as an abstract data type in its own right, so too can continuations, as was done in Hope [46]. Here we give the formulation of continuations as an abstract type.

Historically, continuations are a direct predecessor of monads. Continuations, like monads, arose in denotational semantics, originally as a way to model flow of control. The canonical formulation is due to Plotkin [51], and an engaging history has been penned by Reynolds [52].

Review of interaction by continuations. In the continuation model, each operation takes an additional argument, itself called the *continuation*, that denotes the the entire remainder of the computation. At first sight this appears to be a remarkably unmodular notion — every action incorporates all succeeding actions! But, paradoxically, this style actually increases modularity.

In continuation style, the final result of the program is given the type `Answer`. In Hope `Answer` is an abstract type, while in earlier versions of Haskell `Answer` it is taken as equivalent to the `Dialogue` type of the previous section.

There are two primitives, one to write a character to the output and one to read a character from the input.

```
putcK :: Char -> Answer -> Answer
getcK :: (Char -> Answer) -> Answer
```

Excecuting `putcK c k` writes the character `c` to the output and then behaves as the continuation k, while executing `getc k` reads a character from the input, say `c`, and then behaves as the continuation `k c`. In the first case the continuation simply has type `k :: Answer` since writing a character yields no result, while in the second case the continuation has type `k :: Char -> Answer` since reading a character yield a result of type `Char` which is passed to the continuation.

Each primitive yields an `Answer`, and requires a continuation involving `Answer` as its argument. To terminate this infinite regress, there is a primitive corresponding to no action at all.

```
doneK :: Answer
```

The behaviour of the entire program is specified by the distinguished variable `mainK`.

```
mainK :: Answer
```

This should be bound to the final answer.

Here is the program which echoes a line of its input to its output, implemented with continuations.

```
echoK  :: Answer
echoK  =  getcK (\c->
          if (c == '\n') then
            doneK
          else
            putcK c (
            echoK))
```

This is remarkably close to the monad style, except that all appearances of `>>=` and `>>` have been elided. More precisely, they have been built-in to the corresponding primitives: where before one wrote `getc >>= k` now one writes `getcK k`, and where before one wrote `putc c >> k` now one writes `putcK c k`.

However, there is an essential difference between the monad and continuation styles. As we saw before, with monads one may simply write

```
main :: IO ()
main =  echo >> echo
```

to echo two lines. There is no equivalent form of composition that works for the echo program above. But, unlike with synchronous streams, there is an easy fix. Just rewrite the echo program so that, like the primitives, it too accepts a continuation.

```
echoK'    :: Answer -> Answer
echoK' k  =  getcK (\c->
             if (c == '\n') then
               k
             else
               putcK c (
               echoK' k))
```

The new program accepts a continuation `k`, which appears where `doneK` appeared formerly. Now one may simply write

```
mainK :: Answer
mainK =  echoK' (echoK' doneK)
```

to echo two lines from the input to the output.

The moral: with streams, it is difficult to write a version of echo that composes; with continuations, it is easy to write a version of echo that composes, if you remember to include a continuation argument; and with monads, it is *impossible* to write a version of echo that does *not* compose.

What is the trade off between monads and continuations? The program structures used are almost identical, so in that sense there is little to choose between them. Monads have the adavantage of being slightly more abstract, in that code remains uncluttered by mention of a continuation variable. Continuations have the advantage of directly supporting error jumps and other changes in flow of control. However, error jumps can also be supported by monads, as in the current version of Haskell [49], and monads can support the full power of continuations by building in a 'call with current continuation' operation, as described in [61].

Nonetheless, monads and continuations support very similar program structures. Perhaps the most remarkable comparison between monads and continuations is that the former has so far outstripped the latter in terms of popularity, when the underlying concepts are so similar.

From continuations to monads. We now consider how to define monads in terms of continuations. The type `IO a` stands for a function that accepts a continuation, which accepts a value of type `a` and yields an answer, and itself yields an answer.

```
type  IO a  =  (a -> Answer) -> Answer
```

A typical use of the computation `m :: IO a` has the form `m k :: Answer`, where `k :: a -> Answer` is the continuation. Here `m k` denotes an action that first behaves as specified by `m`, yielding the value `x`, and then behaves as specified by `k x`.

Given this formulation, it is straightforward to define monads in terms of continuations.

```
(>>=)      :: IO a -> (a -> IO b) -> IO b
m >>= k    =  \j-> m (\x-> k x (\y -> j y))

return     :: a -> IO a
return x   =  \j-> j x

putc       :: Char -> IO ()
putc c     =  \j-> putcK c (j ())

getc       :: IO Char
getc       =  \j-> getcK j
```

In each case, the types work out. For instance, in the definition of `(>>=)` we have `m :: (a->Answer)->Answer`, `k :: a->(b->Answer)->Answer`, `j :: b->Answer`, `x :: a`, and `y :: b`. Hence, `(\x-> k x (\y -> j y)) :: a->Answer`, as required. Observe that `k` and `j` both behave somewhat like continuations, but at different levels.

It is equally straightforward to relate the distinguished variables for continuations and monads.

```
mainK  ::  Answer
mainK  =   main (\()-> doneK)
```

Here `main :: IO ()` takes the continuation `(\()-> doneK) :: ()->Answer`, which accepts the trivial value and does nothing more.

Laws. Again, it is a straightforward exercise to show that the three monad laws are satisfied. For instance, we show

```
return x >>= k   =   k x
```

by the following calculation.

```
  (return x >>= k)
=       { definition >>= }
  \j-> return x (\x-> k x j)
=       { definition return }
  \j-> (\j-> j x) (\x-> k x j)
=       { simplify }
  \j-> (\x-> k x j) x
=       { simplify }
  \j-> k x j
=       { simplify }
  k x
```

The other two laws are proved similarly.

From monads to continuations. It is also easy to define continuations in terms of monads, so in this sense the models are equivalent. For details and further discussion, see [61].

3.3 Interaction by linear logic

Here is a naive model of interaction, based on state: an interactive program is represented by a function from the initial state to the world to the final state of the world. Each interaction is represented by a function that takes a current state and a request, and returns the next state and a response.

The difficulty here is that the same current state may be passed to two different invocations of the interaction function, yielding two different next states. In this case, which of the two interactions has actually occurred?

One possible solution is to wait until the final state is returned, as it may encode the series of interactions that produced it. This solution was considered for an early version of Haskell, but rejected, in part because it excludes the useful class of interactive programs that run forever, never yielding a final state.

A different solution is to guarantee that the current state is never duplicated. Hence, each state is passed to at most one function representing an interaction. Since there is no confusion as to which interaction occurs, each interaction may be performed as the program executes. Even a program that runs forever will produce a well-defined sequence of interactions.

Linear logic, as proposed by Girard [12], is a logic in which some propositions may not be duplicated in a proof. Via the Curry-Howard isomorphism, a logic corresponds to a programming language, with proofs corresponding to programs and propositions corresponding to types [17]. Hence, linear logic gives rise to a programming language with types that prohibit duplication. Tutorial explanations of this correspondence have been written by Abramsky [1] and Wadler [63]. Various systems, more or less practical, and based on linear logic to a greater or less degree, have been proposed by Holmstrom [16], Lafont [30], Wadler [60], Guzman and Hudak [15], and Barendsen and Smetsers [3]. The theoretical application to interaction was stressed by Girard and further elaborated by Lafont, but the first suggestion of practical application to interaction appears to be in my own work.

The first practical application of this idea is in the lazy functional language Clean [2], based on the work of Barendsen and Smetsers [3]. The Clean system has been used to program a number of impressive applications, with an interface that offers (at a high level) the same power as the Macintosh graphics toolkit.

Review of interaction by linear logic. In the linear logic model, there is a distinguished type `World`, representing the entire state of the external world. Values of this distinguished type must be treated linearly, that is, they may not be duplicated. Each interactive operation accepts an argument this type, representing the state at the beginning of the operation, and returns a result of this type, representing the state at the end of the operation.

There are two primitives, one to write a character to the output and one to read a character from the input.

```
putcL :: Char -> *World -> *World
getcL :: *World -> *(Char,*World)
```

Following the notation used in Clean, we preface linear types with a star. Values of type `*World` may not be duplicated, nor may the pair of type `*(Char,*World)`, since duplicating the pair would duplicate its second component. But it is permitted to extract and duplicate the first component of this pair.

This is a greatly simplified version of the actual type system used in Clean. Further complications arise because Clean requires that interactive functions should be strict in the argument representing the world, and because it is sometimes necessary to parameterise over whether a given type is linear or not, so that the same function can operate, for instance, on both linear and non-linear pairs.

The behaviour of the entire program is specified by the distinguished variable `mainL`.

```
mainL :: *World -> *World
```

This variable should be bound to a function from the state of the world at the beginning of the programs to the state of the world at the end.

Here is the program which echoes a line of its input to its output, implemented with linear state.

```
echoL    :: *World -> *World
echoL w  =  let (c,w') = getcL w in
            if (c == '\n') then
              w'
            else
              let w'' = putcL c w' in
              echo w''
```

This program is similar in structure to the monad program, though decorated throughout with variables denoting the current state of the world, `w`, `w'`, and `w''`.

Accidentally switching, say, an occurence of `w'` with one of `w''`, might drastically change the meaning of the program. Fortunately, many such errors will be caught by the linear type system.

It remains relatively easy to compose programs. One may simply write

```
mainL    :: *World -> *World
mainL w  =  echoL (echoL w)
```

to echo two lines.

What is the trade off between monads and linear logic? Linear logic requires a sophisticated type system, and forces the code to be cluttered by passing around the current state. We show below how to define monads in terms of linear state. Some users of Clean have found it convenient to make just such definitions and thereafter work in terms of monads, as this eliminates the clutter [26].

But while mentioning the state explicitly is something of a pain when there is just one state, it may become a boon if one fragments the state into separate components representing portions of the world that do not interact – for instance, one to represent the state of the screen, and a different one to represent the state of the file store. Further practical experience is needed to determine where the balance lies.

From linear state to monads. We now consider how to define monads in terms of linear state. The type `IO a` stands for a function that accepts the current state of the world, and returns a value of type `a` and a new state.

```
type  IO a = *World -> *(a, *World)
```

Given this formulation, it is straightforward to define monads in terms of linear state.

```
(>>=)      :: IO a -> (a -> IO b) -> IO b
m >>= f    =  \w-> let
                     (x,w')  = m w
                     (y,w'') = k x w'
                   in
                     (y,w'')

return     :: a -> IO a
return x   =  \w-> (x,w)

putc       :: Char -> IO ()
putc c     =  \w-> ((), putcL c)

getc       :: IO Char
getc       =  \w-> getcL w
```

It is equally straightforward to relate the distinguished variables for continuations and monads.

```
mainL  ::  Answer
mainL  =   \w-> let ((),w') = main w in w'
```

Here `main :: IO ()` takes the initial world `w` and returns a pair consisting of the trivial value `()` and the final world `w'`.

Laws. Again, it is a straightforward exercise to show that the three monad laws are satisfied. For instance, we show

```
return x >>= f  =  f x
```

by the following calculation.

```
  (return x >>= f)
=       { definition >>= }
  \w-> let
         (x',w') = return x w
         (y,w'') = f x' w'
       in
         (y,w'')
=       { definition return }
  \w-> let
         (x',w') = (x,w)
         (y,w'') = f x' w'
       in
         (y,w'')
=       { simplify }
  \w-> let
         (y,w'') = f x w
       in
         (y,w'')
=       { simplify }
  \w-> f x w
=       { simplify }
  f x
```

The other two laws are proved similarly.

From monads to linear state. There is no obvious way to make the converse definition, of linear state in terms of monads.

3.4 Interaction by side effect

Traditionally, in strict languages interaction occurs via side effects. This idea has roots at least as far back as Lisp and Iswim, and is carried on in Scheme and SML.

For this section, we switch our presentation language from Haskell to SML. There are some minor syntactic differences between Haskell and SML, some of which are indicated in the following.

Haskell	SML
`() :: ()`	`() : unit`
`'\n' :: Char`	`#"\n" : char`
`(\x->x) :: a->a`	`(fn x=>x) : 'a -> 'a`

Type variables in Haskell are distinguished by beginning with a small letter and type constructors precede their arguments, as in `IO a`, while type variables in SML are distinguished by begining with a backquote and type constructors follow their arguments, so the same type might be written `'a io` in SML. Types are indicated with two colons in Haskell and with one in SML. In SML, type signatures are preceded by the keyword `val`, and definitions are preceded by the keyword `val` or `fun`. We maul SML slightly by placing type signatures next to the corresponding definitions; in SML proper, definitions are grouped into modules and the signature appears separately.

Review of side effects. Again we assume two primitives, one to write a character to the output and one to read a character from the input. Their types are pleasingly symmetric.

```
val putcML : char -> unit
val getcML : unit -> char
```

Each primitive must be a function, and the desired interaction occurs *when the function is applied*. The type `unit` appears when a function is required to mediate the time at which a side effect occurs, but no actual argument or result is necessary.

The phrase "*when the function is applied*" matches our previous phrase "*when the action is performed*". For instance, evaluating the lambda abstraction

```
(fn c => putcML c; putcML c)
```

has no side effects, and returns a function, let's call it `f`, of type `char -> unit`. It is only when this function is applied that the side effects occur, so evaluating `f #"!"` prints two exclamation marks. Thus, our previous distinction between *thinking* and *doing* is here matched by a distinction between *abstraction* and *application*. The main difference is `main`: with side effects, unlike any of the other models we have studied, there is no need for a distinguished top-level variable.

Here is the program which echoes a line of its input to its output, implemented with side effects.

```
val echoML    : unit -> unit
fun echoML () = let val c = getcML () in
                if c = #"\n" then
                  ()
                else
                  (putcML c; echoML ())
                end
```

Neither the argument nor result of this function contain any information; it is executed solely for its side effects.

Two lines may be echoed by executing the following code.

```
echoML (); echoML ()
```

There is a world of difference between the value `echoML` which has no side effects when evaluated, and the computation `echoML ()`, which does.

For completists, here is how to define `putcML` and `getcML` in terms of the primitives provided in the Standard ML library.

```
fun putcML c =
  TextIO.output1(TextIO.stdOut,c);
```

```
fun getcML () =
  valOf(TextIO.input1(TextIO.stdIn));
```

From side effects to monads. Even in a strict language with side effects, it is still possible to encapsulate interaction within a monad. Thus, it is entirely possible to intermix the side effect and monad approaches to interaction.

The type `'a io` is represented by a function expecting a dummy argument of type `unit` and returning a value of type `'a`.

```
type 'a io  =  unit -> 'a
```

Here we exploit the fact that wrapping an expression inside a function allows us to control the point at which any side effects of that expression will occur.

Given this formulation, it is straightforward to define monads in terms of side effects.

```
infix >>=
val >>=      : 'a io * ('a -> 'b io) -> 'b io
fun m >>= k  = fn () => let
                          val x = m ()
                          val y = k x ()
                        in
                          y
                        end

val return   : 'a -> 'a io
fun return x = fn () => x

val putc     : char -> unit io
fun putc c   = fn () => putcML c

val getc     : char io
val getc     = fn () => getcML ()
```

The infix symbol `>>=` is curried in Haskell but takes a pair in SML, hence the first `->` in its Haskell type becomes `*` in its SML type.

As in the Haskell formulation, `>>` and `done` may be defined as special case of `>>=` and `return`.

```
infix >>
val >>      : unit io * unit io -> unit io
fun m >> n  = m >>= (fn () => n)

val done    : unit io
val done    = return ()
```

SML only allows recursive definition of functions. In order to treat `'a io` as an abstract type, we define a fixpoint operator, `fix`.

```
val fix     : ('a io -> 'a io) -> 'a io
fun fix h   = let fun f () = h f () in f end
```

The definition depends on the fact that `'a io` is the same as `unit -> 'a`, but once the function has been defined it may be used in a scope where `'a io` is taken as an abstract type.

Finally, we need a function to take on the same role as the distinguished variable, a pineal gland to convert thought into action. Its definition is simplicity itself.

```
val execute   : unit io -> unit
fun execute m = m ()
```

As an example of the use of these functions, here is `echo` rewritten in SML.

```
val echo  :  unit io
val echo  =  fix (fn echo =>
               getc >>= (fn c =>
                 if (c = #"\n") then
                   done
                 else
                   putc c >>
                   echo))
```

Apart from the explicit use of a fixpoint operator and a few minor syntactic differences, this is identical to the Haskell code.

Laws. Again, we ask whether the three monad laws follow from the definitions of `return` and `>>=` given above.

```
return v >>=fn x=> m   =  m[x:=v]
m >>=fn x=> return x   =  m
(m >>=fn x=> n) >>=fn y=> o   =
         m >>=fn x=>(n >>=fn y=> o)
```

The second and third laws can indeed be shown valid, for any expressions `m`, `n`, and `o`. But the first law holds only if both `v` and `m[x:=v]` are values. The first restriction is not too surprising, since the law

```
(fn x => m)v = m[x:=v]
```

also holds in SML only if `v` is a value. But the second restriction *is* surprising. Strict langauges (like SML) do restrict reasoning in ways that lazy languages (like Haskell) do not.

The usual call-by-value calculus λ_v of Plotkin [51] is not strong enough to prove these laws. One must use the stronger computational lambda calculus λ_c of Moggi [42], which has been studied by Sabry and Wadler [54].

4. RELATED WORK

Monads have been used for a variety of purposes beyond those described here.

As noted, Eugenio Moggi introduced monads to computing science as a way of structuring denotational semantics [42; 43]. Many different language features, including non-termination, state, exceptions, continuations, and interaction, can be viewed as monads. Independently of Moggi, but at about the same time, Michael Spivey noted that monads provided a useful way of structuring exception handling in functional programs [55]. Inspired by Moggi and Spivey, I proposed monads as a general technique for functional programming [59; 61; 62].

As we have seen, monads are used to structure interaction in Haskell, and they also provide interaction with C and mutable state in Glasgow's local extension of Haskell [50]. Monads are also used to structure interaction in the declarative language Escher [37].

Monads are also used to structure the Glasgow Haskell compiler, which is itself written in Haskell [61; 20]. Each phase of the compiler uses a monad for bookkeeping information. For instance, the type checker uses a monad that combines state (to maintain a current substitution), a name supply (for fresh type variable names), and exceptions (to report type errors). If additional bookkeeping information is required, it is easy to change the monad without requiring extensive modification to the rest of the program. For instance, the type checker was easily altered to maintain information about the current line number, which enabled better error messages.

The use of monads for updateable state is described by Wadler [59], and for input-output is described by Peyton Jones and Wadler [50]. Monads for updateable state have been further elaborated by Launchbury [32] and Launchbury and Peyton Jones [34], and applied to to functional graph algorithms by King and Launchbury [28] and Launchbury [33]. Monads for interaction have been extended to include concurrency by Peyton Jones, Gordon, and Finne [48], and applied to user interface design by Peyton Jones and Finne [47].

The use of monads to structure interpreters and evaluators is described by Wadler [61; 62], Steele [56] and Liang, Hudak, and Jones [36]. Additional structuring techniques based on monads are described by Meijer and Jeuring [39].

There are standard call-by-value and call-by-name translations of lambda calculus into continuation passing style [51]. Monads provide a generalisation of these translations. The relation of monads to continuation passing style have been described by Moggi [42; 43], Wadler [59; 61; 64], Hatcliff and Danvy [19], Filinski [10], and Sabry and Wadler [54]. Filinski also describes an ingenious way to embed arbitrary monads in a call-by-value language with state and continuations, such as SML/NJ [10].

Researchers have proposed various special type systems and syntaxes to support monads. Jones devised an overloaded type system suitable for use with monads [23; 25], and has implemented this system in Gofer [24]. Wadler proposed a notation for monads based on an analogy with list comprehensions [59], and Jones proposed a `do` notation that bears a remarkable resemblance to C [25]; both of these notations are implemented in Gofer, and have been incorporated in Haskell 1.3 [49].

In addition to the three general-purpose monad laws, one requires specific laws to reason about specific effects. Laws to reason about monads that manipulate state are given by Wadler [59], Odersky, Rabin and Hudak [44], Launchbury [33], and Sabry and Launchbury [35]. A formal link between monads and linear state is drawn by Chen and Hudak [8]. Hughes [18] uses monads to illustrate a clever technique for deriving an efficient representation of a data type from an algebraic specification.

Each monad incorporates a different effect, such as input-output, state, or exceptions. So it is important to consider ways in which monads can be combined. This is discussed by Barr and Wells [4], Moggi [42; 43], King and Wadler [29], Jones and Duponcheel [27], Liang, Hudak, and Jones [36], and Jones [25].

5. CONCLUSIONS

Interaction is becoming increasingly important, and declarative languages must develop suitable methods of incorporating interaction. Monads provide one such approach, and are becoming widely adopted within parts of the declarative programming community.

It is a sign of the increasing maturity of computing science that it is no longer acceptable for a programming language to work in isolation. Programmers no longer work from scratch, but assemble systems by combining existing components. Reuse is essential. A language must provide connections to databases, networks, and graphics. We are forced to find ways to start standing on our colleagues's shoulders, and to stop standing on their toes.

Thus it is becoming increasingly important for different languages, and different language paradigms, to communicate. The meagre facility to integrate Haskell with C described here is a start in this direction. Research trends within the community point to further integration: hot topics include how to pass more complex data structures between C and Haskell; how to integrate storage management; how to add concurrency; and better support for graphic user interfaces. Monads have a useful role to play in these developments.

Having praised monads to the hilt, let me level one criticism. Monads tend to be an all-or-nothing proposition. If you discover that you need interaction deep within your program, you must rewrite that segment to use a monad. If you discover that you need two sorts of interaction, you tend to make a single monad support both sorts. It seems to me that instead we should be able to move smoothly from no monads (no interactions) to one monad (a single form of interaction) to many monads (several independent forms of interactions). How to achieve this remains a challenge for the future.

5.1 First-order versus higher-order

In my first paper on monads I wrote: "the higher-order nature of the solution means it cannot be applied in first-order languages such as Prolog" [59]. Certainly, the monad combinator `>>=` is higher-order. However, I have come to believe that my assertion is misleading.

From the beginning, Moggi took pains to stress that monads could be applied to a first-order language, independent of the machinery required for a higher-order language [42]. What was clear to Moggi from the start has become clear to me at last. Just as the higher-order `(\x-> n) m` mimics the first-order `let x=m in n`, so the higher-order `m >>= \x-> n` mimics a monadic `let` construct that is essentially first-order. We've already suggested that it be written `let x <- m in n`.

Thus, there is no problem in adding monads to a first-order typed language. Just add a type constructor `IO`, and add language constructs for `return v` and `let x <- m in n` (where `v`, `m`, `n` are terms, and `x` is a variable). The type rules follow directly.

$$\frac{\vdash \texttt{v :: a}}{\vdash \texttt{return v :: IO a}} \qquad \frac{\vdash \texttt{m :: IO a} \quad \texttt{x :: a} \vdash \texttt{n :: IO b}}{\vdash \texttt{let x <- m in n :: IO b}}$$

A "pineal gland" is required: as with `main` in Haskell, some value of `IO` type must be designated to represent the effect of the program. All this works as well in an untyped language, save that static checks at compile-time may need to be replaced by dynamic checks at run-time.

Nonetheless, higher-order languages have an advantage over first-order languages. In a higher-order language, one can encode a new binding construct simply by adding a new constant. This observation goes back to Church [6], who modelled universal quantification $\forall x.A$ by adding a constant Π and writing $\Pi\,(\lambda x.A)$. Similarly, I modelled `let x <- m in n` by adding a constant `>>=` and writing `m >>= \x-> n`. In a first-order language, such encodings are not available: the only way to add a new binding construct to the language is to add a new binding construct to the language.

Eminent declarative programmers, such as Warren [66] and Goguen [13], have claimed that higher-order languages offer no essential advantages over first-order languages. The preceding paragraph shows why I reject this claim.

The higher-order nature of Haskell made it easy to experiment with monads. But now that higher-order functional programmers have done the experiment, first-order logic programmers may wish to consider extending their languages with monads. Working out the details makes a fascinating challenge.

5.2 Backtracking

If one considers adding monads for input-output to a logic programming language, there is one final hitch. How should interaction interact with backtracking? The mind can easily reverse a thought, but the body has more difficulty undoing an action.

In functional languages, backtracking is often modelled by considering a list (or set) of possible solutions; and if the language is lazy, the operational behaviour is much the same as with backtracking [58]. This use of lists (or sets) was one of the motivating examples for monads [42; 55; 59]. And just as parsing is of special interest to logic programmers as an application of backtracking, so too is parsing of special interest to functional programmers as an application of lists and monads [11; 45; 58; 59; 62].

Monads offer insights into interaction. Do they also offer insights into backtracking? Or into the relation between the two?

Acknowledgements

I thank John Lloyd and the program committee of ILPS'95 for the opportunity to present this material, and I thank Andrew Black, Franklin Chen, John Lloyd, Harald Sondergaard, and the referees for comments on this paper.

REFERENCES

[1] S. Abramsky, Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Peter Achten and Rinus Plasmeijer. The ins and outs of Clean I/O. Journal of Functional Programming, 5(1):81-110, January 1995.

[3] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). Proceeding of the 13'th conference on the Foundations of Software Technology and Theoretical Computer Science, Bombay, India, 1993.

[4] M. Barr and C. Wells, *Toposes, Triples, and Theories.* Springer Verlag, 1985.

[5] R. Bird and P. Wadler, *Introduction to Functional Programming.* Prentice Hall, 1987.

[6] A. Church, A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[7] J. Cupitt, A brief walk through KAOS. Technical Report 52, Computing Laboratory, University of Kent at Canterbury, 1989.

[8] C.-P. Chen and P. Hudak, Rolling your own mutable ADT — a connection between linear types and monads. In *24th Symposium on Principles of Programming Languages*, Paris, France, ACM, January 1997.

[9] D. Dennett, *Consciousness Explained*, Little, Brown, and Company, 1991.

[10] A. Filinski, Representing monads. In *21st Symposium on Principles of Programming Languages*, Portland, Oregon, ACM, January 1994.

[11] J. Fokker, Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[12] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[13] J. Goguen, Higher-order functions considered unnecessary for higher-order programming. In D. Turner, editor, *Research Topics in Functional Programming*, Addison Wesley, 1990.

[14] A. Gordon, *Functional programming and input/output.* Distinguished dissertations in computer science, Cambridge University Press, 1993.

[15] Juan Guzman and Paul Hudak, Single-threaded polymorphic lambda calculus. 5'th IEEE Symposium on Logic in Computer Science, Philadelphia, June 1990.

[16] S. Holmström, A linear functional language. Proceedings of the workshop on Implementation of Lazy Functional Languages, Prgramming Methodology Group report 53, Chalmers University of Technology, September 1988.

[17] W. A. Howard, The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980. (The original version was circulated privately in 1969.)

[18] J. Hughes, The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[19] J. Hatcliff and O. Danvy, A generic account of continuation-passing styles. In *21st Symposium on Principles of Programming Languages*, Portland, Oregon, ACM, January 1994.

[20] C. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler, The Glasgow Haskell compiler: a retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Ayr, Scotland, Springer Verlag Workshops in Computing Series, 134–143, July 1992.

[21] P. Hudak, S. Peyton Jones and P. Wadler, editors, Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.2. *Sigplan Notices* 27(5), May 1992.

[22] P. Hudak and R. S. Sundaresh, On the expressiveness of purely functional I/O systems. Technical report YALEU/DCS/RR-665, Yale University Department of Computer Science, March 1989.

[23] M. P. Jones, A system of constructor classes: overloading and implicit higher-order polymorphism. In *Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, ACM, June 1993.

[24] M. P. Jones, Gofer 2.30 implementation, 1994. Available by ftp from `ftp.cs.nott.ac.uk/nott-fp/languages/gofer`.

[25] M. P. Jones, Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[26] S. B. Jones, Experiences with Clean I/O, Proceedings of the Glasgow Workshop on Functional Programming, Ullapool, Scotland, Electronic Workshops in Computing, Springer-Verlag, July 1995.

[27] M. P. Jones and L. Duponcheel, Composing monads. Research report YALE/DCS/RR-1004, Yale University, New Haven, Connecticut, December 1993.

[28] D. King and J. Launchbury, Structuring depth-first search algorithms in Haskell. In *22nd Symposium on Principles of Programming Languages*, San Francisco, California, ACM, January 1995.

[29] D. King and P. Wadler, Combining monads. In *Glasgow Workshop on Functional Programming*, Ayr, Scotland, Workshops in Computing Series, Springer Verlag, July 1992.

[30] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[31] P. J. Landin, A correspondence between ALGOL 60 and Church's lambda notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101,158–165, February and March 1965.

[32] J. Launchbury, Lazy imperative programming. In *Workshop on State in Programming Languages*, Copenhagen, Denmark, ACM, 1993.

[33] J. Launchbury, Graph algorithms with a functional flavour. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[34] J. Launchbury and S. L. Peyton Jones, Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, Orlando, Florida, ACM, 1994.

[35] J. Launchbury and A. Sabry, Monadic State: Axiomatization and Type Safety (ICFP 1997) In *2nd International Conference on Functional Programming*, Amsterdam, ACM, July 1997.

[36] S. Liang, P. Hudak, and M. P. Jones, Monad transformers and modular interpreters. In *22nd Symposium on Principles of Programming Languages*, San Francisco, California, ACM, January 1995.

[37] J. W. Lloyd, Declarative programming in Escher. Technical report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

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

[39] E. Meijer and J. Jeuring, Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[40] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. MIT Press, 1990.

[41] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[42] E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California, IEEE, June 1989.

[43] E. Moggi, Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[44] M. Odersky, D. Rabin, and P. Hudak, Call-by-name, assignment, and the lambda calculus. In *20th Symposium on Principles of Programming Languages*, Charleston, South Carolina, ACM, January 1993.

[45] L. C. Paulson, *ML for the Working Programmer*. Cambridge University Press, 1991.

[46] N. Perry, I/O and Inter-language Calling for Functional Languages, Proceedings 9'th International Conference of the Chilean Computer Society and 15'th Latin American Conference on Informatics, Chile, 1989. Also available as `ftp://smis-asterix/pub/ResearchPapers/FL_IO_IL_Chile_Jul89.ps.Z`.

[47] S. L. Peyton Jones and S. Finne, Composing Haggis. Manuscript, 1995.

[48] S. L. Peyton Jones, A. Gordon, and S. Finne, Concurrent Haskell. In *23rd Symposium on Principles of Programming Languages*, St Petersburg, Florida, ACM, January 1996.

[49] John Peterson, and Kevin Hammond, editors, Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.3. Technical report, Yale University, May 1996.

[50] S. L. Peyton Jones and P. Wadler, Imperative functional programming. In *20th Symposium on Principles of Programming Languages*, Charleston, South Carolina, ACM, January 1993.

[51] G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[52] J. C. Reynolds, The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[53] D. A. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, 7:299–310, 1985.

[54] A. Sabry and P. Wadler, A reflection on call-by-value. In *1st International Conference on Functional Programming*, ACM Press, Philadelphia, May 1996.

[55] M. Spivey, A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.

[56] G. L. Steele, Jr., Building interpreters by composing monads. In *21st Symposium on Principles of Programming Languages*, Portland, Oregon, ACM, January 1994.

[57] W. Stoye, Message-based functional operating systems. *Science of Computer Programming*, 6(3):291–311, 1986.

[58] P. Wadler, How to replace failure by a list of successes. *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, LNCS 201, Springer-Verlag, September 1985.

[59] P. Wadler, Comprehending monads. In *Conference on Lisp and Functional Programming*, Nice, France, ACM, June 1990.

[60] P. Wadler, Linear types can change the world! In M. Broy and C. Jones, editors, Programming Concepts and Methods, North Holland, Sea of Galilee, Israel, April 1990.

[61] P. Wadler, The essence of functional programming (invited talk). In *19th Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, ACM, January 1992.

[62] P. Wadler, Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, Springer Verlag, 1993. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[63] P. Wadler, A taste of linear logic (invited talk). In *Mathematical Foundations of Computing Science*, Gdansk, Poland, LNCS 711, Springer Verlag, August 1993.

[64] P. Wadler, Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–55, January 1994.

[65] P. Wadler, How to declare an imperative. In John Lloyd, editor, *International Logic Programming Symposium*, MIT Press, December 1995.

[66] D. H. D. Warren, Higher-order extensions to Prolog — are they needed? In D. Michie, *et al.*, editors, *Machine Intelligence 10*, Ellis Horwood, 1981.