Imperative functional programming

Simon L Peyton Jones  Philip Wadler

Dept of Computing Science, University of Glasgow
Email: \{simonpj, wadler\}@dcs.gla.ac.uk

October 1992

This paper appears in
ACM Symposium on Principles Of Programming Languages (POPL), Charleston, Jan 1993, pp.71-84. This copy corrects a few minor typographical errors in the published version.

Abstract

We present a new model, based on monads, for performing input/output in a non-strict, purely functional language. It is composable, extensible, efficient, requires no extensions to the type system, and extends smoothly to incorporate mixed-language working and in-place array updates.

1 Introduction

Input/output has always appeared to be one of the less satisfactory features of purely functional languages: fitting action into the functional paradigm feels like fitting a square block into a round hole. Closely related difficulties are associated with performing in-place update operations on arrays, and calling arbitrary procedures written in some other (possibly side-effecting) language.

Some mostly-functional languages, such as Lisp or SML, deal successfully with input/output by using side effects. We focus on purely-functional solutions, which rule out side effects, for two reasons. Firstly, the absence of side effects permits unrestricted use of equational reasoning and program transformation. Secondly, we are interested in non-strict languages, in which the order of evaluation (and hence the order of any side effects) is deliberately unspecified; laziness and side effect are fundamentally inimical.

There is no shortage of proposals for input/output in lazy functional languages, some of which we survey later, but no one solution has become accepted as the consensus.

This paper outlines a new approach based on monads (Moggi [1989]; Wadler [1992]; Wadler [1990]), with a number of noteworthy features.

- It is composable. Large programs which engage in I/O are constructed by gluing together smaller programs that do so (Section 2). Combined with higher-order functions and lazy evaluation, this gives a highly expressive medium in which to express I/O-performing computations (Section 2.2) — quite the reverse of the sentiment with which we began this section.

We compare the monadic approach to I/O with other standard approaches: dialogues and continuations (Section 3), and effect systems and linear types (Section 7).

- It is easily extensible. The key to our implementation is to extend Haskell with a single form that allows one to call any procedure written in the programming language C (Kernighan & Ritchie [1978]), without losing referential transparency (Section 2.3). Using it programmers can readily extend the power of the I/O system, by writing Haskell functions which call operating system procedures.

- It is efficient. Our Haskell compiler has C as its target code. Given a Haskell program performing an I/O loop, the compiler can produce C code which is very similar to that which one would write by hand (Section 4).

- Its efficiency is achieved by applying simple program transformations. We use unboxed data types (Peyton Jones & Launchbury [1991]) to expose representation and order-of-evaluation detail to code-improving transformations, rather than relying on ad hoc optimisations in the code generator (Section 4.1).

- It extends uniformly to provide interleaved I/O and reference types (Section 5).

- It extends uniformly to support incremental arrays with in-place update (Section 6). Our implementation is efficient enough that we can define monolithic
Haskell array operations in terms of incremental arrays. Hudak have proposed a similar method based on continuations. Our method is more general than his in the following sense: monads can implement continuations, but not the converse.

- **It is based (only) on the Hindley-Milner type system.** Some other proposals require linear types or existential types; ours does not.

We have implemented all that we describe in the context of a compiler for Haskell [Hudak et al. [1992]], with the exception of the extension to arrays and reference types. The entire I/O system provided by our compiler is written in Haskell, using the non-standard extensions we describe below. The language’s standard Dialogue interface for I/O is supported by providing a function to convert a Dialogue into our IO monad. The system is freely available by FTP.

We do not claim any fundamental expressiveness or efficiency which is not obtainable through existing systems, except where arrays are concerned. Nevertheless we feel that the entire system works particularly smoothly as a whole, from the standpoint of both programmer and implementor.

### 2 Overview

We need a way to reconcile being with doing: an expression in a functional language denotes a value, while an I/O command should perform an action. We integrate these worlds by providing a type IO a denoting actions that, when performed, may do some I/O and then return a value of type a. The following provide simple Unix-flavoured I/O operations.

```plaintext
getcIO :: IO Char
putcIO :: Char -> IO ()
```

Here getcIO is an action which, when performed, reads a character from the standard input, and returns that character; and putcIO a is an action which, when performed, writes the character a to the standard output. Actions which have nothing interesting to return, such as putcIO, return the empty tuple (()), whose type is also written ()

Notice the distinction between an action and its performance. Think of an action as a “script”, which is performed by executing it. Actions themselves are first-class citizens. How, then, are actions performed? In our system, the value of the entire program is a single (perhaps large) action, called mainIO, and the program is executed by performing this action. For example, the following is a legal Haskell program.

```haskell
mainIO :: IO ()
mainIO = putcIO '!'
```

This is the point at which being is converted to doing: when executed, the putcIO action will be performed, and write an exclamation mark to the standard output.

#### 2.1 Composing I/O operations

The functions defined above allow one to define a single action, but how can actions be combined? For example, how can we write a program to print two exclamation marks? To do so, we introduce two “glue” combinators:

```haskell
doneIO :: IO ()
seqIO :: IO a -> IO b -> IO b
```

The compound action \( m \ `\text{seqIO}` \ n \) is performed, by first performing \( m \) and then performing \( n \), returning whatever \( n \) returns as the result of the compound action. (Backquotes are Haskell’s syntax for an infix operator.) The action doneIO does no I/O and returns the unit value, (()). To illustrate, here is an action putsIO, which puts a string to the standard output:

```haskell
putsIO :: [Char] -> IO ()
putsIO [] = doneIO
putsIO (a:as) = putcIO a `seqIO` putsIO as
```

We can now use putsIO to define a program which prints “hello” twice:

```haskell
mainIO = hello `seqIO` hello
  where
    hello = putsIO "hello"
```

This example illustrates the distinction between an action and its performance: hello is an action which happens to be performed twice. The program is precisely equivalent to one in which putsIO "hello" is substituted for either or both of the occurrences of hello. In short, programs remain referentially transparent.

In general, an action may also return a value. Again, there are two combinators. The first is again trivial:

```haskell
unitIO :: a -> IO a
```

If \( x \) is of type \( a \), then unitIO \( x \) denotes the action that, when performed, does nothing save return \( x \). The second combines two actions:

```haskell
bindIO :: IO a -> (a -> IO b) -> IO b
```

If \( m :: IO a \) and \( k :: a -> IO b \) then \( m \ `\text{bindIO}` \ k \) denotes the action that, when performed, behaves as fol-
lows: first perform action \( m \), yielding a value \( x \) of type \( a \), then perform action \( k x \), yielding a value \( y \) of type \( b \), and then return value \( y \). To illustrate, here is an action that echoes the standard input to the standard output. (In Haskell, \( \lambda x \rightarrow e \) stands for a lambda abstraction; the body of the abstraction extends as far as possible.)

\[
\begin{align*}
echo &:: \mathbb{I}O () \\
echo &\equiv \text{getcIO } 'bindIO' \ \lambda a \rightarrow \\
&\quad \begin{cases} \\
&\quad \text{if } (a == \text{eof}) \text{ then } \\
&\quad \quad \text{doneIO} \\
&\quad \quad \text{putcIO } a \ 'seqIO' \\
&\quad \quad \text{echo} \\
&\end{cases}
\end{align*}
\]

The combinators \( \text{bindIO} \) and \( \text{unitIO} \) are generalisations of \( \text{seqIO} \) and \( \text{doneIO} \). Here are definitions for the latter in terms of the former:

\[
\begin{align*}
\text{doneIO} &\equiv \text{unitIO } () \\
m \ 'seqIO' n &\equiv m \ 'bindIO' \ \lambda a \rightarrow n
\end{align*}
\]

The combinators have a useful algebra: \( \text{doneIO} \) and \( \text{seqIO} \) form a monoid, while \( \text{bindIO} \) and \( \text{unitIO} \) form a monad (Moggi [1989]; Wadler [1992]; Wadler [1990]).

2.2 Imperative programming

It will not have escaped the reader’s notice that programs written in the monadic style look rather similar to imperative programs. For example, the \texttt{echo} program in C might look something like this:

\[
\begin{align*}
\text{echo}(a) \{ \\
\quad \text{loop: } a = \text{getchar}(a); \\
\quad \quad \text{if } (a == \text{eof}) \\
\quad \quad \quad \text{return} ; \\
\quad \quad \text{else } \{ \text{putchar}(a); \\
\quad \quad \quad \text{goto loop}; \} \\
\}
\end{align*}
\]

(Indeed, as we discuss later, our compiler translates the \texttt{echo} function into essentially this C code.) Does the monadic style force one, in effect, to write a functional facsimile of an imperative program, thereby losing any advantages of writing in a functional language? We believe not.

Firstly, the style in which one writes the functional program’s internal computation is unaffected. For instance, the argument to \texttt{putsIO} can be computed using the usual list-processing operations provided by a functional language (list comprehensions, map, append, and the like).

Secondly, the power of higher-order functions and non-strict semantics can be used to make I/O programming easier, by defining new action-manipulating combinators. For example, the definition of \texttt{putsIO} given above uses explicit recursion. Here is an alternative way to write \texttt{putsIO} which does not do so:

\[
\text{putsIO } as = \text{seqsIO } (\text{map putcIO } as)
\]

The \texttt{map} applies \texttt{putcIO} to each character in the list \( as \) to produce a list of actions. The combinator \texttt{seqsIO} takes a list of actions and performs them in sequence; that is, it encapsulates the recursion. It is easy to define \texttt{seqsIO} thus:

\[
\begin{align*}
\text{seqsIO} &:: [\mathbb{I}O a] \rightarrow \mathbb{I}O () \\
\text{seqsIO } [] &\equiv \text{doneIO} \\
\text{seqsIO } (a:as) &\equiv a \ 'seqIO' \ \text{seqsIO } as \\
\end{align*}
\]

or even, using the standard list-processing function \texttt{foldr}, thus:

\[
\text{seqsIO } = \text{foldr seqIO } \text{doneIO}
\]

To take another example, here is a function which writes a given number of spaces to the standard output:

\[
\begin{align*}
\text{spaceIO} &:: \mathbb{I}nt \rightarrow \mathbb{I}O () \\
\text{spaceIO } n &\equiv \text{seqsIO } (\text{take } n \ (\text{repeat } (\text{putcIO } '\ ')))
\end{align*}
\]

The functions \texttt{take} and \texttt{repeat} are standard list-processing functions (with nothing to do with I/O) from Haskell’s standard prelude. The function \texttt{repeat} takes a value and returns an infinite list each of whose elements is the given value. The function \texttt{take} takes a prefix of given length from a list.

These necessarily small examples could easily be programmed with explicit recursion without significant loss of clarity (or even a gain!). The point we are making is that it is easy for the programmer to define new “glue” to combine actions in just the way which is suitable for the program being written. It’s a bit like being able to define your own control structures in an imperative language.

2.3 Calling C directly

Since the “primitive” functions \texttt{putcIO}, \texttt{getcIO}, and so on must ultimately be implemented by a call to the underlying operating system, it is natural to provide the ability to call any operating system function directly. To achieve this, we provide a new form of expression, the \texttt{ccall}, whose general form is:

\[
\text{ccall proc } e_1 \ldots e_n
\]

Here, \texttt{proc} is the name of a C procedure, and \( e_1, \ldots, e_n \) are the parameters to be passed to it. This expression
is an action, with type IO Int; when performed, it calls
the named procedure, and delivers its result as the value
of the action. Here, for example, are the definitions of
getcIO and putcIO:

\[
\begin{align*}
\text{putcIO } a & = \text{ccall putchar } a \\
\text{getcIO } & = \text{ccall getchar}
\end{align*}
\]

These calls directly invoke the system-provided func-
tions; no further runtime support is necessary. Using this
single primitive allows us to implement our entire I/O
system in Haskell.

We define ccall to be a language construct rather than
simply a function because:

- The first “argument” must be the literal name of the
  C procedures to be called, and not (say) an expression
  which evaluates to a string which is the name of
  the function. Type information alone cannot express
  this.

- Different C procedures take different numbers of
  arguments, and some take a variable number of
  arguments. (It would be possible to check the type-
  correctness of the C call by reading the signature of
  the C procedure, but we do not at present do so.)

- Different C procedures take arguments of different
  types and sizes. (At present, we only permit the
  arguments to be of base types, such as Char, Int,
  Float, Double and so on, though we are working on
  extensions which allow structured arguments to be
  built.)

Treating ccall as a construct allows these variations to
be accommodated without difficulty.

3 Comparison with other I/O styles

In this section we briefly compare our approach with two
other popular ones, dialogues and continuations.

3.1 Dialogues

The I/O system specified for the Haskell language (Hudak
et al. [1992]) is based on dialogues, also called lazy streams
(Dwelly [1989]; O’Donnell [1985]; Thompson [1989]). In
Haskell, the value of the program has type Dialogue, a
synonym for a function between a list of I/O responses to
a list of I/O requests:

\[
\begin{align*}
\text{type Dialogue } & = \text{[Response] } \rightarrow \text{[Request]} \\
\text{main } : : \text{Dialogue}
\end{align*}
\]

Request and Response are algebraic data types which
embodies all the possible I/O operations and their results,
respectively:

\[
\begin{align*}
\text{data Request } & = \text{Putc Char } | \text{Getc} \\
\text{data Response } & = \text{OK } | \text{OKCh Char}
\end{align*}
\]

(For the purposes of exposition we have grossly simpli-
ied these data types compared with those in standard
Haskell.) A system “wrapper program” repeatedly gets
the next request from the list of requests returned by
main, interprets and performs it, and attaches the re-
response to the end of the response list to which main is
applied.

Here, for example, is the echo program written using a
Dialogue (In Haskell xs!!n extracts the n'th element
from the list xs.)

\[
\begin{align*}
\text{echo } : : \text{Dialogue} \\
\text{echo resps } = \text{Getc :} \\
\quad \text{if } (a == \text{eof}) \\
\quad \text{then } [] \\
\quad \text{else Putc } a : \\
\quad \text{echo } (\text{drop } 2 \text{ resps}) \\
\text{where} \\
\quad \text{OKCh } a = \text{resps}!!1
\end{align*}
\]

The difficulties with this programming style are all too
obvious, and have been well rehearsed elsewhere (Perry
[1991]):

- It is easy to extract the wrong element of the re-
sponses, a synchronisation error. This may show up
in a variety of ways. If the “2” in the above program
was erroneously written as “4” the program would
fail with a pattern-matching error in getCharIO; if it
were written “3” it would deadlock.

- The Response data type has to contain a constructor
for every possible response to every request. Even
though Putc may only ever return a response OKChar,
the pattern-matching performed by get has to take
account of all these other responses.

- Even more seriously, the style is not composable: there
is no direct way to take two values of type
Dialogue and combine them to make a larger value
of type Dialogue (try it!).

Dialogues and the IO monad have equal expressive power,
as Figure 1 demonstrates, by using Dialogues to emu-
late the IO monad, and vice versa. The function dToIO
which emulates Dialogues in terms of IO is rather cu-
rious, because it involves applying the single dialogue
d to both bottom (⊥) and (later) to the “real” list
### Dialogue to IO

#### dToIO

```haskell
dToIO :: Dialogue -> IO ()
dToIO d = case (d bottom) of
  [] -> doneIO
  (q:qs) -> doReq q 'bindIO' \r ->
    dToIO (\rs -> tail (d (r:rs)))
```

底 :: a

bottom = error "Should never be evaluated"

#### doReq

```haskell
doReq :: Request -> IO Response
doReq (GetChar f) = getCharIO f 'bindIO'
  (\c ->
    unitIO (OKChar c))
doReq (PutChar f c) = putCharIO f c 'seqIO'
  unitIO OK
```

### IO to Dialogue

#### type IO a

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

#### ioToD

```haskell
ioToD :: IO () -> Dialogue
ioToD action = \rs -> case (action rs) of
  (_, qs, _) -> qs

unitIO v = \rs -> (v, [], rs)
bindIO op fop
  = \rs -> let (v1, qs1, rs1) = op rs
            (v2, qs2, rs2) = fop v1 rs1
             in (v2, qs1++qs2, rs2)
```

Figure 1: Converting between Dialogue and IO

of responses (Hudak & Sundaresh [1989]; Peyton Jones [1988]). This causes both duplicated work and a space leak, but no more efficient purely-functional emulation is known. The reverse function, `ioToD` does not suffer from these problems, and this asymmetry is the main reason that Dialogue is specified as primitive in Haskell. We return to this this matter in Section 5.3.

#### 3.2 Continuations

The continuation-style I/O model (Gordon [1989]; Hudak & Sundaresh [1989]; Karlsson [1982]; Perry [1991]) provides primitive I/O operations which take as one of their arguments a continuation which says what to do after the I/O operation is performed:

```haskell
main :: Result
putcC :: (Char -> Result) -> Result
doneC :: Result
```

Using these primitives, the `echo` program can be written as follows:

```haskell
echo :: Result -> Result
echo c = getcC (\a ->
  if (a == eof) then
    then c
  else putcC a (echo c))
```

Since we might want to do some more I/O after the echoing is completed, we must provide `echo` with a continuation, `c`, to express what to do when `echo` is finished. This "extra argument" is required for every I/O-performing function if it is to be composable, a pervasive and tiresome feature.

The above presentation of continuation-style I/O is a little different from those cited above. In all those descriptions, `Result` is an algebraic data type, with a constructor for each primitive I/O operation. As with Dialogue, execution is driven by a "wrapper" program, which evaluates `main`, performs the operation indicated by the constructor, and applies the continuation inside the constructor to the result. This approach has the disadvantage that it requires existential types if polymorphic operations, such as those we introduce later in Section 5.3, are to be supported.

An obvious improvement, which we have not seen previously suggested, is to implement the primitive continuation operations (such as `putcC`, `getcC` and `doneC`) directly, making the `Result` type an abstract data type with no operations defined on it other than the primitives themselves. This solves the problem.

Continuations are easily emulated by the `IO` monad, and vice versa, as Figure 2 shows. The comparison between the monadic and continuation approach is further explored in Section 6.

### 4 Implementing monadic I/O

So far we have shown that an entire I/O system can be expressed in terms of `ccall`, `bindIO`, and `unitIO`, and of course the `IO` type itself. How are these combinators to be implemented? One possibility is to build them in as primitives, but it turns out to be both simpler and more efficient to implement all except `ccall` in Haskell.

The idea is that an action of type `IO a` is implemented as a `function`, which takes as its input a value representing the entire current state of the world, and returns a pair, consisting of (a value representing) the new state of the
Continuations to IO

type Result = IO ()

cToIO :: Result -> IO ()
cToIO r = r

putCharC :: File -> Char -> Result -> Result
putCharC f c k = putCharIO f c 'seqIO' k

getCharC :: File -> Char
    -> (Char -> Result) -> Result
getCharC f k = getCharIO f 'thenIO' k

TO to continuations

type IO a = (a -> Result) -> Result

ioToC :: IO () -> Result
ioToC action = action (
    () -> nopC)

unitIO v = \k -> k v

bindIO op fop = \k a -> fop a k

putCharIO f c = \k -> putCharC f c (k ())

getcharIO f = \k -> getCharC f (c -> k c)

Figure 2: Converting between continuations and IO

world, and the result of type a:

type IO a = World -> IORes a
data IORes a = MkIORes a World

The type declaration introduces a type synonym for IO, and the auxiliary algebraic datatype IORes simply pairs
the result with the new world. Recall that the value of the entire program is of type IO (). The type World is
abstract, with only one operation defined on it, namely ccall. Conceptually, the program is executed by applying
main to a value of type World representing current
state of the world, extracting the resulting World value from the MkIORes constructor, and applying any changes
embodied therein to the real world.

If implemented literally, such a system would be unworkably expensive. The key to making it cheap is to ensure
that the world state is used in a single-threaded way, so that I/O operations can be applied immediately to the real
world. One way to ensure this would be to do a global analysis of the program. A much simpler way is to make
IO into an abstract data type which encapsulates the data
types IO and IORes, and the combinators bindIO and
unitIO. Here are suitable definitions for the latter:

unitIO a w = MkIORes a w
bindIO m k w = case (m w) of
    MkIORes a w' -> k a w'

Notice that bindIO and unitIO carefully avoid duplicating
the world. Provided that the primitive ccall actions are combined only with these combinators, we can
guarantee that the ccalls will be linked in a single linear
chain, connected by data dependencies in which each
call consumes the world state produced by the previous one. In turn this means that the ccall operations can
update the real world “in place”.

4.1 Implementing ccall

So much for the combinators. All that remains is the implementation of ccall. The only complication here is
that we must arrange to evaluate the arguments to the
call before passing them to C.

This is very similar to the argument evaluation required
for built-in functions such as addition, for which we have
earlier developed the idea of unboxed data types (Peyton
Jones & Launchbury [1991]). These allow representation
and order-of-evaluation information to be exposed to
code-improving transformations. For example, consider
the expression x+x where x is of type Int. The improvement
we want to express is that x need only be evaluated
once.

The key idea is to define the type Int (which is usually
primitive) as a structured algebraic data type with a single
constructor, MkInt, like this:

data Int = MkInt Int#

A value of type Int is represented by a pointer to a heap-
allocated object, which may either be an unevaluated suspension, or a MkInt constructor containing the machine
bit-pattern for the integer. This bit-pattern is of type
Int#.

Now that Int is given structure, we can make explicit the
evaluation performed by +, by giving the following
definition, which expresses + in terms of the primitive
machine operation #+:

a #+ b = case a of
    MkInt a# ->
    case b of
    MkInt b# ->
    MkInt (a# #+ b#)

Inlining this definition of + in the expression x+x, and
performing simple, routine simplifications, gives the fol-
lowing, in which x is evaluated only once:
case x of
  MkInt x# -> MkInt (x# +# x#)

(Unboxed types and ccall are not part of standard Haskell. They are mainly used internally in our compiler, though we do also make them available to programmers as a non-standard extension.)

We apply exactly the same ideas to ccall. In particular, instead of implementing ccall directly, we unfold every use of ccall to make the argument evaluation explicit before using the truly primitive operation ccall#. For example, the use of ccall in the definitions of putcIO and getcIO given above (Section 2.3), are unfolded thus:

putcIO a = \w ->
  case a of
    MkChar a# ->
      case (ccall# putchar a# w) of
        MkIORes# n# w' -> MkIORes () w'

getcIO = \w ->
  case (ccall# getchar w) of
    MkIORes# n# w' ->
      MkIORes (MkChar n#) w'

Like Int, the type Char is implemented as an algebraic data type thus:

data Char = MkChar Int#

The outer case expression of putcIO, therefore, evaluates a and extracts the bit-pattern a#, which is passed to ccall#. The inner case expression evaluates the expression (ccall# putchar a# w), which returns a pair, constructed by MkIORes#, consisting of the value n# returned by the C procedure putchar (which is ignored), and a new world w' (which is returned).

In the case of getcIO, the (primitive, unboxed) value n# returned by getchar is not ignored as it is in putcIO; rather it is wrapped in a MkChar constructor, and returned as part of the result.

The differences between ccall and ccall# are as follows. Firstly, ccall# takes only unboxed arguments, ready to call C directly. Secondly, it returns a pair built with MkIORes# containing an unboxed integer result direct from the C call. The IORes# type is very similar to IORes:

data IORes# = MkIORes# Int# World

(ITORes# and IORes# are distinct types, because while our extended type system recognises unboxed types, it does not permit polymorphic type constructors, such as IORes, to be instantiated at an unboxed type, such as Int#.)

Thirdly, the ccall# primitive is recognised by the code generator and expanded to an actual call to C. Specifically, the expression:

case (ccall# proc a# b# c# w) of
  MkIORes# n# w' -> ...

generates the C statement

n# = proc(a#,b#,c#);
...

This simple translation is all that the code generator is required to do. The rest is done by generic program transformations; that is, transformations which are not specific to I/O or even to unboxing (Peyton Jones & Launchbury [1991]).

4.2 Where has the world gone?

But what has become of the world values in the final C code? The world value manipulated by the program represents the current state of the real world, but since the real world is updated “in place” the world value carries no useful information. Hence we simply arrange that no code is ever generated to move values of type World. This is easy to do, as type information is preserved throughout the compiler. In particular, the world is never loaded into a register, stored in a data structure, or passed to C procedure calls.

Is it possible, then, to dispense with the world in the functional part of the implementation as well? For example, can we define the IORes type and bindIO combinators like this?

data IORes a = MkIORes a
bindIO m k w = case (m w) of
  MkIORes a -> k a w

No, we cannot! To see this, suppose that bindIO was applied to a function k which discarded its argument. Then, if bindIO was unfolded, and the expression (k r v) was simplified, there would be no remaining data dependency to force the call of k to occur after that of m. A compiler would be free to call them in either order, which destroys the I/O sequencing.

To reiterate, the world is there to form a linear chain of data dependencies between successive ccalls. It is quite safe to expose the representation of the I/O type to code-improving transformations, because the chain of data dependencies will prevent any transformations which reorder the ccalls. Once the code generator is reached,
though, the work of the world values is done, so it is safe to generate no code for them.

4.3 echo revisited

The implementation we have outlined is certainly simple, but is it efficient? Perhaps surprisingly, the answer is an emphatic yes. The reason for this is that because the combinators are written in Haskell, the compiler can unfold them at all their call sites; that is, perform procedure inlining.

Very little special-purpose code is required in the compiler to achieve this effect — essentially all that is required is that the Haskell definitions of bindIO, unitIO, putcIO, and so on, be unfolded by the compiler. In contrast, if bindIO were primitive, then every call to bindIO will require the construction of two heap-allocated closures for its two arguments. Even if bindIO itself took no time at all, this would be a heavy cost.

To illustrate the effectiveness of the approach we have outlined, we return to the echo program of Section 2.1. If we take the code there, unfold the calls of seqIO, doneIO, eof, putcIO and getcIO, and do some simplification, we get the following:

```plaintext
echo = \x ->
  case (ccall# getchar w) of
    MkIORes# a w1 ->
      case (a ==# eof#) of
        T# -> MkIORes# w1
        F# -> case (ccall# putchar a w1) of
            MkIORes# n w2 -> echo w2
```

When this is compiled using the simple code-generator described, the following C is produced:

```c
echo() {
  int a;
  a = getchar();
  if (a == eof) {
    retval = unitTuple;
    RETURN;
  } else {
    putchar(a);
    JMP( echo );
  }
}
```

(\textit{JUMP} and \textit{RETURN} are artefacts of our use of C as a target “machine code” (Peyton Jones [1992]). They expand only to a machine instruction or two.) This is very close to the C one would write by hand! We know of no other implementation of I/O with better efficiency.

4.4 A continuation-passing implementation

Like most abstract data types, there is more than one way to implement IO. In particular, it is possible to implement the IO abstract type using a continuation-passing style. The type IO a is represented by a function which takes a continuation expecting a value of type a, and returns a value of the opaque type Result.

```plaintext
type IO a = (a -> Result) -> Result
```

It is easy to implement bindIO and unitIO:

```plaintext
bindIO m k cont = m (\a -> k a cont)
unitIO r cont = cont r
```

What is there to choose between these this representation of the IO type and the one we described initially (Section 4)? The major tradeoff seems to be this: with the continuation-passing representation, every use of bindIO (even if unfolded) requires the construction of one heap-allocated continuation. In contrast, the implementation we described earlier keeps the continuation implicitly on the stack, which is slightly cheaper in our system.

There is a cost to pay for the earlier representation, namely that a heavily left-skewed composition of bindIOs can cause the stack to grow rather large. In contrast, the continuation-passing implementation may use a lot of heap for such a composition, but its stack usage is constant.

The main point is that the implementor is free to choose the representation for IO based only on considerations of efficiency and resource usage; the choice makes no difference to the interface seen by the programmer.

5 Extensions to the IO monad

5.1 Delayed I/O

So far all I/O operations have been strictly sequenced along a single “trunk”. Sometimes, though, such strict sequencing is unwanted. For example, almost all lazy functional-language I/O systems provide a readFile primitive, which returns the entire contents of a specified file as a list of characters. It is often vital that this primitive should have lazy semantics; that is, the file is opened, but only actually read when the resulting list is evaluated. The relative ordering of other I/O operations and the reading of the file is immaterial (provided the file is not simultaneously written). This lazy read is usually implemented by some ad hoc “magic” in the runtime system, but within the monadic framework it is easy to generalise the idea.
What is required is a new combinator for the IO monad, \( \text{delayIO} \), which forks off a new branch from the main "trunk":

\[
\text{delayIO} :: \text{IO } a \rightarrow \text{IO } a
\]

When performed, \( (\text{delayIO } \text{action}) \) immediately returns a suspension which when it is subsequently forced will perform the I/O specified by \( \text{action} \). The relative interleaving of the I/O operations on the "trunk" and the "branch" is therefore dependent on the evaluation order of the program.

The \( \text{delayIO} \) combinator is dangerous (albeit useful), because the correctness of the program now requires that arbitrary interleaving of I/O operations on the "trunk" and "branch" cannot affect the result. This condition cannot be guaranteed by the compiler; it is a proof obligation for the programmer. In practice, we expect that \( \text{delayIO} \) will be used mainly by system programmers.

With the aid of \( \text{delayIO} \) (and a few new primitives such as \( \text{fOpenIO} \)), it is easy to write a lazy readFile:

\[
\text{readFile} :: [	ext{Char}] \rightarrow \text{IO } [	ext{Char}]
\]

\[
\text{readFile } s = \text{fOpenIO } s \text{ ‘bindIO’ } \backslash f \rightarrow \\
\text{delayIO } (\text{lazyRd } f)
\]

\[
\text{lasyRd} :: \text{File } \rightarrow \text{IO } [\text{Char}]
\]

\[
\text{lasyRd } f = \text{readChar } f \text{ ‘bindIO’ } \backslash a \rightarrow \\
\text{if } (a /= \text{eof}) \text{ then } \\
\text{fCloseIO } f \text{ ‘seqIO’ } \\
\text{unitIO } [] \\
\text{else } \\
\text{delayIO } (\text{lasyRd } f) \text{ ‘bindIO’ } \backslash a:w \rightarrow \\
\text{unitIO } (a:a:w)
\]

The \( \text{delayIO} \) combinator provides essentially the power of Gordon's \texttt{suspend} operator (Gordon [1989]).

\section{Asynchronous I/O}

An even more dangerous but still useful combinator is \( \text{performIO} \), whose type is as follows:

\[
\text{performIO} :: \text{IO } a \rightarrow a
\]

It allows potentially side-effecting operations to take place which are not attached to the main "trunk" at all! The proof obligation here is that any such side effects do not affect the behaviour of the rest of the program. An obvious application is when one wishes to call a C procedure which really is a pure function; procedures from a numerical analysis library are one example.

\section{Assignment and reference variables}

Earlier, in Section 3.1, we discussed the apparently insoluble inefficiency of \( \text{dTToIO} \), the function which emulates \texttt{Dialogues} using the IO monad. We can solve this problem by providing an extra general-purpose mechanism, that of \textit{assignable reference types} and operations over them (Ireland [1989]):

\[
\text{newVar} :: a \rightarrow \text{IO } (\text{Ref } a)
\]

\[
\text{assignVar} :: \text{Ref } a \rightarrow a \rightarrow \text{IO } ()
\]

\[
\text{deRefVar} :: \text{Ref } a \rightarrow \text{IO } a
\]

The call \( \text{newVar } x \) allocates a fresh variable containing the value \( x \); the call \( \text{assignVar } v x \) assigns value \( x \) to variable \( v \); and the call \( \text{deRefVar } v \) fetches the value in variable \( v \). By making these side-effecting operations part of the IO monad, we make sure that their order of evaluation, and hence semantics, is readily explicable.

With the aid of these primitives it is possible to write an efficient emulation of \texttt{Dialogues} using IO (Figure 3). The idea is to mimic a system which directly implements \texttt{Dialogues}, which follows the processing of each request with a destructive update to add a new response to the end of the list of responses. Notice the uses of \( \text{delayIO} \) which reflects the fact that there is no guarantee that \texttt{dialogue} will not evaluate a response before it has emitted a request. If this occurs, the un-assigned variable is evaluated, which elicits a suitable error message.

References in languages such as ML require a weakened form of polymorphism in order to maintain type safety
Figure 3: Efficient conversion from Dialogue to IO

(Tofte [1990]). For instance, in ML a fresh reference to an empty list has type `\_a list ref`, where the type variable `\_a` is weak and so may be instantiated only once. In contrast, here a fresh reference to an empty list has type IO (Ref `a`), and the type variable `a` is normal. But no lack of safety arises, because an expression of this type allocates a new reference each time it is evaluated. The only way to change a value of type IO (Ref `a`) to one of type Ref `a` is via bindIO, but now the variable of type Ref `a` is not let-bound, and so can only be instantiated once anyway. Hence the extra complication of weak type variables, required in languages with side effects, seems unnecessary here. (We’re indebted to Martin Odersky for this observation.)

6 Arrays

The approach we take to I/O smoothly extends to arrays with in-place update. Hudak has recently proposed a similar method based on continuations. For I/O, the monad and continuation approaches are interdefinable. For arrays, it turns out that monads can implement continuations, but not the converse.

Let Arr be the type of arrays taking indexes of type Ind and yielding values of type Val. There are three operations on this type.

\[
\begin{align*}
\text{new} & \colon \text{Val} \to \text{Arr} \\
\text{lookup} & \colon \text{Ind} \to \text{Arr} \to \text{Val} \\
\text{update} & \colon \text{Ind} \to \text{Val} \to \text{Arr} \to \text{Arr}
\end{align*}
\]

The call new `v` returns an array with all entries set to `v`; the call lookup `i` `x` returns the value at index `i` in array `x`; and the call update `i` `v` `x` returns an array where index `i` has value `v` and the remainder is identical to `x`.

The behaviour of these operations is specified by the usual laws.

\[
\begin{align*}
\text{lookup} \ i \ (\text{new} \ v) &= v \\
\text{lookup} \ i \ (\text{update} \ i \ v \ x) &= v \\
\text{lookup} \ i \ (\text{update} \ j \ v \ x) &= \text{lookup} \ i \ x
\end{align*}
\]

where `i ≠ j` in the last equation. In practice, these operations would be more complex; one needs a way to specify the array bounds, for instance. But the above suffices to explicate the main points.

The efficient way to implement the update operation is to overwrite the specified entry of the array, but in a pure functional language this is only safe if there are no other pointers to the array extant when the update operation is performed. An array satisfying this property is called single threaded, following Schmidt (Schmidt [1985]).

As an example, consider the following problem. An occurrence is either a definition pairing an index with a value, or a use of an index.

\[
\begin{align*}
data \ Occ &= \ \text{Def} \ \text{Ind} \ \text{Val} \mid \ \text{Use} \ \text{Ind} \\
\text{uses} \ os &= \ \text{loop} \ os \ (\text{new} \ '-' \ )
\end{align*}
\]

For illustration take Ind = Int and Val = Char. Given a list os of occurrences, the call uses `os` returns for each use the most recently defined value (or `'-'` if there is no previous definition). If

\[
\text{os} = [\text{Def} \ 1 \ 'a', \ \text{Def} \ 2 \ 'b', \ \text{Use} \ 1, \\
\text{Def} \ 1 \ 'c', \ \text{Use} \ 2, \ \text{Use} \ 1]
\]

then

\[
\text{uses} \ os = ['a', 'b', 'c']
\]

Here is the code.

\[
\begin{align*}
\text{uses} & :: [\text{Occ}] \to [\text{Val}] \\
\text{uses} \ os &= \ \text{loop} \ os \ (\text{new} \ '-' \ )
\end{align*}
\]

loop `[]` `x` = `[]`
loop `(\text{Def} \ i \ v : \ os)` `x` = loop `os` (update `i` `v` `x`)
loop `(\text{Use} \ i : \ os)` `x` = lookup `i` `x` : loop `os` `x`

The update in this program can be performed by overwriting, but some care is required with the order of evaluation. In the last line, the lookup must occur before the recursive call which may update the array. Some work has been done on analysing when update can be performed in-place, but it is rather tricky (Bloss [1989]; Hudak [1986]).
6.1 Monadic arrays

We believe that single threading is too important to leave to the vagaries of an analyzer. Instead, we use monads to guarantee single threading, in much the same way as was done with I/O. Analogous to the type \texttt{IO a} (the monad of I/O actions), we provide an abstract type \texttt{A a} (the monad of array transformers).

\begin{verbatim}
newA :: Val -> A a -> a
lookupA :: Ind -> A Val
updateA :: Ind -> Val -> A ()
unitA :: a -> A a
bindA :: A a -> (a -> A b) -> A b
\end{verbatim}

For purposes of specification, we can define these in terms of the preceding operations as follows:

\begin{verbatim}
type A a = Arr -> (a, Arr)
newA v m = fst (m (new v))
lookupA i = \x -> (lookup i x, x)
updateA i v = \x -> ((), update i v x)
unitA a = \x -> (a,x)
m 'bindA' k = \x -> let (a,y) = m x in k a y
\end{verbatim}

A little thought shows that these operations are indeed single threaded. The only operation that could duplicate the array is \texttt{lookupA} but this may be implemented as follows: first fetch the entry at the given index in the array, and then return the pair consisting of this value and the pointer to the array. To enforce the necessary sequencing, we augment the above specification with the requirement that \texttt{lookupA} and \texttt{updateA} are strict in the index and array arguments (but need not be strict in the value).

The above is given for purposes of specification only—the actual implementation is along the lines of Section 4.

For convenience, define \texttt{seqA} in terms of \texttt{bindA} in the usual way:

\begin{verbatim}
m 'seqA' n = m 'bindA' \rightarrow n
\end{verbatim}

This is somewhat lengthier than the previous example, but it is guaranteed safe to implement update by over-writing.

6.2 Continuation arrays

An alternative method of guaranteeing single threading for arrays has been proposed by Hudak [1992]. Like the previous work of Swarup, Reddy & Ireland [1991], it is based on continuations, but unlike that work it requires no change to the type system.

As with the array monad, one defines an abstract type supporting various operations. The type is \texttt{C z}, and the operations are as follows:

\begin{verbatim}
newC :: Val -> C z -> z
lookupC :: Ind -> (Val -> C z) -> C z
updateC :: Ind -> Val -> C z -> C z
unitC :: z -> C z
\end{verbatim}

Here a continuation, of type \texttt{C z}, represents the remaining series of actions to be performed on the array, eventually returning (via \texttt{unitC}) a value of type \texttt{z}.

For purposes of specification, we can define these in terms of the array operations as follows:

\begin{verbatim}
type C z = Arr -> z
newC v c = c (new v)
lookupC i d = \x -> d (lookup i x) x
updateC i v c = \x -> c (update i v x)
unitC z = \x -> z
\end{verbatim}

Again, these operations are single threaded if \texttt{lookupC} and \texttt{updateC} are strict in the index and array arguments.

For convenience, define

\begin{verbatim}
m $ c = m c
\end{verbatim}

This lets us omit some parentheses, since \texttt{m (\x -> n)} becomes \texttt{m $ \x -> n}.

Here is the 'definition-use' problem, recoded in monadic style.

\begin{verbatim}
uses :: [Occ] -> [Val]
uses os = newA ' ' (loopA os)
loopA :: [Occ] -> A [Val]
loopA [] = unitA []
loopA (Def i v : os) = updateA i v 'seqA' loopA os
loopA (Use i : os) = lookupA i 'bindA' \rightarrow loopA os 'bindA' \rightarrow
\end{verbatim}

For convenience, define
loopC (Use i : os) c = lookupC i $ \v \rightarrow
  loopC os $ \vs \rightarrow
  c (\v : vs)

This is remarkably similar to the monadic style, where $ takes the place of bindA and seqA, and the current continuation c takes the place of unitA. (If c plays the role of unitA, why do we need unitC? Because it acts as the ‘top level’ continuation.)

However, there are two things to note about the continuation style. First, the types are rather more complex - compare the types of loopA and loopC. Second, the monadic style abstracts away from the notion of continuation - so there are no occurrences of c cluttering the definition of loopA.

6.3 Monads vs. continuations

We can formally compare the power of the two approaches by attempting to implement each in terms of the other. Despite their similarities, the two approaches are not equivalent. Monads are powerful enough to implement continuations, but not (quite) vice versa.

To implement continuations in terms of monads is simplicity itself.

type C z = A z
newC v c = newA v c
lookupC i d = lookupA i 'bindA' d
updateC i v c = updateA i v 'seqA' c
unitC = unitA

It is an easy exercise in equational reasoning to prove that this implementation is correct in terms of the specifications in Sections 6.1 and 6.2.

The reverse implementation is not possible. The trouble is the annoying extra type variable, z, appearing in the types of lookupC and updateC. This forces the introduction of a spurious type variable into any attempt to define monads in terms of continuations. Instead of a type A a, the best one can do is to define a type B a z. Here are the types of the new operations.

newB :: Val -> B a a -> a
lookupB :: Ind -> B Val z
updateB :: Ind -> Val -> B () z
unitB :: a -> B a z
bindB :: B a z -> (a -> B b z) -> B b z

And here are the implementations in terms of continuations.

type B a z = (a -> C z) -> C z
newB v m = newC v (m unitC)
lookupB i = \d \rightarrow lookupC i d
updateB i v = \d \rightarrow updateC i v (d ())
unitB a = \d \rightarrow d a
m 'bindB' k = \d \rightarrow m (\a \rightarrow k a d)

Again, it is easy to prove this implementation satisfies the given specifications.

So monads are more powerful than continuations, but only because of the types! It is not clear whether this is simply an artifact of the Hindley-Milner type system, or whether the types are revealing a difference of fundamental importance. (Our own intuition is the latter - but it’s only an intuition.)

6.4 Conclusion

The I/O approach outlined earlier manipulates a global state, namely the entire state of the machine accessible via a C program. What has been shown in this section is that this approach extends smoothly to manipulating local state, such as a single array. Further, although the monad and continuation approaches are interconvertible for I/O, they are not for arrays: monads are powerful enough to define continuations, but not the reverse.

For actual use with Haskell, we require a slightly more sophisticated set of operations. The type A must take extra parameters corresponding to the index and value types, the operation newA should take the array bounds, and so on. By using a variant of newA that creates an uninitialised array, and returns the array after all updates are finished, it is possible to implement Haskell primitives for creating arrays in terms of the simpler monad operations. Thus the same strategy that works for implementing I/O should work for implementing arrays: use a small set of primitives based on monads, and depend on program transformation to make this adequately efficient.

One question that remains is how well this approach extends to situations where one wishes to manipulate more than one state at a time, as when combining I/O with array operations, or operating on two arrays. In this respect effect systems or linear types may be superior; see below.

7 Related work

7.1 Effect systems

Gifford and Lucassen introduced ‘effect systems’ which use types to record the side-effects performed by a pro-
gram, and to determine which components of a program
can run in parallel without interference (Gifford & Lu-
cassen [1986]). The original notion of effect was fairly
crude, there being only four possible effects: pure (no ef-
fect), allocate (may allocate storage), function (may read
storage), procedure (may write storage). New systems are
more refined, allowing effects to be expressed separately
for different regions of store (Jouvelot & Gifford [1991]).

A theoretical precursor of the effects work is that of
Reynolds, which also used types to record where effects
could occur and where parallelism was allowed (Reynolds
[1981]; Reynolds [1989]).

Our work is similar to the above in its commitment to
use types to indicate effects. But effect systems are de-
signed for impure, strict functional languages, where the
order of sequencing is implicit. Our work is designed for
pure, lazy functional languages, and the purpose of the
`bind` operation is to make sequencing explicit where it is
required.

With effect systems, one may use the usual laws of
equational reasoning on any program segment without
a `write` side effect. Our work differs in that the laws of
equational reasoning apply even where side effects are
allowed. This is essential, because the optimisation phase
of our compiler is based on equational reasoning.

On the other hand, effect systems make it very easy
to combine programs with different effects. In our ap-
proach, each different effect would correspond to a differ-
et monad type (one for IO, one for each array manipu-
lated, and so on), and it is not so clear how one goes
about combining effects.

7.2 Linear types

The implementation of the IO monad given in Section 4
is safe because (and only because) the code that manipu-
lates the world never duplicates or destroys it. We guar-
antee safety by making the IO type abstract, so that user
has no direct access to the world.

An alternative is to allow the user access to the world,
but introduce a type system that guarantees that the
world can never be duplicated or destroyed. A number of
type systems have been proposed along such lines. Some
have been based on Girard’s linear logic (Girard [1987]),
and this remains an area of active exploration (Abram-
sky [1990]; Guzman & Hudak [1990]; Wadler [1990]). An-
other is the type system proposed by the Nijmegen Clean
group, which is more ad-hoc but has been tested in prac-
tical applications similar to our own (Achten, Groningen
& Plasmeijer [1992]).

For example, here is the `echo` program again, written in
the style suggested by the Clean I/O system:

```
echo :: File -> File -> World -> World
echo fi fo w = if a == eof
  then w1
  else echo (putChar fo a w)
where
  (w1,a) = getChar fi w
```

Compared to the monad approach, this suffers from a
number of drawbacks: programs become more cluttered;
the linear type system has to be explained to the pro-
grammer and implemented in the compiler; and code-
important transformations need to be re-examined to en-
sure they preserve linearity. The latter problem may be
important; Wakeling found that some standard transfor-
mations could not be performed in the presence of linear-
ity (Wakeling [1990]).

The big advantage of a linear type system is that it en-
ables us to write programs which manipulate more than
one piece of updatable state at a time. The monadic and
continuation-passing presentations of arrays given above
pass the array around implicitly, and hence can only eas-
ily handle one at a time. This is an important area for
future work.

On the practical side, the Clean work is impressive. They
have written a library of high-level routines to call the
Macintosh window system, and demonstrated that it is
possible to build pure functional programs with sophisti-
cated user interfaces. The same approach should work for
monads, and another area for future work is to confirm
that this is the case.

8 Conclusions and further work

We have been pleasantly surprised by both the expres-
siveness and the efficiency of the approach we have de-
scribed. For example, we have found that while it is pos-
sible to write composable I/O programs in other styles,
itis almost impossible not to do so in using the monadic
approach.

Plenty remains to be done. We are working on our im-
plementation of arrays; this in turn feeds into the ability
to pass structured values in calls; we have not yet im-
plemented assignable reference types.

More importantly, the model we have described concerns
only the I/O infrastructure. Much more work needs to be
done to design libraries of functions, built on top of this
infrastructure, which present a higher-level interface to
the programmer (Achten, Groningen & Plasmeijer [1992]);
Acknowledgements

This work took place in the context of the team building the Glasgow Haskell compiler: Cody Hall, Kevin Hammond and Will Partain. David Watt, Joe Morris, John Launchbury also made very helpful suggestions about our presentation. We gratefully acknowledge their help.

References

S Abramsky [1990], “Computational interpretations of linear logic,” DOC 90/20, Dept of Computing, Imperial College.


A Bloss [Sept 1989], “Update analysis and the efficient implementation of functional aggregates,” in Functional Programming Languages and Computer Architecture, London, ACM.


P Hudak [July 1992], “Continuation-based mutable abstract datatypes, or how to have your state and munge it too.” YALEU/DCS/RR-014. Department of Computer Science, Yale University.


P Jouvelot & D Gifford [Jan 1991], “Algebraic reconstruction of types and effects,” in 18th ACM Symposium on Principles of Programming Languages (POPL), Orlando, ACM.


E Moggi [June 1989], “Computational lambda calculus and monads,” in Logic in Computer Science, California. IEEE.


SL Peyton Jones [Oct 1988], “Converting streams to continuations and vice versa,” Electronic mail on Haskell mailing list.


J Reynolds [1989], “Syntactic control of interference, part II,” in International Colloquium on Automata, Languages, and Programming.


M Tofte [Nov 1990], “Type inference for polymorphic references,” Information and Computation 89.

PL Wadler [1990], “Linear types can change the world!,” in Programming concepts and methods, M Broy & C Jones, eds., North Holland.

PL Wadler [Jan 1992], “The essence of functional programming,” in Proc Principles of Programming Languages, ACM.

PL Wadler [June 1990], “Comprehending monads,” in Proc ACM Conference on Lisp and Functional Programming, Nice, ACM.