# Imperative functional programming

Simon L Peyton Jones Philip Wadler*

January 1993

**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

*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: `simonpj,wadler@dcs.glasgow.ac.uk`.

This paper appears in *20'th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

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 an 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.

```
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.

```
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:

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

The compound action ``m `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:

```
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:

```
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:

```
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:

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

If `m :: IO a` and `k :: a -> IO b` then ``m `bindIO` k`` denotes the action that, when performed, behaves as follows: 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, `\x -> e` stands for a lambda abstraction; the body of the abstraction extends as far as possible.)

```
echo :: IO ()
echo = getcIO `bindIO` \a ->
       if (a == eof) then
            doneIO
       else
            putcIO a `seqIO`
            echo
```

The combinators `bindIO` and `unitIO` are generalisations of `seqIO` and `doneIO`. Here are definitions for the latter in terms of the former:

```
doneIO       =  unitIO ()
m `seqIO` n  =  m `bindIO` \a -> n
```

The combinators have a useful algebra: `doneIO` and `seqIO` form a *monoid*, while `bindIO` and `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 `echo` program in C might look something like this:

```
echo() {
loop: a = getchar(a);
      if (a == eof)
             return;
      else { putchar(a);
             goto loop; }
}
```

(Indeed, as we discuss later, our compiler translates the `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 `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 `putsIO` given above uses explicit recursion. Here is an alternative way to write `putsIO` which does not do so:

```
putsIO as = seqsIO (map putcIO as)
```

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

```
seqsIO :: [IO a] -> IO ()
seqsIO []     = doneIO
seqsIO (a:as) = a `seqIO` seqsIO as
```

or even, using the standard list-processing function `foldr`, thus:

```
seqsIO = foldr seqIO doneIO
```

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

```
spaceIO :: Int -> IO ()
spaceIO n
  = seqsIO (take n (repeat (putcIO ' ')))
```

The functions `take` and `repeat` are standard list-processing functions (with nothing to do with I/O) from Haskell's standard prelude. The function `repeat` takes a value and returns an infinite list each of whose elements is the given value. The function `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 `putcIO`, `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 `ccall`, whose general form is:

$$\texttt{ccall}\ \mathit{proc}\ e_1\ \ldots\ e_n$$

Here, *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`:

```
putcIO a = ccall putchar a
getcIO   = ccall getchar
```

These `ccall`s directly invoke the system-provided functions; 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 accomodated 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:

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

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

```
data Request = Putc Char | Getc
data Response = OK | OKCh Char
```

(For the purposes of exposition we have grossly simplified 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 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`.)

```
echo :: Dialogue
echo resps = Getc :
             if (a == eof)
             then []
             else Putc a :
                  echo (drop 2 resps)
            where
             OKCh a = resps!!1
```

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 responses, a *synchronisation error*. This may show up in a variety of ways. If the "2" in the above program was erroneously written as "1" the program would fail with a pattern-mathing 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 emulate the `IO` monad, and vice versa. The function `dToIO`, which emulates `Dialogue`s in terms of `IO` is rather curious, because it involves applying the single dialogue `d` to both bottom ($\perp$) and (later) to the "real" list 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`s are 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:

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

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

```
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`s, execution is driven by a "wrapper" program, which evaluates `main`, performs the operation indicated by the constructor, and

Dialogue to IO

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

bottom :: a
bottom = error \mbox{$\it Should\ never\ be\ evaluated$}

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 =  [Response]
          -> (a, [Request], [Response])

ioToD :: IO () -> Dialogue
ioToD action = \rs -> case (io 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`

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
```

IO to continuations

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

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

unitIO v      = \k -> k v
bindIO op fop = \k -> op (\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`

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 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 synomym 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* `ccall`*s will be linked in a single, linear chain*, connected by data dependencies in which each `ccall` 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 `ccall` 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 following, 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 uses 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
```

(IORes 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 w)` 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 `ccall`s. It is quite safe to expose the representation of the `IO` type to code-improving transformations, because the chain of data dependencies will prevent any transformations which reorder the `ccall`s. 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:

```
echo = \w ->
 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:

```
echo() {
  int a;
  a = getchar();
```

```
  if (a == eof) {
        retVal = unitTuple;
        RETURN;
  } else {
        putchar(a);
        JUMP( echo );
} }
```

(JUMP and 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`.

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

It is easy to implement `bindIO` and `unitIO`:

```
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 `bindIO`s 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, `delayIO`, which forks off a new branch from the main "trunk":

```
delayIO :: IO a -> IO a
```

When performed, `(delayIO action)` immediately returns a suspension which *when it is subsequently forced* will perform the I/O specified by `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 `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 `delayIO` will be used mainly by system programmers.

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

```
readFile :: [Char] -> IO [Char]
readFile s = fOpenIO s `bindIO` \f ->
             delayIO (lazyRd f)

lazyRd :: File -> IO [Char]
lazyRd f
 = readChar f `bindIO` \a ->
   if (a == eof) then
     fCloseIO f `seqIO`
     unitIO []
   else
     delayIO (lazyRd f) `bindIO` \as ->
     unitIO (a:as)
```

The `delayIO` combinator provides essentially the power of Gordon's `suspend` operator (Gordon [1989]).

*Implementation.* A nice feature of the implementation technique outlined in Section 4 is that `delayIO` is very easy to define:

```
delayIO m = \w -> MkIORes res w
  where
   res = case (m w) of
           MkIORes r w' -> r
```

In contrast to `bindIO`, notice how `delayIO` duplicates the world `w`, and then discards the final world `w'` of the delayed branch; it is this which allows the unsynchronised interleaving of I/O operations on the "branch" with those on the "trunk".

## 5.2 Asynchronous I/O

An even more dangerous but still useful combinator is `performIO`, whose type is as follows:

```
performIO :: IO a -> 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.

*Implementation.* The implementation is quite simple:

```
performIO m = case (m newWorld) of
                MkIORes r w' -> r
```

Here, `newWorld` is a value of type `World` conjured up out of thin air, and discarded when the action `m` has been performed.

## 5.3 Assignment and reference variables

Earlier, in Section 3.1, we discussed the apparently insoluble inefficiency of `dToIO`, the function which emulates `Dialogue`s using the `IO` monad. We can solve this problem by providing an extra general-purpose mechanism, that of *assignable reference types* and operations over them (Ireland [1989]):

```
newVar    :: a -> IO (Ref a)
assignVar :: Ref a -> a -> IO ()
deRefVar  :: Ref a -> IO a
```

The call `newVar x` allocates a fresh variable containing the value `x`; the call `assignVar v x` assigns value `x` to variable `v`; and the call `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 efficent emulation of `Dialogues` using `IO` (Figure 3). The idea is to mimic a system which directly implements `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 `delayIO`, which reflects the fact that there is no guarantee that `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.

```
dToIO :: Dialogue -> IO ()
dToIO dialogue
 = newVar (error \mbox{$\it Synch$}) `bindIO` \rsV ->
   delayIO (deRefVar rsV) `bindIO` \rs ->
   run (dialogue rs) rsV

run :: [Request] -> Ref [Response] -> IO ()
run []         v = doneIO
run (req:reqs) v
 = doReq req             `bindIO` \r ->
   newVar (error \mbox{$\it Synch$}) `bindIO` \rsV ->
   delayIO (deRefVar rsV) `bindIO` \rs ->
   assignVar v (r:rs)     `seqIO`
   run reqs rsV
```

Figure 3: Efficient conversion from `Dialogue` to `IO`

References in languages such as ML require a weakened form of polymorphism in order to maintain type safety (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.

```
new    :: Val -> Arr
lookup :: Ind -> Arr -> Val
update :: Ind -> Val -> Arr -> Arr
```

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.

```
lookup i (new v)   =  v
lookup i (update i v x) = v
lookup i (update j v x) = lookup i x
```

where $i \neq j$ 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.

```
data  Occ  =  Def Ind Val | Use Ind
```

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

```
os = [Def 1 'a', Def 2 'b', Use 1,
      Def 1 'c', Use 2, Use 1]
```

then

```
uses os = ['a', 'b', 'c'].
```

Here is the code.

```
uses     :: [Occ] -> [Val]
uses os   =  loop os (new '-')

loop :: [Occ] -> Arr -> [Val]
loop []              x =  []
loop (Def i v : os) x = loop os (update i v x)
loop (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 analyser. Instead, we use monads to *guarantee* single threading, in much the same way as was done with I/O. Analogous to the type `IO a` (the monad of I/O actions), we provide an abstract type `A a` (the monad of array transformers).

```
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
```

For purposes of specification, we can define these in terms of the proceeding operations as follows.

```
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
```

A little thought shows that these operations are indeed single threaded. The only operation that could duplicate the array is `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 `lookupA` and `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 `seqA` in terms of `bindA` in the usual way.

```
m `seqA` n  =  m `bindA` \a -> n
```

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

```
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` \v  ->
```

```
                        loopA os  `bindA` \vs ->
                        unitA (v:vs)
```

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

## 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 `C z`, and the operations are as follows.

```
newC    :: Val -> C z -> z
lookupC :: Ind -> (Val -> C z) -> C z
updateC :: Ind -> Val -> C z -> C z
unitC   :: z -> C z
```

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

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

```
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
```

Again, these operations are single threaded if `lookupC` and `updateC` are strict in the index and array arguments.

For convenience, define

```
m $ c  =  m c
```

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

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

```
uses     :: [Occ] -> [Val]
uses os  =  newC '-' (loopC os unitC)

loopC :: [Occ] -> ([Val] -> C z) -> C z
loopC []              c = c []
loopC (Def i v : os) c = updateC i v  $
```

```
                             loopC os c
loopC (Use i : os) c    = lookupC i $ \v  ->
                          loopC os  $ \vs ->
                          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 defintion 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 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 -> lookupC i d
updateB i v   =  \d -> updateC i v (d ())
unitB a       =  \d -> d a
m `bindB` k   =  \d -> m (\a -> 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 program, and to determine which components of a program can run in parallel without interference (Gifford & Lucassen [1986]). The original notion of effect was fairly crude, there being only four possible effects: pure (no effect), 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 designed for impure, strict functional languaes, 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 approach, each different effect would correspond to a different monad type (one for IO, one for each array manipulated, 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 manipulates the world never duplicates or destroys it. We guarantee 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 (Abramsky [1990]; Guzman & Hudak [1990]; Wadler [1990]). Another is the type system proposed by the Nijmegen Clean group, which is more ad-hoc but has been tested in practical 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 w1)
               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 programmer and implemented in the compiler; and code-improving transformations need to be re-examined to ensure they preserve linearity. The latter problem may be important; Wakeling found that some standard transformations could not be performed in the presence of linearity (Wakeling [1990]).

The big advantage of a linear type system is that it enables 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 easily 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 sophisticated 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 expressiveness and the efficiency of the approach we have described. For example, we have found that while it is possible to write composable I/O programs in other styles, it is almost impossible *not* to do so in using the monadic approach.

Plenty remains to be done. We are working on our implementation of arrays; this in turn feeds into the ability to pass structured values in `ccall`s; we have not yet implemented assignable reference types.

More importantly, the model we have desribed 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]; Hammond, Wadler & Brady [1991]).

# Acknowledgements

This work took place in the context of the team building the Glasgow Haskell compiler: Cordy 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.

PM Achten, JHG van Groningen & MJ Plasmeijer [1992], "High-level specification of I/O in functional languages," in *Proc Glasgow Workshop on Functional Programming*, Launchbury *et al*, ed., Springer Verlag.

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

A Dwelly [Sept 1989], "Dialogue combinators and dynamic user interfaces," in *Functional Programming Languages and Computer Architecture, London*, ACM.

DK Gifford & JM Lucassen [Aug 1986], "Integrating functional and imperative programming," in *ACM Conference on Lisp and Functional Programming, MIT*, ACM, 28–38.

J-Y Girard [1987], "Linear Logic," *Theoretical Computer Science* 50, 1–102.

A Gordon [Feb 1989], "PFL+: a kernel scheme for functional I/O," TR 160, Computer Lab, University of Cambridge.

JC Guzman & P Hudak [1990], "Single-threaded polymorphic lambda calculus," in *Proc 5th Annual IEEE Symposium on Logic in Computer Science*.

K Hammond, PL Wadler & D Brady [1991], "Imperate: be imperative," Department of Computer Science, Univ of Glasgow.

P Hudak [July 1992], "Continuation-based mutable abstract datatypes, or how to have your state and munge it too," YALEU/DCS/RR-914, Department of Computer Science, Yale University.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27.

P Hudak & RS Sundaresh [March 1989], "On the expressiveness of purely-functional I/O systems," YALEU/DCS/RR-665, Department of Computer Science, Yale University.

Paul Hudak [Aug 1986], "A semantic model of reference counting and its abstraction," *Proc ACM Conference on Lisp and Functional Programming*.

E Ireland [March 1989], "Writing interactive and file-processing functional programs," MSc thesis, Victoria University of Wellington.

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

Kent Karlsson [1982], "Nebula - a functional operating system," Chalmers Inst, Goteborg.

BW Kernighan & DM Ritchie [1978], *The C programming language*, Prentice Hall.

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

JT O'Donnell [1985], "Dialogues: a basis for constructing programming environments," in *Proc ACM Symposium on Language Issues in Programming Environments, Seattle*, ACM, 19–27.

N Perry [1991], "The implementation of practical functional programming languages," PhD thesis, Imperial College, London.

SL Peyton Jones [Apr 1992], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

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

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag.

J Reynolds [1981], "The essence of Algol," in *Algorithmic Languages*, de Bakker & van Vliet, eds., North Holland, 345–372.

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

DA Schmidt [Apr 1985], "Detecting global variables in denotational specifications," *TOPLAS* 7, 299–310.

V Swarup, US Reddy & E Ireland [Sept 1991], "Assignments for applicative languages," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 192–214.

SJ Thompson [1989], "Interactive functional programs - a method and a formal semantics," in *Declarative Programming*, DA Turner, ed., Addison Wesley.

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.

D Wakeling [Nov 1990], "Linearity and laziness," PhD thesis, Department of Computer Science, University of York.