Last Modified On Mon Jul 18 11:05:23 1994 By Andy Gordon

A Proposal for Monadic I/O in Haskell 1.3

If you are browsing using Mosaic, you can print this document using the Print.. option from the File menu (in either PostScript or ASCII formats).

We have frozen the proposal (from 940718) to allow implementation and comments from would be users. For the next few months we only intend to make changes to fix typos, mistakes and ambiguities, before a final revision. Please email comments to Andy Gordon *adg@cl.cam.ac.uk* and Kevin Hammond *kh@dcs.gla.ac.uk*. We're maintaining a log of recent changes to the proposal.

ranging over arbitrary phrases of the object language. For instance, Haskell function `ident` applied to an arbitrary *x* equals *x*.

2 New Types and Values

`this` font and use *this* font for metavariables

We propose to remove the old stream-based I/O mechanism from Haskell and install in its place a new monadic I/O mechanism. We specify in this document a new module for I/O, whose header is as follows.

```
module PreludeMonadicIO (

-- primitive types

 Handle,

-- primitive values

 stdin, stdout, stderr, stdnull, openFile, openChan, flush, close,
 ready, hGetChar, getContents, hPutChar, setBuffering, setEchoing,
 seek, query, select, deleteFile, statusFile, statusChan, getArgs,
 getProgName, getEnv, setEnv, getClock, getCpuTime, runProcess,
 setInterrupt,

-- derived types

 IO, IOError(..), IOMode(..), BufferMode(..), HandleState(..),
 HandleKind(..), OpenClosed(..), FileChan(..), SelectData,

-- derived values

 showError, return, (>>=), failwith, try, (>>), fail, getChar,
 hPutStr, hPutText, putChar, putStr, putText, interact, readFile,
 readChan, appendFile, appendChan, writeFile, system, accumulate,
 accumulate_)
```

We draw a distinction between *primitive* types and values, which are builtin to Haskell, like `Char` and `ord`, and *derived* ones, which can be expressed using Haskell declarations. To specify these we use a local primitive type constructor `PrimIO`, a local derived type constructor `Either`, and local values `returnPrimIO`, `thenPrimIO` and `either`. These locals are not exported from the module.

In more detail than the module header, here is a complete listing of the types and values used in the proposal. Their intended meaning comes later. Locals are marked with '`****`'.

Primitive Types

```
****    PrimIO,
        Handle
```

Primitive Values

```
****    returnPrimIO     :: a -> PrimIO a
****    thenPrimIO       :: PrimIO a -> (a -> PrimIO b) -> PrimIO b
```

```
        stdin           :: Handle
        stdout          :: Handle
        stderr          :: Handle
        stdnull         :: Handle
        openFile        :: IOMode -> String -> IO Handle
        openChan        :: String -> IO Handle
        flush           :: Handle -> IO ()
        close           :: Handle -> IO ()
        ready           :: Handle -> IO Bool
        hGetChar        :: Handle -> IO Char
        getContents     :: Handle -> IO String
        hPutChar        :: Handle -> Char -> IO ()
        setBuffering    :: Handle -> BufferMode -> IO ()
        setEchoing      :: Handle -> Bool -> IO ()
        seek            :: Handle -> Integer -> IO ()
        query           :: Handle -> IO HandleState
        select          :: SelectData -> IO (Maybe SelectData)
        deleteFile      :: String -> IO ()
        statusFile      :: String -> IO String
        statusChan      :: String -> IO String
        getArgs         :: IO [String]
        getProgName     :: IO String
        getEnv          :: String -> IO String
        setEnv          :: String -> String -> IO ()
        getClock        :: IO Integer
        getCpuTime      :: IO Integer
        runProcess      :: String -> [Handle] -> IO ()
        setInterrupt    :: IO () -> IO (IO ())
```

Fixities

```
        infixr 1 >>, >>=
```

Derived Types

```
        type IO a       = PrimIO (Either a IOError)
****    data Either a b = Left a | Right b
        data IOError    = WriteError    String
                        | ReadError     String
                        | SearchError   String
                        | FormatError   String
                        | OtherError    String
                        | EOF
        data IOMode     = ReadMode | WriteMode | AppendMode
        data BufferMode = UnBuffered | LineBuffered | BlockBuffered
        data HandleState= HandleState String HandleKind
                              OpenClosed FileChan BufferMode Bool
        data HandleKind = InputOnly | OutputOnly | InputOutput
        data OpenClosed = IsOpen | IsClosed | IsSemiClosed
        data FileChan   = IsChannel
                        | IsFile Integer Integer   -- size, posn
        type SelectData = ([Handle], [Handle], Maybe Integer)
```

Derived Values

```
        showError       :: IOError -> String
****    either          :: (a -> c) -> (b -> c) -> (Either a b) -> c
        return          :: a -> IO a
        (>>=)           :: IO a -> (a -> IO b) -> IO b
        failwith        :: IOError -> IO a
        try             :: IO a -> (IOError -> IO a) -> IO a
        (>>)            :: IO a -> IO b -> IO b
```

```
fail            :: String -> IO a
getChar         :: IO Char
hPutStr         :: Handle -> String -> IO ()
hPutText        :: Text a => Handle -> a -> IO ()
putChar         :: Char -> IO ()
putStr          :: String -> IO ()
putText         :: Text a => a -> IO ()
interact        :: (String -> String) -> IO ()
readFile        :: String -> IO String
readChan        :: String -> IO String
appendFile      :: String -> String -> IO ()
appendChan      :: String -> String -> IO ()
writeFile       :: String -> String -> IO ()
system          :: String -> IO String
accumulate      :: [IO a] -> IO [a]
accumulate_     :: [IO a] -> IO ()
```

---

3 Computations

Our model is based on two monads, `PrimIO` and `IO`. One, `PrimIO`, is primitive. The other, `IO`, type is a specialised form of `PrimIO`, derived using a `type` declaration.

(The `PrimIO` type constructor is called `IO` in the prototype `ghc` implementation. Our type `IO` provides more functionality than the `ghc` `IO` type.)

3.1 The `PrimIO` monad

An expression of type `PrimIO` *a*, for some type *a*, denotes a *computation* that may engage in I/O, and, if it terminates, returns an answer of type *a*. Here are two primitive operations.

```
> returnPrimIO  :: a -> PrimIO a
> thenPrimIO    :: PrimIO a -> (a -> PrimIO b) -> PrimIO b
```

Computation `returnPrimIO` *x* immediately terminates with answer *x*. Computation *p* `` `thenPrimIO` `` *f* begins with `p`, and if `p` returns an answer *x*, continues with computation *f x*.

The structure (`PrimIO`, `returnPrimIO`, `thenPrimIO`) is intended to be a monad, in the sense that the following laws hold.

1. `returnPrimIO` *x* `` `thenPrimIO` `` *f* = *f x*.
2. *p* `` `thenPrimIO` `` `returnPrimIO` = *p*.
3. *p* `` `thenPrimIO` `` \x -> (*f x* `` `thenPrimIO` `` *g*) = (*p* `` `thenPrimIO` `` *f*) `` `thenPrimIO` `` *g*.

These three laws are an equational characterisation of the categorical notion of a monad; see Wadler (92), for instance, for a discussion.

3.2 The `IO` monad

I/O operations may need to indicate errors. We provide for handling of user or system generated errors in a specialised form of the `PrimIO` monad, called `IO`.

```
> type IO a       = PrimIO (Either IOError a)
> data Either a b = Left a | Right b
>
> data IOError   = WriteError    String
>                | ReadError     String
>                | SearchError   String
>                | FormatError   String
>                | OtherError    String
>                | EOF
```

```
>
> showError      :: IOError -> String
> showError (ReadError s)    =  "Read Error: " ++ s
> showError (WriteError s)   =  "Write Error: " ++ s
> showError (FormatError s) =  "Format Error: " ++ s
> showError (SearchError s) =  "Search Error: " ++ s
> showError (OtherError s)   =  "Other Error: " ++ s
> showError (EOF)            =  "End of IO Handle"
```

Type `IOError` is the same as in Haskell 1.2, except it includes a new constructor `EOF`; `IOError` provides a standard format for system–generated errors. Users can generate errors of the form `OtherError` *msg*. An expression of type `IO` *a*, for some type *a*, denotes a computation whose answer (if any is returned) is either a result of type *a* or an *error* of type `IOError`. We say that the computation *fails with x* if its answer is `Left` *x*. It is worth emphasising that the type system delimits the possibility of failure: only expressions of some type `IO` *a* can fail.

To derive operations on the `IO` monad we begin with a combinator for defining functions on types of the form `Either` *a b*.

```
> either :: (a -> c) -> (b -> c) -> (Either a b) -> c
> either f g (Left x)  = f x
> either f g (Right x) = g x
```

Now we can derive monadic operations `return` and >>= (corresponding to `returnPrimIO` and `thenPrimIO` respectively)

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

and a complementary pair for failing with and recovering from errors

```
> failwith       :: IOError -> IO a
> try            :: IO a -> (IOError -> IO a) -> IO a
```

as follows.

```
> return x       = returnPrimIO (Right x)
> p >>= f        = p `thenPrimIO` either failwith f
> failwith x     = returnPrimIO (Left x)
> try p f        = p `thenPrimIO` either f return
```

Structure (`IO`, `return`, >>=) is a monad in the same sense as the `PrimIO` structure. The following restricted form of >>=,

```
> (>>)           :: IO a -> IO b -> IO b
> p >> q         = p >>= const q
```

is useful for sequential composition, when we are uninterested in the result of the first computation. Here is the fixity of these two infix operators.

```
> infixr 1 >>, >>=
```

As a convention for user–generated errors, to return an error message *msg* `::` `String`, return the error value `OtherError` *msg* via the computation `fail` *msg*.

```
> fail :: String -> IO a
> fail = failwith . OtherError
```

We recommend that `fail` be used instead of Haskell's `error :: String -> a` operation wherever convenient. Recall that the expression `error` *x* is intended to have the same semantics as a divergent program ("bottom"), but implementations are to abort with an error message.

4 Handles

The Haskell 1.2 model of the external world includes both persistent *files* and ephemeral *channels*. The file system is a mapping from file names to file contents, that persists between program executions. On the other hand, on each program execution there is a mapping between channel names and sequences of values, the channel contents for that execution. File and channel names are values of type `String`. File and channel contents are values of type `String`. There are several standard channels for input and output. Programs can open new anonymous channels for interaction with other operating system processes.

Haskell 1.2 input operations returned streams, that is, `ReadFile` and `ReadChan` returned a `String` representing either the contents of the file or the whole series of values communicated on the channel throughout the program run. Instead, we propose to read and write finite sequences of characters from/to files and channels, represented by values of type `Handle`. Each value of this type is a *handle*: a record used by the Haskell run−time system to *manage* I/O with files and channels.

A handle has the following properties:

- a string name;
- whether it manages input or output or both;
- whether it is open, closed or semi−closed;
- whether it manages a file or a channel and, if the former, a position and length of the file;
- whether buffering is disabled, or enabled on a line or block basis;
- whether input echoing is enabled;
- a buffer (whose length may be zero).

We say that a handle is *readable* if it manages only input or both input and output; likewise, it is *writable* if it manages only output or both input and output. A handle is open when first allocated. Once it is closed it can no longer be used for input or output. The operation `getContents` puts it into an intermediate state, 'semi−closed', during which it is effectively closed but its input values are accumulated by demand in a special stream returned by `getContents`. Any operation except for `close` that fails because a handle is closed, also fails if a handle is semi−closed. A semi−closed handle becomes closed either if the `close` operation is applied to it (at which point the contents of the associated stream becomes fixed as if end−of−file had been encountered), or once all the characters available have been accumulated in its special stream. After a handle has been closed, an implementation cannot re−use its storage while references remain to it.

4.1 Standard Handles

Four handles are allocated during program initialisation.

```
> stdin, stdout, stderr, stdnull :: Handle
```

The first two manage input or output from the Haskell program's standard input or output channel respectively. The third manages output to the standard error channel. The fourth is a bidirectional handle. Input from `stdnull` always fails; output to it is always accepted but is not externally observable. The four handles are initially open.

4.2 Opening and Closing Handles

```
> data IOMode   = ReadMode | WriteMode | AppendMode
> openFile      :: IOMode -> String -> IO Handle
> openChan      :: String -> IO Handle
> flush         :: Handle -> IO ()
> close         :: Handle -> IO ()
```

Computation `openFile` *mode file* allocates and returns a new, open, input−only or output−only handle to manage the file *file*. It manages reading if *mode* is `ReadMode`, otherwise writing. In the latter case, if the file does not exist, it should be created in the file system with zero length. If *mode* is `WriteMode` and the file already exists, then it should be truncated to zero length. The handle is positioned at the end of the file if *mode* is `AppendMode`, otherwise at the beginning. The computation fails with

- `SearchError` *msg* if *mode* is `WriteMode` or `AppendMode` and file *file* does not exist and cannot be created;
- `SearchError` *msg* if *mode* is `ReadMode` and file *file* does not exist;
- `WriteError` *msg* if *mode* is `WriteMode` or `AppendMode` and the the handle cannot be allocated for some other reason;
- `ReadError` *msg* if *mode* is `ReadMode` and the handle cannot be allocated.

Implementations should enforce, locally to the Haskell process, multiple–reader single–writer locking on files, which is to say that there may either be many handles on the same file for reading, or just one handle on the file for writing. If any open or semi–closed handle is managing a file for writing, no new handle can be allocated for that file. If any open or semi–closed handle is managing a file for reading, only new handles with mode `ReadOnly` can be allocated for that file. Although the property of two files being the same may vary between implementations, a sufficient but not necessary condition is for two files to have the same name (e.g., pathname).

Computation `openChan` *chan* allocates and returns a new handle to manage the channel identified by string *chan*. If *chan* is the empty string `""` the computation returns a bidirectional handle managing a new anonymous channel, whose input and output are connected. Such a channel is intended for interaction with an operating system process created using `runProcess`. Interpretation of non–empty strings is implementation–dependent; no two handles may be opened with the same non–empty string. The computation fails with

- `SearchError` *msg* if channel *chan* cannot be opened or is already managed by a handle.

Computation `flush` *hdl* causes any characters buffered for output in handle *hdl* to be sent immediately to the operating system. The computation fails with

- `WriteError` *msg* if *hdl* is either not writable or closed.

Computation `close` *hdl* makes handle *hdl* closed. Before the computation finishes, any characters buffered for output and not already sent to the operating system are flushed as for `flush`. The computation fails with

- `OtherError` *msg* if handle *hdl* is closed.

4.3 Character Input via Handles

```
> ready          :: Handle -> IO Bool
> hGetChar       :: Handle -> IO Char
> getContents    :: Handle -> IO String
> getChar        :: IO Char
```

Computation `ready` *hdl* indicates whether at least one character is available for input from handle *hdl*. It fails with

- `ReadError` *msg* if *hdl* is closed or not readable.

Computation `hGetChar` *hdl* reads the next character from handle *hdl*, blocking until a character is available. It fails with

- `EOF` if no further input is possible from *hdl*;
- `ReadError` *msg* if *hdl* is closed or not readable.

Computation `getContents` *hdl* returns the list of characters corresponding to the unread portion of the channel or file managed by *hdl*, which is made semi–closed. It fails with

- `ReadError` *msg* if *hdl* is already closed (or semi–closed).

We derive a specialised form of `hGetChar` to act on standard input.

```
> getChar        = hGetChar stdin
```

4.4 Character Output via Handles

```
> hPutChar       :: Handle -> Char -> IO ()
> hPutStr        :: Handle -> String -> IO ()
> hPutText       :: Text a => Handle -> a -> IO ()
> putChar        :: Char -> IO ()
> putStr         :: String -> IO ()
> putText        :: Text a => a -> IO ()
```

Computation `hPutChar` *hdl c* writes the character *c* to the file or channel managed by *hdl*. Characters may be buffered if buffering is enabled for *hdl*. The computation fails with

- `WriteError` *msg* if *hdl* is either not writable or not open.

Combinator `hPutStr` may be implemented as a primitive, but in principle is derivable as follows.

```
> hPutStr handle = foldr (>>) (return ()) . map (hPutChar handle)
```

Combinator `hPutText` admits output of any type in the `Text` class.

```
> hPutText hdl  = hPutStr hdl . show
```

We derive the following specialised operations acting on standard output.

```
> putChar        = hPutChar stdout
> putStr         = hPutStr stdout
> putText        = hPutText stdout
```

4.5 Buffering and Echoing

```
> setBuffering  :: Handle -> BufferMode -> IO ()
> data BufferMode = UnBuffered | LineBuffered | BlockBuffered
> setEchoing    :: Handle -> Bool -> IO ()
```

Three kinds of buffering are supported: handles may be line–buffered, block–buffered or unbuffered. These modes have the following effects. For output, characters are written out from the internal buffer according to the buffer mode:

line–buffered
: the entire output buffer is written out whenever a newline is output, the output buffer overflows, a flush is issued, or the handle is closed.

block–buffered
: the entire output buffer is written out whenever it overflows, a flush is issued, or the handle is closed.

unbuffered
: characters are written immediately, and never stored in the output buffer.

The output buffer is emptied as soon as it has been written out.

Similarly, input occurs as a result of `hGetChar` *hdl* according to the buffer mode for handle *hdl*.

line–buffered

when the input buffer for *hdl* is not empty, the next character is obtained from the buffer; otherwise, when the input buffer is empty, n characters up to and including the next newline character are read into the buffer. No characters are available until the newline character is available.

block–buffered

when the input buffer for *hdl* becomes empty, the next block of data is read into this buffer.

unbuffered

the next character is read and returned.

The `hGetChar` operation always blocks if no input is available. For most implementations, files will normally be block–buffered and terminals will normally be line–buffered. The size of the buffer is implementation–dependent.

Computation `setBuffering` *hdl mode* sets the mode of buffering for handle *hdl* on subsequent reads and writes.

- If *mode* is `LineBuffered`, line–buffering should be enabled. The `BufferMode` returned by `query` is `LineBuffered`.
- If *mode* is `BlockBuffered`, then either block–buffering or line–buffering should be enabled as appropriate for the device attached to *hdl*. The `BufferMode` returned by `query` is `LineBuffered` if line–buffering is enabled, otherwise `BlockBuffered`.
- If *mode* is `UnBuffered`, then buffering is disabled for *hdl*. The `BufferMode` returned by `query` is `UnBuffered`.

If the buffer mode is changed from `BlockBuffered` or `LineBuffered` to `UnBuffered`, then any characters in the output buffer are written to the device, and any characters in the input buffer are discarded. The default buffering mode when a handle is opened is `UnBuffered` if buffering is not supported for the device, otherwise the mode that would be set for `BlockBuffered`. This is the `BufferMode` returned by `query`. The computation fails with

- `WriteError` *msg* if the handle is not open for writing, if the device attached to *hdl* does not support the requested mode, or if some other error occurs.

An implementation is free to fail with `WriteError` *msg* if the handle has already been read from or written to.

Computation `setEchoing` *hdl flag* enables echoing on input from handle *hdl* if *flag* is `True`, and disables it if *flag* is `False`. Echoed characters appear on the device from which input for *hdl* is taken. The computation fails with

- `OtherError` *msg* if handle *hdl* is not open for input from a user terminal, or if echoing is not supported for *hdl*.

Similarly, an implementation is free to fail with `WriteError` *msg* if the handle has already been read from or written to.

4.6 Landin–stream I/O

```
> interact :: (String -> String) -> IO ()
> interact f = getContents stdin >>= (putStr . f)
```

Derived combinator `interact` supports classical Landin–stream character I/O.

4.7 String–based file operations

```
> readFile :: String -> IO String
> readChan :: String -> IO String
> appendFile :: String -> String -> IO ()
> appendChan :: String -> String -> IO ()
> writeFile :: String -> String -> IO ()
```

We can simulate some of the Haskell 1.2 string–based file operations as follows.

```
> readFile nme = openFile ReadMode nme >>= getContents
>
> readChan "stdin" = getContents stdin
> readChan nme     = openChan nme >>= getContents
>
> appendFile nme str =
>   openFile AppendMode nme >>= \hdl -> hPutStr hdl str >> close hdl
>
> appendChan "stdout" = hPutStr stdout
> appendChan "stderr" = hPutStr stderr
> appendChan _        = const (fail "Unexpected 1.2 channel")
>
> writeFile nme str =
>   openFile WriteMode nme >>= \hdl -> hPutStr hdl str >> close hdl
```

4.8 Handle Access

```
> seek  :: Handle -> Integer -> IO ()
> query :: Handle -> IO HandleState
> data HandleState = HandleState String HandleKind
>                          OpenClosed FileChan BufferMode Bool
> data HandleKind = InputOnly | OutputOnly | InputOutput
> data OpenClosed = IsOpen | IsClosed | IsSemiClosed
> data FileChan   = IsChannel | IsFile Integer Integer   -- size, posn
```

Computation `seek` *handle i* sets the position of handle *hdl* to integer *i*. The size of the file is extended to at least the position requested if it is open for writing and position *i* is beyond its end. When first opened a handle is at position 0. It fails with

- `SeekError` *msg* if *hdl* does not manage a file, or if both *i* is beyond the end of the file and the handle is read–only, or if *i* is negative.

Computation `query` *hdl* returns a value `HandleState` *nme kind oc fc mode echo* where *nme* is the handle name and *kind* indicates whether it manages input, output or both, *oc* whether it is opened or closed, *fc* whether it manages a file or channel, *mode* its buffering mode and *echo* whether echoing is enabled.

4.9 Select

```
> type SelectData = ([Handle], [Handle], Maybe Integer)
> select :: SelectData -> Maybe SelectData
```

`SelectData` consists of a set of input handles, a set of output handles, and an optional time interval.

Computation `select` (*ihs*, *ohs*, *mb*) waits for input or output to be possible on handles *ihs* or *ohs* respectively. If *mb* is `Just` *i* the computation waits at most *i* microseconds. When I/O becomes possible on any of the handles the computation returns the sets of handles on which I/O is possible. If an interval `Maybe` *i* was given the remaining time of that is returned as well. If the timeout expires with I/O still not possible the computation returns `Nothing`.

Implementation notes:

- Using `select` on buffered handles may cause unpredictable results unless the implementation is careful.
- Returning the remaining time interval can be problematic in UNIX version where `select` is not properly implemented, but you can still do it (by calling `gettimeofday` before and after `select`).

5 Operating System Interaction

```
> deleteFile    :: String -> IO ()
> statusFile    :: String -> IO String
> statusChan    :: String -> IO String
> getArgs       :: IO [String]
> getProgName   :: IO String
> getEnv        :: String -> IO String
> setEnv        :: String -> String -> IO ()
> getClock      :: IO Integer
> getCpuTime    :: IO Integer
> runProcess    :: String -> [Handle] -> IO ()
> system        :: String -> IO String
```

Computation `deleteFile` *file* deletes file *file*. Computations `statusFile` *file* and `statusChan` *chan* return status strings for file *file* and channel *chan*, as indicated in the Haskell 1.2 report (Hudak 92). These computations fails with

- `SeekError` *msg* if file *file* or channel *chan* does not exist or, in the case of `deleteFile` *file*, cannot be deleted.

Computation `getArgs` returns a list of the program's command line arguments (not including the program name).

Computation `getProgName` returns the name of the program.

Computation `getEnv` *var* returns the value of the environment variable *var*. It fails with

- `SeekError` *msg* if variable *var* does not exist.

Computation `setEnv` *var val* sets environment variable *var* to new value *val*. Variable *var* is created if it does not already exist.

Computation `getClock` returns the number of microseconds since midnight, GMT, on January 1 1970.

Computation `getCpuTime` returns the number of microseconds of cpu time devoted to the program so far.

Computation `runProcess` *cmd hdls* creates a new operating system process executing command *cmd*. If they are given, the first three handles in *hdls* are connected to the new process's standard input, output and error channels, respectively. The first of these must be readable and the next two must be writable. The remaining handles may be connected to other process channels in an implementation–dependent way. The computation fails with

- `OtherError` *msg* if the process cannot be created.

Derived computation `system` *cmd* returns the output produced when command *cmd* is run as a separate operating system process.

```
> system cmd    =
>   openChan "" >>= \handle ->
>   runProcess cmd [stdnull, handle, stdnull] >>
>   getContents handle
```

5.1 Interrupts

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

Whenever a console interrupt occurs, the executing program is to be stopped and the *interrupt handler*, by default equal to the computation `return ()`, executed in its place. Computation `setInterrupt` *p* changes this default behaviour by installing *p* as the interrupt handler; it returns the previous interrupt handler as its answer.

6 Monadic combinator

```
> accumulate  :: [IO a] -> IO [a]
> accumulate  =
>   foldr mcons (return [])
>     where
>       mcons :: IO a -> IO [a] -> IO [a]
>       mcons p q = p >>= \x -> q >>= \y -> return (x : y)
>
> accumulate_ :: [IO a] -> IO ()
> accumulate_ = foldr (>>) (return ())
```

Computation `accumulate` *ps* runs each of the computations in *ps* in sequence, and accumulates the results, which are returned as a list. Computation `accumulate_` *ps* runs the same computations, but discards the results and returns `()`.

7 Changes to Haskell 1.2

We propose that the `PreludeIO` module be removed from Haskell, and that a `PreludeMonadicIO` module, whose header is given in Section 2, be inserted in its place. A Haskell program is determined by the identifier `main` in module `Main`, whose type must be `IO ()`, that is, `PrimIO (Either IOError ())`. The Haskell program begins with computation `main`. If `main` returns answer `Right ()`, the Haskell program immediately terminates, indicating successful termination to the operating system. If `main` returns answer `Left` *error*, the Haskell program displays a textual form of the error *error* and then terminates, indicating unsuccessful termination to the operating system.

The exact mechanism is likely to be implementation dependent, but we encourage systems supporting Haskell 1.2 I/O to continue to do so via some form of compatibility module or command−line flag.

Here is a summary of the main Haskell 1.2 I/O operations that cannot be directly expressed in the proposed I/O mechanism.

- We omit the `Dialogue` type and stream−based I/O. Continuation−passing I/O can be simulated using computations of type `IO ()`.
- We omit I/O of `Bin` values, but see the proposal of a new `Binary` class.
- We drop the `echo` channel (although echoing is supported on `stdin` via `setEcho`) and do not introduce any monadic operations corresponding to the optional 1.2 requests. These requests have not been widely implemented.

Bibliography

John Cupitt, **A brief walk through KAOS**. Technical Report 58, Computing Laboratory, University of Kent at Canterbury, February 1989.

Andrew D. Gordon, **Functional Programming and Input/Output**. PhD dissertation, University of Cambridge, August 1992. To be published by Cambridge University Press.

Paul Hudak and Simon L. Peyton Jones and Philip Wadler et al, Report on the Functional Programming Language Haskell: A Non−strict, Purely Functional Language Version 1.2. **SIGPLAN Notices**, 27(5) (ACM, March 1992).

John Launchbury and Simon L. Peyton Jones, Lazy Functional State Threads. **Proceedings Programming Language Design and Implementation 1994,** (to appear).

Simon L. Peyton Jones and Philip Wadler, Imperative functional programming. **Proceedings 20th ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993** (ACM, 1993).

Philip Wadler, The essence of functional programming. **Proceedings of the 19th ACM Symposium on Principles of Programming Languages, January 1992** (ACM, 1992).

Appendix: Unresolved Issues

We are considering the following extensions to this proposal. We haven't worked out all the details. We would like to include some form of each of these proposals in the final 1.3 Definition, but are reluctant to do so without implementation experience. Therefore we'd strongly encourage experimentation!

Binary Input/Output

Haskell 1.2 called for a `Bin` datatype, whose elements were suitable for "binary" I/O using specialised versions of the file and channel I/O operations. Values of other types could be embedded as values of type `Bin`. We want to support binary input/output of data, but it is awkward to implement the `Bin`. Instead we are considering a new class, `Binary`,

```
> class Binary a where
>   putBin :: Handle -> a -> IO ()
>   getBin :: Handle -> IO a
```

whose instances would include (at least) `Float`. Full details of this scheme remain to be worked out.

Blocking `runProcess`

There is no way at present to examine the return code of an operating system process started by a Haskell program. One way to do so would be to include a primitive operation

```
  runProcessNow :: String -> [Handle] -> IO Bool
```

with the same behaviour as `runProcess`, except that the computation blocks until the spawned process has terminated and then its answer indicates whether termination was successful.

More Flexible Interrupts

We are considering a more flexible interrupt mechanism (suggested by Jim Mattson) in which there is an enumerated type of possible interrupts,

```
data Interrupt = InternalAbort
                 | RealTimeAlarm
                 | FloatingPointException
                 | LostConnection
                 | IllegalInstruction
                 | KeyboardInterrupt
                 | OpenEndedPipe
                 | KeyboardTermination
                 | SegmentationViolation
                 | SoftwareTermination
                 | UserDefinedSignal Int
```

to each of which is associated a handler of type `IO ()`. Each default handler is `failwith "Interrupt!"`, but given

```
setHandler :: Interrupt -> IO () -> IO (IO ())
```

the handler for interrupt *i* may be updated to be *h* by the computation `setHandler` *i h*, which returns the previous handler as its answer. Finally given

```
unsetHandler :: Interrupt -> IO (IO ())
```

the computation `unsetHandler` *i* disables *i* interrupts, and returns the previously installed handler.

Exposing `PrimIO`

We're considering making global the `PrimIO` type and its values `returnPrimIO` and `thenPrimIO`. The advantage would be that monads could be freely built on top of `PrimIO` without the failure–handling baggage of `IO`. The disadvantage would be that efficient implementations of `IO` may be hampered if it is no longer an abstract type.

Please ignore this section!

> type Integer = Int > data PrimIO a = PrimIO a > data Handle = Handle > (returnPrimIO, thenPrimIO, stdin, stdout, stderr, stdnull, > openFile, openChan, flush, close, ready, hGetChar, getContents, hPutChar, > setBuffering, seek, query, select, getClock, runProcess, deleteFile, > statusFile, statusChan, getArgs, getProgName, allowInterrupt, > getEnv, setEnv, select) = x where x=x

---

Recent Changes to the Proposal

940706
    Fixed some minor glitches reported by Jim Mattson.
940708
    Fixed infelicitous namings following complaints from Jon Fairbairn (including renaming the constructors of type `BufferMode`).
940718
    Restored main programs to type `IO ()`.
    `(..)` added to algebraic datatypes in the module header, to export their (data) constructors.

A Proposal for Monadic I/O in Haskell 1.3

Haskell 1.3 Committee
haskell1.3@comp.vuw.ac.nz