JFP **13** (1): 205–218, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803002314 Printed in the United Kingdom

Chapter 21

Input/Output

```
module IO (
    Handle, HandlePosn,
    IOMode(ReadMode,WriteMode,AppendMode,ReadWriteMode),
    BufferMode(NoBuffering,LineBuffering,BlockBuffering),
    SeekMode(AbsoluteSeek,RelativeSeek,SeekFromEnd),
    stdin, stdout, stderr,
    openFile, hClose, hFileSize, hIsEOF, isEOF,
    hSetBuffering, hGetBuffering, hFlush,
    hGetPosn, hSetPosn, hSeek,
    hWaitForInput, hReady, hGetChar, hGetLine, hLookAhead, hGetContents,
    hPutChar, hPutStr, hPutStrLn, hPrint,
    hIsOpen, hIsClosed, hIsReadable, hIsWritable, hIsSeekable,
    isAlreadyExistsError, isDoesNotExistError, isAlreadyInUseError,
    isFullError, isEOFError,
    isIllegalOperation, isPermissionError, isUserError,
    ioeGetErrorString, ioeGetHandle, ioeGetFileName,
    try, bracket, bracket_,
    -- ...and what the Prelude exports
    IO, FilePath, IOError, ioError, userError, catch, interact,
    putChar, putStr, putStrLn, print, getChar, getLine, getContents,
    readFile, writeFile, appendFile, readIO, readLn
    ) where

import Ix(Ix)
```

```
data Handle = ...                          -- implementation-dependent
instance Eq Handle where ...
instance Show Handle where ..              -- implementation-dependent

data HandlePosn = ...                      -- implementation-dependent
instance Eq HandlePosn where ...
instance Show HandlePosn where ---         -- implementation-dependent

data IOMode      =  ReadMode | WriteMode | AppendMode | ReadWriteMode
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
data BufferMode  =  NoBuffering | LineBuffering
                 |  BlockBuffering (Maybe Int)
                    deriving (Eq, Ord, Read, Show)
data SeekMode    =  AbsoluteSeek | RelativeSeek | SeekFromEnd
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)

stdin, stdout, stderr :: Handle

openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()

hFileSize             :: Handle -> IO Integer
hIsEOF                :: Handle -> IO Bool
isEOF                 :: IO Bool
isEOF                 =  hIsEOF stdin

hSetBuffering         :: Handle  -> BufferMode -> IO ()
hGetBuffering         :: Handle  -> IO BufferMode
hFlush                :: Handle -> IO ()
hGetPosn              :: Handle -> IO HandlePosn
hSetPosn              :: HandlePosn -> IO ()
hSeek                 :: Handle -> SeekMode -> Integer -> IO ()

hWaitForInput         :: Handle -> Int -> IO Bool
hReady                :: Handle -> IO Bool
hReady h              =  hWaitForInput h 0
hGetChar              :: Handle -> IO Char
hGetLine              :: Handle -> IO String
hLookAhead            :: Handle -> IO Char
hGetContents          :: Handle -> IO String
hPutChar              :: Handle -> Char -> IO ()
hPutStr               :: Handle -> String -> IO ()
hPutStrLn             :: Handle -> String -> IO ()
hPrint                :: Show a => Handle -> a -> IO ()
```

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

```
hIsOpen               :: Handle -> IO Bool
hIsClosed             :: Handle -> IO Bool
hIsReadable           :: Handle -> IO Bool
hIsWritable           :: Handle -> IO Bool
hIsSeekable           :: Handle -> IO Bool

isAlreadyExistsError  :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isAlreadyInUseError   :: IOError -> Bool
isFullError           :: IOError -> Bool
isEOFError            :: IOError -> Bool
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool

ioeGetErrorString     :: IOError -> String
ioeGetHandle          :: IOError -> Maybe Handle
ioeGetFileName        :: IOError -> Maybe FilePath

try                   :: IO a -> IO (Either IOError a)
bracket               :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_              :: IO a -> (a -> IO b) -> IO c -> IO c
```

The monadic I/O system used in Haskell is described by the Haskell language report. Commonly used I/O functions such as `print` are part of the standard prelude and need not be explicitly imported. This library contain more advanced I/O features. Some related operations on file systems are contained in the `Directory` library.

21.1 I/O Errors

Errors of type `IOError` are used by the I/O monad. This is an abstract type; the library provides functions to interrogate and construct values in `IOError`:

- `isAlreadyExistsError` – the operation failed because one of its arguments already exists.
- `isDoesNotExistError` – the operation failed because one of its arguments does not exist.
- `isAlreadyInUseError` – the operation failed because one of its arguments is a single-use resource, which is already being used (for example, opening the same file twice for writing might give this error).
- `isFullError` – the operation failed because the device is full.
- `isEOFError` – the operation failed because the end of file has been reached.
- `isIllegalOperation` – the operation is not possible.

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

- `isPermissionError` – the operation failed because the user does not have sufficient operating system privilege to perform that operation.
- `isUserError` – a programmer-defined error value has been raised using `fail`.

All these functions return a `Bool`, which is `True` if its argument is the corresponding kind of error, and `False` otherwise.

Any computation which returns an `IO` result may fail with `isIllegalOperation`. Additional errors which could be raised by an implementation are listed after the corresponding operation. In some cases, an implementation will not be able to distinguish between the possible error causes. In this case it should return `isIllegalOperation`.

Three additional functions are provided to obtain information about an error value. These are `ioeGetHandle` which returns `Just` hdl if the error value refers to handle hdl and `Nothing` otherwise; `ioeGetFileName` which returns `Just` $name$ if the error value refers to file $name$, and `Nothing` otherwise; and `ioeGetErrorString` which returns a string. For "user" errors (those which are raised using `fail`), the string returned by `ioeGetErrorString` is the argument that was passed to `fail`; for all other errors, the string is implementation-dependent.

The `try` function returns an error in a computation explicitly using the `Either` type.

The `bracket` function captures a common allocate, compute, deallocate idiom in which the deallocation step must occur even in the case of an error during computation. This is similar to try-catch-finally in Java.

21.2 Files and Handles

Haskell interfaces to the external world through an abstract *file system*. This file system is a collection of named *file system objects*, which may be organised in *directories* (see `Directory`). In some implementations, directories may themselves be file system objects and could be entries in other directories. For simplicity, any non-directory file system object is termed a *file*, although it could in fact be a communication channel, or any other object recognised by the operating system. *Physical files* are persistent, ordered files, and normally reside on disk.

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

Haskell defines operations to read and write characters from and to files, 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 file system objects. A handle has at least the following properties:

- whether it manages input or output or both;

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

- whether it is *open*, *closed* or *semi-closed*;
- whether the object is seekable;
- whether buffering is disabled, or enabled on a line or block basis;
- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur. 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 either input or output, though an implementation cannot re-use its storage while references remain to it. Handles are in the `Show` and `Eq` classes. The string produced by showing a handle is system dependent; it should include enough information to identify the handle for debugging. A handle is equal according to `==` only to itself; no attempt is made to compare the internal state of different handles for equality.

21.2.1 Standard Handles

Three handles are allocated during program initialisation. The first two (`stdin` and `stdout`) manage input or output from the Haskell program's standard input or output channel respectively. The third (`stderr`) manages output to the standard error channel. These handles are initially open.

21.2.2 Semi-Closed Handles

The operation `hGetContents` hdl (Section 21.9.4) puts a handle hdl into an intermediate state, *semi-closed*. In this state, hdl is effectively closed, but items are read from hdl on demand and accumulated in a special list returned by `hGetContents` hdl.

Any operation that fails because a handle is closed, also fails if a handle is semi-closed. The only exception is `hClose`. A semi-closed handle becomes closed:

- if `hClose` is applied to it;
- if an I/O error occurs when reading an item from the handle;
- or once the entire contents of the handle has been read.

Once a semi-closed handle becomes closed, the contents of the associated list becomes fixed. The contents of this final list is only partially specified: it will contain at least all the items of the stream that were evaluated prior to the handle becoming closed.

Any I/O errors encountered while a handle is semi-closed are simply discarded.

21.2.3 File Locking

Implementations should enforce as far as possible, at least locally to the Haskell process, multiple-reader single-writer locking on files. That is, *there may either be many handles on the same file which manage input, or just one handle on the file which manages output*. If any open or semi-closed handle is managing a file for output, no new handle can be allocated for that file. If any open or semi-closed handle is managing a file for input, new handles can only be allocated if they do not manage output. Whether two files are the same is implementation-dependent, but they should normally be the same if they have the same absolute path name and neither has been renamed, for example.

Warning: the `readFile` operation (Section 7.1 of the Haskell Language Report) holds a semi-closed handle on the file until the entire contents of the file have been consumed. It follows that an attempt to write to a file (using `writeFile`, for example) that was earlier opened by `readFile` will usually result in failure with `isAlreadyInUseError`.

21.3 Opening and Closing Files

21.3.1 Opening Files

Computation `openFile` $file$ $mode$ allocates and returns a new, open handle to manage the file $file$. It manages input if $mode$ is `ReadMode`, output if $mode$ is `WriteMode` or `AppendMode`, and both input and output if mode is `ReadWriteMode`.

If the file does not exist and it is opened for output, it should be created as a new file. If $mode$ is `WriteMode` and the file already exists, then it should be truncated to zero length. Some operating systems delete empty files, so there is no guarantee that the file will exist following an `openFile` with $mode$ `WriteMode` unless it is subsequently written to successfully. The handle is positioned at the end of the file if $mode$ is `AppendMode`, and otherwise at the beginning (in which case its internal I/O position is 0). The initial buffer mode is implementation-dependent.

If `openFile` fails on a file opened for output, the file may still have been created if it did not already exist.

Error reporting: the `openFile` computation may fail with `isAlreadyInUseError` if the file is already open and cannot be reopened; `isDoesNotExistError` if the file does not exist; or `isPermissionError` if the user does not have permission to open the file.

21.3.2 Closing Files

Computation `hClose` hdl makes handle hdl closed. Before the computation finishes, if hdl is writable its buffer is flushed as for `hFlush`. Performing `hClose` on a handle that has already been closed has no effect; doing so not an error. All other operations on a closed handle will fail. If

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

`hClose` fails for any reason, any further operations (apart from `hClose`) on the handle will still fail as if hdl had been successfully closed.

21.4 Determining the Size of a File

For a handle hdl which is attached to a physical file, `hFileSize` hdl returns the size of that file in 8-bit bytes (≥ 0).

21.5 Detecting the End of Input

For a readable handle hdl, computation `hIsEOF` hdl returns `True` if no further input can be taken from hdl; for a handle attached to a physical file this means that the current I/O position is equal to the length of the file. Otherwise, it returns `False`. The computation `isEOF` is identical, except that it works only on `stdin`.

21.6 Buffering Operations

Three kinds of buffering are supported: line-buffering, block-buffering or no-buffering. These modes have the following effects. For output, items are written out, or *flushed*, from the internal buffer according to the buffer mode:

- **line-buffering:** the entire buffer is flushed whenever a newline is output, the buffer overflows, a `hFlush` is issued, or the handle is closed.
- **block-buffering:** the entire buffer is written out whenever it overflows, a `hFlush` is issued, or the handle is closed.
- **no-buffering:** output is written immediately, and never stored in the buffer.

An implementation is free to flush the buffer more frequently, but not less frequently, than specified above. The buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for handle hdl.

- **line-buffering:** when the buffer for hdl is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters are read into the buffer until the next newline character is encountered or the buffer is full. No characters are available until the newline character is available or the buffer is full.

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

- **block-buffering:** when the buffer for hdl becomes empty, the next block of data is read into the buffer.
- **no-buffering:** the next input item is read and returned. The `hLookAhead` operation (Section 21.9.3) implies that even a no-buffered handle may require a one-character buffer.

For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

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

- If $mode$ is `LineBuffering`, line-buffering is enabled if possible.
- If $mode$ is `BlockBuffering` $size$, then block-buffering is enabled if possible. The size of the buffer is n items if $size$ is `Just` n and is otherwise implementation-dependent.
- If $mode$ is `NoBuffering`, then buffering is disabled if possible.

If the buffer mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

- if hdl is writable, the buffer is flushed as for `hFlush`;
- if hdl is not writable, the contents of the buffer is discarded.

Error reporting: the `hSetBuffering` computation may fail with `isPermissionError` if the handle has already been used for reading or writing and the implementation does not allow the buffering mode to be changed.

Computation `hGetBuffering` hdl returns the current buffering mode for hdl.

The default buffering mode when a handle is opened is implementation-dependent and may depend on the file system object which is attached to that handle.

21.6.1 Flushing Buffers

Computation `hFlush` hdl causes any items buffered for output in handle hdl to be sent immediately to the operating system.

Error reporting: the `hFlush` computation may fail with: `isFullError` if the device is full; `isPermissionError` if a system resource limit would be exceeded. It is unspecified whether the characters in the buffer are discarded or retained under these circumstances.

21.7 Repositioning Handles

21.7.1 Revisiting an I/O Position

Computation `hGetPosn` hdl returns the current I/O position of hdl as a value of the abstract type `HandlePosn`. If a call to `hGetPosn` h returns a position p, then computation `hSetPosn` p sets the position of h to the position it held at the time of the call to `hGetPosn`.

Error reporting: the `hSetPosn` computation may fail with: `isPermissionError` if a system resource limit would be exceeded.

21.7.2 Seeking to a New Position

Computation `hSeek` hdl $mode$ i sets the position of handle hdl depending on $mode$. If $mode$ is:

- `AbsoluteSeek`: the position of hdl is set to i.
- `RelativeSeek`: the position of hdl is set to offset i from the current position.
- `SeekFromEnd`: the position of hdl is set to offset i from the end of the file.

The offset is given in terms of 8-bit bytes.

If hdl is block- or line-buffered, then seeking to a position which is not in the current buffer will first cause any items in the output buffer to be written to the device, and then cause the input buffer to be discarded. Some handles may not be seekable (see `hIsSeekable`), or only support a subset of the possible positioning operations (for instance, it may only be possible to seek to the end of a tape, or to a positive offset from the beginning or current position). It is not possible to set a negative I/O position, or for a physical file, an I/O position beyond the current end-of-file.

Error reporting: the `hSeek` computation may fail with: `isPermissionError` if a system resource limit would be exceeded.

21.8 Handle Properties

The functions `hIsOpen`, `hIsClosed`, `hIsReadable`, `hIsWritable` and `hIsSeekable` return information about the properties of a handle. Each of these returns `True` if the handle has the specified property, and `False` otherwise.

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

21.9 Text Input and Output

Here we define a standard set of input operations for reading characters and strings from text files, using handles. Many of these functions are generalizations of Prelude functions. I/O in the Prelude generally uses `stdin` and `stdout`; here, handles are explicitly specified by the I/O operation.

21.9.1 Checking for Input

Computation `hWaitForInput` hdl t waits until input is available on handle hdl. It returns `True` as soon as input is available on hdl, or `False` if no input is available within t milliseconds.

Computation `hReady` hdl indicates whether at least one item is available for input from handle hdl.

Error reporting. The `hWaitForInput` and `hReady` computations fail with `isEOFError` if the end of file has been reached.

21.9.2 Reading Input

Computation `hGetChar` hdl reads a character from the file or channel managed by hdl.

Computation `hGetLine` hdl reads a line from the file or channel managed by hdl. The Prelude's `getLine` is a shorthand for `hGetLine stdin`.

Error reporting. The `hGetChar` computation fails with `isEOFError` if the end of file has been reached. The `hGetLine` computation fails with `isEOFError` if the end of file is encountered when reading the *first* character of the line. If `hGetLine` encounters end-of-file at any other point while reading in a line, it is treated as a line terminator and the (partial) line is returned.

21.9.3 Reading Ahead

Computation `hLookAhead` hdl returns the next character from handle hdl without removing it from the input buffer, blocking until a character is available.

Error reporting: the `hLookAhead` computation may fail with: `isEOFError` if the end of file has been reached.

21.9.4 Reading the Entire Input

Computation `hGetContents` hdl returns the list of characters corresponding to the unread portion of the channel or file managed by hdl, which is made semi-closed.

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

Error reporting: the `hGetContents` computation may fail with: `isEOFError` if the end of file has been reached.

21.9.5 Text Output

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.

Computation `hPutStr` hdl s writes the string s to the file or channel managed by hdl.

Computation `hPrint` hdl t writes the string representation of t given by the `shows` function to the file or channel managed by hdl and appends a newline.

Error reporting: the `hPutChar`, `hPutStr` and `hPrint` computations may fail with: `isFullError` if the device is full; or `isPermissionError` if another system resource limit would be exceeded.

21.10 Examples

Here are some simple examples to illustrate Haskell I/O.

21.10.1 Summing Two Numbers

This program reads and sums two `Integer`s.

```
import IO

main = do
         hSetBuffering stdout NoBuffering
         putStr   "Enter an integer: "
         x1 <- readNum
         putStr   "Enter another integer: "
         x2 <- readNum
         putStr  ("Their sum is " ++ show (x1+x2) ++ "\n")
       where readNum :: IO Integer
                -- Providing a type signature avoids reliance on
                -- the defaulting rule to fix the type of x1,x2
             readNum = readLn
```

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press

21.10.2 Copying Files

A simple program to create a copy of a file, with all lower-case characters translated to upper-case. This program will not allow a file to be copied to itself. This version uses character-level I/O. Note that exactly two arguments must be supplied to the program.

```
import IO
import System
import Char( toUpper )

main = do
         [f1,f2] <- getArgs
         h1 <- openFile f1 ReadMode
         h2 <- openFile f2 WriteMode
         copyFile h1 h2
         hClose h1
         hClose h2

copyFile h1 h2 = do
                   eof <- hIsEOF h1
                   if eof then return () else
                      do
                        c <- hGetChar h1
                        hPutChar h2 (toUpper c)
                        copyFile h1 h2
```

An equivalent but much shorter version, using string I/O is:

```
import System
import Char( toUpper )

main = do
         [f1,f2] <- getArgs
         s <- readFile f1
         writeFile f2 (map toUpper s)
```

21.11 Library IO

```
module IO {- export list omitted -} where

-- Just provide an implementation of the system-independent
-- actions that IO exports.

try            :: IO a -> IO (Either IOError a)
try f          =  catch (do r <- f
                            return (Right r))
                        (return . Left)
```

```
bracket        :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after m = do
        x  <- before
        rs <- try (m x)
        after x
        case rs of
           Right r -> return r
           Left  e -> ioError e

-- variant of the above where middle computation doesn't want x
bracket_        :: IO a -> (a -> IO b) -> IO c -> IO c
bracket_ before after m = do
         x  <- before
         rs <- try m
         after x
         case rs of
            Right r -> return r
            Left  e -> ioError e
```

https://doi.org/10.1017/S0956796803002314 Published online by Cambridge University Press