JFP **13** (1): 97–102, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803000911 Printed in the United Kingdom

Chapter 7

Basic Input/Output

The I/O system in Haskell is purely functional, yet has all of the expressive power found in conventional programming languages. To achieve this, Haskell uses a *monad* to integrate I/O operations into a purely functional context.

The I/O monad used by Haskell mediates between the *values* natural to a functional language and the *actions* that characterize I/O operations and imperative programming in general. The order of evaluation of expressions in Haskell is constrained only by data dependencies; an implementation has a great deal of freedom in choosing this order. Actions, however, must be ordered in a well-defined manner for program execution – and I/O in particular – to be meaningful. Haskell's I/O monad provides the user with a way to specify the sequential chaining of actions, and an implementation is obliged to preserve this order.

The term *monad* comes from a branch of mathematics known as *category theory*. From the perspective of a Haskell programmer, however, it is best to think of a monad as an *abstract datatype*. In the case of the I/O monad, the abstract values are the *actions* mentioned above. Some operations are primitive actions, corresponding to conventional I/O operations. Special operations (methods in the class `Monad`, see Section 6.3.6) sequentially compose actions, corresponding to sequencing operators (such as the semicolon) in imperative languages.

7.1 Standard I/O Functions

Although Haskell provides fairly sophisticated I/O facilities, as defined in the `IO` library, it is possible to write many Haskell programs using only the few simple functions that are exported from the Prelude, and which are described in this section.

All I/O functions defined here are character oriented. The treatment of the newline character will vary on different systems. For example, two characters of input, return and linefeed, may read as a single newline character. These functions cannot be used portably for binary I/O.

In the following, recall that `String` is a synonym for `[Char]` (Section 6.1.2).

Output Functions These functions write to the standard output device (this is normally the user's terminal).

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()  -- adds a newline
print    :: Show a => a -> IO ()
```

The `print` function outputs a value of any printable type to the standard output device. Printable types are those that are instances of class `Show`; `print` converts values to strings for output using the `show` operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

Input Functions These functions read input from the standard input device (normally the user's terminal).

```
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String
interact    :: (String -> String) -> IO ()
readIO      :: Read a => String -> IO a
readLn      :: Read a => IO a
```

The `getChar` operation raises an exception (Section 7.3) on end-of-file; a predicate `isEOFError` that identifies this exception is defined in the `IO` library. The `getLine` operation raises an exception under the same circumstances as `hGetLine`, defined the `IO` library.

The `getContents` operation returns all user input as a single string, which is read lazily as it is needed. The `interact` function takes a function of type `String->String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

Typically, the `read` operation from class `Read` is used to convert the string to a value. The `readIO` function is similar to `read` except that it signals parse failure to the I/O monad instead of terminating the program. The `readLn` function combines `getLine` and `readIO`.

The following program simply removes all non-ASCII characters from its standard input and echoes the result on its standard output. (The `isAscii` function is defined in a library.)

```
main = interact (filter isAscii)
```

Files These functions operate on files of characters. Files are named by strings using some implementation-specific method to resolve strings as file names.

The `writeFile` and `appendFile` functions write or append the string, their second argument, to the file, their first argument. The `readFile` function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with `getContents`.

```
type FilePath =  String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO String
```

Note that `writeFile` and `appendFile` write a literal string to a file. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first.

```
main = appendFile "squares" (show [(x,x*x) | x <- [0,0.1..2]])
```

7.2 Sequencing I/O Operations

The type constructor `IO` is an instance of the `Monad` class. The two monadic binding functions, methods in the `Monad` class, are used to compose a series of I/O operations. The >> function is used where the result of the first operation is uninteresting, for example when it is `()`. The >>= operation passes the result of the first operation as an argument to the second operation.

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

For example,

```
main = readFile "input-file"                       >>= \ s ->
       writeFile "output-file" (filter isAscii s)  >>
       putStr "Filtering successful\n"
```

is similar to the previous example using `interact`, but takes its input from `"input-file"` and writes its output to `"output-file"`. A message is printed on the standard output before the program completes.

https://doi.org/10.1017/S0956796803000911 Published online by Cambridge University Press

The `do` notation allows programming in a more imperative syntactic style. A slightly more elaborate version of the previous example would be:

```
main = do
        putStr "Input file: "
        ifile <- getLine
        putStr "Output file: "
        ofile <- getLine
        s <- readFile ifile
        writeFile ofile (filter isAscii s)
        putStr "Filtering successful\n"
```

The `return` function is used to define the result of an I/O operation. For example, `getLine` is defined in terms of `getChar`, using `return` to define the result:

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then return ""
                          else do s <- getLine
                                  return (c:s)
```

7.3 Exception Handling in the I/O Monad

The I/O monad includes a simple exception handling system. Any I/O operation may raise an exception instead of returning a result.

Exceptions in the I/O monad are represented by values of type `IOError`. This is an abstract type: its constructors are hidden from the user. The `IO` library defines functions that construct and examine `IOError` values. The only Prelude function that creates an `IOError` value is `userError`. User error values include a string describing the error.

```
userError :: String -> IOError
```

Exceptions are raised and caught using the following functions:

```
ioError :: IOError -> IO a
catch   :: IO a    -> (IOError -> IO a) -> IO a
```

The `ioError` function raises an exception; the `catch` function establishes a handler that receives any exception raised in the action protected by `catch`. An exception is caught by the most recent handler established by `catch`. These handlers are not selective: all exceptions are caught. Exception propagation must be explicitly provided in a handler by re-raising any unwanted exceptions. For example, in

https://doi.org/10.1017/S0956796803000911 Published online by Cambridge University Press

```
f = catch g (\e -> if IO.isEOFError e
                   then return []
                   else ioError e)
```

the function `f` returns `[]` when an end-of-file exception occurs in `g`; otherwise, the exception is propagated to the next outer handler. The `isEOFError` function is part of `IO` library.

When an exception propagates outside the main program, the Haskell system prints the associated `IOError` value and exits the program.

The `fail` method of the `IO` instance of the `Monad` class (Section 6.3.6) raises a `userError`, thus:

```
instance Monad IO where
  ...bindings for return, (>>=), (>>)

  fail s = ioError (userError s)
```

The exceptions raised by the I/O functions in the Prelude are defined in Chapter 21.

https://doi.org/10.1017/S0956796803000911 Published online by Cambridge University Press