The Monad.Reader/Issue2 /Bzlib2Binding

From HaskellWiki

< The Monad.Reader | Issue2

Contents

1 A binding for Bzlib2

by Peter Eriksen for The Monad.Reader, Issue Two

The article presents a Haskell FFI binding for the C library Bzlib2. This article also exists as a PostScript file (http://www.student.dtu.dk/~s022018/Bzlib2Hs/tmrartikel.ps) .

1.1 Introduction

This text is about the Haskell Foreign Function Interface (FFI) which is part of the standard library. It enables Haskell functions to call procedures in other languages and other languages to call Haskell functions. This text is for those who like me have found it difficult to cross the gap in the FFI documentation between trivial examples and reference texts.

Having a basic understanding of the FFI is really worthwhile, because it adds a new dimension to the language toolbox and gives access to a vast amount of non-Haskell

libraries in Haskell programs. Many libraries have already had more or less complete Haskell bindings through the FFI including WxHaskell, Gtk2Hs, HOpenGL (http://www.haskell.org/HOpenGL/) , SDL and PCRE.

This text is nowhere near a complete introduction for the FFI, in fact it is rather limited, but it's intended to answering some of the basic question I myself had, when I first tried to understand the functioning of the FFI. I'll describe a simple binding I made for the Bzip2 library for data (de)compression, and I encourage the reader to download the binding and study and play around with it. Of course patches are welcome.

We will end up with an interface consisting of the two useful procedures:

```
readBzip2 :: Handle -> IO String
writeBzip2 :: Handle -> String -> IO ()
```

1.2 Prerequisites

You definitely want to know where to find documentation on the FFI, so here is a small list of useful references:

- The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report (http://www.cse.unsw.edu.au/~chak/haskell/ffi/)
- Haskell Hierarchical Libraries Foreign package (http://www.haskell.org/ghc/docs/6.4/html/libraries/base/Foreign.html)
- Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, p. 46 (http://research.microsoft.com/Users/simonpj/papers/marktoberdorf/)

To play around with the library you need to have the binding package, libbzip2 (from the Bzip2 package) version 1.0 and GHC 6.4. I'm sure other versions of GHC will work, but I Haven't tested it.

Some knowledge of C is required since we'll make a binding for a C library.

1.3 Get on with it

The binding is in a Darcs repository, which can be fetched and build like this:

```
$ darcs get "http://www.student.dtu.dk/~s022018/Bzlib2Hs"
$ cd Bzlib2Hs/
$ ghc -c Bzip2.lhs
$ ghc --make Main.hs -lbz2
$ ./a.out
```

If you want to subsequently try it out interactively in GHCi say:

```
$ ghci Bzip2.lhs -lbz2
```

1.4 The library

libbzip2 is the library written by Julian R Seward used in bzip2(1) for compressing and decompressing files in the bzip2 data format. The interface file is named bzlib.h and can be found somewhere in your system usually in /usr/include/ or /usr/local/include. On GoboLinux it's /System/Links/Headers/bzlib.h. It provides many of procedures of which we will only need use the high level ones. They a still a bit complicated, so in the following they are briefly described.

1.5 Signatures

The high level procedures is comprised of the following signatures as seen in the header file:

```
#!syntax c
typedef void BZFILE;

BZFILE *BZ2_bzReadOpen ( int *bzerror, FILE *f,
 int small, int verbosity,
 void *unused, int nUnused );

int BZ2_bzRead ( int *bzerror, BZFILE *b, void *buf, int len );

void BZ2_bzReadClose ( int *bzerror, BZFILE *b );

BZFILE *BZ2_bzWriteOpen ( int *bzerror, FILE *f,
 int blockSize100k, int verbosity,
 int workFactor );

void BZ2_bzWrite ( int *bzerror, BZFILE *b, void *buf, int len );

void BZ2_bzWriteClose ( int *bzerror, BZFILE* f,
 int abandon,
 unsigned int* nbytes_in,
 unsigned int* nbytes_out );
```

The names of the procedures are pretty descriptive. The open procedures opens a stream `BZFILE` of data from the file handle `f`, which refers to a file already opened for reading or writing respectively. The arguments `small`, `verbosity`, `unused` and `nUnused`, `blockSize100k`, `verbosity` and `workFactor` are for our simple binding not so not so important, and will get reasonable default values of 0, 0, `NULL`, 0, 9, 0 and 30 respectively. Refer to the Bzip2 documentation for explanations of the values.

The read procedure reads at most `len` uncompressed bytes into the buffer `buf` from the stream `BZFILE` of compressed bytes and returns the number of bytes read. If the logical end-of-stream marker was read, `bzerror` is set to `BZ_STREAM_END`.

The write procedure compresses and writes `len` bytes from the buffer `buf` to the stream `b`.

The read close procedure frees all memory created when opening and reading from `b` and is also used to clean up after all error situations when reading.

The write close procedure flushes all data still hanging around the stream `f`, and then

frees all memory created when opening and writing to `f`. It's used to clean up after all error situations when writing. The arguments `abandon`, `nbytes_in` and `nbytes_out` are also not relevant to us, and will just get reasonable default values of 0, `NULL` and `NULL`.

In all procedures if an error occurs `bzerror` is set accordingly.

1.6 The Foreign Function Interface

Now where we are a bit familiar with the Bzip2 library procedures, we can introduce some concepts of the FFI. Making the binding will commence in three steps:

1. Find appropriate Haskell types for the C types.
2. Write an foreign import declaration for each procedure.
3. Write wrapper procedures to get a nice abstraction and a clean interface.

These steps are explained in the following.

1.6.1 Corresponding Haskell and C types

To be able to bridge the gap between C world and Haskell world, the FFI has defined Haskell types which corresponds to the normal C data types like integers, reals, arrays and so on. The whole lot is listed in the library documentation (http://www.haskell.org/ghc/docs/6.4/html/libraries/base/Foreign.C.html) . For convenience the types needed for this binding is listed in the following table:

```
|| C type || Haskell type ||
|| FILE || CFile ||
|| int || CInt ||
|| unzigned int || CUInt ||
|| char* || CString ||
|| a* || Ptr a ||
|| void || () ||
```

On the right is the Haskell type which corresponds to the C type on the left. The integer types are straight forward. The type `CFile` is an opaque data constructor and can thus only be constructed in a C program an then passed to Haskell. The `CString` is actually just an alias for `Ptr CChar` and the void type is represented by the Haskell type `()`. As an example the C type for a pointer to a value of any type, `void*` can be represented in Haskell as `Ptr ()`.

1.6.2 The foreign import declaration

As an exercise before proceeding go back and try to write Haskell type signatures corresponding to the C procedure signatures given above. Generally the foreign import declaration looks like:

```
foreign import ccall "[header name.h] C_name" Haskell_name ::
 Haskell_names_argument_types -> IO return_type
```

It consists of the `foreign import` keywords followed by a keyword, `ccall`, telling the compiler to use the C calling convention on the procedure. Then comes a string with an optional name of a header file containing the signature of the procedure and the procedure name. Then follows the Haskell function name, we want to give the procedure and then it's Haskell type which in our case returns something wrapped in the *IO* Monad, since the procedures have side effects.

1.7 The Haskell binding

The road is now paved for understanding the binding which is situated in `Bzlib2.lhs` in the Darcs repository. Let's go over the main parts one at a time. Some new things will be introduced along the way and explained. The first thing in the file is the import declarations.

```
{-# OPTIONS -fffi #-}

module Bzip2 where
import Control.Monad
import System.IO
import System.Posix.IO
import System.Posix.Types
import Foreign.Ptr
import Foreign.Storable
import Foreign.C.Types (CFile, CInt, CUInt)
import Foreign.C.String (CString, newCString, peekCStringLen)
import Foreign.Marshal.Alloc (malloc, mallocBytes)
```

Since the library has to be passed a handle to an already open file, the problem quickly arises as who to obtain a `Ptr CFile`, when we cannot construct a `CFile` from Haskell. The solution lies in a little trick with which we get a obtain a such through these steps: `Handle` -> `Fd` -> `Ptr CFile`:

```
-- FILE *fdopen(int fildes, const char *mode);
foreign import ccall "" fdopen :: Fd -> CString -> IO (Ptr CFile)

handleToCFile :: Handle -> String -> IO (Ptr CFile)
handleToCFile h m =
 do iomode <- newCString m
 fd <- handleToFd h
 fdopen fd iomode
```

Here we wrap the standard C procedure `fdopen` in a function, `handleToCFile`, which takes a handle to an already open file (for instance obtained with `openFile` from `System.IO`) together with a string which can be ``r *for read mode or* ``*w* for write mode and is passed to `fdopen`. Since the latter takes a `CString` and not a normal Haskell string we need to convert it with the function `newCString`.

1.7.1 Wrapping the read procedures

Here are the foreign import declarations for all the C procedures introduced in the beginning.

```
bz_OK = 0 -- #define BZ_OK 0
```

```
bz_STREAM_END = 4 -- #define BZ_STREAM_END 4

type BZFile = () -- typedef void BZFILE;

foreign import ccall "bzip.h BZ2_bzReadOpen" bzReadOpen ::
 Ptr CInt -> Ptr CFile -> CInt -> CInt -> Ptr () -> CInt -> IO (Ptr BZFile)

foreign import ccall "bzip.h BZ2_bzRead" bzRead ::
 Ptr CInt -> Ptr BZFile -> Ptr () -> CInt -> IO CInt

foreign import ccall "bzip.h BZ2_bzReadClose" bzReadClose ::
 Ptr CInt -> Ptr BZFile -> IO ()
```

This should be straight forward by now. The `bzip.h` defines some constants, which we will use, so they are declared correspondingly in Haskell. Notice that the type alias for `BZFile` and the corresponding C definition in the comment.

1.7.1.1 The read wrapper procedures

Now we can wrap those procedures in a higher abstraction closer to the way a Haskell programmer would think. `readBzip2` takes a handle *h* of a file, which is already opened for reading, and returns its decompressed content as a string.

```
readBzip2 :: Handle -> IO String
readBzip2 h =
 let len = 1024
 verbosity = 0
 small = 0
 unused = nullPtr
 nUnused = 0
 in
 do bzerror <- malloc
 buf <- mallocBytes len -- How big should this be to be efficient?
 cfile <- handleToCFile h "r"
 bzfile <- bzReadOpen bzerror cfile verbosity small unused nUnused
 s <- readIter bzerror bzfile buf len
 bzReadClose bzerror bzfile
 return s

readIter :: Ptr CInt -> Ptr BZFile -> Ptr () -> Int -> IO String
readIter bzerror bzfile buf len =
 do nbytes <- bzRead bzerror bzfile buf (fromIntegral len)
 s <- peekCStringLen (castPtr buf, fromIntegral nbytes)
 e <- peek bzerror
 if e==bz_STREAM_END
 then return s
 else do s' <- readIter bzerror bzfile buf len
 return (s++s')
```

The first procedure defines the default constants, then allocates some variables and opens the handle for reading compressed content. Then it calls the second procedure which loops reading bytes into the supplied buffer and returning it in a Haskell string. In the end the handle is closed and the accumulated string is returned.

The value `nullPtr` is the name of `NULL` in C. The polymorphic function `malloc` returns a pointer to a freshly allocated value, and since `bzerror` is a `Ptr CInt`, `malloc` will allocate an integer. `mallocBytes` allocates an array of bytes and returns a pointer to the first byte. We can convert between most integral Haskell and C types using `fromIntegral` as seen in `readIter`.

Two new procedures are `peekCStringLen` and `peek`. The latter returns the value pointed to by it's pointer argument; it's dereferencing the pointer. Like `malloc` it's polymorphic a thus derives it's correct type and behavior from the context. The former is more complicated. Here's it's type:

```
peekCStringLen :: (Ptr CChar, Int) -> IO String
```

It takes a pair of a character array and a length *n* and delivers the first *n* bytes from the array as a Haskell string. Because of the type conflict between `buf`, which has type `Ptr ()` and the `Ptr CChar` in `peekCStringLen`, we need to explicitly cast the pointer type using the polymorphic `castPtr`.

I rather sure that the entire file will be read in before the string is returned, which obviously is a place for improvement. Can you see how that can be done?

1.7.1.2 Wrapping the write procedures

Now to the corresponding write procedures. The foreign import declarations should not cause surprises any more:

```
foreign import ccall "bzip.h BZ2_bzWriteOpen" bzWriteOpen ::
 Ptr CInt -> Ptr CFile -> CInt -> CInt -> CInt -> IO (Ptr BZFile)

foreign import ccall "bzip.h BZ2_bzWrite" bzWrite ::
 Ptr CInt -> Ptr BZFile -> Ptr () -> CInt -> IO ()

foreign import ccall "bzip.h BZ2_bzWriteClose" bzWriteClose ::
 Ptr CInt -> Ptr BZFile -> CInt -> Ptr CUInt -> Ptr CUInt -> IO ()
```

The write procedure takes a handle to a file already opened for writing and a string and compresses the string into the file.

```
writeBzip2 :: Handle -> String -> IO ()
writeBzip2 h s =
 let len = length s
 blockSize100k = 9
 verbosity = 0
 workFactor = 30
 abandon = 0
 nbytes_in = nullPtr
 nbytes_out = nullPtr
 in
 do bzerror <- malloc
 cs <- newCString s
 cfile <- handleToCFile h "w"
 bzfile <- bzWriteOpen bzerror cfile blockSize100k verbosity workFactor
 bzWrite bzerror bzfile (castPtr cs) (fromIntegral len)
 bzWriteClose bzerror bzfile abandon nbytes_in nbytes_out
```

Most of the things in here were explained in the read section. The procedure `newCString s` just takes a Haskell string and returns a pointer to a freshly allocated character array initialised with the string.

The reflecting reader might already have realized that the input string is allocated in a

buffer in one big chunk, which could be very big and in fact is as big as the file itself. Also neither of the procedures handles error conditions reported through the variable `bzerror`.

1.8 Conclusion

We've written a wrapper for the C library Bzlib2 with a very simple and useful interface consisting of the two procedures

```
readBzip2 :: Handle -> IO String
writeBzip2 :: Handle -> String -> IO ()
```

Although simple the binding has some severe shortcomings, namely lack of error checking and lack of laziness in the strings passed to and from the procedures. Error checking was avoided to keep the implementation uncluttered and could be implemented by after each call which can raise an error to observe the value of `bzerror` with `e <- peek bzerror` and acting appropriately. The laziness problem is harder; when should we call `bzReadClose bzerror bzfile` if we only read in half of the file and moving on? In the case of writing it's a tad easier. Write a loop which lazily consumes the input string and writes a small chunk to the files at the time.

Patches and comments are most welcome.

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue2/Bzlib2Binding"

Category: Article
