Understanding The Haskell FFI

Rebecca Skinner

rebecca@rebeccaskinner.net

September 5, 2013

What is the FFI?

The FFI allows Haskell code to interoperate with native code by allowing haskell applications to call or be called by native functions through static and shared libraries and object files.

What is Native code?

The Haskell 98 Addendum on FFI defines a mechanism for interoperating with code that uses the platforms C calling convention. The standard leaves room for implementations to support other conventions, such as C++ or Java, but these are not supported by GHC.

Using the FFI with GHC

The FFI is not part of the Haskell 98 standard, and must be included as a language extension. In GHC you can include the FFI pragma in your code:

```
{-# LANGUAGE ForeignFunctionInterface #-}
```

or pass the `-XForeignFunctionInterface` or `-fglasgow-exts` options on the command line.

A Brief Aside on Platform Dependence

Since the FFI deals with implementation defined and platform specific code, we will pick a reference platform for the examples. In this case:

- GNU + Linux
- AMD64 System V ABI
- ELF File Format
- GHC 7.4
- GCC 4.7
- libc 4.6

Background

To understand how the FFI on works on our target platform we need to understand how C applications work. Let's look at how we go from source code to a running application on our target platform.

Definition: Symbol

Symbols represent things such as data, functions, ELF sections, or debugging resources. The way that symbol names are created is language and compiler specific, and is part of the compiler ABI.

Getting Into Specifics

Let's take a look at example. We'll create a program in C that calls a function, `generate_message`, and see what happens.

Our first example - `hello.c`

hello.c

```c
/* GNU99 C Source; compile with gcc -std=gnu99 */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>

char* generate_message(const char* name)
{
    char* s = NULL;
    asprintf(&s, "Hello, %s",name);
    return s;
}

int main(int argc, char** argv)
{
    char* s = generate_message("world");
    printf("%s\n",s);
    free(s);
    return EXIT_SUCCESS;
}
```

Compiling Files

Although we can generate an executable directly from our source code, it's illustrative to first generate an object file:

```
user@host$ gcc -std=gnu99 -c hello.c -o hello.o
```

Next we can link our object file with the system libraries to generate our final executable. gcc is helping us out here by defining some default parameters, but we could also do this manually by running `ld` directly.

```
user@host$ gcc hello.o -o hello
```

The ELF Object File Format

The ELF file format consists of an ELF header containing metadata information and offets to a number of sections. The specific sections that are included in a file vary depending on the type of file. Of specific interest to us are the *Symbol Table* and the *Relocations*

We can use the `readelf` command to look at the contents of an ELF file.

ELF Object File Symbol Table

```
Symbol table '.symtab' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000    52 FUNC    GLOBAL DEFAULT    1 generate_message
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND asprintf
    11: 0000000000000034    60 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND free
```

A Side Note on C++

If we'd used a C++ compiler to compile our code, the entry with our `generate_message` symbol would have looked more like this:

Name-Mangled Symbol

```
52:  00000000004005ac 52 FUNC GLOBAL DEFAULT 13 _Z16generate_messagePKc
```

C++ uses name mangling to manage polymorphism. You can get around this by using `extern "C"`, but we are just going to avoid it for this talk.

ELF Object File .text Relocations

```
      Relocation section '.rela.text' at offset 0x678 contains 6 entries:
Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000001d  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000002a  000a00000002 R_X86_64_PC32     0000000000000000 asprintf - 4
000000000044  00050000000a R_X86_64_32       0000000000000000 .rodata + a
000000000049  000900000002 R_X86_64_PC32     0000000000000000 generate_message - 4
000000000059  000c00000002 R_X86_64_PC32     0000000000000000 puts - 4
000000000065  000d00000002 R_X86_64_PC32     0000000000000000 free - 4
```

Meaning of the ELF sections

The symbol table is a persistant hash table that is used for looking up symbols [1]. A Relocation section [2] contains offsets used at load time by the linker.

[1] The `.dynsym` section in executables servers a similar purpose

[2] There are relocation sections for several different sections in an ELF file

Moving On

So we have symbols and relocations for our function. What now?

Generating an Assembly Language File

We can look at the assembly being generated by gcc by running:

```
user@host$ gcc -S hello.c -o hello.s
```

An Excerpt from `hello.s`

```
# The start of our function
generate_message:
.LFB0:
# .LC1, which contains "World", was pushed to %edi,
# which our ABI uses as the first function parameter register
.cfi_startproc
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $32, %rsp
movq    %rdi, -24(%rbp)
movq    $0, -8(%rbp)
movq    -24(%rbp), %rdx
leaq    -8(%rbp), %rax

# Push .LC0, which contains "Hello, %s", into %esi, which is the
# second parameter register
movl    $.LC0, %esi
movq    %rax, %rdi
movl    $0, %eax
call    asprintf
movq    -8(%rbp), %rax
leave
```

The End Result

We don't need to care much about the details of that code. Here are our takeaway points:

- The compiler ABI defines how we generate symbol names
- Symbol names are the keys for entries in symbol tables
- The linker relocates code, we can find it thanks to relocations
- The calling convention defines how we call functions
- The FFI ensures that our haskell code interoperates with native code by ensuring that the ABI and calling conventions are met

FFI Supplied Types

The FFI provides for Haskell analogues to many basic C datatypes in `Foreign.C.Types`, and a set of utility functions for dealing with C Strings in `Foreign.C.String`

Common Types in `Foreign.C.Types`

C Type	Haskell Type
`int8_t, char`	`CChar`
`int, int32_t, long`	`CInt`
`unsigned int, uint32_t`	`CUInt`
`long long, int64_t`	`CLong`
`time_t`	`CTime`
`size_t`	`CSize`
`ptrdiff_t`	`CPtrdiff`

Table: A Mapping Between C and Haskell Types

C Strings

The FFI provides a number of utility functions for working with C Strings. `Data.ByteString` also provides functions for marshalling between ByteString and CString types.

Useful C String Functions

Useful CString Definitions

```
type CString    = Ptr CChar
type CStringLen = (Ptr CChar, Int)

newCString   :: String -> IO CString
peekCString  :: CString -> IO String
withCString  :: String -> (CString -> IO a) -> IO a
packCString  :: CString -> IO ByteString
useAsCString :: ByteString -> (CString -> IO a) -> IO a
```

Marshalling

Marshalling is how we make data shared between C and Haskell mutually intelligable. Native C types are mapped to Haskell types through `Foreign.C.Types`. Additional support functions for C strings are available in `Foreign.C.String`

Marshalling Gotchas

There are a few specific things that we need to be aware of before we get started:

- You may need to account for Endianness of data
- Fundamental types may have different bit widths between Haskell and C, e.g. Ints
- The width of some types may be architecture dependant
- Pointer operations are impure

Pointers

A pointer represents a raw machine address. The FFI defines three[3] types of pointers that we are interested in:

- `Ptr a` A raw machine address. In many cases, `a` is a storable.
- `FunPtr a` A pointer to a foreign function. On some architectures it is possible to cast between a `Ptr a` and a `FunPtr a`
- `StablePtr a` A pointer to a Haskell expression that will not be touched by the garbage collector. This may be necessary if you exposing a native API implemented in Haskell

[3] There are additional pointer types defined by the FFI that are analogous to C's `intptr_t` and `uintptr_t` types

Definition: Opaque Pointer

An opaque pointer does not need to be marshalled and can in most cases be treated as a pointer to an existential type. Using mutators instead of direct structure access in native APIs can simplify their use in the FFI because of this.

Creating Opaque Types

Opaque Pointer Types

```haskell
data MyType = MyType
type MyTypeHandle = Ptr MyType
newtype MyOneshotHandle = Ptr MyOneshotHandle
```

Storable Typeclass

Storable Typeclass

```
class Storable a where
sizeOf       :: Storable a => a -> Int
alignment    :: Storable a => a -> int

peek         :: Storable a => Ptr a -> IO a
peekElemOff  :: Storable a => Ptr a -> Int -> IO a
peekByteOff  :: Storable a => Ptr b -> Int -> IO a

poke         :: Storable a => Ptr a -> a -> IO ()
pokeElemOff  :: Storable a => Ptr a -> Int -> a -> IO ()
pokeByteOff  :: Storable a => Ptr b -> Int -> a -> IO ()
```

Implementing Storable

- `sizeOf`
  Return the size in bytes of the data structure
- `alignment`
  Return the byte alignment of the data structure
- One of: `peek`, `peekElemOff`, or `peekByteOff`
  Read data from the provided memory address
- One of: `poke`, `pokeElemOff`, or `pokeByteOff`
  Write data to the provided memory address

Storable Example: NetfilterQueue.hs

Storable Example

```
data NfGenMsg = NfGenMsg {packet_id :: CUInt, hw_protocol :: CUShort, hook :: CUChar}
instance Storable NfGenMsg where
    sizeOf _ =
        sizeOf (0 :: CUInt) + sizeOf (0 :: CUShort) + sizeOf (0 :: CUChar)
    alignment _ = 16 -- > 8 bytes so we should be 16-byte aligned on x86_64
    peek p =
        let ptr1 = castPtr p
            ptr2 = castPtr $ ptr1 `plusPtr` sizeOf (0 :: CUInt)
            ptr3 = castPtr $ ptr2 `plusPtr` sizeOf (0 :: CUShort)
        in do
            v1 <- peek ptr1
            v2 <- peek ptr2
            v3 <- peek ptr3
            return $ NfGenMsg (fromBigEndian v1) (fromBigEndian v2) v3
    poke ptr (NfGenMsg pkt_id hw_proto hk) =
        let ptr1 = castPtr ptr
            ptr2 = castPtr $ ptr1 `plusPtr` sizeOf (0 :: CUInt)
            ptr3 = castPtr $ ptr2 `plusPtr` sizeOf (0 :: CUShort)
        in do
            poke ptr1 $ toBigEndian pkt_id
            poke ptr2 $ toBigEndian hw_proto
            poke ptr3 hk
            return ()
```

Foreign Functions in Haskell

In order to use a foreign function in Haskell you must create a foreign declaration. The syntax defined for foreign declarations in the FFI addendum is:

Foreign Declaration Syntax in Haskell

topdecl	→	foreign *fdecl*	
fdecl	→	import *callconv* [*safety*] *impent var* :: *ftype*	(*define variable*)
	|	export *callconv expent var* :: *ftype*	(*expose variable*)
callconv	→	ccall | stdcall | cplusplus | jvm | dotnet	
	|	system-specific-calling-convention	(*calling convention*)
impent	→	[string]	(*imported external entity*)
expent	→	[string]	(*exported entity*)
safety	→	safe | unsafe	

Note that foreign declarations may reference any type of foreign data, not just functions.

Importing and Exporting

When we are importing or exporting a foreign declaration we need to define both what the name for it is in our haskell application (the *var*), and the name that appears in the ELF symbol table (the *impent* or *expent*).
The calling convention allows us to specify what standard calling convention should be used. Although there are several reserved keywords for calling conventions, only `ccall` is widely supported at this time.

Safety

`safe` and `unsafe` refers to whether the behavior of the application is well defined if a native function executes a callback into the Haskell application. Any data access, other than the formal parameters of the function or stable pointers, accessed by an `unsafe` function, results in undefined behavior.
When unspecified, `safe` is the default behavior. `unsafe` calls are generally faster.

Types of Native Declarations

Native declarations must be well typed. Just like haskell functions, native functions that have side effects should return a value in the `IO` monad. Pure native functions need not return their value inside of `IO`.

Foreign Declaration: Example

Sample Foreign Function Declarations

```
-- Create a new FLTK Window
foreign import ccall unsafe "fl_window_new" flWindowNew ::
    CInt ->         -- size X
    CInt ->         -- size Y
    CString ->      -- title
    IO FltkWindow -- newly created window

-- Create a Queue Handle from the Netfiler Handle
foreign import ccall unsafe "nfq_create_queue" nfq_create_queue ::
    NetfilterHandle ->        -- The netfiler handle to create the queue handle from
    CShort ->                 -- The queue number to bind to
    NetfilterCallback ->      -- The callback function to use when processing packets
    NetfilterUserData ->      -- User data passed into the callback
    IO NetfilterQueueHandle   -- The queue handle
```

Putting it all together

C Program Exporting A Native Library

```c
#define _GNU_SOURCE
#include <stdio.h>
char* gen_message(const char* name)
{
    char* message = NULL;
    asprintf(&message,"Hello, %s",name);
    return message;
}
```

Haskell Application Using FFI

```haskell
import Foreign.C.String
foreign import ccall safe "gen_message" genMessage :: CString -> IO CString
main = withCString "World" genMessage >>= peekCString >>= putStrLn
```

Putting it all together (cont.)

Building and Running

```
user@host$ ghc -c hello_hs.hs -o hello_hs.o
```

```
user@host$ gcc -c hello_c.c -o hello_c.o
```

```
user@host$ ghc hello_hs.o hello_c.o -o hello
```

Some More Examples

- fltk-haskell
- netfilter-haskell

Guidelines

There are no hard and fast rules on how or when to use the FFI. Here are some guidelines I've come up with based on my own experiences.

- Creating Haskell bindings to Native libraries is a bit easier than going the other way around
- You can create a wrapper around a native library in it's own language, then wrap that, to make things go more smoothly
- Use mutators to keep pointers opaque to avoid doing a bunch of marshalling
- `const`-correctness in C libraries makes managing side effects much easier
- Bang patterns can help manage complications introduced by eagerness mismatches between Haskell and native libraries
- When possible, know your target architecture(s) well. It will save you a ton of pain when dealing with marshalling

hsc2hs

So far we've talked about using the FFI manually. `hsc2hs` helps automate the process of creating haskell bindings to C libraries. It works well in the general case, but it doesn't abstract away the details of the FFI, and sometimes requires manual intervention, so it's best to understand what's going on under the hood before getting started wtih it.

Questions?

Questions?