Connecting Haskell to C/C++

Michael Tolly

Why would you need C/C++

- The free software C library ecosystem: basically packages starting with “lib”
- Audio/video stuff tends to be in C because of real-time requirements
- Lot of utilities for game development are C++, due to professional game dev

C as lingua franca

- Almost every other language has a way to connect to C code
- 2 approaches:
  - Extension API: write C code to connect a C lib to an interpreter
  - Foreign function interface: import C stuff and do the translation work in the host language
- FFI is generally superior in terms of maintainability

This talk

1) How C becomes machine code, what does calling a C function look like
2) The Haskell foreign function interface
3) The “higher FFI”: working with C headers
4) How to locate code/libraries and link a program
5) How to package a program with libraries and ship to users

Part 1/5

C to machine code (briefly)

How C is compiled/executed

- A processor runs “machine code”, the compact binary format for instructions the processor can execute
- Most instructions are simple, like “add two numbers” or “load a number from memory”, so generally one C statement can compile to several instructions
- Instructions generally run in sequence, except for a “jump”/”branch” instruction, which instead jumps to somewhere else in the code
  - Higher-level control flow constructs like `if/then/else`, `while`, `switch` are converted to these jumps

C functions

- C code is organized into “functions” (ehhh), a block of code with extra abilities
  - from elsewhere, you can “call” it by jumping to it
    - you can pass arguments to it when you do so
  - it runs its code, does whatever it does
  - it then jumps back to you (to the location right after where you did your jump)
    - afterward, it may have returned something back to you that you can retrieve
- In machine code, a function is just a block of code that does all these things according to a set of rules
  - The exact way you do it is a “calling convention”, which can differ by processor type, operating system, and compiler

C function example

```
int blackbox(int x, int y) {
  return 2 * x + 3 * y;
}

int main() {
  int result = blackbox(12, 34);
  return result;
}
```

C function example

```
int blackbox(int x, int y) {
  return 2 * x + 3 * y;
}
```

```
blackbox:
  push  rbp
  mov rbp, rsp
  mov DWORD PTR -4[rbp], edi  # x is passed in "edi" register
  mov DWORD PTR -8[rbp], esi  # y is passed in "esi" register
  mov eax, DWORD PTR -4[rbp]  # eax ← x
  lea ecx, [rax+rax]          # ecx ← eax + eax = x * 2
  mov edx, DWORD PTR -8[rbp]  # edx ← y
  mov eax, edx                # eax ← edx = y
  add eax, eax                # eax ← eax + eax = y * 2
  add eax, edx                # eax ← eax + edx = y * 3
  add eax, ecx                # eax ← eax + ecx = x * 2 + y * 3
  pop rbp
  ret                         # jump back to the caller
```

C function example

```
int main() {
  int result = blackbox(12, 34);
  return result;
}
```

```
    main:
        push  rbp
        mov rbp, rsp
        sub rsp, 16
        mov esi, 34                  # put 34 in "esi" (2nd arg)
        mov edi, 12                  # put 12 in "edi" (1st arg)
        call  blackbox@PLT           # push our return address and jump
        mov DWORD PTR -4[rbp], eax   # eax has the returned value
        mov eax, DWORD PTR -4[rbp]
        leave
        ret
```

C function example

```
$ gcc -S function.c main.c
$ gcc function.s main.s
$ ./a.out
$ echo $?
126
```

- So, calling a C function just means generating a snippet of code that follows the rules
  - If you know the function's type, you can generate this code without seeing the function's implementation
- The Foreign Function Interface gives us this ability from within Haskell

Part 2/5

The Foreign Function Interface

The Foreign Function Interface

- Originally an add-on to the Haskell 98 standard, later an official part of Haskell 2010
- Adds new top-level declarations for importing foreign functions as Haskell functions, as well as exporting Haskell functions so foreign code can call them
- Initially designed for C, but designed at a high level so it can be extended to other language environments
  - This has since been done with JavaScript (GHCJS, Asterius) and Java (Eta)

Example 1

```
// .c
int doubleMe(int x) {
  return x * 2;
}

-- .hs
foreign import ccall "doubleMe"
  doubleMe :: CInt -> IO CInt
```

Example 1 (running)

```
main :: IO ()
main = doubleMe 25 >>= print
```

```
$ ghc main.hs double.c
[1 of 1] Compiling Main  ( main.hs, main.o )
Linking main ...
$ ./main
50
```

Parts of the import declaration

```
foreign import ccall "doubleMe"
  doubleMe :: CInt -> IO CInt
```

- `foreign import`
  - New top-level declaration type added by the FFI
  - `export` also available
- `ccall`
  - Calling convention
  - Almost always `ccall`, unless 32-bit Windows
- String literal: name of function in C
  - Defaults to same as Haskell name if not given
- Haskell identifier and type signature: used to generate the calling code

Modules for C-Hs translation

- **`Foreign`, `Foreign.C`**
  - Set of C numerical types: `CInt`, `CLong`, `CFloat`, `CDouble`, etc.
    - `newtype` wrappers around standard Haskell types depending on your system
    - All have `Num`, `Real`, `Integral`/`Fractional`, etc. instances so you can use numeric literals directly
  - `Ptr`: standard pointer type
    - Carries a phantom type parameter
    - The `Storable` typeclass lets you use this parameter in the usual C way to access values in memory
    - Or, you can fill it with an abstract type: `data Foo; getFoo :: IO (Ptr Foo)`
  - `FunPtr`: function pointer type, can be translated to/from a Haskell function
  - `ForeignPtr`: pointer that hooks into the Haskell garbage collector, so a destructor/free function can be called when the Haskell value goes out of scope

Example 2: pointers

```
// .c
void doubleMe(int *input, int *output) {
  *output = *input * 2;
}

-- .hs
foreign import ccall "doubleMe"
  c_doubleMe :: Ptr CInt -> Ptr CInt -> IO ()
```

Example 2: pointers

```
foreign import ccall "doubleMe"
  c_doubleMe :: Ptr CInt -> Ptr CInt -> IO ()
```

```
doubleMeSimple
  :: CInt -> IO CInt
doubleMeSimple n = do
  pin <- malloc
  poke pin n
  pout <- malloc
  c_doubleMe pin pout
  out <- peek pout
  free pin
  free pout
  return out
```

```
doubleMeBetter
  :: CInt -> IO CInt
doubleMeBetter n =
  with n $ \pin -> do
    alloca $ \pout -> do
      c_doubleMe pin pout
      peek pout
```

Modules for C-Hs translation

- **Arrays**
  - C arrays are just a sequence of elements packed sequentially in memory, plus a length (passed separately)
  - `Storable` includes functions to read/write Haskell lists as packed arrays
  - `vector` package provides `Data.Vector.Storable`, which is a nice Haskell data structure but stored in memory as a C array
    - Can construct a storable vector directly from a `Ptr`/`ForeignPtr`, optionally with no copying

Example 3: arrays

```c
// .c
void doubleEach
(int *inputs, int length)
{
  for (int i = 0; i < length; i++) {
    inputs[i] *= 2;
  }
}
```

```haskell
-- .hs
foreign import ccall
"doubleEach"
  c_doubleEach
    :: Ptr CInt
    -> CInt
    -> IO ()
```

```haskell
doubleEach :: [CInt] -> IO [CInt]
doubleEach nums = withArrayLen nums $ \len p -> do
  c_doubleEach p (fromIntegral len)
  peekArray len p
```

Modules for C-Hs translation

- **Strings**
  - `char*` can correspond to `ByteString` or `Text` depending on context
  - Need to know character encoding to read as `String`/`Text`
    - UTF-8, common on Mac/Linux as well as some cross-platform APIs
      - `Data.ByteString.packCString`, then `Data.Text.Encoding.decodeUtf8`
    - UTF-16, common on Windows
      - Stored in a `wchar_t*` (`Ptr CWchar`)
      - `Foreign.C.String.peekCWString`, then `Data.Text.pack`
    - System locale encoding, used by readFile/writeFile/etc.
      - `Foreign.C.String.peekCString`, then `Data.Text.pack`

Example 4: C++

```
class Box
{
public:
  Box() {
  }
  ~Box() {
  }
  void setVal(int x) {
    val = x;
  }
  int getVal() {
    return val;
  }
private:
  int val;
};
```

```
extern "C" {

typedef void * CBOX;

CBOX makeBox() {
  return (CBOX)(new Box());
}
void destroyBox(CBOX b) {
  delete (Box *) b;
}
void setVal(CBOX b, int x) {
  ((Box *) b)->setVal(x);
}
int getVal(CBOX b) {
  return ((Box *) b)->getVal();
}

}
```

Example 4: C++

```
extern "C" {

typedef void * CBOX;

CBOX makeBox() {
  return (void *)(new Box());
}
void destroyBox(CBOX b) {
  delete (Box *) b;
}
void setVal(CBOX b, int x) {
  ((Box *) b)->setVal(x);
}
int getVal(CBOX b) {
  return ((Box *) b)->getVal();
}

}
```

```
newtype Box = Box (Ptr Box)

foreign import ccall "makeBox"
  makeBox :: IO Box

foreign import ccall "destroyBox"
  destroyBox :: Box -> IO ()

foreign import ccall "getVal"
  getVal :: Box -> IO CInt

foreign import ccall "setVal"
  setVal :: Box -> CInt -> IO ()
```

Example 4: C++ (running)

```
main :: IO ()
main = do
  b <- makeBox
  setVal b 123
  getVal b >>= print
  setVal b 456
  getVal b >>= print
  destroyBox b
```

```
$ ghc main.hs stuff.cpp
    -lstdc++
$ ./main
123
456
```

Part 3/5

Working with C headers

Next steps

- We can now bind to simple functions and deal with memory, but we're missing some things:
  - Our bindings aren't verified in any way – if we make a mistake, we'll get garbage data or a crash
  - No facilities to read/write the fields of structs
  - Can't deal with enumeration values or flags, which can come from an `enum` or a C preprocessor `#define`
  - Doing the wrapping/unwrapping to translate a function to a more Haskell-friendly form is a lot of boilerplate, that gets cumbersome when you have more complex functions
- Tools that fill these gaps: `c2hs`, `hsc2hs`
  - Both supported as a preprocessor by Cabal packages
  - I'm more familiar with `c2hs` so that's what I'll show

c2hs example 1

```c
// stuff.h
int doubleMe(int x);

// stuff.c
int doubleMe(int x) {
  return x * 2;
}
```

```haskell
-- Main.chs
module Main where

import Foreign.C

#include "stuff.h"

{#fun doubleMe
  { `CInt'
  } -> `CInt'
#}

main :: IO ()
main = doubleMe 25 >>= print
```

c2hs example 1 (translation)

```
-- Main.chs
{#fun doubleMe
  { `CInt'
  } -> `CInt'
#}
```

```
-- Main.hs (generated)
doubleMe :: (CInt) -> IO ((CInt))
doubleMe a1 =
  let {a1' = fromIntegral a1} in
  doubleMe'_ a1' >>= \res ->
  let {res' = fromIntegral res} in
  return (res')

foreign import ccall safe "Main.chs.h doubleMe"
  doubleMe'_
    :: (C2HSImp.CInt -> (IO C2HSImp.CInt))
```

c2hs example 2: in/out conversions

```c
// stuff.h
int doubleMe(int x);

// stuff.c
int doubleMe(int x) {
  return x * 2;
}
```

```haskell
-- Main.chs
#include "stuff.h"

{#fun doubleMe as doubleFloat
  { round `Float'
  } -> `Float' fromIntegral
#}

main :: IO ()
main = doubleFloat 12.4 >>= print

-- prints:
-- 24.0
```

c2hs example 3: #define

```
#define RED    0
#define YELLOW 1
#define GREEN  2

const char *redFruit
  = "apple";
const char *yellowFruit
  = "banana";
const char *greenFruit
  = "lime";
```

```
const char *getFruit(int color) {
  switch (color) {
    case RED:
      return redFruit;
    case YELLOW:
      return yellowFruit;
    case GREEN:
      return greenFruit;
    default:
      return NULL;
  }
}
```

c2hs example 3: #define (version 1)

```c
#define RED    0
#define YELLOW 1
#define GREEN  2

const char *getFruit
  (int color);
```

```haskell
newtype Color = Color
  { unColor :: CInt }

red, yellow, green :: Color
red    = Color {#const RED    #}
yellow = Color {#const YELLOW #}
green  = Color {#const GREEN  #}

{#fun getFruit
  { unColor `Color'
  } -> `CString'
#}

-- $ getFruit yellow >>= peekCString
-- “banana”
```

c2hs example 3: #define (version 2)

```
#define RED    0
#define YELLOW 1
#define GREEN  2

const char *getFruit
  (int color);
```

```
{#enum define ColorEnum
  { RED    as RED
  , YELLOW as YELLOW
  , GREEN  as GREEN
  } deriving (Eq, Ord)
#}
-- produces:
-- data ColorEnum = RED | YELLOW | GREEN

{#fun getFruit as getFruitEnum
  { `ColorEnum'
  } -> `CString'
#}

-- $ getFruit GREEN >>= peekCString
-- “lime”
```

Part 4/5

How to install libraries and link to them

Simple methods

- Earlier we used GHC directly: `ghc file.hs file.c`
  - This also works for `.cpp`, `.m`, `.s`, `.o`: anything `gcc` can handle
- Putting C files in Cabal packages
  - `c-sources: path/to/file.c`
    - Goes in library or executable section
    - Path can be relative to .cabal, or absolute
  - Most package authors make a folder called `cbits`
  - `cc-options` lets you pass options to `gcc` like defines (`-DENABLE_FOO`)
  - `cxx-sources` and `cxx-options` let you pass separate C++ options
  - Works for a lot of use cases, but C code often needs a complicated `configure` script to select the right build options

Linking to a library

- Usual way to install libraries is from a package manager
  - Mac (Homebrew): `brew install libsndfile`
  - Arch: `pacman -S libsndfile`
  - Ubuntu/Debian: `apt-get install libsndfile-dev`
  - Windows (MSYS2, comes with stack): `pacman -S mingw-w64-x86_64-libsndfile`
- Compiled libraries usually come with static and dynamic files
  - In theory either can work with GHC or GHCi but I've had better luck with dynamic linking
- If a library is at /foo/bar/libbaz.{dll,dylib,so}:
  - `extra-libraries: baz`
  - `extra-lib-dirs: /foo/bar`
  - Also need `include-dirs` for c2hs to see the headers

Linking to a library

- Better solution is `pkg-config`
  - Standard database for installed libraries on Unix-likes
  - Command-line tool used like so:
    - `pkg-config --libs foo` (prints linker flags like `-lfoo`)
    - `pkg-config --cflags foo` (prints C compiler flags like `-I/usr/include/foo`)
  - `.pc` files installed automatically by `apt-get`, `pacman`, `brew`, etc.
  - Direct support in Cabal package
    - `pkgconfig-depends: foo`
- macOS also has Frameworks, bundles of libs + headers
  - `frameworks: foo`

Part 5/5

How to ship your program

Distributing program + libraries

- When you install libraries via package manager, it will likely put them in a standard location that your OS searches for dynamic libraries in (like `/usr/lib` or `/usr/local/lib`)
- Running your program at the command line will search this same folder, so everything works
- If you want to send the program to users, you need to:
  - Include any libraries that aren't already on their system
  - Set up the executable so it knows how to find them

Windows

- On Windows, an .exe will automatically search the folder it's in for any .dlls it needs
- Dependency Manager (depends.exe) can tell you what .dlls a program looks for
- After installing in stack's MSYS2, .dlls will be at someplace like `C:\Users\<You>\AppData\Local\Programs\stack\x86_64-windows\msys2-20180531\mingw64\bin`
- Option 1
  - Make a folder with the .exe and needed .dlls, zip it, send to users
- Option 2
  - Take that same folder, make an installer which extracts those contents to a Program Files directory
  - `nsis` package lets you write an installer generator script in Haskell
    - https://github.com/mtolly/onyxite-customs/blob/master/haskell/Installer.hs

Mac

- `otool -L exepath` prints out the libraries required by an executable
- `https://github.com/auriamg/macdylibbundler` can rewrite a program to point to a relative path for its libraries
  - Can also find all needed libraries recursively, and copy them over
  - Knows which libraries are system provided
- For GUI apps, you should make an .app bundle
  - Simple directory structure, containing Info.plist (XML file)
  - Copy executable and libraries into subfolder via dylibbundler
  - https://github.com/mtolly/onyxite-customs/blob/master/haskell/Makefile

Linux

- I don’t have experience here, haven’t distributed Linux binaries
- `ldd exepath` prints out the library information
- The equivalent setting to find libraries locally is “rpath” or “runpath”
  - https://linux.die.net/man/1/chrpath
  - https://enchildfone.wordpress.com/2010/03/23/a-description-of-rpath-origin-ld_library_path-and-portable-linux-binaries/
- Probably also need a “bundle” solution like Flatpak, Snappy, AppImage to work across Linux distributions

Bindings I've written

- https://github.com/mtolly/rubberband
  - librubberband audio stretch library
  - Demonstrates ForeignPtr usage, and turning a set of flags into a Haskell record
- https://github.com/mtolly/JuicyPixels-stbir
  - STB image resize library, connected to JuicyPixels Haskell image lib
  - Complex API condensed to a single function + record type, also typeclass usage
- https://github.com/mtolly/tinyfiledialogs
- https://github.com/mtolly/conduit-audio/tree/master/conduit-audio-lame
- https://github.com/mtolly/conduit-audio/tree/master/conduit-audio-samplerate
- https://github.com/mtolly/onyxite-customs/tree/master/haskell/ArkTool
  - Demonstrates wrapping a C++ API into C functions
- https://github.com/mtolly/onyxite-customs/tree/master/haskell/kakasi
  - Example of a library converted entirely to a Cabal package