Using Haskell in an Xcode Cocoa project

Jump to: navigation, search

In the interest of full disclosure—I'm not an old time Mac person. I've had my Mac for two years. I only decided to learn Objective-C and Cocoa after I explored doing an application using Python and Qt, and someone in the Python community said "Objective-C is easy. Why don't you write a real Mac application". Neither am I a Haskell expert.

Contents hide:

1 What not, what, and why?

This is not the missing tutorial. *Haskell for MacOS fans*. It is a description of how to add a Haskell module (callable from C) to an Xcode/Cocoa/Interface builder project on your Mac.

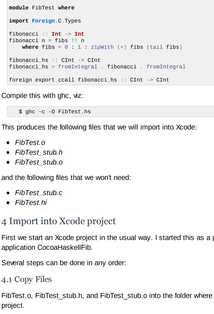
Why? Well, to paraphrase Haskell for MacOS fans, this provides a way to use Haskell to create the *Models* in the Apple Model–View–Controller design pattern while using Xcode/Interface Builder/Cocoa to build "insanely great Mac applications". Modeling that immediately comes to mind includes Parsing, and information visualization (see The Monad Reader, 14).

2 Overview of the process

The "how to" consists of the following steps.

1. Develop and test Haskell module, callable from C.
2. Build an Xcode project with a dummy *c* file for the Haskell file.
3. Add Haskell *.h* and *.o* files to your Xcode project.
4. Add dependency required Haskell libraries to your Xcode project.
5. Resolve naming conflicts between Haskell libraries and other libraries.

Steps 1 and 2 can be done in either order. Step 4 should be easy, using facilities of GHC and/or Cabal, but I haven't been able to make these work. I show how to do this iteratively. This is a bit tedious, but not bad if you don't plan on changing your Haskell module frequently.



Here is a screen shot of the upper left corner of the Mac screen, including the app menu (generated by Interface builder) and the application window itself. When the user types a number in the entry box, the line below changes to give the corresponding Fibonacci Number.

3 The Haskell module

For this test I used the same code as used in Calling Haskell from C, with some slight modifications. Here is the Haskell code, in a file called *FibTest.hs*.

```
{-# LANGUAGE ForeignFunctionInterface #-}

module FibTest where

import Foreign.C.Types

fibonacci :: Int -> Int
fibonacci n = fibs !! n
    where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibonacci_hs :: CInt -> CInt
fibonacci_hs = fromIntegral . fibonacci . fromIntegral

foreign export ccall fibonacci_hs :: CInt -> CInt
```

Compile this with ghc, *viz*:

```
$ ghc -c -O FibTest.hs
```

This produces the following files that we will import into Xcode:

- *FibTest.o*
- *FibTest_stub.h*
- *FibTest_stub.o*

and the following files that we won't need:

- *FibTest_stub.c*
- *FibTest.hi*

4 Import into Xcode project

First we start an Xcode project in the usual way. I started this as a plain Cocoa application. I called my application CocoaHaskellFib.

Several steps can be done in any order:

4.1 Copy Files

FibTest.o, FibTest_stub.h, and FibTest_stub.o into the folder where you have saved your CocoaHaskellFib project.

Then add them to the project by Project Add To Project .

4.2 Create application class

Next, create a Cocoa Objective-C class using the Xcode menu, File → New. I named mine CocoaFib. Xcode will create (bare bones)both an interface file (.h) and an implementation file (.m) for you to add code to.

Here is my interface file (.h file), after adding outlets and a method declaration:

```
#import <Cocoa/Cocoa.h>

@interface CocoaFib : NSObject {
    IBOutlet NSTextField *integerInput;
    IBOutlet NSTextField *fibOutput;
    IBOutlet NSTextField *formIn;
}

-(IBAction)generate:(id)sender;

@end
```

The three outlets are for the user input, fibonacci number output, and the number following "n =" in the label.

It's worth noting that the three outlets are connected to their respective parts of the user interface by dragging in Interface Builder.

Here is my implementation code (.m file), after adding some includes, the *generate* method, and a call to hs_*init* in the *dealloc* method:

```
#import "CocoaFib.h"
#include "HsFFI.h"
#include "FibTest_stub.h"

@implementation CocoaFib

-(IBAction)generate:(id)sender{
    int i = [integerInput intValue];
    int j = fibonacci_hs(i);
    [formIn setIntValue:i];
    [fibOutput setIntValue:j];
}

-(void)dealloc{
    hs_exit();
    [super dealloc];
}

@end
```

4.3 Modify main.m

When creating a Cocoa project, Xcode automatically generates a file *main.m*. Normally, one never touches this file. However, when using a Haskell module, one needs to initialize the Haskell run-time by calling *hs_init*. Normally, I would do this in an *init* call of *CocoaFib.m*, rather than in *main.m*, but since *hs_init* needs an *argc* and *argv*, and since *main.m already* has them hanging around, I took the easy way out.

Here is my *main.m*

```
#import <Cocoa/Cocoa.h>
#include "FibTest_stub.h"

int main(int argc, char *argv[])
{
    hs_init(&argc, &argv);
    return NSApplicationMain(argc,  (const char **) argv);
}
```

The only changes to the main.m provided by Xcode are the addition of *include "FibTest_stub.h"* and the call to *hs_init*.

5 Add Haskell libraries, compile and run

This is the easy part. Just kidding. It should be easy, if it were easy to make a library or executable containing *FibTest.o, FibTest_stub.o* and all their dependencies. Unfortunately I've been unable to do that, in spite of spending most of my spare time for a week trying, and posting questions on *Haskell-cafe*.

Anyway, here is how I did it. If you have a better way, here is a good place to edit this tutorial!

5.1 Build and go, the first time

First, I added *libffi.a* to my project for good measure.

Then select *Build and go* from the menu or the toolbar. This will result something like 26 failures, all due to undefined symbols.

What to do?

What I did was go to the *.../usr/lib/ghc-6.10.4* directory of my installation, and run:

```
find . -name "*lib*.a" | xargs nm > ~/develop/haskell/libInfo/libInfo
```

This results in a file with a list of all the available symbols from all the libraries in the Haskell installation.

Entries look something like this:

```
./array-0.2.0.0/libHSarray-0.2.0.0.a(Base__1.o):
                 U _arrayzm0z12zi0zi0_DataziArrayziBase_STUArray_con_info
00000000 D _arrayzm0z12zi0zi0_DataziArrayziBase_zdWSTUArray_closure
00000000 T _arrayzm0z12zi0zi0_DataziArrayziBase_zdWSTUArray_info
00000000 t _arrayzm0z12zi0zi0_DataziArrayziBase_zdWSTUArray_info_dsp
00000078 t _s7RK_info
00000078 t _s7RK_info_dsp
...
```

Entries with a flag T (for text) are code. U of course indicates undefined, and is no help to us. D indicates data, and is used (if I understood correctly) for defined global constants.

OK. So what do you do with this?

Your Xcode failure output will say something like:

```
"_newCAF", referenced from:
      _FibTest_a3_info in FibTest.o
"_base_GHCziBase_plusInt_closure", referenced from:
      _FibTest_a3_info in FibTest.o
...
```

Now go look in your *libInfo* file (or whatever you called it) and search for *T _newCAF*. It is in *libHSrts.a*.

Now go back to Xcode. In a Finder window, locate the file *libHSrts.a*. Drag *libHSrts.a* into the *Groups and Files* pane of the Xcode window.

In Xcode, this doesn't move the file, it provides a reference to it for the linker. You will get a dialog asking if you want to add the file to the project, and you'll be given an opportunity to copy the file into the project. Add the file to the project, but it is not necessary to copy it.

Now when you redo *Build and go*, you will no doubt get even more failures due to undefined symbols.

But don't despair, iterate.

5.2 Iterate

After a few interations of adding a library containing missing symbols followed by *Build and go* you'll get a successful build.

Well, except I had one further problem.

5.3 Resolve name conflicts of Haskell libraries

I have a bunch of libgmp.a, libgmp.so, and libgmp.dylib on my machine, in addition to the one in *.../usr/lib/ghc-6.10.4*. I don't know where they all came from. Xcode linking tries to use the ones early in its search path, and they don't work with Haskell.

To get around this, I made a symbolic link in the *usr/lib/ghc-6.10.4* directory as follows:

```
ln -s libgmp.a lib-h-gmp.a
```

and added lib-h-gmp.a to my project.

6 Creating the interface

Double-click the file called MainMenu.xib, and add the necessery GUI components. From XCode, drag the CocoaFib.h onto Interface Builder. Then drag an NSObject (which is a plain blue box) from the library into the outline view. Select the newly created NSObject, and change its class to CocoaFib (in the inspector). Then right-click on the object in the outline view, this will give a HUD window with the outlets and actions. Just drag the connections to the corresponding GUI elements (a line for each outlet and a line from generate: to the button).

7 That's all there is to it

Well, its a bit tedious, but consider the following:

Once you import this into your Xcode project, with all the attendant adding of .a files, you shouldn't have to change any of the .a file additions again. Now you can concentrate on the View and Control part of your project.

8 To do

The biggest *to do* is to generate a single library or .o file that includes all the dependencies. There should be a way to do this, but I haven't been able to get it to work. I'll continue to try. Any help will be appreciated.

One might want to write an Objective-C wrapper for your haskell module. This would encapsulate the module as an Objective-C class, and each function call would be a method. I'm not sure why you would want to do this generally. If your Haskell module was managing some persistent data store it might make sense.

Category: Tutorials