Cactus

Developing iPhone applications in Haskell — a tutorial

13 February 2011 (programming haskell iphone) (6 comments)

I couldn't find a step-by-step tutorial explaining the process of developing iOS applications in Haskell, so after finally getting "Hello World" to run on an iPhone, I decided to write this tutorial. I should also credit Lóry, who did the iOS side of the work.

The basic overview of what we're going to do in this tutorial is the following:

- Write the backend of our application in Haskell
- Create FFI bindings for our backend
- Use Jhc to compile the Haskell code into vanilla C
- Write the front-end using the official iOS SDK, using Objective-C
- Create wrappers on the Objective-C side to make resource managment easier
- Compile and link it all using Objective-C and C compilers targeting the iOS devices

Writing the backend in Haskell

For this tutorial, we will simulate an intricate Haskell backend with the a simple function. For your real application, this is where you go all out with your Haskell-fu.

```
module Engine where

import Data.Char (ord)

engine :: String -> Either String [Int]
engine s | length s < 10 = Left "String not long enough
         | otherwise = Right $ map ord s
```

FFI bindings

To interface our backend with the frontend developed in Objective-C (or C or C++ or...), we need to represent the input and output of our Haskell function in terms of simple C types. For the function `engine`, a straightforward API would be, in pseudo-C:

```
bool engine (in string s, out string error, out int[] r
```

Of course, we have to use `char *`s for strings, and pointers for out parameters and arrays, so our real API will be:

```
int engine (char* s, char* *error, int* *result, int *r
```

with `engine` returning 0 on success (`Right`) and non-zero on failure (`Left`).

The Haskell FFI representation of this signature is:

```
foreign export ccall "engine"
  engineC :: CString -> Ptr CString -> Ptr (Ptr CInt) -                    nt
```

The next step requires us to actually define `engineC` that does all the necessary marshalling. We simply evaluate `engine` and then set the appropriate out-parameters.

```
module Engine.FFI (engineC) where

import Foreign
import Foreign.C
import Control.Monad (zipWithM_)

foreign export ccall "engine"
  engineC :: CString -> Ptr CString -> Ptr (Ptr CInt) -> Ptr CInt -> IO CInt
engineC s ptrErr ptrptrResult ptrLen =
  do
    s' <- peekCString s
    case engine s' of
      Left err -> do
        cErr <- newCString err
        poke ptrErr cErr
        return 1
      Right result -> do
        pokeList ptrptrResult ptrLen $ map fromIntegral result
        return 0

pokeList :: Storable a => Ptr (Ptr a) -> Ptr CInt -> [a] -> IO (Ptr a)
pokeList ptrptrList ptrLen xs =
  do
    let len = length xs
    ptr <- mallocBytes $ len * elemSize
    let ptrs = iterate (`plusPtr` elemSize) ptr
    zipWithM_ poke ptrs xs
    poke ptrptrList ptr
    poke ptrLen $ fromIntegral len
    return ptr
  where
    elemSize = sizeOf $ head xs
```

Compiling Haskell to C

The next step is compiling our Haskell project into C, so that we can use Apple's SDK to compile that for the iPhone, and also call `engine` from other code, like the Objective-C parts that make up the frontend.

Unlike GHC, Jhc doesn't compile individual modules. Instead, it compiles every used definition (but only those) and the runtime into a single C source file. Although we are not going to run our Haskell program directly, and instead call to it from the frontend, Jhc still needs a `main` function in the source code. So let's create a Main module which we will compile with Jhc:

```
module Main where

import Engine.FFI

main :: IO ()
main = return ()
```

We can compile this module into a C file containing the code for `engineC` and everything else it uses (including imported packages):

```
jhc -fffi -fjgc --cross -mle32 -C EngineMain.hs -o
EngineMain.jhc.c
```

The `-fffi` flag turns on FFI support and makes Jhc generate the `engine` function from the `foreign export` declaration and the definition of `engineC`. Note that there is no name clash between `engine` the C function (defined as `engineC` in Haskell-land) and `engine` the Haskell definition. I think in this particular example it is cleaner to use the same name for both.

The `-fjgc` flag generates GC code. Note that we will also need to enable the GC code in the next step, when compiling the C sources.

The `--cross -mle32` flags are important because they instruct Jhc to target little-endian, 32-bit CPUs which is what the ARM is.

Compiling the generated C source code

Everything up to this point can be done without Apple's SDK, and in fact you can run Jhc on any platform you wish. From here on, however, we will use the iOS SDK to compile to ARM.

To compile `EngineMain.jhc.c`, we first need to set some preprocessor macros:

- `_GNU_SOURCE`
- `NDEBUG`
- `_JHC_GC=_JHC_GC_JGC` (turns on the generated GC code)
- `_JHC_STANDALONE=0` (this disables the `main` function defined in our module)

You also need some important C compiler flags (you can ignore the warning settings if you'd like):

```
-std=gnu99 -falign-functions=4 -ffast-math -fno-
strict-aliasing -marm -Wextra -Wall -Wno-unused-
parameter
```

`-marm` is very important because otherwise, GCC (or Clang) and Jhc step on each other's toes, leading to strange crashes seemingly out of nowhere.

Creating the frontend and accessing the backend

You can use the standard SDK to create the frontend; I will not cover that here in detail. You also need to create a header file containing the signature of our exported function. The code generated by Jhc also contains initialization and finalization routines that need to be called before and after calling any functions defined in Haskell:

```
extern void hs_init (int *argc, char **argv[]);
extern void hs_exit (void);

extern int engine (char* s, char* *msgError, int* *result, int *len);
```

You also need to manage the memory returned by the backend. The call to `mallocBytes` in the marshalling code is compiled into vanilla `malloc`, so you can simply call `free` after you're done with it.

To make initialization and memory management easier, Lóry has created a sample XCode project that wraps the C API to use Objective-C types. You can find the tarball here.

« Gettó-szakállvágó All entries Pom-pom poporopo-pom »

Stephen Blackheath 2011-02-13 23:46:17

We've ported GHC to the iPhone (as a cross-compiler running on the Mac). Unfortunately, our web page has disappeared for the moment, but everything is available from this temporary location:

http://hip-to-be-square.com/~blackh/ghc-iphone/

This is the usual URL (currently not working):

http://projects.haskell.org/ghc-iphone/

Gekkor McFadden 2011-02-14 16:09:29

I have to wonder why? COBOL on the iOS would make as much sense as this.

Lóry (http://blog.lorentey.hu) 2011-02-15 00:27:07

@Gekkor: If you have to ask why this is useful, you'll never know. :-)

Richard Zetterberg (http://paralyzed.se) 2011-02-16 16:05:45

I been wanting to learn Haskell for ages and haven't found a good initial project, but this seems like something that would really motivate me to start learning.

Thanks for this Gergő and Lóry. Keep up the good work, I will look forward for some related posts! :)

cactus 2011-02-21 21:01:04

Hi Stephen,

Thanks for the link to ghc-iphone. I saw the empty directory on haskell.org and thought it was an abandoned project.

We actually ended up using ghc-iphone instead of jhc, because jhc kept failing with mysterious internal errors when trying to use Parsec.

ceti331 2011-03-26 00:54:30

roll on dual core, quad core iOS & android devices.

great that this is possible.

New comment

Name:

E-mail address:

Home page:

Comment:

Submit

Blog tags

advogato
agda
correctness
electronics
ELTE
movies
gadget
haskell
iphone
ISC
games
food
exhibition
books
lisp
math
meta
miata
language
programming
SCB
Singapore
personal
sziget
theater
Tilos
titanic
history
unix
VPG
windows
XML
zene