

Если вы видите что-то необычное, просто сообщите мне.

Contributing to GHC 2: Basic Hacking and Organization

In part 1 of this series, we took our first step into the world of GHC, the Glasgow Haskell Compiler. We summarized the packages and tools we needed to install to get it building. We did this even in the rather hostile environment of Windows. But, at the end of the day, we can now build the project with make and create our local version of GHC.

In this part, we'll establish our development cycle by looking at a very simple change we can make to the compiler. We'll also discuss the architecture of the repository so we'll can make some cooler changes, which you can read about in part 3.

GHC is truly a testament to some of the awesome benefits of open source software. Haskell would not be the same language without it. But to understand GHC, you first have to have a decent grasp of Haskell itself! If you've never written a line of Haskell before, take a look at our Liftoff Series for some tips on how to get going. You can also download our Beginners Checklist.

You may have also heard that while Haskell is a neat language, it's useless from an industry perspective. But if you take a look at our Production Checklist, you'll find tons of tools to write more interesting Haskell programs!

GETTING STARTED

Let's start off by writing a very simple program in Main.hs.

```
module Main where

main :: IO ()
main = do
```

```
putStrLn "Using GHC!"
```

We can compile this program into an executable using the `ghc` command. We start by running:

```
ghc -o prog Main.hs
```

This creates our executable `prog.exe` (or just `prog` if you're not using Windows). Then we can run it like we can run any kind of program:

```
./prog.exe  
Using GHC!
```

However, this is using the system level GHC we had to install while building it locally!

```
which ghc  
/mingw/bin/ghc
```

When we build GHC, it creates executables for each stage of the compilation process. It produces these in a directory called `ghc/inplace/bin`. So we can create an alias that will simplify things for us. We'll write `lghc` to be a "local GHC" command:

```
alias lghc="~/ghc/inplace/bin/ghc-stage2.exe -o prog"
```

This will enable us to compile our single module program with `lghc Main.hs`.

HACKING LEVEL 0

Ultimately, we want to be able to verify our changes. So we should be able to modify the compiler, build it again, use it on our program, and then see our changes reflected in the code. One simple way to test the compiler's behavior is to change the error messages. For example, we could try to import a module that doesn't exist:

```
module Main where  
  
import OtherModule (otherModuleString)  
  
main :: IO ()
```

```
main = do
    putStrLn otherModuleString
```

Of course, we'll get an error message:

```
[1 of 1] Compiling Main (Main.hs, Main.o)

Main.hs:3:1: error:
    Could not find module 'OtherModule'
    Use -v to see a list of the files search for.
 |
3 | import OtherModule (otherModuleString)
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Let's try now changing the text of this error message. We can do a quick search for this message in the compiler section of the codebase and find where it's defined:

```
cd ~/ghc/compiler
grep -r "Could not find module" .
./main/Finder.hs:cannotFindModule = cantFindErr (sLit "Could not find module")
```

Let's go ahead and update that string to something a little different:

```
cannotFindModule :: DynFlags -> ModuleName -> FindResult -> SDoc
cannotFindModule = cantFindErr
    (sLit "We were unable to locate the module")
    (sLit "Ambiguous module name")
```

Now let's go ahead and rebuild, except let's use some of the techniques from last week to make the process go a bit faster. First, we'll copy `mk/build.mk.sample` to `mk/build.mk`. We'll uncomment the following line, as per the recommendation from the setup guide:

```
BuildFlavour=devel2
```

We'll also uncomment the line that says `stage=2`. This will restrict the compiler to only building the final stage of the compiler. It will skip past stage 0 and stage 1, which we've already build.

We'll also build from the compiler directory instead of the root ghc directory. Note though that since we've changed our build file, we'll have to boot and configure once again. But after we've re-

compiled, we'll now find that we have our new error message!

```
[1 of 1] Compiling Main (Main.hs, Main.o)
```

```
Main.hs:3:1: error:
```

```
  We were unable to locate the module 'OtherModule'
```

```
  Use -v to see a list of the files search for.
```

```
|
```

```
3 |import OtherModule (otherModuleString)
```

```
|^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

GENERAL ARCHITECTURE

Next week, we'll look into making a more sophisticated change to the compiler. But at least now we've validated that we can develop properly. We can make a change, compile in a short amount of time, and then determine that the change has made a difference. But now let's consider the organization of the GHC repository. This will help us think some more about the types of changes we'll make. I'll be drawing on this description written by Simon Peyton Jones and Simon Marlow.

There are three main parts to the GHC codebase. The first of these is the compiler itself. The job of the compiler is to take our Haskell source code and convert it into machine executable code. Here is a very non-exhaustive list of some of the compiler's tasks

1. Determining the location of referenced modules
2. Reading a single source file
3. Breaking that source into its simplest syntactic representation Then there is the boot section. This section deals with the libraries that the compiler itself depends on. They include things such as low level types like `Int` or else `Data.Map`. This section is somewhat more stable, so we won't look at it too much.

The last major section is the Runtime System (RTS). This takes the code generated by the compiler above and determines how to run it. A lot of magic happens in this part that makes Haskell particularly strong at tasks like concurrency and parallelism. It's also where we handle mechanics like garbage collection.

We'll try to spend most of our time in the compiler section. The compilation pipeline has many stages, like type checking and de-sugaring. This will let us zero in on a particular stage and make a small change. Also the Runtime System is mostly C code, while much of the compiler is in Haskell itself!

CONCLUSION

That concludes part 2 of our series on GHC. In part 3, we'll take a look at a couple more ways to modify the compiler. After that, we'll start looking at taking real issues from GHC and see what we can do to try and fix them!

If you want to start out your Haskell journey, you should read our Liftoff Series! It will help you learn the basics of this awesome language. For more updates, you can also subscribe to our monthly newsletter!

If you've done some Haskell and are more interested in building apps than working on GHC right away, that's great too! Take a look at our Haskell Web Series for more details.

Revision #1

Created 11 March 2022 16:33:38 by gasick

Updated 11 March 2022 17:11:16 by gasick