GHCi in a new dress

a project by Abi Raja and Grant Mathews

What we made

GHCi in a new dress is a browser-based REPL (read-eval-print loop) that wraps GHCi and adds a number of features such as live type annotations for intermediate expressions, variable inspection, documentation lookup and many others. The code for the project can be found at http://github.com/johnfn/ghci-in-a-new-dress

Challenges that we faced

GHCi integration

To talk with GHCi, we used CreateProcess and opened up 3 handles into its stderr, stdout and stdin. Our strategy was that a single GHCi query would consist of sending something to GHCi, then reading from stdout until the handle was finished. Unfortunately, there were many subtleties to this approach. Furthermore, dealing with handles was made difficult for many reasons.

GHC API

Our initial approach was to use the GHC API in order to get static information about programs such as intermediate types as well as to compile the program code in order to evaluate to the results. This approach was highly desirable because it would have been far cleaner. However, the GHC API is documented poorly and was just too complex for us to comprehend and make progress on within the short span of time in this class. We also looked at other projects with similar goals (Scion in particular) but we had a hard time to get it working at all, let alone understand the architecture and use their code.

Buffering

Initially, we had trouble simply with a query/response round (more concretely, send something into GHCi, and get a result). Sometimes we would get results, but sometimes

GHCi would just hang indefinitely. This turned out to be because GHCi buffers its output in a strange way. We discovered the hSetBuffering NoBuffering command and were then happily along our way.

Handles

The official way to check if a handle is hReady. Our approach to get new output from a handle was just to keep using hGetLine until hReady was no longer true, but we kept on running into problems where the hReady would report False even when we knew it was true (because we just sent input), or worse, the other way around (and then hGetLine would just hang indefinitely).

After some consternation, we asked around on IRC, and were told that hReady was unreliable, and we shouldn't really depend on it. The way we got around this problem was to force the handle to emit a sentinel at the end of the output. In that way, we could just keep running hGetLine until we saw the sentinel, and then we could just immediately stop.

Yesod

We used Yesod as our web server that rendered the HTML page that contained our REPL UI and responded to AJAX requests by querying GHCi. Since we did not build a full-fledged website with multiple pages and database, we didn't have to deal with too many of Yesod's features. It worked great for our purpose and our experience with it definitely motivates us to explore its more advanced features in future projects.

unsafePerformIO

Our project makes use of a call to unsafePerformIO to fake having "global variables" for our handles.

The problem was that we needed access to the stdout, stdin and stderr handles inside the Yesod route handles. Unfortunately, there was no way to pass them in as arguments (as we might in other languages). In order to get access to certain types of input and output (like IO, database queries, etc), Yesod wraps up the handler in a respective monads. But we really could not find a way to thread the handles through into the handlers short of rewriting Yesod source.

Data

Once you realize that you can do data definitions just by :loading a file, data seems like it should be fairly trivial. Unfortunately, this is not the case because when you :load a file, you lose all local bindings, which means that all of the values in the data inspector on the right quickly become invalidated. We wanted to retain all local bindings, which means we had to do something slightly more clever.

Our idea was that every time the user enters in a command, categorize it as either a data binding, a let binding, or something else. Write all data bindings sequentially to one file, write all let bindings sequentially to another, and ignore everything else entirely. Obviously, if what the user entered causes an error, then you don't write it to the file. Finally, when a user enters a data binding, first load the data file that we just wrote to, and then replay all the let bindings. We now have the new data binding with all of the old let bindings.

We admit that this is not the most elegant way to solve the problem, but our goal was more to create a better version of GHCi that we would use than a rigorous and efficient GHCi.

Haddock/Hoogle/HLint

We wanted to have great integration with module documentation in our REPL. However, it was frustrating that this could not be done directly by importing these modules as libraries. Instead, in order to extract the Haddock documentation, for instance, we had to use xmlhtml library to parse the HTML that the current version of Haddock generates and process it into a structured form. Of course, since we are scraping, it's not only additional work on our part but is also likely to break if the document structure should change in future versions of Haddock. For both Hoogle and HLint, the best option seemed to be parse the command line output and that's what we did.

Comparison between Haskell and JavaScript

Due to the nature of the project, almost all of the UI code is in JavaScript. We'd have liked to do it in Haskell. However, we did not want to use any Desktop UI because of the limited capabilities these libraries tend to offer and the cross platform issues. And since Haskell, not being JavaScript, is simply incapable of modifying HTML user interfaces on the client. Using ghcjs was one possible alternative but the project looks extremely unstable and not worth the potential benefits of being able to use Haskell in the browser yet.

Future directions

We saw a lot of potential for this project. More generally, we see much in bringing graphical capabilities to the traditionally command line interfaces that come with Haskell.

One feature we considered adding, but decided against, was full file input and processing. Currently, our program only processes one line of Haskell at a time, but there are clearly times when that isn't sufficient. One really handy place to have this would be in debugging traces provided by ghci, where you constantly have to refer back and forth from command line to file to follow the trace backwards. It would be really nice to have a way to scroll through these and see where they are visually, and it could save a lot of time.

Another nice feature we could have is automatic syntax error underlining, not unlike that in Eclipse. Yet another amazing feature that is perfectly suited for a language with an elegant type system such as Haskell is the ability to the visually inspect variables of different types (for example, all Image types could be displayed as images using the <canvas> element rather simply as unreadable pixel values).

On the whole, it was extremely fun to work on this project because we made ghci into something that we could hack to add features that we wanted. Many of the solutions that we came up with weren't particularly elegant but they worked well enough that we would use *ghci in a new dress* over ghci while coding Haskell in the future. Since we believe this project will greatly benefit the Haskell community, we intend to make it more robust (perhaps, by directly using the GHC API now that we have more time) and publish it.