Live Haskell

Andrew Stiles

Introduction

Live Haskell is a system which allows users to store and execute arbitrary code online. Live Haskell provides an advanced editor and allows for simultaneous execution of user code. The Live Haskell project is written entirely in Haskell and consists of two main portions: a Snap web server and the IHaskell project. Live Haskell uses Safe Haskell to ensure the safety of user-submitted code. In this paper, we will present the requisite background knowledge necessary to fully understand the project. We will then present the overall architecture of the project and explain design decisions. Finally, we will conclude with challenges, pitfalls, and future outlook for this project. References to all supporting projects as well as media such as website images will be provided.

Background

There exists a fair bit of background information necessary to understand this project. We will break the background information into three parts, one of which will contain a sizeable subportion. The three principal components necessary for the operation of Live Haskell are the Snap web framework, the IHaskell project, and Safe Haskell. IHaskell will include description of IPython which is its supporting framework.

The Snap Web Framework

The Snap web framework is a Haskell web framework which emphasizes simplicity and modularity. To promote modularity, Snap uses the concept of snaplets which are “self contained pieces of functionality that you can include in your web apps”[1]. The two relevant snaplets used in Live Haskell are the heist snaplet for templating and the auth snaplet for authentication. The Live Haskell application itself is a snaplet meaning it contains its own state and configuration. One aspect of the Snap framework with which one should be somewhat familiar when developing Snap applications is that of lenses. Due to the modular nature of the framework, snaplets must keep track of their state in the context of a base state (that of the entire application). Lenses allow a record in the snaplet to be updated within the context of the base state. A succinct description of lenses is that they are first class references to the subpart of a data type. The Snap framework makes use of lenses in the templating system, heist, to ensure that only one instance of heist is instantiated at any given time. After obtaining a reference to heist, values are bound to heist templates through the bindSplices function. String values may be bound to the heist template using the following form:

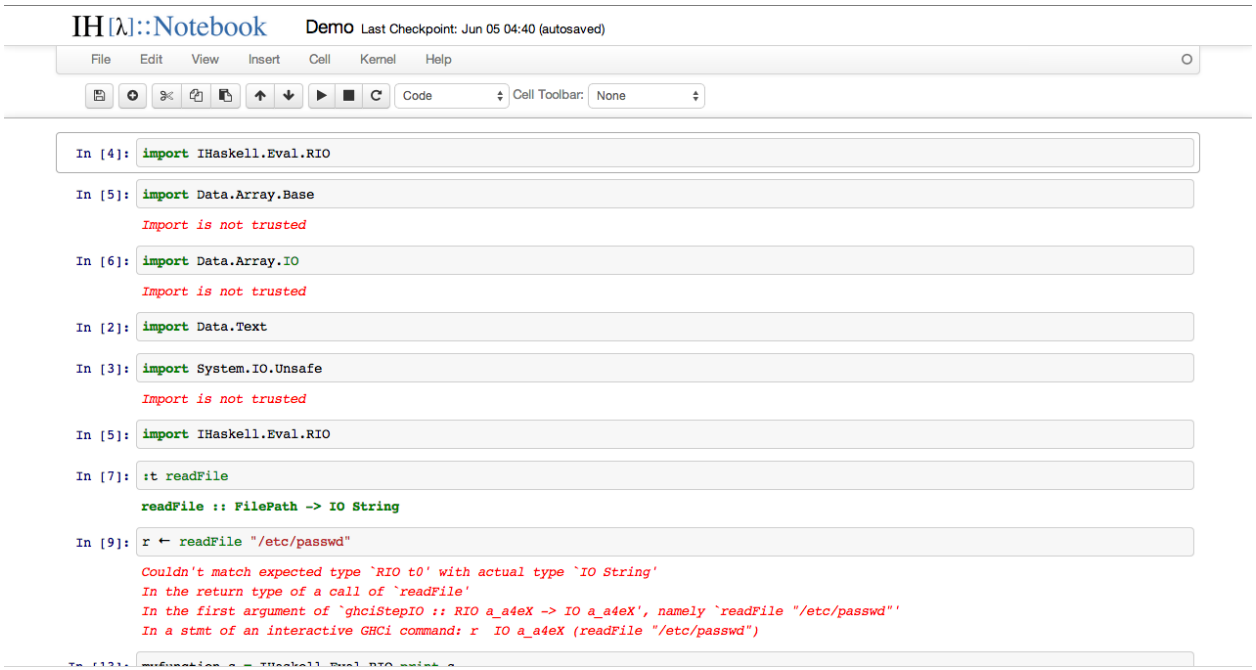
```
"loginError" ## I.textSplice err
```

[1] http://snapframework.com/snaplets

where err is of type Text. Binding using splices was key to the development of Live Haskell as will be further discussed in the architecture section of this paper.

IHaskell

IHaskell is “an implementation of the IPython kernel protocol which allows you to use Haskell inside IPython frontends”.[2] For the uninitiated, an IPython frontend is an interactive shell which provides features such as “syntax highlighting, autocompletion, multi-line input cells, [and] integrated documentation”.[3] Of particular relevance to IHaskell is the IHaskell notebook interface. This interface is complementary to the console interface provided by IHaskell and an example notebook may be found at http://gibiansky.github.io/IHaskell/demo.html. Another example of the notebook running is the following:



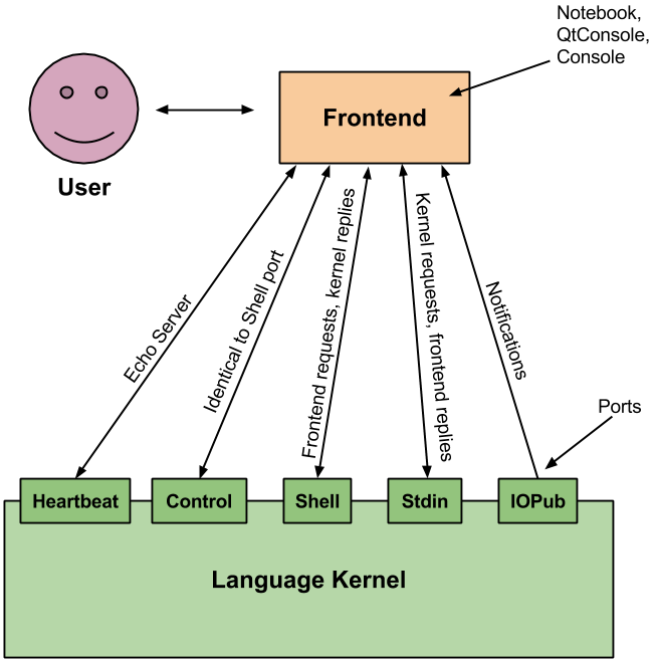
To fully understand the infrastructure of IHaskell, one must understand that of IPython on top of which IHaskell is based.

IPython

IPython has adopted a client/server architecture to accomplish its goal of providing a sophisticated, interactive notebook. Roughly defined, the IPython system may be broken up into two parts: the kernel and frontend. The IPython notebook is an example of such a frontend. It communicates with the IPython backend through a zeromq interface. Briefly, zeromq is a transport layer built on top of TCP. It has the notion of abstract sockets to which nodes can bind. Thus, the overall architecture of the IPython framework is a frontend notebook server which

[2] https://github.com/gibiansky/IHaskell
[3] http://gibiansky.github.io/IHaskell/demo.html

communicates with a kernel running the haskell executable. It is worth noting that in its default configuration, IPython serves notebook files out a directory it creates. Also, custom ports and IP addresses can be specified in IPython but IHaskell provides no such facility. There are four channels by which the framework communicates with its backend: a heartbeat channel, a shell/control channel, the stdin channel, and the IOPub channel. The heartbeat channel is used to determine the status of the kernel and whether it has died. The shell/control channel uses zeromq's router sockets to create a lockstep request/reply mechanism by which code snippets are run. The stdin channel allows the kernel to query the frontend. Finally, the IOPub channel is the channel which actually publishes output. It should be noted that even though the notebook is itself a server, it is the frontend component of IPython. While IPython theoretically supports multiple frontends connecting to multiple kernels, the author of IHaskell has not tried this and believes it requires additional work to set up properly. A diagram of the IPython architecture is included here:



Safe Haskell

Safe Haskell is a system which allows for the safe execution of untrusted code. It is central to the operation of Live Haskell. Specifically, Safe Haskell guarantees referential transparency, module boundary control, and semantic consistency. Referential transparency refers to functions being pure and free of side effects upon evaluation. Module boundary control refers to code respecting the stated exported functions modules provide. Semantic consistency refers to the property that any code executed within the context of Safe Haskell will have the same meaning outside of the context of Safe Haskell.

Safe Haskell allows for the importing of modules. Modules may either be mechanically validated as safe or marked as trustworthy by the user. Currently, Live Haskell uses the -XSafe flag which asks GHC to mechanically verify imports. While one may be concerned that this presents a

security loophole, through testing, the author was unable to import dangerous modules. More testing in the future would help validate that this is the case. Take the following code as an example:

```
{-# LANGUAGE Trustworthy #-}
module FOOBAR where
    myfoo x = unsafePerformIO (x ++ " testing")
```

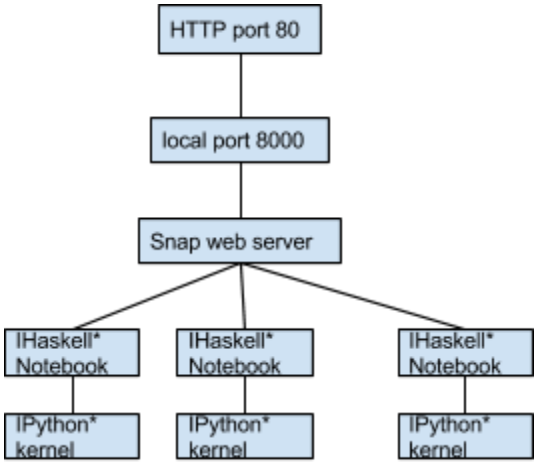
The code is unable to compile and an error message stating that the flags Safe and Trustworthy are incompatible is thrown.

The last piece of the Safe Haskell puzzle is to define a restricted IO monad. Such a construction is necessary for the reason that Safe Haskell does not by itself guarantee that code does not perform dangerous IO actions. One may think of Safe Haskell itself as guaranteeing that code does what it says it will do. However, without a restricted IO monad, user code could simply state that it performs a semantically dangerous action (e.g. stealPasswords()) and then perform it. Thus, in Live Haskell, a restricted IO monad is defined to only allow the IO actions print, putStr, and putStrLn.

Architecture

Now, we will present a description of Live Haskell's architecture. This description will assume knowledge of the preceding sections of this paper.

Live Haskell consists of a Snap web server running atop several instances of IHaskell notebooks. Each individual user is mapped to a separate IHaskell instance. The reason for this design choice is that a single IPython kernel does not yet work out of the box with multiple IHaskell frontends. In planning for this project, it was desired that each user's session be completely independent of each other. Visually, the architecture looks as follows:



* modified from the project's (e.g. IHaskell or IPython) implementation

The main task of the Snap web server is to map IHaskell instances to unique users. It accomplishes this by maintaining a small database of user metadata (such as whether a IHaskell instance has been started for a given user). The state of whether a IHaskell instance has been started for a given user is maintained in memory and the state of whether a user has signed in is maintained in the database.

If a user is found to not have an entry in the in-memory mapping of users to IHaskell instances, a new IHaskell instance is created and an entry inserted into the mapping. A record of the fact that an instance had been started is persisted in the on-disk database. In future versions of Live Haskell, this behavior will be modified as it can (under some circumstances) lead to state synchronization issues between the Snap server and the underlying instances. As it stands, in an ideal scenario, the Snap server would not have to be restarted because doing so would destroy the in-memory mapping.

In brief, when a new user comes into the system, a new IHaskell frontend/kernel is started on a unique port and a link is generated for the user to access that port. As IHaskell must serve files from a particular directory, a unique directory for a user's IPython notebooks is created when a new user enters the system and IHaskell is pointed at that directory.

The coupling between Snap and the underlying IHaskell instances is quite loose. The Live Haskell project actually builds two executables: "LiveHaskell" and "IHaskell". The "LiveHaskell" executable contains the Snap web server and starts the "IHaskell" executable via shell commands. Shelly[4] was used to accomplish this. Parameters are dynamically passed to the "IHaskell" executable to ensure it starts on the proper port, address, and serves the correct directory. While IHaskell previously had an option to serve a specific directory ("--serve=...") the options to specify a port and IP address ("--port=..." and "--ip=..." respectively) did not exist and had to be built into IHaskell.

Architecture Summary

The key portions of the architecture are the following:

- A Snap web server sits atop several running IHaskell notebook/kernels
- A mapping exists between users and IHaskell instances
- Two executables are created, "LiveHaskell" and "IHaskell"
    - "LiveHaskell" contains the web server portions of the system and "IHaskell" contains the underlying IHaskell notebook/kernel
- The IHaskell project was modified to accept additional parameters, namely a port and IP address. The executable is dynamically called by the LiveHaskell executable.
    - e.g. "IHaskell --ip=0.0.0.0 --port=9000 --serve='~/.ihaskell/mydirectory'"

[4] https://hackage.haskell.org/package/shelly

- The IHaskell project was modified to be safe (as will be described in the following section)

IHaskell Modifications

As described earlier, IHaskell was modified to start on a user-supplied IP address and port. However, we will address in this section the modifications necessary to ensure safety.

Firstly, some portions of IHaskell had to be disabled to allow for arbitrary code execution. The two main parts disabled were the display module and the dynamic setting of extensions / magic commands. IHaskell ships with a display module which could not be mechanically verified as safe. Perhaps in future versions of Live Haskell, this can be integrated to allow for some nice display mechanisms shown on the IHaskell web site. Dynamic setting of extensions had to be disabled so that users could not bypass the -XSafe flag. IHaskell also ships with the ability to run terminal commands which had to be disabled for obvious reasons. Some kernel options such as linting were left intact as they did not interfere with Safe Haskell.

To enable Safe Haskell, a dynamic session flag was set in code.

Much of the work in enabling Safe Haskell came in figuring out how to specify and set a custom IO monad (which was called RIO). The RIO monad was defined much as in the documentation for Safe Haskell. It is both an instance of Monad and GHCiSandboxIO. It contains an unexported UnsafeRIO constructor and a function “runRIO” that converts between RIO and IO actions. While there is some understanding necessary to create such a monad, it is not terribly difficult and the Safe Haskell documentation does a good job in facilitating such creation.

Once the custom monad is created, it must be integrated into the IHaskell ecosystem. This proved to be a difficult challenge. The first approach explored was creating a custom GhcMonad instance which used RIO as its IO monad. However, this approach proved less than fruitful. RIO was made an instance of Functor, ExceptionMonad, and MonadIO and then the GhcT monad transformer was used to imbue RIO with GHC-specific functions. However, IHaskell contained some internal machinery which required that IO actions be performed before and after statement evaluation.

Fortunately, David Terei had contributed a function “setGHCiMonad” to the GHC API which sets the monad GHCi lifts user statements into. The only requirement of this function is that it work with an instance of GhcMonad[5]. Fortunately, IHaskell used Ghc[6] by default which is an instance of GhcMonad. While the types checked, there were still errors due to the setup and teardown commands in which IHaskell wraps its user statements. While this was a major roadblock for

[5] http://www.haskell.org/ghc/docs/7.6.1/html/libraries/ghc-7.6.1/GHC.html#t:GhcMonad
[6] http://www.haskell.org/ghc/docs/7.6.1/html/libraries/ghc-7.6.1/GHC.html#t:Ghc

some time, it was eventually solved by dynamically setting the GHCiMonad. IHaskell defines the function “goStmt” which looks like the following:

```
goStmt :: String -> Interpreter RunResult
goStmt s = do
       runStmt s RunToCompletion
```

The function was modified to contain the following:

```
goStmt :: String -> Interpreter RunResult
goStmt s = do
       setGHCiMonad "IO"
       runStmt s RunToCompletion
```

A new function was defined as the following:

```
goStmt’ :: String -> Interpreter RunResult
goStmt’ s = do
       setGHCiMonad "RIO"
       result <- runStmt s RunToCompletion
       setGHCiMonad "IO"
       return result
```

goStmt’ was then wrapped around the user statement execution and goStmt was wrapped around the IHaskell setup and teardown.

Conclusion

Live Haskell was a great learning opportunity. It incorporated several technologies and exposed the author to a diverse range of Haskell topics. The implementation of Live Haskell was successful in that it accomplished its stated goal - execution of arbitrary user code. However, there exists room for improvement. While the model of assigning individual users each an instance of the IHaskell notebook/kernel works on a small scale, it would have to be accommodated to allow for more users. There is not much security for authors of Live Haskell notebooks because a user’s port number can be guessed (e.g. livehaskell.com:<user_port_#>). There also exists an unresolved zeromq issue when many users try to access Live Haskell simultaneously. It seems like using sockets to route users within Snap instead of sending them to different port numbers could resolve some of these issues. Another area which could be made more robust is the communication between the LiveHaskell and IHaskell executables. Perhaps zeromq could be used to maintain decoupling while providing a means of communication.

Even given that there is room for improvement, the author is generally optimistic about the outlook of IHaskell. The goals of providing a rich and safe user experience were realized. The author plans to continue work on Live Haskell for personal fulfillment.

Site/Code

livehaskell.com
https://github.com/aostiles/LiveHaskell

Related Links

https://www.fpcomplete.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial
https://hackage.haskell.org/package/snap-0.13.2.7/docs/Snap-Snaplet.html
http://hackage.haskell.org/package/lens
http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/safe-haskell.html
https://github.com/gibiansky/IHaskell
http://safehaskell.scs.stanford.edu/
https://github.com/ghc/ghc/blob/master/libraries/base/GHC/GHCi.hs
https://github.com/dterei/GhciSafe/blob/master/ghci-safe/Main.hs#L264
http://www.haskell.org/ghc/docs/7.6.1/html/libraries/ghc-7.6.1/GHC.html

Supporting Work

Safe Haskell, IHaskell, Snap, and IPython. Much thanks to David Terei, Andrew Gibiansky, Edward Yang, enthropy (IRC).