Haskell Autograder

Taylor McKinney

Introduction

For my final project, I set out to create an autograder that could dynamically load files and run tests against the functions within them. I found it to be nearly impossible to dynamically load modules purely in Haskell, for reasons described in the Discussion section below, and ultimately resorted to using a simple bash script to load and compile the files, and then pass them as a command line argument to the main driver of the program, which is in Haskell. Being a purely functional language, Haskell can be difficult to use for IO purposes, so finding a way to complete an IO-centric project required some creativity and flexibility in my approach. The organization and contents, usage techniques, and discussion about development for this project are outlined below in this report.

Project Organization

Project File Structure

- *run.sh* - The bash script that loads each file and passes it to the main driver, *ioTest.hs*.
- *ioTest.hs* - The main module and the driver of the program. Takes a student's solution file, extracts a student ID from a comment, modifies all function names to be concatenated with the student ID, appends the functions to the output file, *ans.hs*, generates HUnit test cases for all functions, and finally appends those to the output file. Organization and functions further described below.
- *ans.hs* - The output file that is written to with the student functions and HUnit test cases. Must have an "import Test.HUnit" statement and a module declaration, but otherwise the initial contents of the file are irrelevant.
- *tests/*.hs* - The directory and student solution files that are opened, processed, and tested.

Main Module Structure

Expectations/Assumptions Made:

- Main assumes that the string passed in as a command line argument is a valid file path to a .hs file. Using *run.sh* ensures this, but when running the executable itself, it's up to the user to provide a valid file.

- Main assumes all functions listed in the 'funcs' variable of the **main** method are at least declared in the student file in a way that type checks. Main does not check for missing functions, and the output file will not compile if a function is missing.
- Main assumes the very first line of the student's file is a comment with their unique student ID. This does not necessarily have to be 9 digits to be a banner ID, but for the output file to compile, there needs to be a comment with at least one digit in the first line, and each file that is processed and added to the output file needs to have a unique ID.

Import Statements and Functions Used:

- System.IO: readFile, appendFile
- Data.Char: isDigit
- System.Environment: getArgs
- Data.List.Utils: replace, contains
    - *Note*: This library gives the warning `Deprecated: "Use Data.List.isInfixOf, will be removed in MissingH 1.1.0"`, but for now it is still working.
- Test.HUnit: assertEqual, TestList, TestLabel, TestCase (in output file)

Main Method: The main method first gets the file name from the command line arguments using **getArgs**, and then uses **readFile** to lazily load the contents. The **getStudID** function then takes the file input and extracts the student ID from the comment on the first line; if there is no comment or the comment is not at least one number, an empty string will be returned; if there is more than one file in the directory without a number or with the same number, the output file will not compile because it will have two declarations for the same function name. The **findFunc** function then uses the student ID and the file input, broken into a list of Strings separated by the newlines using **lines**, to search for each function name and concatenate the ID to it, as well as ignore comment lines and module declarations. The function names that the **findFunc** function looks for and modifies must be hard-coded by adding guards for the cases where the name appears, as well as for the cases where more than one function name is in a single line. The **getCases** function then takes a list of function names and the student ID and generates HUnit test cases for each of them, in String form. Finally the main function uses **appendFile** to append the student code with the updated function names, back in String form using **unlines**, and to append the test cases in a properly formatted TestList, named tests[STUDENTID]. The *run.sh* script takes a directory with .hs files, and then for each file compiles and passes it to ./ioTest as an argument. The main method can also be run using a single student file from the command line, but it does not check if it is a valid file and could potentially have an error at runtime. To make testing easier, a TestList named allTests was hard-coded into the output file to link together all of the student tests, but without any further modification, student tests from *ans.hs* can individually be run in GHCi using the command `runTestTT tests[STUDENTID]`.

```
main :: IO ()
main = do
         args <- getArgs
         let file = parse args
         inp <- readFile file
         let id = getStudID inp
             new = findFunc id (lines inp)
             funcs = ["countZeros", "printRes"]
             cases = getCases funcs id
         appendFile "ans.hs" (unlines new)
         appendFile "ans.hs" (" tests" ++id ++ " = TestList [ \n")
         mapM (appendFile "ans.hs") cases
         appendFile "ans.hs" "\t\t] \n"
```

Helper Functions:

- **parse :: [String] -> [String]** parses the command line arguments to extract the filename
- **getStudID :: String -> String** extracts the student ID from the comment on the first line of the student file, the input of which is its parameter.
- **findFunc :: String -> [String] -> [String]** steps through the student file line by line, looking for instances of hard-coded function names and concatenating the student's ID to them. This function excludes comment lines and module declarations.
- **getCases :: [String] -> String -> [String]** generates the HUnit tests for each function in the first parameter, appending the student ID to the names of the functions and tests. This function takes into account the fact that the last TestCase in a TestList should not have a trailing comma, but all of the others should. To make altering test cases easier, the *correct* answer and *input* fields for each function have been separated into external strings.

Usage

To compile and run using run.sh:

```
>  ghc ioTest.hs
>  run.sh
>  ghci ans.hs
Main> runTestTT allTests
```

To compile and run using a single file:

```
> ghc ioTest.hs
>  ./ioTest "file.hs"
> ghci ans.hs
Main> runTestTT tests[STUDENTID]
```

Where [STUDENTID] is the number commented out in the first line of *file.hs*, without brackets.

How to Modify Test Suite:

Unfortunately, because the **main** method is using Strings to represent the student code rather than dynamically loading the modules within one file, there is a fair amount of hard-coding required to add or modify functions and test cases. For the sake of demonstrating the project works, I used two simple functions, **countZeros :: [Integer] -> Integer**, which counts the occurrences of 0 in a list, and **printRes :: Integer -> String**, which produces a string saying "there are x zeros" where x is the integer passed to it. I implemented these functions in two simple test files: *tests/test.hs,* which has a correct implementation, and *tests/test2.hs*, which has a purposefully incorrect implementation and output. To add a new function to the test suite: add cases to the guards in **findFunc** and **getCases**, as well as cases to **findFunc** for when some or all of the functions are in a single file, and add the function name to the 'funcs' variable in the **main** method. You will also need to declare a few Strings to represent the *correct* answer and *input* to each function, paying attention to the formatting guidelines in the function documentation. To change the test case itself, simply modify the appropriate *correct* and *input* fields. An alternative way to test different cases for each function would be to hard code more TestCase strings in each case of the **getCases** function and combine them using (:), but for clarity in the already complex string composition I chose to just leave it at one TestCase per function, with the correct and input strings separated out to avoid redundancy.

Discussion

When I first started on this project, I really wanted to do all of the file handling / module loading dynamically within a single Haskell file, but I very quickly learned that is not an easy feat. This isn't Haskell's strong point at all, and most of the forum and blog posts I found when I started researching told me I should choose another language or approach the project differently. I was able to find two packages that exist for dynamically loading / linking modules by interfacing the GHC API: plugins and hint, but they both have their problems. Plugins is mostly antiquated, and I found this reddit thread with a comment from the user stepcut251, who is the maintainer of the plugins library, saying that it essentially is no longer a viable option without major editing, of which requires far more Haskell expertise than I possess. I tried using hint, but depended on versions of other packages that I could not install, because that would then conflict with even more packages. After scouring through all of the resources I could find on the package, most of which were upwards of three years old, I found more problems than I did solutions. I tried a few other libraries I found, including Dynamic Loader, but kept running into dependency issues and runtime errors due to missing functions in the packages. After tinkering with all of these packages and finding no success, I ultimately decided to change course and use a bash script to help with the file handling.

I still wanted as much of the work to be done in Haskell as I could manage, so I limited just the scanning of the directory, obtaining the file name, and compiling of the student file to the bash script *run.sh*. Because I couldn't find a way to have the **main** method dynamically open modules, and I knew that all of the student files would have identical function declarations, I needed to find a way to differentiate the functions between students, but still run all of the tests using one GHCi command. This is when I came up with the idea of processing the student files as strings, modifying each function with an identifier, and outputting the code as a String to a separate file, which would also contain the tests. This way, the Main module doesn't need any of the student modules, or any of the actual testing code. The biggest downfall of this approach is the amount of hard-coding necessary, but the tradeoff is that the user only needs to minorly modify the *ioTest.hs* file once, and potentially the *ans.hs* file, to setup the test suite to run every test on every function in every file in the given directory. Another downfall of this implementation is the assumptions the program makes, but the use of *run.sh* mitigates some of them, such as knowing that the file exists. I briefly considered using the file name as an identifier, but I recalled the way AsULearn sometimes modifies file names, so I chose to just use a comment in the first line with the student's Banner ID. A benefit to this method is that in the output file, all the functions are clearly associated with a student, and the Banner ID is printed for each failing test, hopefully making it easier to quickly grade an entire assignment.

Conclusion

Haskell is not made for IO, and it certainly is not made for dynamic loading of source files. I sincerely tried to use the packages I found and follow the few tutorials available, but it seems that most of the tools to do this have depreciated with newer versions of GHC and other packages. The final solution uses metaprogramming techniques to create a separate Haskell file, which I think ultimately helped me learn more than I would have if those packages had worked the way I hoped. I originally wanted the driver to be able to process multiple student files on its own, but the nature of Haskell made that exceedingly difficult, so I am happy with it being able to process one, and using the bash script to add multiple file capability. In future extensions I would like to reduce the amount of hard-coding required for setup, but it definitely will take some thought on how to do that.