Prof. Dr. Peter Thiemann
Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Installation and Setup

Overview

This document will guide you through the installation process of the Haskell toolchain and provides basic usage examples.

In particular, it covers how to install and use the following tools:

- *Glasgow Haskell Compiler (GHC)*, which consists of
  - a compiler, `ghc`, which compiles Haskell code to machine code;
  - a REPL, `ghci`, which allows you to interactively play around with the definitions of your source code - similar to running `python` without arguments; and
  - an interpreter, `runhaskell`, which can run Haskell code directly.
- `cabal`, which is Haskell's package manager and project manager.
- *Haskell Language Server (HLS)*, which is a Language Server Protocol (LSP) implementation for Haskell, which provides the usual IDE functionality to the code editor of your choice. The provided IDE functionality includes auto-completion, goto-definition, intellisense, code actions, and identifier renaming. Supported editors include VSCode, vim, neovim, and emacs.
- *GHCup*, which is a version manager for the Haskell toolchain, i.e. it provides a simple way of installing all of the above tools and also allows to quickly change their versions if needed.

This document additionally covers how to set up VSCode for Haskell development. For other editors, please refer to the documentation of HLS for further information: https://haskell-language-server.readthedocs.io/en/latest/configuration.html#configuring-your-editor

Installing GHCup

Follow the installation instructions at https://www.haskell.org/ghcup. Make sure to read the messages printed to the screen carefully. Below we provide some additional pointers and suggestions which answers to provide to the prompts during the installation.

*nix Systems

If the command fails immediately you might have to install `curl` first, e. g. by issuing `apt install curl` or similar.

Suggested answers:

- **p**repend the binary location to `PATH`.
- **y**es, install the Haskell Language Server.
- **n**o, do not improve integration with *stack*.

Windows

The official Getting Started instructions contain a video showing the installation process.

The default choices mostly suffice except for

- **y**es, install the Haskell Language Server.
- **n**o, do not install *stack*.

The installation will then proceed to download the actual GHCup executable. On *nix Systems you will have to verify that the listed system dependencies are installed before downloading GHC, `cabal`, HLS, and finally adjusting your `PATH` variable.

Verifying the Installation

If the installation completed as planned, you should be able to run `ghcup list` to get a listing of available and installed components. On *nix systems `ghcup tui` provides a (poor man's) graphical user interface. Remember to execute these commands in a new shell session, e.g. by opening a new terminal window.

Verify that the output of `ghc --version` and `cabal --version` matches the versions shown by `ghcup list`. If the output tells you that the `ghc` or `cabal` commands were not found, or if the displayed versions are different from the `ghcup list` output, then most likely there was a problem in setting up the `PATH` variable.

Using GHC Directly

In this section, we show how to compile, interpret and interactively examine a simple Haskell file using GHC directly.

Create a file `test.hs` with the following content:

```
myFunction :: Int -> Int
myFunction x = x + 1
```

Open a terminal in the same directory as `test.hs` and run `ghci test.hs`. This should start the interactive interpreter REPL and present you with an output similar to the following:

```
$ ghci test.hs
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /nix/store/iicbh0v8fim7r8ilfr5aagymbwzymgz4-hm_ghci
[1 of 2] Compiling Main             ( test.hs, interpreted )
Ok, one module loaded.
>>>
```

All definitions of your `test.hs` file are available in this REPL, so you can for example run `myFunction` as follows:

```
>>> myFunction 41
42
```

The REPL accepts special commands which start with a colon. In particular, `:q` quits the ghci program, and `:r` reloads the `test.hs` file, which can be useful to quickly reload and test your code after you changed it.

To compile or interpret a Haskell file, it needs to provide a `main` definition. For this purpose, add the following code to your `test.hs` file:

```
main :: IO ()
main = print (myFunction 41)
```

You can now write `runhaskell test.hs` in the terminal to directly interpret the Haskell source code, e.g.

```
$ runhaskell test.hs
42
```

To compile the code, you can instead run

```
$ ghc test.hs
[2 of 2] Linking test
```

which produces a file `test` that contains native machine code of your CPU architecture. You can then run this file with

```
$ ./test
42
```

Note: on Windows, the file generated by `ghc test.hs` might be called `test.exe` instead of `test`.

Using GHC via cabal

Using GHC directly can be nice for quick tests, but for an actual project it makes sense to use `cabal` instead.

We recommend to always use `cabal` instead of `ghc` and `ghci`, as some code editors assume that Haskell files are opened in the context of a project and might misbehave if they don't find a corresponding `.cabal` file.

To create a Haskell project with `cabal`, first create a new directory `test-project` and open a terminal in that directory.

Running the command `cabal init` will then lead you through an interactive process of initializing a Haskell project in that directory. You can simply choose the default for all questions by repeatedly pressing the *Enter* key. Most questions are only relevant if you plan to publish your project to Hackage[1] at some point.

After `cabal init` finished, the `test-project` directory will contain the following files and directories:

```
test-project
├── app
│   └── Main.hs
├── cabal-project.cabal
├── CHANGELOG.md
└── LICENSE
```

The files have the following meaning:

- The `app` directory is where you place all the Haskell source files of your application. In particular, the `app/Main.hs` is the entry point to your application and needs to define `main`. It is initialized by `cabal` with the following hello world program:

  ```
  module Main where

  main :: IO ()
  main = putStrLn "Hello, Haskell!"
  ```

[1] Hackage is Haskell's package repository. It contains all libraries and applications, which can be installed via `cabal`. Its webfrontend can be found at https://hackage.haskell.org/, which is commonly used to search for libraries and to look up documentation.

- The `cabal-project.cabal` file contains a description of the project, its dependencies and lists its source files. `cabal` needs this information to derive the correct GHC invocations.
- The `CHANGELOG.md` and `LICENSE` are for documentation in case you want to publish your project and you can simply delete them if you want.

By executing `cabal run` from inside the project directory you can compile and run your executable, similar as we did manually in the previous section. After some output regarding the build you should see `Hello, Haskell!` printed to the screen. Afterwards you will be able to see an additional directory called `dist-newstyle` which contains the build products and intermediate files. If your project depends on libraries, then those are also automatically downloaded into the `dist-newstyle` directory the first time you run `cabal run`.

The command `cabal repl` will drop you into a `ghci` session in the context of the project, similar as we did with `ghci test.hs` in the previous section. However, in contrast to `ghci test.hs`, the dependencies of your project are also available and all source files will be loaded automatically. If you are working in the context of a project it is always preferable to launch GHCi sessions via `cabal repl` compared to bare `ghci` invocations.

Using Libraries

In this section, we show how to use a library in a `cabal` project. As an example we use the QuickCheck library for property based testing, which was also briefly shown in the lecture.

Open the `cabal-project.cabal` file from the previous section and replace the line

```
    build-depends:    base ^>=4.18.2.1
```

with

```
    build-depends:    base ^>=4.18.2.1, QuickCheck
```

If you now execute `cabal run`, then cabal will automatically download and install the QuickCheck library into this projects `dist-newstyle` directory.

This enables you to import the modules provided by this library in your source code. For example, you can replace the content of your `app/Main.hs` with

```
module Main where

import Test.QuickCheck

myFunction :: Int -> Int
myFunction x = x + 1

prop_myFunction :: Int -> Bool
prop_myFunction x = myFunction x == x + 1

main :: IO ()
main = quickCheck prop_myFunction
```

If you now execute `cabal run` again, then QuickCheck will make sure that your `myFunction` indeed increments its argument by calling `prop_myFunction` with 100 random integers and checking that all calls return `True`:

```
$ cabal run
+++ OK, passed 100 tests.
```

The reference documentation for Haskell libraries can be found on Hackage, e.g. `https://hackage.haskell.org/package/QuickCheck`

VSCode Setup

VSCode can be downloaded from `https://code.visualstudio.com`. Then install the Haskell extension.

When you open your first Haskell file, let's say `Main.hs` from the previous section, the extension will ask you how it should manage/discover the Haskell Language Server. Choose "Automatically via GHCup."

For a quick example, remove the type signature (that is all of line 3). In the beginning it may take a few seconds to startup but eventually you should see the type signature re-appear in small gray text. Clicking on said text will insert the type signature. Removing the `putStrLn` will give you an example how type errors are presented.

Note: (but speaking only from one experience) on a Windows system you might have to close and reopen VSCode a few times for everything to get settled. As soon as you can follow along with the steps described in the paragraph above, everything works as expected.