Making Haskell Projects

CIS 194 Week 13
22 April 2015

Suggested reading:

- **How to Write a Haskell Program**

Encapsulation in Modules

When you are writing a Haskell module, you will often include helper functions that are useful internally, but are not useful to users of your module. Haskell provides a very easy way to hide these helper function. Let's say that we are writing a Binary Search Tree module. Up until now, we would have decleared the module header like this:

```
module BST where
...
```

This exports *everything* that is visible inside the body of the file. And yes, *everything* includes functions and datatypes that are imported from other modules. To fix this, we are going to explicitly say what we would like to export.

```
module BST ( Tree
           , insert
           , remove
           ) where
```

Right now, we have only one datatype and two functions that we are planning to write. We will update the module declaration later if we choose to export more stuff.

Notice that we are only exporting the data *type* `Tree`. This means that users of our module will not be able to see what data *constructors* `Tree` has. If we wanted to export the constructors too, we would write `Tree(..)`. Is it a good idea to hide the constructors?

In this case, yes it is a good idea. Binary Search Trees have very specific invariants. If we give users access to the underlying structure of the tree, then they can easily break the invariants. By hiding the constructors, we can ensure that users will only be able to manipulate a `Tree` using the functions that we expose to them.

Is there a situation where it would be a good idea to expose only some of the constructors to the user?

Building with Cabal

So far, we have only used cabal to install packages from hackage. However, cabal is also a very powerful build system. Whenever you are writing a sizeable Haskell project, you should create a `.cabal` file that states how to build your project, and what the external dependencies are. Having this around will also make it really easy to upload your code to Hackage!

To get started making your `.cabal` file, just use the `cabal init` command in your project directory. You will be prompted with a series of questions about your project. For most of them, the default answer will be fine. The `.cabal` file for our BST module might look like this:

```
name:                bst
version:             0.1.0.0
synopsis:            A module for efficient BSTs
```

```
license-file:        LICENSE
author:              Haskell Curry
maintainer:          hcurry@haskell.org
category:            Data
build-type:          Simple
cabal-version:       >=1.10

library
  exposed-modules:     BST
  build-depends:       base >=4.7 && <4.8
  default-language:    Haskell2010
```

Once your `.cabal` file is created, you can simply type `cabal build` to build your project!

For more information on creating projects with cabal, check out this **guide**.

Adding Test Suites

You will probably write a bunch of tests for your project, and you might want an easy way to run them all. Good news! Cabal supports test suites too!

You can write your test cases using the framework of your choice (ie QuickCheck, HUnit, etc). You can also name your test files whatever you want, but make sure that the *module* name is `Main`. Let's say we wrote some QuickCheck tests in a file called `TreeTests.hs`. We could then add the following bit of code to our `.cabal` file:

```
test-suite tests
  main-is:             TreeTests.hs
  type:                exitcode-stdio-1.0
  build-depends:       base >=4.7 && <4.8, QuickCheck >=2.6 && <2.7
  default-language:    Haskell2010
```

To run these tests, we would simply type `cabal test` in the project directory. This makes it really easy to repeatedly run tests as part of your development process!

You have to be a little careful about how you use QuickCheck with cabal. Notice that the *type* of the test suite is `exitcode-stdio-1.0`. Cabal determines whether or not the tests passed based on the exit code of the program. running QuickCheck has no effect on the program output, so you have to manage this yourself. By default, all Haskell programs return exit code `0`, meaning that they ran successfully. To return an error exit code, you can use the `exitFailure` function from `System.Exit`. You therefore need to check to see if *any* of you QuickCheck properties failed, and if so, call `exitFailure`. See the provided code for an example of how to do this.

Documentation

Documentation is a necessary evil for programmers. Luckily, it is really easy to generate pretty documentation for Haskell code! To do this, we will use a tool called Haddock. The name Haddock is a clever pun that combines the words *Haskell* and *Documentation*.

To get basic Haddock documentation you really don't need to do anything, just run the command `haddock <filename>.hs -o <path to docs dir> -h`. You will get a bunch of HTML files and if you open them you will se something that looks very similar to the documentation you see on Hackage. It looks very similar because Hackage docs are generated using Haddock.

If you just run Haddock without doing anything special to you code, it will be very sad and you will se something like this:

```
Haddock coverage:
   0% (  0 /  6) in 'BST'
```

Whenever you run Haddock, it reports to you how much documentation *coverage* you have. Right now we have 0% coverage since we have not added any anotations to our code. Let's revisit our `BST` module and write some annotations for the `insert` function:

```
-- |Insert an element into a BST
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node Empty x Empty
insert x (Node l y r) | x < y     = Node (insert x l) y r
                      | x > y     = Node l y (insert x r)
                      | otherwise = Node l x r
```

Comments that start with `-- |` are Haddock comments. When you run Haddock, these comments get extracted and used in the generated files.

You can get really fancy and annotate the function arguments as well:

```
-- |Insert an element into a BST
insert :: Ord a
       => a      -- ^Value to be inserted
       -> Tree a -- ^Tree to insert into
       -> Tree a
insert x Empty = Node Empty x Empty
insert x (Node l y r) | x < y     = Node (insert x l) y r
                      | x > y     = Node l y (insert x r)
                      | otherwise = Node l x r
```

The `-- ^` comments do the same thing as `-- |` except they refer to the thing before them instead of after. There are all kinds of other markup techniques you can use to make your Haddock documentation look really pretty. To find out about them, check out **this guide**.

In general, it is good practice to prove a short comment about what each function you write does. Comments in the bodies of functions are generally discouraged since each function should perform one *small* operation, and the body should be fairly self-documenting. Comments for arguments should only be used when it is not obvious what each argument is for. For example, when a function takes in multiple arguments of the same type, it is often unclear which argument is which.

Other Best Practices

Using version control is always a good idea when working on a large project. Git is a standard version control tool to use that many people are familiar with.If you want to be super cool, you can use darcs which is written in Haskell!

It is also a good idea to use HLint to catch style violations! HLint does a really good job of catching common mistakes and suggests how you can fix them, so you barely even need to think about it.

Generated 2015-05-17 10:25:31.363569