Lecture 8a. Project management

Functional Programming 2017/18

Alejandro Serrano

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

The big picture

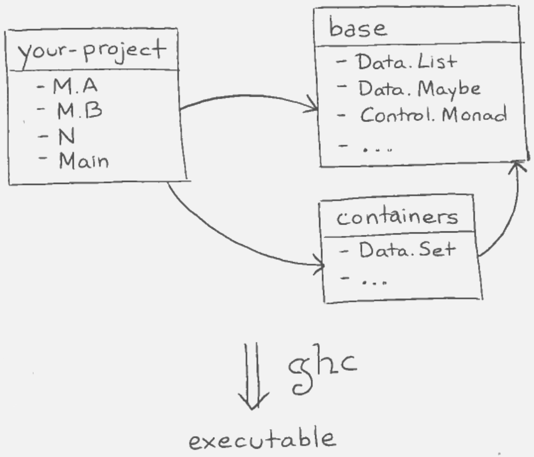

Figure 1:

Packages and modules

- **Packages** are the unit of distibution of code
  - You can *depend* on them
  - Hackage is a repository of freely available packages
- Each packages provides one or more **modules**
  - Modules provide namespacing to Haskell
  - Each module declares which functions, data types and type classes it *exports*
    - “Public” declarations in other terminology
  - You use elements from other modules by *importing*

Project in the filesystem

```
your-project ................................ root folder
 |__ your-project.cabal ....... info about dependencies
 |__ src ........................... source files live here
      |__ M
      |    |__ A.hs ..................... defines module M.A
      |    |__ B.hs ..................... defines module M.B
      |__ M.hs ......................... defines module M
      |__ N.hs ......................... defines module N
```

- The project file – ending in `.cabal` – usually matches the name of the folder
- The name of a module *matches* its place
  - `A.B.C` lives in `src/A/B/C.hs`

Haskell file `M/A.hs`

```haskell
module M.A (
  thing1, thing2  -- Declarations to export
) where

-- Imports from other modules in the project
import M.B (fn, ...)
-- Import from other packages
import Data.List (nub, filter)

thing1 :: X -> A
thing1 = ...

-- Non-exported declarations are private
localthing :: X -> [A] -> B
localthing = ...
```

Different ways to import

- `import Data.List`
  - Import every function and type from `Data.List`
  - The imported declarations are used simply by their name, without any qualifier
- `import Data.List (nub, permutations)`
  - Import only the declarations in the list
- `import Data.List hiding (nub)`
  - Import all the declarations *except* those in the list
- `import qualified Data.List as L`
  - Import every function from `Data.List`
  - The uses must be qualified by `L`, that is, we need to write `L.nub`, `L.permutations` and so on

Exporting data types

There are two ways to present a data type to the outer world

1. *Abstract*: the implementation is not exposed
   - Values can only be created and inspected using the functions provided by the module
     - Data constructors and pattern matching are not available
   - Implementation may change without rewriting the code which depends on it $\implies$ *decoupling*

   ```
   module M (..., Type, ...) where
   ```

2. *Exposed*: constructors are available to the outside world

   ```
   module M (..., Type(..), ...) where
   ```

Import cycles

Cyclic dependencies between modules are **not** allowed

- `A` imports some things from `B`
- `B` imports some things from `A`

Solution: move common parts to a separate module

Note: there is another solution based on `.hs-boot` files

- In practice, cyclic dependencies = bad design

Cabal: build and package manager

Cabal is a tool for managing Haskell projects

- Downloads and installs dependencies
- Builds libraries and executables
  - No need to call `ghc` yourself
- Supports test suites and documentation
- Well integrated with the Haskell ecosystem

Stack is a newer tool with similar goals

Initializing a project

1. Create a folder `your-project`

```
$ mkdir your-project
$ cd your-project
```

2. Initialize the project file

```
$ cabal init
Package name? [default: your-project]
...
What does the package build:
1) Library
2) Executable
Your choice? 2
...
```

Initializing a project

2. Initialize the project file (cntd.)

```
...
Source directory:
* 1) (none)
  2) src
  3) Other (specify)
Your choice? [default: (none)] 2
...
```

3. An empty project structure is created

```
your-project
 ├── your-project.cabal
 └── src
```

The project (`.cabal`) file

```
-- General information about the package
name:    your-project
version: 0.1.0.0
author:  Alejandro Serrano
...

-- How to build an executable (program)
executable your-project
  main-is:         Main.hs
  hs-source-dirs: src
  build-depends:  base
  ...
```

Dependencies

Dependencies are declared in the `build-depends` field of a Cabal stanza such as `executable`

- Just a comma-separated list of packages
- Packages names as found in Hackage
- Upper and lower bounds for version may be declared
    - A change in the major version of a package usually involves a breakage in the library interface

```
build-depends: base,
               transformers >= 0.5 && < 1.0
```

Executables

In an `executable` stanza you have a `main-is` field

- Tells which file is the *entry point* of your program

```
module Main where

import M.A
import M.B

main :: IO ()
main = -- Start running here
```

- In later lectures we shall learn how to interact with the user, read and write files, and so on
  - This is the *impure* part of your program

Building and running

0. Initialize a sandbox *only once*

```
$ cabal sandbox init
```

1. Install the dependencies

```
$ cabal update  # Obtain package information
$ cabal install --only-dependencies
```

   - Not needed if you use `cabal build`

2. Compile and link the code

```
$ cabal build
```

3. Run the executable

```
$ cabal run your-project
```