Utrecht University

Tools

Advanced Functional Programming Summer School 2019

Alejandro Serrano

Main tools in the Haskell ecosystem

- GHC: the compiler
    - GHCi: the interpreter
- Cabal and Stack: the build tools
    - Hackage and Stackage: the package repos
- HLint: the linter
- Haddock: the docs authoring tool

Modules

Code in the large

Once you start to organize larger units of code, you typically want to split this over several different files

In Haskell, each file contains a separate *module*

A simple module system

Goals of the module system

- Namespace management
- Units of separate compilation (not supported by all compilers)

Non-goals

- First-class interfaces or signatures
  - Available in Agda and ML
  - Also in GHC using the Backpack extension

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**

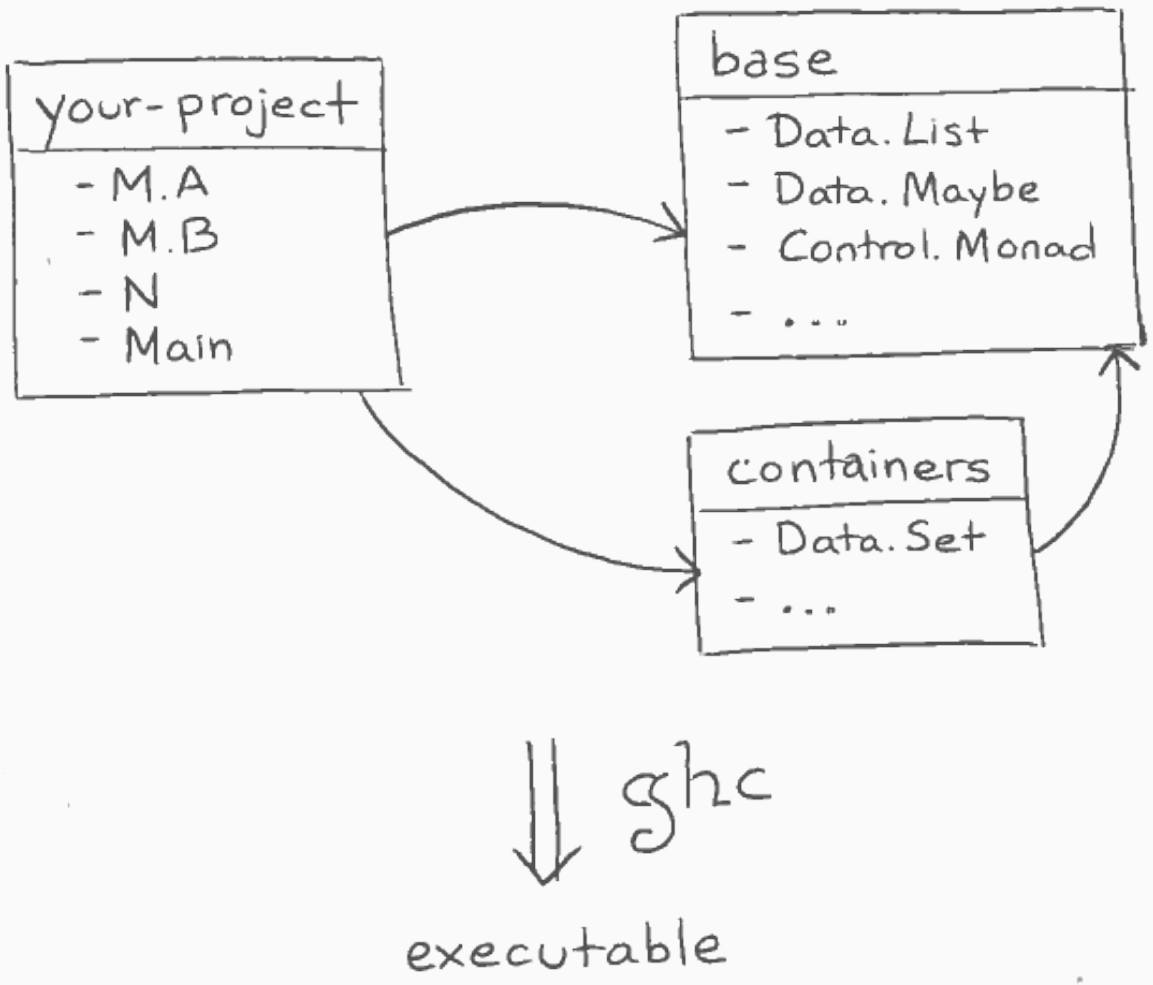
Solution: move common parts to a separate module

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

- In practice, cyclic dependencies = bad design

Packages

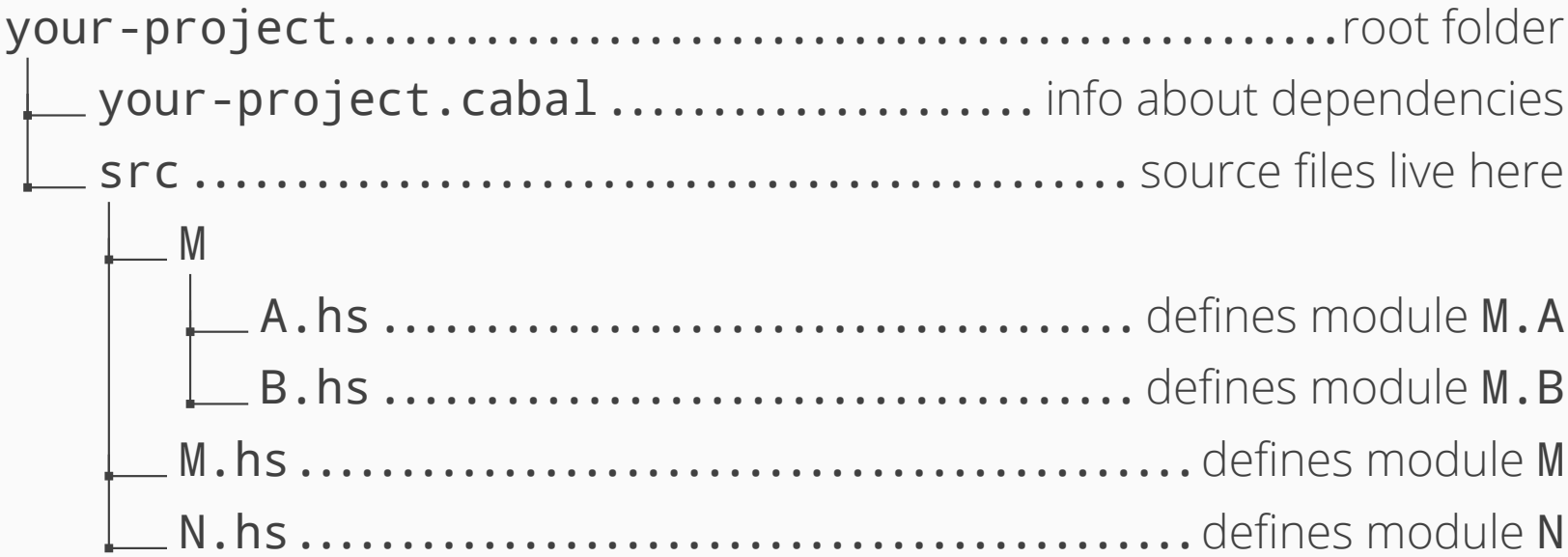
Packages


your-project
- M.A
- M.B
- N
- Main
base
- Data.List
- Data.Maybe
- Control.Monad
- ...
containers
- Data.Set
- ...
ghc
executable

Packages and modules

- **Packages** are the unit of distibution of code
  - You can *depend* on them
- Each packages provides one or more **modules**
  - Modules provide namespacing to Haskell.
  - Each module declares which functions, data types, etcetera it *exports*
  - You use elements from other modules by *importing*
- In the presence of packages, an identifier is **no** longer **uniquely determined** by module + name, but additionally needs package name + version.

Project in the filesystem



- The project (`.cabal`) file 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`

Cabal versus Cabal

In Haskell the name **Cabal** is used for two things:

1. The format in which packages are described
2. One particular build tool

The Cabal format (1) is shared by several build tools in the Haskell ecosystem, including Cabal (2) and Stack

Cabal versus Cabal

In Haskell the name **Cabal** is used for two things:

1. The format in which packages are described
2. One particular build tool

The Cabal format (1) is shared by several build tools in the Haskell ecosystem, including Cabal (2) and Stack

hpack is a version of Cabal with YAML syntax and common fields

- Compile them to Cabal using **hpack** or use Stack

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
```

Build tools: Cabal and Stack

Both tools provide similar features and UI

Cabal

- Uses Hackage as source of dependencies
- Does not manage your GHC installation
- Uses sandboxes if you use the **`new-`** commands

Stack

- Uses Stackage as source of dependencies
- You must declare a *snapshot* to be used in a **`stack.yaml`** file
  - This defines a GHC version, which is automatically downloaded
  - You can create the file using **`stack init`**
- Uses sandboxes by default

Package repositories: Hackage and Stackage

Hackage is a open, community-managed repository of Cabal projects

- Anybody can upload their packages
  - This is even automated using `cabal upload`
- *Pro*: you always access the latest version of the packages
- *Con*: you might get into trouble with dependencies

Stackage provides *snapshots* of those packages

- A subset of Hackage known to compile together
  - in a specific version of the GHC compiler
- *Pro*: **reproducibility**, every member of the team uses the same compiler and package versions
- *Con*: new major versions take time to produce
  - Bugfixes are usually backported

Compile and run

Cabal

```
$ cabal new-update  # from time to time, update Hackage info
$ cabal new-build
$ cabal new-run your-project
$ cabal new-repl    # interpreter
```

Stack

```
$ stack init  # once, to create `stack.yaml`
$ stack build
$ stack run your-project
$ stack ghci  # interpreter, also `stack repl`
```

Linting

`-Wall` is your friend

GHC includes a lot of warnings for suspicious code:

- Unused bindings or type variables,
- Incomplete pattern matching,
- Instance declaration without the minimal methods…

Enable this option in your Cabal stanzas.

```
library
   build-depends:  base, transformers, ...
   ghc-options:    -Wall
   ...
```

HLint

- A simple tool to improve your Haskell style
  - Get it using `cabal install hlint`
  - Run it with `hlint path/to/your/source`
- Scans source code, provides suggestions
  - Suggests only correct transformations
  - New suggestions can be added, existing ones can be selectively disabled
  - Refactoring can be automatically applied

```
Found:
  and (map even xs)
Why not:
  all even xs
```

HLint, larger example

```
i = (3) + 4
nm_With_Underscore = i

y = foldr (:) [] (map (+1) [3,4])

z = \x -> 5
p = \x y -> y
```

HLint in our FP labs

```
- functions:
  - { name: unsafePerformIO, within: [] }

- error: { lhs: "x == []"
         , rhs: match on the list instead
         , name: Pattern match on list head }
- error: { lhs: "length x == 0"
         , rhs: match on the list instead
         , name: Pattern match on list head }

- ignore: { name: Use <$> }
```

Documentation

Haddock

Haddock is the standard tool for documenting Haskell modules

- Think of the Javadoc, RDoc, Sphinx... of Haskell
- All Hackage documentation is produced by Haddock

Haddock uses comments starting with `|` or `^`

```
-- | Obtains the first element.
head :: [a] -> a

tail :: [a] -> [a]
-- ^ Obtains all elements but the first one.
```

Haddock, larger example

```
-- | 'filter', applied to a predicate and a list,
--   returns the list of those elements that
--   /satisfy/ the predicate.
filter :: (a -> Bool)  -- ^ Predicate over 'a'
       -> [a]          -- ^ List to be filtered
       -> [a]
```

- Single quotes as in `'filter'` indicate the name of a Haskell function, and cause automatic hyperlinking
  - Referring to qualified names is also possible, even if the identifier is not normally in scope
- Emphasis with forward slashes: `/satisfy/`
  - Many more markup is available

Profiling

Laziness

Haskell uses a **lazy** evaluation strategy

- Expressions are not evaluated *until needed*
- Duplicate expressions are *shared*

On Thursday, you will dive into this

Laziness is a double-edged sword

- With laziness, we are sure that things are evaluated only as much as needed to get the result
- But, being lazy means holding lots of thunks in memory:
  - Memory consumption can grow quickly
  - Performance is not uniformly distributed

Question: how to find out where memory is spent?

Laziness is a double-edged sword

- With laziness, we are sure that things are evaluated only as much as needed to get the result
- But, being lazy means holding lots of thunks in memory:
  - Memory consumption can grow quickly
  - Performance is not uniformly distributed

Question: how to find out where memory is spent?

Answer: use profiling

```
$ stack build --profile
$ stack run -- +RTS -hc -p
$ hp2ps executable.hp
```

Example heap profile