Lecture 8. Project management and Software Design

Functional Programming

Utrecht University

Goals

- Build a complete Haskell application
  - Deal with multiple files and modules
  - Depend on other libraries
- Design a “large” program in Haskell

Take note for your own game practical

Organizing code

Haskell supports modules to organize code

- One Module per file.

```haskell
module MyModuleName where
```

- One concept per Module.

  e.g. Data.List for functionality concerning lists

Name of the file should correspond to the Module Name

Prefix corresponds to directory path i.e.

My.Long.Prefix.MyModule in ‘My/Long/Prefix/MyModule.hs’

Importing code from other modules

- `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 code from a module

- Specify to export only a subset of the functions and data types:

```haskell
module MyModule(
    thing1, thing2  -- Declarations to export
  , Foo(..), Bar
  ) where
```

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**
- For example: ‘containers’ for data structures or ‘gloss’ for building games.

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-executable
  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

```haskell
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

Cabal and Stack : build and package managers

Cabal and stack are tools 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

Building and running with cabal

0. Update the list of available packages

```
$ cabal update
```

1. Build the project (installing dependencies when required)

```
$ cabal build
```

2. Run the executable

```
$ cabal run your-executable
```

Software design in a functional language

Separate pure and impure parts

Pure functions deal only with values

- Always the same output for the same input
- The Haskell you have learnt until now

Impure functions communicate with the outside world

- Input and output, networking, interaction, …
- Marked in Haskell with the `IO` type constructor

Most common pattern

1. Impure part which obtains the input
2. Pure part which manipulates the data
3. Impure part which communicates the result

Software Architecture

- Big topic; large body of literature
- Some design patterns from OO carry over. For example MVC.
- FP Specific Concepts: Extensible Effects, Monad Transformers, etc.

Model View Controller

- Model : All state / data of your program

```
data Model = .....
```

- View : How to display the Model

```
view :: Model -> Picture
```

- Controller: Business Logic, i.e. how to modify the Model.

```
update :: Input -> Model -> Model
```

Designing the Model

Main ideas:

- Make impossible states impossible to represent.
- One type per concept.
- Abstract using modules and typeclasses.

Make impossible states impossible to represent.

```
type Boolean = Int
-- convention: 0 means False and 1 Means True
```

vs

```
data Boolean = False | True
```

Impossible states: ADTs vs OO

- BAD:

```haskell
data Object = Ship ...
            | Player { score    :: Int
                     , numLives :: Int , ...}
            | Enemy | Wall | Empty
```

Impossible states: ADTs vs OO

- BAD:

```
data Object = Ship ...
            | Player { score    :: Int
                     , numLives :: Int , ...}
            | Enemy | Wall | Empty

getNumLives      :: Object -> Int
getNumLives Wall = ?
```

- Type signature unhelpful
- partial functions may lead to runtime errors.

Introduce one type per concept

Even if types are isomorphic, a separate one

- Improves readability and documents intention
- Prevents confusing one for the other
  - The compiler shouts if that is the case

1. Prevent “Boolean blindness”

```haskell
data PlayerStatus = Alive | Dead
data LevelStatus  = Finished | InProgress
-- instead of reusing Bool
```

Introduce one type per concept

Even if types are isomorphic, a separate one

- Improves readability and documents intention
- Prevents confusing one for the other
    - The compiler shouts if that is the case

1. Prevent “Boolean blindness”

```haskell
data PlayerStatus = Alive | Dead
data LevelStatus  = Finished | InProgress
-- instead of reusing Bool
```

```haskell
computeScore :: Bool -> Bool -> Int
```

vs

```haskell
computeScore :: PlayerStatus -> LevelStatus -> Int
```

Introduce one type per concept

2. Distinguish between points and vectors

```
data Point  = Point  Float Float
data Vector = Vector Float Float
-- Moves a point along a direction
translate :: Point -> Vector -> Point
```

Introduce one type per concept

2. Distinguish between points and vectors

```haskell
data Point  = Point  Float Float
data Vector = Vector Float Float
-- Moves a point along a direction
translate :: Point -> Vector -> Point
```

```haskell
lengthOf :: Vector -> Float
```

Type classes declare common abstractions

Haskell already comes with many common abstractions

- Equality with `Eq`, ordering with `Ord`, …

Type classes declare common abstractions

Haskell already comes with many common abstractions

- Equality with `Eq`, ordering with `Ord`, ...
- Design your own.

Type classes declare common abstractions

- Types that have a position and can be moved

```
class HasPosition a where
   getPosition :: a -> Point
   move        :: a -> Vector -> a
```

- Types that can be rendered to the screen

```
class CanRender a where
    render :: a -> Picture
```

- In general, *types that ...*

Modules for Abstraction

- Use modules to maintain invariants
- Export only subset of functions and constructors for others to use

Modules for Abstraction (example)

- “names always start with a capital (and the rest is lower case)”

Modules for Abstraction (example)

- “names always start with a capital (and the rest is lower case)”

Initial attempt:

```
type Name = String

isValidName :: String -> Bool

asValidName :: String -> Maybe Name

-- hope for the best....
```

Modules for Abstraction (example)

- “names always start with a capital (and the rest is lower case)”

```haskell
module Name( Name, mkName , render ) where
import Data.Char

newtype Name = MkName String deriving Eq

mkName         :: String -> Name
mkName []      = MkName []
mkName (c:cs)  = MkName $ toUpper c : map toLower cs

render             :: Name -> String
render (MkName s)  = s
```

Exporting Data Types

2 ways to present a data type to the outer world

1. *Exposed*: constructors available to the outside world

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

2. *Abstract*: the implementation is not exposed

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

- 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 $\Longrightarrow$ *decoupling*