Lecture 8. Project management, design and testing

Functional Programming 2018/19

Alejandro Serrano

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Goals

- Build a complete Haskell application
  - Deal with multiple files and modules
  - Depend on other libraries
- Learn good practices for Haskell programming
- Introduce randomized testing with QuickCheck

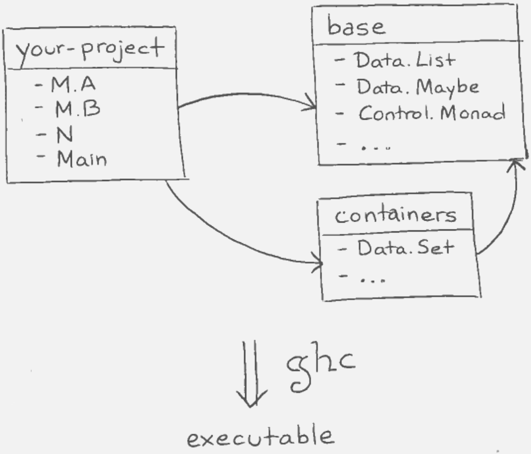
Take note for your own game practical

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Project management

The big picture



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 $\Longrightarrow$ *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-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

```
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. Update the list of available packages

   ```
   $ cabal new-update
   ```

1. Build the project (installing dependencies when required)

   ```
   $ cabal new-build
   ```

2. Run the executable

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

Some words on design

Architectural and coding practices

Two different kinds of good practices

- *Architectural* practices describe how to arrange your types and functions to create a cohesive and understandable design
  - For example, “use classes for common abstractions”
  - The “macro” level of coding
- *Coding* practices describe patterns to write simpler and cleaner source
  - For example, “prefer guards over `if-then-else`”
  - The “micro” level of coding \vspace{-0.2cm**

This is not a black-or-white classification

Software Architecture

- Big topic; large body of literature
- Some design patterns from OO carry over. For example:
- **M**odel **View** C**ontroller
  - Model : All state / data of your program
  - View : How to display the Model
  - Controller: Business Logic, i.e. how to modify the Model.

Software Architecture

- Big topic; large body of literature
- Some design patterns from OO carry over. For example:
- **M**odel **View** C**ontroller
  - Model : All state / data of your program
  - View : How to display the Model
  - Controller: Business Logic, i.e. how to modify the Model.
- FP Specific Concepts: Extensible Effects, Monad Transformers, etc.

Designing Data Types

Main idea:

- Make impossible states impossible to represent.

Designing Data Types

Main idea:

- Make impossible states impossible to represent.

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

vs

```haskell
data MyBoolean = False | True
```

Model the domain precisely using ADT's

Declare closed sets of variants of a concept as constructors of a single data type

```
type Level = Int
data Enemy = Orc Level
           | Nazgul
           | Sauron
```

ADTs + type classes have a different flavor than OOP

- Do not try to import patterns from OOP into Haskell
- In particular, Haskell has no *inheritance*

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”

```
data Status = Alive | Dead
data Level  = 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”

```
data Status = Alive | Dead
data Level  = Finished | InProgress
-- instead of reusing Bool
```

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

vs

```
computeScore :: Status -> Level -> 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 Positioned a where
   getPosition :: a -> Point
   move        :: a -> Vector -> a
```

- Types that can be rendered to the screen

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

- In general, *types that ...*

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

Coding practices from previous lectures

- Do not use magic numbers to handle special conditions
  - Use your custom ADT, or use `Maybe` and `Either`
- Prefer pattern matching with guards over conditionals
- Favour higher-order functions over explicit recursion
- Write type signatures for every declaration

Extra hints on style

- Compile your code with the maximum level of warnings
  - In the command line, use `ghc -Wall`
  - In your Cabal file, add to the stanza

```
executable your-executable
    ...
    ghc-options:  -Wall
```

- Run HLint in your source files
  - HLint suggests improvements to your code

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

Testing

Why testing?

- Gain confidence in the correctness of your program
- Show that common cases work correctly
- Show that *corner* cases work correctly

Why testing?

- Gain confidence in the correctness of your program
- Show that common cases work correctly
- Show that *corner* cases work correctly

Testing cannot prove the absence of bugs

When is a program correct?

- What is a specification?
- How to establish a relation between the specification and the implementation?
- What about bugs in the specification?

More in *Software Testing and Verification*, period 4

Property Testing using QuickCheck

QuickCheck, an *automated* testing library/tool for Haskell

- Describe properties as Haskell programs
- Automatic datatype-driven random test case generation
- Extensible, e.g. test case generators can be adapted

Case study: insertion sort

A buggy insertion sort

```
sort :: [Int] -> [Int]
sort []      =  []
sort (x:xs)  =  insert x xs

insert :: Int -> [Int] -> [Int]
insert x []                       =  [x]
insert x (y:ys)  | x <= y         =  x : ys
                 | otherwise      =  y : insert x ys
```

Let's try to debug it using QuickCheck

How to write a specification?

A good specification is

- as precise as necessary
- but no more precise than necessary

A good specification for a particular problem, such as sorting, should:

1. distinguish sorting from all other operations on lists,
2. without forcing us to use a particular sorting algorithm

A first approximation

Certainly, sorting a list should not change its length

```
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs =
  length (sort xs) == length xs
```

We can test by invoking the function:

```
> quickCheck sortPreservesLength
Failed! Falsifiable, after 4 tests:
[0,3]
```

QuickCheck gives back a *counterexample*

Correcting the bug

```
sort :: [Int] -> [Int]
sort []       =  []
sort (x:xs)   =  insert x xs

insert :: Int -> [Int] -> [Int]
insert x []                      =  [x]
insert x (y:ys)  | x <= y        =  x : ys
                 | otherwise     =  y : insert x ys
```

Which branch does not preserve the list length?

A new attempt

```
> quickCheck sortPreservesLength
OK, passed 100 tests.
```

Looks better. Are our tests good enough?

Good enough?

```
(f `preserves` p) x = p x == p (f x)

sortPreservesLength  =  sort `preserves` length
idPreservesLength    =  id `preserves` length
```

Good enough?

```
(f `preserves` p) x = p x == p (f x)

sortPreservesLength  =  sort `preserves` length
idPreservesLength    =  id `preserves` length
```

So `id` also preserves the lists length:

```
> quickCheck idPreservesLength
OK, passed 100 tests.
```

We need to refine our specification

When is a list sorted?

We can define a predicate that checks if a list is sorted:

```
isSorted :: [Int] -> Bool
isSorted []          = True
isSorted [x]         = True
isSorted (x:y:xs)    = x < y && isSorted (y:xs)
```

And use this to check that sorting a list produces a list that `isSorted`

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Testing again

```
sortEnsuresSorted :: [Int] -> Bool
sortEnsuresSorted xs = isSorted (sort xs)
```

```
> quickCheck sortEnsuresSorted
Falsifiable, after 5 tests:
[5,0,-2]
> sort [5,0,-2]
[0,-2,5]
```

We're still not quite there…

Debugging sort

What's wrong now?

```
sort :: [Int] -> [Int]
sort []       =  []
sort (x:xs)   =  insert x xs

insert :: Int -> [Int] -> [Int]
```

Debugging `sort`

What's wrong now?

```
sort :: [Int] -> [Int]
sort []      =  []
sort (x:xs)  =  insert x xs

insert :: Int -> [Int] -> [Int]
```

We are not recursively sorting the tail in `sort`!

Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]
> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]
> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

```
> isSorted [2,2,4]
False
```

Fixing the specification

The `isSorted` specification reads:

```
sorted :: [Int] -> Bool
sorted []           =  True
sorted (x:[])       =  True
sorted (x:y:ys)     =  x < y && sorted (y : ys)
```

Why does it return `False`? How can we fix it?

Are we done yet?

Is sorting specified completely by saying that

- sorting preserves the length of the input list,
- the resulting list is sorted?

Are we done yet?

Is sorting specified completely by saying that

- sorting preserves the length of the input list,
- the resulting list is sorted?

Not really…

```
evilNoSort :: [Int] -> [Int]
evilNoSort xs = replicate (length xs) 1
```

This function fulfills both specifications, but does not sort

Specifying sorting

```
permutes :: ([Int] -> [Int]) -> [Int] -> Bool
permutes f xs = f xs `elem` permutations xs

sortPermutes :: [Int] -> Bool
sortPermutes xs = sort `permutes` xs
```

This completely specifies sorting and our algorithm passes the corresponding tests

QuickCheck in general

The type of `quickCheck`

The type of is an *overloaded* type:

```
quickCheck :: Testable prop => prop -> IO ()
```

- The argument of is a property of type `prop`
- The only restriction on the type is that it is in the `Testable` *type class*.
- When executed, prints the results of the test to the screen – hence the `IO ()` result type.

Which properties are `Testable`?

So far, all our properties have been of type:

```
sortPreservesLength :: [Int] -> Bool
sortEnsuresSorted :: [Int] -> Bool
sortPermutes :: [Int] -> Bool
```

When used on such properties, QuickCheck generates random integer lists:

- If the result is `True` for 100 cases, this success is reported in a message
- If the result is `False` for a test case, the input triggering the failure is printed

Other example properties

```
appendLength :: [Int] -> [Int] -> Bool
appendLength xs ys =
  length xs + length ys == length (xs ++ ys)

plusIsCommutative :: Int -> Int -> Bool
plusIsCommutative m n = m + n == n + m

takeDrop :: Int -> [Int] -> Bool
takeDrop n xs = take n xs ++ drop n xs == xs

dropTwice :: Int -> Int -> [Int] -> Bool
dropTwice m n xs =
  drop m (drop n xs) == drop (m + n) xs
```

Other forms of properties – contd.

```
> quickCheck takeDrop
OK, passed 100 tests.

> quickCheck dropTwice
Falsifiable after 7 tests.
1
-1
[0]

> drop (-1) [0]
[0]

> drop 1 (drop (-1) [0])
[]
```

Nullary properties

A property without arguments is also possible:

```
lengthEmpty :: Bool
lengthEmpty = length [] == 0
wrong :: Bool
wrong = False
```

```
> quickCheck lengthEmpty
OK, passed 100 tests.
> quickCheck wrong
Falsifiable, after 0 tests.
```

QuickCheck subsumes unit tests

Properties

Recall the type of `quickCheck`:

```
quickCheck :: Testable prop => prop -> IO ()
```

We can now say more about when types are `Testable`:

- testable properties usually are functions (with any number of arguments) resulting in a `Bool`

What argument types are admissible?

- QuickCheck has to know how to produce random test cases of such types

Properties – continued

A `Testable` thing is something which can be turned into a `Property`:

```
class Testable prop where
  property :: prop -> Property
```

A `Bool` is testable:

```
instance Testable Bool where ...
```

If a type is testable, we can add a function argument, as long as we know how to generate and print test cases:

```
instance (Arbitrary a, Show a, Testable b) =>
         Testable (a -> b) where
```

Information about test data

We can show the actual data that is tested:

```
> quickCheck (\ xs -> collect xs (sPL xs))
OK, passed 100 tests:
6% []
1% [9,4,-6,7]
1% [9,-1,0,-22,25,32,32,0,9,...
...
```

Why is it important to have access to the test data?

Implications

The function insert preserves an ordered list:

```
implies :: Bool -> Bool -> Bool
implies x y = not x || y

insertPreservesOrdered :: Int -> [Int] -> Bool
insertPreservesOrdered x xs =
  sorted xs `implies` sorted (insert x xs)
```

Implications – contd.

```
> quickCheck insertPreservesOrdered
OK, passed 100 tests.
```

But:

```
> let iPO = insertPreservesOrdered
> quickCheck (\x xs -> collect (sorted xs)
                               (iPO x xs))
OK, passed 100 tests.
88% False
12% True
```

For **88** test cases, `insert` has not actually been relevant!

Implications – contd.

The solution is to use the QuickCheck implication operator:

```
(==>) :: Testable prop => Bool -> prop -> Property
```

```
iPO :: Int -> [Int] -> Property
iPO x xs = sorted xs ==> sorted (insert x xs)
```

Now, lists that are not sorted are discarded and do not contribute towards the goal of 100 test cases

Implications – contd.

We can now easily run into a new problem:

```
iPO :: Int -> [Int] -> Property
iPO x xs = length xs > 2 && sorted xs ==>
             sorted (insert x xs)
```

We try to ensure that lists are not too short, but:

```
> quickCheck (\x xs -> collect (sorted xs)
                               (iPO x xs))
Arguments exhausted after 20 tests (100% True).
```

The chance that a random list is sorted is extremely small

Custom generators

The solution is to generate values in a better way

- In this case, generate only sorted lists

```
class Arbitrary a where
  arbitrary :: Gen a
```

Loose ends

- Haskell can deal with infinite values, and so can QuickCheck
  - Properties must *not* inspect infinitely many values
  - Solution: only inspect finite parts
- QuickCheck can also generate functional values
  - Tequires defining an instance of another class `Coarbitrary`
  - Showing functional values is still problematic
- QuickCheck has facilities for testing properties that involve `IO`