Advanced Functional Programming

01 - Introduction

Wouter Swierstra & Lawrence Chonavel

Utrecht University

Today

1. Introduction to AFP
2. Programming style
3. Package management
4. Tools

Introduction

Who are we?

- Wouter Swierstra
- Lawrence Chonavel

Who are you?

- This is an elective, so:
  - Why did you choose this course?
  - What do you hope to learn?

Course structure

Topics

- Lambda calculus, lazy & strict
- Algebraic datatypes & generic programming
- Design patterns and common abstractions
- Type-level programming
- Programming and proving with dependent types
- ....

Languages of choice

- Haskell – first half (Wouter)
- Agda – second half (Lawrence)

(Although we may have to mix things up a bit.)

Prerequisites

- Familiarity with Haskell and GHC
  (course: “Functional Programming”)
- Familiarity with higher-order functions and folds (optional)
  (course: “Languages and Compilers”)
- Familiarity with type systems and semantics (optional)
  (course: “Concepts of program design”)

Goals

At the end of the course, you should be:

- able to use a wide range of Haskell tools and libraries,
- know how to structure and write large programs,
- proficient in the theoretical underpinnings of FP, i.e. familiar with lambda calculus and type systems,
- able to understand formal texts and research papers on FP language concepts,
- familiar with current FP research

Course homepage & Communication

`https://utrechtuniversity.github.io/infoafp/index.html`

Feel free to let us know if you find any broken links, missing slides, etc.

Course homepage & Communication

`https://utrechtuniversity.github.io/infoafp/index.html`

Feel free to let us know if you find any broken links, missing slides, etc.

Do you have any preferences for Brightspace/Teams/XXX?

It's good to have a lightweight chat to send out announcements & ask questions.

Sessions

Lectures:

- Monday, 11:00–12:45
- Wednesday, 11:00 –12:45

Participation in all lectures is expected (and you'll get much more out of the course!)

Course components

Four components:

- Exam (50%)
- Weekly assignments (20%)
- Programming project (20%)
- Active Participation (10%)

To pass, the mark on the final exam must be at least a 5 or higher.

Lectures and exam

- Lectures usually have a specific topic
- Often based on one or more research papers
- The exam will be about the topics covered in the lectures and the papers
- In the exam, you will be allowed to consult a one page (hand written) summary of the lectures and the research papers we have discussed

Assignments

- Team size: 1 person
- Weekly assignments, containing both practical and theoretical questions
- Theoretical assignments may serve as an indicator for the kind of questions asked in the exam
- Peer & self review & advisory grading of assignments
  - Reviewing other people's code is an excellent way to learn!
  - Every week you'll be given two assignments to review
- I would be cautious using Generative AI tooling.

Project

- Team size: 3 people
- Develop a realistic library or application in Haskell
- Use concepts and techniques from the course
- Again, style counts. Use version control, test your code. Try to write simple and concise code. Write documentation
- Grading: difficulty, ambition, the code, final presentation, report

Software installation

A recent version of GHC, which you can get via:

- ghcup: https://www.haskell.org/ghcup/
- stack: https://docs.haskellstack.org/en/stable/install_and_upgrade/
- or your system package manager

Please use git & GitHub or our local GitLab installation

Course structure

- Basics and fundamentals
- Patterns and libraries
- Language and types

There is some overlap between the blocks/courses

Basics and fundamentals

Everything you need to know about developing Haskell projects

- Debugging and testing
- Simple programming techniques
- (Typed) lambda calculus
- Evaluation and profiling

Knowledge you are expected to apply in the programming task

Some suggested reading

- *Fun of Programming* edited by Jeremy Gibbons and Oege de Moor
- *Parallel and concurrent programming in Haskell* by Simon Marlow
- *Purely Functional Data Structures* by Chris Okasaki
- *Real World Haskell* by Bryan O'Sullivan, Don Stewart, and John Goerzen
- *Haskell in Depth* by Vitaly Bragilevsky
- *Effective Haskell* by Rebecca Skinner
- *Types and Programming Languages* by Benjamin Pierce

Programming style

Warnings

- Turn on warnings with `-Wall` (and listen to the suggestions)
- Add `:set -Wall` to `~/.ghc/ghci.conf`

Never use TABs

- Haskell uses layout to delimit language constructs
- Haskell interprets TABs to have 8 spaces
- Editors often display them with a different (user-configurable) width
- TABs lead to layout-related errors that are difficult to debug
- Even worse: mixing TABs with spaces to indent a line

Never use TABs

- Never use TABs
- Configure your editor to expand TABs to spaces, and/or highlight TABs in source code

Never use TABs

- Never use TABs
- Configure your editor to expand TABs to spaces, and/or highlight TABs in source code
- *unless*... you travel back in time to 1980 to edit GNU Makefiles

Alignment

- Use alignment to highlight structure in the code!
- Do not use long lines
- Do not indent by more than a few spaces

```
map :: (a -> b) -> [a] -> [b]
map f []       =  []
map f (x : xs) =  f x : map f xs
```

Identifier names

- Use informative names for functions
- Use CamelCase for long names
- Use short names for function arguments
- Use similar naming schemes for arguments of similar types

Spaces and parentheses

- Generally use exactly as many parentheses as are needed
- Use extra parentheses in selected places to highlight grouping, particularly in expressions with many less known infix operators
- Function application should always be denoted with a space
- In most cases, infix operators should be surrounded by spaces

Blank lines

- Use blank lines to separate top-level functions
- Also use blank lines for long sequences of `let`-bindings or long `do`-blocks, in order to group logical units

Avoid large functions

- Try to keep individual functions small
- Introduce many functions for small tasks
- Avoid local functions if they need not be local (why?)

Type signatures

- Always give type signatures for top-level functions
- Give type signatures for more complicated local definitions, too
- Use type synonyms

```
checkTime :: Int -> Int -> Int -> Bool
```

Type signatures

- Always give type signatures for top-level functions
- Give type signatures for more complicated local definitions, too
- Use type synonyms

```
checkTime :: Int -> Int -> Int -> Bool

checkTime :: Hours -> Minutes -> Seconds -> Bool


type Hours = Int
type Minutes = Int
type Seconds = Int
```

Even better

```
checkTime :: Hours -> Minutes -> Seconds -> Bool

newtype Hours   = Hours Int
newtype Minutes = Minutes Int
newtype Seconds = Seconds Int
```

Define separate types and carefully control how they can be constructed

Hiding the constructors, for example, makes it impossible to extract the underlying integers

Comments

- Comment top-level functions
- Also comment tricky code
- Write useful comments, avoid redundant comments!
- Use Haddock

Booleans

Keep in mind that Booleans are first-class values

Negative examples:

```
f x | isSpace x == True = ...
```

```
if x then True else False
```

Use (data)types!

- Whenever possible, define your own datatypes
- Use `Maybe` or user-defined types to capture failure, rather than `error` or default values
- Use `Maybe` or user-defined types to capture optional arguments, rather than passing `undefined` or dummy values
- Don’t use integers for enumeration types
- By using meaningful names for constructors and types, or by defining type synonyms, you can make code more self-documenting

Use common library functions

- Don’t reinvent the wheel. If you can use a `Prelude` function or a function from one of the basic libraries, then do not define it yourself
- If a function is a simple instance of a higher-order function such as `map` or `foldr`, then use those functions

Pattern matching

- When defining functions via pattern matching, make sure you cover all cases
- Try to use simple cases
- Do not include unnecessary cases
- Do not include unreachable cases

Pattern matching

- When defining functions via pattern matching, make sure you cover all cases
- Try to use simple cases
- Do not include unnecessary cases
- Do not include unreachable cases

- GHC will *sometimes* warn you about missing/unnecessary/unreachable patterns

Avoid partial functions

- Always try to define functions that are total on their domain, otherwise try to refine the domain type
- Avoid using functions that are partial

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

Use `let` instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- Is there a semantic difference between the two pieces of code?
- Could/should the compiler optimize from the second to the first version internally?

Use `let` instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- Is there a semantic difference between the two pieces of code?
- Could/should the compiler optimize from the second to the first version internally?
- https://gitlab.haskell.org/ghc/ghc/-/issues/701

Let the types guide your programming

- Try to make your functions as generic as possible
- If you have to write a function of type `Foo -> Bar`, consider how you can destruct a `Foo` and how you can construct a `Bar`
- When you tackle an unknown problem, think about its type first

How to design programs

- Write down example inputs and outputs that your program should accept and produce
- Generalize these examples into data *types*
- Turn these examples into *tests*
- Try to implement the program
- If you struggle, try to break your problem into smaller pieces and repeat

Packages and 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 consists of a separate *module*

Let's start with a quick recap and reviewing the strengths and weaknesses of Haskell's module system

Goals of the Haskell module system

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

There is no language concept of interfaces or signatures in Haskell, except for the class system

Syntax

```
module M(D(),f,g) where
import Data.List(unfoldr)
import qualified Data.Map as M
import Control.Monad hiding (mapM)
```

- Hierarchical modules
- Export list
- Import list, hiding list
- Qualified, unqualified
- Renaming of modules

Module `Main`

- If the module header is omitted, the module is automatically named `Main`
- Each full Haskell program has to have a module `Main` that defines a function

```haskell
main :: IO ()
```

Hierarchical modules

Module names consist of at least one identifier starting with an uppercase letter, where each identifier is separated from the rest by a period

- This former extension to Haskell 98, has been formalized in an addendum to the Haskell 98 Report and is now widely used
- Implementations expect a module named `X.Y.Z` to be located at `X/Y/Z.hs` or `X/Y/Z.lhs`
- There are no relative module names – every module is always referred to by a unique name

Hierarchical modules

Most of Haskell 98 standard libraries have been extended and placed in the module hierarchy – moving `List` to `Data.List`

Good practice: Use the hierarchical modules where possible. In most cases, the top-level module should only refer to other modules in other directories

Importing modules

- The `import` declarations can only appear in the module header, i.e., after the `module` declaration but before any other declarations
- A module can be imported multiple times in different ways
- If a module is imported qualified, only the qualified names are brought into scope. Otherwise, the qualified and unqualified names are brought into scope
- A module can be renamed using `as`. Then, the qualified names that are brought into scope are using the new `modid`
- Name clashes are reported lazily

Prelude

The module `Prelude` is imported implicitly as if

```
import Prelude
```

has been specified, but this can be disabled by placing

```
{-# LANGUAGE NoImplicitPrelude #-}
```

at the top of the file.

An explicit `import` declaration for `Prelude` causes all names from `Prelude` to be available *only* in their qualified form:

```
import qualified Prelude
```

Module dependencies

- Modules are allowed to be mutually recursive
- This is not well supported by GHC, and therefore somewhat discouraged

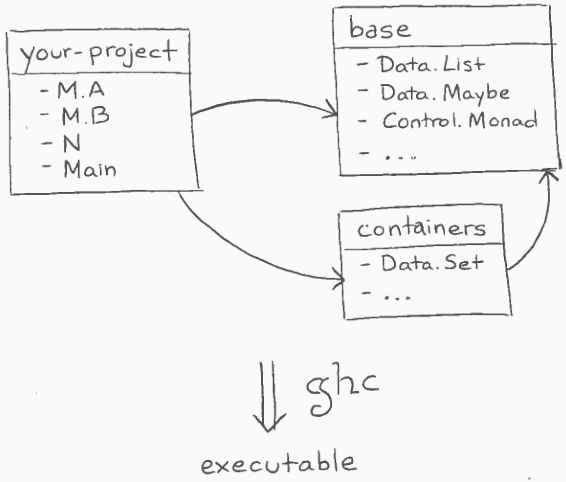
Question:

Why might it be difficult?

Good practice

- Use qualified names instead of pre- and suffixes to disambiguate
- Use renaming of modules to shorten qualified names
- Avoid `hiding`
- Recall that you can import the same module multiple times

Packages and modules



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*
  - 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

The GHC package manager

- The GHC package manager is called `ghc-pkg`
- The set of packages GHC knows about is stored in a package configuration database, `package.conf`
- Multiple package configuration databases:
  - one global per installation of GHC
  - one local per user
  - one per sandboxed project
  - more local databases for special purposes

Listing known packages

```
$ ghc-pkg list
/usr/lib/ghc-6.8.2/package.conf:
Cabal-1.2.3.0, GLUT-2.1.1.1, HDBC-1.1.3,
HUnit-1.2.0.0, OpenGL-2.2.1.1, QuickCheck-1.1.0.0,
array-0.1.0.0, base-3.0.1.0, binary-0.4.1,
cairo-0.9.12.1, containers-0.1.0.1, cpphs-1.5,
fgl-5.4.1.1, filepath-1.1.0.0, gconf-0.9.12.1,
(ghc-6.8.2), glade-0.9.12.1, glib-0.9.12.1,
...
/home/wouter/.ghc/i386-linux-6.8.2/package.conf:
binary-0.4.1, vty-3.0.0, zlib-0.4.0.2
```

- Parenthesized packages are hidden
- Exposed packages are usually available automatically

The GHC package manager

Golden rule: you only use `ghc-pkg` to solve problems with your installation

```
$ ghc-pkg check
% Empty or only warnings means the
% package database is in good shape
```

The GHC package manager

Golden rule: you only use `ghc-pkg` to solve problems with your installation

```
$ ghc-pkg check
% Empty or only warnings means the
% package database is in good shape
```

… but at that point it's probably easier to just `rm -rf`

Instead, use a *package manager*, such as `cabal` or `stack`, to manipulate the database

Cabal: a Haskell package manager

- A unified package description format
- A build system for Haskell applications and libraries, which is easy to use
  - Tracks dependencies between Haskell packages
  - Platform-independent, compiler-independent
  - Generic support for preprocessors, inter-module dependencies, etc.
- Specifically tailored to the needs of a “normal” package
- Integrated into the set of packages shipped with GHC

Cabal is under *active* development, but very *stable* (TLM: >_>)

Hackage

Online Cabal package database

- Everybody can upload their Cabal-based packages
- Automated building of packages
- Allows automatic online access to Haddock documentation

`http://hackage.haskell.org/`

Project in the filesystem

- 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`

Initializing a project (cabal)

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 (cont.)

```
...
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:    my-awesome-afp-project
version: 0.1.0.0
author:  Wouter Swierstra
...

-- 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
  - This is an honour system: the Haskell ecosystem has no agreed upon versioning scheme (SemVer vs. PvP)

```
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
```

Building and running

1. Install the dependencies

   ```
   $ cabal update
   $ 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
   ```

Stack and Stackage

Besides cabal, there is a another package manager, *Stack*

- Unlike Cabal, Stack manages your GHC installation
- Uses sandboxes and local database by default

Stack uses *Stackage* as its source of packages: https://www.stackage.org

- Curated set of packages (subset of Hackage)
- Pro: installation plan always succeeds
- Con: package versions may lag behind Hackage

Right now, both tools work ‘flawlessly’ for normal usage

There are vocal advocates of both approaches

Initialise a project (stack)

1. Create a new project, either…
   - From scratch

     ```
     $ stack new your-project
     ```
   - If you already have a Cabal file

     ```
     $ stack init
     ```
2. Initialize the project *only once*, which downloads all necessary tools including GHC

   ```
   $ stack setup
   ```
3. Compile and link the code

   ```
   $ stack build
   ```
4. Run the executable

   ```
   $ stack exec your-project
   ```

   ```
   $ stack run
   ```

Other useful tools

`-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
- Developed by Neil Mitchell
- Scans source code, provides suggestions
- Makes use of generic programming (Uniplate)
- Suggests only correct transformations
- New suggestions can be added, and some suggestions can be selectively disabled
- Easy to install (via `cabal install hlint`)

HLint, simple example

Run it with `hlint path/to/your/source`

- Source might be a file or a full folder

```
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
```

- What does HLint complain about, why?
- Would you always want such complaints?

HLint

All hints
- Error: Redundant bracket (1)
- Error: Redundant lambda (2)
- Warning: Use . (1)
- Warning: Use camelCase (1)
- Warning: Use const (1)

All files
- HLintDemo.hs (6)

Report generated by HLint v1.8.49 - a tool to suggest improvements to your Haskell code.

HLintDemo.hs:3:5: Error: Redundant bracket
Found
```
(3)
```
Why not
```
3
```

HLintDemo.hs:4:1: Warning: Use camelCase
Found
```
nm_with_underscore = ...
```
Why not
```
nmwithunderscore = ...
```

HLintDemo.hs:6:5: Warning: Use .
Found
```
foldr (:) [] (map (+ 1) [3, 4])
```
Why not
```
foldr ((:) . (+ 1)) [] [3, 4]
```

HLintDemo.hs:8:1: Error: Redundant lambda
Found
```
z = \ x -> 5
```
Why not
```
z x = 5
```

HLintDemo.hs:8:5: Warning: Use const
Found
```
\ x -> 5
```
Why not
```
const 5
```

HLintDemo.hs:9:1: Error: Redundant lambda
Found
```
p = \ x y -> y
```
Why not
```
p x y = y
```

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/`

More markup

Haddock supports several more forms of markup:

- Sectioning to structure a module
- Code blocks in documentation
  - These can be checked automatically using `doctest`
- References to whole modules
- Itemized, enumerated, and definition lists
- Hyperlinks
- Images

Next time...

We will kick off with the lectures in earnest

- Start assembling a team for your project – we have a few suggested topics on the website, but are happy to discuss others that match your interests!
- Make sure you have access to a modern Haskell installation