Intro, packages & tools

Advanced functional programming - Lecture 1

Wouter Swierstra and Alejandro Serrano

Today

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

Topics

- Lambda calculus, lazy & strict
- Types and type inference
- Data structures
- Effects in functional programming languages
- Interfacing with other languages
- Design patterns and common abstractions
- Type-level programming
- Programming and proving with dependent types

Languages of choice

- Haskell
- Agda

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 (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 such as lambda calculus and type systems,
- able to understand formal texts and research papers on FP language concepts,
- familiar with current FP research.

Homepage

- Course homepage:

`https://www.cs.uu.nl/docs/vakken/afp`

- Source on GitHub (pull requests welcome):

`https://github.com/wouter-swierstra/2018-AFP`

- Homepage from previous years is still online:

`http://foswiki.cs.uu.nl/foswiki/Afp/`

Sessions

Lectures:

- Tue, 13:15-15:00, lecture
- Thu, 9:00-10:45, lecture
- Tue, 15:15-17:00, labs

Participation in all sessions is expected.

Course components

Four components:

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

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 the slides from the lectures and the research papers we have discussed.

Assignments

- 'Weekly' assignments, both practical and theoretical.
- Team size: 1 person.
- Theoretical assignments may serve as an indicator for the kind of questions being asked in the exam.
- Use all options for help: labs, homepage, etc.
- Peer & self review & advisory grading of assignments.

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. Write elegant and concise code. Write documentation.
- Grading: difficulty, the code, amount of supervision required, final presentation, report.

Software installation

- A recent version of GHC, such as the one shipped with the Haskell Platform.
- We recommend using the Haskell Platform (libraries, Cabal, Haddock, Alex, Happy).
- 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.

Patterns and libraries

Using Haskell for real-world problems.

- (Functional) data structures
- Foreign Function Interface
- Concurrency
- Monads, Applicative Functors
- Combinator libraries
- Domain-specific languages

Knowledge that may be helpful to the programming task.

Language and types

Advanced concepts of functional programming languages.

- Type inference
- Advanced type classes
  - multiple parameters
  - functional dependencies
  - associated types
- Advanced data types
  - kinds
  - polymorphic fields
  - GADTs, existentials
  - type families
- Generic Programming
- Dependently Types Programming

Some suggested reading

- *Real World Haskell* by Bryan O'Sullivan, Don Stewart, and John Goerzen
- *Parallel and concurrent programming in Haskell* by Simon Marlow
- *Fun of Programming* editted by Jeremy Gibbons and Oege de Moor
- *Purely Functional Data Structures* by Chris Okasaki
- *Types and Programming Languages* by Benjamin Pierce
- *AFP summer school* series of lecture notes

Programming style

Never use TABs

- Haskell uses layout to delimit language constructs.
- Haskell interprets TABs to have 8 spaces.
- Editors often display them with a different 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.

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
```

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.

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?

Let the types guide your programming

- Try to make your functions as generic as possible (why?).
- 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.

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 contains 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

```haskell
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

```
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 `X.Y.Z` to be named `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.

- An explicit `import` declaration for `Prelude` overrides that behaviour

  ```
  qualified Prelude
  ```

  causes all names from `Prelude` to be available only in their qualified form.

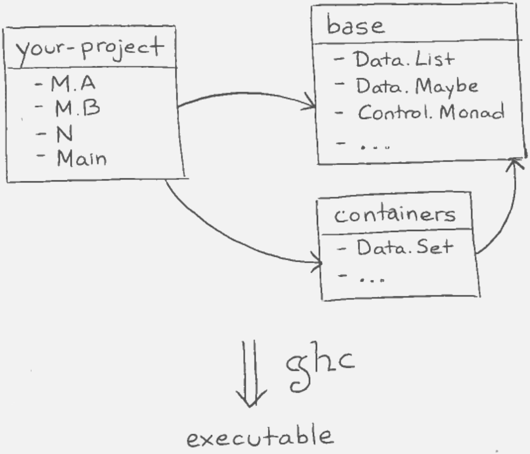
Module dependencies

- Modules are allowed to be mutually recursive.
- This is not supported well 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
% package database in good shape
```

You use Cabal (or Stack) to manipulate the database.

Universiteit Utrecht

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

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*.

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

```
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`.

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
```

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
```

Stack and Stackage

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

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

Stack uses *Stackage* instead of Hackage.

- Curated set of packages.
- Pro: installation plan always succeeds.
- Con: package versions lag behind Hackage.

Right now, both tools work flawlessly for normal usage.

- There are string advocates of both tools.

Using Stack

1. Create a new project.

   ```
   $ stack new your-project && cd your-project
   ```

   - If you already have a Cabal file

   ```
   $ cd your-project && stack init
   ```

2. Initialize the project *only once*.

   - Downloads all needed tools, including GHC.

   ```
   $ stack setup
   ```

3. Compile and link the code.

   ```
   $ stack build
   ```

4. Run the executable.

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

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

```haskell
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 ^.

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

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

Haddock, larger example

```haskell
-- | '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.
- References to whole modules.
- Itemized, enumerated, and definition lists.
- Hyperlinks.