Intro, packages & tools

Advanced functional programming - Lecture 1

Wouter Swierstra and Trevor McDonell
1. Intro to AFP
2. Programming style
3. Package management
4. Tools
Course structure
Topics

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

- Haskell – first half (Trevor McDonell)
- Agda – second half (Wouter Swierstra)
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 such as lambda calculus and type systems,
- able to understand formal texts and research papers on FP language concepts,
- familiar with current FP research.
• Course homepage:

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

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

Lectures:

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

Participation in all lectures is expected.
Four components:

- Exam (50%)
- ‘Weekly’ assignments (20%)
- Programming project (20%)
- Active Participation (10%)
• 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 summary of 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.
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
- Dependent 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* edited 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.
• Use alignment to highlight structure in the code!
• Do not use long lines.
• Do not indent by more than a few spaces.

\[
\begin{align*}
\text{map} & : (a \to b) \to [a] \to [b] \\
\text{map } f \; [] & = [] \\
\text{map } f \; (x : xs) & = f \; x : \text{map } f \; xs
\end{align*}
\]
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.

```haskell
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.

\[
\text{checkTime} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Bool}
\]

\[
\text{checkTime} :: \text{Hours} \to \text{Minutes} \to \text{Seconds} \to \text{Bool}
\]

\[
\text{type Hours} = \text{Int}
\]

\[
\text{type Minutes} = \text{Int}
\]

\[
\text{type Seconds} = \text{Int}
\]
• Comment top-level functions.
• Also comment tricky code.
• Write useful comments, avoid redundant comments!
• Use Haddock.
Keep in mind that Booleans are first-class values.

Negative examples:

```haskell
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.
• 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

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

Use pattern matching!
Use `let` instead of repeating complicated code

Write

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

rather than

```plaintext
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 \( \text{Foo} \rightarrow \text{Bar} \), consider how you can destruct a \text{Foo} and how you can construct a \text{Bar}.
- When you tackle an unknown problem, think about its type first.
Packages and modules
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

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 \(X \cdot Y \cdot Z\) 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.
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.
• 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

\texttt{import Prelude}

has been specified.

• An explicit import declaration for Prelude overrides that behaviour

\texttt{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

```
your-project
- M.A
- M.B
- N
- Main

base
- Data.List
- Data.Maybe
- Control.Monad
- ...

containers
- Data.Set
- ...

\[ \text{ghc} \]

\[ \text{executable} \]