Intro, packages, and quality assurance

Advanced functional programming - Lecture 1

Wouter Swierstra

Today

1. Intro to AFP
2. Package management
3. Testing and quality assurance

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/2017-AFP`

- Last year's homepage 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 or 2, preferably 2.
- 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.

Programming Project

- Team size: 3 to 4.
- Task will be put online shortly, but is loosely based on a problem from the ICFP Programming Contest from a few years back.
- 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.
- *Task:* Get a working Haskell environment by next week; try to install wxHaskell (currently best from `https://github.com/wxHaskell/wxHaskell`)
- (Alternative task: I'm willing to accept pull requests refactoring the visualization, replacing wxHaskell with suitable browser-based technology)

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 on various topics

I'll try to collect links online – but previous year's website has more info for now.

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

```
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.

Haskell package management

- Packages are collections of modules that are distributed together.
- Packages are *not* part of the Haskell standard.
- Packages are versioned and can depend on other packages.
- Packages contain modules. Some of those modules may be hidden.

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, usually called `package.conf`.
- There may be multiple package configuration databases:
  - one global per installation of GHC
  - one local per user
  - 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.

Package descriptions

```
$ ghc-pkg describe containers
name: containers
version: 0.2.0.0
license: BSD3
copyright:
maintainer: libraries@haskell.org
stability:
homepage:
package-url:
description: This package contains efficient
     general-purpose implementations of ...
...
```

More about GHC packages

- The GHC package manager can also be used to register, unregister and update packages, but this is usually done via Cabal (next in this lecture).
- The presence of packages can cause several modules of the same name to be involved in the compilation of a single program (different packages, different versions of a package).
- In the presence of packages, an is no longer uniquely determined by its name and the module it is defined in, but additionally needs the package name and version.

Cabal

- Cabal is itself packaged using Cabal.
- Cabal is integrated into the set of packages shipped with GHC, so if you have GHC, you have Cabal as well.

Homepage `http://haskell.org/cabal/`

A Cabal package description

```
Name: QuickCheck
Version: 2.0
Cabal-Version: >= 1.2
Build-type: Simple
License: BSD4
License-file: LICENSE
Copyright: Koen Claessen <koen@cs.chalmers.se>
Author: Koen Claessen <koen@cs.chalmers.se>
Maintainer: Koen Claessen <koen@cs.chalmers.se>
Homepage: http://www.haskell.org/QuickCheck/
Description:
  QuickCheck is a library for random testing
  of program properties.
```

```
flag splitBase
Description:
  Choose the new smaller, split-up
  base package.

library
Build-depends: mtl
if flag(splitBase)
Build-depends: base >= 3, random
else
Build-depends: base < 3
Exposed-Modules:
Test.QuickCheck, Test.QuickCheck.Arbitrary,
Test.QuickCheck.Gen, Test.QuickCheck.Monadic,
...
```

A Setup file

```
import Distribution.Simple

main = defaultMain
```

In most cases, this together with a Cabal file is sufficient (and often not even needed with `cabal install`). If you need to do extra stuff (for instance, install some additional files that have nothing to do with Haskell), there are variants of `defaultMain` that offer hooks.

Using Cabal

- `runghc Setup configure` resolves dependencies. You can specify via `--prefix` where you want the package installed, and `--user` is the user-specific package configuration database should be used.
- `runghc Setup build` builds the package.
- `runghc Setup install` installs the package and registers it as a GHC package if required.

HackageDB

- Online Cabal package database.
- Everybody can upload their Cabal-based Haskell packages.
- Automated building of packages.
- Allows automatic online access to Haddock documentation.

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

cabal-install

- A frontend to Cabal.
- Resolves dependencies of packages automatically, then downloads and installs all of them.
- Once cabal-install is present, installing a new library is usually as easy as:

```
cabal update
cabal install <packagename>
```

- You can also run `cabal install` within a directory containing a `.cabal` file (and the required source code).

Stack and stackage

Besides cabal, there is a newer Haskell package manager `stack`.

Unlike cabal, stack will manage your GHC installation in addition to the libraries you have installed.

It doesn't use Hackage, but a curated list of packages (Stackage).

Both have their advantages and disadvantages: there are vocal advocates of both tools.

Program Correctness

Testing and correctness

- When is a program correct?

Testing and correctness

- 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?

Equational reasoning

- "Equals can be substituted for equals"
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

Equational reasoning

- "Equals can be substituted for equals"
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

```
ones    = 1:    ones
ones'   = 1:1:  ones'
```

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

```
let   val   x     =   ref 0
      fun   f n   =   (x := !x + n; !x)
in    f 1 + f 2
```

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

```
let  val  x    =  ref 0
     fun  f n  =  (x := !x + n; !x)
in   f 1 + f 2
```

But we cannot replace the last line with `1 + f 2`, even though `f 1 = 1`.

Referential transparency in Haskell

- Haskell is referentially transparent – all side-effects are tracked by the IO monad.

```
do
  x  <-  newIORef 0
  let f n = do modifyIORef x (+n); readIORef x
  r  <-  f 1
  s  <-  f 2
  return (r + s)
```

Note that the type of `f` is `Int -> IO Int` – we cannot safely make the substitution we proposed previously.

Referential transparency

Because we can safely replace equals for equals, we can *reason* about our programs – this is something you already saw in the course on functional programming.

For example to prove some statement `P xs` holds for all lists `xs`, we need to show:

- `P []` – the base case;
- for all `x` and `xs`, `P xs` implies `P (x:xs)`.

Example: insertion sort

```
isort :: Ord a => [a] -> [a]
isort []       =  []
isort (x:xs)   =  insert x (isort xs)


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

Properties of insertion sort

We can now try to prove that for all lists `xs`,

`length (sort xs) == length xs.`

- The base case is trivial.
- The inductive case requires a lemma relating `insert` and `length` – suggestions?

Equational reasoning

- Equational reasoning can be an elegant way to prove properties of a program.
- Equational reasoning can be used to establish a relation between an “obviously correct” Haskell program (a specification) and an efficient Haskell program.
- Equational reasoning can become quite long…
- Careful with special cases (laziness):
  - undefined values;
  - partial functions;
  - infinite values.

Later we’ll see how to formalize such proofs using Agda.

QuickCheck

QuickCheck, an automated testing library/tool for Haskell

Features:

- Describe properties as Haskell programs using an embedded domain-specific language (EDSL).
- Automatic datatype-driven random test case generation.
- Extensible, e.g. test case generators can be adapted.

History

- Developed in 2000 by Koen Claessen and John Hughes.
- Copied to other programming languages: Common Lisp, Scheme, Erlang, Python, Ruby, SML, Clean, Java, Scala, F#
- Erlang version is sold by a company, QuviQ, founded by the authors of QuickCheck.

Case study: insertion sort

Consider the following (buggy) implementation of 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,
- no more precise than necessary.

A good specification for a particular problem, such as sorting, should distinguish sorting from all other operations on lists, 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]
```

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. But have we tested enough?

Properties are first-class objects

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

sortPreservesLength  =  sort `preserves` length

idPreservesLength    =  id `preserves` length
```

Properties are first-class objects

```
(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 spec.

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`.

Testing again

```
> 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`.

Universiteit Utrecht
Faculty of Science
Information and Computing Sciences

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 spec

The `isSorted` spec 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?

No, not quite.

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

This function fulfills both specifications, but still does not sort.

We need to make the relation between the input and output lists precise: both should contain the same elements – or one should be a permutation of the other.

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.

How to use QuickCheck

To use QuickCheck in your program:

```haskell
import Test.QuickCheck
```

Define properties.

Then call to test the properties.

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


Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

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 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 and verifies that the result is `True`.

If the result is for 100 cases, this success is reported in a message.

If the result is `False` for a test case, the input triggering the result 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 vs unit tests

No random test cases are involved for nullary properties.

QuickCheck subsumes unit tests.

Properties

Recall the type of `quickCheck`:

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

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

- 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

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

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

We can test any Boolean value or any testable function for which we can generate arbitrary input.

More information about test data

```
collect :: (Testable prop, Show a) =>
  a -> prop -> Property
```

The function gathers statistics about test cases. This information is displayed when a test passes:

```
> let sPL = sortPreservesLength
> quickCheck (\ xs -> collect (null xs) (sPL xs))
OK, passed 100 tests.
96% False
4% True.
```

Note that the result implies that not all test cases must be distinct.

More information about test data – contd.

```
> quickCheck (\ xs -> collect (length xs `div` 10)
                              (sPL xs))
+++ OK, passed 100 tests.
 26% 0.
 21% 1.
 15% 2.
 10% 5.
 10% 3.
 ...
```

Most lists are small in size: QuickCheck generates small test cases first, and increases the test case size for later tests.

More information about test data (contd.)

In the extreme case, 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 for the result.