Universiteit Utrecht

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

Advanced Functional Programming

2010-2011, periode 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

November 13, 2012

1. Introduction

1.1 What is AFP?

Topics

- Lambda calculus
- Evaluation strategies, eager and lazy evaluation
- Types and type inference
- Data structures
- Effects in functional programming languages
- Interfacing with other languages
- Design patterns and common abstractions
- Modularity and reuse
- Domain-specific languages
- Type-level programming

Language of choice: Haskell

Prerequisites

- Familiarity with Haskell and GHC
  (course: "Functional Programming")
- Helpful: familiarity with higher-order functions and folds
  (course: "Grammars and Parsing / Languages and Compilers")
- Helpful: familiarity with type systems
  (course: "Implementation of Programming Languages")

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.

This lecture

1.2 Administration

Wiki

`http://www.cs.uu.nl/wiki/Afp`

- Check the Wiki page regularly. Unfortunately system management is moving our servers today, so it will be temporarily inaccessible.
- If you do not have a wiki account, create one; we will use it for the non-public parts of the site.
- Mail me your wiki name.

Sessions

Lectures:

- Tue, 9-10.45, BBL-077, lecture
- Tue, 11-12.45, BBL-103, joint discussion about papers read, working on project
- Thu, 15.15-17, BBL-077, lecture

The last two hours on Tuesday are classified as "werkcollege", but we will use it for various purposes, in particular to explain more practical aspects of the course material or to discuss assignments and project progress and discussing papers read.

Participation in all sessions is required.

Course components

Four components:

- Lectures, Exam (50%)
- Weekly assignments (20%)
- Programming task (20%)
- Active Participation (10%)

Lectures and exam

- Lectures usually have a specific topic.
- Often based on one or more research papers.
- Papers should be read prior to the lecture.
- 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 (not all weeks), both practical and theoretical.
- Team size: 1 or 2.
- Theoretical assignments may serve as an indicator for the kind of questions being asked in the exam.
- Use all options for help: Tuesday morning "werkcolleges", Wiki.
- Assignments come available on Thursday and have to be handed in before the Friday one week later.

`http://www.cs.uu.nl/wiki/Afp/Assignments`

Programming Task, paper and Final presentation

- Team size: 2 to 3.
- Task are found at `http://www.cs.uu.nl/wiki/bin/view/Afp/ProgrammingTask`.
- Again, style counts. Use version control, use testing. Write elegant and concise code. Write documentation.
- Grading: difficulty, the code, amount of supervision required, final presentation, paper.

Software

- GHC (current platform version)!
- Use the Haskell Platform (libraries, Cabal, Haddock, Alex, Happy)!
- recommended: git, svn, darcs
- I prefer to use Mac OS, but I will try to help with other platforms as far as I can.

- **Task:** Get a work environment until the end of the week; try to install wxHaskell.

1.3 Course overview

Overall structure

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

There is some overlap between the blocks.

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
  - existentials
  - polymorphic fields
  - GADTs
- Dependently typed programming (Agda)
- Curry-Howard (propositions as types)

Literature

- The Haskell Report
- Real World Haskell
- Fun of Programming
- Purely Functional Data Structures
- Types and Programming Languages
- AFP summer school lecture notes
- papers, tutorials, blogs, TMR, movies

See also

`http://www.cs.uu.nl/wiki/Afp/CourseLiterature`

1.4 Extra activities

Dutch HUG

Dutch Haskell Users' Group

Talking about Haskell in an informal atmosphere. Drinks. Sometimes with presentations

Next lecture

- Thursday 15-17: Testing Haskell with QuickCheck.
- Read: Koen Claessen, John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.
- `http://www.cs.uu.nl/wiki/Afp/CourseLiterature`

1.5 Modules

Question

Why modules?

Goals of the Haskell module system

- Units of separate compilation (but: JHC, EHC, . . .)
- Namespace management

Goals of the Haskell module system

- Units of separate compilation (but: JHC, EHC, . . .)
- Namespace management

There is no language concept of interfaces/signatures in Haskell.

Syntax

```
module M (D (..), f, g) where

import Data.List (unfoldr)
import qualified Data.Map as M
import Control.Monad hiding (mapM)
```

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.

  > **module** Main **where** ...

- 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 (contd.)

Ratio	Data.Ratio
Complex	Data.Complex
Ix	Data.Ix
Array	Data.Array
List	Data.List
Maybe	Data.Maybe
Char	Data.Char
Monad	Control.Monad
IO	System.IO
Directory	System.Directory
System	System.Exit, System.Environment, System.Cmd
Time	System.Time
Locale	System.Locale
CPUTime	System.CPUTime
Random	System.Random

Hierarchical modules (contd.)

Most of Haskell 98 standard libraries have been extended and placed in the module hierarchy.

Good practice

Use the hierarchical modules where possible. In most cases, the only top-level module your programs should refer to is Prelude.

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 –

  > **import** 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.

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.

1.6 Haskell packages

Packages

- 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, HTTP-3001.0.0,
HUnit-1.2.0.0, OpenGL-2.2.1.1, QuickCheck-1.1.0.0, X11-1.4.1,
array-0.1.0.0, base-3.0.1.0, binary-0.4.1, bytestring-0.9.0.1,
cairo-0.9.12.1, containers-0.1.0.1, cpphs-1.5, directory-1.0.0.0,
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, gtk-0.9.12.1, gtkglext-0.9.12.1,
haddock-2.0.0.0, haskell-src-1.0.1.1, haskell98-1.0.1.0,
hpc-0.5.0.0, html-1.0.1.1, hxt-7.3, mozembed-0.9.12.1, mtl-1.1.0.0,
network-2.1.0.0, old-locale-1.0.0.0, old-time-1.0.0.0,
packedstring-0.1.0.0, parallel-1.0.0.0, parsec-2.1.0.0,
pretty-1.0.0.0, process-1.0.0.0, random-1.0.0.0, readline-1.0.1.0,
rts-1.0, soegtk-0.9.12.1, sourceview-0.9.12.1, svgcairo-0.9.12.1,
template-haskell-2.2.0.0, time-1.1.2.0, unix-2.3.0.0, uulib-0.9.2,
xmonad-0.5, zlib-0.4.0.1
/home/andres/.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.
- Packages can explicitly be requested by passing a -package flag to the compiler.

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 various basic immutable container types.  The declared cost of
     each operation is either worst-case or amortized, but remains
     valid even if structures are shared.
category:
author:
...
```

Package descriptions (contd.)

```
exposed-modules: Data.Graph Data.Sequence Data.Tree Data.IntMap
                 Data.IntSet Data.Map Data.Set
hidden-modules:
import-dirs: /Library/Frameworks/GHC.framework/Versions/7.0.3-i386/usr/lib/ghc-7.0.3/containers-
library-dirs: /Library/Frameworks/GHC.framework/Versions/7.0.3-i386/usr/lib/ghc-7.0.3/containers
hs-libraries: HScontainers-0.4.0.0
extra-libraries:
extra-ghci-libraries:
include-dirs:
includes:
depends: array-0.3.0.2-ecfce597e0f16c4cd1df0e1d22fd66d4
         base-4.3.1.0-167743fc0dd86f7f2a24843a933b9dce
hugs-options:
cc-options:
ld-options:
framework-dirs:
frameworks:
haddock-interfaces: /Library/Frameworks/GHC.framework/Versions/7.0.3-i386/usr/share/doc/ghc/html
!4.0.0/containers.haddock
haddock-html: /Library/Frameworks/GHC.framework/Versions/7.0.3-i386/usr/share/doc/ghc/html/libra
```

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 entity is no longer uniquely determined by its name and the module it is defined in, but additionally needs the package name and version.

1.7 Cabal

Goals of Cabal

- A build system for Haskell applications and libraries, which is easy to use.
- Specifically tailored to the needs of a "normal" Haskell package.
- Tracks dependencies between Haskell packages.
- A unified package description format that can be used by a database.
- Platform-independent, Compiler-independent.
- Generic support for preprocessors, inter-module dependencies, etc. (`make` replacement).

Cabal is under *active* development (constraint solver).

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.Function,
Test.QuickCheck.Gen, Test.QuickCheck.Monadic, Test.QuickCheck.Property,
Test.QuickCheck.Test
Other-Modules:
Test.QuickCheck.Exception, Test.QuickCheck.Text
```

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.