Advanced Programming 2017

Introduction to (the course and) Haskell

Andrzej Filinski
andrzej@di.ku.dk

(Administrative info adapted from slides by Ken Friis Larsen)

Department of Computer Science
University of Copenhagen

September 5, 2017

Today's Menu

- General course information
- Course content and motivation
- Introduction to Haskell

What This Course Is About

The purpose of this course is to provide practical experience with sophisticated programming techniques and paradigms from a language-based perspective. The focus is on high-level programming and systematic construction of well-behaved programs.

– `http://kurser.ku.dk/course/ndaa09013u/2017-2018`

The Languages We'll Use

- Haskell: lazy, pure, statically typed, functional programming
  - `http://haskell.org/`
- Erlang: eager, fail-safe, distributed programming
  - `http://erlang.org/`
- Prolog: declarative logic programming
  - SWI-Prolog (`http://www.swi-prolog.org/`)
  - or GNU Prolog (`http://www.gprolog.org/`)

The Skills You Will Practice

- Use program structuring principles and design patterns, such as monads, to structure the code so that there is a clear separation of concerns.
- Use a parser combinator library to write a parser for a medium-sized language with a given grammar, including changing the grammar so that it is on an appropriate form.
- Use parallel algorithm skeletons such as map-reduce to write data exploring programs.
- Implement simple concurrent/distributed servers using message passing, with appropriate use of synchronous and asynchronous message passing.
- Use program structuring principles and design patterns for making reliable distributed systems in the presence of software errors.
- Write idiomatic programs in a logic programming language.

What We Hope You'll Go Away With

- You can write correct, efficient, and maintainable programs with a clear separation of concerns
- You can quickly acquaint yourself with advanced programming techniques, from academic literature and/or technical documentation
- You can use those techniques to solve challenging, realistic problems
- You can give an assessment of your own code, based on a systematic evaluation of correctness, selection of algorithms and data structures, error scenarios, and elegance.

The Course Team

- **Lecturers**
  -  Andrzej

    Haskell, Prolog
  -  Ken (**course organizer**)

    Erlang, QuickCheck
- **Teaching assistants**
  - Abraham
  - Mikkel
  - Niels
  - Simon
  - Troels

Online Information

- The course home page can be found in Absalon
- The home page for the course contains
  - a detailed lecture plan
  - (links to) reading materials
  - assignments and exercises
  - a forum for questions and discussion
  - latest news and other important course information
- Slides *may* be uploaded some time *after* the lecture
- **Keep an eye** on the course home page throughout the block
- Lectures: Tuesday 10:15–12:00 or 9:15-11:00, and Thursday 10:15-12:00, always at Aud. “Lille UP1”, DIKU.
- Labs: Thursday afternoons and some Tuesdays after the lecture. First time this Thursday.

How Should You Spend Your Time

- A typical week:

Attend lectures:	4 hours
Reading (“preload” and “by-need”)	6 hours
Programming & Documentation :	10–12 hours

  - of which, $\sim$ 3 hours in lab sessions.
- We will try to provide open-ended exercises as inspiration for how to work with the topics.
  - The exercises are excellent preparation for the mandatory assignments
  - False economy to start directly on the assignment problems
- If you spend significantly less or more time on the course, please let us know.

How Should You Spend Your Time

- A typical week:

Attend lectures:	4 hours
Reading (“preload” and “by-need”)	6 hours
Programming & Documentation :	**10–12 hours**

  - of which, $\sim$ 3 hours in lab sessions.

- We will try to provide open-ended exercises as inspiration for how to work with the topics.
  - The exercises are excellent preparation for the mandatory assignments
  - False economy to start directly on the assignment problems

- If you spend significantly less or more time on the course, please let us know.

Getting to the Exam

- Pass ≥ 4 out of 6 mandatory assignments:
  - Assignment 0: Curves (Haskell)
  - **Assignment 1: TBD interpreter (Haskell)**
  - Assignment 2: TBD parser (Haskell)
  - **Assignment 3: TBD (Prolog)**
  - Assignment 4: TBD (Erlang)
  - **Assignment 5: TBD (Erlang)**
- We recommend that you seriously attempt them all
  - *But especially assignments 1, 3, and 5*
- Normally published Tuesday, due Wednesday of following week (at 20:00).
- Pair programming strongly encouraged (max 2 people)
  - Do take turns as “driver” vs. “navigator”!

Exam

- One week take-home exam
- Typically $\sim$4 questions
- Each question is like an assignment
- Estimated $\sim$25 hours of work in total
- **Strictly individual**

Let's Begin!

The purpose of this course is to provide practical experience with sophisticated programming techniques and paradigms from a language-based perspective. The focus is on high-level programming and systematic construction of well-behaved programs.

– `http://kurser.ku.dk/course/ndaa09013u/2017-2018`

Why would you learn a new programming language?

A Language-Based Perspective

Different languages offer:

- Different levels of abstraction
  - Contrast assembly, C, and Python
- Different assurances
  - Static (compile-time) analyses
  - Dynamic (run-time) checking
- Different programming models
  - Functional vs imperative vs declarative programming
  - Lazy evaluation vs eager evaluation vs proof searching
  - Message passing vs shared memory
- Different primitives, libraries, and frameworks

Why Haskell?

- Modern functional programming (FP) language
  - Introduced ∼1990, but has been evolving continuously since
  - Vibrant user and developer community
  - Good cross-platform support
  - Directly used in growing number of application domains
- Useful medium to present general programing abstractions and principles
  - Easier to explain many ideas in a functional setting
  - Many FP concepts and techniques steadily diffusing into "mainstream" languages
- Goal of course is *not* to make you Haskell experts
  - "Program *into* a language, not *in* it." –D.Gries
  - Do exploit constructs and idioms of host language, but don't let it constrain your high-level thinking.

Haskell fundamentals

- *Value-oriented* (*applicative*) paradigm
  - Will see others later in the course
- Main computation model: *evaluation* of *expressions*
  - Not sequential *execution* of *statements*
    - Though that can be accomodated as a special case
  - *Purely* functional
    - No hidden/silent side effects at all
- Strongly, statically typed
  - Surprisingly many problems caught at compile time
- If you already know another typed functional language (SML, OCaml, F#), today will be mainly a refresher
  - Next time: Haskell-specific concepts and constructs
- If not, don't panic!
  - Basic concepts are really quite simple

Types

- Haskell (like Java, unlike C) is *strongly* typed.
  - Types enforce both language-provided and programmer-defined abstractions.
  - Cannot *construct* “ill-formed” values of a type
    - No crashes/segfaults (from casting `int` to pointer)
    - No violation of data-structure invariants
  - Cannot even *observe* interior structure of data values, except through API.
    - No inspecting of heap/stack layout (casting pointer to `int`)
    - No hidden dependencies on particular implementation
- Haskell (like C, unlike Python) is *statically* typed
  - Only well-typed programs may even be run
  - Type system is very flexible, normally unobtrusive
    - A type error almost always reflects logical error in program, not weakness/deficiency of type checker
    - Once program type-checks, usually close to working

Types and values

- Types classify values.
  - Notation: *value* `::` *type*

- Usual complement of *basic* types, including:
  - Integers: `3 :: Int`, `43252003274489856000 :: Integer`
  - Floating point: `2.718281828 :: Double` (`Float` rarely used)
  - Booleans: `True :: Bool`
  - Characters: `'x' :: Char`
  - Strings: `"new\nline" :: String`
    - Actually, `type String = [Char]` (list of characters)

- *Compound* types, including:
  - Tuples: `(2, 3.4, False) :: (Int, Double, Bool)`
  - Lists (homogeneous): `[2,3,5,7] :: [Int]`
  - May be nested:
    `([(1, 2.0), (3, 4.0)], True) :: ([(Int, Double)], Bool)`

Expression evaluation

- *Expressions* also have types
  - The expression `2+2 :: Int` evaluates to the value `4 :: Int`
- *Type safety*: expression of a given type always evaluates to value of that type.
  - Or possibly a runtime error, or nontermination
  - Far from trivial to show, given advanced features in Haskell's type system.
- Haskell implementations generally provide an interactive mode
  - Traditionally called a read-eval-print loop (REPL)
  - In Glasgow Haskell Compiler (GHC), invoked as `ghci -W`
    - The `-W` enables useful warnings; omit at your peril!
    - Ignore `Prelude>` in prompt for now.
  - When using Stack, try `alias ghci='stack exec ghci -- -W'` (or equivalent in your favorite shell).

Using the REPL environment

- Evaluate expressions:

```
> "foo" ++ "bar"
"foobar"
> head "foo"
'f'
> head ""
*** Exception: Prelude.head: empty list
```

- Can also typecheck expressions without evaluating:

```
> :type head ""
head "" :: Char
```

- Useful for debugging and experimentation, but not meant for writing large programs.
  - Can load a set of definitions from a file, experiment interactively.

Expression forms

- Expressions are built up from
  - *Literals* (atomic values): `42`
  - Constructors of compound values: `[3,4]`
  - Constant and variable names (global or local): `pi`, `let x = 3 in x*x`
  - Function calls, prefix and infix: `sqrt 4.0`, `5 + 6`
  - Conditionals: `if x > y then x else y`
    - Later generalized to `case`-expressions
- Large number of builtin constants and functions
  - Most common ones are always available (standard prelude)
  - Others must be `import`ed from relevant module first
  - Hoogle (`haskell.org/hoogle/`) is your friend!
- Can add own definitions:
  - At top level (usually only one-liners)

    ```
    > let courseName = "Advanced Programming"
    > let wordCount s = length (words s)
    ```
  - In separate file (next slide)

Definitions in separate file

- Slightly different syntax (no initial `let`).
- Should always include explicit types for all definitions
  - Not formally required, but makes it *much* easier to understand your code.
- Example: in file `mydefs.hs`

```haskell
courseName :: String
courseName = "Advanced Programming"

wordCount :: String -> Int
wordCount s = length (words s)
```

- Can load from top-level loop

```
> :load mydefs.hs
> wordCount courseName
2
```

- Later: code in files should be organized into *modules*.

More about Haskell definitions

- Haskell syntax is indentation-sensitive!
  - Always use spaces, not tabs
- Multiple definitions in a group must start at same level:

```haskell
let f x = ...
    g y = ...
in ...
```

- *Increase* indentation to continue previous line

```haskell
double x =
  x + x
```

- All definitions (whether local or global) may be mutually recursive

```haskell
isEven, isOdd :: Int -> Bool
isEven x = if x == 0 then True else isOdd (x - 1)
isOdd x = if x == 0 then False else isEven (x - 1)
```

More about Haskell functions

- Functions are values, too, but cannot be printed.

```
> :t wordCount
wordCount :: String -> Int
> wordCount
<interactive>:6:1: No instance for (Show (String -> Int)) ...
```

- Functions may have multiple arguments

```haskell
addt :: (Int, Int) -> Int  -- tupled style
addt (x, y) = x + y

addc :: Int -> Int -> Int  -- curried style (preferred)
addc x y = x + y    -- [named for Haskell Brooks Curry]
```

- Functions may also take other functions as arguments

```
>  map isOdd [2,3,5]
[False,True,True]
```

Anonymous functions

- Can construct functional values without naming them:

```
> map (\x -> x+3) [2,3,5]
[5,6,8]
```

- "\" is pronounced "lambda": ASCII approximation of "λ".
  - In fact, in typeset/pretty-printed Haskell code, you may see the above rendered as "$map\ (\lambda x \rightarrow x + 3)\ [2,3,5]$".
- Could define previous functions more explicitly as:

```haskell
addt :: (Int, Int) -> Int   -- tupled style
addt = \p -> fst p + snd p

addc :: Int -> (Int -> Int) -- curried style
addc = \x -> \y -> x + y
```

- Note: `addc 3` actually returns the function `\y -> 3 + y`.
  - `addc 3 4` $\simeq$ `(\y -> 3 + y) 4` $\simeq$ `3 + 4` $\simeq$ `7`.

Infix operators

- Haskell makes no fundamental distinction between *functions* and *operators*, beyond lexing/parsing
- Two syntactic classes of identifiers:
  - Alphanumeric (prefix): any seq. of letters, digits, underscores
    - ... except a few *reserved* words, e.g., `let`
    - Must *start* with lowercase letter
    - Standard style: `longName`, not `long_name`
  - Symbolic (infix): any seq. of special characters (`!`, `#`, `$`, `+`, ...)
    - Except a few reserved operators, e.g., `->`.
    - Must not *start* with a colon
- Can use any operator as (two-argument) function by enclosing in parentheses: `(+) 2 3` evaluates to `5`.
- Conversely, can use any two-argument function as operator by enclosing in backticks: ``10 `mod` 4`` evaluates to `2`.
  - Can specify desired precedence and/or associativity for non-standard operators with `infix{l,r,}` keyword.

Polymorphism

- Functions (and other values) may be *polymorphic*
  - Have type *schemas*, where some concrete types have been replaced by (lowercase) *type variables*

    ```
    dup :: a -> (a, a)
    dup x = (x, x)
    ```
  - Type system will automatically *instantiate* such types to match use context:
    - `dup 5` evaluates to `(5, 5)`
    - `dup True` evaluates to `(True, True)`.
    - ...
- Sometimes polymorphism limited to certain *classes* of types:
  - Numeric types: `Int`, `Double`, ...
    - `(+) :: Num a => a -> a -> a`
    - `2 + 3` evaluates to `5`
    - `2.0 + 3.0` evaluates to `5.0`
    - `"2" + "3"` is a type error
  - Equality types: almost all except functions
    - `(==) :: Eq a => a -> a -> Bool`
  - More about type classes (including defining your own) next time.

Working with lists

- Have already seen how to take apart *tuples*
  - `let add (x, y) = x + y`
  - ``let (q, r) = 10 `quotRem` 3 in ...``
- For lists, note that `[3,4,5]` syntax is actually *syntactic sugar* for `3 : (4 : (5 : []))`
  - `[] :: [a]` is sometimes called *nil*.
  - `(:) :: a -> [a] -> [a]` is usually called *cons*.
- Any well-formed list (and there is no other kind!) is either empty (`[]`) or of the form $(h : t)$ for some h and t.
- Can define functions over lists by covering both possibilities:

```
myReverse :: [a] -> [a]
myReverse [] = []
myReverse (h : t) = myReverse t ++ [h]
```

Pattern matching, continued

- Can pattern match on several arguments at once

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x<=y then x:merge xs (y:ys)
                               else y:merge (x:xs) ys
```

- In case of overlaps, *first* successful match is chosen
- `ghci -W` *warns* about uncovered cases
  - Runtime error if matching fails
- Can also use `case` for pattern matching

```
   case filter isOK attempts of
     [] -> "no solutions"
     [x] -> "one solution"
     _ -> "several solutions"
```

  (Again indentation is significant.)
- Wildcard pattern _ matches everything

Even more pattern matching

- Patterns must only bind variables at most once; this is illegal:

```haskell
myElem :: a -> [a] -> Bool
myElem x [] = False
myElem x (x:ys) = True
myElem x (_:ys) = myElem x ys
```

- But can write with explicit Boolean *guard* on pattern

```haskell
myElem :: Eq a => a -> [a] -> Bool
myElem x [] = False
myElem x (y:ys) | x == y = True
myElem x (y:ys) = myElem x ys
```

- If guard evaluates to `False`, matching resumes with next case.

Programmer-defined data types

- Most non-trivial Haskell programs contain problem-specific type definitions.
- Simplest kind: *type aliases* (abbreviations)

```
type Name = (String, String)  -- family & given name
```

- Types may be *enumerations*:

```
data Color = Red | Green | Blue
  deriving (Show, Eq)
```

  The `deriving` clause puts `Color` in respective type classes.
- **Note:** both type name and *constructor* names must start with uppercase letter.
- Actually, `Bool` is just a predefined enumeration

```
data Bool = False | True
  deriving (Show, Eq, ...)
```

Value-carrying constructors

- Can associate extra data with some or all constructors:

```
data Figure = Point
            | Disc Double -- radius
            | Rectangle Double Double -- width, height

myFigure = Rectangle 3.0 4.0
```

- Define functions on datatype by pattern matching:

```
area :: Figure -> Double
area Point = 0.0
area (Disc r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

(Note parentheses around non-atomic patterns)

Record notation

- Sometimes not obvious what constructor arguments represent.
  - Simple solution: comments
- Alternative: *named fields*

```
data Figure = Point
            | Disc {radius :: Double}
            | Rectangle {width, height :: Double}
```

- Can use either positional or named style when constructing:

```
myFigure = Rectangle 3.0 4.0
myFigure = Rectangle {height = 4.0, width = 3.0}
```

- Can use field names to *project* out components

```
let a = width fig * height fig in ...
```

- **Note:** runtime error if `fig` is not a `Rectangle`
  - So normally use projections only for datatypes with exactly one constructor

More datatypes

- Datatype definitions may be *recursive*:

```
data Figure = ...
            | Stack Figure Figure
```

- Then functions on them are normally also recursive:

```
...
area (Stack f1 f2) = area f1 + area f2
```

- Datatypes may be polymorphic:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

myTree :: Tree Int
myTree = Node (Leaf 2) (Node (Leaf 3) (Leaf 4))
```

- Mutual recursion, possibly mixing `type` and `data` definitions:

```
data RoseTree a = RoseTree a (Forest a)  -- data, children
type Forest a = [RoseTree a]  -- zero or more
```

A few more built-in datatypes

- Have already seen lists:

  ```
  data [a] = [] | a : [a]  deriving ...
  ```

  Note: infix *constructors* start with colon

  - ... which is why infix *operators* must not.
  - Always possible to tell visually whether a name occurring in pattern is a constructor or a variable.

- Option (or "nullable") types

  ```
  data Maybe a = Nothing | Just a
  ```

  Useful especially for function return types:

  ```
  lookup :: Eq a => a -> [(a, b)] -> Maybe b
  ```

- Disjoint-union types:

  ```
  data Either a b = Left a | Right b
  ```

  So `Maybe a` is almost the same as `Either () a`

Tasks for this week

- Install Haskell on your computer
  - See Absalon page for details
- Talk to a fellow student about forming a group (two is max)
- Work on Exercise Set 0
- Attend lecture & labs on Thursday
  - Next lectures: Thursday 10:15-12:00, Tuesday **9:15-11:00**
- Use discussion forum on Absalon for questions outside of lecture and lab hours
  - Please open new discussion thread for each topic
- Solve Assignment 0, **due 20:00 on Wednesday**, next week
  - Submission instructions being fine-tuned