Introduction to Haskell Seminar Series

Fritz Solms

September 17, 2012

Purpose of seminar series

- Be able to take more formal approach to software development
  - Strengthen understanding of relationship between Mathematics and Programming.
- Learn Haskell as example of a pure functional programming language.
  - Be comfortable in functional paradigm.
- Learn within traditional seminar environment.
- **Note:**
  - We are **ALL** learning.
  - The person giving the seminar might just be 1 day ahead of you.

Lambda Luminaries

- Functional programming enthusiast group.
    - `http://www.meetup.com/lambda-luminaries/`
- Meet regularly.
- Discuss interesting topics.
- Might want to attend both
    - This structured seminar series.
    - General discussion/learning/enthusiast group, Lambda Luminaries

Defintion: Functional Programming

Definition

- Computation = evaluation of functions
  - Avoids state.
- Pure function
  - Pure computation with result.
  - No side effect.
    - can use any evaluation strategy
    - compiler can reorder/reorder evaluation of expressions
  - No state.

- Based on λ-Calculus
  - Formalization of mathematics through the use of functions
    - instead of set theory.
  - A formal system in mathematical logic for expressing computation.

Curry-Howard Correspondence

- Exposed direct relationships between programs and proofs:
  - *Program* **is** *Proof*
  - Natural deduction $\Leftrightarrow$ λ-calculus

Declarative versus Imperative Programming

- **Imperative Programming**
  - Program = instructions what to do.
  - Result = consequences of specified activities
  - imperative programs are composed of statements which change global state when executed.
- **Declarative Programming**
  - Programs are executed by evaluating expressions.
  - No mutable state.
  - Specify desired result / constraints

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Functional Programming Languages

- **Pure functional programming languages:**
  - No imperative features.
  - *Haskell*
  - Experimental languages: *Clean*, *Curry*, ...
- **Impure functional programming languages**
  - Support some imperative features.
  - *Lisp*
  - *Erlang*
  - *Scala*
  - *F#*
  - ...

Some applications of Haskell

- *XMonad*
  - Tiling window manager in under 500 lines of code.
  - Written to demonstrate: Haskell can be used for general app development.
- Haskell IDE *Leksah*
- Usage in industry
  - data transformation
  - trading decisions/systems (e.g. ABN Amro, Tsuru Capital, . . .)
  - complex mathematical models (Amgen)
  - handwriting recognition
  - digital signal processing (e.g. Ericsson)
  - design and verification of vehicle systems (Eaton)
  - game development
  - embedded software for gas trace detection (Sensor Sense)
  - find-it-keep-it web browser (keeps pages visited in DB)

Haskell: Feature Overview

- Referential transparency (pure functions)
  - any expression can be replaced by its value without changing behaviour of program.
- List comprehension
  - First-class list manipulation (can base new lists on existing lists).
- Guards/ constraints (|)
  - used for pattern matching
- Garbage collection
- Higher order functions
  - Functions which have other functions as parameter or return value
- Currying
  - transforming multi-parameter functions to chain of single-parameter functions.
- Lazy evaluation
  - delays the evaluation of an expression until its value is needed, and
  - avoids repeated evaluation.

Haskell uses strong typing

- Fails on type errors on compile time.
- Function as first class type.
    - Constants = functions with same value for all inputs.
    - No variables.
    - Infinite precision integer calculations.
- Type classes
    - Natural type variables and polymorphic functions.
    - Run-time type identification.
- Type inference.

Currying

- Mapping multi-arg func onto single-arg func
  - Returns func with one less parameter.
  - Supports partial application.
  - All Haskell functions: single parameter.
- Example:

```
max :: Int -> Int -> Int
```

  - receives `Int` as parameter
  - returns function `Int -> Int`
- Direct support for higher-order functions
  - take function as parameter and/or
  - return function.

Example: zipWith

- `zipwith` = function which takes & returns function :

```
_zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
_zipWith _ [] _ = []
_zipWith _ _ [] = []
_zipWith f (x:xs) (y:ys) = f x y : _zipWith f xs ys
```

- Usage:

```
:l _zipWith
   Compiling Main ( _zipWith.hs, interpreted )
Ok, modules loaded: Main.

_zipWith (*) [1,2,3] [5,6]
[5,12]
```

Lambda abstractions

- Anonymous functions.
- specify mapping is sufficient
  - e.g. $\lambda x \to x \times x$
  - Haskell:

```
\x -> x*x
```

  - need not give function a name
- Normally lambdas are passed to higher order functions:

```
map (\x -> x*x) [1,2,3]

[1,4,9]
```

Types in Haskell

```
data Polynomial = Line Float Float | Parabola Float Float Float
   deriving (Show)

value :: Polynomial -> Float -> Float
value (Line m c) x = m*x + c
value (Parabola a b c) x = a*x^2 + b*x + c

roots :: Polynomial -> [Float]
roots (Line m c) = [-c/m]
roots (Parabola a b c) =
   let rdet = sqrt(b^2-4*a*c)
   in [(-b-rdet)/(2*a),(-b+rdet)/(2*a)]

turningPoint :: Polynomial -> [(Float,Float)]
turningPoint (Line _ _) = []
turningPoint (Parabola a b c) =
   let   x = -b / (2*a)
   in [(x, value (Parabola a b c) x)]
```

- Also type synonyms/aliases.

Modules

- A source unit (a file)
- Exports selected functions and types.

```
module Some.Module (func1, type1, ...)

func1 :: ...
```

- Import
  - Unqualified import:

```
import Some.Module

or

import Some.Module (func1)
```

  - Qualified import:

```
import qualified Some.Module

Some.Module.func1 as SM

SM.func1 ...
```

Packages

- A package is a unit of distribution.
  - Collection of modules (files).
- Built via Cabal.
- Package meta-data specification:
  - globally unique package name,
  - version id,
  - dependency list (referrring to package dependencies),
  - list of exposed modules.
- Unit of distribution with Cabal-based metadata and build support.
- Collection of modules.

```
import Some.Module

or

import Some.Module (func1)
```

Haskell compilers and interpreters

- Many implementations
  - *Glasgow Haskell Compiler* (ghc)
    - most widely used.
    - optimizing compiler
    - implements Haskell 2010
    - available across most common platforms
    - includes interpreter (ghci)
    - supports concurrent & parallel programming
    - profiling (time & space)

Haskell IDEs

- *Leskah*
  - GTK-based *Haskell* IDE written in *Haskell*
  - Cross platform.
  - Takes some getting used to but encourages
    - best practices
    - Cabal-based development
- *EclipseFP*
  - ghc integration
  - Cabal support
  - ...
- *Visual Haskell*
  - *VisualStudio*-based *Haskell* IDE

Testing tools

- Most widely used: **QuickCheck**
    - Define set of truths as properties
        - Define expected properties with functions
    - QuickCheck asserts truths

Haskell Documentation Tool: Haddock

- Document functions, classes, types, data
  - haddock is executed against modules
- Documentation via comments embedded in code.
  - `-- |This function calculated the meaning of life`
- Generates HTML and/or LaTeX documentation.
  - `haddock --html myModule.hs`
  - `haddock --latex myModule.hs`
- Can link to existing module documentation.

Haskell Package Management

- *What is package management?*
  - Collection of tools automating
    - finding and storing software packages, and
    - installing, configuring, and removing packages on/from a system.
- Haskell package management tools
  - *HackageDB*
  - *Hoogle*
  - *Cabal*

HackageDB

- Central Haskell package repository.
  - Hosts CABAL-based versioned **source packages**.
  - Maintains versioning.
- Extensive library
  - Can be explored via web interface:
  - `http://hackage.haskell.org/packages/archive/pkg-list.html`

Hoogλe search tool

- `www.haskell.org/hoogle`
- Search on
  - Qualified or unqualified function or module name
  - mapping (lambda abstraction)
    - `[a] -> [a]` finds all functions mapping a list onto a lits.

Cabal

- **C**ommon **A**rchitecture for **B**uilding **A**pplications and **L**ibraries.
- Package creation
  - Builds packages to be
    - published through *HackageDB*
    - installed/managed through *Cabal-Install*
- update local copy of *Haskell* package list:
  - `cabal update`
- Install a Cabal-based package:
  - `cabal install haddock`

Definition: Smallest Divider

Definition

- For $n \in \mathbb{N}, n > 1$
  - Let $d = \mathrm{LD}(n)$ as smallest $d \in \mathbb{N} \mid 1 < d \leq \mathbb{N}, \frac{n}{d} = a \in \mathbb{N}$
  - Let $\mathbb{P}(n)$ as the set of prime numbers of n.

- $d = \mathrm{LD}(n)$ exists $\forall n \in \mathbb{N}, n > 1$
  - $d = n$ is always a divider.
  - $\Rightarrow$ set of dividers of n is non-empty.
- Defining `divides` in *Haskell*:

```
divides d n = rem n d == 0
```

Some useful observations:

Theorem

a) *$d = LD(n)$ is a prime number.*

b) *if n not prime, then $(LD(n))^2 \leq n$.*

Proof.

a) Proof by contradiction
   - Assume $d = \mathrm{LD}(n)$ is not *prime*.
   - The $\exists a, b \in \mathbb{N} \mid d = a \cdot b, 1 < a < d$
   - But then $a < d \in \mathbb{N}$ divides n.
   - This contradicts $d = \mathrm{LD}(n)$.
   - $\Rightarrow d = \mathrm{LD}(n)$ must be *prime*.

b) Proof by deduction
   - Assume n not *prime*.
   - Let $p = \mathrm{LD}(n)$
   - Then $\exists a \in \mathbb{N}, a > 1 \mid n = p \cdot a. \Rightarrow a$ divides n.
   - But $p > 1$ smallest divisor of $n. \Rightarrow p \leq a$
   - $\Rightarrow p^2 \leq p \cdot a = n \Rightarrow (\mathrm{LD}(n))^2 \leq n$

Supporting functions

- is prime if `LD n == n`
- Calc `LD` recursively via `LDF`

```haskell
-- Check whether integer d divides integer n (without remainder)
divides :: Integer -> Integer -> Bool
divides d n = rem n d == 0

-- Determine smallest divider large than k
ldf :: Integer -> Integer -> Integer
ldf k n | divides k n    = k
        | k^2 > n        = n
        | otherwise      = ldf (k+1) n

-- Determine smallest divider >1 of n
ld :: Integer -> Integer
ld n = ldf 2 n
```

Haskell Prime Check

- Now define `isPrime`
  - with truth property for unit testing

```
-- determines whether provided integer is a prime number
isPrime :: Integer -> Bool
isPrime n     | n <= 1      = False
              | otherwise = ld n == n
-- unit tests for isPrime
prop_primeIsFactor   n   | isPrime n = length (filter (== 0) (map(\x
    -> rem n x) [2..(n-1)])) == 0
                         | otherwise = True
```