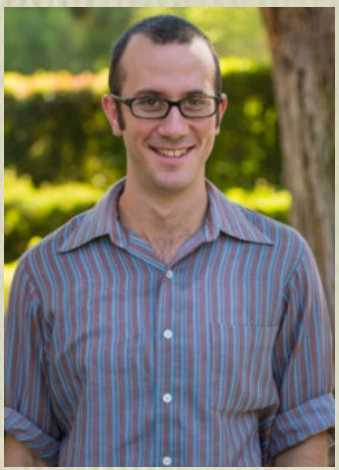
Lecture 1: Haskell



CSC 131
Spring, 2017

Kim Bruce (for Michael Greenberg)

Overview

- Most fundamental tool for programmers
  - Understand what happens at run-time
  - Understand how choice of language affects programmers
- Prof. Greenberg will go in more depth on return.

Partners

- Homework will be done in (randomly chosen) pairs.
  - Watch piazza for pairings for first homework!

Haskell

According to Larry Wall (designer of PERL): ... a language by geniuses for geniuses

He's wrong — at least about the latter part
though you might agree when we talk about monads

Haskell 98

- Purely functional *(unlike ML and Racket)*
- Functions are first-class values
- Statically scoped
- Strong, static typing via type inference *(like ML)*
  - Type-safe
- Parametric polymorphism
- Type classes

Haskell (cont)

- Rich type system including support for ADT's
- Non-strict (lazy) evaluation
- Imperative features emulated using monads.
- Garbage collection
- Compiled or interpreted.
- Named after Haskell Curry -- early contributor to lambda calculus and combinatory logic

Read Haskell Tutorials

- https://www.haskell.org/documentation
- I like “Learn you a Haskell for greater good”
- O’Reilly text: “Real World Haskell” free on-line
- Print Haskell cheat sheet
- Use “The Haskell platform”, available at
  - http://www.haskell.org/

Using GHC

- to enter interactive mode type: ghci
  - :load myfile.hs -- :l also works
  - after changes type :reload myfile.hs
  - Control-d to exit
  - :set +t -- prints more type info when interactive
  - “it” is result of expression

Built-in data types

- Unit has only ( )
- Bool: True, False with not, &&, ||
- Int: 5, -5, with +, -, *, ^, =, /=, <, >, >=, ...
  - div, mod defined as prefix operators (`div` *infix*)
  - Int fixed size (usually 64 bits)
  - Integer gives unbounded size
- Float, Double: 3.17, 2.4e17 w/ +, -, *, /, =, <, >, >=, <=, sin, cos, log, exp, sqrt, sin, atan.

More Basic Types

- Char: ‘n’

list of Char

- String = [Char], *not really primitive*
  - "hello"++" there", length
  - No substring, but `isInfixOf` for all lists
  - Also ‘isPrefixOf`, `isSuffixOf’

Prefix op w/out ``!

import Data.List

- Type classes (later) provide relations between classes.

Interactive Programming with ghci

- Type expressions and run-time will evaluate
- Define abbreviations with “let”
  - let double n = n + n
  - let seven = 7
- “let” not necessary at top level in programs loaded from files

Working with Files

- Examples *(demo)*:
  - mean:: Int -> Int -> Int
  - fact: Int -> Int
  - fib: Int -> Int *(several ways)*

System will infer types, but get much better error messages if you put them in!

Lists

- Lists
  - [2,3,4,9,12]: [Integer]
  - [] -- empty list
  - [m..n] shorthand for [m, m+1, ..., n]
  - fst:rest *pattern matching any non-empty list*
  - Must be homogenous
  - Built-in functions: length, ++, :, map, rev
    - also head, tail, *but normally avoid w/pattern matching!*

Polymorphic Types

- [1,2,3]:: [Integer]
- ["abc", "def"]:: [[Char]], ...
- []:: [a]
- map:: (a → b) → ([a] → [b])
- *Use* :t exp *to get type of* exp

Pattern Matching

- Decompose lists:

- [1,2,3] = 1:(2:(3:[]))

- Define functions by cases using pattern matching:

```
prod [] = 1
prod (fst:rest) = fst * (prod rest)
```

Pattern Matching

- Desugared through case expressions:
  - ```
    head' :: [a] -> a
    head' [] = error "No head for empty lists!"
    head' (x:_) = x
    ```
- equivalent to
  - ```
    head' xs = case xs of
                 [] -> error "No head for empty lists!"
                 (x:_) -> x
    ```

Exercises

- Exercise: Write
  - sum nums = *sum of elts of lst*
  - filterIt nums cond = *sublist of elts of nums satisfying cond*
    - *there is a built-in filter:(a->Bool) -> [a] -> [a]*

Type constructors

- Tuples
  - (17,"abc", True) : (Integer , [Char] , Bool)
  - fst, snd defined only on pairs
- Records exist as well

More Pattern Matching

- (x,y) = (5 `div` 2, 5 `mod` 2)
- hd:tl = [1,2,3]
- hd:_ = [4,5,6]
  - "_" is wildcard.

Static Typing

- Strongly typed via type inference
  - head:: [a] → a
    tail:: [a] → [a]
  - last [x] = x
    last (hd:tail) = last tail
- System deduces most general type, [a] -> a
  - Look at algorithm later

Static Scoping

- What is the answer?

```
- let x = 3
- let g y = x + y
- g 2
- let x = 6
- g 2
```

- What is the answer in original LISP?

```
- (define x 3)
- (define (g y) (+ x y))
- (g 2)
- (define x 6)
- (g 2)
```

Static Scoping

- What is the answer?

```
- let x = 3
- let g y = x + y
- g 2
- let x = 6
- g 2
```

```
{
  const x = 3
  {
    g(y) = x + y
    {
      print (g 2)
      const x = 6
      {
        print (g 2)
      }
    }
  }
}
```

- What is the answer in original LISP?

```
- (define x 3)
- (define (g y) (+ x y))
- (g 2)
- (define x 6)
- (g 2)
```

Local Declarations

```
roots (a,b,c) =
    let      -- indenting is significant
        disc = sqrt(b*b-4.0*a*c)
    in
        ((-b + disc)/(2.0*a),(-b - disc)/(2.0*a))

*Main> roots(1,5,6)
(-2.0,-3.0)
```

or

```
roots' (a,b,c) = ((-b + disc)/(2.0*a),
                  (-b - disc)/(2.0*a))
    where disc = sqrt(b*b-4.0*a*c)
```

Anonymous functions

- dble x = x + x
- *abbreviates*
- dble = \x -> x + x

Defining New Types

- Type abbreviations
  - type Point = (Integer, Integer)
  - type Pair a = (a,a)
- data definitions
  - create new type with constructors as tags.
  - generative
- data Color = Red | Green | Blue

  See more complex examples later

Type Classes Intro

- Specify an interface:
  - class Eq a where
    ```
    (==) :: a -> a -> Bool    -- specify ops
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)     -- optional implementations
    x /= y = not (x == y)
    ```
  - data TrafficLight = Red | Yellow | Green
    instance Eq TrafficLight where
    ```
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
    ```