Lecture 2: Haskell

CSC 131
Spring, 2019
Kim Bruce

Read Haskell Tutorials

- All on links page from course web page
- I like “Learn you a Haskell for greater good”
- O’Reilly text: “Real World Haskell” free on-line
  - Just get overview in class!
- Print Haskell cheat sheet
- Use “The Haskell platform”, available at
  - http://www.haskell.org/

Office Hours Today

- Because of visitor and TA organizing, no office hours today.
- E-mail if want to meet tomorrow (usually don’t have office hours on Friday)

Using GHC

- to enter interactive mode type: ghci
  - :load myfile.hs   -- :l also works
  - after changes type :reload or :r
  - Control-d to exit
  - :set +t   -- prints more type info when interactive
  - “it” is result of expression
  - Evaluate “it + 1” gives one more than previous answer.
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’

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

Lists

• Lists
  - [2,3,4,9,12]: [Integer]
  - [] -- empty list
  - Must be homogenous
  - Functions: length, ++, :, map, rev
    • also head, tail, but normally don't use!
Polymorphic Types

- \([1,2,3]:: [\text{Integer}]\)
- \([\text{“abc”, “def”}]:: [[\text{Char}]], \ldots\)
- \([]:: [a]\)
- map:: (a \rightarrow b) \rightarrow ([a] \rightarrow [b])
- \text{Use } :t \text{ exp to get type of exp}

Pattern Matching

- Decompose lists:
  - \([1,2,3] = 1:(2:(3:[]))\)
- Define functions by cases using pattern matching:
  - \text{prod } [] = 1\n  - \text{prod } (\text{fst:rest}) = \text{fst } * (\text{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

Type constructors

- Tuples
  - \((17, \text{“abc”}, \text{True}) : (\text{Integer}, [\text{Char}], \text{Bool})\)
  - \text{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

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

• 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)
Local Declarations

```haskell
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

```haskell
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.
  - `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`
Common Type Classes

- Eq, Ord, Enum, Bounded, Show, Read
  - See http://www.haskell.org/tutorial/stdclasses.html

- data defs pick up default if add to class:
  - data ... deriving (Show, Eq)

- Can redefine:
  - instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

More Type Classes

- class (Eq a) => Num a where ...
  - instance of Num a must be Eq a

- Polymorphic function types can be prefixed w/ type classes
  - test x y = x < y has type (Ord a) => a -> a -> Bool
  - Can be used w/ x, y of any Ord type.

- More later ...
  - Error messages often refer to actual parameter needing to be instance of a class — to have an operation.

Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a -> b) -> ([a] -> [b])

- Build new control structures
  - listify oper identity [] = identity
    listify oper identity (fst:rest) =
      oper fst (listify oper identity rest)

  - sum' = listify (+) 0
  - mult' = listify (*) 1
  - and' = listify (&&) True
  - or' = listify (||) False

Exercise

- Is listify left or right associative?
  - What is listify (-) o [3,2,1]? 2 or -6 or o ???

- How can we change definition to associate the other way?

  See built-in foldl and foldr