Lecture 7: Haskell

CSCI 131
Fall, 2011

Kim Bruce

Homework 1 Comments

- Present explanations for answers.
  - Convince us you know why answer is correct
- Include name in all files.
- Give complete answers (which language?)
- Turn in a single (zipped if necessary) file.
- For unused features, libraries don't count.
  - looking for features you avoid for some reason
  - e.g., wildcard types, exceptions, inner classes, ...

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]:: [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

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)`

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

Type Classes

- 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
- 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
- :info TypeClass
  - gives interface and instances in scope
    ```