Programming in Haskell

S P Suresh

`http://www.cmi.ac.in/~spsuresh`

Lecture 7

August 30, 2017

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
  - An "answer" is an expression that cannot be further simplified
- Built-in simplifications
  - `3+5` $\Rrightarrow$ `8`
  - `True || False` $\Rrightarrow$ `True`

Computation as rewriting

- Simplifications based on user defined functions
  - ```
    power :: Int -> Int -> Int
    power x 0 = 1
    power x n = x * (power x (n-1))
    ```
    ```

Computation as rewriting

- `power 3 2`

⇛ `3 * (power 3 (2-1))`	user definition
⇛ `3 * (power 3 1)`	built in simplification
⇛ `3 * (3 * (power 3 (1-1)))`	user definition
⇛ `3 * (3 * (power 3 0))`	built in simplification
⇛ `3 * (3 * 1)`	user definition
⇛ `3 * 3`	built in simplification
⇛ `9`	built in simplification

Order of evaluation

- `(8+3)*(5-3)` ⇛ `11*(5-3)` ⇛ `11*2` ⇛ `22`
- `(8+3)*(5-3)` ⇛ `(8+3)*2` ⇛ `11*2` ⇛ `22`
- `power (5+2) (4-4)` ⇛ `power 7 (4-4)`
  ⇛ `power 7 0` ⇛ `1`
- `power (5+2) (4-4)` ⇛ `power (5+2) 0` ⇛ `1`
- What would `power (div 3 0) 0` return?

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- In `head (2:reverse [1..5])`
  - `f` is `head`
  - `e` is `(2:reverse [1..5])`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

Lazy evaluation ...

- The argument is not evaluated if the function definition does not force it to be evaluated.
  - `head (2:reverse [1..5])` ⇛ `2`
- Argument is evaluated if needed
  - `last (2:reverse [1..5])` ⇛
    `last (2:[5,4,3,2,1])` ⇛ `1`

Lazy evaluation ...

- What would `power (div 3 0) 0` return?
  - ```
    power :: Int -> Int -> Int
    power x 0 = 1
    power x n = x * (power x (n-1))
    ```
- First definition ignores value of `x`
- `power (div 3 0) 0` returns `1`
    ```

Lazy evaluation ...

- If all simplifications are possible, order of evaluation does not matter, same answer
- One order may terminate, another may not
- Lazy evaluation expands arguments by “need”
    - Can terminate with an undefined sub-expression if that expression is not used

Infinite lists

- ```
  infinite_list :: [Int]
  infinite_list = inflistaux 0
      where
      inflistaux :: Int -> [Int]
      inflistaux n = n:(inflistaux (n+1))
  ```

- `infinite_list` ⇛ `[0,1,2,3,4,5,6,7,8,9,10,12,…]`

- `head (infinite_list)` ⇛ `head(0:inflistaux 1)` ⇛ `0`

- `take 2 (infinite_list)` ⇛
  `take 2 (0:inflistaux 1)` ⇛
  `0:(take 1 (inflistaux 1))` ⇛
  `0:(take 1 (1:inflistaux 2))` ⇛ `[0,1]`
  ```

Infinite lists

- Range notation extends to infinite lists
  - `[m..]` ⇛ `[m,m+1,m+2,…]`
  - `[m,m+d..]` ⇛ `[m,m+d,m+2d,m+3d,…]`
- Sometimes infinite lists simplify function definition

Primes

- Sometimes infinite lists simplify function definition

- ```
  primes = filterPrime [2..]
      where filterPrime (p:xs) =
                p : filterPrime [x | x <- xs, x `mod` p /= 0]
  ```

- ```
  primes100 = take 100 primes
  ```

Summary

- In functional programming, computation is rewriting
- Haskell uses lazy evaluation — simplifies outermost expression first
- Lazy evaluation allows us to work with infinite lists