Programming in Haskell
Aug-Nov 2015

LECTURE 12

SEPTEMBER 15, 2015

S P Suresh
Chennai Mathematical Institute

Recap

- Recap of efficiency analysis and sorting

Lazy evaluation

- Recall that Haskell uses lazy evaluation
  - Outermost reduction
  - Simplify function definition first
  - Compute argument value only if needed

Infinite lists

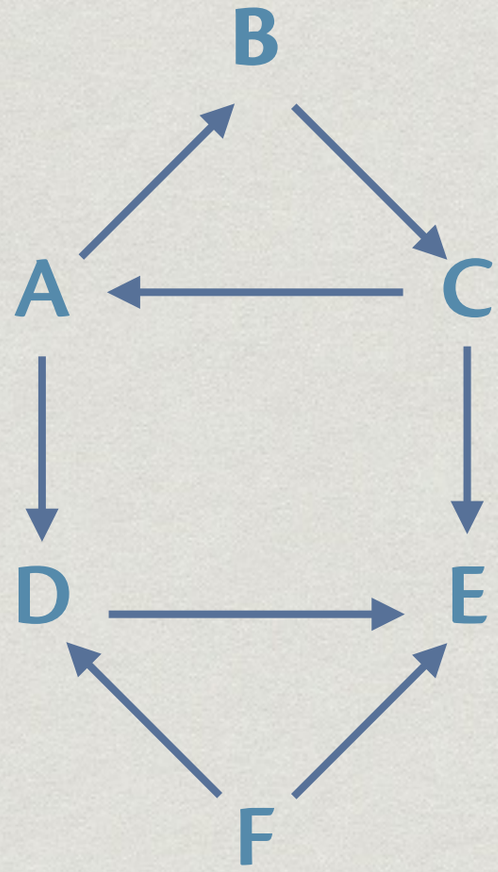
- Lazy evaluation allows meaningful use of infinite lists

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

- head (infinite_list) ⇛ 0
- take 2 (infinite_list) ⇛ [0,1]
- [m..] ⇛ [m,m+1,m+2,…]

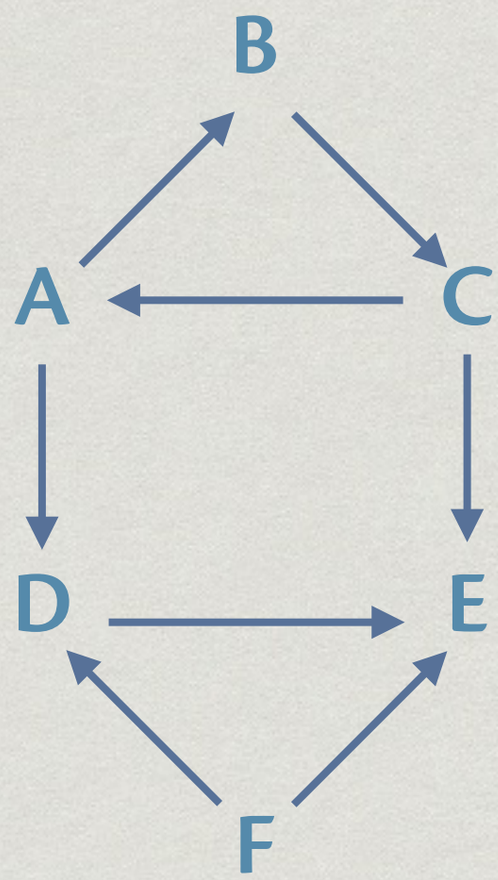
Graphs

- Graphs
  - A, B, … are nodes or vertices
  - (A,B), (A,D), … are (directed edges)



Graphs ...

```
edge :: Char -> Char ->  Bool
edge 'A' 'B' = True
edge 'A' 'D' = True
edge 'B' 'C' = True
edge 'C' 'A' = True
edge 'C' 'E' = True
edge 'D' 'E' = True
edge 'F' 'D' = True
edge 'F' 'E' = True
edge  _   _  = False
```



Connectivity

Connectivity

- Want to check connectivity

Connectivity

- Want to check connectivity

```
connected :: Char -> Char -> Bool
```

Connectivity

- Want to check connectivity

  `connected :: Char -> Char -> Bool`

  - `connected x y` is `True` if and only if there is a path from `x` to `y` using the given set of edges

Connectivity

- Want to check connectivity

  `connected :: Char -> Char -> Bool`

  - `connected x y` is `True` if and only if there is a path from `x` to `y` using the given set of edges

- Inductive definition

Connectivity

- Want to check connectivity

  `connected :: Char -> Char -> Bool`

  - `connected x y` is `True` if and only if there is a path from `x` to `y` using the given set of edges

- Inductive definition

  If `connected x y` and `edge y z` then `connected x z`

Connectivity

- Want to check connectivity

  `connected :: Char -> Char -> Bool`

  - `connected x y` is `True` if and only if there is a path from `x` to `y` using the given set of edges

- Inductive definition

  If `connected x y` and `edge y z` then `connected x z`

- Difficult to translate this directly into Haskell

Building paths

Building paths

- Inductively build up paths

Building paths

- Inductively build up paths
  - Only one path of length 0

Building paths

- Inductively build up paths
  - Only one path of length 0
  - Extend path of length k to length k+1 by adding an edge

Building paths

- Inductively build up paths
  - Only one path of length 0
  - Extend path of length k to length k+1 by adding an edge

```
type Path = [Char]
extendpath :: Path -> [Path]
extendpath [] = [ [c] | c <- [‘A'..'F']]
extendpath p =
 [p++c | c <- ['A'..'F'], edge (last p) c]
```

Building paths …

- `map extendpath` over the list of paths of length `k` to get the list of paths of length `k+1`.

- ```
  extendall :: [Path] -> [Path]
  extendall [] = [[c] | c <- [‘A'..'F']
  extendall l  = concat [extend p | p <- l]
               = [ll | p <- l, ll <- extend p]
  ```
  ```

Building paths …

Building paths …

- Built-in function **iterate**

Building paths …

- Built-in function **iterate**
- `iterate :: (a -> a) -> a -> [a]`

Building paths …

- Built-in function **iterate**
- `iterate :: (a -> a) -> a -> [a]`
  - `iterate f x ⇛`
    `[x, f x, f (f x), f (f (f x))) …]`

Building paths …

- Built-in function **iterate**
- `iterate :: (a -> a) -> a -> [a]`
- `iterate f x ⇛`
  `[x, f x, f (f x), f (f (f x))) …]`
- To generate all paths

Building paths …

- Built-in function **iterate**
- `iterate :: (a -> a) -> a -> [a]`
- `iterate f x ⇛`
  `[x, f x, f (f x), f (f (f x))) …]`
- To generate all paths

```
allpaths = iterate extendall []
```

Connectivity

Connectivity

- To check if x and y are connected, need to check for paths without loops from x to y

Connectivity

- To check if x and y are connected, need to check for paths without loops from x to y
- Given n nodes overall, a loop free path can have at most n-1 edges

Connectivity

- To check if x and y are connected, need to check for paths without loops from x to y
- Given n nodes overall, a loop free path can have at most n-1 edges
- Suffices to examine first n+1 entries in `allpaths`

Connectivity

- To check if x and y are connected, need to check for paths without loops from x to y

- Given n nodes overall, a loop free path can have at most n-1 edges

- Suffices to examine first n+1 entries in `allpaths`

```
take (n+1) allpaths
```

Connectivity …

Connectivity …

- Extract endpoints of paths of length at most n-1

Connectivity …

- Extract endpoints of paths of length at most n-1

```
connectedpairs  =
  [(head p, last p) | l <- firstn, p <- l]
  where
   firstn = take n allpaths
   allpaths = iterate extendall [[]]
```

Connectivity ...

- Extract endpoints of paths of length at most n-1

```
connectedpairs  =
  [(head p, last p) | l <- firstn, p <- l]
    where
    firstn = take n allpaths
    allpaths = iterate extendall [[]]
```

- Finally

Connectivity ...

- Extract endpoints of paths of length at most n-1

```
connectedpairs  =
  [(head p, last p) | l <- firstn, p <- l]
    where
    firstn = take n allpaths
    allpaths = iterate extendall [[]]
```

- Finally

```
connected x y = elem (x,y) connectedpairs
```

Connectivity …

Connectivity …

- `extendall` generates loops, but we don't care

Connectivity …

- `extendall` generates loops, but we don't care
  - For instance, path `['A','B','C','A','B','C']` belongs to the sixth iteration of `extendall []`

Connectivity …

- `extendall` generates loops, but we don't care
  - For instance, path `['A','B','C','A','B','C']` belongs to the sixth iteration of `extendall []`
- We just want to ensure that every pair `(x,y)` in `connected` is enumerated by the step `n`

Connectivity …

- `extendall` generates loops, but we don't care
  - For instance, path `['A','B','C','A','B','C']` belongs to the sixth iteration of `extendall []`
- We just want to ensure that every pair `(x,y)` in `connected` is enumerated by the step `n`
- `connected` is the reflexive transitive closure of the `edge` relation