Programming in Haskell
Aug-Nov 2015

LECTURE 4

AUGUST 13, 2015

S P Suresh
Chennai Mathematical Institute

Lists

- To describe a collection of values
  - [1,2,3,1] is a list of Int
  - [True,False,True] is a list of Bool
- Elements of a list must be of a uniform type
  - Cannot write [1,2,True] or [3,'a']

Lists …

- List with values of type T has type [T]
  - [1,2,3,1] :: [Int]
  - [True,False,True] :: [Bool]
  - [] denotes the empty list, for all types
- Lists can be nested
- [[3,2], [], [7,7,7]] is of type [[Int]]

Internal representation

- To build a list: add one element at a time to the front (left)
  - Operator to append an element is :
  - 1:[2,3] ⇛ [1,2,3]
- All Haskell lists are built this way, starting with []
  - [1,2,3] is actually 1:(2:(3:[]))
  - : is right associative, so 1:2:3:[] is 1:(2:(3:[]))
- 1:[2,3] == 1:2:3:[], 1:2:[3] == [1,2,3], … all return True

Decomposing lists

- Functions head and tail
  - head (x:xs) ⇛ x
  - tail (x:xs) ⇛ xs
  - Both undefined for []
  - head returns a value, tail returns a list

Defining functions on lists

- Recall inductive definition of numeric functions
  - Base case is f 0
  - Define f (n+1) in terms of n+1 and f n
- For lists: induction on list structure
  - Base case is []
  - Define f (x:xs) in terms of x and f xs

Example: length

- Length of [] is 0
- Length of (x:xs) is 1 more than length of xs

```
mylength :: [Int] -> Int
mylength [] = 0
mylength l = 1 + mylength (tail l)
```

Pattern matching

- A nonempty list decomposes uniquely as x:xs
  - Pattern matching implicitly separates head, tail
  - Empty list will not match this pattern
  - Note the bracketing: (x:xs)

```
mylength :: [Int] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

Example: sum of values

- Sum of [] is 0
- Sum of (x:xs) is x plus sum of xs

```
mysum :: [Int] -> Int
mysum [] = 0
mysum (x:xs) = x + mysum xs
```

List notation

- Positions in a list are numbered 0 to n-1
  - l!!j is the value at position j
  - Accessing value j takes time proportional to j
    - Need to “peel off” j applications of : operator
  - Contrast with arrays, which support random access

List notation …

- [m..n] ⇛ [m, m+1, …, n]
    - Empty list if n < m

      [1..7] = [1,2,3,4,5,6,7]
      [3..3] = [3]
      [5..4] = []

List notation …

- Skipping values (arithmetic progressions)

  [1,3..8] ⇛ [1,3,5,7]
  [2,5..19] ⇛ [2,5,8,11,14,17]

- Descending order

  [8,7..5] ⇛ [8,7,6,5]
  [12,8..-9] ⇛ [12,8,4,0,-4,-8]

Example: appendright

- Add a value to the end of the list
  - An empty list becomes a one element list
  - For a nonempty list, recursively append to the tail of the list

```
appendr :: Int -> [Int] -> [Int]
appendr x [] = [x]
appendr x (y:ys) = y:(appendr x ys)
```

Example: attach

- Attach two lists to form a single list
  - attach [3,2] [4,6,7] ⇛ [3,2,4,6,7]
- Induction on the first argument

```
attach :: [Int] -> [Int] -> [Int]
attach [] l = l
attach (x:xs) l = x:(attach xs l)
```

- Built in operator ++
  - [3,2] ++ [4,6,7] ⇛ [3,2,4,6,7]

Example: reverse

- Remove the head
- Recursively reverse the tail
- Attach the head at the end

```
reverse ::[Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Example: is sorted

- Check if a list of integers is in ascending order
- Any list with less than two elements is OK

```
ascending :: [Int] -> Bool
ascending [] = True
ascending [x] = True
ascending (x:y:ys) = (x <= y) &&
                     ascending (y:ys)
```

- Note the two level pattern

Example: alternating

- Check if a list of integers is alternating
  - Values should strictly increase and decrease at alternate positions
- Alternating list can start in increasing order (updown) or decreasing order (downup)
  - tail of a downup list is updown
  - tail of an updown list is downup

Example: alternating ...

```
alternating :: [Int] -> Bool
alternating l = (updown l) || (downup l)

updown :: [Int] -> Bool
updown [] = True
updown [x] = True
updown (x:y:ys) = (x < y) && (downup (y:ys))

downup:: [Int] -> Bool
downup [] = True
downup [x] = True
downup (x:y:ys) = (x > y) && (updown (y:ys))
```

Built in functions on lists

- `head`, `tail`, `length`, `sum`, `reverse`, …
- `init l`, returns all but the last element

  init [1,2,3] ⇛ [1,2]
  init [2] ⇛ []

- `last l`, returns the last element in `l`

  last [1,2,3] ⇛ 3
  last [2] ⇛ 2

Built in functions on lists ...

- `take n l`, returns first `n` values in `l`
- `drop n l`, leaves first `n` values in `l`
  - Do the “obvious” thing for bad values of `n`
  - `l == (take n l) ++ (drop n l)`, always

Built in functions on lists ...

- Defining take

```
mytake :: Int -> [Int] -> [Int]
mytake n [] = []
mytake n (x:xs)
 | n == 0 = []
 | n > 0 = x:(mytake (n-1) xs)
 | otherwise = []
```

Summary

- Functions on lists are typically defined by induction on the structure
- Point to ponder
  - Is there a difference in how length works for [Int], [Float], [Bool], …?
  - Can we assign a more generic type to such functions?