Programming in Haskell
Aug–Nov 2015

LECTURE 21

NOVEMBER 3, 2015

S P Suresh
Chennai Mathematical Institute

Arrays in Haskell

- Lists store a collection of elements
- Accessing the i-th element takes i steps
- Would be useful to access any element in constant time
- **Arrays** in Haskell offer this feature
- The module **Data.Array** has to be imported to use arrays

Arrays in Haskell

- ```
  import Data.Array
  myArray :: Array Int Char
  ```
- The **indices** of the array come from `Int`
  The **values** stored in the array come from `Char`
- ```
  myArray = listArray (0,2) ['a','b','c']
  ```

Index	0	1	2
Value	'a'	'b'	'c'

Creating arrays: listArray

- `listArray ::`
  `Ix i => (i,i) -> [e] -> Array i e`
- `Ix` is the class of all **index** types, those that can be used as indices in arrays
- If `Ix a`, `x` and `y` are of type `a` and `x < y`, then **the range of values between `x` and `y` is defined and finite**

Creating arrays: listArray

- The class `Ix` includes `Int`, `Char`, `(Int,Int)`, `(Int,Int,Char)` etc. but not `Float` or `[Int]`
- The first argument of `listArray` specifies the smallest and largest index of the array
- The second argument is the list of values to be stored in the array

Creating arrays: listArray

* `listArray (1,1) [100..199]`
  `array (1,1) [(1,100)]`

* `listArray ('m','p') [0,2..]`
  `array ('m','p') [('m',0),('n',2),('o',4),('p',6)]`

* `listArray ('b','a') [1..]`
  `array ('b','a') []`

* `listArray (0,4) [100..]`
  `array (0,4) [(0,100),(1,101),(2,102),(3,103),(4,104)]`

* `listArray (1,3) ['a','b']`
  `array (1,3) [(1,'a'),(2,'b'),(3,*** Exception: (Array.!): undefined array element`

Creating arrays: listArray

- The value at index `i` of array `arr` is accessed using `arr!i` (unlike `!!` for list access)
- `arr!i` returns an exception if no value has been defined for index `i`
- `myArr = listArray (1,3) ['a','b','c']`
- ```
  myArr ! 4
  *** Exception: Ix{Integer}.index: Index (4) out of range ((1,3))
  ```
  ```

Creating arrays: listArray

- Haskell arrays are **lazy**: the whole array need not be defined before some elements are accessed
- For example, we can fill in locations `0` and `1` of `arr`, and define `arr!i` in terms of `arr!(i-1)` and `arr!(i-2)`, for `i >= 2`
- `listArray` takes time proportional to the range of indices

First example: Fibonacci

- Recall the function `fib`, which computes the n-th Fibonacci number F(n)

- ```
  fib 0 = 1
  fib 1 = 1
  fib n = fib (n-1) + fib (n-2)
  ```

- Lots of recursive calls, computing the same value over and over again

- Computes F(n) in unary, in effect
  ```

Fibonacci using arrays

- ```
  import Data.Array
  fib :: Int -> Integer
  fib n = fibA!n
    where
      fibA :: Array Int Integer
      fibA = listArray (0,n) [f i | i <-[0..n]]
      f 0 = 1
      f 1 = 1
      f i = fibA!(i-1) + fibA!(i-2)
  ```
- The `fibA` array is used even before it is completely defined, thanks to Haskell's laziness
- Works in O(n) time
  ```

Creating arrays: array

- `array :: Ix i => (i, i) -> [(i, e)] -> Array i e`
  Creates an array from an associative list

- The associative list need not be in ascending order of indices

```
myArray = array (0,2)
          [(1,"one"),(0,"zero"),(2,"two")]
```

- The associative list may also omit elements

```
myArray = array (0,2) [(0,"abc"), (2,"xyz")]
```

- `array` also takes time proportional to the range of indices

More on indices

- Any type `a` belonging to the type class `Ix` must provide the functions

```
range       :: (a,a) -> [a]
index       :: (a,a) -> a -> Int
inRange     :: (a,a) -> a -> Bool
rangeSize   :: (a,a) -> Int
```

More on indices

* `range        :: (a,a) -> [a]`
  `range` gives the list of indices in the subrange defined by the bounding pair

* ```
  range (1,2) = [1,2]
  range ('m','p') = "mnop"
  range ('z','a') = ""
  ```
  ```

More on indices

* `index :: (a,a) -> a -> Int`
  The position of a subscript in the subrange

* ```
  index (-50,60) (-50) = 0
  index (-50,60) 35    = 85
  index ('m','p') 'o'  = 2
  index ('m','p') 'a'
        *** Exception: Ix{Char}.index: Index ('a')
  out of range (('m','p'))
  ```
  ```

More on indices

- `inRange    :: (a,a) -> a -> Bool`
  Returns `True` if the given subscript lies in the range defined by the bounding pair

- ```
  inRange (-50,60) (-50) = True
  inRange (-50,60) 35    = True
  inRange ('m','p') 'o'  = True
  inRange ('m','p') 'a'  = False
  ```
  ```

More on indices

- `rangeSize  :: (a,a) -> Int`
  The size of the subrange defined by the bounding pair

- ```
  rangeSize (-50,60)  = 111
  rangeSize ('m','p') = 4
  rangeSize (50,0)    = 0
  ```
  ```

Functions on arrays

* `(!)      :: Ix i => Array i e -> i -> e`
  The value at the given index in an array

* `bounds   :: Ix i => Array i e -> (i,i)`
  The bounds with which an array was constructed

* `indices :: Ix i => Array i e -> [i]`
  The list of indices of an array in ascending order

Functions on arrays

- `elems   :: Ix i => Array i e -> [e]`
  The list of elements of an array in index order

- `assocs  :: Ix i => Array i e  -> [(i,e)]`
  The list of associations of an array in index order

- `(//) :: Ix i => Array i e -> [(i,e)] -> Array i e`
  Update the array using the association list provided

Second example: `lcss`

- Given two strings **str1** and **str2**, find the **length** of the **longest common subsequence** of **str1** and **str2**

- ```
  lcss "agcat" "gact" = 3
             – "gat" is the subsequence
  lcss "abracadabra" "bacarrat" = 6
          – "bacara" is the subsequence
  ```
  ```

Second example: `lcss`

- ```
  lcss ""      _      = 0
  lcss _       ""     = 0
  lcss (c:cs) (d:ds)
      | c == d        = 1 + lcss cs ds
      | otherwise     = max (lcss (c:cs) ds)
                            (lcss cs (d:ds))
  ```

- `lcss cs ds` takes time >= 2^n, when cs and ds are of length n

- Similar problem to `fib`, same recursive call made multiple times

- **Store the computed values for efficiency**
  ```

lcss using arrays

- We restate the recursive `lcss` in terms of indices

- ```
  lcss :: String -> String -> Int
  lcss str1 str2 = lcss' 0 0
      where
        m = length str1
        n = length str2
        lcss' i j
          | i >= m || j >= n   = 0
          | str1!!i == str2!!j = 1 + lcss' (i+1) (j+1)
          | otherwise          = max (lcss' i (j+1))
                                     (lcss' (i+1) j)
  ```
  ```

lcss using arrays

- ```
  lcss :: String -> String -> Int
  lcss str1 str2 = lcssA!(0,0)
     where
       m = length str1
       n = length str2
       lcssA = array ((0,0),(m,n))
                [((i,j),f i j) | i <- [0..m],j <- [0..n]]
       f i j
        | i >= m || j >= n   = 0
        | str1!!i == str2!!j = 1 + lcssA!((i+1),(j+1))
        | otherwise          = max (lcssA ! (i,(j+1)))
                                   (lcssA ! ((i+1),j))
  ```
  ```

lcss using arrays

- ```
  lcss :: String -> String -> Int
  lcss str1 str2 = lcssA!(0,0)
      where
        m = length str1
        n = length str2
        lcssA = array ((0,0),(m,n))
                  [((i,j),f i j) | i <- [0..m],j <- [0..n] ]
  ```

- `lcssA` is a two-dimensional array. Indices are of type `(Int,Int)`

- Drawback?? The repeated use of `(!!)` in accessing `str1` and `str2`

- Solution? Turn the strings to arrays!
  ```

lcss using arrays

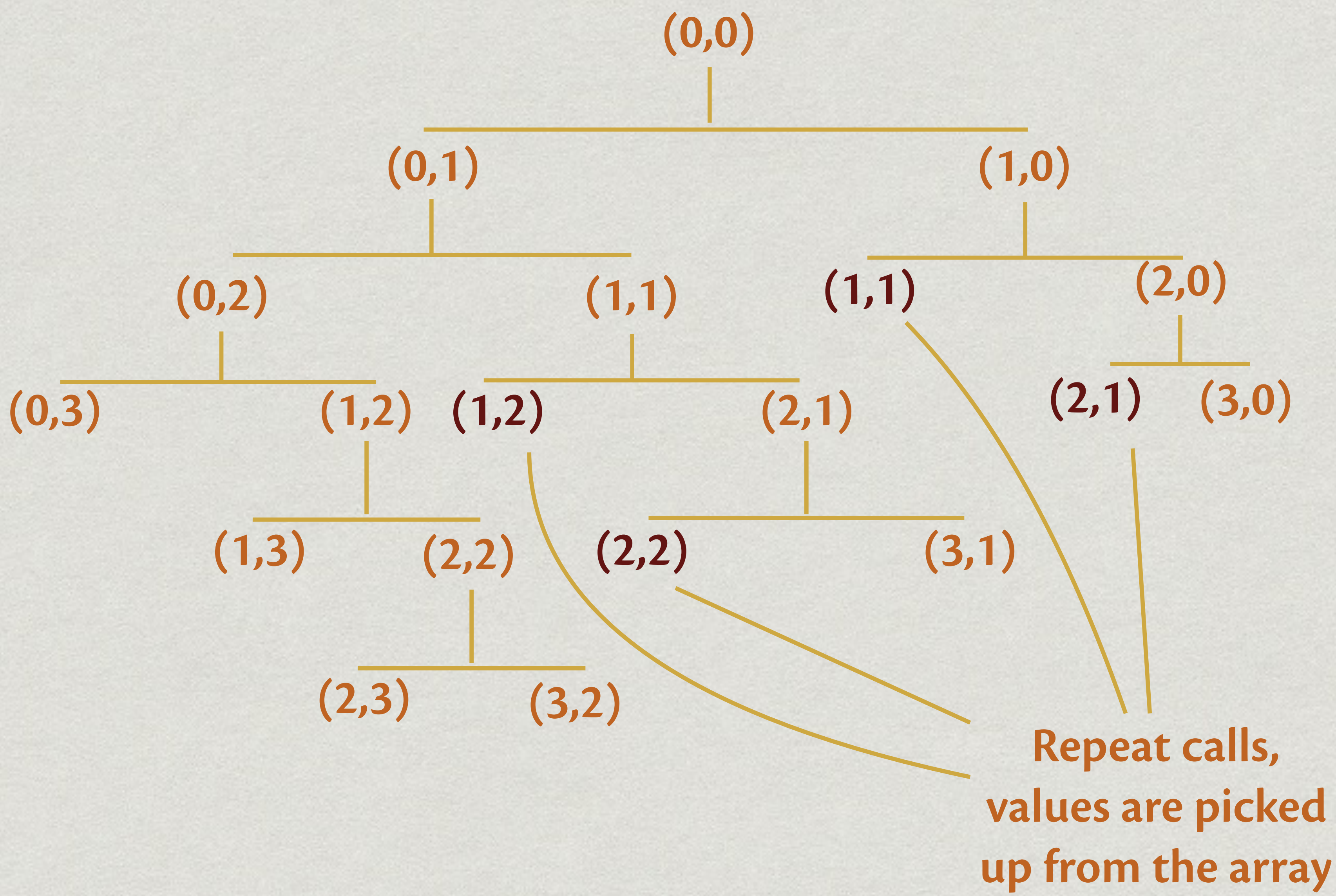
- ```
  lcss :: String -> String -> Int
  lcss str1 str2 = lcssA!(0,0)
   where
    m = length str1
    n = length str2
    ar1 = listArray (0,m-1) str1
    ar2 = listArray (0,n-1) str2
    lcssA = array ((0,0),(m,n))
                   [((i,j),f i j) | i <- [0..m],j <- [0..n] ]
    f i j
     | i >= m || j >= n   = 0
     | ar1!i == ar2!j     = 1 +  lcssA ! ((i+1),(j+1))
     | otherwise          = max (lcssA ! (i,(j+1)))
                                (lcssA ! ((i+1),j))
  ```

- This program runs in time O(mn)
  ```

lcss using arrays

- The first call to f i j stores the value in lcssA!(i,j)
- Subsequent calls with the same values of i and j return the value from the array
- **Memoization**: important technique in algorithm design

Call tree for m = n = 3



Creating arrays: accumArray

* ```
  accumArray
      :: Ix i
      => (e -> a -> e)   -- accumulating function
      -> e               -- initial entry (at each index)
      -> (i,i)           -- bounds of the array
      -> [(i,a)]         -- association list
      -> Array i e       -- array
  ```
  ```

Creating arrays: accumArray

- ```
  accumArray (*) 1 ('a','d')
         [('a',2),('b',3),('c',0),('a',2),('c',4)]
  array ('a','d') [('a',4),('b',3),('c',0),('d',1)]
  ```

- ```
  accumArray (+) 0 (1,3)
              [(1,1),(2,1),(2,1),(1,1),(3,1),(2,1)]
  array (1,3) [(1,2),(2,3),(3,1)]
  ```

- ```
  accumArray (flip (:)) [] (1,3)
              [(1,2),(2,3),(2,8),(1,6),(3,5),(2,4)]
  array (1,3) [(1,[6,2]),(2,[4,8,3]),(3,[5])]
  ```
  ```

Creating arrays: accumArray

- ```
  accumArray
      :: Ix i
      => (e -> a -> e) -> e -> (i,i) -> [(i,a)]
                  -> Array i e
  ```

- `accumArray f e (l,u) list` creates an array with indices `l..u`, in time proportional to `u-l`, provided `f` can be computed in constant time
  ```

Creating arrays: accumArray

- For a particular `i` between `l` and `u`, if `(i,a1)`, `(i,a2)`, ..., `(i,an)` are all the elements with index `i` appearing in `list`, the value for `i` in the array is `f (...(f (f e a1) a2)...) an`
- The entry at index `i` thus **accumulates** (using `f`) all the `ai` associated with `i` in `list`

Linear-time sort

- Given a list of n integers, each between 0 and 9999, sort the list
- Easy to do with arrays
- Count the number of occurrences of each j ∈ {0, ..., 9999} in the list, storing in an array counts
- Output count[j] copies of j, j ranging from 0 to 9999

Sorting with accumArray

* [2,3,4,1,2,5,7,8,1,3,1]

  ⇛ zip [2,3,4,1,2,5,7,8,1,3,1] [1,1,1,1,1,1,1,...]
  = [(2,1),(3,1),(4,1),(1,1),(2,1),(5,1),(7,1),(8,1),(1,1),(3,1),(1,1)]

  (repeat 1 = [1, 1, 1, 1,...])

  ⇛ array (1,8) [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)] – counts number of repetitions of each entry

Sorting with accumArray

- `array (1,8) [(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]`

  – counts number of repetitions of each entry

  ⇛ `[(1,3),(2,2),(3,2),(4,1),(5,1),(6,0),(7,1),(8,1)]`

  ⇛ `replicate 3 1 ++ replicate 2 2 ++ replicate 2 3 ++ replicate 1 4 ++ replicate 1 5 ++ replicate 0 6 ++ replicate 1 7 ++ replicate 1 8`
  = `[1,1,1]++[2,2]++[3,3]++[4]++[5]++[]++[7]++[8]`
  = `[1,1,1,2,2,3,3,4,5,7,8]`

Sorting with accumArray

* ```
  counts :: [Int] -> [(Int,Int)]
  counts xs = assocs (
                     accumArray (+) 0 (l,u) (zip xs ones)
                     )
      where
          ones = repeat 1
          l    = minimum xs
          u    = maximum xs
  ```

* ```
  arraysort :: [Int] -> [Int]
  arraysort xs = concat [replicate n i | (i,n) <- ys]
      where
          ys   = counts xs
  ```

Example: minout

- Assuming that all entries in `l` are distinct and non-negative numbers, find the minimum non-negative number not in `l`

- ```
  minout :: [Int] -> Int
  minout [3,1,2] = 0
  minout [1,5,3,0,2] = 4
  minout [11,5,3,0] = 1
  ```
  ```

Final example: minout

- ```
  minout :: [Int] -> Int
  minout = minoutAux 0
    where
      minoutAux :: Int -> [Int] -> Int
      minoutAux i l
        | i `elem` l     = minoutAux (i+1) l
        | otherwise      = i
  ```

- This program takes $O(N^2)$ time, where N is the `length l` **(Why?)**
  ```

Final example: minout

* ```haskell
  minout :: [Int] -> Int
  minout l = minout' 0 (sort l)
    where
      minout' n []       = n
      minout' n (x:xs)
        | n == x         = minout' (n+1) xs
        | otherwise      = n
  ```

* This program takes O(N logN) time to sort, and O(N) time for `minout'`, where N is the length of the list
  ```

minout using arrays

- We can use arrays for an O(N) solution, where N is the length of the list
- The minimum element outside the list l has to lie between 0 and N
- Select all elements from l that are ≤ N
- Count the number of occurrences of each in l in O(N) time (using **accumArray**)
- Pick the smallest number with count 0

minout using arrays

- ```
  minout :: [Int] -> Int
  minout l = search countlist
    where
      n         = length l
      ones      = repeat 1

      countlist :: [(Int,Int)]
      countlist = assocs (accumArray (+) 0 (0,n)
                         (zip (filter (<=n) l) ones))

      search             :: [(Int,Int)] -> Int
      search ((x,y):l) = if (y == 0) then x
                           else search l
  ```
  ```

Summary

- Recursive programs can sometimes be very inefficient, recomputing the same value again and again
- **Memoization** is a technique that renders this process efficient, by storing values the first time they are computed
- Haskell **arrays** provides an efficient implementation of these techniques
- Important tool to keep in our arsenal