Recursion and Induction: Examples of Programming with Lists

Greg Plaxton
Theory in Programming Practice, Fall 2005
Department of Computer Science
University of Texas at Austin
Mutual Recursion

- The function divide0 splits a given list of length $n$ into two lists of length $\left\lceil \frac{n}{2} \right\rceil$ and $\left\lfloor \frac{n}{2} \right\rfloor$ by alternately prepending elements of the input list into the two output lists.

- The function divide1 is similar except it starts by prepending to the second output list instead of the first.

\[
\begin{align*}
\text{divide0} \; [] &= ([], []) \\
\text{divide0} \; (x: xs) &= (x:f, s) \\
& \quad \text{where } (f, s) = \text{divide1} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{divide1} \; [] &= ([], []) \\
\text{divide1} \; (x: xs) &= (f, x:s) \\
& \quad \text{where } (f, s) = \text{divide0} \; xs
\end{align*}
\]
Mutual Recursion: Example

- Here are two sample executions of divide0

  Main> divide0 [1,2,3]
  ([1,3],[2])
  Main> divide0 [1,2,3,4]
  ([1,3],[2,4])
Appending an Element to a List

• Consider the following function for appending an element to a list

\[
\text{snoc } x \; [] = [x]
\]
\[
\text{snoc } x \; (y: \; xs) = y: (\text{snoc } x \; xs)
\]

• Characterize the asymptotic running time of \text{snoc}
**Concatenating Two Lists**

- Consider the following function for concatenating two lists
  
  \[
  \begin{align*}
  \text{conc} \; [] \; ys &= ys \\
  \text{conc} \; (x:xs) \; ys &= x : (\text{conc} \; xs \; ys)
  \end{align*}
  \]

- Characterize the asymptotic running time of `conc`

- There is a built-in operator that does the same job; `conc xs ys` is written as `xs ++ ys`
“Flattening” a List

• Function `flatten` takes a list of lists, like

  \[
  \begin{array}{c}
  \text{[ [1,2,3], [10,20], [], [30] ]}
  \end{array}
  \]

  and flattens it out by putting all the elements into a single list, like

  \[
  \begin{array}{c}
  \text{[1,2,3,10,20,30]}
  \end{array}
  \]

• Here is the definition of `flatten`

  \[
  \begin{array}{c}
  \text{flatten \[\]\, = \[\]}
  \\
  \text{flatten (xs : xss) = xs ++ (flatten xss)}
  \end{array}
  \]

• What is the type of `flatten`?
Reversing a List

- The following function reverses the order of the items in a list

\[
\text{rev \ } [] \quad = \quad [] \\
\text{rev \ } (x: \ xs) \quad = \quad (\text{rev} \ xs) \ +\ + \ [x]
\]

- Characterize the asymptotic running time of \text{rev}
Reversing a List: A More Efficient Algorithm

• We use the technique of function generalization as in the quickMlt example of an earlier lecture
  \[
  \text{reverse} \; [] \; ys = ys \\
  \text{reverse} \; (x:xs) \; ys = \text{reverse} \; xs \; (x:ys)
  \]

• Characterize the asymptotic running time of \text{reverse}

• How can we define the single-argument function \text{rev} in terms of \text{reverse}?
Towers of Hanoi

- Three pegs a, b, and c
- There is a stack of \( n \) disks of increasing size on peg a; the other two pegs are empty
- In one step, we can move the topmost disk of some stack to a different peg, provided that we never place a larger disk on top of a smaller disk
- Goal: Move the entire stack of \( n \) disks from peg a to peg b in the minimum number of moves
- Here is a 7-move solution for the case \( n = 3 \)
  
  \[
  [(1, 'a', 'b'), (2, 'a', 'c'), (1, 'b', 'c'), (3, 'a', 'b'), \\
  (1, 'c', 'a'), (2, 'c', 'b'), (1, 'a', 'b')]
  \]
Towers of Hanoi: Iterative Solution

- There is an iterative solution for this problem, which goes like this
  - Disk 1 moves in every alternate step starting with the first step
  - If $n$ is odd, disk 1 moves cyclically from 'a' to 'b' to 'c' to 'a' . . . , and if $n$ is even, disk 1 moves cyclically from 'a' to 'c' to 'b' to 'a' . . .
  - In each remaining step, there is exactly one possible move: ignore the stack of which disk 1 is the top; compare the tops of the two remaining stacks and move the smaller one to the top of the other stack (if one stack is empty, move the top of the other stack to its top)
- This is somewhat messy and difficult to prove correct; let’s look at an obviously correct recursive scheme instead
Towers of Hanoi: Recursive Solution

- Observation: There must be a step in which disk $n$ is moved from peg $a$ to one of the two remaining pegs, call it peg $x$, while all other disks are stacked (in increasing order) on the third peg.

- Furthermore, we can deduce that in an optimal solution, $x$ is equal to $b$.
  - By symmetry, the optimal cost of reaching this point is the same whether $x$ is $b$ or $c$.
  - If $x$ is $b$, the task that remains corresponds to an instance of the original problem of size $n - 1$.
  - If $x$ is $c$, the task that remains properly includes an instance of the original problem of size $n - 1$. 
Towers of Hanoi: Recursive Solution

• Here is our recursive solution

```haskell
tower n a b c
  | n == 0    = []
  | otherwise = (tower (n-1) a c b)
              ++ [(n,a,b)]
              ++ (tower (n-1) c b a)
```

• Here is a sample execution of the tower function

```haskell```
Main> tower 3 'a' 'b' 'c' [(1,'a','b'),(2,'a','c'),(1,'b','c'),(3,'a','b'), (1,'c','a'),(2,'c','b'),(1,'a','b')]
```

• What is the total number of moves, as a function of $n$?
Gray Code

- If you are asked to list all 3-bit strings, you will probably write them in increasing order of their magnitudes.

- It is possible to list these strings so that consecutive strings (the first and the last strings are also considered consecutive here) differ in exactly one bit position.

- Goal: Generate such a list for any given number of bits $n > 0$.
  - For $n = 0$, the list ["" ] is a Gray code.
  - For $n = 1$, the list ["0" "1"] is a Gray code.
  - For $n = 2$, the list ["00" "01" "11" "10"] is a Gray code.

- Recursive construction?
Gray Code: A Recursive Implementation

• First we define two useful auxiliary functions

\[
\text{cons0} \ [\ ] = [\ ] \\
\text{cons0} \ (x:xs) = (’0’:x):(\text{cons0} \ xs)
\]

\[
\text{cons1} \ [\ ] = [\ ] \\
\text{cons1} \ (x:xs) = (’1’:x):(\text{cons1} \ xs)
\]

• The desired function \(\text{gray}\) may be defined as follows

\[
\text{gray} \ 0 = [””] \\
\text{gray} \ (n+1) = ((\text{cons0} \ a) ++ (\text{rev} \ (\text{cons1} \ a))) \\
\text{where} \ a = \text{gray} \ n
\]

• Characterize the asymptotic running time of \(\text{gray}\)
Sorting

- Consider a list of items drawn from some totally ordered domain such as the integers
- We develop a number of algorithms for sorting such a list, that is, for producing a list in which the same set of numbers are arranged in nondecreasing order
Insertion Sort

- First, we define a function for performing a single insertion

  \[
  \begin{align*}
  \text{insert } y \; [] &= [y] \\
  \text{insert } y \; (z:zs) &= \begin{cases} \\
  y \leq z & \Rightarrow y:(z:zs) \\
  y > z & \Rightarrow z: (\text{insert } y \; zs) \\
  \end{cases}
  \end{align*}
  \]

- Now we define the sorting function

  \[
  \begin{align*}
  \text{isort } [] &= [] \\
  \text{isort } (x:xs) &= \text{insert } x \; (\text{isort } xs)
  \end{align*}
  \]

- Characterize the asymptotic running time of \text{isort}
Merge Sort

• First, we define a function for merging two sorted lists

  \[
  \text{merge } \text{xs} \ [\ ] = \text{xs} \\
  \text{merge } [\ ] \ \text{ys} = \text{ys} \\
  \text{merge } (x:xs) \ (y:ys) \\
  \hspace{1em} | \ x \leq y \quad = \ x : (\text{merge } xs \ (y:ys)) \\
  \hspace{1em} | \ x > y \quad = \ y : (\text{merge } (x:xs) \ ys) \\
  \]

• Now we define mergesort using divide0 for partitioning

  \[
  \text{mergesort } [\ ] = [\ ] \\
  \text{mergesort } [x] = [x] \\
  \text{mergesort } \text{xs} = \text{merge } \text{left} \ \text{right} \\
  \hspace{1em} \text{where} \\
  \hspace{2em} (\text{xsl},\text{xsr}) = \text{divide0 } \text{xs} \\
  \hspace{2em} \text{left} = \text{mergesort } \text{xsl} \\
  \hspace{2em} \text{right} = \text{mergesort } \text{xsr} \\
  \]

• Characterize the asymptotic running time of mergesort
Quicksort

- In quicksort, we partition the input list about the first (other choices are possible) and recursively sort each partition.

- We first define our partitioning function:

  ```haskell
  partition v [] = ([],[])
  partition v (x:xs)
  | x <= v = (x:ys),zs
  | x > v = (ys,(x:zs))
  where (ys,zs) = partition v xs
  ```

- Now we define the sorting function:

  ```haskell
  qsort [] = []
  qsort (x:xs) = (qsort ys) ++ [x] ++ (qsort zs)
  where (ys,zs) = partition x xs
  ```

- Characterize the asymptotic running time of `qsort`