More Lists

COMP 302

PART 05

Example – Generate [1, 2, ..., n]

```
fun countup_from1 (x:int) =
    let fun count (from:int, to:int) =
         if from=to
         then to::[] (* note: can also write [to] *)
         else from :: count(from+1,to)
    in
      count(1,x)
    end
```

This is poor style since functions can use any bindings if the environment of definition.

Alternative Version

```
fun countup_from1_better (x:int) =
    let fun count (from:int) =
         if from=x
         then x::[]
         else from :: count(from+1)
    in
     count 1
    end
```

Local variables can be used to keep code more readable

Reverse

Define a function to reverse a list:

```
reverse [2, 4, 6, 9, 11] = [11, 9, 6, 4, 2]
```

Using structural recursion we should have a pattern like

```
fun reverse (lst : int list) : int list =
  case lst of
       [] => []
     | x::xs => … (reverse xs) …
```

If we assume the recursive call works for our example it would give `[11, 9, 6, 4]`

To get the result we want, we have to append [2] to the end of the list

Reverse

```
fun append (lst1 : int list, lst2 : int list) : int list =
  case lst1 of
       [] => lst2
     | x::xs => x :: append (xs, lst2)
```

Append is actually a built-in function, @ so we didn't really have to define it

```
fun reverse (lst :: int list) : int list =
  case lst of
       [] => []
     | x::xs => (reverse xs) @ [x]
```

What is the complexity

For functions defined by structural recursion, we can usually characterize the complexity by a recurrence. We measure the cost by counting the number of steps in an evaluation.

Let A be the cost of append. Then

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

Solving this recurrence gives A(n) = O(n)

Let R be the cost of reverse. Then

```
R(0) = c0
R(n) = c1 + A(n-1) + R(n-1)
```

Solving this recurrence gives R(n) = O(n^2)

A Fast Reverse

Think of representing the list by a deck of cards in a pile

Just reverse them by adding them one by one to a second pile

This motivates an auxiliary function

```
fun reverse2 (lst1 : int list, lst2 : int list) : int list =
  case lst1 of
       [] => lst2
     | x::xs => reverse2 (xs, x::lst2)
```

Fast Reverse

```
fun reverse (lst: int list) : int list =
  let
    fun reverse2 (lst1 : int list, lst2 : int list) : int list =
      case lst1 of
           [] => lst2
         | x::xs => reverse2 (xs, x::lst2)
   in
     reverse2 (lst, [])
   end
```

The cost of reverse2, R2 is given by the recurrence

```
R2(0) = c0
R2(n) = c1 + R2(n-1)
```

Fast Reverse Cost

The cost of reverse2, R2 is given by the recurrence

```
R2(0) = c0
R2(n) = c1 + R2(n-1)
and
R(n) = c2 + R2(n)
```

This gives a cost of O(n)

We generalized the original problem by adding a second argument and were able to get a faster algorithm

As with many inductive arguments in Mathematics, we see that sometimes harder problems are easier

Mergesort

Mergesort is a well known O(n log n) sorting algorithm

It uses a divide and conquer methodology

Given a list of values, at each step

- divide the list in half
- recursively mergesort each half
- merge the two sorted lists back together

Our first task will be to divide the list into two piles of approximately equal size

Mergesort - Split

```
(* Deal the list into two piles *)
fun split (lst : int list) : int list * int list =
    case lst of
        [] => ([] , [])
      | [ x ] => ([ x ] , [])
      | x :: y :: xs => let val (pile1 , pile2) = split xs
                        in (x :: pile1 , y :: pile2)
                        end
```

Patterns

The general form of a case expression is

```
case e of
       p1 => e1
     | p2 => e2
     | …
```

where each p is a pattern.

The patterns must be consistent with the type of e.

Patterns

What are patterns?

pattern	type	match
x	any	Anything (binds to x)
_	any	Anything
[]	int list	[]
p1 :: p2	int list	A list v1::v2 where v1 matches p1 and v2 matches p2
(p1 , p2)	t1 * t2	A pair (v1, v2) where v1 matches p1 and v2 matches p2

Pattern Checking

ML performs exhaustiveness and redundancy checking

- If we omit a case, ML says that the pattern-match is non-exhaustive

```
case (x, y) of
      ([] , y) => y
   | (x :: xs , y :: ys) =>
```

- If we add an extra case, ML says that the pattern-match is redundant

```
case (x, y) of
      ([] , y) => y
   | (x , []) => x
   | ([] , []) => x
   | (x :: xs , y :: ys) =>
```

Merge

```
(* Purpose: merge two sorted lists into one *)
fun merge (lst1 : int list , lst2 : int list) : int list =
    case (lst1 , lst2) of
        ([] , lst2) => lst2
      | (lst1 , []) => lst1
      | (x :: xs , y :: ys) =>
            (case x < y of
                 true => x :: (merge (xs , lst2))
               | false => y :: (merge (lst1 , ys)))
```

Mergesort – First Try

```
(* Purpose: sort the list in O(n log n) work *)
(* This version is incorrect but gets us thinking on the right
   path *)
(* what is mergesort ([2, 1])?      *)
fun mergesort (lst : int list) : int list =
|  let val (pile1,pile2) = split lst
   in
       merge (mergesort pile1, mergesort pile2)
   end
```

Mergesort Corrected

```
(* Purpose: sort the list in O(n log n) work *)
fun mergesort (lst : int list) : int list =
    case lst of
        [] => []
      | [x] => [x]
      |  _ => let val (pile1,pile2) = split lst
              in
                 merge (mergesort pile1, mergesort pile2)
              end
```

Proving Correctness

To prove correctness, we need another variant of induction called strong (or complete) induction

The structure of the proof follows the structure of the code

We give a broad outline of how to go about the proof without giving details

Correctness

We want to prove that when mergesort is applied to any list of integers, the value produced is a sorted list.

To do this we first have to be precise about what sorted means

`lst sorted` that is defined as :

1. `[] sorted,`
2. `x :: xs sorted iff x is valuable, xs is sorted and for all values y in xs, x < y)`

Sorted lists are valuable so there is no substitution problem

Theorem: For all values lst : `int list, (mergesort lst) sorted`

Correctness of Mergesort

Theorem: For all values lst : int list, (mergesort lst) sorted

- Base case 1: mergesort [] == []
  - (2 computation steps) and [] is sorted
- Base case 2: mergesort [x] == [x] .
  - We argue that [x] is sorted because x is valuable, [] is sorted and x<y vacuously.

General Case

```
General case outline:

mergesort lst ==

let val (pile1,pile2) = split lst

                          in merge (mergesort p1, mergesort p2)end
```

we have to show

1. `split lst` returns two smaller lists p1 and p2
2. **IH tells us that the recursive calls are sorted FOR ALL SMALLER LISTS (strong induction)**
3. merge produces a sorted list when given two sorted lists