CSC/MAT-220: Discrete Structures

Lab 5: ML

Due October 25, 2017

Recursive Functions

In ML, the primary means of working out an iterative task is through the recursive function. A *recursive function* computes the result of a call by reducing the problem to smaller subproblems until the smaller problem can be solved trivially. The *recursive step* is the simplifying step used to make the problem smaller, and results in the function calling itself until the *base case* (smallest subproblem) has been reached.

In order for a function to call itself, it must have a name by which it can refer to itself. In ML, this is achieved by using a *recursive value binding*. Consider the example below.

```
val rec factorial : int→int =
    fn 0 => 1 | n:int => n*factorial(n-1)
```

As with non-recursive bindings the left hand side is a pattern, but in this case the right hand side is a function expressions, since only functions may be defined recursively in ML. As we saw in Lab 3, we can use the *fun* syntax to make this function declaration much more concise.

```
fun factorial(0) = 1
  | factorial(n:int) = n*factorial(n-1)
```

In general, recursive value bindings will have the following form.

```
val rec var : type = val
```

In order to be well-typed, the *type* of *val* must match the *tpe* of *var*. It follows that in recursive value bindings *var* will always have a function type. Let us now consider how applications of recursive functions are evaluated by noting the following example, where we are to evaluate *factorial 3*. We proceed by retrieving the binding of *factorial* and evaluating

```
(fn 0 => 1 | n:int => n*factorial(n-1))(3)
```

Considering each clause, we note that the first doesn't match the *val*, but the second does, since 3 is an integer. We therefore continue by evaluating the expression *n*factorial(n-1)*, where $n = 3$. We are left with the subproblem of evaluating the expression

```
3*(fn 0 => 1 | n:int => n*factorial(n-1))(2)
```

Using the same clausal reason as before, we can reduce this to the subproblem of evaluating the expression

$$3*2*(fn\ 0 => 1 \mid n{:}int => n*factorial(n\text{-}1))(1)$$

Which reduces to the subproblem of evaluating the expression

$$3*2*1*(fn\ 0 => 1 \mid n{:}int => n*factorial(n\text{-}1))(0)$$

Now that we have reduced our problem to the base case all that is left is to evaluate the expression

$$3*2*1*1$$

Note that the size of our subproblem grows until we reach the base case, at which point there are no more recursive calls and the computation can complete. The growth in the expression size corresponds directly to a growth in the runtime storage required to record the state of the pending computation. This can be a huge problem with recursive functions, fortunately this can often be resolved using *tail recursion*, where we perform calculations before making the recursive call.

$$\begin{array}{l} local \\ \quad val\ rec\ helper\ :\ int * int \to int\ = \\ \quad\quad fn\ (0, r : int)\ =>\ r \mid (n : int, r : int)\ =>\ helper(n-1, n*r) \\ in \\ \quad val\ factorial\ :\ int \to int\ = \\ \quad\quad fn\ n : int\ =>\ helper(n, 1) \\ end \end{array}$$

This time the evaluation of *factorial(3)* results in a call to *helper* with arguments (3,1) and results in the following evaluation trace

i. helper(3,1)

ii. helper(2,3)

iii. helper(1,6)

iv. helper(0,6)

v. 6

Note that there is no growth in the size of the expression because there are no pending computations to be resumed upon completion of the recursive call.

Mutual Recursion It is often useful to define two functions simultaneously, each of which depends on the other. As a simple example, consider the following functions *even* and *odd*.

$$
\begin{aligned}
&fun\ even\ 0\ =\ true\\
&|\ even\ n\ =\ odd(n-1)\\
&and\ odd\ 0\ =\ false\\
&|\ odd\ n\ =\ even(n-1)
\end{aligned}
$$

Assignment Please do each of the following problems and put your code/comments in a file named *yourname_lab5.sml*. In addition, include the following commented line at the top of the file

(Name, Lab #, Date *)*

i. Consider the following recursive function that computes the nth Fibonacci number:

$$
\begin{aligned}
&val\ rec\ fib1\ :\ int \rightarrow int\ =\\
&\quad fn\ 0\ =>\ 1\\
&\quad |\ 1\ =>\ 1\\
&\quad |\ n : int\ =>\ fib1(n-1) + fib1(n-2)
\end{aligned}
$$

Perform a cost analysis of this function, ignoring the cost of addition in the step *fib1(n-1)+fib2(n-2)*.

ii. Consider the following recursive function that computes the nth Fibonacci number:

```
local
    val rec helper : int → int * int
    fn 0 => (1,0)
    | 1 => (1,1)
    | n : int =>
        let
            val (a : int, b : int) = helper(n − 1)
        in
            (a + b, a)
        end
in
    val fib2 : int → int =
    fn n : int =>
        let
            val(a : int, _) = helper(n)
        in
            a
        end
end
```

Describe what the helper function is doing and perform a cost analysis. Then, provide a comparison between the cost analysis in (i) and (ii), which of the corresponding algorithms is faster?

iii. Write a recursive function for evaluating a nth degree polynomial whose coefficients are real numbers $a_0, a_1, \ldots, a_n$ at a real number x. Be sure to store the coefficients of a polynomial in a list, to take advantage of the SML List Structure and its member functions, and to use tail recursion. Provide a cost analysis of your algorithm, including the number of additions and multiplications performed.