CSC/MAT-220: Discrete Structures
Lab 7: ML

Due November 21, 2017

List Primitives. In Lab 3, we saw that aggregate data structures were easy to work with in ML. In this lab we study another important aggregate type, the *list* type. The values of type *typ list* are the finite lists of values of type *typ*. That is, the list type is an example of a type constructor, or parametrized type, where the type of the list reveals the type of its elements. Furthermore, we can define the values of type *typ list* recursively as follows:

i. *nil* is a value of type *typ list*,

ii. if h is a value of type *typ*, and t is a value of type *typ list*, then *h::t* is a value of type *typ list*,

iii. nothing else is a value of type *typ list*.

The null constructor *nil* denotes the empty list. The binary constructor *::* constructs a non-empty list from a value h of type *typ* and another value t of type *typ list*. The resulting list of type *typ list* has h as its head and t as its tail. In general, the value *val* of type *typ list* has the form

$$val_0 :: (val_1 :: \ (\cdots \ :: \ (val_{n-1} :: nil)))$$

for some $n \geq 1$, where val_i is a value of type *typ* for each $i = 0, 1, \ldots, n$. It is important to note that both *nil* and *::* are polymorphic in the type of the underlying elements of the list. Also, we may omit the paranthesis in the above general form and just write

$$val_0 :: val_1 :: \cdots :: val_{n-1} :: nil,$$

and this can be written in *list notation* as

$$[val_0, \ val_1, \ \cdots, \ val_{n-1}].$$

Computing with Lists. Since values of the list type are define recursively, it is natural that functions on lists be defined recursively. Below is code for a function *length* that computes the number of elements in a list.

$$\begin{aligned} &val\ rec\ length\ :\ 'a\ list\ \rightarrow\ int\ = \\ &\quad fn\ nil\ =>\ 0 \\ &\quad |\ (\_ :: t)\ =>\ 1 + length(t) \end{aligned}$$

The above recursive function has a base case of the empty list, *nil*, and returns 0. The recursive step is on the non-empty list *_::t*, we can ignore the head of the list since the definition is given in terms of the tail of the list t,

which is smaller than the list *_::t*. We can follow a similar pattern to define a function to *append* two lists.

```
val rec append : 'a list * 'a list -> 'a list =
    fn (nil, l) => nil
    | (h :: t, l) => h :: append(t, l)
```

The *append* function is built into ML, and it is written using infix notation as *l1 @ l2*. We will make use of this function for defining another function *rev1* for reversing the entries of a list.

```
val rec rev1 : 'a list -> 'a list =
    fn nil => nil
    | (h :: t) => rev(t) @ [h]
```

While this certainly seems the most natural way to define the reversal function, it is not the best. Below is a better implementation of this function.

```
local
    val rec helper : 'a list * 'a list -> 'a list =
        fn (nil, l) => l
        | (h :: t, l) => helper(t, h :: l)
in
    val rec rev2 : 'a list -> 'a list =
        fn l => helper(l, nil)
end
```

Note that the *helper* function is appending the head of the first argument to the second argument, rather than appending the entire tail of a list to its head, as is done in *rev1*.

Assignment. Please do each of the following problems and put your code/comments in a file named *yourname_lab7.sml*. In addition, include the following commented line at the top of the file

(Name, Lab #, Date *)*

i. Determine the running time of the *append* function in terms of the length of the first list. Use this information to determine the running time of the function *rev1*. You must show your work, by setting up and solving a recursive formula. Report your answers in big O notation.

ii. Determine the running time of the *rev2* function. You must show your work, by setting up and solving a recursive formula. How much faster is *rev2* than *rev1*?