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* (non-empty 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. More important, we can do this recursion on the list itself, rather than on the size of the list. Below is code for a function *length* that computes the number of elements in a list.

```
val rec length : 'a list -> int =
    fn nil => 0
  | (_ :: t) => 1 + length(t)
```

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.

$$\begin{aligned} &val\ rec\ append\ :\ 'a\ list * 'a\ list\ \rightarrow\ 'a\ list\ = \\ &\quad fn\ (nil, l)\ =>\ l \\ &\quad |\ (h :: t, l)\ =>\ h :: append(t, l) \end{aligned}$$

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.

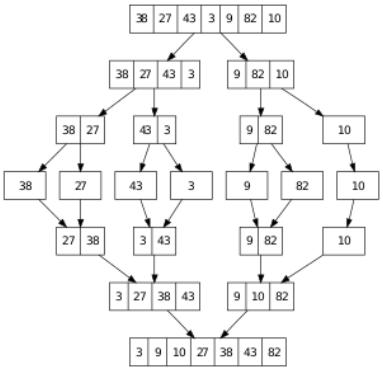
$$\begin{aligned} &val\ rec\ rev1\ :\ 'a\ list\ \rightarrow\ 'a\ list\ = \\ &\quad fn\ nil\ =>\ nil \\ &\quad |\ (h :: t)\ =>\ rev1(t)\ @\ [h] \end{aligned}$$

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.

$$\begin{aligned} &local \\ &\quad val\ rec\ helper\ :\ 'a\ list * 'a\ list\ \rightarrow\ 'a\ list\ = \\ &\quad\quad fn\ (nil, l)\ =>\ l \\ &\quad\quad |\ (h :: t, l)\ =>\ helper(t, h :: l) \\ &in \\ &\quad val\ rev2\ :\ 'a\ list\ \rightarrow\ 'a\ list\ = \\ &\quad\quad fn\ l\ =>\ helper(l, nil) \\ &end \end{aligned}$$

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*.

Merge Sort. Merge sort employs a divide and conquer (and combine) principle to sort a list in $O(n \log n)$ time. In essence, the algorithm recursively splits a list in half, sorts the smaller halves, and then merges them back together.



So, the first task we must be able to accomplish is the splitting of a list. To this end, we create the following function binding.

$$
\begin{array}{l}
val\ split\ :\ 'a\ list\ \rightarrow\ 'a\ list * 'a\ list\ = \\
\quad fn\ l\ => \\
\quad\quad let \\
\quad\quad\quad val\ n\ =\ (List.length(l))\ div\ 2 \\
\quad\quad in \\
\quad\quad\quad (List.take(l, n), List.drop(l, n)) \\
\quad\quad end
\end{array}
$$

It is essential to note that this is a polymorphic function expression with principle type scheme $'a\ list\ \rightarrow\ 'a\ list * 'a\ list$. Therefore, this expression can split a list of any type. Our list eventually gets split down into singleton lists. At this point, we need to be able to merge the smaller sublists while adhering to the overall intended order (non-decreasing). The function expression that accomplishes this task will need to make comparisons. We would like to use the $<$ operator, but its default use is on integers. It can be used on other types, but this is an instance of overloading, so additional context must be provided by the programmer. We will provide this context by including a function argument *less* that will define how these comparisons should be made. Below is the *merge* function binding.

$$
\begin{array}{l}
val\ rec\ merge\ :\ ('a * 'a\ \rightarrow\ bool) * 'a\ list * 'a\ list\ \rightarrow\ 'a\ list\ = \\
\quad fn\ (less, nil, m)\ =>\ m \\
\quad |\ (less, l, nil)\ =>\ l \\
\quad |\ (less, l\ as\ x :: xs, m\ as\ y :: ys)\ => \\
\quad\quad if\ less(x, y)\ then\ x :: merge(less, xs, m)\ else\ y :: merge(less, l, ys)
\end{array}
$$

Note how the lists l and m get referenced using *as* in terms of their head and tail. Furthermore, the function *merge* takes on another function argument *less*, this is how we will provide the context of how comparisons should be made. For instance, if we intended to use integer comparisons we could provide the following functional expression as an argument for *less*.

$$fn\ (x,y)\ =>\ x < y$$

We can now put all of this together to create our function *merge_sort*.

```
val rec merge_sort : ('a * 'a -> bool) * 'a list -> 'a list =
    fn (less, nil) => nil
    | (less, h :: nil) => [h]
    | (less, l) =>
        let
            val (a, b) = split(l)
        in
            merge(less, merge_sort(less, a), merge_sort(less, b))
        end
```

Binary Search. Binary search also employs a divide and conquer method on a sorted list in order to find a target value. Below I have provided a function expression bound to *int_search* that performs binary search on an ordered (non-decreasing) list of type *int list*.

```
local
    val rec helper : int * int * int * int list -> int =
        fn (s, e, t, nil) => ~1
        | (s, e, t, l) =>
            let
                val m = (s + e) div 2
                val n = (List.length(l) - 1) div 2
                val v = List.nth(l, n)
            in
                if t < v then helper(s, m - 1, t, List.take(l, n))
                else if t > v then helper(m + 1, e, t, List.drop(l, n + 1))
                else m
            end
in
    val int_search : int * int list -> int =
        fn (t, l) => helper(0, List.length(l) - 1, t, l)
end
```

Note that the variables s and e record the start and end indexes, respectively, to keep track of where in the original list you are currently searching. The variable m and n denote the current midpoint and size of the sublist, respectively. If $t < v$, then we proceed to search on the sublist defined by indexes $s = s$ and $e = m - 1$. If $t > v$, then we proceed to search on the sublist defined by the indexes $s = m + 1$ and $e = e$. Otherwise, $t = v$ and we're done.

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 answer 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, report your answer in big O notation. How much faster is *rev2* than *rev1*?

iii. Create a list of size 7 of type *int*, a list of size 8 of type *real*, a list of type *char* of size 9, and a list of type *string* of size 10. Be sure that they don't start out sorted. Then, sort these lists using *merge_sort*. You will have to provide the proper context for SML to determine how to make comparisons. See the hint provided on p. 3.

iv. Rewrite the function *int_search* so that the function is polymorphic. Name your new function *binary_search*. You will need to give SML the ability to determine the context of comparisons by giving two function arguments *less* and *great* to replace the operators $<$ and $>$, respectively.

v. Take the sorted lists from part iii. and use *binary_search* to perform a search on 4 target values, one for each list. Exactly once perform a search on a target value that is not in the given list. Again, you will have to provide the proper context for SML to determine how to make comparisons.