Lecture 10: Working with Lists

refer chapter 7 of notes

Abstract Data Type (ADT) α–LIST

$$\alpha\text{–}LIST = \alpha^* = \bigcup_{n=0}^{\infty} \alpha^n$$

The abstract data-type α–LIST, which a list of elements of α, is defined recursively as follows.

1. The empty list [] is an element of α–LIST.
2. α–LIST = α × α–LIST

Thus an instance of α–LIST is a finite sequence of a basic data-type i.e a member of α–LIST may be empty or may contain an arbitrarily long sequence of the elements of the set α.

Basic Operations on ADT α–LIST

1. $attach : \alpha \times \alpha\text{–}LIST \to \alpha\text{–}LIST$, which given an element from the set α and a list (which may be empty) attaches the element to the front of the list. For example, if $ls = [1, 2, 3]$, then $attach(0, ls)$ should return the list $ls = [0, 1, 2, 3]$, and $attach(0, [])$ should return the list $[0]$.

2. $empty : \alpha\text{–}LIST \to \{true, false\}$, which given an input list determines whether it is empty or not.

3. $head : \alpha\text{–}LIST^+ \to \alpha$, which given a non-empty list as its input returns the first element of the type α.

4. $tail : \alpha\text{–}LIST^+ \to \alpha\text{–}LIST$ which given a non-empty list as its input returns the sub-list without the first element. It returns the empty list if the input list has only one element.

SML-Lists

```
canonical list definition: item::(item::(item::….::nil))))

val guest_list = "Mom" :: ("Dad" :: ("Aunt" :: ("Uncle" :: [])))

val guest_list = "Mom" :: "Dad" :: [ "Aunt", "Uncle" ]
(* guest_list has the value [“Mom", “Dad", "Aunt", “Uncle”]  *)
```

SML fundamental List operations

`null l`

returns `true` if the list *l* is empty.

`hd l`

returns the first element of *l*. It raises `Empty` if *l* is `nil`.

`tl l`

returns all but the first element of *l*. It raises `Empty` if *l* is `nil`.

`cons ::`

Is Singleton

We can define the function $singleton : \alpha\text{–}LIST \rightarrow \{true, false\}$ as

$$singleton(ls) = \begin{cases} false & \text{if } empty(ls) \\ empty(tail(ls)) & \text{otherwise} \end{cases}$$

```
fun singleton(ls) =
        if null(ls) then false else null(tl(ls));
```

Length of a List

Example 7.3 *Computing the length of a list.*
We can define the function $length : \alpha\text{--}LIST \to \mathbb{N}$ recursively as

$$length(ls) = \begin{cases} 0 & \text{if } empty(ls) \\ 1 + length(tail(ls)) & \text{otherwise} \end{cases}$$

```
fun len(ls) =  if null(ls) then 0 else 1 + len(tl(ls));

fun len_it(len,l)=

        if null(l) then len else len_it(len+1,tl(l));

fun itlen(ls)=len_it(0,ls);
```

Append two lists (eq of @)

Example 7.4 *Appending two lists.*
The functiona $append : \alpha\text{–}LIST \times \alpha\text{–}LIST \to \alpha\text{–}LIST$ can be written as follows.

$$append(l1, l2) = \begin{cases} l2 & \text{if } empty(l1) \\ attach(head(l1), append(tail(l1), l2)) & \text{otherwise} \end{cases}$$

```
else   hd(l1)::append(tl(l1),l2);
```

Append Example

l1=[3,5,7,9]   l2=[4,2,1,0]

[3,5,7,9,4,2,1,0] —Result

(3 :: append([5,7,9]   l2=[4,2,1,0])   3:: [5,7,9,4,2,1,0]

(5 :: append([7,9]   l2=[4,2,1,0])   5:: [7,9,4,2,1,0]

(7 :: append([9]   l2=[4,2,1,0])   7:: [9,4,2,1,0]

(9:: append ([]   l2=[4,2,1,0])   9:: [4,2,1,0]

Analysis: Correctness & Complexity

The correctness of the function *append* can be established by **PMI**.
Correctness: To show that if $l1 = [l1_1\ l1_2 \ldots l1_n]$ and $l2 = [l2_1\ l2_2 \ldots l2_m]$, then $append(l1, l2)$ returns the list $[l1_1\ l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]$.

Proof: By induction on n (the length of $l1$).

Basis. $n = 0$ or $l1 = [])$. $append(l1, l2) = l2 = [l2_1\ l2_2 \ldots l2_m]$ by function definition.

Induction hypothesis. For all $0 \leq k \leq n$ such that $k = n - i + 1$ is the length of $l1 = [l1_i\ l1_2 \ldots l1_n]$, $append(l1, l2)$ returns the list $[l1_i\ l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]$.

Induction step. Consider $l1 = [l1_1\ l1_2 \ldots l1_n]$. We have that

$$
\begin{array}{rcll}
append(l1, l2) & = & attach(head(l1), append(tail(l1), l2)) & \text{by function definition} \\
& = & attach(l1_1, append(tail(l1), l2)) & \text{by definition of } head \\
& = & attach(l1_1, [l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m]) & \text{by induction hypothesis} \\
& = & [l1_1, l1_2 \ldots l1_n\ l2_1\ l2_2 \ldots l2_m] & \text{by definition of attach}
\end{array}
$$

$\square$

Exercise 7.3 Show that the time complexity of *append* is $O(n)$ where n is the size of $l1$. What is the space complexity?

Working with Lists

Find max (Largest) number in list

$$MAXM : \alpha\text{–}LIST^{+} \to \alpha$$

An algorithm for the function $MAXM$ can be specified as

$$MAXM(ls) = \begin{cases} head(ls) & \text{if } singleton?(ls) \\ max(head(ls), MAXM(tail(ls))) & \text{otherwise} \end{cases}$$

```
        val y=largest(xs);

    in if x > y then x else y

end;
```

Analysis: Correctness

Correctness: We can establish the correctness of the above functional description by demonstrating that

1. the number returned by Largest is and an element of the input list, and

2. it is the largest element of the list (note that there may be more than one occurrence of the largest value).

Analysis: Correctness

Base case. (n = 1) or singleton?. If the list has only one element then *largest*(ls) = head(ls) which is an element of the list and is trivially the largest.

Induction hypothesis. *Largest*(ls) returns the largest value in the list if the size of the list is (n – 1).

Induction step. Consider a list ls such that the size is (n > 1). Note that tail(ls) is a list of size (n – 1).

Now, *Largest*(ls) = max(head(ls), *Largest*(tail(ls))) = max(a,b) where a = head(ls) is an element of the list and b = *Largest*(tail(ls)) is the largest element in the sub-list tail(ls) by the induction hypothesis. By the definition of the binary function max, whose correctness is trivially established, Largest thus returns the largest element in the list ls.

The time complexity of the above algorithm is obviously O(n).

Higher order list functions: map and filter.

The higher order function map is of the type map : $(\alpha \rightarrow \alpha) \times \alpha\text{–LIST} \rightarrow \alpha\text{–LIST}$.

Given a function and a list as its input map returns the list formed by applying the input function on every element of the input list.

For example, if the input list is ls = [1, 2, 3, 4, 5]

then map(square,ls) should return the list [1,4,9,16,25]

map(cube,ls) should return the list [1, 8, 27, 64, 125].

```
fun map(foo,list)= if null(list) then [] else foo(hd(list))::map(foo,tl(list));
```

Higher order list functions: map and filter.

Example 7.5 The higher order function filter is of the type filter : $(\alpha \to \alpha) \times \alpha\text{–LIST} \to \alpha\text{–LIST}$.

It accepts a predicate (boolean function) of the input type α and a list as its input and returns a sub-list of those elements for which the predicate is true.

For example, if the input list is ls = [1, 2, 3, 4, 5]
filter(odd, ls) should return the list [1, 5] and
filter(prime, ls) should return the list [2, 3, 5].

```
fun filter(foo,list)= if null(list) then [] else if foo(hd(list))::filter(foo,tl(list));
```

map & filter example

val marks=[(“ch1200068",17.5),("ee1200452",12.2),("ch1200070",18.4)];

scale marks out of 10

fun scale(x,y)= (x,y/2.0);

test: scale(hd(marks)); => ("ch1200068", 8.75): *string * real*;

map(scale, marks); =>

[("ch1200068", 8.75), ("ee1200452", 6.1), ("ch1200070", 9.2)]: *(string * real)*

map & filter example

```
val marks=[(“ch1200068",17.5),("ee1200452",12.2),("ch1200070",18.4)];

fun filter(foo,list)= if null(list) then nil else
    if foo(hd(list)) then hd(list)::filter(foo,tl(list))
        else filter(foo,tl(list));
fun high(x,y)=y>=16.0;
filter(high,marks);
=> [("ch1200068", 17.5), ("ch1200070", 18.4)]
```