F28PL1 Programming Languages

Lecture 14: Standard ML 4

Polymorphic list operations

- length of list
- base case: `[] ==> 0`
- recursion case: `(h::t) =>` 1 more than length of `t`

```
- fun length [] = 0 |
      length (_::t) = 1+length t;
> val length = fn : 'a list -> int
- length ["a","b","c"];
> 3 : int
```

length ["a","b","c"] ==> 1+length ["b","c"] ==>

1+1+length ["c"] ==> 1+1+1+length [] ==> 1+1+1+0 ==> 3

Polymorphic list operations

- append lists
- e.g. append `[1,2,3] [4,5,6] ==> [1,2,3,4,5,6]`
- NB not `[1,2,3]::[4,5,6]`
- `::` wants `'a` and `'a list` not `'a list` and `'a list`
- recurse on 1st list
- base case: `[]` ==> 2nd list
- recursion case: `(h::t)` ==> put `h` on front of appending `t` to 2nd list

Polymorphic list operations

```
- fun append [] l2 = l2 |
      append (h1::t1) l2 =
       h1::append t1 l2;
> val append =
   fn : ‘a list -> ‘a list -> ‘a list
- append [“a”,”b”,”c”] [“d”,”e”,”f”];
> [“a”,”b”,”c”,”d”,”e”,”f”] : string list
```

Polymorphic list operations

```
- append [“a”,”b”,”c”] [“d”,”e”,”f”] ==>
“a”::append [”b”,”c”] [“d”,”e”,”f”] ==>
“a”::”b”::append [“c”] [“d”,”e”,”f”] ==>
“a”::”b”::”c”::append [] [“d”,”e”,”f”] ==>
“a”::”b”::”c”::[“d”,”e”,”f”] ==>
[“a”,”b”,”c”,”d”,”e”,”f”]
```

- @ - infix append operator

```
- [1,2,3]@[4,5,6];
> [1,2,3,4,5,6] : int list
```

Polymorphic list operations

- is value `e1` in list?
- base case: `[] ==> e1` not in list- `false`
- recursion case 1: `(e2::t) ==> e1=e2` - `true`
- recursion case 2: `(e2::t) ==> e1<>e2` - is `e1` in `t`

```
- fun member _ [] = false |
      member e1 (e2::t) =
       e1=e2 orelse member e1 t;
> val member =
   fn : ''a -> ''a list -> bool
```

Polymorphic list operations

```
- member 7 [1,9,7,4];
> true : bool


member 7 [1,9,7,4] ==>
member 7 [9,7,4] ==>
member 7 [7,4] ==>
true
```

Polymorphic list operations

- add value `e1` to list if not in list already
- base case: `[] ==>` make new list for `e`
- recursion case 1: `(e2::t) ==> e1=e2`
  - return `(e2::t)`
- recursion case 2: `(e2::t) ==> e1<>e2`
  - put `e2` back after adding `e1` to `t`
- will place new value at end of list

Polymorphic list operations

```
- fun add e [] = [e] |
      add e1 (e2::t) =
       if e1=e2
       then e2::t
       else e2::add e1 t;
> val add =
   fn : ''a -> ''a list -> ''a list
```

Polymorphic list operations

```
- add 1 [2,5,4];
> [2,5,4,1] : int list
add 1 [2,5,4] ==> 2::add 1 [5,4] ==>
2::5::add 1 [4] ==> 2::5::4::add 1[] ==>
2::5::4::[1] ==> [2,5,4,1]

- add 4 [2,5,4,1];
> [2,5,4,1] : int list
add 4 [2,5,4,1] ==> 2::add 4 [5,4,1] ==>
2::5::add 4 [4,1] ==> 2::5::[4,1] ==>
[2,5,4,1]
```

Polymorphic list operations

- delete value `e1` from list
- base case: `[] ==>` can't find `e1` so return empty list
- recursion case 1: `(e2::t) ==> e1=e2`
  - return `t`
- recursion case 2: `(e2::t) ==> e1<>e2`
  - put `e2` back after deleting `e1` from `t`

Polymorphic list operations

```
- fun delete _ [] = [] |
      delete e1 (e2::t) =
       if e1=e2
       then t
       else e2::delete e1 t;
> val delete =
   fn : ''a -> ''a list -> ''a list
```

Polymorphic list operations

```
-delete “c” [“a”,”b”,”c”,”d”];
> [“a”,”b”,”d”] : string list

delete “c” [“a”,”b”,”c”,”d”] ==>
“a”::delete “c” [“b”,”c”,”d”] ==>
“a”::”b”::delete “c” [“c”,”d”] ==>
“a”::”b”::[“d”] ==>
[“a”,”b”,”d”];
```

Higher order functions

- function which:
  1. takes another function as parameter

  or:

  3. returns a function as result
- natural in functional languages
- high degrees of:
  - abstraction
  - reuse
- polymorphic

Higher order function: filter

- often want to select those elements of a list for which some property holds
- *filter* list with predicate

```
- fun filter _ [] = [] |
     filter p (h::t) =
      if p h
      then h::filter p t
      else filter p t;
> val filter =
   fn : (‘a -> bool) -> ‘a list -> ‘a list
```

- if `p` holds for `h` then keep it

Higher order function: filter

```
- fun filter _ [] = [] |
     filter p (h::t) =
      if p h
      then h::filter p t
      else filter p t;
> val filter =
   fn : (‘a -> bool) -> ‘a list -> ‘a list
```

- p may be any `‘a -> bool` function
- `(h::t)` must be `‘a list`
- result must be `‘a list`

Higher order function: filter

- e.g. find all in a list of integer > 0

```
- fun isPos x = x>0;
> val isPos = fn : int -> bool
- filter isPos [~2,1,0,2];
> [1,2] : int list
filter isPos [~2,1,0,2] ==>
filter isPos [1,0,2] ==>
1::filter isPos [0,2] ==>
1::filter isPos [2] ==>
1::2::filter isPos [] ==>
1::2::[] ==> [1,2]
```

Higher order function: map

- often want to create a new list by doing the same thing to each element of an old list
- *map* function over list

```
- fun map _ [] = [] |
      map f (h::t) = f h::map f t;
> val map = fn : (‘a -> ‘b) ->
                 ‘a list -> ‘b list
```

- `f` may be any `‘a -> ‘b` function
- `(h::t)` must be a `‘a list`
- result must be a `‘b list`

Higher order function: map

- e.g. find list of sizes for string list

```
- map size [“a”,”bc”,”def”];
> [1,2,3] : int list
```

- `size: string -> int`
- `f : ‘a -> ‘b`
- so: `‘a == string; ‘b == int`

Higher order function: map

```
map size [“a”,”bc”,”def”] ==>

size “a”::map size [“bc”,”def”] ==>

size “a”::size “bc”::map size [“def”] ==>

size “a”::size ”bc”::size “def”::map size
[] ==>

size “a”::size ”bc”::size “def”::[] ==>

[1,2,3]
```

Higher order function: map

- e.g. find list of squares and cubes from integer list

```
- fun powers (x:int) = (x,x*x,x*x*x);
> val powers = fn : int -> int * int * int
- map powers [1,2,3];
> [(1,1,1),(2,4,8),(3,9,27) :
   (int * int * int) list
```

- `powers: int -> int * int * int`
- `f: 'a -> 'b`
- so: `'a == int`; `'b == int * int * int`

Higher order function: map

```
map powers [1,2,3] ==>
powers 1::map powers [2,3] ==>
powers 1::powers 2::map powers [3] ==>
powers 1::powers 2::powers 3::
 map powers [] ==>
powers 1::powers 2::powers 3::[] ==>
[(1,1,1),(2,4,8),(3,9,27)]
```

Insert

- to insert an integer `i1` into an ordered integer sequence in ascending order of first element
- base case: `[] ==> [i1]`
- recursion case 1: `(i2::t) ==> i1<i2`
  - put i1 on front of `(i2::t)`
- recursion case 2: `(i2::t) ==> i1>=i2`
  - put `i2` on front of inserting `i1` into `t`

Insert

```
- fun insert(i:int) [] = [i] |
      insert i1 (i2::t) =
       if i1<i2
       then i1::(i2,e2)::t
       else i2::insert i1 t;
> fn : int  ->  int  list -> int  list
- insert 7 [5,9];
> [5,7,9] :int list
insert 7 [5,9] ==>
5::insert 7 [9] ==>
5::7::[9] ==> [5,7,9]
```

Sort

- to sort a list of integers
  - insert head into sorted tail
- base case: `[] ==> []`
- recursion case: `(h::t) ==>` insert `h` into sorted `t`

```
- fun sort [] = [] |
      sort (h::t) = insert h (sort t);
> fn : int list -> int list
sort [7,9,5];
> [5,7,9] : int list
```

Sort

```
sort [7,9,5] ==>
insert 7 (sort [9,5]) ==>
insert 7 (insert 9 (sort [5])) ==> i
insert 7 (insert 9 (insert 5 (sort []))) ==>
insert 7 (insert 9 (insert 5 []) ) ==>
insert 7 (insert 9 [5])) ==>
insert 7 [5,9] ==>
[5,7,9]
```

Higher order function: foldr

- consider adding all elements of a list together:

```
- fun sum [] = 0 |
      sum (h::t) = h+sum t;
> val sum = fn: int list -> int
- sum [1,2,3];
> 6 : int
```

```
sum [1,2,3] ==> 1+sum [2,3] ==> 1+(2+sum [3]) ==>
1+(2+(3+sum [])) ==> 1+(2+(3+0)) ==> 1+2+3+0
```

- like doing + *between* elements of list

Higher order function: foldr

- consider doing `f` between elements of list
- *fold*
- base case: `[]` ==> return some base value `b`
- recursion case: `(h::t)` ==> apply `f` to `h` and result of folding `f` over `t`

```
- fun foldr f b [] = b |
      foldr f b (h::t) = f h (foldr f b t);
> val foldr =
   fn: (‘a->’b->’b) -> ‘b -> ‘a list -> ’b
```

Higher order function: foldr

e.g use `foldr` to join all elements of string list together

```
fun sJoin s1 s2 = s1^s2;

val sJoin = string -> string -> string;

foldr sJoin “” [“a”,”bc”,”def”];

“abcdef” : string
```

Higher order function: foldr

```
foldr sJoin “” [“a”,”bc”,”def”] ==>
sJoin “a” (foldr sJoin “” [“bc”,”def2]) ==>
sjoin “a”
 (sJoin “bc” (foldr sJoin “” [“def”])) ==>
sJoin “a”
 (sJoin “bc”
   (sJoin “def” (foldr sJoin “” []))) ==>
sJoin “a” (sJoin “bc” (sJoin “def” “”)) ==>
“abcdef”
```

Higher order function: foldr

- use `foldr` to make `sum`

```
- fun add (x:int) y = x+y;
> val add = fn: int -> int -> int
```

- do `add` between elements of list
- when list empty, return `0`

```
- val sum = foldr add 0;
> val sum = fn : int list -> int
```

- `sum` is like `foldr` with `f==add` and `b==0`

Higher order function: foldr

```
- sum [1,2,3];
> 6 : int
sum [1,2,3] ==>
foldr add 0 [1,2,3] ==>
add 1 (foldr add 0 [2,3]) ==>
add 1 (add 2 (foldr add 0 [3])) ==>
add 1 (add 2 (add 3 (foldr add 0 []))) ==>
add 1 (add 2 (add 3 0)) ==>
6
```

Higher order function: foldr

- use `foldr` to make `sort`
- do `insert` in between elements of list
- when list empty, return `[]`

```
- val sort = foldr insert [];
> val sort = fn : 'a list -> 'a list
- sort [3,2,1];
> [1,2,3] : int list
```

Higher order function: foldr

```
sort [3,2,1] ==>
foldr insert [] [3,2,1] ==>
insert 3 (foldr insert [] [2,1]) ==>
insert 3
 (insert 2 (foldr insert [] [1])) ==>
insert 3
 (insert 2
  (insert 1 (foldr insert [] []))) ==>
insert 3 (insert 2 (insert 1 [])) ==>
[1,2,3]
```

Higher order insert

- generalise `insert` to work with list of arbitrary type

```
- fun gInsert p v [] = [v] |
      gInsert p v (h::t) =
       if p v h
       then v::h::t
       else h::gInsert p v t
> val gInsert = fn : (‘a ->’a->bool)->
                     ’a -> ’a list -> ‘a list
```

- if `p` holds between `v` and `h` then put `v` on front of list
- otherwise put `h` on front of inserting `v` into `t` with `p`

Higher order insert

```
- fun iLess (x:int) y = x<y;
> val iLess = fn : int -> int -> bool
- val insert = gInsert iLess;
> val insert =
   fn : int -> int list -> int list
```

- `insert` is like `gInsert` with `p` set to `iLess`

Higher order sort

```
- fun gSort p [] = [] |
      gSort p (h::t) =
         gInsert p h (gSort p t);
> val gSort = fn : (‘a -> ‘a -> bool) ->
                   ‘a list ->’a list
```

- to sort a list with p, insert h with p into sorting t with p

```
- val sort = gSort iLess;
> val sort = fn : int list -> intlist
```

- sort is like gSort with p set to iLess

Higher order sort

```
- fun gSort p = foldr (gInsert p) [];
> val gSort = fn : (‘a -> ‘a -> bool) ->
                   ‘a -> ‘a list -> ‘a list
```

- sorting with p is like folding with inserting with p