FCS Notes

Kevalee Shah

April 20, 2019

Contents

1 ML Syntax

- Negatives are
- Not equal is $<>$
- equality check is $=$
- `and` is written as `andalso`
- `or` is written as `orelse`
- characters are #"C"
- `let D in E end`, where `D` is a `val` and/or `fun`

2 Recursive vs Iterative

Best shown by example Recursive:

```
fun nsum n =
   if n=0 then 0
   else n + nsum(n-1);
```

Iterative (tail-recursive):

```
fun summing (n,total) =
   if n=0 then total
   else summing(n-1, n + total);
```

- tail-recursion saves space
- iterative usually a loop

3 Big O Notation

Simple reccurence relationships:

- Linear: $T(n+1) = T(n) + 1 \qquad \mathcal{O}(n)$

  e.g.

  ```
  fun nsum n =
     if n=0 then 0 else n + nsum (n-1)
  ```

  We get $T(0) = 1$, $T(n+1) = T(n) + 1$
- Quadratic: $T(n+1) = T(n) + n \qquad \mathcal{O}(n^2)$
- Logarithmic: $T(n+1) = T(\frac{n}{2}) + 1 \qquad \mathcal{O}(\log n)$

  e.g. Power function: $T(2^n) = n + 1 \therefore T(n) = \mathcal{O}(\log n)$
- Quasi Linear: $2T(\frac{n}{2}) + n$

Example:
Show that $f(n) = \mathcal{O}(a_1 g_1(n) + \dots a_k g_k(n)) \implies f(n) = \mathcal{O}(g_1(n) \dots g_k(n))$

$$\begin{aligned}
&f(n) = \mathcal{O}(a_1 g_1(n) + \dots a_k g_k(n)) \\
&f(n) = \mathcal{O}(|a_1|\,|a_2| \dots |a_k|)(g_0(n) + \cdots + g_k(n)) \geq f(n) = \mathcal{O}(a_1 g_1(n) + \dots a_k g_k(n)) \\
&k = |a_1|\,|a_2| \dots |a_k| \\
\implies &f(n) = \mathcal{O}k(g_0(n) + \cdots + g_k(n)) \\
\implies &f(n) = \mathcal{O}(g_1(n) \dots g_k(n))
\end{aligned}$$

4 Lists

- Can be of any type but all elements must be of the same type
- Reversing a list in $\mathcal{O}(n)$ - uses consing which is just $\mathcal{O}(1)$

```
fun rev ([], ys) = ys
  | rev (x::xs, ys) = rev (xs, x::ys)

fn: 'a list * a' list -> a' list
```

5 Sorting

We usually determine the efficiency of the sorting algorithm based on the number of comparisons made $C(n)$
For n elements there are $n!$ permutations
Each comparison eliminates half the permutations, $\therefore 2^{C(n)} \geq n!$
$\implies C(n) \geq \log(n!) \approx n\log(n) - 1.44n$

5.1 Insertion Sort

```
fun ins (x:real, []) = [x]
  | ins (x:real, y::ys) =
      if   x<=y then x::y::ys
      else y :: ins(x, ys)

(* on average n/2 comparisons)

fun insort []     = []
  | insort (x::xs) = ins(x, insort xs);

(* n * n/2 operations therefore O(n^2)*)
```

5.2 Quicksort

Choose a pivot point and then partition the list if the number is greater than the pivot or not. Then sort the sublists in the same way. Append them all together at the end.

Average case: $\mathcal{O}(n \log n)$ Worst case: when the given list is reversed or nearly sorted. Then each partition is very uneven. $\mathcal{O}(n^2)$

5.3 Merge Sort

Merge sort - keep on dividing the list into two, until you get to individual elements and then merge lists in a way that results in an ordered list.
Worst case is $\mathcal{O}(n \log n)$ and therefore a very good algorithm, but slower than quicksort

6 Constructors

Can create our own datatypes: e.g.

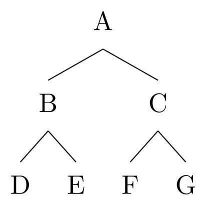
```
datatype vehicle = Bike
                 | Motorcycle of int
                 | Car        of bool
                 | Lorry      of int

(* the of type represents the argument of the constructors *)
```

7 Trees

```
datatype 'a tree = Lf
                 | Br of a' * a' tree * a' tree
```

- Binary search trees: Each branch carries a *(key, value)* pair, left subtree holds smaller keys and right holds greater keys. If the tree is balanced, then $\mathcal{O}(\log n)$ lookup time
- number of branch nodes $\leq \quad 2^{depth} - 1$
- Ways to traverse a tree:



  Preorder: ABDECFG

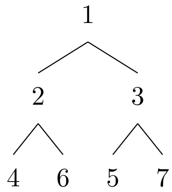
  Inorder: DBEAFCG

  Postorder: DEBFGCA

7.1 Arrays

- updated in place
- functional array is a map from integers to data

- can be represented by a binary tree - with evens to the left and odds to the right



- The number doesn't represent the value, but the position.
- Array access is often $\mathcal{O}(1)$ and always $\mathcal{O}(\log n)$

```
exception Subscript;

fun lookup (Lf, _) = raise Subscript
  | lookup (Br(v, t1, t2), k) =
          if k=1 then v
          else if k mod 2 = 0
               then lookup(t1, k div 2)
               esle lookup(t2, k div 2);
```

8 Functions

- Functions without names:
  e.g.

```
(fn x => x*2) 5;
> val it = 10 : int
```

- Curried functions return another function as its result

    e.g.

```
fun prefix = (fn a => (fn b => a^b));
> val prefix = fn: string -> (string -> string)

(* prefix gives a function of type string -> string *)

fun promote = prefix "Miss"
> val promote = fn: string -> string
```

Shortform for the above is

```
fn prefix a b = a^b;
dub = prefix "Miss ";

(* curried functions allow partial applications of the first
    argument *)
```

- What is the type of `fun S x y z = x z (y z)` type(z) = ‘a

type(y) = ‘b

type(x) = ‘c

input-type(y) = ‘a

input-type(x) = ‘a -¿ ‘b

type(y) = ‘a - ‘b

type(z) = ‘a

type(x) = ‘a -¿ ‘b -¿ ‘c

```
fun S x y z = x z (y z);
fn: ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

9 Lazy Lists

- Elements computed upon demand
- avoids waste if not many solutions
- can be used for infinite lists
- delay computation of the tail
- delayed version of E is $fn() => E$

```
datatype 'a seq = Nil
              | Cons of 'a * (unit -> 'a seq)
```

- e.g

```
fun from k = Cons(k, fn() => from(k + 1));
```

- Ways of joining two streams:

  Append

```
fun appendq (Nil, yq)      = yq
  | appendq (Cons(x, xf), yq) = Cons(x, fn() => appendq(xf(), yq));

(* may never get to yq if xf is infinite *)
```

Interleaving

```
fun inter (Nil, yq)      = yq
  | inter (Cons(x, xf), yq) = Cons(x, fn() => inter(yq, yqfx()));

(* switching the arguments ensures that both lists are included *)
```

10 Tree Traversal

10.1 Queues

Represents a sequence where items taken from the head, and added to the tail.
To ensure that access is $\mathcal{O}(1)$ when amortized we represent the queue $x_1, x_2, \ldots, x_m, y_n, \ldots, y_1$ as:

$$([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_n])$$

```
datatype 'a queue = Q of 'a list * 'a list

fun norm (Q([], tls)) = Q(rev tls, [])
  | norm q           = q;

(* to make sure the front list is never empty *)
```

Examples of functions:

- qempty
- qnull
- qhd
- enq
- deq

10.2 Breadth-first Tree Traversal

This is made easier using queues:

```
fun breadth q =
    if qnull q then []
    else
    case qhd q of
         Lf        => breadth (deq q)
         Br(v,t,u) => v::breadth(enq(enq(deq q, t), u));
```

Breadth first search examines $\mathcal{O}(b^d)$ nodes, where b is the branching factor and d is the depth

10.3 Iterative Deepening

Combines the space-efficiency of depth first and the 'nearest-first' of the breadth first.
Performs repeated depth-first searches with increasing depth bounds, each time discarding the results of the previous search.
Complexity still $\mathcal{O}(b^d)$, but space complexity is $\mathcal{O}(d)$

10.4 Stacks

Another datatype where items can be added or removed from the head only.
Obeys a —*Last-In-First-Out* discipline
Difference to lists: push or a pop changes the existing stack, doesn't return a new one
Functions include:

- `empty`
- `null`
- `top`
- `pop`
- `push`

10.5 Search Methods Summary

- Depth-first: use a stack
- Breadth-first: use a queue
- Iterative deepening: use depth for benefits of breadth

11 Polynomials

To represent $a_n x^n + \cdots + a_0 x^0$ we can use a list of tuples: $[(n, a_n), \ldots, (0, a_0)]$
Exaples of functions:

- `poly`
- `makepoly` - list to poly
- `destpoly` - poly to list
- `polysum`
- `polyprod`
- `polyquorem`

Can do polynomial multiplication in the traditional way of cross multiplication, but there is a better algorithm, inspired by merge sort which requires fewer merges

```
fun termprod (m,a) (n,b) = (m+n, a*b) : (int * real);

fun polyprod [] us    = []
  | polyprod [(m,a)] us = map (termprod(m,a)) us
  | polyprod ts us    =
          let val k = length ts div 2
          in polysum (polyprod (take(ts, k)) us)
                 (polyprod (drop(ts, k)) us)
          end;
```

12 Code Examples

•

```
fun npower(x, n) : real = \\ type constraint
   if n=0
   then 1.0
   else x * npower(x, n-1);
```

• membership test

```
fun member (x, []) = false
  | member(x, y::l) = (x=y) orelse member(x, l)
(* the orelse means that if (x=y) is true, then dont need to
    compute member(x,l) *)
```

• Zipping and unzipping

```
fun zip (x::xs, y::ys) = (x,y) :: zip(xs, ys)
  | zip _            = [];

fun unzip ([], xs, ys)        = (xs, ys)
  | unzip ((x,y)::pairs, xs, ys) = unzip(pairs, x::xs, y::ys);
```

• Using the `case` expression

```
fun wheels v =
    case v of Bike  => 2
        | Motorbike => 2
        | Car       => 4
        | Lorry w   => w
```

13 Procedural Programs

Change the *machine state* - updating variables/ arrays/ sending/ receiving data
Use control structures - branching, iteration, procedures
Data abstractions of the computer's memory

- Reference - to a memory cells
- Arrays - block of memory cells
- Linked Lists

ML makes no distinction between commands and expressions

13.1 References

- τref - type of references to type τ
- $ref\ E$ - creates a ref, new allocation in memory, initally holding value E

- $!P$ - returns the current contents of reference P
- $P := E$ - update the contents of P with E

  $C_1; C_2; \ldots; C_n$ - a series of expressions are evaluated and C_n is returned

13.2 while

Can be used as following:

```
fun length xs =
   let val lp = ref xs
      val np = ref 0
   in
      while not (null (!lp)) do
              (lp := taill (!lp); np := 1 + !np);
              !np
   end;
```

13.3 Arrays

- τ $Array.array$
- $Array.tabulate(n, f)$
- $Array.sub(A, f)$
- $Array.update(A, i, E)$

13.4 Linked List

```
datatype of mlist = Nil
                | Cons of 'a * 'a mlist ref
```