CS223, 4/28/2010
Lecture 13

Imperative features of SML -- ref and array types

1. References and Arrays -- mutable state in SML
[See Paulson, Chapter 8, p313.]

References:

```
   type 'a ref

   val ref : 'a -> 'a ref
   val ! : 'a ref -> 'a
   val := : 'a ref * 'a -> unit
```

ref counts as a constructor and can be used in patterns

```
   fun inc (r as ref n) = r := n+1
```

Imperative version of factorial:

```
   fun fact n =
       let val c = ref n
           val r = ref 1
        in while !c > 0 do
             (r := !r * !c; c := !c - 1);
           !r
       end;
```

Imperative version of summing a list:

```
   fun sum l =
       let val s = ref 0
           val m = ref l
        in while not(null(!m)) do
             (s := !s + hd(!m); m := tl(!m));
           !s
       end;
```

or, a semi-imperative version:

```
  fun sum l =
```

```
    let val s = ref 0
        fun loop nil = ()
          | loop (x::xs) = (s := !s + x; loop xs)
     in loop l;
        !s
    end;
```

Mutable lists (like scheme):

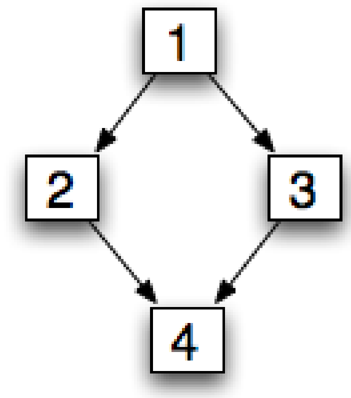
```
datatype 'a mlist = NIL | CONS of 'a ref * 'a mlist ref

val l = CONS(ref 3, CONS(ref 4, ref(NIL)))

fun cons (x, l) = CONS(ref x, ref l)
val l = cons(3, cons(4, NIL))

fun gethd NIL = raise Empty
  | gethd (CONS(ref x,_)) = x

fun sethd (NIL, x) = raise Empty
  | sethd (CONS(r,_), x) = r := x

fun gettl NIL = raise Empty
  | gettl (CONS(_, ref x)) = x

fun settl (NIL, x) = raise Empty
  | settl (CONS(_,r), x) = r := x

fun lastCons NIL = raise Empty
  | lastCons (l as CONS(_,ref NIL)) = l
  | lastCons (CONS(_,ref l)) = lastCons l

fun mappend(NIL, l) = l
  | mappend(l1, l2) = settl(lastCons l1, l2)
```

Graphs:

It is not hard to represent acyclic directed graphs in SML using datatypes:

```
datatype 'a graph = Node of 'a * 'a graph list
```

Then the graph



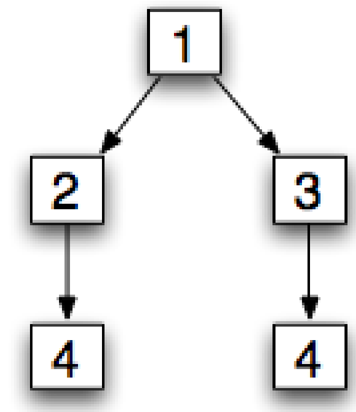
could be represented by:

```
  val g = let val n4 = Node(4,[])
              val n2 = Node(2,[n4])
              val n3 = Node(3,[n4])
           in  Node(1, [n2,n3])
          end
```

But could we tell the difference between this value
and the following?

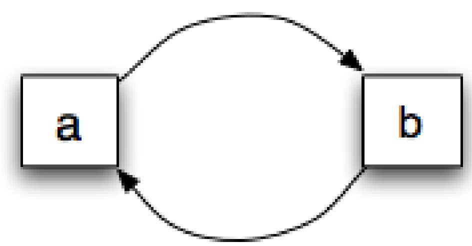
```
  val g' = let val n2 = Node(2,[Node(4,[])])
           val n3 = Node(3,[Node(4,[])])
        in  Node(1, [n2,n3])
        end
```

which represents the "tree" graph:



The difference only matters if the Node has
state, e.g. Node(ref(4),[]).

How about cyclic graphs, like



We need references to tie the knot in the
data structure.

```
   datatype 'a node
     = Node of 'a * 'a graph list ref


   val g = let val asucc = ref []
               val anode = Node("a", asucc)
               val bnode = Node("b", ref [anode])
            in asucc := [bnode];
               anode
           end;
```

Note: In Haskell, we could do something like

```
  data Graph a = Node a [Graph a]

  anode = Node "a" [bnode]
  bnode = Node "b" [anode]
```

This works for simple, fixed shape graphs,
but it doesn't scale to more complicated
graphs that need to be "computed".

More complicated graphs would involve two types,
nodes and edges, which both might contain values
(e.g. labels or weights or costs for edges).

Unification:

Terms constructed out of variables and function symbols
(each with its arty). Here is a sample term "language".

```
   (* infinite collection of variables *)
   type variable = string

   datatype term
     = V of variable   (* variables as terms *)
     | A | B | C       (* constants (0-ary functions) *)
     | F of term | G of term  (* some unary functions *)
     | H of term * term       (* a binary function *)
```

Given two terms, say

```
    H(F(V "x"), G(V "y"))  =?=  H(F(G A), V "z")
```

can we find a substitution (mapping from variables to
terms) making the terms equal?

```
   type subst = (string * term) list
```

In the example, the substitution

```
   val s = [("x", G A), ("z", G(V "y"))]
```

will work:

```
app(s, term1) = app(s, term2) = H(F(G A), G(V "y"))
```

where

```
val term1 = H(F(V "x"), G(V "y"))
val term2 = H(F(G A), V "z")
```

Here is a unification algorithm:
We assume auxiliary functions:

```
val occurs : variable * term -> bool
val appsub : subst * term -> term
val compose : subst * subst -> subst


(* unify : term * term -> subst option *)
fun unify (A,A) = SOME []
  | unify (B,B) = SOME []
  | unify (C,C) = SOME []
  | unify (V s1, V s2) =
     if s1 = s2 then SOME []
     else SOME[(s1, V s2)]
  | unify ((V s, t) | (t, V s)) =    (* OR pattern! *)
     if occurs(s,t) then NONE
     else SOME [(s,t)]
  | unify (F t1, F t2) = unify (t1,t2)
  | unify (G t1, G t2) = unify (t1,t2)
  | unify (H(t1,t2), H(t3,t4)) =
     (case unify(t1,t3)
        of SOME sub1 =>
           (case unify(app(sub1,t2),app(sub1,t4))
              of SOME sub2 => SOME(compose(sub2,sub1))
               | NONE => NONE)
         | NONE => NONE)
  | unify _ = NONE (* e.g. head function symbols don't match *)
```

This involves a lot of application of substitutions and
composition of substitutions. We can simplify unification
and make it more efficient by representing variables using
refs:

```
   (* note that varstate and term are mutually recursive
datatypes,
    * where term depends on varstate through the associated type
    * abbreviation variable *)

   datatype varstate
     = OPEN
     | INST of term

   and term
     = V of variable   (* infinite collection of variables *)
     | A | B | C       (* constants (0-ary function symbols) *)
     | F of term | G of term  (* some unary functions *)
     | H of term * term       (* a binary function *)

   withtype variable = varstate ref

   val x = ref OPEN
   val y = ref OPEN
   val z = ref OPEN
   val term1 = H(F(V x), G(V y))
   val term2 = H(F(G A), V z)
```

Now the unification function looks like:

```
   (* unify : term * term -> bool *)
   fun unify (A,A) = true
     | unify (B,B) = true
     | unify (C,C) = true
     | unify (V (v1 as ref(OPEN)), V (v2 as ref(OPEN))) =
        if v1 = v2 then true
        else (v1 := INST(V v2); true)
     | unify (V v1, V v2) =
        unify(varToTerm v1, varToTerm v2)
     | unify ((V v, t) | (t, V v)) =    (* OR pattern! *)
        (case varToTerm v
           of V(ref(OPEN)) =>
              if occurs(v,t) then false
              else (v := INST t; true)
            | t' => unify (t',t))
     | unify (F t1, F t2) = unify (t1,t2)
```

```
     | unify (G t1, G t2) = unify (t1,t2)
     | unify (H(t1,t2), H(t3,t4)) =
        unify(t1,t3) andalso unify(t2,t4)
     | unify _ = false
```

where varToTerm maps an instantiated variable to
the term that instantiated it:

```
   fun varToTerm(ref(INST t)) =
       (case t
          of V v => varToTerm v
           | _ => t)
     | varToTerm(v as (ref OPEN)) = V v
```

[See the source files unify1.sml and unify2.sml for complete
tested code for the unification example.]

Arrays:

Module Array:

```
   type 'a array
   val array : int * 'a -> 'a array
   val sub : 'a array * int -> 'a
   val update: 'a array * int * 'a -> unit
```

See Array module spec in Basis Library documentation.

Hash tables: