Tutorial 7

Week of March 5, 2007

1 Variant Types

```
(* A new type: dollars, which is either US dollars or Canadian dollars. *)
datatype dollars = USD of real |
                   CAD of real;

(* euro = fn: dollar -> real
 * return the equivalent in EURO *)
fun euro (CAD x) = 0.75 * x
|   euro (USD x) = 0.85 * x;

(* A new type: account.
    chequing: amount, interest rate, service charges per year
    savings:  amount, interest rate
    invest:   amount, interest rate, minimum balance *)
datatype account = chequing of  dollars*real*dollars |
                   savings of   dollars*real         |
                   invest of    dollars*real*dollars;

(* calculate = fn : account -> real
 * return the bank balance in euro after 1 year. *)
fun calculate (chequing (amt, rate, charge)) =
     (1.0+rate)*euro(amt) - euro(charge)

|   calculate (savings  (amt, rate)) =
     (1.0+rate)*euro(amt)

|   calculate (invest   (amt, rate, min)) =
         if euro(amt) < euro(min) then euro(amt)
         else (1.0+rate)*euro(amt);

calculate(chequing (100.0, 0.1, 5));
- Error: operator and operand don't agree [tycon mismatch]

calculate(chequing (dollars 100.0, 0.1, dollars 5));
- Error: unbound variable or constructor: dollars

calculate(chequing (CAD 100.0, 0.1, CAD 5.0));
val it = 68.25 : real

calculate(invest (CAD 100.0, 0.25, USD 50.0));
val it = 81.25 : real
```

Note that the rest of these tutorial notes are lecture slides that Sheila was unable to cover in Wednesday's lecture.

Recursive Types

A datatype can be recursive: e.g. **linked list**.

```
1  -datatype llist= Nil | Node of int*llist;
   datatype llist = Nil | Node of int*llist

2  -val x = Nil;
   val x=Nil: ??

3  -val y = Node (5, Nil);
   ??
4  -val z = Node(3, Node(2,Node(1,Nil)));
   ??
 (*computing the length of a linked list*)
5  -fun len Nil =0
6       |len(Node(_,rest))= 1 + len rest;
  val len = fn : ??

7  -len z;
   ??
```

Recursive Types (continue...)

Example: a *polymorphic* linked list

```
1 -datatype 'a llist= Nil|Node of 'a*('a llist);

2  -val x = Nil;
   val x=Nil: ??

3  -val y = Node (5, Nil);
   val y = Node (5,Nil) : ??

4  -val z = Node("Test", Node("B",Nil));
     ???
```

A binary tree where only leaves have data:

```
6  -datatype 'a tree= L of 'a
                    | N of ('a tree)*('a tree);
7  -val mytree= N(L(1),N(L(2),L(3)));

8  -fun max (x,y)= if x>y then x else y;
9  -fun depth(L _)=0
10     |depth(N(ltree,rtree))=
            1+max (depth ltree, depth rtree);
```

Mutual Recursive Types

Want to represent a tree with arbitrary #of branches.

See the diagram first ...

Defining mutually recursive datatypes (using **and**).

```
1 -datatype tree = Empty| Node of int*forest
2       and forest= Nil  | Cons of  tree*forest
  datatype tree = Empty | Node of int * forest
  datatype forest = Cons of tree * forest | Nil

3 -val t1=Node(2,Nil);
  ??
4 -val t2=Node(3,Nil);
  ??
5 -val t3=Node(7,Cons(t1,Cons(t2,Nil)));
  ??
6 -val t4=Node(5,Nil);
  ??
7 -val t5=Node(1,Nil);
  ??
8 -val t6=Node(2,Cons(t5,Cons(t4,Cons(t3,Nil))));
  ??
```

Mutual Recursive Types: function example...

We want to count how many nodes are in a tree.

solution: 1+ #of nodes in its subtrees (i.e. forest)

```
1 -fun numnodeT (Empty)=0
2     | numnodeT (Node(data,f))= 1+ numnodeF(f)
3   and
4       numnodeF(Nil) = 0
5      |numnodeF(Cons(t,f))= ???


   val numnodeT = fn : tree -> int
   val numnodeF = fn : forest -> int

(* Note that numnodeT and numnodeF are
   mutually recursive.*)

6  -numnodeT(t6)
   ??
```