Department of Computer Science and Engineering | CSE 230
University of California at San Diego | Winter 1998

Problem Set 4 Solutions

1. Define your own list append function in ML, and use it in the obvious function to flatten binary trees. What is the time complexity of this function?

   Define a less obvious function that flattens binary trees in linear time. Use structural induction to prove your two functions equivalent.

   [Solution due to Giovanni Di Crescenzo.] From before, the type for binary trees is

```
datatype btree = empty | leaf of int | node of btree * btree;
```

   and the append function is

```
fun myappend (nil,list) = list
  | myappend (h :: tail,list) = h :: myappend(tail,list);
```

   The obvious function to flatten binary trees is

```
fun flatten1(empty) = nil
  | flatten1(leaf x) = [x]
  | flatten1(node(l,r)) = myappend(flatten1(l),flatten1(r));
```

   A function that flattens binary trees in linear time is

```
fun fl2(empty,fringe) = fringe
  | fl2(leaf x,fringe) = x :: fringe
  | fl2(node(l,r),fringe) = fl2(l,fl2(r,fringe));
fun flatten2(T) = fl2(T,nil);
```

   We have to prove that th `flatten1(T) = flatten2(T)` for each binary tree `T` of type `btree`. As `fl2(T,nil) = flatten2(T)` for each `T` of type `btree` and `myappend(L,nil) = L` for each `L` of type `int list`, we can reduce to proving that for each `T` of type `btree` it holds that for each `L` of type `int list`, `myappend(flatten1(T),L) = fl2(T,L)`. This statement can be proved by structural induction on `T`.

   For the case `T = empty`

```
myappend(flatten1(T),L) = myappend(flatten1(empty),L) =
    myappend(nil,L) = L = fl2(empty,L) = fl2(T,L).
```

   For the case `T = leaf`

```
myappend(flatten1(T),L) = myappend(flatten1(leaf x),L) =
    myappend([x],L) = x::myappend(nil,L) = x::L = fl2(leaf x,L).
```

For the case `T = node(l,r)`

```
myappend(flatten1(T),L) = myappend(flatten1(node(l,r)),L) =
    myappend(myappend(flatten1(l),flatten1(r)),L) =
    myappend(flatten1(l),myappend(flatten1(r),L)) =
    myappend(flatten1(l),fl2(r,L)) = fl2(l,fl2(r,L)) =
    fl2(node(l,r),L)
```

where the fourth and fifth equalities come from the inductive hypothesis and the third from the associativity of `myappend`:

```
myappend(a,myappend(b,c)) = myappend(myappend(a,b),c)
```

for all `a,b,c` of type `int list`, which we can prove by structural induction on `a` in the following way.

For the case `a = nil`

```
myappend(a,myappend(b,c)) = myappend(nil,myappend(b,c)) =
    myappend(b,c) =
    myappend(myappend(nil,b),c) =
    myappend(a,myappend(b,c)).
```

For the case `a = h::tail`

```
myappend(a,myappend(b,c)) = myappend(h::tail,myappend(b,c)) =
    h::myappend(tail,myappend(b,c)) =
    h::myappend(myappend(tail,b),c) =
    myappend(h::myappend(tail,b),c) =
    myappend(myappend(h::tail,b),c) =
    myappend(myappend((a,b),c))
```

where the third equality follows from the inductive hypothesis.

2. An expression of the form $[E|V \in L]$ is called a list comprehension and evaluates to the list of values of E as the value of V ranges over items of the list L. An example is $[2*x|x \in 1 \ upto\ 20]$. With more than one variable, the syntax is

$$[E|V_1 \in L_1; V_2 \in L_2; \cdots]$$

and the value of L_k may depend on the value of V_1 up to V_{k-1}. For example, the value of

$$[x * y|x \in 1 \ upto\ 10; y \in 1 \ upto\ x]$$

is a list of 55 integers.

Specify an abstract syntax for list comprehension expressions, and then define an ML datatype to represent this. (The arithmetic expressions allowed inside comprehensions may be limited to very simple integer expressions.) Using this datatype, implement an ML "interpreter" function that evaluates list comprehensions.

[Solution due to Ming-Ter Kuo.] For simplicity, we allow only two arithmetic operators, + and `*`. A range for a variable is intensionally represented by a lower bound and an upper bound, where each bound is an expression that may contain variables that appeared previously.

The abstract syntax and ML datatypes to represent it are as follows.

Expressions (E)	L	(literal)
	V	(variable)
	$E_1 + E_2$	(addition)
	$E_1 * E_2$	(multiplication)
Ranges (R)	(V, E_1, E_2)	(variable, lower bound, upper bound)
	$R_1\ R_2$	
List comprehensions (LC)	$E\ R$	

```
datatype Exp = Lit of int                      (* literal *)
             | Var of string                   (* variable *)
             | Plus of Exp * Exp               (* addition *)
             | Times of Exp * Exp              (* multiplication *)

datatype Range = Upto of string * Exp * Exp  (* range of variable *)

datatype ListComp = LC of Exp * Range list   (* list comprehension *)
```

For example the list comprehension $[x * y \mid x \in 1\ \mathit{upto}\ 4; y \in 1\ \mathit{upto}\ x]$ is

```
- val exp1 = Times((Var "x"),(Var "y"));
- val range1 = [Upto("x",(Lit 1),(Lit 4)), Upto("y",(Lit 1),(Var "x"))];
- val lc_exp1 = LC(exp1, range1) ;
```

and the list comprehension $[2x + y * z \mid x \in 1\ \mathit{upto}\ 2; y \in 1\ \mathit{upto}\ x; z \in 1\ \mathit{upto}\ x + y]$ is

```
- val exp2 = Plus(Times((Lit 2),(Var "x")), Times((Var "y"),(Var "z")));
- val range2 = [Upto("x",(Lit 1),(Lit 2)), Upto("y",(Lit 1),(Var "x")),
                Upto("z", (Lit 1), (Plus((Var "x"), (Var "y"))))] ;
- val lc_exp2 = LC(exp2, range2) ;
```

We will build up to a final function that evaluates list comprehensions through a sequence of auxiliary functions. The first of these, `evalVar`, searches for the value a variable is bound to in an environment, which is a list of bindings of variables. If a variable has no bindings, the program is aborted–no exception handler is implemented.

```
datatype Env = Bind of string * int
exception Free_Variable
fun evalVar v [] = raise Free_Variable
  | evalVar v (Bind(w, n) :: env) = if v = w then n else evalVar v env
```

The function `evalExp` evaluates an arithmetic expression according to an environment.

```
fun evalExp (Lit n) env = n
  | evalExp (Var v) env = evalVar v env
  | evalExp (Plus(exp1,exp2)) env = (evalExp exp1 env)+(evalExp exp2 env)
  | evalExp (Times(exp1,exp2)) env = (evalExp exp1 env)*(evalExp exp2 env)
```

```
- val b1 = Bind("x", 1);
- val b2 = Bind("y", 2);

- evalExp (Times((Var "x"), (Var "y"))) [b1,b2];
val it = 2 : int
- evalExp (Plus((Var "x"), (Times((Lit 2), (Var "y"))))) [b1, b2]
val it = 5 : int
```

The function `range2list` generates a list of integers from the lower bound up to the upper bound.

```
fun range2list (lb,ub) = if lb > ub then []
                         else lb :: range2list(lb+1, ub)
- range2list(3, 9);
val it = [3,4,5,6,7,8,9] : int list
```

The function `consonto` is a curried versions of `::`.

```
fun consonto x a = a :: x
```

The function `link` appends together all the lists in a list into a single list.

```
fun link [] = []
  | link (a :: x) = a @ link x

- link [[1,2,3],[4,5],[6,7]];
val it = [1,2,3,4,5,6,7] : int list
```

The function `bind` is a curried version of the constructor `Bind`, which makes a binding between a variable name and an integer.

```
fun bind v n = Bind(v,n)
```

The recursive function `gen_env` generates a list of environments from a list of variable ranges. The first argument of `gen_env` is a list of ranges to be expanded and the second argument is an environment formed by previous variables.

Algorithm. Given a list of ranges `[r1, r2, ..., rn]` remaining to be expanded and a list of bindings `env`, `gen_env` generates a list of environments combining the previous environment with each remaining variable expanded over its range.

- Step 1. `evalExp` evaluates the lower and upper bounds (`l, u`) of the range `r1 = Upto(v, l_exp, u_exp)` using environment `env`.
- Step 2. `range2list` generates a list `[l, l+1, ..., u]`.
- Step 3. `map (bind v) ...` constructs a list of bindings `[(v,l),(v,l+1),...,(v,u)]`.
- Step 4. `map (consonto env) ...` conses each binding from Step 3 onto the previous environment `env` to form a new environment with one additional binding.
- Step 5. `map (gen_env tail_ranges) ...` recursively generates environments with respect to the remaining range list (`tail_ranges`) and the list of new environments.
- Step 6. Each application of `gen_env tail_ranges` to an environment produces a list of environments; these are appended together by `link`.

```
fun  gen_env [] env = [env]
  |  gen_env (Upto(v, l, u) :: tail_ranges) env =
         let val lb = (evalExp l env) in
            let val ub = (evalExp u env) in
              link (map (gen_env tail_ranges)
                        (map (consonto env)
                             (map (bind v)
                                  (range2list(lb, ub)))))
            end
         end

- gen_env  [Upto ("x",Lit 1,Lit 4),Upto ("y",Lit 1,Var "x")] [];
val it = [[Bind ("y",1),Bind ("x",1)],[Bind ("y",1),Bind ("x",2)],
          [Bind ("y",2),Bind ("x",2)],[Bind ("y",1),Bind ("x",3)],
          [Bind ("y",2),Bind ("x",3)],[Bind ("y",3),Bind ("x",3)],
          [Bind ("y",1),Bind ("x",4)],[Bind ("y",2),Bind ("x",4)],
          [Bind ("y",3),Bind ("x",4)],[Bind ("y",4),Bind ("x",4)]]
        : Env list list

- gen_env [Upto("x",Lit 1,Lit 2),Upto("y",Lit 1,Var "x"),
           Upto("z",Lit 1, Plus(Var "x", Var "y"))] [];
val it = [[Bind ("z",1),Bind ("y",1),Bind ("x",1)],
          [Bind ("z",2),Bind ("y",1),Bind ("x",1)],
          [Bind ("z",1),Bind ("y",1),Bind ("x",2)],
          [Bind ("z",2),Bind ("y",1),Bind ("x",2)],
          [Bind ("z",3),Bind ("y",1),Bind ("x",2)],
          [Bind ("z",1),Bind ("y",2),Bind ("x",2)],
          [Bind ("z",2),Bind ("y",2),Bind ("x",2)],
          [Bind ("z",3),Bind ("y",2),Bind ("x",2)],
          [Bind ("z",4),Bind ("y",2),Bind ("x",2)]] : Env list list
```

Finally, the function `evalLC` is the desired "interpreter" that evaluates a list comprehension expression `LC(exp, ranges)` where `exp` is the expression and `ranges` is the list of variable ranges. `evalLC` calls `gen_env ranges []` to generate all the possible environments and applies `evalExp exp` to each environment to evaluate the expression `exp`.

```
fun  evalLC (LC(exp, ranges)) = map (evalExp exp) (gen_env ranges [])
```

Here is one sample invocation, on $[x * y \mid x \in 1 \; upto \; 4; y \in 1 \; upto \; x]$:

```
- exp1;
val it = Times (Var "x",Var "y") : Exp
- range1;
val it = [Upto ("x",Lit 1,Lit 4),Upto ("y",Lit 1,Var "x")] : Range list

- evalLC(LC(exp1, range1));
val it = [1,2,4,3,6,9,4,8,12,16] : int list
```

And another sample invocation, on $[2x + y * z \mid x \in 1 \; upto \; 2; y \in 1 \; upto \; x; z \in 1 \; upto \; x + y]$:

```
- exp2;
val it = Plus (Times (Lit 2,Var "x"),Times (Var "y",Var "z")) : Exp
- range2;
val it = [Upto ("x",Lit 1,Lit 2),Upto ("y",Lit 1,Var "x"),
          Upto ("z",Lit 1,Plus ((Var "x"),(Var "y")))] : Range list
- evalLC (LC(exp2, range2));
val it = [3,4,5,6,7,6,8,10,12] : int list
```