```
signature TERM =
sig

  type const = string                           (* Constants c,f  *)
  type parm = string                            (* Parameters a   *)
  type var = string                             (* Variables x    *)

                                                (* Terms          *)
  datatype term =                               (* t ::=          *)
    Parm of parm                                (*   a            *)
  | Var of var                                  (* | x            *)
  | Root of const * term list                   (* | f(t1,...,tn) *)
  | EVar of parm list * (term option) ref (* | X^P     *)

  (* subst (t,x) s ===> [t/x] s,  t closed *)
  val subst : (term * var) -> term -> term

  (* eq (t, s) ===> true  iff  t = s, t and s closed *)
  (* eq’ (ts, ss) ===> true  iff  ts = ss, ts and ss closed *)
  val eq : term * term -> bool
  and eq’ : term list * term list -> bool

  (* closed P t ===> true  iff t closed wrt P *)
  val closed : parm list -> term -> bool
```

```
  (* newParm () ===> a,  where a is a new internal parameter *)
  (* Internal parameters are natural numbers. *)
  val newParm : unit -> parm

  val newEVar : parm list -> term

  exception Unify of string
  val unify : term * term -> unit         (* may raise Unify(msg) *)
  val unify' : term list * term list -> unit
                                          (* may raise Unify(msg) *)

end;  (* signature TERM *)
```

```
structure Term :> TERM =
struct

  type const = string                     (* Constants c,f  *)
  type parm = string                      (* Parameters a   *)
  type var = string                       (* Variables x    *)

                                          (* Terms          *)
  datatype term =                         (* t ::=          *)
    Parm of parm                          (*   a            *)
  | Var of var                            (* | x            *)
  | Root of const * term list             (* | f(t1,...,tn) *)
  | EVar of parm list * (term option) ref (* | X^P<=t>      *)

  (*
     We say "t is closed" if t contains no free variables.
     We say "ts is closed" if all terms in ts are closed.
  *)

  (* Global invariant: If an EVar is instantiated to t,
     t is closed and all parameters in t are mentioned in P

     Global invariant: For any two EVar's which might be unified,
     the parameter context of one is a prefix of the other.
  *)
```

```
(* subst (t,x) s ===> [t/x] s; t must be closed *)
fun subst (t, x) s =
    let fun sbst (s as Parm _) = s
          | sbst (s as Var(y)) =
            if x = y then t else s
          | sbst (Root (g, ss)) =
              Root (g, map sbst ss)
          | sbst (X as EVar _) = X
    in
      sbst s
    end

(* eq (t, s) ===> true iff t = s; t and s must be closed *)
(* eq' (ts, ss) ===> true iff ts = ss; ts and ss must be closed *)
fun eq (Parm(a), Parm(b)) = (a = b)
  | eq (Root(f, ts), Root(g, ss)) =
      (f = g) andalso eq' (ts, ss)
  | eq (EVar(P, r), EVar(P', r')) = (r = r')
  | eq _ = false                         (* Var(_) not permitted! *)

and eq' (nil, nil) = true
  | eq' (t::ts, s::ss) = eq (t, s) andalso eq' (ts, ss)
  | eq' _ = false
```

```
(* member (x, l) ===> true iff x is a member of l *)
(* uses standard equality *)
fun member (x, nil) = false
  | member (x, y::l) = x = y orelse member (x, l)

(* closed P t ===> true  iff t closed wrt P *)
(*
   t is closed wrt P if it is closed and
   all parameters in t are declared in parameter context P.
*)
fun closed P =
    let fun cld (Parm(a)) = member (a, P)
          | cld (Var _) = false
          | cld (Root (f, ts)) = cld' ts
          | cld (EVar _) = true
        and cld' (nil) = true
          | cld' (t::ts) = cld t andalso cld' ts
    in cld end
```

```
local
  (* local counter to guarantee new internal parameters *)
  val counter = ref 0
in
  (* newParm () ===> a,  where a is a new internal parameter *)
  (*
     Internal parameters are natural numbers,
     so they cannot conflict with external parameters.
  *)
  fun newParm () =
      (counter := !counter+1;
       Int.toString (!counter))
end

fun newEVar (P) = EVar (P, ref(NONE))
```

```
(* contained (P’, P) => true  iff P’ is contained in P
   Invariant: either P is a prefix of P’ or P’ a prefix of P
*)
fun contained (nil, _) = true
  | contained (a’::P’, a::P) = contained (P’, P)
  | contained _ = false

(* restrict (P’, P) => P (= intersection of P’ and P)
   Invariant: P is a prefix of P’
*)
fun restrict (P’, P) = P

(* excection Unify(reason) signals failure *)
exception Unify of string
```

```
(* check (X^P<>, t) => ()
   raises Unify(reason) if X or a parameter not in P occurs in t
   Effect: might restrict parameter occurrence in EVars by
           instantiation
*)
fun check (EVar(P, r), t) =
    let fun chk (Parm(a)) =
            if member (a, P) then ()
            else raise Unify ("Parameter occurrence")
          | chk (Root(f, ts)) = chk' ts
          | chk (EVar (P', ref(SOME(t)))) = chk t
          | chk (EVar (P', r')) =
            if r = r' then raise Unify ("Variable occurrence")
            else if contained (P', P) then ()
                 else instantiate (r', newEVar (restrict (P', P)))
          (* Var(x) impossible *)
        and chk' (nil) = ()
          | chk' (t::ts) = (chk t; chk' ts)
    in
      chk t
    end
```

```
(* unifyEVar (X<>, t) => ()
    Effect: instantiate EVars due to parameter restrictions
            instantiate X
            raises Unify (reason) if not unifiable
 *)
 fun unifyEVar (X as EVar (P, r), t) =
       (check (X, t); instantiate (r, t))
```

```
 (* unify (t, s) => ()
     unify' (ts, ss) => ()
     t, s, ts, ss closed (but may contain parameters or EVars)
     Effect: instantiate EVars
             raises Unify (reason) if not unifiable
  *)
  fun unify (Parm(a), Parm(a')) =
      if a = a' then ()
      else raise Unify ("Parameter clash")
    | unify (Root(f, ts), Root(g, ss)) =
      if f = g then unify' (ts, ss)
      else raise Unify ("Constant clash")
    | unify (EVar(_, ref(SOME(t))), s) = unify (t, s)
    | unify (t, EVar(_, ref(SOME(s)))) = unify (t, s)
    | unify (X as EVar(_, r), X' as EVar(_, r')) =
      if r = r' then () else unifyEVar (X, X')
    | unify (X as EVar _, s) = unifyEVar (X, s)
    | unify (t, X as EVar _) = unifyEVar (X, t)
    (* Var(x) on either side impossible *)

  and unify' (nil, nil) = ()
    | unify' (t::ts, s::ss) = (unify (t, s); unify' (ts, ss))
    | unify' _ = raise Unify ("Argument number")

end;  (* structure Term *)
```