```
(* some general functions *)

(* cut x l = (l1,l2) with l = l1@(x@l2) if x occurs in l, and l1 = l, l2 = []
   if x does not occur in l *)

let rec cut x = function
 [] -> ([],[])
|y::rest -> if x = y then ([],rest) else
             let (l1,l2) = cut x rest in (y::l1,l2) ;;

let member x = orec where rec orec = function
 [] -> false
|y::rest -> if y=x then true else orec rest ;;

let cons_mem x l = if member x l then l else x::l ;;

(* non_repeat l has the same elements as l, but without repetition *)

let rec non_repeat = function
 []       -> []
|x::rest -> cons_mem x (non_repeat rest) ;;

(* add x l, if l is ordered and without repitition, then so is add x l, and
   add x l have the same elements as x::l *)

let rec add x l = match l with
 []    -> [x]
|y::l1 -> if x < y then x::l else
           if x = y then l else y::(add x l1) ;;

(* new_of_list 0<=n1<...<np gives the first n>=0 not in n1,..., np.
   nrec k l, if k <= n1 < ... < np gives the first element >= k not in
   n1,...,np *)

let new_and_list = nrec 0 where rec nrec k = function
 [] -> (k,[k])
|x::l -> if k < x then (k,k::(x::l)) else
          let (n,l1) = nrec (k+1) l in (n,x::l1) ;;

let new_of_list = fst o new_and_list ;;

let rec news_of_list k l = if k = 0 then [] else
let (n,nl) = new_and_list l in n::(news_of_list (k-1) nl) ;;

(* for dealing with gensym *)

(* break x_n in x and n, with 0 if no _ in x_n. Gives 0 if
  it is x_n with n has alphabetical character *)

exception Error of string;;
let error string = raise Error(string);;

let break s = let (l1,l2) = cut "_" (explode s) in match l2 with
 [] -> (s,0)
| _ -> (try (implode l1,num_of_string (implode l2))
        with _ -> (s,0)) ;;

let mude x_n = let (x,_) = break x_n in x = "h" ;;

(* extract all variables x_k *)

(* start_with x k output [] if x_k does not occur in then list, and
   output the list of all x_ki in the list if x_k is in the list *)

let start_with x k l =
```

```
let rec srec = function

 [] -> ([],false)

|y_n::rest ->
let (y,n) = break y_n in
let (li,b) = srec rest in
if y = x then if k = n then (add n li,true) else (add n li,b) else (li,b)

in let (lr,b) = srec l in if b then lr else [] ;;

let fresh x_n l = let (x,n) = break x_n in match start_with x n l with
 [] -> x_n
|ln -> let k = string_of_num (new_of_list ln) in x^("_"^k) ;;

let gen_look_up x = lrec where rec lrec = function
 (x1,d1)::dl -> if x = x1 then d1 else lrec dl
| _ -> error"look_up" ;;

let look_up x = lrec where rec lrec = function
 (x1,d1)::dl -> if x = x1 then d1 else lrec dl
| _ -> error(x^" is not in the list") ;;

let test_occur s p = trec s where rec trec = function
 [] -> false
|(q,_)::rest -> if q = p then true else trec rest ;;

let rec or_map f = function
 [] -> false
|x::rest -> f x or or_map f rest ;;

(* union of two lists *)

let cons_mem x l = if member x l then l else x::l ;;

let rec union l1 = function
 [] -> l1
|x::rest -> union (cons_mem x l1) rest ;;

(* union of two num list *)

let rec union_add l = function
 [] -> l
|x::rest -> union_add (add x l) rest ;;


(* we have a substitution s, and a list of name that may occur in an
   expression. We want to know if the subtitution mentionned one of these
   names *)

let mention s = or_map (test_occur s) ;;

let rec extend l1 l2 = match (l1,l2) with
 (_,[]) -> true
|(x1::rest1,x2::rest2) -> if x1 = x2 then extend rest1 rest2 else false
| _ -> false ;;

let ext_rev l1 l2 = extend (rev l1) (rev l2) ;;

(* union of two lists *)

let member_eq eq x = orec where rec orec = function
 [] -> false
|y::rest -> if eq y x then true else orec rest ;;

let cons_mem_eq eq x l = if member_eq eq x l then l else x::l ;;
```

```
let rec union_eq eq l1 = function
 [] -> l1
|x::rest -> union_eq eq (cons_mem_eq eq x l1) rest ;;

let rec flat_map eq f = function
 [] -> []
|x::rest -> union_eq eq (f x) (flat_map eq f rest) ;;

let gen_eq x y = x = y ;;

let union_rewrite_eq eq =

let rec arec cl = function

 [] -> cl

|x::rest -> arec (x::(crec x cl)) rest

and crec x = function

 [] -> []

|y::rest -> if eq y x then crec x rest else y::(crec x rest)

in arec ;;

let rec concac l1 l2 = match l2 with
 [] -> l1
|x::rest -> concac (x::l1) rest ;;
```

```
(* abstract syntax of expressions and values *)

type exp =
 Var of string
|Abs of string * exp * exp
|Prd of string * exp * exp
|Let of string * exp * exp * exp
|App of exp * exp
|Meta of num * (string * exp) list
|Sort of sort

and sort = Type of num | Prop ;;
```

→ term

→ type

```
type cond = isU | ctype of num | isProp ;;

type pattern == exp ;;
```

[x:A] M

```
let satisfy c s = match (c,s) with
  (isU,_) -> true
|(ctype p,Type q) -> q <= p
|(_,Prop) -> true
| _ -> false ;;
```

λ(x:) M =

```
let pred = function      (* condition for typing a product *)
 isU -> isU
|isProp -> isU
|ctype p -> ctype p ;;

let prod_sort s1 s2 = match (s1,s2) with
 (_,Prop) -> Prop
|(Prop,Type k)      -> Type k
|(Type k1,Type k2)  -> Type (max(k1+1) k2) ;;
```

c = λx λy case + =

```
let pred_of_sort = function
 Prop -> isU
|Type(k) -> ctype k ;;

let type_of_sort = function
 Type k -> Type (k+1)
|Prop -> Type 0 ;;
```

Γ ξ:

```
type def =
 Undefined
|Def of exp      * exp
|Prim of exp
|Rec of string list * exp * (pattern * exp) list ;;
```

x =

```
type decl == exp * def ;;

type env == (string * exp) list ;;

(* the identity environment, for initialising the meta-variables *)

type context == (string * decl) list ;;

type vals == (string * def) list ;;

type typs  == (string * exp) list ;;

let (idenv_con:context -> env) = map (fun (x,_) -> (x,Var x)) ;;

let rec is_id = function
 [] -> true
|(x,e)::rest -> Var x = e & is_id rest ;;
```

```
(* derived notions of application and abstraction *)

let rec applist = function
 (e,[]) -> e
|(e,e1::l1) -> applist(App(e,e1),l1) ;;

let hyp_ext con x v = (x,(v,Undefined))::con ;;

let def_ext con x w v = (x,(w,Def v))::con ;;

let prim_ext con x w v = (x,(w,Prim v))::con ;;

let rec_ext con x w li l e = (x,(w,Rec (li,e,l)))::con ;;

(* the global context *)

let CONT = ref([]:context);;

let get_context() = !CONT;;

let declared x con = member x (map fst (con@(!CONT))) ;;

let decl x con = snd (look_up x (con@(!CONT))) ;;

let typage_ident x con = fst (look_up x (con@(!CONT))) ;;

let gensym con x = fresh x (map fst (con@(!CONT))) ;;

let new_ident con = gensym con "X";;
```

```
(* computation of the beta-normal form; this is an untyped algorithm. *)

let pattern_matching e l = prec([],e,l) where rec prec = function

 (_,_,[]) -> error"pattern_matching"

|(env,Var x,(Var y,e0)::rest) ->
 if x = y then (e0,env) else prec([],e,rest)

|(env,App(e1,e2),(App(p1,Var x),e0)::rest) ->
 prec((x,e2)::env,e1,(p1,e0)::rest)

|(env,_,_::rest) -> prec([],e,rest) ;;


(* we do pattern matching only with first-order pattern, i.e. if
   c e1...en is a constructor, then its type is canonical not a product.
   We don't want c:(A:Type)(A)A, and then [x:N]c((N)N,id,x) = c((N)N,x).
   I.e. a constructor has an arity, and it is when we compute the
   delta normal form of its type (x1:A1)...(xp:Ap)X(t1,...,tq) where
   X is NOT a variable *)

let (fetch:string->env->exp) x = frec where rec frec = function
 [] -> Var x
|(x1,v)::env -> if x = x1 then v else frec env ;;

let (map_env:(exp->exp) -> (env->env)) f = map (fun (x,e) -> (x,f e)) ;;

(* invariant for inst: the string list contains the free variables of all
   the expressions in env. The result is a correct expression if all the
   expressions of env are correct *)

let rec (inst:string list * env * exp -> exp) = function

 (_,env,Var x) -> fetch x env

|(l,env,Meta(n,env1)) -> Meta(n,map_env (fun x -> inst (l,env,x)) env1)

|(l,env,Abs(x,e1,e2)) ->
let x1 = fresh x l in Abs(x1,inst(l,env,e1),inst(x1::l,(x,Var x1)::env,e2))

|(l,env,Prd(x,e1,e2)) ->
let x1 = fresh x l in Prd(x1,inst(l,env,e1),inst(x1::l,(x,Var x1)::env,e2))

|(l,env,Let(x,e1,e2,e3)) ->
let x1 = fresh x l in
Let(x1,inst(l,env,e1),inst(l,env,e2),inst(x1::l,(x,Var x1)::env,e3))

|(l,env,App(e1,e2)) -> (match inst(l,env,e1) with
                          Abs(x,_,e) -> inst(l,[x,inst(l,env,e2)],e)
                        | e -> App(e,inst(l,env,e2)))

|(l,env,e) -> e ;;

let (rename: string -> string -> exp -> exp) x x1 e =
if x = x1 then e else inst([x1],[x,Var x1],e) ;;

(* two algorithms defined recursively, one is one step reduction, the
   other is  the computation of the canonical form *)

let symbs con = map fst (con@(get_context())) ;;

let list_vals con =
map (fun (x,(e,d)) -> (x,d)) (con@(get_context())) ;;

let list_typs con =
```

```
map (fun (x,(e,d)) -> (x,e)) (con@(get_context())) ;;

let not_occur x =

let rec nrec = function

 Var y -> not (x=y)

|Abs(y,e1,e2) -> nrec e1 & (x=y or nrec e2)

|Prd(y,e1,e2) -> nrec e1 & (x=y or nrec e2)

|Let(y,e1,e2,e3) -> nrec e1 & nrec e2 & (x=y or nrec e3)

|App(e1,e2) -> nrec e1 & nrec e2

| _ -> true

in nrec ;;

let subst con x e1 e2 = if not_occur x e2 then e2 else
inst(symbs con,[x,e1],e2) ;;

let subst con x e1 e2 = if not_occur x e2 then e2 else
inst(symbs con,[x,e1],e2) ;;

(* one_step con e fail iff e cannot be reduced *)

(* the evaluation of a term can stopped either because of "meta-variables"
   or because of ordinary variables, or because the term is canonical *)

type output =
   Stop_var
|  Stop_meta
|  Step of exp
|  Clos of env * string list * exp * (pattern * exp) list ;;

(* problem with let, should not contain meta-variables *)


let rec (one_step: vals -> exp -> output) lval = function

 Var x ->
(match look_up x lval with
 Def e -> Step e
|Prim e -> Stop_meta
| _ -> Stop_var)

|Meta(_,_) -> Stop_meta

|App(e1,e2) -> (match one_step1 lval e1 with
                      Step (Abs(x,_,e)) -> Step (inst(map fst lval,[x,e2],e))
                     |Step e -> Step (App(e,e2))
                     |Clos(env,[x],e,lp) ->
 let li = map fst lval in
 let env2 = (x,e2)::env in
 let (e',em) = can_with_em lval (inst(li,env2,e)) in
 (try let (e3,env3) = pattern_matching e' lp in
  Step (inst(li,env3@env2,e3)) with Error _ -> em)
                     | em -> em)

|Let(x,e1,_,e3) -> Step (inst(map fst lval,[x,e1],e3))

| _ -> Stop_var

and (one_step1: vals -> exp -> output) lval = function
```

```
  Var x -> (match look_up x lval with
             Undefined -> Stop_var
            |Prim e -> Stop_meta
            |Def e -> Step e
            |Rec(l,e,lp) -> Clos([],l,e,lp))

 |Meta _ -> Stop_meta

 |App(e1,e2) -> (match one_step1 lval e1 with
                  Step (Abs(x,_,e)) -> Step (inst(map fst lval,[x,e2],e))
                 |Step e -> Step (App(e,e2))
                 |Clos(env,[x],e,lp) ->
  let li = map fst lval in
  let env2 = (x,e2)::env in
  let (e',em) = can_with_em lval (inst(li,env2,e)) in
  (try let (e3,env3) = pattern_matching e' lp in
   Step (inst(li,env3@env2,e3)) with Error _ -> em)
                 |Clos(env,x::li,e,lp) -> Clos((x,e2)::env,li,e,lp)
                 | em -> em)

 |Let(x,e1,_,e3) -> Step (inst(map fst lval,[x,e1],e3))

 |(Abs(_) as e) -> Step e

 | _ -> Stop_var

and (can_with_em: vals -> exp -> exp * output) lval u =
match one_step lval u with
 Step e -> can_with_em lval e
|em -> (u,em)

and can lv u = fst (can_with_em lv u) ;;

let can_con con = can (list_vals con) ;;

type occurence == num list ;;

let unfold occ con e =
let lv = list_vals con in

let rec urec = function

 (lv,[],e) ->
(match one_step lv e with Step e1 -> e1 | _ -> error"cannot be unfolded")

|(lv,1::occ,Prd(x,e1,e2)) -> Prd(x,urec(lv,occ,e1),e2)

|(lv,2::occ,Prd(x,e1,e2)) -> Prd(x,e1,urec((x,Undefined)::lv,occ,e2))

|(lv,1::occ,Abs(x,e1,e2)) -> Abs(x,urec(lv,occ,e1),e2)

|(lv,2::occ,Abs(x,e1,e2)) -> Abs(x,e1,urec((x,Undefined)::lv,occ,e2))

|(lv,1::occ,App(e1,e2)) -> App(urec(lv,occ,e1),e2)

|(lv,2::occ,App(e1,e2)) -> App(e1,urec(lv,occ,e2))

|(lv,1::occ,Let(x,e1,e2,e3)) -> Let(x,urec(lv,occ,e1),e2,e3)

|(lv,2::occ,Let(x,e1,e2,e3)) -> Let(x,e1,urec(lv,occ,e2),e3)

|(lv,3::occ,Let(x,e1,e2,e3)) -> Let(x,e1,e2,urec((x,Def e1)::lv,occ,e3))

| _ -> error"unfold"
```

```
in urec(lv,occ,e) ;;

let can_occ occ con e =
let lv = list_vals con in

let rec urec = function

 (lv,[],e) -> can lv e

|(lv,1::occ,Prd(x,e1,e2)) -> Prd(x,urec(lv,occ,e1),e2)

|(lv,2::occ,Prd(x,e1,e2)) -> Prd(x,e1,urec((x,Undefined)::lv,occ,e2))

|(lv,1::occ,Abs(x,e1,e2)) -> Abs(x,urec(lv,occ,e1),e2)

|(lv,2::occ,Abs(x,e1,e2)) -> Abs(x,e1,urec((x,Undefined)::lv,occ,e2))

|(lv,1::occ,App(e1,e2)) -> App(urec(lv,occ,e1),e2)

|(lv,2::occ,App(e1,e2)) -> App(e1,urec(lv,occ,e2))

|(lv,1::occ,Let(x,e1,e2,e3)) -> Let(x,urec(lv,occ,e1),e2,e3)

|(lv,2::occ,Let(x,e1,e2,e3)) -> Let(x,e1,urec(lv,occ,e2),e3)

|(lv,3::occ,Let(x,e1,e2,e3)) -> Let(x,e1,e2,urec((x,Def e1)::lv,occ,e3))

| _ -> error"can_occ"

in urec(lv,occ,e) ;;


let can_subst con x e1 e2 = can_con con (subst con x e1 e2) ;;
```

```
(* the conversion algorithm *)

(* simplification of equational constraint, with eta-conversion *)

exception Eq_error of exp * exp ;;
let eq_error(e1,e2) = raise Eq_error(e1,e2) ;;

let are_equal con e1 e2 =

let rec eqrec con u1 u2 =
if u1 = u2 then true else
match (u1,u2) with

 (Abs(x1,u1,v1),Abs(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u1 in
(eqrec con (can_con con u1) (can_con con u2)) &
eqrec con1 (can_con con1 (rename x1 x v1)) (can_con con1 (rename x2 x v2))

|(Abs(x1,u1,v1),e2) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u1 in
eqrec con1 (can_con con1 (rename x1 x v1)) (App(e2,Var x))

|(e1,Abs(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u2 in
eqrec con1  (App(e1,Var x)) (can_con con1 (rename x2 x v2))

|(Prd(x1,u1,v1),Prd(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u1 in
(eqrec con (can_con con u1) (can_con con u2)) &
eqrec con1 (can_con con1 (rename x1 x v1)) (can_con con1 (rename x2 x v2))

|(App(x1,v1),App(x2,v2)) ->
eqrec con x1 x2 & eqrec con (can_con con v1) (can_con con v2)

|(Sort (Type(p)),Sort(Type(q))) -> q = p


| _ -> false

in if e1 = e2 then true else
eqrec con (can_con con e1) (can_con con e2) ;;

(* if the terms are equal, the output is [] *)

let eq_simpl con e1 e2 =
```

?1 u_1 v_1 u_2 v_2

```
let rec eqrec leq con u1 u2 =
if u1 = u2 then leq else
let ((v1,em1),(v2,em2)) =
    (can_with_em (list_vals con) u1,can_with_em (list_vals con) u2) in
if v1 = v2 then leq else
if em1 = Stop_meta or em2 = Stop_meta then
if are_equal con v1 v2 then leq else (con,v1,v2)::leq else
match (v1,v2) with
```

u_1 u_2

?1 = $\boxed{v_2}$

```
 (Abs(x1,u1,v1),Abs(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u1 in
eqrec (eqrec leq con u1 u2) con1 (rename x1 x v1) (rename x2 x v2)

|(Abs(x1,u1,v1),e2) ->
let x = fresh "X" (symbs con) in
```

```
let con1 = hyp_ext con x u1 in
eqrec leq con1  (rename x1 x v1) (App(e2,Var x))

|(e1,Abs(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u2 in
eqrec leq con1 (App(e1,Var x)) (rename x2 x v2)

|(Prd(x1,u1,v1),Prd(x2,u2,v2)) ->
let x = fresh "X" (symbs con) in
let con1 = hyp_ext con x u1 in
eqrec (eqrec leq con u1 u2) con1 (rename x1 x v1) (rename x2 x v2)

|(App(x1,v1),App(x2,v2)) ->
eqrec (eqrec leq con v1 v2) con x1 x2

|(Sort (Type(p)),Sort(Type(q))) ->
if q = p then leq else eq_error(u1,u2)

| _ -> eq_error(u1,u2)

in eqrec [] con e1 e2 ;;
```

```
let x1 = gensym con x in
let leq1      = type_check lt (occ,con,leq,isU,e1) in
let (t2,leq2) = exp_check lt (2::occ,hyp_ext con x1 e1,leq1,rename x x1 e2) in
(Prd(x1,e1,t2),leq2)

|Prd(x,e1,e2) ->
let x1 = gensym con x in
let (t1,leq1) = exp_check lt (occ,con,leq,e1) in
let con1      = hyp_ext con x1 e1 in
let (t2,leq2) = exp_check lt (2::occ,con1,leq1,rename x x1 e2) in
(match (can_con con t1,can_con con1 t2) with
  (Sort s1,Sort s2) -> (Sort (prod_sort s1 s2),leq)
 | _ -> type_error(occ,e))

|Let(x,e1,e2,e3) ->
let x1 = gensym con x in
let leq2 = type_check lt (2::occ,con,leq,isU,e2) in
let leq1 = typing_check lt (1::occ,con,leq2,e2,e1) in
let (t3,leq3) = exp_check lt (3::occ,def_ext con x1 e2 e1,leq1,rename x x1 e3)
in (Let(x,e1,e2,t3),leq3)

|Sort s -> (Sort (type_of_sort s),leq)

and typing_check lt (occ,con,leq,t,e) =
let (t1,leq1) = exp_check lt (occ,con,leq,e) in (occ,con,t1,t)::leq1

and type_check lt (occ,con,leq,c,e) =
let (t1,leq1) = exp_check lt (occ,con,leq,e) in
match can_con con t1 with
 Sort s -> if satisfy c s then leq else type_error(occ,e)
| _      -> type_error(occ,e) ;;

(* take a list of equations and output the simplified list *)

let list_eq_simpl =
let eq con e1 e2 =
(try eq_simpl con e1 e2 with
  Eq_error (e1,e2) -> type_eq_error(e1,e2)
 | _ -> error"list_eq_simpl") in
let eqsimpl (occ,con,e1,e2) = map (fun x -> (occ,x)) (eq con e1 e2) in
flat_map gen_eq eqsimpl ;;

let typing_simpl lt u = list_eq_simpl (typing_check lt u) ;;

let type_simpl lt u = list_eq_simpl (type_check lt u) ;;

let triv_check_meta x = error"contains place holders" ;;

let is_a_type con e =
type_simpl [] ([],con,[],isU,e) = [] ;;

let is_of_type lt con e1 e2 =
let dummy = typing_simpl lt ([],con,[],e1,e2) in true ;;

(* type computation, knowing that the expression is correct *)

let try_type occ lt con e = fst (exp_check lt (occ,con,[],e)) ;;

let can_try_type occ lt con e = can_con con (try_type occ lt con e) ;;

let rec typing_correct_check lt (occ,con,leq,t,e) =
(occ,con,t,can_try_type occ lt con e)::leq ;;
```

```
let remove n = rmrec where rec rmrec = function
 [] -> []
|((occ,con,tc,p) as c)::rest -> if p = n then rest else c::(rmrec rest) ;;

(* we have a list of typing constraints and we want to know the type of the
   place holders. It fails if it is not at a correct occurence *)

let rec list_meta = function
 [] -> []
|(occ,con,tc,n)::rest -> add n (list_meta rest) ;;

(* backtract is safe only if there are no equational constraints *)

let clean_up1 occ = crec where rec crec = function
 [] -> []
|(occ1,con,x)::rest ->
if extend (rev occ1) occ then crec rest else (occ1,con,x)::(crec rest) ;;

let backtrack occ t (e,lt,leq) =
let z = new_of_list (list_meta lt) in
let ((con,t),e',leq') = deleterec lt z ([],t,e,occ,[]) in
(e',(rev occ,con,Is_of_type t,z)::(clean_up1 occ lt),
 union (clean_up1 occ leq) (list_eq_simpl leq')) ;;

type subst == (num * exp) list ;;

(* for operations on terms *)

(* we suppose that the term is correct w.r.t. its scope *)

let (inst_meta:(string list * subst * exp -> exp)) =

let rec irec (l,sub,e) = match e with

 Meta(n,env) ->
let (env1,b1) = mrec l sub env in
(try (inst(l,env1,gen_look_up n sub),false)
 with Error _ -> if b1 then (e,true) else (Meta(n,env1),false))

|Abs(x,e1,e2) ->
let (u1,b1) = irec(l,sub,e1) in
let (u2,b2) = irec(x::l,sub,e2) in
if b1 & b2 then (e,true) else (Abs(x,u1,u2),false)

|Prd(x,e1,e2) ->
let (u1,b1) = irec(l,sub,e1) in
let (u2,b2) = irec(x::l,sub,e2) in
if b1 & b2 then (e,true) else (Prd(x,u1,u2),false)

|Let(x,e1,e2,e3) ->
let (u1,b1) = irec(l,sub,e1) in
let (u2,b2) = irec(l,sub,e2) in
let (u3,b3) = irec(x::l,sub,e3) in
if b1 & b2 & b3 then (e,true) else (Let(x,u1,u2,u3),false)

|App(e1,e2) ->
let (u1,b1) = irec(l,sub,e1) in
let (u2,b2) = irec(l,sub,e2) in
if b1 & b2 then (e,true) else
 (match u1 with
   Abs(x,_,e) -> (inst(l,[x,u2],e),false)
  | e -> (App(e,u2),false))

| _ -> (e,true)

and mrec l sub env = match env with
```

```
 [] -> ([],true)

|(x,e)::rest ->
let (u1,b1) = irec(l,sub,e) in
let (l2,b2) = mrec l sub rest in
if b1 & b2 then (env,true) else ((x,u1)::l2,false)

in fst o irec ;;

let subst_exp sub con e = inst_meta(symbs con,sub,e) ;;

let subst_con sub =

let rec srec = function

 [] -> []

|(x,(e,d))::rest -> (x,(subst_exp sub rest e,drec rest d))::(srec rest)

and drec con = function

 Def e -> Def (subst_exp sub con e)

| d -> d

in srec ;;



(* application of a substitution in a equation *)

let subst_eq_elem sub (occ,con,e1,e2) =
let f x = inst_meta(symbs con,sub,x) in
map (fun x -> (occ,x)) (eq_simpl con (f e1) (f e2)) ;;

let subst_eq sub = flat_map gen_eq (subst_eq_elem sub) ;;

type goal == exp * type_constraints * eq_constraints ;;

(* application of a substitution in a equation *)

let map_cts_subst sub = mrec where rec mrec = function
 [] -> []
|(occ,con,Is_of_type e,p)::rest ->
 (occ,con,Is_of_type (subst_exp sub con e),p)::(mrec rest)
|c::rest -> c::(mrec rest) ;;

(* don't need type-checking *)

let subst_tp sub = srec where rec srec = function
 [] -> []
|(occ,con,Is_of_type e,p)::rest ->
 (occ,subst_con sub con,Is_of_type (subst_exp sub con e),p)::(srec rest)
|(occ,con,tc,p)::rest ->
 (occ,subst_con sub con,tc,p)::(srec rest) ;;

(* apply the operation f on the type of the nth subgoal *)

let apply_subgoal n f = srec where rec srec = function

 [] -> error"apply_subgoal"

|((occ,con,Is_of_type(e),p) as c)::rest ->
if p = n then (occ,con,Is_of_type(f con e),p)::rest else c::(srec rest)
```

```
| c::rest -> c::(srec rest) ;;



(* simplification by the equational constraints *)

(* extraction of a substitution from a goal *)

let meta_of_exp = mrec [] where rec mrec l = function
 Meta(n,_) -> add n l
|App(e1,e2) -> mrec (mrec l e1) e2
|Abs(_,e1,e2) -> mrec (mrec l e1) e2
|Prd(_,e1,e2) -> mrec (mrec l e1) e2
|Let(_,e1,e2,e3) -> mrec (mrec (mrec l e1) e2) e3
| _ -> l ;;

let rmlist l = rmrec where rec rmrec = function
 [] -> []
|((occ,con,tc,p) as c)::rest ->
 if member p l then c::(rmrec rest) else rmrec rest ;;

let precedes lt n p =
let conp = fst (snd (occ_con_type p lt)) and
    conn = fst (snd (occ_con_type n lt)) in
extend (rev conp) (rev conn) ;;

let elem_subst_of_cts lt = function
 (_,_,Meta(n,envn),Meta(p,envp)) ->
if precedes lt n p & is_id envp then [p,Meta(n,envn)] else
if precedes lt p n & is_id envn then [n,Meta(p,envp)] else []
|(_,_,Meta(n,envn),v) -> if is_id envn then [n,v] else []
|(_,_,v,Meta(n,envn)) -> if is_id envn then [n,v] else []
| _ -> [] ;;

let subst_of_eql lt el = srec [] el where rec srec l = function
 [] -> (l,[])
|x::rest -> (match elem_subst_of_cts lt x with
              [] -> let (l1,l2) = srec l rest in (l1,x::l2)
             |[u] -> let (l1,l2) = srec l rest in (u::l1,l2)
             | _ -> error"subst_of_cts") ;;

(* check the correctness of the substitution with respect to the occurences *)

let rec is_correct l = function
 Var x -> member x l
|App(e1,e2) -> is_correct l e1 & is_correct l e2
|Abs(x,e1,e2) -> is_correct l e1 & is_correct (x::l) e2
|Prd(x,e1,e2) -> is_correct l e1 & is_correct (x::l) e2
|Let(x,e1,e2,e3) -> is_correct l e1 & is_correct l e2 & is_correct (x::l) e3
| _ -> true ;;

let is_correct_exp n lt e = let (_,con,_) = occ_con_type n lt in
is_correct (symbs con) e ;;

let is_correct_sub lt = irec where rec irec = function
 [] -> true
|(n,e)::rest -> is_correct_exp n lt e & irec rest ;;

let (simpl:goal -> goal) (e,lt,leq) =
let (sub,leql) = subst_of_eql lt leq in
if is_correct_sub lt sub then
 let el = subst_exp sub [] e in
 let l1 = meta_of_exp el in
 (el,subst_tp sub (rmlist l1 lt),subst_eq sub leq)
else error"the substitution is not correct w.r.t. the scope" ;;
```

```
(* try to guess a predicate: if we have the equation ?k(x,y) = t, then tries
   ?k = \x,y.t *)

let rec elem_try_subst con v = function
 Meta(n,envn) -> if is_id envn then [n,v] else []
|App(u,Var x) -> elem_try_subst con (Abs(x,typage_ident x con,v)) u
| _ -> [] ;;

let elem_try_subst_of_eql (_,con,u,v) =
 (match elem_try_subst con u v with [] -> elem_try_subst con v u | l -> l) ;;

let try_subst_of_eql el = srec [] el where rec srec l = function
 [] -> (l,[])
|x::rest -> (match elem_try_subst_of_eql x with
               [] -> let (l1,l2) = srec l rest in (l1,x::l2)
              |[u] -> let (l1,l2) = srec l rest in (u::l1,l2)
              | _ -> error"subst_of_cts") ;;

let (try_simpl:goal -> goal) (e,lt,leq) =
let (sub,leq1) = try_subst_of_eql leq in
if is_correct_sub lt sub then
 let e1 = subst_exp sub [] e in
 let l1 = meta_of_exp e1 in
 (e1,subst_tp sub (rmlist l1 lt),subst_eq sub leq)
else error"the substitution is not correct w.r.t. the scope" ;;

(* introduction tactic, with anonymous intro *)

(* we can do an introduction only if the place holders do not occur in other
   goals *)

let occur_meta p = orec where rec orec = function
 Meta(q,_)   -> q = p
|App(e1,e2) -> orec e1 or orec e2
|Abs(_,e1,e2) -> orec e1 or orec e2
|Prd(_,e1,e2) -> orec e1 or orec e2
|Let(_,e1,e2,e3) -> orec e1 or orec e2 or orec e3
| _ -> false ;;

let occur_in_goal p = orec where rec orec = function
 [] -> false
|(_,_,Is_of_type(e),_)::rest -> occur_meta p e or orec rest
|_::rest -> orec rest ;;


let rec build_abs n con l t = match (l,can_con con t) with

 ([],_) -> (Meta(n,idenv_con con),t,con)

|(s::rest,Prd(x,e1,e2)) ->
 let s1 = gensym con s in
 let (M,t,con1) = build_abs n (hyp_ext con s1 e1) rest (rename x s1 e2) in
 (Abs(s1,e1,M),t,con1)

| _ -> error"is not enough functional" ;;



let intro_anonymous n (M,lt,leq) =

match occ_con_type n lt with
```

```
  (occ,con,Is_of_type t) ->
  (match can_con con t with

      Prd(x,e1,e2) ->
     let x1 = gensym con x in
     let new = Abs(x1,e1,Meta(n,(x1,Var x1)::(idenv_con con))) in
     (subst_exp [n,new] [] M,
      (2::occ,hyp_ext con x1 e1,Is_of_type (rename x x1 e2),n)::
                                           (subst_tp [n,new] (remove n lt)),
      subst_eq [n,new] leq)
     | _ -> error"is not enough functional")

| _ -> error"intro_anonymous" ;;


let intros n l (M,lt,leq) =

match occ_con_type n lt with

 (occ,con,Is_of_type e) ->
    let (new,t,con1) = build_abs n con l e in
    let occ1 = map (fun x -> 2) l in
    (subst_exp [n,new] [] M,
     (occ1@occ,con1,Is_of_type t,n)::(subst_tp [n,new] (remove n lt)),
      subst_eq [n,new] leq)

| _ -> error"intro_anonymous" ;;



let intro_with_names n l (e,lt,leq) =

intros n l (e,lt,leq) ;;


(* refinement tactic *)

let build_exp con occ =

let rec brec nl t e = match can_con con t with

 Prd(x,t1,v) ->
let (n1,nl1) = new_and_list nl in
let v1 = subst con x (Meta(n1,idenv_con con)) v in
let (occ2,l2,t2,M2) = brec nl1 v1 (App(e,Meta(n1,idenv_con con))) in
(1::occ2,(2::occ2,con,Is_of_type t1,n1)::l2,t2,M2)

| _ -> (occ,[],t,e)

in brec ;;

let refine n x (e,lt,leq) = match occ_con_type n lt with

 (occ,con,Is_of_type t) ->
let nl = list_meta lt in
let (_,l,t1,c) = build_exp con occ nl (typage_ident x con) (Var x) in
(subst_exp [n,c] [] e,l@(subst_tp [n,c] (remove n lt)),
 (map (fun x -> (occ,x)) (eq_simpl con t t1))@(subst_eq [n,c] leq))

| _ -> error((string_of_num n)^" is of type a sort") ;;


let build_exp_num con occ =

let rec brec k nl t e = if k = 0 then (occ,[],t,e) else
```

```
match can_con con t with

 Prd(x,t1,v) ->
let (n1,nl1) = new_and_list nl in
let v1 = subst con x (Meta(n1,idenv_con con)) v in
let (occ2,l2,t2,M2) = brec (k-1) nl1 v1 (App(e,Meta(n1,idenv_con con))) in
(1::occ2,(2::occ2,con,Is_of_type t1,n1)::l2,t2,M2)

| _ -> (occ,[],t,e)

in brec ;;

let refine_num k n x (e,lt,leq) = match occ_con_type n lt with

 (occ,con,Is_of_type t) ->
let nl = list_meta lt in
let (_,l,t1,c) = build_exp_num con occ k nl (typage_ident x con) (Var x) in
(subst_exp [n,c] [] e,l@(remove n lt),
 (map (fun x -> (occ,x)) (eq_simpl con t t1))@(subst_eq [n,c] leq))

| _ -> error((string_of_num n)^" is of type a sort") ;;



let change_constraints n e (e1,lt1,leq1) =
let f con x =
if type_simpl lt1 ([],con,[],isU,e) = []
& are_equal con e x then e else error"change" in
(e1,apply_subgoal n f lt1,leq1) ;;


let unfold_goal n occ (e1,lt1,leq1) =
(e1,apply_subgoal n (unfold occ) lt1,leq1) ;;

(* try to simplify an equation of the form c(e1,...,en) = c(f1,...,fn) *)

let elim_simpl_eq ((occ,con,e1,e2) as c) =

let rec srec le = function

 (Var x,Var y) -> if x = y then le else error"srec"

|(App(u1,v1),App(u2,v2)) -> srec ((v1,v2)::le) (u1,u2)

| _ -> error"srec"

in (try list_eq_simpl (map (fun x -> (occ,con,x)) (srec [] (e1,e2)))
    with Error _ -> [c]) ;;

let try_simpl_elim (e,lt,leq) = (e,lt,flat (map elim_simpl_eq leq)) ;;


(* computation of the type and the context of one occurence *)

let type_and_con_occ f occ e =

let rec urec (occ,occ0,con,e) = match (occ,e) with

 ([],_) -> (con,try_type occ0 f con e)

|(2::occ1,Abs(x,u1,e1)) -> urec(occ1,2::occ0,hyp_ext con x u1,e1)

|(2::occ1,Prd(x,e1,e2)) -> urec(occ1,2::occ0,hyp_ext con x e1,e2)
```

```
|(1::occ1,Prd(x,e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(1::occ1,Abs(x,e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(1::occ1,Let(x,e1,e2,e3)) -> urec(occ1,1::occ0,con,e1)
|(2::occ1,Let(x,e1,e2,e3)) -> urec(occ1,2::occ0,con,e2)
|(3::occ1,Let(x,e1,e2,e3)) -> urec(occ1,3::occ0,def_ext con x e1 e2,e3)
|(1::occ1,App(e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(2::occ1,App(e1,e2)) -> urec(occ1,2::occ0,con,e2)
| _ -> error"this case is not allowed by type_occ"
in urec(occ,[],[],e) ;;
let con_occ occ e =
let rec urec (occ,occ0,con,e) = match (occ,e) with
 ([],_) -> con
|(2::occ1,Abs(x,u1,e1)) -> urec(occ1,2::occ0,hyp_ext con x u1,e1)
|(2::occ1,Prd(x,e1,e2)) -> urec(occ1,2::occ0,hyp_ext con x e1,e2)
|(1::occ1,Prd(x,e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(1::occ1,Abs(x,e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(1::occ1,Let(x,e1,e2,e3)) -> urec(occ1,1::occ0,con,e1)
|(2::occ1,Let(x,e1,e2,e3)) -> urec(occ1,2::occ0,con,e2)
|(3::occ1,Let(x,e1,e2,e3)) -> urec(occ1,3::occ0,def_ext con x e1 e2,e3)
|(1::occ1,App(e1,e2)) -> urec(occ1,1::occ0,con,e1)
|(2::occ1,App(e1,e2)) -> urec(occ1,2::occ0,con,e2)
| _ -> error"this case is not allowed by con_occ"
in urec(occ,[],[],e) ;;
```

```
(* concrete syntax *)

type expc =
 Varc of string
|Appc of expc * expc
|Metac of num
|Sortc of sort
|Qm
|Prdc of string * expc * expc
|Arrowc of expc * expc
|Letc of string * expc * expc * expc
|Absc of string * expc ;;

let rec traduc = function
 Varc x -> Var x
|Appc(e1,e2) -> App(traduc e1,traduc e2)
| _ -> error"is not allowed here" ;;

let S = ref((Sort (Type(0)),(Sort Prop,[],[])):exp * goal) ;;

let get_goal () = fst !S ;;

let get_proof () = fst (snd !S) ;;

let get_constraints () = snd (snd !S) ;;

let get_typs () = fst (get_constraints());;

let get_eqs ()  = snd (get_constraints());;

let typec_error x = error"the term is not well-typed" ;;

let rnc x x1 e = if x = x1 then e else nrec e where rec nrec e = match e with
 Varc y -> if y = x then Varc x1 else e
|Appc(e1,e2) -> Appc(nrec e1,nrec e2)
|Arrowc(e1,e2) -> Arrowc(nrec e1,nrec e2)
|Absc(y,e2) -> if y = x then e else Absc(y,nrec e2)
|Prdc(y,e1,e2) -> if y = x then Prdc(y,nrec e1,e2) else Prdc(y,nrec e1,nrec e2)
|Letc(y,e1,e2,e3) -> if y = x then Letc(y,nrec e1,nrec e2,e3) else
Letc(y,nrec e1,nrec e2,nrec e3)
| _ -> e ;;

let rec expc_check f (nl,occ,con,lt,leq,e) = match e with

 Varc x   -> (can_con con (typage_ident x con),Var x,lt,leq,nl)

|Appc(e1,e2) ->
 (match expc_check f (nl,1::occ,con,lt,leq,e1) with
   (Prd(x,u1,v1),M1,lt1,leq1,nl1) ->
   let M2,lt2,leq2,nl2 = typingc_check f (nl1,2::occ,con,lt1,leq1,u1,e2) in
   (can_subst con x M2 v1,App(M1,M2),lt2,leq2,nl2)
  | _ -> typec_error(occ,e))

|Metac n -> (check_meta f occ n,Meta(n,idenv_con con),lt,leq,nl)

| _ -> error"cannot be typed"

and typingc_check f (nl,occ,con,lt,leq,t,e) =
match (can_con con t,e) with

 (_,Varc x) -> (Var x,lt,(occ,con,t,typage_ident x con)::leq,nl)

|(_,Appc(e1,e2)) ->
 (match expc_check f (nl,1::occ,con,lt,leq,e1) with
   (Prd(x,u1,v1),M1,lt1,leq1,nl1) ->
let M2,lt2,leq2,nl2 = typingc_check f (nl1,2::occ,con,lt1,leq1,u1,e2) in
```

```
(* the type-checking algorithm *)

(* input: an occurence "backward", the context, the type, or a condition of
   typing, a list of pairs (occ,constraint)

   output: a list of pairs (occ,constraint)

and recursively, we call an algorithm that has as

   input: an occurence "backward", the context, a list of pairs
   (occ,constraint)

   output: a type and an updated list of pairs (occ,constraint)

   and these programs may fail with an occurence "backward", the term *)

(* we have a function that checks the place-holders and compute their types *)

exception Type_error of occurence * exp ;;
exception Type_eq_error of exp * exp ;;

let type_error (occ,e) = raise Type_error(occ,e);;
let type_eq_error e = raise Type_eq_error e ;;

type subgoal =
 Is_of_type of exp
|Is_type of cond ;;

type condition_type == subgoal * num ;;

type condition_eq   == occurence * context * exp * exp ;;

type eq_constraints == condition_eq list ;;

type type_constraints == (occurence * context * condition_type) list ;;

type constraints == type_constraints * eq_constraints ;;

(* the output is a type and a list of eq_constraints *)

let occ_con_type n =
let rec orec = function
 [] -> error((string_of_num n)^" is not a goal")
|(occ,con,typ,s)::rest -> if s = n then (occ,con,typ) else orec rest
in orec ;;

let check_meta lt occ n =
match occ_con_type n lt with
 (occ1,con,Is_of_type(e)) ->
if extend (rev occ) (rev occ1) then e
else type_error(occ,Meta(n,idenv_con con))
| _ -> error("?"^(string_of_num n)^" has no types in this local context") ;;

let rec exp_check lt (occ,con,leq,e) = match e with

 Var x -> (typage_ident x con,leq)

|Meta(p,env) -> (inst(symbs con,env,check_meta lt occ p),leq)

|App(e1,e2) ->
 let (t1,leq1) = exp_check lt  (1::occ,con,leq,e1) in
 (match can_con con t1 with
   Prd(x,u1,v1) -> (subst con x e2 v1,typing_check lt (2::occ,con,leq1,u1,e2))
  | _ -> type_error(occ,App(e1,e2)))

|Abs(x,e1,e2) ->
```

```
            (App(M1,M2),lt2,(occ,con,t,subst con x M2 v1)::leq2,nl2)
    | _ -> typec_error(occ,e))

  |(_,Qm) ->
  let (n1,nl1) = new_and_list nl in
  (Meta(n1,idenv_con con),(occ,con,Is_of_type t,n1)::lt,leq,nl1)

  |(_,Metac n) ->
   (Meta(n,idenv_con con),lt,(occ,con,check_meta f occ n,t)::leq,nl)

  |(Prd(y,u1,v),Absc(x,e1)) ->
  let x1 = gensym con x in
  let M1,lt1,leq1,nl1 =
  typingc_check f
  (nl,2::occ,hyp_ext con x1 u1,lt,leq,rename y x1 v,rnc x x1 e1) in
  (Abs(x1,u1,M1),lt1,leq1,nl1)

  |(Sort s,Prdc(x,e1,e2)) ->
  let x1 = gensym con x in
  let M1,lt1,leq1,nl1 =
      typec_check f (nl,1::occ,con,lt,leq,pred_of_sort s,e1) in
  let M2,lt2,leq2,nl2 =
  typingc_check f (nl1,2::occ,hyp_ext con x1 M1,lt1,leq1,Sort s,rnc x x1 e2)
  in (Prd(x1,M1,M2),lt2,leq2,nl2)

  |(Sort s,Arrowc(e1,e2)) ->
  let x1 = gensym con "h" in
  let M1,lt1,leq1,nl1 =
      typec_check f (nl,1::occ,con,lt,leq,pred_of_sort s,e1) in
  let M2,lt2,leq2,nl2 =
      typingc_check f (nl1,2::occ,con,lt1,leq1,Sort s,e2) in
  (Prd(x1,M1,M2),lt2,leq2,nl2)

  |(_,Letc(x,e1,e2,e3)) ->
  let x1 = gensym con x in
  let M2,lt2,leq2,nl2 =
      typec_check f (nl,2::occ,con,lt,leq,isU,e2) in
  let M1,lt1,leq1,nl1 =
      typingc_check f (nl2,1::occ,con,lt2,leq2,M2,e1) in
  let M3,lt3,leq3,nl3 =
  typingc_check f (nl1,3::occ,def_ext con x1 M2 M1,lt1,leq1,t,rnc x x1 e3) in
  (Let(x1,M1,M2,M3),lt3,leq3,nl3)

  |(Sort s2,Sortc s1) ->
  if satisfy (pred_of_sort s2) s1 then Sort s1,lt,leq,nl else typec_error(occ,e)

  | _ -> typec_error(occ,e)

and typec_check f (nl,occ,con,lt,leq,c,e) = match e with

   Sortc s ->
  if satisfy c (type_of_sort s) then Sort s,lt,leq,nl else typec_error(occ,e)

  |Qm ->
  let (n1,nl1) = new_and_list nl in
  (Meta(n1,idenv_con con),(occ,con,Is_type c,n1)::lt,leq,nl1)

  |Varc x  ->
  (match can (list_vals con) (typage_ident x con) with
    Sort s -> if satisfy c s then Var x,lt,leq,nl else typec_error(occ,e)
   | _ -> typec_error(occ,e))

  |Prdc(x,e1,e2) ->
  let x1 = gensym con x in
  let M1,lt1,leq1,nl1 =
      typec_check f (nl,1::occ,con,lt,leq,pred c,e1) in
```