

# Examen du 17 septembre 2003 - dure 2h

Tous documents, autres que les notes de cours et les documents distribus en cours, sont formellement interdits

## Exercice 1 (6 pts)

1. Dmontrer que la fonction dfinie sur les entiers naturels par

$$f(x, y, z) = \max(|x - y|, |y - z|)$$

est une fonction calculable.

2. On rappelle que l'ensemble des fonctions calculables est effectivement dnombrable. On choisit un dnombrement et on note  $\phi_n$  la fonction calculable de rang  $n$  dans ce dnombrement. Construire une fonction  $f$  de  $N$  dans  $N$ , non calculable, telle que  $f(5) = 10$  et  $f(6)$  est indefini ( $f(6) = \perp$ ). Vous pourrez reprendre la construction vue en cours et la modifier (lgrement!) pour construire  $f$ .

## Exercice 2 (7 pts)

L'exercice porte sur le travail effectu avec l'article "Security properties of type applets". Le langage, la dfnition de ce qu'est une valeur, la smantique big-step sont donnees en annexe du texte de l'examen.

Les fonctions  $\phi$  permettent d'implanter une politique de securit. On rappelle que  $l$  est une adresse-mmoire et  $\phi(l)$  un ensemble de valeurs stockables l'adresse  $l$ . On notera si besoin l'ensemble vide par **Vide**.

1. Soit  $E_1$  un environnement dfini par  $E_1 = [A \leftarrow lA]$  et une mmoire  $M_1$  (store) dfinie par  $M_1 = [lA \leftarrow 10]$ .

Dfinir une fonction  $\phi$ , note  $\phi_1$ , qui autorise l'criture d'un nombre entier entre 1 et 6 l'adresse  $lA$  et aucune criture une adresse diffrente de  $lA$ . Justifier votre rponse.

Dfinir une expression dont l'valuation conduise **err** lorsque la smantique est dcrite avec  $\phi_1$ .

2. On considere maintenant l'environnement  $E_2 = [A \leftarrow lA, X \leftarrow lX]$  et la mmoire  $M_2 = [lA \leftarrow 10, lX \leftarrow lA]$

Soit le terme  $P \equiv (\lambda z. !X := !A + z;)$ .

a - Effectuer en Ocaml les dclarations ncessaires la cration de l'tat  $E_2, M_2$ . Dfinir alors une fonction Ocaml, nomme  $fP$ , implantant le terme  $P$ . Expliquer rapidement l'valuation de  $fP(25)$ .

b - Le terme  $S$  est dfini par  $S = P (!(!X))$ .

Dfinir une fonction  $\phi$ , note  $\phi_2$ , et un environnement  $E_3$ , liant les mmes identificateurs que  $E_2$ , tels que :

$$\phi_2, E_3, M_2 \vdash S \rightarrow () M_3$$

telle que  $M_3 = [lA \leftarrow 45, lX \leftarrow lA]$ .

Vous justifierez soigneusement votre rponse.

c - Dfinir une fonction  $\phi$ , note  $\phi_3$ , et un environnement  $E_4$ , liant les mmes identificateurs que  $E_2$ , tels que :

$$\phi_3, E_4, M_2 \vdash S \rightarrow \mathbf{err}$$

Vous justifierez soigneusement votre rponse.

On considère dans ce qui suit le langage d'expressions fonctionnelles implémenté dans le TME sur le typage et dont la syntaxe abstraite est définie dans le fichier `syntaxe_abstraite.mli`. On se propose d'étendre le langage des expressions en ajoutant une construction permettant de considérer des paires (i.e., des expressions de la forme  $(e_1, e_2)$  où  $e_1$  et  $e_2$  sont des expressions). Pour répondre aux questions qui suivent, il suffit de compléter les définitions des fonctions sur le code source qui vous est fourni à la fin de ce document. Les fonctions appelées dont la définition n'apparaît pas dans le code fourni sont exactement celles qui ont été définies en TME.

1. Ajouter un constructeur au type `expr` correspondant aux paires d'expressions.
2. Ajouter un constructeur au type `typ` afin de pouvoir exprimer le type des paires d'expressions.
3. Modifier les fonctions relatives à l'unification de manière à prendre en compte l'extension réalisée.

```

type decl = Ldef of (string * expr)
and expr =
  Lvar of string                (* variables *)
| Lconst of immediat           (* constantes *)
| Lconditionnelle of expr * expr * expr (* conditionnelle *)
| Lbinop of binop * expr * expr (* operateurs binaires *)
| Lsoit of string * expr * expr (* de'finitions locales *)
| Lfonction of string * expr    (* fonctions *)
| Lapp of expr * expr          (* applications *)
| Lrec of string * string * expr (* re'cursions *)
| ??? A COMPLETER ???         (* paires *)

and immediat = Lint of int | Lbool of bool
and binop = ...
and typ =
  Tvar of var_type             (* variable de type *)
| Tint                         (* entier *)
| Tbool                        (* boole'en *)
| Tfleche of typ * typ        (* fonction *)
| ??? A COMPLETER ???         (* paires *)

and var_type = int
and schema_de_type =
  Tpourtout of var_type list * typ (* sche'ma de type *)
;; (* repre'sente \ / a1,...,an.ty *)

(** fichier unification.ml **)
exception Unification;;

(* ge'ne'ration de symboles *)
let gen_var_type,reinit_var_de_type = ...

(* variables libres d'un type *)
let vars env_liees env_libres t =
  let rec vars env_libres t =
    match t with
    Tvar(x) ->
      if mem x env_liees or mem x env_libres then env_libres
      else x::env_libres
    | (Tint | Tbool) -> env_libres
    | Tfleche(t1,t2) -> vars (vars env_libres t1) t2 in
  vars env_libres t;;
  | ??? A COMPLETER ???

let var_libres t = vars [] [] t;;

(* variables libres dans un environnement de type *)
let var_libres_env env_type = ...

(* appliquer une substitution de type *)
let substituer subst_type t =

```

```

| (Tint | Tbool) -> t
| Tfleche(t1,t2) -> Tfleche(sub t1,sub t2) in sub t;;
| ??? A COMPLETER ???

(* appliquer une substitution de type a' un environnement de type *)
let substituer_env subst_type env_type = ...

(* ajouter une substitution (v,ty) a' une substitution existante. On doit *)
(* "saturer" la substitution existante avec la nouvelle, c'est a' dire *)
(* remplacer toutes les occurrences de v par ty *)
let compose_substitution (v,ty) subst_type = ...

(* produit une substitution *)
let rec unifier subst_type t1 t2 =
  match t1,t2 with
  | Tvar(x1),Tvar(x2) when x1 = x2 -> subst_type
  | Tvar(x1),t2 ->
    begin
      try
        unifier subst_type (assoc x1 subst_type) t2
      with
      | Not_found ->
        let t2 = substituer subst_type t2 in
        begin match t2 with
          | Tvar(x2) when x1 = x2 -> subst_type
          | t2 ->
            if mem x1 (var_libres t2) then raise Unification
            else compose_substitution (x1,t2) subst_type
        end
      end
    end
  | t1,Tvar(x2) ->
    begin
      try
        unifier subst_type t1 (assoc x2 subst_type)
      with
      | Not_found ->
        let t1 = substituer subst_type t1 in
        begin match t1 with
          | Tvar(x1) when x1 = x2 -> subst_type
          | t1 ->
            if mem x2 (var_libres t1) then raise Unification
            else compose_substitution (x2,t1) subst_type
        end
      end
    end
  | ((Tint,Tint) | (Tbool,Tbool)) -> subst_type
  | Tfleche(t1,t2),Tfleche(t'1,t'2) ->
    unifier (unifier subst_type t1 t'1) t2 t'2
  | ??? A COMPLETER ???

```

```

(* ge'ne'raliser un type *)
let generaliser env_type (ty,subst_type) = ...

(* instantiation d'un type ge'ne'ralise' *)
let instancier (Tpourtout(l,ty)) =
  let nouvelles_variables =
    map (fun n -> (n,Tvar(gen_var_type ()))) l in
  let rec instrec ty =
    match ty with
    | Tvar(n) ->
      begin try
        assoc n nouvelles_variables
      with
        | Not_found -> ty
      end
    | (Tbool | Tint) as ty -> ty
    | Tfleche(ty1,ty2) -> Tfleche(instrec ty1,instrec ty2) in
  instrec ty;;
  | ??? A COMPLETER ???

```