Lundi 8 janvier 2007
Durée : 3 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master 1 Informatique 2006-2007

Corrigé préliminaire.

Examen

Remarque : *l'indication des points par exercice, indicative de l'évaluation à venir, est susceptible d'être modifiée.*

Unification

Exercice 1. (3 points) Pour chaque problème d'unification

1. `[ ( f(g(k(x)),y) , f(y,g(x)) ) ]`

   Solution.

$$\begin{aligned}
[(f(g(k(x)),y),f(y,g(x)))] &\rightsquigarrow [(g(k(x)),y);(y,g(x))] \\
&\rightsquigarrow [(g(k(x)),g(x))] && \sigma_0 = \{y \to g(k(x))\} \\
&\rightsquigarrow [(k(x),x)]
\end{aligned}$$

   L'algorithme s'arrête ici car x est une variable libre dans $k(x)$. □

2. `[ ( f(g(x),x) , f(y,g(z)) ) ; ( g(x) , y ) ]`

   Solution.

$$\begin{aligned}
[(f(g(x),x),f(y,g(z)));(g(x),y)] &\rightsquigarrow [(g(x),y);(g(z),x);(g(x),y)] \\
&\rightsquigarrow [(g(z),x);(g(x),g(x))] && \sigma_0 = \{y \to g(x)\} \\
&\rightsquigarrow [(g(g(z)),g(g(z)))] && \sigma_1 = \{x \to g(z)\} \\
&\rightsquigarrow [(g(z),g(z))] \\
&\rightsquigarrow [(z,z)] \\
&\rightsquigarrow [\,]\,.
\end{aligned}$$

   A ce point l'algorithme retournera la substitution identité ι. Cette substition sera composée avec σ_1 et σ_0 pour donner comme résultat final

$$\iota \circ \sigma_1 \circ \sigma_0 = \left\{ \begin{array}{cccccc} y & \to^{\sigma_0} & g(x) & \to^{\sigma_1} & g(g(z)), \\ x & \to^{\sigma_0} & x & \to^{\sigma_1} & g(z) \end{array} \right\}$$

   □

3. `[ ( f(y,k(y),g(x)) , f(k(x),k(y),y) ) ]`

   Solution.

$$\begin{aligned}
[(f(y,k(y),g(x)),f(k(x),k(y),y))] &\rightsquigarrow [(y,k(x));(k(y),k(y));(g(x),y)] \\
&\rightsquigarrow [(k(k(x)),k(k(x)));(g(x),k(x))] && \sigma_0 = \{x \to k(x)\} \\
&\rightsquigarrow [(k(x),k(x));(g(x),k(x))] \\
&\rightsquigarrow [(x,x);(g(x),k(x))] \\
&\rightsquigarrow [(g(x),k(x))]
\end{aligned}$$

   L'algorithme s'arrête ici car les symboles de fonction de $g(x), k(x)$ sont différents. □

détaillez le déroulement de l'algorithme d'unification sur le problème.

Exercice 2. (2 points)

1. Expliquez ce que veut dire qu'une substitution est un unificateur principal d'un problème d'unification.

   Solution. Soit

   $$P = [(t_1, s_1); \ldots; (t_n; s_n)]$$

   Une substitution σ est un unificateur de P si $\sigma t_i = \sigma s_i$ pour $i = 1, \ldots, n$.
   Une substitution σ est un unificateur principal de P si σ est un unificateur de P et pour tour unificateur τ de P on a $\tau = \sigma' \circ \sigma$ pour une autre substitution σ'. □

2. Construisez un problème d'unification et un unificateur non principal de ce problème d'unification.

   Solution. Le problème

   $$P = [(x, x)]$$

   possède comme unificateur $\sigma = \{x \to f(y)\}$. Cette substitution n'est pas unificateur principal car la substitution identité ι est un unificateur mais on ne peut pas écrire $\iota = \sigma' \circ \sigma$. □

3. Peut on avoir deux unificateurs principaux d'un même problème ? Donnez éventuellement un exemple.

   Solution. Oui. Considérons le problème précédent $P = [(x, x)]$. Alors l'identité et la substitution $\sigma = \{x \to y, y \to x\}$ sont deux unificateurs principaux de P. □

Le calcul de la résolution

Exercice 3. (4 points) Considérez les deux clauses suivantes :

$$P(f(x), y) \Rightarrow Q(f(x)), Q(y) \tag{1}$$
$$Q(z) \Rightarrow P(z, x) \tag{2}$$

Appliquez, de toute façon possible, les règles du calcul de la résolution à ces deux clauses.

Solution.

$$\frac{P(f(x), y) \Rightarrow \overbrace{Q(f(x))}^{A}, \overbrace{Q(y)}^{B}}{P(f(x), f(x)) \Rightarrow Q(f(x))}\, Fac$$

où on a utilisé l'unificateur principal $\{y \to f(x)\}$.

$$\frac{P(f(x), y) \Rightarrow \overbrace{Q(f(x))}^{A}, Q(y) \quad \overbrace{Q(z)}^{B} \Rightarrow P(z, x)}{P(f(x), y) \Rightarrow Q(y), P(f(x), x)}\, Res$$

où on a utilisé l'unificateur principal $\{z \to f(x)\}$.

$$\frac{P(f(x), y) \Rightarrow Q(f(x)), \overbrace{Q(y)}^{A} \quad \overbrace{Q(z)}^{B} \Rightarrow P(z, x)}{P(f(x), y) \Rightarrow Q(f(x)), P(y, x)}\, Res$$

où on a utilisé l'unificateur principal $\{z \to y\}$.

$$\frac{Q(z) \Rightarrow \overbrace{P(z, x)}^{A} \quad \overbrace{P(f(x), y)}^{B} \Rightarrow Q(f(x)), Q(y)}{Q(f(x)) \Rightarrow Q(f(y)), Q(y)}\, Res$$

où on a utilisé l'unificateur principal $\{z \to f(x), x \to y\}$. □

Exercice 4. (2 points) Considérez la théorie suivante :

$$R(x, y) \Rightarrow R(x, g(y)) \tag{3}$$
$$\Rightarrow R(0, 0) \tag{4}$$

1. Que se passe-t-il si on démarre l'algorithme `saturer` avec cette théorie en entrée ?

   Solution. L'algorithme `saturer` engendrera, l'une après l'autre, toutes les clauses de la forme $\Rightarrow R(0, g^n(0))$ et de la forme $R(x,y) \Rightarrow R(x, g^n(y))$. En particulier l'algorithme ne s'arrêtera jamais. □

2. Est ce que cette théorie est cohérente ? Justifiez votre réponse en fonction de votre réponse à la question précédente (4.1).

   Solution. Étant donné que l'algorithme ne s'arrête jamais, l'ensemble des clauses à l'origine est cohérent. En effet, la clause vide ne sera engendrée par l'algorithme, et car l'algorithme et le calcul de la résolution sont complets, la théorie est cohérente. □

Sémantique

Exercice 5. (3 points)

1. Rappelez la définition de équivalence entre commandes du langage IML que nous avons proposé à l'aide de la sémantique opérationnelle.

   Solution. Nous avons definie une relation ternaire $\rightarrow \subseteq \mathcal{C}om \times \mathcal{S} \times \mathcal{S}$, de la forme $(c,\sigma) \rightarrow \sigma'$ pour dire que l ;'execution de la commande c amène la machine de l'etat σ à l'état σ'. Nous avons ainsi défini

   $$c \sim c' \text{ ssi } \forall \sigma, \sigma' (c,\sigma) \rightarrow \sigma' \text{ iff } (c',\sigma) \rightarrow \sigma' \ .$$

   □

2. A l'aide de la sémantique opérationnelle, prouvez que, pour toute expression Booléenne `b` et pour tout couple de commandes `c0`,`c1`, la commande

   ```
   if b then c0 else c1
   ```

   est equivalente à la commande

   ```
   if not b then c1 else c0
   ```

   Solution. Supposons que $(\texttt{if b then c0 else c1}, \sigma) \rightarrow \sigma'$. Nous avons remrqué (en TD) que ou bien $(\texttt{b}, \sigma) \rightarrow 1$, ou bien $(\texttt{b}, \sigma) \rightarrow 0$ (et non tous les deux cas.
   Si $(\texttt{b}, \sigma) \rightarrow 1$ alors on peut avoir dérivé la relation $(\texttt{if b then c0 else c1}, \sigma) \rightarrow \sigma'$ de cette façon seulement :

   $$\dfrac{\dfrac{\vdots}{(\texttt{b},\sigma) \rightarrow 1} \quad \dfrac{\vdots}{(\texttt{c0},\sigma) \rightarrow \sigma'}}{(\texttt{if b then c0 else c1},\sigma) \rightarrow \sigma'}$$

   D'ici on peut dériver

   $$\dfrac{\dfrac{\dfrac{\vdots}{(\texttt{b},\sigma) \rightarrow 1}}{(\texttt{not b},\sigma) \rightarrow 0} \quad \dfrac{\vdots}{(\texttt{c0},\sigma) \rightarrow \sigma'}}{(\texttt{if not b then c1 else c0},\sigma) \rightarrow \sigma'}$$

   Si $(\texttt{b}, \sigma) \rightarrow 0$ alors on peut avoir dérivé la relation $(\texttt{if b then c0 else c1}, \sigma) \rightarrow \sigma'$ seulement de cette façon :

   $$\dfrac{\dfrac{\vdots}{(\texttt{b},\sigma) \rightarrow 0} \quad \dfrac{\vdots}{(\texttt{c1},\sigma) \rightarrow \sigma'}}{(\texttt{if b then c0 else c1},\sigma) \rightarrow \sigma'}$$

   D'ici on peut dériver

   $$\dfrac{\dfrac{\dfrac{\vdots}{(\texttt{b},\sigma) \rightarrow 0}}{(\texttt{not b},\sigma) \rightarrow 1} \quad \dfrac{\vdots}{(\texttt{c1},\sigma) \rightarrow \sigma'}}{(\texttt{if not b then c1 else c0},\sigma) \rightarrow \sigma'}$$

   Cela montre que $(\texttt{if b then c0 else c1}, \sigma) \rightarrow \sigma'$ implique $(\texttt{if not b then c1 else c0}, \sigma) \rightarrow \sigma'$.
   On démontre l'implication inverse de façon analogue. □

Le langage OCaml

Exercice 6 : listes infinies d'entiers. (4 points) Considérez le code suivant :

```
1  type ilist = Pair of int*(unit->ilist);;
2  let hd (Pair (t,q)) = t;;
3  let tl (Pair (t,q)) = q ();;
4  let rec map2 f l1 l2 = Pair (f (hd l1) (hd l2), (fun () -> map2 f (tl l1) (tl l2)));;
5  let rec
6    f1 = Pair (0,(fun () -> f2))
7  and
8    f2 = Pair (1,fun () -> (map2 (fun x y  -> x +y) f1 f2));;
9  let rec take n l = if n <= 0 then [] else (hd l)::(take (n-1) (tl l));;
10 let fst10 = take 10 f1;;
```

1. Calculez le type de chaque valeur définit.
2. Quel est la valeur de l'expression `fst10` ?

   Solution.

```
# type ilist = Pair of int*(unit->ilist);;
type ilist = Pair of int * (unit -> ilist)
# let hd (Pair (t,q)) = t;;
val hd : ilist -> int = <fun>
# let tl (Pair (t,q)) = q ();;
val tl : ilist -> ilist = <fun>
# let rec map2 f l1 l2 = Pair (f (hd l1) (hd l2), (fun () -> map2 f (tl l1) (tl l2)));;
val map2 : (int -> int -> int) -> ilist -> ilist -> ilist = <fun>
# let rec
  f1 = Pair (0,(fun () -> f2))
and
  f2 = Pair (1,fun () -> (map2 (fun x y  -> x +y) f1 f2));;
      val f1 : ilist = Pair (0, <fun>)
val f2 : ilist = Pair (1, <fun>)
# let rec take n l = if n <= 0 then [] else (hd l)::(take (n-1) (tl l));;
val take : int -> ilist -> int list = <fun>
# let fst10 = take 10 f1;;
val fst10 : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

   □

3. Décrivez en bref ce qu'il se passe dans ce code. En particulier, expliquez pourquoi on y utilise le type `unit`.

   Solution. Nous avons défini un type `ilist` pour coder les listes infinies d'entiers. Une liste infinie est composée par une tête, un entier, et par une queue, une autre liste infinie. En général, calculer la queue d'une liste infinie peut entraîner dans de calculs infinis. Pour éviter ce résultat nous retardons l'évaluation : c-à-d. nous plaçons l'expression dénotant la queue de la liste sous un `fun`, de façon à transformer cette expression en valeur (c'est ici qu'on se sert du type `unit`). La queue de la liste sera alors calculé de façon explicite quand il se rendra nécessaire.
   Nous définissons alors `hd` et `tl`, comme d'habitude pour le listes, et une fonction `map2` qui applique un fonction de deux arguments à deux listes infinies, éléments par éléments.
   De cette façon nous pouvons donner une définition récursive `f1` de la liste infinie de Fibonacci, dont on affichera les premier dix éléments. □

4. On peut vouloir améliorer la lisibilité du code en introduisant les définitions suivantes :

```
let cons t q = Pair (t, fun ()-> q);;
let rec autre_map2 f l1 l2 = cons  (f (hd l1) (hd l2)) (autre_map2 f (tl l1) (tl l2));;
```

   Que se passe-t'il à l'évaluation de

```
take 10 (autre_map2 (fun x y  -> x +y) f1 f2);;
```

   ?

   Solution. Observons que l'expression `cons t q` force l'évaluation de `q` (évaluation par valeur). De cette façon, nos effort pour retarder les évaluations des queues sont rendus inutiles. L'évaluation de l'expression

```
take 10 (autre_map2 (fun x y  -> x +y) f1 f2);;
```

ne donnera pas de résultats. □

Exercice 7. (5 points) Pour cet exercice vous allez supposer que le module `Terme`, décrit par l'interface

```
(** Type des termes *)
type ('b,'c) t

(** Type des substitutions *)
type ('b,'c) s

(** Appliquer une substitution à un terme *)
val apply : ('b,'c) s -> ('b,'c) t -> ('b,'c) t

(** Calculer un unificateur principal
  d'un problème d'unification *)
val unify : (('b,'c) t * ('b,'c) t) list -> ('b,'c) s
```

est déjà implémenté. De même, vous pouvez utiliser les fonctions des modules que vous connaissez (le module `List` par exemple).

Implémentez le module `Logique` décrit part l'interface :

```
(** Type des formules atomiques *)
type ('a,'b,'c) fa

(** Type des clauses *)
type ('a,'b,'c) clause

(** Appliquer une substitution à une formule atomique *)
val apply : ('b,'c) Terme.s ->  ('a,'b,'c) fa ->  ('a,'b,'c) fa

(** Calculer un unificateur principal de deux formules atomiques *)
val unify : ('a,'b,'c) fa -> ('a,'b,'c) fa -> ('b,'c) Terme.s

(** Appliquer la règle de résolution
  à deux clauses avec 2 formules atomiques choisies *)
val resolve :
  (('a,'b,'c) clause*int) -> (('a,'b,'c) clause*int) -> ('a,'b,'c) clause
```

Solution.

```
type ('a,'b,'c) fa = 'a*(('b,'c) Terme.t list) ;;
type ('a,'b,'c) clause =
    {
      gauche : ('a,'b,'c) fa list;
      droite : ('a,'b,'c) fa list;
    };;


let apply sigma (p,ts) =
  (p,List.map (Terme.apply sigma) ts)
;;

let unify (p1,t1s) (p2,t2s) =
  if not (p1 = p2) then
    failwith "unify : prédicats différentes"
  else
    try
      Terme.unify (List.combine t1s t2s)
    with
        Invalid_argument "List.combine"
        -> failwith "unify : arités différentes"
      | _ -> failwith "unify : pas d'unificateur"
```

```
;;

(** Effacer l'élement i de la liste l *)
let list_remove l i =
   if i < 0 then
     raise (Invalid_argument "list_remove")
   else
     let rec
       list_remove_rec l j = match l with
           [] -> []
         | t::q -> if j = 0 then q else
             t::(list_remove_rec q (j-1))
     in
       list_remove_rec l i
;;

let resolve (c1,i) (c2,j) =
  let
    sigma = unify (List.nth c1.droite i) (List.nth c2.gauche j)
  in
    {
      gauche = List.map (apply sigma)
                 (c1.gauche@(list_remove c2.gauche i));
      droite = List.map (apply sigma)
                 ((list_remove c1.droite j)@c2.droite)
    }
;;
```

□