Lundi 9 janvier 2006
Durée : 3 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master 1 Informatique 2005-2006

Examen

Unification et résolution (7 points)

Exercice 1. (2 points) Est ce que les couples

1. $(f(z), z)$,

   Solution. Non, car la variable z apparait dans $f(z)$. □

2. $(g(x, y), g(y, x))$,

   Solution. Oui, car la substitution $\sigma = \{y \to x\}$ unifie les deux termes. □

3. $(h(x, y), g(y, x))$,

   Solution. No, car les deux termes débutent avec deux symboles de fonction différents. □

4. $(h(x, y, z), h(x, f(w, u)))$.

   Solution. No, car le symbole de fonction h est utilisé d'un coté avec arité 3, de l'autre coté avec arité 2. □

sont unifiables ? Justifiez brièvement votre réponse.

Exercice 2. (2 points) Décrivez, étape par étape, le déroulement de l'algorithme d'unification avec en entrée les problèmes suivants :

1. $[\,(f(w, g(h(z))), f(g(z), w))\,]$,

   Solution.

$$\begin{array}{rl} & (f(w, g(h(z))), f(g(z), w)) \\ & (w, g(z))(g(h(z)), w) \\ \sigma_1 : \{w \to g(z)\} & (g(h(z)), g(z)) \\ & (h(z), z) \end{array}$$

   L'algorithme s'arrête à ce point, le couple n'est pas unifiable. □

2. $[\,(h(f(y), z), h(x, g(x, w)))\,]$.

   Solution.

$$\begin{array}{rl} & (h(f(y), z), h(x, g(x, w))) \\ & (f(y), x)(z, g(x, w)) \\ \sigma_1 : \{x \to f(y)\} & (z, g(f(y), w)) \\ \sigma_2 : \{z \to g(f(y), w)\} & \end{array}$$

   L'algorithme s'arrête à ce point, et retourne :

$$\sigma_2 \circ \sigma_1 : \{x \to f(y),\, z \to g(f(y), w)\}\,.$$

   □

Exercice 3. (3 points) Considérez un langage contenant le symbole de fonction unaire s, la constante 0 et le prédicat binaire P.

1. Utilisez le calcul de la résolution pour montrer que la théorie composée par les trois clauses

$$\begin{aligned} P(x,0) &\Rightarrow && (1)\\ P(s(x),s(y)) &\Rightarrow P(x,y) && (2)\\ &\Rightarrow P(s(s(s(0))),s(s(0))) && (3)\end{aligned}$$

n'est pas cohérente.

Solution.

$$\begin{aligned} &\Rightarrow P(s(s(0)),s(0)) && (4,\text{ Rés, } 3.1,2.1)\\ &\Rightarrow P(s(0),0) && (5,\text{ Rés, } 4.1,2.1)\\ &\Rightarrow && (6,\text{ Res, } 5.1,1.1)\end{aligned}$$

□

2. Comment peut-on se servir du calcul de la résolution pour montrer que la théorie composée par les deux clauses

$$\begin{aligned} &\Rightarrow P(0,s(0)) && (1)\\ P(x,y) &\Rightarrow P(s(x),s(y)) && (2)\end{aligned}$$

est cohérente ?

Solution. Les seules clauses qu'on peut dériver (par itération de la règle de résolution) à partir des clauses 1 et 2 sont de la forme

$$P(x,y) \Rightarrow P(s^n(x),s^n(y)) \qquad\qquad \Rightarrow P(s^n(0),s^{n+1}(0))$$

En particulier, on ne peut pas dériver la clause vide. Le calcul de la résolution étant complet, cette théorie est cohérente. □

Sémantique (6 points)

Exercice 4. (3 points) À partir de deux expressions arithmétiques a_1, a_2 du langage $\mathcal{IML}$, on construit deux commandes c_1, c_2 du langage $\mathcal{IML}$ comme suit :

$$\begin{aligned} c_1 &\equiv\ X := a_1;\ Y := a_2\\ c_2 &\equiv\ Y := a_2;\ X := a_1\end{aligned}$$

1. Proposer deux expressions arithmétiques a_1, a_2 telles que les commandes c_1 et c_2 ne sont pas équivalentes. Se servir de la sémantique formelle pour démontrer que ces deux commandes ne sont pas équivalents.

Solution. Soit $a_1 \equiv Y$, et $a_2 \equiv Y + \hat{1}$. On trouve alors que
– On calcule $\|c_1\|_{\mathcal{C}om}$:

$$\begin{aligned}\|c_1\|_{\mathcal{C}om} &= \{\,(\sigma,\sigma') \mid \exists\tilde{\sigma}\,(\sigma,\tilde{\sigma}) \in \|X := Y\| \text{ et } (\tilde{\sigma},\sigma') \in \|Y := Y + \hat{1}\|\,\}\\ &= \{\,(\sigma,\sigma') \mid (\sigma[\sigma(Y)/X],\sigma') \in \|Y := Y + \hat{1}\|\,\}.\end{aligned}$$

Ces calculs montrent que $\|c_1\|_{\mathcal{C}om}$ est l'ensemble des couples d'états (σ,σ') ou $\sigma'(Y) = \sigma(Y)+1$ et $\sigma'(X) = \sigma(Y)$ (et $\sigma'(Z) = \sigma(Z)$ si $Z \notin \{X,Y\}$).
– On calcule $\|c_2\|_{\mathcal{C}om}$:

$$\begin{aligned}\|c_2\|_{\mathcal{C}om} &= \{\,(\sigma,\sigma') \mid \exists\tilde{\sigma}\,(\sigma,\tilde{\sigma}) \in \|Y := Y + \hat{1}\| \text{ et } (\tilde{\sigma},\sigma') \in \|X := Y\|\,\}\\ &= \{\,(\sigma,\sigma') \mid (\sigma[\sigma(Y)+1/Y],\sigma') \in \|X := Y\|\,\}.\end{aligned}$$

$\|c_2\|_{\mathcal{C}om}$ est donc l'ensemble des couples (σ,σ') ou $\sigma'(X) = \sigma'(Y) = \sigma(Y)+1$ (et $\sigma'(Z) = \sigma(Z)$ si $Z \notin \{X,Y\}$).. Les ensembles $\|c_2\|_{\mathcal{C}om}$ et $\|c_2\|_{\mathcal{C}om}$ étant différents, les commandes c_1 et c_2 ne sont pas équivalentes. □

2. Proposer deux expressions arithmétiques a_1, a_2 telles que les commandes c_1 et c_2 sont équivalentes. Se servir de la sémantique formelle pour démontrer que ces deux commandes sont équivalentes.

   Solution. Soit $a_1 \equiv X$, et $a_2 \equiv Y$. On trouve alors que $\|c_1\|_{\mathcal{C}om} = \|c_2\|_{\mathcal{C}om}$ est l'ensemble des couples de la forme (σ, σ). □

Exercice 5. (3 points)

1. Énoncer le théorème de Tarski.

   Solution. Soit L un treillis complet. Alors toute fonction $f : L \to L$ croissante (ou monotone) possède un point fixe. □

2. Soient P, Q deux treillis complets, et soit $f : P \times Q \longrightarrow P$ une fonction croissante (c.-à-d. monotone) en toutes ses variables. Montrer que la fonction $\mu.f : Q \longrightarrow P$, telle que $\mu.f(q)$ est le plus petit point fixe de la fonction croissante qui envoie $p \in P$ vers $f(p, q)$, est elle même croissante.

   Solution. Soient $q, q' \in Q$ tels que $q \leq q'$. On a

   $$\begin{aligned} f(\mu.f(q'), q) &\leq f(\mu.f(q'), q') && \text{car } f \text{ croissante} \\ &\leq \mu.f(q') && \text{car } \mu.f(q') \text{ est un point préfixe de la fonction } f(\cdot, q'). \end{aligned}$$

   Il en découle que $\mu.f(q')$ est un point préfixe de la fonction $f(\cdot, q)$, et donc il est plus grand de son plus petit point préfixe :

   $$\mu.f(q) \leq \mu.f(q')\,.$$

   □

Le langage Caml (12 points)

Listes infinies

Une liste infinie est composée par une tête et par une queue, la queue étant une liste infinie. Le module `Ilist` permet de faire des calculs avec les listes infinies. Son interface (fichier `ilist.mli`) est :

```
(** Le type des listes infinies *)
type 'a t

(** Renvoie la tête d'une liste *)
val hd : 'a t -> 'a

(** Renvoie la queue d'une liste *)
val tl : 'a t -> 'a t

(** Construit une nouvelle liste,
  à partir de la tête et de la queue *)
val cons : 'a -> 'a t -> 'a t

(** La liste infinie de tous le entiers,
  à partir d'un entier donné *)
val from : int -> int t

(** Sélectionne tous les éléments d'une liste
  satisfaisant un prédicat donné *)
val filter : ('a -> bool) -> 'a t -> 'a t
```

Exercice 6. (3 points) La fonction `fun_x` :

```
let fun_x k =
  let rec fun_x_acc acc seq =
    if List.length acc >= k then  List.rev acc else
      let
        tete = Ilist.hd seq
      and
        queue = Ilist.tl seq
      in
        fun_x_acc (tete::acc)
          (Ilist.filter (fun x -> not (x mod tete = 0)) queue)
  in
    fun_x_acc [] (Ilist.from 2);;
```

se sert du module `Ilist`.

1. Calculez le type de cette fonction.

   Solution.

   ```
   val fun_x : int -> int list = <fun>
   ```

   □

2. Quel est le résultat de l'évaluation de

   ```
   fun_x 5;;
   ```

   ?

   Solution.

   ```
   # fun_x 5;;
   - : int list = [2; 3; 5; 7; 11]
   ```

   □

3. Expliquez ce qui est achevé en général par cette fonction.

   Solution. Cette fonction, avec en entrée l'entier k, calcule la liste finie des premiers k nombres premiers. □

Exercice 7. (4 points) Écrivez une implémentation du module `Ilist`. On prendra garde que les calculs sur les listes infinies ne produisent pas de calculs infinis.

Solution.

```
type 'a t = Cons of 'a * (unit -> 'a t);;

let hd = function
    Cons(t,q) -> t;;

let tl = function
    Cons(t,q) -> q ();;

let cons t q = Cons(t, fun () -> q);;

let rec from k = Cons(k, fun () -> from (k + 1));;

let rec filter p  (Cons(t,q)) =
  if p t then
    Cons(t, fun () -> filter p (q ())) else
      filter p (q ());;
```

□

Types récursifs et forme récursive terminale

Exercice 8. (2 points) Un *tas* est un arbre avec la propriété que chaque noeud est étiqueté par un nombre entier. Un noeud peut avoir aucun, un, ou deux fils.
Un tas est *bien formé* si la propriété suivante est vraie. Soit n l'étiquette d'un noeud x et m l'étiquette d'un noeud y. Si y se trouve dans le sous-arbre à gauche de x alors $m < n$ et s'il se trouve dans le sous-arbre à droite de x alors $n < m$.

1. Proposer une définition du type `tas` dans le langage OCaml.

   Solution.

   ```
   type tas = Feuille | Neoud of int * tas * tas ;;
   ```

   □

2. Définir une fonction `chercher`

   ```
   chercher : int -> tas -> bool
   ```

   qui cherche si un entier est l'étiquette d'un noeud d'un tas bien formé.

   Solution.

   ```
   let rec chercher n  = function
       Feuille -> false
     | Neoud (m,g,d) -> if m = n then true else
         if n < m then chercher n g else
           chercher n d ;;
   ```

   □

Exercice 9. (3 points) La fonction `integrale` suivante

```
let rec integrale f n1 n2 =
  if n1 > n2 then 0
  else
    f n1 + integrale f (n1 + 1) n2 ;;
```

calcule l'intégrale discrète de la fonction f sur l'intervalle $[n1, n2]$.

1. Calculez le type de cette fonction.

   Solution.

   ```
   val integrale : (int -> int) -> int -> int -> int = <fun>
   ```

   □

2. Ré-écrivez cette fonction en forme récursive terminale.

   Solution.

   ```
   let integrale_recterm f n1 n2 =
     let rec integrale_acc f n1 n2 acc =
       if n1 > n2 then acc
       else
         integrale_acc f (n1 + 1) n2 (acc + f n1)
     in
       integrale_acc f n1 n2 0
   ;;
   ```

   □

3. Pour quelle raison la forme récursive terminale est préférable ?

   Solution. Dans la forme récursive non terminale, à chaque appels récursif, on construit une expressions

   $$v1 + (v2 + (v3 \ldots +)) \tag{1}$$

   qu'il faudra évaluer à la fin des appels récursifs. La taille de cette expression est proportionnelle à la taille de l'intervalle, et donc il en découle aussi une utilisation de l'espace mémoire par conséquent.
   La forme récursive terminale force l'évaluation des résultats partiels dans la variable *acc*. L'utilisation de la mémoire est par conséquent limité, car au lieu de stoquer l'expression (1), on stoque seulement son valeur.
   On rappelle aussi qu'après compilation, dans le code assembleur, la récursion terminale est remplacée par une boucle, et une boucle (contrairement aux appels récursifs) n'incrémente pas la dimension de la pile. D'ici, une autre amélioration pour l'utilisation de la mémoire. □

4. Proposez une expérience permettant de vérifier votre dernière réponse.

   Solution. On s'attend d'observer des différences entre les deux implémentations de la fonction *integrale* si on considère des intervaux larges. Par exemple :

   ```
   # let f x = x + 1;;
   val f : int -> int = <fun>
   # let max = 100000;;
   val max : int = 100000
   # integral f 0 max;;
   Stack overflow during evaluation (looping recursion?).
   # integral_recterm f 0 max;;
   - : int = 705182705
   ```

   □