Lundi 10 janvier 2005
Durée : 3 heures
Documents autorisés

Examen : Logique, Déduction et Programmation
Master informatique 2004-2005

Examen

Le langage Caml (9 points)

Exercice 1. (2 points) Considérons le module `Ensemble` : son interface est

ensemble.mli

```
type 'a t;;
val ajouter : 'a -> 'a t -> 'a t ;;
```

et son implémentation est

ensemble.ml

```
type 'a t = 'a list
let rec ajouter a = function
    [] -> [a]
  | t::q -> if t = a then t::q else t::(ajouter a q);;
```

Après l'avoir compilé :

```
$ ocamlc ensemble.mli ensemble.ml
```

on essai de s'en servir :

```
$ ocaml
        Objective Caml version 3.08.0

# #load "ensemble.cmo";;
# Ensemble.ajouter 1 [];;
                     --
This expression has type 'a list but is here used with type int Ensemble.t
#
```

Expliquer la raison du dernier message d'erreur.

Solution. Le problème est du au principe du masquage de l'implémentation et, dans ce cas en particulier, au fait que le type `Ensemble.t` est un type abstrait : dans l'interface `ensemble.mli` on y trouve seulement la déclaration de son existence, et non la déclaration de son implémentation. Par conséquent, en dehors du fichier `ensemble.ml` on a pas le droit a connaître que l'implémentation du type `'a Ensemble.t` est faite à l'aide de listes.
On peut résoudre donc le problème de deux façons :

1. Déclarer dans l'interface l'existence d'un ensemble vide :

   ensemble1.mli

   ```
   type 'a t;;
   val ajouter : 'a -> 'a t -> 'a t ;;
   val empty : 'a t;;
   ```

   ensemble1.ml

   ```
   type 'a t = 'a list
   let rec ajouter a = function
       [] -> [a]
     | t::q -> if t = a then t::q else t::(ajouter a q);;
   let empty  = [];;
   ```

   On obtient donc :

```
$ ocamlc ensemble1.mli ensemble1.ml
$ ocaml
       Objective Caml version 3.08.0

# #load "ensemble1.cmo";;
# Ensemble1.ajouter 1 Ensemble1.empty;;
- : int Ensemble1.t = <abstr>
#
```

2. Une deuxième possibilité est de ne pas utiliser le type `'a Ensemble.t` en tant que type abstrait, par exemple en introduisant la définition de son implémentation dans l'interface `ensemble2.mli` :

ensemble2.mli

```
type 'a t = 'a list;;
val ajouter : 'a -> 'a t -> 'a t ;;
```

ensemble2.ml

```
type 'a t = 'a list
let rec ajouter a = function
    [] -> [a]
  | t::q -> if t = a then t::q else t::(ajouter a q);;
```

On obtient donc :

```
$ ocamlc ensemble2.mli ensemble2.ml
$ ocaml
       Objective Caml version 3.08.0

# #load "ensemble2.cmo";;
# Ensemble2.ajouter 1 [];;
- : int Ensemble2.t = [1]
#
```

□

Exercice 2. Voici un petit programme Caml pour traiter les séquences, c.-à-d. les listes possiblement infinies :

seq.ml

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;
let hd = function
    Cons(t,q) -> t
  | Nil -> failwith "hd";;
let tl = function
    Cons(t,q) -> q ()
  | Nil -> failwith "tl";;
let cons t q = Cons(t, fun () -> q);;
let from_list l = List.fold_right cons l Nil ;;
let rec from k = Cons(k, fun () -> from (k + 1));;
let rec take s n =
  if n = 0 then [] else
    match s with
        Nil -> failwith "take"
      | Cons(t,q) -> t::(take (q ()) (n-1));;
```

1. (1.5 points) Calculer le type de chaque fonction définie dans le fichier `seq.ml`.

Solution.

```
val hd : 'a seq -> 'a
val tl : 'a seq -> 'a seq
val cons : 'a -> 'a seq -> 'a seq
val from_list : 'a list -> 'a seq
val from : int -> int seq
val take : 'a seq -> int -> 'a list = <fun>
```

□

2. (1.5 points) Dire ce qui est produit par l'évaluation de `take (from 30) 2`.

Solution.

```
# take (from 30) 2;;
- : int list = [30; 31]
```

□

3. (2 points) Considérons la définition du type `'a seq` :

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;
```

Expliquer la raison pour laquelle on y trouve le type `unit` dans cette définition.
Par exemple, on pourra :
a) rappeler les conditions sous lesquelles une expression de la forme `Cons(e1,e2)` est un valeur,
b) rappeler les conditions sous lesquelles une expression de type `unit -> 'a` est un valeur,
c) définir un type `'a seqp` à la façon des listes :

```
type 'a seqp = Nilp | Consp of 'a * 'a seqp;;
```

d) écrire une fonction `fromp` qui est une modification de la fonction `from` au type `seqp`.
e) comparer et justifier l'évaluation de `from 1` et `fromp 1`.

Solution. On rappelle d'abord que `Cons(e1,e2)` est un valeur si et seulement si `e1` et `e2` sont des valeurs. Aussi, un expression du type `unit -> 'a` est un valeur ssi est de la forme `fun () -> e` (avec `e : 'a`).
Voici la définition du type `'a seqp` et de la fonction `fromp` :

seqp.ml

```
type 'a seqp = Nilp | Consp of 'a * 'a seqp;;
let rec fromp k = Consp(k, fromp (k + 1));;
```

L'évaluation de `fromp 1` et `from 1` produit respectivement :

```
# from 1;;
- : int seq = Cons (1, <fun>)
# fromp 1;;
Stack overflow during evaluation (looping recursion?).
```

On peut donc observer que pendant l'évaluation du deuxième argument de `Consp` on n'obtient jamais un valeur ce qui produit une suite infinie d'étapes d'évaluation.
Par contre, pendant du deuxième argument de `Cons` est de la forme `fun () -> e`, et donc il est un valeur. L'évaluation s'arrete à ce moment.
A l'aide du type `unit` on peut donc simuler une évaluation paresseuse ou par nom. □

4. (2 points) Donner les étapes de l'évaluation (par valeur) de l'expression `take (from 30) 2`.

Solution.

```
take (from 30) 2
-->
take (Cons (30, fun () -> from (30 + 1) ) 2
-->
if ( 2 = 0 ) then [] else match ...
-->
match  (Cons (30, fun () -> from (30 + 1) ) with
    Nil -> failwith "take"
  | Cons(t,q) -> t::(take (q ()) (n-1))
-->
30::(take (fun () -> from (30 + 1) ()) (2-1))
-->
30::(take (fun () -> from (30 + 1) ()) 1)
-->
30::(take (from (30 + 1)  1)
```

```
-->
30::(take (from 31)  1)
-->
30::(take (Cons (31, fun () -> from (31 + 1) )) 1 )
-->
30::(if ( 1 = 0 ) then [] else match ... )
-->
30::(match  (Cons (31, fun () -> from (31 + 1) )) with
    Nil -> failwith "take"
  | Cons(t,q) -> t::(take (q ()) (n-1)) )
-->
30::31::(take (fun () -> from (31 + 1) ()) (1-1))
-->
30::31::(take (fun () -> from (31 + 1) ()) 0)
-->
30::31::(take (from (31 + 1)) 0)
-->
30::31::(take (from 32) 0)
-->
30::31::(if ( 0 = 0 ) then [] else match ... )
-->
30::31::[] = [30;31]
```

□

Unification et résolution (9 points)

Exercice 3. Existe t-il un unificateur principal des couples de termes suivants ?

1. (1 points) $f(g(x), y, k(x))$, $f(y, h(z), k(w))$,

   Solution.

```
        Objective Caml version 3.08.0

# #load "resolution.cma";;
# #install_printer Terme.print_subst;;
# #install_printer Terme.print;;
# let t1 = Terme.of_string "f(g(x),y,k(x))";;

f(g(x),y,k(x))
val t1 : (string, string) Terme.t =
# let t2 = Terme.of_string "f(y,h(z),k(w))";;

f(y,h(z),k(w))
val t2 : (string, string) Terme.t =
# Terme.unify [(t1,t2)];;
Exception: Failure "unify".
```

   En effet, il existe un unificateur ssi il existe un unificateur du problème $(g(x), y), (y, h(z)), (k(x), k(w)$, et cela ssi il existe un unificateur de $(g(x), h(z)), (k(x), k(w))$ – on a appliqué la substitution $y \to g(x)$ à la queue du problème $(g(x), y), (y, h(z)), (k(x), k(w))$. Il est évident maintenant que ce dernier problème n'a pas de solution, car le couple $(g(x), h(z))$ n'est pas unifiable. □

2. (1 points) $f(x, h(x))$, $f(g(y), z)$,

   Solution.

```
# let t1 = Terme.of_string "f(x,h(x))";;
```

```
f(x,h(x))
val t1 : (string, string) Terme.t =
# let t2 = Terme.of_string "f(g(y),z)";;

f(g(y),z)
val t2 : (string, string) Terme.t =
# Terme.unify [(t1,t2)];;
{
 x -> g(y)
 z -> h(g(y))
}
- : (string, string) Terme.subst =
```

□

3. (1 points) $g(h(x, y), z)$, $g(z, h(f(u), w))$.

   Solution.

```
# let t1 = Terme.of_string "g(h(x,y),z)";;

g(h(x,y),z)
val t1 : (string, string) Terme.t =
# let t2 = Terme.of_string "g(z,h(f(u),w))";;

g(z,h(f(u),w))
val t2 : (string, string) Terme.t =
# Terme.unify [(t1,t2)];;
{
 z -> h(f(u),w)
 x -> f(u)
 y -> w
}
- : (string, string) Terme.subst =
```

□

Si oui, écrire cette substitution, si non justifier votre réponse.

Exercice 4. (2 points)

1. Rappeler la forme générale de la règle de factorisation (à droite).

   Solution.

$$\frac{\Gamma \Rightarrow \Delta, A, B}{\sigma\Gamma \Rightarrow \sigma\Delta, \sigma A}$$

   où σ est un unificateur principal des formules atomiques A et B. □

2. Appliquer la règle de factorisation à droite à la clause

$$Q(g(y), x) \Rightarrow P(f(x), y), P(y, f(x))$$

   Solution.

$$\frac{Q(g(y), x) \Rightarrow P(f(x), y), P(y, f(x))}{Q(g(f(x)), x) \Rightarrow P(f(x), f(x))}$$

□

Exercice 5. (2 points)

1. Rappeler la forme générale de la règle de résolution.

   Solution.
   $$\frac{\Gamma_1 \Rightarrow \Delta_1, A \qquad B, \Gamma_2 \Rightarrow \Delta_2}{\sigma\Gamma_1, \sigma\Gamma_2 \Rightarrow \sigma\Delta_1, \sigma\Delta_2}$$
   où σ est un unificateur principal des formules atomiques A et B. □

2. Appliquer la règle de résolution au couple de clauses suivantes :
   $$P(x) \Rightarrow P(f(x)), R(y) \quad P(y) \Rightarrow Q(y, g(y))$$

   Solution.
   $$\frac{P(x) \Rightarrow P(f(x)), R(y) \quad P(y) \Rightarrow Q(y, g(y))}{P(x) \Rightarrow R(f(x)), Q(f(x), g(f(x)))}$$
   □

Exercice 6. On a discuté, dans le cours, la structure d'un démonstrateur automatique. À un tel démonstrateur on passe en argument une théorie logique $\mathbb{T}$ sous la forme d'un ensemble de clauses. Le démonstrateur essaie de dériver la clause vide en saturant la théorie : il applique toutes les règles du calcul de la résolution à toutes clauses déjà produites pour en produire des nouvelles.
Pour chacun de trois cas suivants, dire si la théorie $\mathbb{T}$ est consistante ou non.

1. (0.5 points) Le démonstrateur s'arrête par ce qu'il a dérivé la clause vide.

   Solution. La théorie est inconsistante : en effet la clause vide est une contradiction. □

2. (0.5 points) Le démonstrateur s'arrête par ce qu'il ne peut plus appliquer des règles du calcul.

   Solution. La théorie est consistante. □

3. (1 points) Le démonstrateur ne s'arrête jamais : il produit un nombre infini de clauses.

   Solution. La théorie est consistante : en effet on démontre que le calcul de la résolution est complet à ce sens : si la théorie est inconsistante (c.-à-d., si une contradiction est dérivable à l'aide des règles usuelles de la logique du premier ordre) alors la clause vide sera tôt ou tard dérivée par l'application des règles du calcul de la résolution ; et dans ce cas, tôt ou tard, le démonstrateur s'arrêtera. □

Sémantique (7 points)

Exercice 7.

1. (0.5 points) Définir en Caml (Camllight ou Ocaml) un type `loc` pour y coder les locations du langage IML.
2. (1.5 points) Définir en Caml un type `etat` pour y coder les états du langage IML.
3. (1.5 points) Définir en Caml un type récursif `aexpr` pour y coder les expression arithmétiques du langage IML.
4. (1.5 points) Écrire en Caml une fonction `eval`, dont le type est `eval : aexpr -> etat -> int`, telle que, pour toute expression `a : aexpr`, pour tout expression `e : etat` et pour tout `n : int`, on a

```
# eval a e;;
- : int = n
```
ssi $(a, e) \to n$ ssi $(e, n) \in \mathcal{A}\|a\|$.

Solution.

iml.ml

```
type loc = string;;
type etat = loc*int list;;

type aexpr = Int of int | Loc of loc
      | Somme of aexpr*aexpr
      | Difference of aexpr*aexpr
      | Produit of aexpr*aexpr
;;

let rec eval a e = match a with
    Int(n) -> n
  | Loc(l) -> List.assoc l e
  | Somme(a1,a2) -> (eval a1 e) + (eval a2 e)
  | Difference(a1,a2) -> (eval a1 e) - (eval a2 e)
  | Produit(a1,a2) -> (eval a1 e) * (eval a2 e)
;;
```

□

Exercice 8. (2 points) Soit L un ensemble partiellement ordonné, soit $f : L \to L$ une fonction monotone, et soit μ tel que

$$\begin{gathered} f(\mu) \leq \mu \\ f(x) \leq x \quad \Rightarrow \quad \mu \leq x\,, \qquad \text{pour tout } x \in L\,. \end{gathered}$$

Démontrer que μ est un point fixe de f, c.-à-d. démontrer que $f(\mu) = \mu$.

Solution. Par hypothèse $f(\mu) \leq \mu$, et car f est monotone on obtient $f(f(\mu)) \leq f(\mu)$. Posons $x = f(\mu)$: on a donc $f(x) \leq x$ et donc $\mu \leq x = f(\mu)$. Par conséquent, $f(\mu) \leq \mu$ et $\mu \leq f(\mu)$ impliquent $f(\mu) = \mu$. □