Master d'informatique | **module TEP** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2h* | **Année 2013-2014**

Examen du 21 novembre 2013

Exercice 1 - λ-calcul sur les entiers

Un entier de Church est représenté comme un terme dont le corps comprend n (si le nombre en question est n) applications successives d'une fonction à un argument. Voici quelques exemples d'entiers de Church :
- $\overline{0} = \lambda fx.x$
- $\overline{1} = \lambda fx.fx$
- $\overline{n} = \lambda fx.f^n x$ ce qui équivaut à $\lambda fx.(f(f \ldots (f\ x) \ldots))$ où f est appliquée n fois.

On étudie le terme de Maurey définit ainsi : $M = \lambda nm.(nF(\lambda x.a)(mF(\lambda x.b)))$
où $F = \lambda fg.(gf)$ et a et b sont deux variables libres.

1. Donner la forme normale des λ-termes suivants, et si oui indiquer une réduction permettant de l'obtenir.
   - $\overline{0}\ F$, $\overline{1}\ F\ (\lambda x.a)$, $\overline{2}\ F\ (\lambda x.b)$
   - Appliquer M aux différents couples construits à partir des entiers $\overline{0}, \overline{1}, \overline{2}$ (ex : $M\ \overline{1}\ \overline{2}$).
2. A quoi correspond le terme M pour ces premiers entiers ? Expliquer le comportement de ce terme pour tout couple d'entiers de Church.
3. Ecrire un λ-terme clos correspondant à la fonction *min* pour deux entiers de Church.

Exercice 2 : typage, partage d'environnement et fonctionnelles

1. Donner le type OCaml, s'il existe, des déclarations OCaml suivantes :

```
let make_gensym s =
    let c = ref 0 in
    let g facc  = facc s c  in g ;;

let r = make_gensym "TEP";;

let f_reset s c = c:=0; s;;

let f_new s c = incr c; s^(string_of_int !c);;
```

2. Calculer les valeurs des expressions OCaml suivantes en justifiant les résultats obtenus :

```
r f_new;;
r f_new;;
r f_new;;
r f_reset;;
r f_new;;
r f_new;;
```

3. Indiquer en le justifiant le type des déclarations et expressions suivantes, en précisant la valeur retournée par l'appel de la fonction `r` :

```
let f_mystere s c = c:= !c + 2;  c;;
r f_mystere;;
```

Exercice 3 : Résolution de la surcharge

Dans cet exercice on cherche à comparer différentes résolution de surcharge en Java en faisant varier la relation d'ordre sur les signatures de méthodes.

Soient les classes Java suivantes :

```
class A {
  void m(B x) {        /* A1 */ }
  void m(C x, B y) {   /* A2 */ }
}
class B extends A {
  void m (A x) {        /* B1 */}
  void m (B x , A y) { /* B2 */ }
  void m (C x , B y) { /* B3 */ }
}
class C extends B {
  void m (C x) {        /* C1 */ }
  void m (C x, C y) {   /* C2 */ }
  void m (C x, B y) {   /* C3 */ }
}
```

```
class D {
  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    A a1 = b;
    C c = new C();
    B b1 = c;
    A a2 = b1;
    a.m(b);     a.m(a1);     b.m(a1);  //  QUESTION 1
    b.m(b);     c.m(c);      c.m(b2);
    a.m(b,b);   a.m(c,c);    a.m(c,b); //  QUESTION 2
    b.m(b,b);   b.m(c,c);    b.m(c,b1);
    c.m(b,b);   c.m(c,c);    c.m(c,b1);
  }
}
```

Pour chacune des questions suivantes vous rappellerez ou préciserez la relation d'ordre employée pour la comparaison des signatures des méthodes.

1. Dans un tableau, indiquer quand elle existe la signature sélectionnée de la méthode `m` en Java 1.7 lors des appels de `m` avec un paramètre, puis préciser quelle méthode sera exécutée au lancement du programme.
2. même question pour les appels à 2 paramètres
3. En utilisant une autre relation d'ordre, qui compare ici le produit cartésien de la classe de construction et des types des paramètres d'une méthode, pour la résolution de la surcharge reconstruire les tableaux des deux premières questions en expliquant les différences qui y apparaissent.
4. Même question en supprimant la résolution de la surcharge et la relation de subsomption, c'est-à-dire que l'on recherche la signature d'une méthode ayant exactement les types des paramètres d'appel. Indiquer quand cela est possible quel *upcast* explicite permet de faire passer certains appels.

Exercice 4 : Files d'attente génériques

Soit la classe OCaml `['a] queue` suivante :

```
1  exception Empty
2
3  class ['a] oqueue3 () =
4    object(self)
5      val mutable q = ([] : 'a list)
6      method enq x =  q <-  q @ [x]
7      method deq () = match q with
8          [] -> raise Empty
9      |  h::r -> q <- r ; h
10     method reset () = q <- []
11     method fold : 'b.  ('b -> 'a -> 'b) -> 'b -> 'b =  fun f  accu ->
12        List.fold_left f accu q
13   end;;
```

où `fold_left` est un itérateur sur les listes : `fold_left f a [x;y;z]` donne `(f (f (f a x) y) z)`.
La méthode `fold` est une méthode polymorphe où le type du résultat `'b` ne dépend que des types de ces paramètres `f` de type `'b -> 'a -> 'b` et `accu` de type `'b`, d'où la déclaration indiquant que la variable de type `'b` est locale à déclaration de `fold`.

1. Donner le type OCaml des méthodes `enq` et `deq`.
2. Ecrire une classe paramétrée Java (1.7) équivalente, nommée Queue<T>. Dans une première version on n'écrira pas la méthode `fold`.
3. Sans l'implanter (voir question 8), indiquez comment feriez-vous pour simuler le comportement de la méthode `fold` en Java. Précisez cette simultation pour au moins deux fonctions vérifiant dans une Queue<String> d'une part l'appartenance du mot `"FIN"` et d'autre part qui calcule la somme des longueurs des mots de la Queue<String>. Vous pouvez définir une méthode générique `fold_left` Java avec une autre signature.
4. Construire deux instances `qa` et `qb` de cette classe ; `qa` est une queue contenant des noms (`Queue<String>`) et `qb` des numéros (`Queue<Integer>`).
5. Indiquer ce qui se passe à la compilation et à l'exécution de ce fragment de programme par rapport à la classe que vous avez écrite. Si une ligne ne compile pas, traitez néanmoins la suite du programme.

```
1  Queue qq = qb;
2  qq.enq("Essai");
3  Object o = qq.deq();
4  String s = (String)o;
5
6  qq.enq(o);
7  qq.enq(new Integer(33));
8  System.out.println(qq.deq());
9  System.out.println(qq.deq());
10
11 qq.enq(o);
12 qq.enq(new Integer(12));
13 int r = qb.deq() + qb.deq();
14 System.out.println(r);
```

6. On définit la méthode de copie suivante associée à une classe quelconque `C` :

```
    public static <T> void copy(
            Queue<? super T> destination,
            Queue<? extends T> source);
```

   Indiquez le sens des deux *wildcards* (`?`) en terme de co-variance et de contra-variance sur les paramètres de type des types des paramètres.
7. Expliquez pour chaque ligne du programme suivant utilisant la fonction `copy` celles qui sont correctes du point de vue des types et celles qui déclencheront une erreur à la compilation. Justifiez vos réponses. On rappelle que les classes `Integer` et `Double` héritent indépendamment de `Number`, et que toutes les classes Java sont sous-classes d'`Object`.

```
Queue<Object> qo;
Queue<Double> qd;
Queue<Integer> qi;
Queue<Number> qn;

C.copy(qn,qn);
C.copy(qi,qn);
C.copy(qn,qi);
C.copy(qi,qd);
C.copy(qo,qd);
C.copy(qd,qo);
```

8. Implanter la méthode `fold` en Java comme vous l'avez expliqué à la question 3.