Master d'informatique | **module TEP** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2008-2009**

Examen du 17 novembre 2008

1 - λ-calcul : stratégie d'évaluation et typage

Soient les λ-termes suivants :
$T = \lambda xy.x$
$F = \lambda xy.y$
$COND = \lambda bxy.bxy$
$NON = ...$
$OU = ...$
$ET = ...$
$\Delta = \lambda x.xx$
$I = \lambda x.x$
$Q_1 = OU\ T\ (ET\ T\ F)$
$Q_2 = OU\ T\ (\Delta\ \Delta)$

1. En prenant comme représentation des booléens les termes T pour vrai et F pour faux, et $COND$ pour la conditionnelle, écrivez les opérateurs booléens NON, ET et OU sous forme d'application de la conditionnelle $COND$ que vous réduirez et en indiquant leur type dans le *lambda*-calcul simplement typé.
2. Réduisez quand cela est possible les λ-termes Q_1 et Q_2 suivant la stratégie d'évaluation retardée (l'argument est passé tel quel à la fonction) et la stratégie d'évaluation immédiate (l'argument est évalué avant d'être passé à la fonction).

2 - Typage : fonctionnelles et références en O'Caml

Donnez le type O'Caml, s'il existe, des déclarations O'Caml suivantes et indiquez les erreurs de typage, quand elles ont lieu, en précisant la raison :

1. ```
   let succ x  x + 1;;
   let rs = ref succ;;
   let id x = x ;;
   let ri = ref id;;
   ```
2. ```
   let app f x =  f x;;
   let rapp = ref app;;
   let rec f x = f x;;
   let rf = ref f;;
   ```
3. ```
   let set a b = a := b;;
   let ex1 = set ri succ ; ri;;
   let ex2 = set rs id; rs;;
   let ex3 = set rapp succ; rapp;;
   ```
4. ```
   let ex4 = set rf app; rf;;
   let ex5 = set rapp rf; rapp;;
   let ex6 = set rf succ; rf;;
   let ex7 = (!rapp) succ ; rapp;;
   ```

3 - Surcharge et liaison tardive en Java

On cherche à écrire son propre résolveur de surcharge qui transforme un programme Java avec surcharge en un programme Java sans surcharge équivalent selon un algorithme de surcharge donné. Pour cela on distinguera les méthodes surchargées en leur associant un nom particulier en fonction de leur arité et du types des paramètres formels de la manière suivante :
soit la méthode m de type de retour t_r, qui a n paramètres p_1, ... p_n de type respectif t_1, ... t_n :

```
tr m (t1 p1, t2 p2, ... , tn pn))
```

le nom de la méthode est alors transformée en `m_n_p1_p2_..._pn`.

Voici un exemple de transformation de la méthode m :

programme d'origine	programme transformé

`class C {`
`  A m (A x, B y){ ... }`
`}`
 	
`...`
`  A a = new A();`
`  B b = new B();`
`  a.m(a,b);`
`...`
	`class C {`
	`  A m_2_A_B (A x, B y) { ... }`
	`}`
	`...`
	`  A a = new A();`
	`  B b = new B();`
	`  a.m_2_A_B(a,b);`
	`...`

1. On utilisera dans un premier temps une résolution de surcharge qui sélectionne une méthode de la bonne arité dont la signature (produit cartésien des types de paramètres formels) est égale aux types des paramètres d'appel sans effectuer de conversions de types implicites en cas de sous-typage. Effectuez à la main la transformation décrite précédemment sur le code Java suivant :

```
class A {
    void m(A x, Ay) {System.out.println("cas 1");}
}
class B extends A {
    void m(A x, B y) {System.out.println("cas 2");}
    void m(A x, A y) {System.out.println("cas 3");}
}
class C extends B {
    void m(B x, C y) {System.out.println("cas 3");}
    void m(A x, C y) {System.out.println("cas 3");}
    void m(A x, A y) {System.out.println("cas 3");}
}

class p3 {
  public static void main(String[] a) {
      A a1 = new A();
      A a2 = new A();
      B b1 = new B();
      B b2 = new B();
      C c1 = new C();
      C c2 = new C();
      A a3 = b1;
      A a4 = c1;
      B b3 = c1;
```

```
  }
}

// ...
a1.m(a1,a2);
a1.m(b1,a2);
b1.m(a1,b1);
b1.m(a1,a3);
b1.m(a1,(A)b1);
c1.m(a1,a2);
c1.m(a1,b1);
c1.m(b1,a1);
c1.m(b1,b1);
c1.m(b1,c1);
```

puis indiquez ce qu'affichera le programme on enlevant les lignes, s'il en existe, provoquant une erreur à la compilation en expliquant la nature de l'erreur.

2. On cherche maintenant à utiliser l'algorithme de résolution de la surcharge sélectionnant la méthode de bonne arité dont la signature (produit cartésien des types des paramètres formels) est la plus petite par rapport à la relation de sous-typage Java. Indiquez s'il y a des différences à la compilation et à l'exécution sur les deux programmes précédents. Si oui expliquez-les.

4 - Classes paramétrées en Java

On cherche à écrire une classe paramétrée (générique) pour coder des piles homogènes, c'est-à-dire contenant des éléments du même type, avec les comportements classiques : `push` empile un élément sur la pile, `pop` dépile le sommet de pile et `top` retourne le sommet de pile sans le dépiler.
Soit l'interface suivante :

```
interface IPile<ALPHA> {
    ALPHA  pop();
    void push (ALPHA x);
}
```

1. Ecrivez une classe `Pile <ALPHA>` qui implante l'interface précédente. On pourra utiliser des `RuntimeException` pour simplifier les échappements si besoin est.
2. En fonction de votre implantation de cette classe, indiquez les lignes qui provoquent une erreur ou un warning de typage à la compilation et indiquez ce qu'exécute le programme suivant en ne tenant pas compte des lignes d'erreur. On rappelle que `Double` est sous-classe de la classe abstraite `Number`.

```
class p4 {
    static void main (String[] a) {
//      partie 1
        Double d1 = 3.14;
        Double d2 = (Double)2.12;
        Pile<Double> pd1 = new Pile<Double>();
        Pile<Number> pn1 = new Pile<Number>();
        Pile<? super Number> psupn1 = pn1;
        Pile<? extends Number> pextn1 =  pn1;
        Pile<Pile<Double>> ppd1 = new Pile<Pile<Double>>();
//  partie 2
        pd1.push(d1);
```

```
        pd1.push(d2);
        pn1.push(d1);
        pn1.push(d2);
//       partie 3
        Pile pp = pd1;
        pp.push(d1);
        System.out.println(pd1.pop() + pd1.pop());
//       partie 4
        psupn1.push(d1);
        pextn1.push(d2);
        System.out.println(psupn1.pop() + pextn1.pop());
    }
}
```

3. Soit l'interface `IEPile<ALPHA>` suivante :

```
interface IEPile<ALPHA> extends IPile<ALPHA> {
    ALPHA top();
}
```

écrivez une sous-classe de `EPile <ALPHA>`, sous classe de `Pile` qui implante cette interface.

4. Toujours en fonction de votre implantation, indiquez d'une part les lignes qui provoquent une erreur ou un warning de typage à la compilation en expliquant pourquoi, et d'autre part faite une trace de l'exécution du programme sans les lignes non compilables.

```
class p5 {
    static void main (String[] a) {
//           mêmes déclarations que la partie 1 de la question précédente
//       partie 1
        EPile<Double> epd1 = new EPile<Double>();
        EPile<Number> epn1 = new EPile<Number>();
        EPile<? super Number> epsupn1 = epn1;
        EPile<? extends Number> epextn1 = epn1;
        EPile<EPile<Double>> epepd1 = new EPile<EPile<Double>>();
//       partie 2
        epd1.push(d1);
        epd1.push(d2);
        Pile epp = epd1;
        epp.push(d1);
        System.out.println(pd1.pop() + pd1.pop());
//       partie 3
        epsupn1.push(3);
        epextn1.push(4);
        System.out.println(epsupn1.top());
        System.out.println(epextn1.top());
//       partie 4
        EPile<? super Number> pn3 = epsupn1;
        EPile<? extends Number> pn4 = epextn1;
        epsupn1.push(3.22);
        epextn1.push(4.33);
        System.out.println(pn3.pop());
        System.out.println(pn4.pop());    }}
```