Master d'informatique | **module TAS (5I555)** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2h* | **Année 2018-2019**

Examen du 22 novembre 2018

Exercice 1 - λ-calcul typé

Dans le cadre du λ-calcul simplement typé, on cherche à vérifier que le type associé à un λ-terme est correct, c'est-à-dire que l'on arrive à construire l'arbre de vérification du type en fonction des règles de typage suivantes. Pour cela on rappelle la définition des types simples et des trois règles de ce système de types.

Soit $\mathcal{P}$ un ensemble de variables de types. On définit l'ensemble des types simples bien formés $\mathcal{T}$ (construit à partir de $\mathcal{T}$ et du constructeur $\rightarrow$) par :
— si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$
— si $\sigma, \tau \in \mathcal{T}$ alors $\sigma \rightarrow \tau \in \mathcal{T}$.

Les contextes sont des listes de couples $C = (x_1 : \sigma_1), \ldots, (x_n : \sigma_n)$.

Les règles de typage sont les suivantes :

(Var)
$$(x_1 : \sigma_1), \ldots, (x_n : \sigma_n) \vdash x_i : \sigma_i$$

(App)
$$\frac{C \vdash M : \sigma \rightarrow \tau \quad C \vdash N : \sigma}{C \vdash MN : \tau}$$

(Abs)
$$\frac{(x : \sigma), C \vdash M : \tau}{C \vdash \lambda x.M : \sigma \rightarrow \tau}$$

On s'intéresse aux trois λ-terme suivants :

1. $(\lambda x.x)(\lambda y.y)$
2. SKK avec $K = \lambda x.\lambda y.x$ et $S = \lambda x.\lambda y.\lambda z.xz(yz)$
3. $\Omega = \Delta\Delta$ avec $\Delta = \lambda x.xx$

Donner pour chaqu'un d'eux le code équivalent en OCaml ou Java, et indiquer le type attendu sans se soucier de leur correction, puis constuire l'arbre de vérification de leur type quand cela est possible, et sinon préciser quelle règle de typage ne peut pas être respecté pour le type proposé.

Exercice 2 - Files d'attente en fonctionnel (OCaml ou Java)

On cherche à représenter une file d'attente dans un style fonctionnel relativement efficace. Pour cela on représente une telle file par un couple (ou un enregistrement à deux champs) de listes : le premier champ contient la liste des éléments entrants, le second la liste des élément sortants. Ces files d'attente sont homogènes. L'idée est d'optimiser les opérations. Pour cela la liste des éléments rentrants a le dernier élément arrivé en tête de liste, et la liste des éléments sortants a le premier élément arrivé en tête de liste.

```
entrants    sortants

[R E Z A]   []          // enqueue  A  puis enqueue Z puis E puis R
[]          [A Z E R]   // copyover
[]          [Z E R]     // dequeue retourne A
[I U Y T]   [Z E R]     // enqueue T, puis enqueue Y puis U puis I
[I U Y T]   []          // 3 appels à dequeue retournent Z, puis E puis R
[]          [Y U I]     // dequeue provoque copyover et retourne T
[]          []          // 3 appels à dequeue retourne Y puis U puis I
[]          []          // dequeue déclenche une exception
```

1. Donner le type paramétré pour de telles files d'attente fonctionnelles génériques. On s'interdira toute modification physique sur de telles structures.
2. Ecrire une fonction `enqueue` qui prend un élément et une file et retourne une nouvelle file où la branche entrante est enrichie par l'élément passé.
3. Ecrire une fonction `copyover` qui prend une file d'attente en entrée et retourne une nouvelle file où tous les éléments entrants se retrouvent maintenant dans la branche de sortie.
4. Ecrire une fonction `dequeue` qui prend une file, retourne le premier élément de la branche de sortie si elle n'est pas vide. Si la branche de sortie est vide, on copie (avec la fonction précédente) la branche d'entrée dans la branche de sortie. Si les deux branches étaient vides, une exception est alors déclenchée.
5. Sur l'exemple donné, indiquer le nombre d'opérations élémentaires de création de liste effectuées ainsi que le nombre d'éléments de liste alloués.

Exercice 3 - Surcharge et liaison tardive (Java)

On utilise dans la suite de la question l'interface `Comparable<T>`. et on définit les classes suivantes :

```java
class Point
implements Comparable<Point>{
  private int x;
  private int y;
  Point(){x=0;y=0;}
  Point(int x, int y){
     this.x=x;
     this.y= y;
  }
  int getX(){return x;}
  int getY(){return y;}
  public int compareTo(Point c) {
    int cx=c.getX();
    int cy=getY();
    if (x < cx)  return -1;
    else if (x > cx)  return 1;
    else if (y < cy) return -1 ;
    else if (y > cy) return 1 ;
    else return 0 ;
  }
  int myeq(Point c) {
    System.out.println("P MYEQ(P)");
    return compareTo(c);
  }
}

class PointColore extends Point {
 String c;
 PointColore(){c="VERT";}
 PointColore(int x, int y, String c) {
  super(x,y); this.c=c;
 }
```

```java
  // suite de PointColore
 String getC(){return c;}

 int myeq(PointColore pc) {
  System.out.println("PC MYEQ(PC)");
  int  r1 = super.myeq(pc);
  if (r1 != 0) return r1;
  else return c.compareTo(pc.getC());
 }
}

class PointSousColore extends PointColore {
 String c2;
 PointSousColore(int x,int y,String c,String c2) {
  super(x,y,c); this.c2=c2;
 }
 String getC2(){return c2;}
 int myeq(PointSousColore psc) {
  System.out.println("PSC MYEQ(PSC)");
  int  r1 = super.myeq(psc);
  if (r1 != 0) return r1;
  else return c2.compareTo(psc.getC2());
 }
 int myeq(PointColore pc) {
  System.out.print("PSC MYEQ(PC)-->");
  return super.myeq(pc);
 }
 int myeq(Point p) {
  System.out.print("PSC MYEQ(P)-->");
  return super.myeq(p);
 }
}
```

1. Indiquer ce qu'affiche le programme suivant en justifiant vos réponses pour chaque affichage.

```
Point p = new Point();
PointColore pc = new PointColore(2,3,"R");
Point np = (Point) pc;
PointSousColore psc = new PointSousColore(1,2,"R","B");
PointColore npc = (PointColore) psc;
int a = p.myeq(p);
int b = p.myeq(pc);
int c = p.myeq(np);
int d = pc.myeq(p);
int e = pc.myeq(pc);
int f = pc.myeq(np);
```

2. Même question pour le programme suivant :

```
 int g = psc.myeq(p);
 int h = psc.myeq(pc);
 int i = psc.myeq(np);
 int j = psc.myeq(npc);
 int k = psc.myeq(psc);
```

3. On remplace les différents appels à `super.myeq(x)` par un `super.myeq((Point)x)`. Indiquer les lignes et méthodes touchées puis préciser les différences d'affichage en expliquant pourquoi.
4. Expliquer l'erreur lors de la compilation de la classe `PC` suivante et indiquer un moyen s'il y en a un de la contourner :

```
$  cat PC.java
class PC extends Point implements Comparable<PC> {
}
$  javac PC.java
PC.java:1: error: Comparable cannot be inherited with different arguments: <PC> and <Point>
class PC extends Point implements Comparable<PC> {
^ 1 error
```

Exercice 4 : Sous-typage (OCaml et Java)

On définit le code objet OCaml suivant :

```
class p nom =

  object(self:'a)

    val nom = nom

    method get_nom = nom

    method to_string () = nom^""

    method eq(p2:'a) =
      self#get_nom = p2#get_nom
  end

  let p1 = new p "A"
```

```
class pa nom age =

  object(self:'b)
    inherit p nom as super

    val age = age

    method get_age = age

    method to_string () =
      super#to_string() ^
      (string_of_int age)

    method eq(p2:'b) =
      (self#get_nom = p2#get_nom) &&
      (self#get_age = p2#get_age)
    end

    let pa1 = new pa "B" 20
```

1. Donner le type des méthodes de chaque classe.

2. Pourquoi la déclaration suivante :

```
1    let p2 = (pa :> p)
```

déclenche-t-elle une erreur de typage en indiquant bien les raisons.

3. Traduire les deux classes OCaml en Java en indiquant explicitement le type de la classe de définition comme type du paramètre quand un paramètre a le type de l'instance (ici `self`).
4. Traduire ensuite en Java les trois déclarations : p1, pa1 et p2. Y a-t-il la même erreur dans la déclaration de p2 ? Expliquer votre réponse en précisant bien les mécanismes utilisés.

Exercice 5 : Sous-typage borné (Java)

Dans le cadre du polymorphisme borné à la java, la fonction `copy` qui copie les éléments de `src` vers `dst` possède la signature suivante :

```
1  public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

1. Expliquer cette signature en précisant les relations de sous-typage des types génériques des paramètres. Ecrire la fonction `copy` et donner un exemple d'appel de cette fonction respectant bien la signature.
2. On suppose une nouvelle version de la fonction de copie, ici appelée `badcopy`, avec le code précédent mais dont la signature est incorrecte :

```
1  public static <T> void badcopy(List<? extends T> dest, List<? extends T> src)
```

En supposant que cette fonction compile, construire un exemple utilisant cette fonction `badcopy` et dont l'utilisation du résultat de la copie est incorrecte vis-à-vis du typage Java. Indiquer alors la ligne du code de la fonction qui devrait entraîner une erreur de typage lors de la compilation en en précisant les raisons.