Master d'informatique | **module TAS (MU5IN555)** | **Sorbonne Université**
Documents autorisés | *Durée 2h* | **Année 2019-2020**

Examen du 21 novembre 2019

Exercice 1 - λ-calcul : triplets et conditionnelle

On cherche d'une part à définir les triplets et d'autre à s'en servir pour coder une structure de contrôle conditionnelle.

1. Pour cela, définir les quatre λ-termes correspondant respectivement aux constructeur ($makeTriplet$) et accesseurs de triplets ($proj_i$ pour la 1ère, 2ème et 3ème projection) qui vérifient les propriétés suivantes :

$$proj1(makeTriplet\ A\ B\ C) \rightarrow_* A$$
$$proj2(makeTriplet\ A\ B\ C) \rightarrow_* B$$
$$proj3(makeTriplet\ A\ B\ C) \rightarrow_* C$$

2. On cherche maintenant à définir une structure de contrôle conditionnelle ITE (if then else) prenant un triplet contenant le terme de la condition en première position, le terme de la branche then en 2ème position et le terme de la branche else en 3ème position. On codera les booléens de manière classique :

$$T = \lambda xy.x$$
$$F = \lambda xy.y$$

   Ecrire alors un terme, nommé ITE, prenant un triplet dont le premier terme s'évalue en un booléen, selon la valeur de celui-ci il retourne le terme de la branche then ou else.

$$ITE(makeTriplet\ T\ E1\ E2) \rightarrow_* E1$$
$$ITE(makeTriplet\ F\ E1\ E2) \rightarrow_* E2$$

   Montrer que votre terme ITE vérifie bien les deux résultats précédents avec $E1$ et $E2$ deux termes quelconques.

3. On essaie maintenant de définir sur les expressions conditionnelles manipulant des booléens (dans les branches then et else) un opérateur de négation, appelé NOT, qui vérifie :

$$NOT\ T \rightarrow_* F$$
$$NOT\ F \rightarrow_* T$$

   Donner NOT et montrer en calculant les termes que le terme $NOT(ITE\ (makeTriplet\ A\ B\ C))$ est β-équivalent au terme $ITE\ (makeTriplet\ A\ (NOT\ B)\ (NOT\ C))$.

4. Définir les opérateurs AND et OR, correspondant respectivement à la conjonction et à la disjonction sur les booléens, à partir de ITE et vérifier que $NOT(AND\ A\ B)$ est équivalent à $OR\ (NOT\ A)\ (NOT\ B)$.

Exercice 2 : Polymorphisme paramétrique en fonctionnel/impératif (OCaml)

Dans cet exercice, on cherche à comprendre l'influence sur le polymorphisme paramétrique des traits impératifs dans le langage OCaml. Pour chaque question, donnez le type de la déclaration ou de l'expression si elle est typable, et dans le cas inverse expliquez pourquoi elle ne l'est pas.

On rappelle les constructions et opérateurs sur les références : `ref` est le constructeur des références prenant une valeur (`'a`) et construisant une référence sur cette valeur (`'a ref`), `!` est l'accesseur qui prend une référence et retourne la valeur référencée, de type `'a ref -> 'a`, et `:=` le modifieur qui prend une référence et une valeur en modifiant la valeur référencée par celle-ci, de type `'a ref -> 'a -> unit`.

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

```
let f x = x ;;
let g x = x + 1;;
let h x = not x;;
```

2. Donner les types, s'ils existent ou sinon indiquer pourquoi la déclaration ou l'expression n'est pas typable, de la déclaration et des expressions suivantes :

```
let rf = ref f;;
!rf 12 ;;
rf :=  g ;;
!rf 14 ;;
rf := f ;;
!rf true ;;
```

3. Donner les types, s'ils existent ou sinon indiquer pourquoi la déclaration ou l'expression n'est pas typable, de la déclaration et des expressions suivantes :

```
let nrf = ref f;;
nrf := h ;;
!nrf true ;;
nrf := g ;;
!nrf 14 ;;
```

4. Que calculerait l'ensemble du programme des 3 questions précédentes si toutes les déclarations et expressions étaient typables ? Indiquer alors le problème rencontré.

Exercice 3 : Objet et fonctionnel (Java)

On cherche à implanter en Java un mécanisme d'évaluation retardée L'idée principale est de geler le calcul d'une expression classique dans une lambda-expression, au sens de Java 1.8.

```
interface Freezable<T> {
  T eval() ;
}
class LateEval<T> implements Freezable<T> {
  private Freezable<T> closure;
  LateEval(Freezable<T> c){ closure=c; }
  public T eval(){ return (closure.eval()); }
}
```

1. On définit alors une macro au sens de CPP (le préprocesseur de C) :
`#define lazy(e)  new LateEval(() -> e)` qui remplace `lazy(e)` par `new LateEval(() -> e)`. Indiquer l'évaluation du programme suivant (méthode `main`) en précisant les affichages.

classe A	classe B

class A {
private int x;
A(int x){this.x = x;}
int getX(){return x;}
void incrX(int dx){x = x + dx;}
public String toString(){return ("[" + x +"]");}
void display(){
System.out.println(this.toString()); }
}
	class B extends A {
	private int y;
	B(int x, int y){super(x); this.y =y;}
	int getY(){return y;}
	void incrY(int dy){y = y + dy;}
	public String toString(){return (
	super.toString()+"[" + y +"]");}
	}

```
class Q3 {
 public static void main(String[] args) {
  A a = new A(3);
  B b = new B(3,9);
  A c = a;
  Freezable<String> fa = lazy(a.toString());
  Freezable<String> fb = lazy(b.toString());
  System.out.println(fa.eval());
  System.out.println(fb.eval());
```

```
10    b.incr();
11    System.out.println(fa.eval());
12    System.out.println(fb.eval());
13   }}
```

2. Ecrire une classe `Lazy` sous-classe de `LateEval` qui évalue la valeur de l'expression gelée la première fois qu'on lui demande de le faire (méthode `eval`), conserve cette valeur et la retourne. Les prochains appels à cette évaluation retournent directement le résultat conservé.

3. Indiquer le comportement du programme précédent avec la définition suivante de la macro `lazy` : `#define lazy(e)  new Lazy(() -> e)`

4. L'expression retardée peut déclencher une exception lors de son évaluation, par exemple `lazy(throw new MyE())` où `MyE` est une sous-classe d'Exception. On est dans le cas où une évaluation (même tardive) risque de déclencher une exception, mais pas forcément au moment prévu dans le programme. Quel problème cela peut-il soulever dans la correction des types Java, en particulier sur les exception tracées ? Que faut-il modifier dans les déclarations d'interface et de classes précédentes pour tenir compte de cela.

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. Effectuer à la main la transformation décrite précédemment sur le code Java suivant (classes A, B et Q4) puis indiquer ce qu'affichera le programme en enlevant les lignes, s'il en existe, provoquant une erreur à la compilation en expliquant alors 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. Indiquer s'il y a des différences à la compilation et à l'exécution sur les deux programmes précédents, et si oui les expliquer.

classes A et B	classe Q4 et méthode main

`class A {`
`  void m(A x, A y) {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 4");}`
`  void m(A x, C y) {System.out.println("cas 5");}`
`  void m(A x, A y) {System.out.println("cas 6");}`
`}`
	`class Q4 {`
	`  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);`
	`  }`
	`}`

Exercice 5 : Types et génériques (Java)

En Java 1.8 le programme suivant compile sans aucun warning :

ee

```
1  class Q5<T, U> {
2   class Contrainte<B extends U> {}
3   final Contrainte<? super T> contrainte;
4   final U u;
5
6   Q5(T t) {
7    u = coerce(t);
8    contrainte = getContrainte();
9   }
10
11  <B extends U> U upcast(Contrainte<B> contrainte, B b) { return b; }
12
13  U coerce(T t) { return upcast(contrainte, t); }
14
15  Contrainte<? super T> getContrainte() { return contrainte; }
16
17  public static void main(String[] args) {
18   Object zo = new Q5<Integer,Object>(0).u;
19   System.out.println(zo);
20   Integer zero = new Q5<Object,Integer>(0).u;
21   System.out.println(zero);
22   String zs = new Q5<Integer,String>(0).u;
23   System.out.println(zs);
24  }
25 }
```

Par contre son exécution provoque une erreur de typage :

```
$ javac Q5.java
$ java Q5
0
0
Exception in thread "main" java.lang.ClassCastException:
  java.lang.Integer cannot be cast to java.lang.String
at Q5.main(Q5.java:22)
```

1. Indiquer ce que le programme est censé faire en détaillant les méthodes utilisées.
2. Expliquer l'erreur de typage à l'exécution, et ce que cela implique dans le système de types des génériques de Java.