Master d'informatique — *Documents autorisés*

module TEP — *Durée 2h30*

Université Pierre et Marie Curie — **Année 2012-2013**

Examen du 12 novembre 2012

Exercice 1 : Additionneur binaire (λ-calcul)

On va construire des additionneurs bit à bit à partir du codage des booléens et de la condition. On rappelle un codage possible des deux valeurs booléennes : $T = \lambda xy.x$ et $F = \lambda xy.y$, ainsi que celui de la conditionnelle : $cond = \lambda ce_1e_2.c\ e_1\ e_2$. Voici les tables de vérité d'un demi-additioneur à 1 bit et d'un additionneur complet (prenant en compte une retenue en entrée) à un bit :

demi-additionneur

A	B	A + B	retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

additionneur complet

A	B	R_e : retenue en entrée	A xor B xor REntrée	R_s : retenue en sortie
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1. demi-additionneur. Ecrire deux λ-termes, l'un calculant la somme $A + B$ et l'autre la retenue en sortie d'un demi-additionneur.
2. additionneur complet. Ecrire deux λ-termes l'un calculant la somme $A + B + R_e$ et l'autre la retenue en sortie d'un additionneur complet.
3. additionneur 2 bits (il y a donc 4 entrées : $a_0\ b_0\ a_1\ b_1$). Ecrire trois λ-termes : les deux premiers calculant les deux résultats des additions et le troisième la retenue globale. Vous pouvez utiliser les termes des questions précédentes.
4. chaînage d'additionneurs : Proposer sans l'implanter un mécanisme pour pouvoir additionner et calculer la retenue de n bits ($a_0\ b_0\ a_1\ b_1\ a_2\ b_2\ ...\ a_n\ b_n$). Vous pouvez utiliser d'autres structures de données.
5. (Bonus) Implanter ce mécanisme.

Exercice 2 : Typage fonctionnel/impératif (OCaml)

On cherche ici à apprécier le système de types fonctionnel-impératif d'OCaml. On définit tout d'abord le type `t` suivant :

```
1 type ('a,'b) t = {a : 'a list; mutable b : 'b list};;
```

1. Soit ce premier programme OCaml :

```
1 let v1 = {a=[2]; b=["hello"]};;
2 let r1 = if v1.a = v1.b then v1.a else v1.b;;
3 let r2 = if List.length(v1.a) < List.length(v1.b) then v1.a else v1.b;;
4 let r3 = if List.length(v1.a) < List.length(v1.b) then []   else v1.b;;
```

Donnez quand cela est possible le type des déclarations et des expressions de ce programme, en indiquant la valeur du résultat calculé. Différenciez bien les variables des inconnues de types.

2. Même question pour le programme suivant :

```
let v2 = {a=[]; b=[]};;
let r4 = if v1.a = v1.b then v1.a else v1.b;;
v2.b <- [3];;
v2;;
r4;;
```

3. Même question pour ce 3ème programme :

```
let v3 = {a=[]; b=[]};;
let f x = List.length v3.a;;
let g y = if y = ["Hello"] then 1 else List.length y;;
v3.b <- [ List.length ];;
v3;;
let r5 = if [f] == v3.b then [f] else v3.b;;
v3.b <-[ g ];;
v3;;
r5;;
```

Exercice 3 : Objet et fonctionnel (Java)

On cherche à permettre en Java d'avoir un style fonctionnel/objet à la OCaml, c'est-à-dire de pouvoir écrire ce genre de programme :

```
# let f o  = o#m() ^"\n";;
val f : < m : unit -> string; .. > -> string = <fun>
```

où `o` est un objet, instance d'une classe quelconque possédant la méthode `m` et retournant une chaîne de caractères.

Pour cela on va comparer deux techniques d'implantation en appréciant leurs avantages et limitations. La première utilise une interface `Fable` :

```
interface Fable {
 String m();
}
class A {
 public static String apply_f(Fable o, Object x){return o.m();}
}
```

et la deuxième les mécanismes d'introspection de Java. Le listing suivant montre une version allégée utilisant l'introspection (`apply_f`) :

```
class B {
 public static String apply_f(Object o, String m) {
  Class c=(o.getClass());
  Method f = c.getMethod(m,null);
  Object r = f.invoke(o,null);
  return (String)r^"\n";
 }
}
```

Soit le squelette de programme principal :

```
public static void main(String[] args) {
  Point p =new Point();
  String s = apply_f(p,"m");
  System.out.println(s);
}
```

On cherche à détailler les comportements du programme à la compilation et à l'excution dans les 3 cas suivants selon les propriétés de la classe `Point`.

1. Soit une classe `PointA` qui possède une méthode `m` de signature `String m()` et qui implante l'interface `Fable`.
2. Soit une classe `PointB` qui possède une méthode `m` de signature `String m()` et qui n'implante pas `Fable`
3. Soit une classe `PointC` qui ne possède pas de méthode `m`.

Exercice 4 : Surcharge et liaison tardive

Soient les classes suivantes :

```
1  class A {int a;
2    A(int a){this.a=a;}
3    public String toString(){return "["+a+"]";}
4    A max(A x){
5      if (this.compareTo(x) <= 0) return x;
6      else return this;
7    }
8    int getA(){ return a; }
9    int compareTo(A x){
10     int a2 = x.getA();
11     if (a < a2) return (-1);
12     else if (a > a2) return 1; else return 0;
13   }
14 }
```

```
1  class B extends A { int b;
2    B(int b){super(b/2);this.b=b;}
3    public String toString(){return "("+b+","+super.toString()+")";}
4    A max (A y) {if (this.compareTo(y) <= 0) return y; else return this;}
5    B max (B y) {if (this.compareTo(y) <= 0) return y; else return this;}
6    int getB(){return b;}
7    int compareTo(B y){
8      int b2 = y.getB();
9      int r = super.compareTo(y);
10     if (r == 0)
11       if (b < b2) return -1; else if (b > b2) return 1; else return 0;
12     else return r;
13   }
14 }
```

Indiquer sur le code suivant la signature (quand elle existe) de la méthode `max` en le justifiant, puis préciser quelle méthode sera exécutée et quel sera alors l'affichage obtenu.

1.

```
1        A a1 = new A(400);
2        B b1 = new B(100);
3        B b2 = new B(200);
4        A a2 = b2;
5        System.out.println(a1);
6        System.out.println(b1);
7        System.out.println(b2);
8        System.out.println(a2);
```

2.

```
1        A a3 = a1.max(a2);
2        A a4 = a2.max(a1);
3        B b3 = b1.max(b2);
4        B b4 = b2.max(b1);
5        System.out.println(a3);
6        System.out.println(a4);
7        System.out.println(b3);
8        System.out.println(b4);
```

3.

```
A a5 = b2.max(b1);
A a6 = b2.max(a1);
A a7 = b2.max(a2);
A a8 = a2.max(b2);
System.out.println(a5);
System.out.println(a6);
System.out.println(a7);
System.out.println(a8);
```

Exercice 5 : Génériques

On cherche à construire des visiteurs pour coder différents traitements sur des expressions arithmétiques simplifiées (constante, variable, addition et multiplication). Un visiteur sera une classe qui implante l'interface suivante :

```
interface IVisiteur<T> {
    T visite(Constante c);
    T visite(Add e);
    T visite(Mul e);
    T visite(Var v);
}
```

Pour cela on va définir une classe abstraite `Expression` qui possède une méthode abstraite `accepte` qui accepte un `IVisiteur<T>` et qui retourne une valeur de type `T`. Il faudra définir cette classe, ainsi qu'au moins les 4 classes filles utilisées dans l'interface (`Constante`, `Var`, `Addition` et `Multiplication`).

Voici par exemple un visiteur qui convertit une expression en chaînes de caractères :

```
class VisiteurTS implements IVisiteur<String> {
    public String visite(Constante c){return ( ""+c.getValue());}
    public String visite(Var v){return v.getNom();}
    public String visite(Add a){
        String s1 = a.getFg().accepte(this);
        String s2 = a.getFd().accepte(this);
        return "( "+ s1 + " + " + s2 + " )";
    }
    public String visite(Mul a){
        String s1 = a.getFg().accepte(this);
        String s2 = a.getFd().accepte(this);
        return "( "+ s1 + " * " + s2 + " )";
    }
}
```

1. Ecrire une hiérarchie de classes : `Expression`, `Constante` (entière), `Var`, `Add` et `Mul` qui respecte cette spécification.
2. Ecrire un visiteur d'évaluation des expressions en `Integer`. Ce visiteur ne sait évaluer que les expressions sans variable et déclenche une `RuntimeException` de votre choix s'il rencontre une variable.
3. Indiquer comment se comporte le programme suivant en traçant les appels aux visiteurs :

```
VisiteurTS v1 = new VisiteurTS();
VisiteurEval v2 = nw VisiteurEval();
// ... soit e1 une expression sans variable
  String s = e1.accepte(v1);
  Integer i = e1.accepte(v2);
  System.out.println(s + "=" + i);
```

4. Détailler du point de vue des types comment construire des visiteurs retournant des types différents en vous appuyant sur le code donné et le code que cous avez écrit.