Master d'informatique | **module TEP** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2h30* | **Année 2011-2012**

Examen du 14 novembre 2011

Exercice 1 - codage des listes en λ-calcul

Le but de cet exercice est de construire les fonctions classiques sur les listes à partir du codage suivant. La liste $[e_1\,;\,e_2\,;\,...\,e_n]$ est représentée par $\lambda fx.fe_1(fe_2\ldots(fe_nx)\ldots)$. La liste vide est représentée alors par *nil* : $\lambda fx.x$ et la fonction *cons* qui prend un élément e_0 et une liste $[e_1\,;\,e_2\,;\,\ldots\,;\,e_n]$ et retourne la liste $[e_0\,;\,e_1\,;\,e_2\,;\ldots\,;\,e_n]$ par *cons* : $\lambda alfx.fa(lfx)$.

1. Montrer que *hd* la fonction qui prend une liste et retourne sa tête réprésentée par : $hd : \lambda l.l\ (\lambda xy.x)\ nil$ vérifie bien :

   $hd\ (cons\ a\ l) \rightarrow^* a$

   $hd\ nil \rightarrow^* nil$

   On ne demande pas de coder la fonction *tl* qui retourne la queue d'une liste non nulle et nil sinon, mais on pourra la supposer connue par la suite.
2. Ecrire une fonction *length* qui prend une liste et retourne sa longueur sous la forme d'un entier de Church dont la définition est rappelée : un entier naturel n de Church est représenté par $\lambda fx.f^nx$
   - $\overline{0} = \lambda fx.x$
   - $\overline{1} = \lambda fx.fx$
   - $\overline{n} = \lambda fx.f^nx$ ce qui équivaut à $\lambda fx.(f(f\ldots(f\ x)\ldots))$ avec n applications de f
3. Soit le codage suivant de la fonction `map` : $\lambda gl.l\ (\lambda al_2.\ cons\ (g\ a)\ l_2)\ nil$ qui applique une fonction à une liste et retourne la liste des résultat de l'application de cette fonction aux éléments de la liste, montrer que : $map\ g\ [e_1;e_2\ldots.;e_n] \equiv_\beta [g\ e_1; g\ e_2;\ldots;g\ e_n]$ sachant que $l\ h\ r \equiv_\beta h\ e_1(l_2\ h\ r)$ où $l_2 = tl\ l$.
4. Que retourne le terme *myst* suivant : $\lambda ll_2.l\ cons\ l_2$ quand on l'applique à deux listes ?

Exercice 2 - de l'évaluation retardée en Caml

On cherche à implanter un mécanisme de liaison retardée dans un langage fonctionnel-impératif statiquement typé. Pour cela on va retarder le calcul des expressions retardées en capturant celles-ci dans des fermetures.

On définit tout d'abord un nouveau type pour les valeurs retardées :

```
type 'a c = Immediat of 'a | Retard of (unit -> 'a);;
type 'a v = {v : 'a c};;
```

sous la forme d'un enregistrement à un champ qui contient soit une valeur calculée, soit une expression dont l'évaluation a été retardée.

Ainsi on peut introduire une construction, nommée `late expr`, qui construit une expression dont l'évaluation est retardée. Par exemple l'expression `let x = 20 in late (3-x)` se traduit en `let x = 20 in { v = Retard (fun () -> 3 - x)}`.

1. Ecrire une fonction `eval` de type `'a v -> 'a` qui évalue une expression déjà évaluée ou bien retardée, et retourne la valeur évaluée.

2. On définit les deux fonctions suivantes :

```
let si e1 e2 e3 = if eval e1 then eval e2 else eval e3;;
let rec sigma n = si (late (n = 0)) (late 0) (late (n + sigma(n-1)));;
```

Donner le type de la fonction `si`, puis réécrivez la fonction `sigma` en expansant le code de `late`, et enfin donner le type de la fonction `sigma`.

3. Qu'affichent et que calculent les lignes suivantes :

```
let u = late (print_int 10; 10);;
sigma (eval u);;
sigma (eval u);;
```

4. Pour éviter de calculer à chaque demande la valeur d'une expression retardée, on modifie le type `'a v` en rendant `mutable` son unique champ de la manière suivante :

```
type 'a v = {mutable v : 'a c}
```

Modifier en conséquence la fonction `eval` précédente pour conserver la valeur évaluée lors de la première demande d'évaluation d'une expression à évaluation retardée.

5. Qu'affichent et que calculent alors les lignes suivantes :

```
let u = late (print_int 10; 10);;
sigma (eval u);;
sigma (eval u);;
```

justifiez s'il y en a les changements pas rapport à votre réponse à la question 3.

Exercice 3 : typage fonctionnel impératif

On cherche ici à apprécier le système de types fonctionnel-impératif d'OCaml.

1. Soit ce premier programme OCaml :

```
let id n = n;;
let l = ref [id] ;;
let add g l = l:=g :: !l;;
add (fun x -> x + 1) l;;
List.hd (!l) 1;;
add (fun x -> not x) l;;
List.hd (!l) true ;;
```

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. Soit ce deuxième programme OCaml :

```
let id n = n;;
let l = ref [] ;;
let add g l = l:=g :: !l;;
add (fun x -> [x]) l;;
List.hd (!l) 1;;
add (fun x -> [not x]) l;;
List.hd (!l) true;;
```

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.

Exercice 4 : Résolution de la surcharge

Soient les classes Java suivantes :

```
class A {
  void m(C x) {       /* A1 */ }
  void m(C x, B y) {  /* A2 */ }
}
class B extends A {
  void m (B x) {       /* B1 */}
  void m (B x , C y) { /* B2 */ }
  void m (C x , B y) { /* B3 */ }
}
class C extends B {
  void m (A x) {       /* C1 */ }
  void m (C x, A 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(c);     b.m(c);  //  QUESTION 1
    b.m(b1);   c.m(b1);    c.m(a2);
    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);
  }
}
```

1. Dans un tableau, indiquer quand elle existe la signature de la méthode `m` sélectionnée en Java 1.6 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 différents 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 5 : Files d'attente génériques

Soit la classe O'Caml `['a] queue` suivante :

```
exception Empty

class ['a] oqueue3 () =
  object(self)
    val mutable q = ([] : 'a list)
    method enq x =  q <-  q @ [x]
    method deq () = match q with
       [] -> raise Empty
    |  h::r -> q <- r ; h
    method reset () = q <- []
    method fold : 'b.  ('b -> 'a -> 'b) -> 'b -> 'b =  fun f  accu ->
       List.fold_left f accu q
  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)`.

1. Donner le type OCaml des méthodes `enq` et `deq`.
2. Ecrire une classe paramétrée Java (1.6) équivalente, nommée Queue<T>. Dans une première version on n'écrira pas la méthode `fold`.

3. Sans l'implanter, indiquez comment feriez-vous pour simuler le comportement de `fold_left` 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.

```
Queue qq = qb;
qq.enq("Essai");
Object o = qq.deq();
String s = (String)o;

qq.enq(o);
qq.enq(new Integer(33));
System.out.println(qq.deq());
System.out.println(qq.deq());

qq.enq(o);
qq.enq(new Integer(12));
int r = qb.deq() + qb.deq();
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 `Double`, et que toutes les classes héritent 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);
```