Typage et Analyse Statique

Cours 5

Emmanuel Chailloux

Parcours Science et Technologie du Logiciel
Master mention Informatique
Sorbonne Université

année 2021-2022

Plan du cours 5

Surchage: (overloading - polymorphisme *ad hoc*)

- Surcharge en Java
  - par l'exemple
  - algorithmes de résolution
- Autres modèles
  - Classes de type en Haskell
  - Inlining en F#
  - Modules implicites en OCaml

Résolution de la surcharge

- choix du type de la méthode à employer lors d'un appel de méthode
- résolue STATIQUEMENT selon une relation d'ordre sur la classe de définition et le type des arguments
- le type du résultat n'est pas pris en compte
- il y a des cas où la résolution échoue

```
1  class A :  m2(A) m2(A,A)
2  class B :  m2(B) m2(A,B) m2(B,A)
3  class C :  m2(A) m2(B,C) m2(C,A)
```

avec C qui hérite de B qui hérite de A.

```
1  A a1 = new A();
2  B b1 = new B();
3  C c1 = new C();
4  c1.m2(x,y)
```

Quel est le type de la méthode à utiliser en fonction des types de x et y ?

Sélection du type de la méthode (1)

Au niveau de la classe C on a 8 méthodes dont 5 à 2 arguments

	classe de définition	type de la méthode
m2	A	(A)
	A	(A,A)
	B	(B)
	B	(A,B)
	B	(B,A)
	C	(C)
	C	(B,C)
	C	(C,A)

Sélection du type de la méthode (2)

```
1  c1.m(x,y);
```

Sur l'ensemble des méthodes `m2` du receveur (ici `c1`), on ne conserve que celles dont le type (t1,t2) vérifie :

- type de x $\leq t_1$
- type de y $\leq t_2$

c1.m2(a1,b1) ;

	classe de définition	type de la méthode
m2	A	(A,A)
	B	(A,B)

Le choix de la signature de la méthode se portera sur la méthode la plus spécifique au sens du sous-typage (ici celle de signature `(A,B)`.

Cas d'ambiguïté

Soit une classe B avec 2 méthodes `m2(A,B)` et `m2(B,A)` :

```
1  A a1 = new A();
2  B b1 = new B();
3  b1.m2(a1,a1);
```

Sur les types on obtient B x (A, B) et B x (B, A)
Aucune de ces deux méthodes ne possède un type tel que (A,A) soit plus petit. il y a un clash à la compilation ! ! !

```
1  b1.m2(b1,b1)
```

Ici les deux méthodes ont un type tel que (B,B) soit plus petit, mais aucune des deux méthodes est plus petite que l'autre il y a un clash à la compilation ! ! !

Surcharge et liaison tardive

```
1  package pobj.cours4;
2  class A {
3      void m2(A a) {System.out.println("A1");}
4      void m2(A a1, A a2) {System.out.println("A2");}    }
5  class B extends A {
6      void m2(B  b) {System.out.println("B1");}
7      void m2(B b1, A a2) {System.out.println("B2");}    }
8  class C extends B {
9      void m2(C  c) {System.out.println("C1");}
10     void m2(A a1, A a2) {System.out.println("C2");}
11     void m2(B b1, B b2) {System.out.println("C3");}    }
12 class TestSurcharge {
13   public static void main(String [] args) {
14      A a1 = new A();
15      B b1 = new B();
16      C c1 = new C();
17      B b2 = c1;
18      A a2 = c1;
19      a1.m2(c1,c1);  // A2
20      b1.m2(c1,c1);  // B2
21      c1.m2(c1,c1);  // C3   b2, c1 et a2 partagent un objet
22      b2.m2(c1,c1);  // B2   mais ne sont pas de me^me type
23      a2.m2(a2,a2);  // C2
24 }}
```

Visiteur : extension des traitements

On cherche à séparer les données des traitements dans le but d'étendre les traitements. Pour cela les classes pour les expressions arithmétiques accepteront de recevoir un visiteur qui effectuera un traitement particulier (calcul de la valeur de l'expression, transformation en chaînes de caractères, ...).
On défini alors une classe abstraite visiteur qui pourra visiter chaque élément des expressions arithmétiques :

```
1 abstract class Visiteur {
2   public abstract void visite(CteV c);
3   public abstract void visite(AddV a);
4   public abstract void visite(MultV m);
5 }
```

la classe abstaite pour les expressions devient alors :

```
1 abstract class ExprArV {
2   public abstract void accepte(Visiteur v);
3 }
```

séparation effective des données et des traitements.

Expressions arithmétiques acceptant un visiteur

```
1  class CteV extends ExprArV {
2    private int val;
3    public CteV(int v) {val=v;}
4    public int getVal(){return val;}
5    public void accepte(Visiteur v){v.visite(this);}
6  }
7  abstract class OpBinV extends ExprArV {
8    protected ExprArV fg, fd;
9    ExprArV sous_expr_g(){return fg;}
10   ExprArV sous_expr_d(){return fd;}
11 }
12 class AddV extends OpBinV {
13   public AddV(ExprArV fg, ExprArV fd){
14     this.fg = fg; this.fd = fd;
15   }
16   public void accepte(Visiteur v){v.visite(this);}
17 }
18 class MultV extends OpBinV {
19   public MultV(ExprArV fg, ExprArV fd){
20     this.fg = fg; this.fd = fd;
21   }
22   public void accepte(Visiteur v){v.visite(this);}
23 }
```

Un visiteur de traitement de calcul

```
1  class VisiteurEval extends Visiteur  {
2    private int   res;
3    VisiteurEval(){res=0;}
4    public int  getRes(){return res;}
5
6    public void visite(CteV c){res=c.getVal();}
7
8    public void visite(AddV a){
9      int i;
10     a.sous_expr_g().accepte(this);
11     i=res;
12     a.sous_expr_d().accepte(this);
13     res=res + i;
14   }
15
16   public void visite(MultV m){
17     int i;
18     m.sous_expr_g().accepte(this);
19     i=res;
20     m.sous_expr_d().accepte(this);
21     res=res * i;
22   }
23 }
```

Test sur les expressions avec visiteur

```
1  package pobj.cours4;
2
3  class TestExprArV {
4    public static void main(String[] args) {
5      VisiteurEval v = new VisiteurEval();
6
7      ExprArV e1 = new CteV(10);
8      ExprArV e2 = new CteV(20);
9      ExprArV e3 = new MultV(e1,e2);
10     ExprArV e4 = new AddV(e1,e3);
11
12     e4.accepte(v);
13     System.out.println(e4 + " = " + v.getRes());
14   }
15 }
```

```
1  > java pobj/cours4/TestExprArV
2  pobj.cours4.AddV@1a0f73c1 = 210
```

Il manque un visiteur de conversion en chaîne de caractères pour afficher la formmule, mais le résultat est correct.

Redéfinition et co-variance du résultat

Depuis la version 1.5, il est possible dans le cadre d'une redéfinition de préciser (au sens du sous-typage) le type de retour. Le type du résultat de la méthode redéfinie est en relation de sous-typage avec celui défini dans la sur-classe. On parle de co-variance du type résultat.

- toujours pour éviter les *cast* inutiles

Exemple :

- dans `Point` :

```
1 protected Point clone () {
2   return new Point(getX(), getY());  }
```

- dans `PointColore` :

```
1 protected PointColore  clone () {
2   return new PointColore(getX(), getY());  }
```

- pas besoin de retourner un `Object` pour le transtyper ensuite en `Point` ou `PointColore`.

Surcharge et autoboxing

```
1  package pobj.cours4 ;
2
3  class TS2 {
4   public static int m1(int y){return 2 * y;}
5   public static Integer m1(Integer z){return 5 * z;}
6   public static int m2(int z){return 10 * z;}
7   public static void main(String[] a) {
8   int i = 1;
9   Integer k = new Integer(10);
10  System.out.println(m1(i) + "," + m1(k)); // 2,50
11  System.out.println(m1(m1(i)) +","+ m1(m1(k))); // 4,250
12  System.out.println(m2(i) +","+ m2(k)); // 10,100
13  System.out.println(m2(m2(i)) +","+ m2(m2(k))); // 100,1000
14  }
15 }
```

Algorithmes de résolution de la surcharge (1)

- First Match : on garde la première signature de méthode trouvée.
  simple mais peu efficace

- Perfect Match :
  ne garder que la signature de méthode pour laquelle tous les types sont identiques aux expressions passées en paramètre.
  Pas d'ambiguité mais très restrictive.

Algorithmes de résolution de la surcharge (2)

- Mult : Multiplication des distances entre les arguments et les paramètres de la méthode.

  calcul d'une distance de parentée entre deux types correspondant au nombre de liens de parenté nécessaires pour aller d'un type à l'autre : si B hérite de A et C hérite de B , alors la distance entre A et C est de 2. On calcule donc l'ensemble des distances, auxquelles on ajoute 1, et on les multiplie. On gardera bien entendue la méthode ayant la valeur minimale (si tous les arguments ont le même type que les paramètres de la méthode, on obtient la plus petite valeur possible : 1).

Algorithme de résolution de la surcharge (3)

- Java 1.2 : produit cartésien de sa classe de définition et des paramètres de la méthode.

  Les méthodes plus précises possèdent alors des valeurs plus grandes que les autres, ce qui permet de les sélectionner. A noter que plusieurs méthodes peuvent avoir la même valeur, ce qui peut mener soit à une impossibilité à typer, soit à une sélection purement arbitraire (au choix, par exemple la première méthode trouvée).

- Java 1.5 : produit cartésien des paramètres de la méthode. La classe fournissant la méthode n'est donc plus prise en compte. Cela évite qu'une méthode moins précise mais issue d'une sous-classe soit sélectionnée.

Algorithme de résolution de la surcharge (4)

- Mixte : On prend la signature la plus petite dans la classe la plus petite. A la différence de java 1.2 on ne recherche que dans les méthodes définies les dernières (recherche dans la classe la plus petite).

Exemple en Java 1.2 et 1.5 (1)

B hérite de A :

```java
class A {
  int m (A x) { System.out.println(1+" "); return (1); }
  boolean n (B x) {System.out.println(2+" "); return (true); }
}

class B extends A {
  int m (B x) { System.out.println(5+" "); return (5); }
  boolean n (A x) {System.out.println(6+" "); return(true); }
}
```

```java
A a1 = new A ();
B b1 = new B();
A a2 = b1;
```

Exemple en Java 1.2 et 1.5 (2)

1er appel :

```
1  b1.m(b1);
```

on a 2 méthodes candidates : A.m(A x) et B.m(B x) La résolution ne posera ici aucun souci : la méthode B.m(B x) est dans la plus petite classe, et possède une signature qui colle parfaitement avec le type de l'argument. C'est donc celle qui sera sélectionnée.

Exemple en Java 1.2 et 1.5 (3)

2ème appel :

```
1  b1.n(b1);
```

Les 2 candidats sont ici A.n(B x) et B.n(A x).

- Pour Java 1.2, ces méthodes sont représentées par les couples (A , B) et (B, A), en on n'en trouve pas un plus petit que l'autre : il y a ambiguïté.
- Pour Java 1.5, les méthodes sont représentées par (B) et (A), et la méthode A.n(B x) est donc plus précise. C'est cette signature qui sera sélectionnée.
- Pour Perfect Match et Multiply, la même méthode sera sélectionnée.

Cette différence de traitement entre Java 1.2 et les autres algorithmes montre sur un exemple simple l'importance de l'algorithme de résolution de la surcharge, en particulier dans les cas où il n'y a pas d'erreurs de compilation ; c'est-à-dire quand l'algorithme sélectionne une signature.

Exercice (1)

```
class A {
  void m(B x) {        /* A1 */ }
  void m(C x, B y) {   /* A2 */ }
}
class B extends A {
  void m (A x) {        /* B1 */}
  void m (B x , A y) { /* B2 */ }
  void m (C x , B y) { /* B3 */ }
}
class C extends B {
  void m (C x) {        /* C1 */ }
  void m (C x, C y) {   /* C2 */ }
  void m (C x, B y) {   /* C3 */ }
}
```

Exercice (2)

```
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(a1);    b.m(a1);  //  QUESTION 1
    b.m(b);    c.m(c);     c.m(b2);
    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);
  }
}
```

Exercice (3)

Pour chacune des questions suivantes vous rappellerez ou préciserez la relation d'ordre employée pour la comparaison des signatures des méthodes.

1. Dans un tableau, indiquer quand elle existe la signature sélectionnée de la méthode `m` en Java 1.7 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

Haskell

- fonctionnel
- pur
  sans effets de bord
- à évaluation paresseuse
  stratégie standard du λ-calcul
- typé statiquement (fortement)
  polymorphe paramétrique et
  ad hoc : types de classes pour la surcharge
- avec inférence de types

plusieurs compilateurs (ghc (Glasgox Haskell Compiler)), livres, sites dont le très riche `http://www.haskell.org`.

de OCaml à Haskell (1)

déclaration de type:

`type` pour les synonymes

`data` pour les types avec constructeurs

```
1 data Couleur = Pique | Coeur | Carreau | Trefle
2 data Maybe a = Just a | Nothing
3 data Arbre a b = Feuille b | Noeud a (Arbre a b) (Arbre a b)
```

contrainte de type pour une déclaration :

```
1 succ :: Integer -> Integer
2 succ n = n + 1
```

de OCaml à Haskell (2)

fonctions et applications:

fonction anonyme ($\lambda xy.x + y$)

```
1 \x y ->  x + y
```

composition ($f\ o\ g\ o\ h$)

```
1 f . g . h
```

opérateur d'application $

```
1 a b c x = ((a b) c) x
2 a $ b $ c $ x = a ( b ( c x))
```

de OCaml à Haskell (3)

une écriture équationnelle :

```
1  sign x |  x >  0       =   1
2         |  x == 0       =   0
3         |  x <  0       =  -1
```

la fonction map :

```
1      map :: (a -> b) -> [a] -> [b]
2      map _ [] = []
3      map f (x:xs) = f x : map f xs
4
5  > map (+3) [1,5,3,1,6]
6  [4,8,6,4,9]
7  > map reverse ["abc","cda","1234"]
8  ["cba","adc","4321"]
```

Surcharge

- en OCaml
  - un opérateur différent pour chaque type argument
    - opérateurs de calcul : `+` pour les *int*, `+.` pour les *float*
    - ou des fonctions de conversion explicite : `string_f_int`, `string_of_float`
  - écrire des fonctions paramétriques qui explorent la structure des valeurs (builtin)
    - égalité structurelle : (=) : `'a -> 'a -> bool` qui explore la structure des arguments en exécutant un code différent
      limitation : impossible à écrire dans le langage
- en Haskell
  - classes de types (ou type class)

Classes de types

Une classe de types définit une propriété sur les types.
Par exemple les types avec égalité :

```
1  class Eq a where
2    (==) :: a -> a -> Bool
```

indique que les types de la classe `Eq` possèdent l'opérateur d'égalité `==`

une classe de types peut avoir une ou plusieurs implantations (instances) pour des types donnés :

```
1  instance Eq Bool where
2  x == y = if x then y else not y
3
4  instance Eq Couleur where
5    x == y = ...
```

Contraintes de typage

On peut ajouter des contraintes de types sur des paramètres de types de fonctions polymorphes : ici le prédicat `mem` qui retourne True si un élément appartient à une liste et False sinon doit pouvoir tester l'égalité du paramètre x avec les éléments de la liste.

```
1 mem : Eq a => a -> [a] -> Bool
2 mem x [] = False
3 mem x (y :: ys) = x == y || mem x ys
```

```
1 > mem True [False, False, False]
2 False
3 > mem True [False, True, False]
4 True
```

Conjonctions de contraintes

Soit la classe `Show a` définissant `show` :

```haskell
class Show a where
  show :: a -> String
```

on peut utiliser des conjonctions de contraintes comme sur l'exemple suivant :

```haskell
filter_show :: (Eq a, Show a) => (a -> Bool) -> [a] -> String
filter_show p l = foldl  write ""  (filter p l)
   where write ""  obj = show obj
         write str obj = str ++ "," ++ show obj
```

la fonction `filter_show` filtre les éléments d'un liste en fonction d'un prédicat et convertit en une seule chaîne de caractères les éléments sélectionnés. Ces éléments de liste sont des instances de `Eq a` et `Sort a`.

contraintes sur les instances

Une instance peut être être paramétrée, et donc on peut faire porter une contrainte sur les paramètres de types.

```
1 data Ordering = LT | EQ | GT
2 class Ord a where
3 compare :: a -> a -> Ordering
```

voici une instance des éléments ordonnés pour des listes quelconques dont les éléments sont ordonnés.

```
1 instance Ord a => Ord [a] where
2 compare [] [] = EQ
3 compare _  [] = GT
4 compare [] _  = LT
5 compare (x : xs) (y : ys) = case compare x y of
6    EQ -> compare xs ys
7    other -> other
```

Contraintes sur les classes

On peut contraindre qu'une classe de type `SC` fournisse toutes les fonctions d'une classe de type `C` à la manière de l'héritage.
Par exemple la classe des éléments bornés (Bounded) intègre les méthodes de la classe des éléments ordonnés (Ord)

```haskell
class Ord a => Bounded a where
  maxBound :: a
  minBound :: a
```

Ainsi toute instance de `Bounded a` aura d'une part la fonction `compare` mais aussi les bornes `maxBound` et `minBound`.

Classes standards

- types avec égalité et ordre :
  - Eq a – x == y
  - Ord a – compare, (<) , (<=)
  - Bounded – minBound, maxBound
- types numériques
  - Num a – structure d'anneau (+, -, *, ...)
  - Fractional a – inverse, quotient (/)
  - Floating - ajoute les fonctions trigonométriques et transcendentes
- utilitaires
  - Show a – show : : a -> String
  - Enum a – types indexables par des entiers (fromEnum, toEnum)

Dérivations

- définition automatique sur les énumérations :

```
1  data Couleur = Pique | Coeur | Carreau | Trefle deriving Enum
```

on obtient un fromEnum : `Couleur -> Int` et la fonction inverse de type `Int -> Couleur`

```
1  data Arbre a = Feuille a | Noeud a (Arbre a) (Arbre a)
2  deriving (Eq, Show)
```

engendrent (automatiquement) les instances suivantes :

```
1  instance Eq a => Eq (Arbre a) where ...
2  instance Show a => Show (Arbre a) where ...
```

Surcharge en F# - en objets

```
type Vector(x: float, y : float) =
   member this.x = x
   member this.y = y
   static member (~-) (v : Vector) =
     Vector(-1.0 * v.x, -1.0 * v.y)
   static member (*) (v : Vector, a) =
     Vector(a * v.x, a * v.y)
   static member (*) (a, v: Vector) =
     Vector(a * v.x, a * v.y)
   override this.ToString() =
     this.x.ToString() + " " + this.y.ToString()
```

```
> let v1 = Vector(1.0, 2.0);;
val v1 : Vector = 1 2
> let v2 = v1 * 2.0;;
val v2 : Vector = 2 4
> let v3 = 2.0 * v1;;
val v3 : Vector = 2 4
> let v4 = - v2;;
val v4 : Vector = -2 -4
```

Surcharge en F# - inlining (1)

le mot clé `inline` indique que l'intégration des appels de la fonction avec une contrainte sur les paramètres de type :

```
1  > let add x y = x + y ;;
2  val add : x:int -> y:int -> int
3  > let inline add x y = x + y ;;
4  val inline add :
5    x: ^a -> y: ^b ->  ^c
6      when ( ^a or  ^b) : (static member ( + ) :  ^a *  ^b ->  ^c)
```

ic ^a ou ^b, un des types des paramètres de (add doit avoir une fonction + infixe définie ; si oui cela répercute les contraintes de type sur l'autres paramètre

```
1  > add "hello" "world" ;;
2  val it : string = "helloworld"
3  > add 1.2 4.3 ;;
4  val it : float = 5.5
```

Surcharge en F# - inlining (2)

et sinon c'est une erreur de typage.

```
> add (true,true);;
  add (true,true);;
  ^^^
/Users/emmanuel/All/cvs/Enseignement/P6/2015-2016/L/mpil/SVN/2015-2016/cours/stdin(9,1): error FS0071:
Type constraint mismatch when applying the default type '(bool * bool)' for a type inference variable.
 Expecting a type supporting the operator '+' but given a tuple type Consider adding further type constraints
```

Surcharge en F# - inlining (3)

```
> type point = {x:int; y : int}
   with static member
      (+) (p1,p2) = {x=p1.x + p2. x; y = p1.y + p2.y};;
type point =
  {x: int;
   y: int;}
  with
    static member ( + ) : p1:point * p2:point -> point
  end
> let p1 = {x = 10; y = 20} ;;
val p1 : point = {x = 10;
                  y = 20;}
> add p1 p1 ;;
val it : point = {x = 20;
                  y = 40;}
> add 1 2 ;;
val it : int = 3
```

Modules implicites pour la surcharge en OCaml

Dans le cadre du polymorphisme paramétré, il est difficile de résoudre la surcharge en cas de types paramétrés. Par exemple s'il y a plusieurs opérateurs +, de types `(int * int) -> int` et `(float * float) -> float` une solution est de passer une information de typage du paramètre de type au moment de l'appel.

```
1  let add x y = x + y ;;
2
3  (* add 3 4 ---> add [^Int^] 3 4  *)
```

facilitant ainsi la résolution de la surcharge de l'opérateur + car cet argument supplémentaire permet de savoir quel est l'opérateur + à utiliser.

Cet argument supplémentaire peut être passer implicitement ce qui évite d'alourdir les points d'appel à add.

Proposition d'implicites via les modules

- inspirations :
  - classes de types : à la Haskell

```haskell
class Show a where show :: a -> String
show :: Show a => a -> String
```

  - implicites comme en Scala : un paramètre peut-être indiqué comme implicite
  - modular type class : utilisation des types de modules comme classes de type (à la Haskell)
- papier de référence :

Modular implicits

Léo White, Frédéric Bour,Jeremy Yallop

http://arxiv.org/pdf/1512.01895.pdf

Exemple : syntaxe expérimentale

```
module type Show =
sig
  val show : {S: Show} -> S.t -> string
end
let show {S : Show} (v : S.t) = S.show v
implicit module Show_int =
struct
 type t = int
 let show = string_of_int
end

show {Show_int} 3
show 3

implicit module Show_list {A: Show} =
struct
 type t = A.t list
 let show l =
   "["^ String.concat ", "  (List.map A.show l) ^"]"
end
```