Typage et Analyse Statique

Cours 5

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Sorbonne Université

année 2019-2020

Plan du cours 5

Programmation par objets:

- Surcharge en Java
  - par l'exemple
  - algorithmes de résolution
- Classes de type en Haskell

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 :

```
1  class A {
2    int m (A x) { System.out.println(1+" "); return (1); }
3    boolean n (B x) {System.out.println(2+" "); return (true); }
4  }
5
6  class B extends A {
7    int m (B x) { System.out.println(5+" "); return (5); }
8    boolean n (A x) {System.out.println(6+" "); return(true); }
9  }
```

```
1  A a1 = new A ();
2  B b1 = new B();
3  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 foctions paramétriques qui explorent la strucutre 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 pour des types donnés :

```
1  instance Eq Bool where
2    x == y = if x then y else not y
3  instance Eq Couleur where
4    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 par 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    XEQ -> compare xs ys
7    other -> other
```

Contraintes sur les classes

On peut contraindre qu'une classe de type `SC` fournisse toutes les fonctions d'uen 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)

```
1  class Ord a => Bounded a where
2    maxBound :: a
3    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 égalitét ordre :
  - Eq a – x == y
  - Ord a – compare, (<) , (<=)
  - Bounded – minBound, maxBound
- types numériques
  - Num a – structure d'anneau (+, -, *, ...)
  - Frational 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 un `Int -> Couleur`

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

engendrent les instances suivantes :

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