

Typage et Analyse Statique

Cours 6

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2015-2016

Plan du cours

- ▶ λ -expression en Java 1.8 (cours 05)
- ▶ exemple simple de génériques en Java
- ▶ GADT en OCaml
- ▶ panorama des langages
- ▶ présentation (rapide) de Scala (slides 10-20 popl06 Odersky)
 - ▶ objets
 - ▶ filtrage par motifs
 - ▶ fonctions
- ▶ un petit mot sur Swift (noyau fonctionnel-impératif) pour le programmeur OCaml

Exemple génériques en Java (1)

retour sur la méthode copy :

```
1  class Test {
2
3      static <T> void copy( List<? extends T> src,
4                          List<? super T> dst) {
5          for (T element : src) { dst.add(element); }
6      }
7
8      public static void main(String[]a) {
9          List<Integer> li = new ArrayList<Integer>(10);
10         li.add(10); li.add(20);
11         List<Object> lo = new ArrayList<>();
12         lo.add("Hello");
13         // ok
14         copy(lo,lo);
15         copy(li,li);
16         copy(li,lo);
17         // ...
```

Exemple génériques en Java (2)

```
1 // pas ok
2 //   copy (lo,li);
3   }
4 }
```

```
1 Test.java:19: error: method copy in class Test cannot be applied
2   to given types;
3     copy (lo,li);
4     ^
5   required: List<? extends T>,List<? super T>
6   found: List<Object>,List<Integer>
7   reason: inferred type does not conform to upper bound(s)
8     inferred: Object
9     upper bound(s): Integer,Object
10  where T is a type-variable:
11    T extends Object declared in method <T>copy(List<? extends T>,List<? ←
12    super T>)
1 error
```

Types algébriques généralisés : GADT

But: préciser le typage sur les paramètres de types

- ▶ les contraintes sur les paramètres de type peuvent changer en fonction des constructeurs
- ▶ les variables de types sont :
 - ▶ existentielles quand elles sont en position argument d'un constructeur

Syntaxe :

```
constr-decl ::= ...
```

```
| constr-name : typexpr { * typexpr } -> typexpr
```

```
type-param ::= ...
```

```
| [variance] _
```

Exemple sur les listes

nouvelles listes : 1 paramètre de type pour distinguer liste vide et liste nonvide, l'autre paramètre de type pour les éléments de la liste

```
1 type vide ;;
2 type pasvide ;;
3 type ('a, 'b) liste =
4   Nil : (vide,'b) liste
5 | Cons : 'b * ('a, 'b) liste -> (pasvide, 'b) liste ;;
6
7 # let l1 = Nil;;
8 val l1 : (vide, 'a) liste = Nil
9 # let l2 = Cons(3,Nil);;
10 val l2 : (pasvide, int) liste = Cons (3, Nil)
11 # let tete l = match l with Cons(x,_) -> x;;
12 val tete : (pasvide, 'a) liste -> 'a = <fun>
```

pas de *warning* sur des cas de filtrage non exhaustifs non pertinents.

Exemple : un évaluateur sans GADT

```
1  type expr = I of int | B of bool | Add of expr * expr
2  | If of expr * expr * expr;;
3
4  # let add e1 e2 = match (e1,e2) with
5    I i1, I i2 -> i1+i2
6  | _ -> failwith "add";;
7  val add : expr -> expr -> int = <fun>
8  # let rec eval e = match e with
9    I i -> I i
10   | B b -> B b
11   | Add (e1,e2) -> I (add (eval e1) (eval e2))
12   | If (e1, e2, e3) -> (match (eval e1) with B b -> if b then eval e2 else ←
    eval e3 | _ -> failwith "If") ;;
13  val eval : expr -> expr = <fun>
14
15  # let e1 = Add (I 3, If (B true, I 10, I 20));;
16  val e1 : expr = Add (I 3, If (B true, I 10, I 20))
17  # eval e1;;
18  - : expr = I 13
19  # let e2 = Add (I 3, B true);;
20  val e2 : expr = Add (I 3, B true)
21  # eval e2;;
22  Exception: Failure "add".
```

Exemple (avec GADT)

```
1 type 'a expr =
2   I : int -> int expr
3 | B : bool -> bool expr
4 | Add : int expr * int expr -> int expr
5 | If : bool expr * 'a expr * 'a expr -> 'a expr ;;
6
7 # let rec eval : type a. a expr -> a = function e -> match e with
8     I i -> i
9   | B b -> b
10  | Add (e1,e2) -> (eval e1) + (eval e2)
11  | If (e1,e2,e3) -> if (eval e1) then (eval e2) else (eval e3);;
12 val eval : 'a expr -> 'a = <fun>
13
14 # let e1 = Add (I 3, If (B true, I 10, I 20));;
15 val e1 : int expr = Add (I 3, If (B true, I 10, I 20))
16 # eval e1;;
17 - : int = 13
18
19 # let e2 = Add (I 3, B true);;
20 Error: This expression has type bool expr
21      but an expression was expected of type int expr
```


Langages et types

- ▶ typage statique : Ada, OCaml, Eiffel, Haskell
- ▶ typage dynamique : Lisp, Scheme, CLOS, Smalltalk
- ▶ typage statique ET dynamique :
 - ▶ Java 1.4, C# 1.0
 - ▶ Java 1.5, C# 2.0

tendance typage statique : sûreté + généricité

Polymorphismes

Dans le cadre du typage statique :

- ▶ paramétrique (à la ML)
- ▶ *ad hoc* ou de surcharge
- ▶ objets ou d'inclusion

volonté d'intégration des différents polymorphismes dans les langages statiquement typés

Inférence de types

- ▶ totale en OCaml, Haskell
 - ▶ sauf dans les interfaces .mli
 - ▶ peut être nécessaire de l'aider
 - ▶ extension objet, ...
- ▶ partielle pour la résolution des wildcard (<?>) en Java 1.5 ou des diamants (<>) en 1.7

Surcharge et inférence - Ocaml (1)

en OCaml : difficultés liées à l'inférence

- ▶ polymorphisme paramétrique

```
1 # let g o = o#m();;  
2 val g : < m : unit -> 'a; .. > -> 'a = <fun>  
3 # let f x y = x#m y;;  
4 val f : < m : 'a -> 'b; .. > -> 'a -> 'b = <fun>
```

Surcharge et inférence - OCaml (2)

- ▶ application partielle :
Soit une classe `c` ayant 2 méthodes :
 - ▶ `m : int -> int -> int`
 - ▶ `m : int -> float`

```
1 let g (o:c) (i:int) = o#m i;;
```

Quelle méthode `m` choisir ?

Surcharge et inférence - Haskell

En Haskell avec les types de classes :

- ▶ définir des types de classes regroupant des ensembles de fonctions surchargées.
- ▶ Une déclaration de classe définit une nouvelle classe et les opérateurs que celle-ci autorise.
- ▶ Une déclaration d'instance (d'une classe) indique qu'un certain type est une instance d'une classe. Cela inclue la définition des opérateurs surchargés de sa classe pour ce type.

```
1 class Num a where
2   (+)    :: a -> a -> a
3   negate :: a -> a
```

Surcharge et inférence - OCaml extension

Extension du système de types en OCaml

- ▶ Langage reFLect (Intel) et surcharge :

```
1 overload print : IO.output -> 'a -> unit
2 overload _+_ : 'a -> 'a -> 'a
3
4 instance print Int.print Float.print
5 instance _+_ Int._+_ Float._+_
```

voir : <http://gallium.inria.fr/~pouillar>

Génériques (1)

classes paramétrées :

- ▶ C++ : code spécialisé pour chaque instance de template
- ▶ OCaml : reste dans le cadre du polymorphisme paramétrique de la couche fonctionnelle
- ▶ Java : même machine virtuelle, compatibilité ascendante, code engendré compatible avec la JVM, polymorphisme borné
- ▶ C# 2.0 : machine virtuelle avec instructions génériques
 - ▶ Design and Implementation of Generics for the .NET Common Language Runtime Andrew Kennedy and Don Syme. PLDI 2001
 - ▶ research.microsoft.com/en-us/um/people/akenn/generics/index.html

Génériques (2)

- ▶ même code pour tout type :
 - ▶ représentation uniforme des données (α)
 - ▶ paramètre supplémentaire (class restriction) puis dispatch
 - ▶ monomorphisation : code spécialisé pour chaque instance :
⇒ code plus important
 - ▶ casts sûrs dans le code :
⇒ code moins rapide
- ⇒ conséquences sur les performances (GC, ...)

Intégration (1)

- ▶ Styles de programmation
 - ▶ fonctionnel (paramétrique)/objet (à la OCaml)
 - ▶ fonctionnel (paramétrique)/*ad hoc* : fonctions génériques (à la CLOS ou à la Haskell)
 - ▶ objet/*ad hoc* : Java 1.4, C# 1.0
 - ▶ objet/*ad hoc*/paramétrique : Java 1.5, C# 2.0
- ▶ Mélange
 - ▶ F# : C# pour l'objet + caml-light pour le fonctionnel/impératif
 - ▶ C# 3.0 : λ -expressions, inférence de types (var locales)

```
1 x => x + 1 // Implicitly typed, expression ↔  
    body  
2 x => { return x + 1; } // Implicitly typed, statement body  
3 (int x) => x + 1 // Explicitly typed, expression ↔  
    body  
4 (int x) => { return x + 1; } // Explicitly typed, statement body  
5 (x, y) => x * y // Multiple parameters  
6 () => Console.WriteLine() // No parameters
```

- ▶ Java 1.8 : λ -expressions, streams

Intégration (2)

- ▶ Comparaison :

- ▶ Narbel Ph :

- Programmation fonctionnelle, générique et objet. Vuibert. 2005.

- comparaison des structurations modules et classes pour la généricité.*

- ▶ Autres langages

- ▶ Scala : langage statiquement typé, fonctionnel, impératif, objet pour Java et .NET, en Java permet d'avoir accès aux bibliothèques Java directement ; syntaxe à la Java, simplifie les génériques, filtrage de motifs, modularité à la mixin
prochain cours : <http://www.scala-lang.org>
 - ▶ Swift : langage statiquement typé, fonctionnel, impératif, objet pour Objectif C, permet d'avoir accès aux bibliothèques Objective C, environnement de développement sous Xcode

Interopérabilité

- ▶ FFI : foreign function interface (external OCaml, JNI Java)
- ▶ IDL : fonctions, données (COM)
- ▶ IDL : objets (exemple O'Jacaré)
- ▶ types abstraits de données, Goji (OCaml - JS) :
<http://ocaml.org/meetings/ocaml/2013/slides/canou.pdf>
<https://github.com/klakplok/goji>

runtime commun : facilite l'interopérabilité (O'Jacaré.NET) pour la gestion mémoire (GC), la concurrence (threading), ...

- ▶ ocamil, ocamljava, ocamlcc, js_of_ocaml, ...

Swift - noyau fonctionnel-impératif(1)

▶ let et var

```
1 let x = 3
2 var y = 8
3 // x = 5
4 y = 5
```

▶ fonctions et paramètres (in, out et inout)

```
1 func add1(n:Int)->Int { return n+1;}
2 func incremp (inout x : Int) { x = x + 1}
3 func incr(inout x : Int) -> Int {
4     let y = x
5     x = x + 1
6     return y
7 }
8 var u = 4
9 println(u) ; println(add1(u)); println(u)
10 println(incr(&u)) ;println(u)
11 // 4 5 4 4 5
```

Swift - noyau fonctionnel-impératif (2)

fonction et fermeture : compose

```
1 func compose (f : (Int)->Int,
2               g: (Int) -> Int,
3               x : Int) -> Int {
4     return f (g( x))
5 }
6
7 let v = compose(add1,add1,8)
8 println(v)
9
10 let a = 8
11 let w = compose(add1, {(n : Int) -> Int in return n * a}, 2)
12 println(w)
13
14 // 10 17
```

Swift - noyau fonctionnel-impératif (3)

avec du polymorphisme paramétrique

```
1 func long<T>(l : [T]) -> Int {return l.count}
2
3
4 func mapAlphaList<T,P>(f : (T) -> P, l : [T]) -> [P] {
5     var e = [P]()
6     for item in l {
7         e.append(f(item))
8     }
9     return e
10 }
11
12 let nums = [1, 4 , 2]
13 let z = mapAlphaList(add1,nums)
14 println(z)
15 // [2, 5, 3]
16
17 // let z = let z = mapAlphaList(incr,nums)
18 // clash de type : ((Int-> Int) -> Int, [Int])
```

Typage statique ou dynamique

Tendance :

- ▶ vers le typage statique
- ▶ avec du typage dynamique pour les valeurs/programmes venant de l'extérieur (sérialisation, réseau)

Passage facilité par des nouveaux systèmes de types :

- ▶ Typed Scheme : The Design and Implementation of Typed Scheme (POPL 2008)
- ▶ Thorn : intégration of typed and untyped code in a scripting language (POPL2010)
- ▶ TypeScript (Microsoft) : <http://www.typescriptlang.org/>
- ▶ Hack (Facebook) : <http://hacklang.org/>
vidéo : <https://www.youtube.com/watch?v=BnJQJNGkUdM>
- ▶ Dart (Google) : <https://www.dartlang.org/>
- ▶ Rust (Mozilla) : <http://www.rust-lang.org/>
- ▶ Swift (Apple) : <https://developer.apple.com/swift/>