

Analyse des noms

Martin Odersky

version 1.3

Plan du cours

- 1 Symboles et types
- 2 Résolution des noms
 - Les propriétés dépendantes du contexte
 - Représentation des contextes : les portées
 - Implantation des portées
- 3 Performance de l'analyse de noms

Analyse de noms

Dans un programme, chaque nom — de classe, de variable, etc. — identifie un objet du programme.

- Un même nom peut identifier plusieurs éléments différents.
- L'utilisation d'un nom qui n'identifie rien, ou quelque chose d'un type incompatible, doit être rejeté.

```
class A {  
    field a: Int = 12  
    method f(a: Int; b: Int): Null = {  
        printInt(a + this.a)  
        return null  
    }  
}
```

Symboles

Définition : Symbole

Un **symbole** représente uniquement un identificateur déclaré dans le programme et conserve les informations le concernant utiles au compilateur.

Le **nom** et le **type** de l'identifiant sont toujours utiles ;

- suivant les besoins du compilateur, on ajoutera plus d'information.

Exemple : Classes pour les symboles

```
abstract class Symbol {  
  def name: String  
}
```

```
case class ClassSymbol(name: String, superclass: Option[Name])  
  extends Symbol
```

```
case class MethodSymbol(name: String,  
                        paramtypes: List[Type],  
                        restype: Type)  
  extends Symbol
```

```
...
```

Types

Définition : Type

Un **type** est une approximation de la valeur d'une expression ou d'un symbole que le compilateur peut connaître.

Il existe différents types en Vier : types classes, `Int` et `Null`.

Ce qui amène à la syntaxe abstraite suivante pour les types :

```
Type = ClassType name  
      | IntType  
      | NullType
```

Des langages modernes disposent d'information de types bien plus riches — le cours «type systems» aborde cela.

Une classe pour les types

En transformant systématiquement la syntaxe abstraite vers les classes d'arbre on obtient :

Exemple : Classes pour les types

```
abstract class DataType  
  
case class ClassType(clazz: ClassSymbol) extends DataType  
  
case object IntType extends DataType  
  
case object NullType extends DataType
```

Exemple Java : Classes pour les types

```
abstract class DataType {  
  
    static class ClassType extends DataType {  
        private ClassSymbol c;  
        ClassType(ClassSymbol c) {  
            this.c = c;  
        }  
        ClassSymbol getClassSymbol() { return c; }  
    }  
  
    static final DataType IntType = new DataType();  
  
    static final DataType NullType = new DataType();  
}
```


Les langages sont dépendants du contexte

La propriété « *chaque identificateur a besoin d'être déclaré* » dépend du contexte.

En théorie la syntaxe des langages de programmation peut être entièrement spécifiée dans une grammaire dépendante du contexte.

En pratique on définit un sur-ensemble non-contextuel du langage en EBNF, et on élimine les programmes illégaux avec d'autres règles.

Typiquement, ces règles ont besoin d'accéder à la déclaration d'un identificateur.

Chaque identificateur a une **portée**, c'est-à-dire une zone dans le texte du programme à l'intérieur de laquelle on peut s'y référer.

- La portée d'un identificateur s'étend de l'endroit de sa déclaration jusqu'à la fin du bloc englobant
- Il est illégal de se référer à un identificateur en dehors de sa portée.
- Il est illégal de déclarer deux identificateurs avec le même nom si l'un est dans la portée de l'autre.

Portées

Définition : Portée

Une **portée** (*scope*) est une zone de visibilité pour des variables.

La structure de données **Scope** contient les symboles de tous les identifiants déclarés à l'intérieur d'une portée.

La définition exacte d'une portée dépend des règles de visibilité du langage.

Portées de Vier

En Vier, on distingue deux types de portées :

- la portée globale contenant toutes les classes,
- les portées locales, spécifiques à une classe.

Les portées locales ne sont pas imbriquées.

Exercice : portées

Quelles sont les portées de ce programme ?

```
class A {  
  val a = 4  
  def f(b: Int): Int = {  
    val c = {  
      val d = 5  
      return d * 2  
    }  
    val b = 3  
    ...  
  }  
}
```

Nous allons examiner deux solutions pour implanter les portées dans une structure `Scope` :

- 1 en utilisant une structure `Scope` plate ;
- 2 en utilisant une série de structures `Scope` imbriquées.

Portée «plate»

La structure `Scope`, contient une opération pour :

- entrer un symbole dans le `Scope` : c'est la méthode `enter`
- trouver un symbole par son nom : c'est la méthode `lookup`
- supprimer un symbole du `Scope` une fois que l'on quitte la portée : c'est la méthode `remove`

Exercice

Pourquoi a-t-on besoin de la méthode `remove` ?

On peut obtenir tout cela avec les classes «dictionnaires» de la librairie standard.

En **Scala**, on utilise `scala.collection.immutable.ListMap` qui est invariant et conserve l'ordre des éléments. Dans ce cas :

- `enter` devient `scope = scope.update(sym.name, sym)`
- `lookup` devient `scope.get(sym.name)`
- `remove` devient `scope = scope - sym.name`

En **Java**, on utilise `java.util.LinkedHashMap` qui est mutable et conserve l'ordre des éléments. Dans ce cas :

- `enter` devient `scope.put(sym.name, sym)`
- `lookup` devient `(Symbol)scope.get(sym.name)`
- `remove` devient `scope.remove(sym.name)`

Ensembles et tables (non mutables)

Exemple : Map

```
package scala.collection.immutable
class ListMap[K, D]
  extends Map[K, D] ...
trait Map[K, D] {
  def get(key: K): Option[D]
  def update(key:K,value:D):Map[K,D]
  def -(key: K): Map[K, D]
  def keys: Iterator[K]
  ...
}
```

Exemple : Set

```
package scala.collection.immutable
class ListSet[A]
  extends Set[A] ...
class Set[A] {
  def contains(elem: A): Boolean
  def +(elem: A): Set[A]
  def -(elem: A): Set[A]
  def elements: Iterator[A]
  ...
}
```

```
var scope = new ListMap[String, Symbol]()
scope = scope.update(sym.name, sym)
val symOrNull: Option[Symbol] = scope.get(name)
```

- Voir aussi : TreeMap, TreeSet.

Ensembles et tables (mutables)

Exemple : Map

```
package scala.collection.mutable
class HashMap[K, D]
  extends Map[K, D] ...
trait Map[K, D] {
  def get(key: K): Option[D]
  def update(key:K,value:D): Unit
  def -=(key: K): Unit
  def keys: Iterator[K]
  ...
}
```

Exemple : Set

```
package scala.collection.mutable
class HashSet[A]
  extends Set[A] ...
class Set[A] {
  def contains(elem: A): Boolean
  def +=(elem: A): Unit
  def -=(elem: A): Unit
  def elements: Iterator[A]
  ...
}
```

```
val scope = new HashMap[String, Symbol]()
scope.update(sym.name, sym) // ou: scope(sym.name) = sym
val symOrNode: Option[Symbol] = scope.get(name)
```

- Voir aussi : ListMap, TreeMap, ListSet, TreeSet.

Portées «imbriquées»

Pour des langages qui permettent à des portées «masquantes» d'être imbriquées à des profondeurs illimitées, il est plus approprié d'avoir des structures `Scope` imbriquées, une pour chaque bloc :

- On peut supprimer le `Scope` le plus élevé quand on quitte un bloc.
- La méthode `remove` devient obsolète.

On peut à nouveau obtenir tout cela avec les classes «dictionnaires» de la librairie standard.

L'analyse de noms se fait par descente récursive de l'AST,

- la portée courante suit la descente, comme paramètre de chaque méthode d'analyse.

La portée courante fait partie de l'*environnement* d'analyse.

Lorsqu'on analyse un élément de l'AST qui définit une nouvelle portée (méthode, bloc, définition de variable),

- 1 on stocke l'ancienne portée si elle doit être restituée après l'analyse (bloc),
- 2 on analyse le sous-arbre, dans le nouvel environnement,
- 3 si l'on modifie la portée extérieure (définition de variable), on retourne cette nouvelle portée.

Exemple : Analyse de noms

```
...
def analyzeStatement(varScope: VarScope, tree: Stat): VarScope =
  tree match {
    ...
    case While(cond, body) =>
      ...
      analyzeStatement(varScope, body)
      varScope
    ...
  }
...
```

Exercice : Portées mutables

Pourquoi cette technique ne marche-t-elle pas avec les classes (mutables) de la bibliothèque standard de Java ?

Exemple Java : Classe pour les portées

```
class Scope {
    Symbol first = null; Scope outer;
    Scope(Scope outer) { this.outer = outer; }

    /** find symbol with given name in this scope.
     * return null if none exists */
    Symbol lookup(String name) { ... }

    /** enter given symbol in current scope */
    void enter(Symbol symbol) {
        if (first = null) first = symbol; else {
            Symbol last = first;
            while (last.next != null) last = last.next;
            last.next = symbol;
        }
    }
}
```

Comment tout cela marche ensemble

Exemple : programme Vier

```
class A {  
  method length(xs:List):Int = {...}  
  method sort0(xs:List;n:Int):List={  
    return  
      if n < 2 then xs else {...}  
  }  
  method sort(xs: List): List =  
    this.sort0(xs; this.length(xs)  
  ...  
}
```

Définition de List

```
class List {  
  field head: Int = 0  
  field tail: List = null  
}
```


Gestion de la mémoire

Les entrées de la table des symboles pour les variables locales des blocs qui ont fini d'être analysés ne sont plus nécessaires.

Comment s'en débarrasser ?

- En Java/Scala, le ramasse-miettes, ou glaneur de cellules (*garbage collector*), s'en occupe.
- En C/C++ la stratégie la plus efficace est un alloueur de mémoire personnalisé qui utilise le marquage (*mark/release*).
 - En entrant dans un bloc : marquer le sommet du tas courant.
 - En sortant du bloc : réinitialiser le sommet du tas à la marque précédente.

Optimisation

Le schéma courant utilise une recherche linéaire pour les identificateurs.

Dans un compilateur de production c'est beaucoup trop lent.

Meilleures solutions :

- En plus, lier les entrées comme un arbre binaire et utiliser cela pour la recherche.
- Utiliser une table de hachage (*hash table*) pour chaque bloc.
- Utiliser une table de hachage globale (plus rapide).