

Analyse des noms

Martin Odersky

28 novembre 2006
version 1.2

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 {  
  val a = 12  
  def f(a: Int, b: Int) = {  
    val a = b  
    a + this.a  
    ...  
  }  
}
```

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.

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 différentes types en Drexel : types classes et `Int` mais aussi l'absence de type (le type de retour d'une procédure).

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

```
Type = ClassType name  
      | IntType  
      | NoType
```

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 Type;  
  
case class IClassType(clazz: ClassSymbol) extends Type;  
  
case object IIntType extends Type;  
  
case object INoType extends Type;
```

(Les `I` sont ajoutés pour éviter des conflits des noms avec les classes de l'AST.)

Exemple Java : Classes pour les types

```
abstract class Type {  
  
  static class IClassType extends Type {  
    private ClassSymbol c;  
    IClassType(ClassSymbol c) {  
      this.c = c;  
    }  
    ClassSymbol getClassSymbol() { return c; }  
  }  
  
  static final Type IIntType = new Type(){};  
  
  static final Type INoType = new Type(){};  
}
```

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 Drei

En Drei, on distingue deux types de portées :

- la portée globale contenant toutes les classes,
- les portées locales, spécifiques à une classe, à une méthode ou à un block dans le corps d'une méthode.

Les portées locales sont imbriquées,

- les identifiants définis dans une portée intérieure cachent les identifiants de même nom des portées extérieures ;
- dans une portée, les identifiants doivent avoir des noms disjoints.

Exercice : portées

Quels sont les portées de ce programme ?

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

Nous allons montrer 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 (immuables)

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 (muables)

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 symOrNull: Option[Symbol] = scope.get(name);
```

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

Portée «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'*environment* d'analyse.

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

- 1 on stocke l'ancienne portée si elle doit être restituée après l'analyse (block),
- 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 muables

Pourquoi cette technique ne marche-t-elle pas avec les classes (muables) 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 Drei

```
class A {
  def length(lst: List): Int = {...}
  def sort(lst: List): List = {
    var len: Int = this.length(lst);
    var cond: Int = len < 2;
    return if (cond) {
      lst
    } else {
      firstHalf: List = ... Ω ...
    }
  }
}
```

Définition de List

```
class List {
  Int head;
  List tail;
}
```

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).