Analyse des noms

Martin Odersky

5 et 6 décembre 2005
version 1.1

Plan du cours

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.

Règles de contexte pour Zwei

Pour les identificateurs, Zwei adopte les règles de visibilité standards basées sur la structure en blocs.

Nous considérerons un bloc comme étant :

- n'importe quoi entre accolades {} ou
- la zone incluant la liste des paramètres et le corps d'une méthode.

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.

Exercice

Comparez cette dernière propriété avec celle de Java et C.

Représentation des contextes

On représente les contextes par une table des symboles.

Définition : Table des symboles

On appelle table des symboles une structure de données globale qui stocke pour chaque identificateur visible des informations concernant sa déclaration.

On appelle entrée dans la table des symboles (ou entrée pour abréger) l'information associée à un identificateur.

Symboles

Définition : Symbole

Un symbole est une structure de donnée qui contient toutes les informations concernant un identificateur déclaré, et que le compilateur doit connaître.

Les symboles ont un nom et un type et sont regroupés en portées.

Exemple : Classes pour les symboles

```
abstract class Symbol {
  def name: String;
  var next: Symbol;
}
case class ClassSymbol(name: String, superclass: Option[Name])
           extends Symbol;

case class FieldSymbol(name: String, fieldtype: Type)
           extends Symbol;

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

case class VarSymbol(name: String, vartype: Type)
           extends Symbol;
```

Exemple Java : Classes pour les symboles

```
abstract class Symbol {
    String name;
    Symbol next;
}
class ClassSymbol extends Symbol {
    ClassSymbol superclass;
    ...
}
class FieldSymbol extends Symbol {
    Type fieldtype;
}
class MethodSymbol extends Symbol {
    Type[] paramtypes;
    Type restype;
}
class VarSymbol extends Symbol {
    Type vartype;
}
```

Types

Définition : Type

Un type est une structure de données qui contient toutes les informations concernant la valeur d'une expression ou d'un symbole (excepté son nom) que le compilateur doit connaître.

Il existe différentes types en Zwei : types classes, `Int` et `Null`.

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

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

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 INullType 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 INullType = new Type(){};
}
```

Portées

Définition : Portée

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

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

La définition exacte d'une portée dépend des règles de visibilité du langage. En général, on définit une structure `Scope` par portée où :

- les `Scope` sont imbriqués ;
- les identifiants définis dans un `Scope` cachent les identifiants englobants de même nom ;
- les identifiants des variables dans un `Scope` doivent avoir des noms disjoints (pas si des méthodes récursives sont permises).

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

1. Pour un langage comme Zwei en utilisant des classes de collection «dictionnaire» (`HashMap` ou `TreeMap`) ;
2. Pour un langage comme Scala ou Java en implantant le `Scope` à la main.

Portées pour Zwei

En Zwei, la profondeur des portées est de trois :

1. Tout à l'extérieur, il y a la portée de toutes les classes.
2. Chaque classe à deux portées pour ses membres :
   1. une pour les champs (*fields*) ;
   2. une pour les méthodes (*methods*).
3. Chaque méthode a une portée contenant ses paramètres et les valeurs définies localement.

Il suffit ainsi d'utiliser une pile de trois structures `Scope` : `classScope`, `memberScope` et `varScope`.

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

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

Portées pour Scala ou Java

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.

Nous allons présenter une implantation bas-niveau de ce concept en 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;
        }
    }
}
```

Il est parfois nécessaire de parcourir tous les symboles d'une portée dans l'ordre de leurs déclarations :

- les portées se réfèrent au premier symbole déclaré dans la portée ;
- on accède aux autres symboles par le champ `next` de la classe `Symbol`

Exercice

Implantez la méthode `lookup`.

Comment tout cela marche ensemble

Exemple : programme Zwei

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

Spécification des règles de contexte

Comment les tables de symboles sont-elles utilisées dans un compilateur ?
Premier chose à se demander : Comment spécifie-t-on l'utilisation des tables des symboles dans les règles de contexte d'un langage ?

Plus généralement : Comment spécifie-t-on les règles de contexte ?

On peut utiliser une méthode semi-formelle, qui ajoute des attributs aux symboles et connecte les attributs au moyen de contraintes.

Squelette de spécification des règles de visibilité

Exemple : règles de visibilité pour Zwei

P = Program D E(t_e)
"crée une nouvelle portée pour les classes."

D = ClassDef name [name] M
"crée un symbole dans la portée des classes pour `name`, avec la sorte `ClassSymbol` et le type `ClassType` ; traite les membres `{ M }` dans la portée de la classe"

M = MethodDef name { name T(t_f) } T(t_r) E(t_e)
"traite les paramètres `{ name T }` dans une portée nouvelle contenant un seul symbole pour `this` et le type `ClassType` référant la classe courante ; crée un symbole dans la portée de la classe courante avec le nom `name`, les types { t_f } et le type de résultat t_r."

S = Var name T(t_f) E(t_e)
"crée un nouveau symbole de la sorte `VarSymbol` dans la portée courante avec le nom `name` et le type t_f donné."

...

Grammaires attribuées

- Une syntaxe dépendant du contexte est parfois spécifiée en utilisant une grammaire attribuée.
- C'est similaire à ce que l'on a fait, mais complètement formel.
- Les grammaires attribuées reposent sur la syntaxe non-contextuelle concrète :
  - On donne des attributs aux symboles, pouvant avoir un type quelconque.
  - Les attributs sont évalués par des affectations similaires à nos contraintes.
  - On représente les attributs comme des variables d'instance des noeuds de l'arbre.

Systèmes de types

L'idée ici est d'exprimer une syntaxe dépendant du contexte comme un système déductif.

- Les jugements sont de la forme $\Gamma \vdash < \mathit{terme} > : < \mathit{type} >$
- Un programme P est bien typé si un jugement $\Gamma \vdash P : T$ est prouvable.

Exemple : Une règle de typage pour l'addition

$$(\text{Add}) \frac{\Gamma \vdash e_1 : \mathit{Int} \qquad \Gamma \vdash e_2 : \mathit{Int}}{\Gamma \vdash e_1 + e_2 : \mathit{Int}}$$

- Généralement on garde aussi dans un jugement un environnement Γ qui représente la table des symboles courante.
- Les systèmes de types sont souvent plus concis et lisibles que les grammaires attribuées.
- Les grammaires attribuées sont plus proches d'une implémentation.

Structure de données efficace pour la table des symboles

- Avec la solution précédente, trouver un symbole requiert $O(N)$ étapes, où N est le nombre de symboles visibles au point où le symbole est consulté.
- Si N croît linéairement avec la longueur du programme, le temps total nécessaire pour accéder aux symboles croît de façon quadratique.
- Cela peut être amélioré en maintenant des structures de données qui accélèrent la consultation des symboles.
- Deux solutions classiques :
  - les arbres de recherche
  - les tables de hachage

- Un arbre de recherche binaire peut être utilisé si les clés sont totalement ordonnées.
- Un noeud dans l'arbre de recherche contient une entrée de la table ainsi que des pointeurs `left` et `right` pour les sous-arbres.
- Invariant pour chaque noeud n :
  - toutes les entrées rangées dans le sous-arbre gauche sont plus petites que l'entrée rangée dans n, et
  - toutes les entrées rangées dans le sous-arbre droit sont plus grandes que l'entrée rangée dans n.

Donc, la consultation nécessite de rechercher dans au plus un sous-arbre.

Une classe Scope avec arbre de recherche

Exemple : arbre de recherche en Java

```
class TreeScope extends Scope {
    static class SymTree {
        SymTree left = null, right = null;
        Symbol sym;
        SymTree (Symbol sym) { this.sym = sym; }
    }
    SymTree root = null;
    SymTree insert(SymTree t, Symbol sym) {
        if (t == null)
            return new SymTree(sym);
        else {
            int rel = sym.name.compareTo(t.sym.name);
            if (rel < 0)
                t.left = insert(t.left, sym);
            else if (rel > 0)
                t.right = insert(t.right, sym); return t;
        }
    } ...
```

```
  Symbol find(SymTree t, String name) {  ... }

  Symbol lookup(String name) {
    return find(root, name);
  }

  Symbol enter(Symbol sym) {
    root = insert(root, sym);
  }
}
```

Suppressions

- Les suppressions dans un arbre de recherche binaire sont un peu plus difficiles.
- Soit n le noeud à supprimer. Par quoi doit être remplacé n ?

Cas facile un sous-arbre de n est nul. Prendre l'autre.

Cas difficile aucun sous-arbre n'est nul. Il faut trouver une nouvelle racine.

Solution : Prendre le plus petit noeud du sous-arbre droit (on pourrait prendre aussi le plus grand noeud du sous-arbre gauche).

Exemple : La méthode delete

```
SymTree delete(SymTree t, Symbol sym) {
    if (t == null)
        return t;
    else {
        int rel = sym.name.compareTo(t.sym.name);
        if (rel < 0)
            t.left = delete(t.left, sym);
        else if (rel > 0)
            t.right = delete(t.right, sym);
        else {
            if (t.right == null)
                t = t.left;
            else if (t.left == null)
                t = t.right;
            // ...
```

```
            else {
                SymTree prev = null;
                SymTree newt = t.right;
                while (newt.left!= null) {
                    prev = newt; newt = newt.left;
                }
                if (prev == null)
                    t.right = newt.right;
                else
                    prev.left = newt.right;
                newt.left = t.left;
                newt.right = t.right;
                t = newt;
            }
        }
        return t;
    }
}
```

Complexité des opérations sur l'arbre

Coût

	en moyenne	au pire
rechercher	$O(logN)$	$O(N)$
insérer	$O(logN)$	$O(N)$
supprimer	$O(logN)$	$O(N)$

On obtient un meilleur comportement dans le pire des cas en utilisant un arbre équilibré (*balanced tree*) : arbre AVL, arbre rouge-noir, ..

Pour des insertions aléatoires, le nombre de comparaisons est environ 40% plus important pour un arbre non équilibré que le $log(N)$ pour un arbre parfaitement équilibré.

Solutions à base de tables de hachage

- Une table de hachage (*hash table*) est similaire à un tableau indexé par les entrées de la table (temps d'accès constant).
- Il faut tout d'abord associer des entiers aux entrées de la table. C'est le rôle de la méthode `int hashCode()` dans la classe `Object`.
- Il reste deux problèmes :
  - intervalle d'indices trop grand (typiquement 28 à 32 bits).
    Solution : utiliser le code de hachage modulo la taille du tableau désiré.
  - des collisions sont possibles.
    Solution : les éléments du tableau sont des listes linéaires d'entrées avec le même code de hachage.

Exemple : Code avec table de hachage en Java

```
class HashScope extends Scope {
    static class Entry {
        Entry next; Symbol sym;

        Entry (Entry next, Symbol sym) {
            this.next = next; this.sym = sym;
        }
    }

    final int N = 1024;
    Entry [] elems = new Entry [N];

    Symbol enter(Symbol sym) {
        int i = sym.name.hashCode() % N;
        elems[i] = new Entry (elems[i], sym);
    }

    Symbol lookup(String name) { ... }
```

```
    void deleteAll(Symbol sym) {
        if (sym != null) {
            deleteAll(sym.next);
            int i = sym.name.hashCode() % N;
            elems[i] = elems[i].next;
        }
    }

    void leave() { deleteAll(first); }
}
```

Remarques :

- On utilise une seule table pour toutes les portées.
- L'initialisation et la consultation des portées sont ainsi plus efficaces.
- Par contre, on doit supprimer les entrées de la table en sortant d'une portée.