Analyse de Données Structurées - Cours 9

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

2 avril 2015

Analyse Sémantique

- Nous savons maintenant construire un arbre de syntaxe abstraite à partir d'une représentation textuelle (syntaxe concrète).
- Par exemple : syntaxe abstraite pour un langage de programmation, pour un langage de données structurées comme XML ou JSON.
- Nous avons aussi vu la semaine dernière que certaines restrictions sytaxiques ne peuvent pas être réalisées par une une grammaire.
- C'est maintenant le rôle de l'analyse sémantique.
- Les techniques sont les mêmes que pour l'interprétation.

Théorie et Pratique

Tâche	Model théorique	Réalisation
Analyse lexicale	Expressions régulières	Fichier JFlex
Analyse syntaxique	Grammaires	Implémentation d'un parseur LL(1)
Syntaxe abstraite	Définition inductive	Classes Java
Sémantique	Règles sémantiques	Parcours d'arbre

Définition Inductive

L'ensemble E des arbres de syntaxe abstraite représentant des expressions régulières est défini comme suit :

- tout $Int(n)$, où $n \in \mathbb{N}$, est un élément de E
- si $e_1, e_2 \in E$, alors $Sum(e_1, e_2) \in E$
- si $e_1, e_2 \in E$, alors $Product(e_1, e_2) \in E$

Ce sont les seules façons de construire un élément de E.

Exemples

- $Int(42)$
- $Product(Sum(Int(2), Int(49)), Int(5))$

Remarques

- Attention : il s'agit d'une notation mathématique pour des *arbres*. Ce n'est pas une définition d'une syntaxe concrète.
- On utilise parfois aussi pour la définition des arbres de syntaxe abstraite un formalisme appelé *grammaire d'arbres*, ce qui risque créer une confusion. Pour nous, une grammaire défini toujours une syntaxe concrète.
- L'intérêt de cette notation est qu'elle nous permet de rédiger les règles de calcul sur la syntaxe abstraite. Cette notation est plus compacte que de dessiner un arbre.
- Souvent on défini plusieurs types d'arbre de syntaxe abstraite , par exemple un type pour les expressions arithmétiques, un autre pour les instructions d'un langage de programmation.

Traduction en JAVA

- Chaque type d'arbre correspond à une classe abstraite.
- Pour chacun des cas dans la définition il y a une classe dérivée.
- Le constructeur de cette classe dérivée a comme arguments les valeurs différentes utilisées dans la construction de l'arbre.
- Le constructeur stocke ces valeurs dans des champs privés de la classe.
- Éventuellement : des méthodes pour savoir dans quel cas on est, et pour récupérer les valeurs utilisées dans la construction.

Définition de la Syntaxe Abstraite sur l'exemple I

```
abstract class Expression {
    public boolean isInt() {
        return false;
    }
    public boolean IsSum() {
        return false;
    }
    public boolean IsProduct() {
        return false;
    }
}

class Int extends Expression {
    private int value;
```


Définition de la Syntaxe Abstraite sur l'exemple II

```
    public boolean isInt() {
        return true;
    }
    public Int (int i) {
        value=i;
    }
    public int getValue() {
        return value;
    }
}

class Sum extends Expression {
    private Expression left;
    private Expression right;
    public boolean isSum() {
```

Définition de la Syntaxe Abstraite sur l'exemple III

```
        return true;
    }
    public Sum(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public Expression getLeft() {
        return left;
    }
    public Expression getRight() {
        return right;
    }
}

class Product extends Expression {
```


Définition de la Syntaxe Abstraite sur l'exemple IV

```
    private Expression left;
    private Expression right;
    public boolean isProduct() {
        return true;
    }
    public Product(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public Expression getLeft() {
        return left;
    }
    public Expression getRight() {
        return right;
    }
```


Définition de la Syntaxe Abstraite sur l'exemple V

```
}
```


La même chose dans le langage OCaml

```
type expression =
  | Integer of int
  | Sum of expression*expression
  | Product of expression*expression;;
```

(voir le cours *Programmation Fonctionnelle* du L3)

Objectif

- Jusqu'à maintenant : syntaxe concrète et syntaxe abstraite
- Reste à faire :
  - Les restrictions supplémentaires qu'on ne peut pas exprimer par une grammaire : *analyse sémantique*
  - Exécution ou compilation d'un programme : *sémantique*
- Il nous faut d'abord une notation qui nous permet de spécifier précisément les restrictions supplémentaire, et la sémantique.
- Puis : codage (en Java, par exemple)

Langages de programmation : Trois étapes

1. De la syntaxe concrète à la syntaxe abstraite.
2. Analyse sémantique : par exemple assurer que toutes les variables sont déclarées, toutes les méthodes définies, que le typage est correct.
3. Exécution de la syntaxe abstraite (interprétation), ou traduction vers un autre langage (compilation, ou transformation de code)

L'enchaînement des trois étapes

1. Normalement : Construction de la syntaxe abstraite pour le programme entier avant toute analyse sémantique ou exécution (attention : il y des cas ou analyse syntaxique et exécution sont intercalées)
2. La vérification des conditions sémantiques est faite *avant* l'exécution. Pour cette raison on parle aussi parfois d'une sémantique *statique*.
3. L'exécution (ou la compilation) peut utiliser des résultats de l'étape (2). Dans ce cas, on l'étape (3) se base sur une syntaxe abstraite *annotée* (exemple : inférence de types).

Notion de jugement

- Un jugement est une affirmation portant sur une expression abstraite.
- Quelques exemples, en langue naturelle :
  - « l'expression $Sum(Int(1), Product(Int(2), Int(3)))$ a pour valeur 7 » ;
  - « l'expression $Sum(Int(1), Product(Int(2), Int(3)))$ a pour type int » ;
  - « l'instruction $\mathrm{Write}(Sum(Int(3), Int(4)))$ affiche 7 » ;
  - « l'expression $Sum(Int(3), Int(5))$ a pour code machine [PUSHI 3 ;PUSHI 5 ;ADD] ».

Jugements avec contexte

- « dans le contexte où x est une variable valant 4, l'expression $Sum(Int(1), Product(Var(x), Int(3)))$ a pour valeur 13 » ;
- « dans le contexte où x est une variable de type real, l'expression $Sum(Int(1), Product(Var(x), Int(3)))$ a pour type real » ;
- pour un compilateur devant engendrer du code machine à une adresse donnée, on pourra avoir des jugements du style « dans le contexte où l'adresse courante du code est 250 et x est une variable dont l'adresse est 100, l'expression $Product(Int(3), Var(x))$ a pour code machine [250 PUSHI 3 ; 251 PUSH 100 ; 252 MUL] ».

Définition

Un *jugement* est un triplet $(C, a, v) \in D^c \times A \times D^v$ tel que :

- A est un ensemble d'arbres de syntaxe abstraite, contenant l'arbre a ;
- D^v est un ensemble appelé *domaine de valeurs* ; la valeur v appartient au sous-ensemble D^v des valeurs ;
- D^c est un ensemble appelé *domaine de contextes* ; la valeur C appartient au sous-ensemble D^c des valeurs.

Remarques

- Quand le domaine de contextes est trivial, le jugement se réduit à une paire (a, v).
- Les contextes et les valeurs peuvent être des paires, triplets, ...

Notation et exemples

- On note le jugement $(C, a, v) \in D^c \times A \times D^v$ par « $C \vdash a \Rightarrow v$ ».
- Dans le cas d'un domaine de contexte trivial : $a \Rightarrow v$.
- Exemples :
  - $$Sum(Int(1), Product(Int(2), Int(3))) \Rightarrow 7$$
    est le jugement « l'expression $1 + 2 \times 3$ a pour valeur 7 ».
  - $$\{x \Rightarrow 4\} \vdash Sum(Int(1), Product(Var(x), Int(3))) \Rightarrow 13$$
    est le jugement « dans le contexte où x est une variable valant 4, l'expression $1 + x \times 3$ a pour valeur 13 », ici $D^v = \mathbb{N}$ et D^c est l'ensemble des *environnements d'évaluation*.

Variante de notation

Lorsque le domaine de valeurs D^v est un ensemble de types, par exemple $D^v = \{\text{int}, \text{bool}, \text{real}\}$, nous utiliserons une variante classique de cette notation, $C \vdash a : t$, où $t \in D^v$. On écrira donc :

- $$Sum(Int(1), Product(Int(2), Int(3))) : \text{int}$$
  pour le jugement « l'expression $1 + 2 \times 3$ a pour type int » ;
- $$\{x : \text{real}\} \vdash Sum(Int(1), Produit(Var(x), Int(3))) : \text{real}$$
  pour le jugement « dans le contexte où x est une variable de type real, l'expression $1 + x \times 3$ a pour type real ».

Jugements vrais et faux

La définition précédente autorise l'écriture de jugements vrais aussi bien que de jugements faux :

- « l'expression $Sum(Int(1), Mult(Int(2), Int(3)))$ a pour valeur 12 » ;
- « l'expression $Sum(Int(1), Mult(Int(2), Int(3)))$ a pour type bool » ;

Axiomes

- Les règles sémantiques permettent d'établir qu'un certain jugement est vrai.
- Cas de base : Les axiomes : ce sont des jugements pour lesquels on affirme qu'ils sont vrais.
- Exemples (voir plus tard pour la notation) :
  - si n est une constante entière, alors le jugement $Int(n) \Rightarrow n$ est vrai ;
  - si n est une constante entière, alors le jugement $Int(n) : int$ est vrai ;
  - dans un contexte C où x est une variable de type bool, le jugement $C \vdash Var(x)$: bool est vrai.

Les règles en général

- Les règles proprement dites correspondent au cas général de la récurrence.
- Elles affirment la vérité d'un jugement qui constitue la *conclusion* de la règle sous l'hypothèse (de récurrence) de la vérité d'un ou plusieurs jugements appartenant aux *prémisses* de la règle.
- Par exemple :
  Si le jugement $e_1 : int$ est vrai et si le jugement $e_2 : int$ est vrai alors le jugement $Sum(e_1, e_2) : int$ est également vrai.

Règles sémantiques

Définition

Une *règle sémantique* est un k-uplet de jugements, avec $k \geq 1$. Si $k = 1$, la règle est appelée un *axiome*. Dans le cas d'une règle $(J_1, \ldots, J_k)$ avec $k \geq 2$, $J_1, \ldots, J_{k-1}$ constituent les *prémisses* de la règle et J_k en est la *conclusion*.

Notation habituelle

Pour la règle $(J_1, \ldots, J_k)$

$$\frac{J_1, \ldots, J_{k-1}}{J_k}$$

Dans le cas d'un axiome, rien ne figure au dessus du trait de fraction.

Les schémas de règles

- Dans le cas de l'évaluation d'une expression arithmétique il y a déjà une infinité d'axiomes :

$$\frac{}{Int(0) \Rightarrow 0} \qquad \frac{}{Int(1) \Rightarrow 1} \qquad \text{etc.}$$

- Pour cette raison, on utilise des *schéma d'axiomes*

$$\text{pour tout } n \in \mathbb{N} : \quad \frac{}{Int(n) \Rightarrow n}$$

- De la même façon, on a les schémas de règles suivantes pour l'addition et la multiplication d'entiers :

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{Sum(e_1, e_2) \Rightarrow v_1 +_{int} v_2}$$

et

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{Product(e_1, e_2) \Rightarrow v_1 \times_{int} v_2}$$

Les schémas de règles

- Dans un schéma de règles comme

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{Sum(e_1, e_2) \Rightarrow v_1 +_{int} v_2}$$

  e_1 et e_2 sont des *méta-variables* qu'il faut remplacer par des arbres de syntaxe abstraite pour obtenir une instance.
- Souvent on confond les notions de règle et de schéma de règle.
- Les opérateurs $+_{int}$ et $\times_{int}$ sont les opérations arithmétiques.

Des règles sémantique à des méthodes récursives

- Dans des cas simples : les jugements spécifient une fonction, par exemple la fonction d'évaluation qui associe à chaque arbre de syntaxe abstraite une valeur *unique*.
- Cette fonction peut être partielle.
- Dans notre cas de l'évaluation : trois schémas de règles, chacune correspondant à un cas dans la définition des arbres de syntaxe abstraite.
- Il y a une méthode abstraite `eval`.
- Chaque sous-classe défini cette méthode selon le schéma donnée :
  - L'argument de la méthode est le contexte.
  - Les prémisses correspondent à des appels récursifs.
  - La conclusion dit alors comment calculer le résultat.

Évaluation : code I

```
abstract class Expression {
    abstract int eval();
}

class Int extends Expression {
    private int value;
    public Int (int i) {
        value=i;
    }
    public int eval() {
        return value;
    }
}
```

Évaluation : code II

```
class Sum extends Expression {
    private Expression left;
    private Expression right;
    public Sum(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public int eval () {
        return left.eval() + right.eval();
    }
}

class Product extends Expression {
    private Expression left;
    private Expression right;
```


Évaluation : code III

```
    public Product(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public int eval () {
        return left.eval() * right.eval();
    }
}
```