Analyse de Données Structurées - Cours 10

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

8 avril 2015

Rappel : évaluation d'expressions sans variables

- 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 $Product(e_1, e_2) \in E$
- Règles sémantiques :

$$\frac{}{Int(n) \Rightarrow n} \qquad n \in \mathbb{N}$$

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

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

Expressions avec variables

- 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'
  - tout $Ident(s)$, où $s \in \Sigma^*$, 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 $Product(e_1, e_2) \in E'$
- Σ est l'alphabet utilisé pour former des noms de variables (par exemple, tous les caractères UNICODE).
- Éventuellement, l'ensemble des noms de variables possibles dans la syntaxe abstraite est plus riche que les noms de variables permis par la syntaxe concrète.

Évaluation d'expressions avec variables

- Maintenant, l'évaluation d'une expression dépend aussi d'un *contexte*.
- La définition du contexte dépend de l'application. Pour l'évaluation : Il s'agit d'une fonction partielle

$$\Sigma^* \to \mathbb{N}$$

aussi appelée *environnement de valeurs*.
- Le jugement d'évaluation :

$$\Gamma \vdash e \Rightarrow v$$

exprime : Dans l'environnement de valeur Γ, la valeur de l'expression e est v.
- Exemple d'un jugement vrai :

$$\{x = 3, y = 5\} \vdash Sum(Product(Ident(x), Int(2)), Ident(y)) \Rightarrow 11$$

Règles pour l'évaluation des expressions avec variables

- Axiomes :

$$\frac{}{\Gamma \vdash \mathit{Int}(n) \Rightarrow n} \quad n \in \mathbb{N}$$

$$\frac{}{\Gamma \vdash \mathit{Ident}(s) \Rightarrow n} \quad \Gamma(s) = n$$

- Règles récursives :

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

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

Remarques

- Quand $s \notin \mathrm{domaine}(\Gamma)$, alors le schéma d'axiome ne s'applique pas à $\mathit{Ident}(s)$. Dans ce cas on ne peut donc pas calculer la valeur d'une expression qui contient $\mathit{Ident}(s)$.
- Dans les règles récursives, comme

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

le contexte Γ utilisé dans les hypothèses est le même que dans la conclusion. Le calcul correspondant aux deux hypothèses peut donc se faire en parallèle.

Implémentation : `ValueEnvironment.java`

- Implémentation des environnements de valeur : table de hachage.

```
import java.util.HashMap;

class ValueEnvironment extends HashMap<String,Integer> {
}
```


Implémentation : `Expression.java` I

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

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

Implémentation : Expression.java II

```
class Ident extends Expression {
    private String name;
    public Ident (String s) {
        name=s;
    }
    public int eval(ValueEnvironment env) {
        return env.get(name);
    }
}

class Sum extends Expression {
    private Expression left;
    private Expression right;
    public Sum(Expression e1, Expression e2) {
        left=e1;
```


Implémentation : Expression.java III

```
        right=e2;
    }
    public int eval (ValueEnvironment env) {
        return left.eval(env) + right.eval(env);
    }
}

class Product extends Expression {
    private Expression left;
    private Expression right;
    public Product(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public int eval (ValueEnvironment env) {
```


Implémentation : Expression.java IV

```
        return left.eval(env) * right.eval(env);
    }
}
```


Affectation et Affichage

- Objectif : exécution de programmes.
- Première étape : séquences d'affectation à des variables, et d'instructions d'affichage.
- Instruction "print" pour afficher, symbole " :=" pour l'affectation.
- Instructions terminées par le symbole ";"
- Fragment de grammaire (le reste est comme avant) :

$$\begin{aligned} S &\rightarrow \text{IL EOF} \\ \text{IL} &\rightarrow \epsilon \mid \text{I ; IL} \\ \text{I} &\rightarrow \texttt{ident} := \text{E} \mid \texttt{print E} \end{aligned}$$

Règles sémantiques

- On s'intéresse surtout à la modification des environnements.
- Jugement $\Gamma_1 \vdash i \Rightarrow \Gamma_2$: l'exécution de l'instruction (ou de la liste d'instructions) i dans l'environnement Γ_1 résulte dans l'environnement Γ_2.
- Modélisation des affichages : omise.

Instructions

- Syntaxe abstraite : L'ensemble I des arbres de syntaxe abstraite de sorte *Instruction* est défini comme suit :
  - tout $Assign(s, e)$, où $s \in \Sigma^*$ et $e \in E'$, est un élément de I ;
  - tout $Print(e)$, où $e \in E'$, est un élément de I.
- Règles sémantiques :

$$\frac{\Gamma_1 \vdash e \Rightarrow v_1}{\Gamma_1 \vdash Affect(s, e) \Rightarrow \Gamma_2[s \mapsto v]}$$

$$\frac{}{\Gamma_1 \vdash Print(s, e) \Rightarrow \Gamma_1}$$

Remarques

- Maintenant, le contexte dans les hypothèses n'est plus toujours le même que dans la conclusion.
- Les hypothèses des règles peuvent être des jugements qui portent sur des expressions de syntaxe abstraite de sorte différente que la conclusion (exemple : règle pour *Affect*).
- $Gamma[s \mapsto v]$ est l'environnement qui associe à s la valeur v, et qui associe $\Gamma(x)$ à tout $x \neq s$.
- La deuxième règle exprime que l'exécution de *Print* ne modifie pas l'environnement.

Listes d'instructions

- Syntaxe abstraite : L'ensemble IL des arbres de syntaxe abstraite de sorte *InstructionList* est défini comme suit :
  - Nil est un élément de IL ;
  - Si $il \in IL$ et $i \in I$, alors $Seq(i, il)$ est un élément de IL.
- Règles sémantiques :

$$\frac{}{\Gamma_1 \vdash Nil \Rightarrow \Gamma_1}$$

$$\frac{\Gamma_1 \vdash i \Rightarrow \Gamma_2 \qquad \Gamma_2 \vdash il \Rightarrow \Gamma_3}{\Gamma_1 \vdash Seq(i, il) \Rightarrow \Gamma_3}$$

Remarques

- La première règle (qui est un axiome) exprime que l'exécution de *Nil* ne modifie pas l'environnement.
- La deuxième règle impose maintenant un ordre sur l'exécution des deux composantes i et il :
  - Donné initialement : Γ_1 et $Seq(i, il)$.
  - Il faut d'abord exécuter la première hypothèse qui donne, à partir de Γ_1 et de i, l'environnement Γ_2.
  - Puis on peut exécuter la deuxième hypothèse qui donne, à partir de Γ_2 et de il, l'environnement Γ_3.
  - L'implémentation utilise des tables de hachage modifiables, il faut bien respecter l'ordre !

Implémentation : `Instruction.java` I

```
abstract class Instruction {
    abstract void exec(ValueEnvironment env);
}

class Assignment extends Instruction {
    private Expression expression;
    private String variable;
    public Assignment(String v, Expression e) {
        expression=e;
        variable=v;
    }
    public void exec(ValueEnvironment env) {
        env.put(variable,expression.eval(env));
    }
```


Implémentation : `Instruction.java` II

```
}

class Print extends Instruction {
    private Expression expression;
    public Print(Expression e) {
        expression=e;
    }
    public void exec(ValueEnvironment env) {
        System.out.println(expression.eval(env));
    }
}
```


Implémentation : `InstructionList.java` I

```
abstract class InstructionList {
    abstract void exec(ValueEnvironment env);
}

class Nil extends InstructionList {
    public Nil() {
    }
    public void exec(ValueEnvironment env) {
    }
}

class Seq extends InstructionList {
    private Instruction head;
    private InstructionList rest;
```

Implémentation : `InstructionList.java` II

```
    public Seq(Instruction i, InstructionList il) {
        head=i;
        rest=il;
    }
    public void exec(ValueEnvironment env) {
        head.exec(env);
        rest.exec(env);
    }
}
```


Expressions et Types

- Pour l'exécution des programmes avec, par exemple des instruction conditionnelles et des boucles, nous avons besoin des expressions booléennes.
- On ne peut pas faire la distinction entre expressions entières et expressions booléennes par la grammaire à cause des variables.

Typage statique et typage dynamique

- Typage statique : vérification des types après construction de la syntaxe abstraite, et *avant* l'exécution.
- Avantage : Plus de sécurité, une fois le typage vérifié par le compilateur on sait que des erreurs de *type* ne peut plus se produire (sauf certains langages de programmations qui permettent de contourner le typage).
- Avantage : le typage peut annoter la syntaxe abstraite par des types des expressions, ce qui simplifie l'exécution. Désambiguïson des opérateur surchargés.
- Exemples : Java, Ocaml, ...

Typage statique et typage dynamique

- Typage dynamique : vérification des types pendant *exécution* du programme, quand on applique un opérateur à des valeurs.
- Avantage : plus de flexibilité (mais : flexibilité aussi possible dans le cas statique, par ex. polymorphie).
- Avantage : programmes moins verbeux, car il n'est pas nécessaire de déclarer les variables avec leur type (mais : le même avantage peut être obtenu par une *inférence de type*).
- Exemples : Python, bash, Perl.

Exemple (Python)

```
#!/usr/bin/python
# une variable peut prendre des valeurs de
# type different.

x=42
print x
x=True
print x
```


Exemple (Python)

```
#!/usr/bin/python

# erreur de typage seulement detectee quand
# la branche else est executee

import sys
read=sys.stdin.readline()
if read != 'coocoo\n':
    print 17+21
else:
    print ('abc' & 'def')
```


Typage statique et typage dynamique

- Dans le cas d'un typage statique, des types sont initialement par des déclarations associés à des *identificateurs*. Puis, une inférence de type peut déduire le type pour chaque expression, et vérifier que tous les opérateurs sont utilisés en cohérence avec le type des expressions.
- Dans le cas d'un typage dynamique, les types sont associés aux *valeurs* (éventuellement résultat d'un calcul). C'est au moment de l'application de l'opérateur qu'une incompatibilité de types peut être détectée.

Règles de typage

- Environnement de typage : asscocie des types à des variables
- Jugement $\Gamma \vdash e : t$: Dans l'environnement Γ, l'expression e a le type t.

$$\frac{}{\Gamma \vdash Int(n) : int} \quad n \in \mathbb{N}$$

$$\frac{}{\Gamma \vdash True : bool} \qquad \frac{}{\Gamma \vdash False : bool}$$

$$\frac{}{\Gamma \vdash Ident(s) : \Gamma(s)} \quad s \in \Sigma^*$$

Règles de typage

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Plus(e_1, e_2) : int}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Mult(e_1, e_2) : int}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Equal(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash Greater(e_1, e_2) : bool}$$

Analyse de Données Structurées - Cours 10
└Typage

Règles de typage

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash And(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash Or(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash Not(e) : bool}$$