Programmation Dirigée par la Syntaxe (PDS)

CM2 - Calcul dirigé par la syntaxe

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

PDS, M1 info

Plan

Plan

Introduction

- Une ASD (définition de syntaxe abstraite) définit :
  - un langage
  - un ensemble de phrases
  - les structures syntaxiques de ces phrases
- On veut
  - construire, manipuler ces structures syntaxiques
  - faire du *calcul dirigé par la syntaxe*
  - définir des fonctions
    - dont les arguments sont des AST
    - dont le type d'entrée est l'ASD
  - exemples de fonction : pretty-printing, compilation, vérification de types, etc.

Calcul dirigé par la syntaxe

Propriétés souhaitées pour les calculs :

1. localité
   - calculs locaux à chaque variant
   - $\Rightarrow$ facilite extensions ASD (ajouts de cas)
   - $\Rightarrow$ facilite les preuves de correction
   - *forme de raisonnement/programmation par cas*
2. compositionalité
   - propagation des calculs à travers l'arbre (AST)
     - synthèse : agrégation des feuilles vers la racine
     - héritage : propagation de la racine vers les feuilles
     - ...
   - localité des règles de propagation
   - le calcul global est distribué entre les noeuds de l'AST

Analogie OO

Analogie avec les méthodes en programmation OO

- chaque méthode définit un calcul local à une classe
- chaque méthode peut
  - transmettre des informations aux objets voisins
  - déclencher des calculs chez des objets voisins
    appels de méthodes
  - agréger des informations reçues d'objets voisins
- le calcul global est distribué entre les objets

ASD attribuée

- une puissante structure de calcul
- calquée sur les grammaires attribuées
  - ASD au lieu de grammaires
  - donc variants au lieu de productions
  - et propagation sur AST au lieu de *parse trees*

Plan

ASD attribuée

Definition

Une ASD attribuée est une ASD où :

- chaque type τ est décoré par des attributs $\tau.a, \tau.b, \ldots$
- chaque variant est décoré par des calculs sur ces attributs
  - définissant certains attributs en fonction des autres

Attributs

Un attribut est caractérisé par :

1. un nom (ex. "val")
2. le type de ses valeurs (ex. int)
3. les types τ_i de l'ASD qu'il décore (ex. *expr*)
4. sa polarité (ex. synthétisé)
   - hérité : entrée pour le calcul local
   - synthétisé : sortie pour le calcul local

Attributs *self*

Un attribut *self* est prédéfini pour chaque type de base et chaque type τ d'un ASD

- nom : *self*
- type des valeurs : τ (les valeurs sont des valeurs de base ou des AST)
- polarité : synthétisé

Remarque

Ces attributs servent en général à récupérer les valeurs aux feuilles des AST.

Calculs attachés aux variants

Chaque *calcul* est :

- attaché à un variant $\tau_0 ::= \ldots \mid C(\tau_1, \ldots, \tau_n) \mid \ldots$
- de la forme $\tau_0.a_o := f(\tau_1.a_1, \ldots, \tau_n.a_n)$
  $\tau_0.a_o$ est défini en fonction des attributs $\tau_1.a_1, \ldots, \tau_n.a_n$

Contraintes sur la place des attributs dans les calculs

$\tau_i.a$	$\tau_i = \tau_0$ (type défini)	$\tau_i \in \tau_{1..n}$ (type argument)
a synthétisé	$\tau_0.a := \ldots$	$\ldots := f(\ldots\tau_i.a\ldots)$
a hérité	$\ldots := f(\ldots\tau_0.a\ldots)$	$\tau_i.a := \ldots$

Rmq : même contraintes que pour les appels de fonctions en cascade. Une fonction peut :

- définir sa valeur de retour et les paramètres des fonctions qu'elle appelle
- utiliser ses paramètres et les valeurs de retour des fonctions qu'elle appelle

Calculs attachés aux variants

Chaque *calcul* est :

- attaché à un variant $\tau_0 ::= \ldots \mid C(\tau_1, \ldots, \tau_n) \mid \ldots$
- de la forme $\tau_0.a_o := f(\tau_1.a_1, \ldots, \tau_n.a_n)$
  $\tau_0.a_o$ est défini en fonction des attributs $\tau_1.a_1, \ldots, \tau_n.a_n$

Contraintes sur la place des attributs dans les calculs

$\tau_i.a$	$\tau_i = \tau_0$ (type défini)	$\tau_i \in \tau_{1..n}$ (type argument)
a synthétisé	$\tau_0.a := \ldots$	$\ldots := f(\ldots\tau_i.a\ldots)$
a hérité	$\ldots := f(\ldots\tau_0.a\ldots)$	$\tau_i.a := \ldots$

Rmq : même contraintes que pour les appels de fonctions en cascade. Une fonction peut :

- définir sa valeur de retour et les paramètres des fonctions qu'elle appelle
- utiliser ses paramètres et les valeurs de retour des fonctions qu'elle appelle

Exemple : Expressions : attributs

(1)……

Exemple : Expressions : calculs

(2)……

Une ASD attribuée par fonction

- Une ASD attribuée X définit une fonction sur X
- Des fonctions différentes nécessitent des attributs et calculs différents sur une même ASD
- Plusieurs fonctions peuvent être définies sur une même ASD
  - ex : expressions : évaluation, pretty-printing, compilation, dérivation
- $\Rightarrow$ modularité

AST attribué

Definition

Un AST attribué est un AST d'une ASD attribuée où :

- chaque noeud de type τ a une copie de
  - chaque attribut $\tau.a$ attaché à τ dans l'ASD attribuée

Analogie avec la programmation OO :

- chaque objet instance d'une classe C a une copie de
  - chaque attribut $C.a$ de la classe C

Exemple d'AST attribué (avant calcul)

(3)……

Dépendances entre attributs d'un AST

Question

Un attribut dépend de quels attributs pour le calcul de sa valeur ?

Pour chaque calcul $\tau_0.a_0 := f(\tau_1.a_1, \ldots, \tau_n.a_n)$:

- $\tau_0.a_0$ dépend de $\tau_1.a_1, \ldots, \tau_n.a_n$
- soit n relations de dépendance

Exemple d'AST attribué avec dépendances

(3'). . . . . .

Composition des calculs et propagation des valeurs

Processus de calcul global :

- initialisation : les valeurs des attributs hérités du noeud racine sont données
  - ce sont les données d'entrée du calcul global
- progression : on calcule la valeur d'un attribut dès que la valeur est connue pour les attributs dont il dépend
- terminaison : quand les valeurs des attributs synthétisés du noeud racine sont calculées
  - ce sont les données de sortie du calcul global

Remarque

Le processus termine s'il n'y a pas de dépendances circulaires !

Composition des calculs et propagation des valeurs

Processus de calcul global :

- initialisation : les valeurs des attributs hérités du noeud racine sont données
  - ce sont les données d'entrée du calcul global
- progression : on calcule la valeur d'un attribut dès que la valeur est connue pour les attributs dont il dépend
- terminaison : quand les valeurs des attributs synthétisés du noeud racine sont calculées
  - ce sont les données de sortie du calcul global

Remarque

Le processus termine s'il n'y a pas de dépendances circulaires !

Exemple de calcul des valeurs d'un AST attribué

(3")$\ldots\ldots$

Plan

Implémentation d'une ASD attribuée

- On s'appuie sur l'implémentation de l'ASD
- On fusionne
  - la définition des calculs
  - le processus de calcul (propagation)

Implémentation en ML

- ASD attribuée $\Rightarrow$ ens. fonctions mutuellement récursives
  - type $\tau \Rightarrow$ fonction f_τ
  - attribut hérité $\tau.h \Rightarrow$ paramètre de f_τ
  - attribut synthétisé $\tau.s \Rightarrow$ résultat de f_τ
- Chaque fonction f_τ est définie par pattern matching
  - avec un cas pour chaque variant de τ
  - avec un appel à f_{τ_i} pour chaque argument τ_i du variant

Exemple : évaluation d'expressions en OCaml

(4)……

Exemple : éxécution de l'évaluateur

```
# let expr = Binop (Plus, Var "x", Const 1) in
  let mem = function
            | "x" -> 10
            | "y" -> 2
            | _ -> invalid_arg "unbound variable" in
  expr |> eval_expr mem;;
- : int = 11
```

Implémentation en OO

- ASD attribuée $\Rightarrow$ ens. méthodes de même nom m
  - type $\tau \Rightarrow$ méthode abstraite $\tau.m$
  - attribut hérité $\tau.h \Rightarrow$ paramètre de la méthode
  - attribut synthétisé $\tau.s \Rightarrow$ résultat de la méthode
    si plusieurs résultats, les empaqueter dans une classe auxiliaire
- Chaque classe concrète implémente la méthode abstraite
  - avec les calculs du variant correspondant
  - avec un appel à $\tau_i.m$ pour chaque argument τ_i du variant
    choix implémentation par liaison dynamique

Exemple : évaluation d'expressions en Java

(6)……

Exemple : éxécution de l'évaluateur

```
// defining the expression AST
Expr e = new Binop(new Plus(),
                   new Var("x"),
                   new Const(1));
// defining the initial memory
Map<String,Integer> mem = new HashMap();
mem.put("x", new Integer(10));
mem.put("y", new Integer(2));
// running the evaluation
int val = e.eval(mem);
```

Implémentation à base de visiteurs

Problème :

- une méthode pour chaque fonction sur un ASD
- chaque classe mélange toutes les fonctions
- chaque fonction est éclatée entre toutes les classes

Solution : le design pattern Visitor permet de grouper le code par fonction

1. une interface *Visitor* avec une méthode par variant
2. une méthode `accept(Visitor v)` dans chaque classe
3. une implémentation de *Visitor* pour chaque fonction implémentant chaque méthode-variant avec les calculs associés

Exemple : interface Visitor

```
// interface Visitor pour les expressions
// H : type des attributs herites
// S : type des attributs synthetises
interface Visitor<H,S> {
    public S exprConst(H h,int val);
    public S exprVar(H h,String v);
    public S exprBinop(H h,Op op,Expr left,Expr right);
}
```

Exemple : méthode `accept`

```
// abstract class Expr
    abstract S accept(Visitor<S,H> v, H h);

// class Const extends Expr
    public S accept(Visitor<H,S> v, H h) {
        return v.exprConst(h, val); }

// class Var extends Expr
    public S accept(Visitor<H,S> v, H h) {
        return v.exprVar(h, v); }

// class Binop extends Expr
    public S accept(Visitor<H,S> v, H h) {
        return v.exprBinop(h, op, left, right); }
```

Exemple : visiteur pour l'évaluation d'expressions

```
class Eval implements Visitor<Map<String,Integer>, Integer> {
    public Integer exprConst(Map<String,Integer> mem, int val) {
        return new Integer(val);
    }
    public Integer exprVar(Map<String,Integer> mem, String v) {
        return mem.get(v);
    }
    public Integer exprBinop(Map<String,Integer> mem,
                             Op op, Expr left, Expr right) {
        int val_1 = left.accept(this, mem).intValue();
        int val_2 = right.accept(this,mem).intValue();
        if (op instanceof Plus)
            return new Integer(val_1 + val_2);
        else if (op instanceof Minus)
            return new Integer(val_1 - val_2);
        else if (op instanceof Times)
            return new Integer(val_1 * val_2);
        else
            throw new Exception("unknown operator");
    }
}
```

Comparaison ML/OO

implémentation	ML	OO-méthodes	OO-visiteur
concision	+	-	- -
nouvelle fonction	+	-	+
nouveau type/variant	-	+	-

Il y a une antagonie entre

- la facilité à définir de nouvelles fonctions sur l'ASD
- la facilité à ajouter des types/variants à l'ASD

Les langages OO permettent les deux, mais pas en même temps !

Plan

Pretty-printing des expressions : solution 1

(8)……

Exemples pour la solution 1

(8'). . . . . .

Pretty-printing des expressions : solution 2

Prise en compte des priorités des opérateurs avec les attributs suivants :

- *prio* : *int* (*s*) on *op* — priority of the operator
- *prioLeft* : *int* (*s*) on *op* — context priority for left argument
- *prioRight* : *int* (*s*) on *op* — context priority for right argument
- *prioCtx* : *int* (*h*) on *expr* — priority of the expression context

...

Pretty-printing des expressions : solution 2

$$
\begin{array}{lll}
op & ::= & \textbf{Plus} \\
 & & \left\{ \begin{array}{ll} op.string & := \texttt{"+"} \\ op.(prio, prioLeft, prioRight) & := \texttt{(1,1,1)} \end{array} \right. \\
 & | & \textbf{Minus} \\
 & & \left\{ \begin{array}{ll} op.string & := \texttt{"-"} \\ op.(prio, prioLeft, prioRight) & := \texttt{(1,1,2)} \end{array} \right. \\
 & | & \textbf{Times} \\
 & & \left\{ \begin{array}{ll} op.string & := \texttt{"*"} \\ op.(prio, prioLeft, prioRight) & := \texttt{(2,2,2)} \end{array} \right.
\end{array}
$$

Pretty-printing des expressions : solution 2

$$
\begin{array}{lll}
expr & ::= & \textbf{Const}(int) \\
 & & \{\ expr.string := \texttt{stringOfInt}(int) \\
 & | & \textbf{Var}(string) \\
 & & \{\ expr.string := string \\
 & | & \textbf{Binop}(op, expr_1, expr_2) \\
 & & \left\{ \begin{array}{lll}
expr_1.prioCtx & := & op.prioLeft \\
expr_2.prioCtx & := & op.prioRight \\
expr.string & := & \texttt{let s =}\ expr_1.string \\
 & & \quad \texttt{++}\ op.string \\
 & & \quad \texttt{++}\ expr_2.string\ \texttt{in} \\
 & & \texttt{if}\ op.prio < expr.prioCtx \\
 & & \texttt{then "("} \texttt{++ s ++}\ \texttt{")"} \\
 & & \texttt{else s}
\end{array} \right.
\end{array}
$$

Exemples pour la solution 2

(9)……

Compilation d'expressions régulières en automates

Quelle fonction ?

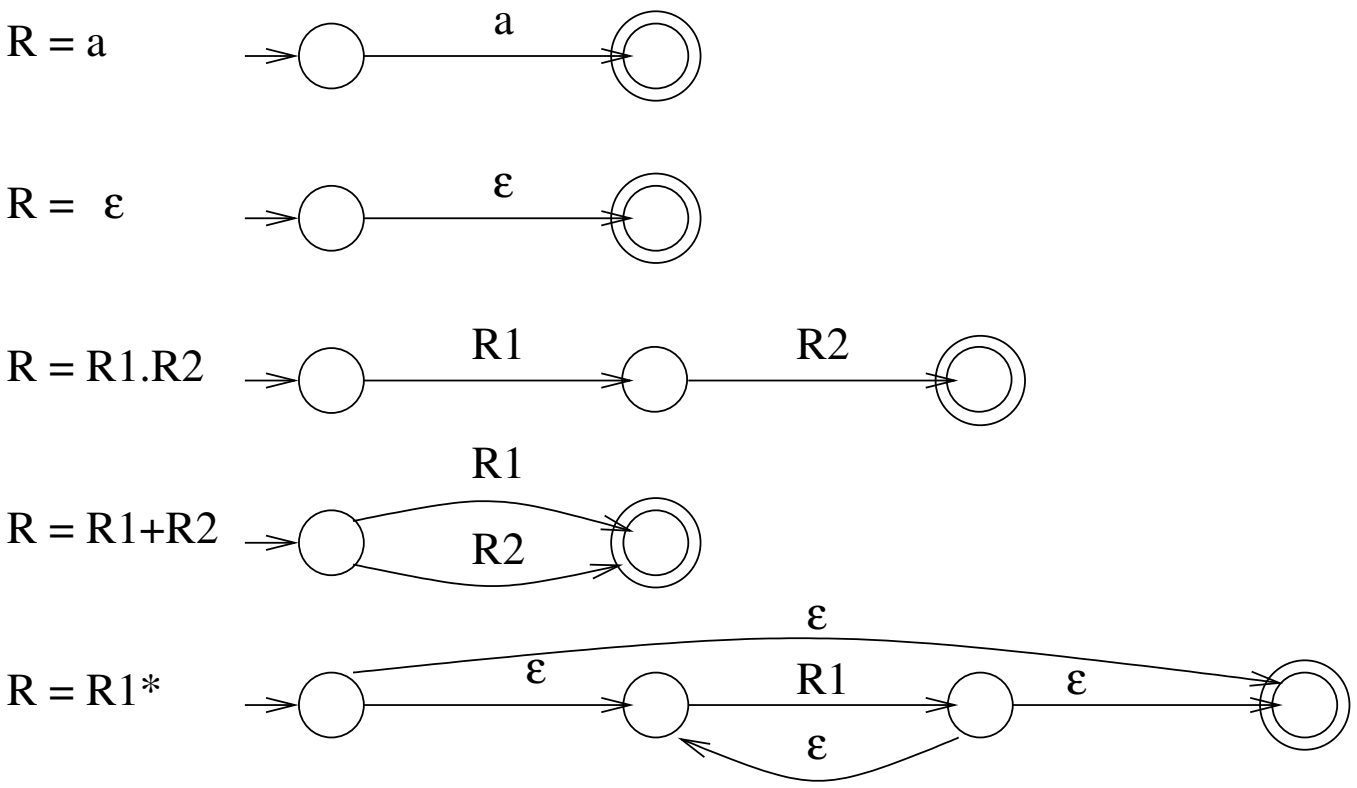
- entrée : expression régulière
  - AST appartenant à l'ASD des expressions régulières
  - représente un ensemble de mots
- sortie : automate fini
  - ensemble d'états Q et ensemble de transitions Δ
  - reconnait un ensemble de mots

C'est bien de la compilation

- le résultat en sortie est "éxécutable" (automate)
- la sémantique est préservée : même ensemble de mots

Algorithme de Thompson

Production récursive d'un automate fini non-déterministe



L'automate obtenu peut ensuite être déterminisé de façon classique.

Attributs pour la compilation d'expressions régulières

- *delta* : *set*(*transition*) (*s*) sur *regexp* — ens. de transitions
- *init* : *state* (*h*) sur *regexp* — état initial
- *final* : *state* (*h*) sur *regexp* — état final

ASD attribuée pour la compilation d'expressions régulières (1/2)

$$
\begin{array}{lll}
\mathit{regexp} & ::= & \textbf{Char}(\mathit{char}) \\
 & & \left\{ \; \mathit{regexp.delta} := \{(\mathit{regexp.init}, \mathit{char}, \mathit{regexp.final})\} \right. \\
 & | & \textbf{Epsilon} \\
 & & \left\{ \; \mathit{regexp.delta} := \{(\mathit{regexp.init}, \varepsilon, \mathit{regexp.final})\} \right. \\
 & | & \textbf{Concat}(\mathit{regexp}_1, \mathit{regexp}_2) \\
 & & \left\{ \begin{array}{lll}
 x & := & \texttt{newState()} \\
 \mathit{regexp}_1.(\mathit{init}, \mathit{final}) & := & (\mathit{regexp.init}, x) \\
 \mathit{regexp}_2.(\mathit{init}, \mathit{final}) & := & (x, \mathit{regexp.final}) \\
 \mathit{regexp.delta} & := & \mathit{regexp}_1.\mathit{delta} \\
 & \cup & \mathit{regexp}_2.\mathit{delta}
\end{array} \right.
\end{array}
$$

ASD attribuée pour la compilation d'expressions régulières (2/2)

| **Alt**($regexp_1$, $regexp_2$)

$$\begin{cases} regexp_1.(init, final) & := & (regexp.init, regexp.final) \\ regexp_2.(init, final) & := & (regexp.init, regexp.final) \\ regexp.delta & := & regexp_1.delta \\ & \cup & regexp_2.delta \end{cases}$$

| **Closure**($regexp_1$)

$$\begin{cases} x & := & \texttt{newState()} \\ y & := & \texttt{newState()} \\ regexp_1.(init, final) & := & (x, y) \\ regexp.delta & := & regexp_1.delta \\ & \cup & \{(regexp.init, \varepsilon, x)\} \\ & \cup & \{(y, \varepsilon, regexp.final)\} \\ & \cup & \{(regexp.init, \varepsilon, regexp.final)\} \\ & \cup & \{(y, \varepsilon, x)\} \end{cases}$$

Soyons réflexifs ! Compilation d'ASD en Java

Quelle fonction ?

- entrée : une définition de syntaxe abstraite (ASD)
  - AST appartenant à l'ASD des ASD
  - représente un langage, un ensemble de structures syntaxiques
- sortie : code Java

C'est bien de la compilation

- le résultat en sortie est "éxécutable" (après compilation du Java)
- la sémantique est préservée : l'ensemble des objets bien typés coïncide avec l'ensemble des structures syntaxiques

Rappel : l'ASD des ASD

$$
\begin{array}{lcl}
\textit{asd} & ::= & \textbf{ASD}(\textit{typedef}*) \\
\textit{typedef} & ::= & \textbf{Typedef}(\textit{string}, \textit{variant}*) \\
\textit{variant} & ::= & \textbf{Variant}(\textit{string}, \textit{argument}*) \\
\textit{argument} & ::= & \textbf{One}(\textit{string}) \\
 & | & \textbf{Optional}(\textit{string}) \\
 & | & \textbf{List}(\textit{string})
\end{array}
$$

Attributs pour la compilation d'ASD en Java

- *java* : *string* (*s*) sur tous les types
  code Java généré
- *type* : *string* (*h*) sur *variant*
  type auquel le variant appartient
- *rank* : *int* (*h*) sur *arg*
  rang de l'argument dans son variant
- *name* : *string* (*s*) sur *arg*
  nom de l'attribut Java pour l'argument

ASD attribuée pour la compilation d'ASD en Java (1/2)

$$
\begin{array}{lll}
\textit{asd} & ::= & \textbf{ASD}(\textit{typedef}*) \\
 & & \{\ \textit{asd.java} := \texttt{concat(}\textit{typedef}*\texttt{.java)} \\
\textit{typedef} & ::= & \textbf{Typedef}(\textit{string}, \textit{variant}*) \\
 & & \left\{ \begin{array}{lll}
\textit{variant}*\texttt{.type} & := & \textit{string} \\
\textit{typedef.java} & := & \texttt{"abstract class "++}\textit{string}\texttt{++"\{ \}"} \\
 & @@ & \texttt{concat(}\textit{variant}*\texttt{.java)}
\end{array} \right. \\
\textit{variant} & ::= & \textbf{Variant}(\textit{string}, \textit{arg}*) \\
 & & \left\{ \begin{array}{lll}
\textit{arg}_i\texttt{.rank} & := & i \\
\textit{variant.java} & := & \texttt{"class "++}\textit{string} \\
 & ++ & \texttt{" extends "++}\textit{variant.type}\texttt{++" \{"} \\
 & @@ & \texttt{concat(}\textit{arg}*\texttt{.java++";")} \\
 & @@ & \texttt{"public "++}\textit{string}\texttt{++"("} \\
 & ++ & \texttt{concatSep(}\textit{arg}*\texttt{.java,",")++") \{"} \\
 & @@ & \textit{arg}*\texttt{.name++" = "++}\textit{arg}*\texttt{.name++";"} \\
 & @@ & \texttt{"\} \}"}
\end{array} \right.
\end{array}
$$

ASD attribuée pour la compilation d'ASD en Java (2/2)

$$
\begin{array}{lll}
\textit{arg} & ::= & \textbf{One}(\textit{string}) \\
 & & \left\{ \begin{array}{lll} \textit{arg.name} & := & \textit{string}{+}{+}\textit{arg.rank} \\ \textit{arg.java} & := & \textit{string}{+}{+}\texttt{" "}{+}{+}\textit{arg.name} \end{array} \right. \\
 & | & \textbf{Optional}(\textit{string}) \\
 & & \left\{ \begin{array}{lll} \textit{arg.name} & := & \textit{string}{+}{+}\textit{arg.rank} \\ \textit{arg.java} & := & \textit{string}{+}{+}\texttt{" "}{+}{+}\textit{arg.name} \end{array} \right. \\
 & | & \textbf{List}(\textit{string}) \\
 & & \left\{ \begin{array}{lll} \textit{arg.name} & := & \textit{string}{+}{+}\textit{arg.rank} \\ \textit{arg.java} & := & \texttt{List<}\textit{string}\texttt{>}{+}{+}\texttt{" "}{+}{+}\textit{arg.name} \end{array} \right.
\end{array}
$$

The end