Grammaires Attribuées

Martin Odersky

18 décembre 2006
version 1.1

En collaboration avec :

Gilles Dubochet et Stéphane Micheloud

Plan du cours

Grammaire attribuée

Une syntaxe dépendant du contexte est parfois spécifiée en utilisant une grammaire attribuée.

- Les grammaires attribuées peuvent reposer sur la syntaxe non-contextuelle concrète ou abstraite.
- La syntaxe est enrichie par des attributs et les règles d'attribution.
- Les attributs sont attachés aux symboles ; ils peuvent avoir un type quelconque.
- On représente les attributs comme des variables d'instance des noeuds de l'arbre.

Règles d'attribution

- On ajoute des attributs et des règles d'attribution aux grammaires non-contextuelles.
- A chaque symbole peuvent être attachés des attributs.
- A chaque production peuvent être associées des règles d'attribution qui mettent en relation les attributs des symboles de la production.

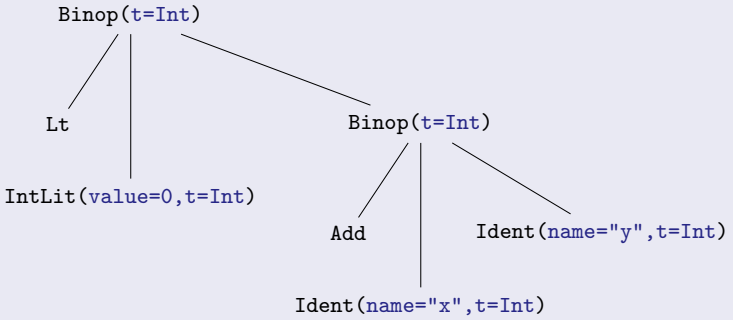
Exemple : Attributs

```
E(env, t) = Binop ADD                t1 = t2 = Int, t = Int
            E(env, t1) E(env, t2)
          | Binop EQ                 t1 <: t2 || t2 <: t1
            E(env, t1) E(env, t2)    t = lub(t1, t2)
          | Ident name               sym = env.lookup(name)
                                     t = sym.tpe
          | IntLit n                 t = Int
```

Une attribution est une affectation de tous les attributs d'un arbre syntaxique satisfaisant toutes les règles d'attribution.

Exemple : Attribution de Binop

`0 < (x + y)` où `x` et `y` sont de type `Int`



Un programme est légal si :

- c'est une phrase du langage décrit par la grammaire non-contextuelle, et
- il existe une attribution pour son arbre.

Un langage est donc complètement caractérisé par sa grammaire non-contextuelle et sa grammaire contextuelle.
Toutefois, rien n'est encore dit au sujet de la signification d'un programme (qu'on appelle aussi sa sémantique).

Attributs

Des exemples typiques d'attributs sont :

- Le *type* d'une expression
- Le *symbole* produit par une déclaration
- La *table des symboles* (ou portée) produite par un ensemble de déclarations.

Les tables des symboles sont souvent représentées par des variables globales plutôt que par un ensemble d'attributs.
Voilà une différence entre la théorie et la pratique !
Il est toutefois important de s'assurer que la pratique ne détruit pas les concepts théoriques. Les tables des symboles peuvent être vues comme un attribut, mais on choisit une représentation centralisée plus performante.

Grammaire attribuée pour MFL

Integers	i, j			
Identifiers	$x, f,$	$\ldots$		
Expressions	E, F	$::=$	i	Literal
		$\mid$	x	Identifier
		$\mid$	$E_1 + E_2$	Addition
		$\mid$	$E_1 = E_2$	Comparison
		$\mid$	$E_1(E_2)$	Application
		$\mid$	**if** $(E_1)\ E_2$ **else** E_3	Conditional
		$\mid$	**val** $x : T = E_1;\ E_2$	Value def
		$\mid$	**def** $f(x : T_1) : T_2 = E_1;\ E_2$	Function def
Types	S, T	$::=$	**int**	
		$\mid$	**boolean**	
		$\mid$	$T_1 \Rightarrow T_2$	

Attributs pour MFL

- Attribut de type : `t = int, boolean, t -> t`
- Attribut de symbole : `sym`
- Attribut de environnement : `env`

Règles d'attribution

voir tableau

La relation de sous-typage

Définition : Relation de sous-typage

S est un sous-type de T, ou S est compatible avec T, ssi S = T ou `S = NullType`, `T = ClassType(C)` ou S et une class qui a T comme une de ses superclasses.
La relation de sous-typage s'écrit `S <: T`

Exemples : Sous-type dans *Zwei*

Les trois relations suivantes sont vrais :
`Null <: List`
`List <: List` (relation réflexive)
`List <: Collection` (héritage)
Mais pas :
`Int <: List`

Borne supérieure de 2 types

Le type de la valeur effectivement retournée par une expression conditionnelle **if** (E_1) E_2 **else** E_3 est la borne supérieure des types de E_2 et E_3 : lub(T_2, T_3).
Voici comment la borne supérieure peut être calculée :

La méthode `lub`

```
def lub(t1: Type, t2: Type): Type = Pair(t1, t2) match {
  case Pair(IIntType, IIntType) => IIntType
  case Pair(IClassType(x), INullType) => t1
  case Pair(INullType, IClassType(x)) => t2
  case Pair(IClassType(c1), IClassType(c2)) =>
    intersection(c1.superclasses, c2.superclasses) match {
      case t :: ts => t
      case List() => badLub(t1, t2)
    }
  case x => badLub(t1, t2)
}
```


- Question : Comment calculer l'intersection de deux listes ?
- Donnez une implantation de la méthode suivante :

La méthode `intersection`

```
def intersection[a](xs: List[a], ys: List[a]): List[a]
  = ?
```

- Truc : Considérez utiliser les méthodes `filter` et `contains` de la classe `List`.

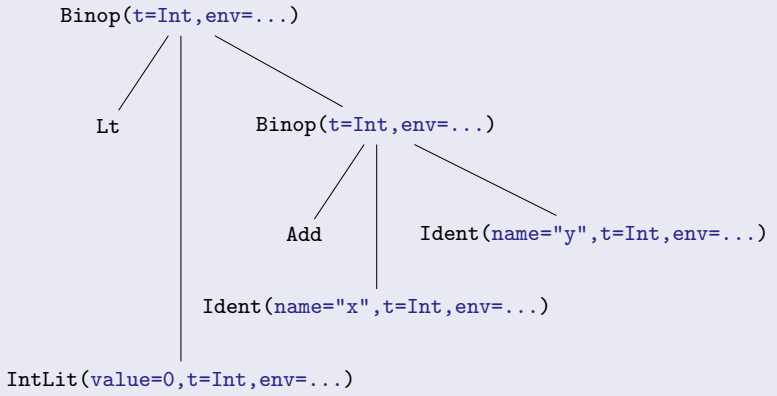
Comment implanter la spécification ?

Au lieu de deviner les attributs et de vérifier ensuite qu'ils satisfont les règles d'attribution, on doit les *calculer*.
Les attributs sont en général calculés à partir de la valeur d'autres attributs.
Important : les attributs doivent être affectés une et une seule fois !
Les attributs peuvent être distingués en fonction de la manière dont ils *traversent* l'arbre syntaxique.

- Certains le traversent de bas en haut - ils sont dits *synthétisés*.
- Certains le traversent de haut en bas - ils sont dits *hérités*.

Dans un compilateur, les attributs synthétisés sont représentés par les types de retour (ou paramètres de sortie en C/C++) des visiteurs.
Les attributs hérités sont représentés par les paramètres (d'entrée) des visiteurs.

Exemple : Attributs synthétisés et hérités



`s` est hérité, `t` est synthétisé et `name` et `value` sont intrinsèques.

Représentation des attributs

Définition : Attribut persistant

Un attribut utilisé également dans des phases ultérieures de la compilation est dit *persistant*. Un attribut persistant est en général stocké comme champ supplémentaire dans l'arbre syntaxique.

Définition : Attribut temporaire

Un attribut utilisé uniquement lors la vérification de type est dit *temporaire*. Un attribut temporaire est soit le paramètre, soit le résultat des méthodes des visiteurs :

- Attribut temporaire synthétisé = résultat de visiteur.
- Attribut temporaire hérité = paramètre de visiteur.

Certains attributs peuvent aussi être représentés par des variables globales. Cela est plus simple si ces attributs changent peu souvent.

Exemple : Attributs synthétisés et hérités

```
E(t,s)  =  Binop IntOp E(t1, s1) E(t2, s2)
        |  Binop CmpOp E(t1, s1) E(t2, s2)
        |  Ident ident (sym)
        |  ...
```

Les attributs suivants existent :

- `t` : (type) synthétisé, persistant
- `sym` : (symbole) intrinsèque, persistant
- `env` : (portée) hérité, temporaire

Concrètement on utilise pour l'analyse des visiteurs d'arbres en Java resp. des fitrages de motifs en Scala.

Exemple : Analyse des types en Scala (1/3)

```
class Analyzer {
  // Global scope of classes
  type ClassScope = Map[String, ClassSymbol];
  var classScope: ClassScope = ListMap.Empty;

  // For local scopes of variables
  type VarScope = Map[String, VarSymbol];
  val emptyVarScope: VarScope = ListMap.Empty;

  // Analyze a program
  def analyzeProgram(tree: Program): Unit = tree match {
    case Program(classes, main) =>
      classes.foreach(analyzeClass);
      analyzeExpr(emptyVarScope, main);
      ()
  }

  // Analyze a class
  def analyzeClass(tree: ClassDef): Unit = tree match { /* .. */ }
```


Exemple : Analyse des types en Scala (2/3)

```
  // Analyze an expression
  def analyzeExpr(varScope: VarScope, tree: Expr): Type = tree match {
    case Ident(name) =>
      varScope.get(name.name) match {
        case Some(v) =>
          name.sym = v;
          v.vartype
        case None =>
          Report.error(tree.pos, "Unknown␣variable␣" + name);
          IBadType
      }
    // ...
  }
```


Exemple : Analyse des types en Scala (3/3)

```
  // Analyze an expression
  def analyzeExpr(varScope: VarScope, tree: Expr): Type = tree match {
    case Binop(op, left, right) =>
      val t1 = analyzeExpr(varScope, left);
      val t2 = analyzeExpr(varScope, right);
      if (op == EQ || op == NE) {
        if (! (t1.isSubtype(t2) || t2.isSubtype(t1)))
          Report.error(tree.pos,
                       "Incompatible␣types:␣" + t1 + "␣<->␣" + t2);
      }
      else {
        checkIntType(left.pos, t1);
        checkIntType(right.pos, t2)
      }
      IIntType
    // ...
  }
```


Exemple : Analyse des types en Java (1/5)

```
class Analyzer {
    private final ProgramVisitor programVisitor;
    private final ExprVisitor    exprVisitor;
    // ...

    // Global scope of classes
    private final ClassScope classScope = new ClassScope();

    public Analyzer() {
        programVisitor = new ProgramVisitor();
        exprVisitor    = new ExprVisitor();
    }

    // Analyze the tree
    public Type analyze(Tree tree) {
        programVisitor.analyze(tree);
        return tree.type;
    }
    // ...
}
```

Exemple : Analyse des types en Java (2/5)

```
// Analyze a program
private class ProgramVisitor extends DefaultVisitor {

    public void analyze(Tree tree) {
        tree.apply(this);
    }

    public void caseProgram(Program tree) {
        classVisitor.analyzeEach(tree.decls);
        exprVisitor.analyze(new VarScope(), tree.main);
    }
    // ...
}

// Analyze a class
private class ClassVisitor extends DefaultVisitor { /* ..*/ }
```


Exemple : Analyse des types en Java (3/5)

```
// Analyze an expression
private class ExprVisitor extends DefaultVisitor {
    private VarScope varScope;

    public Type analyze(VarScope scope, Tree expr) {
        VarScope oldScope = varScope;
        varScope = new VarScope(scope);
        expr.apply(this);
        varScope = oldScope;
        return expr.type;
    }

    public void caseIdent(Ident tree) {
        VarSymbol v = varScope.lookup(tree.name.name);
        if (v != null) {
            tree.name.sym = v; tree.type = v.vartype;
        } else {
            Report.error(tree.pos, "Unknown␣variable␣" + tree.name);
            tree.type = Type.IBadType;
    }}
}
```


Exemple : Analyse des types en Java (4/5)

```
// Analyze an expression
private class ExprVisitor extends DefaultVisitor {
    private VarScope varScope;

    // ...
    public void caseBinop(Binop tree) {
        Type t1 = analyze(varScope, tree.left);
        Type t2 = analyze(varScope, tree.right);
        if (tree.op == Tree.EQ || tree.op == Tree.NE) {
            if (! (t1.isSubtype(t2) || t2.isSubtype(t1)))
                Report.error(tree.pos,
                    "Incompatible␣types:␣" + t1 + "␣<->␣" + t2);
        } else {
            checkIntType(tree.left.pos, t1);
            checkIntType(tree.right.pos, t2);
        }
        tree.type = Type.IIntType;
    }
}
```


Exemple : Analyse des types en Java (5/5)

```
// For the global scope of classes
private static class ClassScope {
    private final Map/*<String, ClassSymbol>*/ map;
    ClassScope() { map = new LinkedHashMap(); }
    void enter(String name, ClassSymbol c) { map.put(name, c); }
    ClassSymbol lookup(String name) {
        return (ClassSymbol) map.get(name);
    }
}

// For local scopes of variables
private static class VarScope {
    private final Map/*<String, VarSymbol>*/ map;
    VarScope() { map = new LinkedHashMap(); }
    VarScope(VarScope scope) { map = new LinkedHashMap(scope.map); }
    void enter(String name, VarSymbol v) { map.put(name, v); }
    VarSymbol lookup(String name) {
        return (VarSymbol) map.get(name);
    }
}
```

Optimisations

La création d'un visiteur pour chaque noeud visité prend du temps $\Longrightarrow$ il est préférable de réutiliser le visiteur courant.

```
public Type analyze(VarScope scope, Tree expr) {
    VarScope oldScope = varScope;
    varScope = new VarScope(scope);
    expr.apply(this);
    varScope = oldScope;
    return expr.type;
}
```

L'attribut `scope` ne varie pas fréquemment $\Longrightarrow$ on peut utiliser une méthode d'analyse plus efficace qui ne fait pas la sauvegarde/changement/restauration de la portée.

```
public Type analyze(Tree tree) {
    tree.apply(this);
}
```


On doit souvent comparer deux types et signaler une erreur s'ils ne sont pas compatibles $\Longrightarrow$ on peut créer une fonction `checkSameType` qui se charge de faire cela :

```
private void checkSameType(int pos, Type found, Type expected) {
    if (found.isSameType(expected)) return found;
    error(pos, "type␣error:␣expected␣" + expected
                          + ",␣found␣" + found);
}
private void checkIntType(int pos, Type found) {
    checkSameType(pos, found, Type.IIntType);
}
```

Dès lors `caseBinop` peut s'écrire de manière plus concise.

```
public void caseBinop(Binop tree) {
    if (tree.op == ADD) {
        checkIntType(tree.left.pos, analyze(tree.left));
        checkIntType(tree.right.pos, analyze(tree.right));
        tree.type = Type.IIntType;
    } // ...
}
```


Résumé

L'analyse de noms et la vérification des types sont parmi les tâches les plus complexes d'un compilateur.
Tâches :

1. Vérifier que l'arbre donné est légal par rapport aux règles du langage (grammaire contextuelle).
2. Déterminer certains attributs de l'arbre (p.ex. : `t`, `sym`), qui seront nécessaires dans des phases ultérieures.

La grammaire contextuelle peut être spécifiée au moyen d'une grammaire attribuée.
Les attributs sont des champs des noeuds de l'arbre, calculés au moyen des règles d'attribution.

On distingue 3 classes d'attributs :

- hérités,
- synthétisés,
- intrinsèques.

Les attributs sont représentés par :

- des champs de l'arbre, ou
- des paramètres ou des résultats de la méthode du visiteur, ou
- des champs de la classe du visiteur.

Mise en oeuvre :

- visiteurs d'arbres en Java,
- filtrages de motifs en Scala.