Construction de l'arbre abstrait avec SableCC

Alexis Nasr
Franck Dary
Pacôme Perrotin

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

Principe

- La construction de l'arbre abstrait se fait lors d'un parcours de l'arbre de dérivation à l'aide du visiteur `DepthFirstAdapter` généré par `Sablecc`.
- Une classe pour chaque type de nœud, définie dans le package `sa`.

Le package sa

Classe	Interface	implémente	hérite de
	SaNode		
	SaVar SaDec SaExp SaInst		SaNode SaNode SaNode SaNode
SaInst*		SaInst	
SaExp*		SaExp	
SaDecVar SaDecFonc SaDecTab		SaDec SaDec SaDec	
SaVarIndicee SaVarSimple		SaVar SaVar	
SaAppel		SaExp, SaInst	
SaLExp SaLInst SaLDec		SaNode SaNode SaNode	
SaProg		SaNode	

Parcours de l'arbre de dérivation

- Le visiteur `DepthFirstAdapter` permet de réaliser un parcours en profondeur de l'arbre de dérivation.
- Il définit pour chaque type de nœud `X` la méthode :

```
public void caseX(X node)
```

- Qui réalise le parcours d'un nœud de la classe `X`.
- Exemple pour la classe `APlusExp3` :

```
public void caseAPlusExp3(APlusExp3 node){
    inAPlusExp3(node);
    if(node.getExp3() != null) node.getExp3().apply(this);
    if(node.getPlus() != null) node.getPlus().apply(this);
    if(node.getExp4() != null) node.getExp4().apply(this);
    outAPlusExp3(node);
}
```

- Les classes correspondant aux nœuds de l'arbre de dérivation définissent la méthode

```
public void apply(Switch sw)
```

- Exemple de la classe `APlusExp3` :

```
public void apply(Switch sw)
  {((Analysis) sw).caseAPlusExp3(this);}
```

Parcours de l'arbre de dérivation

- On définit un visiteur `Sc2sa` fondé sur `DepthFirstAdapter`

```
public class Sc2sa extends DepthFirstAdapter
```

- Qui définit la méthode `caseX(X node)` pour chaque type de nœud `X`.
- Pour la classe `APlusExp3` on aurait aimé écrire :

```
// exp3 = {plus} exp3 plus exp4
public SaExp caseAPlusExp3(APlusExp3 node){
    SaExp op1 = node.getExp3().apply(this);
    SaExp op2 = node.getExp4().apply(this);
    return new SaExpAdd(op1, op2);
}
```

- Mais les méthodes `apply` et `case` ne renvoient rien (`void`).

```
public void apply(Switch sw)
public void caseAPlusExp3(APlusExp3 node)
```

Parcours de l'arbre de dérivation

- On définit dans la classe `Sc2sa` une variable d'instance `returnValue`, qui permet de simuler le retour des fonctions.
- On écrit `caseAPlusExp3` de la façon suivante :

```
public class Sc2sa extends DepthFirstAdapter{
    private SaNode returnValue;
    // exp3 = {plus} exp3 plus exp4
    public void caseAPlusExp3(APlusExp3 node)
    {
        SaExp op1 = null;
        SaExp op2 = null;
        node.getExp3().apply(this);
        op1 = (SaExp) this.returnValue;
        node.getExp4().apply(this);
        op2 = (SaExp) this.returnValue;
        this.returnValue = new SaExpAdd(op1, op2);
    }
```

- La valeur calculée par la fonction appelée est mise dans `returnValue` et sera accessible pour la fonction appelante.
- Attention : il est indispensable de sauvegarder la valeur de `returnValue` après un appel de fonction car elle sera écrasée par le prochain appel !

Pas de création de nœuds

- Dans certains cas, un nœud de l'arbre de dérivation ne provoque pas la création d'un nœud de l'arbre abstrait.
- Sinon l'arbre abstrait ne serait pas plus simple que l'arbre de dérivation !
- C'est le cas notamment pour les règles unaires des expressions, du type : $E_3 \rightarrow E_4$

```
public void caseAExp4Exp3(AExp4Exp3 node){
        if(node.getExp4() != null)
        {
            node.getExp4().apply(this);
        }
    }
```

- Dans ce cas, on ne fait rien : la valeur se trouvant dans `returnValue` à l'issue de l'appel `node.getExp4().apply(this)`, reste dans `returnValue`.

Affichage

- Les fonctions de `DepthFirstAdapter`, de la forme `caseX` commencent toutes par un appel à la fonction `inX` et se terminent par un appel à la fonction `outX`

```
public void caseAExp4Exp3(AExp4Exp3 node){
        inAExp4Exp3(node);
        ...
        outAExp4Exp3(node);
    }
```

- Ces fonctions sont définies par un appel aux fonctions `defaultIn` et `defaultOut`

```
public void inAExp4Exp3(AExp4Exp3 node){defaultIn(node);}
public void outAExp4Exp3(AExp4Exp3 node){defaultOut(node);}
```

- Il est conseillé de définir `defaultIn` et `defaultOut` de manière à afficher le nœud visité, par exemple :

```
public void defaultIn(Node node){
    System.out.println("<" + node.getClass().getSimpleName() + ">");}
public void defaultOut(Node node){
    System.out.println("</" + node.getClass().getSimpleName() + ">");}
```

- Cela vous sera très utile pour debugger !

Représentation de l'arbre abstrait en `xml`

- Il est possible grâce à la classe `Sa2Xml` de représenter l'arbre abstrait au format `xml`
- C'est fait automatiquement par `Compiler` lorsqu'il est lancé avec l'option `-v 2`

```
System.out.println("[BUILD SA] ");
Sc2sa sc2sa = new Sc2sa();
tree.apply(sc2sa);
SaNode saRoot = sc2sa.getRoot();

if(verboseLevel > 1){
  System.out.println("[PRINT SA]");
  new Sa2Xml(saRoot, baseName);
}
```

- Le résultat est écrit dans le fichier `fichier.xml` où `fichier.l` est le fichier sur lequel est lancé la compilation.

Représentation de l'arbre abstrait en xml

```
<programme type="SaProg">
 <variables type="SaLDec">
  <tete type="SaDecVar" nom="a"/>
 </variables>
 <fonctions type="SaLDec">
  <tete type="SaDecFonc" nom="main">
   <corps type="SaInstBloc">
    <val type="SaLInst">
     <tete type="SaInstAffect">
      <lhs type="SaVarSimple" nom="a"/>
      <rhs type="SaExpInt" val="1"/>
     </tete>
     <queue type="SaLInst">
      <tete type="SaInstEcriture">
       <arg type="SaExpVar">
        <var type="SaVarSimple" nom="a"/>
       </arg>
      </tete>
     </queue>
    </val>
   </corps>
  </tete>
 </fonctions>
</programme>
```