La syntaxe abstraite de Drei

Martin Odersky

20 novembre 2006

version 1.1

La syntaxe abstraite de Drei

```
P = Program { D } S
D = ClassDef name [ name ] { M }
M = FieldDecl name T
  | MethodDef name { name T } T E
T = ClassType name
  | IntType
  | NoType
S = While E { S }
  | If E S S
  | Var name T E
  | Set name E
  | Do E
  | PrintInt E
  | PrintChar E
  | PrintChar E
  | Compound { S }
```

```
E = Ident name
  | New name { E }
  | Select E name
  | Call E name { E }
  | IntLit int
  | Unop U E
  | Binop B E E
  | ReadInt
  | ReadChar
  | Block { S } E
  | Empty
U = Not | Neg
B = Add | Sub | Mul
  | Div | Mod | Eq
  | Ne | Lt | Le
  | Gt | Ge | And
```

Quelques alternatives de la grammaire méritent un commentaire :

- dans `ClassDef` $\texttt{name}_1$ `[` $\texttt{name}_2$ `] { M }`, l'attribut $\texttt{name}_2$ est le nom de la super-classe, qui peut être vide ;
- dans `MethodDef` $\texttt{name}_1$ `{` $\texttt{name}_2$ `T } T E`, les couples $\texttt{name}_2$ `T` représentent les noms et types des paramètres ;
- `Var` déclare une variable, `Set` redéfinit la valeur d'une variable et `Do` est une instruction sans modification de variable ;
- `Select` est l'accès à un champ d'une instance, `Call` est l'appel d'une méthode d'une instance.
- `Compound` est une instruction (*statement*) composée de plusieurs instructions, `Block` est similaire mais se termine par une expression dont il retourne la valeur, ce qui en fait lui-même une expression.

Obtenir un arbre de syntaxe abstraite (AST)

L'AST de Drei est obtenu en définissant :

- une classe abstraite `Tree` ;
- une sous-classe abstraite de `Tree` pour chaque non-terminal `P`, `D`, etc. de la grammaire ;
- une sous-classe `case` concrète de la classe du non-terminal pour chaque alternative `Program`, `ClassDef`, `FieldDecl`, `MethodDef`, etc. de la grammaire.

Définition de Tree pour Drei

```
abstract class Tree {
  private var p: Int = Position.UNDEFINED;
  def pos: Int = p;
  def setPos(p: Int): this.type = { this.p = p; this }
}

/** P = Program { D } E */
case class Program (classes: List[ClassDef], main: Expr) extends Tree;

/** A common superclass for tree nodes designating types */
abstract class TypeTree extends Tree;

/** T = Int */
case class IntType extends TypeTree;
// ...
```

On notera dans le définition de `Tree` pour Drei que :

- le constructeur de chaque classe se compose des différents sous-arbres de l'alternative correspondante dans la grammaire ;
- l'option est exprimée par la classe `Option` de Scala : `[ name ]` dans la grammaire devient `Option[Name]` dans l'arbre ;
- la répétition est exprimée par des listes : `{ T }` dans la grammaire devient `List[T]` dans l'arbre ;

Il est pratique de représenter les identifiants par des instances d'une classe `Name`. Pour le moment, celle-ci est un simple enrobage pour une chaîne de caractères — mais l'utilisation d'une classe dédiée sera très utile par la suite.

```
case class Name(val name: String) {
  override def toString() = name
}
```

Le champ `pos` contient la position courante dans l'arbre — importante pour produire des messages d'erreur :

- `pos` est commun à tous les types d'arbres ; c'est pourquoi il est membre de la classe `Tree`.
- `pos` ne fait pas partie du constructeur des classes, ceci afin de ne pas «polluer» le filtrage de motif :
  - `setPos` doit être appelée juste après la construction de la classe — c'est une sorte de post-constructeur.
  - `setPos` retourne `this` : il est ainsi possible de chaîner l'appel à `setPos` directement au constructeur :

    ```
    val prog = Program(classes, main) setPos pos;
    ```