## RefactoringNG

### Introducción

RefactoringNG [ref] es una herramienta de refactorización general para java. La refactorización se define en un fichero que se compone de un conjunto de reglas de refactorización. Cada regla describe una transformación de un árbol de sintaxis abstracta (AST) a otro. Cada regla se compone de los siguientes elementos principales:

Patrón -> Transformación

El patrón es un árbol AST del código fuente original y la transformación define los cambios que se aplicarán al patrón original. Por ejemplo, la siguiente regla transforma “p = null” en “p = 0”:

Assignment {

Identifier [name: "p"],

Literal [kind: NULL\_LITERAL]

} ->

Assignment {

Identifier [name: "p"],

Literal [kind: INT\_LITERAL, value: 0]

}

El patrón y la transformación cumplen con la misma estructura:

#### Árbol, atributos y contenido

Los árboles toman el nombre de los ASTs y los atributos los de las propiedades del compilador para Java de Sun. Sólo el árbol es obligatorio, tanto los atributos como el contenido son descriptores opcionales. Los atributos están encerrados entre corchetes “[]” y están separados por comas. Éstos especifican información adicional del árbol. Por ejemplo, en:

Literal [kind: NULL\_LITERAL]

el atributo “kind” indica que el literal es el literal “null”. Dentro del patrón, si el atributo no es especificado, puede tomar cualquier valor. Por ejemplo:

Identifier

indica cualquier identificador y :

Literal [kind: INT\_LITERAL]

indica cualquier literal “int”. En la transformación el árbol debe ser descrito completamente para que un nuevo árbol pueda ser creado. Por ejemplo cada identificador en la transformación debe poseer el atributo “name”.

El contenido estára encerrado entre llaves “{}” y es una lista separada por comas de los hijos del nodo del árbol dado. Por ejemplo:

Binary [kind: PLUS] {

Literal [kind: INT\_LITERAL],

Literal [kind: INT\_LITERAL]

}

representa la suma de dos literales enteros. Los hijos de un árbol deben pertenecer a un tipo apropiado y todos ellos deben ser especificados si el árbol tiene contenido. Por ejemplo, si se define contenido en un operador binario debe tener siempre dos hijos (operandos) y ambos deben ser del tipo expresión. Si no se define el contenido del operador binario, entonces los operandos pueden tener cualquier valor. Así:

Binary [kind: MINUS]

significa cualquier substracción. El mismo árbol podría ser definido de la siguiente manera:

Binary [kind: MINUS] {

Expression,

Expression

}

Como no se especifica ningún atributo de “Expression”, entonces “Expression” representa cualquier tipo de expresión. En aquellas posiciones en las que corresponde un árbol, cualquier subclase de un árbol puede ser utilizado. Por ejemplo, los operandos de “Binary” pueden ser cualquiera de las subclases de “Expression”.

Binary [kind: MULTIPLY] {

Identifier,

Literal [kind: INT\_LITERAL, value: 0]

}

La jerarquía de los árboles es la misma que en el compilador para Java de Sun.

Algunos atributos pueden tener más de un valor. En dichos casos, la lista de valores se define separando los elementos con el carácter “|”. Por ejemplo:

Binary [kind: PLUS | MINUS]

indica bien una suma o una substracción.

Cada árbol en el patron puede tener el atributo “id”. El valor de este atributo debe ser único en una regla dada y se utiliza para referirse a un árbol en la transformación. Por ejemplo:

Assignment {

Identifier [id: p],

Literal [kind: NULL\_LITERAL]

} ->

Assignment {

Identifier [ref: p],

Literal [kind: INT\_LITERAL, value: 0]

}

reescribe “p = null” a “p = 0” donde “p” se refiere a cualquier identificador.

Las referencias a los atributos se marcan con la almohadilla “#”. Por ejemplo, b#kind apunta al atributo “kind” de “b”. La referencia a un atributo se puede utilizar en las transformaciones como nuevo valor del atributo. Por ejemplo, para cambiar el orden de los operandos en una división se definiría de la siguiente manera:

Binary [id: b, kind: DIVIDE | REMAINDER] {

Identifier [id: x],

Identifier [id: y]

} ->

Binary [kind: b#kind] {

Identifier [ref: y],

Identifier [ref: x]

}

El valor especial “null” indica que el árbol no debe existir. Por ejempo, la regla siguiente añade un valor inicial a las declaraciones de variable:

Variable [id: v] {

Modifiers [id: m],

PrimitiveType [primitiveTypeKind: INT],

null

} ->

Variable [name: v#name] {

Modifiers [ref: m],

PrimitiveType [primitiveTypeKind: INT],

Literal [kind: INT\_LITERAL, value: 42]

}

#### Listas

Las listas utilizan la misma sintaxis que las listas genéricas en Java. “List<T>” es una lista de elementos del tipo “T”.

List<Expression>

Es un ejemplo de lista de expresiones. Una lista puede ser utilizada como parte de otro árbol o como nivel raíz. Para transformar un bloque vacío en un bloque con una sentencia vacía se haría de la siguiente manera:

Block {

List<Statement> { }

} ->

Block {

List<Statement> {

EmptyStatement

}

}

Y la regla que transforma una lista de cadenas en otra lista de cadenas distintas se definiría de la siguiente manera:

List<Expression> {

Literal [kind: STRING\_LITERAL, value: "London"],

Literal [kind: STRING\_LITERAL, value: "Paris"]

} ->

List<Expression> {

Literal [kind: STRING\_LITERAL, value: "Prague"]

}

El atributo “size” especifíca el número de elementos.

List<Literal> [size: 2]

->

List<Literal> {

Literal [kind: CHAR\_LITERAL, value: '@']

}

Mientras que “minSize” y “maxSize” permiten especificar un rango de elementos:

List<Expression> [minSize: 2, maxSize: 3]

El valor “\*” para el atributo “maxSize” indica sin límite máximo.

List<Catch> [minSize: 2, maxSize: \*]

->

List<Catch> {

Catch {

Variable [name: "e"] {

Modifiers {

List<Annotation> { },

Set<Modifier> { }

},

Identifier [name: "Exception"],

null

},

Block {

List<Statement> { }

}

}

}

#### NoneOf

“NoneOf” indica que el árbol puede ser cualquier cosa excepto los árboles señalados. La regla a continuación reescribe la asignación sólo si el nombre de la variable no es “x” ni “y”.

Assignment {

NoneOf<Expression> [id: i] {

Identifier [name: "x"],

Identifier [name: "y"]

},

Literal [kind: DOUBLE\_LITERAL, value: 3.14]

} ->

Assignment {

Expression [ref: i],

MemberSelect [identifier: "PI"] {

Identifier [name: "Math"]

}

}