Índice

1. Metodología

1.1. Implementación de las nociones semánticas

1.1.1. Sintaxis concreta y abstracta

La sintaxis concreta de un lenguaje, es la interfaz que utilizan los usuarios para crear programas; la sintaxis abstracta es la representación semántica de dicho lenguaje.

La sintaxis abstracta, es una estructura de datos o modelo, que actúa como una API para aplicar herramientas de validación, transformación y generación de código.

Existen dos estrategias para el desarrollo de las sintaxis concretas y abstractas:

Iniciar por la definición de la sintaxis concreta: la sintaxis abstracta es derivada a partir de la sintaxis concreta, ya sea de forma automática o con la ayuda de "hints" en la especificación de la sintaxis concreta.

Iniciar por la definición de la sintaxis abstracta: la sintaxis concreta se define a partir de las especificaciones de la sintaxis abstracta.

Para la creación de la sintaxis abstracta, existen dos técnicas:

1. "Parsers": utilizan una definición formal llamada gramática, para poder derivar la sintaxis abstracta a partir de la sintaxis concreta.
2. Proyección: la sintaxis abstracta es generada a partir de acciones realizadas por el usuario en un editor. La sintaxis concreta es dinámica y es generada a partir de los cambios ocurridos en la sintaxis abstracta. La proyección no utiliza gramática.

Los compiladores tradicionales utilizan "parsers" creados en forma manual, la consecuencia de esto son programas grandes y monolíticos.

"Parser Generator" es una técnica mencionada por (Fowler 2010) en donde el parser se genera en forma automática en base a la especificación de una gramática. Este es el enfoque que utilizan la mayoría de los compiladores actuales.

La siguiente tabla muestra, las diferencias entre ambos enfoques:

Enfoque	Ventajas	Desventajas
Generador de "parsers"	El "parser" se genera automáticamente. El desarrollador un experto en la creación de "par	No ofrece el mismo rendimiento en comparación a un "parser" no tiene que ser hecho a la medida de un experto tecnología de sers"
"Parser" manual	Es más rápido	El desarrollador debe ser un experto en la creación de lenguajes

1.2. Xtext

Xtext es una herramienta que sirve para implementar lenguajes de programación (DSLs o GPLs), dentro del marco de trabajo de Eclipse. Xtext es un generador de "parsers" y ofrece toda la infraestructura para definir restricciones, manejo de tipos, "scoping", generación de código, interpretes, "quickfixes" y todas las

características de un lenguaje de programación moderno.

Para la generación del "parser", Xtext se apoya en ANTLR (antlr) (pronunciado: "Antler, ANother Tool for Language Recognition"). La definición de la gramática en ANTLR se realiza en un solo archivo.

Un **modelos semánticos** (Fowler 2010) es un modelo de objetos en memoria, que un DSL debe popular. El modelo semántico y la sintaxis abstracta son términos equivalentes. Xtext se apoya en EMF Ecore para persistir el modelo semántico en memoria.

EMF Ecore

"Eclipse Modeling Framework" (EMF) (Steinberg et al. 2009), es el corazón de las herramientas de modelado de Eclipse. EMF expone una amplia variedad de servicios y herramientas para persistir, editar y transformar modelos.

Para la definición de meta-modelos, el EMF utiliza Ecore. Las principales características de Ecore son:

- Diversas representaciones (código Java, XML, UML) del meta-modelo.
- Mecanismos automáticos de transformación entre las diferentes representaciones.
- Clases generadas por EMF.Edit que sirven como punto de partida para la implementación de código y el desarrollo de una aplicación.
- Un editor genérico para la creación, visualización y edición de modelos.
- Una API de consultas para la obtención de la estructura del meta-modelo.
- Mecanismos de reflexión para la manipulación de instancias del meta-modelo.

Los modelos básicos de Ecore son:

1. *EClass*: representa los modelos del lenguaje (elementos de la sintaxis abstracta).
2. *EAtrribute*: describe el estado de un EClass.
3. *EDataType*: indica el tipo de un atributo. Un tipo de dato puede ser primitivo o un tipo de objetos como ser `java.util.Date`.
4. *EReference*: representa asociaciones entre EClases. En forma opcional los EReferences pueden tener semántica de contenedores.
5. *EObject*, representa instancias de EClasses (por ejemplo nodos AST). Cada EObject puede ser contenido de al menos una instancia de EReference.
6. *EPackage*: agrupa clases y tipos de datos relacionados.

1.2.1. Sintaxis y Gramática

La gramática (piedra angular de Xtext) es la definición formal de la sintaxis concreta. El objetivo de la gramática es indicar como se mapea la sintaxis concreta con la sintaxis abstracta representada en memoria. El modelos es contruído dinámicamente por el "parser" cuando se consume una entrada válida de texto.

Declaración del lenguaje

El lenguaje se declara en el encabezado de la gramática. Ejemplo:

```
grammar cl.pleiad.ram.Textram
   with org.eclipse.xtext.common.Terminals
```

En el encabezado también se indica la reutilización de otras gramáticas, por ejemplo: `org.eclipse.xtext.common.Terminals`

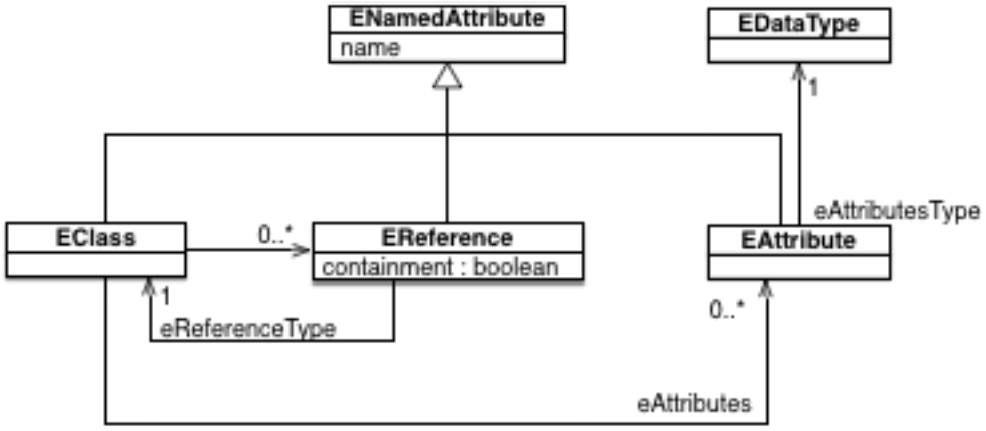

Figura 1: Meta-modelo básico de Ecore

1.2.2. Reglas Terminales

Terminales

"Lexing" es el proceso de creación de "tokens" mediante el consumo de una secuencia de caracteres de entrada. Los "tokens" son uno o más caracteres que corresponden a una regla terminal o "keyword". Por ejemplo `ID` es una regla terminal, definida en `org.eclipse.xtext.common.Terminals`, su implementación es la siguiente:

```
terminal ID:
  ('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

El orden de las reglas "terminal" es importante para la gramática, debido a que se sobre-escriben unas a otras.

Tipos de retorno

Cada regla terminal, retorna un EDataType de Ecore (por defecto ecore::EString). A continuación se presenta la definición de una regla llamada *INT* que retorna ecore::EInt:

```
terminal INT returns
  ecore::EInt: ('0'..'9')+`
```

Expresiones Backus-Naur extendendidas

Las reglas terminales son descritas por medio de expresiones "Extended Backus-Naur Form" (EBNF).

Las cardinalidades de EBNF son:

1. Exactamente uno (sin operador, configurado por defecto)
2. Uno o más (operador `?`)
3. Cualquiera (cero o más, operador `*`)
4. Uno o muchos (operador `+`)

Rango de caracteres

Un rango de caracteres puede ser declarado con el operador `..`:

```
terminal INT returns
  ecore::EInt: ('0'..'9')+;
```

En este caso INT retorna un o más caracteres en '0' y '9'

"Wildcards"

La gramática de Xtext, soporta el uso de "wildcards". Por ejemplo el `.` (punto), permite el uso de cualquier carácter:

```
terminal FOO: 'f' . 'o';
```

Esta regla permite las siguientes expresiones: *foo, f0o, f_o*.

"Until Token"

Indica que todo debe ser consumido hasta la aparición de un "token":

```
terminal ML_COMMENT: '/*' -> '*/';
```

El ejemplo anterior es un comentario regular de Java.

"Negated Token"

Todos los "tokens" pueden ser negados, mediante el uso del signo de admiración:

```
terminal BETWEEN_HASHES: '#' (!'#') * '#';
```

Llamada de reglas

Las reglas pueden referenciar otras reglas:

```
terminal DOUBLE : INT ',' INT;
```

En este caso la regla `DOUBLE` está llamando a la regla `INT`.

Alternativas

Permiten definir una lista de opciones válidas. Por ejemplo el la regla del espacio en blanco, utiliza una alternativa de la siguiente forma:

```
terminal WS : (' '|'\t'|'\r'|'\n')+;
```

Grupos

Se llama grupo a la secuencia de "tokens" que se definen uno detrás de otro:

```
terminal ASCII : '0x' ('0'..'7') ('0'..'9'|'A'..'F');
```

El ejemplo anterior, representa 2 dígitos hexadecimales de caracteres ASCII.

Fragmentos terminales

Debido a que las reglas terminales son usadas en un contexto sin estado, no es fácil reutilizar partes de su definición; para resolver este problema se utilizan los fragmentos.

Los fragmentos permiten los mismos elementos EBNF como reglas terminales, pero no son consumidas por el "lexer". En su lugar son definidas para ser usadas por otras reglas terminales:

```
terminal fragment ESCAPED_CHAR : '\\' ('n'|'t'|'r'|'\\');
terminal STRING :
  '"' ( ESCAPED_CHAR | !('\\'|'"') )* '"' |
  "'" ( ESCAPED_CHAR | !('\\'|"'") )* "'"
;
```

Fin de archivo (EOF)

El "token" fin de archivo, puede ser usado para describir que el final del flujo de entrada es valido en un cierto punto de una regla terminal:

```
terminal UNCLOSED STRING : '"' (!'"')* EOF;
```

1.2.3. "Parser"

Las reglas del "parser" son vistas como un plan para la creación de EObjects que forman el modelo semántico. Los elementos que están diponibles en las reglas del "parser" son:

1. Grupos
2. Alternativas
3. Palabras claves.
4. Reglas de llamada.

A continuación, se describe como algunas expresiones dan "hints" para la construcción directa del AST:

Asignaciones

La creación de instancias se realiza en las asignaciones. El tipo de la entidad asignada es un EClass inferido desde el lado derecho de la asignación. El nombre del EClass puede ser definido explícitamente o implícitamente utilizando el nombre de la regla.

A continuación se presenta una regla que retorna un EClass llamado `Aspect`:

```
Aspect:
    'aspect'
    name=ID
    '{'
        structure=Structure
    '}';
```

La declaración sintáctica de `Aspect` inicia con la palabra clave "aspect" seguido de la asignación `name=ID`; el lado derecho de dicha asignación puede ser una llamada a otra regla, una palabra clave, una referencia cruzada o una alternativa. Los tipos involucrados en la asignación deben ser compatibles.

Operadores de asignación

1. El carácter `=`: es usado cuando se espera un solo elemento.
2. El carácter `+=`: espera una lista con múltiples valores y adiciona el valor a dicha lista.
3. El carácter `?=`: espera un elemento de tipo EBoolean y asigna `true` si el elemento del lado derecho fue consumido.

1.2.4. Referencias cruzadas

Xtext permite la definición de referencias cruzadas dentro de la gramática. Ejemplo:

```
Transition :
    event=[Event] '=>' state=[State] ;
```

La regla "Transition" está formada por dos referencias cruzadas, una apuntanto a "event" y otra a "state". El texto entre corchetes no se refiere a otra regla, se refiere a un EClass.

En Ecore, la clase EReference tiene una propiedad que indica si la referencia es contenedora o no. Una referencia contenedora es un apuntador a un objeto. Un objeto puede tener un solo contenedor.

Las referencias cruzadas son referencias no contenedoras. La determinación de alcance ("scoping") se encarga de resolver este tipo de referencia mediante el uso de un identificador único ("index") que almacena toda la información del objeto. La resolución de las referencias cruzadas se hace en una etapa llamada enlazamiento "linking".

1.2.5. Determinación de alcance y enlazamiento

El **enlazamiento** resuelve las referencias de los símbolos de un lenguaje basado en "parser". La resolución de referencias es por medio de la convención de nombres.

La **determinación de alcance**, es el mecanismo principal detrás de la visibilidad y la resolución de referencias cruzadas. Desde el momento en que el DSL necesita estructurar el código, se necesita una implementación de la definición de alcance.

Por lo general, le determinación de alcance de una referencia cruzada, depende de:

1. El espacio de nombres en donde viven los elementos.
2. La ubicación dentro de la estructura del sitio que contiene la referencia cruzada.
3. Algún aspecto, que no necesariamente es estructural por naturaleza.

La determinación de alcance, también puede ser de ayuda para:

1. Popular el menú del "code-completion" del IDE cuando el usuario presiona `Ctrl-Space` en el sitio referenciado.
2. Validar una referencia existente.

Xtext proporciona una API de Java para implementar la determinación de alcance. Con dicha API un desarrollador puede definir varios niveles de alcance:

1. Alcance simple y local.
2. Alcance anidado.
3. Alcance global.

1.2.6. Restricciones

No todos los programas que están conformes con el meta-modelo son válidos. La definición de un lenguaje incluye restrucciones que no pueden ser expresadas solamente por el meta-modelo.

Las restricciones son condiciones "Boolean" que deben ser evaluadas como verdaderas, para poder indicar la validez de un modelo. Un mensaje de error debe ser reportado si la expresión a evaluar es falsa.

Se pueden distinguir dos tipos de restricciones:

1. Restricciones que exigen que los elementos estén bien formados. Ejemplo: unicidad de los nombres en una lista de elementos.
2. Sistema de tipos: las reglas de los sistemas de tipos, son diferentes porque verifican la correcta definición de los tipos dentro de un programa. Por ejemplo, el sistema de tipos se asegura que no se pueda asignar un `float` a un `int`.

Las restricciones pueden ser implementadas por cualquier lenguaje o "framework" que pueda consultar un modelo y reportar errores al usuario.

La definición eficiente de restricciones, debe contemplar:

1. Instrucciones para navegar y filtrar el modelo.
2. Soporte a "higher-order-functions", para la escritura de algoritmos genéricos y estrategias transversales.
3. Definición declarativa de las restricciones, con asociación a los conceptos del lenguaje (o patrones estructurales).

1.2.7. Sistema de tipos

Los sistemas de tipos, implementan cálculo de tipos y verificación de tipos.

Conceptos básico del sistema de tipos

Para introducir los conceptos básicos del sistema de tipos, se hará uso del AST de la Figura 3.

El AST de la Figura 3, contiene cajas que representan instancias de los conceptos del lenguaje, las líneas sólidas representan contenedores y las líneas punteadas representan referencias-cruzadas.

Un sistema de tipos, debe contemplar:

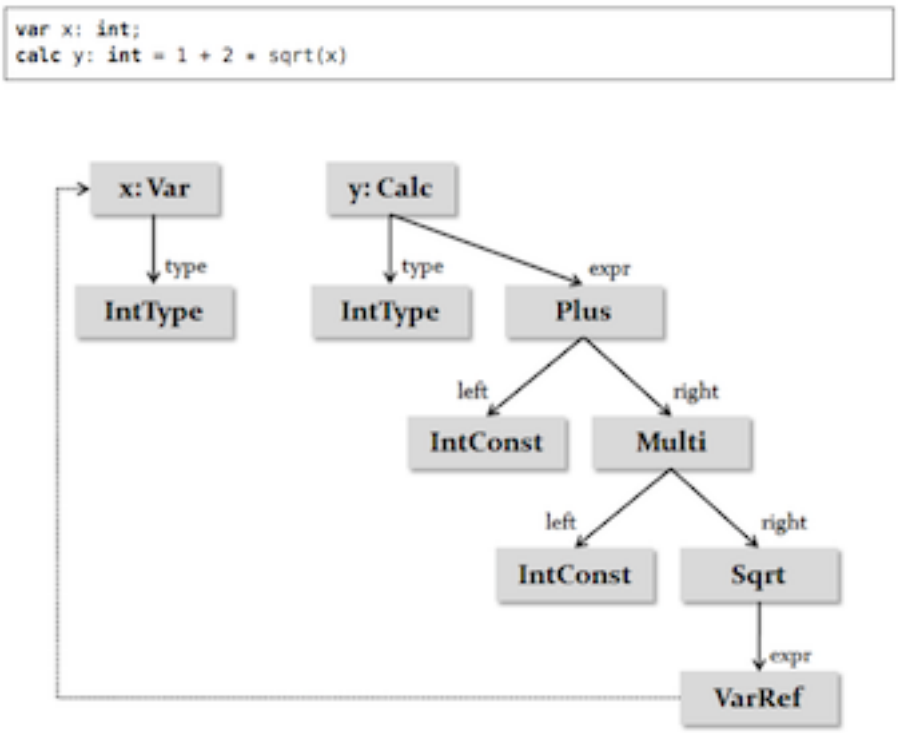

Figura 2: "Figura 3. AST de un fragmento de código. Copiado de (Voelter et al. 2013)"

1. *Declaración de tipos fijos*: algunos elementos de los programas tienen tipos fijos. Los mismos no deben ser derivados o calculados. Por lo general permanecen igual y son definidos con anticipación. Algunos ejemplos son: la constante entera `IntConst` (cuyo tipo es `IntType`), el concepto de raíz cuadrada `sqrt` (cuyo tipo es `double`), así como las declaraciones de tipos por si mismas (el tipo de `IntType` es `IntType`, el tipo de `DoubleType` es `Double-Type`).

2. *Derivación de tipos*: para algunos elementos del programa, el tipo debe ser derivado a partir de los tipos de otros elementos. Por ejemplo, el tipo `VarRef` (variable de referencia) es del tipo de la variable referenciada. El tipo de la variable es del tipo de su tipo declarado. Por ejemplo, el tipo de `x` y de la referencia a `x` es `IntType`.

3. *Cálculo de tipos comunes*: muchos de los sistemas de tipo, tienen alguna especie de jerarquía de tipos. En el ejemplo, `IntType` es un sub-tipo de `DoubleType` (por tanto, `IntType` puede ser usado, siempre que `DoubleType` sea esperado). El sistema de tipos debe soportar la especificación de relación de subtipos.

4. *Verificación de tipos*: finalmente, un sistema de tipo debe verificar los errores de tipo y reportarlos al usuario. Ejemplo: un error de tipo ocurre si una variable `DoubleType` es asignada a una variable `IntType`.

Inferencia del modelo de Ecore

Como se mencionó en [??] el AST de Xtext es Ecore. Los elementos de Ecore son inferidos a partir de las reglas definidas en la gramática de Xtext. La tabla es un resumen de la inferencia hecha por Xtext a los modelos de Ecore.

| Tipo de regla | Tipo de retorno | Descripción | |:==========+==========+==========+ | Parser | EPackage | Creado después de la directiva `generate`. El nombre del paquete se forma a partir de sus parámetros y de su `nsUri`, opcionalmente se puede utilizar un alias. | +————+————-+———-+———+

| Enum | Enum | Se crea por reglas que utilizan enumeración | +————+————-+———-+———+ | Tipo de datos | EDataType (por defecto es EString) | se crea a partir del tipo de dato de cada regla terminal o de una regla de tipo de dato | +————+————-+———-+———+ | Parser | EAttribute | Es `EBoolean` si se utiliza el operador `?=` si se utilizan los operadores `=` o `+=` en las reglas terminales, se crea un atributo con un tipo igual al tipo de retorno de la clase llamada | +———— +————-+———-+———+ | Parser | EReference | se crea un `EReference` si hay una regla que llama a otra regla, también se crea una `EReference` por cada asignación de una acción; en ambos casos el tipo será igual al tipo de retorno de la regla llamada. | +————+————-+———-+———+

Tabla inferencia del modelo Ecore {#tablaInferenciaEcore}

Tipo de regla	Tipo de retorno	Descripción
Parser	EPackage	Creado después de la directiva `generate`. El nombre del paquete se
Enum	Enum	Se crea por reglas que utilizan enumeración
Tipo de datos	EDataType (por defecto es EString)	EDataType (por defecto es EString)
Parser	EAttribute	Es `EBoolean` si se utiliza el operador `?=` si se utilizan los operadores
Parser	EReference	se crea un `EReference` si hay una regla que llama a otra regla, tamb

1.3. Transformaciones

La transformación se refiere a la creación de un artefacto desde un modelo semántico. Puede ser de dos tipos:

1. Modelo a texto (M2T): los modelos son convertidos a texto (usualmente código fuente, XML, archivos de configuración, etc).
2. Modelo a modelo (M2M): los modelos son transformados en otros modelos (conversión de un modelo semántico a otro equivalente).

Xtend es un lenguaje de programación inspirado en Java, pero que remueve su ruido sintáctico. Xtend puede ser usado como una alternativa de la transformación M2T. Las principales características de Xtext son:

- Métodos de extensión.
- Inferencia de tipos.
- Expresions lambda.
- Múltiples "dispatch" para invocación polimorfica de métodos.
- "Closures".
- Expresiones de plantillas (usado para la generación de código).
- Facilidad para navegar y consultar modelos, gracias a abstracciones funcionales ("high-order functions").
- Todo es una expresión y no una sentencia (ejemplo: dentro de una función, la última expresión es la expresión de retorno, por tanto es opcional explicitar la palabra clave "return").

antlr. "ANTLR (ANother Tool for Language Recognition)." http://www.antlr.org.

Fowler, Martin. 2010. *Domain Specific Languages*. 1st ed. Addison-Wesley Professional.

Steinberg, David, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0*. 2nd ed. Addison-Wesley Professional.

Voelter, Markus, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-specific Languages*. dslbook.org.