Theoria, Vol. 13: 39-49, 2004 ISSN 0717-196X

Artículo

ANÁLISIS SEMÁNTICO DE PROGRAMAS ESCRITOS EN JAVA

SEMANTIC ANALYSIS OF PROGRAMS WRITTEN IN JAVA

Luis Gajardo Díaz[1] y **Luis Mateu Brulé**[2]

[1]Depto. de Auditoría e Informática. Facultad de Ciencias Empresariales. Universidad del Bío-Bío. Avenida La Castilla s/n Chillán, Chile. Fono (42) 203414. Fax (42) 203421. E-mail: lgajardo@ubiobio.cl.
[2]Depto. de Ciencias de la Computación. Facultad de Ingeniería. Universidad de Chile. Santiago, Chile.

RESUMEN

Actualmente los lenguajes orientados a objetos son ampliamente utilizados en todas las áreas de desarrollo de software. Uno de sus principales exponentes, el lenguaje java, permite la construcción de programas de manera sencilla y elegante. Esto ha provocado que java sea muy utilizado en investigación, sirviendo de lenguaje base sobre el cual implementar nuevas mejoras, funcionalidades o incluso dialectos. Generalmente estas mejoras requieren de cambios a nivel de diseño del lenguaje, produciendo como resultado un analizador semántico de programas con estas nuevas características. Dado que no existen herramientas que apoyen el análisis semántico de programas escritos en java, es común que los investigadores tengan que reescribir gran parte de la funcionalidad del compilador de java para agregar sólo unos pocos cambios a nivel semántico. La construcción de un analizador de programas en java involucra pasos como análisis léxico, análisis sintáctico y análisis semántico. Las dos primeras etapas son abordadas por lo general por medio de la utilización de herramientas que generan de manera automática analizadores sintácticos. Estos analizadores sintácticos construyen una estructura tipo árbol que representa el programa fuente analizado. El problema que surge aquí es que estos árboles, al ser construidos automáticamente, son de un tamaño excesivamente grande ya que contienen más información de la que realmente se necesita almacenar. La manipulación de estos árboles es compleja, tediosa y es ineficiente en cuanto a espacio de almacenamiento y tiempo empleado en recorrer sus nodos. Nuestro objetivo entonces es realizar un diseño ad-hoc que permita trabajar con un árbol mucho más pequeño y manejable el cual puede ser fácilmente modificado para incluir nuevas funcionalidades al lenguaje. El resultado de este trabajo es un analizador semántico de programas escritos en lenguaje java, diseñado en capas de manera de permitir la incorporación de cambios al nivel que se necesite sin reprogramar el analizador completo desde cero.

Palabras claves: Análisis de tipos, análisis semántico, árbol de sintaxis abstracta, java.

ABSTRACT

Actually, object oriented programming languages are used in all software development areas widely. One of the most known languages is Java, this programming language allows developing programs in a simple and elegant style. Java is used in researching very much because this fact, new facilities are implemented or even new dialects are created on it. These improvements require changes to language design level generally; the result is a program semantic analyzer with these new characteristics. Because there are no tools supporting semantic analysis of Java programs, researchers have to rewrite the most of functionality of Java Compiler just for adding few semantic changes. Building a Java program analyzer involves lexical analysis, syntax analysis, semantic analysis, etc. Two first steps are usually solved using tools that generate syntax analyzers automatically. These analyzers build a tree, which represents the traced source program. But with this solution appears a problem; the automatically built trees are huge because they contain more information that it is necessary to store. Besides, working with these trees is complex, tedious, and inefficient in storage space and time for visiting their nodes. So our goal is to design a tool that works with a smaller and more manage-

able tree, a tree that may be modified easily when new facilities are added to the language. The result of this work is a Java program semantic analyzer designed in levels so changes may be incorporated in any desired level without reprogramming all the analyzer.

KEYWORDS: Types analysis, semantic analysis, abstract syntax tree, java.

Recepción: 05/05/04. Revisión: 27/08/04. Aprobación: 22/10/04

1. INTRODUCCIÓN

En la actualidad los lenguajes orientados a objetos son utilizados en una amplia gama de áreas, ya que permiten modelar procesos y entidades de una manera natural y sencilla. El lenguaje java está basado en el paradigma de la orientación a objetos desde su concepción, lo que ha dado origen a un lenguaje que posee construcciones sintácticas limpias y elegantes. Esto ha provocado que java sea muy utilizado para educación e investigación.

Es común encontrar investigadores que desarrollan nuevas funcionalidades, mejoras o incluso nuevos dialectos de Java por medio de modificaciones a su gramática original. Nuestro estudio se centra principalmente en este punto, ya que la complejidad de escribir nuevas funcionalidades o herramientas sintácticas significa grandes esfuerzos y tiempo de programación. Pese a que existen herramientas que realizan de manera automática el análisis sintáctico y la generación de un árbol de análisis, éstas no lo hacen de manera eficiente y sencilla provocando demoras en las operaciones realizadas sobre el árbol y complejidad en la mantención del mismo.

Es por este motivo que nosotros proponemos un diseño ad-hoc que permite trabajar con un árbol mucho más pequeño y manejable el cual puede ser fácilmente modificado para incluir nuevas funcionalidades al lenguaje.

2. ANÁLISIS SINTÁCTICO

El análisis sintáctico, también llamado *parsing*, consiste en agrupar los *tokens* del programa fuente formando frases gramaticales las cuales son usadas para comprobar si se deriva a partir de la gramática del lenguaje. Como resultado de este proceso se genera, de una forma explícita o implícita, un árbol de análisis.

La literatura clásica en el área de los compiladores, como Aho *et al.* (1990), describe dos tipos principales de analizadores sintácticos utilizados hoy en día: los métodos descendentes y los métodos ascendentes. Es importante mencionar que estos métodos sólo trabajan con subclases de gramáticas y no con todas las existentes.

Los analizadores descendentes, también llamados *top-down*, construyen el árbol sintáctico desde arriba (la raíz) hacia abajo (las hojas). Los analizadores ascendentes o *bottom-up* construyen el árbol comenzando por las hojas hasta concluir en la raíz.

Los métodos descendentes son más populares, ya que se pueden implementar fácilmente a mano. Sin embargo, los métodos ascendentes pueden manejar una gama más amplia de gramáticas. De forma que las herramientas de software que generan directamente analizadores sintácticos suelen utilizar los métodos de análisis ascendente.

Nuestro trabajo realiza el análisis sintáctico de un programa fuente en java apoyado en herramientas automáticas, las cuales serán comentadas más adelante. Sin embargo, es importante destacar que el resultado de este análisis es un árbol, el cual posee un diseño creado íntegramente por nosotros, no por las herramientas automáticas, y que permitirá facilitar las etapas posteriores. La Figura 1 muestra el diagrama que representa el análisis sintáctico.

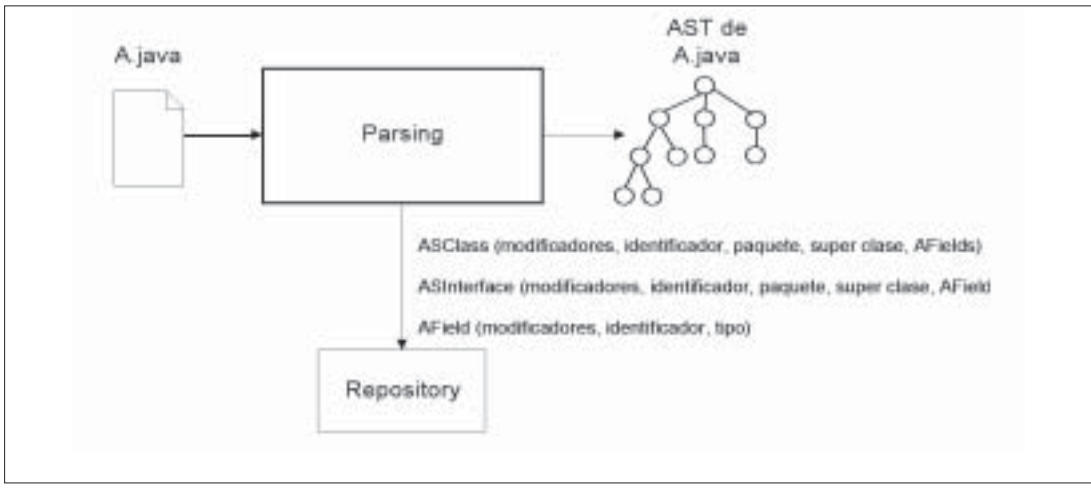

Figura 1. Análisis sintáctico o Parsing.

Como el objetivo principal de este trabajo se orienta al análisis semántico, en esta etapa reuniremos información útil para llevarlo a cabo, con esto reducimos el número de recorridos al árbol en la etapa siguiente. La información recolectada consiste en declaraciones de clases, interfaces y atributos, las cuales serán almacenadas en una estructura denominada *Repository* que consiste en un diccionario que asocia el nombre completo de la clase o interfaz con un descriptor (un objeto ASClass o ASInterfaz respectivamente). Además existe sólo un repositorio durante todo el proceso de análisis global, el cual se va completando en cada etapa. Finalmente como salida de esta etapa se obtiene una representación del programa en forma de un Arbol de Sintaxis Abstracta o AST el cual será utilizado para la etapa siguiente.

2.1. Herramientas para generar analizadores sintácticos

Las herramientas que permiten generar analizadores sintácticos dan la posibilidad al programador de trabajar a un nivel de abstracción más elevado que el que deberían usar si las escribieran a mano. Existen 2 críticas clásicas sobre los analizadores sintácticos creados automáticamente: 1) Los *parsers* generados automáticamente son lentos y usan mucha memoria. 2) El reporte de errores no es tan exacto.

Generalmente las etapas de análisis léxico y sintáctico son realizadas por distintas herramientas, esto permite independizar las funciones con lo cual podríamos lograr mayor flexibilidad. Las herramientas de análisis sintáctico se basan en uno de los dos métodos explicados anteriormente, descendente o ascendente. Entre las que utilizan métodos descendentes podemos mencionar JavaCC (Java Compiler Compiler), desarrollada por Sun Microsystems (2000) y ANTLR (Another Tool for Language Recognition), escrita por Parr (1995). Por el contrario, entre las que utilizan métodos ascendentes podemos encontrar SableCC, desarrollada por Gagnon y Hendren (1998), y CUP (Constructor of Useful Parsers), escrita por Hudson (1997).

Con el objeto de justificar la elección de la herramienta adecuada se presenta, en la Tabla I, una comparación de las características de cada herramienta.

Los métodos de análisis ascendente son mejores porque sus gramáticas no requieren

TABLA I. Comparación entre las principales características de las herramientas para generar analizadores sibntácticos.

Herramientas vs características	CUP	SableCC	JavaCC	ANTLR
Método de análisis ascendente	√	√	X	X
Integración	X	√	√	√
Reglas de desambiguación	√	X	√	√
Asociatividad de acciones a producciones	√	X	√	√
Soporta código en lenguaje Java	√	√	√	√
Disponibilidad del código fuente	√	√	X	√
Su código puede ser modificado por el programador	√	√	X	√
Documentación y soporte	√	√	√	√

demasiadas modificaciones para que puedan ser analizadas como en el caso de los métodos descendentes.

En lo que se refiere a integración, JavaCC posee una herramienta denominada JTB (Wang *et al.*, 1997), la cual permite la generación de árboles de derivación. De las cuatro herramientas analizadas, CUP no apoya de manera automática la generación de un árbol de derivación. La integración de varios utilitarios bajo una sola herramienta es muy favorable, sobre todo cuando se da la libertad al usuario para utilizarlas o no. SableCC posee una excelente integración entre sus módulos, ya que permite la generación de analizadores léxicos y sintácticos, construcción automática del árbol de derivación y sus visitadores. Esta ventaja se transforma en desventaja a la hora de querer modificar el árbol de derivación debido a que es demasiado rígido, lo que hace compleja su manipulación. El problema de los árboles de derivación demasiado grandes es común a todas las herramientas que generan automáticamente estos árboles a partir de la gramática del programa. Por eso, en este trabajo no se recurrirá a estas herramientas, en cambio se diseñará un árbol que permitirá procesar en forma eficiente y más comprensible los programas, a pesar de que esto significará un trabajo adicional al rediseñar este árbol.

Con respecto a la asociatividad de las acciones semánticas con las producciones de la gramática, en JavaCC y ANTLR ésta se pierde. SableCC por su parte posee un esquema totalmente orientado a objeto, ya que el programador debe escribir el código de las acciones semánticas directamente dentro de las clases generadas por el framework. Según Appel (2002), esto provoca dificultades en la lectura y mantención del analizador. En este sentido, CUP al igual que YACC (Johnson, 1975), herramienta de la cual desciende, asocia las acciones semánticas directamente con las producciones de la gramática, lo que facilita su mantención.

La disponibilidad de cada herramienta y las posibilidades de modificación por parte del programador están relacionadas con el copyright. Todas las herramientas analizadas son open source o GNU salvo JavaCC, la cual es una herramienta gratuita que no provee su código fuente. Todas poseen buen soporte y documentación, pero destaca SableCC.

2.2. Árboles de derivación vs árboles de sintaxis abstracta

Como ya mencionamos, la salida de la etapa de análisis sintáctico corresponde a una estructura de tipo jerárquica del programa que se analiza. El árbol resultante puede ser de derivación o de sintaxis abstracta.

Un árbol de derivación, también llamado *parse tree*, posee un diseño en el cual cada pro-

ducción de la gramática tiene su correspondiente objeto. Esto significa que el tamaño del árbol será proporcional al tamaño de la gramática. En lenguajes con gran cantidad de construcciones sintácticas como java esto se traduce en la utilización de una gran cantidad de memoria para almacenar el árbol.

En la gramática de java existen reglas sintácticas que sólo se incluyen con el propósito de desambiguar la gramática y no es necesario crear su correspondiente representación en el árbol de derivación, ya que no poseen información útil desde el punto de vista semántico.

A diferencia de un *parse tree*, un árbol de sintaxis abstracta (AST) posee un diseño acotado en el cual sólo se conservan aquellas clases que aportan un significado semántico, desechando aquellas que corresponden a reglas de desambiguación o parentización. Esto permite reducir considerablemente el tamaño del árbol resultante.

Otra ventaja producto de esta disminución en el tamaño, está dada por el tiempo utilizado en recorrerlo. Esto significa que las operaciones realizadas sobre ramas y nodos serán más rápidas que en el árbol de derivación.

Un ejemplo en el cual se aprecia esta diferencia se puede observar en la Figura 2 y Figura 3 con la expresión aritmética "5 + 3".

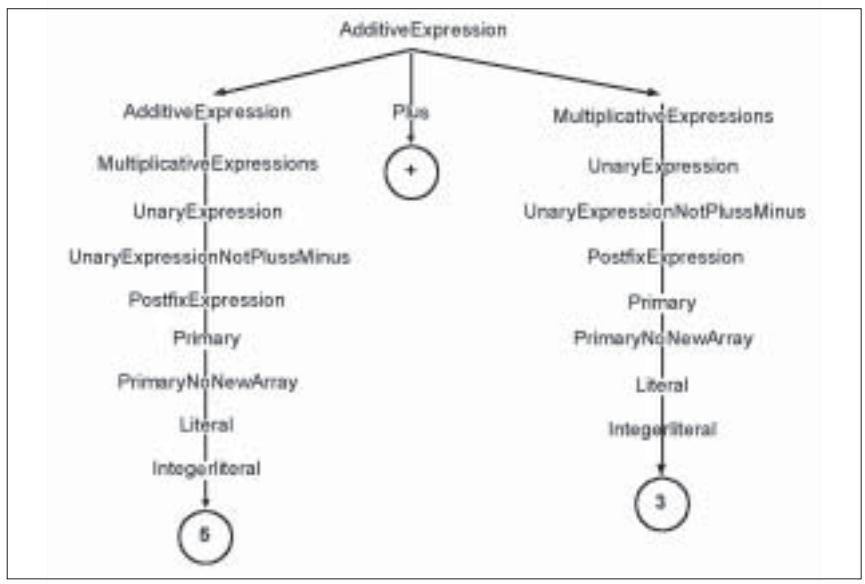

FIGURA 2. Arbol de derivación o parse tree.

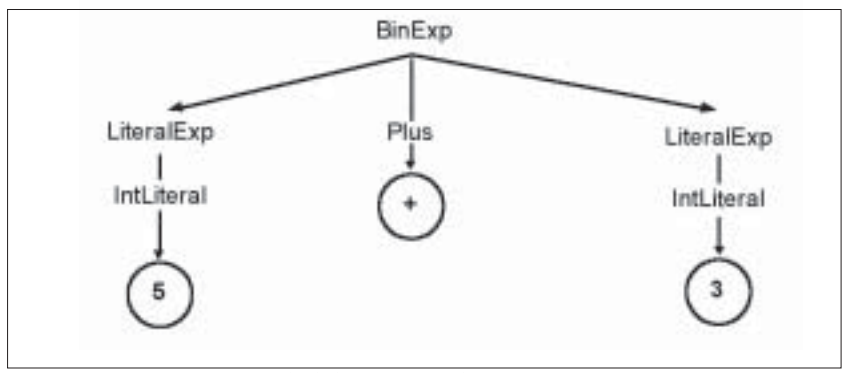

FIGURA 3. Arbol de Sintaxis Abstracta o AST.

Como desventaja, un AST es más difícil de diseñar que su contraparte, ya que requiere un detallado estudio de las construcciones sintácticas de la gramática a fin de no perder información relevante para la representación del programa.

En la Figura 4 se presenta el diseño desarrollado mediante un modelo ad-hoc para la generación de un AST. Para mayor comprensión por parte del lector, este modelo muestra solamente las categorías de clases fundamentales que pueden ser relacionadas con la gramática del lenguaje java, estas son: declaraciones (Dcl), sentencias (Stmt), expresiones (Exp), nombres e identificadores (Name), literales (Literal) y por último los tipos de datos (Type). Estos últimos son de una gran importancia debido a que en esta etapa son clasificados como "no conocidos" y luego en el análisis de tipos son reclasificados de manera más exacta mediante la utilización de las clases que identifican una clase abreviada (AbbrevClassType), una clase interna (InternalClassType) o una clase con su nombre de paquete incluido (PackageClassType).

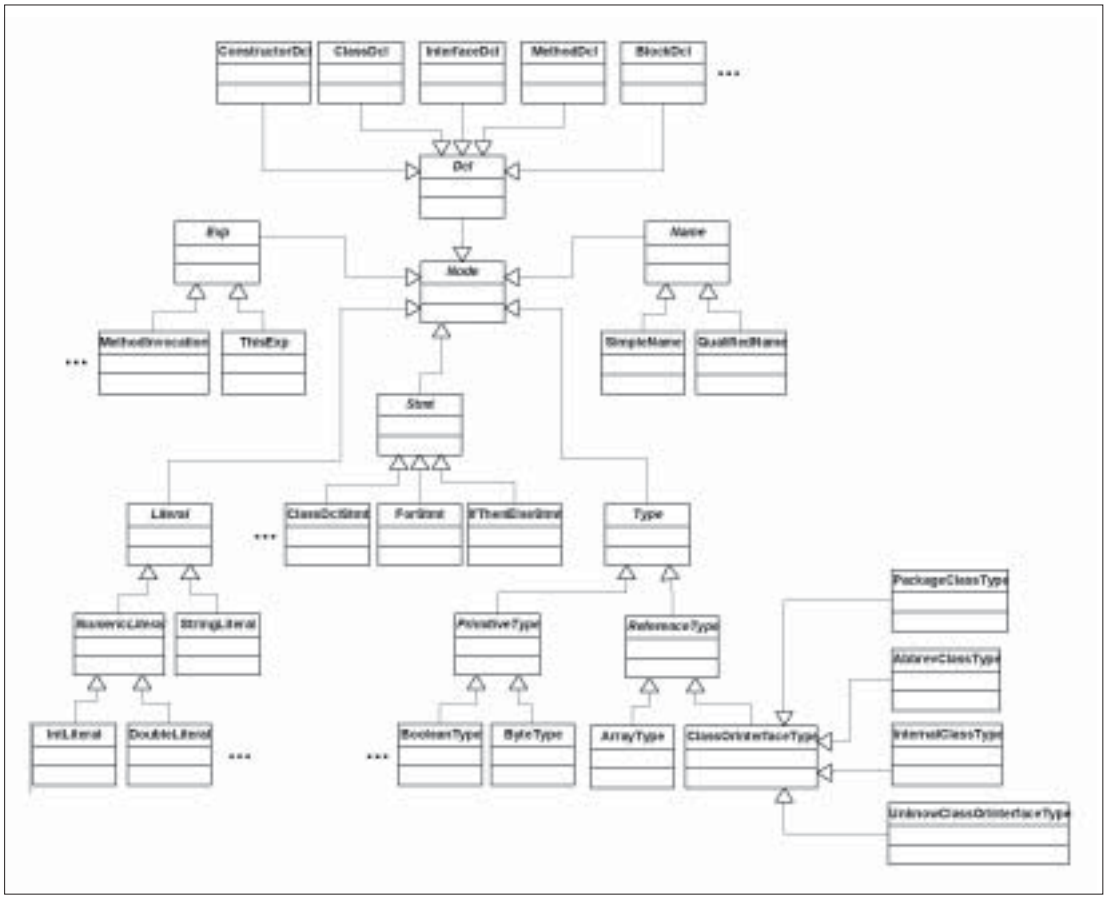

FIGURA 4. Modelo UML del árbol de sintaxis abstracta (AST).

2.3. Patrón de diseño Visitor

Para realizar análisis sobre el árbol abstracto es necesario recorrer sus nodos realizando diversas operaciones, esto toma una especial importancia ya que por cada operación que se desee efectuar se debe modificar todas o por lo menos la mayoría de las clases que conforman el AST. En un ambiente normal de desarrollo es común efectuar continuos cambios y mejoras. Tal dinamismo implica muchos cambios sobre las clases, lo que conlleva un excesivo esfuerzo en mantención y la complejidad de cada clase aumenta considerablemente. Esto hace necesario algún mecanismo que permita facilitar esta labor de manera sencilla y flexible.

El lenguaje de programación java, por ser un lenguaje orientado a objetos, permite emplear una técnica denominada patrones de diseño. La literatura en esta área (Fowler, 1996 y Cooper, 1998) aclara que los patrones de diseño no son una forma más de Análisis y Diseño Orientado a Objetos (ADOO), sino una recopilación de los esfuerzos hechos en el que sirve cómo perfecto complemento. Los patrones de diseño surgen de diseños cuya utilidad o necesidad se ha manifestado repetidamente en algunos proyectos y que han ido evolucionando hasta conseguir una cierta estabilidad y generalidad que los convierten en soluciones generalizadas y con ello altamente reutilizables.

El patrón de diseño llamado *Visitors* (Gamma *et al.*, 1994) consiste de una clase externa que es agregada al modelo con el cual trabajamos y que actúa sobre los datos contenidos en las instancias de las clases del modelo. Tradicionalmente este patrón es utilizado cuando i) Muchas operaciones distintas y sin relación entre sí deben ser realizadas sobre objetos en una estructura de objetos intentando evitar una "contaminación" de sus clases con estas operaciones y ii) La estructura de objetos de una aplicación es estable, pero se quieren añadir con frecuencia nuevas operaciones a los objetos de esta estructura.

El patrón de diseño *Visitors* no resulta demasiado útil en modelos que cambian continuamente su estructura, ya sea eliminando clases existente o agregando otras nuevas, puesto que el costo asociado sería la redefinición de la interface en todos los visitantes. En este caso sería probablemente mejor definir las operaciones dentro de cada clase del modelo.

En definitiva, dado que se necesita realizar variadas operaciones sobre un AST que posee un diseño bastante estático este patrón es una excelente opción, ya que nos permite definir nuevas operaciones sin modificar la estructura de los objetos sobre los que opera.

3. EL MODELO PROPUESTO

La Figura 5 muestra el modelo que nosotros hemos diseñado para permitir el análisis semántico de programas escritos en java. A continuación abordaremos la explicación de este modelo mediante un ejemplo concreto.

3.1. Ejemplo de análisis semántico: Análisis de tipos

El análisis de tipo, de un programa fuente en java, es un claro ejemplo que muestra cómo se puede realizar análisis semántico sobre un AST (Fig. 4).

Este modelo está compuesto por un diccionario (Repository) que almacenará descriptores de clases (AClass), interfaces (AInterface), atributos (AField), métodos (AMethod) y constructores (AConstructor); un administrador de clases (ClassManager) que se encargará de manejar los nombres abreviados y la lista de *imports*; el *scope* será administrado mediante una pila que almacenará

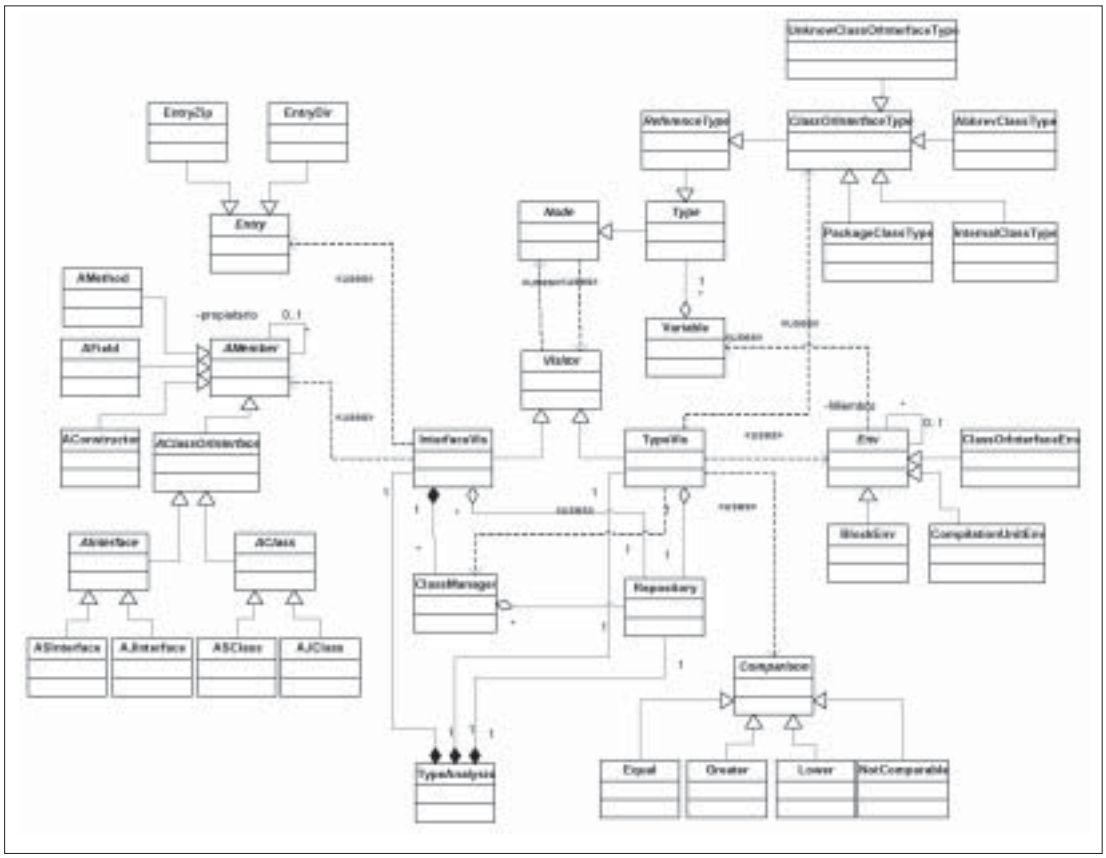

FIGURA 5. Modelo UML para el análisis semántico.

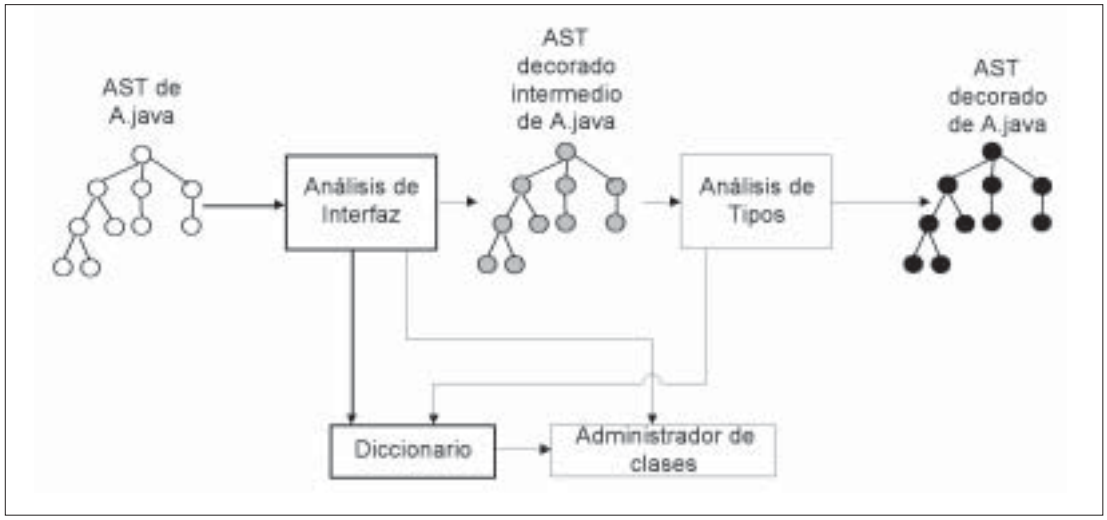

FIGURA 6. Análisis de Interfaz y de Tipos.

las clases que representan los ambientes (Env), de esta manera cada vez que se ingrese en un nuevo ambiente se apilará un objeto en la pila; la determinación de orden de magnitud entre los tipos se realizará mediante la clase de comparación (Comparison) y por último para el recorrido del árbol se emplearán visitadores (InterfaceVis y TiposVis).

Antes de explicar en qué consiste el análisis de tipos, debemos saber que cada expresión de un programa en java posee un tipo estático y otro dinámico. El primero puede ser comprobado en la compilación del programa, el segundo no puede ser comprobado por el compilador ya que se necesita información que debe ser obtenida en tiempo de ejecución, es decir, después de la compilación.

El análisis de tipos entonces es la etapa en la cual se determina el tipo estático de cada expresión del programa fuente.

Para realizar análisis de tipos se necesitan algunos pasos previos los cuales consisten en reunir información sobre la interfaz de cada clase, determinación del ámbito de las variables, determinación de los métodos de cada clase y su firma; a esta etapa previa la llamaremos análisis de interfaz y en ella nos concentraremos en llenar de información útil nuestro diccionario (Fig. 6).

El análisis de interfaz nos permite conocer la interfaz definida por una clase, es decir, corresponde a lo que es visible para otras clases de otros paquetes o del mismo paquete.

Recordemos que el archivo fuente, que estamos analizando, lo denominamos A.java del cual ya tenemos una representación jerárquica, el AST del análisis sintáctico. Para recorrer el árbol utilizaremos un visitador (InterfazVis), el cual construirá una clase denominada ClassManager en la cual almacenaremos la lista de imports y los nombre abreviados, a continuación se encargará de actualizar el diccionario con información adicional por cada clase o interfaz ya almacenada, esta información se compone de lista de métodos, lista de constructores y lista de interfaces implementadas. Además por cada método encontrado se agrega un descriptor (AMethod) asociado con su firma, este descriptor contiene la siguiente información: nombre, firma, tipo de retorno, tipo de los parámetros, tipo de las excepciones. Por último, en aquellas declaraciones de variables de instancia o clase, parámetros de un método, interfaces implementadas o super clase, donde aparezca un tipo referenciado (ReferenceType), se buscará esa clase en el diccionario. De no existir se buscará en disco utilizando el *classpath* y la lista de *imports*. Si se encuentra en disco se pueden dar tres casos:

a) Que exista el archivo de esa clase como un archivo fuente (.java). En este caso se debe realizar sólo la etapa parsing que explicamos en la sección de análisis de interfaz.
b) Que exista el archivo de esa clase como un archivo binario (.class). En este caso se debe utilizar una herramienta que permita el análisis de *bytecode* para obtener información de la clase.
c) Que existan ambos, el fuente y el binario. En este caso se consulta la fecha de última modificación del archivo, con lo cual se utiliza el más reciente y por ende la correspondiente fase de parseo anterior.

Finalmente, el resultado de todo este proceso será el mismo AST que utilizamos como entrada para esta etapa pero ahora con información adicional, por lo que lo denominaremos AST decorado intermedio.

Una vez que hemos completado el análisis de interfaz ya podemos comenzar con el análisis de tipos. Una de las cosas más complejas de realizar en esta etapa corresponde a la inferencia de tipos, que consiste en determinar para cada expresión del programa el tipo resultante por medio de reglas for-

males definidas para cada sentencia. Recordemos que los tipos de datos en java pueden ser básicos o referenciados.

En el caso de los tipos numéricos básicos, es importante destacar que existe un orden parcial entre ellos (Gosling *et al.*, 2000). Por ejemplo los tipos byte, short, int, long, float, double poseen la siguiente organización: byte < short < int < long < float < double.

Esto significa que un valor almacenado en una variable de tipo long puede ser asignada a otra variable de un tipo mayor sin pérdida de magnitud. Por el contrario, esta variable, no puede ser asignada a una variable con un tipo menor ya que ahora si habría pérdida en la precisión del valor almacenado.

Un ejemplo de las reglas de inferencia de tipos para la expresión aritmética suma se presenta a continuación.

$$\frac{T(e1)\leq int \wedge T(e2)\leq int}{T(e1+e2) \equiv int}$$

$$\frac{(T(e1)\equiv long \wedge T(e2)\leq long) \vee (T(e1)\leq long \wedge T(e2)\equiv long)}{T(e1+e2) \equiv long}$$

$$\frac{(T(e1)\equiv float \wedge T(e2)\leq float) \vee (T(e1)\leq float \wedge T(e2)\equiv float)}{T(e1+e2) \equiv float}$$

$$\frac{(T(e1)\equiv double \wedge T(e2)\leq double) \vee (T(e1)\leq double \wedge T(e2)\equiv double)}{T(e1+e2) \equiv double}$$

Según las reglas anteriores, si tenemos las siguientes expresiones:

short s= 20;
byte b= 80;
T(s + b) ≡ int

El caso de los tipos referenciados es similar al de los básicos ya que las clases también se organizan mediante un cierto orden definido según su línea de herencia. Por ejemplo, una clase B que hereda de otra A es menor que ella y ambas son menores que la clase Object, la cual es la base de todo el sistema de clases de java.

4. CONCLUSIONES

Mostramos que un árbol de sintaxis abstracta (AST) ad-hoc es más conveniente que un árbol de derivación ya que permite ventajas en términos del espacio de memoria necesario para almacenarlo, tiempo requerido para recorrerlo y facilidad de mantención.

También hacemos entrega a la comunidad de una herramienta de libre distribución, que no existe actualmente sobre internet y que permite realizar análisis semántico sobre programas escritos en jshield y java.

Los investigadores pueden enriquecer experimentalmente la gramática de java agregando nuevas capas, por ejemplo, sobre el análisis de tipos para desarrollar nuevas funcionalidades en java.

AGRADECIMIENTOS

Esta investigación ha sido financiada por el proyecto 032614 2/I - año 2003. Universidad del Bío-Bío.

REFERENCIAS

AHO R., SETHI R. y ULLMAN J. (1990) Compiladores: principios, técnicas y herramientas". Massachussets:Addison-Wesley, pp. 159-277.

APPEL, A. (2002) Modern compiler implementation in java. Cambridge:Cambridge University Press, pp. 86-111.

COOPER, J. (1998). The design patterns – java companion. Addison-Wesley, pp. 75-250.

FOWLER, M. (1996). Analysis patterns : reusable object models. Addison-Wesley, pp. 120-212.

GAGNON, E. y HENDREN, L. (1998) Sablecc, an object-oriented compiler framework. Published at the tools-98 conference. Montreal: McGill University.

GAMMA, E., HELM, R., JOHNSON, R. y Vlissides J. (1994) Design patterns. elements of reusable object-oriented software". Addison-Wesley. pp. 1-330.

GOSLING, J., JOY, B., STEELE, G. y BRANCHA, G. (2000) The java language specification. Addison-Wesley, pp 1-447.

HUDSON, S. (1997) CUP parser generator for java. http://www.cs.princeton.edu/appel/modern/java/CUP.

JOHNSON, S. (1975) YACC – yet another compiler compiler. technical report computing science, technical report 32 AT&AT Bell Laboratories, Murray Hill N.J.

PARR, T. (1995) ANTLR: a predicated-LL(k) parser generator. Journal of Software Practice and Experience, Vol. 25, 789-810.

SUN MICROSYSTEMS (2000) Javacc – the java parser generator. http://www.metamata.com/javacc.

WANG, W., TAO, K. y PALSBERG, J. (1997). Java tree builder, Indiana:Purdue University. http://www.cs.purdue.edu/jtb