Avances en Procesadores de Lenguajes y *Proof-Carrying Code*

J. Aguirre[2], R. Medel[1],M.Arroyo[2], N. Florio[2], F.Bavera[2], P. Caymes Scutari[2], D. Nordio[2]

RESUMEN

En este trabajo se presentan las líneas del grupo de investigación "Procesadores de Lenguajes" perteneciente al Departamento de Computación de la UNRC. Los aspectos fundamentales son la creación de modelos y herramientas de última generación para la generación de procesadores de lenguajes incluyendo la generación de Código Móvil Seguro. Se reseñan los trabajos del último año que han permitido: construir un generador de analizadores léxicos traductores, construir un generador de evaluadores concurrentes sin comunicación de gramáticas de atributos, realizar un análisis comparativo de los generadores de procesadores de lenguajes más usados y de *Japlage* y definir un lenguaje assembler tipado para la ejecución segura de código móvil no confiable. Además, se presentan los trabajos iniciados por el grupo destacando la obtención de un prototipo de un compilador certificante y entorno de ejecución para Proof-Carrying Code (PCC) – una técnica para garantizar código móvil seguro – .

INTRODUCCIÓN

El avance hacia la sociedad informatizada, sustentado por el gran desarrollo de las redes de computadoras y de las actividades que estas soportan y el consecuente uso masivo de las redes públicas de computadoras, impone la utilización de código que se mueva de un equipo a otro. Esta práctica es usada hoy tanto en grandes aplicaciones como en entretenimientos familiares. Resulta, por lo tanto, de primordial importancia que se pueda garantizar que la ejecución del código importado no dañe la integridad y/o privacidad del ambiente local en el que se ejecuta, es decir, que se trate de Código Móvil Seguro. Un importante ejemplo de software inseguro es el código que se importa mediante navegadores de Internet y un claro ejemplo de software invasor lo constituyen los virus que usan ese código como medio de propagación.

Proof-Carrying Code (PCC) – fue introducido por Necula y Lee [Nec98] y ha generado una activa línea de investigación, esta línea ha tenido una importante producción [App99][App01][Col00] y aún presenta muchos problemas abiertos – es una propuesta para garantizar código móvil seguro. Este mecanismo soporta tanto la construcción de pruebas matemáticas de propiedades de programas, las cuales son fácilmente chequeables, como la especificación formal de propiedades comportamentales de seguridad.

El campo de trabajo del grupo aborda los aspectos de la problemática citada. Como resultado instrumental se espera obtener un prototipo de compilador certificante y entorno de ejecución para PCC. Además se esperan obtener conclusiones sobre la posibilidad efectiva del uso de tipos para soportar PCC. Una conclusión relevante del trabajo sería clarificar la factibilidad del efectivo uso industrial de PCC.

RESULTADOS OBTENIDOS EN ÚLTIMO AÑO

Generador de Analizadores Léxicos Traductores "JTLex"

[1] Universidad Nacional de Río Cuarto, Stevens Insitute of Technology New Jerssey EE. UU, rhmedel@dc.exa.unrc.edu.ar.
[2] Universidad Nacional de Río Cuarto. e-mails: {jaguirre, marroyo, nflorio, pancho, pcaymesscutari, nordio}@dc.exa.unrc.edu.ar.
Este trabajado ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Córdoba Ciencia.

Se diseñó e implementó *JTLex*, un generador de analizadores léxicos *JTLex*. Al contrario que los generadores existentes, permite la especificación conjunta de la sintaxis y la semántica de los componentes léxicos, siguiendo el estilo de los esquemas de traducción. Para ello se basa en un nuevo formalismo, las Expresiones Regulares Traductoras (ERT), [Agu99]. Esta herramienta genera automáticamente un analizador léxico a partir de la especificación provista por el usuario. Los analizadores-traductores generados por *JTLex* procesan una cadena α en tiempo $O(|\alpha|)$ y el tiempo de generación de *JTLex* es del mismo orden que el requerido por los algoritmos usados tradicionalmente. Tanto su diseño como la especificación de los procedimientos con que el usuario implementa la semántica asociada a los símbolos son Orientados a Objetos. El lenguaje de implementación de *JTLex* es *Java*, como así también, el del código que genera y el que usa el usuario para definir la semántica. El lenguaje de especificación brindado por *JTLex* sigue el estilo de *Lex* – que es prácticamente un estándar -. *JTLex* se integrará, como un generador de analizadores léxicos alternativo al tradicional, a *japlage.*

Generador de Evaluadores Concurrentes de Gramáticas de Atributos

Se diseñó e implementó un generador de evaluadores concurrentes sin comunicación entre procesos para gramáticas de atributos no circulares de la familia NC(1). Esta familia contiene a la familia de las ANCAG (Absolutely Non-Circular Attribute Grammars), que hasta ahora se consideraba la mayor familia que permitía generar evaluadores estáticos eficientes. Los evaluadores generados son del tipo multivisita y utilizan información computada estáticamente para la selección de los planes de evaluación y segmentos de atributos independientes en el árbol sintáctico de entrada. Los segmentos se evalúan concurrentemente y su independencia hace que no se requiera ningún mecanismo de sincronización ni comunicación entre procesos. La partición creada sobre las instancias de los atributos de un árbol atribuido se basa en un algoritmo propuesto por Yang [Yang99]. El evaluador produce estáticamente los planes de evaluación proyectados en función de las posibles particiones. El diseño del generador de evaluadores y el modelo de evaluación es Orientado a Objetos y ha sido implementado en Java.

Análisis Comparativo de los Generadores de Procesadores de Lenguajes más usados y Japlage

Se determinaron características ortogonales para realizar el análisis, obteniéndose como síntesis los resultados de la tabla 1; así mismo fue comparada la eficiencia de la herramientas elegidas obteniéndose los resultados presentados en las tablas 2 y 3. Se puede concluir que ***Japlage*** es el generador que admite la clase más amplia de especificaciones y está bien posicionado en cuanto a performance, excepto en cuanto al tiempo de ejecución de los procesadores generados que es significativamente superior, se estima que la motivación principal es la política de scheduling de procesos usada por Java. Se ha iniciado el trabajo para estudiar y solucionar este problema.

Otra contribución de este trabajo fue la determinación del alcance real de ***JavaCC***. En la literatura [Jcc00] se afirma que extendiendo el *lookahead JavaCC* puede procesar cualquier gramática de la clase LL(K). Se pudo demostrar que no es así y que el alcance real es una subclase de la mencionada, la clase *Strong LL(K).*

	JAVACC	*YACC*	*JAVACUP*	*ELI*	*JAPLAGE*
Tipo de análisis	Descendente. Parsing Recursivo Descencente Predictivo	Ascendente. Parsing LALR	Ascendente. Parsing LALR	Ascendente. Parsing LALR	Ascendente. Parsing LALR
Tipos de Gramáticas	Gramáticas *LL(1)* y *Strong LL(k)*	Gramáticas *LALR(1)* Desambiguación explícita	Gramáticas *LALR(1)* Desambiguación explícita	Gramáticas *LALR(1)* Desambiguación implícita	Gramáticas *LALR(1)* Desambiguación explícita
Especificación semántica	Esquemas de traducción	Esquemas de traducción	Esquemas de traducción	Gramáticas de atributos	Esquemas de traducción con atributos
Alcance	Gramáticas l-atribuídas, si no se utiliza JJTree	Gramáticas S-atribuídas	Gramáticas S-atribuídas	Gramáticas de atributos Ordenadas –AOG-. L-atribuídas si usa la directiva BOTTOMUP	ATSs, en particular Gramáticas de atributos bien definidas sin ningún tipo de restricción.
Construcción de Arboles	El preprocesador JJTree permite la construcción de árboles sintácticos.	No	No	Si. LIGA, que procesa especificaciones LIDO	No, no los necesita, porque usa un grafo de dependencias implícito
Estrategia de Recuperación de errores	Producciones error	Producciones error	Producciones error	A nivel de frase/modo pánico	No está implementada
Opciones de personalización	Posee muchas opciones de personalización, relativas al lookahead, documentación, reporte de errores, debbuging, etc.	Sólo provee algunas opciones de personalización básicas, como legibilidad y utilización de prefijos en los nombres de los archivos generados.	Posee muchas opciones de personalización, relativas a generación de documentación, progreso de la compilación, resolución de conflictos, paquetes de almacenamiento, etc.	Posee muchas opciones de personalización, relativas a generación de documentación, progreso de la compilación, debugging, obtención de especificaciones concretas y abstractas, etc.	Provee sólo algunas opciones de personalización respecto del debugging, la economización de código generado o ejecutado del parser.
Lenguaje del procesador generado	Java	C,C++,FORTRAN,Pascal, APL,RATFOR,etc.	Java	C, C++	Java

Tabla 1: Comparación entre LPGs

	JavaCC	*Yacc*	*JavaCup*	*Eli*	*Japlage*
Caso 1	0m27,931s	0m1,126s	0m6,454s	0m0,137s	0m26,119s
Caso 2	0m29,391s	0m2,191s	0m36,707s	0m6,508s	0m30,812s

Tabla 2: Comparación de los tiempos de generación

JavaCC	*Yacc*	*JavaCup*	*Eli*	*Japlage*
0m3,737s	0m0,028s	0m2,970s	0m0,072s	0m39,875s

Tabla 3: Comparación de tiempos de ejecución de los procesadores generados

Lenguaje *Assembler* Tipado para la Ejecución Segura de Código de Origen no Confiable

En el marco del enfoque PCC para la ejecución segura de código de origen no confiable, se estudia la posibilidad de utilizar lenguajes *assembler* tipados a fin de proveer garantías de seguridad sobre el uso limitado de recursos en la máquina anfitriona y la detección temprana de código automodificante. Estos trabajos se abordan en cooperación con el grupo de PCC del Stevens Institute of Technology – New Jerssey, USA - dirigido por Adriana Compagnoni. En este contexto se ha desarrollado HBAL [Asp03], un lenguaje *assembler* tipado orientado a asegurar que cada programa de tipo válido ejecuta en un espacio de memoria constante: si el tipo de un programa HBAL es verificable este ejecutará en forma segura en cualquier memoria que satisfaga las condiciones iniciales de tipado. A fin de garantizar la ejecución segura de un programa *P* se debe:

1. chequear el tipado de *P* una única vez,
2. antes de cada ejecución, chequear que la memoria satisface las asunciones de tipado del contexto inicial.

Otros TALs (*Typed Assembly Languages*) basan su sistema de tipos en una configuración particular de memoria, por lo que se debe volver a chequear el tipado del programa antes de cada ejecución. El lenguaje HBAL asegura, mediante análisis estático, que los límites del espacio de *heap* asignado al programa son respetados (no se permite modificación de lugares arbitrarios en memoria), no se realiza lectura de direcciones no inicializadas previamente (no se permite lectura de “basura”) y no se realizan lecturas de direcciones que el programa no escribió él mismo o recibió como datos (no se permite el recorrido libre del espacio de memoria y no hay pérdida de información privada).

Actualmente se estudia la posibilidad de adaptar HBAL a otros escenarios, donde el principal énfasis de la seguridad está puesto en detectar tempranamente (preferiblemente antes de su ejecución) la presencia de código automodificante y la filtración de información confidencial (propiedad de no-interferencia).

HBAL realiza una distinción explícita entre las direcciones dedicadas a almacenar código y aquellas dedicadas a datos, por lo que, mediante una apropiada disciplina de tipos, se puede eliminar la posibilidad de la reescritura de direcciones donde se almacena código. Agregando a ésto la restricción del flujo de control a través del tipado de los rótulos, se elimina el peligro de la ejecución de instrucciones ocultas (o reescritas) en las direcciones donde se almacenan datos.

En cuanto a la verificación de la propiedad de no-interferencia, esto es, asegurar que en la ejecución de un programa la información pública no depende de información confidencial, se han estudiado las nociones de "flujo de información" y su análisis estático, advirtiéndose una escasez de trabajos en esta área que sean aplicables a código de bajo nivel, por lo que pretende inciar trabajos en el tema.

TRABAJOS INICIADOS

Extensiones del Entorno *Japlage*

- **Incorporación de JTLex a Japlage**: Se incorporará el generador de analizadores léxicos traductores JTLex a Japlage permitiendole al usuario seleccionar entre un generador de analizadores clásicos y JTLex.

- **Mejora de la Performance de los Traductores Generados:** se analizarán alternativas al mecanismo de scheduling de procesos de Java para mejorar el tiempo insumido por la ejecución concurrente de los procesos generados. en un futuro se incorporará la posibilidad de utilizar el modulo generador desarrollado para gramáticas NC(1).

Prototipo de Compilador Certificante y Entorno de Ejecución para PCC

La técnica PCC permite a un productor de software proveer un programa conjuntamente con la prueba formal de su seguridad. La política de seguridad del destinatario es dada mediante un sistema de axiomas y reglas. La demostración construida por el productor se basa en dicho sistema. El consumidor verifica la prueba recibida y sólo ejecuta el código si la prueba es satisfecha. PCC no requiere autenticación del productor, ya que el programa sólo correrá si locamente se ha demostrado su seguridad. PCC tampoco requiere verificación en tiempo de ejecución ni introduce ninguna merma en el tiempo de ejecución del programa ya que la prueba se efectúa antes de la corrida. Esta ventaja del enfoque estático de PCC se pierde en los enfoques dinámicos que controlan la seguridad operación a operación mientras el código se ejecuta; estos métodos de verificación, como el implementado por Java, son los que primero aparecieron para garantizar seguridad al Código Móvil.

Esta línea se focaliza en dos aspectos: uno atinente al productor de código y el otro a la funcionalidad del receptor.

El primero avanza hacia la obtención de un prototipo de compilador certificante. En ella se espera obtener el diseño de un compilador que genere PCC para dicha política. Sólo el desarrollo de compiladores certificantes eficientes podrá hacer que PCC pueda ser usado masivamente. Ellos permitirán compilar programas escritos en lenguajes de alto nivel a binarios PCC automáticamente.

La conjunción de PCC y compilación certificante proveerá una solución eficiente al problema de código móvil seguro. Un compilador certificante podrá producir no sólo código objeto seguro, sino también información de tipos y anotaciones de soporte para las optimizaciones que se pueden haber implementado.

El segundo avanza hacia la construcción de un entorno de ejcución de código PCC. En el se definirá una Política de Seguridad y se diseñaran el correspondiente verificador estático - ejecutor de la prueba recibida sobre el código- y el generador de código objeto a partir del código intermedio. Se estudiará la posibilidad de utilizar como código intermedio el assembler tipado ya desarrollado y de basar la política de seguridad en un sistema de tipos usando tipos dependientes. Sería ventajoso poder basar las técnicas de seguridad en sistemas de tipos; ello acarrearía las siguientes ventajas, inherentes a los tipos: Son sumamente flexibles y fáciles de configurar. Una vez que un sistema de tipos se selecciona, interesantes políticas de seguridad pueden ser obtenidas sólo declarando los tipos de valores almacenados en ciertos registros y locaciones de memoria. Más aun, distintas políticas de seguridad pueden ser obtenidas variando el sistema de tipos. Otra gran ventaja es que, para muchos sistemas de tipos de interés, los invariantes de ciclo pueden ser generados automáticamente por un compilador certificante lo que facilita el uso de un probador de teoremas para verificar las condiciones de seguridad.

Productos que se Esperan Obtener

Se espera poder conjugar los conocimientos adquiridos en el área de PCC en el desarrollo de un prototipo de entorno de ejecución que demuestre la factibilidad de su uso. Además se esperan obtener conclusiones sobre la posibilidad efectiva del uso de tipos para soportar PCC.

Como resultados concretos se esperan obtener: la especificación de la política de seguridad y del lenguaje fuente; la implementación de un compilador certificante, del generador de pruebas, de un prototipo de entorno de ejecución de Proof-Carrying Code, del ejecutor de pruebas del prototipo y la implementación del generador de código ejecutable a partir del código intermedio.

BIBLIOGRAFIA

[App01] Andrew W. Appel y Edward W. Felten. ”Models for Security Policies in Proof-Carrying Code”. 2001. Department of Computer Science, Princeton University, USA.

[Agu99] J. Aguirre, G. Maidana, M. Arroyo. Incorcorando Traducción a las Expresiones Regulares. Anales del WAIT 99 JAIIO, pp 139-154, 1999.

[Bav02] F. Bavera, D. Nordio, M Arroyo y J. Aguirre. JTLex: Un Generador de analizadores Léxicos Traductores – Anales del CACIC 2002. Universidad de Buenos Aires – 2002, pp 124-134.

[Jcc00] http://www.webgain.com/cgi-bin/printer_friendly.cgi (2000).

[Yang99] Wuu Yang. 1999. A classification of Non-Circular Attribute grammars based on the look-ahead behabior. Internal report, Ntional Chiao-Tung University.

[Yang99] Wuu Yang. 1999. A finest partitioning algorithm for attribute grammars. Second workshop on Attribute grammars and their applications - WAGA 99.

[Med02] Ricardo Medel, Mathieu Lucotte y Adriana Campagnoni. Implementing a Typed Assembly Language and its Machine Model. CACIC 2002. UBA.

[Asp03] Aspinall, David and Compagnoni, Adriana, “Heap Bounded Assembly Language”, accepted for publication in Journal of Automated Reasoning, Special issue on Proof-Carrying Code. To appear 2003.

[App99] Andrew W. Appel, Edward W. Felten y Zhong Shao. “Scaling Proof-Carrying Code to Production Compilers and Security Policies” . 1999. DARPA-funded research project.

[Col00] Christopher Colby, George Necula, Peter Lee, Fren Blan, Mark Plesko y Kenneth Cline. A Certifying Compiler for Java. 2000. ACM SIGPLAN PLDI’00.

[NEC98] George Nécula. “Compiling with Proof”. 1998. School of Computer Science. Carnegie Mellon University.