PROPOSICION DE UN LENGUAJE DE NIVEL INTERMEDIO PARA UN SISTEMA DE PRODUCCION DE COMPILADORES

A. Campos, Y. Eterovic, G. García, R. Santis
Depto. Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 6177 - Correo 22
Santiago - CHILE

RESUMEN

Se presenta el diseño de una máquina objeto virtual y de su lenguaje de ensamble, basado en las características principales de los lenguajes de programación modernos de mayor uso, y orientado a facilitar la generación automática del generador de código intermedio en un sistema de producción de compiladores. En particular, se describe la organización de la memoria, la representación de los tipos de datos básicos y estructurados, el manejo de subprogramas y de múltiples procesos, y las características relevantes del lenguaje de ensamble.

1.- Introducción

El estado del conocimiento en el área de construcción de compiladores ha permitido desarrollar generadores automáticos para las diferentes etapas del proceso de compilación: análisis léxico, sintáctico y semántico, y generación de código. Estos generadores automáticos son programas computacionales que, a partir de una especificación formal de un lenguaje de programación, generan la etapa correspondiente de un compilador para ese lenguaje.

En particular, es posible construir con cierta facilidad generadores de analizadores léxicos y sintácticos, a partir de especificaciones basadas en expresiones regulares y gramáticas libres de contexto. Mayor dificultad presenta la construcción de generadores de analizadores semánticos y generadores de código, debido a la falta de formalismos apropiados para especificar la semántica de los lenguajes de programación y las características de las máquinas objetos para las cuales se desea generar código (Campos y Eterovic, 1988).

Un sistema de producción de compiladores es un conjunto de generadores automáticos para todas o algunas etapas del proceso de compilación, organizadas de modo de reducir la cantidad y complejidad de la programación necesaria para construir compiladores para diferentes lenguajes y máquinas objetos.

En este artículo se presenta el diseño de una máquina objeto virtual con un único procesador y de su lenguaje de ensamble. El propósito de este diseño es proporcionar a un sistema de producción de compiladores un lenguaje común de nivel intermedio (el lenguaje de ensamble) que facilite la generación automática del generador de código intermedio de los diferentes compiladores producidos por el sistema. El diseño está basado en las principales características de los lenguajes de programación modernos de mayor uso, como se describe en las siguientes secciones.

2.- Los Registros y la Organización de la Memoria

Una característica común a todos los lenguajes de programación modernos es la estructura de bloques, normalmente conseguida a través de subprogramas que pueden anidarse estáticamente y activarse recursivamente hasta niveles arbitrarios de profundidad. El modelo de implementación más apropiado para este tipo de estructuras está basado en el funcionamiento de stacks que crecen o decrecen cuando se inicia o termina la ejecución de un subprograma, y cuyos elementos son representaciones de los subprogramas que están siendo ejecutados (Tanenbaum, 1978).

Por otra parte, los lenguajes más recientes también proveen facilidades para definir en un mismo programa y ejecutar simultáneamente varios procesos independientes, cada uno de los cuales requiere de su propio stack de ejecución (USDD, 1983; Wirth, 1985). Finalmente, también es posible para el usuario, en la mayoría de los lenguajes modernos, manejar parte de la memoria disponible en forma dinámica, esto es, solicitar y liberar memoria durante la ejecución de los programas.

En base a estas consideraciones, se ha diseñado la organización de la memoria de la máquina virtual en la forma de un arreglo lineal de bytes de 8 bits de longitud, conceptualmente dividido en 3 áreas durante la ejecución de los programas:

- Un área central, o área de código, destinada al almacenamiento de los programas una vez traducidos al lenguaje de ensamble de la máquina. La máquina provee acceso a esta área a través del registro Program Counter (PC), que contiene la dirección de la instrucción que debe ser ejecutada a continuación de aquélla que está siendo ejecutada;

- Un área organizada en la forma de un conjunto de stacks, utilizada para asignar un stack a cada uno de los procesos simultáneamente activos. En estos stacks se almacenan los registros de activación de los procesos activos y de los subprogramas que están siendo ejecutados. La máquina provee acceso a esta área a través de los registros Stack Pointer (SP), que contiene la dirección del tope del stack del proceso que está siendo ejecutado (el stack vigente), y Activation Record (AR), que contiene la dirección base del registro de activación que está en el tope del stack vigente;

- Un área organizada en la forma de un heap, definida como un área de memoria común para todos los procesos activos, que permite implementar las facilidades de manejo dinámico de memoria presentes en varios lenguajes de programación (Jensen y Wirth, 1975). La máquina provee dos instrucciones para manejar esta área: Allocate almacena en el stack vigente la dirección a partir de la cual hay un determinado número de bytes disponibles; Deallocate libera un determinado número de bytes, a partir de una dirección almacenada en el stack vigente. En ambos casos, el número de bytes también se encuentra en el stack vigente.

Cada uno de los tres registros mencionados, además de un cuarto registro de libre disposición, Mark (M), tiene 32 bits de longitud. Como las direcciones de memoria se representan en 4 bytes consecutivos o en éstos registros, es posible direccionar 2^{32} bytes.

3.- Tipos de Datos Básicos y Estructurados

Los siete tipos de datos básicos provistos por la máquina objeto y su longitud en bytes son los siguientes: Byte (1 byte), Word (4 bytes), DoubleWord (8 bytes), QuadWord (16 bytes), Floating (5 bytes), DoubleFloating (9 bytes), QuadFloating (17 bytes). En los tres últimos casos, el primer byte es utilizado para representar un exponente y el resto para representar una mantisa. Este conjunto posee la flexibilidad suficiente para permitir la representación de la totalidad de los tipos de datos elementales presentes en los lenguajes de programación modernos: integer, real, character, boolean, etc (Pratt, 1984).

Por otra parte, la forma más simple de tipos de datos estructurados presente en los lenguajes de programación modernos es la de los arreglos homogéneos, en las cuales todos los elementos de un arreglo pertenecen a un mismo tipo de datos. La estructura de estos arreglos puede ser estática, si es definida en el momento de la compilación de un programa y no puede ser modificada durante su ejecución, semidinámica, si es definida en el momento de la activación de un subprograma y no puede ser modificada durante su ejecución, o dinámica, si puede ser definida y modificada en cualquier momento durante la ejecución de un programa (Ghezzi y Jazayeri, 1982). La máquina virtual propuesta permite la representación y proveer las instrucciones necesarias para el manejo de arreglos de estructura semidinámica.

En el área de stacks de la máquina, un arreglo es representado por la dirección de su descriptor (Eterovic y Fuller, 1985). Un descriptor de arreglo (AD) es una estructura de bytes que contiene la información que define la estructura del arreglo y que permite localizar un elemento determinado de éste. Esta información consiste en el número de dimensiones del arreglo, el número de elementos en cada dimensión, y el número de bytes ocupados por cada elemento. La información contenida en el arreglo propiamente tal es almacenada en el área de stacks, a continuación del descriptor. Las instrucciones provistas para el manejo de los arreglos son las siguientes:

- PushAD almacena en el stack vigente el descriptor de un arreglo;

- CopyAD copia en el stack vigente el descriptor de un arreglo cuya dirección está en el tope del stack;

- PushElement almacena en el stack vigente la dirección de un elemento de un arreglo, a partir de los valores de los índices que identifican al elemento y que están almacenados en el tope del stack vigente;

- ArrayCopy copia el contenido del arreglo cuya dirección está en el tope del stack vigente en el arreglo cuya dirección está a continuación del tope del stack. Esta instrucción no verifica que los descriptores de ambos arreglos sean compatibles, con el propósito de facilitar la reestructuración dinámica de arreglos para el caso de los lenguajes que pudieran requerirla.

4.- Representación y Manejo de Subprogramas

En algunos lenguajes de programación modernos, los subprogramas son considerados como variables pertenecientes a una forma especial de tipos de datos estructurados (Tanenbaum, 1976). Estos tipos de datos están normalmente definidos por el número, posición y tipo de datos de sus parámetros formales y, en el caso de las funciones, además por el tipo de datos del valor que retornan. En estos lenguajes, el valor de una variable perteneciente a alguno de estos tipos de datos, es decir, el valor de un subprograma particular, habitualmente está definido por los nombres de sus parámetros formales y por el conjunto de sentencias que especifican su acción.

La máquina objeto virtual propuesta permite representar los subprogramas como variables pertenecientes a algún tipo de datos como los descritos. Para esto, el código de un subprograma, incluyendo el código necesario para asociar los parámetros formales con los correspondientes parámetros reales, es almacenado en el área de código de la memoria, una vez traducido al lenguaje de ensamble de la máquina. En el área de stacks, el subprograma es representado por la dirección de su descriptor. Un descriptor de subprograma (SD) es una estructura de bytes que contiene la información necesaria para permitir la ejecución del subprograma. Esta información consiste en la dirección de la primera instrucción del código correspondiente, el nivel de anidación estático con respecto al programa principal, la dirección base del registro de activación del subprograma en el cual está declarado, y una especificación del tipo de datos del valor que eventualmente retorna.

Normalmente, la interpretación de una llamada a un subprograma se traduce en el almacenamiento en el stack vigente de un nuevo registro de activación. La información básica contenida en estos registros permite hacer referencias a objetos que no son locales al subprograma que está siendo ejecutado, así como retornar el control al subprograma que hizo la llamada, una vez terminada tal ejecución. Esta información consiste en:

- La dirección de la instrucción que debe ser ejecutada inmediatamente después del retorno (dirección de retorno); corresponde al contenido del registro PC en el momento en que se produce la llamada;

- La dirección base del registro de activación en el que se produjo la llamada (enlace dinámico); corresponde al contenido del registro AR en el momento en que se produce la llamada;

- El nivel de anidación estático con respecto al programa principal; se obtiene del descriptor del subprograma;

- La dirección base del registro de activación más reciente en el cual el subprograma está declarado (enlace estático); también se obtiene del descriptor del subprograma.

En conjunto, la dirección de retorno y el enlace dinámico permiten retornar al subprograma que hizo la llamada. Por su parte, el nivel de anidación y el enlace estáticos permiten manejar las referencias a los objetos no locales al subprograma que está siendo ejecutado, en aquellos lenguajes en los que la asociación entre nombres y objetos es de tipo estática. Las instrucciones provistas para el manejo de los subprogramas son las siguientes:

- PushSD almacena en el stack vigente el descriptor de un subprograma; el nivel de anidación estático y la dirección base del código del subprograma deben ser especificados como operandos explícitos de la instrucción;

- SaveB reserva en el stack vigente el espacio necesario para almacenar un valor de tipo Byte; hay una instrucción similar para cada uno de los siete tipos de datos básicos de la máquina;

- CreateAR reserva en el stack vigente el espacio necesario para almacenar la información básica de un nuevo registro de activación;

- Call inicializa el contenido del registro de activación recién creado para comenzar la ejecución de un subprograma, con la información básica correspondiente;

- AssignB almacena en el registro de activación en el que se produjo la llamada el valor de tipo Byte retornado por el subprograma cuya ejecución está terminando; hay una instrucción similar para cada uno de los siete tipos de datos básicos de la máquina;

- Return reestablece, al terminar la ejecución de un subprograma, los contenidos que los registros SP, PC y AR tenían antes de iniciarse tal ejecución.

5.- Representación y Manejo de Múltiples Procesos

Los lenguajes de programación modernos de más reciente aparición, y otros no tan recientes, proveen facilidades para la definición y existencia simultánea de varios procesos activos. En general, estas facilidades no dependen de la posibilidad real de ejecutar en un computador dos o más procesos distintos en paralelo, sino que se basan en la existencia de interfaces que adecúan el número de procesos activos simultáneamente al número de procesadores disponibles. En el caso particular de la máquina propuesta, que consta de un único procesador, la estrategia consiste en asignar secuencialmente un intervalo igual de tiempo de ejecución a cada proceso activo.

Al iniciarse el intervalo de tiempo de ejecución asignado a un determinado proceso, la máquina obtiene la información necesaria para tal ejecución a partir del descriptor del proceso, almacenado en el área del heap. Un descriptor de proceso (PD) es una estructura de bytes que almacena el contenido que tenía cada uno de los cuatro registros del computador la última vez que se suspendió la ejecución del proceso correspondiente. Además, contiene las direcciones de los límites del área de stacks asignada al proceso y la dirección base del stack del proceso que hizo la activación. Durante la ejecución de un programa, existe un descriptor para cada uno de los procesos activos.

El tamaño máximo del stack asignado al proceso principal es definido por la máquina. El tamaño máximo del stack asignado a cada proceso activado durante la ejecución del proceso principal debe ser especificado en el instante de la activación. Para esto la máquina provee la instrucción CreateProcess que permite especificar en sus operandos explícitos dicho tamaño, así como de la dirección base del código correspondiente en el área central.

La activación de un proceso se traduce en la creación de un nuevo stack, que es almacenado en el área de stacks e inicializado con el registro de activación correspondiente al proceso. En estos registros, cuya estructura es idéntica a la de los subprogramas, el nivel de anidación es 0, la dirección de retorno y el enlace estático contienen ambos una dirección nula, y el enlace dinámico apunta al descriptor del proceso en el que se produjo la activación. En el caso particular del registro de activación del proceso inicial el enlace dinámico también contiene una dirección nula.

Debe notarse que no es posible asegurar la validez de los enlaces estáticos o dinámicos entre los procesos y los subprogramas en los cuales han sido declarados o activados. Esto se debe a que la ejecución de un subprograma puede terminar antes que la ejecución del proceso declarado o activado dentro de él, de modo que no puede asegurarse la existencia del registro de activación del subprograma durante todo el tiempo de ejecución del proceso: si durante la ejecución de un proceso se produce una referencia a un objeto no local, no puede asegurarse que ésta se pueda resolver siguiendo un enlace estático o dinámico al subprograma que contiene la declaración o produjo la activación del proceso, ya que este subprograma pudo haber terminado su ejecución antes de que se produjera la referencia.

De acuerdo con las características de los lenguajes que proveen procesos múltiples, el proceso (y no el subprograma) en el que se produce la activación de un segundo proceso, debe permanecer activo durante la ejecución de este último. Esto permite hacer referencias a objetos no locales al proceso que está siendo ejecutado, las que se resuelven siguiendo la cadena de enlaces dinámicos entre los registros de activación de los procesos y los descriptores de los procesos que hicieron las activaciones.

La búsqueda de un objeto en el stack de un proceso puede ser implementada de dos maneras diferentes. Por una parte, puede hacerse sólo entre las variables globales del proceso, para las que en todo instante existe una única instancia en el registro de activación inicial del stack correspondiente. Por otra, puede hacerse a partir del registro de activación que está en el tope del stack del proceso, y siguiendo la cadena de enlaces dinámicos entre los demás registros de activación. En este caso, se tiene acceso a todos los objetos activos en el stack, pero dos referencias a un mismo objeto pueden conducir a instancias diferentes de éste (ya que entre una referencia y otra puede producirse una nueva activación del subprograma al que pertenece el objeto), siendo responsabilidad del diseño del lenguaje correespondiente definir en forma adecuada el manejo de estas referencias.

Al finalizar la ejecución de un proceso, se libera el espacio ocupado por éste en el área de stacks y se elimina el descriptor correspondiente en el área del heap. Las eventuales referencias desde un proceso activo a objetos locales a un proceso cuya ejecución ya terminó detienen la ejecución del programa. La ejecución de un programa termina normalmente cuando ha terminado la ejecución de todos los procesos activados por el programa.

6.- Características del Lenguaje de Ensamble

Las instrucciones del lenguaje de ensamble de la máquina virtual son ejecutadas por un intérprete escrito en el lenguaje de programación C (Kernighan y Ritchie, 1978). La forma general de las instrucciones consiste en el nombre de la instrucción seguido por los operandos explícitos que puedan corresponder. La mayoría de las instrucciones, sin embargo, no tiene operandos explícitos. En estos casos, los operandos necesarios para la ejecución de las instrucciones se encuentran normalmente en el área de stacks o bien en el área del heap, y las instrucciones tienen acceso a ellos a través de los registros, ya sea directa o indirectamente (Hunter y Farquar, 1984).

En el caso de las instrucciones cuyos operandos, explícitos o implícitos, pertenecen a alguno de los siete tipos de datos básicos de la máquina, existe una versión de la instrucción para cada tipo de datos. Por ejemplo, el computador provee las instrucciones MultB, MultSW, MultDW, MultQW, MultSF, MultDF y MultQF para realizar la multiplicación entre dos valores de un mismo tipo de datos, estando ambos en el tope del stack vigente.

Similarmente, en el caso de las instrucciones cuyos operandos, explícitos o implícitos, representan direcciones de memoria, existe una versión de la instrucción para manejar direcciones absolutas y otra para manejar distancias relativas entre direcciones absolutas. Por ejemplo, el computador provee las instrucciones JumpAbs y JumpRel para transferir el control a una instrucción particular. El operando de la primera instrucción representa una dirección absoluta en el área de código de la memoria; el operando de la segunda instrucción representa una distancia, en bytes, con respecto a la dirección actual.

Las instrucciones provistas también facilitan la generación de código para las estructuras de control más comunes de los lenguajes de programación modernos. Por ejemplo, una sentencia del tipo

if A = B then sent1 else sent2,

puede ser fácilmente traducida a una secuencia de instrucciones tal como

```
        {instrucciones para almacenar los valores de A y B en el stack}
        BranchEQAbs   dir1
        JumpAbs       dir2
        ...
dir1:   {instrucciones correspondientes a sent1}
        BlockReturn
dir2:   {instrucciones correspondientes a sent2}
        BlockReturn
```

En este ejemplo, dir1 y dir2 son las direcciones en el área central de los códigos correspondiente a las instrucciones generadas para sent1 y sent2, respectivamente. Así, una vez que los valores de A y B han sido almacenados en el tope del stack vigente, la instrucción BranchEQAbs almacena el contenido actual del registro PC y, si ambos valores son iguales, lo reemplaza por la dirección dir1; en caso contrario, la instrucción JumpAbs almacena en el registro PC la dirección dir2. La secuencia de instrucciones correcta se ejecuta a continuación y luego la instrucción BlockReturn reestablece el contenido original del registro PC para continuar la ejecución del programa.

Además se incluyen instrucciones del tipo Branch para los diferentes tipos de comparaciones (LT, GT, NE, LE, GE), y las instrucciones BlockReturnT y BlockReturnF para terminar la ejecución de iteraciones de los tipos for, repeat y while.

La máquina virtual no provee instrucciones para realizar operaciones de entrada y/o salida de datos; provee, sin embargo, la instrucción ExecuteExternal que permite ejecutar rutinas definidas externamente al lenguaje de ensamble, como parte de la definición de un determinado lenguaje de alto nivel. El nombre de la rutina externa debe ser especificado como operando de la instrucción. Esto es, el código generado por el compilador de un lenguaje de alto nivel puede contener llamadas a estas rutinas, las cuales serán procesadas correctamente por el intérprete del lenguaje

de ensamble si han sido definidas apropiadamente. Estas definiciones pueden incluir el uso de otras instrucciones del lenguaje de ensamble, y deben informarse al intérprete mediante la inserción de los nombres de las rutinas externas en una tabla especial. La comunicación de parámetros entre las rutinas externas y el intérprete se hace a través del área del heap. Para esto, el computador provee la instrucción Lock, que impide el acceso al byte cuya dirección ha sido especificada como operando de esta instrucción.

Finalmente, el código generado por un compilador puede contener labels para identificar determinadas instrucciones, así como referencias a estas instrucciones hechas a través de los labels. El intérprete del lenguaje de máquina solucionará estas referencias en el momento de cargar el código en el área central de la memoria, traduciéndolas a direcciones absolutas o bien a distancias relativas entre direcciones absolutas.

7.- Conclusiones

Se ha descrito una máquina virtual que provee las herramientas básicas para representar las construcciones usuales presentes en la mayoría de los lenguajes de programación modernos. En especial, se ha puesto énfasis en facilitar la implementación de arreglos que pueden ser reestructurados dinámicamente durante la ejecución de un programa, en la representación de subprogramas como tipos de datos, y en el manejo de múltiples procesos en un único procesador. En este sentido, se ha propuesto un esquema para resolver las referencias a los objetos no locales a un determinado proceso. Además, los recursos descritos ofrecen la flexibilidad necesaria para incorporar variaciones; en particular, el conjunto de instrucciones del lenguaje de ensamble puede ser fácilmente extendido por el usuario del sistema.

AGRADECIMIENTOS

Este trabajo es parte del proyecto DIUC Nº 19/86 "Desarrollo de un Ambiente para la Generación de Compiladores", financiado por la Dirección de Investigación de la Universidad Católica de Chile, organismo al que los autores desean manifestar su agradecimeinto.

REFERENCIAS

1.- Campos, A., Eterovic, Y. "A Compiler Production Environment". Presentado para consideración a The Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, Enero, 1988.

2.- Eterovic, Y., Fuller, D. "Preliminary Design of a High Level Language Machine Architecture". Anales VI Congreso Chileno de Ingeniería Eléctrica, 469-474, Santiago, Chile, 1985.

3.- Ghezzi, C., Jazayeri, M. Programming Languages Concepts. John Wiley, New York, 1982.

4.- Hunter, C.B., Farquar, E. "Introduction to the NS16000 Architecture". IEEE Micro, 26-47, April 1984.

5.- Jensen, K., Wirth, N. Pascal User Manual and Report. Springer Verlag, New York, 1975.

6.- Kernighan, B.W., Ritchie, D.M. The C Programming Language. Prentice- Hall, Englewood Cliffs, 1978.

7.- Pratt, T.W. Programming Languages: Design and Implementation. Prentice-Hall, Englewood Cliffs, 1984.

8.- Tanenbaum, A.S. "A Tutorial on ALGOL 68". ACM Computing Surveys, 155-190, June 1976.

9.- Tanenbaum, A.S. "Implications of Structured Programming for Machine Architecture". Communications of the ACM, 237-246, March 1978.

10.- USDD. Reference Manual for the Ada Programming Language. U.S. Department of Defense, Washington, D.C., 1983.

11.- Wirth, N. Programming in Modula-2. Springer-Verlag, Berlin, 1985.