Compiladores
Facultad de Ciencias UNAM
Back-end: introducción

Lourdes Del Carmen González Huesca [*]

29 de abril de 2024

Resumen

Durante esta nota se presenta un panorama sobre las fases que componen al *back-end*. Además se presentan algunos términos preliminares que serán útiles para el estudio de esta última parte del compilador.

Términos clave:
Es: Selección de instrucciones, Asignación de registros, Generación de código, Registros de Activación, Pila, Heap, Static.
En: Instruction selection, Register Allocation, Optimal code generation, Activation Record / Frame, Stack, Heap, Static.

En esta parte del compilador se toma como entrada una o varias representaciones intermedias junto con la información de la tabla de símbolos que se produjeron con anterioridad, la salida esperada es un *código objeto* semánticamente equivalente al original. Este código debe preservar el significado semántico del programa original y no sólo eso, sino que se debe asegurar que es de mejor calidad que el original; debe hacer buen uso de los recursos disponibles de la máquina. Como por ejemplo se debe manejar de forma eficiente lo siguiente:

- Creación y manejo de un ambiente de ejecución
- Almacenamiento y manejo las posiciones de memoria
- Acceso a las variables
- Ligado de procedimientos
- Manejo de paso de parámetros
- Comunicación con el Sistema Operativo y dispositivos periféricos.

Una consideración importante para el back-end es el problema de generar un código objeto óptimo para cualquier código fuente dado. Este problema es indecidible, por tanto se utilizan técnicas heurísticas para generar *buen* código, respecto a las características físicas de la máquina, aunque no sea necesariamente el más óptimo.

[*]Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

En este curso se aborda el estudio del *back-end* a partir de las siguientes fases:

1. Selección de instrucciones
   escoger las instrucciones adecuadas dependiendo de la máquina en que se está compilando (*target-machine instructions*)

2. Asignación de registros
   decidir qué valores e información se guarda en los registros del procesador de forma eficiente

3. Generación de código
   ordenar las instrucciones para una ejecución óptima

Cabe mencionar que entre estas fases pueden realizarse procedimientos para la optimizacón de código de bajo nivel.

Para la segunda fase se utiliza un lenguaje intermedio y varias transformaciones para realizar una traducción a otro lenguaje de bajo nivel que permita asignar registros del procesador de forma eficiente:

1. Transformación en un lenguaje de registros (Register Transfer Language)

2. Transformación en un lenguaje de registros explícito (Explicit register transfer language)

3. Asignación de registros físicos del procesador (Location transfer language)

En esta nota describimos de forma breve las fases del back-end y estudiaremos la primera para seleccionar las instrucciones. Las demás fases se revisarán en notas subsecuentes. A continuación una sección de conceptos y nociones preliminares para entrar en contexto con las fases de síntesis de código de bajo nivel.

Preliminares

Dentro del back-end de los compiladores se tratan a las rutinas o funciones como unidades independientes llamada bloques, también se trata a la memoria de una forma particular dado que la interacción con los registros del procesador será fundamental para la ejecución del código.

La memoria en tiempo de ejecución es administrada como una pila o *stack* y cada vez que una rutina es llamada se le asigna un espacio en el *stack* para que pueda ser ejecutada. Este proceso tiene muchos pormenores, pero antes de dar un bosquejo sobre la estructura general de la memoria en tiempo de ejecución es necesario entender el ambiente necesario para poder ejecutar cada rutina.

Registros de activación Una llamada a una función o rutina está compuesta por los siguientes elementos:

1. Un ambiente de ejecución que contiene una pequeña memoria temporal y variables locales.
2. Los parámetros y argumentos con los que fue llamada la rutina.
3. Los valores de retorno o la información que se espera que devuelva la rutina, no necesariamente la que se comunique al usuario.

La lista anterior muestra que una llamada a una rutina requiere la creación de un ambiente. A la estructura que engloba este ambiente se le llama registro de activación o *frame*.

Por tanto un registro de activación contiene los datos pertinentes a la llamada o invocación de una rutina o función; representa una rutina iniciada y aún no terminada ya que el registro de activación es eliminado una vez se terminó la ejecución de la función. En específico contiene datos del usuario como variables locales, parámetros, valores de retorno, contenido de los registros, direcciones de memoria y apuntadores a otros registros de activación.

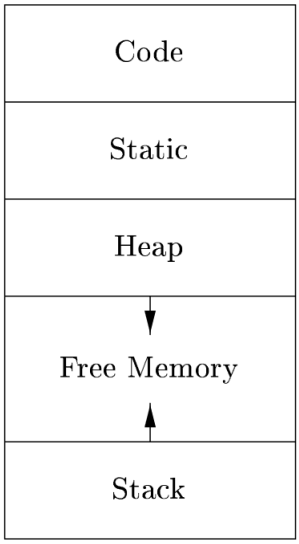
Caller y Callee Como resultado de las estructuras anteriores, a la función que *llama* a otra función se le conoce como *caller* y la función llamada se le conoce como *callee*.

Una vez que se ha acordado que cada rutina es un elemento independiente y que cada una de ellas está envuelta en un registro de activación, lo restante es idear una forma de mantener la comunicación entre las rutinas.

La solución que se le ha dado a este problema es tratar a la memoria como una pila o *stack*. Con esta estructura, siempre que se haga una llamada a una rutina se agregará su registro de activación en el tope de la pila y será eliminado del tope de la pila una vez que termine su ejecución.

Notemos que esta estructura nos permite compartir el espacio entre llamadas a rutinas que no se traslapan en el tiempo. Además nos permite compilar el código de una rutina de tal forma que las direcciones relativas de sus variables no locales son siempre las mismas.

Memoria en tiempo de ejecución Dado que la ejecución de un programa requiere su propio espacio designado de memoria, una vista general de la memoria de un programa durante su ejecución sería la siguiente.



Donde se asume que la memoria es un bloque contiguo de *bytes*. En este tipo de memoria la cantidad de espacio requerido para almacenar un objeto está determinado por el *tipo* del objeto, mientras que el espacio requerido para almacenar un tipo primitivo `char`, entero `integer`, un flotante `float` es fijo.

Padding La capa de almacenamiento para los objetos depende fuertemente del *hardware*.

Por ejemplo, en el manejo de memoria se espera que los datos se encuentren alineados, es decir que estén ubicados en un espacio de memoria contiguo. En algunos lenguajes de programación, en el caso de los enteros y bajo una arquitectura de 32 bits, el tamaño de este tipo de datos es de 4 bytes, por lo tanto la alineación debe ser múltiplo de 4. Esto facilita las operaciones, en este caso la instrucción de suma de enteros.

Otro ejemplo es en el lenguaje de programación C donde un arreglo de caracteres de tamaño 10 sólo necesita 10 bytes para su almacenamiento, pero el compilador reservará 12 bytes de almacenamiento para mantener la alineación de múltiplos de 4. A esta adición de bytes vacíos que se deja para alinear los datos se le conoce como padding.

Code El tamaño del código generado está determinado una vez que se concluye la compilación y este tamaño no cambia durante la ejecución del programa. Por tanto es posible almacenar el texto literal del código en una área estática llamada `Code`.

Static De manera similar al finalizar el tiempo de compilación también se conoce el tamaño fijo de otros elementos tales como constantes globales o información que permita el correcto uso del recolector de basura. Esta información suele ser almacenada en el área `Static`.

Existen razones estadísticas que para almacenar de forma estática tantos elementos como sea posible; las direcciones fijas de dichos elementos pueden ser compiladas en el código objeto.

Heap El heap es un área de memoria virtual que le permite a los objetos y otros elementos dinámicos obtener un espacio de almacenamiento al momento de ser creados y de devolver el espacio una vez que han sido invalidados. Para mantener este espacio de forma eficiente el recolector de basura en tiempo de ejecución un sistema para detectar objetos que ya han sido invalidados y reusar su espacio de almacenamiento aún si el programador no devuelve el espacio de forma explícita

Stack Esta sección es utilizada para almacenar los registros de activación de cada rutina en tiempo de ejecución.

Se advierte que el Heap y Stack se encuentran en extremos opuestos del espacio restante de la memoria designada para el programa. Además son dinámicas ya que su tamaño varía durante la ejecución del programa y crecen la una hacia la otra conforme es requerido durante la ejecución.

Selección de instrucciones

En esta fase el objetivo es seleccionar instrucciones adecuadas del procesador y utilizarlas para crear nuevas representaciones intermedias pero de más bajo nivel.

Esta tarea se divide principalmente en:

- Reemplazar las operaciones ariméticas del lenguaje de las representaciones intermedias (IR) por las instrucciones explícitas del procesador. Esto debe hacerser seleccionando las operaciones que mejor optimicen al código, entre algunas de las operaciones que se pueden elegir estan la suma de registros y constantes, corrimiento de bits para multiplicar o dividir, comparaciones básicas, entre otras.
- Reemplazar el acceso a campos de estructuras por operaciones explícitas en la memoria. Esto debe hacerse respetando la evaluación del lenguaje de programación (perezosa, estricta, paso de valores por valor o por referencia)

Lo anterior considerando que para cada instrucción de la IR se debe asegurar una nueva traducción adecuada y eficiente. Para estas traducciones se ocupan las operaciones explícitas de la memoria `load` y `store`, donde `load` carga o trae un dato de memoria y `store` almacena un un valor en un espacio de memoria.

Es importante tomar en cuenta que en esta fase los registros siguen siendo registros lógicos, no físicos. Es decir aún no representan una localidad física de memoria, sino que son simbólicos para fines de compilación.

Ejemplo 1. Ejemplo de selección de instrucciones para una suma.

La expresión:

$$x := y + z$$

Se transforma en

```
load $19, y
load $20, z
add $21, $19, $20
store $21, x
```

Ejemplo 2. Ejemplo de selección de instrucciones para dos sumas.
La expresión:

$$a := b + c$$

$$d := a + e$$

Se transforma en

```
load $22, b
load $23, c
add $24, $22, $23
store $24, a
load $25, a
load $26, e
add $27, $25, $26
store $27, d
```

Notemos que se hace un `store` en el registro $24 del valor de la variable a y en seguida se vuelve a cargar ese mismo valor en el registro $25. Un ejemplo de las optimizaciones que se realizan en esta fase es evitar la redundancia en la carga del valor a.
De tal forma que la versión final de las instrucciones es:

```
load $22, b
load $23, c
add $24, $22, $23
store $24, a
load $26, e
add $27, $24, $26
store $27, d
```

Así se evita cargar el valor a en un nuevo registro y se ocupa el registro $24 que ya contiene su valor. Aunque estas optimizaciones vuelvan más complejo al proceso de compilación conviene realizarlas ya que generan código eficiente para ejecutarse.

Observación: En los compiladores reales las instrucciones `load` y `store` son reemplazadas por `lw` y `sw` que significan *load word* y *store word* respectivamente, esto dado que en un procesador los espacios para el manejo de datos son llamados palabras o `words` y suelen ser de 32 ó 64 bits según la arquitectura de la máquina.

Asignación de registros

Durante esta fase del backend se presentan diferentes estrategias para decidir en cada punto de un programa qué valores deben de residir en los registros y cómo elegir adecuadamente el registro a utilizar. El objetivo es sustituir los registros lógicos por registros físicos del procesador o su ubicación dentro del stack de ejecución.

Dada la complejidad de este proceso se divide en las siguientes fases:

1. Análisis de supervivencia (liveness analysis)
2. Creación de la gráfica de interferencia
3. Asignación de registros

Generación de código

En esta fase el compilador debe de completar la tarea de producir el código objeto, es decir traduce el código fuente en operaciones de máquina. Dado que esta traducción ya es de más bajo nivel y está relacionada con la máquina donde se ejecutará el código es posible considerar las instrucciones particulares del procesador.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Torben Ægidius Mogensen. *Basics of Compiler Design*. Lulu Press, 2010.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[7] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.