Compiladores
Unidad 1: Introducción
Etapa de síntesis

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

La fase de síntesis constituye la segunda gran etapa en el proceso de compilación. Su propósito es transformar la representación intermedia en la fase de análisis (normalmente un ASA) enriquecido con información semántica en un programa objeto que pueda ejecutarse en una máquina concreta. Esta fase no sólo busca traducir, sino también optimizar: generar un código que respete la semántica del programa y, al mismo tiempo, aproveche los recursos del hardware.

En esta nota recorreremos las distintas etapas que conforman la síntesis: comenzaremos con la generación de código intermedio, para después revisar cómo se aplican optimizaciones en distintos niveles, tanto independientes de la máquina como específicas de la arquitectura. Posteriormente, veremos el proceso de traducción a código objeto y su respectiva optimización, analizando los retos de eficiencia que surgen en procesadores modernos. Nuestro objetivo es comprender cómo, a partir de una representación abstracta del programa, se llega a un ejecutable eficiente que aprovecha las características del hardware sin sacrificar la corrección semántica.

1. Generación de código intermedio

El primer paso en la síntesis es producir una representación equivalente en un lenguaje intermedio, el cual funciona como un puente entre el análisis del programa fuente y la generación de código objeto. Este lenguaje debe ser independiente de la máquina, pero lo suficientemente cercano a la arquitectura para facilitar la traducción posterior.

Ejemplo 1

Un ejemplo clásico es el código de tres direcciones, donde operaciones complejas se descomponen en secuencias simples como:

```
t1 = x < 10
if t1 goto L1
```

Este tipo de representación cumple un papel fundamental: desacopla el compilador de los detalles de la máquina, facilitando la portabilidad. Así, un mismo *frontend* puede generar código intermedio que luego diferentes *backends* traducirán arquitecturas. Ell *frontend* es el conjunto de analizadores que conforman la etapa de análisis, mientras que el *backend* es el conjunto de programas responsables de la etapa de síntesis.

Además, la adopción de un lenguaje intermedio resuelve el problema de la explosión combinatoria cuando existen múltiples lenguajes fuente y múltiples arquitecturas destino. Si se desea compilar L lenguajes para M máquinas, la solución inmediata sería desarrollar $L \times M$ compiladores distintos. Sin embargo, con un lenguaje intermedio basta implementar L traductores que lleven los lenguajes fuente al intermedio y M traductores que lleven el intermedio a cada máquina. En total, sólo se requieren $L + M$ compiladores, reduciendo drácticamente el esfuerzo de desarrollo.

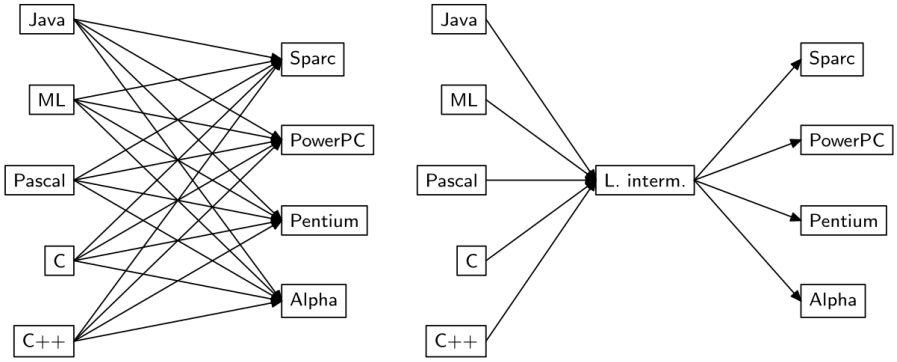

Figura 1: Utilidad de las representaciones intermedias

Este enfoque modular hace posible que, al añadir un nuevo lenguaje de programación a la colección, no sea necesario crear M compiladores nuevos, sino únicamente un traductor hacia el lenguaje intermedio. De manera análoga, al incorporar una arquitectura de hardware, basta con un traductor desde el intermedio hacia esa máquina. Esta idea, ilustrada en la literatura clásica de compiladores, constituye uno de los pilares para el diseño de compiladores portables y escalables.

2. Optimización de código intermedio

La segunda etapa busca mejorar el rendimiento del programa sin alterar su semántica. Aquí entran en juego las optimizaciones independientes de la máquina, que aprovechan la estructura del lenguaje intermedio.

Algunos ejemplos son:

- ⋆ Remover instrucciones cuyo resultado no afecta la ejecución.
- ⋆ Sustituir expresiones como $3 + 2$ por 5.
- ⋆ Sacar expresiones que siempre calculan el mismo valor dentro de un ciclo.

Estas optimizaciones, además de producir programas más rápidos, contribuyen a reducir el consumo de memoria y energía, lo que resulta esencial en arquitecturas modernas y en dispositivos móviles.

3. Generación de código objeto

Una vez optimizado el código intermedio, el compilador procede a traducirlo en código objeto, es decir, instrucciones específicas de la máquina destino. Este paso requiere tener en cuenta aspectos como:

- ⋆ Decidir qué valores se guardan en registros y cuáles en memoria.
- ⋆ Elegir la instrucción de máquina adecuada para cada operación.
- ⋆ Organizar variables en la prila, el *heap* o la memoria estática.

Ejemplo 2

La operación intermedia `t1 = x + y` puede traducirse en un procesador x86 como:

```
MOV R1, x
ADD R1, y
MOV t1, R1
```

Este paso es particularmente delicado porque combina precisión semántica con eficiencia técnica. Una mala asignación de registros, por ejemplo, puede degradar significativamente el desempeño del programa.

4. Optimización de código objeto

Finalmente, el compilador aplica una optimización específica de la máquina destino, ajustando el programa objeto para aprovechar las particularidades del hardware.

Ejemplos típicos incluyen:

- ⋆ Reordenamiento de instrucciones para evitar esperas en la segmentación de intrucciones (*pipeline*) del procesador.
- ⋆ Sustitución de secuencias largas por instrucciones más rápidas disponibles en la arquitectura.
- ⋆ Minimización de accesos a memoria mediante el uso eficiente de registros.

En esta etapa, el compilador se convierte en una obra ingenieril de bajo nivel, capaz de producir código que no sólo es correcto, sino también competitivo en términos de rendimiento.

5. El rol de la síntesis en la actualidad

Mientras que la fase de análisis reduce la brecha semántica entre el lenguaje humano y la representación formal, la síntesis busca reducir la brecha entre la representación abstracta y el hardware físico.

En este sentido, la síntesis puede verse como un proceso de *traducción descendente:* desde estructuras lógicas y abstractas hacia instrucciones concretas y ejecutables. Cada paso de optimización y traducción

contribuye a hacer que los programas escritos por humanos, con estructuras de alto nivel y a menudo redundancias, se transformen en secuencias eficientes que aprovechan las características de la máquina.

En la actualidad, este papel se vuelve especialmente interesante con el auge de la Inteligencia Artificial Generativa aplicada a la programación. Herramientas como ChatGPT pueden escribir código en lenguajes de alto nivel, pero aún dependen de compiladores y sistemas de síntesis para traducir esos programas en ejecutables eficientes. El desafío contemporáneo consiste en integrar estas dos visiones: la generación probabilística de código y la traducción rigurosa y optimizada de los compiladores. Ambos procesos comparten un mismo objetivo: acercar el pensamiento humano al lenguaje de la máquina de manera confiable y eficaz.

6. Conclusión

En conclusión, la síntesis es la etapa donde las ideas del programa se transforman en instrucciones reales para la máquina. No sólo traduce, también optimiza, asegurando que el resultado sea correcto y eficiente en su ejecución.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2.ª ed. Pearson, 2006.
[2] K. D. Cooper y L. Torczon, *Engineering a Compiler*, 2.ª ed. Morgan Kaufmann, 2011.
[3] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.