UNIVERSIDAD SAN PABLO DE GUATEMALA

Facultad de Ciencias Empresariales

Escuela de Ingeniería

Ingeniería en Ciencias y Sistemas de la Computación

Tarea

Semana 16

Transformación y optimización en compiladores.

Impartido por Ing. Walter Oswaldo Ordoñez Sierra

Manuel de Jesus Alvarez Xi 1700224

Guatemala, 07 de noviembre de 2025

Contenido

Resumen 3
Introducción 4
Fases de análisis del compilador 4
Representaciones intermedias 5
Técnicas de optimización 5
Compiladores modernos y optimización avanzada 6
Conclusión 7
Referencias 8

Resumen
Este documento explora el proceso mediante el cual los compiladores transforman el código fuente en representaciones intermedias y aplican técnicas de optimización antes de la generación de código final. Se describen los fundamentos teóricos y prácticos basados en las obras de Aho, Lam, Sethi y Ullman (2008), Appel (2002) y Cooper & Torczon (2012), abordando las fases de análisis, las estructuras intermedias como el Árbol de Sintaxis Abstracta (AST) y el Código de Tres Direcciones (TAC), así como las estrategias de optimización más utilizadas en los compiladores modernos.

Introducción

Un compilador es un programa complejo que traduce código fuente escrito en un lenguaje de alto nivel a un código ejecutable por una máquina. Este proceso no es directo, sino que implica múltiples fases de análisis, transformación y optimización. Las representaciones intermedias (IR) juegan un papel central, ya que permiten que el compilador opere sobre una versión abstracta del código antes de su conversión final.

Los compiladores modernos, como GCC o LLVM, emplean sofisticadas técnicas de optimización basadas en teoría de grafos, análisis de flujo de datos y representaciones únicas como SSA (Static Single Assignment). El propósito de este trabajo es analizar cómo el código fuente se transforma en estas representaciones y cómo las optimizaciones contribuyen a mejorar la eficiencia del programa resultante.

Fases de análisis del compilador

Además de las formas clásicas de representación como el AST, TAC, DAG y SSA, las representaciones intermedias pueden adoptar estructuras más complejas según el enfoque del compilador. Entre las más comunes se encuentran los gráficos semánticos, las tuplas y el código de pila. El gráfico semántico suele derivarse de un árbol de sintaxis abstracta ampliado y anotado con información de tipos y dependencias; su función es mantener una representación estructural del programa que permita reconstruir el flujo lógico del código. Las tuplas, por otro lado, representan instrucciones de forma más cercana al nivel máquina, siguiendo el modelo de una arquitectura con registros ilimitados. Cada tupla consta de un operador y un conjunto de operandos, lo que permite expresar operaciones aritméticas, lógicas y de control con precisión formal. El código de pila se utiliza en compiladores diseñados para máquinas virtuales basadas en pila, donde las operaciones se ejecutan mediante inserciones (push) y extracciones (pop) en una estructura de pila de ejecución. Estas formas intermedias difieren en su nivel de abstracción y en la cantidad de información semántica que conservan del código fuente. Sin embargo, todas comparten el objetivo de ofrecer una representación independiente del lenguaje de programación y del hardware, facilitando la optimización y la posterior generación de código. En consecuencia, una buena representación intermedia debe ser eficiente de manipular, expresiva y lo suficientemente flexible para adaptarse a distintas arquitecturas y estrategias de optimización (Wikipedia, 2024).

Según Aho et al. (2008), el proceso de compilación se divide en dos grandes etapas: el análisis y la síntesis. La fase de análisis examina el código fuente para comprender su estructura y significado.

1. Análisis léxico: divide el código fuente en unidades léxicas llamadas tokens. Por ejemplo, la expresión 'a = b + c;' se convierte en los tokens [IDENT(a), OP(=), IDENT(b), OP(+), IDENT(c), DEL(;)].

2. Análisis sintáctico: organiza los tokens en una estructura jerárquica (árbol de análisis) que representa la gramática del lenguaje.
3. Análisis semántico: valida los tipos, el alcance de las variables y las reglas semánticas del lenguaje. El resultado de estas fases es un Árbol de Sintaxis Abstracta (AST).

Representaciones intermedias
Las representaciones intermedias (IR) actúan como un puente entre el código fuente y el código máquina. Permiten aplicar optimizaciones y simplificar la traducción. Existen diferentes niveles de IR:

- **AST (Árbol de Sintaxis Abstracta):** estructura jerárquica que representa las relaciones gramaticales del programa. Por ejemplo, para la expresión 'x = a + b * c', el nodo raíz es la asignación, con hijos 'x' y una expresión aritmética 'a + (b * c)'.
- **TAC (Código de Tres Direcciones):** conjunto de instrucciones simples del tipo t1 = b * c; x = a + t1.
- **DAG (Directed Acyclic Graph):** representación que elimina subexpresiones redundantes.
- **SSA (Static Single Assignment):** cada variable se asigna una sola vez, lo que facilita optimizaciones globales.

Técnicas de optimización
La optimización busca mejorar la eficiencia del programa sin alterar su comportamiento. Cooper y Torczon (2012) clasifican las optimizaciones en locales y globales:

- **Optimizaciones locales:** se aplican dentro de un bloque básico. Ejemplos: eliminación de código muerto, propagación de constantes, plegado de expresiones y simplificación algebraica.
- **Optimizaciones globales:** abarcan varios bloques o funciones, como el movimiento de invariantes de bucle, análisis de alcance y reducción de fuerza.

Appel (2002) resalta que las optimizaciones deben balancear la eficiencia con la complejidad del compilador. Un ejemplo clásico es la sustitución de subexpresiones comunes:
Antes:
```
t1 = a * 2
t2 = a * 2
b = t1 + t2
```
Después:
```
t1 = a * 2
b = t1 + t1
```

Compiladores modernos y optimización avanzada

Los compiladores contemporáneos, como LLVM y GCC, integran fases de optimización en múltiples niveles. LLVM utiliza una representación intermedia en SSA que facilita la aplicación de más de 200 pases de optimización. Entre ellas destacan la inlining de funciones, la eliminación de bucles innecesarios y la vectorización automática.

Por otro lado, los compiladores Just-In-Time (JIT), utilizados en lenguajes como Java o Python (PyPy), aplican optimizaciones dinámicas en tiempo de ejecución, analizando los patrones de ejecución reales para ajustar la generación de código. Estas técnicas permiten alcanzar un rendimiento comparable al del código compilado estáticamente.

Conclusión

Las representaciones intermedias constituyen el núcleo de los compiladores modernos. Su diseño determina la calidad y el tipo de optimizaciones posibles. A través de estructuras como el AST, TAC o SSA, los compiladores logran transformar código fuente en versiones más eficientes sin alterar su semántica. El estudio de estas fases no solo es fundamental para el desarrollo de compiladores, sino también para entender cómo los programas alcanzan niveles de rendimiento óptimos.

Referencias

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores: principios, técnicas y herramientas.* Pearson Educación.

Appel, A. W. (2002). *Modern Compiler Implementation in C (2nd ed.).* Cambridge University Press.

Cooper, K. D., & Torczon, L. (2012). *Engineering a Compiler (2nd ed.).* Morgan Kaufmann.

Wikipedia. (2024, junio 20). Código intermedio. En Wikipedia, La enciclopedia libre. https://es.wikipedia.org/wiki/C%C3%B3digo_intermedio