Chapter 1

Análise de Fluxo de Dados

1.1 Objetivos

- Revisar os conceitos de Grafos de Fluxo de Controle (CFG).
- Compreender o propósito e a importância da Análise de Fluxo de Dados para otimizações globais.
- Estudar os princípios matemáticos e lógicos das Equações de Fluxo e Funções de Transferência.
- Analisar em detalhes os três principais problemas clássicos: *Reaching Definitions*, *Liveness Analysis* e *Available Expressions*.
- Entender o algoritmo iterativo de *Worklist* para resolução das equações de fluxo.

1.2 Introdução

No capítulo anterior, estudamos otimizações que ocorrem no escopo estrito de um único Bloco Básico (Análise Local). Embora eficazes, essas otimizações são míopes: elas não enxergam as consequências de saltos e desvios de controle. Para que o compilador realize otimizações **globais** (que abrangem múltiplos blocos básicos e laços de repetição), ele precisa compreender como as informações e valores fluem através de todo o programa.

A **Análise de Fluxo de Dados** (Data-Flow Analysis) é a técnica clássica de coleta de informações sobre a dinâmica do programa em tempo de execução, realizada de forma estática (em tempo de compilação). Através dela, modelamos o programa como um **Grafo de Fluxo de Controle (CFG)**, onde cada nó é um bloco básico e cada aresta representa uma transição de fluxo (como um `if` ou a volta de um `while`).

1.3 O Arcabouço Teórico (Data-Flow Framework)

Para raciocinarmos formalmente sobre o fluxo de dados, definimos dois pontos virtuais para cada bloco básico B:

- **IN[B]**: A informação disponível imediatamente *antes* da primeira instrução do bloco.
- **OUT[B]**: A informação resultante imediatamente *após* a última instrução do bloco.

O comportamento de cada bloco B é modelado por uma **Função de Transferência** f_B, que recebe a informação de entrada e produz a informação de saída (ou vice-versa, no caso de análises retrógradas). O arcabouço matemático é composto por $\langle V, \sqcap, F \rangle$:

1. **Domínio** V: Um conjunto de valores estruturado como um semirreticulado (*semilattice*), representando o "tipo" de informação que estamos coletando (ex: conjuntos de variáveis).
2. **Operador de Confluência** ($\sqcap$): Quando múltiplos caminhos do CFG se encontram (ex: o fim do ramo `then` e do ramo `else`), precisamos unir as informações usando o operador *meet*.
3. **Família de Funções** F: As funções $f_B : V \to V$, que devem possuir propriedade de monotonicidade para garantir a convergência do algoritmo.

Na vasta maioria das análises (conhecidas como *Bit-Vector Problems*), a função de transferência assume a forma clássica:

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B]) \tag{1.1}$$

Onde $GEN[B]$ representa a nova informação gerada pelo bloco e $KILL[B]$ representa a informação que foi invalidada pelo bloco.

1.4 Problemas Clássicos de Fluxo de Dados

Existem dezenas de análises de fluxo utilizadas por compiladores modernos (LLVM, GCC). Focaremos nos três problemas fundamentais que embasam quase todas as otimizações.

1.4.1 Reaching Definitions (Definições Alcançáveis)

Uma definição de uma variável `x` (ex: `d: x = y + z`) "alcança" um ponto p do programa se existe algum caminho no CFG de d até p onde `x` **não é redefinida**.

- **Direção:** *Forward* (do início para o fim do programa).
- **Operador Confluência:** União ($\cup$). Se a definição sobrevive por pelo menos um caminho, ela alcança o ponto.
- **Aplicações:** Utilizada para verificar o uso de variáveis não inicializadas (se uma definição "vazia" alcança um uso) e essencial para a Construção de SSA e Propagação de Constantes.

1.4.2 Liveness Analysis (Análise de Variáveis Vivas)

Uma variável está "viva" num ponto p se o seu valor atual pode ser lido em alguma instrução futura ao longo de algum caminho, antes de ser sobrescrita. Se a variável nunca mais for lida, ela está "morta".

- **Direção:** *Backward* (a informação flui do fim do programa para o início, pois "estar viva" depende do futuro).
- **Operador Confluência:** União ($\cup$).
- **Aplicações:** A aplicação primária é a **Alocação de Registradores**. Duas variáveis não podem ocupar o mesmo registrador se ambas estiverem vivas simultaneamente. Também é a base para a Eliminação de Código Morto (Dead Code Elimination).

1.4.3 Available Expressions (Expressões Disponíveis)

Uma expressão matemática (ex: `x + y`) está disponível num ponto p se ela foi computada em **absolutamente todos** os caminhos que chegam a p, e nem `x` nem `y` sofreram alterações desde então.

- **Direção:** *Forward*.
- **Operador Confluência:** Interseção ($\cap$). Exige um grau de certeza forte: a expressão tem que estar garantida, independentemente do caminho que o fluxo de execução tomou.
- **Aplicações:** Eliminação Global de Subexpressões Comuns (Global CSE). Se o compilador vê `z = x + y` e constata que `x + y` já está "disponível", ele pode reutilizar o valor computado anteriormente e poupar a CPU de calcular novamente.

1.5 O Algoritmo Iterativo de Worklist

A presença de ciclos (laços `while`, `for`) no CFG significa que não podemos resolver as equações passando pelos nós apenas uma vez em ordem topológica. A informação de um bloco dentro do laço precisa "dar a volta" e influenciar a entrada do mesmo laço.

A solução é o **Algoritmo de Ponto Fixo Iterativo**.

1 Inicializamos o IN/OUT de todos os blocos com o elemento *Top* do reticulado (conjunto vazio $\emptyset$ para União, conjunto universo U para Interseção).

2 Inserimos todos os blocos numa fila de trabalho (*Worklist*).

3 Enquanto a fila não estiver vazia, removemos um bloco B e:

- Computamos $IN[B]$ calculando o operador $\sqcap$ dos seus vizinhos de fluxo.
- Computamos $OUT[B]$ aplicando a função de transferência local $f_B(IN[B])$.
- Se o $OUT[B]$ resultante for **diferente** do valor que ele possuía antes, nós reinsirimos todos os sucessores de B de volta na fila.

Graças à propriedade de monotonicidade das equações de conjuntos, garantimos que o conjunto só cresce (ou só diminui) a cada iteração. Como o número de variáveis ou definições no programa é finito (um reticulado de altura finita), o algoritmo inevitavelmente atinge um ponto de estabilidade (Ponto Fixo Maximal - MFP), onde a fila se esvazia e a análise é concluída com sucesso.

1.6 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd Ed.). Addison-Wesley.
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler* (2nd Ed.). Morgan Kaufmann.
- Appel, A. W. (1998). *Modern Compiler Implementation in Java.* Cambridge University Press.