Chapter 1

Introdução ao LLVM IR

1.1 Objetivos

- Conhecer a infraestrutura LLVM e seu pipeline de compilação.
- Compreender a anatomia e a hierarquia da LLVM IR.
- Entender a relação da IR com conceitos de 3AC (Three-Address Code) e SSA.
- Manipular e visualizar código intermediário utilizando ferramentas como `clang`, `opt` e `llc`.

1.2 O Ecossistema LLVM

O LLVM (Low Level Virtual Machine) é atualmente a plataforma de referência para pesquisa e desenvolvimento de compiladores na indústria. Ele atua como base para compiladores de linguagens modernas como C/C++ (via Clang), Rust, Swift, Julia, Zig e Kotlin/Native, além de back-ends para GPUs como CUDA e ROCm.

O grande motivo de sua popularidade reside em sua arquitetura altamente **modular** (baseada em bibliotecas independentes) e em sua Representação Intermediária (IR) bem definida, que atua como uma linguagem universal para a otimização de código. Isso permite que engenheiros foquem em aspectos sintáticos da linguagem ou em análises complexas sem precisar reimplementar geradores de código de máquina para cada arquitetura (x86, ARM, RISC-V, WebAssembly, etc).

1.2.1 O Pipeline do LLVM

A arquitetura do LLVM divide o processo de compilação em três fases principais:

- **Frontend (ex: Clang, rustc):** Realiza a análise léxica, sintática e semântica do código-fonte e emite a representação **LLVM IR** (geralmente em formato de texto `.ll` ou bitcode binário `.bc`).

- **Passes de Otimização (`opt`):** Aplicam transformações sobre a IR (ex: inlining, simplificação de loops, eliminação de código morto). A IR é transformada sucessivamente de forma independente de arquitetura.
- **Backend (`llc`):** Traduz a LLVM IR otimizada para assembly nativo específico da arquitetura alvo (arquivos `.s`) ou diretamente para código objeto (`.o`).

1.3 Anatomia da LLVM IR

A LLVM IR assemelha-se a uma linguagem de montagem (assembly) tipada e portável, construída fundamentalmente sobre dois pilares conceituais: **3AC (Three-Address Code)** e **SSA (Static Single Assignment)**.

1.3.1 Características Fundamentais

- **3AC em SSA:** Cada instrução é simples, possui um único resultado, e as expressões complexas são achatadas.
- **SSA Nativa e a Função ϕ:** O formato SSA restringe o ciclo de vida da variável: cada registrador virtual é atribuído uma única vez. Ao garantir isso, criam-se ligações implícitas de Uso-Definição (*Use-Def*) de complexidade $O(1)$. Quando o fluxo de controle sofre bifurcações (ex: `if/else` ou loops), o SSA é contornado pelas instruções ϕ **(Phi)**, que selecionam magicamente o valor correto dependendo de **qual bloco básico** o fluxo de execução veio.
- **Fortemente Tipada:** Toda instrução especifica o tipo de seus operandos, tornando o código mais seguro para o otimizador.
- **Registradores Virtuais Infinitos:** Diferente do assembly real, a LLVM IR não possui limite de registradores. Eles são prefixados por `%` (ex: `%0`, `%result`, `%x`).

O LLVM também usa um truque para facilitar a vida dos frontends que geram a IR: o passe `-mem2reg`. Gerar código estritamente em SSA com nós ϕ diretamente da AST é complexo. Em vez disso, o frontend gera código burro alocando variáveis mutáveis na pilha com `alloca`, `load` e `store`. Posteriormente, o passe de otimização `-mem2reg` entra em ação, analisa o fluxo e promove automaticamente essas variáveis de memória para registradores SSA puros, inserindo os nós ϕ onde necessário.

1.3.2 Sistema de Tipos e Instruções Principais

Sendo fortemente tipada, os tipos fundamentais da LLVM IR incluem:

- `i1`: Booleano (1 bit), muito usado em condições.
- `i8`, `i32`, `i64`: Inteiros de vários tamanhos.
- `float`, `double`: Ponto flutuante.
- `ptr`: Ponteiro genérico (opaco).

- `void`: Procedimentos sem retorno.
- `<4 x float>`: Vetores SIMD (ex: SSE/AVX).

As instruções se dividem em categorias bem delineadas:

- **Aritmética:** `add`, `sub`, `mul`, `sdiv` (para inteiros); e as equivalentes com prefixo `f` para flutuantes (`fadd`, `fsub`).
- **Comparação:** `icmp` (*integer compare*) com modificadores como `eq` (igual), `ne` (diferente), `slt` (*signed less than*), `sgt` (*signed greater than*). Para reais, usa-se `fcmp`.
- **Memória:** `alloca` (aloca na pilha), `load` (lê para registrador), `store` (grava na memória).
- **Controle:** `br` (salto incondicional ou condicional), `ret` (retorna da função), `call` (invoca função), `phi` (resolve fluxos).

1.3.3 Hierarquia Lógica da LLVM IR

1 **Module (Módulo):** Representa uma unidade de compilação inteira (ex: `programa.c`). Agrupa funções, variáveis globais e metadados.

2 **Function (Função):** Contém argumentos e um grafo de **Blocos Básicos**.

3 **Basic Block (Bloco Básico):** Sequência linear de instruções, executada sempre da primeira à última sem interrupção. O bloco **obrigatoriamente termina** com uma instrução terminadora (`br` ou `ret`).

4 **Instruction (Instrução):** A unidade operacional mais atômica.

1.4 O Acesso à Memória: A Instrução GEP

Na LLVM IR, a aritmética de ponteiros para acessar vetores e structs (ex: acessar `A[3]`) é feita pela instrução **`getelementptr`** (GEP). Atenção à **Analogia do Carteiro**: A instrução GEP age como um *GPS*. Ela recebe o endereço de um Condomínio e uma coordenada. O GEP **não entra no apartamento** (`load`) e **não deixa o pacote** (`store`); ele apenas **calcula o endereço geográfico (o ponteiro)** exato da porta do apartamento.

Ao dessecarmos um acesso a array, como `int A[10]; A[3] = 42;`, a representação em LLVM exige dois índices:

```
1 %ptr = getelementptr [10 x i32], ptr %A, i32 0, i32 3
2 store i32 42, ptr %ptr
```

- `[10 x i32]` é a planta base arquitetural do tipo (o tipo original).
- `ptr %A` é o endereço do Condomínio.
- O **primeiro índice** (`i32 0`) orienta o ponteiro a andar zero posições gigantes (ou seja, ficar no próprio array base em vez de navegar em um array de arrays).

- O **segundo índice** (`i32 3`) adentra a estrutura, caminhando 3 posições lógicas do tipo `i32` até o "Apartamento 3".

O Combate ao Aliasing: A grande força do GEP reside na otimização. Em vez de somar bytes cegamente (ex: `ptr + 12`), a instrução diz ao otimizador que estamos navegando hierarquicamente em uma estrutura de tipos. Isso fornece à LLVM a **certeza matemática irrefutável** de que o índice 3 e o índice 4 de um mesmo array não colidem em memória (*alias*), destravando vetorizações severas.

1.5 Exemplos Práticos de Tradução

Vejamos como construções de alto nível são convertidas no pipeline LLVM.

1.5.1 A Jornada de uma Expressão

Uma expressão simples como `x = a + b * c` primeiramente vira uma AST (onde a adição é a raiz), em seguida vira um 3AC genérico e finalmente a LLVM IR estrita:

```
%1 = mul i32 %b, %c
%2 = add i32 %a, %1
store i32 %2, ptr %x
```

O processo "achata" a árvore da expressão.

1.5.2 Uma Função e o Controle de Fluxo (If/Else)

Para uma função que calcula o maior entre dois números (If `a > b` return `a` else return `b`):

```
define i32 @max(i32 %a, i32 %b) {
entry:
  %cmp = icmp sgt i32 %a, %b
  br i1 %cmp, label %then, label %else
then:
  br label %merge
else:
  br label %merge
merge:
  %result = phi i32 [%a, %then], [%b, %else]
  ret i32 %result
}
```

Note os três blocos (`entry`, `then`, `else`, e o de fechamento `merge`). O terminador `br i1 %cmp` bifurca a execução. No fechamento, o `phi` consolida qual valor de retorno utilizar preservando o formato SSA.

1.5.3 Loops em SSA

As iterações de laços de repetição também exigem nós ϕ, pois uma variável que atua como contador virá tanto da entrada inicial do loop quanto do fim da iteração anterior:

```
define i32 @somaAteN(i32 %n) {
entry:
  br label %loop
loop:
  %i    = phi i32 [0, %entry], [%next, %loop]
  %sum  = phi i32 [0, %entry], [%new,  %loop]
  %new  = add i32 %sum, %i
  %next = add i32 %i, 1
  %cmp  = icmp slt i32 %next, %n
  br i1 %cmp, label %loop, label %exit
exit:
  ret i32 %new
}
```

1.6 Ferramentas LLVM na Prática

O canivete suíço da plataforma envolve a integração do Frontend com otimizadores modulares. No terminal, o processo básico é o seguinte:

```
# 1. Gerar LLVM IR a partir de código C:
clang -S -emit-llvm programa.c -o programa.ll

# 2. Aplicar passes de otimização sobre a IR:
opt -S -mem2reg -instcombine programa.ll -o opt.ll

# 3. Gerar código assembly nativo a partir da IR:
llc opt.ll -o programa.s
```

Os passos de otimização (*passes*) são ativados no `opt`. Alguns essenciais incluem:

- `-mem2reg`: Promove uso de variáveis em memória (stack) para registradores puros SSA.
- `-instcombine`: Simplifica instruções mesclando operações redundantes.
- `-dce`: *Dead Code Elimination* (remove instruções que não afetam resultados).
- `-inline`: Embutimento de funções pequenas dentro de seus chamadores.

1.6.1 Visualizando o Grafo de Controle de Fluxo (CFG)

A infraestrutura fornece meios fáceis para gerar visualizações das funções compiladas, fundamentais para debug. Basta utilizar o `opt` junto ao utilitário *Graphviz*:

```
opt -dot-cfg programa.ll
dot -Tpdf .nomeDaFuncao.dot -o cfg.pdf
```

O resultado exibe diagramas interligados mapeando visualmente as instruções de salto `br` entre os blocos lógicos.

1.7 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools.*
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler.*
- Lattner, C. (2002). *LLVM: An Infrastructure for Multi-Stage Optimization.* Tese de Mestrado, Universidade de Illinois em Urbana-Champaign.