Aula 3 - Representação Intermediária do LLVM (LLVM-IR)

Daniel A. G. de Oliveira

19 de março de 2025

A principal referência para o assembly do LLVM (LLVM-IR) é o LangRef do próprio LLVM [1].

Uma outra forma que ajuda a entender como gerar LLVM-IR é utilizar o compilador clang para emitir código em LLVM-IR. Para isso, utilize o comando `clang -S -emit-llvm -O0 codigo.c` que vai gerar um arquivo `codigo.ll`.

Caso o clang em sua máquina seja antiga, instale uma versão mais nova, como clang-19, para produzir um LLVM-IR atualizado. Especialmente uma versão a partir da 15, pois a forma de usar ponteiros foi atualizada.

1 Identificadores

Identificadores são formados pelo seguinte padrão:

```
[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*
```

O primeiro caracter poder ser um '%' ou um '@'. O @ significa que o identificador é global (variáveis globais, funções) em seu escopo, enquanto o % implica que o identificador é local a determinada função. Portanto, identificadores como @xyz, ou @.minha-string são identificadores globais, enquanto %xyz e %xyz.2 são locais.

1.1 Variáveis globais

Variáveis globais, em LLVM-IR, são regiões alocadas em memória, e devem ser inicializadas. Portanto, serão basicamente um ponteiro para uma região de memória.

Listing 1: Exemplo de variável global do tipo inteiro

```
@x = global i32 2
```

O exemplo 1 define uma variável global chamada @x, do tipo inteiro com 32 bits, e inicializa com o valor dois.

2 Tipos

LLVM-IR é fortemente tipada, e o tipo inteiro pode possuir tamanho arbitrário (quantidade de bits). O inteiro mais comum, com 32 bits, já foi visto e é definido como `i32`. O formato para inteiros é `i<quantidade de bits>`, portanto uma variável declarada como `i1` é um inteiro de apenas um bit (usado em comparações, como se fosse um tipo booleano).

Ponto flutante é definido, primariamente, como `float` e `double`, e as instruções para esse tipo de dado são diferentes das instruções que operam sobre inteiros. Tipos agregados também são suportados, mas não vamos tratar disso aqui. Por fim, também existe o tipo `void`, que pode ser usado para definir uma função que não retorna valor algum.

3 Estrutura

Programas LLVM são compostos de módulos, onde cada módulo contém funções e variáveis globais. Cada função será composta de um ou mais blocos básicos, e cada bloco básico contém instruções.

3.1 Bloco Básico

Cada bloco básico inicia com um label, ou recebe de forma implicita um label, contém um conjunto de instruções e finaliza com uma instrução *terminadora*.

Um bloco básico tem uma entrada e uma saída (instrução terminadora), portanto não tem instruções de desvio 'no meio' do bloco. As instruções terminadores podem ser um desvio (*br*) ou retorno(*ret*), bem como outros casos como a instrução *unreachable*.

O trecho de código 2 possui três blocos básicos. O primeiro bloco básico começa com o rótulo de entrada *Test* e termina com uma instrução de desvio *br*. O segundo trecho inicia com o rótulo *IfEqual* e termina com um retorno. Por fim, o terceiro bloco básico inicia com o rótulo *IfUnequal* e termina com um retorno.

O trecho de código implementa uma comparação de igualdade entre os registradores %a e %b, caso sejam iguais, o bloco básico que inicia com *IfEqual* será executado e o valor 1 retornado. Caso não sejma iguais, o bloco que inicia com *IfUnequal* será executado e o valor zero retornado.

Listing 2: Exemplo de blocos básicos

```
Test:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

4 Funções

Funções são criadas com a palavra-chave *define*. Elas podem possuir um tipo de retorno e parâmetros de entrada. Parâmetros de entrada podem ser variáveis simples, ou ponteiros. Na verdade, muitos detalhes podem ser alterados, como convenção de chamada e outros. Porém, a intenção deste tutorial é ser bem simples.

O código 3 demonstra uma função simples, chamada @soma, que recebe dois parâmetros de entrada do tipo inteiro, e retorna a soma dos dois. Como funções tem o escopo global, seu identificador deve começar com @.

Listing 3: Exemplo de função

```
define i32 @soma(i32 %x, i32 %y) {
  %1 = add i32 %x, %y
  ret i32 %1
}
```

Para executar uma função usamos a instrução `call`. Existe, também, a instrução `invoke` onde é possível tratar erros e trabalhar com exceções, mas não iremos tratar disso. O códido em 4 tem um exemplo completo que utiliza a função soma e também imprime um inteiro na tela. Nesse código, as variáveis %1 e %2 são inteiros inicializados, respectivamente, com os valores 1 e 2. A função vai receber esses dois valores por cópia, e retornar a soma deles. Por fim, esse resultado é impresso na tela com uma quebra de linha no final.

Listing 4: Exemplo de função

```
declare i32 @printf(ptr noundef, ...)
@write_int = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define i32 @soma(i32 %x, i32 %y) {
  %1 = add i32 %x, %y
  ret i32 %1
}

define i32 @main() {
    %1 = add i32 0, 1
```

```
    %2 = add i32 0, 2
    %3 = call i32 @soma(i32 %1, i32 %2)
    %4 = call i32 (ptr, ...) @printf(ptr noundef @write_int, i32 %3)
    ret i32 0
}
```

5 Static Single Assignment (SSA)

LLVM-IR é uma linguagem do tipo Static Single Assignment (SSA). Isso significa que todo registrador em uma função possui apenas uma única atribuição (single Assignment), seu valor não é atualizado. Porém, visto de forma estática, e atribuições dentro de um laço de repetição são permitidas (daí vem a definição de Static). Veremos, posteriormente, essas atribuições dinâmicas e a instrução *phi* (ϕ). A linguagem estar na forma SSA permite que muitas técnicas de otimização sejam facilmente implementadas.

Em resumo, instruções que reescrevem no mesmo registrador devem ser reescritas para utilizar um novo registrador. Por exemplo, o código `x= 10; x = x + 1;` não é permitido em LLVM-IR pois a variável x possui duas atribuições. O seguinte trecho de código poderia ser traduzido como em 5, criando um novo registrador a cada atribuição.

Listing 5: Exemplo de função

```
%x.0 = add i32 0, 0
%x.1 = add i32 %x.0, 1
```

6 Memória: alloca, store e load

Uma certa *facilidade* que LLVM-IR ofereece é a gerência de memória de forma automática, sem que o usuário se preocupe na forma que a pilha, ou alguma estrutura de memória, seja organizada. Assim, a instrução `alloca` vai retornar um ponteiro para uma região de memória que será automaticamente desalocada ao final da função.

Em conjunto com essa instrução, vamos usar as instruções `store` e `load` para armazenar e fazer a leitura dessa região de memória.

Como não vamos usar tipos agregados, arrays e etc., um simples store e load seria o suficiente. Porém, ao usar tipos agregados, onde aritmética de ponteiros é importante para calcular a posição de um elemento na memória, LLVM-IR oferece uma função que faz essa aritmética (`getelementptr`), que não iremos cobrir neste texto.

O trecho de código 6 demonstra como alocar um espaço na memória, armazenar o valor 15 e fazer a leitura para uma nova variável. Depois, esse valor da nova variável é incrementado e salvo novamente na posição de memória. Note que para acesso a memória, podemos fazer quantos `store` forem necessários para a mesma posição (ou ponteiro) sem quebrar a regra de SSA de variáveis.

Listing 6: Exemplo de função

```
%x = alloca i32
store i32 15, ptr %x
%1 = load i32, ptr %x
%2 = add i32 %1, 1
store i32 %2, ptr %x
```

7 Condicional e Desvio

Para executar condicionais podemos usar as instruções `icmp` para inteiros, e `fcmp` para ponto-flutuante. A instrução `icmp` permite usar as seguintes comparações:

1. eq: igual

2. ne: não é igual
3. ugt: maior que, sem sinal
4. uge: maior ou igual, sem sinal
5. ult: menor que, sem sinal
6. ule: menor ou igual, sem sinal
7. sgt: maior que, com sinal
8. sge: maior ou igual, com sinal
9. slt: menor que, com sinal
10. sle: menor ou igual, com sinal

A instrução `fcmp` é similar, mas em vez de considerar com e sem sinal ('u' e 's'), ela utiliza ordered e unordered ('o' e 'u') que tem considerações diferentes em relação a números *Not a Number (NaN)*.

Essas instruções de comparação retornam um valor do tipo `i1`, ou seja, verdadeiro (um) ou falso (zero).

Para executar um desvio, vamos utilizar esse resultado do tipo `i1` e a instrução `br`. Essa instrução vai pular para um primeiro rótulo caso o valor da variável do tipo `i1` seja verdadeiro, ou então vai pular para o segundo rótulo caso contrário. O trecho de código 7 implementa um `if` simples que verifica se dois valores (%a e %b) são iguais, caso sejam iguais o valor 1 é retornado, caso contrário o valor 0 é retornado.

Listing 7: Exemplo de função

```
Test:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

No trecho de código 7, ainda podemos observar que três blocos básicos foram criados. Lembrando que um bloco básico inicia com um rótulo (criado sem o uso do '%' no início, porém é referenciado com o uso do '%'), e termina com uma intrução de término, como `br` ou `ret`.

8 Exemplo completo: Fatoria recursivo

Laços de repetição serão tratados posteriormente, porém, com tudo o que já vimos é suficiente para criar programas recursivos. No exemplo 8, o fatorial é computado de forma recursiva na função `@fat`.

Neste exemplo, iniciamos o programa com dois `declare` que avisa ao módulo que usaremos duas funções externas, no caso a scanf e printf da libc. Ainda criamos duas strings que são constantes, a primeira é a string usada para o scanf que informa sobre a leitura de um inteiro ("%d"); a segunda é similar, mas tem uma quebra de linha no final ("%d\n"), porém usar o valor hexadecimal para o caracter de quebra de linha (0A).

A função scanf espera um ponteiro para cada valor que será lido, já a função printf recebe valores por cópia para o que será impresso na tela. As duas funções são chamadas pela instrução `call`. A função scanf vai salvar, na posição de memória apontada pelo ponteiro %1, o valor lido do teclado. Já a função printf vai imprimir na tela o valor salvo na variável %4, que possui o retorno da função `@fat`.

Listing 8: Exemplo de função

```
declare i32 @printf(ptr noundef, ...)
declare i32 @__isoc99_scanf(ptr noundef, ...)
@read_int = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@write_int = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
```

```
define i32 @fat(i32 %x) {
  %1 = icmp sle i32 %x, 0
  br i1 %1, label %retFim, label %retRec
retFim:
  ret i32 1
retRec:
  %2 = sub i32 %x, 1
  %3 = call i32 @fat(i32 %2)
  %4 = mul i32 %x, %3
  ret i32 %4
}

define i32 @main() {
    %1 = alloca i32
    %2 = call i32 (ptr, ...) @__isoc99_scanf(ptr @read_int, ptr %1)
    %3 = load i32, ptr %1
    %4 = call i32 @fat(i32 %3)
    %5 = call i32 (ptr, ...) @printf(ptr noundef @write_int, i32 %4)
    ret i32 0
}
```

Referências

[1] LLVM, "Llvm language reference manual." https://llvm.org/docs/LangRef.html, 2025. [Online; accessed 18-March-2025].