Geração de Código

Guilherme B. Del Rio[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR) – Campo Mourão, PR – BRASIL

guilhermerio@alunos.utfpr.edu.br

Abstract. *This document describes the implementation of the code generator, which is the last step of the compiler, through the trees created from the input code and after its pruning in the previous step, when going through it, a code is created through the LLVM library, in assembly that can be used for transformation in other programming languages, as a result we have this file that, when executed, returns the result of the input file.*

Resumo. *Este documento descreve a implementação de gerador de código, que é a ultima etapa do compilador, através da árvores criada à partir do código de entrada e depois de sua poda na etapa anterior, ao percorre-la é criado através da biblioteca LLVM, um código em assembly que pode ser utilizado para transformação em outras linguagens de programação, como resultado temos este arquivo que ao ser executado retorna o resultado do arquivo de entrada.*

1. Introdução

Este trabalho descreve o gerador de código do compilador desenvolvido, o gerador de código é a ultima etapa de um compilador, após o tratamento dos erros e da poda da árvore, é possível compila-lo e gerar seu código respectivo em linguagem de máquina (assembly) que então poderá ser passado para outras linguagens.

2. Detalhes da Implementação

2.1. Verificação dos resultados semânticos

Primeiramente é necessário a verificação da saída da análise semântica, onde detectamos todos os erros (caso hajam) no código de entrada recebido, caso houver mensagens de erro retornadas não será feita a geração de código e o algoritmo será terminado.

2.2. Inicialização da LLVM

Após a verificação de erros, é iniciada a LLVM através dos comandos:

- llvm.initialize()
- llvm.initialize_all_targets()
- llvm.initialize_native_target()
- llvm.initialize_native_asmprinter()

O LLVM, ou Low Level Virtual Machine, é o responsável pela leitura de um código intermediário e geração de um código otimizado para código de montagem, no caso, assembly. Este processo é possível através da árvore de execução do código, ao percorre-la é criado seu respectivo comando.

Para que estas operações sejam salvas de forma correta é criado um módulo, onde serão salvos em blocos todas as funções geradas.

2.3. Inicialização das funções de controle

Com o modulo feito, para que seja feita a leitura de um inteiro ou ponto flutuante para a linguagem de máquina é utilizado duas funções de leitura:

- leiaInteiro = ir.Function(module, ir.FunctionType(ir.IntType(32), []), name=”leiaInteiro”)
- leiaFlutuante = ir.Function(module, ir.FunctionType(ir.FloatType(), []), name=”leiaFlutuante”)

Podemos observar que estas funções são conectadas com o módulo criado anteriormente, e tem o tipo de leitura definido logo em seguida, como são apenas para leitura de variáveis, não possuem argumento, no final é declarado seu respectivo nome.

O mesmo é feito para a escrita de variáveis de ponto flutuante, no caso, utilizando estas duas funções:

- escrevaInteiro = ir.Function(module, ir.FunctionType(ir.VoidType(), [ir.IntType(32)]), name=”escrevaInteiro”)
- escrevaFlutuante = ir.Function(module, ir.FunctionType(ir.VoidType(), [ir.FloatType()]), name=”escrevaFlutuante”)

Com a criação similar a anterior, apenas diferindo no final onde é passado o argumento de escrita, se é ponto flutuante ou inteiro.

2.4. Geração de código

Para o começo da geração do código é chamada a função *generate_code* e passado a árvore que será percorrida onde será verificado se os primeiros nós filhos são declarações de variáveis ou declarações de funções. Caso seja uma variável, significa que ela foi declarada em escopo global, sendo assim chamado a função de declaração de variáveis globais.

Esta função se inicia salvando todos os dados da variável, sendo eles nome, tipo e se tiver, sua dimensão, para que sejam convertidos para o formato da LVM, e sejam definidos como globais e colocados no módulo.

Semelhante às variáveis, as funções globais tem seu tipo, retorno e bloco criados, dentro do bloco caso hajam variáveis locais, estas são declaradas em seu bloco, caso não é feita a finalização da criação da função. Caso houver condicionais como *se, :=, retorna, leia, escreva e repita*, será chamada uma função de controle que identificará qual a condição, e então a colocará no arquivo com o código de montagem respectivo.

3. Exemplo entrada e saída

3.1. Entrada

Como arquivo de entrada é enviado um arquivo com a extensão tpp, e para executa-lo utilizamos o comando:

- python3 codeGenerator.py ./geracao-codigo-testes/gencode-001.tpp

No caso, foi utilizado o código de exemplo *gencode-001.tpp* Figura 1.

```
{Declaração de variáveis}
inteiro: a

inteiro principal()
    inteiro: b

    a := 10

    b := a

    retorna(b)
fim
```

Figura 1. Código de entrada em TPP

3.2. Saída

Como saída temos um arquivo com extensão *.ll* com o mesmo nome do arquivo de entrada, dentro deste arquivo há o código gerado no formato assembly, além deste código ser mostrado no terminal.

Para executa-lo utilizamos o *clang* e definimos um nome para o executável que será gerado, então iniciamos o executável para que ele retorne o resultado (Figura 2) devido através dos comandos:

- clang ./geracao-codigo-testes/gencode-002.ll -o gencode
- echo $?

```
Arquivo de destino: ./geracao-codigo-testes/gencode-001.tpp.cut.unique.ast.png
node.name retorna
return_element b
var %".6" = load i32, i32* %"b"
; ModuleID = "gencode-001.bc"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

declare void @"escrevaInteiro"(i32 %".1")

declare void @"escrevaFlutuante"(float %".1")

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

@"a" = common global i32 0, align 4
define i32 @"main"()
{
entry:
  %"b" = alloca i32, align 4
  %"expression" = add i32 0, 10
  store i32 %"expression", i32* @"a"
  %".3" = load i32, i32* @"a"
  %"expression.1" = add i32 0, %".3"
  store i32 %"expression.1", i32* %"b"
  br label %"exit"
exit:
  %".6" = load i32, i32* %"b"
  ret i32 %".6"
}
```

Figura 2. Código de saída no terminal

4. Conclusão

Através deste trabalho foi possível aprender sobre a geração de código de um compilador e como ele traduz os dados para a linguagem de máquina através da árvore resultante de um código. Também foi possível aprender sobre a biblioteca LLVM, que traduz de forma otimizada os dados.

Referências

Gonçalves, R. A. (2021a). Aula 017: Geração de código intermediário, técnicas, estruturas de dados e de controle, funções e procedimentos (slides).

Gonçalves, R. A. (2021b). Vídeo aula 18: Geração de código intermediário llvm-ir).