Geração de Código Intermediário

Mateus Tomoo Yonemoto Peixoto

DACOM – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil
{mateustomoo}@gmail.com

Abstract. This paper describes the development of the last part of the discipline implementation work, intermediate code generation for a compiler being designed for the T++ programming language.

Resumo. Este artigo descreve o desenvolvimento da última parte do trabalho de implementação da disciplina, a geração de código intermediário para um compilador projetado para a linguagem de programação T++.

1. Introdução

Um compilador exige alguns passos até que um código intermediário seja obtido. Esses passos são divididos em análise léxica, análise sintática, análise semântica e geração de código.

A Geração de Código é a fase final de um compilador e também a fase mais complexa, pois não depende exclusivamente da linguagem-fonte, mas também de informações e detalhes da arquitetura-alvo, da estrutura do ambiente de execução e do Sistema Operacional da máquina-alvo.

Para o desenvolvimento deste trabalho, foi utilizado a linguagem de programação Python (versão 3.6.2) e o LLVMLite.

2. Geração de Código Intermediário

Uma estrutura de dados que represente o programa-fonte durante a tradução é denominada Representação Intermediária (IR).

Chamamos de código intermediário a forma de representação intermediária gerada a partir da árvore sintática que se assemelhe melhor ao código-alvo. O código intermediário pode assumir muitas formas, porém todas representam a linearização da árvore sintática (representação sequencial da árvore sintática). Nele também pode conter informações detalhadas sobre a máquina-alvo e o ambiente de execução.

3. LLVM – IR

Os principais componentes de um código LLVM-IR são:

- Módulo – representa um arquivo com código-fonte ou uma unidade de tradução, onde todo o restante do código deve estar dentro de um módulo.
- Funções – módulos contém funções, que são declaradas com seus nomes e argumentos. Uma função é um container de Basic Blocks.
- Basic Block (bloco básico) – pedaço contíguo de instruções.
- Instruções – operação única expressa em um código. Operações como adição, subtração ou até mesmo uma instrução de load ou store.

4. LLVMLite

Nesta seção será mostrado e explicado algumas das funções presente no LLVMLite que foram utilizadas na construção do trabalho para a geração de código intermediário.

- **`llvmlite.ir.GlobalVariable(module, type, name)`** – função utilizada para declaração de variável global.
- **`llvmlite.ir.Constant(type, constant)`** – função que atribui uma constante a variável.
- **`llvmlite.ir.Function(modulo, type, name)`** – função utilizada para definição de função.
- **`llvmlite.ir.IRBuilder(block)`** – função que cria um novo builder, que inicia ao final do bloco.
- **`IRBuilder.append_basic_block(block_name)`** – função que adiciona um bloco básico, com o nome passado.
- **`IRBuilder.position_at_end(block)`** – função que posiciona ao final do bloco básico.
- **`IRBuilder.branch(block)`** – função que faz um “jump” até o alvo passado.
- **`IRBuilder.cbranch(condition, truebr, falsebr)`** – função que faz um “jump” para “truebr” ou “falsebr” dependendo da condição.
- **`IRBuilder.ret(value)`** – função que retorna um valor ta função atual.

5. Implementação

Para a implementação da geração de código intermediário, a árvore abstrata é percorrida. A medida que a árvore é percorrida, a geração de código é feita.

A seguir é mostrado alguns exemplos de funções implementadas para que a geração de código ocorra.

```
def declaracao_variaveis(self, node):
    var_type = node.child[0].type
    lista_vars = self.lista_variaveis(node.child[1])

    for varname in lista_vars:
        if self.scope == "global":
            if var_type == "inteiro":
                var = ir.GlobalVariable(self.module, ir.IntType(32), varname)
                var.initializer = ir.Constant(ir.IntType(32), 0)
            else:
                var = ir.GlobalVariable(self.module, ir.FloatType(), varname)
                var.initializer = ir.Constant(ir.FloatType(), 0)
        else:
            if var_type == "inteiro":
                var = self.builder.alloca(ir.IntType(32), name=varname)
            else:
                var = self.builder.alloca(ir.FloatType(), name=varname)
            #num = ir.Constant(ir.FloatType(), 0)
            #self.builder.store(num, var)

        var.linkage = "common"
        var.align = 4
        self.symbols[self.scope + "-" + varname] = ["variavel", varname, False, False, var_type, self.scope, var]
```

"*Imagem01. Declaração de variáveis.*"

Na Imagem01, é mostrado a função que tem como objetivo a declaração de variáveis. Funciona da seguinte maneira: percorre-se uma lista de variáveis e verifica-se se a variável possui escopo "global" ou não. Caso seja "global", utiliza-se a função "GlobalVariable" do llvmlite (importada como "ir") e inicializa a variável com uma constante 0 (zero) e o seu tipo (inteiro ou flutuante). Caso a variável pertença a algum escopo que não seja "global", essa variável é alocada. Logo depois, é adicionado na tabela de símbolos (da mesma forma que na semântica, porém com uma posição a mais que é a variável no llvm).

```
def declaracao_funcao(self, node):
    offset = 0
    if len(node.child) == 1:
        retorno = None
        llvm_retorno = ir.VoidType()
    else:
        offset = 1
        if node.child[0].value == "inteiro":
            retorno = "inteiro"
            llvm_retorno = ir.IntType(32)
        else:
            retorno = "flutuante"
            llvm_retorno = ir.FloatType()

    cabeca = self.cabecalho(retorno, node.child[offset])
    lista_param = cabeca[0]
    llvm_params = []
    corpo_node = cabeca[1]
    func_nome = cabeca[2]

    for param in lista_param:
        tipo = param[0]

        if tipo == "inteiro":
            llvm_params.append(ir.IntType(32))
        else:
            llvm_params.append(ir.FloatType())

    llvm_func_type = ir.FunctionType(llvm_retorno, llvm_params)
    self.func = ir.Function(self.module, llvm_func_type, name=func_nome)
    scope_nome = func_nome
    self.scope_list.append(scope_nome)
    self.scope = scope_nome

    entry_block = self.func.append_basic_block('entry')

    self.builder = ir.IRBuilder(entry_block)
    self.bloco = entry_block
    self.symbols[func_nome] = ["funcao", func_nome, lista_param, llvm_retorno, 0, scope_nome]
    self.corpo(corpo_node)
```

"*Imagem02. Declaração de função*."

A Imagem02 mostra a função que faz as declarações de funções. Como mostrado na análise sintática, essa função deve gerar tipo e cabeçalho ou apenas cabeçalho. A variável "offset" é encarregada de diferenciar isso. Caso a função tenha tipo, é verificado qual o tipo dela.

A função cabeçalho retorna 3 coisas: a lista de parâmetros, o corpo da função e seu ID. Desta forma, as variáveis "lista_param", "corpo_node" e "func_nome" recebem isso respectivamente. Logo em seguida é verificado o tipo dos parâmetros. Depois, a função "FunctionType" do llvmlite é chamada, passando o tipo de retorno da função e seu parâmetro já na forma do llvmlite. Assim, cria-se um bloco, chamada de "entry", onde ficará todo o corpo da função. No final, assim como na declaração de variáveis, a função também é adicionada na tabela de símbolos, apenas para um melhor controle e entendimento.

6. Exemplos

Nesta seção, será mostrado alguns exemplos de entrada (código escrito na linguagem T++) e suas respectivas saídas para a geração de código intermediário.

```
1  inteiro: n
2  inteiro: soma
3
4  inteiro principal()
5      n := 10
6      soma := 0
7      repita
8          soma := soma + n
9          n := n - 1
10     até n = 0
11
12     retorna(0)
13 fim
```

"Imagem03. Exemplo de entrada com repita."

```
declare float @"leia_float"()

declare i32 @"leia_int"()

declare i32 @"escreve_float"(float %".1")

declare i32 @"escreve_int"(i32 %".1")

@"n" = common global i32 0, align 4
@"soma" = common global i32 0, align 4
define i32 @"principal"()
{
entry:
  store i32 10, i32* @"n"
  store i32 0, i32* @"soma"
REPITA:
  %".4" = load i32, i32* @"soma"
  %".5" = load i32, i32* @"n"
  %".6" = add i32 %".4", %".5"
  store i32 %".6", i32* @"soma"
  %".8" = load i32, i32* @"n"
  %".9" = sub i32 %".8", 1
  store i32 %".9", i32* @"n"
  %".11" = load i32, i32* @"n"
  %".12" = icmp eq i32 %".11", 0
  br i1 %".12", label %"FIM-REPITA", label %"REPITA"
FIM-REPITA:
  ret i32 0
}
```

"Imagem04. Exemplo de saída com repita."

```
2   inteiro: a
3
4   inteiro principal()
5       inteiro: b
6
7       a := 10
8
9       b := a
10
11      retorna(0)
12  fim
```

“Imagem05. Exemplo de entrada simples.”

```
@"a" = common global i32 0, align 4
define i32 @"principal"()
{
entry:
  %"b" = alloca i32, align 4
  store i32 10, i32* @"a"
  %".3" = load i32, i32* @"a"
  store i32 %".3", i32* %"b"
  ret i32 0
}
```

“Imagem06. Exemplo de saída simples.”

7. Referências

LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

PLY (Python Lex-Yacc). Disponível em: <http://www.dabeaz.com/ply/>.

LLVMLite Documentation. Disponível em: http://llvmlite.readthedocs.io/en/latest/user-guide/ir/values.html