Geração de Codigo LLVM - π Framework

Fernando Mendes

Universidade Federal Fluminense

fjmendes1994@gmail.com

October 25, 2018

Overview

1. LLVM
   - Introdução
   - Para que estamos usando
   - IR
2. π-lib $\rightarrow$ *LLVM IR*
   - Introdução
   - Expressões aritméticas e booleanas
   - Variáveis locais
   - Loops
   - Geração de código LLVM para π-lib em Python

LLVM - Introdução

LLVM é um conjunto de ferramentas utilizado na construção de compiladores para produção de código intermediário, otimização e produção de código de máquina.

LLVM - Para que estamos usando?

Faremos uso do LLVM como framework para geração de código de máquina partindo de sua representação intermediária, provendo um front-end que transforma código π-lib em LLVM IR, tirando proveito assim de uma seria de otimizações que o LLVM é capaz de realizar partindo de sua IR.

LLVM - IR

- Mescla características de linguagens de alto nível com de linguagem baixo nível
  - Tipagem forte
  - Funções e Módulos
  - Blocos e Ramificações

LLVM - IR

- Baseada em SSA(Static single assignment)
  - Cada variável é atribuída exatamente uma vez durante a escrita do programa
  - Facilita algumas otimizações

Introdução

- Transformar contruções π-lib para LLVM IR
- Expressões
  - Id(String)
  - Add(Exp, Exp)
  - Mul(Exp, Exp)
  - Eq(Exp, Exp)
  - Not(BoolExp)
- Comandos
  - Assing(Id(String), Exp)
  - Loop(BoolExp, Cmd)
  - Cseq(Cmd)
- Declarações
  - Blk(Decl, Cmd)
  - Bind(Id, Ref)
  - Ref(Exp)
  - Dseq(Decl, Decl)

Estrutura de um programa escrito em LLVM IR

Como não ainda não possuímos funções e módulos, todo nosso código estará contido Dentro de um módulo e uma função principal que será nosso ponto de entrada e onde colocaremos nossas instruções.

```
; ModuleID = "main_module"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = ""

define i64 @"main_function"()
{
entry:
  ...

  ret i64 0
}
```

Expressões aritméticas e booleanas

Add(Num(10),Num(1))

```
%"tmp_add" = add i64 10, 1
```

Mul(Id('z'),Num(10))

```
%"tmp_mul" = mul i64 %"val", 10
```

Eq(Id('x'),Num(1))

```
%"temp_eq" = icmp eq i64 %"val.1", 0
```

Not(Id('z'))

```
%"temp_not" = xor i1 %"temp_eq", -1
```

Variáveis locais

Bind(Id('z'),Ref(Num(1)))

```
%"ptr" = alloca i64
store i64 1, i64* %"ptr"
```

Id('z')

```
%"val" = load i64, i64* %"ptr"
```

Assign(Id('z'), Num(0))

```
store i64 %"tmp_mul", i64* %"ptr"
```

Loops

Loop(
  Not(Eq(Id('y'), Num(0))),
  Assign(Id('y'), Sub(Id('y'), Num(1)))))

```
loop:
  %"val" = load i64, i64* %"ptr"
  %"temp_eq" = icmp eq i64 %"val", 0
  %"temp_not" = xor i1 %"temp_eq", -1
  %"val.1" = load i64, i64* %"ptr"
  %"tmp_sub" = sub i64 %"val.1", 1
  store i64 %"tmp_sub", i64* %"ptr"
  br i1 %"temp_not", label %"loop", label %"after_loop"
after_loop:
  ...
```

Geração de código LLVM para π-lib em Python

`https://github.com/ChristianoBraga/PiFramework/blob/master/python/pillvm.py`

Fatorial π-lib

```
Blk(
   Bind(Id(z), Ref(Num(1))),
   Blk(
      Bind(Id(y), Ref(Num(10))),
      Loop(
         Not(Eq(Id(y), Num(0))),
         CSeq(
            Assign(Id(z), Mul(Id(z), Id(y))),
            Assign(Id(y), Sub(Id(y), Num(1)))))))
```

Fatorial LLVM IR

```
; ModuleID = "main_module"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = ""

define i64 @"main_function"()
{
entry:
  %"ptr" = alloca i64
  store i64 1, i64* %"ptr"
  %"ptr.1" = alloca i64
  store i64 10, i64* %"ptr.1"
  br label %"loop"
loop:
  %"val" = load i64, i64* %"ptr.1"
  %"temp_eq" = icmp eq i64 %"val", 0
  %"temp_not" = xor i1 %"temp_eq", -1
  %"val.1" = load i64, i64* %"ptr"
  %"val.2" = load i64, i64* %"ptr.1"
  %"tmp_mul" = mul i64 %"val.1", %"val.2"
  store i64 %"tmp_mul", i64* %"ptr"
  %"val.3" = load i64, i64* %"ptr.1"
  %"tmp_sub" = sub i64 %"val.3", 1
  store i64 %"tmp_sub", i64* %"ptr.1"
  br i1 %"temp_not", label %"loop", label %"after_loop"
after_loop:
  ret i64 0
}
```