LLVM 影响下的编译器新生态

高一 4 班 - 梁韬 - GitHub@Origami404

2019 年 2 月 14 日

1 前言

21 世纪以来, 新的编程语言的产生速度越来越快, 包括 DSL 在内的全新编程语言对于编译技术的质和量的要求也越来越高. 而反观 Intel 等硬件厂商, 指令集也越发复杂, 同时编译优化技术特别是 GC 及 JIT 技术的发展对于编译器作者的水平也提出了极高的要求. 面对这种困难, LLVM 诞生了.

LLVM 是一个**自由软件项目**, 是一套编译器基础设施, 基于 C++. LLVM 的命名最早源于所谓**底层虚拟机 (Low Level Virtual Machine)** 但随着其本身的发展及壮大, 现在的 LLVM 只是单纯的作为一个无意义的名称, 用于统称 LLVM 项目下产生的各种工具, 如 LLVM IR, lldg, LLVM C++ Standard Library 等.

本综述主要叙述 LLVM 对于编译器编写这一传统领域所带来的影响及基于 LLVM 编写编译器前端的基本流程.

2 LLVM 对传统编译编译器编写的影响

传统的编译器实现一般包括两大部分七大模块. 第一个部分是**分析**. 分析部分将源代码分解为多个组成要素, 并在这些组成要素上添加语法结构. 在此部分编译器将捕获并处理全部的词法与语法错误, 收集整理与代码有关的其他信息, 最后构造出一种**中间表示 (IR)**.

第二个部分是**综合**. 在此部分编译器开始着手优化并产生目标代码. 在这一部分编译器可能使用某些巧妙的算法进行机器无关的代码优化, 又或者使用某种依赖于目标平台或目标语言特性的优化机理进行优化, 最后产生的便是目标代码.

在此划分之上编译器又被按功能分为七个模块, 分别是**词法分析**, **语法分析**, **语义分析**, **中间代码生成**, **机器无关代码优化**, **代码生成**, **机器相关代码生成**. 其中词法分析, 语法分析, 语义分析及中间代码生成属于分析部分; 机器无关代码优化, 代码生成, 机器相关代码生成属于综合部分.

读者不难发现, IR 作为联系前端和后端的形式, 在编译器编写中具有重要作用. 在一门语言的实现中, 除了语言本身的语法定义, 就属 IR 的定义最为重要. 泛观世界上成熟的编译器工具集和 VM 如 GCC, LLVM, JVM, 甚至是 LuaVM 都定义了极为成熟和在一定程度上泛用的 IR 表示. 这些 IR 为平台的二次开发奠定了基础, 也是其能流行于世界的重要原因. 而相比起其他编译平台, LLVM 的优势在于其 IR 的开发一开始就基于一个目标—-多语言通用, 多范式支持, 多用途使用. 因而在快速实现编译器时 LLVM 具有天然的优势, 而其完全透明, 类型完备的 IR 也为基于该语言的开发工具提供了实现包括语法高亮, 语义代码补全等面给用户的功能的极大便利. 诸如 Apple 的 Xcode, Microsoft 的 VSCode 的 C++ 插件等都支持对 LLVM 编译出的 IR 进行利用而改善用户体验的功能. 这也是 LLVM 的一大优势.

3 利用 LLVM 实现编译器前端的基本流程

LLVM C++ 库对中间代码生成, 优化及跨平台机器代码生成提供了简洁易用的接口. 在一般的编写流程中, 开发人员主要编写词法与语法分析器, 即俗称的 Scanner 与 Parser, 而后通过在 Parser 得出的 AST 继承体系中加入调用 LLVM 生成 IR 的方法来生成 LLVM IR, 极大的简化了开发流程. 对于某些对速度要求不高的需求, 开发人员甚至可以使用诸如 Bison 和 flex 等分析器生成器生成语法词法分析器, 从而极大地降低了编译器开发的门槛. 同时也因为 LLVM IR 的生成一般基于 AST, 而在 AST 层面往往可以对语言作出极其灵活的修改而不改变用户代码的感知, 故这种方法也具有极高的灵活性, 甚至在某些情况下能实现超出 Python 的 metaclass 和 Java 的 Reflection 的令人惊奇的灵活修改, 能在源代码层面实现原本许多嵌入式语言中在宿主层面才实现的特性.

为更好的介绍 LLVM C++ 接口, 在这里实现一个简短的带变量二元运算解析. 我们的目标是支持诸如:

```
    def x = 2.44
    def y_s = 300.45 + x
    def c = (x + y_s)*3
    c * 2 + x / y_s
```

的变量赋值与二元运算. 二元运算支持 +, -, *, / 与 (), 为了简便所有量的类型均为 double. 更严谨的, 此语言的 BNF 如下: (也是一个 LL(1) 文法)

$$programmer \rightarrow define \mid expression \tag{1}$$

$$define \rightarrow \textbf{def}\ Id = expression \tag{2}$$

$$expression \rightarrow primary\ [binop] \tag{3}$$

$$primary \rightarrow Id \mid Num \mid (\ expression\ ) \tag{4}$$

$$binop \rightarrow primary\ Op\ binop \tag{5}$$

由于不是重点, 这里直接给出 AST 以及 Scanner 和 parser 的定义:

```
// 编译命令
// clang++ -g -O3 *.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core`
#include "llvm/ADT/APFloat.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <cstdlib>
#include <map>
#include <memory>
#include <string>
#include <vector>
```

```
using std::unique_ptr;
using std::string;
using std::map;
using std::move;
using namespace llvm;
```

头文件

```
// 这里是AST节点的定义
namespace {
    class ExprAST {
        public:
        virtual ~ExprAST() {}
        // 用于生成IR
        virtual Value* codegen() = 0;
    };
    using ASTptr = unique_ptr<ExprAST>;
    class NumAST : public ExprAST {
        double val;
        public:
        NumAST(double v): val(v) {}
        virtual Value* codegen();
    };
    class VarAST : public ExprAST {
        string name;
        public:
        VarAST(const string& n): name(n) {}
        virtual Value* codegen();
    };
    class BinOpAST : public ExprAST {
        char op;
        unique_ptr<ExprAST> lhs, rhs;
        public:
        BinOpAST(char op, ASTptr LHS, ASTptr RHS)
            : op(op), lhs(move(LHS)), rhs(move(RHS)) {}
        virtual Value* codegen();
    };
    class DefAST {
        string name;
        ASTptr val;
        public:
        DefAST(const string& name, ASTptr val)
            : name(name), val(move(val)) {}
        virtual void codegen();
    };
}
```

AST

下面是一些主要环境变量和辅助函数:

```
enum Token {
    tok_eof = -1,
    tok_identifier = -2,
    tok_number = -3,
    tok_def = -4,
    tok_end = -5
};
```

```
8
9  static string identifier_str;
10 static double num_val;
11
12 int gettok() {
13     static int last_char = ' ';
14     while (isspace(last_char))
15         last_char = getchar();
16     // identifier || def
17     if (isalpha(last_char)) {
18         identifier_str = last_char;
19         while (isalnum(last_char = getchar()))
20             identifier_str += last_char;
21         // def
22         if (identifier_str == "def")
23             return tok_def;
24         return tok_identifier;
25     }
26     // Number, a small bug
27     if (isdigit(last_char) || last_char == '.') {
28         string num_str;
29         do {
30             num_str += last_char;
31             last_char = getchar();
32         } while (isdigit(last_char) || last_char == '.');
33         num_val = strtod(num_str.c_str(), nullptr);
34         return tok_number;
35     }
36     // EOF
37     if (last_char == EOF)
38         return tok_eof;
39     if (last_char == ';'){
40         last_char = getchar();
41         return tok_end;
42     }
43
44     // Operators
45     int this_char = last_char;
46     last_char = getchar();
47     return this_char;
48 }
49 // 递归下降解析器的预读字符缓冲
50 int cur_tok;
51 int next_token() {
52     return (cur_tok = next_token());
53 }
54 ASTptr Error(const char* str) {
55     fprintf(stderr, "Error: %s\n", str);
56     return nullptr;
57 }
58 // 用于递归下降解析的运算符优先级表
59 map<char, int> op_prec;
60 int get_prec(char c) {
61     auto prec = op_prec.find(c);
62     if (prec == op_prec.end())
63         return -1;
64     else return prec->second;
```

```
}

ASTptr Error(const char* str) {
    fprintf(stderr, "Error: %s\n", str);
    return nullptr;
}
```

环境

同时一个简单的解析器也在下方给出, 注意本代码出于演示目的忽略了很多错误处理.

```
ASTptr parse_expression();
unique_ptr<DefAST> parse_define();
ASTptr parse_primary();
ASTptr parse_binop(int expr_prec, ASTptr lhs);
ASTptr parse_id();
ASTptr parse_num();
ASTptr parse_paren();

/// programmer ::= define | expression
// 同时承担程序入口的责任
void parse_programmer() {
    while (true) {
        printf("ready> ");
        switch (cur_tok) {
            case tok_eof: return;
            case tok_def: parse_define()->codegen(); break;
            case tok_end: break;
            default: parse_expression()->codegen()->print(errs());
        }
        next_token();
    }
}
/// define ::= 'def' Id '=' expression
unique_ptr<DefAST> parse_define() {
    next_token(); // eat def
    auto name = identifier_str;
    next_token(); // eat '='
    next_token();
    auto val = parse_expression();
    return llvm::make_unique<DefAST>(DefAST(name, move(val)));
}
/// expr ::= primary [binop]
ASTptr parse_expression() {
    return parse_binop(0, parse_primary());
}
/// primary ::= Id | Num | paren
ASTptr parse_primary() {
    switch (cur_tok) {
        case tok_identifier:
            return parse_id();
        case tok_number:
            return parse_num();
        case '(':
            return parse_paren();
        default:
            return Error("Unknown token in primary");
    }
```

```
}
/// paren ::= '(' expression ')'
ASTptr parse_paren() {
    next_token(); // eat (
    auto expr = parse_expression();
    if (cur_tok != ')')
        return Error("Unmatch paren");
    next_token(); // eat )
    return expr;
}
/// binop ::= primary Op binop
ASTptr parse_binop(int expr_prec, ASTptr lhs) {
    while (true) {
        int now_prec = get_prec(cur_tok);
        if (now_prec < expr_prec)
            return lhs;

        int binop = cur_tok;
        next_token();
        auto rhs = parse_primary();

        int next_prec = get_prec(cur_tok);
        if (now_prec < next_prec)
            rhs = parse_binop(expr_prec + 1, move(rhs));
        lhs = llvm::make_unique<BinOpAST>(binop, move(lhs), move(rhs));
    }
}

ASTptr parse_num() {
    auto res = llvm::make_unique<NumAST>(num_val);
    next_token();
    return move(res);
}

ASTptr parse_id() {
    auto res = llvm::make_unique<VarAST>(identifier_str);
    next_token();
    return move(res);
}
```

解析器

下面就是本节讲述的重点: LLVM IR C++ API. 我们使用用 AST 节点的 codegen 虚方法来递归地生成 IR, 您将看到这种生成方法是如此强大与方便, 以至于代码量还不及解析器:

```
// LLVM API 文档(英文): https://llvm.org/docs/LangRef.html
static LLVMContext my_context;
static IRBuilder<> builder(my_context);
static unique_ptr<Module> my_module;
static map<string, Value *> named_vals;
Value *LogErrorV(const char *str) {
  LogError(str);
  return nullptr;
}

Value* NumAST::codegen() {
    return ConstantFP::get(my_context, APFloat(this->val));
}
```

```
Value* VarAST::codegen() {
    Value* v = named_vals[this->name];
    if (!v)
        return LogErrorV("undefine variable name");
    return v;
}
Value* BinOpAST::codegen() {
    Value* l = this->lhs->codegen();
    Value* r = this->rhs->codegen();
    if (!l || !r)
        return nullptr;
    switch (this->op) {
        case '+':
            return builder.CreateFAdd(l, r, "addtmp");
        case '-':
            return builder.CreateFSub(l, r, "subtmp");
        case '*':
            return builder.CreateFMul(l, r, "multmp");
        case '/':
            return builder.CreateFDiv(l, r, "divtmp");
        default: return LogErrorV("unexpect binop");
    }
}
void DefAST::codegen() {
    printf("Name: %s defined\n", this->name.c_str());
    named_vals[this->name] = this->val->codegen();
}
```

IR 生成

从上述代码可以看出 LLVM IR C++ API 的完善与简洁. 在保留了比 C 更贴近一点底层的表示的同时很好地维护了 IR 的类型信息. 完整的 LLVM API 可以在 LLVM 的网站上查询.

使用测试样例运行程序:

```
ready> def x = 42.0;
ready> def y = 3.2*x;
ready> x+y;
double 1.764000e+02
ready>
```

测试

可以看到 LLVM 会在生成 IR 中就为我们完成简单的常量常量折叠, 事实上在这个小例子中 LLVM 甚至能帮我们完成**全部**的运算并折叠成一个常量. 注意这在 LLVM 的分类中 **" 连优化都算不上"**, 是默认开启的. LLVM 对机器无关的代码优化能力可见一斑.

但即使是这种小例子也足以折射出 LLVM IR 的一个特点: 强类型. LLVM IR 对于类型是极为严格的, 甚至不允许 C 的很多符合常理的隐式类型转换, 这一点也在后续的优化中发挥很大的作用.

4 总结与参考引用

本综述阐述了基于 LLVM 编写编译器前端的基本流程, 为今后开展编译技术方面研究性学习奠定了坚实的基础. 今后的研究性学习将着手于编译器前端技术的学习, 实现一个简单的 Scheme 解释器与 C 编译器.

本文主要参考: LLVM Tutorial : https://llvm.org/docs/tutorial/index.html
LLVM Document : https://llvm.org/docs/LangRef.html
Wikipedia : https://www.wikipedia.org/
« 编译原理 » 机械工业出版社, 第二版.
« 深入理解计算机系统 » 机械工业出版社, 第三版.
代码地址: