

# 编译原理Lab3

专业	姓名	学号
计算机科学与技术	吴浩岚	19335209

## 一、LLVM IR的学习

### 1. Source Filename

source\_filename 的值就是包含clang 输入的源文件（原封不动）的字符串。

### 2. Data Layout

target datalayout 描述了目标机器中数据的内存布局方式，包括：字节序、类型大小以及对齐方式。

- 第一个字母表示目标机器的字节序。可选项：小写字母e（小端字节序）、大写字母E（大端字节序）。
- m:<mangling>, 指定输出结果中的名称重编风格。(e:ELF、m:Mips、o:Mach-o、x:x86)
- p[n]:<size>:<abi>:<pref>:<idx>, 指定在某个地址空间中指针的大小及其对齐方式
- i<size>:<abi>:<pref>, 指定整数的对齐方式
- f<size>:<abi>:<pref>, 指定浮点数的对齐方式
- n<size1>:<size2>:<size3>..., 指定目标处理器原生支持的整数宽度
- S<size>, 指定栈的对齐方式（单位：比特）

### 3. Target Triple

target triple 描述了目标机器是什么，从而指示后端生成相应的目标代码。

### 4. Identifiers

LLVM IR 中的标识符分为：全局标识符和局部标识符。

**全局标识符**以@开头，比如：全局函数、全局变量。

**局部标识符**以%开头，类似于汇编语言中的寄存器。

标识符有如下 3 种形式：

1. 有名称的值（Named Value），表示为带有前缀（@或%）的字符串。比如：%val、@name。
2. 无名称的值（Unnamed Value），表示为带前缀（@或%）的无符号数值。比如：%0、%1、@2。
3. 常量。

## 5. Functions

`define` 用于定义一个函数。

- `define void @foo(i32 %x) { ... }`，表示定义一个函数。其函数名称为 `foo`，返回值的数据类型为 `void`，参数（用 `%x` 表示）的数据类型为 `i32`（占用 4 字节的整型）。
- `#0`，用于修饰函数时表示一组函数属性。这些属性定义在文件末尾。

举一个具体的例子：

```
define weak dso_local void @foo(i32 %x) #0 {
entry:
%x.addr = alloca i32, align 4
%y = alloca i32, align 4
%z = alloca i32, align 4
store i32 %x, i32* %x.addr, align 4
%0 = load i32, i32* %x.addr, align 4
%cmp = icmp eq i32 %0, 0
br i1 %cmp, label %if.then, label %if.end

if.then:
store i32 5, i32* %y, align 4
br label %if.end

if.end:
%1 = load i32, i32* %x.addr, align 4
%tobool = icmp ne i32 %1, 0
br i1 %tobool, label %if.end2, label %if.then1

if.then1:
store i32 6, i32* %z, align 4
br label %if.end2

if.end2:
ret void
}
```

源码如下：

```
void foo(int x) {
    int y, z;
    if (x == 0)
        y = 5;
    if (!x)
        z = 6;
}
```

## 二、相关命令

语法分析：

```
clang -xcclang -ast-dump -fsyntax-only ../tester/wuhlan.c

//或
( export PATH=~/.sysu/bin:$PATH CPATH=~/.sysu/include:$CPATH
  LIBRARY_PATH=~/.sysu/lib:$LIBRARY_PATH
  LD_LIBRARY_PATH=~/.sysu/lib:$LD_LIBRARY_PATH && clang -E tester/wuhlan.c |
  clang -cc1 -ast-dump=json ) > wuhlan.json

//或
/home/wuhlan3/lab2/SYSU-lang/preprocessor/sysu-preprocessor tester/wuhlan.c |
/usr/lib/llvm-11/bin/clang-11 -cc1 -dump-tokens 2>&1 |
/home/wuhlan3/sysu/build/parser/sysu-parser
```

可视化过程：

```
clang -emit-llvm -S tester/wuhlan.c
opt -dot-cfg wuhlan.ll
到http://viz-js.com/查看可视化
```

运行我自己的generator：

```
cmake --build ~/.sysu/build &&
cmake --build ~/.sysu/build -t install

clang -E tester/functional/000_main.sysu.c | clang -cc1 -ast-dump=json |
/home/wuhlan3/sysu/build/generator/sysu-generator

//测试
CTEST_OUTPUT_ON_FAILURE=1 cmake --build ~/.sysu/build -t test
```

输出如下所示：

```
; ModuleID = '-'
source_filename = "-"

define i32 @main() {
entry:
    ret i32 3
}
```

测试我自己写的C文件：

```
clang -E tester/wuhlan.c | clang -cc1 -ast-dump=json |
/home/wuhlan3/sysu/build/generator/sysu-generator
```

单个测试：

```
( export PATH=~/.sysu/bin:$PATH \  
CPATH=~/.sysu/include:$CPATH \  
LIBRARY_PATH=~/.sysu/lib:$LIBRARY_PATH \  
LD_LIBRARY_PATH=~/.sysu/lib:$LD_LIBRARY_PATH &&  
# sysu-compiler --unittest=benchmark_generator_and_optimizer_1 "**/*.sysu.c" )  
# sysu-compiler --unittest=benchmark_generator_and_optimizer_1  
"functional/*.sysu.c" )  
sysu-compiler --unittest=benchmark_generator_and_optimizer_1 \  
/home/wuhlan3/lab3-new/SYSU-lang/tester/wuhlan.c )
```

clang运行程序:

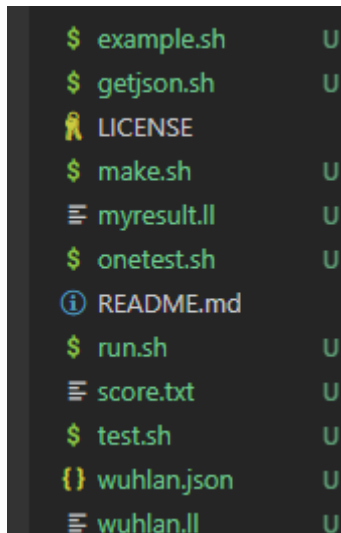
```
export PATH=~/.sysu/bin:$PATH \  
CPATH=~/.sysu/include:$CPATH \  
LIBRARY_PATH=~/.sysu/lib:$LIBRARY_PATH \  
LD_LIBRARY_PATH=~/.sysu/lib:$LD_LIBRARY_PATH &&  
clang tester/h_functional/084_expr_eval.sysu.c -lsysy -lsysu -o a.out &&  
./a.out ;
```

打包:

```
cmake --build ~/.sysu/build -t package_source
```

## 提高效率

写了一些.sh文件来缩减命令的长度。



```
$ example.sh U  
$ getjson.sh U  
📄 LICENSE  
$ make.sh U  
≡ myresult.ll U  
$ onetest.sh U  
📖 README.md  
$ run.sh U  
≡ score.txt U  
$ test.sh U  
📄 wuhlan.json U  
≡ wuhlan.ll U
```

## 三、实验过程

### 1.全局变量和局部变量

由于定义和声明的方式不一样，所以分成了两个函数来实现：

- BuildGlobalVarDecl
- BuildVarDecl

两个函数都要对数据类型，数据名称进行解析，之后使用 `TheModule.getOrInsertGlobal()` 或 `Builder.CreateAlloca()`

其中，参照clang，函数中的局部变量空间分配都放到函数的起始位置：

```
llvm::Function *TheFunction = Builder.GetInsertBlock()->getParent();
llvm::IRBuilder<> tempBuilder(&TheFunction->getEntryBlock(),
                             TheFunction->getEntryBlock().begin());
```

## 2.表达式

### (1)返回全局变量

```
int a = 3;

int main(){
    return a;
}
```

可以看到如下所示：

```
@a = dso_local global i32 3, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    %2 = load i32, i32* @a, align 4
    ret i32 %2
}
```

这一个过程包括：

- 对a进行取值，并赋值到%2
- 返回%2

关键在于在TheModule中查找全局变量，加入到Builder后，再返回该value：

```
llvm::GlobalVariable *globalVar = TheModule.getNamedGlobal(name);
auto val = Builder.CreateLoad(globalVar);
return val;
```

### (2)数组的声明

数组维度的确定，可以通过qualType来构造需要的类型。所以我们可以写一个额外的函数ParseType来进行类型的解析。

数组全部初始化为0，如下：

```
llvm::ConstantAggregateZero* const_array_zero =
    llvm::ConstantAggregateZero::get(type);
globalVar->setInitializer(const_array_zero);
```

## 初始化序列

```
define dso_local i32 @main() {
entry:
  %b = alloca [2 x [1 x i32]], align 4
  %0 = getelementptr inbounds [2 x [1 x i32]], [2 x [1 x i32]]* %b, i32 0, i32 0, i32 0  取出b[0][0]地址到%0
  store i32 13, i32* %0, align 4  将13存到b[0][0]
  %1 = getelementptr inbounds [2 x [1 x i32]], [2 x [1 x i32]]* %b, i32 0, i32 1, i32 0  取出b[1][0]地址到%1
  store i32 17, i32* %1, align 4  将17存到b[1][0]
  ret i32 0
}
```

```
int main(){
    int b[2][1] = {13,17};
    int a[3] = {1,b[1][0],3};
    return 0;
}
```

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [2 x [1 x i32]], align 4  b[2][1]
  %3 = alloca [3 x i32], align 4  a[3]
  store i32 0, i32* %1, align 4  retval
  %4 = bitcast [2 x [1 x i32]]* %2 to i8*  b的初始化
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 4 %4, i8* align 4 bitcast ([2 x [1 x i32]]
  %5 = getelementptr inbounds [3 x i32], [3 x i32]* %3, i64 0, i64 0  获取 a[0]指针
  store i32 1, i32* %5, align 4  将1存到a[0]
  %6 = getelementptr inbounds i32, i32* %5, i64 1  %6是a[1]的指针
  %7 = getelementptr inbounds [2 x [1 x i32]], [2 x [1 x i32]]* %2, i64 0, i64 1  获取b[1]
  %8 = getelementptr inbounds [1 x i32], [1 x i32]* %7, i64 0, i64 0  获取b[1][0]
  %9 = load i32, i32* %8, align 4  取出b[1][0]的值到%9
  store i32 %9, i32* %6, align 4  %9存到%6
  %10 = getelementptr inbounds i32, i32* %6, i64 1  %10是a[2]的指针
  store i32 3, i32* %10, align 4  3存到%10
  ret i32 0
}
```

获取数组的值，耗费我的时间比较长，最终发现自己的代码结构出现了问题，需要进行重构。并且对JSON结构一步步理解清楚，最终解决了这个问题。

## (3)函数参数的问题

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @func(i32 %0) #0 {
  %2 = alloca i32, align 4  给函数参数分配空间
  store i32 %0, i32* %2, align 4  将函数参数的值存到空间中
  %3 = load i32, i32* %2, align 4  取出p的值，到%3
  %4 = sub nsw i32 %3, 1  减法，存到%4
  store i32 %4, i32* %2, align 4  将%4存到%2
  %5 = load i32, i32* %2, align 4  返回的过程
  ret i32 %5
}
```

经过和群友讨论，最终解决的方法采用的是：对符号表中的函数参数增加一个.addr，以便和普通的局部变量进行区分。

```
define dso_local i32 @func(i32 %p) {
entry:
    %p.addr = alloca i32, align 4
    store i32 %p, i32* %p.addr, align 4
    ret i32 0
}
```

## (4)条件语句和循环语句

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 10, i32* @a, align 4
    %2 = load i32, i32* @a, align 4
    %3 = icmp sgt i32 %2, 0
    br i1 %3, label %4, label %5

4: then:                                     ; preds = %0
    store i32 1, i32* %1, align 4  将1存到%1
    br label %6  跳转到%6

5: else:                                     ; preds = %0
    store i32 0, i32* %1, align 4  将0存到%1
    br label %6  跳转到%6

6: ifcont:                                   ; preds = %5, %4
    %7 = load i32, i32* %1, align 4
    ret i32 %7
}
```

比较过程

越写越乱，最后重新梳理了一下Builder相关的API，理清整个逻辑。重点还是在于对BasicBlock的理解要到位。

## (5)Break和Continue的处理

一开始我的做法是设置了两个全局的变量 BreakBB、ContinueBB 来存储将要跳转的块信息。后来发现可能出现这样的情况：

```
while(a >= 1){
    if(a == 3)break;
    while(b >= 1){
        if(b == 2)break;
    }
    if(a == 3)break;
}
```

这种情况下，就有可能出现最后一个break需要跳转的目的地不明确的情况。解决方法是将这两个目的块用栈来存储：

```
stack<llvm::BasicBlock*> BreakBB;
stack<llvm::BasicBlock*> ContinueBB;
```

每当遇到while语句的时候，把将要跳转的块存放到栈里。当该while语句解析完全了，才弹栈。

### 3.卡住很久的“短路”

一直感觉短路非常难处理，所以卡了大概有两三天吧，不敢下手去写代码，因为没有想清楚。

观察clang生成的IR，使用了phi。但是我觉得使用起来还是比较麻烦。

后来考虑到可以使用两个栈，因为短路一般只需要往两个方向判断，要么是往下走，要么就往旁边的进行判断。最终的实现方式如下：

```
//全局定义了两个栈
stack<llvm::BasicBlock*> ShortCircuitThen;
stack<llvm::BasicBlock*> ShortCircuitSkip;

void BuildIfStmt(const llvm::json::Object *O){
    assert(O->getString("kind")->str() == "IfStmt");
    llvm::Function *TheFunction = Builder.GetInsertBlock()->getParent();
    llvm::BasicBlock *ThenBB = llvm::BasicBlock::Create(TheContext, "ifthen");
    llvm::BasicBlock *ElseBB = llvm::BasicBlock::Create(TheContext, "ifelse");
    llvm::BasicBlock *MergeBB = llvm::BasicBlock::Create(TheContext, "ifcont");

    // ShortCircuitElseBB = ElseBB; //保存一下
    // ShortCircuitThenBB = ThenBB; //保存一下
    // ShortCircuitIfContBB = MergeBB; //保存一下

    auto hasElse = O->get("hasElse");

    //短路设置*****
    if(hasElse){
        ShortCircuitThen.push(ThenBB);
        ShortCircuitSkip.push(ElseBB);
    }else{
        ShortCircuitThen.push(ThenBB);
        ShortCircuitSkip.push(MergeBB);
    }

    auto Arr = *O->getArray("inner");
    llvm::Value* condition;
    //条件判断
    string conditionkind = Arr[0].getAsObject()->getString("kind")->str();
    condition = BuildAny(Arr[0].getAsObject());
    if(condition){
        if(condition->getType() == llvm::Type::getInt32Ty(TheContext)){
            condition = Builder.CreateICmpNE(condition,
                llvm::ConstantInt::get(TheContext, llvm::APInt(32, "0", 10)));
        }
    }
}
```

根据不同的情况，来设置将要跳转的两种路径。

### 4.关于符号表的问题

在群里又看到很多解决办法。但是感觉实现起来比较麻烦，因为在层层查找块符号表的时候，需要考虑深度的问题。后来听一位同学说，可以使用JSON语法树上的id来区分同名变量后，就采取了这种“偷懒”的方法。



```

"inner": [
  {
    "id": "0x15730a0",
    "kind": "DeclStmt",
    "range": { ...
  },
  "inner": [
    {
      "id": "0x1572f18",
      "kind": "VarDecl",
      "loc": { ...
    },
    "range": { ...
  },
  "isUsed": true,
  "name": "i",
  "mangledName": "i",
  "type": {
    "qualType": "int"
  }
},

```

该id是会伴随这个变量的。并且作用域不同的时候，同名变量产生的id不同。

所以最终我定义了一个全局的map来存储【id，变量地址】

```
map<string, pair<llvm::Value*, bool>> VarEnv;
```

其中还添加了一个bool变量，用来记录该变量是否是函数的参数。因为函数参数，在获取数组指针的时候会有些不一样。

## 5.关于内存不足的问题

当卡在样例388的时候，非常难受，在评测系统上运行，会直接终止也没有具体报错。感觉自己距离成功就差临门一脚了，但是不明白到底是哪里出现了错误。将程序扒下来后，经过慢慢的排查发现是这条语句出现了问题：

```
return i1 + i2 + i3 + i4 + ... + i998 + i999 + i1000;
```

最终发现，原来前面的同学已经遇到过这个问题了：

### 关于程序内存占用过多的问题 #53

✓ Answered by zhengzhch zhengzhch asked this question in Q&A



zhengzhch 6 days ago

...

在测例integer-divide-optimization-1.sysu.c，函数在return时，有一串很长的加法表达式，这个加法表达式是递归的总共1000层，我的程序在函数递归时，内存占用过多，这个问题有没有什么解决方案呢？

```

i1000 = i1000 / 2;
return i1 + i2 + i3 + i4 + i5 + i6 + i7 + i8 + i9 + i10 + i11 + i12 + i13 + i14 + i15 + i16 + i17 + i18 + i19 + i20 + i21 + i22 + i23 +
}

turnstmt 0x2209600 <line:43:5, col:6855>
BinaryOperator 0x2209848 <col:10, col:6895> 'int' '+'
BinaryOperator 0x22097f0 <col:10, col:6888> 'int' '+'
BinaryOperator 0x2209798 <col:10, col:6881> 'int' '+'
BinaryOperator 0x2209740 <col:10, col:6874> 'int' '+'
BinaryOperator 0x22096e8 <col:10, col:6867> 'int' '+'
BinaryOperator 0x2209690 <col:10, col:6860> 'int' '+'
BinaryOperator 0x2209638 <col:10, col:6853> 'int' '+'
BinaryOperator 0x22095e0 <col:10, col:6846> 'int' '+'
BinaryOperator 0x2209588 <col:10, col:6839> 'int' '+'
BinaryOperator 0x2209530 <col:10, col:6832> 'int' '+'

```

已解决，是因为我在函数中对inner的Array指针解引用了，而这个结构的inner非常巨大，所以会导致内存占用过多。

Marked as answer ↑ 2 😊 2 🍷 2 ❤️ 2 ✏️ 2

0 replies
















最终，我将实验中所有涉及到语法树获取inner块的部分，都修改成直接使用指针来获取：

```
auto Arr = 0->getArray("inner");  
string kind = Arr->begin()->getAsObject()->getString("kind")->str();
```

这样就避免了解引用的操作，从而避免了对超大的JSON结构的拷贝过程。节省了内存和时间。最终也使得我的程序性能上有了很大的提高。

## 四、实验结果

最终实验结果如下：

提交 ID	提交时间	得分	操作
844	2022-05-29 13:44:27	424 / 429	 
843	2022-05-29 13:01:19	389 / 429	 
839	2022-05-29 10:27:47	388 / 429	 
837	2022-05-29 09:43:08	388 / 429	 
836	2022-05-29 08:49:21	388 / 429	 
834	2022-05-28 23:42:37	38 / 429	 
828	2022-05-28 21:37:34	内部错误	 
801	2022-05-27 17:14:53	320 / 429	 

总体性能相比其他同学还算不错，当然，相对于clang还有很大发展空间：

名次	昵称	提交时间	performance	score
11	yinhan	2022-05-25 09:32:07	0.12244472819486825	424
12	助教	2022-03-26 23:46:56	0.1221629353575511	425
13	Wuhlan3	2022-05-29 13:44:27	0.12210063060028752	424
14	PeiWangSYSU	2022-05-25 00:40:16	0.1219168294773924	424
15	zjnyly	2022-05-23 08:18:18	0.12135817840515849	424

## 五、总结

本次实验确实花了很长的时间，大概有7~8天是早晚都在思考该实验的。其中多次想要放弃，比如说5.26的时候，已经能够实现到180个样例左右了，当时觉得不挂科就行，当时卡在短路的问题上，不太清楚自己继续花时间进去会不会有回报。

是什么使自己坚持下去呢？可能是我能够感受到，在做这个实验的过程中，我提高了很多方面的能力、暴露出很多问题。一方面、是懂得使用google，而不是遇事不决就百度，真正让自己发现中文网站资料的贫瘠；另一方面，是尝试去阅读llvm的官方文档、api文档等，这些能力对于我以后的发展是非常重要的；最后，是偶然瞥见其他同学的代码，发现自己的coding能力是那么弱，没能够在实验前先构建出一个整体架构，没有按照面向对象的思想将功能模块化，这些种种问题都导致我在实验过程中踩中无数的坑。我认为自己那种边做边改的习惯是需要摒弃的，必须想清楚了，再动手。

总的来说，对这个编译原理实验，又爱又恨吧。过程是很痛苦的，但是在那一天中午，我将最后一次测试提交到测评网站，准备午休没有睡着的时候，我心中默念着时间，迫不及待地下床看看结果，那一种欣喜若狂是难以描述的。有点小夸张（捂脸），希望能正视自己的问题，再接再厉吧。