LLVM介绍

一、工具下载

针对`Ubuntu`，下载方式为:

```
$ sudo apt-get install llvm
$ sudo apt-get install clang
```

通过以下命令可以测试是否安装成功:

```
$ clang -v
$ lli --version
```

二、工具简单介绍与使用

介绍

程序从高级语言编译成机器语言主要经过前端、中端和后端，前端读入源程序，进行词法、语法分析后生成一棵抽象语法树，然后扫描语法树进行语义分析，检查是否有语义错误；中端对抽象语法树进行扫描并生成中间代码表示形式，可以在其基础上进行通用的优化；后端将中间代码翻译成目标机器码，可以在此基础上进行面向体系结构的优化。中间代码的存在可以作为桥梁衔接编译的前端和后端，不同的高级语言可以生成相同的中间代码，然后在中间代码上进行优化，可以减少编译器开发、优化的代价。

宏观的`LLVM`，指的是整个的`LLVM`的框架，它肯定包含了`Clang`，因为`Clang`是`LLVM`的框架的一部分，是它的一个`C/C++`的前端。虽然这个前端占的比重比较大，但是它依然只是个前端，`LLVM`框架可以有很多个前端和很多个后端，只要你想继续扩展。

微观的`LLVM`指的是`LLVM`的`core`，或者是说实际开发和使用中的具体的`LLVM`。也可以简单的理解为名为`LLVM`的源码包。编译`LLVM`和`Clang`的时候，`LLVM`的源码包是不包含Clang的源码包的，需要单独下载`Clang`的源码包。

`Clang`和微观`LLVM`一起构成了一个完整的编译器，`Clang`是前端，中端优化和后端都在微观`LLVM`之中。

由于课时限制，我们的实验只涉及到`LLVM IR`生成部分。

工具使用

这里介绍下如果利用`Clang`和`LLVM`帮助我们更好地完成实验。

编写一个简单的C语言程序如下:

```c
// main.c
int main(){
  int a = 10;
  int b = 5;
  return a + b;
}
```

使用`Clang`编译该文件生成`LLVM`中间代码：

```
$ clang -S  -emit-llvm main.c -o main.ll -O0
```

你会发现你得目录下生成了`main.ll`文件，它的内容大概是这样的：

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %2, align 4
  store i32 5, i32* %3, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %3, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

attributes #0 = { noinline nounwind optnone sspstrong uwtable "frame-
pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-
protector-buffer-size"="8" "target-cpu"="x86-64" "target-
features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 13.0.1"}
```

根据个人电脑硬件与系统，有些内容会有差异，比如`target triple`和`target datalayout`，我们的实验中并不要求生成它们；`align`说明4字节对齐，`dso_local`表明该变量或函数会在同一个链接单元内解析符号；`;`是`LLVM IR`注释的开头……这些内容都不要求在本实验中生成（当然如果生成了也不会影响实验结果）。

把无关的东西删除后，我们得到如下的`.ll`文件，为了便于同学们理解，我们添加了注释：

```
;函数定义以`define`开头，i32是返回值类型(32位int)，main是函数的名字
define i32 @main() {
  %1 = alloca i32, align 4  ;为%1分配空间,这里%1是i32*类型，可以理解成int*
  %2 = alloca i32, align 4  ;为%2分配空间，对应a
  %3 = alloca i32, align 4  ;为%3分配空间，对应b
  store i32 0, i32* %1, align 4
  store i32 10, i32* %2, align 4    ;将10存到%2中，对应a=10
  store i32 5, i32* %3, align 4     ;将5存到%3中，对应b=5
  %4 = load i32, i32* %2, align 4   ;将%2的值取出加载到%4中
  %5 = load i32, i32* %3, align 4   ;将%3的值取出加载到%5中
  %6 = add nsw i32 %4, %5           ;%4和%5加起来赋值给%6
  ret i32 %6                        ;返回%6，类型为i32
}
```

上面这个中间代码还是比较容易理解的。当然你利用第三方jar包生成的中间代码不一定跟生面`Clang`生成的一致，比如下面是助教的程序生成的中间代码：

```
define i32 @main() {
mainEntry:
  %a = alloca i32, align 4
  store i32 10, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 5, i32* %b, align 4
  %a1 = load i32, i32* %a, align 4
  %b2 = load i32, i32* %b, align 4
  %tmp_ = add i32 %a1, %b2
  ret i32 %tmp_
}
```

大体逻辑是一样的，其中`%a, %b, %a1, %b2, %tmp`等都是助教自定义的临时变量名，你可以在代码中自己设计一套命名方式，不同变量命名不冲突即可。

现在你有了`main.ll`文件，你可以通过`lli`命令来执行该文件：

```
$ lli main.ll
```

Linux每个程序执行都有一个返回值，你可以通过`$?`来获取：

```
$ echo $?
```

上面这条命令会在命令行打印出你刚刚执行`main.ll`的返回值，助教这里显示是`15`，这与`main.c`的实际返回值一致。(`shell`的返回值会在`0-255`之间，如果你返回`-1`或者`255`，那么打印出的都是`255`，即它会对你程序的返回结果模`256`)

知道`Clang`和`lli`的基本使用对你完成本地的意义在于：

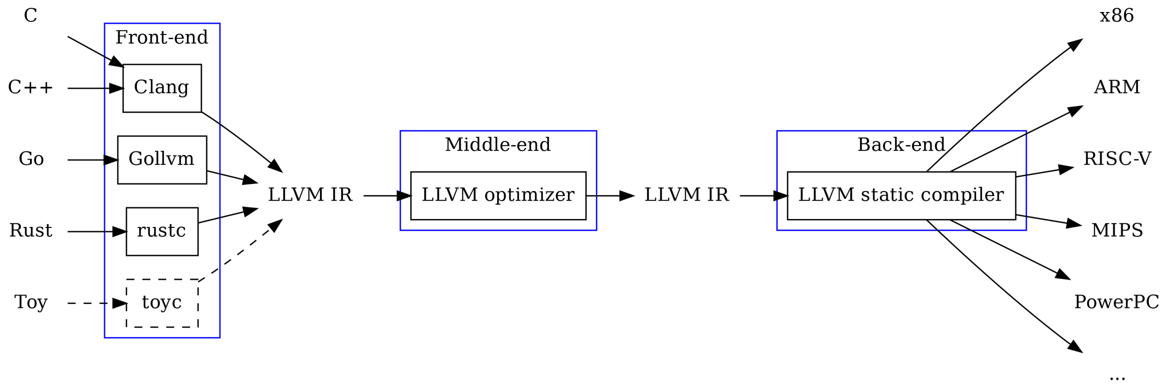
1. 学习`LLVM IR`：你可以自己写一些符合`SysY`语言定义的程序，并利用`Clang`编译生成`LLVM IR`，通过逐行对比的方式，了解、学习`Clang`是如何翻译源代码的，并且模仿着在你的程序里调用第三方库生成正确的`LLVM IR`。
2. 验证你的程序生成的`LLVM IR`的正确性：上面已经演示过如何利用`lli`运行`LLVM IR`并且获取返回值，你可以手动创建一个`*.ll`文件，把你的`Java`程序生成的`LLVM IR`拷贝进去并且执行、查

看返回结果，如果返回结果是正确的，你可以认为你的程序生成了正确的`LLVM IR`。

三、`LLVM`简介

在阅读这部分时，你可以参照这里给出的例子学习如何在`java`代码中调用`LLVM`的库生成`LLVM IR`。

`LLVM`项目的整体架构如下图所示：



`LLVM IR`有三种表示形式，并且这三种形式是完全等价的：

- 在内存中的编译中间语言（我们无法通过文件的形式得到）
- 在硬盘上存储的二进制中间语言（格式为`.bc`）
- 人类可读的代码语言（格式为`.ll`）

我们的中间代码生成部分的实验要求生成`.ll`格式的`LLVM IR`。

`LLVM IR`的介绍

我十分推荐你自己写一些`C`语言的代码并利用`Clang`编译成`LLVM IR`的形式进行学习。

`Module` 模块

`Module`相当于一个`.c`文件，每个`Module`之间相互独立，`Module`主要包含了声明或者定义的函数、声明或定义的全局变量等。

为了获取`Module`存储的信息，调用`LLVMDumpModule(LLVMModuleRef var0)`方法就会在屏幕上打印出全部信息。当然打印的前提是你已经在遍历抽象语法树(`AST`)时将生成`LLVM IR`需要用到的信息存储到`Module`中了。

`Context`

`Context`包含了`LLVM`在一个线程中正常运行（比如一个编译任务）所需要的数据（即`LLVM`程序状态数据）。在很久以前，也就是`LLVM`的老版本中，这些状态数据都是全局数据。后来，它们被一股脑儿打包到了一个名叫`LLVMContext`的对象中，这样，`LLVM`就能支持在多线程中运行编译任务了。作为`api`的调用者，我们可以简单地把它当做一个代表，它代表了`LLVM`这个引擎库。我们只需记住，当一个函数需要`context`的时候，我们就把它传进去即可。

`IRBuilder`

`IRBuilder`的作用比较纯粹，它的存在就是为了让我们方便快捷地创建IR指令。

`IRBuilder`就是一个工具，它提供了一套统一的`API`，以便用户用来创建IR指令，并插入到代码块中。插入的位置可以是在代码块的结尾处，也可以是在代码块中的其它位置。当每次插入`Intruction`之后，`IRBuilder`都会更新并记录下新的插入点，以便下一次能插到正确的位置。

`IRBuilder`的工作方式是，首先指定当前代码块，然后逐指令插入。当完成了一个代码块后，通过`LLVMPositionBuilderAtEnd(LLVMBuilderRef var0, LLVMBasicBlockRef var1)`更改当前代码块。`IRBuilder`在其内部记录了当前的代码块(`BasicBlock`)，以及当前代码块中的当前指令。每次插入新的指令时，总是在当前指令的后面插入。

`Basic Block 基本块`

一个基本块是包含了若干指令以及一个终结指令的代码序列。

基本块的执行是原子性的，基本块只会从终结指令退出。也就是说，如果基本块中的一条指令执行了，那么块内其他所有的指令也都会执行。基本块内部没有控制流，控制流是由多个基本块之间通过跳转指令实现的。

例如你有一份不含跳转（没有分支、循环）也没有函数调用的、只会顺序执行的代码，那么这份代码只有一个基本块。

然而，一旦在中间加入一个 `if-else` 语句，那么代码就会变成四个基本块：`if` 上面的代码仍然是顺序执行的，在一个基本块中；`then` 和 `else` 各自部分的代码也都是顺序执行的，因此各有一个基本块；`if` 之后的代码也是顺序执行的，也在一个基本块中。所以总共四个基本块。

例子：

```
int main(){
    int x = 2;
    int y;
    if(x > 1)
        y = 2;
    else
        y = 1;
    return y;
}
```

利用`Clang`生成的`LLVM IR`如下：

```
define dso_local i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = icmp sgt i32 %4, 1
  br i1 %5, label %6, label %7

6:                                                ; preds = %0
  store i32 2, i32* %3, align 4
```

```
  br label %8

7:                                                ; preds = %0
  store i32 1, i32* %3, align 4
  br label %8

8:                                                ; preds = %7, %6
  %9 = load i32, i32* %3, align 4
  ret i32 %9
}
```

它就有四个基本块，其中第一个基本块是匿名的。当然你自己在对上面`C`代码生成`LLVM IR`时可以翻译成任意多的基本块，只要执行结果正确即可。

`Instruction 指令`

指令指的`LLVM IR`中的非分支指令（non-branching Instruction），通常用来进行某种计算或者是访存（比如上面例子中的 `add`、`load`），这些指令并不会改变程序的控制流。

值得一提的是，`call` 指令也是非分支指令，因为在使用 `call` 调用函数时，我们并不关系被调用函数内部的具体情况（即使被调用函数内部存在的控制流），而是只关心我们传入的参数以及被调用函数的返回值，因此这并不会影响我们当前程序的控制流。

`Terminator instruction 终结指令`

终结指令**一定**位于某个基本块的末尾（否则中间就改变了基本块内的控制流）；反过来，每个基本块的末尾也**一定**是一条终结指令（否则仍然是顺序执行的，基本块不应该结束）。终结指令决定了程序控制流的执行方向。例如，`ret` 指令会使程序的控制流返回到当前函数的调用者（可以理解为 `return`），`br` 指令表示根据标识符选择一个控制流的方向（可以理解为 `if`）。

`br`指令一共有两种形式：

- `br + 标志位 + truelabel + falselabel`条件跳转，如果标志位为`1`，会跳往`truelabel`标记的基本块，如果标志位为`0`，会跳往`falselabel`标记的基本块。例如上面例子中的`br i1 %5, label %6, label %7`，`i1`可以视作一位的`bool`，取值为`0`或`1`，如果`%5`为`1`，会跳到基本块`%6`，如果为`0`，会跳到基本块`%7`。`%5`是`icmp sgt i32 %4, 1`执行的结果，我认为这里`icmp sgt`的意思是`int类型(i)，比较(cmp),符号的(s)，%4是否大于(gt)1`。
- `br + label`无条件跳转，程序会无条件跳转到目标基本块。例如上面例子中的基本块`%6`和`%7`中的`br label %8`。

类型系统

接下来我们介绍`LLVM IR`的类型系统。 类型系统是`LLVM IR`中最重要的部分之一，强大的类型系统在很大程度上降低了读取和分析`LLVM IR`的难度，并且可以实现一些在一般的的三地址码`IR`中难以实现的优化。`LLVM IR` 的类型多种多样，我们在此只介绍可能和实验关系紧密的几种。

Void Type

仅占位用，不代表任何值也不占任何空间。 比如：

```
define void @foo(){
  ret void
}
```

Integer Type

最简单的类型，后面数字决定的位宽，比如`i1`代表的就是`1bit`长的 integer（可以看作是 `bool`），`i32`就是`32bit`长的 integer。 比如：

```
ret i32 0
br i1 %2,label  %3,label  %4
```

Label Type

标签类型，用作代码标签，比如：

```
br i1 %9, label %10, label %11
br label %12
```

SSA（静态单赋值）介绍（选读）

在`LLVM IR`中，每个变量都在使用前都必须先定义，且每个变量只能被赋值一次（只能被赋值一次可以理解为只能在等号左边出现一次），所以我们称`LLVM IR`是静态单一赋值的。举个例子的，假如你想返回`a*b+c`的值，你觉得可能可以这么写：

```
%0 = mul i32 %a, %b
%0 = add i32 %0, %c
ret i32 %0
```

但是这里%0被赋值了两次，是不合法的，我们需要把它修改成这样：

```
%0 = mul i32 %a, %b
%1 = add i32 %0, %c
ret i32 %1
```

使用SSA的好处

`SSA`可以简化编译器的优化过程，比如对于如下代码：

```
d1: y := 1
d2: y := 2
d3: x := y
```

我们很容易可以看出第一次对`y`赋值是不必要的，在对`x`赋值时使用的`y`的值时第二次赋值的结果，但是编译器必须要经过一个定义可达性(Reaching definition)分析才能做出判断。编译器是怎么分析呢？首先我们先介绍几个概念：

- 定义：对变量 `x` 进行定义的意思是在某处会/可能给 `x` 进行赋值，比如上面的 `d1` 处就是一个对 `y` 的定义。
- kill：当一个变量有了新的定义后，旧有的定义就会被 kill，在上面的语句中 `d2` 就 kill 了 `d1` 中对 `y` 的定义。
- 定义到达某点：定义 `p` 到达某点 `q` 的意思是存在一条路径，沿着这条路径行进，`p` 在到达到点 `q` 之前不会被 kill。

- reaching definition：`a` 是 `b` 的 reaching definition 的意思是存在一条从 `a` 到达 `b` 的路径，沿着这条路径走可以自然得到 `a` 要赋值的变量的值，而不需要额外的信息。

按照上面的写法，`d1` 便不再是 `d3` 的 reaching definition, 因为 `d2` 使它不再可能被到达。

对我们来说，这件事情是一目了然的，但是如果控制流再复杂一点，对于编译器来说，它便无法确切知道 `d3` 的 reaching definition 是 `d1` 或者 `d2` 了，也不清楚 `d1` 和 `d2` 到底是谁 kill 了谁。但是，如果我们的代码是 SSA 形式的，那它就会长成这样。

```
d1: y1 := 1
d2: y2 := 2
d3: x := y2
```

编译器很容易就能够发现 `x` 是由 `y2` 赋值得到，而 `y2` 被赋值了 2，且 `x` 和 `y2` 都只能被赋值一次，显然得到 `x` 的值的路径就是唯一确定的，`d2` 就是 `d3` 的 reaching definition。而这样的信息，在编译器想要进行优化时会起到很大的作用。

四、参考链接

https://www.llvm.org/docs/ProgrammersManual.html#the-core-llvm-cl

https://llvm.org/doxygen/classllvm_1_1Value.html

https://zhuanlan.zhihu.com/p/26223459

https://buaa-se-compiling.github.io/miniSysY-tutorial/pre/llvm_ir_quick_primer.html

https://codeantenna.com/a/48iqkDbL6V

https://blog.csdn.net/Zhanglin_Wu/article/details/125943137

https://zhuanlan.zhihu.com/p/66793637