『编译技术』SysY-Mips编译器设计——中间代码LLVM生成

章节目录

任务目标

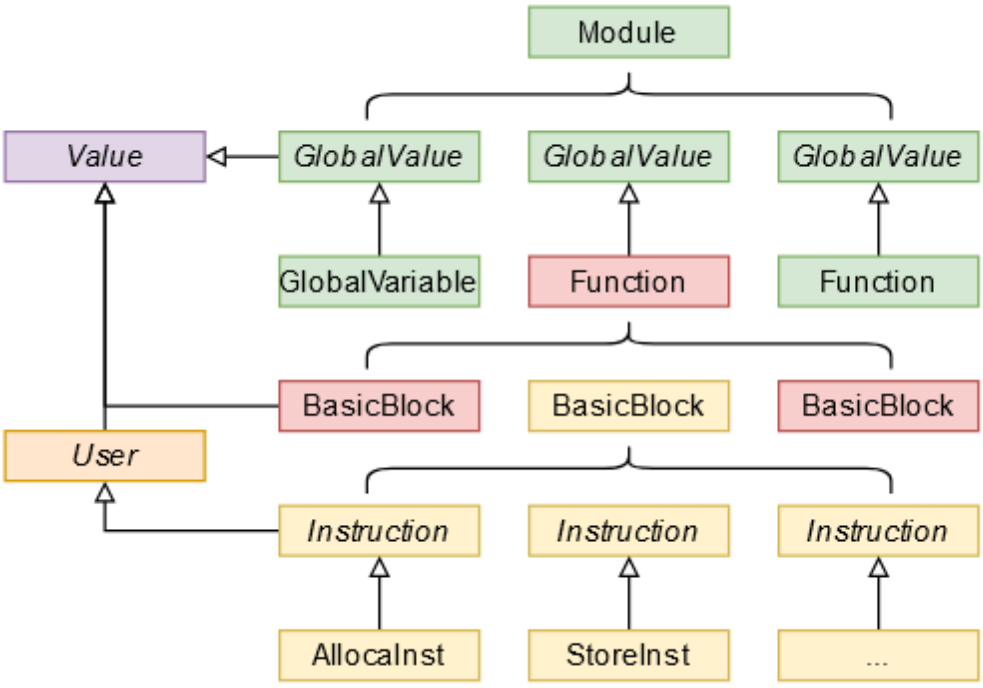
狭义的LLVM（Low Level Virtual Machine）是一种常用的通用中间代码表示。在本部分内容中，我们由建立好的语法树生成中间代码LLVM，不考虑复杂优化方法。

前置准备

1. LLVM生成时将常量全部带入，在后续中我们了解到属于**常量折叠**优化。
2. 在编译开始前遍历字符串，清 `+,-` 连着的长串符号，这样可以减少部分无意义的运算指令。
3. 将错误处理和中间代码生成解耦合，即默认输入代码不出错。
4. 本方案采用第一遍扫描生成语法树，第二遍扫描语法树生成LLVM。

LLVM架构

LLVM采用**User,Value,Use**的架构，具体的继承关系如图。



每一条LLVM语句都可以看作一个`Value`，`Value`是LLVM一切类的父类。从直观来看，`'='`左边的值便是一条语句的Value，又可以叫做综合属性或返回值，当然存在诸如`Store`这类没有返回值，我们可以认为其Value为Null。

`User`类继承`Value`类，而`Instruction`又都继承User类，从直观来看，`'='`右边用到的值，即一些操作数，便是一条语句的`User`，也可以称作继承属性或传入值。

`Use`类记录了`Value`和`User`之间的联系，可以看作是`def-use`图中的一条边，本架构中用列表替代了`Use`的作用，应当是等价的。

对于一条`Instruction`，它既作为一个`User`，又作为一个`Value`。作为`User`时，它记录了`Value`的列表，即记录了使用了哪些`Value`，又或者说使用了哪些语句的返回值；作为`Value`时，它又记录了一个`User`的列表，即记录了这条语句被哪些语句作为操作数使用。通过这种引用关系，我们可以很容易的得到`def-use`链和`use-def`链等，为后续优化做好了基础。

Value

```
public class Value {
    public String value; // 虚拟变量名
    public int valueType; // 记录Value类型，具体如下
    public ArrayList<User> users; // 记录被哪些语句所使用
    public int firstSize; // 当为指针类型时，需要的额外维度参数
    public int secondSize;
}
```

Value的类型人为规定如下，仍建议采用枚举类：

值	类型	描述
-1	常规`store`，`move`指令	用于表示无返回值的`store`和`move`指令
0	`i32`	32位整数
1	`i32*`	一维指针
2	`[n x i32]*`	一维数组的首地址或二维指针
3	`[m x [n x i32]]*`	二维数组的首地址
4	`i1`	1位布尔值
5	`func`	函数类型
6	`basicblock`	基本块引用
7	`i32**`	一维指针的地址
8	`[n x i32]**`	二维指针的地址
9(本次无关)	`fakevalue`	记录后续`mem2reg`时需要的假标签，在重命名步骤时被替换为真`Value`
10(本次无关)	`storeValue`	图着色优化新增，将溢出的变量 store 入内存所用的`Value`，仅作为标识

以上12种类型可以表示所有类型Value，在LLVM生成以及在代码优化中会发挥大作用

User

```
public class User extends Value {
    public ArrayList<Value> usedValues;
}
```

在这里简化了`Use`类，记录了当前`Value`使用过的`Value`，构建起`def-use`链。

其余全部结构皆继承`User`与`Value`，以`Value`为最终父类，构建起了LLVM的中端架构。

符号表

是否重新建表?

建树时已经生成过一份完整符号表，在每一个`Block`内记录符号表的`id`，由此生成LLVM时可实时获得当前层的符号表，**复用旧符号表即可**，无需再重建新的符号表。

使用全局符号表的问题：符号进入顺序无法确定，如处理下述语句时，第一个`a`会查到该作用域新定义的`a`，而不是全局变量的`a`。

解决方法：记录下符号表中每个符号的递增id，在定义语句出现时记录当前符号id，在非定义语句需要查表时，必须满足**查到的符号id小于等于当前符号id**，否则查到的符号在当前位置是**还没定义过**的，需要前往上层符号表继续查找。

```
int a = 10;
int main() {
    int b = a, a;
}
```

符号表的改动

符号表中每个表项新增`Value`属性，记录**存储该符号对应的Value**，在后续使用到该符号时，我们可直接查符号表获取对应的`Value`，以生成中间代码。

短路求值与条件语句，循环语句

and优先级高于or

无论条件还是循环语句，其本质均是依赖于`icmp`跳转指令，仅仅是一些为有条件跳转，一些为无条件跳转的区别。故需要弄清，（1）跳转语句的位置，（2）标签的位置。

条件语句的LLVM生成

涉及文法

```
Stmt    → 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
Cond    → LOrExp
RelExp  → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
EqExp   → RelExp | EqExp ('==' | '!=') RelExp
LAndExp → EqExp | LAndExp '&&' EqExp
LOrExp  → LAndExp | LOrExp '||' LAndExp
```

跳转语句位置

1. 每个 `&&` ， `||` 处， `Cond` 末尾处
2. `Stmt` 末尾
3. `elseStmt` 末尾

Label位置（即创建一个新基本块的位置）

- `Stmt` 之前： `stmtLabel`
- `elseStmt` 之前： `elseStmtLabel`
- `if` 语句整体结束之后： `ifEndLabel`
- 每个 `&&` ， `||` 处： `andLabel` ， `orLabel` （在相应的跳转语句之后）

短路求值

不是全部的逻辑判断语句均要执行，我们称之为短路求值规则。在此给出每个位置跳转语句的跳转位置。

当前位置	后续逻辑	结果	跳转位置
`&&` ，“或”， `Cond` 结尾	`&&`	真	下一个 `&&` 的 `andLabel`
		假	下一个 或 的 `orLabel` ，若无 “或”，则跳转至 `elseStmtLabel`
	“或”	真	`stmtLabel`
		假	`orLabel`
	无	真	`stmtLabel`
		假	`elseStmtLabel`
`Stmt` 末尾	-	-	`ifEndLabel`
`elseStmt` 末尾	-	-	`ifEndLabel`

重填法

由于Label出现的位置在跳转指令之后，无法第一时间填写Label名，因此需要在Label出现后重填跳转指令的标签。我们规定生成跳转指令时先使用如下假标签，采用上述规则生成:

1. `!nextOrLabel`
2. `!nextAndLabel`
3. `!stmtLabel`
4. `!elseStmtLabel`
5. `!ifEndLabel`

为每个条件语句开一个跳转指令列表，每当跳转指令出现时推入，每当真标签语句生成时，**立即**重填跳转指令列表中所有能匹配上的假标签。

立即重填的原因： `nextOrLabel` 和 `nextAndLabel` 代表的位置会随着语句分析而变化。

循环语句的LLVM生成

涉及文法

```
Stmt    → 'for' '(' [ForStmt1] ';' [Cond] ';' [ForStmt2] ')' Stmt
        | 'break' ';'
        | 'continue' ';'
ForStmt → LVal '=' Exp
```

循环语句流程

1. 执行初始化表达式 `ForStmt1`
2. 执行条件表达式 `Cond`，如果为真执行循环体 `Stmt`，否则结束循环执行后续 `BasicBlock`
3. 执行完循环体 `Stmt` 后执行增量/减量表达式 `ForStmt2`
4. 重复执行步骤2和步骤3

Label位置

1. 循环体语句 `Stmt` 之前：`stmtLabel`
2. 循环结束之后的第一条语句：`forEndLabel`
3. 循环条件改变语句 `forStmt` 之前：`forStmtLabel`

跳转语句位置

1. `Cond` 结尾，在此仅将 `Cond` 语句结尾作为一个跳转语句位置
2. `break` 后
3. `continue` 后
4. `forStmt2` 后
5. `Stmt` 后

这部分多是无条件跳转，逻辑较简单，不过多赘述。

Break, Continue的处理

`Break` 和 `Continue` 语法树节点生成时就要保存当前的 `LoopStmt` 节点引用信息，方便快速找到它是属于哪个循环语句的。

`Continue` 跳转到 `forStmtLabel`，`Break` 跳转到 `forEndLabel`。

注：条件或循环语句嵌套问题：每个 `LoopStmt`，`CondStmt` 都需要一个待重填指令的列表，即将各个语句的跳转指令分离开来，重填时避免混淆。

高维数组

符号表中记录的符号是一个地址，使用符号时从符号表中得到地址，再从地址中取出值；而定义符号时，先申请一块地址，再将值存入对应地址中。**一维和二维数组首地址指针不会被存在内存中，但是它客观存在。**

GetElementPtr

该指令为地址转化函数，使用的 `Value(%1)` 和返回的 `Value(%2)` 都一定是指针类型，对本文法而言，只有 `i32*`，`[n x i32]*`，`[m x [n x i32]]*` 三种。

```
%2 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0, i32 0, i32 0
```

对于该条语句，`%1`为基地址，`[2 x [3 x i32]]*`为该基地址指针的类型(若存到内存中均是4字节的地址，维度表明了该指针产生1单位偏移量时地址会随之偏移的多少)，`[2 x [3 x i32]]`为该基地址指针指向的值的类型，返回值`%2`为目标地址，指令后面跟的每一个索引（`i32 0, i32 0, i32 0`），第一个索引在原维度上偏移，自第二个起，先对目标地址降一维再偏移，依次类推，具体参考指令文档。

处理方式

高维数组传的是**地址**，不再是值。 第一个维度可以省略的原因是，需要后面的维度进行计算索引，而第一维不再需要，因为我们不需要关注数组越界问题。

- 定义
  - 申请内存
    - 一维数组：`alloca [2 x i32]`
    - 二维数组：`alloca [3 x [2 x i32]]`，开辟一块指定大小的空间, 返回值是数组的首地址
  - 赋初值：遍历数组，根据基地址计算出需要赋初值的地址，接着同数值赋初值一样
- **表达式计算中使用**：由于表达式计算中不涉及指针运算，先根据基地址和偏移量读取出使用位置的目标地址，再取值即可。
- **传参**：规定为统一起见，在涉及高维数组计算时，都要使用`getelementptr`指令进行基地址到目标地址的转化，即使出现`%2 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0`这种无意义指令。
  - 作为实参传递：根据符号表中查到的符号类型(分为一维数组，二维数组，一维指针，二维指针)与目标形参参数的类型(数值，一维指针，二维指针)，所有情况进行枚举分析。其中一定会使用到`GetElementPtr`指令，这在后续生成Mips有所用处。
  - 作为形参接收：函数的形参实际上也是数个定义语句，将传过来的参数保存进定义的地址中即可。

I1 ➔ I32

处理条件语句时，涉及`i1`和`i32`之间的相互转化(在MIPS中并无这类问题)，具体来说：

1. `icmp`语句的`Value`是`i1`类型，使用的两个操作数是i32类型。

```
%2 = icmp ne i32 0, 0
```

2. `br`语句使用的比较操作数是`i1`类型。

```
br i1 %2, label %12, label %3
```

条件跳转指令依赖于`EqExp`，我们统一规定`EqExp`返回`i1`类型的`Value`，`RelExp`返回`i32`类型的`Value`，当涉及到链式计算时，应使用`zext`指令将`i1`转化为`i32`类型；使用`cmp ne 0`将所有值为非零的`i32`转化为`i1 1`；值为零的`i32`转化为`i1 0`。

优化方向

生成LLVM后，并没有用到`Value`的`User`的列表，即并没有记录当前指令被哪些指令所使用，同时由于LLVM的特殊性，除了定义语句，其他值都是随用随从对应地址中取值，因此非定义的每个`Value`当且仅当被后续的一条指令所使用，记录下来并没有意义。

可以预见到如此做会产生极大量的内存读取指令，必然可以优化，根据`def-use`链的思想，在一个变量未赋新值前，都可以使用上一次赋值的虚拟变量，这样便省去了读取内存指令，然而这又涉及到基本块的关系以及分支间的选择(phi指令)，暂且留到优化文档中进行更深一步的探索。

编译命令

- 链接两`.ll`文件，生成可执行文件`clang main.ll lib.ll -o out.exe`
- 执行`.\out.exe`
- 控制台输入