Lab4 实验报告

杨杰 201908010705

实验要求

主要工作

1. 阅读cminus-f 的语义规则~~成为语言律师~~，我们将按照语义实现程度进行评分
2. 阅读LightIR 核心类介绍
3. 阅读实验框架，理解如何使用框架以及注意事项
4. 修改 `src/cminusfc/cminusf_builder.cpp` 来实现自动 IR 产生的算法，使得它能正确编译任何合法的 cminus-f 程序
5. 在 `report.md` 中解释你们的设计，遇到的困难和解决方案

PS：在lab3 里面由人将 c 翻译成 cpp 的工作，在 lab4 里，我们需要通过对抽象语法树的结点**自顶向下遍历**，让机器自动生成对应的 cpp 文件。

需要阅读的材料

- cminus-f 的语义规则
- LightIR 核心类介绍：重点看 c++ 的核心类介绍，以及各个类的使用方法，他们实现在 `/lab4/src/lightir` 中。
- 理解访问者模式下，算法的执行流程，参考– 打印抽象语法树的算法 `/src/common/ast.cpp`
- 理解助教给出的，用于存储作用域的类 Scope：它是用来辅助我们在遍历语法树时，管理不同作用域中的变量。它提供以下接口：

```
// 进入一个新的作用域
void enter();
// 退出一个作用域
void exit();
// 往当前作用域插入新的名字->值映射
bool push(std::string name, Value *val);
// 根据名字，寻找到值
Value* find(std::string name);
// 判断当前是否在全局作用域内
bool in_global();
```

  你们需要根据语义合理调用 `enter` 与 `exit`，并且在变量声明和使用时正确调用 `push` 与 `find`。

  在类 `CminusfBuilder` 中，有一个 `Scope` 类型的成员变量 `scope`，**它在初始化时已经将 `input`、`output`、`neg_idx_except` 等函数加入了作用域中**。因此，你们在进行名字查找时不需要顾虑是否需要对特殊函数进行特殊操作。

工作内容的理解

1. 我们需要实现的函数是 CminusfBuilder 类中的一系列成员函数。

   由 .cpp 生成 .ll 的过程可以理解为：自顶向下遍历抽象语法树的每一个结点，利用访问者模式，调用我们写的成员函数，对每一个抽象语法树的结点进行分析，使之符合 cminusf 的语法和语义规则。

2. 从函数的传入参数开始，理解 visit 函数的工作流程：以第一个函数为例

```
void CminusfBuilder::visit(ASTProgram &node){}
```

传入参数为--抽象语法生成树上的一个结点（这里是根节点），结点类型为 `ASTProgram`

它的孩子是 declaration-list：

```
//ast.hpp
struct ASTProgram : ASTNode {
    virtual void accept(ASTVisitor &) override final;
    std::vector<std::shared_ptr<ASTDeclaration>>
        declarations;
};
```

也就是说，当我们遇到 ASTProgram 这个结点，需要做的事情是，向下遍历 (accept) 它的孩子，正如 cminusf 语法规则所要求的那样：`program → declaration-list`

此时，如果对应的 cminusf 语义有其他要求【具体信息来自实验文档】，也应该在这里实现【使用 lightir 核心类】。

最终的函数代码实现：

```
void CminusfBuilder::visit(ASTProgram &node) {
    // program -> declaration-list 深度优先遍历
    for(auto n : node.declarations){ // n 获取 declaration-list
        n->accept(*this);  // visit all declaration-list
    }
 }
```

3. accept() 函数的理解：

参考打印抽象语法生成树的代码，可以将 accept() 理解成一个递归调用，即向下递归遍历孩子结点。

实验难点

VarDeclaration

- 需要利用 `scope.in_global()` 来区分变量是在全局还是局部；
- 如果声明的是数组，需要检查数组下标 `node.num->i_val` 的值是否大于零；
- 根据 cminus-f 语义要求，全局变量需要初始化为 0；

FunDeclaration

一共需要进入两个子域：

1. 维护函数的参数及其返回值类型；
2. 维护函数输入的参数值和函数体内部语句；

紧跟着的 `Param` 用来分析函数的参数类型，并把参数 push 到作用域中；

- 注意如果参数是数组，则为数组引用传递（指针），且不需要考虑数组下标是否合法，因为根据 cminus-f 语法，这里的数组不会有下标。

`CompoundStmt` 维护函数体内部语句，根据语法向下遍历即可；

AssignExpression

$expression -> var = expression$

把 expression 的值赋给 var；

1. `var` 可以是一个整型变量、浮点变量，或者一个取了下标的数组变量。
2. 通过全局变量 `ret` 获取的 expression 的返回值有：int , float, int*, float* [表示返回的是存放 expression 值的地址]，int1，五种类型。需要分别讨论这些情况。
3. 在 `expression_value` 和 `var_value` 的类型不同时，需要以 `var_value_type` 为标准，对 `expression_value` 做强制类型转换。

SimpleExpression

simple-expression -> additive-expression relop additive-expression | additive-expression

- 浮点数和整型数一起运算时，整型值需要进行类型提升，转成浮点类型，其运算结果也是浮点类型；

Selection statement

- 首先，我们需要向下遍历if的判据（即expression）：
    1. expression类型：指针（int,float型），int，float，bool。
    2. 如果是指针的话，我们需要将指针指向的值load出来。
    3. 如果是bool类型，则将其强制类型转换为int32类型。
    4. 这样最后都会转化为int32或者float类型。用对应的cmp函数比较其值是否为0即可。
    5. 作为分支判据的expression的处理都是类似的，后面的iterationstmt实现方案和selectionstmt是非常相似的，所以后面iteration部分不再赘述
- 其次还要检测selection stmt展开之后是什么类型，有没有else stmt，如果有的话就创建真假分支后创建跳出分支，没有的话只需要创建真分支和跳出分支即可
- 注意到，在每个statement内部，我们需要考虑statement内return
- 在selection stmt和后面的iteration stmt中，为了解决if嵌套if，while嵌套while的情况，建议在创建Basicblock的时候，使用默认的寄存器编号作为名字，不然可能会出现嵌套时分支重名的情况

Return statement

- 由于expression可能类型的多样性，如果是指针类型，我们需要把指针load成一个对应的值
- 若return void；则直接创建空返回就好
- 要注意到：我们需要用`builder->get_insert_block()->get_parent()->get_return_type()`来检测函数返回值类型与我们实际要返回的类型是否相同，如果不同，则要进行**强制类型转换**，比如

```
else if ((ret->get_type() == typeint && builder->get_insert_block()->get_parent()->get_return_type() == typefloat))
            {
                auto ret_load = builder->create_sitofp(ret, typefloat);
                builder->create_ret(ret_load);
            }
```

Var

- 首先需要在scope中找到id
- 因为var可能被展开成单纯的ID或者ID[expression]，即数组类型，于是我们需要根据是否有expression进行讨论
- 如果是纯id，则返回id地址即可
- 如果不是，则是数组类型，即ID【expression】
- 先讨论expression，因为expression类型的多样性，这里对于最终的数组下标而言，我们需要的是一个int32，所以需要将它们全部转化为int32。
- 然后由于数组下标非负的限制，我们需要创建一个判断分支去看是否大于0（value*不能直接用if语句比大小），true分支里面，讨论一下id的类型，id的类型可能是：int**,float**,int*,float*，后面两个类型是普通的整数和浮点数数组类型，对于int**和float**类型，它本质上是一个指针，指向了一块内存空间，这个内存空间存放了数组首地址。我们希望将数组取出来，这部分主要是用于数组传参，后面实验设计部分会有更加详细的解释。到最后，任意id都会成为int*,float*类型，然后我们用create_gep取值并返回即可
- 如果数组下标小于0，则调用neg_idx_except报错

ASTAdditiveExpression

additive_expression 可以生成两种表达式，分别是`term`和`additive_expression addop term`。

1. 如果是`term`，那么直接调用它的accept函数，就可以完成访问者遍历并跳转到子节点（term）
2. 如果是`additive_expression addop term`，那么要对加减法符号两端对操作数进行类型转换。
3. 最后根据操作数是int还是float来调用相应的加减法函数

ASTTerm

term 和 additive_expression 的产生式形式相同，处理方式几乎完全一样

ASTCall

call分为两部分，即函数名的处理和函数传入参数列表的处理。

对于函数名，需要做的是进入到它对应的作用域里面，并取出函数声明时的形参列表（用于检查和传入参数是否一致）。

对于传入参数，需要将指针/数组元素对应的值“取出来”，存入vector中，并与声明时的参数列表比较类型

这个函数的难点：

1. 函数调用的参数可以为空；
2. 需要检查调用时实参和函数声明时参数的个数和类型是否一致，如果不一致，在必要时需要进行强制类型转换；

获取被调用函数的参数类型列表的方法：

```
/* enter the scope of terminal symbol ID */
        auto IDalloc = scope.find(node.id);
        auto fun = IDalloc->get_type();
        auto callFun = static_cast<FunctionType *>(fun);
        std::vector<Type *> params; // get the params value of the function
```

> 这里不用考虑数组的类型不一致问题。

实验设计

全局变量 ret

存放 expression 的返回值，以便需要时可以直接获取；

- 该返回值可以从 Num 中得到 -> 一个数字(int or float)；
- 从关系计算式得到 -> int1
- 通过简单的运算或者从数组中取出，这时存放的是对应值的地址 -> int* or float*

如何处理数组传参

请看这个函数

`int func_array(int a[])`

在这个函数中，**数组首地址**作为形参传入`int func_array(int a[])`,函数会开辟一块内存来存储形参。在`int func_array(int a[])`中，形参类型是int*,指向存储int*内存空间的指针类型是int**。针对这类情况，我们处理int**或者float**的流程如下，在下面函数中，`id_type`是`id`的类型：

- 首先，将id当成一个二维数组，使用create_gep函数`id_load = builder->create_gep(id_type, {CONST_INT(0), CONST_INT(0)});`取出内存空间中存放的数组首地址。
- 再用`id_load = builder->create_gep(id_load, {temp_ret});`(temp_ret是数组下标)取得id[temp_ret]的地址

通过这两个步骤，我们可以访问传进子函数的数组中的任意元素

降低生成 IR 中的冗余 -- 丢掉同一块内 return 后的语句

主要思想：

1. 设置全局变量标识遍历到的结点之前是否出现过`return`语句，初始化为0；`int isAfterRetrun = 0;`
2. 在每个结点的函数开始处，判断 isAfterRetrun 是否为0；若是，则可以继续分析，若不是，则不对其进行任何处理（丢掉冗余结点的分析）。
3. 每个 return 结点遍历结束时，`isAfterRetrun = 1;`
4. 每个函数执行结束时，重置`isAfterRetrun = 0;`
5. 每个 while 块执行结束时，`isAfterRetrun = 0;`
6. `if-statement`，没有 else 分支，且在 if 分支内部出现了 return：

```
node.if_statement->accept(*this);

if (isAfterReturn){
    localFlag = 1; // 区分 if-then 和 if-then-else
    isAfterReturn = 0;
}
```

7. `if-then-else` 分别讨论只在一个分支出现 return 和两个分支都出现 return 的情况：

```
if (isAfterReturn){
    localFlag_2 = 1;
    if (localFlag){
        isAfterReturn = 1; // if & else 都有 return ，则 if-else 后的代码块没有必要执行;
    }
    else
    {
        isAfterReturn = 0;
    }
}
```

结果验证

```
(base) yj@myubuntu:~/Documents/编译原理/cminus_compiler-2021-fall/tests/lab4$ python lab4_test.py
==========TEST START==========
Case 01:        Success
Case 02:        Success
Case 03:        Success
Case 04:        Success
Case 05:        Success
Case 06:        Success
Case 07:        Success
Case 08:        Success
Case 09:        Success
Case 10:        Success
Case 11:        Success
Case 12:        Success
============TEST END============
```

实验总结

深入理解了中间代码的生成过程。

最大的感受是发明编译器的前辈们实在是太厉害了，clang 生成的 .ll 文件精简又准确，在函数需要返回语句，但缺少时，甚至能自动补全一句 return；

编译器就像一个优秀的翻译官，将我们的代码翻译成机器能听懂的语言的同时，对我们的语言进行了精简，使得机器运行的效率提高。

自己在写 c 程序的时候并没有关心过自己写的 c 语句是如何被机器识别成二进制代码。从学了编译原理，写了一次次实验之后，终于对代码的编译过程甚至是计算机行业有了更深入的了解。越来越觉得，计算机相关行业无一不是在让人们的生活变得更加便利和快捷，未来可期！

最开始写实验的时候尝试从 call 函数倒着写，最后发现是一个相当大的失误。对语法树的处理是自顶向下的，思维随着编译器，对一条语句自program（root）而下遍历，对途径的产生式展开和选择，每一个结点的处理才逐渐浮现在眼前。在后期调试和debug的过程中，也有一点别的体会，比如对于return之后冗余语句的省略，除了本次实验的实现方法外，我觉得还可以通过修改语法规则来实现优化：

$\text{compound-stmt} \rightarrow \textbf{{}\ \text{local-declarations}\ \text{statement-list}\ \text{return-stmt}\ \text{redundancy-stmt} \textbf{}}$

$\text{redundancy-stmt}\to\text{statement-list}$

$\text{statement}\to \text{expression-stmt}\mid \text{compound-stmt}\mid \text{selection-stmt}\ \mid\ \text{iteration-stmt}$