编译原理实验报告三

161220070 李鑫烨

完成的功能点

实验三在之前实验排除语法、语义错误并建立语法分析树的基础上，递归地遍历分析树，对于树中特定节点进行翻译，生成线性中间代码并进行调整优化。在完成实验必做内容同时，完成了以下内容:

- 支持结构体变量与高维数组的翻译。
- 结构体与数组类型均能作为函数参数。
- 保证运行正确条件下消除冗余临时变量与常量计算，尽可能提高运行效率。

实现思路

实验三内容为将源代码翻译为机器无关的中间代码，主要步骤为

- 设计中间代码数据结构，使其存储尽可能多信息并有利于翻译与调整优化过程。
- 遍历语法分析树，对于特定语法节点调用对应翻译函数生成中间代码。
- 对中间代码进行调整优化，通过消除连续赋值与运用代数恒等式消除冗余临时变量与常量计算，提高中间代码效率。

数据结构

编译器中非常核心的数据结构即为中间代码，其中包含哪些信息与其内部表示需要仔细设计。实验中采用了线性中间代码及四元式表示，保持灵活性以便于之后进行调整优化。

中间代码逻辑最顶层的数据结构为 CodeBlock，存储中间代码段，对中间代码段的线性链表结构进行封装并提供拼接、获取结果等接口，为以后重构中间代码逻辑结构提供灵活性，也为中间代码生成完毕之后的调整优化提供载体。

```
1  class CodeBlock {
2    list<InterCode> code;
3  public:
4    void append(CodeBlock);
5    Operand* getResult();
6    void optimize();
7  };
```

中间代码最关键的数据结构为 InterCode，存储单条中间代码中操作符类型、操作数等，并针对不同指令提供不同的构造函数，提供获取操作符类型接口，并对于存在返回值的指令提供获取结果接口。实现中，枚举变量 kind 存储操作符类型，操作数指针 op1 和 op2 存储操作数，若指令存在返回值，则将指令运算结果存储于指针 result 中。

```
class InterCode {
  enum interCodeType kind;
  Operand *result, *op1, *op2;
pulic:
  Operand *getResult();
  enum interCodeType getType();
  InterCode(enum interCodeType kind, Operand *op);
  InterCode(enum interCodeType kind, Operand *op1, Operand *op2);
};
```

存储操作数的数据结构为 Operand。由于 Operand 类型变量多以指针方式被访问，同时操作数具有多种形态，如临时变量、常量、符号变量以及函数标号等等，实验中采取继承的方式描述不同类型的操作数。对于 Operand 类型，实验中仅存储其操作数类型并提供接口，某些操作数类型特有的操作均封装于其对应派生类中。Operand 类型的实现相当简单，枚举变量 kind 存储操作数类型。Operand 类的派生类较多，不在此一一罗列具体实现。

```
class Operand {
  enum operandType kind;
public:
  enum operandType getType();
};
```

数组与结构体类型处理

考虑它们作为函数参数的情形，数组与结构体类型变量如何处理的问题难以处理。语言并未提供指针这一机制，但在翻译数组或结构体类型变量过程中，我们需要计算相对偏移量并进行取值，需要用到临时变量存储地址，不可避免地用到指针这一概念。当数组或结构体类型变量作为函数参数传递时，由于涉及到之后实验运行时环境与语言传值调用与传址调用的问题，我们并不能简单地将整个变量传递。

实验给出解决方案为将操作数区分为值类型与指针类型，具体实现中运用指示变量予以区分，对指针型操作数进行赋值、算术运算或比较时运用取值操作转换为值类型变量。对于数组与结构体类型变量，将其视为指针类型，初始化为指向其首地址的指针。处理之后进行函数传参时只需传入其指针，类似于传址调用。

由于虚拟机中为变量申请一段内存空间之后，该变量不能转化为指针类型。具体实现中将数组或结构体类型变量转化为指针运用了 trick，先为变量申请相应大小的内存空间，然后创建同名符号指针类型变量，初始化为这段内存空间首地址。因此之后取用该数组或结构体类型变量时，将直接取到该指针类型变量。比如翻译如下代码

```
int main() {
  int a[10][10];
  a[10][10] = 1;
}
```

编译器将其翻译为中间代码如下

```
FUNCTION main :
DEC a_var 400
a := &a_var
t0 := a + #80
t1 := a + #8
*t1 := #1
return #0
```

中间代码优化

实验中并未采用复杂的全局优化方法或进行中间代码结构调整，而是进行局部优化，通过消除连续赋值与运用代数恒等式消除冗余临时变量与常量计算，提高中间代码运行效率。

主要优化思路在于消除中间代码中冗余的赋值语句。观察后发现，中间代码的主要冗余指令在于对于计算结果的连续赋值，从而产生冗余临时变量。比如以下代码，

```
a = 0 + 1;
```

完全不经优化时，编译器可能翻译为

```
t0 := #0
t1 := #1
t2 := t0 + t1
a := t2
```

消除冗余赋值语句优化方法分为过程中优化和翻译完毕后优化。过程中优化主要方法为

- 在翻译赋值语句 $E = E_1$ 过程中，对于 E_1 进行进一步向下发掘，如果 $E_1 \to E_2 + E_3$，整个语句可展开为 $E = E_2 + E_3$ 能用一条基本指令表达的形式，直接翻译为一条基本指令，简化了其中冗余过程。
- 在不经优化时，编译器对于源代码中常数或符号，往往产生赋值语句来对其进行取值，产生冗余临时变量。因此我们在翻译常量或者符号表达式时，不产生中间代码而是直接返回操作数，简化了不必要的取值过程。

过程中优化后，上述代码将翻译为

```
a := #0 + #1
```

翻译完毕后优化主要方法为提前计算常量以及应用代数恒等式。对于可提前计算常量的中间指令可计算结果后优化为赋值指令。对于 $t0 := t0 + \#0$ 此类指令可运用代数恒等式优化为赋值指令。在提前计算常量和应用代数恒等式过程中，产生了新的赋值指令，可能形成连续赋值的可优化的情形。为保证中间代码正确性，我们将中间代码划分为基本块进行处理。将中间代码划分为基本块之后，在块内搜索赋值指令，将块内所有赋值指令左边操作数替换为等价的右边操作数，最终消除该赋值指令。在这过程中可能产生可提前计算常量和应用代数恒等式的情形，因此迭代优化直至无赋值指令可消除。上述代码翻译为

```
a := #1
```

编译运行方法

在 lab3/Code 中输入以下指令

- make parser: 生成可执行文件 parser 并替换上级目录中 parser。
- make run: 已生成可执行文件 parser 条件下，测试 lab3/Test 中四个样例。
- make clean: 删除可执行文件以中间文件

若需测试 lab3/Test 中 filename.cmm，则在 lab3 目录下输入如下指令

./parser Test/filename.cmm > Out/filename.ir

运行示例

对 Test/quick_sort.cmm 进行测试，样例中采用快速排序，对长度为 5 的数组进行排序。

- 未进行优化的情况下，中间代码长度为 185，虚拟机运行结果正确，共用 644 条指令。
- 进行优化的情况下，中间代码长度为 100，虚拟机运行结果正确，共用 354 条指令。

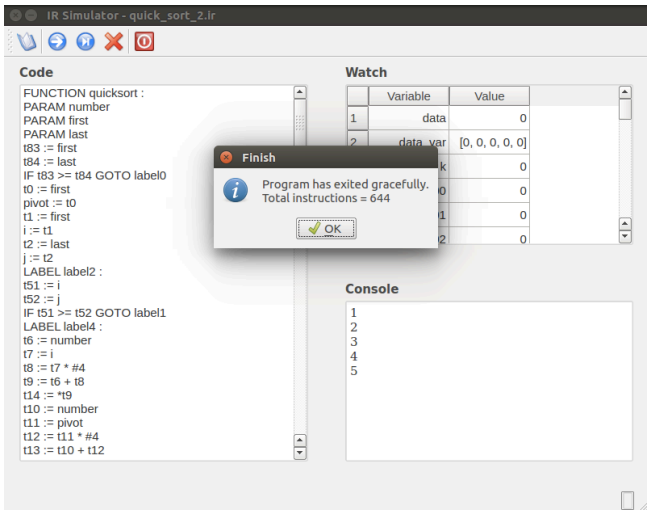

Figure 1: 未优化运行结果

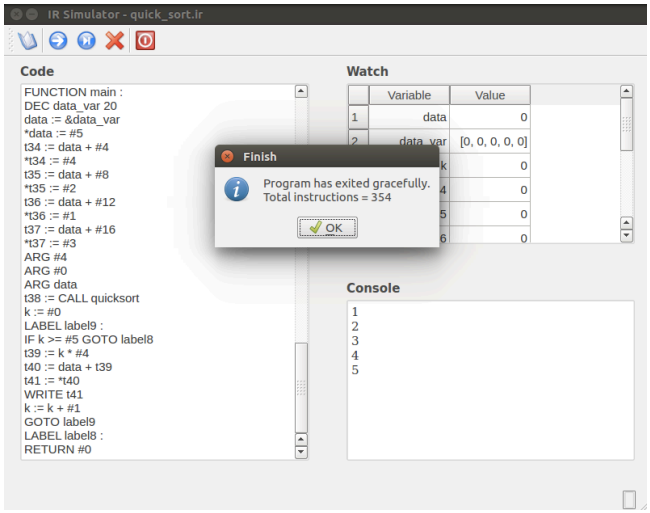

Figure 2: 优化后运行结果

在排序这种数值运算相对较少，控制流相对复杂的场景下，编译器能够翻译产生正确的中间代码并且起到一定的优化作用。