编译原理实验报告

Lab3 中间代码生成

151220069 罗义力
luoyl233@foxmail.com

一、已完成功能

1.中间代码生成：将 C—源代码翻译为中间代码
2.选做 3.1：源代码中可以出现结构体类型变量，结构体类型变量可以作为函数参数
3.基本块优化

二、实验环境

GNU Linux Release: Ubuntu 12.04, kernel version 4.4.0-66
GCC version 5.4.0
GNU Flex version 2.6.0
GNU Bison version 3.0.4

三、使用说明

make: 编译生成分析器 parser。
make clean: 清除所有的中间文件和目标文件。
程序执行方式：

./parser <input_file> <output_file>

将输入文件翻译为中间代码，并保存在输出文件中。

四、实验过程

1. 操作数和中间代码结构体定义

操作数结构体体 struct Operand 和中间代码结构体 struct InterCode 基本与实验文档一致，不过在操作数和中间代码的类型上要更加详细。

```
typedef enum OP_TYPE {
    VARIABLE,              //变量类型，包括整型变量，结构体，函数等，使用 name 域
    CONSTANT,              //常数，使用 Operand 的 value 域
    TEMP,                  //临时变量，使用 var_no 域
    LABEL,                 //标签，使用 var_no 域
    REF,                   //引用，即指向变量 op 的地址
    DEREF                  //解引用，指向 op 代表的地址所指向的值
} OP_TYPE;
struct Operand_ {
    OP_TYPE kind;
    union {
        int var_no;        //临时变量/标签
        int value;         //整型数的值
        char *name;        //变量名字
        Operand op;        //引用或解引用对应的操作数
```

```
    };
};
```

可以看到，我是把引用和解引用放到了操作数结构体中表示的，这样做一方面是为了精简代码，另一方面出于选做部分的需要，这点在选做实现部分具体说明。如果把引用和解引用设置为中间代码类型，为了详细地指出引用和解引用对应的是哪个操作数，则需要更加具体地定义中间代码类型。如下面的中间代码类型：

```
t12 := &t3 + #4
*t12 := t3 + #4
t12 := #4 + &t3
```

可能就得需要为上述三种类型的中间代码分别定义三种类型了。

如果把引用和解引用都设置在操作数结构体中，那么上述的三种中间代码可以只用一种中间代码表示：

```
rlt := op1 + op2。
```

其中 rlt 根据操作数类型可以是 t12 或者*t12，op1 可以是&t3 或者 t3 或者#4。

中间代码无论哪种类型，都是以<op, rlt, op1, op2>的格式储存。对于一元类型的中间代码，如 PARAM，LABEL 等，rlt 和 op2 为 NULL，即<op, NULL, op1, NULL>。对于二元类型的中间代码，如 ASSIGN，op2 为 NULL,即<op, rlt, op1, NULL>。

2. 线形方式表示中间代码

中间代码以双向链表的形式储存，与实验文档保持了一致。在后面优化的时候，双向链表修改更加方便。

3. 必做部分的翻译

这部分基本就是将实验文档提供的伪代码实现一遍，为每种类型的语法树结点，实现函数 translate_xxx()。

4. 选做 3.1(结构体的访问和传参)

对结构体部分的翻译涉及到对内存地址的访问，这部分应该是引用和解引用类型真正会被使用到的地方。如果只考虑对简单结构体的访问，根据 Exp1 -> Exp2 DOT ID，从 Exp2 中获取结构体变量名，再从符号表中获取结构体域，根据如下公式：

$$\mathrm{ADDR}(\mathrm{st}.field_n) = \mathrm{ADDR}(\mathrm{st}) + \sum_{t=0}^{n-1} \mathrm{SIZEOF}(\mathrm{st}.field_t)$$

就能算出偏移量了。

但是结构体可能是嵌套的，Exp2 可能是一维数组、结构体或者 ID。在计算当前结构体偏移量之前，需要先通过 Exp1 获取结构体链表(通过修改参数 type 指向的值)，并计算偏移量。

我将结构体的翻译实现在函数：

```
InterCodes* translate_Structure(TreeNode* exp, Operand place, Type *type)
```

首先判断 Exp2 的类型，若为 ID，直接从符号表中获取变量类型，计算偏移量，如果要获取的域变量是 INT 型，则返回值，若为数组或者结构体类型，则返回地址；若为结构体，则先递归调用 translate_Structure()计算外层偏移量，并从 type 中获取结构体链表，计算当前域的偏移量；若为数组，则调用 translate_Array()计算结构体基地址，从 type 中获取结构体链表，计算当前域偏移量。

另外需要修改的地方是 translate_Exp()函数中对赋值语句的操作。赋值号左边或者右边可能出现对数组和结构体的访问，右边的和普通情况一样处理，但出现在左边就意味这对结构体和数组的修改，直接传值无法正确给结构体和数组赋值。因此，需要使用解引用类型的操作数，比如对结构体 st，在 translate_Structure()中对 INT 型域返回的并不是 place := st.a，

因为这样无法修改 st.a 的值，而是将 place 修改为*(st.addr+offset)，对 place 的修改就是对相应域的修改。

5. 基本块优化

根据前面生成的中间代码，对每个基本块生成有向无环图(DAG)，再转化为中间代码，达到优化的目的。

DAG 节点数据结构如下所示：

```
struct DAGNode_
 {
      BOOL isLeaf;              //DAG 节点分为叶子节点(初始变量)和内部节点(操作符)
      IC_TYPE kind;             //内部节点的操作符类型
      Operand signList[SIGN_SIZE];     //所有节点都有可能与多个符号相关联，但一个符号同
                                       //时只能与一个节点关联
      int signSize;             //signList 用数组表示更方便，signSize 确定关联符号数目
      Operand op;               //叶子节点的初始变量
      DAGNode left, right;      //内部节点的左右子女
      Operand activeSign;       //在 DAG 转化为中间代码的时候使用
 };
```

对没有指针出现的基本块，优化效果非常好，所有多余的临时变量赋值操作都会被优化掉；但如果出现了指针，如*p := xxx，这对整个基本块将是灾难性，因为这种类型的指针操作会杀死所有的节点，所以，对指针出现的基本块，我直接跳过优化。