编译原理实验三

111220174 张赫
mattzhang9@gmail.com

总体说明：

本次实验的内容是使用语法树生成中间代码并进行优化。我采用生成语法树之后遍历树的方式生成中间代码。完全生成后再进行优化然后输出为中间代码。

中间代码采用双向链表的数据结构。DAG虽然便于优化，尤其是优化死代码块和重复代码，但实现起来太过复杂故没有被采用。双向链表在保证开发效率的同时最大限度地保证了对代码优化的便捷。除了控制流以外基本使用指导攻略里的翻译模板。为了优化，控制流使用了FALL的方式翻译。

翻译成中间代码：

1.从根节点开始遍历树，对于重要节点进行相关处理。对于声明部分完全不做处理。对于数组（分组二），在第一次使用的时候分配空间。对于倒序的函数形参实参，采用栈的方式将其倒转为正确的顺序。

2.Exp的翻译是重中之重，也是代码量最大的。但主要来说都是按照模板翻译，不是按模板翻译的也是用FALL方式翻译和跳转相关的内容。

3.各种跳转的翻译在早期我尝试了很多自己想出来的方法，但都在某种情况下有问题。最后想到书上曾经提到过IF false和FALL结合的方式翻译，遂采用。

4.数组：数组的翻译在指导攻略上并没有写，实际上也非常复杂。因为有的时候是要取数组的值，有的时候是要数组的地址。于是定义了一个Wrap(translateArray)，用这个包裹需要被递归调用的Iter(translateArrayIter)。并且，对于需要地址或者数值的情况进行分类，最终返回一个被需要的值。

5.数据结构：对于操作数和中间代码，分别定义了两个struct。根据各种不同的操作数和中间代码设定相应地Type和相关信息。操作数型包含区分操作数类型，变量标号（对于代码中的变量会将这个变量标号加入到符号表）和变量是否为指针以方便输出。对于中间代码型，包括中间代码类型，操作名（或理解为对于操作数的函数）和操作数。

中间代码优化：

因为没有使用DAG，使得优化非常困难。再加上有跳转的问题，更不能像线性执行一样进行优化。于是只能仅最大可能进行优化。优化了三个方面：相邻的Label，临时变量和无用代码。

1.相邻Label：这种主要会出现在多个else if的时候，最后的LabelFalse会相邻，事实上只要有一个就可以了。对于没一个重复的Label，向上扫描直到找到跳转到这个Label的语句并将其跳转到到Label改为唯一剩下的那个Label即可。

2.临时变量：因为经常会有place，所以会有巨大量的临时变量。一种简单且无错的方法是合并相邻赋值语句中得临时变量。这种方法虽然非常简单，但可以减少很多无用的代码。当然，还有其他更加复杂的方法。对于二目操作，IF的条件，返回等操作，均可以与赋值语句合并，其中一些语句还可以和二目操作合并。

但是在这之中需要考虑临时变量是否改变（当exp的操作符为RELOP等的时候），被删除的临时变量是否还会被用到（数组）的问题。简单的方法是只合并相邻的，在我的优化中用optVerboseTemp()来用这种方式进行合并。但这种方式并不能优化隔了几个的。比如t = v3; ARG v4; ARG t。所以我用optAdvance()来优化可能会相隔几个的语句。在这个优化中，只要碰到有可能不顺序执行或者改变、使用被优化临时变量的地方就停止。

3.常量优化：当一个临时变量被赋值为一个常量的时候，一般都会在优化临时变量的时候优化掉。但是当这个语句是一个算数表达式的时候，并不会被优化。于是，如果发现代码是个双目操作且操作数都是常量，则将双目操作改为赋值，赋的值即为运算结果。这样一来，可以在下一次循环的临时变量优化中将其优化。

4.无用代码：在一个程序中，事实上只有和Write有关的操作是有意义的。推广一下，对于函数而言，只有关于Write和Return的部分是有意义的。于是最直接的想法是可以将所有相关代码标记出来，其他删掉。但是，有两类代码虽然不直接相关但有可能间接相关。一类是跳转，另一类是数组。所以对于这两类也要认为其代码是有意义的。事实上，对于这两部分的进一步优化只能采用虚拟机的方式完成，否则很难做到。

本来比较想优化重复代码的，但尝试了很久还是没有成功，这也是不用DAG的最大问题。

总结:

本次试验花了整整一周半。因为上个实验有些问题，所以这次实验先把上个实验的漏洞修复了。同样因为吸取了上次的教训，这次的实验做了充分的测试，甚至使用快排来测试。

本实验翻译部分的难点在于理解place、想到用FALL处理控制流以及处理既可能需要值又可能需要地址的数组。这些部分都花去了很多写和调试的时间。优化的问题在于没有成熟的算法，DAG太过麻烦只能自己想Tricky的方式优化。而且优化之路永无止境，不满足的话就没时间做其他大作业了。而且优化的时候会有很多自己没有想到的情况，如果忽略了这些情况就会使代码无法运行。这些问题包括但不仅限于：*t = 2 + 3的形式非法、翻译带Relop的Exp会隔一个IF对同一个临时变量两次赋值，Label实际上是不可以在临时变量时穿越的。这些都大大增加了测试的麻烦。

使用方法:

编译`parser`请先进入source文件夹。
输入 `make parser` 来编译 parser。
然后输入 `./parser 文件名` 以获得中间代码，中间代码会被存储在 code.ir 文件中。
输入 `bash irsim.sh` 来启动irsim。