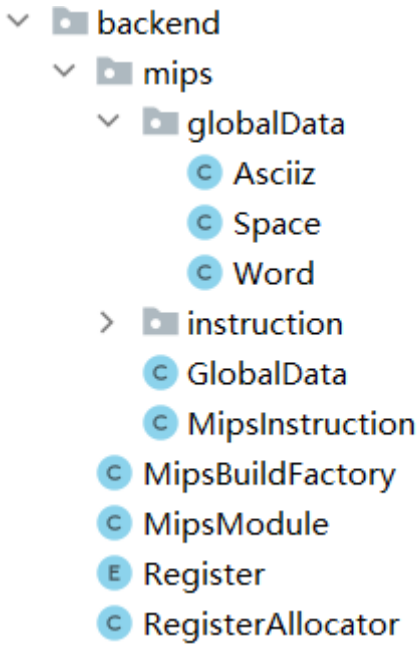
优化文章

21373216 李嘉鹏

目标代码生成

总体架构

在mips总体架构上，可以仿照llvm的总体架构实现，单例模式，工厂模式等等。与llvm实现类似，这里就不再赘述。



由llvm翻译mips

两者比较与关系

llvm:

- 标识符变量均存于内存中
- 寄存器存储内存地址[i32*]/值[i32]
- 临时寄存器"无穷多"

mips:

- 标识符变量在栈中分配内存（在不进行任何优化的情况下）
- 寄存器存储栈地址/值
- 两组映射关系：
  - 存地址[i32*]/存值[i32]：每个value均有可能分配一个寄存器【Value2Reg】
    - 即：Value2Reg，Reg中存Value的值
  - 若生成出来Value的时候，没有分配到寄存器，在栈中分配内存【Value2Offset】
    - 即：Value2Offset，栈中`$sp+offset`存Value的值

翻译指令

在每个需要翻译的llvm的Value类中，重写toMips()方法，将llvm翻译成mips对象，放入到mips框架容器之中

```
public class LoadInstr extends Instruction {
    private Value srcPointer;

    public LoadInstr(String name, Value srcPointer) {  //Value pointer 参数，通过查符号表得到
        super(((PointerType) srcPointer.getType()).getTargetType(), name);
        this.srcPointer = srcPointer;
        addOperands(srcPointer);
    }

    @Override
    public String toString() {
        return name + " = load " + type + ", " + srcPointer.getType() + " " + srcPointer.getName();
    }


    @Override
    public void toMips() {
        super.toMips();
        MipsBuildFactory mB = MipsBuildFactory.getInstance();
        // 读baseReg，写tarReg；因此默认初始寄存器可以均是K0
        Register baseReg = getRegForI32Pointer(srcPointer, Register.K0);
        Register tarReg = mB.getRegOf(this);
        if (tarReg == null) tarReg = Register.K0;
        //执行核心功能
        new MemoryMipsInstr(MemoryMipsInstr.Op.LW, tarReg, baseReg, 0);
        //  Load 是 i32*
        // 生成了一个Value，保存到寄存器（若未分配寄存器，则保存到栈中，并做映射）
```

```
        if (mB.getRegOf(this) == null) {
            saveValueIfNotExistMatchRag(this, tarReg);
        }
    }

}
```

函数调用

函数调用是mips翻译的难点

核心思路如下所示：

```
    @Override
    public void toMips() {

        /*
        1.保存现场
         */

        // 寄存器如何分配，注意一下，！！比如哪些并没有在分配表中，但还是使用了，比如a0(syscall使用),sp和ra...??k0,k1(好像无所谓)
        // 保存分配表中寄存器

        // 保存$sp,$ra

        /*
       2. 传递实参
         */

            /*
            注意！！！：syscall打印整数与字符串的时候，为了性能，没有保存$a0现场值。因此函数传参的时候，不要使用$a0寄存器！！
            第一个参数 -> $a1
            第二个参数 -> $a2
            第三个参数 -> $a3
            其余参数，入栈
             */
            // 为被调用函数填写a1-a3寄存器，但仍然在栈中预留空间

            // 其余实参存入栈中


        /*
        3.栈帧设置
         */

        new AluMipsInstr(AluMipsInstr.Op.ADDI, Register.SP, Register.SP, curOffset - numOfReg * 4);


        /*
```

```
        4.调用函数【核心功能】
         */

        new JumpMipsInstr(JumpMipsInstr.Op.JAL,
targetFunc.getName().substring(1));


        /*
        5.还原现场
         */

        // 恢复$sp $ra

        // !!注意：恢复完$sp后，$sp已经指向当前函数的栈底，偏移量和未调用函数之前相同
        // 恢复分配表中寄存器

        /*
        6.处理返回值
         */

        // v0中获取返回值，也即call指令的值
        // 生成了一个Value，保存到寄存器（若未分配寄存器，则保存到栈中，并做映射）

    }
}
```

其中的坑点是，如果实参本身就是当前函数的形参的情况，当前函数形参的a0-a3寄存器 已经被填写为被调用函数的形参！需要从栈中保存的现场中获取当前函数形参，需要注意考虑！

总的来说，生成的目标代码，有两部分

- ***①是程序本身的代码；***
- ***②是栈和现场维护的代码****（模拟实际中的栈的运行情况，相当于一个大模拟）*

函数调用call指令翻译主要生成②的代码

其他大部分指令翻译都是生成①的代码

【区分】：函数定义生成代码（①），与函数调用生成代码（②）

优化

常量直取

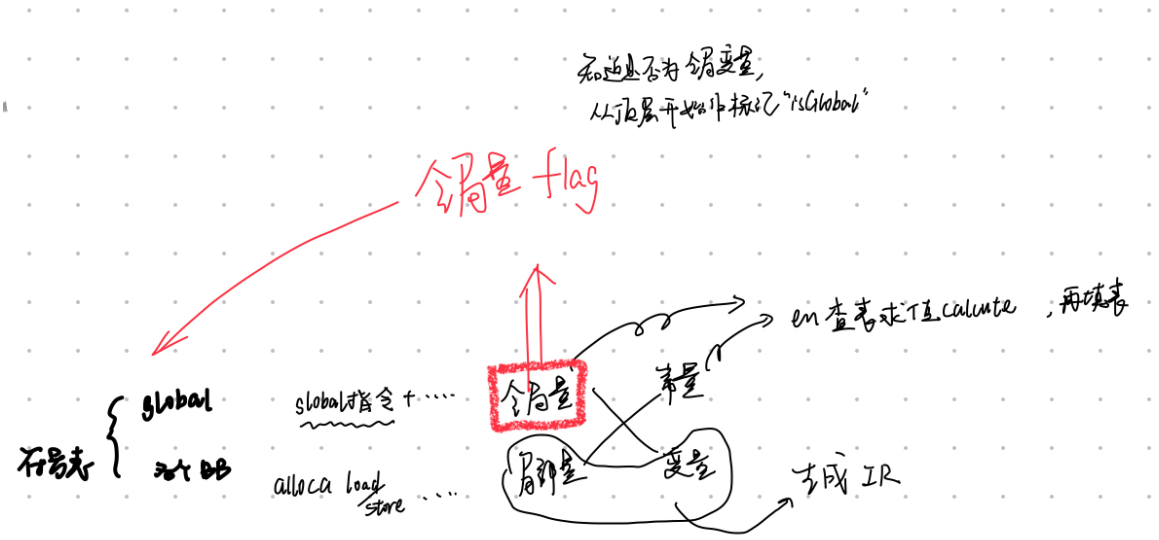
在建立符号表时，遇到const常量无论是int还是数组，都直接将其初值存储在符号表以及中间代码类之中，以便之后生成目标代码时直接取用。

这里需要在涉及常量or全局变量定义的语句，为Initial语法成分生成中间代码的时候，对其自顶向下计算求值。

与genIR的实现方法类似，实现位于Middle类的calculateEnsureNotArrayType方法，它是一个递归函数。为其传入语法树的根节点root，然后自顶向下递归调用，为所有的子节点生成llvm中间代码。

```
public static Value calculateEnsureNotArrayType(Node node) {
        SymbolManager symbolManager = SymbolManager.getInstance();
        ArrayList<Node> children = node.getChildren();
        switch (node.getSyntaxSymbolType()) {
            // CompUnit → {Decl} {FuncDef} MainFuncDef
            case "CompUnit":
                。。。
            case "Decl":
                。。。
            case "ConstDecl":
                。。。
            default:
                throw new IllegalStateException("Unexpected value: " +
node.getSyntaxSymbolType());
        }
    return -1;
}
```

详细逻辑可详见下图



数据流分析

这一部分都在Analyzer类中实现，主要涉及如下变量的维护与传递

```
    //数据流分析
    private ArrayList<BasicBlock> preListForFG = new ArrayList<>(); //flow graph
    private ArrayList<BasicBlock> sucListForFG = new ArrayList<>();
    private ArrayList<BasicBlock> domList = new ArrayList<>();
    private BasicBlock parentOfDom; // 支配树
    private ArrayList<BasicBlock> childrenOfDom = new ArrayList<>();
    //活跃变量分析
    private HashSet<Value> def = new HashSet<>();
    private HashSet<Value> use = new HashSet<>();
    private HashSet<Value> in = new HashSet<>();
    private HashSet<Value> out = new HashSet<>();
```

程序流图

构建程序流图比较简单，只需要通过跳转指令的目标BasicBlock对象，将各个基本块之间的关系存储到`HashMap<BasicBlock, ArrayList<BasicBlock>> preMap`和`HashMap<BasicBlock, ArrayList<BasicBlock>> sucMap`中即可

支配集合

由定义可将问题转化为，n1支配 n2，当且仅当，n1是从入口节点到 n2的必经节点。我们可以反向求解，求不经过n1可达的节点，然后取补集，即为所求。求解可达节点，就可以使用DFS搜索，`DFSForReachableBBWithout(n1, reachableBBSet, entry);`

支配树的构建

支配树的建立，即为求每个节点的直接支配节点集合，由定义：n1直接支配 n2 等价于 n1严格支配n2，且不严格支配任何严格支配n2的节点。可以求解

```
for (BasicBlock n1 : basicBlocks) {
    for (BasicBlock n2 : n1.getDomList()) {
        if (!n1.getDomList().contains(n2) || n1.equals(n2)) continue;
        boolean flag = true;
        for (BasicBlock other : n1.getDomList()) {
            if (!other.equals(n1) && !other.equals(n2) &&
other.getDomList().contains(n2)) {
                flag = false;
                break;
            }
        }
        // 满足直接支配关系
        if (flag) {
            parentMap.put(n2, n1);
            childMap.get(n1).add(n2);
        }
    }
}
```

活跃变量分析

根据编译理论课讲述的内容，由数据流公式in = (out - def) + use计算IN和OUT集合，循环计算直至OUT集合稳定不变。实现上，设置`change`标记变量来检查是否仍继续迭代。`change = !newIn.equals(inMap.get(basicBlock))`

由于llvm的SSA形式，可以省去做到达定义数据流分析的过程。

寄存器分配

由于中间代码为llvm，在寄存器分配的时候，我们实质上是为llvm的每个Value分配寄存器。且每一个llvm指令，至多只需要2个临时寄存器，即可完成操作。由于我们的代码没有和操作系统交互，k0和k1寄存器没有被使用的，可以作为临时寄存器，t0-t9,s0-s7寄存器全都作为全局寄存器来参与分配。

可以利用之前数据流分析得到的结果，进行图着色寄存器分配。

分配的过程中主要维护了两个映射关系reg2var和var2reg，其中var2reg为真正会在翻译指令的时候使用到的映射关系。所有可以将var2reg视为真正的分配结果，reg2var视为一个辅助temp映射关系。

```
public class RegisterAllocator {

    private Module module;
    private HashMap<Register, Value> reg2var = new HashMap<>(); //temp辅助
    private HashMap<Value, Register> var2reg = new HashMap<>();
    private ArrayList<Register> globalRegisters = new ArrayList<>();
}
```

整体实现如下：

1. 遍历基本块的指令列表，记录每个变量在该基本块里最后一次被使用的位置，并存储在`lastUse`中。
2. 再次遍历基本块的指令列表，对每个指令进行处理：
    - 如果该指令的某个操作数（operand）是该基本块内的最后一次使用，并且该基本块的输出（out）中不包含该操作数，并且该操作数在`var2reg`映射表中有对应的寄存器，则可以暂时释放该变量所占用的寄存器（从`reg2var`中移除），并将该变量加入到`neverUsedAfter`集合中。
    - 如果该指令属于定义语句，并且不是创建数组的alloc指令，则将该指令添加到`defined`集合中，并为该指令分配一个寄存器（通过调用`allocRegFor`函数），然后更新`reg2var`和`var2reg`映射表。
3. 遍历当前基本块的直接子节点，对每个子节点进行寄存器分配：
    - 如果当前寄存器所对应的变量在子节点中不会被使用到（即子节点的输入（in）中不包含该变量），则将该映射关系记录到`buffer`中，并从`reg2var`中移除该映射关系。
    - 调用`allocaForBB`函数为子节点进行寄存器分配。
    - 将`buffer`中记录的被删除的映射关系恢复到`reg2var`中，以免影响子节点的兄弟节点的寄存器分配。
4. 将该基本块定义的变量对应的寄存器释放，即从`reg2var`中移除这些映射关系。
5. 将“后继不再使用但是是从前驱基本块传过来”的变量对应的寄存器映射恢复回来：
    - 遍历`neverUsedAfter`集合中的每个变量，如果该变量在`var2reg`中有对应的寄存器映射，并且不是当前基本块定义的变量，则将映射关系添加到`reg2var`中。

```
public Register allocRegFor(Value value) {
    Set<Register> AllocatedRecordedInBoth = reg2var.keySet();
    for (Register reg : globalRegisters) {
        if (!AllocatedRecordedInBoth.contains(reg)) {
            return reg;
        }
    }
    return globalRegisters.iterator().next();
}
```

其中，整体思想为DFS遍历支配树进行寄存器分配，如果用通俗的方式来解释的话：

首先先为块内Value真正的分配寄存器，写到var2reg分配结果中。然后在第2步，根据活跃变量分析，如果某变量在该块之后便不再活跃，则将其设置为“虚位以待”的状态，一旦有其他变量有分配寄存器的需求，则将不活跃变量真正清除掉，否则，其仍保留在真正的var2reg分配结果中。在第三步，如果该块的直接支配节点，根本不用某个变量，则仍然设置为“虚位以待”的状态，递归遍历子直接支配节点；第四步，将该块的新定义的变量设置为“虚位以待”的状态，由于是DFS搜索过程，这样，其他兄弟节点定义更早的变量可以抢占定义更晚的变量的寄存器。

总体来说，可以更优先得让定义更早的，活的更久的变量，分配到寄存器。具有显著的优化效果。

窥孔优化

删除无用的`lw,sw`指令

删除无用的`move`指令

删除无用的`j,beq,bne`等跳转指令

乘除优化

乘法优化，如果操作数为常数，将乘法指令转换为左移和加法指令

除法优化，如果操作数为常数，可以按照公式，将指令转化为乘法指令，进一步优化

取模优化，可以将取模运算转化为除法指令和减法指令再进一步优化：$a \% b = a - a/b$