编译实习 MiniC 报告

罗昊　1700010686

2018 年 12 月 30 日

hcc(Hao miniC Compiler) is a simple compiler for MiniC.
It contains 2 parts: eeyore and tigger.
eeyore convert MiniC code to Eeyore code (a kind of three-address code)
tigger convert eeyore code to Tigger code || Riscv64 asm || Riscv32 asm.
Get source code from https://github.com/vangohao/hcc

1 supported additional C rules besides MiniC

1. 支持空语句 (;)
2. 支持逻辑表达式与算术表达式互相自动转换.
3. 支持无返回值调用函数.
4. 支持调用函数时使用表达式作为参数.
5. 支持在程序体内声明函数.
6. 支持 C 风格多行注释和 C++ 风格单行注释.

2 error report

1. 报告语法错误和词法错误及其行号, 并给出该处正确的 token 类型提示.
2. 检查标识符使用, 如果使用了未定义的标识符会报错.
3. 检查标识符重复, 对于重复定义的报错, 函数名与变量名冲突的报错, 如果在 {} 程序块内使用与程序块外同名的变量, 则不会报错.

4. 检查函数参数表, 对于重复声明但参数表不一致, 或定义与声明参数表不一致, 或调用时所用的参数表与声明的类型不一致时报错.

5. 检查 +,-,*,/,% 运算符对于数组类型变量的不合法操作给出错误提示, 这些操作中除了 (int[])+(int), (int)+(int[]), (int[])-(int), 外涉及数组的运算都是不合法的.

6. 检查对数组变量的赋值, 无法将数值赋给数组变量.

7. 检查 a[b] 使用, 如果 a 不是数组变量, 会报错.

3 warning report

1. 控制流到达函数结尾 (Control reaches end of function)

4 Eeyore

4.1 简介

1. 使用 flex,bison 和 C++ 构建, 将输入的 MiniC 代码转换为 Eeyore 三地址代码.

2. 使用 STL 的 map 模板制作符号表, 使用链表串联内外层程序块的符号表.

3. 使用回填法构建 eeyore 中的标号及 goto 语句.

4.2 代码结构

代码由 symbol.cpp/h node.cpp/h gotolist.cpp/h fault.cpp/h main.cpp parser.y lexer.l 构成

1. symbol.cpp/h 包含两个类的定义:

   (1) Symbol 类, 即符号类, 每个 Symbol 对象对应一个符号 (包括常数, 原生变量和临时变量)

   (2) SymbolTable 类, 即符号表.

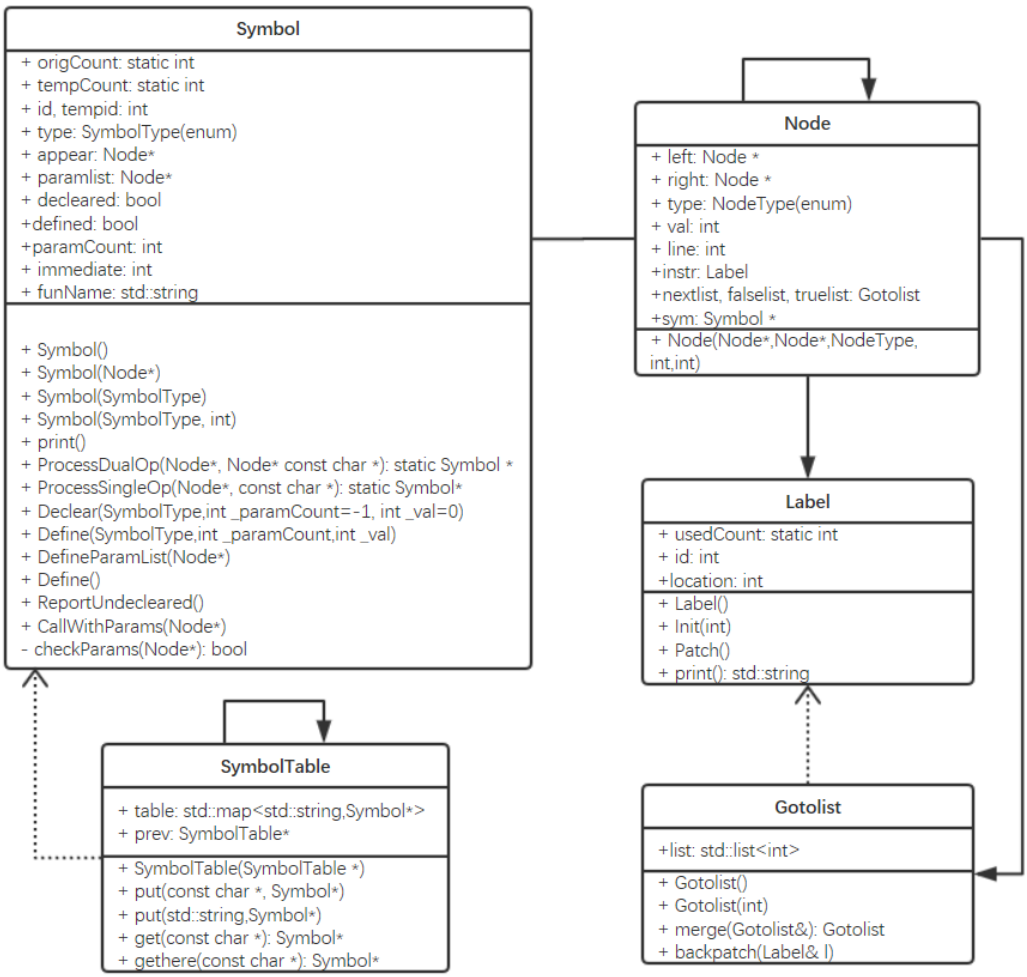
2. node.cpp/h 包含两个类的定义:

   (1) Node 类, 即语法树节点.

   (2) Label 类, 即标号类, 用于翻译生成的标号.

3. gotolist.cpp/h 包含 Gotolist 类的定义,gotolist 就是跳转表, 用于建立跳转关系.
4. fault.cpp/h 包含用于报错的函数.
5. main.cpp 主程序
6. parser.y 语法分析器, 进行语法制导翻译.
7. lexer.l 词法分析器, 进行词法分析并传递 token 及其基本属性给语法分析器

4.3 类结构



5 Tigger

5.1 代码结构

代码由 analyz.cpp/h lexer.l tigger.y tigger.cpp/h main.cpp 构成

analyz.cpp: 包含三个类的定义:

1.Expression 类, 每个 Expression 对象对应一条 Tigger/RiscV 指令;

2.Func 类, 每个 Func 对象对应一个函数;

3.Analyz 类, 单例类, 负责整体处理工作以及函数间优化工作.

analyz.h: 包含上述三个类的声明.

lexer.l: 词法分析器, 负责词法分析, 其中对于变量, 区分全局变量, 局部变量和形参, 使用 Analyz 中的 vcount 属性为变量分配 id, 并包含在 yylval 中.

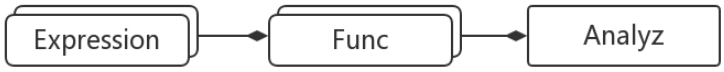
tigger.y: 语法分析器, 负责语法分析, 负责将 Eeyore 语句转换为 Expression 对象, 以及负责 Func 类对象创建, Func 类对象负责 tigger 的主要函数体内工作.

tigger.cpp: 包含一些公共函数.

tigger.h: 公共头文件.

main.cpp: 主程序函数和少量全局变量.

5.2 类结构



1. Expression 类:

```
class Expression
{
public:
    ExprType type;          //类型
    bool isMove;            //是否为传送指令
    bool dead;              //是否为死代码
    bool visited;           //是否被访问过(用于活性分析)
    vector<int> left;       //左值变量
    vector<int> right;      //右值变量
    vector<int> imm;        //直接数
    vector<int> use;        //使用的变量集合
    vector<int> def;        //定义的变量集合
    set<int> in;            //入口活跃集合
    set<int> out;           //出口活跃集合
    list<Expression*> nexts;//下一条指令集合
    list<Expression*> prevs;//上一条指令集合
    map<int,int> constant;  //常数变量表,用于常数传播
```

```
    string funtocall;       //仅用于call语句
     //构造函数，用于创建Expression并自动将指针加入当前所在函数的exprs表
    Expression(ExprType _type,vector<int> _left,
    vector<int> _right,vector<int> _imm,
    string _funtocall="",bool push=true);
};
```

2.Func 类

```
class Func
{
public:
    Func(int _paramCount,string _name); //构造函数
    void Processor();                   //主函数
    int insert(int s,int v);            //向栈空间添加变量，s为大小，
                                        //v为变量id，返回值为栈上的编号
    void ReturnFunc(int v,int t);       //处理Return语句，v为变量id或常数值，
                                        //t为选项(0表示常数，1表示变量)
    void CallParam(int v,int t);        //处理Param语句，v和t的含义同上
    void CallFunc(int v,string f);      //处理call语句，v为存返回值变量，f为函数名称
    int getParamVar(int r);             //获取形参对应的局部变量编号
    friend class Expression;            //将Expression声明为友元
private:
    int paramCount;                     //参数数量
    int paramToCallWithCount;           //调用参数计数器
    int frameSize;                      //栈空间大小(不含保存调用者保存寄存器的临时空间)
    int frameMaxSize;                   //栈空间大小
    unordered_map<int,int> frameSaveTable;  //记录被调用者保存寄存器的保存位置
    unordered_map<int,int> frameArrayTable; //从栈上数组变量id映射到栈上位置
    string name;                        //函数名称
    vector<int> offset;                 //栈上数据位置
    vector<int> size;                   //栈上数据大小
    vector<int> paramTable;             //形参对应局部变量表
    list<Expression*> exprs;            //语句表
    vector<int> spilledVariableFrameMap; //由变量id映射到上面的offset和size数组的下标


    //Color Algorithm图染色算法
    static int colorNumber;             //颜色数
    list<int> initial;                  //待初始化的节点表
    list<int> simplifyWorklist;         //低度数传送无关节点表
    list<int> freezeWorklist;           //低度数传送相关节点表
    list<int> spillWorklist;            //高度数节点表
    list<int> spilledNodes;             //溢出节点
    list<int> coalescedNodes;           //已合并节点
    list<int> coloredNodes;             //已染色节点
    list<int> selectStack;              //栈
    list<Expression*> coalescedMoves;   //已合并传送指令
    list<Expression*> constrainedMoves; //已约束传送指令
    list<Expression*> frozenMoves;      //已冻结传送指令
    list<Expression*> worklistMoves;    //待合并的传送指令
    list<Expression*> activeMoves;      //活跃的传送指令
    vector<vector<int>> adjMatrix;      //邻接矩阵
    vector<list<int>> adjList;          //邻接表
    vector<int> degrees;                //度
```

```
47      vector<int> alias;                  //别名
48      vector<int> color;                  //颜色
49      vector<NodeStatus> status;          //顶点状态
50      vector<list<Expression*>> useList;  //使用该变量的指令表
51      vector<list<Expression*>> defList;  //定义该变量的指令表
52      vector<list<Expression*>> moveList; //与该变量有关的传送指令表
53
54
55      void ColorAlgorithmMain();          //图染色主函数
56      void AddEdge(int x,int y);          //建图添加边
57      void livelyAnalyz();                //活性分析
58      void InitColorAlgorithm();          //初始化
59      void DecrementDegree(int m);        //顶点度数减1
60      bool MoveRelated(int n);            //n是否是传送相关的
61      list<Expression*>& NodeMoves(int n);//与n相关的传送指令(未冻结的)
62      list<int>& Adjacent(int n);         //n的邻点集
63      void Simplify();                    //简化
64      void Coalesce();                    //合并
65      int GetAlias(int x);                //获取别名(由合并产生)
66      void AddWorklist(int u);            //加入工作表
67      bool TestPrecoloredCombine(int u/*precolored*/,int v); //测试预着色节点相关传送指令是否可以合并
68      bool TestConservative(int u,int v); //测试传送指令是否可以保守合并
69      void Combine(int u,int v);          //合并
70      void EnableMoves(int m);            //将m相关的传送指令设为待合并的
71      void FreezeMoves(int u);            //将u相关的传送指令冻结
72      void FreezeAction();                //冻结
73      void SelectSpill();                 //选择高度数节点溢出
74      void AssignColors();                //分配颜色
75      void RewriteProgram();              //重写程序,对于有真实溢出的情况
76      void InsertExprForWrite(Expression* e,int v);   //插入栈内存写入
77      void InsertExprForRead(Expression* e,int v);    //插入栈内存读取
78      int GenTempVariable();              //获取一个新临时变量的id
79
80      //程序流处理及优化
81      void InitializeVectorSpace();       //初始化
82      void InitFunEnv();                  //函数入口形参处理
83      int insert();                       //添加一个int变量到栈中
84      void frameFree();                   //释放栈的最后一个空间
85      void SaveReg();                     //call语句出保存调用者保存的寄存器
86      void OptimizeFlow();                //优化程序流(常数传播)
87      void OptimizeDead();                //死代码消除
88      void OptimizeLoadStore();           //优化
89      void genFlow();                     //生成程序流
90
91      //生成代码
92      string opstring(int op);            //获得op对应的运算符
93      string opinstruct(int op);          //获得op对应的RiscV指令
94      void OutputArithRIMul(int reg1,int reg2,unsigned imm);  //处理ArithRI指令的RiscV输出(将乘以2的幂改为左移)
95      void GenCode();                     //生成tigger代码
96      void GenRiscv64();                  //生成RiscV64代码
97      void GenRiscv32();                  //生成Riscv32代码
98      void checkReturn();                 //检查函数退出前是否有返回指令
99  };
```

5.3 程序流生成

Parser 分析到函数体结尾 end func 时会生成一个 Empty 类型语句表示函数体的结束.

程序流分析过程对 Expression 对象的 nexts 属性和 prevs 属性赋值, 使其分别指向 e 的所有前驱和所有后继.

除 Goto,If 类,Return,Empty 语句外, 所有语句在 vector<Expression*> exprs 中的后继都是其程序流中的后继 (称为默认后继).

对于 Goto 语句, 其后继为 Goto 所指向标号对应的 Label 语句;

对于 If 类语句, 其后继为默认后继以及所指向标号对应的 Label 语句;

对于 Return 语句, 其后继为表示函数体结束的 Empty 语句.

5.4 return 语句检查

如果函数结尾处的 Empty 语句有不是 return 语句的前驱, 则说明程序控制流到了函数末尾, 会给出警告;

并会自动在函数结尾处添加一条 return 0;

5.5 活性分析

以语句为基本块进行自下而上的活性分析. 使用队列进行宽度优先搜索, 递推式为:

$$e.in = e.use \cup (e.out - e.def)$$

$$e.out = \cup_{x \in e.nexts}(x.in)$$

如果进行上述计算后 e.in 和 e.out 有所改变, 则将 e 的前驱加入队列.

迭代直到队列为空.

5.6 死代码消除

此处处理的死代码分为两种:

(1)$e.def \bigcap \left(\bigcup_{x \in e.nexts} x.in\right) == \Phi$ 即该语句定义的变量在这条语句的所有后继中都不是入口活跃的.

(2) 不可达语句, 即在活性分析时未处理的语句.

对于第 (1) 情行, 未定义变量的语句和控制流跳转类语句为例外排除.

5.7 常数传播

每个 Expression 语句对象都有一个 constant 属性, 他是一个 map 表, 从变量 id 映射到该处 (出口处) 的变量常数值. 此表只会储存此处为常数值的变量.

按顺序遍历所有语句, 对于每个语句, 计算一个 mp(map 表)

$$mp = \bigcap_{x \in e.prevs} x.constant$$

即 mp 为 e 的所有前驱的 constant 表中项的交集, 也即对于所有前驱语句都具有相同常数值的变量.

对于非跳转类或访存类语句, 如果 $x \in mp \forall x \in e.use$ 成立, 也即所有被 x 使用的变量在此处均为常数, 则可说明此语句定义的变量也为常数. 调用 calcarith 函数计算此算术运算. 并将该语句的类型改为 MoveRI(即将直接数赋值给变量的语句)

对于上面公式不成立的语句, 以及跳转类和访存类语句, 令

$$mp = mp - e.def$$

即此处被定义的变量标记为不是常数.

最后令 $e.constant = mp$ 记录此语句处的常量表

5.8 访存优化

死代码消除和常数传播的优化是在寄存器分配之前就完成的, 而访存优化主要是优化对局部变量的访存指令. 考虑这两种情况: (1) 当寄存器不够用时, 图染色算法将局部变量溢出至栈帧, 然后 RewriteProgram 过程会为每次对该局部变量的读取增加 FrameLoad 指令, 对修改增加 FrameStore 指令.(2) 连续的 call 语句时, 调用者保存的寄存器会被连续存取但却不会改变.

以上两种情况可能产生一些冗余的访存指令.

此处使用两种规则优化:

(1) 设本条语句 FrameLoad n x. 若当前寄存器 x 在上一句 FrameLoad n x 后未改变, 且上条语句和本条语句之间没有跳转类语句, 标号或者数组写入语句, 则当前 FrameLoad 语句可被优化.

(2) 设本条语句 FrameStore n x. 若当前寄存器 x 在上一句 FrameLoad n x 后未改变, 且上条语句和本条语句之间没有跳转类语句, 标号或者数组写入语句, 则当前 FrameStore 语句和之前的 FrameLoad 语句都可被优化.

举例:

(1)

```
//此处设a保存在栈帧中
c = a + b;
d = a + c;
e = c + d;
```

优化前的 tigger 代码:

```
load  0 a0
a1 = a0 + a1
load  0 a0
a1 = a0 + a1
a1 = a1 + a1
```

优化后的 tigger 代码

```
load  0 a0
a1 = a0 + a1
a1 = a0 + a1
a1 = a1 + a1
```

(2)

```
b = a + 2;
d = func(1);
d = func(2);
return b;
```

优化前的 tigger 代码:

```
a1 = a0 + 2
a0 = 1
store a1 0
call func
load 0 a1
a0 = 2
store a1 0
call func
load 0 a1
a0 = a1
return
```

优化后的 tigger 代码

```
a1 = a0 + 2
a0 = 1
store a1 0
call func
a0 = 2
call func
load 0 a1
a0 = a1
return
```

5.9 寄存器分配

使用图着色算法进行寄存器分配.

利用活性分析的结果构造冲突图, 兼用邻接表和邻接矩阵表示冲突图.

图染色有四种操作:

1. 简化对于度数小于 k 的传送无关的点, 将其从图中删除并放入栈中.

2. 合并对于一条传送指令, 对其关联的两个点在保守规则下合并.

3. 冻结冻结一条传送指令, 将他当作非传送指令.

4. 溢出将图中度数最大的点放入栈中, 并从图中删除.

操作按 1.2.3.4 的优先级进行, 直到图中不再有点.

将栈中元素依次弹出, 并为其分配颜色, 如果无颜色可用, 则将该顶点真实溢出.

若有真实溢出发生, 则重写程序, 为溢出的变量分配栈空间, 加入 load 语句和 store 语句来读取和写回, 并创建临时变量进行运算.

5.10 其他说明

Tigger 不再创建符号表, 因为其中所有变量名称的作用域都是全局, 所以直接由词法分析器为其分配一个 id(>27). 全局变量和栈数组不分配寄存器, 直接存到内存中.

5.11 预着色节点的处理

Tigger 有 28 个寄存器, 给其确定 id 为 0-27, 以作为预着色节点, 在活性分析中与其他变量处于同等地位. 在图着色时也可与与其相关的传送节点合并.

5.12 形参的处理

函数参数及返回值需要预着色, 函数体的开头会创建临时变量, 使用传送指令将预着色节点传送到临时变量作为形参, 以避免形参长期占用 a 开头寄存器.

5.13 调用函数

该部分 tigger.y 代码:

```
Expression:
| ...
| PARAM Symbol                                {AnalyzInstance.currentFunc().CallParam($2,1);}
| PARAM INTEGER                                 {AnalyzInstance.currentFunc().CallParam($2,0);}
| Symbol '=' CALL FUNCTION                    {AnalyzInstance.currentFunc().CallFunc($1,$4);}
| ...
```

该部分 analyz.cpp 代码:

```
void Func::CallParam(int v,int t)
{
    if(t == 1)
    {
    new Expression(MoveRR,{(int)(a0)+paramToCallWithCount},{v},{});
    }
    else
    {
    new Expression(MoveRI,{(int)(a0)+paramToCallWithCount},{},{v});
    }
    paramToCallWithCount++;
}
void Func::CallFunc(int v,string f)
{
    vector<int> paramvec;
    paramvec.push_back(int(a0));
    for(int i = 1; i<paramToCallWithCount;i ++)
    {
        paramvec.push_back((int)(a0) + i);
    }
    new Expression(Call,{(int)(a0)},paramvec,{},f);
    new Expression(MoveRR,{v},{(int)(a0)},{});
    paramToCallWithCount = 0;
}
```

Func 类设置了一个计数器计数当前传入参数的个数. 计数器初始化为 0, 每扫描并处理一个 Param 语句, 将计数器值加一. 对于 ‘Param x‘语句, 将其处理为 ‘a0 = x‘(对应 Expression 的类型为 MoveRR), 其中 0 可能为 0-7, 是传参计数器的当前值. 由 Eeyore 的构成, 在 Param 语句和对应的 Call 语句之间不会有其他会改变参数寄存器值的语句, 而 Call 语句的 right 属性会被设置成 a0,a1,..,at, 其中 t 为参数的个数-1; 因此活性分析的结果会阻止 Param 语句和 Call 语句之间的其他无关变量被分配为 a0-at 寄存器.

合并预着色点相关传送节点的例子

MiniC 代码

```
int v0;
v0 = getint();
int v1;
v1 = func(v0);
int v2;
v2 = putint(v1+1);
return v2;
```

tigger 代码

```
call getint
call func
a0 = a0 + 1
call putint
return
```

riscv64 代码

```
main:
        add     sp,sp,-16
        sd      ra,8(sp)
        call    getint
        call    func
        addiw   a0,a0,1
        call    putint
        ld      ra,8(sp)
        addi    sp,sp,16
        jr      ra
        .size   main, .-main
```

变量大量合并, 只用了 a0, 没有一条 move 指令.