编译实习分享

罗昊

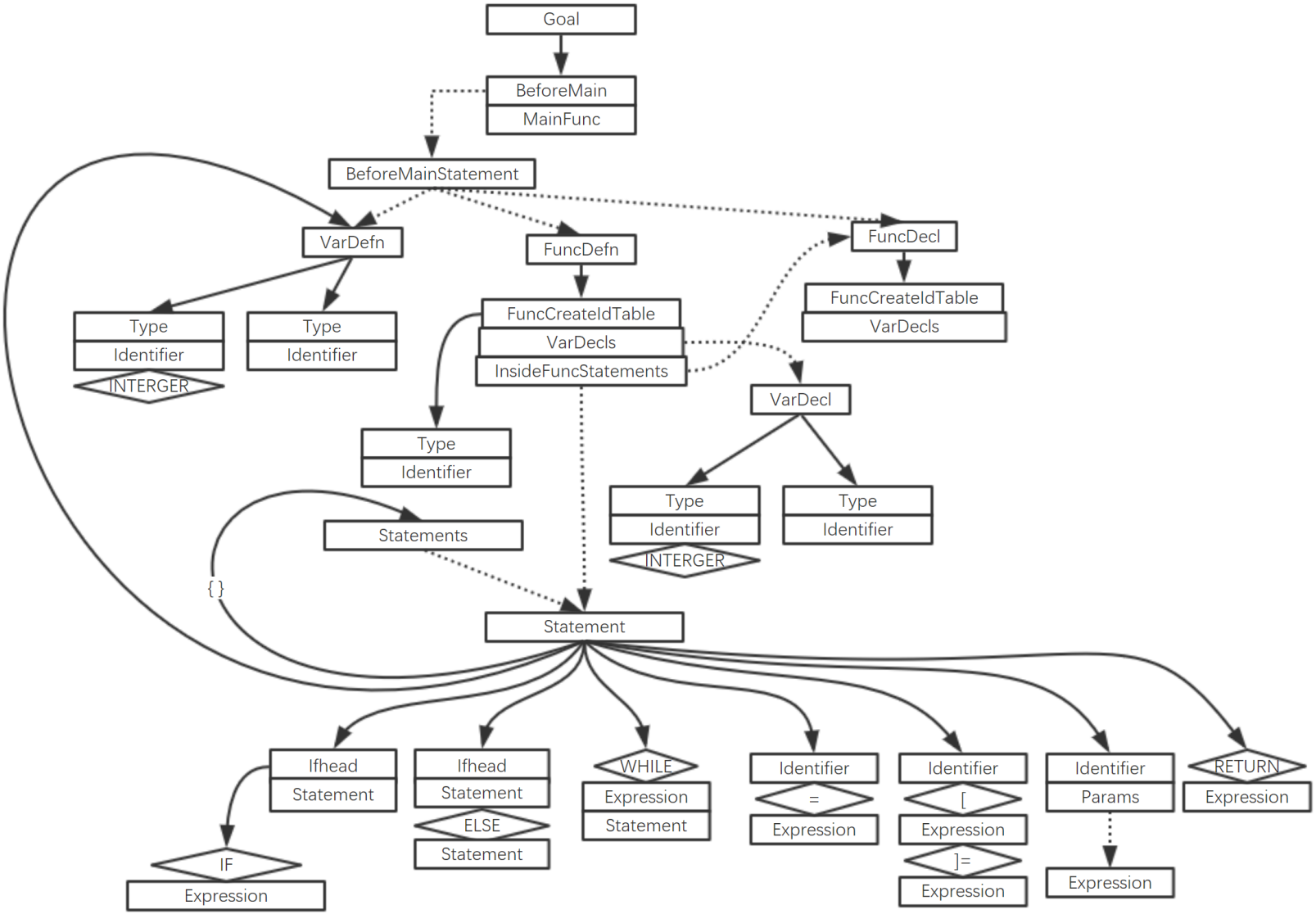
https://github.com/vangohao/hcc

2019 年 1 月 2 日

Eeyore 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.

Eeyore 语法结构


Goal
BeforeMain
MainFunc
BeforeMainStatement
VarDefn
FuncDefn
FuncDecl
Type
Identifier
INTERGER
Type
Identifier
FuncCreateIdTable
VarDecls
InsideFuncStatements
FuncCreateIdTable
VarDecls
VarDecl
Type
Identifier
Type
Identifier
INTERGER
Statements
{ }
Statement
Ifhead
Statement
IF
Expression
Ifhead
Statement
ELSE
Statement
WHILE
Expression
Statement
Identifier
=
Expression
Identifier
[
Expression
]=
Expression
Identifier
Params
Expression
RETURN
Expression

Eeyore 函数声明, 定义及参数检查

1. Declear 方法配合 decleared 属性, 在 VarDefn 中, 调用 Declear 方法将 decleared 属性设为 true. 在 Expression 中遇到相应符号时, 需要检查符号的 decleared 属性, 必须是 true, 否则会报错. 对于函数,Declear 方法会存入参数表.
2. Define 方法, 适用于函数, 作用是声明函数已定义, 若此函数已被声明, 则还需检查此处传入的参数表是否与声明时使用的参数表相匹配, 若不匹配则会报错. 但函数声明和定义都不是调用的必要条件, 没有声明的函数也可以直接调用.
3. checkParams 方法, 检查传入的参数表是否与 paramList 相匹配. 参数表的类型是 Node* , 也就是 VarDecl 对应的语法树节点. 检查的方法是遍历链表, 检查节点的 sym 属性的 type 属性是否对应或可以隐式转换, 不匹配的给出错误提示.

Eeyore 逻辑表达式和算术表达式转换

本编译器将比较表达式以及或与非表达式视为逻辑表达式，五则运算表达式视为算术表达式，支持逻辑表达式和算术表达式的自动转化.

1. 逻辑表达式转算术表达式：生成一个临时变量 x 表示结果，将逻辑表达式的 truelist 指向运行 x=1，然后跳过后面一句 x=0；将逻辑表达式的 falselist 指向运行 x=0.
2. 算术表达式转逻辑表达式：设算数表达式对应的符号为 x，则将该语句翻译为和逻辑比较表达式 x==0 相同的翻译结果. 即为其生成 if-goto-goto 语句，并将 truelist 初始化为第一个 goto 对应的序号，将 falselist 初始化为第二个 goto 对应的序号.

Tigger 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.

Tigger 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到的语句.
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.

Tigger 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

Tigger 访存优化

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

优化前的 tigger 代码:

```
1  load  0 a0
2  a1 = a0 + a1
3  load  0 a0
4  a1 = a0 + a1
5  a1 = a1 + a1
```

优化后的 tigger 代码

```
1  load  0 a0
2  a1 = a0 + a1
3  a1 = a0 + a1
4  a1 = a1 + a1
```

Tigger 访存优化

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

优化前的 tigger 代码:

```
1  a1 = a0 + 2
2  a0 = 1
3  store a1 0
4  call func
5  load 0 a1
6  a0 = 2
7  store a1 0
8  call func
9  load 0 a1
10 a0 = a1
11 return
```

优化后的 tigger 代码

```
1  a1 = a0 + 2
2  a0 = 1
3  store a1 0
4  call func
5  a0 = 2
6  call func
7  load 0 a1
8  a0 = a1
9  return
```

Tigger 寄存器分配

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.

合并条件:

1. 对于关联两个普通节点的传送指令, 合并这两个节点的条件是:
   如果节点 a 和 b 合并产生的节点 ab 的高度数 ($\geq K$) 的邻节点的个数小于 K, 则认为 a 和 b 的合并是保守的.
2. 对于一个普通节点 a 和一个预着色节点 b 的合并, 合并条件是:
   对于 a 的每一个邻接点 t, 或者 t 与 b 已有冲突, 或者 t 是低度数 ($< K$) 节点.