FUTURE GADGET 2015

FG2015
----ACM compiler 2015 project

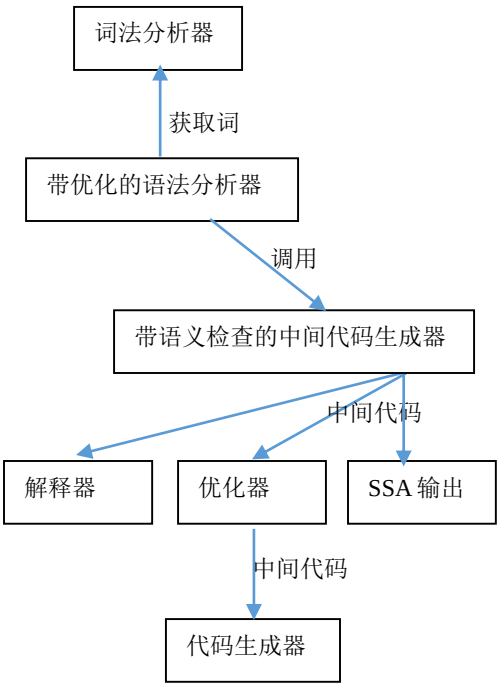
高宇
pmkfb@aliyun.com

摘要
这篇文档中, 我介绍 FG2015, 我的 ACM compiler 2015 project. 同时也包括了我的实现方法, 其中有得到满分的方法和最速得到基本分的方法.

关键字
C 语言, 编译器, 自编译

1. 简介
FG2015(未来道具 2015), 是一个能够自编译的 C 语言编译器, 支持 C 语言的一个子集. 显然它是用 C 语言写的. 它实现了多种窥孔优化和 linear scan 寄存器分配以及二次寄存器分配. 它拥有解释器功能, 能够模拟支持的语言中无键盘输入的所有程序. 它还能打印 single static form, 但是没有利用 single static form 做任何优化.

2. 代码框架
下图展示了 FG2015 的代码框架, 即代码中各个功能模块的交互方式. 这个代码框架能被精简从而用于编写一个编程复杂度最低(*)的 ACM compiler2015 project.



3. 词法分析器
3.1 nextToken()
词法分析器唯一的对外借口是 nextToken()函数, 它返回程序的下一个 token. 下面介绍每种 token 的处理方法.

3.2 空白和注释
nextToken()函数开头必须跳过当前所有的空白和注释. 空白是指空格和制表符以及换行符, 注释则是单行注释和多行注释.

3.3 关键字和标识符
当跳过空白和注释后, 遇到的第一个字符是字母('_', '$', 'A'..'Z', 'a'..'z')时, 获取后面连续的字母和数字作为一个字符串. 如果在关键字表里匹配不到, 就是标识符. 关键字可以以字符串或编号的方式存储在 token 中, 标识符则必须包含字符串.

3.4 整数常量
当跳过空白和注释后, 遇到的第一个字符是数字, 获取后面连续的字母和数字作为一个字符串, 并解析为 8 进制/16 进制/10 进制字符串常量, 将其值存储在 token 中.

3.5 字符常量和字符串常量
当跳过空白和注释后, 遇到的第一个字符是单引号或双引号时, 就获取后面连续的(转义)字符, 直到同样的引号出现.

3.6 衔接符
衔接符('\')必须在词法分析前处理, 衔接符能吃掉它后面紧随的换行. 但是换行必须考虑操作系统的区别.

4. 语法分析器
4.1 概述
由于支持的语言子集使用 CFG 定义的, 所以语法分析器采用了最简单暴力的递归下降+看 2 个 token.

4.2 支持的语言
Top level

```
program: (declaration | function-definition)+
declaration: type-specifier init-declarators? ';'
function-definition: type-specifier plain-declarator '(' parameters? ')' compound-statement
parameters: plain-declaration (',' plain-declaration)*
declarators: declarator (',' declarator)*
init-declarators: init-declarator (',' init-declarator)*
init-declarator: declarator ('=' initializer)?
initializer: assignment-expression
           | '{' initializer (',' initializer)* '}'
 type-specifier: 'void' | 'char' | 'int'
```

```
        | struct-or-union identifier? '{' (type-specifier declarators
';')+ '}'
        | struct-or-union identifier
struct-or-union: 'struct' | 'union'
plain-declaration: type-specifier declarator
declarator: plain-declarator ('[' constant-expression ']')*
plain-declarator: '*'* identifier
```

Statements

```
 statement: expression-statement
        | compound-statement
        | selection-statement
        | iteration-statement
        | jump-statement
expression-statement: expression? ';'
compound-statement: '{' (declaration + statement)* '}'
selection-statement: 'if' '(' expression ')' statement ('else'
statement)?
iteration-statement: 'while' '(' expression ')' statement
                | 'for' '(' expression? ';' expression? ';' expression? ')'
statement
jump-statement: 'continue' ';'
        | 'break' ';'
        | 'return' expression? ';'
```

Expressions

```
expression: assignment-expression (',' assignment-expression)*
assignment-expression: logical-or-expression
                | unary-expression assignment-operator assignment-
expression
assignment-operator: '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' |
'&=' | '^=' | '|='
constant-expression: logical-or-expression
logical-or-expression: logical-and-expression ('||' logical-and-
expression)*
logical-and-expression: inclusive-or-expression ('&&' inclusive-
or-expression)*
inclusive-or-expression: exclusive-or-expression ('|' exclusive-or-
expression)*
exclusive-or-expression: and-expression ('^' and-expression)*
and-expression: equality-expression ('&' equality-expression)*
equality-expression: relational-expression (equality-operator
relational-expression)*
equality-operator: '==' | '!='
relational-expression: shift-expression (relational-operator shift-
expression)*
relational-operator: '<' | '>' | '<=' | '>='
shift-expression: additive-expression (shift-operator additive-
expression)*
shift-operator: '<<' | '>>'
additive-expression: multiplicative-expression (additive-operator
multiplicative-expression)*
additive-operator: '+' | '-'
multiplicative-expression: cast-expression (multiplicative-operator
cast-expression)*
multiplicative-operator: '*' | '/' | '%'
cast-expression: unary-expression
            | '(' type-name ')' cast-expression
type-name: type-specifier '*'*
unary-expression: postfix-expression
            | '++' unary-expression
            | '--' unary-expression
            | unary-operator cast-expression
            | 'sizeof' unary-expression
            | 'sizeof' '(' type-name ')'
unary-operator: '&' | '*' | '+' | '-' | '~' | '!'
postfix-expression: primary-expression postfix*
postfix: '[' expression ']'
       | '(' arguments? ')'
       | '.' identifier
       | '->' identifier
       | '++'
       | '--'
arguments: assignment-expression (',' assignment-expression)*
primary-expression: identifier
            | constant
            | string
            | '(' expression ')'
constant: integer-constant
       | character-constant
```

4.3 Top level 的 parse 方法

由于采用了递归下降法, 对上面的 CFG 的每个非终止状态都写一个函数, 根据下一个 token 确定要采用的转移, 再依次匹配转移的状态即可. 其中变量和函数声明会出现无法区分的情况, 故合在一起写一个函数.

每一层大括号都对应一个符号表, 符号表中有 struct/union 表, 函数表和变量表. 其中函数参数对应函数内符号表, 函数的返回值则对应函数外的符号表.

struct/union 应记录每个成员的类型, 名字和每个成员在内存中相对结构体头的位置.

变量应记录名字和类型.

函数应记录返回值的类型, 名字和各个参数的类型.

类型包含基本类型(int, char, struct/union), 指针阶数和数组各维度. 由于支持的语言限制, 只存在指针数组而没有指向数组的指针.

4.4 Statements 的 parse 方法

大致与 Top level 的 parse 方法一样, 下面讲一下几种控制语句应该如何生成代码.

if 语句: 新建两个 label, 一个代表条件不满足时执行的代码的开始位置, 另一个代表 if 语句结束的位置. 首先 parse 条件表达式, 若不满足, 跳到不满足时执行的代码的开始位置. 然后 parse 满足时执行的代码, 最后跳到 if 语句 结束的位置. 打印不满足时执行的代码的开始位置的 label. parse 不满足时执行的代码. 打印 if 语句结束的 label.

for 语句: 新建三个 label, 一个代表判断循环条件之前, 一个代表修改循环变量之前, 一个代表 for 语句结束之后. 先 parse 初始化表达式. 打印判断循环条件之前的 label. parse 循环条件表达式, 若不满足跳至循环结束位置. parse 循环变量改变表达式, 但是先不打印. parse 循环体. 打印循环变量改变位置的 label. 打印循环变量改变表达式. 打印 for 语句结束 label.

while 语句: 新建两个 label, 代表判断循环条件之前和 while 循环结束之后. 打印判断循环条件之前的 label. parse 循环条件表

达式, 若不满足跳至循环结束之后. parse 循环体. 跳至判断循环条件之前. 打印循环结束 label.

break 语句: 因为 break 语句的缘故, parse statements 时, 要时刻记住当前 break 要跳到哪个 label. 若遇到 for 或者 while 语句, 则 break 跳到的 label 置为循环结束的 label.

continue 语句: 因为 continue 语句的缘故, parse statements 时, 要时刻记住当前 continue 要跳到哪个 label. 若遇到 for 或者 while 语句, 则 continue 跳到的 label 位置为判断循环条件之前的 label.

return 语句: 直接保留在中间代码中.

4.5 expression 的 parse 方法

parse expression 时, callee 要返回 parse 的 expression 的运算结果, 这样 caller 才能根据它所有 callee 返回的结果调用中间代码生成器, 返回它的结果.

其中 sizeof 要返回常量.

其中 unary expression 处要多看几个 token 才能判断需要的转移.

逻辑与和逻辑或表达式要有短路求值, 方法是新建一个标签代表短路以后将结果置为 0(或者 1), 求出参数后如果是 0, 就跳到前面建立的标签处. 到了最后一个参数, 就直接将结果赋值. 就像下面这样, a || b || c 生成的代码为:

if a goto set_result

if b goto set_result

result = c

goto end

set_result:

result = 1

end:

4.6 parser 所做的优化

举个例子, 一个 a = b + c, parser 应该会生成 temp = b + c, a = temp 这样的代码. 于是我们记住生成的最后一个赋值代码, 把 temp 直接替换为 a. 这样能减少许多赋值表达式的指令数.

5. 中间代码生成器

5.1 中间代码表示设计

一个中间代码包含一个指令类型码 type 和三个操作数 a, b, c, 外加一个整数 n 存储指令中可能出现的参数和一个函数 func 来表示函数调用.

一个操作数包含常数信息, 类型信息和操作数编号.

中间代码共有下面几种:

LABEL n:

GOTO goto n

IF_GOTO if a goto n

IF_FALSE_GOTO if_false a goto n

ASSIGN a = b

ASSIGN_LOGICAL_NOT a = !b

ASSIGN_NOT a = ~b

ASSIGN_NEGATE a = -b

ASSIGN_INCLUSIVE_OR a = b |c

ASSIGN_EXCLUSIVE_OR a = b ^ c

ASSIGN_AND a = b & c

ASSIGN_SHR a = b >> c

ASSIGN_SHL a = b << c

ASSIGN_ADD a = b + c

ASSIGN_SUB a = b - c

ASSIGN_MOD a = b % c

ASSIGN_DIV a = b / c

ASSIGN_MUL a = b * c

ASSIGN_NOT_EQUAL_TO a = b != c

ASSIGN_EQUAL_TO a = b == c

ASSIGN_GREATER_THAN_OR_EQUAL_TO a = b >= c

ASSIGN_LESS_THAN_OR_EQUAL_TO a = b <= c

ASSIGN_GREATER_THAN a = b > c

ASSIGN_LESS_THAN a = b < c

ASSIGN_ADDRESS_OF a = &b

ASSIGN_DATA a = 给变量 n 预分配的地址

ASSIGN_STR a = 给字符串 n 预分配的地址

ARGU 将 a 压入栈中

CALL 调用函数 func 并将返回值存入 a

GET_ARGU 从栈顶获得函数的参数

VOID_RETURN 返回值为空的函数的返回

RETURN 返回 a

LD 从内存 b + n 处取一个 int 赋给 a

LD_CHAR 从内存 b + n 处取一个 char 赋给 a

ST 将 b 作为 int 存储在内存 a + n 处

ST_CHAR 将 b 作为 char 存储在内存 a + n 处

MALLOC 令系统分配大小为 b 的内存, 将头指针存在 a 中

GETCHAR 从键盘读一个字符赋给 a

PUTCHAR 在屏幕上以字符(ascii)的形式打印 a

PUTINT 在屏幕上以 int 的形式打印 a

EXIT 退出整个程序

PHI SSA 使用的 phi 函数

NOP no operation.

5.2 中间代码中的数组和 struct/union

中间代码中的数组和 struct/union 都是用头指针表示的. 也就是说对数组/struct/union 中的内容修改, 都对应内存操作.

5.3 中间代码中的类型检查

加这个条目只是为了说明中间代码生成时要类型检查. 关于该检查什么没有必要详细说明, 请参阅助教的要求和 gcc 的检查.

6. 解释器

6.1 解释器中的变量存储

变量按照编号的二进制表示存在一个 trie 里面. 无论原来是什么类型, 全部按照 int 存储.(这相当于我们有一台对内存中两个数操作的机器)

6.2 解释器中的函数调用

为了递归问题, 函数调用时要备份函数中用到的变量.(这对应 mips 代码中的备份寄存器) 函数调用的方式其实是调用一个通用 run()函数, 把函数名传进去.(这对应备份 mips 中的 ra.)

6.3 解释器中的 goto 和 label

需要预处理将所有的 label 对应到一个中间代码指针上去, 跳转时直接置当前指令指针为跳到的 label 对应的指针.(这就对应着 mips 中的 pc)

7. 优化器

7.1 控制流分析

所谓控制流, 是程序运行时执行指令的顺序. 如果没有 if_goto 这种语句, 这个控制流就是确定的, 有了这类语句, 这个控制流就有了分叉. 所谓控制流分析, 就是分析程序运行时指令执行的所有可能顺序. 许多优化是建立在控制流分析之上的.

为了方便一些优化, 先要将代码分为 basic block. 所谓 basic block, 就是程序执行时, 如果执行到某个 basic block 之内的语句, 则一定是从这个 basic block 中语句的第一条按顺序逐个执行这个 basic block 中的每条指令各一次. 即各类 goto 的出现位置和目标位置都要成为 basic block 的分割点. (如果做函数内控制流分析的话, 函数调用和返回就不算 goto 类语句.)

分好 basic block 后, 就根据每个 basic block 最后的 goto 语句建立控制流图. 控制流图中的一条有向边连接两个 basic block, 表示一个 basic block 执行完可能到另一个 block 的开头处继续执行.

7.2 活性分析

一个变量在某个时刻, 若其值可能在以后被用到, 则此时必须将其值保留在某个地方, 称该变量此时为活的. 若其值不会在以后被用到, 则此时该变量便不需要空间保存, 称该变量此时为死的. 活性分析为寄存器分配提供资料: 两个变量如果活的区间无交, 则他们可以共用一个寄存器.

对于一条语句 a=b+c, 无论该条语句之后的瞬间 b, c 的存活情况, 该条语句之前的瞬间 b, c 一定是活的. 如果 a 不等于 b 也不等于 c, 则该条语句之前的瞬间 a 是死的. 根据这条规则, 加上所有语句运行结束后所有变量都是死的, 就能推断出顺序控制流程序中每个时刻(即每两条指令之间)每个变量的存活情况了. 对于分支控制流, 只要一个分支中下一条指令执行之前某个变量是活的, 分支之前最后一条指令执行之后这个变量就是活的. 有了上面的规则, 只要迭代至不动点, 即可求出程序中任意时刻任意变量的存活情况.

7.3 linear scan 寄存器分配

linear scan 假设每个变量的存活区间是连续的, 这样就能采用以下的贪心策略分配较优的寄存器: 按存活区间的左端点从小到大考虑每个变量, 先把所有右断点小于当前变量左端点的变量占用的寄存器释放. 此时如果有空余寄存器, 则为当前变量分配这个寄存器, 并将该寄存器占用. 若此时每个寄存器都被占用, 则在所有占用寄存器的变量和当前变量中取一个右端点最大的, 使它不占用寄存器(如果右端点最大的不是当前变量, 就让当前变量占用它的寄存器, 否则当前变量不对寄存器占用情况产生任何影响.) 这样做完所有变量后, 一些变量有寄存器, 另外一些没有. 没有的变量在生成代码时就只能每次从内存中取.

实际上每个变量的存活区间是不连续的, 那令它的存活区间的左端点为他存活的最小时刻, 右端点为最大时刻. 这就近似地表示了变量的存活区间.

所谓时刻, 并不代表真正运行时的先后顺序. 所以每两条语句之间的"时刻"可以重新排序, 使得存活区间的近似情况更好. 但是 FG2015 没有做这个优化, 直接使用了中间代码生成的顺序作为时刻的顺序.

注意到全局变量因为会被其它函数修改, 不分配寄存器. 被取过地址的变量, 因为可能在写内存的时候被修改, 也不分配寄存器.

7.4 二次寄存器分配

二次寄存器分配是我为了提高寄存器利用率而做的优化. 首先, 对于每个函数都没有使用的寄存器, 可以归全局变量使用. 然后, 每个没有分配到寄存器的变量如果实际的存活区间与某个寄存器的占用区间不交, 就将它分配到那个寄存器.

注意被取过地址的变量还是不能分配寄存器.

8. single static form

8.1 简介

single static form(静态单赋值形式), 是指这样的中间代码, 其中每个变量只被赋值一次. 这种形式允许多种的优化.

但是这样就无法实现控制流分支了, 以为很可能一个变量在不同的分支中被赋以不同的值. 这时我们在不同的分支中使用不同的变量, 并使用一个 phi 函数, 它在两个参数中神奇地帮我们选择需要的再赋给一个新的临时变量.

8.2 简单做法

有许多优化的做法, 但是这里用的是一个简单的做法. 先做好控制流分析. 然后将对同一个变量的每个赋值做不同的标记, 如 a(1) = b + c, a(2) = d / e. 然后对于每个变量, 将标记在控制流中传递, 直到遇到另外一次赋值. 最后, 对于每个 basic block 的开始, 若这个 basic block 中用到了某个变量, 则对该变量添加

足够的 phi 语句(若此时有 n 个不同的标记, 则需要 n-1 个)使该变量此时无歧义.

9. 代码生成

9.1 函数内代码生成

假如没有寄存器分配, 则每个变量使用时都必须到内存中取. 于是我们为每个变量在内存中分配位置. 全局变量分配在.data 里, 局部变量则存储在函数的栈帧中. 每次执行一条中间代码指令, 先把参数 load 到任意 2 个寄存器中, 再生成 mips 代码对其操作, 最后把结果 store 回需要的位置.

有了寄存器分配, 当被分配了寄存器的变量被使用时, 就不需要 load, 同样产生被分配了寄存器的变量时, 也不需要 store. FG2015 规定未被分配寄存器的变量只能使用$v0 和$a0 来临时 load, 只能使用$v0 来生成临时运算结果.

函数内的变量存储在函数对应的栈帧中, FG2015 规定用$

sp 指向当前函数对应的栈帧, 所有变量存储在内存中$sp 加常数的位置.

9.2 函数间代码生成

调用函数时, 使用 jal 指令, 它能在$ra 中保存被调用的函数执行结束后控制流返回的位置(即当前的下一条指令.) 被调用的函数执行结束后, 用 jr $ra 即可返回. FG2015 规定函数将返回值存储在$v0 中. FG2015 规定函数的参数一次压入栈中, 调用函数后, 这些参数对应的栈就归被调用者使用. FG2015 规定使用$fp 作为活动栈指针, 指向最后一个被压入栈中的元素. 即函数的参数在函数的栈帧底部. 在函数开始时存活的参数若被分配了寄存器, 则要在函数开始前 load 到指定寄存器.

一个函数不应该对调用它的函数产生任何影响. 于是一个函数必须备份它用到的寄存器, 即为它里面任何变量分配的寄存器. 除此之外, 函数还应该备份$ra 和$sp, 分别为上一层函数的返回地址和上一层函数的固定栈指针.

函数结束后应该复原备份的所有变量, 同时将$fp 处自己的参数全部弹出.

10. 致谢

感谢孙锴, 蒋舜宁, 廖超, 刘爽, 许文五位助教所做的工作和对我的帮助. 感谢一起写 compiler 2015 project 的同学们对我的启发和帮助.

11. 参考文献和资料

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. China Machine Press, 2009.

[2] Massimiliano Poletto, and Vivek Sarkar. *Linear Scan Register Allocation*. http://www.cs.ucla.edu/~palsberg/course/cs132/linearscan.pdf.

[3] Kai Sun, Shunning Jiang, Chao Liao, Shuang Liu, and Wen Xu. *Compiler 2015*. http://acm.sjtu.edu.cn/wiki/Compiler_2015.