C1 编译器的实现

PB09210183 何春晖

2012.06.17

目录

1 总览

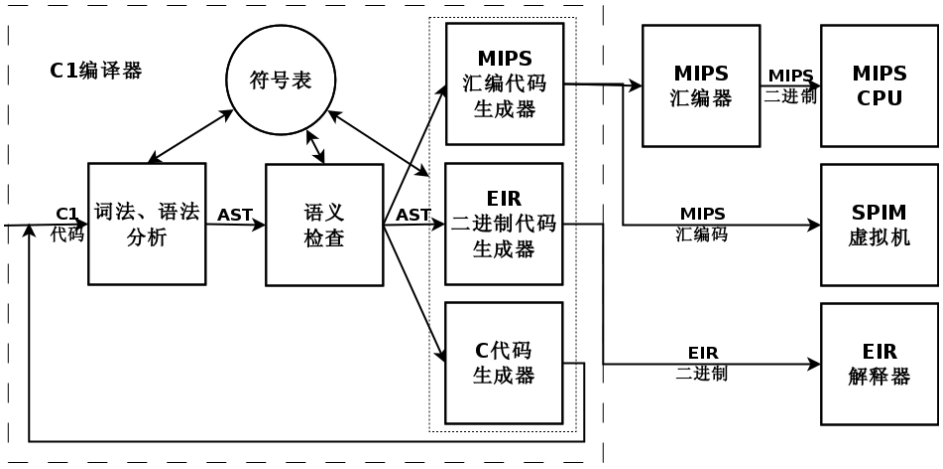


图 1: C1 语言编译器及流程

C1 语言是一个类 C 的语言。语言的特征为：

- 包含 int、float 和 bool 简单类型以及以这些类型为基本类型的多维数组类型。
- 一个 C1 程序包含多个函数、全局变量声明和常量声明，其中必须有一个 void main(void) 主函数。
- 函数可以带参数，也可以不带参数，参数的类型是简单类型。
- 函数的返回类型可以是 void，或者是某简单类型。
- 函数体中可以有常量定义、变量声明和函数声明，包含表达式语句、条件语句、循环语句、函数调用语句、复合语句和空语句。

本文实现的 C1 编译器，其编译流程由词法语法分析、语义检查和代码生成三个阶段组成。其主要的特点是：

- **多目标**：对 C1 源代码，可以生成 MIPS 汇编码、EIR 二进制码和 C 代码；
- **强大的类型系统**：可以识别 C 语言语法的类型定义，输出其类型表达式；
- **实现绝大部分 C1 语言特征**；
- **带有扩展语法**：如 continue、for 等；
- **较详细的错误报告**；

下面根据编译器的阶段，逐一介绍其实现细节。

2 词法、语法分析

2.1 分析方案

本阶段的分析是把字符串流转换为抽象语法树。

词法、语法分析分别使用 Flex 和 Bison 构造。

分析时，只对语句建立树结构。对于符号的定义（变量定义、函数定义等），并不对其语法成分建树，而是顺着分析流程建立符号表，并把符号放在符号表中。

这样，就可以 **避免语法树中出现大量的字符串**，使得树的结构、结点的类型得到了简化。缺点是 **造成复杂的类型分析比较困难**，将类型系统的设计大大复杂化了。

翻译完成后，得到的总入口为全局符号表，从此符号表开始检索，可以得到程序的所有信息。

2.2 词法

与 C 的词法类似，其主要区别为：

- read 和 write 是保留字，用于在 C1 中进行输入输出；
- bool、true 和 false 是保留字，用于实现布尔类型；

其余还有一些区别，如 sizeof 不是单词等，但并不重要。

2.3 语法

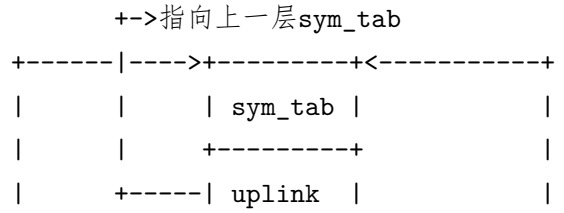
本实现的语法与 C1 的语法基本相同，其主要区别是：

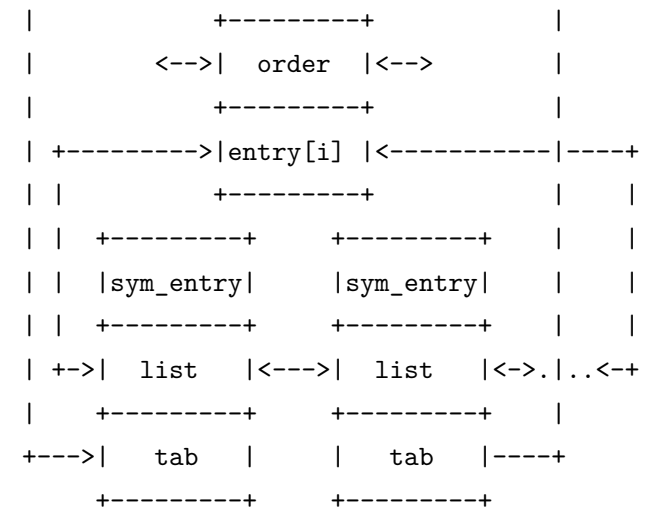
- 没有逗号表达式；
- 包含 for 语句；
- 函数的参数可以是数组类型（值传递语义）；
- 变量初始化语法只能有一层括号，且不能有多余逗号。
- 下列不是运算符

  `++`、`--`、`+=`、`-=`

2.4 符号表

符号表实现在 src/sym_tab.c 中。采用多层结构，图示如下：





上图中，sym_tab 是符号表，sym_entry 是表项。

表项串接在符号表中，有 list 和 order 两个线索。表项的 list 链条是哈希链，order 链条为顺序链。

查找符号时，先在本层的符号表查找。若找不到，则顺着 uplink 向上一层再查，直到找到或到达顶层。

符号表记录符号的名字和类型。根据不同的类型有不同的记录，如函数有函数局部符号表地址、函数语句 AST 指针、函数地址、函数类型等信息。

2.5 类型系统

2.5.1 表示

类型系统实现在 src/type.c 中，其基本结构类似符号表，也是一个哈希链条将所有类型串起来。

每个类型的定义如下：

```
struct type {
      struct list_head list;
      enum {
            TYPE_VOID = 0,
            TYPE_INT,
            TYPE_FLOAT,
            TYPE_BOOL,
            TYPE_ARRAY,
            TYPE_FUNC,
            TYPE_LABEL,
            TYPE_TYPE,
      } type;
      int n;
      int is_const;
```

```
        union {
                struct sym_entry *e;
                struct type *t1;
        };
        struct type *t2;
};
```

有上述定义可见，这个类型的定义是树状的，因而可以表达非常复杂的结构，如函数数组，数组函数等。

2.5.2 名字

上面类型都是匿名的，当需要给类型取名（包括内置类型和用户自定义类型）时，可以构造一个 TYPE_TPYE 类型的类型。其中上述结构体的 e 指向符号表，给出类型名字，t2 指向真实的类型。

在编译器初始化时，默认给内置类型命名：

```
symtab_enter_t(symtab, "int", get_type(TYPE_INT, 0, 0, NULL, NULL));
symtab_enter_t(symtab, "float", get_type(TYPE_FLOAT, 0, 0, NULL, NULL));
symtab_enter_t(symtab, "bool", get_type(TYPE_BOOL, 0, 0, NULL, NULL));
symtab_enter_t(symtab, "void", get_type(TYPE_VOID, 0, 0, NULL, NULL));
```

当用户用 typedef 定义新类型时，可以类似上述方法，在符号表中记录相应类型。

2.5.3 等价

类型等价可以按结构和按名字。

从类型的表示可见，当类型需要按名字等价时，只要比类型指针就可以了。若指针不等，则不是同一类型（匿名的类型总是不等的）：

```
static inline int type_is_equal_byname(struct type *t1, struct type *t2)
{
        return t1 == t2;
}
```

当按结构等价时，则需要递归地比较两个类型树的所有属性：

```
static inline int type_is_equal_bystru(struct type *ty1, struct type *ty2)
{
.....
        if(ty1->type == TYPE_FUNC)
                return type_is_equal_bystru(ty1->t1, ty2->t1) &&
                        type_is_equal_bystru(ty1->t2, ty2->t2);
.....
}
```

2.5.4 解析

C 语言中的类型定义 **并非是书写类型表达式，而是声明其用法**。这造成了这一部分实现的极端复杂。

如类型表达是为 int->array(10,int) 的类型用 C 语法写出为：

```
int type(int a)[10];
```

为了分析这种类型，在 rule/c1.y 中有两个函数来处理之。

2.6 AST

AST 实现在 include/ast_node.h 中。

由于语义的要求，树结点的分叉数是不一样的的，故采用链表 **将儿子和兄弟组成一个双向链表**（从 Linux 内核取出，而非 bison-example），增强通用性。

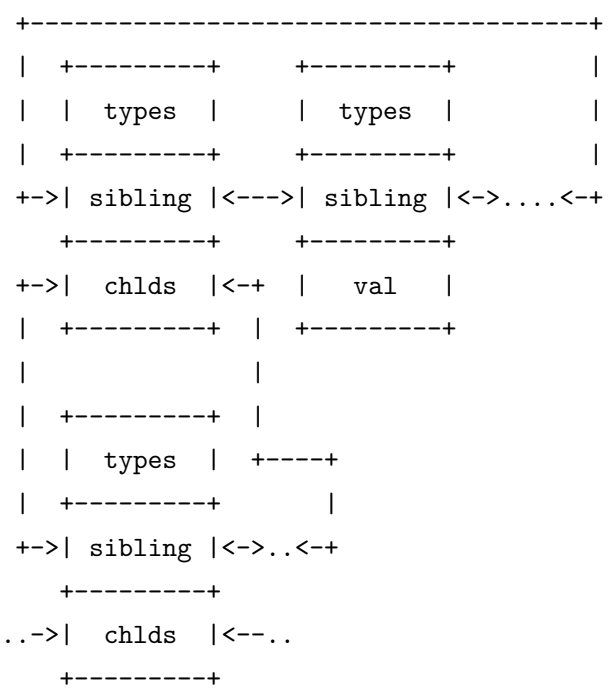
定义如下：

```
struct ast_node {
unsigned short type;
unsigned short id;
struct list_head sibling;
int first_line;
int first_column;
union {
void *pval;
int ival;
float fval;
struct list_head chlds;
};
};
```

各个域含义为：

- type：结点类型（exp、block 等，详见 node_type.h）
- id：结点子类型（'+'、'-' 等）
- sibling：兄弟组成的链表
- first_：位置追踪信息
- chlds：儿子组成的链表
- val：结点属性值

图示如下：



基本操作只有三种：ast_node_new 新建 ast_node_delete 删除 ast_node_add_chld 增加儿子

其余遍历兄弟和儿子的操作使用 list.h 中的 list_for_each_entry 实现。

3 语义检查

此遍较简单，主要要做的检查为：

1. 类型检查和提升
2. continue、break 在 while 或 for 中
3. 变量不能是 void
4. const 变量不能被赋值

4 EIR 代码生成器

EIR 指令模拟的是一种栈式机器，指令类型和意义可见 eir/interp_dbg.c。

此指令集的特点是：**已经将所有的策略定好**，因此指令生成并没有太多灵活的空间，只要对树进行一次遍历，就可以生成代码。

值得一提的是短路运算的翻译方案。如 and 的翻译如下：

```
geni(lit, 0, 0);
gen_exp(l);
cj1 = cx;
```

```
geni(jpc, 0, 0);
gen_exp(r);
cj2 = cx;
geni(jpc, 0, 0);
geno(opr, 0, notnot);
code[cj1].v.i = cx;
code[cj2].v.i = cx;
```

这个翻译方案的特点是：

- 若两个表达式有一个为假，最终栈顶留下数字 0
- 若第一个表达式为真，第二个表达式不求值
- 两个表达式均真时，执行 notnot 操作，将栈顶翻转为 1

因此这个方案是 and 操作的合法方案。这个方案 **用较少的指令达到了准确的翻译，且翻译只需要局部的信息**。缺点是条件较复杂时可能要连续经过多次跳转才能到达目标。

or 的翻译类似可得。

5 MIPS 代码生成器

5.1 寄存器分配

MIPS 是基于寄存器的机器，因此相对于栈式机器，需要进行寄存器分配。

为了简单起见，本生成器基于基本块来分配。

寄存器分配器为每个寄存器维护如下的结构：

```
struct reg_struct {
int dirty;
int loaded;
struct sym_entry *sym;
struct list_head list;
struct list_head avail_list;
};
```

由此可知，这里一个寄存器仅仅可以关联一个符号 sym。符号表中同时有一项指向寄存器结构，表示当前此符号被关联到了哪个寄存器上。

当产生对 sym 对应寄存器修改的指令时，dirty 位置 1。

当到达基本块出口时，调用 reg_wb_all 函数产生指令将 dirty 为 1 的寄存器写回内存。同时将原来所有关联取消，以便下一个基本块分配。

分配函数的核心为 get_reg 函数。生成器将要使用的符号传递给 get_reg。

get_reg 函数首先查看是否符号已经关联，若是则直接返回寄存器号。否则，从 avail_list 链中取出一个可用寄存器，将符号关联到此。若 avail_list 为空，则产生溢出，将 list 上面的一个变量写回内存，在将符号关联到此。

5.2 体系结构相关特性优化

5.2.1 延迟槽的利用

由于这个生成器还十分简单，获取的全局信息也不够，因此 **对一般生成的指令，延迟槽内仅仅填写空操作**。但是 **对于函数框架模板、短路翻译方案等地方，手工做了优化**。

5.2.2 叶子函数

叶子函数是指此函数体内没有进一步函数调用。根据 MIPS 体系结构特点，不需要将返回地址放入内存。

我们在语义检查阶段对函数调用情况进行统计，当生成时，发现可以进行叶子函数的优化时，就产生特殊的指令，提高效率。

6 使用说明

6.1 编译

输入 make，得到 c1c 执行文件。

6.2 运行

从命令行读取参数，使用方法类似GCC：

编译生成EIR中间代码：

```
c1c src_file [-o out_file]
```

编译生成C代码：

```
c1c src_file [-o out_file] -m c
```

编译生成MIPS汇编代码：

```
c1c src_file [-o out_file] -m spim
```

帮助：

```
c1c -h
```