第34卷　第17期　　计 算 机 工 程　　2008年9月
Vol.34　No.17　　Computer Engineering　　September 2008
·软件技术与数据库·　　文章编号：1000—3428(2007)17—0074—03　　文献标识码：A　　中图分类号：TP314

可重定向C编译器中DAG及归约规则

张红光，赵彩云，李海丰，李福才，陈　鹏
(南开大学计算机科学与技术系，天津 300071)

摘　要：以在嵌入式系统中建立C编译器的技术特点为主要内容，用设计实例论述了C编译器实现中前端、后端的主要工作内容。说明了在前、后端之间起桥梁作用的中间描述语言有向无环图(DAG)的设计原理及形成方法，同时还就如何将DAG与目标机系统之间形成映射关系进行描述，提出了在映射中规约规则制定的方法和原则，给出了一些有指导意义的经验性结论。
关键词：宿主机；目标机；可重定向编译器；有向无环图；抽象语法树

DAG and Reduction Rules in Retargetable C Compiler

ZHANG Hong-guang, ZHAO Cai-yun, LI Hai-feng, LI Fu-cai, CHEN Peng
(Department of Computer Science and Technology, Nankai University, Tianjin 300071)

【**Abstract**】This paper, based on the technology of building C compiler in embedded system, discusses the mainly work in realization of C compiler with design examples. The paper presents the design principle and the form method of the middle description language DAG, which has the bridge function between front end and back end. Also this paper describes the relation from DAG to target machine system, puts forward rules in maps about the method and principle of the rule establishment, and gives some experience conclusions that have guided meaning.
【**Key words**】host; target; retargetable compiler; Directed Acyclic Graph(DAG); abstract syntax tree

1　概述

嵌入式系统中的软件开发一般采用主从模式，嵌入式微处理器所在的机器称为目标机(target)，而进行程序开发的机器称为宿主机(host)。目标机主要任务是运行嵌入式系统程序而不包括针对本系统的软件开发工具；宿主机完成代码的编辑、编译、汇编、链接和地址分段等开发工作。在嵌入式系统的宿主机上建立良好的软件工具链和开发环境，是完成嵌入式系统设计的基础。

在编译技术领域，根据不同用途，可以将编译程序分成：专门用于帮助程序开发和调试的诊断编译程序(diagnostic compiler)；着重于提高目标代码效率的优化编译程序(optimizing compiler)。如果一个编译程序产生不同于宿主机的机器代码，则称之为交叉编译器(cross compiler)；如果不需要重新编写编译程序中与机器无关的部分就能改变目标机的编译器称为可重定向编译程序(retargetable compiler)。

2　可重定向编译器与目标机

编译过程是一个复杂的过程，一般可划分为5个阶段：词法分析，语法分析，语义分析，中间代码产生及优化，目标代码产生[1]。在C编译器实现中，又会将编译程序划分为编译前端和编译后端。前端主要由与源语言有关但与目标机无关的那些部分组成。这些部分包括词法分析、语法分析、语义分析与中间代码生成，有时代码优化工作也有一部分在前端进行。后端主要包括编译程序中与目标机有关的那些部分，如与目标机相关的代码优化部分及目标代码生成等内容。编译器前端是依赖于源语言的部分。不同编译器的前端功能都类似，因此可以根据一些自动机和文法来确定，目前已经有一些成熟前端自动生成工具，如LEX, YACC[2]等。正是由于前端具有相对的独立性，许多编译器可以针对不同的高级语言由不同的模块进行处理，生成相同的中间代码。比如GCC的前端就可以接收C, C++, Java等多种高级语言。

后端和前端相比，更多地依赖于目标机的体系结构和指令集特性。根据前端传来的中间代码，不同的后端将生成不同的目标代码。与前端类似，编译器可拥有多个后端，用户在使用编译器时设置一些编译选项即可生成特定目标机代码。在编译器设计中，不需要重新编写编译器中与机器无关部分，精良的体系结构及高度的代码抽象设计就能实现适应新目标机的后端，该编译器即可重定向编译器[3](retargetable compiler)。

3　中间语言DAG

图1展示的是典型编译器前端、后端和中间语言之间的关系。

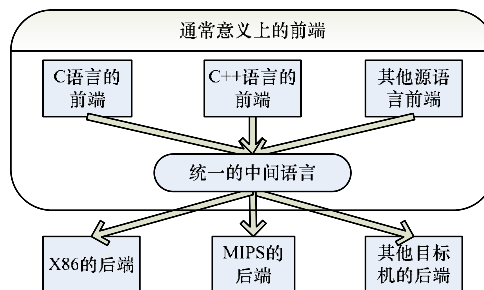

图1　编译器中前、后端及中间语言的关系

一款具有可重定向特性的编译器，其主体思想就是实现

作者简介：张红光(1955－)，女，副教授，主研方向：计算机系统软件，嵌入式系统平台技术，网络技术；赵彩云、李海丰，硕士研究生；李福才，高级工程师；陈　鹏，硕士
收稿日期：2007-10-15　**E-mail**：caiyunzhaonk@sina.com

其后端可以面向不同目标机生成不同的汇编代码。要做到这一点就需要在前、后端之间建立中间描述语言格式，将前后端的工作进行比较完善的分离。分离后在编译器的前、后端之间进行沟通的桥梁就是中间语言。

中间描述语言有多种，有向无环图(Directed Acyclic Graph, DAG)是一种典型的中间表述形式，它在多种编译器中得到了应用。比如 LCC[4]的编译前端主要任务是生成 DAG，而后端完成的主要工作是将 DAG 转化成目标机需要的汇编代码。

3.1 词法分析

C 语言是基于函数进行编译的，也即 C 编译器在编译 C 语言源程序时，发现一个 C 函数就会走一遍编译流程并生成相应的目标代码。这里用一个简单的示例程序做说明，假设最开始的源代码是：

```
int example (f) float f;{
return f+0.5;
}
```

在这里只关注编译过程，对预处理阶段的工作不作讨论。使用标准的预处理后会变为：

```
#1 “example.c”
int example (f) float f;{
return f+0.5;
}
```

预处理后，编译器开始工作，由词法分析器(lexical analyzer)和扫描器(scanner)将源程序分析分解成一个个单词(token)。

3.2 抽象语法树的构成

词法分析之后编译器根据 C 语言的语法规则对单词串进行分析(parse)，同时也分析程序语义(semantic)的正确性。示例源程序经过分析后，将形成 2 棵抽象语法树[3](abstract syntax tree)，如图 2 所示。

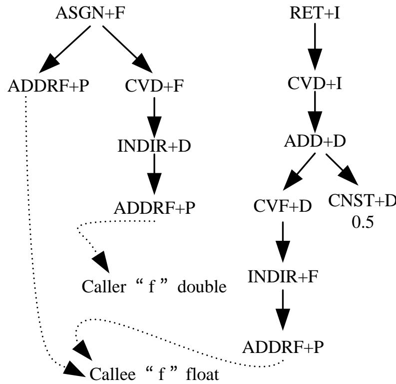

图 2 语法分析后的抽象语法树

树中的每个节点代表一个基本操作。第 1 棵树将传入的 double 类型的参数转换成 float 类型。节点(INDIR+D)从地址为&f 的单元(ADDR+P)取出 double 类型的值，节点(CVD+F)将该值转换为 float 值。节点(ASGN+F)把该值存入地址为&f 的单元(ADDRF+P)。第 2 棵树完成了示例中的语句，返回了一个整数(RET+I)。节点(INDIR+F)从地址为&f 的单元(ADDRF+P)取出 float 值，节点(CVF+D)将该值转换成 double 值，节点(ADD+D)把该值加上 double 常量 0.5(CNST+D)，节点(CVD+I)将结果转换成整数。

这些树使隐含在源代码中的一些含义更加清晰。上面的类型转换在源代码中并没有显式的体现，但是在 ANSI C 标准中是有明确规定的，在这些树中完成了这些类型转换。此外，树中还明确给出了所有操作的类型，如源代码中的加法都没有显式给出操作类型，但是在树中与其对应的节点都标明了具体的操作类型信息。

3.3 生成中间描述 DAG

根据图 2 语法树的含义，编译器前端将生成图 3 所示的 DAG，编号为 1 和 2 的 DAG 是从图 2 中的树转换而来的。操作符标记中不再有“ + ”号，而是将这些节点转换成相应的操作符。树变成 DAG 后，一些隐含的事实变得更加明显化，如树中 CNST+D 节点的常量 0.5，在 DAG 中成了名字为“ 2 ”的静态变量，CNST+D 操作符被替换成取地址操作符 ADDRGP 和取值操作符 INDIRD。图 3 中的第 3 个 DAG 定义了名字为“ 1 ”的标号，该标号将出现在目标程序 example 的末尾处，程序中的返回指令被翻译成跳转到该标号。

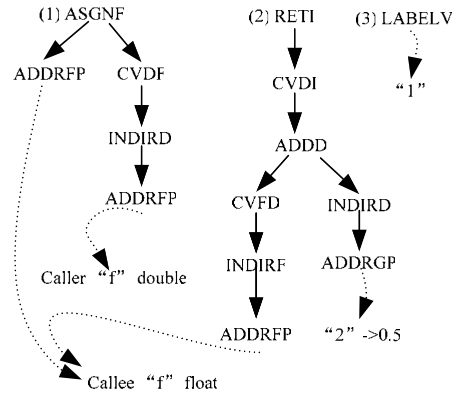

图 3 生成的 DAG

前端生成 DAG 的顺序与前端传给后端的代码表执行顺序是相同的。至此编译器前端的工作就基本完成了，主要成果是：表达式首先被编译为语法树然后转化为 DAG；每个函数的这些 DAG 被串在一起形成一个代码表，它是一个双向链表，代码表包含了函数的代码。前端完成这些工作后将生成的代码表传给后端代码生成器，由代码生成器把这些结构翻译成目标机上的汇编语言。

4 DAG 与目标机代码之间规约规则的制定

编译器后端代码生成器需要完成多项工作，其中一项主要工作就是将前端分析生成的中间代码与目标机代码形成对应，然后才能实现输出目标机代码。在编译器实现中，由 function 函数调用后端接口函数 gencode 实现该功能，目标机代码输出是由 function 函数调用 emitcode 完成的。这部分主要工作是形成目标机代码与中间代码之间的对应关系，并制定相应的规约规则。

4.1 DAG 规约规则

对于一款特定目标机的编译器设计，目标机的指令集和代码编写规则是固定的，与中间代码具有强对应性关系，不允许存在太多的灵活性。为了避免手动编程的易错性和重复性，可以借助工具完成，比如 LCC 提供的一个简单的树文法分析程序 LBURG[2]，它可以把根据规范写成的树文法翻译成 C 程序。与常规的文法类似，树文法也是一个规则列表，每条规则左边是一个非终结符，右边是终结符和非终结符组成的模式。这样生成的 C 程序可以直接作为 gencode 和 emitcode 接口函数的一部分来构成后端代码生成器。在实现中通过编写符合 LBURG 规范的树文法来实现中间代码和目标机代码的对应关系，大大降低了工作量并且保证了代码的正确性，

减少了 bug 出现的几率。由于这些树文法是将一个 DAG 归约为一个非终结符，也称这些树文法为 DAG 归约规则。

4.2 DAG 归约规则的规范

LBURG 接受的 DAG 归约规则有其规范，下面以实例说明为目标机制定的一些 DAG 归约规则：

```
addr:	reg	"%0"
reg:	ADDRGP "mov%c,%a\n" 1
reg:	ADDI(reg,reg)
"?mov%c,%0\nadd %c,%1\n" 3
```

从这些规则中可以看出，每个规则通常分为 3 个部分：第 1 部分(如 reg, addr)称为非终结符，它是用来体现归约关系的，是归约的结果；第 2 部分(如 ADDRGP, ADDI)称为终结符，它是 DAG 中节点的操作符，是 LCC 的代码生成接口的一部分，它代表被编译的 C 源程序的语义；ADDRGP 就是一个取全局变量地址的操作。第 2 部分也可以是一个非终结符，如规则：addr: reg "%0"。

该规则的意思是寄存器不需其他操作就可以直接归约为地址，因为目标机支持寄存器地址。第 3 部分是相应的汇编模板。可以看到汇编模板中还有以“%”开头的字符，它们在汇编输出时将被寄存器名或变量名所替换。其中，%0 代表该节点的左子树归约结果；%1 为右子树归约结果；%a 为该节点操作的变量名；%c 是为该节点操作分配的寄存器。

有的规则还有第 4 部分，它是该规则的代价值，如果缺省即为 0，通常都是相应汇编语句的指令周期。使用该代价值来选取代价最小的汇编实现。代码生成器对每个 DAG 树都进行 2 遍扫描：第 1 遍可以被认为是一个采用自底向上方式的标记程序，得到一个能以最小代价覆盖所有子树的模式集；第 2 遍可视为一个自顶向下方式的化简程序，它从第 1 遍标记程序记录的模式集中选取代价最小的覆盖。这样，代码生成器就可使用代价最小的模式生成代码。

下面来解释一下规则内容：

```
reg: ADDI(reg,reg)
"?mov %c,%0\nadd %c,%1\n" 3
```

该规则的含义是对于 DAG 中操作符为 ADDI(2 个整型数相加)的节点，如果它的左右子节点都归约为非终结符 reg，那么该节点也可归约为 reg，并且以“?mov %c,%0\nadd %c,%1\n”为模板生成汇编指令，模板中的“?”意思是如果根据寄存器分配规则为其分配的寄存器与左子节点归约所得的寄存器相同，那么“?mov %c,%0\n”这部分汇编不输出。

4.3 目标机规约规则的实现

要实现目标机的归约规则，首先必须确定归约规则中的非终结符。有非终结符归约规则才能够完成终结符的向上归约。参考了 LCC 现有的目标机代码生成器中所使用的非终结符，根据目标机特点，定义了 8 个非终结符，它们分别是：

(1)stmt：所有 DAG 归约的终点。

(2)reg ：匹配操作结果存放在寄存器中的树。

(3)addr：匹配地址。

(4)con：匹配常量。

(5)addrfp：匹配函数参数地址。

(6)addrlp：匹配局部变量地址。

(7)rc4：匹配寄存器和常量大小为 16 以内。

(8)ar：匹配存放在寄存器中的标号地址。

这些规约规则制定的依据是：

(1)stmt 是所有 DAG 归约的终点，也就是说 stmt 这个非终结符作为归约结果只会出现在 DAG 的根上，而不会是 DAG 的任何子树中。

(2)非终结符 reg 用来匹配结果存放在寄存器中的 DAG 子树，由于目标机寄存器非常多，该非终结符也是归约规则中使用最多的非终结符。

(3)addr 也是一个使用很多的非终结符，所有普通的地址操作都可归约到该非终结符。

(4)con 用来匹配常量，常量可以是立即数也可是寄存器常量。

(5)addrfp 匹配函数参数地址。

(6)addrlp 用以匹配局部变量地址，虽然局部变量和函数参数的作用域相同，但是由于它们在函数栈帧中的位置不同，还是将它们分成 2 个非终结符。

(7)rc4 用来匹配寄存器和常量，该非终结符是用于匹配移位操作中移位的位数的，因为目标机是 16 位的，所以它所代表的值必须为 16 以内。

(8)ar 匹配所有寄存器中的标号地址，它们包括全局变量地址、调用函数地址和常量指针。

4.4 归约规则制定的经验

经过实践可知，要制定对应于某个具体 DAG 节点的规则并不困难，重要的是要对目标机的特性有充分了解。但是要想制定出一套对应整个 DAG 的规则就比较困难了，因为 C 语言的语义千变万化，生成的 DAG 也纷繁复杂，可能出现的 DAG 节点非常得多，难于统计。如果根据可能出现的 DAG 节点来制定规约规则，那么这项工作几乎无法完成。在长时间完成规约规则制定的实践中，得出了几条有效的经验：

(1)对于每个终结符，如果能够转化为寄存器操作则尽量转化为寄存器操作，即使它可以直接使用内存操作且速度更快。这样可以尽量保证规则之间的可归约性，从而简化规则。而为了保证编译生成代码的效率，可以添加相应的代价值较小的非寄存器规则来做补充。

(2)尽可能少地制定非终结符归约到非终结符的规则。这也意味着尽可能少地定义非终结符。非终结符归约到非终结符意味着更多的归约层数和更多的循环归约，这会大大增加整套规则的复杂度。

(3)对于机器指令集的指令尽量在规则中有所体现，这样可以尽可能地发挥相应机器的性能。

5 结束语

基于以上原则制定了目标机的 DAG 规则。目标机的 DAG 规则一共 155 条，远小于 DAG 节点可能出现的情况数。该套 DAG 规则目前已通过了标准 C 测试集的测试，并在其上完成了一些实用 C 程序的编写与调试。这些成果表明所制定的规则完成了对 DAG 的完全覆盖，而且实践证明以上 3 条制定规则的原则是行之有效的，可以较好地解决重定向中的实际问题。

参考文献

[1] Fraser C W, Hanson D R. The Lcc 4.x Code-generator Interface[R]. Microsoft Research, Tech. Rep.: MSR-TR-2001-64, 2001.

[2] 陈火旺. 程序设计语言编译原理[M]. 3 版. 长沙: 国防工业出版社, 2000.

[3] Fraser C W, Hanson D R. A Retargetable C Compiler: Design and Implementation[M]. 北京: 电子工业出版社, 2005.

[4] 李宝峰, 窦 勇, 周兴铭. 基于 LCC 的 LEAP 编译器设计与实现[J]. 计算机工程与科学, 2005, 27(1): 61-63.