形式规约语言中函数运算的优化*

张荣华 董韫美

(中国科学院软件研究所计算机科学开放研究实验室 北京 100080)

E-mail: ymdong@public.bta.net.cn

摘要 在运行时刻,由于某些函数参数的取值会造成一些函数被重复调用,这在编译时刻是无法被传统的优化器发现的.针对这种情况,提出一种运行时刻的优化方法.它通过调用依赖图来消除被重复调用的函数.

关键词 函数式语言,优化.

中图法分类号 TP311

本文的研究基于SAQ(specification acquisition system)系统中的LFC(language for context free recursive function)语言.该语言是一种函数式语言,它与其他函数式语言最大的不同是,它把上下文无关语言作为其基本的数据类型,并实现了上下文无关语言上的递归函数[1].在LFC中,定义函数是通过对上下文无关语言进行结构归纳和模式匹配进行的[2].由于LFC定义函数的独特方式,如果函数参数中存在重复的结构就可能会造成许多函数被重复调用.如果能够消除这些冗余的函数调用,LFC的执行效率就会得到提高.

由于LFC中出现的重复函数调用是由运行时刻某些参数的取值造成的,因此编译时刻的优化是无法发现它们的.目前已经提出的运行时刻的优化有:值驱动的运行时刻优化[3]、基于数据分析的代码优化[4]、带结果缓冲的函数实现[5]等.而对于值驱动的运行时刻优化和基于数据分析的代码优化都是利用某些变量的值的不正常分布进行优化的,它们也不大适用于LFC.只有带结果缓冲的函数实现是可能运用于LFC上的.但是,该方法对于函数定义中包含有自递归调用的情况效果很好,对于其他情况则显得效率不高.

针对这种情况,我们提出从参数的语法结构入手,通过寻找重复结构来搜索可能的重复函数调用,进而消除冗余调用.文章第1节简要介绍LFC.第2节举例说明重复函数调用的存在.第3节介绍优化算法,并给出对第2节中的实例进行优化的过程.最后给出实验结果.

1 LFC简介

由于LFC把上下文无关语言作为其基本的数据类型,因此在利用LFC定义函数运算之前首先需要定义作为其定义域和值域的上下文无关语言,然后通过对它们进行结构归纳来定义函数运算.

例如,假定我们要讨论的对象是初等函数的某个子类,即在整数和变量x,y,z上有限次施行+、-、*、正弦、余弦和加括号等构造操作后得到的函数类.我们还要在该函数类上定义形式微分运算,在LFC中可以表达如下.首先,定义初等如下函数(产生式后面的数字表示产生式的序号,并不是定义的一部分.Num是系统支持的类型,故略去其定义):

⟨ElemFunc⟩→⟨ElemFunc⟩+⟨Term⟩ (1)

⟨ElemFunc⟩→⟨Term⟩ (2)

⟨ElemFunc⟩→-⟨Term⟩ (3)

* 本文研究得到国家自然科学基金(No.69873042)和国家"九五"重点科技攻关项目基金(No.96-729-06-02)资助.作者张荣华,1975年生,硕士生,主要研究领域为软件设计方法.董韫美,1936年生,研究员,博士生导师,中国科学院院士,主要研究领域为软件设计方法.

本文通讯联系人:董韫美,北京 100080,中国科学院软件研究所计算机科学开放研究实验室

本文2000-01-17收到原稿,2000-04-21收到修改稿

⟨Term⟩→⟨Term⟩ * ⟨Factor⟩ (4)
⟨Term⟩→⟨Factor⟩ (5)
⟨Factor⟩→⟨TriFunc⟩ (6)
⟨Factor⟩→⟨Var⟩ (7)
⟨Factor⟩→⟨Num⟩ (8)
⟨Factor⟩→(⟨ElemFunc⟩) (9)
⟨TriFunc⟩→sin(⟨ElemFunc⟩) (10)
⟨TriFunc⟩→cos(⟨ElemFunc⟩) (11)
⟨Var⟩→x (12)
⟨Var⟩→y (13)
⟨Var⟩→z (14)

然后,定义初等函数的微分运算,其中 tmDiff,faDiff,tfDiff 是为定义 Diff 而引入的辅助函数(下文中$x[]y$表示由 x 和 y 连接产生的值.""用来括纯字符串.例如:$x0[]"+"[]x1$ 就表示由一个初等函数 $x0$,一个加号+和一个项 $x1$ 形成的字符串.concat 表示字符串的连接运算).

```
dec Diff: ElemFunc->ElemFunc;  /* 函数 Diff 定义域为 ElemFunc,值域为 ElemFunc */
var x0: ElemFunc;
    x1,x2,x3: Term;
def Diff(x0[]"-"[]x1)=concat(Diff(x0),"+",tmDiff(x1));
/* 当参数是由产生式⟨ElemFunc⟩->⟨ElemFunc⟩|⟨Term⟩生成时,使用该定义式 */
    Diff(x2)=tmDiff(x2);
/* 当参数是由产生式⟨ElemFunc⟩->⟨Term⟩生成时,使用该定义式 */
    Diff("-"[]x3)=concat("-(",tmDiff(x3),")");
/* 当参数是由产生式⟨ElemFunc⟩->-⟨Term⟩生成时,使用该定义式 */

dec tmDiff: Term->ElemFunc;
var x0: Term;
    x1,x2: Factor;
def tmDiff(x0[_"*"[]x1)=concat("(",tmDiff(x0),")*",x1,"+",x0,"*(",faDiff(x1),")");
    tmDiff(x2)=faDiff(x2);

dec faDiff: Factor->ElemFunc;
var x0: TriFunc;
    x1: Var;
    x2: Num;
    x3: ElemFunc;
def faDiff(x0)=tfDiff(x0);
    faDiff(x1)="1";
    faDiff(x2)="0";
    faDiff("("[]x3[]")")=Diff(x3);

dec tfDiff: TriFunc->ElemFunc;
var x0,x1: ElemFunc;
def tfDiff("sin("[]x0[]")")=concat("cos(",x0,")*(",Diff(x0),")"),
    tfDiff("cos("[]x1[]")")=concat("-sin(",x1,")*(",Diff(x1),").
```

2 实 例

在 LFC 的环境下执行 Diff("sin(x)+cos(sin(x))")将会有如图 1 所示的函数调用树(该调用树是逻辑上的,实际上并不存在)出现.

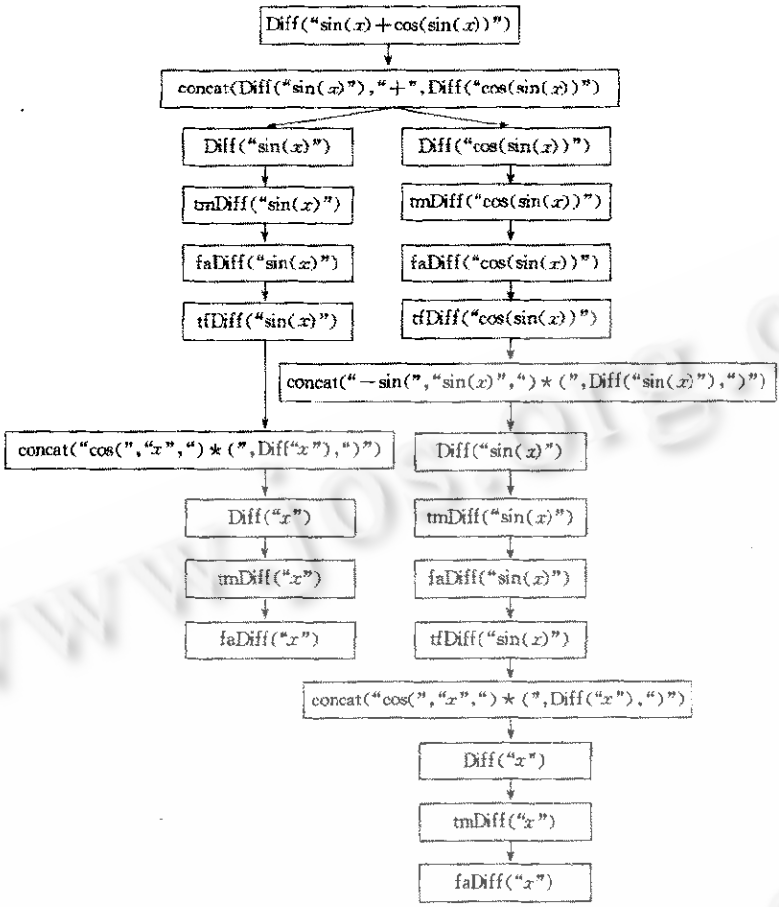

Fig. 1 Function call tree of Diff("sin(x)+cos(sin(x))")
图1 Diff("sin(x)+cos(sin(x))")的函数调用树

从图1可以看出,由于在"sin(x)+cos(sin(x))"中重复子串"sin(x)"(严格地说,应该是重复结构)的存在,由于重复计算Diff("sin(x)")而导致出现了大量的重复函数调用.这种函数调用是无法在编译时刻检测出来的,只可能在运行时刻发现,并且是特定于每次函数调用的.重复子串造成重复函数调用的原因在于LFC的函数定义是根据结构进行归纳的.如果重复子串在语法上等价于重复结构,那么必然导致重复的函数调用.事实上,重复函数调用并不局限于上面的例子,在一些递归函数中有更严重的重复函数调用.例如,斐波那奇函数:

$$Fib(n)=\begin{cases}0, & n=0,\\ 1, & n=1,\\ Fib(n-1)+Fib(n-2), & n>1.\end{cases}$$

3 优化算法

由于大多数的重复函数调用是由于重复结构造成的,我们可以通过检查重复结构来预测出现重复函数调用的可能性.由于函数调用时本来就需要对参数进行语法分析以进行类型检查,因此,这一步骤并不增加很多的开销.发现参数有重复结构以后,我们就可以生成函数调用依赖图,然后根据该依赖图进行计算而得到最后结果.

由于这里的语法分析图和调用依赖图都是有向无环图,我们将它们看成树的特殊形式.因此,在下面的算法描述中,我们使用了有关树的一些术语,如,子树、根节点等.

步骤1. 建立函数参数的语法分析图

我们首先使用语法分析的通用算法——Earley算法[5]对最外层函数调用的参数进行语法分析,在得到其最

右分析序列以后,使用算法1建立其语法分析图.该算法从左到右处理该分析序列,并且使用了一个栈来保存当前生成的各子树的根节点.

算法1. 根据最右生成序列建立语法分析图.

输入:最右分析序列:$a_1,a_2,\ldots,a_n$.其中a_i为产生式序号;

输出:语法分析图.

步骤:(1) 将栈清空,取分析序列中的第1个序号A,转(2).

(2) 对于序号为A的产生式,记它右部的非终极符个数为$k(k\geqslant 0)$,又记栈顶的k个元素为$P_1,\ldots,P_k$.如果在已经生成的节点中存在节点J,其标号为A(节点标号即以该节点为根的子树所用产生式的序号),其k株子树为$P_1,\ldots,P_k$,转(3);否则转(4).

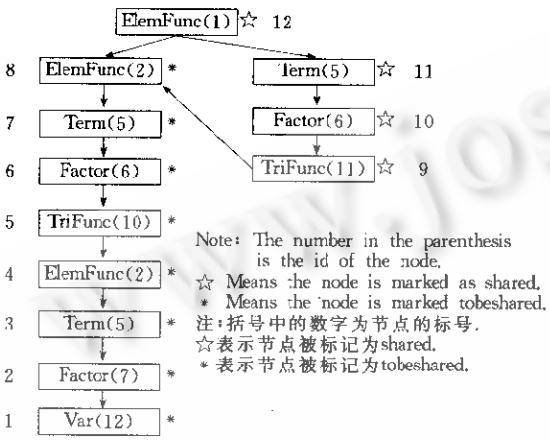

Fig. 2 Syntax graph of sentence: $\sin(x)+\cos(\sin(x))$
图2 $\sin(x)+\cos(\sin(x))$的语法分析图

(3) 将J节点标记为shared,并将从J所在子树的树根到J的父节点这条路径上的所有未被标记成shared的节点均标记成tobeshared.将栈顶的k个元素弹出,将J压入栈中.转(5).

(4) 生成1个新节点J,将其标号设置为A.将栈顶的k个元素作为节点J的子树(最上面的元素作为最后一株子树).将栈顶的k个元素弹出,将J压入栈中.如果J的k株子树中至少有1株被标记为shared或tobeshared,将J标记为tobeshared.转(5).

(5) 取分析序列中的下一个序号A,转(2);如果已经全部取完,结束.此时,栈中只有唯一的元素,即语法分析图的根节点.

例如,假定按照第1部分中ElemFunc的定义分析$\sin(x)+\cos(\sin(x))$,最终形成的分析图如图2所示.

步骤2. 建立函数调用依赖图

定义1. 函数调用$P_i(p_1,\ldots,p_n)$是可扩展的当且仅当:$p_1,\ldots,p_n$之一所对应的语法分析图节点被标记为shared或tobeshared,或者$p_1,\ldots,p_n$之一是可扩展函数调用.

定义2. 函数调用依赖图是一个二元组$\langle V,E\rangle$,V中的每个元素都代表函数调用,而边$\langle i,j\rangle\in E$当且仅当函数调用i使用了函数调用j的运算结果.

算法2. 建立函数调用依赖图.

输入:初始函数调用$I(i_1,\ldots,i_s)$,$s>0$(当$s=0$时,函数调用没有参数,不必使用本算法进行优化);函数参数的语法分析图;函数的定义;

输出:函数调用依赖图.

步骤:(1) 节点队列清空.为初始函数调用$I(i_1,\ldots,i_s)$生成一节点I_0.将I_0放入节点队列,转(5).

(2) 记当前节点为$C=F(p_1,p_2,\ldots,p_n)$,$n>0$(节点队列中都是可扩展的函数调用.根据可扩展函数调用的定义,函数的参数个数一定大于0).如果$p_1,p_2,\ldots,p_n$都不包含函数调用,转(3).否则,取第1个可扩展的函数调用p_i,转(4);如果没有,转(5).

(3) 根据函数F的定义获得与参数$p_1,p_2,\ldots,p_n$相匹配的定义式Q.

(a) 如果Q不是函数调用,转(5);否则,转(b).

(b) Q是函数调用,记其参数为$q_1,\ldots,q_m(m\geqslant 0)$.

① 如果已经存在节点$J=Q(q_1,\ldots,q_m)$,将J作为C的子节点,转(5);否则,转②.

② 为函数调用Q生成一个新节点I,作为C的一个子节点.如果$Q(q_1,\ldots,q_m)$是可扩展的,I进入节点队列,转(5).

(4) 记 C 的当前参数 $p_i=P_i(v_1,\ldots,v_t)$,$t>0$(节点队列中都是可扩展的函数调用.根据可扩展函数调用的定义,函数的参数个数一定大于 0).

(a) 如果已经存在节点 $J=P_i(v_1,\ldots,v_t)$,将 J 作为 C 的一个子节点,转(c);否则,转(b).

(b) 为 p_i 生成一个新节点 I,作为 C 的一个子节点.若 $P_i(v_1,\ldots,v_t)$是可扩展的,I 进入节点队列,转(c).

(c) 取 C 的下一个可扩展的函数调用,转(4);如果全部取完,转(5).

(5) 如果节点队列为空,转(6);否则,取节点队列的下一个节点,转(2).

(6) 此时,初始调用 $I(i_1,\ldots,i_s)$对应的节点 I_0 即为图的根节点.

例如,假定按照第 1 节中 Diff 的定义分析 Diff("sin(x)+cos(sin(x))"),可以得到如图 3 所示的调用依赖图.

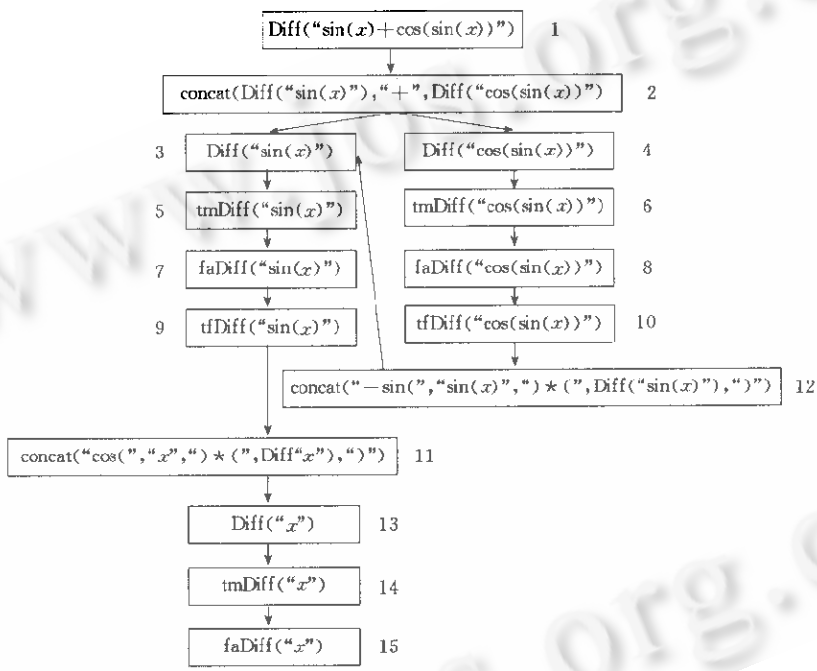

Fig. 3 Call dependency graph of Diff("sin(x)+cos(sin(x))")

图3 Diff("sin(x)+cos(sin(x))")的调用依赖图

步骤 3. 根据调用依赖图进行运算

(1) 对生成的调用依赖图进行拓扑排序.

(2) 根据(1)得到的拓扑排序结果的逆序进行计算.

注:上面介绍的算法是针对上下文语言上的递归函数,它同样也适用于数值上的递归函数.只是对于数值计算函数,由于不是按结构定义的,我们不进行步骤 1.步骤 2 也略有变化,不再考虑参数的可扩展性.

4 实验结果

以上算法已经在微机版的 SAQ 系统上实现,并进行了实验.

例 1:初等函数的形式微分运算,包括一些简单的化简,共有 63 个函数定义.基本上是介绍部分所举例子的扩充.

所求值的函数如下:

(1) Diff("cos(x)+sin(cos(x))"),

(2) Diff("x*y+cos(x*y)+sin(x+x*y)"),

(3) Diff("cos(ln(x))+ln(x)*x").

例 2:斐波那奇函数.

(1) *Fib*("10"),

(2) *Fib*("20"),

(3) *Fib*("25").

从表 1(实验机器为 PII350,运行时间都是指从用户输入函数表达式到得到最后结果所用的时间.由于 Windows不能精确地计算某个进程所使用的时间,因此表中的数据有一定的误差)中可以看出,该方法对于不同的函数调用都有不同程度的优化.由于斐波那奇函数计算过程中会出现极其大量的重复调用,因此其优化效果也格外显著.

Table 1 Experimental result

表 1 实验结果 (ms)

Example①	Example 1			Example 2		
Application②	1	2	3	1	2	3
Before optimization③	2 850	2 960	2 470	60	7 470	85 470
After optimization④	2 470	2 580	2420	50	60	60

①例子,②应用,③优化以前,④优化之后.

5 结 论

本文提出了一种运行时刻的优化方法,并在改进的 LFC 环境下进行了实验.结果表明,该方法对于 LFC 是有效的,对其他函数式语言也可供借鉴.

致谢 在本文的写作过程中得到了陈海明博士的帮助,在此表示感谢.

参考文献

1 Dong Yun-mei. Recursive function on context free language(I). SAQ Report No. 22, ISCAS-LCS-98-14, Laboratory of Computer Science, Institute of Software, the Chinese Academy of Sciences, 1998. 3~6
(董韫美.定义在上下文无关语言上的递归函数(I).中国科学院软件研究所计算机科学开放研究实验室报告,SAQ Report No. 22,ISCAS-LCS-98-14,1998. 3~6)

2 Dong Yun-mei. The design and implementation of formal specification acquisition system (SAQ). SAQ Report No. 9, ISCAS-LCS-96-1, Laboratory of Computer Science, Institute of Software, the Chinese Academy of Sciences, 1996. 12~13
(董韫美.形式规约的获取系统 SAQ 的设计和实现.中国科学院软件研究所计算机科学开放研究实验室报告,SAQ Report No. 9,ISCAS-LCS-96-1,1996. 12~13)

3 Keppel D, Eggers S J, Henry R R. Evaluating runtime-compiled value-specific optimizations. Technical Report, UW-CSE-93-11-02, 1993. 1

4 Muth R, Watterson S, Debray S. Code specialization based on value profiles. 2000. 1~2. http://www.cs.arizona.edu/people/debray/papers/valuespec.ps

5 McNamee P, Hall M. Developing a tool for memoizing functions in C++. ACM SIGPLAN Notices, 1998,33(8):17~20

6 Aho A V, Ullman J D. The Therory of Parsing, Translation and Compiling (Vol. 1: Parsing). Englewood Cliffs, NJ: Prentice-Hall, 1972. 320~330

Optimization of Function Evaluation in Formal Specification Language

ZHANG Rong-hua DONG Yun-mei

(*Laboratory of Computer Science Institute of Software The Chinese Academy of Sciences Beijing* 100080)

Abstract At run-time, some parameter values may cause duplicate function calls which cannot be found at compile-time by traditional optimizer. In this paper, the authors propose a run-time optimization method. It can eliminate the duplicate calls by creating function call dependency graph.

Key words Functional language, optimization.