国防科技大学学报
JOURNAL OF NATIONAL UNIVERSITY OF DEFENSE TECHNOLOGY
第11卷第1期 1989年8月 Vol.11.No.1

编译型PROLOG系统的若干优化实现技术*

张 晨 曦

(计算机系)

摘要 文中论述可应用于设计编译型Prolog系统的若干优化实现技术。这些技术包括：执行驱动编译策略，代码分类以及数据库操作内部谓词的操作模式。

关键词 Prolog系统，编译，优化技术，编译策略，数据库操作

分类号 TP314

引 言

Prolog[1]是一种基于一阶谓词逻辑的实用程序设计语言。它具有很强的表达能力，具有语义清晰、可读性好等优点。虽然人们在标准或扩充了的Prolog能否作为五代机语言的问题上存在不同的看法，但Prolog作为一种具有独特风格的语言存在，已为大多数人所接受。作为一种程序设计语言，其实现效率非常重要。如何提高Prolog的执行效率，是多年来逻辑程序设计领域的一个重要研究课题。

与传统语言的情况类似，采用编译技术是提高Prolog系统性能的一个重要途径。近年来出现的许多高性能Prolog系统都是基于编译技术。然而，编译技术也带来了一些新的问题[2]。例如：引入编译开销，增加实现数据库的动态修改以及交互性的困难等。只有合理地解决这些问题，才有可能实现实用的Prolog系统。本文介绍我们在对编译型Prolog系统的研究过程中提出的若干优化实现技术。这些技术能为有效地解决上述问题提供支持。文中首先介绍执行驱动编译策略，然后讨论代码分类，最后介绍数据库操作内部谓词（简称DBOP内部谓词）的操作模式。

1 执行驱动编译策略

根据编译一个程序（或其中一部分）所发生的时刻不同，作者把编译策略分为以下几种。

1.1 动态编译

这是一般传统语言（例如Pascal、C等）所采取的策略。采用这一策略时，程序的编译和执行是相对独立的两个过程。一个程序只有全部编译成代码后才能执行。如果在程序的执行过程中想修改程序，就必须先退出执行状态，修改后还要重新进行编译和链接/装入。显然这种策略不适用于象Prolog这样交互性很强的语言（静态过程的编译除

*国家自然科学基金资助项目
1988年8月23日收稿

外）。

1.2 动态编译

采用这种编译策略时，可以在运行期间动态地编译一段程序和调用链接/装入程序把由此产生的代码与原有的代码链接起来。因此，允许动态修改程序而无需重新编译整个程序。由于Prolog向用户提供了动态修改程序库的功能，编译型Prolog系统必须采用动态编译策略。虽然传统语言的动态编译实现起来很困难，Prolog的动态编译则要简单得多。其主要原因是Prolog是一种基于逻辑的语言。在Prolog中，变量都是逻辑的，不存在全局变量（DBOP内部谓词的变元除外）。此外，变量的作用范围是在一个子句内，而子句则是一个较小的单位。

Prolog的动态编译策略还可进一步分为两种：

（1）插入驱动(assertion driven)编译：

每当往Prolog数据库中插入新的子句（或过程时），就调用编译器对新插入的子句进行编译。程序在系统中总是以代码的形式出现。该策略只要求系统中有一个代码库。

（2）执行驱动(execution driven)编译：

这是我们提出的一种编译策略。按照这种策略，各过程都以源程序的形式存在于系统中。一个过程只有等到第一次执行时才被编译成代码、并链接/装入代码库。该策略要求系统中设有两个实际的数据库：源程序库和代码库。它们分别用于存放程序的源形式和代码。

用一个过程表来支持这种策略的实现。过程表项中包含有指向相应过程源形式和目标代码的指针和一个编译标志位。编译标志位指出该过程的编译状态。当调用一个过程时，若它已处于编译状态，则直接执行之。否则，就需先编译该过程，然后再执行。

可以看出，采用的执行驱动编译策略是以过程为单位。实际上，还有一种以子句为单位的执行驱动编译策略。由于采用后一种策略时，过程的各子句是递增式地编译，所以不是难以形成带索引的代码，就是构造索引的开销太大。而且每个子句都需要包含有编译状态、源项指针、代码指针等信息，管理的动态开销太大。特别是，在实际应用中，一个Prolog过程通常是或者用于执行，或者用做数据结构。因此，以过程为单位比较合理。

执行驱动编译策略的优点有两个。一是只需编译要执行的过程。这对于只执行大程序中一小部分的情况来说，优越性更加明显。另一个是它隐含DBOP内部谓词的优化实现（见第3节）。

在目前正在开发的系统中，作者拟综合采用上述各编译策略。对于程序中已调试好、且无需修改的过程进行静态编译，而对其余过程则尽可能采用执行驱动动态编译。因为是以过程为单位，所以当对于已处于编译状态的过程插入新的子句时，对该子句而言，是采用插入驱动编译。

2 代码分类

经过分析，我们发现对Prolog数据库中过程的修改操作可分为两类：

（1）动态修改

这是指当修改一个过程时，系统中存在该过程的子句特例的情况。

（2）静态修改

这是指除动态修改以外的所有修改操作。用户在交互式情况下对数据库的修改以及在程序运行过程中的大多数修改都属于这类。

例1

考虑下述过程：

```
main:- assertz(main), fail.
main:- fail.
```

当执行assertz时，在该过程的末尾插入事实：main. 由于系统中存在第一个子句的特例，因此该插入操作属于动态修改。

例2

```
main:- retract(fact(second)),fact(X),write(X),fail.
fact(first).
fact(second).
fact(third).
```

这个程序中的retract将删除fact过程的第二个子句。因为系统中不存在fact过程的任何子句的特例，此删除操作为静态修改。

为了实现代码的优化，可把过程（代码）相应地分为三类：

（1） 静态过程（代码）：不可修改；

（2） 半静态过程（代码）：可静态修改；

（3） 动态过程（代码）：可动态修改。

这里的静态代码直接采用Warren代码[3]。Warren代码增加少量的管理信息和指令，就可成为半静态代码。动态代码与Warren代码在形式上的主要区别是选择点指令不同。为了实现动态候选子句序列语义[4]，动态代码采用了我们增设的两条选择点指令：ctry_me_else和trust_fail.

从修改灵活性来看，动态代码最高，半静态代码次之，静态代码最低。而在执行效率上，其顺序则相反。静态代码和半静态代码都可以带索引。而且，在进行修改操作后，仍尽可能保持半静态代码的索引。这是通过修改或重新构造索引来实现的。动态代码不能带索引，这是因为当采用动态候选子句序列语义时，实现索引动态代码非常困难，开销太大。有关这两种代码的管理方法，见文[5]。

由于动态修改和静态修改有许多质的区别，上述代码分类是非常必要的，否则动态候选子句序列语义就会低效到令人无法接受。这两种修改的实现方法也大不相同。对于动态修改来说，因为系统中可能存在已进入被修改过程，但还未执行完的过程调用，故存在如何确定动态代码的语义的问题。此外，已删除子句空间的回收方法也比较复杂。静态修改则不同，半静态代码的语义总是一致的，而且已删除子句的空间可以立即回收。

作者在Prolog抽象机模拟系统WAM-PLUS-SES[6]上用几个典型程序对动态代码

和半静态代码进行了比较，结果如下：

（1）求反表(reverse)

```
main:- list30(L),nreverse(L,X),write(X),nl.
nreverse([X|L0],L):-
        nreverse(L0,L1),concatenate(L1,[X],L).
nreverse([],[]).
concatenate([X|L1],L2,[X|L3]):-
        concatenate(L1,L2,L3).
concatenate([],X,X).
list30([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
    22,23,24,25,26,27,28,29,30]).
```

表 1

比较项目	半静态	动态	动态/半静态
已执行指令的总数	4535	4539	1.01
失败的次数	0	31	
数据区读操作的次数	2951	3292	1.12
数据区写操作的次数	1643	8092	4.93
dereference操作的次数	467	467	1.00
trailing操作的次数	0	467	
尾栈栈顶指针的最大值	0	467	
局部栈栈顶指针的最大值	192	6136	31.96

（2）快速分类(quicksort)

```
main:- list50(L),qsort(L,X,[]),write(X),nl.
qsort([X|L],R,R0):- partition(L,X,L1,L2),
        qsort(L2,R1,R0),qsort(L1,R,[X|R1]).
qsort([],R,R).
partition([X|L],Y,[X|L1],L2):-
        X<Y,!,partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]):- partition(L,Y,L1,L2).
partition([],_,[],[]).
list50([27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,47,28,
    82,6,11,55,29,39,81,90,97,10,0,66,51,7,21,85,27,31,
    63,75,4,95,99,11,28,61,74,18,92,40,53,59,8]).
```

表　2

比较项目	半静态	动态	动态/半静态
已执行指令的总数	5373	5599	1.04
失败的次数	124	275	2.22
数据区读操作的次数	5462	6976	1.28
数根区写操作的次数	5680	8257	1.45
dereference操作的次数	555	555	1.00
trailing操作的次数	347	505	1.46
尾栈栈顶指针的最大值	224	381	1.70
全局栈栈顶指针的最大值	659	659	1.00
局部栈栈顶指针的最大值	75	3962	52.83

（3）排序(serialize)

（源程序略）

表　3

比较项目	半静态	动态	动态/半静态
已执行指令的总数	3462	3600	1.04
失败的次数	86	169	1.97
数据区读操作的次数	4124	5028	1.22
数据区写操作的次数	2564	4713	1.84
dereference操作的次数	278	278	1.00
trailing操作的次数	131	202	1.54
尾栈栈顶指针的最大值	96	167	1.74
局部栈栈顶指针的最大值	48	1037	21.60

（4）查询(query)

表　4

比较项目	半静态	动态	动态/半静态
已执行指令的总数	23002	52930	2.30
失败的次数	626	16930	27.05
数据区读操作的次数	34504	229454	6.65
数据区写操作的次数	16398	55792	3.40
dereference操作的次数	8927	23201	2.60
trailing操作的次数	2571	2601	1.01
尾栈栈顶指针的最大值	9	9	1.00
局部栈栈顶指针的最大值	57	120	2.11

3 DBOP内部谓词的操作模式

DBOP内部谓词包括assert、retract等。它们给用户提供了动态修改Prolog程序的功能。在实际应用中，它们还经常被用来实现全局变量、中间结果的保存[7]以及全局栈废旧空间的回收[8]。下面举一例说明用assert和retract实现的全局栈空间管理。

例3

```
gc(Goal):-call(Goal),lock(Goal).
gc(Goal):-unlock(Goal).
lock(Term):-abolish(info_lock,1),
            assert(info_lock(Term)),fail.
unlock(Term):-retract(info_lock(Term)),
            abolish(info_lock,1).
```

对于调用目标$g(t_1,\cdots,t_n)$，若用$gc(g(t_1,\cdots,t_n))$调用上述过程，则在执行完$g(t_1,\cdots,t_n)$后，能实现部分全局栈空间的回收（这种方法要求调用目标$g(t_1,\cdots,t_n)$执行完后，不留下选择点）。

例3中lock(Goal)的作用是把整个调用目标（包括其各变元）作为事实info_lock的变元保存到数据库中，然后引起回溯。回溯的结果导致全局栈废旧空间的回收。unlock的作用是在全局栈中恢复调用目标，并且删除在数据库中保存的副本。

在这些应用中，插入或删除的项并不当作子句执行，而只是用做数据项。因此，无需编译成代码。为了优化实现DBOP内部谓词，我们为之定义了两种操作模式：SINGLE和DOUBLE。SINGLE模式内部谓词只对源数据库进行操作，而DOUBLE模式内部谓词则同时对源数据库和代码库进行操作。显然，若在上述应用中采用SINGLE模式的assert和retract，就能大大提高时空效率。

在标准Prolog中，DBOP内部谓词隐含的操作模式为：

listing,clause——SINGLE;

assert,retract,abolish等——DOUBLE.

以往我们没有意识到操作模式的存在，是由于在解释系统中没有代码库的概念。把DBOP内部谓词的模式显式化，并没有破坏它们原来的语义，而只是扩充了一些SINGLE模式内部谓词。我们规定，对于具有两种操作模式的内部谓词来说，若在其谓词名的左上方标记有“ ”（例如 asserta），则表示其操作模式为SINGLE，否则就是DOUBLE。在后一种情况下，其操作语义与标准Prolog中相应内部谓词的语义相同。这里的标记与某些并行Prolog语言[9,10]中的标记在下述意义上是类似的。即它们都向用户提供了在程序中给出控制信息的手段。然而，一个重要的区别是，我们的标记是为了优化实现DBOP内部谓词，而这些内部谓词本身就是非逻辑的。因此，这种标记的非逻辑性是无可非议的。

必须指出，执行驱动编译策略已隐含了DBOP内部谓词的优化实现。因此，当采用这种策略时，无需引入操作模式。

4 结束语

Prolog的编译实现能大幅度地提高效率。本文论述了作者提出的编译型Prolog系统的三个优化实现技术。这些技术对于设计实用的Prolog系统有很大的意义。

参 考 文 献

[1] Clocksin W F and Mellish C S. Programming in Prolog, Springer—Verlag, 1981

[2] 张晨曦，慈云桂. Prolog的编译实现. 小型微型计算机系统，1988；9(1)

[3] Warren D H D. An Abstract Prolog Instruction Set, Technical Note 309, SRI International, 1983

[4] Zhang Chenxi and Ci Yungui.Study of Mechanisms that Support the Implementation of the Non-Logical Components of Prolog in WAM-based Systems.In Proc. of The Frontiers in Computing Conference, 1987

[5] 张晨曦，慈云桂. Prolog代码库的一种管理方法. 电子学报，1989；17

[6] 张晨曦. Prolog抽象机模拟系统WAM-PLUS-SES. 小型微型计算机系统1988；9(11)

[7] Moss C. CUT&PASTE-Defining the Impure Primitives of Prolog.In Proc. of the 3rd Int'l Conf.on Logic Programming, 1986

[8] Roy P V.A PROLOG Compiler for the PLM, Master's Thesis.Computer Science Division, Univ.of Califorhia, Berkeley, 1984

[9] Clark K L and Gregory S. Parlog: Parallel Programming in Logic. Research Report DOC Imperial College, London, 1984；(4)

[10] Shapiro E Y. A Subset of Concurrent Prolog and Its Interpreter. ICOT, Technical Report TR-033. 1983

Some Optimization Techniques for Compiler-based PROLOG Systems

Zhang Chenxi

Abstract

This paper describes some optimization techniques applicable to the design of compiler-based PROLOG systems, including execution driven compilation strategies, classification of procedure code, and operating modes for database operation builtins.

Key words: prolog system, compilation, optimization compilation straregies, database operation