PROLOG逻辑程序设计的结构优化

段 鹰 段文泽
(重庆大学机械学院 400044) (重庆建筑大学机电学院 400045)

摘 要 以减少存储空间消耗为主要目标,讨论逻辑程序设计的结构优化。分析了程序基本结构及其优化方法,提出了一种强制失败-数据旁路结构,作为已有优化方法的补充。由此综合出一种全局结构优化模式,用于机器翻译试验工作,效果显著。

关键词 逻辑程序设计,程序结构优化,存储空间消耗,人工智能
中图法分类号 TP311

任何程序设计的优化目标,主要在于追求时间和空间的最小消耗。在这方面,以PROLOG为代表的逻辑程序设计语言具有若干特色。它以一阶谓词逻辑为基石,属描述性语言,使其在逻辑推理方面明显优于其它所有语言。在程序设计上,只需对问题进行描述,而不必给出求解问题的每一步骤,也优于其它过程式语言。由此,它成为高新技术,特别是人工智能的一个重要支柱。但与此同时,其自身仍面临若干困难。状态空间搜索是其特长,但"组合爆炸"时也要消耗大量的时间和空间,对此,一些作者已进行过研究[1][2]。此外,在数据的传递与存储方面,PROLOG与C语言相比,更好地实现了封装性,它不设全局变量,而把每个谓词(相当于C中的函数)作成一个封闭体,除了可通过谓词名后括号中的"约束-非约束变量"(C中的形参-实参)的渠道与外界进行双向数据传递外,别无它路。这一方面减弱了模块之间的耦合性,增强了程序的可移植性,还减小了出错概率,另一方面却也少了一个重要的数据传递通道。问题在于,要通过"约束-非约束"变量渠道传递数据,谓词的执行必需成功,这样,那些不需传递的变量占用的空间就不能释放。这就是空间消耗与数据传递的矛盾。在程序规模增大时,这一问题尤为突出。这也许是PROLOG不能得到普及的一个重要原因。对程序的效率问题,文献[3]从使用截断、数据结构和求解方法选择、灵活使用规则等方面作了概述。如何解决上述矛盾,如何从节约空间的角度得到程序的优化结构,有待深入,这正是本文研究的重点。本文的形成以机器翻译工作为背景,面向PDC PROLOG 3.2(BOLLAAND公司1990年版本)[4]进行讨论。

1 PROLG中数据的存储方式

根据PROLOG的DOS版本,其基本内存分为几个区域:栈(Stack)、堆与全局栈(Heap and

收稿日期:1998-03-31
段 鹰,男,1971年生,研究生

Global Stack)跟踪(Trail)、生成代码(Generated Code)、和程序原正文(Program Source Text)。PDC PROLOG 在 O/C/Memory 菜单项中提供了设置栈、代码、跟踪空间大小的可能,在分配了这些空间和原程序代码后,余下的内存就作为堆和全局栈。在程序调试中,又提供谓词 storage(S,H,T),可返回 Stack,Heap 和 Trail 三种存储区的大小[4]。

在调试工作中,我们总结各类变量的存储方式如下:

栈用于存放以下数据:(1)调用子目标(谓词)时用于传递参数,(2)子目标调用时的现场保护和返回地址。一般说来,在子目标调用开始时分配动态存储空间,结束时释放。但是遇到非确定性时,却要保留回溯点,以备在执行后面其它子目标失败时,便于回溯,直到全部回溯点搜索完毕,或是遇到截断,才释放存储空间。

全局栈用于对串、项、表这些数据的临时自动存储。在子目标执行成功后,占用的存储单元并不释放。只在子目标失败时,才能释放。

堆存放较为永久的对象,如数据库事实、文件缓冲区、窗口缓冲区等。无论子目标执行成功与否,其所占用的空间都不释放。只有在内部数据库事实被删除、外部数据库被关闭、磁盘(或虚拟磁盘)文件、窗口被关闭的情况下,才能释放。

全局栈和堆分配在一个区域里。

跟踪用来寄存指针变量,如 Reference 域。在数据库查询中,回溯点的数目越多,占用空间越大。子目标的失败可释放跟踪空间。

生成代码是程序的实际机器码。

原程序正文只是在开发环境中用来交互式地运行程序。

2 逻辑程序的基本结构及其优化

PEOLOG 是描述性语言,但是在研究效率问题时,我们被迫从过程的角度对它进行考察并归纳出三种基本结构;顺序结构、选择结构和循环结构。下面从数据传递和存储的角度分析它们,研究其优化问题。

2.1 顺序结构

顺序结构如图 1a 所示,为完成目标 $p(W)$,先执行 $a(Z)$,再执行 $b(Z, W)$ 子目标。这些子目标可以由一些简单语句构成,也可以包含其它复杂结构,如选择结构、循环结构等。在顺序结构中,调用子目标时要分配栈空间,结束时释放,并不构成威胁。但是在临时读入数据时,例如,在执行 readlin(X),file_str(F, S),tef_term(db_selector, domain, Ref, Term)时,读入的串、项、表将占用全局栈空间,占用量取决于数据的长度,且在子目标完成后也不释放。这种情况多了,或是只要不退出 PROLOG 而反复执行这一子目标,占用空间将逐渐积累,最终将导致堆溢出(Heap Overflow)。如后面第 3 节所述,机器翻译中正好出现这种情况。

为了释放 Heap,只有在子目标 $a(Z)$,$b(Z, W)$ 执行成功后又强令失败。但是在失败后需要传递的约束变量 Z 就此消失,无法传递。这就构成了数据传递与内存占用的矛盾。对此,本文提出通过在虚拟盘建立传递文件进行传递的方案,如图 1b 所示。这里,执行子目标 a 时,将需要传递的数据 Z 写入文件 e:file.1 中,然后关闭文件,强令失败。而在执行子目标

$b(W)$时，又从文件中读出 Z，再关闭文件。这样，执行 a 时所占用的内存可全部释放。在EXAM1. PRO中给出了这样的例子程序。

```
EXAM1.PRO
proc(W):-
  a,
  b(W).
a:-
  readln(X),readln(Y),
  concat(X,Y,Z),
  openwrite(first,"e:file.1"),
  writedevice(first),
  write(Z),nl,
  closefile(first),
  fail.
a.
b(W):-
  openread(first,"e:file.1"),
  readdevice(first),
  readln(Z),nl,
  closefile(first),nl,
  readln(U),nl,
  concat(U,Z,W).
```

在选择方案时，要根据情况决定。一般在大型程序内层，传递数据不长，空间问题不突出，取图1a方案。在外层，累积占用空间已相当可观，或传递数据很长时，则取图1b方案，使用的条件是确信目标 a 定能成功，而后强令失败，又在另一同谓词子句中令其成功。

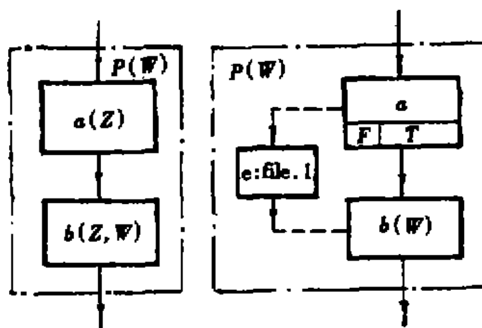

图1　顺序结构及其优化

2.2　选择结构

选择问题就是在若干相同子句中选择一个，这本身是个非确定性问题。但若使用截断(!)，可把非确定性问题变成确定的，达到节约时间和空间的目的。如果选择条件简单，象菜单程序那样，甚至可直接把条件写入规则头中，并紧接其后写上截断，效率更高。选择结构示于图2，例程为EXAP2.PRO。

```
EXAP2.PRO
predicates
  a(integer)
goal
  write("输入选择号(1~3)"),
  readint(K),a(K).
clauses
  a(1):-!,
  ......
```

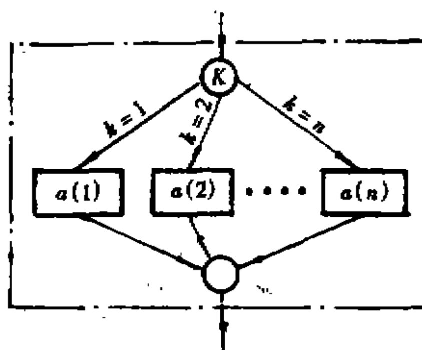

图2　选择结构

```
a(2):-!,
 ......
a(3):-!,
......
a(_):-
 write("输入有误!").
```

在上例中,若去掉截断,虽然执行某一子目标成功,但却留下回溯点,从而占用更多的空间。

2.3 循环结构

PEOLOG中有两类循环结构,一是回溯,二是递归。

2.3.1 回溯结构

PROLOG的回溯机制用于状态空间搜索是一个有力手段,它可从大量事实组合中求得问题的全部解。EXAP3.PRO是其典型结构。

```
EXAP3.PRO
a:-
  b(p,W),write(W),nl,
  fail.
a.
b(p,W):-
  c(W),d(p,W).
c(p1).
c(p3).
c(p6).
......
d(p,p2).
d(p,p3).
d(p,p6).
......
```

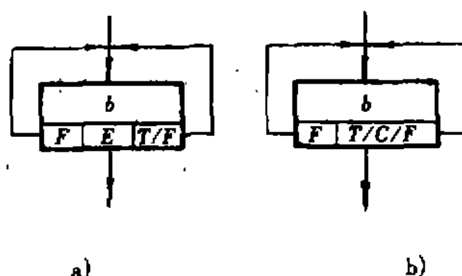

图3 回溯结构

例中要求按规则 $b(p, W)$ 去搜索对象 W,对 W 既提出要求 $c(W)$,又要使它与已知对象 p 满足一定关系 $d(p, W)$,我们用图3a表示这种结构。F 代表自然搜索失败后回溯,T/F 代表搜索成功强令失败回溯,E 代表事实库搜索完毕。由于每次操作都遭遇失败,就不但能释放回溯点占用的栈,还能释放堆,从而使程序优化。

如果只需求得一个解,就不必花费更多时间,可在例EXAM3.PRO中 $b(p, W)$ 后加上截断以消除不确定性。我们用图3b表示这种结构,其中,$T/C/F$ 表示搜索成功/截断/强令失败。也可不必强令失败并取消第二个子句 a,但操作中占用的堆和全局栈就不能释放。

图3的结构可任意多次地重复操作而无堆溢出之患。

即使目标没有多个解,也可使用回溯来构成循环,这需要定义下一简单谓词:

```
repeat.
repeat:-repeat.
```

加入repeat,使PROLOG认为它有无数个不同的解,就去不断回溯。repeat-fail结构经常使用在大型程序的外层。但要指出的是,它不象EXAP3.PRO那样,每次失败只引起回溯点往下移动,它属于多层次回溯,每次失败导致回溯点存储不断积累,栈空间不断消耗,只是每

次都不大(12字节)。至于堆,是能得到释放的。

2.3.2 递归结构

递归是一个调用自身的过程,其典型结构如表1。

一般递归结构中，循环次数及中间结果都能被当作变元从一个循环传递到另一个。这是它与回溯结构相比的优越性。但其一大缺点是占用很多内存：每当一过程调用另一过程时,必需保存调用过程的执行状态,以便在结束被调用过程后能够恢复如前。调用自身次数越多,使用的栈框架也越多。

优化的方案是采用尾递归[4],其要求归纳为两点：

1) 一过程将其对自身的调用作为其最后一步，这样，被调用过程结束后，调用过程就无其它任何事可作，就不必保存其执行状态了。这种操作相当于在一循环中更改控制变量，将 X, Y, Z 改为 $X1, Y1, Z1$。

(2) 在最后一个子句前面没有回溯点，在后面也没留下其它可选的未曾尝试过的调用。否则,每循环一次,就要消耗一个栈框架来保留回溯点。循环次数一多,仍要发生栈溢出。

表1 递归结构

种类	一般递归结构	尾递归结构
结构	a(X,Y,Z):-e(…),!. a(X,Y,Z):- b(…), c(…), a(X1,Y1,Z1), d(…).	a(X,Y,Z):-e(…),!. a(X,Y,Z):- b(…), c(…),!, a(X1,Y1,Z1).

在上述两个要求中,第一个比较容易满足。对第二个常用截断来实现。在表1的尾递归中，第一个截断消除了两个相同谓语子句造成的不确定性，第二个截断在条件 b, c 成功后防止了回溯,消除了回溯点从而释放了栈空间。

在综合应用中,只有尾递归还远不够。循环体中若和数据库打交道,只有在循环体内打开并及时关闭外部数据库或读入事实到内部数据库并及时删除它，才能释放堆。若和标准设备打交道,又只有将此递归结构放入外部的“失败-成功”结构中,堆空间才能释放。

例程EXAP4.PRO处理了以上各种情况。图4给出其图示。图中递归结构的两个标符号 a 正反映了过程调用其自身,P 是跳出循环的判断。

```
 EXAP4.PRO
domains
  database - dba1
    b(integer,string,string)
  database - dba2
    c(integer,string)
predicates
  proc
  a(integer,integer,integer,string)
  goal
  proc.
clauses
```

```
proc: -
  N = 1, readint(K), readint(M), readln(p),
  a(M,N,K,P), fail.
 proc.
a(M,N,K,P): - N = K + 1, !.
a(M,N,K,P): -
  consult("prod.txt", dba1),
  consult("part.txt", dba2),
  b(N,P,W),
  c(M,W), !,
  write(W), nl, N1 = N + 1,
  retractall(_, dba 1), retractall(_, dba2),
  a(M,N1,K,P).
a(M,N,K,P): -
  retractall(_, dba 1), retractall(_, dba2),
  N1 = N + 1,
  a(M,N1,K,P).
```

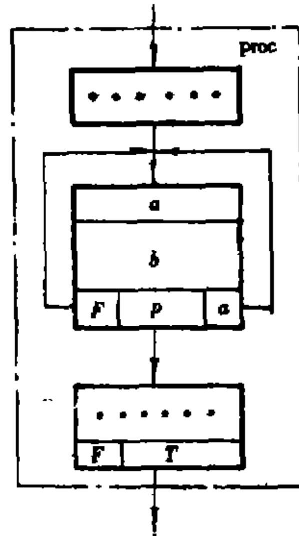

图 4 混合结构

需要指出的是，对数据库的读入和删除都放在循环体中，要消耗更多的时间。因此，如果可以接受，可在循环体外，例如，在 proc 谓词中处理。

3 逻辑程序全局结构的一种优化模式

当程序由若干模块构成一个整体时，PROLOG 称之工程(project)。当工程很大时，内存溢出往往成为很大的问题，我们应灵活地综合应用基本结构的各种优化手段，形成针对具体问题的优化全局结构。在机器翻译工作的基础上，我们在图 5 给出一个这样的全局结构。

图中实线表示的是在内存中进行的过程，虚线表示在内存和词盘(包括虚拟词盘)之间的交换过程。$p(1)$是编辑被译文件。$p(2)$从磁盘调入文件修改以产生被译原文件 *file.sor*。$p(3)$是主要的翻译操作，包括预处理和 4 个子模块，它们逐个产生 file.1 - file.4 最后是译文输出 flie.out。$p(4)$退出翻译系统。

整个程序的最外层是一个 repeat - fail 结构，保证翻译工作在不断重复中能释放出内存。但这还远远不够，在处理数据量很大的情况

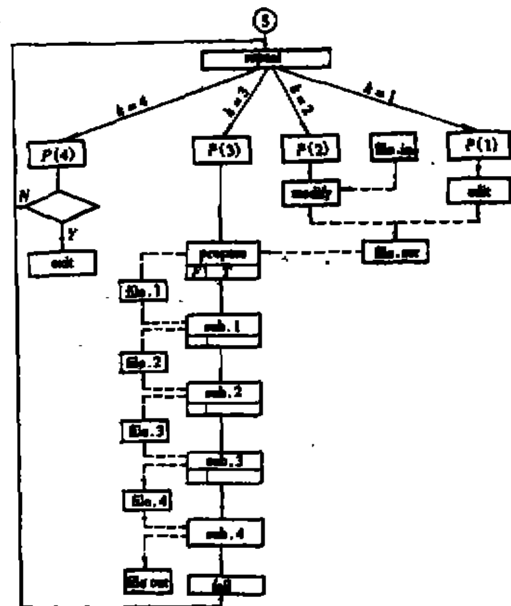

图 5 一个优化全局结构

下，prepare，sub1～sub3子模块都采取了"强令失败－数据旁路"的结构，只是sub4直接进入外层，因为外层的repeat－fail结构已保证了内存释放。每个子模块采用了第2节中所述的各种优化结构的综合。

表2给出了优化前后试验结果的对比。由于未优化的结构中每一子模块都以成功结束，堆不能释放；又由于选择结构未消除不确定性，栈和堆空间的消耗也急剧增长，在运行到第二轮时，就已出现堆溢出。优化结构的运行效果就很理想，每运行一轮，栈消耗仅12字节，这是运行repeat保留回溯点造成的。每轮的堆消耗为1字节，这是因为运行$p(2)$时总是成功，读入(y/n)字符造成的。

表2　优化结果对比

轮次		第一轮			第二轮		
剩余空间(bit)		Stack	Heap	Trail	Stack	Heap	Trail
未优化结构	处理前	30 966	163 133	75	30 844	113 716	75
	改文件后	30 850	159 068	75	30 728	113 705	75
	预处理前	30 718	158 168	75	30 578	113 705	75
	sub.1前	30 718	158 168	75	30 596	113 705	75
	sub.2前	30 718	134 533	75	30 596	93 094	75
	sub.3前	30 718	118 436	75	30 596	76 977	75
	sub.4前	30 718	107 612	75	Heap Overflow		
优化结构	处理前	30 966	174 674	75	30 954	174 673	75
	改文件后	30 960	174 673	75	30 948	174 672	75
	预处理前	30 810	174 673	75	30 798	174 672	75
	sub.1前	30 828	174 673	75	30 816	174 672	75
	sub.2前	30 828	174 673	75	30 816	174 672	75
	sub.3前	30 828	174 673	75	30 816	174 672	75
	sub.4前	30 828	174 673	75	30 816	174 672	75

对翻译工作的重复进行，优化前只能运行一次，优化后，按计算能运行2580次，这实际上相当于已不受限制，因为翻译多次后总要退出菜单。以后重新使用翻译系统时，一切又已恢复如初。可见以上研究为在微机上运行较大型的翻译软件奠定了一个良好基础。

参考文献

1 王鼎兴，温冬婵．逻辑程序设计语言及其实现技术．北京：清华大学出版社，南宁：广西科学技术出版社，1996：99～103

2 段文泽，段　鹰．闭环消除法解耦的递阶智能搜索．计算机应用研究，待发表

3 周立柱．PROLOG逻辑程序设计及应用．北京：清华大学出版社，1992：119～129

4 BOLLAND公司，唐晓程编译，PDC PROLOG实用大全(上下册)，北京：北京科海培训中心，1991(3)

(下转第109页)

On C60 HIgh - performance Concrete of Modified Extra Fine Sand and Its Application

Yang Zaifu Wang Liqiang

(Chongqing lst Construcfion Gtoup Company, 400042)

Ren Shiman

(Dept. of Material Science and Engineering, Chongqing Jianzhu University, 400045)

Abstract Utilizing the local materials in Chongqing, employing the techniques concrete of modified extra fine sand with three additives, we have conducted a cross experiment design test on multi - factors. In addition, in combination with the routine production procedures in ready - mixed pumping concrete, C60 high - performance concrete has been produced. Through the practical application in engineering projects, a conclusion canbe drawn that C60 high - performance concrete consisting of modified extra fine sand is of great value in its spreading and popularization.

Key Words modified fine sand, C60 high - performance concrete research, application.

(上接第 99 页)

Optimizing Configuration of the ROLOG Logic Programming

Duan Ying (Chongqing Univ, 400044)

Duan Wenze (Chongqing Univ. of Architecture, Chongqing Jianzhu University, 400045)

Abstract This paper discusses the optimizing configuration of the Logic Programming, taking the reduction of memory space consumption as the main goal. The elementary confingurations and their optimization are studied. As a supplement of the existing methods, a optimal configuration named"forced fail - data bypass"is advanced with the result of synthesizing an optimal global configuration, which is used in machine translation work and a good effect is obtained.

Key Words logic programming, optimizing program configuration, memory space consumption, artificial intelligence.