国防科技大学学报
JOURNAL OF NATIONAL UNIVERSITY OF DEFENSE TECHNOLOGY
第10卷第2期 1988年6月 Vol.10.No.2

Prolog动态代码的两种语义及其实现方法

张 晨 曦
(计算机科学系)

摘 要 Prolog 数据库操作内部谓词是Prolog非逻辑成份的一个重要组成部分。为了实现Prolog程序的可移植性，这些内部谓词应有一致的操作语义。本文首先讨论了两种比较合理的语义，然后论述了这两种语义在WAM框架下的实现方法。

关键词 Prolog，数据库操作，动态代码，语义，WAM

1. 引 言

Prolog是一种基于一阶谓词逻辑的程序设计语言。它包含逻辑成分和非逻辑成份两部分。在非逻辑成份中，assert、retract等数据库操作内部谓词是一个重要的组成部分。它们在扩展Horn子句的功能上的作用是显著的[1,2]。它们使Prolog程序能动态地修改自身的规则和事实。这一点对于基于Prolog的知识库系统来说尤其重要。因此，有实用意义的Prolog系统中都应提供这些内部谓词。

然而，由于这些内部谓词的存在，Prolog过程代码可能在其执行过程中动态发生变化（为了便于讨论，我们称这些过程为动态过程）。这就存在如何确定动态过程代码的语义的问题。Prolog中对此没有规定。不同Prolog系统实现的语义差别很大。这使得含非逻辑成份的同一Prolog程序在不同系统中的执行结果可能不同。其原因主要是以往只注重这些内部谓词的实现的方便性，而很少考虑Prolog程序的可移植性。我们认为，有必要开展对这些内部谓词的语义（或动态过程的语义）的研究，以寻求比较合理、易于接受的一致性语义。本文首先对我们认为比较合理的两种语义进行了讨论，然后论述了在基于WAM[3]的编译型Prolog系统中，这两种语义的实现方法。

2. 动态过程代码的两种语义

动态过程的变化是以子句为基本单位。对于一个动态过程调用来说，确定动态过程的语义实际上就是确定其候选子句序列。本节中我们讨论几种可能的语义。这些语义的不同之处在于它们处理Prolog数据库动态变化的方法不同。

本文1987年7月1日收到

1） **第一种语义：固定候选子句序列（以下简称FCCS语义）**

按照这种语义，每一个过程调用的候选子句序列为在刚开始执行该过程调用的时刻，被调用过程所包含的子句序列。既使此后被调用过程发生了变化，该过程调用的候选子句序列也不改变。这相当于为每一个过程调用提供一个凝固了的过程副本。

举例：例1 test1:—assertz (test1), fail.

例2 test2:—assertz (test2), fail.

test2:—fail.

例3 test3:—fail.

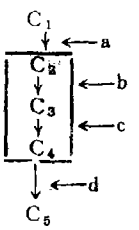
test3:—assertz (test3), fail.

按照上述语义，对test1/0，test2/0和test3/0的任何调用都将导致失败。

2） **第二种语义：动态候选子句序列（以下简称DCCS语义）**

按照这种语义，在一个过程调用的执行过程中，其候选子句序列随着被调用过程的变化而动态变化。数据库操作内部谓词的作用能立即反映到过程调用的执行中。当采用这种语义时，对上述例子test1/0(或test2/0, test3/0)的调用将导致在过程末尾插入事实test1.(或test2., test3.)，并立即加入该过程调用的后继执行。因此，最后的执行结果为成功。

当动态过程采用第二种语义时，retract和assert的联合使用可能会导致复杂的情况。这是因为若在一个子句序列的某个位置上先后删除多个子句，而后又插入一个新子句，则有多种插入方法。例如，对于图1所示的过程来说，若要在C_1和C_5之间插入一个子句，则a、b、c、d等都是可能的插入位置。虽然在这些不同位置上插入子句后，该过程的语义对于与C_1对应的过程调用（对于一个过程来说，正在执行的过程调用与子句特例是一一对应的）及以后进入该过程的过程调用来说是一致的。但对于与已删除子句C_2、C_3、C_4对应的过程调用来说，则是不一致的。因此，我们必须规定某个固定的插入位置。



□表示已删除子句

图 1

尽管在图1中a、b、c、d等4个位置上的插入都是同等地合理，但由于位置b和位置c与该过程的历史（即进行删除操作前的子句序列）有关，故难以实现在这些位置上的插入操作。而位置a和位置d上的插入则比较容易实现。据此，我们认为，当要把一个子句作为第i个子句插入过程时，比较合理的插入位置是紧接在第(i－1)个非删除子句之后（例如，图1中的a），或紧挨着原第i个非删除子句之前（例如，图1中的d）。由于前者在实现效率上略优于后者，故在我们的模型中采用了前者[11]。

此外，按照T.G.Lindholm的观点[4]，还有第三种语义：基于实现的语义。这是指除上述两种语义之外的任何语义。在采用这种语义的Prolog系统中，动态过程的具体行为往往取决于最容易实现的一种。因而是与具体Prolog系统的内部结构有关的。目前大多数Prolog系统就是这种状况。这增加了移植Prolog程序的困难。

3. FCCS语义和DCCS语义的实现方法

本节中我们讨论在基于WAM的Prolog系统中FCCS语义和DCCS语义的实现方法。

之所以选择WAM为框架，是因为WAM是目前世界上普遍采用的编译型 Prolog 执行模型[5,6,7,8,9]。所以，研究在这种框架下的实现方法具有一定的普遍意义。有关WAM的详细介绍见文献[3,9]。

1) 关于WAM的讨论

在WAM中，回溯是借助于选择点实现的。选择点中记录了在建立该选择点时，机器的状态以及下一个候选子句的地址。当调用一个含多个候选子句的过程时，就为之建立一个选择点。而当选择点被删除之后，就再也不可能回溯到该过程。WAM 中的确定性检测是一个重要的优化措施。它是指在进入过程的最后一个子句之前（由trust_me_else fail 指令）删除该过程的选择点。在大多数情况下，确定性检测能显著地减少局部栈的空间开销，并且能提高执行效率。但是，它的存在却使得WAM中动态代码的语义不一致。下面通过例子进一步说明。

当编译成WAM代码时，上述例子中过程test1/0、test2/0和test3/0的形式分别为

```
procedure test1/0
  <assertz(test1)的代码>
  fail

procedure test2/0
  try_me_else C2
  <assertz(test2)的代码>
  fail
C2:trust_me_else fail
  fail

procedure test3/0
  try_me_else C2
  fail
C2:trust_me_else fail
  <assertz(test3)的代码>
  fail
```

当执行过程test 2/0中的 assertz 以后，原过程变成图 2 的形式。由于这时相应的选择点还未被删除，因此当第一个子句和第二子句相继执行失败后，将回溯至新插入的子句。执行成功。这显然是符合 DCCS 语义。

```
procedure test2/0
  try_me_else C2
  <assertz(test2)的代码>
  fail
C2:retry_me_else C3
  fail
C3:trust_me_else fail
  proceed
```

图 2

但是对于对test3/0和test1/0的过程调用来说，由于在执行assertz时，该过程的选择点不存在（或者已被删除，或者本来就没有），故无法回溯至新插入的子句。即新插入子句是不可见的。这与FCCS语义相吻合。

对于带索引的动态过程来说，也同样存在上述语义不一致的问题。

由此可见，WAM只能实现前述第三种语义。为了实现FCCS语义或DCCS语义，必须对WAM进行扩充。

2) FCCS语义的实现方法

为了有效地实现 FCCS 语义，T.G.Lindholm 等人提出了“虚拟拷贝”法[4]。这种方法是为每一个过程调用提供一个过程代码副本。不过这个副本并不需要通过复制得到，而是通过给原过程的各子句加上不同的时间标记得到的。子句的时间标记是指区间（Birth, Death）。它指出了子句被添加到Prolog数据库的时间(Birth)和从 Prolog 数据库中删除的时间（Death）。

用“虚拟拷贝”法实现FCCS语义时，需在WAM中增设一个全局时钟和一个调用时钟寄存器CC。CC用于记录当前调用的调用时间。此外，还需用新的dynamic_else等指令代替WAM中的选择点指令。

当插入子句时，把新插入子句的Birth置为全局时钟的当前值，把其Death置为∞. 插入后，全局时钟加“1”。删除子句时，只需把其 Death 置为全局时钟的当前值，并把全局时钟加“1”。

动态过程的各子句经dynamic_else指令相链接。其中只有一部分是候选子句。仅当过程调用的调用时间落在子句的时间标记之内时，该子句才是候选子句。有关 dynamic_else指令的详细执行过程见文[4]。

3) DCCS语义的实现方法

在我们提出的非逻辑成份的执行模型中，动态过程采用的是DCCS 语义。本小节介绍其实现方法[10, 11]的基本思想。

(1) 对WAM的扩充

1°. 增设两条选择点指令：ctry_me_else LA, Next 和trust_fail。其中 Next 为下一子句地址，LA为Next的地址。ctry_me_else 指令用于替代动态过程中链接子句代码的选择点指令。trust_fail指令的功能是删除当前选择点并引起回溯。它总是动态过程的最后一条指令。

2°. 增设机器状态标志位 CTRYF。选择点中也相应地增加了一个域，用于保留CTRYF的值。每当回溯时，根据该域恢复 CTRYF的值。CTRYF的作用是指示ctry_me_else 指令的执行方式。

(2) 候选子句的确定

为实现Prolog数据库管理而设置的主要数据结构是过程表PT。Prolog程序的每一个过程在 PT 中都有相应的一项。PT项包含有关于过程的源子句和子句代码的信息。对于动态过程来说，有关其子句代码的信息包括：

PT.sadr——非删除子句链表中第一个子句的地址。

PT.ladr——非删除子句链表中最后一个子句的地址。

PT.predel——已删除子句链表表头指针。

与前述“虚拟拷贝”法的管理方法不同，在我们的方法中，每一个动态过程有两个子句链表：非删除子句链表和已删除子句链表。非删除子句链表中的各子句经ctry_me_else指令链接，它们都是候选子句。已删除子句链表包含已被删除但尚未回收其空间的子句。其中的子句是无序的。

(3) 子句的插入与删除

根据我们对DCCS语义的定义，子句的插入比较简单。只需通过修改相应 ctry_me_else 中的Next指针，把子句链入非删除子句链表中相应的位置。删除操作复杂些。它首先把要删除的子句从非删除子句链表中摘除，然后判断能否立即回收其空间。若不能，则把它链入已删除子句链表。由于不能破坏当前删除子句中ctry_me_else指令的 Next 指针，所以当把当前删除子句插入已删除子句链表时，需另建立新的链接指针。最后，还要把已删除子句链表中指向当前删除子句的指针改为指向该子句的下一个子句或 trust_

fail指令（若当前删除子句为过程的最后一个子句）。

(4) ctry_me_else LA, Next指令的执行过程

ctry_me_else 指令有两种执行方式：first 方式和非 first 方式。当 CTRYF=0 时，ctry_me_else按first方式执行。否则，按非first方式执行。在一个动态过程中，只有第一条 ctry_me_else 指令按 first 方式执行。在 first 方式下，ctry_me_else 指令的执行过程与WAM中的 try_me_else 类似。不同的是它在建立的选择点中填入下一子句的间接地址LA，而不是Next。而且，它还把CTRYF置为“1”，进入非 first 方式。在非 first 方式下，ctry_me_else相当于WAM的retry_me_else指令。

4. 结 束 语

我们认为，基于实现的语义是不可取的。只有FCCS语义和DCCS语义可以作为动态过程的语义。文中介绍了这两种语义的实现方法的基本思想。采用 DCCS 语义的动态代码的执行效率比采用FCCS语义的低。这是因为采用DCCS语义时，实现过程代码的索引比较困难，而且不能实现原 WAM 中的确定性检测。然而，DCCS 语义具有更好的实时性。而且，在大多数情况下，往往只有少数一部分过程需动态修改。所以，若使其它无需动态修改的过程仍保留Warren代码的形式，则仍能实现高效的代码。

感谢慈云桂教授的指导。

参 考 文 献

[1] Buettner, K.A.: Fast Decompilation of Compiled Prolog Clauses, Proceedings of the Third International Conference on Logic Programming, July, 1986

[2] Moss, C.: CUT&PASTE—defining the impure Primitives of Prolog, Proceedings of the Third International Conference on Logic Programming, July, 1986

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

[4] Lindholm, T.G. and O'keefe, R.A.: Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code, Proceedings of the Fourth International Conference on Logic Programming, 1987

[5] Bowen, K.A., et al.: The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler, Proceedings of the Third International Conference on Logic Programming, July, 1986

[6] Hermenegildo, M.V.: An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, Proceedings of the Third International Conference on Logic Programming, July, 1986

[7] Dobry, T.P., et al.: Performance Studies of a PROLOG Machine Architecture, Conference Proceedings of the 12th Annual International Symposium on Computer Architecture, June, 1985

[8] Nakazaki, R., et al.: Design of a High-speed Prolog Machine (HPM), Conference Proceedings of the 12th Annual International Symposium on Computer Architecture, June, 1985

[9] Tick, E.: An Overlapped Prolog Processor, Technical Note 308, SRI International, Oct., 1983

[10] Zhang Chenxi, Tzu Yungui, Li Liangliang and Hu Yunfa: An Approach to the Implementation of Prolog Code Database and Source Database in Compiler-Based Systems, Proceedings of the Fourth International Symposium on Logic Programming, 1987

[11] Zhang Chenxi and Tzu Yungui: Study of Mechanisms That Support the Implementation of the Non-Logical Components of Prolog in WAM-based Systems, To Appear in the Proceedings of the Frontiers in Computing Conference, 1987

Schemes for Implementing Two Choices of Semantics of Dynamic Prolog Code

Zhang Chenxi

Abstract

Database operation built-in predicates constitute an important part of the non-logical components of Prolog. Consistent operation semantics are to be defined for these built-ins in order to realize easy transportations of prolog programs among different systems. In this paper, two choices of reasonable semantics are first discussed. Then are described schemes to implement the semantics under the framework of the WAM.

KEY WORDS Prolog, Database operation, Dynamic code, Semantics, WAM