PROLOG语言中的一个逻辑问题

陈有祺（南开大学计算机与系统科学系）

PROLOG is an important logical programming language. However, there is some serious problem about the definition of the predicate"not", which would lead to the unequivalence of predicate P (x) and not (not (P (x))). In this paper, we have salved this problem through revising the definition of the predicate"not".

一、问题的提出

PROLOG语言是一种重要的逻辑型程序设计语言，它是以谓词演算为基础的，由于这种语言的独特风格和深远意义，已引起世界各国的广泛重视。但是，它还存在着若干不足之处，其中关于逻辑非(否定)的定义，就存在着严重的问题。

在micro-PROLOG中，对否定的定义是：

((NOT| X)X/FAIL) (1)

((NOT| X)) (2)

在PROLOG语言的其他版本中(如DEC-10 PROLOG)，也有类似的定义。在这种定义之下，NOT仅仅是逻辑非的一种近似，对于任何谓词P(X)来说，它不能使(NOT NOT P(X))永远等价于P(X)。我们可以用谓词append为例，来分析这一问题。

在micro-PROLOG中，如果我们定义了append，然后提出询问：

? ((append(a b)(c d)X)(PP X))

则输出的X值应为(a b c d)，但是当执行

? ((NOT NOT append(a b)(c d)X) (PP X)) (3)

时，却不能输出X的值，只能输出一个未定值的变量符号。其原因如下：

在(3)式中，首先应用到谓词名NOT，按照PROLOG语言的规定，应当去和定义NOT的第一个子句(即(1)式)匹配，此时将(NOT append(a b)(c d)X)对应于(1)式中的变元X，问题即化为顺次检验谓词序列

(NOT append (a b) (c d) X) /FAIL (4)

是否皆成立。在这个序列中，首先又遇到谓词名NOT，它又与(1)式相匹配。按规定，又归结为顺次检验谓词序列

(append (a b) (c d) X) /FAIL (5)

是否皆成立。由append的定义，此序列的第一个谓词可以满足，且X取值为(a b c d)。然后执行/(即cut)，它的效果是排除了对(2)式的应用。最后遇到FAIL时，导致(5)式不成立，因为没有别的选择，也就宣告了(4)式中第一个谓词失败。但(4)式属于上一层次，它尚未执行到/，故(3)式中第一个谓词仍可应用(2)式。由于(2)式是无条件成立的规则，它对任何变元都自然满足，故(3)式中第一个谓词也满足。尽管如此，可是因为未经过任何匹配工作，所以谓词append中的X没有得到任何值，当在继续执行(PP X)时，就只能输出一个未定值的变量符号了。这样，谓词(append (a b) (c d) X)与加上双NOT以后的效果就不相同。类似的例子还可以举出很多。

这种不等价性会导致PROLOG语言在某些逻辑程序设计中失效，因为它使得许多要重的逻辑公式不成立。这是必须解决也是可能解决的问题。

二、解决方法

因为在PROLOG语言中，NOT并非内部谓词，而是用PROLOG语言的其他功能定义的。我们完全可以修改这一定义，使之适合我们的要求。

在定义中要首先保证(NOT NOT P (X))与P

(X) 的等价性，这将放在第一个子句中加以检验，修改后NOT的定义为：

```
((NOT|[X|Y])(EQ X NOT)/Y)        (1′)
((NOT|X)X/FAIL)                  (2′)
((NOT|X))                        (3′)
```

其中第一个子句的意思是：把NOT后面的谓词中的谓词名字分离出来，看是否等于NOT，如果是，则不再考虑后面的子句(2′)和(3′)，直接执行双NOT之后的谓词Y即可。如果不是这种情况，继续应用子句（2′）和（3′），这和以前的定义完全一样。

对于其他常见的PROLOG语言版本，例如DEC-10 PROLOG，也可以用适当的内部谓词，来实现对not的上述修改：

```
not (X) :-X=..[Y|Z], Y=not, Z=
[W],
                call (W), ! .        (1″)
not (X) :-call (X), ! , fail.        (2″)
not (X) .                            (3″)
```

其中（2″）、（3″）两个子句也是原有的，第一个子句是为了处理双not而加上去的。这算符=..表示将左边的结构（函数结构）变为右边的表结构，目的是将谓词名分离出来。这里的"!"表示cut，其他几个谓词的作用很明显，不必多加解释了。

以上两种修改已在IBM-PC机上试验过，均达到预期的目的。

三、进一步的问题

上段提出的解决办法是针对在谓词之前直接出现双否定情形的，若在逻辑程序中实现谓词演算，则还要考虑各种复杂情形。例如，一个谓词的定义中包含了对另一谓词的否定运算，而当原来的谓词又被否定时将如何处理，会遇到相当复杂的问题。对于较简单的情形，我们仍容易解决，那就是在定义NOT的第三个子句中，再加上一些处理手段。例如对于

```
((T X)(NOT Q X))
```

这种形式的规则，当对谓词(T X)进行NOT运算时，也就是当遇到（NOT T X）这个谓词时，可以使其等价于谓词（Q X）。为此，我们再修改NOT的定义如下：

```
((NOT|[X|Y])(EQ X NOT)/Y)
((NOT|X)X/FAIL)
((NOT|X)(CL(X|Y))(EQ Y((Y1|Y2)))
    (EQ Y1 NOT)/Y2)
((NOT|X))
```

这里共用了四个子句，其安排的次序也有一定的道理，首先在第一个子句中考虑双NOT的情形，若不是，则考虑第二个子句，若X成立，则（NOT|X）不成立，也就不需要深入分析了。若X不成立，才需要进一步分析我们所设想的情形，这时应用第三个子句。在第三个子句中，首先将规则X的体（即条件部分）分离出来，然后检查体部分是否以NOT开头，若是则执行其后面的谓词（相当于我们所考虑的（Q X)），若不是这种情形，只好再求助于第四个子句，使其无条件成立了。

对于一般情形，需要针对所处理问题的复杂程度，由使用者增添适当的检验和处理方法，这里不能详尽讨论。总之，由于NOT是可由用户定义的谓词，用户可根据情况，使之尽量符合逻辑上的需要。

参考文献

[1] W.F.Clocksin, C.S.Mellish, "Programming in Prolog", Springer-Verlag, New York, 1981.

[2] 史忠植等译，"micro-PROLOG程序员使用手册"，中国科学院计算技术研究所十七室，1985.

[3] Robert Kowalski,"逻辑作为一种计算机语言"，鲁汉榕译，《计算机科学》，1986,No. 1。