PROLOG的指称语义和操作语义

Saumya K. Debray & Prateek Mishra

原摘要：PROLOG 程序的语义通常是根据一阶逻辑的模型论给出的。但是，这并不足以刻划PROLOG程序的计算特性。PROLOG的实现主要采用了以程序中子句和字面的正文出现次序为基础的顺序计算策略，并用到诸如"Cut"之类的非逻辑成份。在本文中提出了一种指称语义，它能刻划PROLOG的计算特性。我们给出了不含"Cut"的 PROLOG 的语义，然后，将其推广到含"Cut"的pROLOG。在两这情况下，我们分别证明了该语义与标准操作式解释程序的等价性。作为这一指称语义的应用，还证明了有关PROLOG程序变换的一些标准的"常用"定理的正确性。

1、引 言

想要描述程序设计语言PROLOG的语义就得解决一些错综复杂的问题。从一种角度来看，该问题是可以解决的，因为PROLOG程序就是用一阶逻辑的Horn-子句形式书写的一组陈述，从而可以用一阶逻辑的模型论[1,13]来描述PROLOG的语义。这常被称为PROLOG的说明性语义，即逻辑语义。从计算的角度来看，这种描述方式是不够的，因为它忽略了PROLOG程序性态的某些方面，其中包括使用顺序深度优先搜索策略的终止性和控制搜索的语言成份（例如cut）等问题。而且，在实际使用中，PROLOG似乎更象一种用置换序列来定义计算的语言，而不是刻划一些成真的断言公式的语言（参见〔10〕）。

在本文中，我们提出了PROLOG的一种指称语义[12]，它可以表达人们感兴趣的程序行为特性。该语义涉及到标准PROLOG执行系统所采用的顺序计算策略，能够刻划谓词符号cut的作用。其结果使得谓词的含义成为从置换到置换序列(有可能是无限的)的函数，而不再是一组基本的原子。这种指称语义的正确性是通过它与操作式PROLOG解释程序的等价性来证明的。

我们的工作是从需要证明PROLOG程序各种优化变换的正确性出发的。文章给出了作为这类交换[4,11]理论基础的常用定理的许多参考文献，许多定理涉及到PROLOG程序计算特性的推导，例如：由于cut的插入和移去所引起的终止性问题，以及将谓词视为置换序列转换符时的性态等。在第五部分中，用我们的语义给出了关于含有cut的程序变换的两条并不简单的定理的扼要证明。目前，正试图用指称语义验证复杂的静态分析模式（如模态推理和确定性推理）的正确性。虽然这些定理也可以通过采用计算归纳法或不动点归纳法从解释程序的性态中推出，但可能很困难，因为解释程序不能支持有关置换序列的推导过程，它是围绕着将其状态中的这类信息进行编码这一概念加以组织的。我们认为，我们所采用的指称语义方法既简明得多，又更加易于理解，

语义等价性的证明往往难以阅读，需要精心设计才稍易接受。在我们的证明中关键问题是使合成的指称语义与主要面向"逐次置换"处理的操作语义相一致。在不含cut的

情况下，通过提出一组定理支持解释程序状态模块化分解来协调两种语义描述。不幸的是，这种直接分解方法对于完整的PROLOG是不适用的，因为cut具有非局部的特性，相应的证明只好表达成将解释程序的状态与指称语义相联系的不变式（参见第四部分）。为了讨论的连贯性，这些定理的证明将在本文的附录中给出。

从指称角度研究逻辑程序设计语言语义的有关工作包括Frandsen的[5，6]和Jones与Mycroft的[7]等。Frandsen讨论了"纯逻辑式"程序，忽略了PROLOG计算时的顺序特性，这使得他的语义难以用来解释程序的一些执行性态。我们的工作与Jones和Mycroft的工作也有许多不同之处。我们的语义定义出于需要验证程序分析与变换方法之正确性，而他们的定义出于产生正确的PROLOG解释程序。与他们用一种特殊的标牌模拟cut的直接语义所不同的是，我们用拓展已有语义的方法，以更加直观的可取方式来模拟cut。而且，我们给出了操作语义和指称语义的等价性证明，并用指称语义验证了PROLOG程序两种并不很简单的优化转换的正确性。

2、 基础知识

2.1 SLD归结

PROLOG中的项可以是变量、常量和复合项$f(t_1, \cdots, t_n)$，其中f是n元函数符序，$t_i(1\leqslant i\leqslant n)$是项。变量、函数符号和谓词符号的集合分别用Var、Func和Pred来表示。项的集合将用Term来表示。置换是从Var到Term的幂等映射，它是除有很多点外的恒等映射。置换的集合用Subst表示。给定两个置换σ_1和σ_2，如果存在置换θ，使得$\sigma_2=\theta\circ\sigma_1$，则称$\sigma_1$比$\sigma_2$较为一般。如果对于两个项$t_1$和$t_2$存在置换$\sigma$，使得$\sigma(t_1)=\sigma(t_2)$，则称$t_1$和$t_2$是可合一的，这时，$\sigma$称为这两个项的合一置换。如果项$t_1$和$t_2$具有合一置换，则它们必定具有最一般的合一置换mgu(t_1, t_2)，并且，在不考虑变量重命名的影响时，它是唯一的。

一个PROLOG程序由一组谓词定义组成，而每个谓词定义又由一组子句组成。子句是由字面组成的序列，字面可以是原子目标或其否定形式。PROLOG的子句通常被限制为特定的Horn子句，它恰好含有一个肯定字面，称为子句的头，而其余的字面（如果存在的话）构成子句的体。只含有否定字面的子句称为否定子句或目标。我们将采用爱丁堡PROLOG的语法，将子句写成以下形式：

$$p:\!-q_1, \cdots, q_n.$$

它可以读作："如果q_1并且…并且q_n均为真，则p亦为真"。

PROLOG采用的计算策略是一种称为SLD归结[1]的较一般定理证明过程的特例。关于子句集合P的一个SLD推导过程是一个否定子句的序列$N_0, N_1, \cdots$，它使得对于每个i，如果有$N_i=a_1, \cdots, a_{n_i}$，则

$N_{i+1}=\theta(a_1, \cdots, a_{k-1}, (b_1, \cdots, b_m), a_{k+1}, \cdots, a_{n_i})$将满足：

（1）$1\leqslant k\leqslant n_i$；

（2）$b:\!-b_1, \cdots, b_m$是P中的一个子句，并且已被适当重命名，使其不含有与N_i中同名的变量；

（3）θ是a_k和b的最一般合一置换。

程序$\langle P, G\rangle$由一组谓词定义P和一个目标子句G组成。给定$\langle P, G\rangle$，其SLD树中的节点将用目标子句来标识，根节点标识成G。如果SLD树中的某个节点N被标识为$a_1, \cdots, a_m$，那么，可以选择其中的一个原子a_k（$1\leqslant k\leqslant m$），对于P中经过适当变量重命名后其头部可以与a_k合一的每条子句$a':\!-b'$，都将为节点N产生一个以合一后的结果目标为标识的子节点。

2.2 顺序PROLOG

一棵SLD树代表着一组从q出发的SLD推导过程。程序的执行可以看成在相应的SLD树上搜索反驳的遍历过程，反驳即为终

止于空子句的路径。可以证明，如果一个程序的某个SLD树中存在以空子句标识的节点，则该程序的任何SLD树中都将存在以空子句标识的节点[1]。因此，搜索任何一棵SLD树就足以解决是否存在反驳的问题。PROLOG的执行策略对应于SLD树的深度优先搜索，并且，总是在目标中选取最左边的原子，而树中节点的子节点是按照相应子句在程序中的正文次序加以排列的。在这一模式下，对原子目标的调用通过依次选择适当的子句并与其合一来完成，从而得出程序在描述上一致而计算时却有所不同。例如，对于如下定义的倒序函数：

```
append ([], X, X).
append ([A|B], Y, [A|D]); —append(B, Y, D).
rev1 ([], []).
rev1 ([A|X], Y); —rev1 (X, Z), append (Z, [A], Y).
rev2 ([], []).
rev2 ([A|X], Y); —append(Z, [A], Y), rev2 (X, Z).
```

读者可以验证，目标rev1 ([1, 2],X) 在X为[2, 1]时终止，而目标 rev2 ([1, 2],X) 将产生一个解，然后发散下去。

2.3 PROLOG程序的可观察性态

PROLOG被引起广泛注意的一个方面是其能够用置换来定义计算。求解目标时所产生的置换的简单语义定义就是在SLD树中从一个节点到另一个节点搜索空子句的同时产生相应的置换。但这忽略了PROLOG计算过程中唯一可观察的是其高层目标中所含变量这一事实。在我们的语义定义中，采用了这种可观察性概念，放弃了置换带来的一些不必要信息。例如，在我们的描述中，下述谓词foo1和foo2具有相同的含义：

```
foo1 (X, Y); —foo (X, Z).
foo ([], []).
foo2 ([], Y).
```

2.4 "cut"的语义

在顺序遍历一棵SLD树时对于遇到cut所采取的动作可以解释如下。我们称考虑中被激活的子句的目标为父目标，当在该子句中遇到cut时，将放弃父目标以下的SLD子树，而该父目标已由解释程序遍历过。事实上，这意味着：（1）cut左边的原子的所有可能置换，除第一个外，都将被放弃；（2）按正文次序在当前子句以后的其它子句可能获得的置换也将全部放弃。我们的语义在模拟cut的行为时显式地采用了这种观点。

2.5 定义与表示方法

给定一可数集S，$S_\perp$代表集合$S\cup\{\perp\}$。S的有限和ω-无限元素序列所构成的集合将表示为S^∞，而S的有限元素序列之集合表示为S^*。第一个元素为a，从第二个元素起以后的序列为L的序列有时写作$a::L$；空序列记为nil。给定两个序列S_1和S_2，它们的并置用$S_1 \Diamond S_2$来表示。设$\perp$代表无定义的序列，则$\Diamond$可以定义为：

$$\perp \Diamond L=\perp;$$
$$nil \Diamond L=L;$$
$$(a::L_1) \Diamond L_2=a::(L_1 \Diamond L_2).$$

对象$a_1, \cdots, a_n$的n元组表示为$(a_1, \cdots, a_n)$。当无需区分元组中的元素时，有时可以用上横写法来表示该元组，例如：$\bar{t}$。

序列上的一种共同的操作是将函数作用于序列的每一分量，并依次汇集其结果。这用"‖"来表示：

$$f\|\perp=\perp;$$
$$f\|nil=nil;$$
$$f\|(a::L)=f(a)::f\|L.$$

最后，我们定义"汇集"操作符，它汇集将以序列为值的函数作用于序列各分量之结果。设$f: S_\perp \longrightarrow S_\perp^\infty$是一个这样的函数，$s\in S_\perp^\infty$是由$S$中对象组成的序列。我们希望将$f$依次作用于$S$的每一分量，并使其结果序列并置后输出。该运算表示为$\bigcirc$：

$$f\bigcirc\perp=\perp;$$
$$f\bigcirc nil=nil;$$
$$f\bigcirc(a::L)=f(a) \Diamond (a\bigcirc L).$$

PROLOG的语法范畴如下：

变量::=Var;
函数符号::=Func;
谓词符号::=Pred;
项::=Term;
项::=变量+函数符号（项$_1$, …, 项$_n$）;
原子::=谓词符号（项$_1$, …, 项$_n$）;
字面::=原子+'¬';
目标::=字面*;
子句::=原子:—目标.

前面谈到置换是从Var到Term的幂等且几乎恒等的映射。对于给定的置换σ，常希望了解我们所感兴趣的一些有限变量集上σ的作用效果。为此，定义了有限置换集FSubst，它是在有限定义域和值域上从Var到Term的映射。自然要求有限置换所"涉及"的任一变量都在其定义域之内。让$Vars$: Term$\longrightarrow 2^{Var}$代表产生项（可自然推广到字面和子句）中出现的变量之集合的函数，于是，有以下的定义：

定义 有限置换ϕ是从变量的有限集V到项的有限集T的幂等映射，它使得对V中的任一v，$vars(\phi(v)) \subseteq V$成立。

有限置换ϕ的定义域记为$dom(\phi)$。给定有限置换ϕ和含有$dom(\phi)$的变量集V，ϕ到V的扩展$\phi\uparrow V$是以V为定义域的映射，其中在$dom(\phi)$上，它与ϕ相同，而在$V-dom(\phi)$上为恒等映射。对于两个有限置换θ和σ，它们的复合$\theta\circ\sigma$是以$dom(\theta) \cup dom(\sigma)$为定义域的有限置换。在极限情况，有限置换到所有变量之集合Var上的扩展也将产生一个置换。在上下文无歧义时，我们有时将不区分有限置换与置换。有限置换集对于有限复合运算是封闭的，但对于无限复合则不然。

函数$unify$: Term×Term$\longrightarrow$(FSubst$\cup\{fail\}$)在可合一时返回两个项的最一般合一置换，否则，返回标牌$fail$。这可以自然推广到原子。函数$rename$: Term$\times 2^{Var}\longrightarrow$Term以项$t$和变量集$V$为输入，产生一个与$t$相等的项$t'$，而$t'$中的所有变量被一致地替换成不在$V$中出现的"新"变量。（假设变量集Var是用自然数来进行枚举的可数无限集$v_0, v_1, \cdots$，于是，对于任一有限变量集V，可以有效地找出V中具有最大下标的元素v_k（$v_k \in V$），并可进一步找出变量$v_m (m>k)$，使得$v_m \notin V$。我们不再进一步细化$rename$的特性。）当引用$rename(t, V)$时，便可以说"t关于V已被重命名"。函数$rename$也可自然地推广到字面和子句。

在下面的讨论中，我们所感兴趣的是有限置换的有限和无限序列以及无定义的序列所组成的集合$FSubst^{\infty}_{\perp}$。FSubst中的元素用小写希腊字母$\sigma, \theta, \phi, \cdots$来表示，而$FSubst^{\infty}_{\perp}$中的元素则用大写希腊字母$\Sigma, \Theta, \Phi, \cdots$来表示。以$V$为定义域的恒等有限置换记作$\in_V$。

设V为目标G中出现的变量之集合，如果θ是"回答G的置换"，则我们所感兴趣的只是θ与V中变量有关的部分。对此还有一种说法是：既然θ是回答的置换，那么，$\theta(G)$的任一实例在该程序中都是可反驳的，这意味着对任一在V中元素部分与θ一致的置换θ'，$\theta'(G)$在该程序中也是可反驳的，从而θ中"感兴趣"的部分就是涉及到V的部分。我们用投影函数$\downarrow$：FSubst$\times 2^{Var}\longrightarrow$FSubst将有限置换限制到某一变量集上。

定义 给定有限置换ϕ：$V\to T$和集合$W \subseteq V$，ϕ在W上的投影函数$\phi\downarrow W$是以$V_1 = W \cup (\cup_{v\in W} vars(\phi(v)))$为定义域的有限置换，它使得对$V_1$中的任一$v$，$(\phi\downarrow W)(v) = if\ v\in W\ then\ \phi(v)\ else\ v$。

3、 不含CUT的PROLOG语义

3.1 不含Cut的PROLOG指称语义

谓词定义的语义是将项映射为有限置换序列的一个函数。有限置换的有限和无限序列加上最小对象⊥所组成的集合$FSubst^{\infty}_{\perp}$在标准前束序关系下具有完全偏序性质。谓词符号通过使用环境以通常的方式与其含义相联系。直观地看，环境将标识符名映射为从项到置换序列的函数。但是，事实表明：由

于技术上的考虑（在归结前使变量重命名），在执行过程中也需要对所遇到的变量集进行动态传播。这通过以参量的形式传递合适的有限置换并根据该置换的定义域进行重命名来完成。因此，我们有：

$Env = Predicate \to Term \to FSubst \to FSubst_{\perp}^{\perp}$

环境一般用 ρ，$\rho_0, \rho_1, \cdots$ 来表示。在某个环境中将标识符 Id 与对象 Obj 相联系的操作记为 $Id \leftarrow Obj$。给定环境 ρ，$\rho[p \leftarrow f]$ 代表除将 p 的值置为 f 外，其余部分均与 ρ 相同的环境。

定义程序指称语义的过程可视为两个部分。首先，必须将各个谓词符号的语义用其组成部分的语义来定义；这用语义函数 $D[\![\,]\!]$ 来完成，它将子句序列的语义定义为关于环境的一个函数。给定子句序列 C^* 和环境 ρ，$D[\![C^*]\!]\rho$ 表示在环境 ρ 中精化子句序列 C^* 后所得到的新环境。其次，必须根据有关的定义刻划目标的语义。这用语义函数 $G[\![\,]\!]$ 来表示：给定字面序列 G，环境 ρ 和输入置换 σ，$G[\![G]\!]\rho\sigma$ 表示在环境 ρ 中计算 $\sigma(G)$ 后所得到的输出置换序列。事实上，这两部分不是互相独立的，因为第一部分涉及到定义子句的语义，而定义子句体的语义要用到语义函数 $G[\![\,]\!]$。我们还将采用两个辅助函数：$C[\![\,]\!]$ 给出单个子句的语义，$L[\![\,]\!]$ 给出子句体中单个字面的语义。

函数 $D[\![\,]\!]$ 以子句序列和环境为输入，输出一个新的环境，其抽象形式为：

$D[\![\,]\!]: Clause^* \to Env \to Env$。

$D[\![\,]\!]$ 的直观意义是：给定其它谓词含义的一些知识（由输入环境刻划），我们可以通过在该环境中求解所给的子句序列来扩充关于那些谓词含义的知识；这将产生一个比输入环境更加"定义良好"的新环境。子句的空序列不定义任何东西，于是，我们有：

$D[\![nil]\!]\rho = \lambda x \lambda y \lambda z \cdot nil$

其中 x 代表输入谓词名，y 代表项的元组，而 z 则为一个置换。

另一方面，对于子句序列 $c_0::C$，根据 PROLOG 的计算策略，用子句 c_0 所能产生的一切解答都将在尝试 C 的解答前产生。因此，如果仅用 cl_0 所获得的置换序列为 s_0 且用 C 获得的为 s_1，则结果的置换序列为 $s_0 \Diamond s_1$。较为形式化的表示是：

$D[\![c_0::C]\!]\rho = \lambda x \lambda y \lambda z \cdot [\,(C[\![c_0]\!]\rho x y z) \Diamond (D[\![C]\!]\rho x y z)\,]$ 其中 x、y 和 z 的含义同前所述。

最后，单个子句的含义是一个以项的元组 $\overline{t}$ 和有限置换 σ 为输入，以置换序列为输出的函数。其基本思想是：首先使子句中的变量一致地重命名，使所用的新变量均不在 $dom(\sigma)$ 中出现，并使 $\sigma(\overline{t})$ 与在重命名后的子句头出现的项的元组合一；然后，在将结果置换作用于它之后求解子句的体；最后，将求解子句体所得置换序列中的每一置换投影到调用该子句时的变量集 $dom(\sigma)$，并返回结果置换序列。之所以要进行这一投影是因为这里所"感兴趣"的变量是 $dom(\sigma)$ 的元素，而求解子句体时所返回的序列中的元素可能置换非 $dom(\sigma)$ 中的变量。正如前面所讨论的，这不是需要的成份，应该投影掉这类多余的成份。因此，函数 $C[\![\,]\!]$ 定义为：

$$
\begin{aligned}
&C[\![\,p(\overline{T_0}) :- G_0]\!]\rho = \\
&\quad P \leftarrow \lambda s \lambda \sigma \cdot [\,let\ \ (p(\overline{T_1}) :- G_1) = rename((p(\overline{T_0}) :- G_0), dom(\sigma)); \\
&\qquad \theta = unify(\sigma(\overline{s}), \overline{T_1}); \\
&\qquad in \\
&\qquad if\theta = fail\ then\ nil\ else\ (\lambda x \cdot x \downarrow dom(\sigma)) \\
&\qquad\qquad \| G[\![G_1]\!]\rho(\theta \cdot \sigma :: nil)\,].
\end{aligned}
$$

函数 G 输入字面的序列、定义各种谓词语义的环境和一个置换的序列，然后，返回另一个置换的序列，其抽象形式为：

$G[\![\,]\!]: Goal \to Env \to FSubst_{\perp}^{\perp} \to FSubst_{\perp}^{\perp}$

空目标代表一次成功的计算，这时，返回的置换流即为其输入流：

$G[\![nil]\!]\rho\theta = \theta$。

字面序列 $L::G$ 的含义是先用输入置换

流求解L，再将其结果置换"流入"G所得的置换流。这意味着对于给定的环境ρ和置换σ，目标$L::G$的语义是：

$$\mathbf{G}[\![L::G]\!]\rho\theta=\mathbf{G}[\![G]\!]\rho\ (L[\![L]\!]\rho\theta)$$

单个字面$P(\overline{T})$在环境ρ中关于有限置换σ的语义是在该环境中对输入流中的每一置换调用相应的谓词，再将其结果置换流依次并置后所得的置换序列θ。这可用函数$\mathbf{L}[\![\,]\!]$来刻划：

$$\mathbf{L}[\![p(\overline{T})]\!]\rho\theta=(\rho\ (p)\quad(\overline{T}))\bigcirc\theta.$$

上述所有语义函数见图1。

$\mathbf{D}[\![\,]\!]: Clause^*\to Env\to Env$.

(D1.1) $\mathbf{D}[\![nil]\!]\rho=\lambda x\lambda y\lambda z.nil$

(D1.2) $\mathbf{D}[\![c_0::C]\!]\rho=\lambda x\lambda y\lambda z.[(C[\![c_0]\!]\rho xy\ z)\ \diamond\ (\mathbf{D}[\![C]\!]\rho xyz)]$

$\mathbf{C}[\![\,]\!]: Clause\to Env\to Env$.

(C1.1) $C[\![p(\overline{T_0}):-G_0]\!]\rho=$

$p\leftarrow\lambda\overline{S}\lambda\sigma.[let\ (p\ (\overline{T_1}):-G_1)=rename\ ((p\ (\overline{T_0}):-G_0),\ dom(\sigma));$

$\theta=unify\ (\sigma(\overline{S}),\ \overline{T_1});$

in

$if\ \theta=fail\ then\ nil\ else\ (\lambda x.x\downarrow dom(\sigma))\|\mathbf{G}[\![G_1]\!]\rho\ (\theta\cdot\sigma::nil)]$.

$\mathbf{G}[\![\,]\!]: Literal^*\to Env\to \mathbf{FSubst}_\perp^\infty\to\mathbf{FSubst}_\perp^\infty$.

(G1.1) $\mathbf{G}[\![nil]\!]\rho\theta=\theta$.

(G1.2) $\mathbf{G}[\![L::G]\!]\rho\theta=\mathbf{G}[\![G]\!]\rho\ (\mathbf{L}[\![L]\!]\rho\theta)$.

$\mathbf{L}[\![\,]\!]: Literal\longrightarrow Env\longrightarrow\mathbf{FSubst}_\perp^\infty\longrightarrow\mathbf{FSubst}_\perp^\infty$.

(L1.1) $\mathbf{L}[\![p(\overline{T})]\!]\rho\theta=(\rho\ (p)\ (\overline{T}))\bigcirc\theta$.

图1 无cut的PROLOG语义函数$\mathbf{D}[\![\,]\!]$和$\mathbf{G}[\![\,]\!]$

设"$fix\ x.f(x)$"表示$\bigsqcup_n f^n(\perp)$，则由子句序列P和目标G组成的程序$\langle P,\ G\rangle$的语义可以定义为$\mathbf{G}[\![G]\!]\rho_0\ (\in_V::nil)$，其中$\rho_0$实际上就是$fix\rho.\mathbf{D}[\![P]\!]\rho$，而$V=vart(G)$。为了精确定义极限环境$\rho_0$，需要区分代表不终止的底置换序列$\perp$和底环境$\perp_{Env}$，后者将程序中有定义的每个谓词映射为$\lambda x\lambda y.\perp$，而将无定义的任何谓词映射为$\lambda x\lambda y.nil$，因为传统PROLOG系统的执行在遇到程序中无定义的谓词时失败。因此，程序P所反映的极限环境被定义为$\rho_0=\bigsqcup_n\mathbf{D}^n[\![P]\!]\perp_{Env}$。由于函数$\mathbf{G}$和$\mathbf{D}$都是用连续函数的复合和（在$\mathbf{G}$的情况下）简单递归来定义的，所以它们也是连续函数（参见[12]），并且，不动点ρ_0一定存在。

3.2 不含Cut的PROLOG操作语义

操作语义是按照以下述方式遍历"最左"SLD树并完成相应的状态转换的解释程序给出的：

在任何一点，总是选择当前目标的最左原子进行归结。

在归结某个字面时，按照程序中的出现次序顺序尝试有关的子句。

如果在某一点回溯失败，回溯至最靠近的选择点并用另一种可能的选择恢复执行过程。

该执行策略实际上对应于程序最左SLD树的深度优先回溯搜索。

解释程序的状态由反映计算状态的一个动态栈和包含某个程序的子句表组成。栈是由记录组成的表，每个记录刻划从SLD树根到某个树中节点的一条路径。它是一个三元组$\langle Frame\ List,\ Subst,\ Clauses\rangle$，其中$Frame\ List$是由$Frame$组成的表，每个$Frame$代表一个待求解的目标；$Subst$是当前的有限置换，它随着求解过程逐步被构造出来；$Clauses$是程序的尾部，由在求解相应目标最左字面时尚未尝试过的子句组成。每个$Frame$是一个目标与变量集的对，这里的目标可以是用户的查询，或者是用来求解查询目标的某个子句的右部，而变量集Var_p则是该目标所关心的父节点变量集。在计算该目标时产生的任一置换，在返回前都要投影到Var_p中的变量上。

$Frame::=\langle Atom^*,\ Var_p\rangle$

$FrameList::=nil\mid Frame::FrameList$

$Stack::=nil\mid\langle Frame\ List, Subst, Clause^*\rangle::Stack$

$State::=Stack$。

该解释程序由函数$interp$来定义，它将状态映射为（可能是无限的）置换序列。解释程序围绕着程序中定义谓词的子句序列运行：

$interp$: $State \times Clause^* \longrightarrow FSubst^{\infty}$。

当动态栈为空时运行结束：

$interp\ (nil,\ P) = nil$。

在当前框架中的待求解目标变为空时，便找到一个解，因此，返回当前的置换，并使执行过程回溯，以寻找其它的解：

$interp\ (\langle nil,\ \phi,\ C\rangle :: St,\ P) = \phi :: interp\ (St,\ P)$。

另一方面，如果已不存在可与（非空）目标匹配的程序子句，则执行失败，并引起回溯：

$interp\ (\langle F,\ \phi,\ nil\rangle :: St,\ P) = interp\ (St,\ P)$，若$F \neq nil$。对于非空目标和非空子句序列，解释程序试图用这些子句求解目标中最左的字面，这对应于调用该谓词所定义的过程。经过子句变量的适当重命名以后，如果第一条子句的头与该最左字面可合一，则在框架表中增加一项刻划由相应的子句体所组成的子目标的框架。原目标和尚未尝试的子句被保留在栈中，以便在回溯时可以寻找其它的解：

$interp\ (\langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ (H_0 :- B_0) :: C\rangle :: St_0,\ P) = interp\ (\langle F_2 :: F_1 :: F_0,\ \theta \cdot \phi,\ P\rangle :: St_1,\ P)$

其中

$H_1 :- B_1 = rename\ ((H_0 :- B_0),\ dom(\phi))$;

$\theta = unify(\phi(L), H_1)$,　$(\neq fail)$;

$V_P' = dom\ (\phi)$;

$F_2 = \langle B_1,\ V_P'\rangle$; $F_1 = \langle G,\ V_P\rangle$;

$St_1 = \langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ C\rangle :: St_0$。

另一方面，如果第一条子句的头与最左字面不可合一，则放弃该子句，而对其余的子句重复上述过程：

$interp\ (\langle Framelist,\ \phi,\ (H_0 :- B_0) :: C\rangle :: st_0,\ P) = interp\ (\langle Framelist, \phi,\ C\rangle :: St_0,\ P)$。

当框架中的目标变为空时（也许在当前的向前执行路径上还存在其它待解目标），适当地投影当前置换，并用投影后的置换和剩下的框架表继续进行计算。这对应于从过程的一次返回。由于进行返回的计算中可能引入新变量，在以后的向前计算中需要重命名时，必须考虑投影后的置换中出现的这些变量的影响：

$interp\ (\langle nil,\ V_P\rangle :: F_0,\ \phi,\ C\rangle :: St,\ P) = interp\ (\langle F_0,\ \phi \downarrow V_P,\ P\rangle :: St,\ P)$

对定义有关谓词的子句序列C求解目标G的过程可以用如下表达式来定义：

$interp(\langle\langle G,\ V\rangle :: nil,\ \in_V,\ P\rangle :: nil\rangle,\ P)$

其中$V = vars\ (G)$，$\in_V$是定义域V中的有限恒等置换。对解释程序在图2中作了概述。

$interp$: $State \times Clause^* \longrightarrow Subst^{\infty}$。

(I1.1) $interp\ (nil,\ P) = nil$.

(I1.2) $interp\ (\langle nil,\ \phi,\ C\rangle :: St,\ P) = \phi :: interp\ (St,\ P)$.

(I1.3) $interp\ (\langle F_0 :: F_1,\ \phi,\ nil\rangle :: St,\ P) = interp\ (St,\ P)$.

(I1.4) $interp\ (\langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ (H_0 :- B_0) :: C\rangle :: St_0,\ P) = interp\ (\langle F_2 :: F_1 :: F_0,\ \theta \cdot \phi,\ P\rangle :: St_1,\ P)$

其中

$H_1 :- B_1 = rename\ ((H_0 :- B_0),\ dom(\varphi))$;

$\theta = unify\ (\phi(L),\ H_1)\ (\neq fail)$;

$V_P' = dom(\phi)$;

$F_2 = \langle B_1,\ V_P'\rangle$; $F_1 = \langle G,\ V_P\rangle$;

$St_1 = \langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ C\rangle :: St_0$。

(I1.5) $interp\ (\langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ (H_0 :- B_0) :: C\rangle :: St_0,\ P) = interp\ (\langle L :: G,\ V_P\rangle :: F_0,\ \phi,\ C\rangle :: St_0,\ P)$

其中

$H_1 :- B_1 = rename\ ((H_0 :- B_0),\ dom\ (\phi))$;

$unify\ (\phi\ (L),\ H_1) = fail$.

(I1.6) $interp\ (\langle nil,\ V_P\rangle :: F_0,\ \phi,\ C\rangle :: St,\ P) = interp\ (\langle F_0,\ \phi \downarrow V_P,\ P\rangle :: St, P)$.

图2　不含Cut的PROLOG抽象解释程序

3.3 指称语义和操作语义的等价性

这一节中我们将证明：上述抽象解释程序赋予谓词、原子和目标的含义实际上就是前述语义函数所给的语义。但是，在证明等价性之前，需要一些有关抽象解释程序性态的结构引理。解释程序定义了如下的函数：

$interp: State \times Clause^* \longrightarrow Subst^*$

而且，由于它是按照一组互斥情况定义的，并且，对于每种情况，各表达式右部由以下成份构作而成：(i) 连续函数，(ii) 符号 *interp*，以及(iii)由(i)和(ii)中成份复合而成，因此，*interp*所定义的是连续泛函，可以采用不动点归纳法推导其最小不动点。

第一条引理的含义是：对于解释程序的动态栈$F::St$，其组成部分F刻划了当前的向前计算，而栈的其余部分St则代表了尚未搜索过的SLD树的剩余部分。

引理3.1 对任何栈框架F和栈St，均有：

$$interp\ (F::St,\ P) \equiv interp(F::nil,\ P)\ \Diamond interp\ (St,\ P)$$

证明：对*interp*进行不动点归纳即可。 □

下两条引理的含义是：解释程序每次以一个元组为计算单位，并按照从左到右的次序求解目标中的字面。首先，我们将证明：该元组按照栈中当前向前计算成份所含框架后进先出（LIFO）的次序进行处理：

引理3.2 对任一框架F、框架表F_1、置换ϕ、程序P和子句表C，均有：

$$interp(\langle F::F_1,\ \phi,\ C\rangle::nil,\ P) \equiv [\lambda\theta\cdot interp\ (\langle F_1,\ \theta,\ P\rangle::nil,\ P)]\bigcirc interp\ (\langle F::nil,\ \phi,\ C\rangle::nil,\ P)$$

证明：对*interp*进行不动点归纳即可。 □

下一引理指出：同一框架中字面的计算按照从左到右的次序每次处理一个元组。

引理3.3 对任一目标$L::G$、变量集V_P、框架表F、置换ϕ、子句表C和程序P，均有：

$$interp\ (\langle\langle L::G,\ V_P\rangle::F,\ \phi,\ C\rangle::nil, P) \equiv [\lambda\theta\cdot interp\ (\langle\langle G,\ V_P\rangle::F,\ \theta,\ P\rangle::nil,\ P)]\bigcirc interp\ (\langle\langle L::nil,\ dom(\phi)\rangle::nil,\ \phi,\ C\rangle::nil,\ P)$$

证明：对*interp*进行不动点归纳即可。 □

最后一条引理涉及到解释程序从一次调用返回时如何投影掉多余的置换，它说明这一工作可以分两步进行。这是证明定理**3.1**所需的纯技术性引理。

引理3.4 对任一目标G、置换θ和ϕ、栈成份St和以C为尾部的程序P，均有：

$$interp\ (\langle\langle G,\ dom\ (\phi)\rangle::nil,\ \theta\circ\phi,\ C\rangle::St,\ P) = (\lambda\sigma\cdot\sigma\downarrow dom\ (\sigma))\ \|\ interp\ (\langle\langle G,\ dom\ (\theta\circ\phi)\rangle::nil,\ \theta\circ\phi,\ C\rangle::St,\ P)$$

证明：用图2中的规则进行化简，并注意到$dom(\phi)\subseteq dom\ (\theta\circ\phi)$即可。 □

最后，考察一下失败对置换序列的影响。失败的目标将返回空置换序列。正如前节所述，对于目标$L::G$，如果字面L失败，则整个目标失败，这可以用下述引理来表达：

引理 3.5 对任一目标 G 和环境 ρ，$G[[G]]\rho nil = nil$。

证明：对G进行结构归纳即可。 □

现在我们可以来证明本节的主要结果了。

定理3.1 对任一目标 G、字面L、以C为尾部的程序P和置换流θ，均有：

$$G[[G]]\rho_0\ (L[[L]]\ D[[C]]\rho_0\theta) \equiv \lambda\sigma\cdot interp(\langle\langle L::G,\ dom\ (\sigma)\rangle::nil,\ \sigma,\ C\rangle::nil,\ P)\bigcirc\theta$$

证明：对$G[[]]$、$D[[]]$和*interp*进行不动点归纳即可。 □

推论（指称语义和操作语义的等价性）。对任一目标G、程序P和置换流θ，均有：

$$G[[G]]\rho_0\theta \equiv \lambda\sigma.interp\ (\langle\langle G,\ dom(\sigma)\rangle::nil,\ \sigma,\ P\rangle::nil,\ P)\bigcirc\theta$$

（未完待续）

〔戴 敏 译自J. LOGIC PROGRAMMING 1988:5:61-91 仲 源校〕