Prolog程序部分计算的研究与实现*

邓铁清 胡运发

(长沙工学院)

RESEARCH AND IMPLEMENTATION OF THE PARTIAL EVALUATION OF PROLOG PROGRAMS

Deng Tieqing and Hu Yunfa

(Changsha Institute of Technology)

ABSTRACT

Partial evaluation is an important program transformation method and compiling optimization technique. Prolog programs are specially suited to be partially evaluated. At present, several principal models and special-purpose tools of the partial evaluation of Prolog programs have been developed in the world, but there are still following problems in these works: (1) There is lack of systematic understanding for the basic principle and characteristics of the partial evaluation of Prolog programs; (2) Both methods which are used to detect loops in logic programs don't bring the termination problem in partial evaluation to a final solution; (3) The treatment of builtins in Prolog is imperfect and there are many implicit semantic errors in it; (4) The efficiency of some algorithms for partial evaluation is very low; (5) The partial evaluators now available are confined to their application fields, respectively, and they are lack of generality. Based on our working practice in researching and developing a practical source-level partial evaluator in GKD-Prolog compiler [14], this paper discusses the partial evaluation of programs in pure Prolog, the detection of loops in logic programs and the treatment of builtins in full Prolog entirely and systematically.

摘 要

部分计算是一种重要的程序变换方法和编译优化技术，Prolog程序特别适

* 1989 年 8 月 9 日收到，1989 年 10 月 14 日定稿。

合于部分计算。目前，国际上已开发了几个Prolog程序部分计算的原理模型和专用工具[5][6][9][12]，但其中存在以下若干问题: (1) 关于Prolog程序部分计算的基本原理和特征缺乏系统的认识; (2) 现有的两种检测逻辑程序中循环的方法，并没有最后解决部分计算的终止性问题; (3) 关于Prolog中内部谓词的处理不够完善，而且其中还隐含了许多语义错误; (4) 部分计算算法相当低效; (5) 现有的部分计算器局限于各自的应用领域，缺乏通用性。本文结合我们研制GKD-Prolog编译系统[14]中一个实用源级部分计算器的工作实践，全面、系统地讨论了纯Prolog的部分计算、逻辑程序的循环检测以及全Prolog的内部谓词处理。

§1. 引 言

部分计算的思想最早出现在1952年由Kleene所著的《元数学引论》[1]一书中，它的一般定义是: 设有函数 $f(x_1,\cdots,x_n)$，且当 $x_1=a_1,\cdots,x_k=a_k(1\le k\le n)$ 时，$f(x_1,\cdots,x_n)=f'(x_{k+1},\cdots,x_n)$，则称 f' 为 f 在 $x_i=a_i(1\le i\le k)$ 下的部分计算。

限于当时的理论研究，部分计算的思想起初并没有引起人们的重视和应用。随着计算机科学的兴起，特别是对程序变换、编译方法、软件生产自动化的研究和开发，程序的部分计算迅速获得发展和应用[2][3][10][11]。从数学的角度看，程序的部分计算是一个从 $PXD\to(PXD)$ 的映射 pe，并且，若 $pe(p,d)=(p',d')$，则 $val(p,d)=val(p',d')$，其中，val 是程序集为P、数据集为D的语言的语义函数。保义性是程序的部分计算的必要条件。

Prolog程序特别适合于部分计算，这是因为:

(1) 在一个程序被提交执行以前，已知目标或子句中部分常量或数据结构信息，因此，Prolog程序的部分计算相对来说能更加有效地进行。

(2) Prolog系统的一致化机制强有力地支持了目标中已知信息的自顶向下的传播、子句中已知信息的自底向上的传播、以及中间部分信息的双向传播。

(3) 程序的数据的统一，为采用一体化的部分计算方法提供了方便性和灵活性。

(4) Prolog语言所具有的明确的操作语义，比较容易保证部分计算的保义性。

近年来，国际上许多知名学者分别从提高Prolog元、目标级混合程序和Prolog—DBMS耦合程序的执行速度，以及自动产生编译器等诸方面，相继对Prolog程序的部分计算进行了一些研究，并开发了几个原理模型和专用工具[5][6][9][12]。但其中存在以下几个方面的问题:

(1) 关于Prolog程序部分计算的基本原理和特征缺乏系统性的认识。

(2) 现有的两种检测循环的方法，并没有最后解决Prolog程序部分计算的终止性问题。

(3) 关于Prolog 中内部谓词的处理不完善，而且，其中还隐含了许多语义错误。

(4) 部分计算方法的低效性。这包含两方面的含义：其一，部分计算算法本身的低效性；其二，Prolog 程序经部分计算后所获得的残余程序的低效性。

(5) 现有的部分计算器局限于各自的应用领域，缺乏通用性。

本文结合我们研制GKD-PROLOG 编译系统[14] 中源级部分计算器的工作实践，系统、全面地讨论了纯Prolog 的部分计算、逻辑程序的循环检测、以及全Prolog 的内部谓词处理。

§2. 纯Prolog 的部分计算

2.1 基本原理

保义性是部分计算的必要条件。在Prolog 系统中，问题求解所依据的语义函数是一种基于自上而下、从左到右和深度优先的SLD 归结，因此，要保证部分计算所得到的残余程序与源程序的语义等价，部分计算的操作语义一定也是受限的SLD 归结。

问题求解与部分计算的区别在于：前者求解时的各种输入信息已经具备，因此，输出将给出明确的回答；后者计算时的输入信息尚不完全，因此，目前还不足以求出一个具体的解，输出可能仍是一段程序(称之为残余程序)，但是，相对源程序来说，这段程序的执行速度一般是提高了。

为避免与逻辑程序设计中的某些概念相混淆，下面首先定义几个我们在Prolog 程序的部分计算中常用的术语。

定义1 设部分计算的子目标为P，如果源程序中存在子句$P':-Q_1,\cdots,Q_n$，且P、P'可一致化，则称P 是可展开的；否则是不可展开的。

一般地，若子目标P 不可展开，P 的部分计算将被挂起或延迟。

定义2 设部分计算的目标为：$-A_1,\cdots,A_i,\cdots,A_n$，其中，$A_1,\cdots,A_{i-1}$不可展开，且存在子句$A_i':-B_1,\cdots,B_m$，使得$A_i$可展开。若$A_i$、$A_i'$的最一般一致化取代为$\theta$，则称：$-(A_1,\cdots,A_{i-1},B_1,\cdots,B_m,A_{i+1},\cdots,A_n)\theta$为目标：$-A_1,\cdots,A_i,\cdots,A_n$相对子目标$A_i$的一次展开。$(A_1,\cdots,A_{i-1})\theta$称为$A_i$展开时的向后一致化。$(B_1,\cdots,B_m,A_{i+1},\cdots,A_n)\theta$称为$A_i$展开时的向前一致化。

根据受限的SLD 归结原理，Prolog 程序的部分计算过程可描述为：

(1) 按从左到右的顺序，选择目标G 的最左文字P 为当前子目标。

(2) 若P 不可展开，则P 被挂起，P 的部分计算结果是P 本身。下一个子目标为G 中P 的后继文字。

(3) 若P 可展开，按自上而下的原则，选择源程序中第一条其头可与P 一致化的子句$P':-Q_1,\cdots,Q_m$，展开G 中的P。按深度优先策略，下一个子目标为$(Q_1)\theta$，其中，θ为P、P′的最一般一致化取代。

(4) 重复过程(2)和(3), 直至所有子目标均被部分计算为止。

(5) 回溯, 求出目标G的其它部分计算结果。

此外, 在Prolog程序的部分计算中, 已知的信息除目标或子句中的数据或结构之外, 还有众多的逻辑常量true和fail, 对它们的处理可明显简化计算结果和切断计算过程。

(6) 设部分计算的目标为: $-A_1,\cdots,A_i,\cdots,A_n$, 其中, $A_1,\cdots,A_{i-1}$ 是已求出的部分结果, A_i 为当前子目标。若 A_i 为逻辑常量true, 则 A_i 展开的结果是: $A_1,\cdots,A_{i-1},A_{i+1},\cdots,A_n$, 且 A_{i+1} 为下一个子目标。若 A_i 为逻辑常量fail, 则终止计算过程, 且最后结果为: $-A_1,\cdots,A_{i-1},fail$。

例1 设地图着色问题的Prolog程序是:

```
color(A,B,C,D,E) :-next(A,B),next(A,C),next(A,D),next(B,C),
                   next(C,D),next(B,E),next(C,E),next(D,E).
next(X,Y(:-next1(X,Y).
next(X,Y) :-next2(X,Y).

next1(green,red).
next1(green,yellow).
next1(green,blue).
next1(red,blue).
next1(red,yellow).
next1(blue,yellow)

next2(X,Y) :-next1(Y,X).
```

给定目标: $-next(X,Y)$, 部分计算后的残余程序是:

```
next(green,red).        next(red,green).
next(green,yellow).     next(yellow,green).
next(green,blue).       next(blue,green).
next(red,blue).         next(blue,red).
next(red,yellow).       next(yellow,red).
next(blue,yellow).      next(yellow,blue).
```

若给定目标: $-color(A,B,C,D,E)$, 部分计算后的残余程序将给出地图着色问题的全部72个解。

2.2 实现算法

如前所述, 在Prolog程序的部分计算中, 输入和输出均是Prolog程序。因此, 从逻辑程序设计的角度看, 部分计算的实现过程具有下列特点:

(1) 程序作为数据进行处理。

(2) 数据作为程序进行求值。

(3) 计算结果当作数据进行输出。

由此可见，用Prolog描述部分计算算法需要运用元级程序设计技术[13]。为提高算法的时空效率，我们采取了元级与目标级相结合的程序设计技术，其核心算法是：

```
peval(P,(A,B),Suf,Acc,Out):-!,
                peval(P,A,(B,Suf),Acc,Out).
peval(P,(A,B),Suf,Acc,Out):-!,
                (peval(P,A,Suf),Acc,Out);
                (peval(P,B,Suf),Acc,Out)).
peval(P,true,Suf,Acc,Out):-!,
                peval_rest(P,Suf,Acc,Out).
peval(_,fail,_,Acc,(Acc,fail));-!.
                peval(P,nil,Suf,Acc,out):-!
                peval_rest(P,Suf,Acc,Out).
peval(P,A,Suf,Acc,Out):-
                clause(A,B),
                peval(P,B,Suf,Acc,Out).
peval_rest(_,nil,Out,Out):-!.
peval_rest(P,(A,B),Acc,Out):-
                peval(P,A,B,Acc,Out).
```

在谓词$peval(P,Subgoal,Suf,Acc,Out)$中，P是部分计算的源目标，Subgoal为当前子目标，Suf表示待处理的后续子目标，Acc是已求得的部分结果，Out为最终结果。

谓词peval/5的含义是：

(1) 若Subgoal是多个子目标的"与"，则优先求解最左边的那个子目标。

(2) 若Subgoal是多个子目标的"或"，则分别求解各子目标。

(3) 如果Subgoal为逻辑常量true，则消除之。

(4) 如果Subgoal为逻辑常量fail，则终止计算过程，并输出最终结果。

(5) 若Subgoal为空nil，则继续处理后续子目标。

(6) 如果源程序中存在某个子句，其头能与Subgoal匹配，则用它的体来替换Subgoal，然后求解之。

2.3. 主要特征

经过分析和研究，我们认为，与传统程序的部分计算相比，Prolog程序的部分计算具有下列主要特征：

(1) 子目标是通过展开求解的。即如果源程序中存在与该子目标匹配的子句，则用此子句的体替换该子目标并求解之。

(2) 值的传播是借助Prolog系统的一致化机制自动完成的。即如果子目标含已知信息，则通过向前一致化自顶向下传播；如果子句含常量或数据结构，则通过向后一致化自底向上传播。在谓词peval/5的最后一个子句中，设$clause(A,B)$成功的最一般一致化取代为θ，则向前一致化是$(B,Suf)\theta$，向后一致化为$(P,Acc)\theta$。

(3) 全解是利用回溯获得的。即残余程序相对源程序的完整语义，是通过谓词peval/5的最后一个子句中，对$clause(A,B)$的强制性失败而得到的。

§3. 逻辑程序的循环检测

3.1 循环问题及其处理

例2 设判断表的成员关系的Prolog程序是:

$$member(E,[E|\_]).$$
$$member(E,[\_|L]):-member(E,L).$$

给定目标: $-member(E,L)$, 部分计算后的残余程序是:

$$member(E,[E|\_]).$$
$$member(E,[\_,E|\_]).$$
$$member(E,[\_,\_,E|\_]).$$
$$\cdots$$

显然，上述计算过程是不会终止的。

实际上，Prolog程序的部分计算是一种对目标的SLD树进行搜索的过程。如果SLD树上的所有通路均是成功或者有穷失败，我们说部分计算的过程最终是会终止的。然而，递归是Prolog程序的重要特色，因此，许多目标的SLD树都会在不同程度上含有无穷的通路，如果我们在部分计算的过程中，不禁止那些导致无穷通路的循环子目标的展开，求解过程必将不能终止。因此，关于循环问题的处理是:

设部分计算的目标为: $-A_1,\cdots,A_i,\cdots,A_n$, A_i是当前子目标，且A_i的展开将造成循环，则A_i的部分计算是:

(1) 在该目标的部分计算中，A_i作为不可展开的子目标进行处理。

(2) 在该目标的计算终目之后，继续求解新的且未曾部分计算的目标: $-A_i$，新目标的部分计算结果最后构成源目标部分计算后残余程序的另一部分。

按以上循环处理方法，例2中目标: $-member(E,L)$经部分计算后的残余程序与源程序完全相同。

3.2. 循环的滞后与超前检测

循环的正确处理依赖于对循环的有效检测。目前，国际上关于Prolog程序的循环检测方法共有两种，即滞后[6][9]与超前[12]检测，并分别对应于下面两个关于循环的不同定义:

定义3 设已部分计算的子目标序列为$A_1,\cdots,A_{i-1}$，当前子目标为A_i。若存在某个$A_j(1\le j\le i-1)$，满足:

(1) A_j与A_i同构，即除变元命名不同之外，其余均相同；或者

(2) A_j较A_i更一般，即A_i是A_j的实例。则称A_i与A_j构成(滞后)循环，A_i称为(滞后)循环子目标。

定义4 设部分计算的当前子目标是A，且A按SLD树展开的序列为$A_1,A_2\cdots$。若存在某个$A_i(i\ge 1)$，A、A_i可一致化，则称A与A_i构成(超前)循环，A称为(超前)循环子目标。

例3 设有下列按抽象数据类型定义表拼接的Prolog程序:

$$
\begin{aligned}
&append\_list(L1,L2,L3):-append(L1,L2,L3).\\
&empty([\ ]).\\
&cons(E,L,[E|L]).\\
&append(L,R,R):-empty(L).\\
&append(L1,L2,L3):-cons(E,Part1,L1),\\
&\qquad\qquad\qquad append(Part1,L2,Part3),\\
&\qquad\qquad\qquad cons(E,Part3,L3).
\end{aligned}
$$

给定目标: $-append\_list(L1,L2,L3)$，若按照滞后检测及其循环处理，部分计算后的残余程序是:

$$
\begin{aligned}
&append\_list([\ ],L,L).\\
&append\_list([E|L1],L2,[E|L3]):-append(L1,L2,L3).
\end{aligned}
$$

$$
\begin{aligned}
&append([\ ],L,L):\\
&append([E|L1],L2,[E|L3]):-append(L1,L2,L3).
\end{aligned}
$$

如果按照超前检测及其循环处理，则部分计算后的残余程序是:

$$
\begin{aligned}
&append\_list(L1,L2,L3):-append(L1,L2,L3).
\end{aligned}
$$

$$
\begin{aligned}
&append([\ ],L,L).\\
&append([E|L1],L2,[E|L3]):-append(L1,L2,L3).
\end{aligned}
$$

3.3 循环检测的性能分析和评价

经过大量的研究与实践，我们认为，一种循环检测算法的优劣将主要取决于下面三个性能指标的好坏:

(1) 可靠性。如果一种算法能检测出部分计算中子目标所有不安全的展开，则称之为可靠的；反之为不可靠的。一个子目标的展开称为不安全的，是指此次展开将在目标的SLD树上构成一条带圈的回路，或者进入一条无穷通路。

(2) 准确性。如果一种算法能保证部分计算中子目标所有安全的展开均不是循环，则称之为准确的；反之为不准确的。一个子目标的展开称为安全的，是指此次展开在目标的SLD树上还不足以构成带圈的回路，或者尚未进入无穷通路。

(3) 高效性。如果一种算法本身的时空性能很好，则称之为高效的；反之为低效的。

在上述三个性能指标中，可靠性是根本，是解决部分计算的终止性的基础；准确性是保证部分计算所得到的残余程序的有效性的前提；高效性是算法本身实用性的一个重要标志。

经过典型程序的测试和理论分析，关于循环的滞后和超前检测方法，我们可以得出如下结论:

(1) 两种检测方法均是不可靠的。两者都只识别了两类非终止性：其一，子目标的展开已经(滞后检测时)或者即将(超前检测时)在目标的SLD树上构成一条带圈的回路；其二，子目标中由变量不断例示为无穷项而在目标的SLD树上即将进入一条无穷通路。但是，我们发现，在Prolog程序的部分计算中，还可能存在第三类非终止性：子目标中的变量由于是一个计数器或累加器(如差表)，而在目标的SLD树上形成一条无穷通路，典型程序参见皇后问题的Prolog描述[7]。

(2) 滞后检测的准确性总的来说要优于超前检测，这是因为：滞后检测时，子目标中的各变量已在先前的部分计算中获得了充分的特例化；而且，滞后检测中的条件较之超前检测更弱，有利于象$append([X1,X2,X3],Y,Z)$中第一参数的结构信息的传播。但是，在一个子目标的展开确实会进入循环时，滞后检测比超前检测发现得晚，因此，对该子目标的部分计算必将导致一次冗余的展开(见前一节)。

(3) 滞后检测的效率显然要比超前检测高。

基于以上讨论，我们提出并实现了一个新的循环检测方法——混合检测。

3.4. 循环检测的一种新方法——混合检测

混合检测的基本思想是:

(1) 立足点为超前检测，并通过设置一个计数器检测第三类非终止性。

(2) 检测条件采用滞后检测中的定义。

(3) 在超前检测中实施一些可能的计算。

用 Prolog 描述是:

```
loop(L) :- assertz($$$$$(L)), retract($$$$$(L1)),
           clause(L1, Body), loop(L, Body, nil, [ ]).
loop(L, (Subgoal, Body), Suf, Unfoldings) :- !,
           Suf == nil -> (Body0 = Body; Body0 = (Body, Suf)),
           loop(L, Subgoal, Body0, Unfoldings).
loop(L, (Subgoal; Body), Suf, Unfoldings) :- !,
           (loop(L, Subgoal, Suf, Unfoldings);
           loop(L, Body, Suf, Unfoldings)).
loop(_, nil, _, _) :- !, fail.
loop(L, Subgoal, _, Unfoldings) :-
           (not(not(instance_of(Subgoal, L)));
           length(Unfoldings, Breadth), Breadth > 15).
loop(L, Subgoal, Suf, Unfoldings) :-
           system(Subgoal), evaluable(Subgoal), !,
           call(Subgoal), loop(L, Suf, nil, Unfoldings).
loop(L, Subgoal, Suf, Unfoldings) :-
           (system(Subgoal);
           not(not(instance_literal(Subgoal, Unfoldings)))), !,
           loop(L, Suf, nil, Unfoldings).
loop(L, Subgoal, Suf, Unfoldings) :-
           assertz($$$$$(Subgoal)), retract($$$$$(Subgoal1)),
           clause(Subgoal1, Body),
           loop(L, Body, Suf, [Subgoal|Unfoldings]).

instance_literal(L, [H|_]) :- instance_of(L, H).
instance_literal(L, [_|Ls]) :- instance_literal(L, Ls).
```

其中，*loop*(*L*) 判部分计算的当前子目标L在未来的展开过程中是否遇到循环。

在谓词*loop*(*L*, *Subgoal*, *Suf*, *Unfoldings*), L 是部分计算的当前子目标; *Subgoal* 是 L 展开式中的某个子目标; *Suf* 是此展开式中 *Subgoal* 的后续子目标; *Unfoldings* 是 *Subgoal* 之前展开的各子目标的栈。*loop*/4 的含义是显然的，其中第四个子句体的第一个"或"(;) 分量检测前两类非终止性，第二个"或"分量检测第三类非终止性。

instance_of(T_1, T_2) 判 T_1 是否为 T_2 的实例(含同构)。

evaluable(*Subgoal*) 判 *Subgoal* 目前是否可求值。

§4. 全 Prolog 的内部谓词处理[15]

一般地，对于内部谓词的部分计算，当条件充分时(如比较型谓词的两个参数均不含变量)，则先求出解，然后根据解的结果(true 或 fail)进行展开。若条件不充分(如 var/1 的参数为变量)，或者谓词本身在部分计算时并不能求解(如 I/O 或数据库操作)，则按不可展开进行处理。

例4 设有下列求阶乘的Prolog程序:

$$factorial(O) : -write(1).$$
$$factorial(N) : -N > O, write(N), write('*'),$$
$$N1 \ is \ N - 1, factorial(N1).$$

给定目标: factorial (5)，部分计算后的残余程序是:

$$factorial(5) : - \ write(5), write('*'),$$
$$- \ write(4), write('*'),$$
$$- \ write(3), write('*'),$$
$$- \ write(2), write('*'),$$
$$- \ write(1).$$

然而，内部谓词大多不是真正意义上的关系，它们有其特殊的含义，甚至对SLD 树的搜索也会产生影响。因此，在纯Prolog 的部分计算的基础上，对全Prolog 的内部谓词必须专门处理。

4.1. 向后一致化

一般地，向后一致化可能改变某些变量的约束关系，这对那些对变量值的类型较为敏感的内部谓词来说，部分计算有时会导致求解结果不正确。

例5 设有下列Prolog 程序:

$$p(X) : -var(X), q(X)$$
$$q(a).$$

给定目标: $-p(X)$，按纯Prolog 的部分计算残余程序是:

$$p(a) : -var(a).$$

对于询问 $?-p(X)$，源程序的回答是 $X = a$; 而残余程序的回答却是NO (即不成功)。

设部分计算的目标为: $-A_1, \cdots, A_i, \cdots, A_n$，其中，$A_i$ 为当前子目标，且是不可展开的内部谓词，$X_1, \cdots, X_m$ 是出现在 A_i 中的所有不能被向后一致化的变量，则关于 A_i 的部分计算是:

(1) A_i 的计算结果置为$(A_i, X_1 = Y_1, \cdots, X_m = Y_m)$, 其中, $Y_1, \cdots, Y_m$ 是不在目标中出现的m个新变量。

(2) 分别用$Y_j(1 \le j \le m)$替换$A_{i+1}, \cdots, A_n$中关于X_j的所有出现, 并形成新的待部分计算的子目标序列$A'_{i+1}, \cdots, A'_n$。

(3) 置当前子目标为A'_{i+1}, 继续计算过程。

(4) 对求解结果中的每个Y_j, 如果Y_j未被特例化, 则X_j与Y_j进行一致化。

(5) 消去求解结果中的恒等比较。

按上述处理方法, 例5的残余程序是:

$$p(X) : -var(X), X = a.$$

4.2 向前一致化

有些内部谓词中, 变元的作用域是局部的, 它的值只是因测试的需要而发生临时的变化, 一旦脱离这些谓词, 该变元就恢复成进入时的状态。因此, 变元的临时约束值是不能向后一致化或向前一致化的。

例6 设有下列Prolog程序:

$$p(X) : -not(not(q(X)), write(X).$$
$$q(a).$$

给定目标: $-p(X)$, 按纯*Prolog*的部分计算, 残余程序是:

$$p(a) : -write(a).$$

对于询问? $-p(X)$, 源程序的输出是变元X; 而残余程序的输出是常量a。

设部分计算的目标为: $-A_1, \cdots, A_i, A_{i+1}, \cdots, A_n$, 其中, A_i是当前子目标, $X_1, \cdots, X_m$是出现在A_i中的所有不能被向前一致化的变元, 则A_i的部分计算是:

(1) 在源程序中插入新子句$newP(X_1, \cdots, X_m) : -A_i$。

(2) 部分计算目标: $-newP(X_1, \cdots, X_m)$。

(3) 在目标: $-newP(X_1, \cdots, X_m)$的残余程序中, 收集头为$newP(X_1, \cdots, X_m)$的子句体送*Bodies*, 体间的关系是"或"(;)。

(4) A_i的计算结果置为*Bodies*。

(5) 置当前子目标为A_{i+1}, 继续计算过程。

按上述处理方法, 例6的残余程序是:

$$p(X) : -not(not(X = a)), write(X).$$

4.3 副作用

当子目标可与多个子句匹配时，若采用回溯求解，某些内部谓词的副作用可能发生变化。

例7. 设有下列Prolog 程序:

$$\begin{aligned} &Program(X,Y):-write(title),\\ &\qquad\qquad database(X,Y),nl,\\ &\qquad\qquad write(X),tab(5),write(Y).\\ &database(a,b).\\ &database(a,c). \end{aligned}$$

给定目标: $-program(X,Y)$, 按纯 Prolog 的部分计算，残余程序是:

$$\begin{aligned} &program(a,b):-write(title),nl,\\ &\qquad\qquad write(a),tab(5),write(b).\\ &program(a,c):-write(title),nl,\\ &\qquad\qquad write(a),tab(5),write(c). \end{aligned}$$

对于询问 $?-program(X,Y)$, 源程序的输出中只有一个title; 而残余程序的输出中却有两个title。

设部分计算的目标为: $-A_1,\cdots,A_i,A_{i+1},\cdots,A_n$, 其中，$A_i$ 是当前子目标，且是带副作用的内部谓词，$X_1,\cdots,X_m$ 是同时出现在 $(A_1,\cdots,A_i)$ 和 $(A_{i+1},\cdots,A_n)$ 中的所有变元。则 A_i 的部分计算是:

(1) 在源程序中插入新子句 $newP(X_1,\cdots,X_m):-A_{i+1},\cdots,A_n$。

(2) 部分计算目标: $-newP(X_1,\cdots,X_m)$。

(3) 在目标: $-newP(X_1,\cdots,X_m)$ 的残余程序中，收集头为 $newP(X_1,\cdots,X_m)$ 的子句体送 *Bodies*, 体间的关系是"或" (;)。

(4) 置 A_i 的计算结果为 A_i, $(A_{i+1},\cdots,A_n)$ 的计算结果为 Bodies。

(5) 回溯，求源目标的其它部分计算结果。

按上述处理方法，例7 的残余程序是:

$$\begin{aligned} &program(X,Y):-write(title),\\ &\qquad\qquad (X=a,Y=b,nl,write(a),tab(5),write(b);\\ &\qquad\qquad X=a,Y=c,nl,write(a),tab(5),write(c)). \end{aligned}$$

4.4 !

按纯Prolog 的部分计算，"!" 可能导致如下两个问题:

(1) 向后一致化有时改变了"!" 以前的变元的约束关系，可能引起错误。

(2) 若与当前子目标匹配的子句体中含"!", 此次展开可能改变原"!" 的作用范围。

例8 设有下列Prolog 程序:

$$p(X):-q(X),r(X),s(X).$$
$$p(\cdots):-\cdots$$

$$r(X):-t(X),!,u(X).$$

若$q(X),s(X),t(X)$ 和$u(X)$ 均不可展开(如循环谓词), 按纯Prolog 的部分计算, 目标: $-p(X)$ 的残余程序是:

$$p(X):-q(X),t(X),!,u(X),s(X).$$
$$p(\cdots):-(\cdots)$$

这里, 由于$r(X)$ 的展开, "!" 的作用范围已了生了如下变化: (1) "!" 切断了$q(X)$ 可能存在的其它选择; 但在源程序中, 即使$u(X)$ 失败导致$r(X)$ 不成功, $q(X)$ 还是可回溯的。(2)"!" 对谓词p/1 作了剪枝; 而在源程序中, "!" 的剪枝只作用于谓词r/1, 对谓词p/1 不产生影响。

"!" 引起的第(1) 个问题已在本章第一节中解决, 关于第(2) 个问题, 我们的处理方法是:

设部分计算的目标为: $-A_1,\cdots,A_i,A_{i+1},\cdots,A_n$, 其中, A_i 是当前子目标, 且存在子句C, C 的头与A_i 可一致化, C 的体中含"!"。则A_i 的部分计算是:

(1) 部分计算目标: $-A_i$, 设该结果为子句集$\{C_j|1\le j\le m\}$, C_j 的体记作B_j。

(2) 尽可能消除$\{C_j|1\le j\le m\}$ 中的"! "。消除原则有两种: 其一, 若子句C_j 的体B_j 的最左文字为"! ", 则首先消除$\{C_j|1\le j\le m\}$ 中其头与C_j 的头同构的所有后续子句$C_k(j<k\le m)$, 设新的子句集为$\{C_j|1\le j\le l\}$, 其中l 为剩余子句个数; 然后, 若$j=l$, 即C_j 成为新的求解结果中的最后一个子句, 则从体B_j 中抹去最左文字"!"。其二, 采用与文献[8][9] 相同的方法, 尽可能用$if\cdots then\cdots else$ 替代子句集$\{C_j|1\le j\le l\}$ 中的"!", 设该结果是$\{C'_j|1\le j\le t\}$, 其中t 为最后的剩余子句个数。

(3) 若子句C'_j 的体$B'_j(1\le j\le t)$ 均不含! , 则A_i 的部分计算是分别用$B'_1\cdots,B'_t$ 去展开。

(4) 否则, A_i 按不可展开处理, 且子名组$C'_1\cdots,C'_t$ 作为目标的残余程序的一部分进行输出。

§5. 结论

本文关于Prolog 程序部分计算的主要贡献是:

(1) 在较为严格的定义和类形式描述的基础上, 全面、深入地阐述了Prolog 程序部分计算的基本原理和主要特征。

(2) 通过对现有的两种检测逻辑程序中循环方法的分析和研究，提出了一种新的循环检测——混合检测，有效地解决了部分计算的终止性问题。

(3) 系统地分析了全 Prolog 中内部谓词对纯 Prolog 的部分计算的影响，并提出了一系列有效的处理方法，从而消除了[6][9][12]中存在的语义错误。

(4) 文中提出并实现的许多算法尽可能采用了元、目标级相结合的程序设计技术，因而具有较好的时空性能。

(5) 文中关于 Prolog 程序的部分计算面向 Prolog 的全集，而不是仅仅处理 Prolog 的某个子集，因而具有通用性。

参考文献

[1] Kleene, S. C., Introduction to Meta-Mathematics. North-Holland Publishing Co., Amsterdam, 1952.

[2] Futamura, Y., Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler systems, Computers, Controls, Vol. 2, No. 5, 1971.

[3] Komorowski, H. J., Partial Evaluation as a Means for Inferencing Data Structure in an Applicative Language: A Theory and Implementation in the Case of Prolog. Ninth ACM Symp. on Principles of Programming Languages, Albuquer, New Mexico, 1982.

[4] Kahn, K. M., A Partial Evaluator of Lisp Written in a Prolog Intended to Be Applied to the Prolog and Itself Which in Turn Is Intended to Be Given to Itself together with the Prolog to Produce a Prolog Compiler. Technical Report, UPNAIL, Uppsala University, Sweden, 1982.

[5] Kahn, K. M. and Carlsson, M., The compilation of Prolog Programs without the Use of Prolog Compilers. International Conference on FGCS, Tokyo, Japan, 1984.

[6] Venken, R., A PROLOG Meta-Interpreter for Partial Evaluation and Its Application to Source Transformation and Query Optimization. in ECAI-84.

[7] Sterling, L. and Shapiro, E., The Art of Prolog, MIT Press, 1986.

[8] O'Keefe, R. A., On the Treatment of Cuts in Prolog Source Level Tools. Proc. Int'l Symp. on Logic Programming, Boston, Mass, 1985.

[9] Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming. in Information Processing 86, Dublin, Irelannd, North-Holland, 1986.

[10] Bjrner, D., Ershov, A. P. and Jones, N. D. eds, Workshop on Partial Evaluation and Mixed Computation, GL. Avernaees, Denmark, 1987.

[11] International Conference on FGCS, Tokyo, Japan, 1988.

[12] Li Lei, Design and Implementation of a Partial Evaluator for (Almost) Full Prolog Programs, ICO 89 Quebec, Ganada 29, 1989.

[13] 苏金树，邓铁清，吴泉源，元级程序设计技术研究，《知识工程进展1988》，中国地质大学出版社，1988, 12。

[14] 胡运发，高洪奎，胡子昂，卢肇川，邓铁清，逻辑语言及其环境合成系统GKD-PROLOG/WICK，全国第四届逻辑程序设计会议，1989。

[15] 邓铁清，胡运发，高洪奎，Prolog 中非逻辑成分的部分计算，全国第四届逻辑程序设计会议，1989。