PROLOG-DBMS耦合：半解释半编译的混合式方法

李 磊 G.H. Moll J. Kouloumdjian*)

本文介绍了一种混合式Prolog-DBMS耦合技术，它既能保持解释方法和编译方法的优点，又能克服其缺点。这种技术使得随时使用关系数据库算子(比如连接算子)成为可能，同时避免了无用数据(即归结过程中不用的数据)的加载，最后，"历史"管理机制保证了同一数据不会被多次加载。

1. 引 言

在实现知识库管理的诸多方法中，那些基于逻辑程序设计与数据库联合的方法近几年为了建造原型系统而得到了广泛研究。这些方法的不同之处在于其中Prolog-DBMS耦合的性质和实现耦合的技术两个方面。也就是，如果Prolog能直接访问数据库，则称为紧耦合；如果使用了查询语言，则称为"松耦合"。

另一个区别来自用户访问数据的透明程度。用户可以把系统看成仿佛数据库中的全部事实都在Prolog工作空间的主存里，事实检索和管理完全由系统完成，这是全透明；用户也可通过一些起数据操纵语言作用（查询、更改等）的内部谓词访问数据库。

这些技术在逻辑部分与数据管理部分的通讯策略上也可能有差异：数据库可以一次只给出一个元组（当推理机提出询问时），也可以一次给出一大块元组。

许多研究者都考虑了递归，因为这在当前的应用中是一个实际问题，而且，在使用Prolog策略时不考虑数据库关系，则递归的处理所需的时间和/或空间代价很大。

无论哪种方法，所面临的主要问题都是基于耦合的系统效率问题，怎样才能使一个系统不但代价合理（无须对Prolog解释器及DBMS的源码进行修改）、用户友好（数据分布透明），而且效率高。这些原则是里昂一大（Univ. of Lyon 1）的Epsilon工程的基础。第一个原型系统是用C-Prolog标准解释器和一个以SQL作查询语言的关系数据库Informix建造的。系统中，数据的重新划分是完全透明的。优化策略依赖于用户请求(Prolog目标)，是以一种Prolog程序编译技术——部分求值技术为基础的。但是，在这个原型系统中，递归问题没有得到满意的解决。

本文旨在介绍这一原型系统的改进版，尤其是在递归的场合效率有所提高。所选用的方法结合了部分求值（称为编译的静态步骤）和一个解释步骤以保证只有有用的元组（对推理机而言）才被提取，而且在Prolog工作空间中一给定元组只被提取一次。

2. SLD归结与元程序设计

这一节介绍一些元程序设计中要用到的记号和定义。

设P和Q是两个一阶谓词，我们说{P,Q}是可合一的，如果存在一个置换θ使得：

$$P\theta = Q\theta$$

我们说"P包含了Q"当且仅当存在一个

*) 第一位作者（吉林大学计算机系），后二位作者(Université Claude Bernard Lyon 1. France.)

置换θ使得：

$$P\theta = Q$$

记为Q←P，并称P比Q更广义。

一个程序就是一组Horn子句。如果一个谓词P只出现于子句的体中，而不出现在子句的头部，则称之为"终极谓词"。

搜索一棵SLD树的策略可以由下述三个选择函数来定义：

·在当前子目标中选择句节（Literal）

·选择子句执行

·选择下一步要处理的SLD树结点。这一选择是带根本性的，它定义了回溯策略。

我们可以回想一下，在Prolog中，句节选择是"自左至右"，子句选择是按程序中的顺序，而回溯策略则是"深度优先"。

采用元程序设计技术可以实现一种不同于上述策略的策略。

设$G_k = C_1, C_2, \cdots, C_k$是SLD归结中的当前子目标，它挂在归结树的结点$N_c$上。我们可以给每个三元组$(G_k, C_i, N_c)$一个值，称为子目标$G_k$中的句节$C_i$在结点$N_c$的取值，记为：

$V(G_k, C_i, N_c)$：布尔值，定义为：$V(G_k, C_i, N_c)$="真"，如果C_i可以导出，即在程序中有一子句，其头部可与C_i合一，或者C_i是一个内部谓词，且是可由Prolog求值的，其结果为"真"；在其他任何情况下，$V(G_k, C_i, N_c)$="假"，意思是C_i立即失败。

3. 部分求值

设一个Prolog程序通过edb谓词引用数据库关系。edb谓词的形式是：

$$edb(R_i(x_1, \cdots, x_n))$$

部分求值是一种源-源编译技术，其方法是通过冻结一些子目标句节来搜索一棵归结树，其中的一些被冻结的句节就是edb谓词。

这就是说$V(G_k, C_i, N_c)$不仅具有前述完全求值情况下的两个值，而且可以有第三个值，即在如前同一情况中取"真"值，在C_i不是终极句节而且不存在头部可与之合一的子句，或者在C_i是一个内部谓词且求值结果为"失败"情况下取"假"值。在任何其他情形下，尤其在C_i为edb谓词情况下，$V(G_k, C_i, N_c)$的值为"不知道"。

这样，对无递归和无Cut的程序就很容易实现一个部分求值器：部分求值之后，在编译过的程序中，子句的体由被冻结的目标构成。但是，递归和Cut可以在部分求值期间处理，而且，这样的部分求值器已经开发出来了；所用的技术主要是在适当的时候阻塞求值，重新命名句节，以及控制前向/后向合一。

4. 解释和编译耦合技术

解释方法 这种方法采用了标准的Prolog归结策略：当前待处理句节是一个edb谓词时，Prolog即停止其推理过程，将该句节翻译成SQL语句，然后查询数据库，并将提取的事实加载到Prolog工作空间中，Prolog便可以继续求值了。回溯策略将自动用完所提取的事实。

实现这个过程的一种方法就是让edb谓词具有Prolog语法结构：

```
edb(R(x1, ···xn)).
```

然后再将edb定义为一个谓词：

```
1   edb(-relation) :-
2       SQL-translator(-relation),
3       System("SQL-system"),
4       Fail
5   edb(-relation) :-
6       Fact-translator,
7       Load-facts,
8       Call(-relation).
```

其中，第2行程序是将相应句节翻译成SQL形式，第3行是访问数据库，第6行是将提取的关系表转换成Prolog事实，第8行则使利用提取的事实继续求值成为可能。

这个方法的主要特点在于使用Prolog标准策略，并且提供给用户一个完整的Prolog。其主要不足点在于，当同一edb关系在归结中不止一次出现时，所提取的数据常出现冗余，即被多次提取，这一点当然很容易克服。但另一个问题是对数据库的访问太频繁，而且，数据库仅仅只是作为一个文件系

统被利用。这样一种系统，同时采用了子句间优化技术和包孕概念。

编译方法 与前一方法相反，这种方法在对给定的用户目标，将Prolog程序作为一个整体处理，从而将其中涉及推理机的部分和涉及数据库的部分分离开来。编译之后，对DB的询问被翻译成数据操纵语言语句，并且只对数据库作一次访问（因为所有的询问都被分组，并经过优化），一旦事实都加载到了主存之中，最终的求值便可以开始。

这样，Prolog的标准策略就被修改了，新的策略受一个元解释器控制，该元解释器冻结所有的edb谓词，它可以看作是一个部分求值器。这个方法的优点是只对数据库做一次访问。这样一个系统，结合了优化技术，并利用了定义域约束和函数相关性等等。

5. EPSILON方法

这种方法是对编译方法的扩充和改进，在纯编译方法中，部分求值的结果是一棵归结树，其叶结点（即被冻结的部分）是edb谓词的合取式。对部分求值器的功能进行扩充，可以使叶结点成为edb谓词及其他被冻谓词的混合。然后，必须将edb谓词分组集中，以减少对DB的独立查询次数，并尽可能从关系算子，尤其是"连接（Join）"受益。

在这个方法中，数据有时会被多次提取，也可能有些数据被提取后却对推理过程毫无用处。相应的部分求值器以及一个基于Prolog与DBMS耦合的原型系统已经在里昂一大的计算机科学实验室开发出来。

6. 编译-解释混合式方法

逻辑与集合 用到的一些记号：

设C_1和C_2是两个集合，C_1包含C_2，记作$C_1 \supseteq C_2$，当且仅当$\forall x[x \in C_2 \Rightarrow x \in C_1]$（"⇒"表示蕴含）。

对每个集合P，通过公式$P=\{X|\ P(X)\}$有一个谓词P(　)与之相联系。

逻辑与集合中几种运算之间的关系由如下的几个经典公式给出：

$$S \cup P=\{X/S(X),\ P(X)\}$$
$$S \cap P=\{X/S(X),\ P(X)\}$$
$$S-P=\{X/S(X),\ \neg P(X)\}$$

在本文和耦合式原型系统中，封闭世界假设和唯一名假设是成立的。

混合方法的基本思想是保持编译和解释方法的优点，而克服其缺点，即：

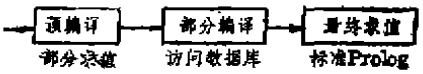
—只要可能，就使用关系型DBMS的各种功能，尤其是连接算子。

—使对DB的访问减到最少。

—通过最大限度地实例化 edb 谓词而生成尽可能使用选择运算的数据库查询语句。

—为用户提供一个完整的Prolog，包括函数、Cut、递归、否定，以及尽可能多的内部谓词。

EPSILON编译方法可分为三个步骤：



这一方法的缺点在于将EDB谓词组从部分求值程序中的子句的体中分离出来（这一步骤称为动作体生成），并对它们进行独立静态处理。我们可以举一个例子来说明：

设一个用户程序为：

```
mc(-x, -y):-chief ( x, -z1),  mc(-z1, -z2),
    ser(-s, -z2), ser(-s, -z3),
    mc(-z3, -z4), chief(-y, -z4).
```

其中chief和ser均为EDB谓词，即它们要访问数据库元组。预编译后将产生两部分程序，第一部分是：

```
mc(-x, -y):-activanto(-x, -z1),
    mc(-z1, -z2),
    activant1(-s, -z2, -z3),
    mc(-z3, -z4),
    activanto(-y, -z4).
```

与Prolog相关，而第二部分则为：

```
activanto(-x, -y):-edb(chief(-x, -y)).
activant1(-x, -y, -z):-edb(ser(-x, -y)),
    edb(ser(-x, -z)).
```

与数据库相关。

记为 activanto(-x, -y) 和 activant 1 (-x, -y, -z)的数据库元组集合：

{(x, y)/chief(x, y)}和{(x, y, z)/ser(x, y), ser(x, z)}将被提取。

但是，当进行最终求值时，标准SLD归结在调用activant1时将产生变量实例化，

置换θ_1，由于对activant0(-x, -y)归结的结果；置换θ_2，由于对$mc(-z_1, -z_2)\theta_1$归结果的结果。这样，调用实际上将是activant1(-x, -y, -z) $\theta_1\theta_2$。

当然，activant1(-x, -y, -z)$\theta_1\theta_2$是包孕于activant1(-x, -y, -z)的，这样，在查询数据库时就很可能加载了无用事实。

一般而言，各种动作体之间都存在着某些依赖关系，比如交集非空、包含或等价。对相依动作体进行的优化称为"子句间优化"。在EPSILON原型系统中，只考虑了等价关系，即检测出两个动作体等价后，给它们赋以相同的名字，从而只对数据库产生一次查询。在动作体之间存在包孕关系时，例如：

activant1(-x, -y,-z):-edb(mc(-x,-y -z)).

activant2(-x, -y):-edb(mc(a, -x, -y)).

很明显，与"mc(a, -x, -y)"匹配的事实被提取了两次。

包孕检测可以解决这个问题，但可惜太耗时间。在动作体之间的交集非空而又并非一个包含于另一个之内的场合下，除了对数据库做两次访问之外别无他法。

对于递归，在纯编译方法中，由于在部分求值期间阻断了前向合一，将导致加载过多的事实。

混合式方法解答了这诸多问题。该方法的原理很类似于EPSILON编译方法：

—部分求值步骤是完全一样的；

—动作体生成步骤也是完全一样的，只是动作体（同构edb谓词组）被称为"edb调用"；

—最终的求值由Prolog解释器处理。在解释过程中，每个"edb调用"原语都被动态求值。这是求值策略的共同点。

附录中给出了所开发的原型系统结构，它由两个主要模块组成：一个编译器和一个求值器。

源-源编译器包括EPSILON部分求值器和一个静态翻译器，其作用是把部分求值后的程序转换成Prolog可解释的形式。

这样，就避免了最终进行的一次元解释，但是这要用到一个透明的内部谓词：edb-call，其形式为：

edb-call(activant*(x_1, x_2, …, x_n), [$\gamma_1(x_{11}, \cdots, x_{1k})$, …, $\gamma_n(x_{n1}, \cdots, x_{nj})$])

其中"activants"（动作体）是自动生成的句节，可以区别edb谓词的各种合取式。

γ_1, …, γ_n是edb谓词，而且有下式成立：

$$\{x_1, \cdots, x_n\} \subseteq \{x_{11}, \cdots, x_{1k}\} \cup \cdots \cup \{x_{n1}, \cdots, x_{nj}\}$$

该edb-call谓词的含义是，子句：

activant*:-edb(γ_1),…, edb(γ_n).

必须由DBMS处理。

现在，我们来定义所谓EDB表：

令L_i:-e_1, …, e_n是部分求值过的程序中的一个子句，则：

[e_j]是一个edb表，如果e_j是edb谓词；

[e_j, e_{j+1}, …, e_k]是edb表，如果：每个e_p都是edb谓词，并且每个e_{j+i}($0 \leqslant i \leqslant k-j$)都至少与一个$e_{j+p}$ ($p \neq i$)有一个公用变量。这一约束保证了数据库管理系统不会做笛卡尔乘积。

[e_j, …, e_k]是一个极大edb表，当且仅当[e_{j-1}, …, e_k]和[e_j, …, e_k, e_{k+1}]均不是edb表。

为了减少对数据库的独立访问次数，很明显，edb表必须是极大表。例如，对用户程序

```
mc(-x, -y):-
    edb(chief(-x, -z1)), mc(-z1, -z2),
    edb(ser(-s, -z2)), edb(ser(-s, -z3)),
    mc(-z3, -z4), edb(chief(-y, -z4)).
```

所生成的极大edb表为：

1 edb(chief(-x, -z_1))

2 edb(ser(-s, -z_1)), edb(ser(-s, -z_3))

3 edb(chief(-y, -z_4))

每个edb表都将包含在一个edb调用原语中，下面介绍edb调用原语的生成。

定义 表〔e_1'，…，e_n'〕被称为是与edb表〔e_1，…，e_n〕相联系的edb公式，如果edb表中的每个常量都被变量代换，而且相同的常量被同一个变量代换。

例 与edb表〔edb(father(-x, b)), edb(father(b, -y))〕相联系的edb公式是：

〔edb(father(-x, -z)), edb(father(-z, -y))〕

设L_1'和L_2'是分别与edb表L_1和L_2相联系的两个edb公式，则：

L_1与L_2只生成一个动作体，当且仅当$L_1'=L_2'$（变量名可以不同）。

例 考虑两个edb表：

〔ser(z, x), ser(z, y)〕

和〔ser(z, x), ser(z, a)〕，它们将只生成一个动作体，不妨称之为activanti。相应的极大edb表将分别被代之以：

```
edb-call(activanti(-x, -y, -z),
    〔ser(-z, -x), ser(-z, -y)〕)
edb-call(activanti(-x, a, -z),
    〔ser(-z, -x), ser(-z, a)〕)
```

对前面的例子，静态翻译器将产生如下的程序：

```
mc(-x, -y):-
    edb-call(activanto(-x, -z),
      〔chief(-x, z1)〕),
    mc(-z, -z2),
    edb-call(activanti(-z2, -z3, -s),
      〔ser(-s, -z2), ser(-s, z3)〕),
    mc(-z3, -z4),
    edb-call(activanto(-y, -z1),
      〔chief(-y, -z1)〕).
```

附录中我们给出了一个更完全、更复杂的例子。

混合式方法比较类似于解释方法，但有一个根本差别：在解释方法中，每个edb谓词都产生一次数据库访问，而在混合式方法中，一次数据库访问是由一个edb调用生成的。

混合式方法中，最终求值可以用两种方式完成：第一种方式是元解释，没被采用，原因之一是效率问题，原因之二是为了避免元解释器处理Cut的困难。这样，我们便采用了第二种方式，即由Prolog标准解释器作最终解释。

对DBMS的访问是通过edb-call谓词处理的，如下例所示：

```
edb-call(-activant*, -edb-list):-
    SQL-translator(-activant*, -edb-list),
    System("SQL system"),
    Fail.
edb-call(-activant*, -):-
    Fact-translator,
    Load-facts,
    Call(-activant*).
```

其中，SQL-translator，"system"，Fact-translator，Lood-facts以及Call都与解释式方法中的一样。SQL-translator将头为"activant"而体为"edb-list（edb表）"的子句翻译成SQL语言语句。

这种策略可用于解决不完全实例化动作体的问题。在最终求值期间，实际调用的动作体是$activant1\theta_1\theta_2$而不是activant1。

仍然存在一个问题，事实冗余。许多极大edb表，如果它们具有相同的edb公式的话，则将对应于一给定动作体。这样，

〔father(a, s), father(s, y)〕

和〔father(x, s), father(s, y)〕

相应于同一动作体，从而带来了"双份"的问题。

我们记$|A|=\{t/A(t)\}$，其中t为元组变量，$|A|$表示从数据库中提取的所有数据。

设A_1=activant0(a, -y)

A_2=activant0(-x, -y)

为两个动作体，显然它们有相同的edb公式。于是，activant0(a, -y)包含于activant0(-x, -y)，这意谓着$|A_1|\subseteq|A_2|$，也即$|A_1|$中的事实被提取了两次，这不仅是低效的，而且是不正确的。

在 $|A_1|$ 被提取后，应该提取的是 $|A_2|-|A_1|$ 而不是 $|A_2|$，用集合来表达就是

$$\{t\,|A_2(t),\ \neg A_1(t)\}。$$

这个问题由“optimizer(优化器)”模块处理，该模块是“edb-call”的一个子程序。这个模块管理一个元库，元库中存贮着所遇到的每一个“更广义”的动作体。这里的所谓“广义”是在集合的“包孕”序关系的意义上讲的。这个模块中的主要谓词是“history（历史)”，其作用之一是将当前动作体A_c与元库中那些更一般化的同名动作体进行比较。

重要的是指出：

$|A_1|\cap|A_2|\neq\phi\Rightarrow\{A_1,\ A_2\}$是可合一的，这是因为$A_1$、$A_2$是同名的。

这样，为了不断更新元库，就必须做这种简单的可合一性检验，而不必作包孕检验。“History”是一个二元谓词：

history(A_c, A_o)

其中A_c是当前动作体，A_o是元库中极大动作体表，$A_o=[A_1,\ \cdots,\ A_n]$，满足$\{A_i,\ A_c\}$可合一。

元库的管理办法如下：

1. 如果元库中没有可与A_c合一的极大动作体，则$|A_c|$被加载到Prolog主存工作空间中。

2. 如果元库中有极大动作体A_i，A_i包孕A_c，则$|A_c|$不再加载，因为$|A_i|\supseteq|A_c|$。

3. 否则，如前定义$A_o=\{A_1,\ \cdots,\ A_n\}$，且$A_c$就成为元库中一个更广义的动作体，所有满足$|A_i|\subset|A_c|$的$A_i$均被清出元库，并且加载$|A_c|-(|A_1|\cup\cdots\cup|A_n|)$——这一集合的逻辑表达式是：$A_c,\ \neg A_1,\ \cdots,\ \neg A_n$。

现在让我们来考察一个数据库查询的实际处理过程，该查询的形式为：

$$A_c,\ \neg A_1,\ \cdots,\ \neg A_n$$

第一个解决方案是使用关系型求差算子，这个算子不是在任何DBMS中都提供的（例如在INFORMIX中）。

但是，考虑到A_c，A_1，…，A_n同名，而且只被变量或常量所实例化，就有可能基于谓词演算对其进行形式化的优化。作为一个例子，让我们考虑

$A_c=A(a,\ y,\ z)$

$A_1=A(a,\ b,\ z)$

$A_2=A(x,\ b,\ c)$

$|A_c|-(|A_1|\cup|A_2|)=\{(x,\ y,\ z)/A(x,\ y,\ z),$

$x=a,$

$y\neq b,$

$(y\neq b;\ z\neq c)\}$

很重要的一点是，一个变量不管在A_i中是否实例化，在A_c中都将被实例化，这是由于A_c和A_i是可合一的，所以，所有的负约束都是施加在A_c中的未实例化变量上。

这样，优化器便起着形式逻辑公式处理器的作用。它处理并简化“与-或”表达式，产生一个由负约束的析取式构成的合取式。

把与当前动作体A_c相关联的EDB表译成SQL语句，再加上实例化约束、连接约束以及负实例化约束，这一切都由一个经典的翻译器处理。数据库被访问，元组以Prolog事实的形式被加载到主存并赋以与相应动作体相同的名称，这样，求值便可以继续下去了。

现在，在最终求值阶段处理数据库调用的谓词edb-call的新定义中将包括优化器和“历史”管理谓词：

```
edb-call(-activant*, -edb-list):-
    history(-activant*, A_o),
    s-edb-call(-activant*, -edb-list, A_o).
edb-call(activant*, -):-!,call(activant*).
s-edb-call(-activant*, -edb-list, A_o):-
    difference(A_o, -and-or-list).
    db-translator(-activant*,
                  -edb-list, -and-or-list),
    system("SQL system"),
    fail.
s-edb-call(-activant*, -, -):-
    fact-tranrsiator,
    load-facts,
    call(-activant*).
```

7. 结束语

本文介绍了一种用于Prolog与关系数据库耦合的混合式方法，这种半编译、半解释的方法具有下

述优点：

·由于采用了将edb谓词分组归并为edb表的技术，对数据库的访问没有纯解释方法频繁。这种技术是从纯编译方法继承来的。

·与纯编译方法相比，提取的事实较少，这是因为对数据库的查询语句实例化得更多，因而就更带有选择性（有较多的选择操作）。而且，由于对场作体采取了"历史"管理措施，事实不会被多次加载。

·能给用户提供完整的Prolog语言，具有各种函数，Cut，递归，以及内部谓词。

已经在SUN 3/50 UNIX机器上开发了一个原型系统，所用的数据库系统是Informix。

本文中所介绍的技术也适用于其他策略，甚至在逻辑/推理部分与事实管理机制相配合的其他情况亦如此。事实上，不管是什么样的结合，问题都是一样的：减少通讯频率，并减少交换的信息量。另外，这个原型中的运行模块逻辑上都是独立的，这使得它们可被重用于其他策略的框架中：

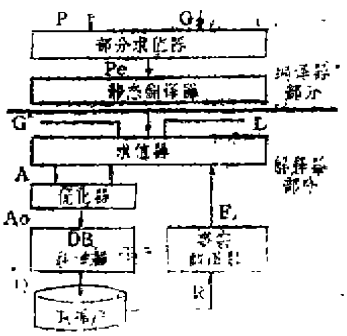
—部分求值器
—静态翻译器
—元库管理器
—求差运算的符号优化器
—SQL翻译器
—关系元组到Prolog事实的格式化程序（见附录）。

原型系统的实现表明用关系数据库做Prolog的事实管理器是可行的、有效的。

参考文献（略）

〔石桥林译自《Proc. of 3rd Int'l Conf. on Data and Knowledge Bases, Jerusalem, Israel, 1988》，李磊校〕

附录　原型系统的结构




```
/*  源程序*/
meta(supplier(-x), [edb(supplier(-x))]).
meta(part(-x), [edb(part(-x))]).
meta(subpart(-x, -y), [edb(subpart
(-x, y))]).
meta(has-supplied(-x, -y),
    [edb(has-supplied(-x, -y))]).
meta(lives(-x, -y), [edb(lives(-x,-y))]).
meta(north(london), []).
meta(north(oslo), []).
meta(south(rome), []).
meta(south(athens), []).
meta(suggest-order(-supplier, -part),
    [good-and-cheap(-supplier,-part)]).
meta(suggest-order(-supplier, -part),
    [not(good-and-cheap(-any-supplier,
-part))
    north-european(-supplier)
    potential-supplier(-supplier, -part)])
meta(good-and-cheap(-supplier, -part).
    [potential-supplier(-supplier, -part)
    south-european(-supplier)])
meta(potential-supplier(-supplier,-part).
    [supplier(-supplier)
    part(-part)
    not(missing-subpart(supplier.-part))])
meta(missing-subpart(-supplier,-part).
meta(north-european(-supplier),
    [lives(-supplier,-city),north(-city)]).
meta(south-european(-supplier),
    [lives(-supplier,-city),south(-city)]).
/*  静态翻译结果*/
|-program(-0, -1):-
    edb-call(activant0(-0),[supplier(-0)])
    edb-call(activant1(-1),[part(-1)])
    not missing-subpart(-0,-1)
    edb-call(activant2(-0, rome),[lives
(-0, rome)])
|-program(-0, -1):-
    edb-call(activant0(-0),[supplier(-0)])
    edb-call(activant1(-1), [part(-1)])
    not missing-subpart(-0, -1)
    edb-call(activant2(-0, athens),
        [lives(-0, athens)]
```

```
|-program(-0, -1):-
    not good-and-cheap(-123, -1)
    edb-call(activant3(-0, london),
        [lives(-0, london), supplier(-0)])
    edb-call(activant1(-1), [part(-1)])
    not missing-subpart(-0, -1)
|-program(-0, -1):-
    not good-and-cheap(-123, -1)
    edb-call(activant3(-0, oslo),
        [lives(-, oslo), supplier(-0)])
    edb-call(activant1(-1), [part(-1)])
    not missing-subpart(-0, -1)
missing-subpart(-111, -112):-
    not has-supplied(-111, -112)
missing-subpart(-111, -112):-
    edb-call(activant4(-123, -112),
        [subpart(-123, -112)])
    missing-subpart(-111, -123)
good-and-cheap(-123, -124):-
    edb-call(activant0(-123), [supplier
(-123)])
    edb-call(activant1(-124),[part(-124)])
    not missing-subpart(-123, -124)
    edb-call(activant2(-123, rome),
        [lives(-123, rome)])
good-and-cheap(-123, -124):-
    edb-call(activant0(-123), [supplier
(-123)])
    edb-call(activant1(-124), [part
(-124)])
    not missing-subpart(-123, -124)
    edb-call(activant2(-123, athens),
        [lives(-123, athens)])
has-supplied(-135, -136):-
    edb-call(activant5(-135, -136),
        [has-supplied(-135, -136)])
/*部分求值结果*/
/*针对询问suggest-order(-x, -y)*/
|-program(-0, -1):-
    edb(supplier(-0))
    edb(part(-1))
    not missing-subpart(-0, -1)
    edb(lives(-0, rome))
|-program(-0, -1):-
    edb(supplier(-0))
    edb(part(-1))
    not missing-subpart(-0, -1)
    edb(lives(-0, athens))
|-program(-0, -1):-
    not good-and-cheap(-123, -1)
    edb(lives(-0, london))
    edb(supplier(-0))
    edb(part(-1))
    not missing-subpart(-0,-1)
|-program(-0, -1):-
    not good-and-cheap(-123, -1)
    edb(lives(-0, oslo))
    edb(supplier(-0))
    edb(part(-1))
    not missing-subpart(-0, -1)
missing-subpart(-111, -112):-
    not has-supplied(-111, -112)
missing-subpart(-111, -112):-
    edb(subpart(-123, -112))
    missing-subpart(-111, -123)
good-and-cheap(-123, -124):-
    edb(supplier(-123))
    edb(part(-124))
    not missing-subpart(-123, -124)
    edb(lives(-123, rome))
good-and-cheap(-123, -124):-
    edb(supplier(-123))
    edb(part(-124))
    not missing-subpart(-123, -124)
    edb(lives(-123, athens))
has-supplied(-135, -136):-
    edb(has-supplied(-135, -136))
```