函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第十四讲
逻辑程序设计

什么是逻辑程序设计

- 传统程序设计如下定义表的 append ：

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

它规定了append 如何进行，但是不能回答下面问题：

- 找出一个表y， (a b)与y的append是(a b c d)
- 找出所有的x和y，使得x与y的append是 (a b c d)

什么是逻辑程序设计

● 在逻辑程序中，给出一些断言和规则，然后就可以在逻辑程序中提出查询，逻辑程序的解释器会自动寻找出答案输出(不能保证总是能找到正确答案）。

●对于append问题，给出两条规则：

➢对任何一个表 y，空表与其append得到的表是 y 本身

```
(append y '()) = y
```

➢任何 u，v，y，z，(cons u v) 与 y的append得到 (cons u z) 的充要条件是 v 与 y 的append得到 z

```
(append (cons u v) y) = (cons u z)
<=>
z = (append v y)
```

什么是逻辑程序设计

- 根据append问题的规则求解x，使得(append （a b） x） = （a b c d）

```
(append (cons u v) y) = (cons u z)
<=>
z = (append v y)

(append (a b) x)  =
(append (cons a (b)) x)  =    ;u = a,v = (b), y = x,
(cons a (append (b) x)) =  ;z = (append (b) x)
(cons a (append (cons b ()) x)) =
(cons a (cons b (append () x))) =
(cons a (cons b x))  = (a b c d)

=> x = (c d)
```

什么是逻辑程序设计

- 在逻辑程序中，针对append的规则设定和查询:

➢规则设定：

```
(assert! (rule (append-to-form () ?y ?y))) ;空表append任何y，结果是y
(assert! (rule (append-to-form (?u . ?v) ?y (?u . ?z)) ;规则的结论
       (append-to-form ?v ?y ?z))) ;规则的体(结论的充要条件)
; v append y形成 z ⇔ (cons u v) append y 会形成 (cons u z)
```

➢查询：

```
(append-to-form (a b) (c d) ?z)
=> (append-to-form (a b) (c d) (a b c d))
(append-to-form ?x ?y (a b c d))
=>
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
```

实例数据库：Microshaft人事数据库

- Microshaft公司人事数据库中有以下断言：

```
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
(job (Bitdiddle Ben) (computer wizard))
(salary (Bitdiddle Ben) 60000)
;在逻辑程序中描述断言的形式是: (assert! (salary (Bitdiddle Ben) 60000))
(address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
(job (Hacker Alyssa P) (computer programmer))
(salary (Hacker Alyssa P) 40000)
(supervisor (Hacker Alyssa P) (Bitdiddle Ben));Ben是P的上司
(address (Fect Cy D) (Cambridge (Ames Street) 3))
(job (Fect Cy D) (computer programmer))
(salary (Fect Cy D) 35000)
(supervisor (Fect Cy D) (Bitdiddle Ben))
(address (Tweakit Lem E) (Boston (Bay State Road) 22))
(job (Tweakit Lem E) (computer technician))
(salary (Tweakit Lem E) 25000)
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
```

实例数据库：Microshaft人事数据库

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))


(supervisor (Bitdiddle Ben) (Warbucks Oliver))
(address (Warbucks Oliver) (Swellesley (Top Heap Road)))
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 150000)
```

实例数据库：Microshaft人事数据库

```
(address (Scrooge Eben) (Weston (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Warbucks Oliver))
(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))


(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(job (Aull DeWitt) (administration secretary))
(salary (Aull DeWitt) 25000)
(supervisor (Aull DeWitt) (Warbucks Oliver))
```

实例数据库：Microshaft人事数据库

```
(can-do-job (computer wizard) (computer programmer))
;computer wizard可以做 computer programmer的任何工作
(can-do-job (computer wizard) (computer technician))

(can-do-job (computer programmer)
            (computer programmer trainee))

(can-do-job (administration secretary)
            (administration big wheel))
; administration secretary可以做administration big wheel的任何工作
```

简单查询

●查询及其结果(查询也称为“模式”或“查询模式”):

查询就是寻找所有使得模式成立(可从数据库中推导出来)的变量的赋值方案(变量以'?'开头，同一变量的不同出现都只能赋同一个值)

```
(job ?x (computer programmer))  ;谁是computer programmer
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))

(address ?x ?y)   ;列出所有人及其地址
(supervisor ?x ?x)  ;谁自己是自己的老板

(job ?x (computer ?type))  ;谁的工作是 computer XXX
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
(job (Tweakit Lem E) (computer technician))

(job (Reasoner Louis) (computer programmer trainee)) 不匹配，因为computer后面有两项
```

简单查询

●查询及其结果：

```
(job ?x (computer . ?type)) ;谁的工作是 computer XXX YYY ....
(job (Reasoner Louis) (computer programmer trainee))
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
(job (Tweakit Lem E) (computer technician))
```

'.' 后面的 ?type，匹配时可以是一个列表（有任意多项），也可以是空表

●如果查询中没有变量，则只看该查询是否出现在数据库中

复合查询and,or,not和lisp-value

- `(and <query1> <query2> ... <queryn>)`

```
(and (job ?person (computer programmer))
     (address ?person ?where))
(and (job (Hacker Alyssa P) (computer programmer))
     (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
(and (job (Fect Cy D) (computer programmer))
     (address (Fect Cy D) (Cambridge (Ames Street) 3)))
```

复合查询and,or,not和lisp-value

- (or <query1> <query2> ... <queryn>)

```
(or (supervisor ?x (Bitdiddle Ben))
    (supervisor ?x (Hacker Alyssa P)))
```

```
(or (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
    (supervisor (Hacker Alyssa P) (Hacker Alyssa P)))
(or (supervisor (Fect Cy D) (Bitdiddle Ben))
    (supervisor (Fect Cy D) (Hacker Alyssa P)))
(or (supervisor (Tweakit Lem E) (Bitdiddle Ben))
    (supervisor (Tweakit Lem E) (Hacker Alyssa P)))
(or (supervisor (Reasoner Louis) (Bitdiddle Ben))
    (supervisor (Reasoner Louis) (Hacker Alyssa P)))
```

复合查询and,or,not和lisp-value

- **(not <query1>)**
  找出所有对变量的赋值方案，使其不满足条件 **<query1>**。

➢ 数理逻辑的“非x”说明x一定为假，此处的 **(not X)** 即便成立，也只是意味着“X为真”这件事不能从已有的事实推导出来。

```
(not (friend alysa BenDog))
=>(not (friend alysa BenDog))
```

虽然数据库里根本没有 **friend,BenDog**这些词，查询的结果也认为 **(not (friend alysa BenDog)** 是成立的

➢ **not和and，or可以一起用：**

```
(and (supervisor ?x (Bitdiddle Ben))
     (not (job ?x (computer programmer))))
```

复合查询and,or,not和lisp-value

- **(lisp-value <predicate> <arg1> ... <argn>)**

➢ **<predicate>是scheme的基本谓词,如 = < > 。也可以扩展解释器，让<predicate>支持scheme的基本过程甚至解释器程序中编写的过程。**

➢**此查询寻找所有使得(<predicate> <arg1> ...<argn>)为真的变量赋值方案**

例如：

```
(and (salary ?person ?amount)
  (lisp-value > ?amount 30000))
```

查找所有工资大于30000的人

规则

- (rule <conclusion> <body>)

  <body>成立 ⇔ <conclusion>成立

;在逻辑程序中描述规则的形式是: (assert! (rule (same ?x ?x)))

下面规定了两个人住得近的条件

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
(rule (same ?x ?x))   ;一个规则可以在另一个规则中使用
```

➢下面规定了一个人是大老板的条件

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

➢下面规定了一个人是另一个人的上级的条件

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
          (and (supervisor ?staff-person ?middle-manager)
               (outranked-by ?middle-manager ?boss))))
```

规则

●一些根据规则的查询及其结果：

```
(lives-near ?x (Bitdiddle Ben))
(lives-near (Reasoner Louis) (Bitdiddle Ben))
(lives-near (Aull DeWitt) (Bitdiddle Ben))
```

要查找和Ben住的近的程序员：

```
(and (job ?x (computer programmer))
     (lives-near ?x (Bitdiddle Ben)))
```

查询系统如何工作

- 用流控制搜索
- 两大核心操作：模式匹配和合一
- 模式匹配用于处理简单查询和复合查询，只需用到断言，不需要用规则。
  断言不包含变量
- 合一是模式匹配的推广，用于处理需要用到规则(rule)的查询。

简单模式匹配器

- 一个简单模式匹配器是一个过程，其检查数据项是否符合一个给定的模式

数据项：`((a b) c (a b))`
能与之匹配的模式：

```
(?x c ?x)          ; x-> (a b)
(?x ?y ?z)         ;x-> (a b)   y-> c   z->(a b)
((?x ?y) c (?x ?y))          ;x->a y->b
```

`(?x a ?y)` 不能与之匹配

简单模式匹配器

● 简单模式匹配器以一个模式，一个数据项和一个框架作为输入，返回值也是一个框架。简单模式匹配器处理不需要用规则的简单查询。

●框架是模式变量到其值的约束的一个列表，每个元素都是一个变量约束。同一变量在一个框架里不能存在两个约束。

●一个变量约束是一个序对，可以是表，也可以不是表。其car是变量，cdr是变量被约束到的值

●框架形如：

```
(  ((? x) a) ((? z) . c) ((? y) a b) ((? u) (a b)) ((? k) ? y) )
```

➢ 变量?x经过解释器预处理后都变成列表(? X)

➢x被约束到(a)， z被约束到c，y被约束到 (a b),u被约束到 ((a b)),k被约束到(? y)，即变量y（k的值必须和y相同）

简单模式匹配器

- 简单模式匹配器以一个模式 M，一个数据项 D和一个框架 F1作为输入，返回值是一个扩充后的框架 F2或失败标志。数据项D来源于数据库中的断言。

- F1里存放了一些模式变量的约束。模式匹配器将M中各个变量约束到D中对应的值，形成变量的赋值方案，该方案如果和F1不矛盾，则认为该赋值方案成功使得M和D匹配，于是将该赋值方案加入到F1,形成F2返回。如果矛盾，则不返回框架，而是返回一个失败标志符号，比如 failed

```
        M :   ((? x) (? y) (? x))
        D :   (a b a)
        F1:   ()
返回值  F2:   (  ((? x) . a)   ((? y) . b)   )
```

若F1是 (((? y) . a)) 则 F2为 failed
若F1是 (((? y) . b)) 则 F2为 (((? x) . a)   ((? y) . b)   )

简单模式匹配器

- 如何应对查询 `(job ?x (computer programmer))`

对于数据库里的每个断言，将其作为D
将`(job (? x) (computer programmer))` 作为M
将`()`作为F1

调用简单模式匹配器

对每一个成功的返回值F2,都用F2里面对x的约束，去实例化：

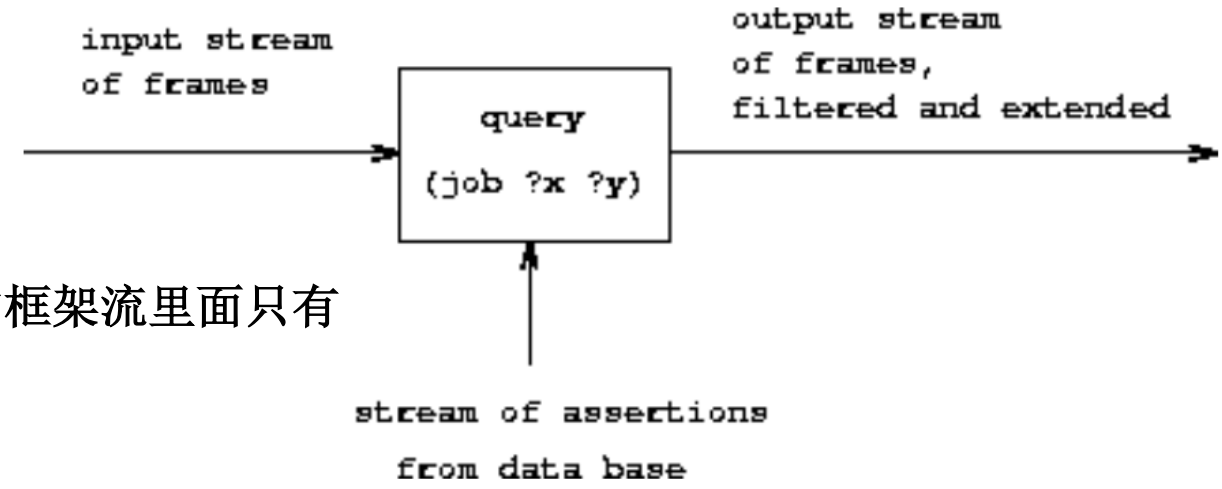
```
(job ?x (computer programmer))
```

即得到一个查询结果(查询结果可以有多个)

框架的流

● 查询系统以一个框架流和一个模式作为输入。对流中的每个框架，扫描数据库中的每个条目，如果模式能匹配上，则返回扩充的框架，如果不能匹配上 ，则返回失败标志。扩充的框架和失败标志被混合成一个流，在该流上过滤掉失败标志。输入框架流中的每个框架都可能产生一个输出框架流。

●把每个框架所产生的流全部都组合起来形成一个更大的流，根据这个流生成要输出的查询结果。



●回答简单查询时，输入的框架流里面只有一个空框架。

复合查询

- and查询

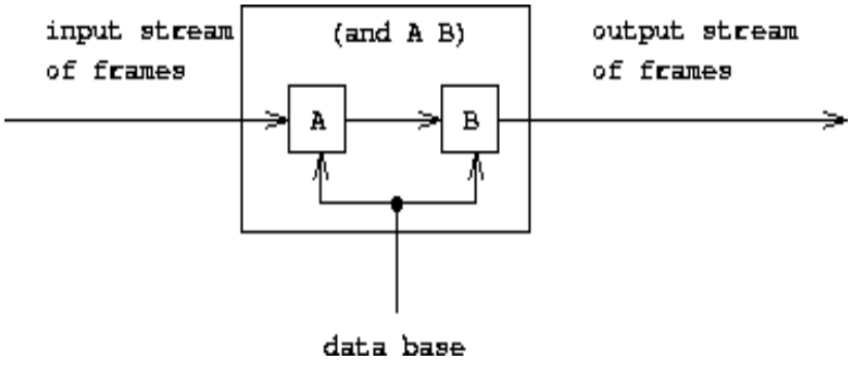
➢查询：(and (can-do-job ?x (computer programmer trainee))
(job ?person ?x))

找出所有这样的人，他们都能做实习程序员的所有工作

➢ (can-do-job ?x (computer programmer trainee)

的查询结果是一个框架流（每个框架都包含对x的约束），
以这个框架流作为输入，去执行查询

(job ?person ?x)

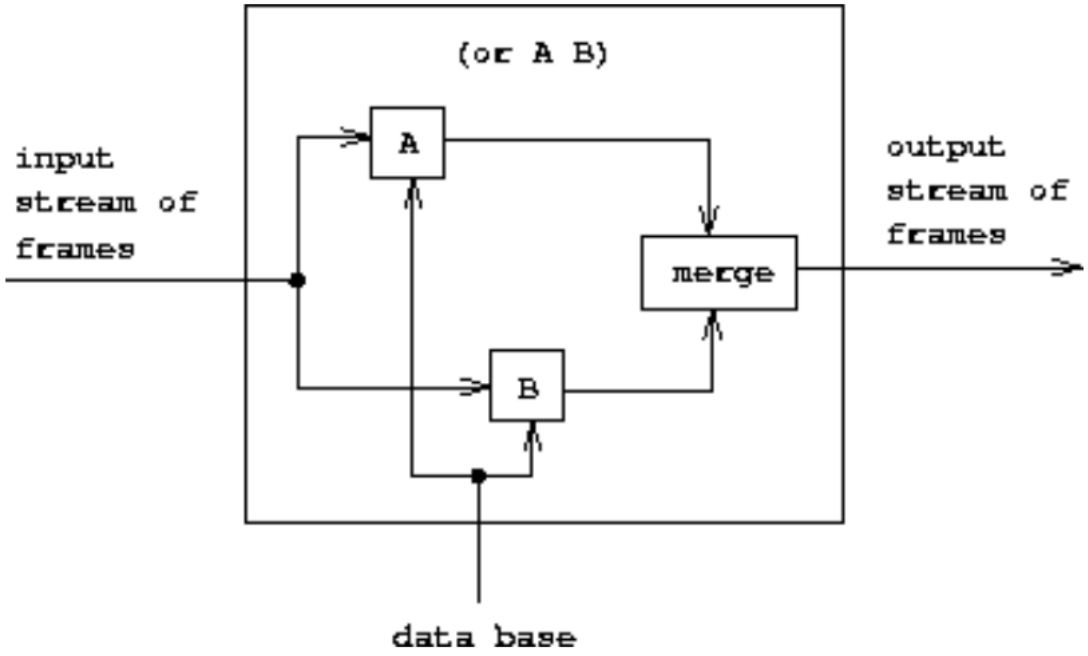


➢查询过程用到了简单模式匹配器

复合查询

- or查询

➢两个查询结果被归并到一起(用解释器中的interleave-delayed)，形成新的查询结果流



复合查询

- not查询

not查询是个过滤器。从输入的框架流中删除满足查询条件的框架。

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
```

(supervisor ?x ?y)产生出一个匹配成功的框架流，not查询在框架流中找到那些满足 (job ?x (computer programmer)) 的框架，删除之。

- lisp-value查询

和not类似，也是个过滤器。从输入框架流中删除不满足查询中所规定的条件的框架。

合一

- 合一用于处理和规则相关的查询

- 模式匹配的模式里有变量，数据项都是断言，没有变量

- 规则的结论部分有变量，也可以看做是一种模式

- 合一器取两个都可以包含常量和变量的模式和一个框架作为参数，设法寻找到某种变量的赋值方案，使得两个模式相等，且和参数框架不矛盾。如果找到，则返回包含这种赋值方案的框架。找不到则返回失败标记。

`(?x a ?y)` 和`(?y ?z a)`加空框架合一的结果是：

```
( ((? x) . a) ((? y) . a) ((? z) . a))
```

`(?x ?y a)` 和`(?x b ?y)`的合一结果是失败

合一

- 成功的模式匹配导致模式里的变量都被约束到常量
- 成功的合一，模式里的变量可能依然没有被约束，也可能被约束到另一个变量

`(?x a)` 和 `((b ?y) ?z)`的合一。结果：

```
( ((? x) b (? y))    ((? z) . a)    )
```

没有对 `y` 的约束，也无法再对`y`进行求解。除非这一框架可能后续作为其他匹配或合一运算的输入，然后又被扩充。如果对`y`的约束值`Y`被加入该框架，则`x`的值也必须是`(b Y)`

规则的应用

●查询的过程，既要考察全部断言，也要考察全部规则，把模式匹配和合一的结果合并

考虑模式： `(lives-near ?x (Hacker Alyssa P))` ,没有断言可以匹配

可以尝试和下面的规则的结论合一：

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))
```

模式和结论合一的结果是个框架：

`(((? x) ? persion-1) ((? person-2) . (Hacker Alyssa P)) )`

用这个框架去匹配规则的体，即进行 `and` 复合查询。如果成功匹配，返回值是个扩充后框架，否则返回值是个失败标记

需要考察规则的简单查询

查询的过程，既要考察全部断言(模式匹配)，也要考察全部规则(合一)，把模式匹配和合一的两个结果流合并成一个新的最终的结果流。

有可能出现的死循环

逻辑程序有可能导致死循环。

```
(assert! (married Minnie Mickey))

(married Mickey ?who)  ;没有答案，返回的是个空流
```

加入规则：

```
(rule (married ?x ?y)
      (married ?y ?x))
```

再次查询 (married Mickey ?who)，导致死循环

有可能出现的死循环

规则的结论 **(married ?x ?y)** 与模式 **(married Mickey ?who)** 成功合一，返回框架：

```
( ((? x) . Mickey)  ((? y) ? who) )
```

用这个框架去匹配规则的体 **(married ?y ?x)** ，即处理查询

```
(married ?who Mickey)
```

数据库有断言 **(married Minnie Mickey)**，得到一个查询结果。

但是处理 **(married ?who Mickey)** 也不能漏掉对规则的合一，因此其和规则的结论合一后，再去匹配规则的体，就等价于查询

```
(married Mickey ?who)
```

于是陷入死循环

逻辑程序解释器（即查询系统）的实现

数据库的维护

●用一个大的流（全流）保存所有的断言。但是如果每次做匹配都要遍历全流，效率低。

●除了全流外，将所有car为符号XXX的断言存成一个流，所有car为符号YYY的断言存成一个流....这样如果碰到car是XXX的模式，则只需要搜寻XXX断言流，提高效率。这些流称为“分流”

●规则用和断言类似的办法保存。结论的car是变量的规则，单独存为一个分流，以?作为索引。因为在匹配car为符号XXX的模式时，不但要考虑结论的car是符号XXX的规则，还要考虑结论的car是变量的规则。因为该变量可以约束到XXX，也许也能导致合一成功。

数据库的维护

```
(define THE-ASSERTIONS the-empty-stream) ;断言全流，开始是空
(define (fetch-assertions pattern frame) ;根据pattern取出需要遍历的断言流
  (if (use-index? pattern) ;判断pattern的car是不是symbol
      (get-indexed-assertions pattern) ;取相应的分流
      (get-all-assertions))) ;取全流
(define (get-all-assertions) THE-ASSERTIONS)
(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))

(define (get-stream key1 key2)
  (let ((s (get key1 key2))) ;get是数据导向程序设计中的二维表格操作
    (if s s the-empty-stream)))
```

数据库的维护

```
(define THE-RULES the-empty-stream) ;规则全流
(define (fetch-rules pattern frame) ;根据pattern取出需要遍历的规则流
  (if (use-index? pattern) ;判断pattern的car是不是symbol
      (get-indexed-rules pattern) ;取相应的分流
      (get-all-rules))) ;取全流
(define (get-all-rules) THE-RULES)
(define (get-indexed-rules pattern)
  (stream-append  ;两个分流合并
   (get-stream (index-key-of pattern) 'rule-stream)
   (get-stream '? 'rule-stream)))
```

数据库的维护

```
(define (add-rule-or-assertion! assertion);添加一条规则或断言
  (if (rule? assertion) ;(define (rule? x)(tagged-list? x 'rule))
      (add-rule! assertion)
      (add-assertion! assertion)))
(define (add-assertion! assertion)
  (store-assertion-in-index assertion) ;更新断言分流
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS ;更新断言全流
          (cons-stream assertion old-assertions))
    'ok))
(define (add-rule! rule)
  (store-rule-in-index rule) ;更新规则分流
  (let ((old-rules THE-RULES)) ;更新规则全流
    (set! THE-RULES (cons-stream rule old-rules))
    'ok))
```

数据库的维护

```
(define (store-assertion-in-index assertion)
  (if (indexable? assertion);判断assertion的car是否是符号或变量
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
               (get-stream key 'assertion-stream)))
          (put key ;put来自数据导向程序设计的二维表格操作
               'assertion-stream
               (cons-stream assertion
                            current-assertion-stream))))))
(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                 (get-stream key 'rule-stream)))
            (put key
                 'rule-stream
                 (cons-stream rule
                              current-rule-stream)))))))
```

框架和约束

```
(define (make-binding variable value) ;构造一个约束。框架里每个元素都是约束
  (cons variable value));cons出来的结果是否是表，取决于 value的形式
;约束的多种形式：
((? x) a), ((? z) . c), ((? y) a b),  ((? u) (a b)) ,((? k) ? y)
(define (binding-variable binding)
  (car binding))
(define (binding-value binding)
  (cdr binding))
(define (binding-in-frame variable frame);在框架中查找变量的约束
  (assoc variable frame))
(define (extend variable value frame);把新约束加入框架
  (cons (make-binding variable value) frame))
```

查询的语法分析

```
(define (type exp)
  (if (pair? exp)
      (car exp)
      (error "Unknown expression TYPE" exp)))
(define (contents exp)
  (if (pair? exp)
      (cdr exp)
      (error "Unknown expression CONTENTS" exp)))

(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))
(define (add-assertion-body exp)
  (car (contents exp)))
```

查询的语法分析

```
;exp中已经除去了 and ,or ,list-value
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps)) ;and的第一个模式
(define (rest-conjuncts exps) (cdr exps)) ;and的剩余模式
(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps)) ;or的第一个模式
(define (rest-disjuncts exps) (cdr exps)) ;or的剩余模式
(define (negated-query exps) (car exps)) ;not中的模式
(define (predicate exps) (car exps)) ;list-value中的谓词
(define (args exps) (cdr exps)) ;list-value中的谓词的参数
```

查询的语法分析

●关于规则的语法分析：

```
(define (rule? statement)
  (tagged-list? statement 'rule))
(define (conclusion rule) (cadr rule)) ;规则的结论
(define (rule-body rule)
  (if (null? (cddr rule))
      '(always-true) ;没有体的规则，结论总是成立。因此返回这个特殊的体。
      (caddr rule))) ;qeval会通过'always-true找到always-true过程
```

规则形如：
```
(rule (wheel (? person))
      (and (supervisor (? middle-manager) (? person) )
           (supervisor (? x) (? middle-manager))))
```

```
(define (always-true ignore frame-stream) frame-stream)
(put 'always-true 'qeval always-true)
```

查询的语法分析

●预处理:模式变量的替换

解释器读入一条规则或模式，会用query-syntax-process预处理，将其中的变量?x变换为 (? x)

```
; ?x -> (? x)  (job ?x ?y) -> (job (? x) (? y))
(define (query-syntax-process exp) ;替换exp中的变量
  (map-over-symbols expand-question-mark exp))
(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp))))
        ((symbol? exp) (proc exp)) ;如果是符号就看看是否变为 (? x)形式
        (else exp)))
(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        ;string=和substring是scheme基本过程
        (list '?
              (string->symbol
               (substring chars 1 (string-length chars))))
        symbol)))
```

查询的语法分析

●合一过程保证变量名唯一

一条规则的规则定义中可能用到别的规则。不同的规则在定义时，可能使用相同变量名。但是在合一操作时，不同规则里的变量名必须不同。

```
(assert! (rule (greater x1 y1)))
(assert! (rule (live-near ?x ?y ?a ?b)
               (lisp-value > ?x ?y)))
(assert! (rule (lisp-value > ?a ?b)   ;规则2
               (greater ?a ?b)))

(live-near x1 y1 a1 b1)
?x -> x1
?y -> y1
?a -> a1
?b -> b1
```

若不改规则2中的名字，则在规则2中，?a->x1， ?b->y1,和frame里面已经有的 ?a->a1 ?b->b1矛盾

查询的语法分析

●合一过程保证变量名唯一

一条规则的规则定义中可能用到别的规则。不同的规则在定义时，可能使用相同变量名。但是在合一操作时，不同规则里的变量名必须不同。

Q代表询问

```
(assert! (rule (append-to-form () ?y ?y)))
(assert! (rule (append-to-form (?u . ?v) ?y (?u . ?z))
          (append-to-form ?v ?y ?z)))

(append-to-form (a b) (c d) ?z)

?z -> (?u . ?z)    ;????

(append-to-form (a b) (c d) ?x)
?x -> (?u . ?z)    不如  规则中的 ?z -> ?z1
```

查询的语法分析

●合一过程保证变量名唯一

每次在apply-a-rule中用到规则时，都要对规则里的变量进行重新命名。重新命名的办法是在变量前面加个数字id,id是个全局变量，每用到一次规则id的值就加1。

```
(define rule-counter 0) ;全局id
(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)

(define (make-new-variable var rule-application-id)
  (cons '? (cons rule-application-id (cdr var))))
;变量名变成形如: (? 29 x)
查询: (same ?x ?x)
=>(same ?x-56 ?x-56)
```

查询的语法分析

- 合一过程保证变量名唯一

```
(define (rename-variables-in rule);为规则rule中的变量重命名
   (let ((rule-application-id (new-rule-application-id)))
     (define (tree-walk exp)
       (cond ((var? exp)
              (make-new-variable exp rule-application-id))
             ((pair? exp)
              (cons (tree-walk (car exp))
                    (tree-walk (cdr exp))))
             (else exp)))
     (tree-walk rule)))
```

驱动循环和实例化

● 系统反复读入表达式并立即预处理。如果是规则或断言，就加入数据库。否则就认为是查询。将查询和一个初始框架流（内含一个空框架）送给求值器 **qeval**。

●**qeval**对查询求值的结果是一个框架流，其中的变量约束能够匹配查询，且和作为**qeval**被调用时的参数的那个流不矛盾。返回结果框架流中的每个框架是一个查询的解。如果找不到匹配查询的办法，**qeval**返回空流。

空流和只有一个空框架的流不同。前者表示无解，后者表示查询本身就是解,无需对查询中的变量进行约束(或查询里面就没有变量)。

例如：有规则 **(rule (same ?x ?x))**则查询 **(same ?x ?x)**本身就是解

●驱动循环拿到**qeval**的返回值后，对每一个框架，根据框架中的变量的约束，把查询中的变量替换成其在框架中约束的值，就得到了一个可以输出的解(这叫将查询实例化)。驱动循环把所有可以输出的解组成一个流，一起输出。

驱动循环和实例化

```
(define (query-driver-loop) ;驱动循环
  (let ((q (query-syntax-process (read)))) ;表达式变量变换,?x->(? x)
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (query-driver-loop)) ;处理下一个输入表达式
          (else
           (display-stream ;显示最终结果流
            (stream-map
             (lambda (frame)
               (instantiate q ;对q进行实例化
                            frame ;一个frame对应一个解
                            (lambda (v f) ;处理解中未约束的变量
                              (contract-question-mark v))))
             (qeval q (singleton-stream '())));以
            ;qeval求得框架流，每个框架就是一个解
           (query-driver-loop))))) ;处理下一个输入表达式
```

```
(define (singleton-stream x)
  (stream-cons x the-empty-stream))
```

驱动循环和实例化

```
;实例化一个表达式(模式)
(define (instantiate exp frame unbound-var-handler)
;unbound-var-handler处理无法实例化的变量
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding ;binding形如 ((? x) . 4) ....
                 (copy (binding-value binding)) ;值也可能是变量
                 (unbound-var-handler exp frame))))
          ((pair? exp)
           (cons (copy (car exp)) (copy (cdr exp))))
          (else exp)))
  (copy exp))

(define (binding-in-frame variable frame);在框架中查找变量的约束
  (assoc variable frame))
```

驱动循环和实例化

```
(define (contract-question-mark variable)
;把unbound的变量变回原来的形式 variable 形如 (? x) 或者 (? 4 x)
分别变成了 ?x 和 ?x-4
   (string->symbol
    (string-append "?"
      (if (number? (cadr variable))
          (string-append (symbol->string (caddr variable))
                         "-"
                         (number->string (cadr variable)))
          (symbol->string (cadr variable))))))
```

求值器

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)));type就是取car
    (if qproc
        (qproc (contents query) frame-stream);复合查询
        (simple-query query frame-stream))));简单查询
```

如果 (type query)不是 and,or,not,lisp-value或always-true，则认为query是简单查询，不是复合查询

```
;qproc是 disjoin, conjoin, negate lisp-value或 always-true
(put 'or 'qeval disjoin) ;处理or
(put 'and 'qeval conjoin) ;处理 and
(put 'not 'qeval negate) ;处理not
(put 'lisp-value 'qeval lisp-value) ;处理 lisp-value
(put 'always-true 'qeval always-true) ;处理查询总是真的情况
```

简单查询

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed ;组合两个流
      (find-assertions query-pattern frame);符合断言的流
      (delay (apply-rules query-pattern frame)))) ;符合规则的流
   frame-stream))
```

frame-stream里面的每个frame都是求了一部分的一组解Si，进一步求完整解的时候必须以Si为基础，不得和它矛盾。一个Si可以扩展出多个更进一步的解（不一定是完整解）

Si1,Si2... Sin

所有的 Sij组合起来形成simple-query返回值流。返回值是空流表示无解。

复合查询and

```
(define (conjoin conjuncts frame-stream) ;conjuncts是去掉and剩下的
  (if (empty-conjunction? conjuncts) ;空表
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts)
                      frame-stream))))
```

先找出and的第一项的所有解（框架流）FS1，然后以这些解为前提条件去求第二项的解，得到所有的一个同时满足第一项和第二项的解（框架流）FS2，然后以FS2为输入去求FS3。返回值是空流表示无解。

复合查询or

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed ;交错地合并两个流
       (qeval (first-disjunct disjuncts) frame-stream)
       (delay (disjoin (rest-disjuncts disjuncts)
                       frame-stream)))))
```

返回值是空流表示无解。

为何要交错合并？

复合查询not

```
(define (negate operands frame-stream);operands里面已经去掉了not
;operands形如((job ?x (computer programmer)))
  (stream-flatmap
   (lambda (frame)
     (if (stream-null? (qeval (negated-query operands)
                              (singleton-stream frame)))
         (singleton-stream frame)
         the-empty-stream));如果frame使得not后面的条件成立，则无解
   frame-stream))
```

返回值是空流表示无解。

```
(define (singleton-stream x)   ;结果:(x)
  (stream-cons x the-empty-stream))

(define (negated-query exps) (car exps)) ;not中的模式
```

exps形如: (not (job ?x (computer programmer)))

复合查询lisp-value

```
(define (lisp-value call frame-stream)
  (stream-flatmap ;call形如 (> ?x 2000)
   (lambda (frame)
     (if (execute
          (instantiate ;要把谓词的参数按照frame实例化
           call
           frame
           (lambda (v f) ;如果有参数不能实例化，则无法执行谓词，报错
             (error "Unknown pat var -- LISP-VALUE" v)))
          (singleton-stream frame)
          the-empty-stream))
   frame-stream))

(define (execute exp) ;求值谓词并执行它
  (apply (eval (predicate exp) (scheme-report-environment 5))
         (args exp))) ;程序开头 (require r5rs)。 args就是取cdr
```

特殊形式alway-true

```
(define (always-true ignore frame-stream) frame-stream)

(put 'always-true 'qeval always-true)
```

描述总为真的查询。因此参数ignore无用

always-true在查询和永真规则（没有体的规则）匹配时会被调用

永真规则例子：(rule (same ?x ?x))

分析永真规则的体返回 '(always-true)

```
(define (rule-body rule)
  (if (null? (cddr rule))
      '(always-true)
      (caddr rule)))
```

求值器

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)));type就是取car
    (if qproc
        (qproc (contents query) frame-stream);复合查询
        (simple-query query frame-stream))));简单查询
```

如果 (type query)不是 and,or,not,lisp-value或always-true，则认为query是简单查询，不是复合查询

```
;qproc是 disjoin, conjoin, negate lisp-value或 always-true
(put 'or 'qeval disjoin) ;处理or
(put 'and 'qeval conjoin) ;处理 and
(put 'not 'qeval negate) ;处理not
(put 'lisp-value 'qeval lisp-value) ;处理 lisp-value
(put 'always-true 'qeval always-true) ;处理查询总是真的情况
```

通过模式匹配找出断言

```
(define (find-assertions pattern frame)
  (stream-flatmap (lambda (datum);datum是断言
                    (check-an-assertion datum pattern frame))
                  (fetch-assertions pattern frame)))
; fetch-assertions返回适合的断言流，缩小了断言的查找范围。frame参数无用


;简单模式匹配器 check-an-assertion
(define (check-an-assertion assertion query-pat query-frame)
  (let ((match-result
         (pattern-match query-pat assertion query-frame)))
    (if (eq? match-result 'failed)
        the-empty-stream ;无解返回空流
        (singleton-stream match-result))))
```

通过模式匹配找出断言

```
(define (pattern-match pat dat frame) ;返回值是个frame，或'failed
  (cond ((eq? frame 'failed) 'failed) ;结束递归
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame)) ;(? x)
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                        (cdr dat)
                        (pattern-match (car pat)
                                       (car dat)
                                       frame)))
        (else 'failed)))
```

extend-if-consistent 在pat取值为dat和frame不矛盾的情况下，将这个pat到dat的约束扩充进frame

通过模式匹配找出断言

●扩展frame：

;如果var取值为dat和frame不矛盾，则扩展frame。 var 形如 (? x) ,dat可以是表，也可以是单个值，如 5 , Mike

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match (binding-value binding) dat frame)
;找到的binding可能是 ((? x) ? y)形式的，那么就要对 (? y)进一步和 dat 去看是否match
        (extend var dat frame))))  ;var在frame里面找不到，那么肯定可以把 (var . dat) 加进去
```

```
(define (extend variable value frame);把新约束加入框架
  (cons (make-binding variable value) frame))
```

通过模式匹配找出断言

●pattern-match的执行过程示例：

```
(assert! (married Minnie Mickey))
(married Minnie ?x)

;in pattern-match, pat=(married Minnie (? x))
dat=(married Minnie Mickey) frame=()
;in pattern-match, pat=married dat=married frame=()
;in pattern-match, pat=(Minnie (? x)) dat=(Minnie Mickey)
frame=()
;in pattern-match, pat=Minnie dat=Minnie frame=()
;in pattern-match, pat=((? x)) dat=(Mickey) frame=()
;in pattern-match, pat=(? x) dat=Mickey frame=()
;in pattern-match, pat=() dat=() frame=(((? x) . Mickey))
```

具有带点尾部的模式

●如果一个模式里包含一个圆点，后跟一个模式变量，则该模式变量和正在匹配的数据表里剩余部分匹配，而不是和数据表里的下一个元素匹配。

- 输入 (computer ?type), (read)处理后的表结构是:

```
(cons 'computer (cons '?type '()))
```

- 输入 (computer . ?type), (read)处理后的表结构是:

```
(cons 'computer '?type)
```

此种情况下 ?type就是一个cdr指针,因此做匹配时，也要与数据表的cdr指针匹配。

(cons 'computer '?type)与 (computer programmer trainee)匹配，则会将 ?type 匹配到 (programmer trainee)

规则和合一

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
                    (apply-a-rule rule pattern frame))
                  (fetch-rules pattern frame)))
;fatch-rules缩小了要考察的规则的范围
```

首先在给定的frame条件下，将pattern和rule的结论部分做合一，得到一个扩充的框架F，再以F作为前提条件，去求值(匹配)规则的体。

使用规则前，要先对规则里的变量进行重命名，以确保不会和别的规则重名。

规则和合一

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)));规则变量重命名
    (let ((unify-result
           (unify-match query-pattern ;unify-match是合一
                        (conclusion clean-rule)
                        query-frame)))
      (if (eq? unify-result 'failed);模式和规则的结论合一失败
          the-empty-stream
          (qeval (rule-body clean-rule) ;进一步求值规则的体
                 (singleton-stream unify-result))))))
```

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)));type就是取car
    (if qproc
        (qproc (contents query) frame-stream);复合查询
        (simple-query query frame-stream))));简单查询
```

简单查询

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed ;组合两个流
      (find-assertions query-pattern frame);符合断言的流
      (delay (apply-rules query-pattern frame)))) ;符合规则的流
   frame-stream))
```

为何要delay?

简单查询

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed ;组合两个流
      (find-assertions query-pattern frame);符合断言的流
      (delay (apply-rules query-pattern frame)))) ;符合规则的流
   frame-stream))
```

考虑：

```
(assert! (married Minnie Mickey))
(assert! (rule (married ?x ?y)
               (married ?y ?x)))
(married Mickey ?who)
```

不delay,apply-rules就死循环了

apply-rules->apply-a-rule->
qeval->simple-query->apply-rules

规则和合一

```
(define (unify-match p1 p2 frame) ;将模式p1和p2合一
  ;规则的结论部分也可以看做是一个模式，因其可能包含变量
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
             ;如果将p1约束到p2和frame不矛盾，则扩展frame
        ((var? p2) (extend-if-possible p2 p1 frame))   ; ***
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1)
                                   (car p2)
                                   frame)))
        (else 'failed)))
```

规则和合一

```
(define (extend-if-possible var val frame)
 ;如果将var约束到val和frame不矛盾，则扩展frame
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
           (unify-match
            (binding-value binding) val frame))
          ; (binding-value binding) 可能是变量，如果是，就要进一步匹配
          ((var? val)   ;frame没有对var的约束且val是变量
           (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match
                  var (binding-value binding) frame)
                 (extend var val frame))))
          ((depends-on? val var frame);val不是变量则需要检查其是否依赖var
           'failed)
          (else (extend var val frame)))))
```

不能处理“?x 和 包含?x但不是?x的一个表达式等价，?x该如何赋值” 这个问题

规则和合一

depends-on?查看exp的值是否和var相关。exp里面如果包含var的出现，则其值和var相关。如果exp里面包含 y，且y在frame被约束到 var，则exp也和var相关

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
           (if (equal? var e)
               true
               (let ((b (binding-in-frame e frame)))
                 (if b
                     (tree-walk (binding-value b))
                     false))))
          ((pair? e)
           (or (tree-walk (car e))
               (tree-walk (cdr e))))
          (else false)))
  (tree-walk exp))
```

流操作

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2);delayed-s2是延时求值对象，里面放着一个流
      (cons-stream
       (stream-car s1)
       (stream-append-delayed (stream-cdr s1) delayed-s2))))


(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2) ;delayed-s2是延时求值对象，里面放着一个流
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))
```

流操作

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave-delayed
       (stream-car stream)
       (delay (flatten-stream (stream-cdr stream))))))
```

;stream中的每个元素都是流，要让拉平的结果里，每个流里的元素都有机会出现