Stream 和 Object

——面向对象的并行计算模型及其语言探讨(2)

Yoshida Kaul 和 近山隆

(日本新一代计算机技术开发研究机构，ICOT)

4. 扩充文法

为了严格忠实地表示Stream 计算模型，我们定义了基本文法。但是，对于大型的并行问题，往往需要更简洁、更抽象的文法描述。为了提高描述力，简洁而安全地编写程序，特对基本文法作如下扩充：

- 导入简易（宏）的描述；
- 从Stream变量到Channel变量的语义扩充；
- 对Stream完整化的支持；
- 导入挥发对象。

4.1 简易描述（宏）

为了能简易地书写程序，要提供种种简易描述。如同基本语法由函数表达式组成一样，这些简易描述在某表达式（串）展开时，估价值都指定为输入端、输出端或空。

4.1.1 算术/逻辑演算宏 针对原始对象的演算操作（如整数的四则运算等），我们定义如下宏群。它们的估价值几乎都表示到运算结果对象的Stream输出端。

如同前面介绍的基本表达式一样，我们也可以组合这些宏来描述更为复杂的演算式。例如：3 + 5 = 8 的算式，展开为3:add (5,∧sum)::, sum:eq(8, ∧TRUE)::。其值表示为真对象'true的Stream 输出端 TRUE。

4.1.2 伪变量$self 对象本身即下一世代的Stream端点，由伪变量 $self指定，其语义因场合不同而不同。

- 输出端的访问：在输出端的场合出现时，意味着应访问最新的自身；在省略消息发送式的输出端时，意味着指定输出端为伪变量$self。
- 输入端的更新：在输入端的场合出现时，意味着它将成为之后的自身。

〔**关于自身存取顺序的规则**〕 存取自身的顺序，包括语法内出现的伪变数$self、传送消息的目标对象省略以及关于槽存取的宏出现等情况。根据它们出现先后顺序，可以确定自身存取顺序。具体方法如下：

- 对于方法，从事件到动作串；
- 对于方法的动作串（‹Actions›部分的串），从左到右；
- 对于消息的参数串，从左到右；
- 对于宏展开，从里向外。

按构成方法的表达式的估值顺序，给定自存取顺序。例如：

```
:foo (!a, ∧X)—→
    :foo1(!b, !c, ∧Y)) = $self,
    $self:foo2(Y, ∧Z) = $self,
    X+Z=!d
```

该方法与下面的方法等价。其中"!"‹SlotName›表示槽存取宏，"→"表示继续宏。如后所述。

```
:foo(A, ∧X) —→
    $self:get-outlet(a, ∧A)
          :get-outlet (b, ∧B)
```

```
:get-outlet(c, ∧C)
:foo1 (B, C, ∧Y)
:foo2 (Y, ∧Z)
:set-outlet (d, ∧sum) = $self::,
X:add(Z, ∧sum)::.
```

4.1.3 **世代下降宏** 为了便于书写世代下降，提供如下的继续宏及终结宏。

```
⟨Method⟩ ::=
  ⟨EventOnly⟩ ⟨DescendingMacro⟩ ⟨Action⟩
{"," ⟨Action⟩}
  ⟨EventOnly⟩ ::= ⟨ReceivedMessage⟩ | ⟨Dete-
ctingClosed⟩
  ⟨Received-Message⟩ ::=":" ⟨MessagePatt-
ern⟩
  ⟨DetectingClosed⟩ ::=":"
  ⟨DescendingMacro⟩ ::= ⟨Succession⟩ | ⟨Termi-
nation⟩
  ⟨Succession⟩ ::="—→"
  ⟨Termination⟩ ::="—|"
```

继续宏 (succession)：继续宏如图 7 所示，是增加下列动作的宏。

1. 世代下降。

2. 现世代中，接受了消息后，将消息后的接口Stream的输入端∧Rest联结到下一世代的Stream末尾的输出端Last。

例如，方法：m—→：do展开如下：

```
:m=Rest|⇐∧self
        self:do=∧Rest
```

终结宏 (termination)：终结宏如图 8 所示，是增加下列动作的宏。

1. 世代下降。

2. 现世代中，接受了消息，生成sink对象并与处于消息接受之后的界面Stream的输入端∧Rest连接。

3. 流向下一世代的Stream末端为输出端Last，我们向Last传送终结消息 $ terminate之后，闭锁之。

例如，方法：m—|：do展开如下：

```
:m=Rest|⇐∧self
    ##sink, new(∧Rest)::
    self, do, '$terminate'::
```

4.1.4 **实例生成宏** 所谓实例生成宏，意味着向指定类传送实例生成消息，并意味着指向所生成实例的Stream输出端。例如，#counter展开为##ocunter:new(∧counter)，表示到类counter的例示输出端 counter。

4.1.5 **槽存取宏** 为了便于存取槽，可提供输入端槽存取宏@slotname以及输出端槽存取宏!slotname。槽存取宏和伪变量 $self一样，语义因出现的场合不同而异。

对于输出端槽存取宏：

• 指定输出端部分，则表示槽访问。

例如，!n=∧N 展开后为$self:get-outlet(n, ∧slot)，slot=∧N

• 指定输入端部分，则表示槽更新。

例如，N=!n展开后为$self:set-outlet(n, slot)，N=∧slot。

4.1.6 **更新传送消息的目标对象宏** 在语法内，往往需要分割描述关于槽和自身的顺序动作串，例如，

```
:foo→!a:m1(∧X)=!a::,
   …⟨actions related to X⟩ …
    !a:m2=!a::.
```

向槽传送了某消息之后，在与该结果相关联的动作（含条件分支）的后面，还有向该槽传送其它消息的动作时，往往特别容易忘记自己或槽的更新描述。

于是，有下列形式：

```
:foo→!a:m1(∧X),
    …(actions related to X)…
        !a:m2.
```

类似地，关于自身存取及槽存取，如果把消息传送后隐含地认为是最新的自己或槽，则语法中出现的顺序可以直接反映这些世代。

消息传送式中的传送目标更新宏，对应于目标对象最初给出的输出端种类，自动地支持消息传送目标对象的更新。

程序中，对于直接传送消息表达式，即使不指定，其它的宏展开为消息传送表达式时，该宏也适用。例如，!n+1=!n 展开为

!n:add(1, ∧sum)::, sum =!n::, 使用更新目标对象的宏，则为!n:add(1,∧sum) =!n::, sum=!n::。

4.2 Channel变量

以上各节中，变量用来表示Stream(链)。我们使用Stream变量来描述如下简单的问题：

"解答者（X，Y）依据各种策略求解问题(P)，然后把两人的各个枚举解答(A)综合起来"。综合的方法有两种：一是合并：将两个解答者的解答以任意顺序综合。表示如下：

```
:consult(∧P, ∧A, ∧X, ∧Y) —→
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P=∧P1=∧P2::, A=∧A1=∧A2::.
```

解答者X和Y分别：

• 通过Stream P_1或P_2，独立地讨论问题P；

• 以解答为消息，并向StreamA_1或A_2传送，且合并为StreamA。这里，出现多个输出端变量，用Stream的合并演算表示如下：

```
:consult(∧P, ∧A, ∧X, ∧Y)—→
    X:solve (P, A), Y:solve(P, A).
```

第二种方法是连接：X的解答在Y的解答之前先综合，表示如下：

```
:consult(∧P, ∧A, ∧X, ∧Y)—→
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P=∧P1=∧P2::, A\∧A1\∧A2::.
```

所给解答者的Stream，不同于前一种情况，不同之处为：连接传送解答的Strame A_1和A_2为StreamA。如果用标以序数的输出端变量表示Stream的连接演算，则有：

```
:consult(∧P, ∧A, ∧X, ∧Y, ∧Z)—→
    X:solve(P, A$1), Y:solve(P, A$2)
```

从这个例子，我们可以看出：在以前使用Stream变量的程序设计中，大部分工作花费在如何利用合并及连接，由链来构造半序集。如果将变量的语义从Stream(链)扩充为Channel(半序集)，那么可以把半序集合全体看成一个整体，如：例中的"给解答者一个问题P"或"综合全部的解答为A"等等。这样程序员就可以专心做"对于什么对象作什么"的描述工作，而不是在"如何去连接Stream上花很多时间。注意点从Stream转向对象。

综上所述，可以约定如下Channel变量指定Channel。

1个或1个以下的输入端变量：带"∧"头部的变量名（如：∧X）表示输入端。

0个或0个以上的非顺序输出端变量：仅用变量名（例如：X）表示，代表应合并的Joint输出端。

0个或0个以上的顺序输出端变量：用加上"$"和序号的变量名（如：X$1）表示。表示按序数升序连接的Joint输出端。序数为正整数时，不必从1开始，也不必连续(如:X$1，X$2，X$3)。例如，表达式X=∧$P_1$=∧$P_2$\∧$S_1$\∧$S_2$::表示如图19所示的Channel。图中有输入端变量∧X一个，非顺序输出端变量X两个，顺序输出端变量为X$1，X$2，X$3。

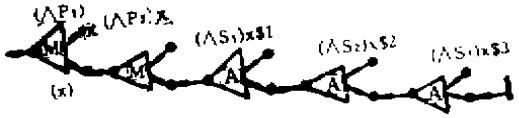

图19 Channel变量

4.3 隐式的Stream完整化

基本语法中，为了防止Stream不完整化所造成的死锁发生，促使计算的结束，给出了以下两个规则：

• Stream变量（输入端变量和输出端变量）应成对出现。

• 方法的动作串（<Actions>部分）应指定空表达式。

但是，为了Stream的完整化，程序员常常要闭锁输出端，从而加重了程序员的负担。前一节，我们将变量的语义从Stream扩充到Channel，已经出现了任意多个输出端，这个问题更显突出。为了便于书写安全的程序，将语法扩充如下：

输入端/输出端(表达式)的闲置：对于方法的动作串（〈Action〉部分），要指定输入端表达式和输出端表达式。各变量表示Channel，即使出现1个以下的输入端和任意个非顺序输出端以及顺序输出端也行。

自动完整化：闲置的非空表达式能自动地完整化。所谓自动完整化，即指支持输出端的自动闭锁、输入端的自动回收、对象的自动结束。

4.3.1 **输出端的自动闭锁** 闲置的输出端的自动闭锁，适用如下情况：

- 动作串所闲置的输出端表达式的结果；
- 输出端不出现的Channel的输出端；
- 槽更新时的当前槽；
- 对象结束时的各输出端槽；
- 对象初始化时的各输入端槽的输出端。

4.3.2 **输入端的自动回收** 自动地回收被闲置的输入端，即生成sink对象并与之连接，可适用如下情形：

- 动作串所闲置的输入端表达式之结果；
- 输入端不出现的Channel的输入端；
- 对象结束时的Channel的输入端；
- 对象结束时的各输入端槽；
- 对象初始化时的各输出端槽的输入端。

4.3.3 **对象的自动结束** 多数对象在检查出它的接口流闭锁时结束。但这个规定常常被遗忘，可以采用上界类定义闭锁检出便立即结束的方法。即：::⟶。这是可行的。

4.4 挥发对象

对象的各世代是：1. 根据观测事件激活各世代；2. 激活的同时，下降到下一世代。前者，相当于条件分支，后者相当于循环。也就是说，对象具有条件处理器的功能。但是，如果每个条件判定都定义一个类，不仅因此要多定义许多零碎类，而且容易使程序的语法过于琐碎。条件判定前的语法和条件判定后的语法中，Stream变量环境是互相独立的，为了使它们发生联系，程序员要从前者向后者发送消息，这无疑加重了负担。为了解决这些问题，我们引入挥发对象的概念。

挥发类：在一个方法内，称临时定义的类为挥发类（volatile class）。一个方法内可以定义任意多个挥发类。挥发类，除类名不同外，和一般定义的外部类几乎没有区别。所谓不同是指：

- 外部类具有从外部访问的类名，借助类名，其它类（包括挥发类）可以对该类进行存取或继承。
- 挥发类不具有从外部访问的类名。

可以任意重嵌套地定义挥发类。即某个挥发类的方法内可以定义其它的挥发类。这和一般的过程型语法中条件判定语句或循环语句的嵌套完全一样。

挥发对象：称挥发类的例示为挥发对象（volatile object）。

生成者对象：生成了挥发对象的对象，即执行（定义）挥发类方法的对象为生成者对象（creator object）。当挥发类的定义嵌套时，对于内侧的挥发对象，定义它的外侧的挥发对象是其生成者对象。

流向生成者对象的Stream通过指定伪变量$ creator，能由挥发对象存取。

作用域：槽名分别出现于各个方法中，并可以跨越世代，存取对应的Stream。而变量名只能在方法中使用，与Stream一一对应。这样，称名字的有效范围为名字的作用域（name scope）。

根据挥发对象和生成者对象作用域的关联，可把挥发对象分为不变挥发对象和可变挥发对象两种。在详细讨论它们之前，我们先来看看两者共同的类定义和对象生成的指定方法。

4.4.1 **挥发对象的生成** 挥发对象生成表达式由挥发对象的界面Stream的指定和挥发

类的定义组成。挥发对象生成表达式是空表达式。

‹VolatileObjeet Creation› ::=
  ‹ImmutableVolatileObjectCreation› | ‹MutableVolatileObjectCreation›
‹ImmutablVolatile ObjectCreation› ::=
  ‹Interface› "?" ‹Immutable Volatile ClassDefinition›
  ‹MutableVolatileObjectCreation› ::=
  ‹Interface› "⇒" ‹MutableVolatileClassDefinition›
‹Interface› ::= ‹InletExpression› | ‹OutletExpression›
‹Immutable VolatileClassDefinition› ::=
    "(" [‹SuperClassDefinition› ";"]
    ‹Method› {";" ‹Method› } ")"
‹MutableVolatile ClassDefinition› ::=
    "(" [‹SuperClassDefinition› ","]
      [‹InletSlotDefinition› ";"]
      [‹Outlet SlotDefinition› ";"]
      ‹Method› {; ‹Method› } ")"

基本生成（以输入端为界面）：首先，我们指定在‹Interface›部分生成的挥发对象的界面Stream输入端。例如，指定

    ∧Huger? (
        :'true ⟶ :eat;
        :'false ⟶ :sleep
)

则，被生成的挥发对象从输入端∧Hunger接受消息'true或'false，由此向自身传送消息eat或sleep。

扩充（对于输出端传送'谁消息'）：包含前述算术宏的很多宏是以输出端为其估价值。它的算术结果往往因某种原因要求条件分支。为了便于描述，提供如下支持：在指定为输出端的情况下，隐含地向输出端传送"谁消息"，Who-are-you (Who)，并设输入端∧Who为界面。例如，分别处理奇数和偶数时，指定 (Xmod2==0)? (……)，则相当于：

    (X mod 2==0):Who-are-you(Who)::,
∧Who?(……)。

4.4.2 **不变挥发对象** 不变挥发对象（immutable voletile object）是仅限于一代的对象。具有和生成者对象相同的作用域（图20），即：

变量名的作用域同一：不变挥发类中出现的变量名和该生成者对象的方法内出现的同一变量各意味着同一Channel。

生成者自身：作为不变挥发对象，生成者对象是自身。即，作为不变挥发对象，自身及槽群意味着生成者对象的内容。伪变量$ creator和$ self能作为同义词使用。

不变挥发对象的生成：不变挥发对象生成表达式的‹Interface›部分指定的输入端（∧I. V. obj）被连接到生成的不变挥发对象的第0世代。

所谓同一作用域，亦即共有作用域。不变挥发对象生成时，除了流到生成者对象本身的Stream之外，还包括传递到作用域对象（scope object）的Stream。生成者对象也类似，要向作用域对象传送Stream。两者带有的Channel群由该作用域对象管理。

作用域对象：一般对象用槽名管理槽群。同样，作用域对象是用变量名管理Channel群的对象。

作用域共享：不变挥发对象在实现时决定由该界面Stream接受哪个消息，并决定存取哪个Channel。

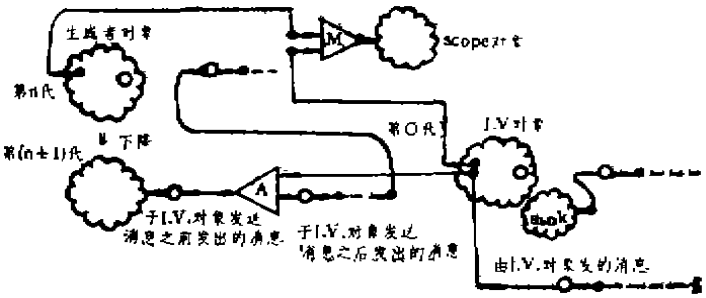

图20 不变挥发对象和生成者对象

例如，访问和生成对象之间的公共Channel变量，对于某方法，设它共出现3个输出端，对于别的方法，出现2个输出端。不

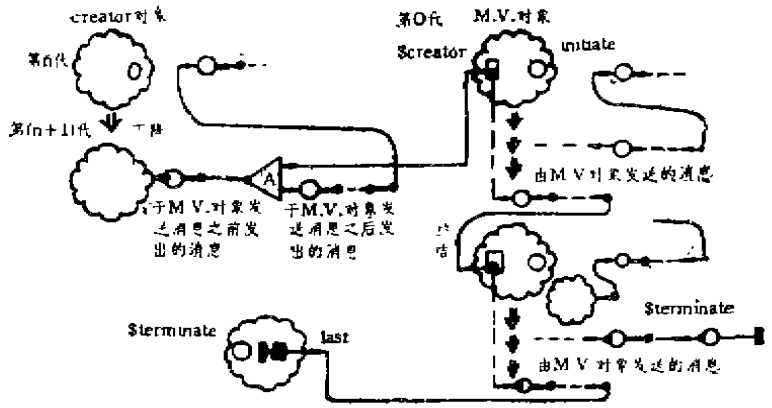

图21 可变挥发对象和生成者对象

论哪种情况，相关的Channel (Stream) 应全部完整化。

不变挥发类的定义可以嵌套。根据各种可能的情况，事先编写好各种代码。但这样一来，就偏离了代码优化原则。

综上所述，我们应尽可能使物理代码最优化，Channel完整化。引入作用域就是为了解决这个问题。即具有相同的作用域的对象之间共有一个作用域对象，而且只能在实现时决定的事应该在实现时决定。

作为不变挥发对象自身：根据语法中出现的顺序，生成者对象的自存取顺序应遵守如下规定：

·不变挥发对象连接、分解流向生成者对象自身的Stream并传送消息，在向不变挥发对象自身传递所有消息后，闭锁。

·保持生成者对象，用于对该可变挥发类定义后所出现的自己本身的存取。

不变挥发对象主要用于条件分支。

4.4.3 可变挥发对象 可变挥发对象 (mutable volatile object) 具有自身的世代，而且，生成者对象是独立对象，生成者对象具有独立的作用域（图21)。即：

变量名的作用域独立：可变挥发类定义中出现的变量名和其生成者对象的方法内出现的同一变量名代表不同的Channel。

生成者是槽之一：可变挥发对象具有表示自身的内部状态的槽群。生成者对象不过是一个以伪变量 creator 为槽名的一个输出端槽。另一方面，伪变量 $ self 表示可变挥发对象自身。这样，伪变量 $ creator和 $ self 的语义不同。

可变挥发对象的生成：可变挥发对象和外部对象完全一样。生成后，隐式地传送初始化消息。初始化消息的方法也可同样定义。

可变挥发对象生成表达式的〈Interface〉部分所指定的输入端（∧M. V. Obj）是紧接该初始化消息之后的Stream的输入端。可变挥发对象也在其生成时，向生成者对象自身流入Stream。

作为可变挥发对象的自身：根据语法中出现顺序，生成者对象的自存取顺序与不变挥发对象的完全相同，遵守如下规定：

·可变挥发对象连接、分解流向生成者对象本身的Stream并传递消息，将其保持于输出端槽 $ creator。

可变挥发对象结束时，输出端槽 $ creator，与其它输出端槽一起闭锁。

·保持生成者对象，用于对该可变挥发类定义后出现的自身的存取。

可变挥发对象主要用于循环。

下面，我们来看一个应用挥发对象的程序例子：

应用例2 （素数生成）

```
class, prime                                    %1
:primes(∧Max, ∧Ps)->                           %2
3=∧x                                            %3
:generate(x, Max,Ns)                            %4
#sift:do(x, ∧NS, Ps:n(2):n(x))                  %5
:generate(∧x, ∧Max, ∧Ns)->                      %6
((x+2=Newx)<Max)?(                              %7
  :/true->                                      %8
  :generate(Newx, Max, Ns:n(newx));
                                                %9
  :/false-> %end%                               %10
```

```
).                                          %11
end.                                        %12
class sift                                  %1
:do(^V, NS, ^PS) →                          %2
  s:initialize(V,Ps)=^Ns                    %3
^s⇒(                                        %4
  out, me, Next, to-next, prmes:            %5
:initialize(^V, ^Ps)→                       %6
V=|me, O=|next, Ps=|primes;                 %7
:n(^x)→                                     %8
((x mod|me)==0)?(                           %9
  :true→;                                   %10
  :false→                                   %11
  (|next==0)?(                              %12
  :true→                                    %13
    x=|next, Ns=|to-next,                   %14
    #sift:do(x, ^Ns,|Primes:
                        n(n(x));            %15
    :false→                                 %16
        |to-next:n(x)                       %17
  )                                         %18
  )                                         %19
)                                           %20
end.                                        %21
class test
  :test→
#prime:primes (20, Ps), :nop(^Res)
:nop(Res)→.
end.
```

图22 生成素数程序

这是一个生成不超过所指定最大值(max)的素数序列的程序。

1. 发送2后，生成以3开始的奇数序列。通过筛选最终得到素数序列。

2. 每找到一个新的素数，便做一个筛子。首先，以3为筛子。

3. 筛选是用筛子去除流入的数。

·能除尽（自己的倍数）时，什么也不做；

·不能除尽时，看筛子是否是当前最大素数：

——是最大素数时，当前流入的数是素数，将它做一个筛子，并设置为"下一个筛子"。

——否则，将当前流入的数交给"下一个筛子"。

如此反复，直到筛子里没有流入数为止。

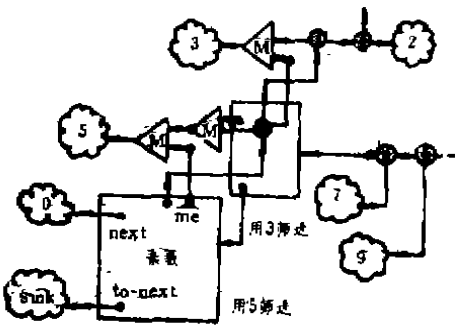

图23 生成素数的"筛子"串

类sift的第4行～20行定义了一个可变挥发对象（M_1），其中第9行～19行定义了一个不变挥发对象(I_1)，第12行～18行定义了另一个不变挥发对象（I_2）。可变挥发对象的外侧（2行～3行）出现的变量V，Ns，Ps和内侧（4行～20行）出现的变量V，Ns，Ps是独立的。通过消息initialize传送到内侧。在同一作用域内，第7、9、14、15、17行出现的槽存取都是最外侧的，即可变挥发对象（M_1）的槽。

5. Stream和Object的融合

下面，我们小结一下Stream和Object融合所起的作用。

5.1 Stream计算和并行计算机

Stream计算的提出不仅是为了进一步构造出更精致的并行计算模型，而且为构造更实用的并行计算机系统带来了希望。

Stream由输入端、输出端两端点定义，输入端一个，输出端一个，换句话说，意味着一个专门读，一个专门写的单一访问、单一代入。

单一代入：如果决定了写者，单一代入什么时候写都行，即意味着应唯一确定执行的顺序和数据存取的语义。在数据流语言中

广泛采用的技术是促进数据和控制分离和演算功能分布化。

单一访问：其读者如能读完信息，由于没有另外的读者，所以可以抛弃它。即为了使计算机系统能持续运行，执行时要回收无用单元（GC）。对于并行计算机的构造，回收无用单元已成为最重要的研究课题之一。与多重访问所产生的问题相同，无用单元长时间堆积时，使整个系统停机来处理GC是完全不现实的。计算机规模愈大，这个问题愈加突出。从而，单一访问发挥的作用也愈大。

5.2 作为不动点的对象

我们从事件顺序出发，比较微观地探讨了并行系统。对象的一生定义为世代的链。视对象的一生为事件的非连通图，则可视各世代为该图的不动点或循环点（$x=f(x)$ 的 x）。对象可以看成是同构子图迭加的图。

从人类社会来看，这是十分自然的。构成人体的细胞时刻都在再生，人的相貌无时无刻不在改变。某一时刻的那个人和一秒钟后的那个人在物理上不同。那个人表示为各个时刻的一些因果产物的集合体。从生到死，对一个人的识别是离不开与这个人相关联的因果链的。根据Stream来描绘对象，真实地反映了这种情况。

同样，微观地定义并行系统的计算模型或语言还有另外一些相关的研究途径。如半序集模型〔Pratt86〕，过程代数语言群〔Milner80、Milner83，Winskel84，Bergstra 84，Winkowski87，Bckker87〕。并行逻辑语言群〔Shapiro83a，Clark84，Ueda85〕，并行函数语言群〔Broy86〕等。它们导入了各自的不动点和递归代数式、递归谓词定义、或递归函数定义的宏描述，但仍都是基于对象的抽象〔Staple83，Winkowski87，Milner 83，Bakker87，Shapiro83b，Broy86〕。

6 .结束语

关于并行问题，本文提出的基本思想是极其朴素的。“并行系统是半序集合，并行程序设计是设计‘事件图’。以不动点（循环点），迭图即成为对象”。这种基本思想也是贯穿于AIUM的基本设计观点。

AIUM包括并行计算机系统、并行程序设计语言、并行排错环境等等，限于篇幅，这里就不一一介绍。AIUM作为日本ICOT第五代机核心语言的组成部分，其本身尚在不断探讨和发展中，但是其中的设计思想对于计算机工作者特别是正在研究新一代计算机语言的人们是很有启发意义的。我们介绍这方面的内容，谨供同行参考，并和大家商榷。（全文完）。

〔樊　红　何克清编译〕

主要参考文献

〔Bakker87〕J. W. de Bakker, J. J. Ch. Meyer, and E. R. Orderog, Infinite Streams and Finite Observations in the Semantics of Uniform Concurrency, Theoretical Computer Science 49, NorthHolland, 1987

〔Bergstra84〕J. A. Bergstra and J. W. Klop, Process Algebra for Synchronous Communication, Information and Control 60, 1984

〔Birkhoff79〕G. Birkhoff, Lattice Theory, American Mathematical Society, Colloquium Publications, Vol. 25, Third edition, 1979