94,21(3) 通信系统 交互作用

1-8,34

计算机科学 1994 Vol. 21 №. 3

TN914

交互作用之基础

Robin Milner

Milner, R 李角年

A

摘 要 这篇文章是 R. Milner 的图灵奖演讲，他回顾了自己在顺序范型的指导下寻找并发的基本模型(通信系统演算 CCS)的艰难历程，并在最后概述一个新的基本并发演算 π-演算。全文思路开阔，追本溯原，深入浅出，读来颇受启迪。

——译校者

我非常荣幸地获得了以 Alan Turing 命名的图灵奖，也许图灵会为自己的母校——剑桥皇家学院所栽培的人感到高兴。1956 年，我在皇家学院写下了我的第一个计算机程序(在 EDSAC 机上)。当然 EDSAC 已成为历史，但惭愧的是这并没吸引我步入计算界，我有四年没接触计算机。1960 年，那时我是一名教师，我想计算机可能比学生更好对付，就申请了伦敦 Ferranti 的一份工作，那正是柏伽索斯时期。在采访中，有人问我是否愿意献身计算机事业，这个吓人的问题我从未想过。我曾有幸同计算机科学一道成长，现在我仍从事这一领域的探索。

这次获奖给了我一个不同寻常的机会，希望能允许我从个人的观点回顾一下学术研究的历程。我想我应当抓住这个机会，因为在我的兴趣中有个思考了二十年之久的线索。如果你记住这是一条个人亲身的线索的话，那么描述这种经验肯定能增长见识，因为科学就是由许多这样的线索交织成的，当每条线索在这种交织中难觅踪迹时，科学就是那条最强的主线。

我将要捕捉的线索是并发计算的语义基础。首先我解释一下我是怎样认识到并发性需要全新的方法，而不仅仅是拓广用以解释顺序计算的实体和结构的全部内容。然后将谈谈我凭借顺序语义的经验为并发寻找基本结构的过程，正是这项工作诞生了通信系统演算(CCS)。就此我将简要地讨论这些结构可以用数学来理解的程度，就象顺序程序可以用函数来理解一样。最后，我将概述一个新的基本并发演算，它将提高原有的命名(naming)或参考(reference)思想，但这一思想至今仍未引起计算理论(界)的高度重视。

我认为，对于像并发计算这样大的问题的所有方面，并不只存在一个唯一的概念模型或一个众人所喜爱的形式框架。在某种意义上并发计算是我们的全部主题，而顺序计算是其中表现良好的特殊领域。我们需要许多解释层，对不同的专业领域有不同的语言、演算和理论。具体应用是多种多样的：保险公司的信息流动、网络通信、飞行控制计算机之间的实时通信、数据库中的并发控制、面向对象并发程序的行为、并发逻辑程序设计中变量的语义分析。我们当然不希望对所有这些应用使用相同的术语来进行讨论和分析。

但是必须补充声明一句：计算机科学家同所有的科学家一样，试图寻找一个公共的框架来连接和组织许多解释层；而且这个公共的框架必须是语义的，因为我们的解释(包括程序)通常使用形式语言(并且常常是几种形式语言的混合物)来处理大的异构系统，这就是我们的任务。对于顺序计算这个小得多的领域来说，公共语义框架是以数学函数这一中心概念为基础的，并可形式地表示成函数演算，Alonzo Church 的 λ-演算就是一个著名的原型。当我们讨论顺序程序设计的语义时，函数是基本成分。但通常对并发程序设计和交互式系统而言，我们找不到可比较的基本成分。

那么，我们到哪里去寻找并发的语义成分呢？或者说我们怎样升华出它们呢？这是一个宏伟的目标，正如我前面提到的，并发性是普遍存在的。我相信，解释并发计算的正确思想只能来自逻辑和数学的模型与对实际经验的适当升华两者之间的辩证统一。

我做了一些辩证工作。我试图缓解两者之间的对立：一方面，函数演算显示了其纯净性和简单性；另一方面，程序设计和通信实现中隐含着关于并发和交互的某些非常具体的思想。

实体

七十年代，我逐渐认识到并发和交互的理论需要一个新的概念框架，而不仅仅是适合于顺序计算的已有框架的精化。

通常，一个人形成某个信念的经验都是不可预见和不深刻的。但我很想回忆一下我自己的一个体会，因为它在几个方面与这里的主题有关。当时，我试图把关于程序设计语言语义的 Scott-Strachey 方法加以扩展来处理并发语言的语义，因为该方法对绝大多数高级顺序语言处理得非常漂亮。我必须做出这样的努力，而且对于成功我非常乐观。

按照该方法，假如无中间输入/输出的话，那么一个顺序程序可以完美地用一个从存贮器到存贮器的函数来表示。（我使用"存贮器"这一术语是指存贮器状态，它包含程序中所有变量的值。）而且 Dana Scott 发展了论域理论（论域是一种具有特殊性质的偏序集），该理论为 λ-演算（一种纯粹的函数演算）提供了语义。所以在 Scott-Strachey 方法中，强制式程序的语义在于通过下列方程给出的论域中：

程序语义＝存贮器→存贮器

该论域是普遍适用的，原因在于：对于任何顺序语言中的每个语法结构，都对应有一个抽象操作，它能根据部分程序的语义来构造复合程序的语义。也就是说，语义是可合成的，这是一条基本性质。

而并发引入的一种现象是非确定性。（当然，在没有并发的情况下也可能有非确定性存在，但在我看来，我们要处理的是并发引起的非确定性。）Plotkin 利用幂论域理论（论域理论的一种扩展特技）来处理非确定性。对任何适当的论域 D，都存在幂论域 $\mathcal{P}$(D)，其元素均为 D 的子集。所以考虑到非确定性，我们可以重新定义程序的语义为：

程序语义＝存贮器→$\mathcal{P}$(存贮器)

即程序语义本质上是存贮器上的关系。通过对顺序语言加入非确定性算子即可获得非确定性语言，对于它来说，上述的程序语义是完美地可合成的。

但是，并发性有其惊奇之处。当你组合子程序并发运行时，就失去了可合成性，因为它们可以互相干扰。确切地说，作为具有相同的关系语义的程序 P_1 和 P_2，当与第三个程序 Q 并发执行时，它们的行为就不同了。一个简单的例子就是：

程序 P_1：x：＝1；x：＝x＋1

程序 P_2：x：＝2

在无干扰的情况下，P_1 和 P_2 都把初始状态下存贮器中 x 的值替换成 2，所以它们具有相同的语义。但对下列程序：

程序 Q：x：＝3

如果让 Q 依次与 P_1 和 P_2 并发运行，即：

程序 R_1：P_1 Par Q

程序 R_2：P_2 Par Q

那么程序 R_1 和 R_2 具有不同的语义。（即使在赋值语句做为一个不可分割的整体执行情况下，R_1 执行结束时 x 可能等于 2、3 或 4，而 R_2 执行结束时 x 只能等于 2 或 3。）所以可合成语义必须进一步精化，必须考虑程序与存贮器之间的交互方式。

事后看来这种现象不足为奇。但如果我们不能利用存贮器上的函数或关系来解释并发程序，那么我们能使用什么手段呢？当然，我们会很自然地在更精细的粒度上给出关系语义，使它能记载程序从一存贮器状态到下一存贮器状态的每一步，而这只利用论域理论就可完成。但是这种现象给了我一个深刻的教训：一旦存贮器不再为单个程序所操纵，那么将程序与存贮器的关系视为主-仆（或函数-值）关系的观点就不复存在了。古谚云：一仆不侍二主。所以最好能建立一个交互系统的通用模型，其中程序与存贮器的交互恰是同类之间交互的一种特殊情况。

下面用视图来帮助理解。图 1 非形式地给出了共享存贮器模型。我们使用不同形状的结点来表明部件之间主动/被动的区别。（在后面的图中，我将一直使用方框代表主动进程，圆圈代表被动进程。）当然，程序一般使用若干个位于 M 中的变量。

为消除图 1 中主动/被动之区别，我们将 M 提升为进程状态，并把变量 x，y，…当作程序与存贮器之间交互的通道名，如图 2 所示。

现在，更一般地，让我们利用存贮器来阐释下述观点：可用任何种类的进程之合成来形成更大的进程。

在顺序情况下，为了方便我们一直假设，存贮器是一个坚实的整体，但对并发程序设计而言，这很不现实，因为存贮器的不同部分可以被同时存取。所以，如图 3 所示，我们进一步把存贮器的每个单元看作一个进程，例如 X，并通过适当的命名通道将其与一个或多个程序（即进程）相连。

软件工程师可能坚决反对这种相似的处理，并顽固地坚持共享存贮器模型，这对他们是十分重要的，因为在共享存贮器模型下允许使用一种方法学，有助于书写出正确的程序。理论学家可能回答，在只能容纳一种实体的基本模型中容纳两种实体，这在科学上是不允许的。他们还可能指出，共享存贮器模型中主动/被动的区分很难适用于混合型（实体），例如对数据库，当你不使用时它可进行自重组。当然这两种看法不无道理。

让我们回头看看对多解释层的需求。计算机系

统的工程师们希望能对不同目的使用不同概念和模型。例如，共享存贮器模型自然是他们的拿手好戏。但计算机科学家还必须寻找各种模型的共同基础，他们不仅对单个设计和系统感兴趣，而且也对其成分的统一理论感兴趣。为了在并发的基本模型中获得统一性，所有的交互，因而所有的交互者都必须予以同样对待，这就是我将本次演讲称为"交互作用之基础"的原因。

为了你们不误认为我考虑的仅是程序、存贮器或计算机系统的交互，在图 4 中我描述了一个汽车电话网，其中信道是无线电频道。通信协议允许汽车把信道切换到离它最近的站上，整个系统是中心监控的。现在，我们希望我们的基本结构能在一个抽象的层次上完美地描述这样的系统，交互的基础不应当是只针对计算机系统的。

在六十年代，Carl-Adam Petri 已经很好地理解了我上面所讲的大部分内容，他是构造离散并发系统的科学模型的先驱。他的工作为并发理论奠定了坚实的基础。他声称他的 Petri 网理论至少既可作为物理世界的模型又可作为计算的模型。他认为，内存单元和程序都可用同一种对象来模拟，这就是所谓的 Petri 网，这样就打破了主动/被动的二分法。Petri 网理论的概念框架极其简洁，但无论是对单个系统的分析还是对各种系统的分类，它的清晰度和深度确实使人受益匪浅。

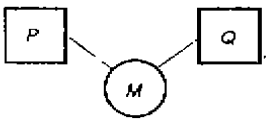

图 1　共享存贮器模型

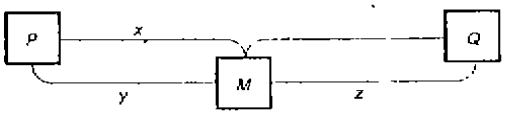

图 2　存贮器作为交互进程

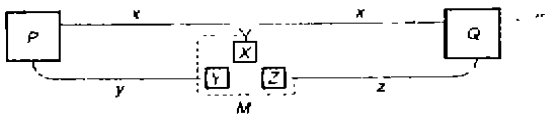

图 3　存贮器作为分布式进程

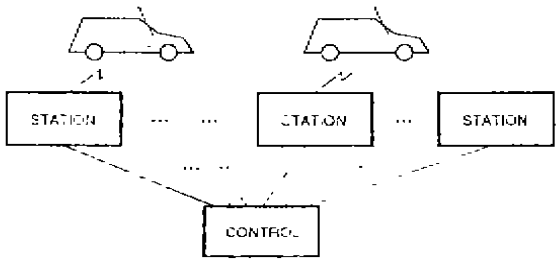

图 4　汽车电话网

静态结构

除存在系统组成实体的主动/被动之二分法这一问题之外，从基本结构上来说，并发需要一个全新的方法。我曾一直试图用程序设计所遵循的关于系统合成的观点来发展和完善 Petri 网理论。这本质上是一种代数观点，因为代数是研究结构及其含意的。对于顺序计算，这种观点体现在 λ-演算中，而有别于自动机的经典理论。

为了处理并发性，我们不仅仅应该为顺序计算语言和理论添砖加瓦，特别应该增添附加结构以便从小系统建造大系统。如果我们这样做的话，那么我们当然会得到更多的基本结构。这很好地证实了我前面的预言，即并发比顺序复杂得多，并且这已成为现实。然而，尽管并发可能复杂得多，但我们不应如此轻易放弃。我们应该把自己的注意力集中在那些本身对并发来说是最基本的结构，那么，我们确实可以把顺序计算看作是一种更高、更特殊的解释层。

考虑顺序合成 P;Q，其中分号";"是大家熟悉的符号，是强制式顺序程序设计的基本连接符。为了获得并发性，我们是否应该在保持顺序合成的同时增加并发合成呢？当然，对一个程序设计语言我们可以这样做，因为我们在为程序员提供新工具的同时必须提供他们已熟练的工具。但是，我们是否应该在基本演算中这样做呢？我认为不应该，因为顺序合成确实是并发合成 P|Q 的一种特殊情况。对于 P|Q，我理解为 P 和 Q"并肩"运行，并按我们为其设计的任何交互方式进行交互。所以顺序合成是并发合成的下述特殊情况：其中交互只发生在 P 刚执行完而 Q 才开始执行时。允许引入特殊类型的交互将破坏我们的基本原则：在基本模型中，所有的交互都是同一种类的。

正是这种对事物的洞察力使得我不再企图从顺序性中进行外推，而是试图用一个新的演算来捕捉

一组关于并发性的基本结构。我想 Church 就是这样利用 λ-演算来处理顺序计算的吧！我希望能找到与函数演算相匹配的并发演算，不是仅仅照搬其结构，而是竭力效仿它的两个属性：①它是可合成的，我们可用它构造系统，因为项的结构表示进程的结构；②它是可计算的，其基本的语义概念是一个计算步。对它的另一个属性，即具有公认的数学解释，我们的并发演算尚不具备（虽然已大有长进）。不过 Church 当时自己也把 λ-演算的项理解为可计算意义上的函数，而不是 Scott 所说的指称。

总之，对我来说，函数演算是为通信系统构造演算的范型，但不是平台。

刚才我指出，顺序合成是并发合成的一种特殊情况。的确，在设计 CCS 时我坚持认为只用一个组合算子即可合成交互的或共存的进程。这看起来似乎是苛求，我还认为，内存单元可模拟为进程，因此这同一个组合算子必须既能把内存单元装配成存贮器，又能将使用这些内存单元的进程合成起来，还能把进程与存贮器组合起来。然而一个组合算子确实足矣！这是因为所有的交互确实可以同等对待。例如，我们可把图 3 的系统写成 P|M|Q，其中 M=X|Y|Z，或者写成 P|X|Y|Z|Q。即使程序 P 和 Q 除通过存贮器进行交互之外，还利用其它方式进行交互，或者 X，Y，Z 不仅仅是内存单元，而可能是充当程序和远程存贮器之间中介的进程，我们仍可使用相同的表达式来进行表达。表达式的形式独立于这五个进程的性质。

由于其核心只有这唯一的并发组合算子，所以并发演算的代数性质逐渐显露出来。并发合成的直观背景是：我们仅简单地把系统的成分汇集起来。所以我们希望组合算子是可结合和可交换的，因而我们的表达式中不必出现括号。成员的不同顺序和括号的不同匹配均表示一种将系统分解成子系统的不同划分。

我们的代数怎样才能更清晰地反映出由系统成分之间的连接所导出的结构呢？我们首先注意到，在图 3 中，系统成分 P，Q，X，…本身就是进程表达式，而且通道 y 只连接成员 P 和 Y，因为 P 和 Y 是通道名 y 在其中出现的仅有的表达式。此处我们不给 Y 这样的存贮单元以进程表达式，可以这样说，每个这样的表达式将以其地址作为通道名。这样我们可以说，从 P 的观点来看，名字 y 定位了单元 Y。图 3 非常清楚地表达了这种定位思想，我们也想在我们的代数中有所体现。为此，我们引入了另一个组合算子来保证：单元 Y 只能由 P 来访问，即通道 y 是局部于 I 和 Y 的。我们将这个新的组合算子称为限制（restriction），例如在表达式 υy(Y|P) 中，通道 y 被限制在 Y 和 P 之中使用。希腊字母 υ 部分地用作双关语 "new"，实际上，限制只是程序设计中局部变量说明这一概念的提炼。这样，对图 3 所示的整个系统，我们可以更精确地写成：

$$\upsilon x(X|\upsilon y(Y|P)|\upsilon z(Z|Q)) \qquad (1)$$

一般说来，这种表达式能很好地表示交互式系统的空间或静态结构，尽管此处我们只用它解释程序和存储器。我们把图 3 这样的图形称为流图，这是相当形式化的对象。的确，限制和合成满足特定的方程式，从而定义了流图的代数理论。

动态结构

现在让我们看一下这种系统的动态（即行为）方面。实际上，正是这种动态性使得我们能将 λ-演算的顺序性、层次性控制与 CCS 的并发性、非层次性控制进行对比。在 λ-演算中，所有的计算都最终化为一件事：称为归约（reduction），即把一个参数传递给函数，我们可以把这叫做 λ-演算的行为原子。同样，一次交互，即进程之间单个数据的传递，是 CCS 的行为原子。现在让我们看看，它是怎样使合作者间保持对称性的，它又是如何反映系统中的每个进程均具有一致的身份这一思想的。

我们将根据两种演算的基本计算规则来看看这种对比。在每个演算中，系统由一个二元组合算子构成。在 λ-演算中，这个组合算子称为施用（application），它既不是可交换的又不是可结合的。当项 M 施用于另一个项 N 时，记为 M(N)，那么第一个项 M（即操作子）取得控制权，操作子将接收 N 这个操作数。当操作子采取 λ 抽象的形式 λx. M[x] 时（其中方括号表示 M 中可能含有约束变元 x），符号 λ 表示操作子的控制轨迹，而且将发生下列归约：(λx. M[x])(N)→M[N]（右边的方括号表明 N 替换了 M 中的 x）。这是 λ-演算的基本计算规则，即函数施用的动态性。众所周知的 Algol 60 的拷贝规则，就是从它导出的。图 5 形象地表示了这种操作，注意操作数 N 用圆圈表示，是归约中的一个被动数据，只有当 M 允许时，作为操作子它才能在以后的阶段获得控制权。

另一方面，在 CCS 中，当两个项 P 和 Q 并发组合成 P|Q 时，二者都持有控制权。P 可能收到 Q 的信息，Q 也可能接收到 P 的信息。一个项可能采取两种所谓的动作（actions）来与另一个项交互。当一个合作者采取输入形式 λx. P[x] 而另一个采取输出形式

$\bar{\lambda}V.Q$ 时，我们有并发合成 $\lambda x.P[x]|\bar{\lambda}V.Q$。其中合作者具有所谓的互补控制轨迹 λ 和 $\bar{\lambda}$，即命名为 λ 的通道的正负端。此时将发生下面的交互：$\lambda x.P[x]|\bar{\lambda}V.Q \rightarrow P[V]|Q$。这是合作者间动作的同步。

但关键是，并发中有许多通道而并非只一个。所以，在 λ-演算中用来表示单一控制轨迹的符号 λ，现在成为可能处于并发活动状态的众多通道中的一个。CCS 用简单名字 a.b.c.…来标记这些通道。这样我们得到 CCS 的基本计算规则，如图 6 所示。计算只不过是反复利用这单一规则，不断地转换表达式。在任何阶段，表达式的结构表示了系统的空间结构。图 6 通过表示空间格局(configuration)的流图之上的迭加，显示了被动数据传输的动态行为。

这里，我们再仔细看一下上述的差异。第一，在 λ-演算中，第二个合作者仅是一个被动数据，但在 CCS 中，它输出数据 V 并继续独立地存在。正是这一点保证了并发进程之间的连续交互，同时每个进程保持自己的身份。

第二，并发合成是可交换的，P|Q 与 Q|P 意义相同。与 λ-演算不同，在 CCS 中任何合作者都可以充当接收者，而且该接收者以后又可能变成发送者。

第三，并发合成是可结合的。在 λ-演算中，项结构表示了其层次性控制，而在 CCS 中，可结合性与可交换性一起使得交互规则不受项结构约束。正如我们在关于程序-存贮器交互的例子中所见到的，控制规则将取决于哪个通道与哪个进程相连。

虽然这是一些技术性细节，但问题的焦点在于：并发计算与顺序计算的差异集中体现在各自演算的单一基本计算规则上。通过修改 λ-演算中归约的概念，我们得到了并发进程的交互规则，这是我们将函数模型的纯净性和现实并发系统的动态性进行综合的第一阶段。

在我用代数形式表示之前，Tony Hoare 已经熟知把同步交互作为程序设计原语的思想。在不同动机的驱使下，我们各自独立地采取了这些做法。这一事实本身就证明这种思想是很自然的。Hoare 把这种思想结合到其程序设计语言 CSP 中，并且后来与他的同事一起发展了一个关于 CSP 的理论，从而为 Ada 的会合机制提供了基础，并且成为程序设计语言 Occam 中的交互原子。我的贡献在于把交互作为一种代数演算的基石。八十年代发展的进程代数，如 CCS，CSP 和 Bergstra 与 Klop 的 ACP，已成为许多语义研究的主题，并且已逐渐成为经常使用的设计工具，特别是规范语言 Lotos 已广泛应用于通信协议。

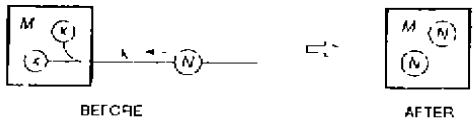

图 5　λ-演算的归约 $(\lambda x.M[x])N \rightarrow M[N]$

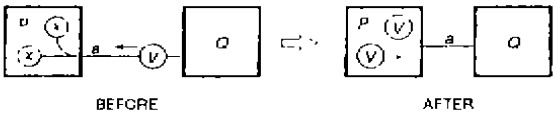

图 6　CCS 的交互 $ax.P[x]|\bar{a}V.Q \rightarrow P[V]|Q$

语义(Meaning)

我很想减轻人们对并发的恐惧心理，至少从我处理并发的方式来看，并发是相当具体和机械的。难道不存在抽象的数学实体可以做为它的全部基础，从而我们可以保留"进程"这一术语而不必把代数表达式叫做"进程"？毕竟，在某种抽象意义上，我们已经使用"函数"这一术语，所以当我们指着 λ-演算中的一个表达式称之为"那个函数"时，人们一般认为我们指的是"那个表达式所指称的数学函数"。如果一个表达式没有公认的指称，我们可能感到不舒服。

关于进程的语义有大量的文献且在激剧增加，它一点也不简单，而且已经出现某些激动人心的进展。在早些时候我谈到了 Scott 的论域在并发程序中的可能应用，的确，已经为进程代数研究了许多有趣的论域。而且对其可观察行为(observable behavior)所确定的进程的语义也进行了大量研究，其主要思想是：观察一个进程就是与它进行交互，两个系统(如 CCS 中的两个进程项)有相同的指称当且仅当我们不能通过观察来区别它们。这项研究极为有益，可划分可观察特征的强弱等价类并进行各种情况下的数学刻画，现已取得相当成果，但尚有许多问题需要进一步研究。

迄今为止，这些关于观察的评论同样适用于顺序进程，但是，正如人们所知道的，并发有它自己的问题。一个重要的话题是因果独立性(causal independence)概念，它是 Petri 网理论的中心。对外部观察者来说，两个具有并发成分的进程可能是不可区分的，然而在可观察到的动作中因果关系不同。我可能观察到你从树上掉下来然后看到救护车到达，但我仍

然不能确定是否一件事引起另一件事。因此观察语义忽略了因果关系。但重视因果关联的语义模型也是有充分理由的，如 Nielsen 等人的事件结构。这个话题过于复杂，一时难以讲清，我之所以提到它，是为了说明并发确实产生新的语义问题。

无论研究何种数学模型，我相信，进程演算为研究提供了一个基本的观点。并发系统的语义理论，只有当它最终成为抽象的同时又是形式化的理论时，许多人才会对它满意。但为了达到这个目标，我们首先必须抓住交互的动态性这个本质，这就是形式化进程演算如 CCS 所要做的。

值

现在我们进入把函数范型与交互的现实性进行综合的第二阶段，这时我关心的是在其自身的概念框架内形成一个具有充分表达能力的并发演算。首先，我要解释一下 λ-演算在这方面是怎样成功的，相比之下 CCS 和类似的并发演算显得有些不足。以后我们将找到一种补救方法。

人们通常非形式地使用 λ-演算，只是为了在熟悉的数据上导出新操作子。当讨论算术时，可能写出如下的公式：$f=\lambda x(x^2+1)$ 或 $F=\lambda f\ \lambda x(f(x)+1)$，即把 λ-演算的构造与算术的构造混为一谈，并不希望用一个来表达另一个。这样的话，λ-演算像一个计算架，而数据（如数字）和操作子（如平方，加法，求微分）就象真正的工人一样在该计算架上爬上爬下。

λ-演算的这种辅助应用是很自然的。但在函数框架内很难导出一个自包含的计算模型。每当我们增加数组、表和树等新的数据结构类型时，事实上扩充了演算。但它对理解程序设计语言是很重要的，例如，我们用以解释它们的基本模型应尽可能同构和完全，不需进行特别的扩充。

很显然，λ-演算确实具有这种同构性和完全性。在其纯粹形式下，它确实足以能表示数据结构和其上的计算，只用计算架我们就可以做任何事。对于算术，Church 证明了这一点，Böhm 和其他人把这一结论推广到其它数据结构形式上。而且，存在类型理论，即著名的 Girard-Reynolds 二阶 λ-演算，允许用受控方式为分析顺序程序设计语言和其相应的类型理论建造一个统一的框架。

那么，并发性又怎样呢？绝大多数并发形式系统也提供了一个不同的计算架，值和操作子可以在架上移动，这些值和操作子与计算架本身大为不同。再看一下 CCS（图 6），在该交互中，数据 V 实际上是一个代表数字之类的表达式，它是与进程表达式 P 大不相同的一类东西。

为了强调这一点，图 7 给出了 CCS 中进程 Double 的定义，它反复从通道 a 输入一个数字，并在通道 b 上输出该数字的两倍。考虑这个简单的定义，它至少包括 5 种不同的东西：进程（Double），通道（a，b），变量（x），操作子（入），值表达式（2入x）。其中四个（不包括操作子）已出现在 CCS 的基本计算规则中（见图 6）。这种情况是否过于混杂了？

对于高层程序设计语言或规范语言，比如 Lotos，它是完全可以接受的，在这些语言中，用户期望有一个取之不尽的工具箱。但对真正的基础演算，这种情况就不那么舒服了。基础演算应该施加尽可能不多的分类学，因为不同的高层解释将施加不同的分类学。

纯 λ-演算仅由两类事物组成：项和变量。对于进程演算，我们能否达到同样的简洁性呢？Carl Hewitt 早就在其 Actor 模型中对这个挑战做出了反应。他声称，一个值，一个对值的操作子，一个进程都可看成同一种事物，即一个 actor，这个目标给我留下了很深的印象，因为它隐含了我们刚才所谈的表示的同构性和完全性。过了很久我才明白如何在代数演算中达到这一目标。我知道 CCS 有所不足，但我希望它能为找到其它方法提供洞察力。

名字

最近，基于 Engberg 和 Nielsen 的重要洞察，我和我的同事 Joachim Parrow 与 David Walker 发现了与纯 λ-演算具有同样内部完全性的演算。它也是精炼的，按照代数进程演算的传统很难进一步简化。我们把它叫做 π-演算，其关键思想是命名或参考（通过对命名给予更大的重视从而得到更大的表达完全性，对这一点我们不应感到惊奇，因为系统中的并发活动赋予其成分以独立的身份，而为利用这种身份需要一种鉴别身份的方法）。

命名，虽然在一些模型（如函数模型）中隐藏得非常好，但在实际计算中是反常的。想想它的所有变种。本质上，它只不过是那些可以存取任何事物的东西，但在不同的场合，我们往往认为变量、地址、位置、指针、通道是不同的，这确实是因为它们可存取不同的东西。在程序设计语言（古老的或现代的）中，其种类是惊人的：Algol 60 的变量、Pascal 的指针、Ada 和 Occam 的通道、逻辑程序设计中的逻辑变量、面向对象程序设计的对象引用等等。所以，以 Hewitt 的精神，我们第一步是要求所有的东西都是同一种东西，都可用项表示或通过名字存取，如值、寄存器、

操作子、进程、对象等，它们应该都是进程。因此，我们把按名存取看作计算的原材料。下一步我们必须定义一种方法，利用该方法进程既可操纵存取，本身也是可存取的。

为了体会一下这种巨大的改变，再看看图 6 所示的 CCS 的计算规则。在该图中，a，x 和 V 是不同的事物，即通道、变量和值。而在 π-演算中，通道和变量都只不过是名字，而像 V 这样的值是由名字定位的进程。其关键的直觉在于：一个值恰是一特殊种类的进程，是可以反复成为同一观察主题的进程。所以表达式 V 确与其它进程表达式属于同一类，并且为了观察它，必须有一个交互通道。这样，正如一个存贮单元 X(见图 3)可以由名字 x 定位一样，值 V 可以由一个名字 u 定位。所以在 π-演算中，计算步采取如下的形式：$ax.P[x]|V|\bar{a}u.Q \rightarrow P[u]|V|Q$，形象地如图 8 所示。在该图中，每个大写字母代表一个进程，而每个小写字母是一个名字。注意一个名字可以主动地用作通道(如 a)或被动地作为数据来使用(如 x 和 u)。

在该交互中，值 V 不再是个参与者，连被动的参与者都不是。这样，交互的最简单表示方式是：$ax.P[x]|\bar{a}u.Q \rightarrow P[u]|Q$。这是 π-演算的基本计算规则，清楚地表明了与 CCS 的关键性区别，即此处传递的数据仅是一个名字，一种进行存取的手段。由于名字没有内部结构，计算步是真正原子的。与此相反，在 λ-演算中，归约的操作数可能是个复杂数据。

对此你可能有两种反应。首先，你可能觉得这毫无新意。因为你曾意识到，在真正的自顶向下计算中，传递的是指针而不是实际的值。对这个问题我的回答是，从程序设计的角度看，如果交互的基础已经很熟悉了，那么一切都好解决了，而且我们还可以在程序设计之外的领域使用它。

第二，你可能反对。因为当我们开始用一个很抽象的方式来思考计算时，所有这些机械性事务(指针或其它)都被隐藏起来而且假定隐藏得很好。在这个意义上我同意你的观点：虽然我们用一个非常漂亮的抽象方式来考虑顺序计算，但对于并发，我们在这个方向上前进不大。关键的问题在于：一方面当我们面对现实时，参考的概念不再保持隐蔽；另一方面它似乎拒绝理论处理，至少是很难处理。我相信 π-演算将解开这个死结，为参考提供易处理的理论，从而也为并发和交互提供理论。

我不准备更详细地讨论 π-演算，但为了说明它达到了 λ-演算的简洁性，我并列地写出两个演算的语法(见图 9 和 10)。与 CSP、CCS 和 Occam 一样，π-演算的可选择动作结构意味着恰有一种选择将发生。重复结构 !P 的大致意思是 P|P|…，即有充分多的 P 的拷贝并发执行。我们已经例示了其它结构。

为了明白演算是怎样进行的，回忆一下图 4 中的汽车电话网。我们把一个汽车与一个车站之间的通道模拟为 π-演算的两个名字——一个用于对话，一个用于切换，见图 11。该图表明 CAR 进程的定义是以这两个通道为参数的。CAR 有两个可选择动作，它可以对话或当 STATION 提出请求时它可以切换其通道。(此处 CAR 是递归定义的，但递归可由 π-演算中的重复结构导出。)此图还给出了一个可定义整个网络的表达式。其它成份跟 CAR 一样易于定义。

在这个例子中，汽车从一个车站到另一个车站的移动与图 10 中值 V 的移动无任何区别。这样，移动不仅限于值、各种进程彼此之间的移动可以用此种方式来模拟。π-演算的本质在于它对限制与格局的动态变化之间的相互影响做技术性处理，其中限制是对空间格局的模拟(如表示图 3 所示系统的表达式(1))。

总而言之，有下列几种理由使我们可以声称 π-演算具有普遍性，即使在这种简单形式下：

- 它对能改变自身结构的系统给出了一种直接描述，如汽车电话网，或具有作业流和资源分配的分布式操作系统。
- 它允许采用一种统一的方式来定义数据结构。这样，它支持 CCS 这样的进程代数作为较高的解释层。
- 可以把 λ-演算本身简单地翻译成 π-演算，并且忠实于原来的计算行为。这样，它支持把函数式程序设计看作较高的解释层。
- 它为面向对象程序设计和其它程序设计范型提供了一个方便的语义平台。
- 像 λ-演算一样，它遵循类型规则，这将允许我们增加分类学，而分类学对易处理的语义框架是非常重要的。

$$Double \stackrel{def}{=} ax.\bar{b}(2 \times x).\ Double$$

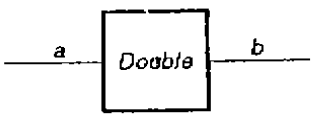

图 7　CCS 中的 Double 进程

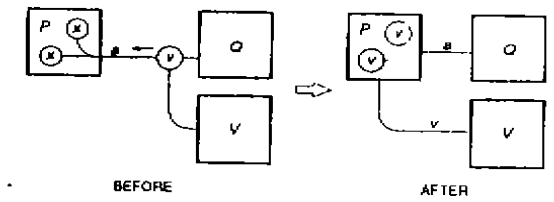

图 8 π-演算的交互

$$ax.P[x] \mid V \mid \bar{a}V.Q \rightarrow P[v] \mid V \mid Q$$

variables $x, y, z, \ldots$

terms	$M ::= x$		variable
		$\lambda y\, M$	abstraction
		$M_1(M_2)$	application

(the occurrence of y is binding)

basic rule of computation $(\lambda y.M_1[y])(M_2) \rightarrow M_1[M_2]$

图 9 λ-演算

names $x, y, z, \ldots$

action terms	$A ::= \bar{x}z.P$		send z along x
		$xy\,P$	receive any y along x
terms	$P ::= A_1 + \cdots + A_n$		alternative action $(n \geq 0)$
		$P_1 \mid P_2$	composition
		$\nu y P$	restriction
		$!P$	replication

(the occurrences of y are binding)

basic rule of computation $xy.P_1[y] \mid \bar{x}z.P_2 \rightarrow P_1[z] \mid P_2$

图 10 π-演算的简单形式

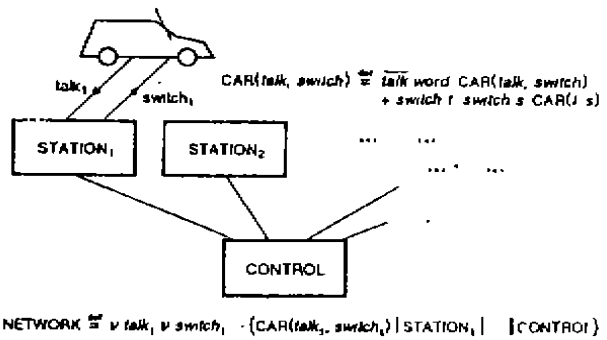

图 11 π-演算描述的汽车电话网

实际上正是这五个性质中的第一个最实在的性质，即描述变动性或改变结构的能力，是我们最先努力获取的。这个目标一直左右着 π-演算的形成。变动性事实上是整个问题的关键，这个发现对我们来说是个意外的惊喜。一旦这一点从技术上被掌握，我们就会发现无需再增加任何东西即可表示出其它性质了。

结束语

我已经回顾了在顺序范型的指引下寻找并发的基本模型的线索。我之所以敢谈论语义，是因为我一直希望我们所有的人都熟悉正确的语义原语，不是我们考虑了这些原语，而是我们很自然地根据它们来锤炼我们的思想。无论我刚才讨论过的原语是否恰当，它们肯定具有那种熟悉的性质。

对我不断地使用归约你可能会生气，一次又一次我用一个东西来解释另一个东西（一个值恰是一个进程，诸如此类）。我知道没有其它方法能达到既通用又易处理。但我重复一下：许多解释层是必不可少的。确实，与低层的实体相比，高层的实体当然呈现出更丰富的多样性。

我已经声明了 π-演算的一些通用性，尽管确实有一些事情不是直接处理的，还需要进一步研究。一项重要的任务是把它与 Hewitt 的 Actor 进行比较，两者有些明确的相同点，那是因为我们沿用 Actor 的直觉，也有一些细微的差别，例如对名字的处理。更一般地来说，π-演算是形式演算，而 Actor 模型本质上更接近于物理方法，试图鉴别支配交互的原始概念的法则。

最后，我们怎样评价或测试进程演算之类的计算模型？即使如 Alan Newell 和 Herbert Simon 在他们 1975 年的图灵奖讲演中所指出的，计算机科学可能不符合实验方法的狭隘旧框框，但在某种意义上讲，计算机科学确实是一门实验科学，因为我们对该领域中的机器、语言和系统都得进行测试。另一方面这些实验性测试也影响到我们的基本模型，因为我们最终是根据这些模型来设计、定义和分析我们的人工产品的。

除外在的实验性测试之外，还有内在的数学检验。把计算实践的真正基础之思想分离出来很难，找到那些同时允许易处理理论的思想则更难。正是这两种测试之间的冲突构成了我已举例说明过的辩证法的基础。我已经努力表明，在一个已建立的逻辑范型的指导下，如何从实践中进行提炼以产生精确定

（下转第 34 页）

在非法引用，则给出警告信息，对约束规则和操作函数的具体修改是由用户完成的。比较难处理的问题是广义操作函数，它是用一般高级语言编写的，若对它进行检查，势必要增加检查高级语言语法的功能，为简便起见，对广义操作函数系统仅提示而不作任何检查。

任何一动态模式操作应具有原子性，在并发修改模式时应和其它数据库操作一样对数据进行封锁，封锁的方式和管理与其它封锁管理一致。例如对某一类删除一属性，由于需要进行实例对象存贮空间压缩，可对该类加 x 封锁。对由于实例对象标识 OID 的改动而引起的需修改的引用对象所属类也施加 x 封锁，在全部修改完成后才可以释放这些封锁。如果在修改的任何一处出现不允许的情况，则应中断修改，利用日志文件使系统恢复到修改前的状态。因此，对每一动态模式操作，O_2S 系统为其产生一事务。

访问权限问题也和动态模式相关，在 O_2S 中，类与类之间存在着语义关系，由于对某一类的模式进行修改时会导致对另一类的修改。如果用户仅对起源对象类拥有修改权限而对另一对象类没有修改权限，迫使从下面三种办法中选择一种，一是放弃全部修改，二是仅修改源类，三是扩大用户权限。采用第二种方法会导致不一致的信息存贮在数据库中，第三种方法可能导致数据的失密。因此 O_2S 系统采用第一种方案，先拒绝操作，当用户获得足够的权限时方允许进行这类操作。

七、结束语

动态模式反映了设计对象在结构和分解方面的动态性，它来源于设计任务的分解性、设计过程的试错性以及自顶向下的设计方法等设计特征，是工程数据库管理系统应支持的重要功能之一。本文提出的动态模式处理策略已在 O_2S 系统中得到应用。

参 考 文 献

[1] *Peter Wegner, Dimensions of Object-Based Language Design, Proceedings of OOPSLA' 87, 1987*

[2] Andrea H. Skarra, Stanley B. Zdonik, Type Evolution in an Object-Oriented Database, In Research Directions in Object-Oriented Programming, B. Shriver & P. Wegner (editors), MIT Press, 1987

[3] Jay Banerjee, et al., Data Model Issues for Object-Oriented Applications, ACM Trans. on Office Information Systems, January, 1987

[4] D. Jason Penney, Jacob Stein, Class Modification in the Gemstone Object-Oriented DBMS, Proceedings of OOPSLA' 87, 1987

（上接第 8 页）

义的候选理论，它必须接受更深的数学和更深的实验检验。

我相信，在这个普遍的科学方法上，即使不是在其实验标准上，计算机科学与物理学差别不大。同许多计算机科学家一样，我希望有一种关于人为和自然现象的广义信息科学能与现有的丰富的物理科学相匹配。如果我所描述的基本思想能在该方向上前进了即使一小步，我将感到很欣慰。

鸣谢

我认为这个荣誉在很大程度上应归功于为此做出努力的许多人，我感谢他们，尤其是与我一直并肩战斗的同事 Rod Burstall 和 Gordon Plotkin，从他们那里我学到了许多东西，得到了大力支持。David Park 在一个要点上对我影响很大并一直鼓励我，Tony Hoare，Carl-Adam Petri 和 Dana Scott 的工作给我以启迪。（参考文献 38 篇略）

[李舟军　刘海燕　译自 CACM Jan, 1993, pp78—89，陈火旺　校]