通信顺序进程(稿)

原著：C. A. R. Hoare
译注：Huailin.Chen

2017年10月

Communicating Sequential Processes

C. A. R. Hoare

May 18, 2015

谨以此译注工作献给令人尊敬的

图灵奖获得者，英国皇家学会院士C.A.R Hoare先生，
中国科学院院士，中国科学院软件所研究员周巢尘先生。

译者序

译者一直好奇这个并发的世界是基于逻辑的，还是基于数学的。似乎爱因斯坦，哥德尔，冯 . 诺伊曼都认为自己是数学家。我想Tony Hoare院士也希望如此。

并发行为无处不在，大到浩瀚星宇，小到蚂蚁世界。其本质是一个分布式系统，各个组成部分在独立的运行，但也彼此影响，交换信息。复杂并发系统蕴含的巨大的不确定性，使得对并发系统的理解变得非常的令人好奇。

感谢图灵奖获得者，英国皇家学会Tony Hoare院士关于通信顺序进程，CSP(Communicating Sequential Process)的创造性的工作。CSP通过引入代数的手段，利用数学的方法来理论研究和精确推演一个复杂并发系统的迭代行为，令人倾佩。

感谢中国科学院周巢尘院士。您上个世纪80年代与Tony Hoare之间的学术合作研究，和在1988年翻译出版了Hoare的CSP这一经典著作，对中国科学在数理逻辑，程序语言，软件可靠性验证的研究做出了巨大的贡献。我们一直心怀感激。

三十年过去，弹指一挥间。沧海桑田。

未来的世界，万物互联。 随着人工智能AI的兴起， 大规模并发计算机系统会迅速的普及，例如，无人驾驶汽车，无人小型飞行器，智能机器人等等。这些互联的大规模智能系统的安全性(Safeness)， 可靠性(Reliability)， 容错性(Fault－Tolerance)， 可用性(Availability)变得对社会，家庭和个人的安全致命相关(Mission Critical)。AI系统的安全性已经成为人类对未来的一个重要忧虑。

译者认为我们需要更多的从并发系统的不确定性， 系统状态空间的爆炸性的角度去理解和设计未来的智能设备和智能机器人。这是为什么时隔30年，译者在Tony Hoare于1985年出版的《Communicating Sequential Processes》[1]一书，2015年发布的CSP电子版本[2]，和周巢尘院士在1988年由北京大学出版社出版的《通信顺序进程》[3]一书工作的基础上，再次翻译和加注CSP这本关于并发理论的经典著作的主要目的。

译者在并发理论方面水平有限，但对系统行为的不确定性，对系统行为的复杂性充满了好奇。希望这一工作对社会有益。

---

[1] Hoare, C. A. R. [1985]. Communicating Sequential Processes. Prentice Hall International. ISBN 0-13-153271-5.

[2] http://www.usingcsp.com/cspbook.pdf

[3] 周巢尘 [1988]. 通信顺序进程. 北京大学出版社. ISBN7-301-00813-9

前言(FORWARD)

因为若干原因，所有知情的人都在焦急的等待着这本书的出版；他们的耐心现在终于得到了回报。

一个简单的原因是因为这本书是Tony Hoare的第一本专著。很多人是Hoare在世界各地不知疲倦地作学术讲演时认识他的；更多的人知道Hoare是因为他是不少学术文章的作者。其论文清晰、严谨，而且涉及面很广泛。他的文章通常还墨迹未干时，就已经成了相关领域的经典文献。但是，专著是一种非常不同于学术研究文章的媒介：在专著中作者可以不受通常十分严格的篇幅限制来表达自己的观点；作者可以获得更方便地表达自己，和对一些话题展开充分讨论的机会。Tony Hoare分地利用了专著的这些优点。不负众望。

另外一个更具体的原因是来自这本书的内容本身。作为一门新型学科，当计算机界在25年前开始认识到并发现象时，整个计算机界都陷入了无穷无尽的困惑之中。这种困惑一方面来自并发现象会出现在许多不同的各类环境中，另一方面是由于并发现象同时导致了许多当代历史事件发生的非确定性。为了解开这一困惑，需要一个成熟的、具有献身精神的而且是幸运的科学家的艰苦劳动来诠释这一复杂的并发理论。Tony Hoare把他科学生涯中的一段主要精力用来了接受这一挑战，我们从各个角度而言都要感谢他。

期待这本书早日出版的最深刻的原因是读过本书早期初稿的人都有的一种强烈感受。Hoare的书稿通过令人惊奇的清晰透彻的诠释并发理论，展现了计算机科学可以——或者甚至说，应该——成为怎样的一门科学。计算机科学家口头说或者自我感觉他们面临的主要挑战是不要被自己研究工作的复杂性弄糊涂了是一回事；而发现并通过一些数学法则的准确无误和优雅来达到这一崇高的目标，则是完全不同的另一回事。在这里，我们从Charles Antony Richard Hoare的科学智慧，数学符号系统的大胆使用，以及处理上的巧妙中受益匪浅。我们对此深深的心怀感谢。

Edsger W. Dijkstra

序言(PREFACE)

本书是写给那些有抱负的，有志于对他们所从事的需要高智能的工作有更深入的理解和掌握更高技能需求的程序员的。本书的组织是通过对一个熟悉的话题采用一种新的探索方法，从而引起读者产生一种很自然的好奇心。这种新方法是通过一组例子来阐述的，这组例子选自一些不同的应用领域，从自动售货机、童话故事，游戏，到计算机操作系统。对这些例子的处理是通过一套系统化的数学代数理论来刻画的。

本书的最根本的目的是让读者具有一种洞察力，能用新的眼光去考察当前和未来的问题，使这些问题能更有效、更可靠地得到解决，甚至在某些情况下，最好是可以避开从不需要考虑这些问题。

本书中阐述的新方法最显而易见的应用场景是可以用在形式描述、设计和实现那些持续不断与环境发生交互作用的计算机系统。基本的思路是这种系统可以很容易地分解成多个并行运行的子系统。这些子系统之间以及与它们的共同环境之间持续地发生交互作用。子系统的并行组合与传统程序设计语言中程序行或程序语句的顺序组合是一样的简单明了。

对并发系统的这种洞察理解也带来许多实际的好处。首先，可避免程序设计中处理并行性时的很多传统问题，如干扰、互斥、中断、多线程、信号量等等。其次，近年来在程序设计语言和程序设计方法学研究中提出的很多高级结构化观念，也都成为本书理论观点下的特例了，例如管程(Monitor)、类程(Class)、模块(Module)、包(Package)、临界区(Critical Region)、封装(Envelop)、形(Form)、以及最普通的子程序(SubRoutine)概念。最后，这种新方法还提供了可以避免严重错误，诸如并发程序的发散、死锁和非终止等，和计算机系统在设计和实现时的正确性可以得到严格保证的可靠的数学基础。

在书的组织安排上，我试图按照逻辑的和循序渐进的顺序逐步介绍我的关于并发理论的想法，先从简单和基本的算子开始，逐渐展向利用这些基本算子的更精巧的应用。好学勤奋的读者可以逐页地阅读本书。但其他读者可以从他们自身比较有兴趣的话题开始；为此本书的每一章都刻意组织，方便读者可以适宜的选读。

1. 每个新观念都给出一个非形式的说明，并用一些小例子来解释，这样可能有助于所有读者。
2. 书里给出了这些新概念的相关代数演算法则。这些法则可以用来刻描述各种运算的重要特性。喜爱数学优美结构的读者会对这方面有兴趣。如果读者希望通过能够保持正确性的变换来优化他们的系统设计，那么这些代数法则对他们也会非常有所帮助。
3. 书里使用了大家比较熟知的程序设计语言LISP的一个简单的功能子集来实现各个算子。这个略微不太一般。对于那些可以用LISP实现的手段来检验和论证其系统设计的读者，对本书的这一安排会很感兴趣。
4. 系统分析员对迹的定义和描述会应该感兴趣。系统分析员在开始一个系统的实现前，需要精确的描述用户的需求。高级程序员对这部分也应该有兴趣，因为

在设计一个系统时，他们需要将一个系统划分为若干个子系统，并且需要清楚的描述子系统间的接口。

5. 证明规则对工程实现人员很有益。工程人员肩负着根据已有的描述、规定的日程、规定的价格，完成可靠的程序的重要责任。
6. 最后，书中的数学理论对进程概念和相关的用来构建各种进程的算子给出了严格的定义。这些定义是相关代数规则，实现和证明规则的基础。

读者对上述任一论题缺乏兴趣，或者感到一时很难理解，可以略去或者暂缓阅读相关章节。

各章间的承上启下也很方便于读者的精读、选读或者按照自己的安排。其中第一章和第二章的前几节是导引，建议所有的读者都阅读，但后面的章节可以略过，或者留待第二遍时再读。第三、四、五章是彼此独立的，读者可按兴趣和爱好，以任意次序或组合进行阅读。我们建议，读者在遇到难懂的内容时，不必中断，可继续阅读下一节，甚至下一章，因为忽略过的材料很可能不会立即又提到。如果需要用到前面的内容时，书中通常会标注好相关引用，从而当读者有了足够的动力时，可以重新捡起来。我希望读者最终能发现书中的每项内容都是有趣的，是值得一读的；但我并不期望每个读者都按本书书写的次序来阅读和掌握。

用来阐述本书中各种新概念的都是些小例子。这是有意这样安排的。解释一个新概念的最初的几个例子就应该十分简单。这样才不会由于例子的复杂或者读者的不熟悉而影响对概念的理解。通常后面的一些例子会比较难理解，因为这些例子涉及的问题本身很容易让人产生困惑而且很容易变得复杂化；由于本书引入的新概念的表达能力的强大，符号系统的精巧，从而对这些问题的解决方案会显得非常简洁。

然而，每个读者以后都会痛苦的遇到一些问题，这些问题会比书中的例子覆盖的范围更大，更复杂，更重要。这类问题看上去是用任何数学理论来处理都是很难的。但切不可因此而泄气或者放弃，应该接受挑战，试着应用本书中的新方法去解决这些现实的问题。

选择一个复杂问题的某些局部点，形成一个简化的模型，然后以此为出发点，在必要的时刻，再逐步加入复杂的成分。令我们惊奇的发现是，最初的、简化的模型往往会增加你对问题的深刻了解，有助于你对整个问题的解决。模型或许可作为一种基本结构，在这个结构上，复杂细节可以安全的地往上添加。令人最惊奇的发现是，在模型中添加的一些复杂细节是没有必要的。如果事情都是这样，掌握一种新的系统构建方法所作的努力是完全值得的，会有不错的回报。

在学习时，人们常常抱怨符号系统。例如，初学俄语的学生经常抱怨陌生的斯拉夫字母造成的障碍，特别是其中不少字母的发音奇特。尽管如此，这还是俄语学习中最容易的阶段。学会书写后，必须学习文法和词汇，然后掌握这一切，必须花时间去学，去练，去学会通过语言表达自己的观点。这一切都不会一蹴而成。学习数学也是这样。开始时，符号也是一道难逾的路障；但是实质性的问题是理解符号的含义和特性，了解如何使用它们，并且学会熟练地使用它们来描述新的问题、寻求解决方案、给出相关证明。最终，你能培养自己具有对数学优美风格的鉴赏力。当达到这一境界时，外在的符号会消失；通过符号你可直接看到其内在含义。数学的优点在于，它的规则比之自然语言的规则简单得多，它的词汇比之自然语言的词汇也少得多。因此，当你遇到一些陌生的数学问题时，你可以使用逻辑演绎和相关创造来解决这些问题，而不必请教书籍或专家。

这就是为什么数学会象程序设计那样有趣。可惜数学并不总是很容易。即使是数学家们在学习研究一个新的专业分支时，也会觉得很困难。通信进程(CSP)的理论是数学的一个新分支；因此，学习通信进程时，程序人员与数学家相比，并不处于劣势。但在程序人员最后在把获得的知识付诸实践时，会有明显的优势。

本书的内容在一些非正式的会议上和正式的学校课程教学上讲过。本书曾用于软件工程专业的硕士课程，讲授一学期。但大部分材料也可用作计算机系的本科最后一年或者大学第二年的课程来讲授。要求的预备知识主要是高中代数、集合论概念、谓词演算符号等。这些都列于序言后的汇总的符号表中。这本书的内容也可以为有经验的程序人员举办为时一周的高强度的集中培训课程。在讲授时，教师应重点讲解例子和定义，而将更多的数学成分留作课后自学。如果时间确实稀少，课时少于一周时，即使只能讲完第二章也是值得的；如果仔细选材的话，还可举办一小时的讨论班，讲到很有启发意义的五个哲学家就餐的故事。

讲授通信顺序进程是十分有趣的事情，因为书中的例子提供讲演者发挥演剧才能的机会。一个例子相当一个剧本，演出时一般着重表演有关人物的感情。听众通常会从演出中感到死锁是特别滑稽的。但是听众要始终警惕这种人格化的危险性。讲演者籍助表演动机、爱憎、感情波动，“使枯燥、荒诞的故事听起来逼真”，而数学公式有意摆脱了这些人为因素。大家应该专心致志于理解数学公式的无人情味的、干巴巴的含义，并且学会欣赏数学抽象的优美。比如，某些递归定义的算法就具有J.S.Bach所创作的逃亡曲般的惊人的优美[4]。

[4] 这一段文字充分借鉴了周巢尘院士的1988年版本的翻译。

概要(SUMMARY)

第一章介绍进程的基本概念[5]，进程是系统及其环境间交互作用的一种数学抽象。我们熟悉的递归技术可用来刻画一个生命周期很长的，或者永不消亡的进程。该章中的概念先通过例子和图形来解释；更完整的解释是由代数法则和用函数式程序设计语言的实施来给出。该章第二部分阐述一个进程的行为可由进程所从事的一系列动作的轨迹(Trace)来记载。我们定义了许多关于迹的代数和逻辑运算。在实现一个进程前，可以利用这个进程的迹的各种特性来描述它。该章中给出了一些规则，从而可以帮助确保一个进程的实现与其形式描述是满足一致性的。

第二章阐述了如何将多个进程组装成系统，在系统中各子部件交互作用，也和系统的外部环境彼此打交道。引入并发性本身并不产生任何非确定性[6]。这一章中的主要例子是经典的五个哲学家的就餐问题，本章第二部分中阐述了，如何利用改变进程能够参与的事件的名字的方法，使得进程可以很方便地具有新用途。这章最后以确定性进程的数学理论为结束，其中包括递归理论的一个简单介绍。

第三章对令人困惑的非确定性问题给出一个最简单的解。非确定性是实现抽象的一种重要技术。当我们决定忽略或者封装掩盖一个系统行为中我们不感兴趣的部分时，就会自然地会出现非确定性。非确定性也具备对称性，这种对称性可见相关算子定义的数学理论部分。非确定性进程的证明方法比确定性进程稍微麻烦些，因为必须证明每一种可能的非确定选择产生的行为都符合进程的描述。幸运的是，我们在本书中提出了一些绕过非确定性的技术，这些技术会在第四和第五章中被广泛的使用。因此，对于第三章的学习或精通掌握可一直推迟到第六章。第六章中就不能再避免引入非确定性了。

第三章的后面一些章节中，给出了非确定性进程概念的一个完整的数学定义。希望能够探究这门学科的理论基础，或者希望用证明的方法来检验书中的代数法则和其它进程特性的有效性的纯数学家们应该对这个定义会有兴趣。而应用数学家(包括程序设计人员)可能认为这些法则是不证自明的，或者通过这些法则的实用性来证明其有效性，这些读者可以不需要去阅读那些比较理论的章节。

第四章开始介绍通信：通信是两个进程间相互作用的一种特例，其中的一个进程输出一个消息，与此同时，另一个进程输入这个消息。因此，通信是同步的；如果要在通道上使用消息缓冲，可在两个进程间插入一个缓存进程。

设计并发系统的重要目的是在解决实际问题时获得高性能计算。为了阐述这一目的，书中设计了一些简单的脉动式(或迭代式)算法。管道[7]就是一个简单例子。管道定义为由一系列进程所组成，其中每一个进程仅从它的先行进程输入消息，仅向它的后继进

---

[5] Tony Hoare的原著术语是"Process"。周巢尘院士在1988年译本翻译为“进程”。本书在符号系统和概念命名上大多数沿用《通信顺序进程》(周巢尘 1988 北京大学出版社)一书。译者认为“Process”如果翻译成“过程”可能略微贴切一些，例如，控制系统中的“过程控制”。读者不要与操作系统中的进程与并发理论CSP中的进程混淆。操作系统中的进程指的是一个程序的实例(Instance)。

[6] 在CSP里，Hoare对并发问题的处理引入的主要是如何诠释同步问题，不考虑非确定性问题。非确定性问题是通过一个单独的算子来表达。

[7] 周巢尘院士的译本中为“导管”。译者认为“管道”应该更为贴切，符号现代操作系统的概念和机制。

程输出消息。管道在实现单向的，具有层次结构的通信协议时很有用。最后，在本章中我们定义了一个重要的抽象数据类型，从属进程。从属进程的每个进程实例只和其从属于的母进程通信[8]。

第五章讲述如何在通信顺序进程的框架内集成传统的顺序程序设计的算子。有经验的程序人员可能会很吃惊的发现，这些程序设计语言算子和常用的数学理论中的算子一样，具有相同漂亮的代数性质；另外，证明顺序程序满足其规约描述，和证明并发程序满足规约描述的方法非常想象。即使外部启动的中断也可用通信顺序进程定义，该章中会说明其用途。中断也遵从通信顺序进程理论的相关法则。

第六章讨论怎样构造和实现一个复杂系统，在这个系统中有很多的进程，它们共享有限的物理资源，诸如磁盘、行式打印机等，而且进程对资源的需求是随着时间而变化的。每一个资源由一个进程来表示。每当用户进程需要一个资源时，就建立一个新的虚拟资源。一个虚拟资源是一个进程，它有点象用户进程的附属进程；但可以和实际第物理资源通信。这些通信是和其它同时活动着的虚拟进程的通信穿插进行的。这里的实际和虚拟进程概念与PASCAL PLUS中的管程(Monitors)和闭体(Envelopes)扮演的角色是一样的。这一章通过用模块式开发一系列完整但是非常简单的操作系统来解释如何构造一个复杂系统。这些例子堪称本书中规模最大的例子。

第七章阐述了研究并发和通信的其他方法，并且解释了导致本书前面章节中理论的技术、历史和作者个人的动因。在本章中，我深深的感谢其他作者对我的影响，并推荐和介绍这个领域中可以进一步阅读的材料。

[8] 原文用了“Subordinate Process"，从属进程。读者可以直观的理解就是类似程序语言中的子程序的概念。

致谢(ACKNOWLEDGEMENTS)

兹深深的感谢Robin Milner先生深刻的、原创性的研究工作。其研究工作在他的开山之作通信系统演算(Calculus for Communicating Systems)中有详尽的说明。他独有的洞察力，他的友谊和他在学问上对我的的鞭策，一直是本书竟成的各项工作的灵感和勇气的源泉[9]。

在过去20年中，我一直思考着并行计算程序设计中的一些问题，并想设计一种程序设计语言以缓解这些问题。这段时间中，我极大得益于与很多科学家的合作，包括Per Brinch Hansen, StePhen Brookes, Dave Bustard, Zhou Chao Chen[10], Ole-Johan Dahl, Edsger W. Dijkstra, John El-der, Jeremy Jacob, Ian Hayes, Jim Kaubisch, John Kennaway, T. Y. Kong, Peter Lauer, Mike McKeag, Carroll Morgan, Ernst-Rudiger Olderog, Rudi Reinecke, Bill Roscoe, Alex Teruel, Alastair Tocher和Jim Welsh。

最后，特别感谢O.-J. Dahl, E. W. Dijkstra, Leslie M. Goldschlager和Jeff Sanders等人，他们仔细阅读了本书的初稿，并且指出了原稿中错误和费解处；也要特别感谢一九八三年一月参加了Wollongong暑期计算机程序设计科学讲座的参加者，一九八三年四月在中国科学院研究生院参加了我的讲习班的人们，以及一九七九年至一九八四年牛津大学计算专业的硕士生们。

[9] Robin Milner(13 January 1934 – 20 March 2010), 1991年图灵奖获得者，英国爱丁堡大学计算机教授和系主任。英国爱丁堡皇家院士，英国皇家学会院士，美国ACM院士，美国工程院外籍院士。著名的爱丁堡大学计算机系LFCS(Laboratory for Foundations of Computer Science)的创办人之一。是机器自动定理证明，函数程序语言，并发系统分析等领域的开创人物。其CCS的工作与Hoare的CSP基本上属于同一个时期的，分别独立的原创性研究工作。

[10] 中国科学院院士周巢尘院士。1958年毕业于北京大学数学力学系。1967年研究生毕业于中国科学院计算技术研究所。中国科学院软件研究所研究员，联合国大学国际软件技术研究所所长。研究生期间，研读数理逻辑，师从中国数理逻辑研究领域科学家胡世华院士。

符号表(GLOSSARY OF SYMBOLS)

逻辑符号(Logic)

记号	含义	例子
=	相等	x=x
≠	不等	x≠ x+1
□	例子或证明的结束符	
P∧Q	P和Q(两者为真)	x≤x+1∧x≠x+1
P∨Q	P或Q(两者或其一为真)	x≤y ∨ y≤x
¬ P	非P(P不真)	¬ 3>5
P=>Q	若P则Q	x<y => x≤y
P≡Q	P当且仅当Q	x<y ≡ y>x
∃x.P	存在x使P真	∃x.x＞y
∀x.P	对一切x,P真	∀x.x＜x+1
∃x：A.P	存在集合A中元素x,使P真	
∃x：A.P	对集合A中一切元素x,P真	

集合(Sets)

记号	含义	例子
∈	属于	拿破仑 ∈ 人类
∉	不属于	拿破仑∉俄罗斯人
{ }	空集(无元素集)	¬(拿破仑 ∈{})
{a}	a组成的单元素集；a是其仅有的元素	x∈{a} ≡ x=a
{a,b,c}	a,b,c组成的集	c∈{a,b,c}
{x | P(x)}	使P(x)为真的全体x的集合	{a}={x | x=a}
A ∪ B	A并以B	A∪B={x | x∈A ∨ x∈B}
A ∩ B	A交以B	A∩B={x | x∈A ∧ x∈B}
A - B	A减去B	A - B={x | x∈A ∧ ¬ x∈B}
A ⊆ B	A包含于B	A⊆B ≡ ∀x：A.x∈B
A ⊇ B	A包含B	A⊇B ≡ B⊆A
{ x ：A | P(x) }	使P(x)真的A中全体x	
ℕ	自然数集	{0,1,2...}
ℙA	A的幂集	ℙA={X | X⊆A}
$\cup_{n\geq 0} A_n$	集合族的并集	$\cup_{n\geq 0} A_n = \{x \mid \exists n\geq 0 . x\in A_n\}$
$\cap_{n\geq 0} A_n$	集合族的交集	$\cap_{n\geq 0} A_n = \{x \mid \forall n\geq 0 . x\in A_n\}$

函数(Functions)

记号	含义	例子
f：A→B	f是将A中每个元素映射到B中元素的一个函数	square：ℕ→ℕ
f(x)	(A中)x经过f得到的B中的映象	

单射	将A中元素映射到B中不同元素的函数	$x \neq y \Rightarrow f(x) \neq f(y)$
f^{-1}	单射f的逆	$x=f(y) \equiv y=f^{-1}(x)$
$\{f(x) \mid P(x)\}$	将f作用于使P为真的全体x所得到的集合	
$f(C)$	由f形成的C的映象集	$\{y \mid \exists x . y=f(x) \wedge x \in C\}$ square：$(\{3,5\})=\{9,25\}$
$f \circ g$	f 复合g	$f \circ g(x)=f(g(x))$
$\lambda x . f(x)$	将x的每个值映射至$f(x)$的函数	$(\lambda x . f(x))(3)=f(3)$

迹(Traces)

节号	记号	含义	例子
1.5	<>	空迹	
1.5	<a>	仅含a的迹(单元序列)	
1.5	<a,b,c>	由a然后b然后c构成的迹	
1.6.1	^	(迹间的)相接	<a,b,c> = <a,b> ^ <> ^ <c>
1.6.1	s^n	重复s共n次	$<a,b>^2$ = <a,b,a,b>
1.6.2	s A	s受限于A	<b,c,d,a> {a,c} = <c,a>
1.6.5	s≤t	s是t的前缀	<a,b> ≤ <a.b.c>
4.2.2	$s \leq^n t$	从t的尾部至多移走n个符号后可得到s	$s \leq^n t$
1.6.5	s in t	s在t中	<c,d> in <b,c,d,a,b>
1.6.6	#s	s的长度	#<b,c,b,a> = 4
1.6.6	s↓b	s中b的个数	<b,c,b,a>↓b =4
1.9.6	s↓c	s中记载的通道c上的通信	<c.1,a.4,c.3,d.1>↓c = <1,3>
1.9.2	^/s	压扁后的s	^/<<a,b>,<>,<c>> = <a,b,c>
1.9.7	s;t	s成功地接以t	(s^<√>);t = s^t
1.6.4	A*	A中元素的序列集	A*={s | s A=s}
1.6.3	s_0	s的首元素	$<a,b,c>_0$ = a
1.6.3	s´	s的尾部	<a,b,c>´ = <b,c>
1.9.4	s[i]	s的第i个元素	<a,b,c>[1] = b
1.9.1	$f^*(s)$	s的f星函数	square*(<1,5,3>)=<1,25,9>
1.9.4	$\overline{s}$	s的逆置	$<a,b,c>$ = <c,b,a>

特殊事件(Special Events)

节号	记号	含义
1.9.7	√	成功(成功终止)
2.6.2	τ.a	名为 τ 的进程参予事件a
4.1	c.v	在通道c上传递值v
4.5	τ.c	在通道τ.c上传递消息v
4.5	τ.c.v	名为τ的进程的通道c
5.4.1	↯	灾难(闪电)
5.4.3	⊗	交换
5.4.4	©	为恢复所设的备查点
6.2	acquire	获取

6.2	release	释放

进程(Processes)

节号	记号	含义
1.1	αP	进程P的字母表
4.1	αc	在通道c上可传递的消息的集合
1.1.1	a→p	a然后P
1.1.3	(a→p | b→Q)	在a然后P与b然后Q间的选择(如果a≠b)
1.1.3	(x：A→P(x))	在A中选取x然后P(x)
1.1.2	μX：A.F(X)	满足X=F(X)的字母表为A的进程X
1.8.3	P/s	执行迹s中事件后的P
2.3	P ‖ Q	P和Q并行执行
2.6.2	*l* ：P	P冠以名*l*
2.6.4	L:P	P冠以集合L中的名字
3.2	P ⊓ Q	P或Q(非确定性的)
3.3	P□Q	P和Q间的选择
3.5	P\C	除去C后的P(藏匿)
3.6	P|||C	P和Q的穿插
4.4	P 》 Q	P链接以Q
4.5	P // Q	P附属于Q
6.4	*l*:: p // Q	远程附庸
5.1	P;Q	P(成功地)随之以Q
5.4	P^Q	P被Q中断
5.4.1	P↯Q	P在遇到灾难后执行Q
5.4.2	$\widehat{p}$	可再启动的P
5.4.3	P⊗Q	P和Q交替执行
5.5	P≮b≯Q	若b则P，否则Q
5.1	*P	重复执行P
5.5	b*P	若b则重复P
5.5	x：=e	x取(值)e
4.2	b!e	在(通道)b上输出(值)e
4.2	b?X	从(通道)b上向x输入
6.2	*l* ！e?x	调用名为*l*的共享子程序，值参为e，结果送至x
1.10.1	P sat S	(进程)P满足(规约)S
1.10.1	tr	给定进程的任意迹
3.7	ref	给定进程的任意拒绝集
5.5.2	$x^{\surd}$	给定进程所产生的x的终止值
5.5.1	var(P)	可由P赋值的变元集
5.5.1	acc(P)	可由P引起的变元集
2.8.2	P⊑Q	(确定性地)Q能完成P能做的事情
3.9	P⊑Q	(非确定性地)Q不比P差
5.5.1	*D* e	表达式e有定义

代数(Algeba)

术语	含义
自反(reflexive)	满足xRx的关系R
反对称(antisymmetric)	满足xRy∧yRx⇒x=y的关系R
传递(transitive)	满足xRy∧yRz⇒xRz的关系R
偏序(partial order)	自反、反对称和传递的关系≤
底(bottom)	满足⊥≤x的最小元素⊥
单调(monotonic)	保持偏序的函数f，即x≤y ⇒f(x) ≤f(y)
严格(strict)	保持底值的函数f，即f(⊥)=⊥
幂等(idempotent)	满足x f x = x的二元算子f
对称(symmetric)	满足x f x = y f x的二元算子f
结合(associative)	满足x f (y f z)= (x f y) f z的二元算子f
分配(distributive)	f可分配入g，若x f (y g z) = (x f y) g (x f z) 而且 (y g z) f x = (y f x) g (z f x)
单元位(unit)	满足x f 1 = 1 f x = x的元素1，是f的单位元
零元(zero)	满足x f 0 = 0 f x = 0的元素0.是f的零元

图(GrαPhs)

术语	含义
图(grαPh)	一种图状的，通过结点和边的关系表达方法，
节点(node)	图中的一个圆点，表示该关系的定义域或值域中的一个元素
弧(arc)	图中一线段或一箭头，连接满足该关系的两个节点
无向图(undirected grαPh)	对称关系的图
有向图(directed grαPh)	非对称关系的图，常画作箭头
有向回路(directed cycle)	由方向相同的箭头组成的回路中的节点集
无向回路(undirected cycle)	方向不定的弧或箭头组成的回路中的节点集

第一章 进程(PROCESSES)

1.1 引言(Introduction)

让我们暂时把计算机和计算机程序设计忘掉一会儿，想想我们周围世界的各种对象[11]，这些对象依据各自的某种行为方式在活动，并与我们或其它对象发生相互作用。这类对象如钟表、筹码机、电话机、博弈游戏和自动售货机等。要描述它们的行为，我们首先要确定我们对这些对象的哪类事件或动作感兴趣，然后给每类事件赋予一个不同的名称。

就拿一个简单的自动售货机来说，可有两类事件：

coin — 将一枚硬币投入自动售货机的硬币槽；

choc — 从机器的发货器送出一块巧克力。

而对较为复杂的自动售货机来说，就有更多种类的事件(event)[12]：

in1p — 投入一枚一便士的硬币；

in2p — 投入一枚两便士的硬币；

small — 送出一块小饼干或小甜饼；

large — 送出一块大饼干或大甜饼；

out1p — 送出一便士的找头。

注意每个事件名代表的是一个事件的*类别(Class)*；同一类的事件会在不同的时间场合出现很多次[13]。事件类别和具体出现的一个事件之间的区别与字母h的情形类似。同一字母h在本书中可以多次出现，只是出现在不同的地方而已。

我们把认为是与描述一个对象行为有关的全体事件的名称的集合叫做这个对象的*字母表(alphabet)*。字母表是一个对象的一个预先规定好的永久属性。从逻辑上讲，对象不可能执行其字母表以外的事件；譬如说，专售巧克力的售货机不可能突然间送出一艘玩具军舰来。但这个命题的逆命题并不成立。对象字母表内的事件也不一定会发生[14]。例如，专售巧克力的售货机可能就真的不能再吐出一个巧克力了——也许是因为售货机里还没装巧克力，也许是售货机坏了，也许是没人买巧克力了。但一经确定choc事件是在自动售货机字母表内，即使该事件从来没有发生过[15]，也仍然被认为是属于自动售货机的事件。

选择一个对象的字母表一般要涉及到一点有意简化的问题，例如，舍弃那些我们不感兴趣的属性和动作。例如，我们不需要刻画自动售货机的颜色、重量和形状。另外，我们也有意舍弃某些即使与售货机本身密切相关的事件，如补充巧克力或倒空硬币盒——因为这些事件与顾客没关系(也不应有什么关系)。

在一个对象的生命周期中，每一个实际发生的事件应被看作是无延时的瞬息动作或原子动作。对于一个延续的或占时间的动作需要通过两个单独的事件来表示，一事件表示动作的开始，另一事件表示动作的结束；从动作的开始事件的发生到动作的结束事件的发生，有一段间隔，动作的延时就由这段间隔来代表；在这段间隔内，还可能发生其它事件。如果有两个延续动作，前一动作的还未结束，后一动作就开始了，则它们在时间上发生重叠。

---

[11] 原文是Objects。泛指一个对象，可以是一个物理的实体，或者一个虚拟的实体。但在CSP中，读者不要与Object Oriented Programming的Object混淆。在OO中，Object是Class的一个Instance。在本书中，Object(对象)有点Class的含义，泛指物体，实体。

[12] 一个事件(event)通常意味着一个动作(action);但一个动作可以是多个原子事件的组合。

[13] 类似一个事件(Class)可以出现许多实例(Object Instance)

[14] 或者永远不发生。只是属于行为集合的一部分。可能发生，但不一定能够触发。类似每个人都可能做总统，但要看运气。

[15] 原文是"that event never actually occurs"，意味着这个事件可以从来没有发生过。而非"不再发生"。

我们有意忽略的另一个细节是事件发生的具体时间。这样做的好处在于简化了事件的设计和论证过程，而且可应用于具有任何运算速度的性能的物理计算系统。在那些对响应时间至关重要的应用场景下，需要单独考虑这些实时性问题。这些考虑是独立于设计逻辑正确性方面的。一个高级程序设计语言是否成功的必要条件就是与时序无关[16]。

忽略了事件的时间这个问题的好处是，我们不需要回答，也不需要问某一事件是否与另一事件丝毫不差地同时发生这些问题。当两个事件发生的同时性很重要时(如同步情形)，我们就把它们当做是单个事件的发生；不重要时，我们就将两个可能同时发生的事件按任意前后次序记录下来。

选择字母表时，没有必要区别由对象引发的事件(如choc，吐出一个巧克力)和由对象外部的某个因素引发的事件(如coin，购物者主动投币)。在CSP事件处理时，避开因果的概念能大大简化CSP理论及其应用。

从现在开始，我们用进程(Process)这个词来代表对象的行为模型，并认为一个进程的行为可以通过由组成其字母表的有限事件的集合来刻画。本书中，我们遵循如下约定：

1. 用小写的字符串表示不同事件，如：
   coin, choc, in2p, out1p
   有时也用字母来表示事件，如：a, b, c, d, e
2. 用大写的字符串表示相关具体的进程，如：
   VMS——简单自动售货机
   VMC——复杂自动售货机
   在后面法则中出现的字母P,Q,R表示任意进程。
3. 字母x, y, z是表示事件的变量。
4. 字母A, B, C表示事件集合。
5. 字母X, Y是进程变量。
6. 进程P的字母表记作αP, 如：
   αVMS= {coin, choc}
   αVMC= {in1p, in2p, small, large, out1p}

以A为字母表，但从不实际执行[17]A中事件的进程，叫做$STOP_A$。$STOP_A$刻画了一个毁坏了的对象的行为：尽管这个对象具备了执行A中事件的物理能力，但它从不执行这些能力。值得注意的是，字母表不同的对象，即使不做任何事情，它们也是不同的。例如，STOPαVMS的行为是本可以吐出一块巧克力的。而STOPαVMC却永远不可能给出一块巧克力来，只能给出饼干。顾客即使根本不知道这两台自动售货机都坏了，也知道上述事实[18]。

在引言的余下部分中，我们要定义一些简单的符号约定，以帮助我们来表述能够实际做一些事情的对象。

1.1.1 前缀(Prefix)

设x为一事件，P为一进程，则

---

[16] 从另外一个角度而言，CSP是从编程语言的角度来探索并发问题，例如，通过并发算子的能力。CSP比较关注的是并发程序的逻辑正确性问题，不涉及时序逻辑问题。

[17] 或者从来不被外在的环境触发。

[18] 在第5章－顺序进程一章中，介绍了一个特殊进程$SKIP_A$。与STOP进程代表一个处理过程的非正常不工作(例如，死锁(Deadlock)或者活锁Livelock))不同的是，SKIP进程代表一个处理过程的正常终止。如果不考虑一个Process的递归调用，一个Process在CSP的符号系统里必须要么是以STOP异常终止，或者SKIP成功结束。读者可以参阅5.1节关于SKIP进程的定义。这个概念建议与STOP同时学习比较好。

(x → P)　(读做“x 然后 P”)

刻画了这样的一个对象的进程：它首先执行事件x，然后按照进程P的说明进行动作[19]。我们定义进程(x→P)与进程P有同样的字母表，所以，只有当x在P的字母表内时，这个记法才有意义[20]；形式地记为：

当x ∈ αP时，　α(x → P) = αP

例子

X1 一台简单自动售货机在损坏前接受了一枚硬币，记为：

(coin→STOPαVMS)　□

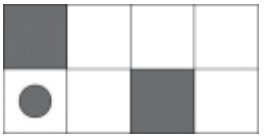
X2 一台简单自动售货机在损坏前成功地为两位顾客服务，记为：

(coin →(choc →(coin →(choc →STOPαVMS))))

最初，售货机只接受投入硬币槽内的一枚硬币，不允许先送出巧克力。当第一枚硬币投入后，硬币槽就关闭，直到一块巧克力被送出后再打开。这台售货机不会连续接受两枚硬币，也不会连续给出两块巧克力。　□

我们约定：→是一个右结合，这样以后我们可略去事件的线性序列中的括号，例如X2中的括号。

X3 一个筹码从一块板的左下方开始移动，只允许向上或向右移入相邻的空白方格内。



αCTR = {up, right}

CTR = (right→up→right→right→STOPαCTR)　□

注意算子→的右侧总是进程，左侧总是单个事件。如果P和Q为进程，则

P→Q

在语法上是不正确的。某一进程首先按P进程动作，然后按Q进程动作，对这类进程的正确表述方法将在第四章中讲解。类似地，如果x，y为事件，从语法上讲，下述写法也是错误的[21]。

---

[19] 是一种递归定义的方法。一个进程的行为，是通过另外的进程行为来表达。

[20] (x → P)的含义是发生x事件发生后，一个处理过程(Process)下一步的处理过程(Process)。

[21] CSP里进程的概念有点不精确。例如，如果y是第五章里定义的成功结束事件"√"，那么x→y 其实应该等价于 x→SKIP的行为。但在"→"的定义下，x→y不是进程的语法，需要x→√→SKIP才是表达了一个进程。

$x \rightarrow y$

这类进程正确表述是

$x \rightarrow (y \rightarrow STOP)$

这样我们就将事件的概念和进程的概念区别开了，一个进程也许参与很多事件，也许什么也不做。

1.1.2 递归(Recursion)

对一个最终要结束的进程，我们可以用上述的前缀记法表述它的全部行为。但是如果将一台自动售货机在其预期的最长生存期内的全部行为都罗列出来，却是件极为无聊的事情；因此，我们需要有一种方法，用短得多的符号语法将重复的行为刻画出来。最好在使用这些语法时，不需要事先确定对象的寿命长短；这样，不受时间限制的不断动作并与其环境互相作用的对象，也就可以被表述了。
如果不考虑上弦的动作，钟是一种最简单的、有可能永远动作的对象，它只会滴答滴答地走，即

$\alpha CLOCK = \{tick\}$

下面我们考虑一个对象，除一开始先走一下外，其行为与这座钟完全相同，可记为

$(tick \rightarrow CLOCK)$

这个对象的行为是与现实中那座钟的行为是无法区分，等价的。所以，这一推理导出下列方程

$CLOCK = ((x \rightarrow P))$

这个方程可以看作是钟的行为的一个隐性的定义，就象2的平方根可定义为下列方程中x的正数解一样

$x = x^2+x-2$

由闹钟的方程可得到一些不言而喻的结果，这些结果可以通过简单的等式代换得到。

$CLOCK = (tick \rightarrow CLOCK)$	原方程
$= (tick \rightarrow (tick \rightarrow CLOCK)$	经过代换
$CLOCK = (tick \rightarrow tick \rightarrow tick \rightarrow CLOCK)$	类似的持续代换得出

方程式可按需要进行多次展开，展开的次数不受限制。钟的潜在的无限行为可定义为

$tick \rightarrow tick \rightarrow tick \rightarrow ...$

这就象2的平方根被认为是十进制数列

1.414...

的极限值。
这种通过自我引用的递归定义一个进程行为的方法，只有当方程式右边所有出现相应

的递归重复调用的进程名之前，至少存在一个前缀事件时[22]，这种递归的进程定义方法才是有效的。例如，递归方程

X=X

不能说明任何问题，因为方程的解是任意的。**我们把以前缀开始的进程表达式称作是卫式表达式(guarded)[23]。如果F(X)是含进程名X的卫式表达式，A为X的字母表，则我们断言方程**

X=F(X)[24]

有唯一解。为方便起见，我们把这个解表示为

μX:A • F(X)[25]

的形式。这里，X是局部名(约束变量)，并且可以随意更换，即

μX:A • F(X) = μY:A•F(Y)

这个等价性可通过方程

X=F(X)

中的变量X的解也是方程

Y=F(Y)

中变量Y的解来理解。

以后，我们或者用方程式，或用更加方便的μ表达方式给出进程的递归定义。在用μX:A.F(X)定义时，如果字母表A从进程的内容或上下文中已经很清楚了，我们就略去不写。

例子

X1 一只永动钟，它的递归定义是

CLOCK = μX : {tick}.(tick→X) □

X2 一台简单自动售货机，投入多少枚硬币就送出多少块巧克力，其方程递归定义是

VMS =(coin→(choc→VMS))

如上所述，这个方程式可以更形式化的通过μ的方式定义为：

VMS =μX : {coin, choc}.(coin→(choc→X))

---

[22] 前缀(prefix)定义为(x → P), 其中x是一个事件，P是一个进程。例如，(tick→CLOCK)的递归定义中，(tick→CLOCK)就是一个前缀: tick是事件，CLOCK是自我引用的递归进程的名字。

[23] 读者可参阅Dijkstra的经典论文"Guarded commands, nondeterminacy and formal derivation of programs", Communications of the ACM Volume 18 Issue 8, Aug. 1975 Pages 453-457.

[24] 这里的大写X是代表递归中的进程的符号(名)，例如，CLOCK，VMS等。

[25] 卫式递归方程式定义为存在唯一解，而非类似 X = X，可以有多个解或者任意多个解。严格的数学证明会在后面给出。

这种μ的方式是方程式的另一种写法[26]。 □

X3 一台换钱机，投入五便士的硬币，它自动根据处理流程，换出零钱，表述为

$$\alpha CH5A = \{in5p, out2p, outlp\}$$
$$CH5A = (in5p \rightarrow out2p \rightarrow outlp \rightarrow out2p \rightarrow CH5A)$$ [27] □

X4 一台换钱方式不同的机器，其进程的字母表与上例相同，定义为

$$CH5B = (in5p \rightarrow out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow CH5B)$$[28] □

关于卫式的方程有解，且这个解是唯一的断言，可以通过字符串代换的方法非形式地证明。用方程的完整右侧对方程式里含有该进程自我引用的地方做一次代换，会导致定义进程行为的表达式随之增长，不断的持续替换会使得原始方程式所刻画的行为的初始部分的行为字符串越来越长。因此，通过这种替换的方法，任何有限长的行为都可以确定下来。在任一时刻动作都相同的两个对象具有同样的行为，例如，两个对象同属于同一个进程。目前觉得这种非形式的推理方法不好理解或不能令人信服的读者，可以暂时把这个结论当作公理加以接受，这条公理的价值和意义会越来越明显[29]。在没有给出进程的精确的数学定义前，严格的形式证明是无法给出的。这个定理的形式证明将在第2.8.3节中给出。在这里对递归的叙述主要以递归方程的卫式特性为基础。我们将在3.8节中讨论非卫式的递归的含义。

1.1.3 选择(Choice)

使用前缀和递归的方法，我们可以刻画那些只具备单一行为的对象，这类对象的行为不受外界影响。然而，有许多对象在其所处的环境中与环境相互作用，致使对象的行为受到环境影响。例如，一台自动售货机可能有两个硬币槽，一个只能投入二便士的硬币，另一个只能投入一便士的硬币；选择触发哪一事件是由顾客来决定的。假设x和y是两个不同事件，则表达式

$$(x \rightarrow P \mid y \rightarrow Q)$$

表述了这样一种对象，一开始时，它既可以执行事件x，也可以执行事件y。当第一个事件发生后，如果发生的是事件x，则对象的后续行为按P进行动作；如果是事件y，则对象按Q进行动作。因为x和y不同，则P和Q之间的选择就是由实际发生的第一个事件来确定。与以前一样，我们继续延用字母表的一致性，即有

---

[26] 通过μ的方式，一个递归进程可以通过X和F(X)来表达，这样可以避免方程表达方式的束缚，然后可以把一个μX.F(X)的表达作为一个符号系统的实体，去参加各种代数(Algebra)演算。这是CSP理论里一个非常重要的技巧从而可以把进程行为的研究转换为进程代数研究。

[27] CH5A的代数形式定义为：$CH5A = \mu X : \{in5p, out2p, out1p\} \cdot (in5p \rightarrow out2p \rightarrow out1p \rightarrow out2p \rightarrow X)$

[28] $CH5B = \mu X : \{in5p, out2p, out1p\} \bullet (in5p \rightarrow out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow X)$

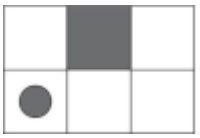
[29] 关于卫式的方程有解，且这个解是唯一的断言

$\alpha(x \rightarrow P \mid y \rightarrow Q) = \alpha P$　假设 $\{x, y\} \subseteq \alpha P$ 且$\alpha P = \alpha Q$时[30]

“|”读做“选择”：“选择x则按P执行，选择y则按Q执行”。

例子

X1 在下面这块板上筹码的移动可由下述进程所定义



$(up \rightarrow STOP \mid right \rightarrow right \rightarrow up \rightarrow STOP)$ □

X2 一台机器同时可提供两种方法破开五便士的硬币(比较1.1.2中X3和X4，在这两例中没有选择余地)，此机器的行为可定义为

$$CH5C = in5p \rightarrow (outlp \rightarrow outlp \rightarrow outlp \rightarrow out2p \rightarrow CH5C \mid out2p \rightarrow outlp \rightarrow out2 \rightarrow CH5C)$$

由换钱的顾客进行选择。 □

X3 一台售货机，其每笔交易或卖巧克力或卖太妃糖，定义为

$$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$$ □

X4 一台更为复杂的售货机，在可使用的硬币、出售的货品和找钱的方式上，都有选择余地，可定义为

$$VMC = (in2p \rightarrow (large \rightarrow VMC \mid small \rightarrow outlp \rightarrow VMC) \mid inlp \rightarrow (small \rightarrow VMC \mid inlp \rightarrow (large \rightarrow VMC \mid inlp \rightarrow STOP)))$$

象许多复杂的机器一样，这台售货机有一个设计上的缺陷，顾客不能接连投入三个便士的硬币[31]。但是改一下用户手册要比改进机器容易的多，所以，我们在机器上加上一条注意事项

“注意：不要接连投入三个便士。” □

X5 一台售货机允许顾客先品尝一块巧克力，然后再付钱。它也允许先付后尝。此进程写为

---

[30] 字母表都是一致的假设很重要。因为后续的P或者Q进程里包括一个递归定义，因此P的字母表里必须有x事件。

[31] 观察VMC的行为定义，如果依次投入3个便士，进程的行为序列会走到STOP的地方。机器不再工作。

$$VMCRED = \mu X \bullet (coin \rightarrow choc \rightarrow X \mid choc \rightarrow coin \rightarrow X) \qquad \square$$

X6 为了避免损失，使用VMCRED时必须先付使用费。改良的进程如下

$$VMS2=(coin \rightarrow VMCRED)$$

这台售货机最多可允许连续两次投入硬币，然后最多连续送出两块巧克力；但是先付了多少钱，售货机不会送出更多的巧克力，决不多给。 □

X7 一个拷贝进程执行下列事件：

in.0 由进程的输入通道输入0
in.1 由进程的输入通道输入1
out.0由进程的输出通道输出0
out.1由进程的输出通道输出1

这一进程的行为是成对事件的重复发生。在每个周期里，它输入一位数，又将这一位数输出，记为

$$COPYBIT = \mu X \bullet (in.0 \rightarrow out.0 \rightarrow X \mid in.1 \rightarrow out.l \rightarrow X)$$

注意这一进程允许由环境选择输入的值，但输出时环境就不能选择(影响)了。在第四章中我们定义和处理通信问题时，这一点是输入和输出的主要区别。 □

选择的定义可以很容易的推广到两个“选择”以上的情形，即

$$(x \rightarrow P \mid y \rightarrow Q \mid \ldots \mid z \rightarrow R)$$

注意选择符号 | 不是进程之间的算子；因此对进程P和Q，P | Q的写法是有语法错误的[32]。不能这样写的原因是，我们想回避给表达式

$$(x \rightarrow P) \mid (x \rightarrow Q)$$

赋予任何含义，这个表达式看上去好象向环境提供了选择第一个事件的权利，但实际上又没有做到这一点[33]。解决这个问题的方法是引入非确定性，这将在3.3节中介绍。现在，假设x，y，z是不同事件，则

$$(x \rightarrow P \mid y \rightarrow Q \mid z \rightarrow R)$$

应被看作是单个算子，这个算子有三个变元P,Q,R。千万不能把它

$$(x \rightarrow P \mid (y \rightarrow Q \mid z \rightarrow R))$$

[32] CSP的Choice需要显示的用(x→P | y→Q)的方式来表达对不同的事件(x或者y)，相应的不同的处理进程。
[33] 假设环境触发了事件x，系统不知道该执行P还是Q。引起混乱。

混为一谈，后者犯有语法错误[34]。

总之，设B为事件集合，对B中每个x，表达式P(x)定义了一个进程，则

$$(x:B \rightarrow P(x))$$

定义了这样一个进程，开始时，它允许选择集合B中任意事件y，然后按$P(y)$继续执行。这个进程定义表达式应读作"从B中选择x然后按进程P(x)动作"。表达式中x是局部变量，因此有

$$(x:B \rightarrow P(x)) = (y:B \rightarrow P(y))$$

集合B定义了进程的初始菜单[35]，因为它给出了进程启动时，可以选择的动作的集合。

例子

X8 一个进程，它在任何时候都能执行其字母表A中的任何事件，则它为

$$\alpha RUN_A = A$$
$$RUN_A = (x:A \rightarrow RUN_A)$$ □

如果在初始菜单内只含一个事件e时，则由于e是唯一可能的初始事件，就有

$$(x:\{e\} \rightarrow P(x)) = (e \rightarrow P(e))$$ [36]

在一种更为特殊的情形里，初始菜单是空的。这时，不会发生任何事件，因此有

$$(x:\{\} \rightarrow P(x)) = (y:\{\} \rightarrow Q(y)) = STOP$$[37]

二元选择算子|也可以用下面更为一般的记法来定义

$$(a \rightarrow P \mid b \rightarrow Q) = (x:B \rightarrow R(x))$$

这里$B = \{a, b\}$, 而且$R(x) = \text{if } x = a \text{ then } P \text{ else } Q$

在三个或三个以上对象之间的选择可以类似地表述出来。这样，我们把选择、前缀和STOP定义为一般选择记法的特殊情形。这在我们制定进程必须遵从的一般法则(1.3节)，以及进程实施(1.4节)中将大有用处。

1.1.4 联立递归(Mutual Recursion)

通过递归手段可以把单个进程定义为一个单一方程和此方程的解[38]。我们可容易地把它

---

[34] $(y \rightarrow Q \mid z \rightarrow R)$是一个进程。不能直接跟在Choice"|"后面。属于语法错误。在"|"后面，需要首先是一个event。

[35] 类似面向对象语言里，一个对象对外的Public的Method方法。通过这些方法，一个对象可以被触发，或者接受外部的调用。

[36] 当选择事件是唯一时，选择就变成了前缀。换言之，前缀是选择的特殊形式。

[37] STOP进程的一个形式定义。不接受任何外部的触发，不响应任何外部的事件了。

[38] $\mu X:A \bullet F(X)$

推广到：多个未知的进程是联立方程组的解。这个推论要成立的条件是：联立方程的右侧都必须为卫式的，且每个未知进程都在某一个方程的左侧恰好出现一次。

例子

X1 饮料机上有两个分别标为橘子汁(ORANGE)和柠檬汁(LEMON)的按键。两个按键触发的动作定义为setorange和setlemon。给出饮料的动作定义为orange和lemon。饮料机最后吐出那种饮料决定于客户按下了那个按键。以下是定义三个进程的字母表和行为的方程

$$\alpha DD = \alpha O = \alpha L = \{setorange, setlemon, orange, lemon\}$$
$$DD = (setorange \rightarrow O \mid setlemon \rightarrow L)$$
$$O = (orange \rightarrow O \mid setlemon \rightarrow L \mid setorange \rightarrow O)$$
$$L = (lemon \rightarrow L \mid setorange \rightarrow O \mid setlemon \rightarrow L)$$

非形式地说，当第一个事件发生后，饮料机就处于O状态或L状态。在这两种状态下，它或者给出相应的饮料，或者还可以改变想法转换到另一选项去。重复选择是可以的，但没有任何印象，因为是一个简单的递归重复。 □

采用带下标的变量，我们就可以定义无穷方程组。

例子

X2 一个对象以地面为起点，可以向上运动。以后任意时刻它都可以向上或向下运动，除非它落到地上，不能再向下落了。而且规定当它在地上时，可向四周运动。假设自然数$n \in \{0, 1, 2...\}$。对每个n，我们引入一个加下标的名字CT_n，用它来表述这个对象离地第n次运动的行为方程。对象的初始行为定义为

$$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$

剩下的无穷方程组为

$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$

其中n为自然数0, 1, 2...

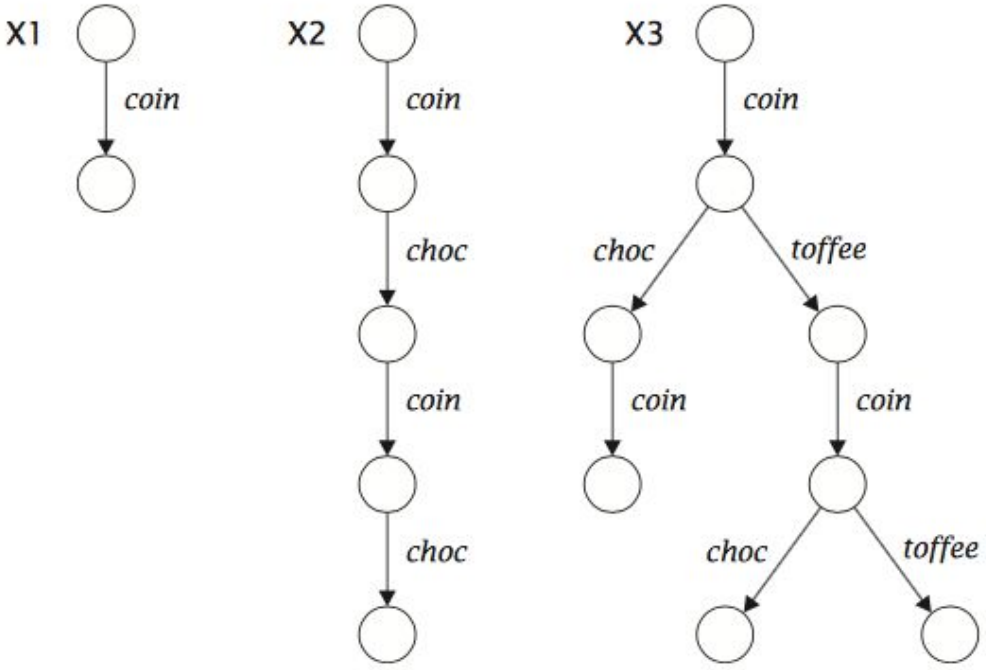
归纳定义的方程式是否有效，一般情形下需要看每个方程右侧所用下标是否比左侧用的小。而这里，CT_{n+1}却是由CT_{n+2}给出的。因此，这个定义式只能被看做是一个无穷的联立递归定义方程组，它是否有效就要看每个方程右侧是否是卫式。 □

1.2 示意图(Pictures)

用树形结构图作为进程行为的示意图，有时是很有帮助的，树形图由箭头连起来的一些小圆圈组成，在状态机的传统术语里，这些圆圈代表进程的各个状态，箭头表示状态间的转换。树根上的圆圈(一般画在树形图上方)是初始状态；整个进程沿着箭头方向向下进行。每个箭头旁边标有实现状态转换时发生的事件。在由同一节点引出的不同

的箭头或者有向边必须是不同的事件触发的。

例子(1.1.1节中X1, X2; 1.1.3节中X3)



□

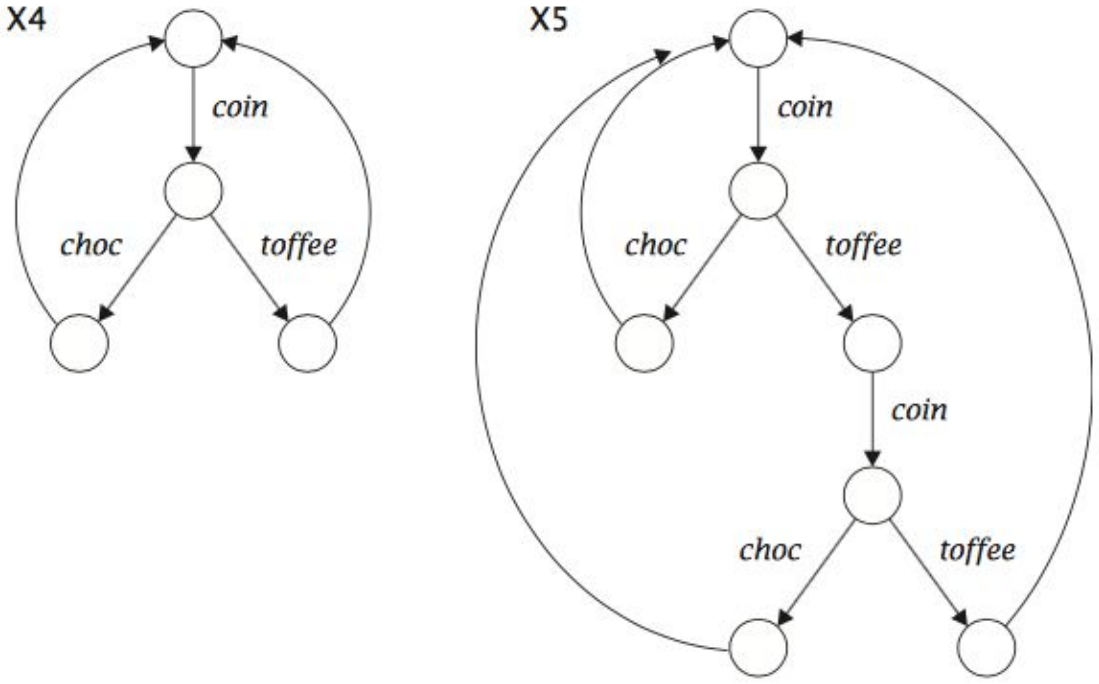
在这三个例子中，每个树的各个分支都以STOP结束，STOP表示为一个不引出箭头的圆圈。如果用图示法表示具有无限行为的进程，我们得引入另一个约定，即有一种没有任何标记的箭头，它可沿树上叶点绕回到前面的某个圆圈去。这个约定是说，当一个进程走到了在这种箭头的尾部所在的节点时，它就立即神不知鬼不觉地回到该箭头所指向的节点[39]。

显然，下面两个不同的示意图所描绘的是同一个进程(1.1.3节的X3)。然而，要用图示的方法来证明它们是相同的[40]，则是很不容易的事。这是图示法的一个弱点。

图示法的另一弱点是不能描绘具有很多很多个或有无穷个状态的进程，例如，我们做联立方程组里的例子CT_0。

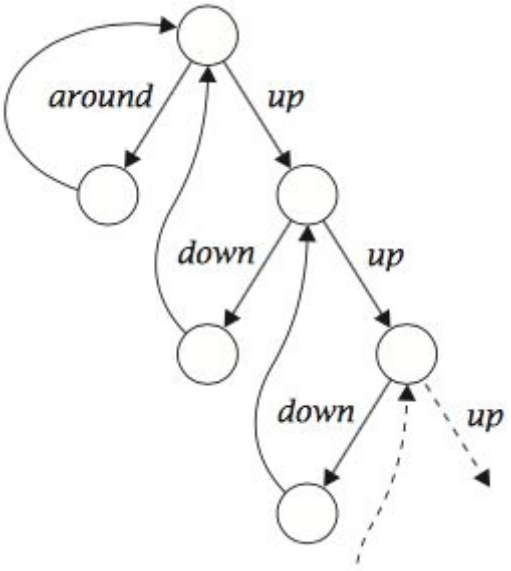
[39] 如果指向根节点，就是一个递归进程的行为。例如，前面谈及的自动售货机。

[40] X4与X5都是对1.1.3节的X3的进程VMCT进程行为的图示化。X5与X4的区别是：X5做了一次递归迭代展开，然后才循环回到根节点的。这种人的理性可以很简单判断出来的两个图是等价的关系，对于图计算来说可能可以很复杂。



□

要画出CTo整个示意图，实在没有那么多地方。只有65536种不同状态的筹码进程的示意图，就够我们画好一阵了。



1.3 法则(Laws)

即使目前为止我们介绍和使用了相对严格的符号系统，我们发现要刻画一个进程的行为，还是存在许多不同的方法。譬如，被选择的事件的表述次序，显然不应该有多大关系，即

$$(x \rightarrow P \mid y \rightarrow Q) = (y \rightarrow Q \mid x \rightarrow P)$$

而另一方面，能动作的进程和什么都不能做的进程显然不等同。

$$(x \rightarrow P) \neq STOP$$

要正确理解符号的含意，且能准确无误地使用它们，就必须学会识别哪些表达式刻画了同一对象，哪些没有。这就象懂代数的人都知道(x+y)和(y+x)代表同一个数。我们可以运用一些代数法则证明或者证伪两个字母表相同的进程的等同性。这些关于进程的代数法则与算术中的法则很类似。
第一条法则(L1)是处理选择算子的(1.1.3节)。它说明两个由选择算子定义的进程，如果它们第一步的选择就不同，或者第一步是相同的，但以后的行为不同，则这两个进程就不同。但是如果两个进程的初始选择相同，其后续所以的行为也相同，则这两个进程显然是一样的[41]。

L1 $(x : A \rightarrow P(x)) = (y : B \rightarrow Q(y))$
$\equiv (A = B \wedge \forall x \in A \bullet P(x) = Q(x))$

在本书的任何地方，我们都假设方程式两边的进程的字母表都相同，这一点以后不再单独说明。

法则L1有几个推论：

L1A $STOP \neq (d \rightarrow P)$

证明 $LHS = (x:\{\} \rightarrow P)$ 由定义(1.1.3节结尾)[42]
$\neq (x:\{d\} \rightarrow P)$ ∵ $\{\} \neq \{d\}$
$= RHS$ 由定义(1.1.3节结尾)

L1B $(c \rightarrow P) \neq (d \neq Q)$ 如果$c \neq d$

证明 $\{c\} \neq \{d\}$[43]

L1C $(c \rightarrow P \mid d \rightarrow Q) = (d \rightarrow Q \mid c \rightarrow P)$

证明

定义
$R(x) = P$ 当x=c时
$= Q$ 当x=d时

$LHS = (x:\{c, d\} \rightarrow R(x))$ 由定义[44]
$= (x:\{d, c\} \rightarrow R(x))$ ∵ $\{c, d\} = \{d, c\}$
$= RHS$ 由定义

L1D $(c \rightarrow P) = (c \rightarrow Q) \equiv P = Q$

---

[41] 从状态图的角度，就是两个进程的状态机对各个事件的响应是完全一致的。两个图是等价的。
[42] STOP进程定义为不响应任何事件的异常进程。不会有任何事件能够触发进程，并进行到P的过程。
[43] 根据L1，最开始的事件不一样，整个进程的行为当然不一样。
[44] 参阅1.1.3节关于Choice的讨论。

证明 $\{c\} = \{c\}$[45]

用以上法则可证明一些简单的命题。

例子

X1 $(coin \rightarrow choc \rightarrow coin \rightarrow choc \rightarrow STOP) \neq (coin \rightarrow STOP)$
证明 由L1D，再由L1A即可得证。 □

X2 $\mu X \bullet (coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X))$
$= \mu X \bullet (coin \rightarrow (toffee \rightarrow X \mid choc \rightarrow X))$
证明由L1C即可得证。

为了证明递归定义进程方面更普通的命题，我们还需引入一条新法则，即每个卫式递归方程有唯一解。

L2 如果$F(X)$是卫式表达式，则有

$$(Y = F(Y)) \equiv (Y = \mu X \bullet F(X))$$

紧接着一个重要推论是，$\mu X \bullet F(X)$确实是有关方程的解：

L2A $\mu X \bullet F(X) = F(\mu X \bullet F(X))$[46]

例子

X3 假设$VM1 = (coin \rightarrow VM2)$, 且$VM2 = (choc \rightarrow VM1)$, 要证明$VM1 = VMS$[47]。
证明

$VM1 = (coin \rightarrow VM2)$ 根据VM1定义
$= (coin \rightarrow (choc \rightarrow VM1))$ 根据VM2定义

所以，VM1是与VMS递归方程完全一样的进程方程的一个解。由于方程式是卫式的，因此它只有一个唯一解。于是，VM1和VMS只是该唯一解的两个不同名字而已。
这个命题的成立很显然的，不需要通过证明来增加可信度。该证明的唯一目的是要通过例子，来说明这些法则的强大，可以用来证明这类事实。当我们证明某些显而易见的事实，而使用的法则却没有那么明显时，要详细检查每一步证明，防止出现循环论证。

法则L2可推广到联立递归。我们可以通过使用下标的方式把联立递归方程组的一般形式记为

---

[45] 根据L1的法则，如果起初事件集合相同，并且整个进程的行为也相同，那么初始事件之后的进程行为必须严格相同。因此，对于L1D而言，起初事件只有相同的c，显然P必须等价于Q如果$(c \rightarrow P) = (c \rightarrow Q)$。反之亦然。
[46] $Y = F(Y)$方程有唯一解，这个解是$Y = \mu X \bullet F(X)$。把解代入方程，即得到$\mu X \bullet F(X) = F(\mu X \bullet F(X))$
[47] VMS方程定义和递归形式定义可参阅1.1.2节的X2。$VMS = (coin \rightarrow (choc \rightarrow VMS))$

$X_i = F(i, X)$　　对所有$i \in S$

这里
S是下标集合，每个方程的下标是S的一个元素；
X是进程数组，其下标变化范围为S；
F(i, X)是卫式表达式。

在这些条件下，法则L3说明只存在一个唯一的数组X，其元素满足所有方程。即

L3 在上述条件下，

if $(\forall\ i : S \bullet (Xi = F(i, X) \wedge Y_i = F(i,Y)))$ 则 $X = Y$[48]

1.4 进程的实施(Implementation of processes)

到目前为止，可表述的进程可表示为以下形式

$(x:B \rightarrow F(x))$

其中F是符号到进程的对应函数，集合B可以是空集(如STOP), 或只有一个元素(如前缀情形)，或多个元素(例如，多种选择的情形)[49]。对递归定义的进程，我们一直强调所用递归式必须是卫式的。因此，递归定义的进程可形式化的记为

$\mu X \bullet (x:B \rightarrow F(x,X))$[50]

而且，应用L2A，这个表达式可按要求的形式展开

$(x:B \rightarrow F(x, \mu X \bullet (x:B \rightarrow F(x,X))))$[51]

这样每个进程均可被看作是一个在函数域B上的函数F。其中B定义了进程一开始准备执行的事件集合；对B中的每个x，F(x)定义了进程执行第一个事件之后的未来行为。根据上述观点，每个CSP的进程可以通过适当的函数式程序设计语言中的函数来表达，譬如LISP语言。进程字母表中的每个事件可表述为一个LISP里的原子，譬如"COIN, "TOFFEE。一个进程是一个可以把这些院长事件作为输入参数的函数。如果这个符号

---

[48] 可以简单证明如下：通过联立递归方程的定义，每个X_i有唯一解，并为$\mu X \bullet F(i, X)$; 同理，每个Y_i有唯一解，并为$\mu Y \bullet F(i, Y)$。因为两个方程组一模一样，而且方程的解是唯一的，因此，$X = Y$。

[49] 本节关于Lisp实现的讨论就是按照STOP进程，前缀，选择和递归进程来逐步介绍的。

[50] 因为对于一个递归进程，非方程的μ的形式化描述是$\mu X \bullet F(X)$。在这里，我们把F可以表达为一种通用多项选择$(x:B \rightarrow F(x))$的符号记法代入，于是得到，$\mu X \bullet (x:B \rightarrow F(x,X))$。要注意的是大写的X是代表递归进程的符号(名字)；小写的x是代表了进程响应的各种事件。

[51] 对于是卫式递归的通用进程的方程式，方程式右侧显然$(x:B \rightarrow F(x, X))$。根据L2A，其意思就是把递归方程的唯一解 $\mu X \bullet (x:B \rightarrow F(x,X))$，可以代入到方程式里替换所有出现该递归进程名(符号)的地方，于是方程式的右边得到：$(x:B \rightarrow F(x, \mu X \bullet (x:B \rightarrow F(x,X))))$。例如，对于一个自动售货机VMCT(1.1.3 X3)，$X=\mu X \bullet (coin \rightarrow (toffee \rightarrow X \mid choc \rightarrow X))$，我们可以理解$(coin \rightarrow (toffee \rightarrow X \mid choc \rightarrow X))$为这里的F(x, X)。显然，进程可以简单的迭代展开一次变成：
$(coin \rightarrow (toffee \rightarrow ((coin \rightarrow (toffee \rightarrow X \mid choc \rightarrow X))) \mid choc \rightarrow ((coin \rightarrow (toffee \rightarrow X \mid choc \rightarrow X)))))$。理论上，可以无限制的展开。

不是该进程的第一个(批)可能执行的事件，这个函数则给出一个特殊符号“BLEEP”作为结果。符号“BLEEP”的用途仅在于此。例如，既然STOP从不执行任何事件，“BLEEP”就是它能给出的唯一的结果，因此它被定义为

```
STOP = λx • "BLEEP
```

如果自变量的值是该进程的可能事件，这个LISP函数则给出另一函数作为结果，代表进程的后续行为。于是，(coin→STOP)就可用如下LISP函数表达

```
λx • if x = "COIN then
       STOP
  else
       "BLEEP
```

这个例子利用了LISP的一个便利条件，即可以返回一个函数(如STOP)作为函数结果。LISP的另一个便利之处是，它允许将一个函数做为一个自变量传递给另一个函数，我们利用这一点来表达通用的前缀操作(c→P)

```
prefix(c, P) = λx • if x =c then
                          P
                   else "BLEEP
```

一个LISP函数如果要表示一个一般的二元选择算子(c→P | d→Q)，需要四个参数：

```
choice2(c, P, d, Q) = λx • if x= c then
                                P
                          else if x= d then
                                       Q
                                else
                                       "BLEEP
```

我们可以利用Lisp的LABEL特性表达CSP的递归进程。例如，简单自动售货机(μX • coin→choc→X)可以表示为

```
LABEL X • prefix( "COIN, prefix( "CHOC, X))
```

LABEL也可以用来表示联立递归式。如CT(1.1.4节X2)就可被看作是由自然数到进程的一个函数，当然进程本身也是函数，但先不考虑这一点。这样，CT就可以定义为

```
CT = LABEL X • (λn • if n=0 then
                        choice2("AROUND,X(O),"UP,X(1))
                        else
                              choice2("UP, X(n+1), "DOWN, X(n-1))
```

CT(0)是以地面为起点的进程。整个进程组从CT(0)开始。

如果P是表示一个进程的函数，且A为包含该进程字母表中符号的表，则LISP函数

menu(A,P)定义为给出可做为P的第一个事件的全部符号。

```
menu(A, P) = if A=NIL then
                    NIL
                    else if P(car(A)) = "BLEEP then
                                menu (crd(A)
                        else
                                cons(car(A), menu(cdr(A), P))
```

假设x在menu(A, P)中，且P(x)不是"BLEEP，那么P(x)是一个函数，它定义了P执行完x事件后的未来行为。由此推出，如果y在menu(A, P(x))中，则P(x)(y)定义的是在x和y都发生之后，进程的后续行为。这条规律为我们探讨进程的行为提示了一个很有用的方法。编写一段程序，先将menu(A,P)的值输出到屏幕上，然后再由键盘输入一个符号，观察结果。如果这个输入的符号不在表menu(A,P)中，则就会发出清晰的响声，但不接受它。如果它在这个表中，就会被接受，然后用键盘输入的符号产生的结果，例如，P(x)来取代P，并一直重复下去。当结束这一过程时键入符号"END。于是，如果k是由键盘输入的符号序列，则以下函数给出相关的输出序列

```
interact(A, P, k) =
        cons(menu(A, P), if car(k) = "END then
                NIL
           else if P(car(k))= "BLEEP then
                   cons( "BLEEP, interact (A, P, cdr(k) ) )
                   else
                           interact (A, P(car(k), cdr(k)))
```

以上定义LISP函数时所用的记号法很不正规，还需要把它们翻译成具体的LISP的S——表达式的形式。例如在LISPkit中，前缀函数可定义为

```
(prefix
        Lambda
        (a  p)
        (lambda (x)(if (eq x a) p (quote BLEEP))))
```

幸运的是，我们使用的只是纯函数式LISP语言的一个很小的子集，因此，在各种类型机器上用各种LISP变种翻译并运行进程时，都不会有什么困难。

如果有好几种LISP版本都能用，我们应该选择带有变量静态约束的那种LISP，使用它时会方便些。而使用惰式求值的LISP还要更方便些，因为这种LISP允许对递归方程直接编码，不需用LABEL特性，因此有

```
VMS = prefix( "COIN, prefix( "CHOC, VMS))
```

如果输入和输出是由惰式求值的LISP实施的，则在调用函数interact时以键盘作为它的第三个参数；进程P的菜单则是它的第一个输出。使用者通过在不断输出菜单中选择并输入符号，就可以交互式地研究进程P的行为。
在其它几种LISP版本中，函数interact应该改写，使用显示输入和输出来达到同样的效果。总之，这样一来，我们就有可能观察计算机执行一LISP函数表示的任一进程了。从这个意义上来说，这种LISP函数就可以被看做是其相应进程的实现。更进一步地说，象prefix这样的LISP函数，它们作用于表示进程的函数上，就可以被看作是相应的进程算子的一种实现。

1.5 迹(Traces)

一个进程行为的迹是进程事件被触发执行的一个有限序列，这个符号序列记录该进程到某一时刻为止执行的各个事件。设想有位观察员带着笔记本，观察这个进程，每一个事件发生时他就将该事件名记下来。我们完全可以忽略两个事件同时发生的可能性；因为即使真有两个事件同时发生了，观察员做记录时也总要有先有后，而且他记录的先后次序也不是重要的。
一个迹表示为事件符号的一个序列，中间由逗号断开，并且括在一对角括号内。

<x, y>由两个事件组成，y跟在x后发生。
<x>只包含一个事件x的序列。
< >不包含任何事件的空序列。

例子

X1 简单自动售货机VMS(1.1.2节中X2)若刚好完成了为两位顾客服务，这时它的迹为

(coin, choc, coin, choc)
□

X2 同一个售货机，在第二位顾客的巧克力被送出之前的迹为

(coin, choc, coin)

不论是进程还是观察员都不具备一个完整的交易这种概念。顾客的饥不可耐和机器满足顾客要求的责任感都不在进程的字母表内，所以也就不能被观察到或记录下来。□

X3 在进程开始执行任何事件之前，观察员的笔记本是空白的。这种情况表示为空迹

< >

空迹是每个进程可能的最短的迹。 □

X4 复杂自动售货机VMC(1.1.3节中X4)有以下七个长度不超过2的迹

< >
< in2p> <inlp>
<in2p, large> <in2p, small> <inlp, inlp> <inlp, small>

对一台给定的机器来说，那四个长度为2的迹中只有一个能真正发生。至于选择哪一个，就要有第一位使用这台售货机的顾客来决定了[52]。 □

X5 如果第一位顾客不理睬VMC机器的注意事项[53]，那么VMC的迹可以是

<inlp, inlp, inlp >

这个迹并不记录机器的损坏。机器的损坏是通过下述事实表现出来的，即在这台机器所有可能的迹中，不存在一个迹是这个迹的延伸，也就是说，不会存在事件x，能使

<inlp, inlp, inlp, x>

成为VMC可能的迹。这时，顾客可能干着急没办法，观察员也是一筹莫展，不会再有事件发生了，不会再有符号能记在小本上了。至于顾客和售货机的结局如何，我们就不得而知了，因为这不在我们选择的字母表内。

1.6 迹的运算(Operations on traces)

在对进程行为进行记录，描述以及正确理解过程中，迹扮演着一个很关键的角色。在这一节里，我们要探讨迹的一般性质以及迹的运算。我们要用到以下约定

s, t, u 表示迹
S, T, U 表示迹的集合
f, g h 表示函数

1.6.1 连接(Catenation)

迄今最重要的迹运算是迹与迹之间的连接，即把一对操作数s和t按照s, t的次序简单的拼到一起，构成一个迹，记为

s ^ t

例如

<coin, choc >^<coin, toffee> = <coin, choc, coin, toffee>
<inlp> ^<inlp> = <inlp, inlp>
<inlp, inlp> ^ < > = <inlp, inlp>

[52] VMC的递归行为定义是：VMC = (in2p→(large→VMC | small→outlp→VMC) | inlp→(small→VMC | inlp→(large→VMC | inlp→STOP)))。支持多种选择。但一旦选择了，执行轨迹就不一样了。
[53] VMC机器的注意事项："不要接连投入三个便士。"

连接的最重要的性质是，连接是满足结合律的(associative)，且其单位元素为< >。
L1 s^ < > = < > ^ s = s

L2 s ^ (t ^ u) = (s ^ t) ^ u

以下法则是很显然的，而且非常有用

L3 s ^ t = s ^ u ≡ t = u

L4 s ^ t = u ^ t ≡ s = u

L5 s ^ t = < > ≡ s = < > ∧ t = < >

假设f是一个映射迹到迹之间的函数。如果它对空迹的映射结果仍然为空迹，即

f(< >) = < >

则函数f被称作是严格的(strict)。
如果f(s ^ t)=f(s) ^ f(t)
则说函数f是满足分配律的(distributive)。所有可分配的函数都是严格的[54]。
设n为自然数，我们定义 t^n为t的n次自相连接。对n进行归纳，即有结果

L6 t^0 = < >

L7 t^{n+1} = t ^ t^n

上述两项很有用的法则，实际上就是定义本身；以下是另外两个可由它们推导证明的法则

L8 t^{n+1}=t^n ^ t

L9 $(s \hat{} t)^{n+1}$ = s ^ $(t \hat{} s)^n$ ^ t

1.6.2 局限(Restriction)

表达式(t A)表示迹t局限于集合A中的符号，即把t中所有不属于A的符号去掉后留下的迹。例如

<around, up, down, around> {up, down} = <up, down>

局限运算是可分配的，因此也是严格的。相关一些法则

L1 <> A = <>

L2 (s ^ t) A = (s A) ^ (t A)

[54] f(<>) = f(<> ^ <>)。因为f满足分配律，所以，f(<>) = f(<>) ^ f(<>)。因此，f(<>) = <>。因此是严格的。

对于单元素(Singleton)序列，显然有

L3 <x> A = <x> 如果x ∈ A

L4 <y> A = <> 如果y ∉ A

一个满足分配律的函数，在定义了它对单元素序列的作用效果后，这个函数整个的行为也就唯一地确定了。因为当作用于更长的序列时，我们总可以使它分别作用于序列的每个元素。然后把各项结果连接起来，然后可得到我们所要的结果。例如，如果x ≠ y

<x, y, x> { x }
= (<x> ^ <y>^ <x>) { x }
= (<x> { x }) ^ (<y> { x }) ^ (<x> { x }) [利用法则L2][55]
= <x> ^ <> ^ <x> [利用法则L3和L4]
= <x, x>

以下是关于局限与集合运算间的关系的法则。一个迹局限于空符号集的结果是一个空迹，连续受限于两个集合的迹的结果就跟这个迹受限于这两个集合的交集的结果相同。对迹s的长度进行归纳，就能严格证明这几项法则

L5 S {} = <>

L6 (s A) B = s (A ∩ B)

1.6.3 首部与尾部(Head and tail)

假设s为一非空序列，取它的第一个符号为s_0，去掉s_0后结果记为s´。如

$<x, y, x>_0 = x$

<x, y, x>= <y, x>

这两个运算对空列无定义。

L1 $(<x>s)_0 = x$

L2 (<x>s)´ = s

L3 $s = (<s_0> \wedge s')$ 如果s ≠ < >

法则L4给出证明两个迹相等的简便的方法。

[55] 通过分配律作用在每个元素上。

L4 $s = t \equiv (s = t = <> \vee (s_0 = t_0 \wedge s' = t'))$[56]

1.6.4 星号(Star)

我们定义集合A^*是由符号表A符号构成的所有的有穷迹(包括< >)的集合。因此，当局限于字母表A时，这些迹都保持不变。通过这个性质我们可引出一个简单定义

$$A^* = \{s \mid s\ A = s\}$$

以下法则是这个定义的推论。

L1 $< > \in A^*$

L2 $< x > \in A^* \equiv x \in A$

L3 $(s \hat{} t) \in A^* \equiv s \in A^* \wedge t \in A^*$

这些法则很有力，足以确定一个迹是否是A^*的一个元素。例如，如果$x \in A$且$y \notin A$，则有

$< x, y > \in A^* \equiv (< x > \hat{} < y >) \in A^*$
$\equiv (< x > \in A^*) \wedge (< y > \in A^*)$　　由L3
$\equiv$ true $\wedge$ false　　由L2

下一法则可作为A^*的一个递归定义。

L4 $A^* = \{t \mid t = <> \vee (t_0 \in A \wedge t' \in A^*)\}$

1.6.5 次序(Ordering)

假设s是序列t的一个初始子序列，则我们总能找到s的某个扩展序列，使$s \hat{} u = t$。因此，我们定义一种次序关系

$$s \leq t = (\exists u \bullet s \hat{} u = t)$$

我们定义这种关系为s是t的一个前缀。例如

$$< x, y> \leq <x, y, x, w>$$

$$< x, y> \leq <z, y, x> \equiv x = z$$

---

[56] 证明很简单，如果s和t都是空串，显然s = t；或者，如果s的第一个符号和后续的符号都与t的第一个符号和后续的符号相同，显然，根据L3，s = t。

如法则L1至L4所述，关系≤是个偏序关系[57]，其最小元为< >。

L1 $< > \leq s$ 最小元

L2 $s \leq s$ 自反性

L3 $s \leq t \wedge t \leq s \Rightarrow (s = t)$ 反对称性

L4 $(s \leq t \wedge t \leq u) \Rightarrow (s \leq u)$ 传递性

下一法则与L1并用，给出计算$s \leq t$是否成立的方法。

L5 $(< x > \hat{} s) \leq t \equiv t \neq < > \wedge x = t_0 \wedge s \leq t'$[58]

对一给定序列，它的所有的前缀是全序的(Total Ordering)[59]。

L6 $s \leq u \wedge t \leq u \Rightarrow s \leq t \vee t \leq s$[60]

假设s为t的一个子序列(不一定是初始子序列)，我们就说s in t；记为

L7 $s \text{ in } t = (\exists u,v \bullet t = (u \hat{} s \hat{} v))$

这个关系也是个偏序关系，满足上述L1至L4[61]。它还满足下述法则

L8 $(< x > \hat{} s) \text{ in } t \equiv t \neq < > \wedge ((t_0 = x \wedge s \leq t') \vee (< x > \hat{} s) \text{ in } t')$

我们定义一个函数单调的性质。定义为如果迹到迹的函数f能够保持前缀的次序关系≤，也就是说

$f(s) \leq f(t)$ 如果有$s \leq t$[62]

所有满足分配律的函数都是单调函数。例如

L9 $s \leq t \Rightarrow (s \ A) \leq (t \ A)$

---

[57] 偏序关系的含义是，给定字母表A组成的任意两个序列，不是任何两个序列都必须需要保证存在≤这种逻辑关系。可以是谁也不是谁的前缀。例如，<x y z m n>与<z x n m>，不存在这种前缀关系。

[58] 只需确保<x>和s的连接形成的序列中，x是t的第一个字符，s是t除了第一个字符之后的子串。

[59] 任何两个前缀，彼此之间一定存在前缀关系。不存在孤立的两个前缀，无法定义其关系。

[60] 要么s是比t还靠前的前缀，或者，t更靠前。

[61] L1 $< > \text{ in } s$最小元 L2 $s \text{ in } s$自反性 L3 $s \text{ in } t \wedge t \text{ in } s \rightarrow (s = t)$反对称性 L4 $(s \text{ in } t \wedge t \text{ in } u) \rightarrow (s \text{ in } u)$传递性

[62] 可以这样理解，通过函数f的变换，前缀的次序关系不会丢掉。

一个二元函数对它的两个自变量中的一个可能是单调的(这时令另一个自变量不变)。例如，连接对第二个自变量(但不能是第一个)是单调的。

L10 $t \leq u \Rightarrow (s \wedge t) \leq (s \wedge u)$

对所有自变量均为单调的函数称为单调函数。

1.6.6 长度(Length)

迹t的长度记做#t。例如

$$\#< x,\ y,\ x > = 3$$

定义长度运算#的法则如下

L1 $\#< > = 0$

L2 $\#<x> = 1$

L3 $\#(s \wedge t) = (\#s) + (\#t)$

A中符号在t中出现的个数为# A

L4 $\#(t\ \ (A \cup B)) = \#(t\ (A) + \#(t\ B) - \#(t\ (A \cap B))$

L5 $s \leq t \Rightarrow \#s \leq \#t$

L6 $\#(t^n) = n \times (\#t)$

符号x在迹中s中出现的次数定义为

$$s\downarrow X = \#(s\ \ (x\})$$

1.7 迹的实施(Implementation of traces)

为了在计算机内表示迹并实现对它的运算，我们需要一种高级的表处理语言。幸运的是，LISP能满足我们的需要。可以用代表事件的原子组成的Lisp表，能很容易地把迹表示出来。

< > = NIL

<coin> = (cons("COIN, NIL))

<coin, choc> = "(COIN CHOC)

其含义为cons("COIN, cons("CHOC, NIL))

迹的运算可轻易的通过Lisp表的函数的功能来实现。例如，一个非空表的首部和尾部可以由原函数car和cdr给出

$$t_0 = car(t)$$

t' = cdr(t)

< x> ^ s = cons(x, s)

我们用大家熟悉的αPpend函数来实现一般的连接运算。αPpend函数是由下述递归式给出定义的

s ^ t = αPpend(s, t)

其中，

αPpend(s, t) =

if s = NIL then t

else
cons(car(s), αPpend(cdr(s), t))

这个定义的正确性由下述法则为依据

< >^ t=t

s ^ t = <s_0> ^ <s ^ t> 其中s≠< >

LISP的αPpend函数的终止可以通过这样的事实保证，即每次递归调用时用作第一个自变量的表都要比它前一级递归调用时的短[63]。自变量的类似特性也保证以下定义的其它运算实施的正确性。
为实施局限(Restricted)运算，我们用有限集合B的元素的表代表B本身，检验(x ∈ B)可通过调用函数来实现

ismember(x, B) =

if B = NIL then
false
else if x = car (B) then
true
else ismember(x, cdr(B))

(s B)可由下列函数实现
restrict(s,B) =

if s= NIL then
NIL
else if ismember(car(s), B) then
cons (car(s), restrict(cdr(s), B))
else
restrict (cdr(s), B)

[63] 最后会遇到递归方程的控制条件：if s = NIL then t 。

检验(s≤t)可由给出结果是true或false的函数来实现；根据1.6.5节的L1和L5这个函数可定义为

```
isprefix(s, t) = if s = NIL then
        true
        else if t = NIL then
        false
        else
              car(s) = car (t) and
        isprefix(cdr(s), cdr(t))
```

1.8 进程的迹(Trace of a process)

在1.6节中，我们定义进程的迹是进程到某一个时刻为止的行为的顺序记录。进程开始前，谁也不知道进程的哪个可能的迹将会被记录下来：这个选择是由不受进程约束的外部环境因素来决定的。然而，进程P的所有可能的迹的全集是能事先知道的，我们定义函数traces(P)，用这个函数来表示产生一个进程可能的行为(迹的集合。

例子

X1 进程STOP的行为只有一个迹，即< >。记录这个进程的观察员的笔记本永远是空白，即

traces(STOP) = {< >} □

X2 在接受一枚硬币后就毁坏的机器只能有两个迹

traces(coin→STOP) = {< >, <coin>} □

X3 只是滴答地走着的钟，它的迹是

traces(μX • tick→X) = {< >, <tick>, <tick, tick>,...}
= {tick}*

就象很多非常有趣的进程一样，尽管这个进程的每个迹都是有限的，其迹的集合却是无穷的。 □

X4 对一台简单自动售货机，其行为的迹为[64]

traces(μX • coin→ choc→X) ={ s | ∃ n • s ≤ <coin, choc>^{n} □

1.8.1 法则(Laws)

在这一节里，我们讲述如何使用目前已经阐述的符号系统来度量一个进程迹的集合。如前所述，进程STOP只有一个迹，并为

[64] 自动售货机比理解什么是一次交易完成。所以迹只是一种单纯的记录，可以包括<coin, choc, coin>。我们通过前缀算子 ≤来精确定义。s ≤ <coin, choc>^{n}

L1 traces(STOP) = {t | t = < >}= {< >}

空迹可以是(c→P)进程的一个迹，因为每个进程，到它开始实行第一个动作的时刻，< >都是它的行为的一个迹。(c→P)的每个非空迹都是以c开始，这个迹的尾部必须是P的一个可能的迹，故有

L2 traces(c→P) ={t | t = < > ∨ (t_0 = c ∧ t' ∈ traces(P))}
= {< >} ∪ {< c> ^ t | t ∈ traces(P)}[65]

对初始事件有多种选择的进程的行为的迹，一定是下面几种可能的迹之一

L3 traces(c→P | d→Q) =
{ t | t = < > ∨ (t_0 = c ∧ t' ∈ traces(P)) ∨(t_0 =d ∧ t' ∈ traces(Q)) }[66]

把上面三个法则(STOP, 选择，多项选择)归结为一个通用选择算子，可以归纳为

L4 traces(x : B→P(x)) ={ t | t = < > ∨ (t_0 ∈ B ∧ t' ∈ traces(P(t_0))) }[67]

对递归定义的进程来说，要定义其迹的集合要略微困难一些。一个递归定义的进程是下面方程的唯一解。

X=F(X)

首先，我们用归纳法定义函数F的迭代式

$F^0(X) = X$
$F^{n+1}(X) = F(F^n(X))$[68]
$= F^n(F(X))$
=F(...(F(F(X)))...)

然后，假设F是卫式，我们可定义

L5 traces(μX : A • F(X)) = $\bigcup_{n\geq 0}$ traces($F^n(STOP_A)$)

例子

X1 回忆一下进程RUN_A在1.1.3节X8中被定义为

μX : A • F(X)

---

[65] 要么是最开始任何事件都还没有触发是的空迹状态，或者是以事件c为最开始的事件，然后跟着的是属于进程P的迹。

[66] (t_0 = c ∧ t' ∈ traces(P)) 代表了选择一"c→P"的进程走向。(t_0 =d ∧ t' ∈ traces(Q))代表了选择一"d→P"的进程走向。两个选择必居其一。

[67] 逻辑其实很简单，但数学表达的很精巧。进程的迹是根据最开始的事件选择的不同，响应的映射到不同的行为函数上。另外，t = t_0^t'，形成该进程完整的迹。

[68] 这里需要明确的是n不为0。$F^1(X)$不能递归为$F(F^0(X))$。进程一旦开始接受事件触发就进入正常的模式了。

这里$F(X) = (x: A \rightarrow X)$，我们要证明

$traces(RUNA) = A^*$

证明 $A^* = \bigcup \{s \mid s \in A^* \wedge (\#s \leq n)\}$ [69]

我们通过归纳发来证明。

1. $traces(STOP_A) = \{< >\}$
$= \{s \mid s \in A^* \wedge \#s \leq 0\}$ [70]

2. $traces(F^{n+1}(STOP_A))$
$= traces(x : A \rightarrow F^n(STOP_A))$ 根据F和F^{n+1}的定义
$= \{t \mid t = < > \vee (t_0 \in A \wedge t' \in traces(F^n(STOP_A)))\}$ 根据L4
$= \{t \mid t = < > \vee (t_0 \in A \wedge (t' \in A^* \wedge \#t' \leq n))\}$ 根据归纳假设[71]
$= \{t \mid t = < > \vee (t_0 \in A \wedge t' \in A^*)) \wedge \#t \leq n+1\}$ #的特性
$= \{t \mid t \in A^* \wedge \#t \leq n+1\}$ 1.6.4节L4

□

X2 我们要证明1.8节中的X4，即证

$traces(VMS) = U_{n \geq 0}\{s \mid s \leq <coin, choc>^n\}$

证明 我们假设迭代为n的时候成立。

$traces(F^n(VMS)) = \{t \mid t \leq <coin, choc>^n\}$

这里$F(X) = (coin \rightarrow choc \rightarrow X)$

(1) $traces(STOP) = \{< >\} = \{s \mid s \leq <coin, choc >^0\}$ 1.6.1节L6
(2) $traces(coin \rightarrow choc \rightarrow F^n(STOP))$
$= \{< >, <coin>\} \cup \{<coin, choc> \hat{} t \mid t \in traces\{F^n(STOP))\}$ L2
$= \{< >, <coin>\} \cup \{<coin, choc> \hat{} t \mid t \leq <coin, choc>^n\}$ 归纳假设
$= \{s \mid s = < > \vee s = < coin > \vee$
$\exists t \bullet s = <coin, choc> \hat{} t \wedge t \leq t < coin, choc>^n\}$
$= \{s \mid s \leq <coin, choc>^{n+1}\}$

再由L5即得结论。 □

在1.5节中，我们阐述了迹是一个符号序列，这个序列记录了进程P到某一时刻为止已执行过的事件。其中< >是任何一个进程在开始执行第一个事件之前的迹[72]。另外，如果$(s \hat{} t)$是进程到某一时刻的迹，则s一定是这个进程到前面某一时刻为止的一个迹。最后，发生的每一事件一定在进程的字母表内。这三个事实我们用法则正式给出。

---

[69]这个例子的证明与Tony Hoare 1988年的版本有一些文字上的变化。

[70]这里面蕴含了，当n=0时，即一个进程还不接受任何event的时候，其行为是STOP进程。

[71]归纳假设 $(t_0 \in A \wedge t' \in traces(F^n(STOP_A))) = (t_0 \in A \wedge (t' \in A^* \wedge \#t' \leq n))$

[72]有点类似宇宙大爆炸之前的时间为0的状态一切都是静止的。从另外一个角度来考察，CSP应该定义一个INIT的进程，其行为等价于STOP和后面定义的SKIP。都表达一个进程什么也不响应的一种状态。但如果在这里CSP用STOP来描述进程最开始的迭代似乎有点不妥当。

L6 $< > \in \text{traces}(P)$

L7 $s^\wedge t \in \text{traces}(P) \rightarrow s \in \text{traces}(P)$

L8 $\text{traces}(P) \subseteq (\alpha P)^*$

进程的迹和进程行为的树状示意图是紧密相关的，对树上的任何节点来说，进程走到该节点时刻行为的迹就是它从树根到该节点沿途遇到的标记的序列。譬如，在图1.1中所示的VMC树形图上，与由树根到那个黑节点的路径对应的迹就是

<in2p, small, outlp >

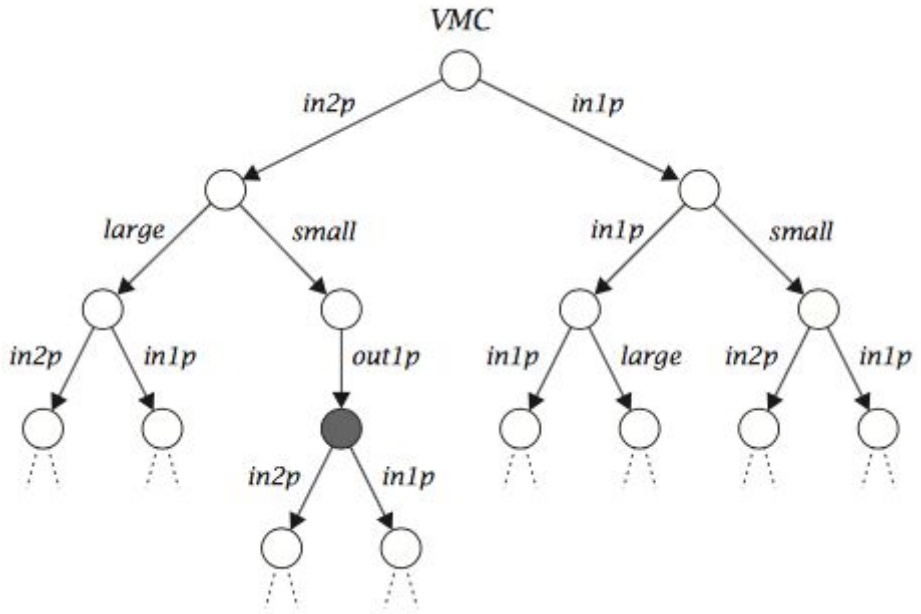


图1.1

很显然，行为树上任意一条路径的所有子路径仍然是这棵树上的路径上述L7形式化的说明了这一点。空迹定义了由树根到其本身的空路径，即L6[73]。进程所有的迹就是由树根引到节点的所有路径的集合。

反之，因为从每个节点引出的分支都用不同事件做了标记，所以进程的每个迹也就唯一地确定了由树根到某一具体节点的一条路径。这样，任何一个满足L6和L7的迹的集合组成了树形图的方便的数学表示式，在该类树形图中由一个节点不会引出两个标记相同的分支。

1.8.2 实现(Implementation)

假设进程已经通过LISP函数P来实现实了，这里s为此进程的一个迹。则通过下面的函数可以来验证是否s是P的一个迹

[73] 在定义递归函数的迹时，L5的$\text{traces}(\mu X : A \bullet F(X)) = \cup_{n\geq 0} \text{traces}(F^n(STOP_A))$通过STOP来表达最开始时的空迹，容易让人产生歧义。

```
istrace(s, P) = if s = NIL then
          true
else if P(car(s)) = "BLEEP then
           false
   else
            istrace(cdr(s),P(car(s)))
```
[74]

由于s是有限的，所以这个递归在处理完进程P行为的有限步骤之后，会终止。正是因为我们不对进程作无穷的探究，我们才能把进程定义为一个无穷的对象，也就是说，进程是个函数，其结果仍为一个函数，这个结果函数的结果还是个函数……。

1.8.3 后继(After)

如果s ∈ traces(P)，则

P/s　　(P中s的后继)

是一个进程，这个进程表示P在执行完迹s所记录的所有动作后的行为。如果s不是P的一个迹，则(P/s)无意义。

例子

X1 (VMS /<coin>) = (choc →VMS) □

X2 (VMS / <coin, choc>) = VMS □

X3 (VMC /<(inlp >[3]) =STOP □

X4 为避免有VMCRED (1.1.3 X5, X6)可能引起的损失，售货机的主人决定自己吃第一块巧克力，则有

(VMCRED /<choc>) = VMS2 □

在进程P的树形图上(如图1.1)，(P/s)表示一棵完整的子树，子树的根就是用s所标记的路径的终点。因此，图1.1中黑色节点以下的子树就记为

VMC / <in2p, small, outlp>

以下几项法则将说明算子/的含义。什么事都还没做的进程不会有任何变化，因此

L1 P/< >=P

---

[74] 此处与Tony Hare 1988年的版本有点小区别。在1988年的书里直接用了CSP的符号表示一个迹的首元素和尾元素。在LISP里，用Car和Cdr来分别表示。The car of a list is, quite simply, the first item in the list. Thus the car of the list (rose violet daisy buttercup) is rose. The cdr of a list is the rest of the list, that is, the cdr function returns the part of the list that follows the first item.

执行完s^t后的迹，与跟P/s执行完再过滤t的行为完全一样，即

L2 $P/(s^\wedge t) = (P/s)/t$

执行完单个事件c后，进程的行为就要由这个初始选择来确定了，即

L3 $(x:B \rightarrow P(x)) / \langle c \rangle = P(c)$ 假设$c \in B$

下面的这条推论说明/ <c>是前缀算子c→P的逆运算。

L3A $(c \rightarrow P) / \langle c \rangle = P$

(P/s)的迹定义为

L4 $traces(P / s) = \{t \mid s^\wedge t \in traces(P)\}$ 假设 $s \in traces(P)$

要证明进程P永远不停止动作，只要能对所有$s \in traces(P)$
证明P/s≠STOP就够了[75]。

我们想讨论的进程的另一个性质就是循环性：如果在任何情况下，进程P都能回到它的z最初始状态，即

$$\forall\ s : traces(P) \bullet \exists\ t \bullet (P / (s^\wedge t) = P)$$

我们就称这个进程P是循环的。STOP明显具备循环属性；但任何其它循环进程都是永远不停止动作的。

例子

X5 以下进程是循环的(1.1.3节中X8, 1.1.2节中X2,1.1.3节中X3,1.1.4节中X2)，即

$$RUN_A, VMS, (choc \rightarrow VMS), VMCT, CT_7 \qquad \square$$

X6 以下进程不是循环的，因为无法使它们回到初始状态(1.1.2节中X2, 1.1.3节中的X3, 1.1.3节中X2)，即

$$(coin \rightarrow VMS), (choc \rightarrow VMCT), (around \rightarrow CT_7)$$

譬如，在(choc→VMCT)的初始状态下，只能拿到巧克力，但是以后，choc和toffee总是同时可供选择；因此这些后续状态就不会与初始状态相同了[76]。 □

注意：如果在递归定义的进程里使用/，会使其可能失去卫式特性[77]，并由此导致递归方程有多解。例如

---

[75] 永远抵达不到STOP状态，例如递归进程。
[76] $VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$
[77] 只有Guarded的卫式方程才具备唯一解。

$X = (a \to (X \,/\, < a >))$

不是卫式，任何形式为

$a \to P$

的进程都是它的解，其中P可为任一进程。

证明 $(a \to ((a \to P) \,/\, <a>)) = (a \to P)$[78] 根据L3A

由于这个原因，我们在递归进程的定义中不使用算子/。

1.9 迹的其它运算(More operations on traces)

这节主要讲述一下迹的其它运算。这部分目前可以跳过不看，在以后章节中如果用到这些运算时，我们会注明相关出处。

1.9.1 符号变换(Change of symbol)

假设函数f为集合A到集合B的符号映射。由f我们还可导出一个由A*至B*的新函数f^*，即用f分别作用于A*中符号序列的每个元素，从而将A*的符号序列对应为B*中的一个序列。例如，设double为一个函数，使其整数自变量增为两倍，则

$\text{double}^*(< 1, 5, 3, 1 >) = < 2, 10, 6, 2 >$[79]

标星号的函数很显然是服从分配律的，因而也是严格的，故有

L1 $f^*(< >) = < >$

L2 $f^*(< x >) = < f(x) >$

L3 $f^*(s \hat{} t) = f^*(s) \hat{} f^*(t)$

其他明显的法则

L4 $f^*(s) = < f(s_0) >$ 若 $s \neq < >$

L5 $\#f^*(s) = \#s$

下述一个法则貌似很“显然”，却其实并不成立

$f^*(s \quad A) = f^*(s) \quad f(A)$

这里 $f(A) = \{ f(x) \mid x \in A \}$。

---

[78] $X = (a \to (X \,/\, < a >))$, 代入$a \to P$得到$a \to P = (a \to (a \to P \,/\, < a >))$。通过L3A, 得到$a \to P = a \to P$。因此P可以为任何进程。不存在唯一解。

[79] 把一个迹的符号序列都相应的换掉。

我们举一个最简单的反例，有这样

一个函数f，满足

$f(b) = f(c) = c$　这里$b \neq c$，于是有

$f^*(\langle b \rangle \ \{c\})$	由于$b \neq c$
$= f^*(\langle\rangle)$	
$= \langle\rangle$	L1
$\neq \langle c \rangle$	
$= \langle c \rangle \ \{c\}$	
$= f^*(\langle c \rangle) \ f(\{c\})$[80]	由于$f(c) = c$

然而，当函数f是1-1对应(单射)函数时，这项法则是成立的，既有

L6 $f^*(s \ A) = f^*(s) \ f(A)$　当f为单射函数时

1.9.2 连接(Catenation)

设s为一序列，这个序列的每个元素本身还是一序列。则将所有元素按原序连接起来的结果记为^/s，例如

$$\hat{}/\langle \langle 1,3 \rangle, \langle\rangle, \langle 7 \rangle \rangle = \langle 1,3 \rangle \hat{} \langle\rangle \hat{} \langle 7 \rangle$$
$$= \langle 1, 3, 7 \rangle$$

这个算子满足分配律

L1 $\hat{}/\langle\rangle = \langle\rangle$

L2 $\hat{}/\langle s \rangle = s$

L3 $\hat{}/(s\hat{}t) = (\hat{}/s) \hat{} (\hat{}/t)$

1.9.3 穿插(Interleaving)

如果一序列s可被拆成一系列的子序列，并且这些字序列交替地为序列t和u的子序列，我们就说s是t和u的穿插。例如

$$s = \langle 1, 6, 3, 1, 5, 4, 2, 7 \rangle$$

就是有t和u穿插而构成，这里

---

[80] $f^*(\langle c \rangle) \ f(\{c\}) = f^*(\langle b \rangle) \ f(\{c\})$。因此$f^*(\langle b \rangle \ \{c\}) \neq f^*(\langle b \rangle) \ f(\{c\})$。这个反证可以这样理解: 在有过滤算子的场景下，函数在过滤后和过滤前应用在一个迹上的结果可能有重要区别。因为函数的使用可以使得一个过滤算子失去效果。

t=<1, 6, 5, 2, 7> 和 u = <3, 1, 4>

穿插算子的递归定义可由下述法则给出

L1 < > interleaves (t, u) ≡ (t = < > ∧ u=< >)

L2 s interleaves (t, u) ≡ s interleaves (u, t)

L3 ((x)^s) interleaves (t, u) ≡ (t ≠ <> ∧ t_0=x ∧ s interleaves (t', u)) ∨
(u ≠ < > ∧ u_0 = x ∧ s interleaves (t, u'))

1.9.4 下标(Subscription)

如果0≤i＜#s，则我们用惯用的记号s[i]，表示序列s的第i个元素，其定义见L1。

L1 s[0] = s_0 ∧ s[i+1] = s´[i] 若s≠< >

L2 $(f^*(s))[i]=f(s[i])$ 若i＜#s

1.9.5 逆置(Reversal)

设s为一序列，则 $\bar{s}$ 为s的逆置，即将s的元素顺序颠倒过来。例如

$\overline{<3,\ 5,\ 37>}$ = <37, 5, 3>

逆置的完整定义由以下法则给出

L1 $\overline{<\ >}$ = < >

L2 $\overline{<x>}$ = <x>

L3 $\overline{<s\hat{}t>}$ = $\bar{s}\hat{}\bar{t}$

逆置还具备几个简单的代数性质，包括

L4 $\bar{\bar{s}}$ = s

逆置的其它性质我们留给读者去推导。还有一个很有用的事实是， 是序列s的最后一个元素，且一般有

L5 $\bar{s}$ [i] = s[#s - i +1]

1.9.6 挑选(Selection)

设s为一个二元序列，我们把s中所有第一个元素为x的元素对挑选出来，然后用各自的第二个元素取代这些元素对，所得

结果定义为s↓x。元素对的元素之间我们用一个黑点隔开。这样如果

s=<a，7，b，9，a，8，c.0 >

则 s↓a = <7,8 >
且 s↓d = < >

L1 < >↓x = < >

L2 (<y.z>^t)↓x = t↓x 若y≠x

L3 (<x.z>^t)↓x = < z> ^ (t↓x)

如果s不是二元序列，则s↓a表示a在s中出现的次数(如1.6.6节中定义)。

1.9.7 组合(Composition)

假设符号√表示进程的成功终止，则这个符号只能出现在迹的结尾[81]。设t为一个迹，它记录的是s已经终止后才开始的事件序列。s和t的组合记为(s;t)。如果√没有在s中出现，则t就无法开始，即

L1 s ; t = s 若 ¬(< √> in s)

如果√在s的结尾出现了，就将它去掉，把t附加到s的剩余的部分，即

L2 s^<√ > ; t = s^t 若 ¬(< √> in s)

符号√有点类似胶水，把s和t粘到一起。没有这个胶水，t就粘不上去。如果符号√出现在一个迹的中间(是不正确的)，为了规则的完整性，我们规定符号√之后的序列都不参与组合，舍弃掉。即

L2A (s^⟨√⟩^u) ; t =s^t if ¬(⟨√⟩ in s)

这大家不太熟悉的组合算子具备几条常见的代数性质的。象连接一样，它是满足结合的。与连接不同的是，它不论对它的第一个变量还是第二个自变量都是单调的。而且组合算子对第一个自变量是严格的，并以<√>为其左单位元，即

L3 s ; (t ; u) = (s ; t) ; u

L4A s ≤t ⇒ ((u ; s)≤ (u ; t))

---

[81] 这个事件对应的是第5章顺序进程中介绍的“SKIP”进程，表示一个进程成功结束。

L4B $s \le t \Rightarrow ((u\ ;\ s) \le (t\ ;\ u))$

L5 $<>\ ;\ t=t$

L6 $<\surd>\ ;\ t=t$

如果√仅在迹的结尾出现，那么$<\surd>$也是迹的右单位元，既有

L7 $s\ ;\ <\surd> = s$ 若 $\neg(<\surd> \text{ in } (s)^{\frown})$

1.10 规约(Specifications)

一般说来，某项产品的功能描述说明这个产品的预期的行为。这类说明是包含一些自由变量的谓词断言[82]，每个自由变量表示这个产品行为的某个可观察到的方面。例如，有这样一个电子放大器的描述，其电压输入范围为1伏，放大约为10倍，表示成谓词就是

$$AMPIO = (0 \le v \le 1 \Rightarrow |v' - 10 \times v| \le 1)$$

在描述中，谓词中的v理解为输入电压，v′为输出电压。在科学和工程中使用数学时，很基本的一点就是要理解变量的含义。
对进程而言，能观察到的和进程行为最直接有关的，就是到某一给定时刻为止发生的事件的迹。我们用特殊变量记号 tr 表示所描述的进程的一个任意迹，这和前一段中我们用记号v和v′记载观察到的电压的任意变化的方法是一样的。

例子

X1 自动售货机的主人不想亏本，他就规定送出巧克力的块数不能超过投入的硬币数

$$NOLOSS = (\#(tr\ \ \{choc\}) \le \#(tr\ \ \{coin\}))$$ □

以后我们用缩写形式

$$tr \downarrow c = \#(tr\ \ \{c\})$$

表示符号c在tr中出现的次数(见1.6.6节)

X2 顾客也希望自动售货机公平合理，投入多少硬币就能得到多少巧克力

$$FAIR1 = (\ (tr \downarrow coin) \le (tr \downarrow choc) + 1)$$ □

X3 简单自动售货机的生产就必须同时满足机器主人和顾客的要求

$$VMSPEC = NOLOSS \wedge FAIR1$$
$$= (0 \le (\ (tr \downarrow coin) - (tr \downarrow choc)) \le 1)$$ □

[82] Predicate的意思。本书里的都应该是一介逻辑的谓词演算。

X4 修改复杂自动售货机的描述，不允许它连续接受三枚硬币

VMCFIX = (¬ <inlp>^{3} in tr) □

X5 改进后的机器的描述为

MENDVMC = (tr ∈ traces(VMC) ∧ VMCFIX) □

X6 VMS2(1.1.3节X6)的描述

0≤((tr↓coin) — (tr ↓ choc)) ≤ 2 □

1.10.1 满足(Satisfaction)

设P为满足描述S的产品，我们说P满足S，缩写为

P sat S

其含义是，可观察到的P的行为都被S所描述的行为刻画了；换句话说，只要S中的变量取的是由观察P得到的值时，S就为真，或更形式地记为∀ tr • tr∈traces(P)⇒S。譬如，下列表格给出对放大器特性的观察结果

	1	2	3	4	5
V	0	.5	.5	2	.1
V'	0	5	4	1	3

所有这些观察结果除了最后一栏外都满足AMP10。第二和第三栏说明放大器的输出并不是完全由输入决定的。第四栏表明，如果输入电压超出了指定范围，输出电压可以是任意值，但这不算是违背描述。(在这个简单的例子里，我们排除过高电压会使产品击穿的可能性。)
下面的一些法则给出"满足"关系的几个最一般的性质。True不对产品施加任何限制，因此所有产品都满足这一要求；甚至是坏掉的产品也能满足这一无意义的、无需求的描述。即

L1 P sat true

如果一产品同时满足两种不同的规约，则也满足它们的并合，即

L2A 如果 P sat S
并且P sat T
则有P sat (S ∧ T)

法则L2A 可推广到无穷个描述的逻辑与，即推广到全称量词。设S(n)为含变量n的谓词

L2 当P不随n变化时
如果∀ n • (P sat S(n))
则 P sat (∀n • S(n))[83]

如果描述S从逻辑上讲蕴含了另一描述T，则由S刻画的观察也可用T来刻画。于是每个满足S的产品也一定满足较弱的描述T，即

L3 如果 P sat S
且S ⇒ T
则P sat T

有了这项法则，我们有时可将证明串起来；如果S⇒T，我们将
P sat S
⇒T
作为下面完整证明的缩写形式。

P sat S
S ⇒T
P sat T 由L3

以上给出的法则及其解释对所有各类产品及各类描述都适用。在下一节里，我们将给出适用于进程的其它法则。

1.10.2 证明(Proofs)

在设计产品时，设计人员的责任就是确保设计的产品能满足相关要求，要做到这一点，可能使用有关数学分支中的论证方法，譬如几何学或微积分学中的方法。在这一节里，我们要给出的一组法则，从而可运用数学中的论证方法来确保进程P满足它的规约S。
我们有时把一个规约写作S(tr)，以表示一个规约通常都含有自由变量tr[84]。然而我们显示的标注变量tr的真正原因在于要说明tr可由别的复杂表达式代换，例如S(tr')。必须强调，S和S(tr)中除了有变量tr之外，还可有其它的自由变量。
不论怎样观察进程STOP，它的迹永远是空迹，因为这个进程什么也不做，故有

L4A STOP sat (tr = < >)

进程(c→P)的迹一开始是空的。每个后续的迹均以c开始，其尾部又是P的一个迹。所以这个尾部一定是由P的任何一个描述所刻画的，即

L4B 如果P sat S(tr)

则 (c→P) sat (tr = < > ∨ (tr_0 = c ∧ S(tr')))

[83] P sat (S(0) ∧ S(1)...∧ S(n))。证明很简单，true ∧true ...∧true = true。

[84] 一介谓词，first order logic。

这个法则的推论可以解决双前缀的问题。

L4C 如果P sat S(tr)
则 (c→d→P) sat (tr ≤ < c, d> ∨ (tr ≥ < c, d> ∧ S(tr")))[85]

二元选择的情况与前缀类似，区别仅在于，迹可以以两个供选择事件中的任意一个开始，而迹的尾部必须由被选中分支的描述来刻画，故有

L4D 如果P sat S(tr)
且Q sat T(tr)

则 (c →P | d→Q) sat (tr = < > ∨ (tr_0 = c ∧ S(tr'))
∨ (tr_0 = d ∧ T(tr')))[86]

以上给出的几条法则其实都是通用法则L4的特例

L4 如果∀ x ∈ B • (P(x) sat S(tr, x))[87]
则 (x : B→ P(x)) sat (tr =< > ∨ (tr_0 ∈B ∧ S(tr', tr_0)))
后继算子的法则简单得出奇。如果tr是(P/s)的迹，则s^tr是P的迹，因此s^tr必须由P所满足的描述来刻画，即

L5 如果 P sat S(tr)
且有s∈traces(P)
则(P/s) sat S(s^tr)[88]

最后，我们还需要一条法则建立递归进程的正确性。

L6 如果F(X)是卫式表达式，
且STOP sat S
且(X sat S) ⇒ (F(X) sat S))
则(μX.F(X))sat S

这项法则的前提条件保证了可用归纳法证明

F^n(STOP) sat S 对一切n[89]

---

[85] tr ≤ < c, d>的含义是tr可以是空, {c}, 或者{c, d}。其演算里包含了<>空迹，可参阅1.6.5关于Ordering的L1。空迹是任何迹的前缀。(tr ≥< c, d>∧ S(tr")))的演算形式化描述了：tr如果长度超过cd之后，去除了前面两个字母之后的迹tr"满足S。这一点也很直接。因为如果(c→d→P)，其后面的迹是P的迹。而P的迹是满足S的。综合起来，(c→d→P)一定满足(tr ≤ < c, d> ∨ (tr ≥< c, d>∧ S(tr")))的两个选择之一。

[86] 通过L4B，可以非常容易的推出这个结论。

[87] S(tr, x)表达了在多重选择的情况下的表达方式。例如，当一个选择是x的时候，其相应的规约是tr。

[88] 因为是P sat S(tr)，所以(P/s)的迹需要补一个s在tr前面，s^tr才能合成一个满足P的迹去满足P。即 sat S(s^tr)。

[89] 证明：1. 对递归函数μX.F(X), STOP属于其迹。STOP sat S成立。2. 假设F^n(STOP) sat S成立。F^{n+1}(STOP) = F(F^n(STOP))。因为已知(X sat S) ⇒ (F(X) sat S))，因此F^{n+1}(STOP) sat S。因此，根据归纳法，F^n(STOP) sat S，对一切n。

由于F(X)是卫式，F^n(STOP)至少完全描述了μX.F(X)的前n步的行为。故μX.F(X)的每个迹都是某个n的F^n(STOP)的迹。所以这个迹必须满足F^n(STOP)的描述，而这个描述对所有n来说都是S。我们将在2.8节中用数学理论给出更形式的证明。

例子

X1 我们欲证(1.1.2节X2, 1.10节X3)

VMS sat VMSPEC

证明

1. STOP sat tr =< > L4A
⇒0 ≤(tr↓coin - tr↓choc)≤ 1
(< >↓coin) = (< >↓choc) =0

这个结论中隐含着对法则L3的应用。

2. 假设 X sat(0≤tr↓coin)-(tr↓choc)) ≤1
于是 (coin →choc →X) sat (tr≤ <coin, choc>)
∨
((tr≥<coin, choc>∧ 0 ≤((tr" ↓ coin) — (tr" ↓ choc)≤1)[90] L4C

⇒ 0 ≤ ((tr ↓ coin) − (tr ↓ choc)) ≤ 1

因为< >↓coin = < >↓choc = < coin>↓choc = 0
并且< coin>↓coin = (<coin, choc>↓coin) = <coin, choc>↓choc = 1
并且tr ≥<coin, choc> ⇒ (tr↓coin = tr"↓coin+1 ∧ tr↓choc = tr"↓choc+1)[91]

应用L3，再应用L6即得证。

进程P满足它的描述并不一定就意味着是可用的。譬如，因为有

tr = < > ⇒ 0 ≤ (tr↓coin-tr↓choc) ≤ 1

应用L3和L4，可证

STOP sat 0 ≤ (tr↓coin-tr↓choc) ≤ 1

---

[90] 通过L4C，基于X sat(0≤tr↓coin)-(tr↓choc)) ≤1，可以扩展定义(coin →choc →X)的规约。

[91] 因为tr↓coin = tr"↓coin+1 ∧ tr↓choc = tr"↓choc+1为真，所以，tr↓coin - tr↓choc = tr"↓coin - tr"↓choc(原因非常直接：前面的序列是一个coin和一个choc)。根据假设 X sat(0≤tr↓coin)-(tr↓choc)) ≤1。因此递归(coin →choc →X)之后的迹仍然满足0 ≤ ((tr ↓ coin) − (tr ↓ choc)) ≤ 1。

然而STOP做为自动售货机可不够，它对主人和顾客都无用。STOP自然不会做错任何事情；但是办法太消极，什么都不做。也正是由于这个原因，STOP能满足任何进程所满足的描述。
幸运的是，很容易地可以推理证明VMS永远不会停止工作。其实任何进程，只要它是仅有前缀、选择以及卫式递归式定义的，就不会停止动作。设计一个能停止的进程只有一个办法，就是把STOP显示的写上去，或者(x：B→P(x))，这里B为空集。只要避开了这类初级错误，我们就能保证写出来的进程就是永远不停止的。然后，在下一章中引入并发性的问题之后[92]，这些简单的防御措施就无法满足不停机的要求了。在第三章的3.7节中我们将给出描述和证明某个进程永远不会停止的更一般方法。

[92] 并发(Concurrency)会导致两个貌似都不错(例如，不停机)的进程，互相死锁，并导致整体系统停机STOP。

第二章　并发(CONCURRENCY)

2.1 引言(Introduction)

进程是通过刻画其全部的潜在行为来定义的。而且经常要在几种不同动作之间做出选择，譬如说，是给VMC(1.1.3节X4)投入一枚大点的硬币呢，还是投入一枚小一点的。每当出现这种情况时，到底会发生哪个事件则完全由进程所处的环境控制。例如，总是由顾客来决定到底想投入什么样的硬币，值得庆幸的是，我们定义进程的这套符号系统记，也可用来定义一个进程的环境的行为，从而将进程的环境也刻画成一个进程，这样，我们就可以把进程连同它所在的环境合到一起，作为一个完整的系统，对其行为加以探讨和研究；在进程及其环境并发的运行过程中，它们各自动作着，且又在交互作用着。这个完整的系统也应当被看作是一个进程，它的行为可由组成它的各个进程的行为决定；而且这个系统也有可能被置于一个更大、更广的环境中去。其实，最好是忘掉进程、环境和系统三者相互之间的区别，它们统统都是进程，这些进程的行为都可以用一种简单而又统一的方式规定、刻画、记录和分析。

2.2 交互作用(Interaction)

当把两个进程放到一起，让它们并行运行时，我们一般是期望它们交互作用。这些交付行为可以被看作是要求这两个进程同时参予的事件[93]。暂时我们还是把注意力集中到这类同时参予的事件上，先不管其它事件。因此我们先假设两个进程的字母表相同。于是每个实际发生的事件，都必须是每个进程独立行为中的一个可能事件。比方说，只有当顾客需要巧克力，而且自动售货机也提供巧克力的时候，巧克力才能送出来[94]。假设P和Q是两个字母表相同的进程，我们引入

$$P \parallel Q$$

表示一个进程，其行为就是P和Q构成的整个系统的行为，P和Q的交互作用按上述说明为严格同步的[95]。

例子

X1 有位贪心的顾客想不付钱就拿到一块太妃糖或一块巧克力。只在他这种欲望得不到满足时，他才不情愿的付一枚硬币；一旦付了钱后，他就非要块巧克力不可了。即有

```
GRCUST = (toffee→GRCUST
    | choc→ GRCUST
    | coin→choc→GRCUST)
```

---

[93] 同时参与是并发算子的本质。是指并发进程要从开始第一个事件触发开始有共同的响应的行为(迹)，否则观察者无法记录这个组合进程下一步是做什么，进程的子进程无法达成共识。在第三章里会引入和接受关于非确定性的算子，例如，两个子进程可以按照自己支持的，不同于另外一个子进程的行为来出来外界环境和之后的处理流程。

[94] 这里蕴含的意思是环境(顾客)与售货机(进程)都需要能够处理“巧克力”这个事件，顾客与售货机才能互动协调起来共同相应“巧克力”这个事件。

[95] 可以认为P是Q的环境。也可以认为Q是P的环境。互相同步，互相牵制，形成一个共同对外的系统。

当这位顾客遇到VMCT(1.1.3节X3)时[96]，就不能得逞了，因为这台机器不允许先尝后付钱。同时，因为这个人付钱之后只要巧克力，所以VMCT也就不会给出太妃糖来[97]。因此有

(GRCUST ‖ VMCT) = μX. (coin→ choc→X)[98]

这个例子说明，不用并发算子‖，也可以把两个子进程组合成的进程描绘成一个单一的简单的进程[99]。

X2 一位傻乎乎的顾客想买一块大饼干，于是就把硬币投入VMC。他并没有注意自己投入是哪种硬币；可是不管怎么说，他非要块大饼干不可，他的行为模式可以形式化的描述为：

FOOLCUST = (in2p →large→FOOLCUST
|inlp → large→ FOOLCUST)

可惜的是，自动售货机可不愿吃亏。这位顾客投入一枚小硬币，它才不会给出大饼干的，故

(FOOLCUST ‖ VMC) = μX • (in2p →large→ X| inlp →STOP)[100]

伴随着inlp后发生的STOP被叫做死锁。尽管每个分进程都准备好进行下一步的动作，但它的各自的下一步动作都是不同的；由于这些进程无法统一意志，下一步就什么也不会发生了[101]。 □

以上例子的故事，违背了科学抽象性和客观性的标准。但切记一点，各种事件都特意定义成为一些中性的过度状态，从而可以由那些不懂得七情六欲的外星人观察和记录下来。这类外星人不懂吃饼干的乐趣，也不了解那位愚蠢的顾客徒然等待粮食时挨饿的痛苦。我们选择有关事件的字母表时有意地排除了这些内在的感情因素；但如果真有这种需要时，我们还可引入某类事件来模拟这种内在的状态变化，详情可参见2.3节中的X1。

2.2.1 法则(Laws)

(P ‖ Q)的行为遵从的法则非常简单直接。第一条法则说明进程与其环境之间的逻辑对称性，

---

[96] VMCT = μX • coin →(choc→X|toffee→ X)。显然，这个售货机只支持先付钱，才有其他可能(糖或者巧克力)的行为。

[97] 两个进程(顾客和售货机)同步，最后达到一个一致对外的行为结果。所以，不可能有糖的选择，不是一个两个进程都共同存在的一个演化路径。

[98] 两个进程并发，是一种合作行为，不是各自为政的非确定性的随机行为。并发的结果是结果的迹在任意一个进程里都是一致的。从而，两个进程合成的进程对外体现的是一个唯一的，一致的行为和记录下来确定的迹。

[99] 也可以反过来思考：任何一个进程，也可以通过并发算子把两个子进程组合起来的行为来表示。

[100] VMC = (in2p→(large→VMC | small→outlp→VMC) | inlp→(small→VMC | inlp→(large→VMC| inlp→STOP)))。可以很容易的观察，当投入一个in1p的时候，进程FoolCust的行为和VMC的行为无法取得下一步的共同的步骤。因此，只能是STOP。

[101] 假设这两个独立的进程，都通过一个共同的外部事件触发，例如，in1p，（这时观察者可以记下这个迹。），但两个进程开始出现分歧，对in1p触发之后的处理不一致，因此导致符合进程无法再往下发展，成为STOP状态。

即

L1 $P \parallel Q = Q \parallel P$[102]

下一个法则说明当三个进程组装在一起时，它们之间的排列顺序没有关系

L2 $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$[103]

第三，一个死锁进程会使整个组合的系统都死锁；但是一个进程如果与$RUN_{\alpha P}$(1.1.3节X8)组合，结果不会受到影响。

L3A $P \parallel STOP_{\alpha P} = STOP_{\alpha P}$

L3B $P \parallel RUN_{\alpha P} = P$[104]

下一条法则说明一对进程或者同时执行同一动作，或者在执行第一个动作时无法取得一致意见，而处于死锁，即

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = (c \rightarrow (P \parallel Q))$[105]

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP$[106]　　若$c \neq d$

这两条法则可以很容易地推广到两个进程都可以选择初始事件的情况；当把这两个进程组合到一起时，只有他们的公共选择部分才能保留下来，即

L4 $(x: A \rightarrow P(x)) \parallel (y: B \rightarrow Q(y)) = (z:(A \cap B) \rightarrow (P(z) \parallel Q(z)))$[107]

规则L4表明使用并发算子定义的系统可以由不使用并发算子的方式来刻画的。

例子

X1 设$P = (a \rightarrow b \rightarrow P \mid b \rightarrow P)$
且$Q = (a \rightarrow b \rightarrow Q \mid c \rightarrow Q)$

则$(P \parallel Q) = a \rightarrow (b \rightarrow P) \parallel (b \rightarrow \mid c \rightarrow Q))$　　由L4A
$= a \rightarrow (b \rightarrow (P \parallel Q))$
$= \mu X \cdot (a \rightarrow b \rightarrow X)$　　因为这个递归方程式是卫式

---

[102] 并发算子满足交换律
[103] 并发算子满足结合律
[104] 有点类似一个子集与一个无穷大的全集做过滤。显然过滤的结果是这个子集。
[105] 两个进程在对外的第一个event上，有共同的事件。因此可以往下走一步。外界观察者能够记录下来c作为trace。不会出现记录不下来的现象。
[106] 对这个法则可以从两个方面来理解。1. 这两个进程组合起来的进程无法在第一个事件上取得共识，从而导致方程右侧的行为无法被观察者做任何记录，形成迹(trace)。2. 从1.8的法则L6，我们可以指导，<>空迹其实属于任何一个进程，例如，这个组合进程。因为这个组合进程的唯一的两个组成部分不可能达成对第一个事件的相应，因此，这个组合进程的行为只能是<>。或者说是一个空进程STOP。
[107] 要么是空集，组合后的进程是一个STOP进程，即虽然内部暗潮涌动(由两个或者多个子进程组成)，但什么也做不了，无法对外接受事件的触发，形成不了一致意见。

2.2.2 实施(Implementation)

显然组合算子 ‖ 的实施要以规则L4为基础

$$\text{intersect}(P, Q) = \lambda z \bullet \text{if } P(z) = \text{"BLEEP} \vee Q(z) = \text{"BLEEP then"BLEEP else intersect}(P(z), Q(z))$$

2.2.3 迹(Traces)

由于(P ‖ Q)的每一个动作都要P和Q同时参与，则这类动作的每个序列必须同时是这两个运算对象的动作序列。由于这个原因，/s对 ‖ 算子有分配律。

L1 $\text{traces}(P \parallel Q) = \text{traces}(P) \cap \text{traces}(Q)$[108]

L2 $(P \parallel Q)/s = (P/s) \parallel (Q/s)$[109]

2.3 并发性(Concurrency)

上节中讲述的算子也可以推广到运算对象P和Q的字母表不同的情形，即

$$\alpha P \neq \alpha Q$$

当把这样的两个进程组装到一起并发运行时，它们字母表中共有的事件就需要P和Q同时参加，其理由见2.2[110]。然而，属于αP但不属于αQ的事件就跟Q无关系了，也没能力去控制甚至过问这类事件。因此每当P在执行这类事件时，它们可以独立于Q发生。类似地，Q也会执行只属于αQ而不属于αP的事件[111]。这样，只需将参加组合的各个进程的字母表简单地并到一起，就可得到整个组合系统逻辑上所有可能事件的集合。故

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

这种运算对象的字母表不相同的算子不多见，而且运算结果又产生了另一个字母表。但在两个进程字母表相同的情况里，它们结合后字母表仍然不变，(P ‖ Q)的含义与上节所述的完全一样。

例子

X1 假设αNOISYVM = {coin, choc, clink, clunk, toffee}

---

[108] 基于处理的每一步，两个进程共同的行为。
[109] 在迹s之后的迹。这里显然s是复合进程的迹。
[110] 共有的事件的含义是：对这些事情的处理是对外可观察到的，需要被记录成为trace的。因此，假设在某一事件之前，所有的动作序列都是符合并发算子的要求的。但是在这个当前事件上(事件属于各个进程的字母表），处理的下一步如果属于共同的事件，则必须一致，否则观察者无法记录迹，类似系统不知道到底是做哪个事件。符号表里的事件的定义是一个进程必须对这个事件有所反应。
[111] 对任何一个只属于某一个进程P的事件，不需要其余那个进程Q行为的支持。这个进程P里事件的发生不会干扰到进程Q的行为。对作为记录复合并发进程(P ‖ Q)的迹的观察者，不会无所适从。

这里clink是硬币掉进自动售货机时的声音，clunk是自动售货机做完一批交易后发出的声音。另外，这台响个不停的售货机里已经售完太妃糖了，其行为可以形式描述为：

NOISYVM =
(coin→clink→choc→clunk→NOISYVM)

但是顾客就想要太妃糖；curse是顾客拿不到太妃糖时的咒骂；后来他还是不得不拿一块巧克力。故

αCUST= {coin, choc, curse, toffee}

CUST =(coin→(toffee →CUST | curse→choc→ CUST))

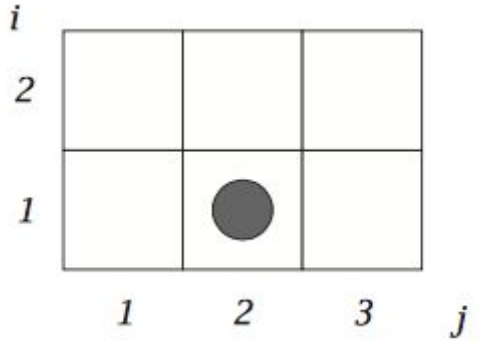
这两个进程并发动作的结果就成了[112]

(NOISYVM ‖ CUST) =
μX • (coin→(clink→curs→ choc→ clunk→X | curse→clink
→ choc →clunk →X))[113]

注意，clink可能发生在curse之前，也可能在它之后。这两个事件甚至有可能同时发生，但记录它们的前后次序是无关紧要的[114]。

还要注意的是，数学公式是无法表现顾客的感情的，他当然情愿要块太妃糖而不愿意说句难听的话。数学公式作为一种对现实的抽象，忽略人的感情因素，只去描述进程字母表的事件发生或不发生的可能性，不管人们是否喜欢这些事情。

X2



一个筹码从如上图所示的方格开始，这个筹码可在板上、 下、左、右移动。

假设αP = {up, down}

P = (up→down→P)

αQ ={left, right}

[112] 在这个复合的并发进程里，其中clink，clunk事件只会被NOISYVM进程所感知和触发（或者被触发）；curse智慧被CUST客户进程所触发。

[113] 如果不考虑那些独立的事件，这两个进程复合的对外的行为，或者是可以被记录下来的迹是：μX • (coin→ choc→X)。一个只接受硬币，吐出巧克力的行为。

[114] clink和curse的事件发生不存在依赖关系。但都必须这choc之前。满足自己对应进程行为描述中事件的前后顺序。

$$Q = (right \rightarrow left \rightarrow Q \mid left \rightarrow right \rightarrow Q)$$

筹码的行为可定义为 $P \parallel Q$。
在此例中，P和Q没有一个共同事件。所以，这个筹码的运动就是P和Q的动作的任意穿插。这种穿插如果不用并发性而用其它方法来刻画，就会很冗长。例如，设R_{ij}表示筹码由板上第i行、第j列的位置上出发的可能的行为，其中$i \in \{1, 2\}, j \in \{1, 2, 3\}$。则有

$$(P \parallel Q) = R_{12} \quad \text{其中}$$

$$R_{21} = (down \rightarrow R_{11} \mid right \rightarrow R_{22})$$

$$R_{11} = (up \rightarrow R_{21} \mid right \rightarrow R_{12})$$

$$R_{22} = (down \rightarrow R_{12} \mid lef \rightarrow R_{21} \mid right \rightarrow R_{23})$$

$$R_{12} = (up \rightarrow R_{22} \mid left \rightarrow R_{11} \mid right \rightarrow R_{13})$$

$$R_{23} = (down \rightarrow R_{13} \mid left \rightarrow R_{22})$$

$$R_{13} = (up \rightarrow R_{23} \mid left \rightarrow R_{12})$$ □

2.3.1 法则(Laws)

前三条法则是对并发性的扩展，与交互作用的法则(2.2.1节)非常类似，它们是

L1, 2 ‖既是对称的又是结合的

L3A $P \parallel STOP_{\alpha P} = STOP^{115}_{\alpha P}$

L3B $P \parallel RUN_{\alpha P} = P$

假设 $a \in ((\alpha P - \alpha Q), b \in (\alpha Q - \alpha P)$，且$\{c, d\} \subseteq ((\alpha P \cap \alpha Q)$。那么以下的几条法则说明，P可单独执行事件a，Q可单独执行事件b，但执行c和d时需要P和Q同时参加[116]。

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP^{117}$ 若$c \neq d$

---

[115] 组合后的进程没法对字母表αP里的任何事件做出响应。
[116] 同时参加，一致，从而复合进程可以完整的体系一个不混乱的迹。
[117] 根据并发算子的定义，$(c \rightarrow P)$进程能够处理的第一个非空的事件是c；$(d \rightarrow Q)$能够处理的第一个非空的事件是d; 而且$c \neq d$。因此，$((c \rightarrow P) \parallel (d \rightarrow Q))$这个进程陷入了混沌状态，没法往任何一个共同的行为方向发展，只能进入STOP。

L5A $(a\to P) \parallel (c\to Q) = a\to(P \parallel (c\to Q))$[118]

L5B $(c\to P) \parallel (b\to Q) = b\to((c\to P) \parallel Q)$[119]

L6 $(a\to P) \parallel (b\to Q) = a\to(P \parallel (b\to Q)) \mid b\to((a\to P) \parallel Q))$[120]

上述法则可推广到通用选择算子上。

L7 设$P = (x: A\to P(x))$
且$Q = (y: B\to Q(y))$
则$(P \parallel Q) = (z: C\to P' \parallel Q')$
这里$C = (A \cap B) \cup (A-\alpha Q) \cup (B-\alpha P)$
且有$P' = P(z)$[121] 如果$z \in A$
$=P$[122] 否则
$Q' = Q(z)$ 如果$z \in B$
$=Q$ 否则

利用这些法则，可以把由并发算子定义的进程简化，重新定义，使其中不出现并发算子，如下例。

例子

X1 设
$\alpha P = \{a, c\}$, $P=(a\to c\to P)$
$\alpha Q = \{b, c\}$, $Q=(c\to b\to Q)$

于是 $P \parallel Q = (a\to c\to P) \parallel (c\to b\to Q)$ 由定义
$=a\to((c\to P) \parallel (c\to b\to Q))$ 由L5A
$=a\to c\to(P \parallel (b\to Q))$ 由L4A...‡

而且$P \parallel (b\to Q)$[123] $= (a\to(c\to P) \parallel (b\to Q)$
$\mid b\to(P \parallel Q))$ 由L6
$= (a\to b\to((c\to P) \parallel Q)$[124]
$\mid b\to(P \parallel Q))$ 由L5B

---

[118] a是$(a\to P)$进程或者说P进程独有的事件。整个复合进程的行为显然可以支持a的发生，而不需要考虑$(c\to Q)$的同步并发问题。如果a发生并且记录下a之后，这个时候要"看"之后的P和$(c\to Q)$了。因为c是P和Q都需要反应的事件。所以，未来的整体行为不好说，a发生后，系统的行为只能是$(P \parallel (c\to Q))$。1. 我们假设P是$(d\to P')$。我们可以很容易的理解，这个复合并发进程的迹最多能记下一个a事件，然后就会进入STOP。原因是$(d\to P') \parallel (c\to Q)$的首发无法能够取得一致。整体系统陷入混乱(chaos)状态，只能STOP。2. 如果P是$(c\to P')$， 整个进程显然可以往下多发展一步，因为，$(c\to P') \parallel (c\to Q) = c\to(P' \parallel Q)$。系统的行为，或者迹可以发展到$<a, c>$了。。。依此逻辑，可以一直推理下去。

[119] 与L5A是一个逻辑。

[120] a与b是互相独立的可以交错的事件。所以整体组合进程的行为可能是a先发生，或者b先发生。值得注意的是，$a\to(P \parallel (b\to Q))$里的不能简单的推导出$b\to(P \parallel Q)$。原因是我们并不知道P的细节。如果P是类似$(c\to P')$，根据L5B，$(P \parallel (b\to Q)) = (c\to P' \parallel (b\to Q)) = b\to(c\to P' \parallel Q) = b\to(P' \parallel Q)$；但如果P还是$(a\to P')$的模式，$(P \parallel (b\to Q)) = (a\to P' \parallel (b\to Q))$，那么后续行为是$a\to(P' \parallel (b\to Q)) \mid b\to((a\to P') \parallel Q))$。

[121] 当C属于 $(A \cap B) \cup (A-\alpha Q)$的时候。

[122] 当C属于$(B-\alpha P)$的时候。

[123] 首先展开为$(a\to c\to P) \parallel (b\to Q)$。因为a, b是独立的两个事件分别属于$\alpha P$和$\alpha Q$，因此应用L6开始推导。

[124] 通过L5B，$(c\to P) \parallel (b\to Q)$可以容易的推导为$b\to((c\to P) \parallel Q$。

= (a→b→c→(P ‖ (b→Q))[125] 由‡
|b→a→c→(P ‖ (b→Q))) [126]
= μX • (a→b→c→X) 因为是卫式
|b→a→c→X

所以

(P ‖ Q)=(a→c→μX • (a→b→c→X
|b→a→c→X)) 由‡

□

2.3.2 实施(Implementation)

算子 ‖ 的实施可直接由L7导出。我们可以把运算对象的字母表表示成符号的有限表A和B。要检验一符号是否在表中，就用我们在1.7节中定义过的函数ismember(x,A)
要实现(P ‖ Q)，可以通过调用函数concurrent(P, αP, αQ, Q)
这个函数是这样定义的

```
concurrent(P, A, B, Q) = aux(P, Q)
这里aux(P, Q)=
λx.if="BLEEP or Q="BLEEP then "BLEEP
else if ismember(x,A) ∧ ismember(x,B) then aux(P(x)
      Q(x))
else if ismember(x,A) then aux(P(x),  Q)
        else if ismember(x,B) then aux(P,  Q(x))
              else"BLEEP
```

2.3.3 迹(Traces)

假设(P ‖ Q)的迹为t。则t中属于αP的每个事件都是在P的生存期内发生的事件；不属于αP的事件发生时，不需P参加。(t αP)是有P参加的所有事件组成的迹，是P的一个迹。同理，(t αQ)也是Q的一个迹。另外，t中每个事件如不在αP中就一定在αQ中。由此，我们有L1

L1 traces(P ‖ Q) = { t | (t αP) ∈ traces(P)
∧ (t αQ) ∈ traces(Q)
∧ t ∈ (αP ∪ αQ)* }[127]

下一个规则是关于算子/s对并行组合有分配律，即

L2 (P ‖ Q)/s = (P/(s αP)) ‖ (Q/(s αP)

当αP=αQ, 则有s αP = s αP = s

---

[125] 开始化简(c→P) ‖ Q。把Q = (c→b→Q)代入，得到(c→P) ‖ (c→b→Q)。因此，得到c→(P ‖ (b→Q))。代入到上一步的方程式中，得到 a→b→c→(P ‖ (b→Q))。

[126] 把前面得到的P ‖ Q =a→c→(P ‖ (b→Q))代入b→(P ‖ Q)，得到b→a→c→(P ‖ (b→Q))。

[127] P ‖ Q的迹一定都满足：如果过滤P的字母表，一定属于进程P的迹；如果过滤Q的字母表，一定属于进程Q的迹；属于P和Q的字母表的幂级的组合。

在这种情况下这些法则就与2.2.3节中的完全一样了。

例子

X1 (参阅2.3节X1)

假设 t1 = <coin, clink, curse>
则t1 ↾ αNOISYVM = <coin, clink>　　属于traces(NOISYVM)
且t1 ↾ αCUST = <coin, curse>　　属于traces(CUST)
因此t1 ∈ traces(NOISYVM ‖ CUST)

我们可类似地推断

<coin, curse, clink> ∈ traces(NOISYVM ‖ CUST)

这说明curse和clink的记录顺序可以不同。它们有可能还会同时发生，只不过我们已决定不提供记录两个事件同时发生的方法罢了。 □

总之，(P ‖ Q)的迹是P和Q的迹的交错混合在一起，其中P和Q字母表中共有的事件只发生一次。如果αP ∩ αQ = {}，则P和Q的迹是单纯的互相交错(1.9.3节)，例如2.3节中X2。另一种极端的情形，即αP=αQ，迹中的每个事件都同时属于P和Q的字母表，此时(P ‖ Q)的含义就与交互作用的含义完全相同了(2.2节)。

L3A 如果 αP ∩ αQ = {}

traces(P ‖ Q) = { s ｜ ∃ t: traces(P); ∃ u: traces(Q) • s interleaves(t, u)}

L3B 如果 αP = αQ

traces(P ‖ Q) = traces(P) ∩ traces(Q)[128]

2.4 示意图(Pictures)

我们用一个标有P的方框来表示字母表为{a, b, c}的进程P，由这个方框再引出几条线段，每条线段都用字母表中不同事件表明(图2.1)。类似地，字母表为{b, c, d}的进程Q如图2.2所示。

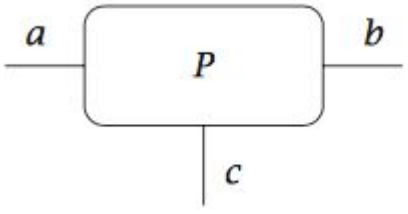


图2.1

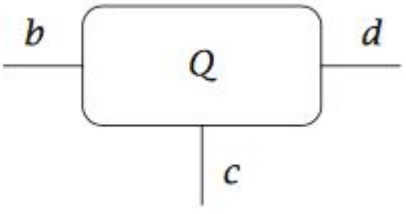


图2.2

[128] 如果没有单独只属于某个进程的事件，则P ‖ Q的迹只能是两个进程迹的交集(含空)。

当把这两个进程连到一起并运行时，所得的系统的示意图就象是个网络，具有相同标记的线段连接在一起，但是如果线段上标记的事件只属于其中一个进程的字母表，则这条线段不与任何其它线段连接(图2.3)。

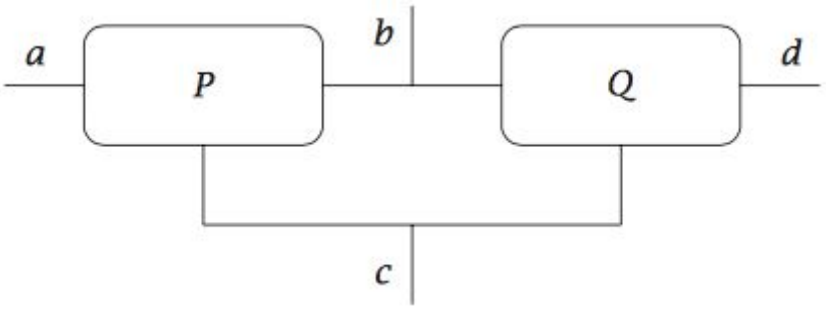


图2.3

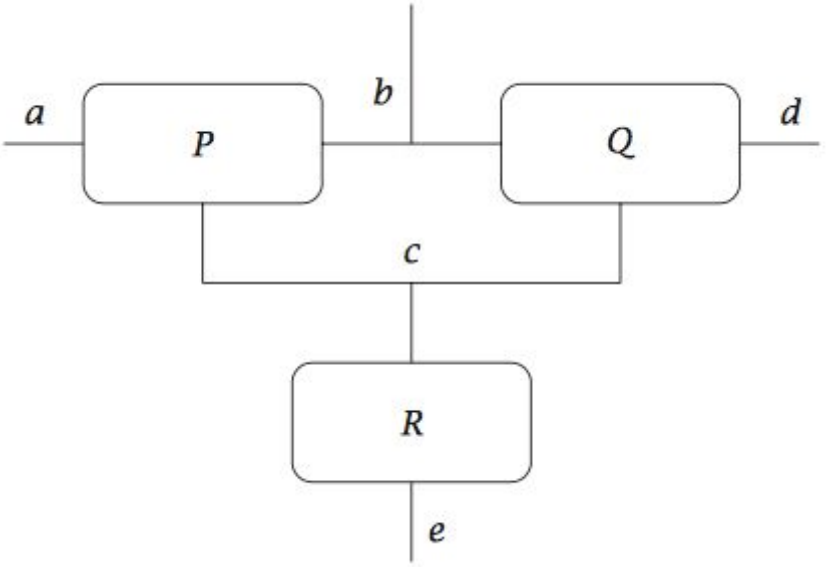


图2.4

在2.3的基础上也可以再添加进程R，αR = {c, e}，进程R加到示意图上去，形成图2.4。在这个框里，事件c的发生需要三个进程P, Q和R同时参加，事件b需要P和Q两个进程同时参加，其它的事件如a, d, e就只与单个进程有关了。这类的示意图被称做为连接框图。

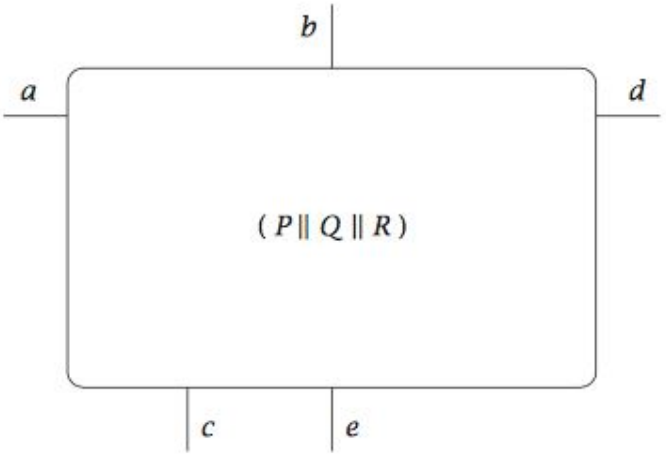


图2.5

但这些示意图很容易引起误解。其实，由这三个进程构成的一个系统也还是一个进程，因此应当画成一个方框(如图2.5)。例如，整数60可由三个数乘积(3×4×5)构成；但一旦构成后，它也只是一个整数而已，和它的构成方式就不再有什么关系了，甚至也不可能从它本身观察到它的构成方式了。

2.5 例子：哲学家就餐问题(The Dining Philosophers)

古时候，一位非常富有的慈善家出钱资助一所学院，让学院为五位杰出的哲学家提供食宿。每个哲学家都有一个房间，他可以在里面专心思考；但是餐厅是公共的，里面摆了一张圆桌、五把椅子，每把椅子上都标有一位哲学家的名字。他们的名字是 $PHIL_0$, $PHIL_1$, $PHIL_2$, $PHIL_3$, $PHIL_4$。并且以逆时针次序围坐在桌旁。在每位哲学家的左侧放着一把金叉，桌子正中间摆着一个大碗，里面盛着意大利细条通心面，而且随时地有人添加着。

哲学家本来就是要把他的主要时间用于思考的；但如果觉得饿了，他就去餐厅，坐在他的椅子上，拿起放在左边的叉子，吃一顿细条通心面。可事情并不这么简单，要把缠结的通心面送到嘴里，必须用第二把叉子。所以这位哲学家非得把他右边的叉子也拿起来不可。一旦吃完了，他就放下两把叉子，站起来回到他房间里继续思考去。当然，一把叉子一次只能由一个人用。如果有另外一位哲学家要用这把叉子，他只能等头一位哲学家用完了再说。

2.5.1 字母表(Alphabets)

现在我们为这个系统构建一个数学模型。首先我们必须选出相关事件的集合。对 $PHIL_i$，集合定义为

$$\alpha PHIL_i = \{i.\text{sits down}, i.\text{gets up}, i.\text{picks up fork.}i, i.\text{picks up fork.}(i \oplus 1), i.\text{puts down fork.}i, i.\text{puts down fork.}(i \oplus 1)\}$$

这里$\oplus$是模数为5的加法，因此$i \oplus 1$就是第i位哲学家右侧的邻座编号。

注意每个哲学家的字母表是相互独立的，他们不在一块做任何事件，所以就不相互作用或通信——这是那个时代哲学家们的行为的客观反映。

在我们这个小剧目里的其它角色就是那五把叉子，每把叉子上贴的号与它主人的号相同。一把叉子或由其主人使用，或由叉子另一边的，主人的邻居使用。则第i把叉子的字母表为

$$\alpha FORK_i = \{i.\text{picks up fork.}i, (i \ominus 1).\text{picks up fork.}i, i.\text{puts down fork.}i, (i \ominus 1).\text{puts down fork.}i\}$$

这里$\ominus$表示模数为5的减法。

这样，除了坐下和站起来的事件之外，其它的每个事件都需要两个密切相关的演员参加，一位哲学家和一把叉子，如连接图2.6所示。

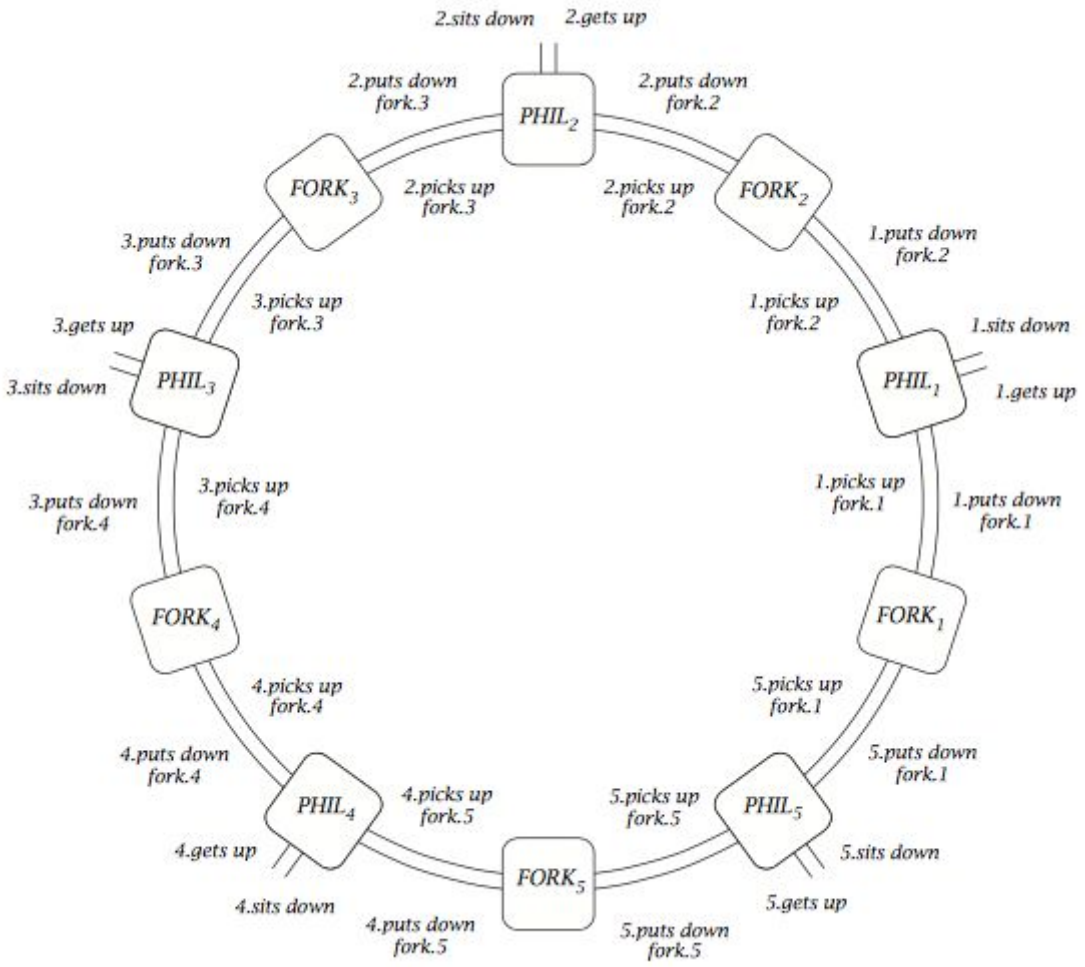


图2.6

2.5.2 行为(Behaviour)

除了我们有意忽略的思考和吃饭两件事件外，哲学家的生活就是六件事件为一个周期，不断重复，即

$$
\begin{aligned}
PHIL_i = (&i.\text{sits down} \rightarrow \\
&i.\text{picks up fork.}i \rightarrow \\
&i.\text{picks up fork.}(i \oplus 1) \rightarrow \\
&i.\text{puts down fork.}i \rightarrow \\
&i.\text{puts down fork.}(i \oplus 1) \rightarrow \\
&i.\text{gets up} \rightarrow PHIL_i)
\end{aligned}
$$

叉子扮演的角色很简单；就是被坐在旁边的哲学家重复地拿起来或放下(两个动作都由同一个人做)，即

$$
\begin{aligned}
FORK_i = (&i.\text{picks up fork.}i \rightarrow i.\text{puts down fork.}i \rightarrow FORK_i \mid (i \ominus 1).\text{picks up fork.}i \rightarrow \\
&(i \ominus 1).\text{puts down fork.}i \rightarrow FORK_i)
\end{aligned}
$$

[129] 拿左手的叉子。i.picks up fork.(i⊕1)拿右手的叉子。

所以整个学院的行为就是各个组成部分的行为的并发组合

$$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$$

$$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$$

$$COLLEGE = PHILOS \parallel FORKS$$

这个故事稍加有趣变动，允许哲学家以任意左右次序拿起或放下他们的叉子[130]，即分别考虑哲学家每只手的行为。每只手都能拿起放下旁边的那把叉子，但是要想坐下或站起来，必须两只手帮忙。这样就有

$$\alpha LEFT_i = \{i.picks\ up\ fork.i,\ i.puts\ down\ fork.i,\ i.sits\ down,\ i.gets\ up\}$$

$$\alpha RIGHT_i = \{i.picks\ up\ fork.(i \oplus 1),\ i.puts\ down\ fork.(i \oplus 1), i.sits\ down,\ i.gets\ up\}$$

$$LEFT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i)$$

$$RIGHT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.(i \oplus 1) \rightarrow i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i)$$

$$PHIL_i = LEFT_i \parallel RIGHT_i$$

坐下和站起来是需要 $LEFT_i$和$RIGHT_i$的同步工作[131]，这一点就保证了只有当哲学家入座后，才能拿起叉子来。除此之外，对两把叉子的操作则是任意交叉的。
我们把这个故事再变一变，当哲学家坐下后，他可以把叉子拿起来或放下去很多次。这样，他两只手的行为就要有所改动，允许有一种重复的动作，例如

$$LEFT_i = (i.sits\ down \rightarrow \mu X\ .\ (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow X \mid i.gets\ up \rightarrow LEFT_i))$$

2.5.3 死锁(Deadlock)!

当数学模型建立起来后，揭示了一个严重的问题。如果说这五位哲学家在大致同一时间里都饿了，他们都到餐厅里坐下来，拿起属于他的那把叉子，然后去拿另一把—但这另一把叉子却不在了。在这种尴尬的情况下，他们必定要挨饿。尽管每个角色都有

---

[130] 之前的行为定义是：先拿自己的叉子(左手)，然后拿别人的(右手)。

[131] $PHIL_i = LEFT_i \parallel RIGHT_i = (i.sits\ down) \rightarrow ((i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i) \parallel (i.picks\ up\ fork.(i \oplus 1) \rightarrow i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i))$。参加2.3.1的L6，显然$PHIL_i$的行为会是坐下来之后，左手，右手的4个独立事件的各种组合。然后在都放下叉子后，同步在站起来的事件上。然后递归回去。

能力做更多的动作，但他们中的任意两个都不能合作地往下发展[132]。
尽管如此，我们这个故事的结尾不会这么惨，一旦发现问题，总可以找到解决的办法。譬如说，他们中的一位拿第一把叉子时总是先拿右邻的—但很难使他们一致同意由谁这样做！同样的理由，也难决定再买一个叉子专归某一位，而再买五把又太贵了。最后采取的解决办法是找位男仆，他的职责就是为每个人挪椅子，他的字母表是

$$\bigcup_{i=0}^{4} \{i.\text{sits down}, i.\text{gets up}\}$$

这位仆人得到秘密指示，最多只能让四个人同时吃饭。则他的行为如果用联立递归式定义就非常简单。

$$U = \bigcup_{i=0}^{4} \{i.\text{gets up}\} \quad D = \bigcup_{i=0}^{4} \{i.\text{sits down}\}$$

$FOOT_j$定义仆人帮助j位哲学家就座后的行为

$$FOOT_0 = (x : D \rightarrow FOOT_1)$$
$$FOOT_j = (x : D \rightarrow FOOT_{j+1} \mid y : U \rightarrow FOOT_{j-1}) \quad j \in \{1, 2, 3\}$$
$$FOOT_4 = (y : U \rightarrow FOOT_3)$$

没有死锁问题的学院定义为

$$NEWCOLLEGE = (COLLEGE \parallel FOOT_0)$$[133]

这个哲学家就餐的故事是来自Edsger W. Dijkstra。雇一位仆人的主意则来自Carel S. Scholten。

2.5.4 死锁不存在的证明(Proof of absence of deadlock)

在COLLEGE进程中，死锁问题并非显而易见；因此，如果说在NEWCOLLEGE中就没有死锁问题，需要认真地证明才行。我们必须要证明的东西可形式地叙述为

$(NEWCOLLEGE / s) \neq STOP$　对全部$s \in traces(NEWCOLLEGE)$都成立。

[132] 可能更是一个starving的状态。每个哲学家都在一直试图反复的看是否可以拿到右手的叉子。
[133] 一旦餐桌有了4个哲学家在就餐，FOOT进程会进入只响应i.gets up的事件。从而导致不会出现系统的迹还出现5个哲学家争抢的现象。

证明时先任取这个进程的一个迹s，然后证明在任何情况下，至少能找到一个事件，以延伸s后得到的序列扔在traces(NEWCOLLEGE)。我们定义入座的哲学家的人数为

$$seated(s) = \#(s \upharpoonright D) - \#(s \upharpoonright U)$$ 其中U和D的定义如上。

因为(由2.3.3节的L1) $s \upharpoonright (U \cup D) \in traces(FOOT_0)$，我们知道有$seated(s) \leq 4$。如果$seated(s) \leq 3$，则至少还能再容下一个人入座，所以不会出现死锁。当$seated(s)=4$时，数一下正在吃的哲学家有几位(正吃的人都举着两把叉子)。如果有人正在吃，则正吃的一位总是可以放下他左手拿的叉子。如果没人在吃，就看着被拿在手里的叉子的个数。如果只有或还不足三把叉子被拿在手里，则有一位就座的哲学家还可以拿起他左边的叉子。如果已拿起四把，则坐在空位左侧的那位不仅左手握叉，而且还能把他右边的叉子也拿起来。如果五把叉子都被拿起来了，则这四位中至少有一位一定在吃了。

非形式化的描述这个具体例子的行为的证明分析要分很多种情况[134]。现在我们换一种证明方法：设计一个计算机程序，用来研究这个系统所有可能的行为，来寻找死锁[135]。一般说来，这样的一个程序无法保证死锁的不存在，因为我们永远无法知道这个程序是否已经遍历了全部可能行为[136]。但是象在COLLEGE这样的有限状态系统中，所需考虑的迹，只是有穷多个，因为它们的长度不会超过可由状态数计算出的某个已知上限。(P ‖ Q)的状态数不超过P的状态数与Q的状态数之积。由于每个哲学家有六个状态，每把叉子有三个状态，则COLLEGE的状态总数不超过

$6^5 \times 3^5$ 大约1.8×10^6

由于仆人(Footman)的字母表已经包含在COLLEGE中，所以NEWCOLLEGE的状态不会比COLLEGE多。又因为在每个状态的中几乎都包含了两个或更多的可能事件，我们要考虑的迹的个数就超过$2^{1.8 \times 10}$了。尽管是有穷多个，但要让一个计算机程序遍历全部这些迹，简直是不可能的。因此，即使对一个很简单的有限状态进程来说，其死锁的不存在性的证明还是很艰难，必须留给并发系统的设计人员去处理。

2.5.5 无限抢先(Infinite overtaking)[137]

除了死锁之外，吃饭的哲学家还面临着另一个危险——总是被他的邻位抢了先，假设一位入座的哲学家左手不太灵活，又有个非常贪吃的左邻座。还没等他把左边的叉子拿起来，那位邻座就闯进来，坐下来，一下子就把他左右的两把叉子都拿起来了，半天吃不完。好不容易吃完了放下两把叉子，离开座位。可是马上又饿了，又跑回来，坐下，抓起两把叉子又饱餐一顿，只苦了等了好久的那位右边的哲学家，他来不及拿到那把他们两个人合用的叉子。这个周期可能无限循环下去，这样就会有一位入座的哲学家总也吃不成。

这个问题没有办法解决，因为如果真有位那么贪吃的哲学家则肯定有人(不是他就是他的邻座)要长时间挨饿。实在没有什么聪明的办法使大家都满意，只能再多买几把叉子

---

[134] 类似穷举所以可能的后续行为。
[135] 有兴趣的读者可以参阅Lesie Lamport的TLA+。
[136] 或者是可以遍历，但状态空间爆炸，遍历需要指数级别的时间。
[137]操作系统中的starving的概念。

，多添几次通心面，才可以解决这个问题。

但是，如果保证入座的人最终都能吃到，可以修改一下男仆的行为：在他帮着一位哲学家入座后，他就一直等到这个人把两把叉子都拿起来之后，才去帮着这个人的左右邻居入座。

2.6 符号变换(Change of symbol)

在前一节所举的例子中有两组进程，一组是哲学家，一组是叉子；每一组中进程的行为相似，只是执行的事件的名字不同。这一节里，我们介绍一个很方便的方法，用以定义行为相似的成组进程。设f为一个单射函数，它将F的字母表映射到符号集合A上去

$$f : \alpha P \rightarrow A$$

我们定义$f(P)$为这样的一个进程：当P 执行事件c时，$f(P)$就执行事件$f(c)$。它遵从以下规则

$$\alpha f(P) = f(\alpha P)$$

$$traces(f(P)) = \{ f^*(s) \mid s \in traces(P) \}$$

(f^*的定义看1.9.1节)。

例子

X1 过了几年之后，所有的东西都要涨价。要把通货膨胀表示出来，我们以下列方程式定义函数

f(in2p) = in10p　　f(large) = large
f(inlp) = in5p　　f(small) = small
f (outlp) = out5p

则新的自动售货机就是

$$NEWVMC = f(VMC)$$ □

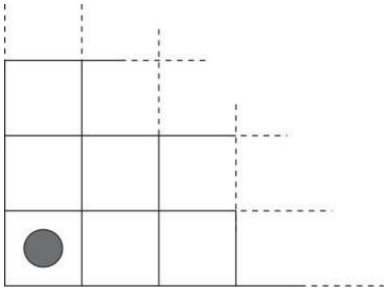
X2 一个筹码，行为跟CT_0相同(1.4.4节X2)，不同的是，它是向左和向右移动，而不是向上或向下移动，令

$$f(up) = right,\ f(down) = left,\ f(around) = around,$$

$$LR_0 = f(CT_0)$$ □

我们对进程的事件名做这种变换，主要的原因是使它们能在并发组合时充分发挥作用。

X3 一个筹码可在板中、上、左、右移动，这块板是上方和右方无穷的，其边界在左边沿和下边沿。



如图筹码从左下角出发，也仅在这个方格里，它才允许转圈。用第2.3节X2中的办法，将垂直和水平移动分别看作是不同进程的独立动作；但是around就需要两个进程同时参加，故LRUD= LR_0 ‖ CT0。

X4 我们希望把两个COPYBIT(参看1.1.3节X7)连接起来，这样由第一个COPYBIT输出的每个数位同时成为第二个COPYBIT的输入。首先，我们改变一下用作内部通信的事件名；引入两个新的事件mid.0和mid.1，再定义函数*f*和g，以变换一个COPYBIT的输出及另一个COPYBIT的输入，即

f (out.0) = g(in.0) = mid.0
f (out. 1) = g(in.1) = mid.1
f (in.0) = in.0, f (in.1) = in.1
g(out.0) = out.0, g(out.1) = out.1

我们需要的结果就是

CHAIN2 = f(COPYBIT) ‖ g(COPYBIT)[138]

值得注意的是，由*f*和g的定义可见，‖左边的运算对象的输出0或1与其右边运算对象的输入0或1完全是同一事件(mid.0,或mid.1)。这样就给出了图2.7所示的二进制数字的同步通信的模型，该通信是在一条连接了两个运算对象的通道上进行的。

[138] 在CHAIN2进程里，通过函数f和g的变换后，符号表或者说能触发的事件是：作用在第一个COPYBIT上的in.0, in.1，作用在第二个COPYBIT上的out.0, out.1。中间的mid.0和mid.1事件。值得非常注意的是：在CSP中，只要是出现在符号表中的事件，都是可以被外界所观察到的（Observer）。这也是为什么作用在两个进程上的并发算子得到的迹一定是两个进程的共同的行为的凝聚。否则，观察者会记录不下来复合进程的迹。

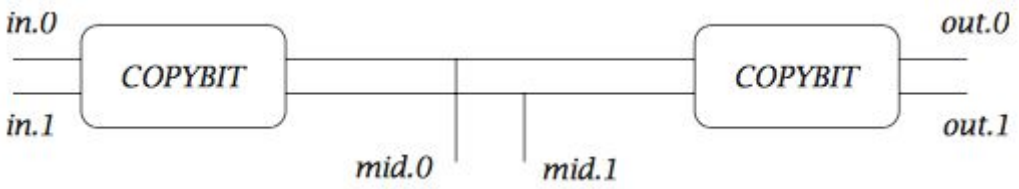


图2.7

左边的运算对象决定在连接通道上传送哪一个值，而右边的运算对象则准备好或者执行事件Mid.0，或者执行事件 Mid.1。所以确定发生哪个事件的总是输出进程。并发进程之间的这种通信方法会在第四章中推广到其通用的情形。
注意内部通信 Mid.0和 Mid.1包含在组合进程的字母表中，并且可被进程的环境观察到(甚至可受环境控制)。有时我们希望把这类内部事件忽略或屏蔽掉，但在一般情况下，做这类屏蔽可能会引入非确定性，所以我们把这个问题退后到3.5节去处理。 □

X5 我们想把计算机程序中使用的布尔变量的行为表示出来。字母表中的事件有

assign0 给变量赋0值
assign1 给变量赋1值
fetch0 当变量为0时取出变量的值
fetch1 当变量为1时取出变量的值

变量的行为与饮料机(1.1.4节X1)的行为非常相似，所以我们定义

$$\mathrm{BOOL} = f(\mathrm{DD})$$

函数f的定义留作小小的练习。要注意的是这个布尔变量在未被赋值前是不会给出它的值的。要取一个未被赋值的变量的值，将导致死锁—这恐怕是程序错误中最容易发觉的出错方式了，因为最简单的分析复盘就能将这类错误指出。 □

把函数f作用于P的树形图各枝的标记上，就得到了f(P)的树形图。由于f是1对1的函数映射，故该变换仍能保持树的原结构，而且由同一结点引出的分叉上的标记仍互不相同。例如NEWVMC的示意图如图2.8。

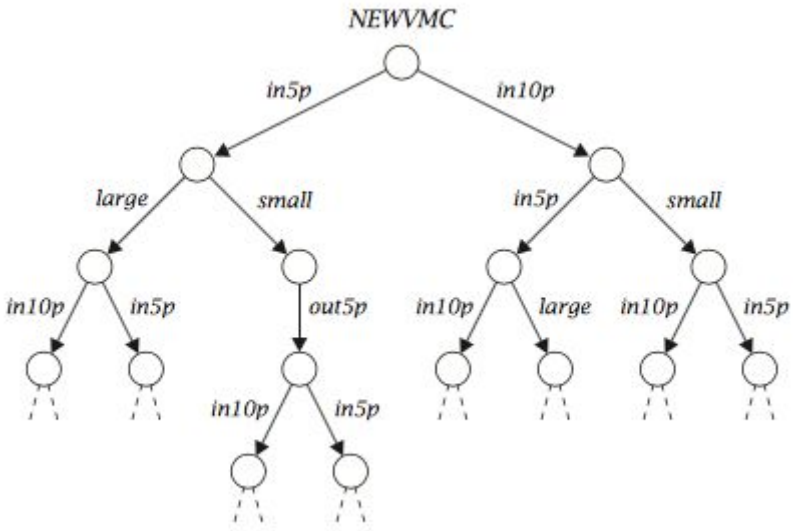


图2.8

2.6.1 法则(Laws)

在应用一对一的函数对进程进行变换符号时不会改变进程的行为结构。这一点由以下事实得出：1-1函数运算对其它一切算子均满足分配律。具体可详见下面的法则。首先我们要用到几个辅助定义如下：

$f(B) = \{f(x) \mid x \in B\}$
f^{-1}　　f的逆函数
$f \circ g$　　f和g的复合
f^*　　定义见1.9.1节

强调f是单射函数，主要是因为在法则中要用到f^{-1}。
经过符号变换后，STOP在改变后的字母表中仍什么都不做，即

L1 $f(STOP_A) = STOP_{f(A)}$

在选择的情况下，供选择的符号变了，且进程的后续行为也要相应变化，故

L2 $f(x : B \rightarrow P(x)) = (y : f(B) \rightarrow f(P(f^{-1}(y))))$

等式右边的f^{-1}可能需要解释一下。我们说过P为一函数，它要根据由集合B中选出的某事件x得到一个进程，但等式右边的变量y是集合$f(B)$的元素。对P来说，y的对应事件就是$f^{-1}(y)$，且$f^{-1}(y)$在B中(因为$y \in f(B)$)。执行完$f^{-1}(y)$之后，P的行为就是$P(f^{-1}(y))$，而且进程$P(f^{-1}(y))$的动作在f的作用下还要继续变换。

符号变换显然对并行组合有分配律，故

L3 $f(P \| Q) = f(P) \| f(Q)$

它对递归式的分配律更稍微复杂些，且使进程字母表相应地变化，有

L4 $f(\mu X: A \bullet F(X)) = (\mu Y : f(A) \bullet f(F(f^{-1}(y)))$

等右边的f^{-1}又有点让人迷惑不解。以前讲过，要使等式左边的递归式有效，要求函数F的自变量为字母表为A的进程，而且函数F的计算结果也是个进程，它与自变量的字母表相同。L4中等式右边，变量Y的变化范围是所有字母表为$f(A)$的进程，因此只有当其字母表经反变回到A之后，才能做为$F(f^{-1}(Y))$的字母表为A，应用f就将字母表变成$f(A)$，以此保证法则L4等式右边递归式的有效性。

两次符号变换的复合就定义为两个符号变换函数的复合，即

L5 $f(g(P)) = (f \circ g)(P)$

经变换符号后的进程的迹，是将原进程的迹中的各个符号都变换后得到的，故

L6 $\text{traces}(f(P)) = \{f^*(s) \mid s \in \text{traces}(P)\}$

最后一条法则的解释说明与L6类似

L7 $f(P)/f^*(s) = f(P/s)$

2.6.2 进程标记(Process labelling)

符号变换在组建由类似的进程组成进程群体时特别有用，这些类似进程并发操作，对其外部环境提供相同的服务，但是相互之间不发生联系。这意味着它们的字母表必须互不相同而且互不相交。为了做到这一点，我们把每个进程都标记上不同的名字；再把进程的每个事件也标记上与其所属进程相同的名字。做了标记的事件是个二元组l.x，其中l是标记，x是代表该事件的符号。

标记为l的进程P表示为

l : P

每当P要执行x时，l : P就执行l.x。定义l : P所用的函数是f_l

$$f_l(x) = l.x \qquad x \in \alpha P$$

而标记的定义是

l : P=$f_l(P)$

例子

X1 两台并排放置的自动售货机记为

(left : VMS) ‖ (right : VMS)

这两个进程的字母表是不相交的，而且在哪台机器上发生的事件就用那台机器的名字做上标记。万一在并排放置之前没给他们赋名，则每个事件的发生都要求它们同时参与。这样一来这两台机器就跟一台没什么区别了。由这一事实我们推出

(VMS ‖ VMS) = VMS □

有了进程标记，就可以象使用高级程序设计语言里的变量那样使用进程了，这类变量只在具体用到它们的程序块中局部地说明[139]。

X2 布尔变量的行为的模型是BOOL(2.6节X5)。有一个程序块其行为表示为进程USER。该进程对两个布尔变量b和c进行赋值和取值。这样αUSER中就包括了如下的复合事件

b.assign.0	给b赋0值
c.fetch.1	当c的值为1时取出其当前值

进程USER与它的两个布尔变量并行运行

b : BOOL ‖ c : BOOL ‖ USER

在程序USER内部，可以等效地表示

b:=false; P	为	(b.assign.0→ P)
b :=¬ c; P	为	(c.fetch.0 →b.assign.l→ P |c.fetch.l→b.assign.0 →P)

要注意的是，我们是通过允许变量在fetch0和fetch1之间做出选择的方法，以发现变量当前值的；还要注意该选择并以适当方式影响USER的后续行为。 □

在X2及以下几例中，如果先定义了赋值，可能就更方便了，例如先定义赋值

b := false

而不是直接定义命令对

b := false; P

[139] 通过使用标记，可以实现对个进程行为的实例，达到变量的效果。

在命令对中明确提到了程序的剩余部分P。单独定义赋值命令的方法将在第五章中介绍。

X3 进程USER需要两个计数变量l和m。给它们赋的初值分别为0和3。USER使每个变量增值的事件为l.down和m.down。检验变量是否为0的事件为l.around和m.around。所以就可以使用进程CT(1.1.4节X2)，但需用l和m作为适当标记。既有

$$(l : CT_0 \parallel m: CT_3 \parallel USER)$$

在进程USER内，可以等效地表示习惯使用的

(m := m + 1; P) 为　　(m.up →P)
if l = 0 then P else Q 为　　(l.around →P | l.down →l.up→Q)

注意检验变量是否为零是这样进行的：进程通过事件l.around检验l是否为零的同时，也通过事件l.down去将计数变量l减少1。这样l就要在这两个事件中选择：如果l的值为0，它就选择l.around，如果它不是0，就选择l.down。但在后一种情况里选择了l.down之后，l的值已经被减少1，故需用事件l.up立即将其恢复为原值。在下一例中，恢复原值的过程要更繁琐些。

(m:=m+l; P)　　由ADD实施

其中ADD递归地定义为

$ADD=DOWN_0$
$DOWN_i = (l.down \rightarrow DOWN_{i+1} \mid l.around \rightarrow UP_i)$
且$UP_0=P$
$UP_{i+1}=l.up \rightarrow m.up \rightarrow UP_i$

进程组$DOWN_i$把l逐步减到0从而发现l的初值，然后由进程组UP_i再把这个初值分别加到m和l上，这样既恢复了l的初值，而且还把该值加到了m上。
用一组并发的进程可以等效实现一个数组变量，这些并发进程用它们在数组的下标做为标记。

X4 进程EL的作用是记录时间in是否已经发生了。第一次发生时，EL的回答是no，以后发生时，EL的回答是yes。令

aEL = {in, no, yes}
$EL=in \rightarrow no \rightarrow \mu X \cdot (in \rightarrow yes \rightarrow X)$

这个进程可用在数组中模拟小整数的集合行为

$SET_3 = (O : EL) \parallel (1 : EL) \parallel (2 : EL) \parallel (3 : EL)$

整个数组在试用前还可再加一次标记，得到

$m : SET_3 \parallel USER$

在$a(m:SET_3)$中的每个事件是一个三元组，例如m，2.in。在USER进程内，由

$m.2.in \rightarrow (m.2.yes \rightarrow P \mid m.2.no \rightarrow Q)$

可取得以下效果

$$\text{if } 2 \in m \text{ then } P \text{ else } (m := m \cup \{2\} ; Q) \qquad \square$$

2.6.3 实施(Implementation)

要实现通用的符号变换，我们需要知道符号变换函数*f*的反函数g。我们还需要确保g的自变量超出了*f*的值域时，g会给出回答"BLEEP。符号变换的实现以2.6.1节L4为依据。令

```
change(g, P) = λx • if g(x) = "BLEEP then
        "BLEEP
        else if P(g(x)) = "BLEEP then
                "BLEEP
        else
                change(g, P(g(x)))
```

进程标记，作为一种特殊情况，它的实现就更简单了。我们用原子对cons("l, "x)。代表复合事件l.x。(l : p)的实现可以表示如下

```
label(l, P) = λ y • if null(y) or atom(y) then[140]
        "BLEEP
    else if car (y) ≠ I then
        "BLEEP
          else if P(cdr(y)) = "BLEEP then
                  "BLEEP
                        else
                        label(l, P(cdr(y)))
```

2.6.4 多重标记(Multiple labelling)

[140] 在1985年版的CSP书里，用的是LISP的逻辑符号“∨”，而非Hoare在2015年电子版用的 "or"。

标记的定义可以扩展到允许事件在集合L中任取标记l。设P是一进程，定义(L:P)是行为与P完全相同的进程，只是一旦P执行c时，(L:P)即执行事件l.c(l ∈ L, c ∈ αP)。标记l的每次选择均由(L:P)的环境独立完成。

例子

X1 一个年轻的男仆LACKEY，这个男仆服侍他的主人入座、离座，主人吃饭时他要站在身后，于是有

aLACKEY = {sits down, gets up}

LACKEY = (sits down→gets up→ LACKEY)

要教会LACKEY为五个主人服务(但每次只为其中一个服务)。我们定义

L={0,1,2,3,4}
SHARED LACKEY = (L : LACKEY)

因此，当那位男仆(2.5.3节)去度假时，这个共享的男仆可以被雇来为就餐的哲学家服务，不至于出现死锁。当然，在这段时间里，哲学家们更要挨饿了，因为每次他们中只能有一个人可以上桌就餐。
如果L中的标记不止一个，L：P的树形图与P的树形图尽管类似，但从一个节点引出的分支增多了，从这一意义上说，L:P的树要更茂盛些。如LACKEY的示意图是一根光秃秃的树干没有树杈(图2.9)。

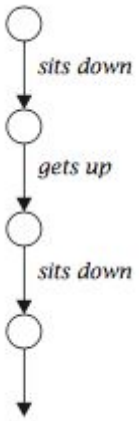


图2.9

而{0,1}：LACKEY的示意图则是一个二叉树(图2.10) SHARED LACKEY的示意图就更复杂了。

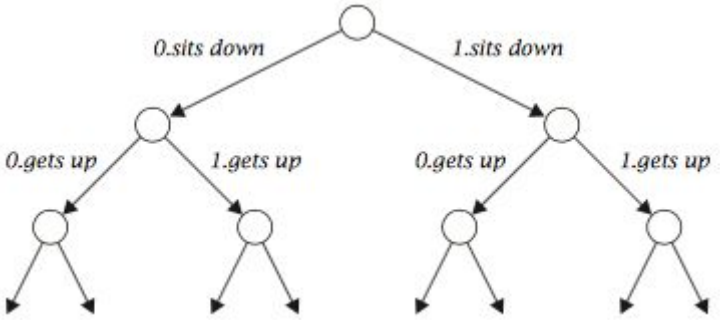


图2.10

总之，有了多重标记，假设预先知道这些标记的符号，就能使多个做了标记的进程共享某一个进程的服务。共享技术将在第六章中更充分地加以讨论。

2.7 功能描述(Specifications)

假设P和Q为并发运行的进程，并已证明P sat S(tr)和Q sat T(tr)
设tr为(P ‖ Q)的一个迹。由2.3.3节L1可知(tr ↾ αP)是P的迹，因而它也满足S，即

S(tr ↾ αP)[141]

同理，(tr ↾ αQ)为Q的迹。即

T(tr ↾ αQ)

上面的论证对(P ‖ Q)的每个迹都成立。于是我们可以推出，

(P ‖ Q) sat (S(tr ↾ αP) ∧ T(tr ↾ αQ))[142]

可以通过非正式的归纳成如下法则

L1 如果 P sat S(tr)
Q sat T(tr)
则(P ‖ Q) sat (S(tr ↾ αP) ∧ T(tr ↾ αQ))

例子

X1 (参见2.3.1节X1)

设 αP = {a, c} αQ= {b, c}

P = (a→c→P)

Q = (c→b→Q)

[141] S(tr ↾ αP)的输入(tr ↾ αP)是P的迹。因此，满足S。
[142] 同时满足两个进程P和Q的迹，就是并发进程P ‖ Q的迹。

我们要证

$$(P \parallel Q) \text{ sat } 0 \le tr\downarrow a - tr\downarrow b \le 2$$

对1.10.2节中X1的证明加以修改，容易证得

$$P \text{ sat } (0 \le tr\downarrow a - tr\downarrow c \le 1)$$
$$Q \text{ sat } (0 \le tr\downarrow c - tr\downarrow b \le 1)$$

由L1就有

$$(P \parallel Q)$$
$$\text{sat } (0 \le (tr\ \alpha P)\downarrow a - (tr\ \alpha P)\downarrow c \le 1 \ \wedge$$
$$0 \le (tr\ \ \alpha Q)\downarrow c - (tr\ \alpha Q)\downarrow b \le 1)$$
$$\Rightarrow 0 \le tr\downarrow a - tr\downarrow b \le 2 \ \ [\text{当} a \in A, \ \text{有} (tr\ A)\downarrow a = tr\downarrow a] \qquad \square$$

由sat的法则我们知道STOP满足一切能够被满足的描述，因此基于sat法则的推理无法证明一个进程死锁的不存在性[143]。在3.7节中我们将给出更强大的法则来。而在目前，要消灭死锁的危险，一个办法是如2.5.4节中我们所做的那样，逐步仔细的证明进程的无死锁，另一个办法是如2.3.1节中X1中我们所做的那样，证明用并行组合算子定义的进程与没有用这个组合算子定义的非停止进程等价。可是这类证明总要牵扯到一大串冗长乏味的代数变换。因此只要可能，我们还是应该用通用法则来证明，例如

L2 设P和Q永不停止，且$(\alpha P \cap \alpha Q)$最多包含一个事件，则$(P \parallel Q)$也永不停止[144]。

例子

X2 在X1中定义的进程$(P \parallel Q)$永不停止，因为

$$\alpha P \cap \alpha Q = \{c\}$$

符号变换的证明规则是

L3 如果P sat S(tr)

则 $f(P)$ sat $S(f^{-1^*}(tr))$

法则结论中的f^{-1^*}可能要解释一下。设tr为$f(P)$的迹，则$f^{-1^*}(tr)$为P的迹。L3中的前提说明P的每个迹都满足S，由此可得$f^{-1^*}(tr)$满足S，这正是L3的结论。

2.8 确定性进程的数学理论(Mathematical theory)

在描述刻画进程的过程中，我们陈述了很多法则，并且有时还将它们用于以些命题的

---

[143] 死锁在CSP里的表现为STOP行为。

[144] 如果超过两个共同的事件，就开始涉及同步的问题。如果没有聚合出来共同的迹或者说，没有相同的可被记录的行为，复合的并发进程就会无所适从，陷入死锁的处境。而只有一个共同的事件不会出现无所适从的问题。因为，这个事件后续的事件可以是属于任何一个进程的，不相关的事件，不需要同步的控制。

证明。如果说这些法则是经过证明了的话，证明的过程也仅限于非形式的解释和略微说明为什么我们认为它们是成立的理由，对于一位有应用数学或工程师的直觉的读者来说，那些解释也就够了，但还免不了要提些问题，譬如，这些法则是够真的成立？它们会不会相互矛盾？还有没有更多的法则？或者，它们是否完全，即有了这些法则就可证明有关进程的一切性质？是否能用更少的法则证同样多的事实呢？只有经过更深入的数学研究才能找到这类问题的答案。

2.8.1 基本定义(The basic definitions)

在建立一个物理系统的数学模型时，用物体的直接或间接可观察或测量的属性定义其相关的基本概念，是一种很好的方法。对确定性进程P，它的两个比较熟悉的属性是

αP	P原则上能执行的事件集合
traces(P)	P能实际参加的所有事件序列的集合

我们已解释过它们为何一定满足1.8.1节中的法则L6，L7，L8。下面我们考虑一个任意集合二元组(A, S)，也满足这三个法则。则这个集合对也就唯一决定了一个进程P，它的迹集是S。其构成如下

设$P^0 = \{x \mid \langle x \rangle \in S\}$

且对所有$x \in P^0$，P(x)进程定义为一个迹为$\{t \mid \langle x \rangle \hat{} t \in S\}$的进程

则有形式定义如下

$$\alpha P = A$$
$$P = (x : P^0 \rightarrow P(x))$$

另外，集合对(A,S)也可被下列方程式还原出来

$$A = \alpha P$$
$$S = \text{traces}(x : P^0 \rightarrow P(x))$$

这样在每个进程P和集合对$(\alpha P, \text{traces}(P))$之间就有了一种一一对应的关系。在数学上，有了这种关系就足以证明这两个概念是等同的，因为可以用其中一个概念定义另一个概念。

D0 一个确定性进程就是一个集合对
(A,S)
其中A是符号的任意集合
而且S是满足以下两个条件的A^*的任意子集合

C0 $<> \in S$
C1 $\forall s, t \bullet s^\frown t \in S \Rightarrow s \in S$

满足这个定义的最简单的例子就是不动作的进程，即
D1 $STOP_A=(A, \{<>\})$

而另一个最极端的例子是任何时刻动作都可以的进程
D2 $RUN_A=(A, A^*)$

现在可以形式地定义有关进程的各种算子了，即通过说明运算结果的字母表和迹是如何由运算对象的字母表和迹演变出来的。

D3 $(x:B \rightarrow (A, S(x))) = (A, \{\langle\rangle\} \cup \{\langle x \rangle s \mid x \in B \wedge s \in S(x)\})$ 其中$B \subseteq A$

D4 $(A, S)/s=(A, \{t \mid (s^\frown t \in S\})$ 其中$s \in S$

D5 $\mu X: A \bullet F(X) = (A, \bigcup_{n \geq 0} traces(F^n(STOP_A)))$ 其中F为卫式

D6 $(A, S) \parallel (B, T) = (A \cup B, \{s \mid s \in (A \cup B)^* \wedge (s \upharpoonright A) \in S \wedge (s \upharpoonright B) \in T\})$

D7 $f(A, S)=(f(A), \{f^*(s) \mid s \in S\})$ 其中f为1-1函数

当然我们还有必要证明这些定义中的等式右侧确实是进程，也就是说证明它们满足D0的条件C0和C1。幸运的是，要证明这些很容易。
但是D0还不是进程概念的完美无缺的定义，这一点在第三章中将越来越明显，因为D0没有把进程可能具有的非确定性体现出来。所以我们还需要一个更普遍、更复杂的进程定义。非确定性进程的一切法则对确定性进程也都成立。但确定性进程遵从一些额外的法则，例如

$$P \parallel P = P$$

为了避免引起混乱，在本书中我们没有引述的法则不仅可以万无一失地应用于确定性进程，而且完全适用于非确定性进程(除了2.2.1 L3A, 2.23 L1, 2.3.1 L3A, 2.3.3 L1, L2, L3。这些规则在含有进程CHAOS(3.8节)中是不成立的。

2.8.2 不动点理论(Fixed point theory)

这一节的目的是给出对于递归基本理论证明的一个轮廓，这个基本定理是：一个递归定义的进程(2.8.1节D5)是相应递归方程的一个解[145]，即

[145] 递归方程 x = F(x)。可参阅1.1.2节对递归方程的引入。在第一章，作者是假设，只要一个CSP进程的prefix是卫士的(Guarded)，则这个递归方程存在着解，即存在一个不动点，使得在那个点上，x = F(x)。

$\mu X.F(X) = F(\mu X.F(X))$

证明的方法遵从Scott的不动点理论。
首先我们需要定义描述进程之间的一种次序关系

D1 $(A, S) \subseteq (B, T) = (A = B \wedge S \subseteq T)$

两个进程如果字母表相同，而且其中之一可以做另一进程做过的每件事——也许还能做得更多——则它们可以按这种次序比较大小。这个次序关系是一种偏序关系，并满足如下关系

L1 $P \subseteq P$

L2 $P \subseteq Q \wedge Q \subseteq P \Rightarrow P = Q$

L3 $P \subseteq Q \wedge Q \subseteq R \Rightarrow P \subseteq R$

在偏序关系里，链(chain)是元素的一个无穷序列

$$\{P_0, P_1, P_2, ...\}$$

并满足，对一切i

$$P_i \subseteq P_{i+1}$$

我们定义这样一个链的极限(最小上界)为

$$\bigsqcup_{i\geq 0} P_i = (\alpha P_0, \bigcup_{i\geq 0} traces(P_i))$$ [146]

以后我们只对链式进程序列使用极限算子[147]。
如果一个偏序有最小元，而且所有的链均有最小上界，我们就说这个偏序是完全的。
字母表为A的所有进程的集合形成一个完全偏序(缩写为c.p.o.)，因为它满足以下法则

L4 $STOP_A \subseteq P$　只要$\alpha P = A$

我们称那个解是：$\mu X: A \bullet F(X)$。在2.8节里，作者开始试图首先形式定义一个进程和变换可以通过(A, S)来定义。然后在这个基础上，利用数学的集合论，代数的手段来推演。

[146] $\bigsqcup_{i\geq 0} P_i$本身通过(A, S)的方式定义为一个进程。 $\bigcup_{i\geq 0} traces(P_i)$意味着这个进程的迹是所有Pi迹的并集。覆盖了一切可能的行为。

[147] 即需要满足$P_i \subseteq P_{i+1}$关系的进程序列。

L5 $P_i \subseteq \bigsqcup_{i\geq 0} P_i$

L6 $(\forall\ i\geq 0 \bullet P_i \subseteq Q) \Rightarrow (\bigsqcup_{i\geq 0} P_i) \subseteq Q$

更进一步，μ算子(2.8.1节D5)可用极限重新正式定义为

L7 $\mu X : A \bullet F(X) = \bigsqcup_{i\geq 0} F^i(STOP_A)$

由一个c.p.o.到另一个c.p.o.(或到其自身)的函数F，如果对所有链的极限算子都满足分配律，即若$\{P_i \mid i\geq 0\}$是一个进程链，

$$F(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} F(P_i)$$

则我们说它是连续的。
(因为对所有P和Q有$P \subseteq Q \Rightarrow F(P) \subseteq F(Q)$，因此所有连续函数都是单调的，因此前一方程的等式右端也是一个升链的极限[148]。)
含多个自变量的函数G如果对其每个自变量分别连续，则我们定义G也是连续的。例如

对一切Q

$$G((\bigsqcup_{i\geq 0} P_i), Q) = \bigsqcup_{i\geq 0} G(P_i, Q)$$

对一切Q

$$G(Q, \bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} G(Q, P_i)$$

连续函数的复合仍是连续的；并且任意个数的连续函数作用于任意多个变量的任意组合上，所构成的表达式对其每个变量仍是连续的，例如，设G, F和H为连续函数，则

$$G(F(X), H(X,Y))$$

对X是连续，即

$$G(F(\bigsqcup_{i\geq 0} P_i), F(\bigsqcup_{i\geq 0} P_i), Y) = \bigsqcup_{i\geq 0} G(F(P_i), H(P_i,Y)) \qquad \text{对一切Y}$$

D3到D7中定义的所有算子(除了/)在上述意义下都是连续的

L8 $(x : B \rightarrow (\bigsqcup_{i\geq 0} P_i(x)) = \bigsqcup_{i\geq 0} (x : B \rightarrow P_i(x))$

---

[148] $\bigsqcup_{i\geq 0} F^i(STOP_A)$是一个chain的limit。

L9 $\mu X : A \bullet F(X, \bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} \mu X : A \bullet F(X, P_i)$ 若F连续

L10 $(\bigsqcup_{i\geq 0} P_i) \parallel Q = Q \parallel (\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0}(Q \parallel P_i)$

L11 $f(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} f(P_i)$

因此如果F(X)是单纯由这些算子构成的一个表达式，则它对X也是连续的。
现在我们可以证明基本不动点定理了

$F(\mu X: A \bullet F(X))$
$= F(\bigsqcup_{i\geq 0} F^i(STOP_A))$ 根据μ定义
$= \bigsqcup_{i\geq 0} F(F^i(STOP_A))$ 因为F是连续的
$= \bigsqcup_{i\geq 1} F^i(STOP_A)$ 根据F^{i+1}定义[149]
$= \bigsqcup_{i\geq 0} F^i(STOP_A)$ 因为$STOP_A \subseteq F(STOP_A)$[150]
$= \mu X : A . F(X)$ D5 μ定义

这个证明只依赖于F为连续函数。而F是卫式，只是保证方程有唯一性[151]。

2.8.3 唯一解(Unique solutions)

在这一节里我们将更形式地处理1.1.2节中给出的论证过程，即证明进程的卫式递归方程只有一个唯一解。证明时，我们先要弄清楚具备解的唯一性的几个更为一般的条件。为简单起见，我们只讨论单一方程的情况，这种处理方法可以很容易地推广到联立方程组。
设P为一进程，n是个自然数，我们定义$(P \upharpoonright n)$仍为一个进程，在执行它的前n个事件时，其动作和P一样，然后它就停止不动；形式定义为

$$(A, S) \upharpoonright n = (A, \{s \mid s \in S \wedge \#s \leq n\})$$

由定义引出
L1 $P \upharpoonright 0 = STOP$
L2 $P \upharpoonright n \subseteq P \upharpoonright (n+1) \subseteq P$
L3 $P = \bigsqcup_{n\geq 0} P \upharpoonright n$
L4 $\bigsqcup_{n\geq 0} P_n = \bigsqcup_{n\geq 0}(P_n \upharpoonright n)$

---

[149] $\bigsqcup_{i\geq 0} F(F^i(STOP_A)) = \bigsqcup_{i\geq 0} F^{i+1}(STOP_A)$。因此按照定义，是$\cup_{i\geq 0}$ $traces(F^{i+1}(STOP_A))$，等于$\cup_{i\geq 1}$ $traces(F^i(STOP_A))$，即从$F(STOP_A)$开始。
[150] 这一步很精妙。因为$STOP_A \subseteq F(STOP_A)$，所以，我们补齐这个$STOP_A$的迹，不会影响$\bigsqcup_{i\geq 1} F^i(STOP_A)$，是等价的。因此，我们得到$\bigsqcup_{i\geq 1} F^i(STOP_A) = \bigsqcup_{i\geq 0} F^i(STOP_A)$。
[151] 唯一的不动点。

设F为进程到进程的一个单调函数，如果对所有X，均有

$$F(X) \ (n+1) = F(X \ n) \ (n+1)$$

则说F是构造性的。其含义是F(X)的前n+1步的行为仅由X的前n步行为决定；
因此如果$s \neq <>$，就有

$$s \in traces(F(X)) = s \in traces(F(X \ (\#s - 1)))$$

前缀运算就是一种最基本的构造性函数，因为

$$(c \rightarrow P) \ (n+1) = (c \rightarrow (P \ n)) \ (n+1)$$[152]

通用选择算子也是构造性的，因为有

$$(x:B \rightarrow P(x)) \ (n+1) = (x:B \rightarrow P(x) \ n)) \ (n+1)$$

恒等函数I不是构造性的，因为

$$I(c \rightarrow P) \ 1 = c \rightarrow STOP$$
$$\neq STOP$$
$$= I((c \rightarrow P) \ 0) \ 1$$

现在我们可以通过构造函数来证明唯一解的基本定理

L5 设F是一构造性函数。方程

$$X = F(X)$$

对X只有一个唯一解

证明　设X为一任意解。首先我们用归纳法证明引理

$$X \ n = F^n(STOP) \ n$$

初始情形。$X \ 0 = STOP = STOP \ 0 = F^O(STOP) \ 0$
归纳：

$X \ (n+1)$	因为X=F(X)
$= F(X) \ (n+1)$	F是构造性的
$= F(X \ n) \ (n+1)$	F是构造性的
$= F(F^n(STOP) \ n) \ (n+1)$	假设条件[153]

[152] $c \rightarrow (P \ n)$中c事件是第1步。因此我们只要再取n步，就构成了n+1步。
[153] 自然归纳法，假设为n时成立。

$= F(F^n(STOP))\ \ (n+1)$
$= F^{n+1}(STOP)\ \ (n+1)$ 由F^n定义

下面我们回到主定理去

$X = \bigsqcup_{n\geq 0} (X\ n)$ 由L3
$= \bigsqcup_{n\geq 0} F^n(STOP)\ n$ 由引理
$= \bigsqcup_{n\geq 0} F^n(STOP)$ 由L4
$= \mu X \cdot F(X)$ 由2.8.2节L7

因此方程$X=F(X)$所有的解都等于$\mu X.F(X)$，换句话说，$\mu X.F(X)$是方程唯一解。
如果我们能清楚地分辨出哪些函数是构造性的，哪些不是，则我们这个唯一解的定理的作用将大大增加。让我们定义一个非破坏性函数G[154]，它满足的条件是

$G(P)\ n = G(P\ n)\ n$ 对所有n和P

字母表变换就是非破坏性的，因为

$f(P)\ n = f(P\ n)\ n$

恒等函数，也是非破坏性的。任何构造性的单调函数同时也是非破坏性的。但是由于

$((c\rightarrow c\rightarrow STOP)\ /<c>\ 1 = c \rightarrow STOP$
$\neq STOP$
$= (c \rightarrow STOP)\ /<c>$
$= (((c\rightarrow c\rightarrow STOP\ 1)/<c>)\ 1$

所以后继算子是破坏性的。
非破坏性函数(如G和H)的任意复合仍为一非破坏性函数，因为有

$G(H(P))\ n=G(H(P)\ n)\ n=G(H(P\ n)\ n=G(H(P\ n)\ n$

更为重要的一点是，一个构造性函数与多个非破坏性函数的复合仍是构造性的。因此，如果函数F,G...，H均为非破坏性的，且它们当中有一个是构造性的，则它们复合的结果

$F(G(...(H(X))...))$

是X的一个构造性函数。
以上结论可很容易地推广到含多个自变量的函数的情形。譬如说，进程的并行组合对两个自变量都是非破坏性的，因为

---

[154] 非破坏性函数是函数语言的一个概念，指的是不改变操作对象的某种性质，例如，经过变换后，对象之前的某种属性还保持着。

(P ‖ Q) n = ((P n) ‖ (Q n)) n

假设E为含有进程变量X的一个表达式。如果对X在表达式E中每个出现的地方，均有一个构造性函数作用于它，而且不存在作用于它的破坏性函数，我们说E关于X是卫式(Guarded)的。例如，以下表达式关于X是卫式

(c→X | d→f(X ‖ P) | e→(f(X) ‖ Q ‖ ((d→X) ‖ R

由此我们可以得到重要结论：函数的构造性可以仿照下面卫式表达式的语法条件定义

D1 凡仅由并发算子、符号变换算子及通用选择算子构成的表达式，是保持卫式特性的。
D2 不含X的表达式，是关于X的卫式。
D3 如果对所有x，P(X, x)是保持卫式特性的，则下面的通用选择是关于X的卫式。
(x:B→P(X,x))
D4 如果P(X)关于X是卫式，则符号变换f(P(X))也是关于X的卫式。
D5 如果P(X)和Q(X)都是关于X的卫式，则它们构成的并发系统P(X) ‖ Q(X)关于X仍是卫式。

最后，我们得到以下结论

L6 如果表达式E是关于X的卫式，则方程
X=E
有唯一解

第三章　非确定性(Nondeterminism)

第四章　通信(COMMUNICATION)

第五章　顺序进程(SEQUENTIAL PROCESSES)

第六章　共享资源(SHARED RESOURCES)

第七章　讨论(DISCUSSION)

文献(BibliogrαPhy)

索引(Index)