925—936

# 程序设计法则

C. A. R. Hoare等

**摘要**：本文对 Dijkstra 的不确定性顺序程序设计语言，给出完整的一组代数法则，用 Scott 的域论把迭代和递归解释为连续泛函的不动点。提出了类似于最弱前置谓词的演算，以帮助人们从程序的规格说明中推导程序。

有些计算机科学家已经放弃了指导传统过程性程序设计的合理法则的研究。他们往往推荐使用函数程序设计[5]或逻辑程序设计[10]。我们在本文中要证实这样一种观点，即传统的过程性程序是数学表达式，它们象数学、工程或自然科学的任何其他分支一样，服从一组丰富多彩的法则。

而且，一组广泛的代数法则，起到了对有关数学记号，特别是程序语言进行有用的（公理化）形式定义的作用—原先由 Igarishi[8] 提出。这些代数法则提供语言的使用者和设计语言的数学工程师之间的界面。当然，数学家也应该设计语言的模型，检验这些法则的完备性和一致性，为程序的规格说明以及正确性证明提供框架。

下面是一些大家熟知的应用于实数乘法的算术法则：

（1）乘法是对称的，即

$x \times y = y \times x$，对任意的数$x$和$y$。

习惯上引用法则时省略了"对有关集合的所有$x$和$y$"这个短语。

（2）乘法是结合的，即

$x \times (y \times z) = (x \times y) \times z$。

习惯上对结合的操作省略括号，记作$x \times y \times z$。

（3）零乘以任何数等于零，即

$0 \times x = 0$。

称数0为乘法的不动点或零元。

（4）1乘以任何数仍等于该数，即

$1 \times x = x$。

称数1为乘法的单位元或么元。

（5）除法是乘法的逆，即

$y \times (x/y) = x$，只要$y \neq 0$。

有关乘法和除法的另一条法则是

$z/(x \times y) = (z/x)/y$，只要$y \neq 0$且$x \neq 0$。

（6）乘法对加法是分配的，即

$(x+y) \times z = (x \times z) + (y \times z)$。

通常省略上述方程右边的括号，习惯上，分配的算子更靠近约束于对之分配的算子。

（7）乘以非负数的乘法，在它对另一个运算对象保序的意义下单调，即

$x \leqslant y \Rightarrow x \times z \leqslant y \times z$，只要$z \geqslant 0$。

如果减小两个乘法因子中的一个，则乘积的值不会增加。

（8）乘法在保持任何收敛数列极限的意义下连续，即

$(\lim_{n\to\infty} x_n) \times y = \lim_{n\to\infty} (x_n \times y)$，只要$x_n$收敛。

（9）如果定义

$x \cap y = x$和$y$的较小者

$x \cup y = x$和$y$的较大者，

则有下列法则：

$x \cap y = y \cap x$

$(x \cap y) \geqslant z \equiv x \geqslant z \wedge y \geqslant z$

$(x \cup y) \leqslant z \equiv x \leqslant z \wedge y \leqslant z$

$x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$。

数学家或工程师本质上熟悉所有这些法则，频繁而直观地使用这些法则。应用数学家、科学家或工程师也熟悉相应的许多自然法则，清晰地使用它们，以寻找解决其他的疑难问题。对这些法则愚昧无知的人，没有

资格从事技术工作。那末，什么是为软件工程职业提供形式基础的程序设计法则呢？

许多程序员甚至连一条法则都不会引用。容忍不可靠程序后果的许多程序员，可能声称程序员不必注意任何法则。这种指责是不公平的，也是不适当的。程序设计法则像算术法则一样。它们描述以合适记号表示的程序的性质，正如算术法则描述以十进制记号表示的数的性质一样。确保程序服从一组有用的、合理的、可清晰陈述的适当法则，是程序语言设计者和实现者的责任。

计算机硬件的设计者有类似的责任，他们须确保其运算装置服从诸如乘法单调性(7)那样的法则。令人遗憾的，有些计算机设计却不能这样。同样的，许多流行的程序语言，甚至不能服从诸如本文列出的那些最显然的法则。Occam[11]是专门设计的服从那种数学法则的首批语言之一。本文使用的语言比Occam简单，本文的语言省略了Occam的通讯和并发。我们也要注意到，在更复杂的语言中有些法则不成立。

## 语　言

为了将数学法则公式化，有必要介绍描述程序的一些记号。我们基于文献[4]中介绍的语言，使用特别简洁的适合于我们目的的（程序语言）记号。有三种基始的语句和五种构成较大语句（程序）的方法。SKIP语句以Ⅱ表示。该语句成功地执行终止，不发生任何变化。

ABORT语句以⊥表示。它对机器执行的行为或失误毫无限制，机器可以做任何事情，也可以不做任何事情；特别是，它可能不终止。于是，⊥表示机器故障的行为或胡乱地运行程序的行为。ABORT最重要的性质在于，没有人想使用这个程序或编写这个程序。ABORT象数学系统的奇异性，有才能的工程师无论如何必须避免这种系统。为了证明这种错误不存在，人们必须使用承认它存在的数学理论。

赋值语句中，设x为不同变量名的表，E为具有相同个数的表达式的表。执行赋值语句x:=E，意味着先计算E中所有表达式（所有变量都取新近赋于的值），然后把各表达式值赋给表x中相同位置的变量。这是众所周知的多重赋值或同时赋值。假设表达式的计算无副作用，规定x中变量的值直到整个计算完成之前不会改变。为简单起见，还假设所有表达式的所有运算，对其变元的值都是有定义的。因此，表达式的计算总是成功地终止。这种假设在无定义的表达式一节中会放松。

顺序复合。如果P和Q是程序，则(P;Q)也是程序。该程序首先执行P，如果P不终止，则（P；Q）也不终止。如果并且当P终止，就执行Q，然后当Q终止时，（P；Q）也终止。

条件。如果P和Q是程序，而b是布尔表达式，则（P◁b▷Q）也是程序。该程序首先计算b，如果b为真，则执行P；如果b为假；则执行Q。更通常的条件记号是**if b then P elseQ**。我们之所以选择中缀记号◁b▷，是因为它简化了有关代数法则的表达式。

不确定性。如果P和Q是程序，则(P∪Q)也是程序。该程序或者执行P或者执行Q。它们之间的选择是任意的。程序员故意推迟这种决定，有可能推迟到程序开发的后面阶段，甚至也有可能把这种决定交给执行该程序的机器。

迭代。如果P是程序，而b是布尔表达式，则(b*P)也是程序。该程序首先计算b，如果b为假，则成功地执行终止，且不发生任何变化；如果b为真，则机器接着执行P；(b*P)。习惯上的迭代记号是**while b do P**。

递归。设x是递归定义的程序名，（包含名x出现的）F(x)是定义其行为的程序文本，则μx·F(x)是行为和F(μx·F(x))一样的一个程序，即所有递归出现的程序名可以用整个递归程序去替换。这种事实用下列法则

表示：

$$\mu x\cdot F(x)=F(\mu x\cdot F(x))。$$

在数学上，相等的替换总是允许的，而且可以无限地重复

$$\mu x\cdot F(x)=F(\mu x\cdot F(x))$$
$$=F(F\cdot(\mu x\cdot F(x)))=\cdots。$$

实质上这是计算机执行递归定义程序的方法。当然，迭代仅仅是递归的特殊情况：$b*P=\mu x\cdot(P;x)\lhd b\rhd \mathrm{I}$。迭代比一般递归要简单，更熟悉一些，因此，值得分开进行处理。本文不去阐述如何理解递归。

作为例子，我们给出一个计算非负数x除以正数y的商q以及余数r的程序。该程序提供了一个选择的方法，当y=0时，其中一个选择终止。

$$(q,\ r:=0,\ x;\ (r\geqslant y*(q,\ r:=q+1,\ r-y)))$$
$$\cup((q,\ r:=x\div y,\ x\ \mathrm{rem}\ y)\lhd y\neq 0\rhd q:=0)。$$

似乎不大熟悉对计算机程序的子表达式自由地使用括号，但是，把程序作为数学公式处理时，却是很自然的。有时省略括号很方便，只要能按下列优先级规则重新插入括号。

，约束级别最高　:=　*　;

$\lhd b\rhd$　$\cup$约束级别最低

正常算术操作的约束优先级比其他操作更高。于是，上例可以不要任何括号。

本文的语言记号能以Dijkstra语言的警戒命令去定义，

$$P\cup Q=\mathbf{if}\ \mathrm{true}\to P\ \square\ \mathrm{true}\to Q\ \mathbf{fi},$$

$$P\lhd b\rhd Q=\mathbf{if}\ b\to P\ \square\ \neg b\to Q\ \mathbf{fi},$$ 其中$\neg b$是$b$的反，

$$b*p=\mathbf{do}\ b\to p\ \mathbf{od}。$$

反过来，警戒命令也能以本文的记号去定义。例如：

$$\mathbf{if}\ b\to P\ \square\ c\to Q\ \mathbf{fi}=((P\cup Q)\lhd c\rhd P)\lhd b\rhd(Q\lhd c\rhd\perp),$$

$$\mathbf{do}\ b\to P\ \square\ C\to Q\ \mathbf{od}=(b\vee c)*(\mathbf{if}\ b\to P\ \square\ c\to Q\ \mathbf{fi})。$$

这样，以Dijkstra语言表示的任一程序，能变成本文语言的程序，不过，可能引起警戒命令代码长度的大量扩张。本文语言表示的任一程序（但把迭代作为递归的唯一形式），能变成Dijkstra语言的程序。除了本文的记号使某些法则表达起来更简洁之外，我们不去理会记号的优劣之处。

本文对程序语言语句的描述是很不形式化的，且有意不给出程序概念的数学定义。有经验的程序员能理解我们的意图。把算术法则的研究，推迟到对传统的基本定义（例如，基数是一类有关一一对应函数的集合）的研究之后，显然是不合适的。的确，甚至数学家，也是通过研究其性质而不是研究其形式定义，去获取深刻清晰的更好理解的概念。本文的目的之一是想指出代数法则也能精巧地解释程序语言的设计细则。

**技术注释**　通常，在理论教科书中，具体记号（例如，以各种基表示的数字）和它们代表的抽象对象（例如，自然数）之间是有严格区别的。本文中，程序这个词表示抽象概念，粗略地说，等价于以各种初始状态执行其文本的可能的可观测作用的值域。当要强调由程序变换处理的特定具体文本时，称之为程序文本。但一般来说，正象大多数应用数学家和工程师那样，我们对这些形式区别采取宽松的态度。

可以把本文的法则作为程序语言完整的形式代数规格说明。这些法则足够强，能把不包括递归的每一个程序化归到标准形式。我们对标准形式，定义一种自然的偏序，然后，使程序等同于该偏序下的理想数。这种理想数构造，使人联想起有点象某种有理数集合的Dedekind实数定义。

理论工作者，通常把不确定性程序等同于一个关系，即所有这样的机器状态的序偶集合。如果程序从该序偶的第一个状态开始执行，则程序终止于该序偶的第二个状态。例如，I是恒等关系，ABORT是全关系。然而，这种定义对本文的语言不正确，因为它不符合几个法则（例如，顺序复合的第3条法则和极限的第4条法则）。为了使这些法则也成立，有必要把程序定义为具有专门性质关系的特殊子集。例如，

程序是一个全关系，

每个状态的象点是有限的或无穷的。

必须用只有由不终止程序才能"到达"的虚构

"无限大状态"来表示不终止。同样的，如果虚构状态在状态的象点中，则该象点是无穷的。本文语言的所有各种操作都保持这些性质。进一步的细节请见文献[1]。

虚构状态既引起争论又引起复杂性，它们和实际程序员的需要无关。对实际工程师来说，数学法则也比基础研究中所构造的计算模型关系更密切。

模型的另一个问题是，当程序设计语言扩充时（例如，引入输入/输出语句），模型要发生很大变化。不过，正象实数算术共享整数算术的许多性质那样，为了保持简单语言服从的大多数法则的有效性，人们能够（而且或许应该）设计这样的扩充。文献[11]中已经为Occam进行过扩充。

**小结** 本文提出的法则，不仅适用于上述程序语言记号表示的具体程序，而且大多数法则也适用于能力更强的一类记号表示的程序规格说明。给出另外的法则帮助从规格说明到设计以及从设计到程序的逐步开发。事实上，我们要研究四类对象系列，其中每一类对象系列包含该对象系列的前承对象，并且服从所有的或大多数的相同法则。

（1）有限程序可以用不包括迭代和递归的程序语言记号所表达。"代数法则"一节中给出有限程序的法则。这些法则足够强，能把每一个有限程序文本化归到简单的标准形式。

（2）具体程序可以用包括递归在内的完整程序语言记号所表达。

（3）抽象程序可以用程序设计记号，另外加上表示一致收敛的程序序列的极限操作所表达。它们涉及论域的概念和法则，在"域的性质"一节中对之进行解释。

称上述三类对象为程序。它们都满足由"代数法则"和"域的性质"解释的程序设计法则。

（4）余下的一类对象称为规格说明对象。这是更一般的一类对象，因为对它们可以表示的记号，没有什么限制。数学或逻辑上的任一有定义的操作，甚至包括取反，都可以在规格说明中自由地使用。在设计和程序的开发阶段为满足其规格说明，适用于规格说明的法则是有用的。人们为表达规格说明而更自由地使用记号换来的是有可能且容易书写任一程序都不能满足的规格说明。另一个潜在的困难是规格说明不服从一些法则，而程序却服从这些法则，

这些对象类之间的区别似乎是复杂的，但实际上，它们和众所周知的不同类的数之间的区别一样简单。

（1）有限程序可以和有理数相比。代数法则允许把所有的算术表达式，化归到容易建立其等式的互素整数之比。

（2）具体程序象代数实数，它们在受限的记号（象多项式方程的解）框架中有定义。它们构成了可数集。

（3）抽象程序象实数，它们享有收敛的序列有极限这样的性质。对诸如演算等许多目的，实数远比代数数在论证上方便。它们形成了不可数集。

（4）规格说明可以和复数相比，其中许多操作（例如平方根）是全函数。起初，人们难于接受虚数概念，因为虚数不能以一维实数连续表示。而且，虚数不服从诸如$x^2 \geqslant 0$这样的法则。不过，甚至对最终结果是实数这样的问题，在定义、演算以及证明中也要使用虚数。同样，即使规格说明决不被计算机执行，但它们在需求分析以及程序开发中是有用的（甚至是必要的）。

报导一下用这些法则来帮助开发实际规模的正确程序的实例研究似乎更好。不巧，这是不可能的。编写一个实际规模的程序，需要比本文介绍的初等代数更多的面向应用的数学知识。人们不能期望通过桥梁设计的实例研究去阐明算术法则的作用。程序设计法则象算术法则一样，是十分广泛而又相当浅显的。人们学习这些代数法则，直观地使用这些法则，就象人们学习和使用外语语法规则那样。

## 代数法则

本节给出大约30条无迭代和无递归程序服从的代数法则。这些法则相当有效，足以把每一个有限程序化归到简单的标准形式，它们可用于检验任意二个文本是否表示相同的程序。

对变量值域，我们采用下述约定：

P、Q、R　代表程序。

b、c、d　代表布尔表达式。

e、f、g　代表单个表达式。

E、F、G　代表表达式表。

x、y、z　代表对之赋值的程序变量表，其中在组合表x、y、z中，变量不会多次出

现。

而且，x、y、z分别和E、F、G长度相同。

**技术注释**

(1) 短语"P代表程序"可能有二义性。P代表以某种程序记号用几种不同的方式表示的某个数学对象吗？或者，P代表一个程序文本，该文本在使用包含P的法则之前必须替换P吗，回答是没有关系的。考虑类似的数学方程

$$n+m=m+n。$$

我们把变量n、m正常地看成代表实际的数，独立于表示它们的方式。但是，如果用数字（例如，三进制数字序列）替换n和m，则该法则恒为有效。而且，如果以任意其他的算术表达式（例如，以$2\times p^2+q$替换n得到$2\times p^2+q+m=m+2\times p^2+q$）替换n，该法则依然恒有为效。

(2) 对体现程序设计代数法则的方程，以及对断定以不同方式编写的二个程序相等的方程，我们要问这些方程的含义是什么？显然，二个相等的程序文本，其含义不会不等。我们把程序含义粗略地理解为，以计算机的每一初始状态，执行程序的可能作用。例如，下列方程为真：

$(x:=037)=(x:=37)$

$(x:=y\times y+2\times y\times z+z\times z)=(x:=(y+z)\times(y+z))$

$(x:=0;\ x:=1)=(x:=1)$

在上述各种条件下，两个程序都是相等的，即使其中一个执行速度可能比另一个要慢一些。允许效率较低的程序文本，由相同程序的效率更高的程序文本替代，这反映了对程序的执行速度进行抽象。这样，程序设计法则可用于保持正确性的程序变换，或用于阐明自动代码优化。这就是设计、实现和使用具有数学性质语言的实际报酬。

(3) 我们在几个场合下，把新的记号引入这些法则，这些记号不在程序语言中，但是它们描述了所表示的程序。这种方法十分类似于数学处理。例如，x的一个多项式语法上可定义为这样一个文本，它或者是一个常数，或者是x乘以一个多项式再加上一个常数。于是，下面是一些多项式：

$0,\ 7,\ 7\times x+4,\ (7\times x+4)\times x+17,\ \cdots$

然而，人们习惯上使用$\Sigma$和幂$\Sigma a_i\times x^i$描述多项式，即使这些记号不在多项式的形式语言中。

**不确定性** 指导不确定选择的法则适用于各类选择。下列方程断定由左右两边描述选择的整个值域都相等。当然，左边得到的一个特殊选择可能和右边得到的特殊选择不一样。从相同的文本中得到的两种选择也成立。因此，在不确定机制中，两次执行相同程序文本未必得到相同的结果。

(1) 很显然，以什么样的次序进行选择不会引起任何的差异。"茶或咖啡？"与"咖啡或茶？"是一样的。

$P\cup Q=Q\cup P$(对称性)

(2) 三个事件（茶，咖啡，或可可）之间的选择，首先看作是一个事件与另外两个事件之间的选择，然后（如果有必要）看作是另外两个事件之间的选择。以什么方式把它们编成一组是无关紧要的。

$P\cup(Q\cup R)=(P\cup Q)\cup R$ （结合性）

(3) 一个事件与其本身之间的选择等于根本不选择(Hobson选择）。

$P\cup P=P$ （幂等性）

(4) ABORT语句完全允许任意的行为，因此，对它进一步的选择变得没有差别。

$\perp\cup P=\perp$ （零$\perp$）

这条法则有时称为Murphy法则，它表示"如果有错，则一直有错"；左边描述可能错误运行（或象P的行为）的机器，而右边描述将继续错误运行的机器。但这条法则真正的含义实际上比这还要坏：程序$\perp$未必总是错误运行—仅当程序$\perp$这样做造成最大灾难时！法则(4)的丰富实践经验意味着应把它作为计算机程序设计的第一条法则。

n+1个事件之间的选择，用下标记号简记成

$$\bigcup_{i=0}^{n}P_i=P_0\cup P_1\cup\cdots\cup P_n$$

下标记号的确不在程序语言中。不过，它对法则公式化有用，因为下标记号能把单一的法则应用于任意数目的选择上。在实际程序文本中，每次使用该法则时，应把选择表完全列出来。

**条件** 对一给定布尔表达式b，选择操作$\lhd b\rhd$规定左右两个事件之间的一个选择。头

二条法则明确表示。如何按b的真假值使用这种选择的准则：

(1) $P \lhd \text{true} \rhd Q = P$。

(2) $P \lhd \text{false} \rhd Q = Q$。

和∪一样，条件操作也有幂等性、结合性：

(3) $P \lhd b \rhd P = P$

(4) $P \lhd b \rhd (Q \lhd b \rhd R) = (P \lhd b \rhd Q) \lhd b \rhd R$。

而且，条件操作还满足不太熟悉的法则

(5) $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$，其中$\neg b$是$b$的反；

(6) $P \lhd c \lhd b \rhd d \rhd Q = (P \lhd c \rhd Q) \lhd b \rhd (P \lhd d \rhd Q)$，其中$c \lhd b \rhd d$是条件表达式，如果$b$为真，则取值$c$，否则，取值$d$。

(7) $P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$。

上述这些法则可分$b$为真和$b$为假两种情况进行检验。例如，法则(7)表明中间的运算对象$Q$，在两种情况下都选择不到。

如果条件的一个运算对象提供了$P$和$Q$之间的不确定选择，则在条件的计算之前或计算之后进行这种选择都是无关紧要的，因为条件的值不受选择影响。

(8) $(P \cup Q) \lhd b \rhd R = (P \lhd b \rhd R) \cup (Q \lhd b \rhd R)$。

对$\lhd b \rhd$的右运算对象，也可推导出类似的法则。

(9) $R \lhd b \rhd (P \cup Q) = (R \lhd b \rhd P) \cup (R \lhd b \rhd Q)$。

证明　左边$= (P \cup Q) \lhd \neg b \rhd R$

(根据法则(5))

$= (P \lhd \neg b \rhd R) \cup (Q \lhd \neg b \rhd R)$

(根据法则(8))

$=$右边　(根据法则(5))。□

象这种对∪分配的操作称为析取操作。

不改变布尔表达式b值的任一操作，将对$\lhd b \rhd$分配。不确定选择是一个例子。在b的计算之前或计算之后进行这种选择是无关紧要的。

(10) $(P \lhd b \rhd Q) \cup R = (P \cup R) \lhd b \rhd (Q \cup R)$。

基于相同的理由，条件$\lhd c \rhd$对另一个不同的条件也是分配的。

(11) $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)$。

使用这些法则，我们可证明下述定理：

(12) $(P \lhd c \rhd R) \lhd b \rhd (Q \lhd d \rhd R) = (P \lhd b \rhd Q) \lhd c \lhd b \rhd d \rhd R$。

证明　右边$= ((P \lhd b \rhd Q) \lhd c \rhd R) \lhd b \rhd ((P \lhd b \rhd Q) \lhd d \rhd R)$　(根据法则(6))

$= ((P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)) \lhd b \rhd ((P \lhd d \rhd R) \lhd b \rhd (Q \lhd d \rhd R))$

(根据法则(11))

$=$左边(根据法则(7)和(4))。□

**顺序复合**

(1) 顺序复合是结合的；为了依次执行三个动作，或者先执行一个动作，再接着执行其他二个动作，或者先执行前二个动作，再接着执行第三个动作。

$P;\ (Q;\ R) = (P;\ Q);\ R$　(结合性)

(2) 在程序$P$之前或之后执行语句Ⅱ(它什么也不改变)不会改变程序$P$的作用。

$(Ⅱ;\ P) = (P;\ Ⅱ) = P$　(么元Ⅱ)

(3) 在程序$P$之前或之后执行语句⊥(它可能做任何事情)导致程序P可能做任何事情—它甚至可能象P的行为！

$(\bot;\ P) = (P;\ \bot) = \bot$

法则$P;\ \bot = \bot$表示，在$P;\ \bot$到达⊥之前我们不能观察P所做的任何事情。这条法则对程序与它的环境(例如输入/输出)有交互作用的语言不成立。

这条法则的非形式解释是弱的。或许把它解释为道德(moral)法则更好。程序员有责任避免程序⊥。同样地，顺序程序员有责任既避免$(\bot;\ P)$又避免$(P;\ \bot)$。划清人们要避免的程序之间的界线是毫无意义的。

(4) 先在P和Q之间作选择的机器，然后在选择到的事件终止时再执行R等价于开始时就选择：不是执行$P$再执行R，就是执行Q再执行R，

$(P \cup Q);R=(P;\ R) \cup (Q;\ R)$。

同理，复合对∪右分配，即$R;\ (P \cup Q)=(R;\ P) \cup (R;\ Q)$。总之，顺序复合是析取操作。

(5) 条件的计算对后来发生的事情是没有影响的。因此，；对条件操作左分配：

$(P \lhd b \rhd Q);R=(P;\ R) \lhd b \rhd (Q;\ R)$。

然而，；对条件操作却不是右分配的。因此，一般来说，$R;\ (P \lhd b \rhd Q)=(R;\ P) \lhd b \rhd (R;\ Q)$不成立。上式左边，b在执行R之后被计算；而右边，b在执行R之前被计算，一般来说，先前执行的R会改变b的值。

**赋值** 当表达式的变量用表示这些变量值的表达式或常数替换时，原表达式的值不变，这是一条数学法则。如果$E(x)$是表达式表，而F是变量x的值表，则$E(F)$是E的一个复写，其中x中每个变量的每次出现都用F表中相同位置的表达式替换。

(1)赋值的第一条法则，允许对相同变量的两个相继赋值语句合并：

$(x:=E;\ x:=F(x))=(x:=F(E))$。

例如，$(x:=x-1;\ x:=2\times x+1)=(x:=2\times(x-1)+1)$。

(2) 赋值的第二条法则，对变量本身的赋值不发生任何变化，即$(x:=x)=\mathrm{II}$。

在访问未初始化的变量导致不同作用的语言中（例如，abortion），该法则不成立。

(3)事实上，虚设的赋值语句加到任意其他的赋值语句上，不会改变这些赋值语句的作用（记住x和y是不相交的），即$(x,y:=E,\ y)=(x:=E)$。

(4) 最后，变量表和表达式表进行相同的交换不会改变赋值语句的作用：

$(x,\ y,\ z:=E,\ F,\ G)=(y,x,z:=F,\ E,\ G)$。

推论：$(x,\ y:=E,\ F)=(y,\ x:=E,\ E)$。

利用上述四个法则，足以把任一赋值序列化归到单一的赋值语句。例如，

$(x,\ y:=F,\ G;\ y,\ z:=H(x,\ y),\ J(x,\ y))$

$=(x,\ y,\ z:=F,\ G,\ z;\ x,\ y,z:=x,\ H(x,\ y),\ J(x,\ y))$（根据法则(3)和(4)）

$=(x,\ y,z:=F,\ H(F,\ G),J(F,\ G))$

（根据法则(1)）。

(5) 赋值对条件操作右分配，且改变条件表达式中赋值变量的出现。

$(x:=E;\ (P \lhd b(x) \rhd Q))=((x:=E;\ P) \lhd b(E) \rhd (x:=E;\ Q))$。

(6) 对两个分支中同一变量赋值的条件语句，可以用条件表达式，对该变量单一的赋值语句去替换，即

$((x:=E) \lhd b \rhd (x:=F))=(x:=(E \lhd b \rhd F))$。

(7) 条件操作对表达式表中的个体成份向内分配，即

$(e,\ E) \lhd b \rhd (f,\ F)=(e \lhd b \rhd f),\ (E \lhd b \rhd F)$。

(8) 使用这些法则，把语句序列的条件语句压入到表达式，从赋值语句序列中删去条件语句。例如，

$(x:=E,\ (x:=F(x) \lhd b(x) \rhd x:=G(x)))$

$=(x:=(F(E) \lhd b(E) \rhd G(E)))$。

下述定理在化归到标准形式中有用。

(9) $((x:=E \lhd b \rhd \perp);\ (x:=F(x) \lhd c(x) \rhd \perp))$

$=(x:=F(E) \lhd c(E) \lhd b \rhd \mathrm{false} \rhd \perp)$。

证明 左边$=(x:=E;\ ((x:=F(x) \lhd c(x) \rhd \perp) \lhd b \rhd (\perp;$

$(x:=F(x) \lhd c(x) \rhd \perp)))$ （根据顺序复合法则(5)）

$=(x:=F(E) \lhd c(E) \rhd \perp) \lhd b \rhd \perp$

（根据法则(1)、(5)和顺序复合法则(3)）

=右边（根据条件法则(2)和(6)）。□

**无定义的表达式** 如果程序语言记号包括某些运算对象的值可能无定义的表达式，则上述一些法则要稍微变弱一点。假设程序语言允许表达式的值域足够地狭窄，对每个表达式（或表达式表）E，存在这样一个布

尔表达式$\mathcal{D}E$，在任何情况下，E的计算成功，其值为真；而当E的计算失败时，其值为假。于是，$\mathcal{D}E$本身的计算总是成功的（这在由程序员任意定义函数的语言中是不可能的）。任意不要把$\mathcal{D}$作为程序语言的一个记号。下面是一些例子。

$\mathcal{D}\text{true}=\mathcal{D}\text{false}=\text{true}$

$\mathcal{D}(E+F)=\mathcal{D}E\wedge\mathcal{D}F$

$\mathcal{D}(E/F)=\mathcal{D}E\wedge\mathcal{D}F\wedge F\neq 0$

$\mathcal{D}(E\lhd b\rhd F)=\mathcal{D}b\wedge(\mathcal{D}E\lhd b\rhd\mathcal{D}F)$

$\mathcal{D}\mathcal{D}E=\text{true}$。

现在我们约定，试图在定义域外计算表达式的作用是完全任意的。因此，

(1) $x:=E=(x:=E\lhd\mathcal{D}E\rhd\perp)$，

(2) $P\lhd b\rhd Q=(P\lhd b\rhd Q)\lhd\mathcal{D}b\rhd\perp$，

(3) $P\lhd b\rhd\perp=P\lhd b\lhd\mathcal{D}b\rhd\text{false}\rhd\perp$。

以这种观点，前述的部分法则应更改如下：

(4) $P\lhd b\rhd P=P\lhd\mathcal{D}b\rhd\perp$ （见条件法则(3)）。

(5) $(P\lhd b\rhd Q)\lhd c\rhd R$
$=((P\lhd c\rhd R)\lhd b\rhd(Q\lhd c\rhd R))\lhd\mathcal{D}b\rhd(\perp\lhd c\rhd R)$（见条件法则(11)）。

(6) $(x:=E;\ x:=F(x))=(x:=F(E)\lhd\mathcal{D}E\rhd\perp)$ （见赋值法则(1)）。

(7) $x:=E;\ (x:=F(x)\lhd b(x)\rhd x:=G(x))$
$=x:=(F(E)\lhd b(E)\rhd G(E))\lhd\mathcal{D}E\rhd\perp$ （见赋值法则(8)）

赋值法则的定理(9)也需要修改，且条件法则(12)的证明也需要修改。

论述无定义的表达式可能使问题复杂化，需要小心谨慎。然而，也得到了一些报酬。例如，空集的最小元无定义，使得Dijkstra的线性搜索定理〔4，pp105～106〕变成简单的公式：

(8) $(i:=0;\ \neg b(i)*(i:=i+1))=(i:=\min\{i\mid b(i)\wedge i\geqslant 0\})$。

注意，一般来说出现在右边的最小函数不能够实现或者不在任一程序语言中，因此，可以把(8)的右边作为其左边程序的确切规格说明。

**标准形式** 为说明迄今给出的法则的能力，把它们用于任一有限程序文本，以化归到简单的标准形式。有限程序文本不包括迭代和递归。在标准形式中，程序看上去象

$(\bigcup_{i\leqslant n}x:=E_i)\lhd b\rhd\perp$，

其中，对所有的$i\leqslant n$，有$b\Rightarrow\mathcal{D}E_i$，而$\Rightarrow$表示逻辑蕴涵。为不失一般性，必要时我们用$\lhd b\lhd\mathcal{D}b\rhd\text{false}\rhd$替换$\lhd b\rhd$（根据无定义的表达式法则(3)），以确保上下文中$\mathcal{D}b=\text{true}$。

为了阐明如何把一个程序文本化归到标准形式，只要说明如何用标准形式书写每一基始语句，以及当每一操作应用到标准形式的运算对象时，如何产生以标准形式表示的结果，赋值这一节已解释过如何改写程序中所有的赋值语句，使得所有赋值语句的左边都有相同的变量表。因此，我们假设已经完成了这部分工作。

(1) SKIP

$\mathbb{I}=((x:=x)\lhd\text{true}\rhd\perp)$

（根据赋值法则（2）以及条件法则(1)）。

(2) ABORT

$\perp=(x:=x\lhd\text{false}\rhd\perp)$ （根据条件法则(2)）。

(3) 赋值

$(x:=E)=(x:=E\lhd\mathcal{D}E\rhd\perp)$

（根据无定义的表达式法则(1)）。

(4) 不确定性

$(P\lhd b\rhd\perp)\cup(Q\lhd c\rhd\perp)$

$=(P\cup(Q\lhd c\rhd\perp))\lhd b\rhd(\perp\cup(Q\lhd c\rhd\perp))$（根据条件法则(10)）

$=((P\cup Q)\lhd c\rhd(P\cup\perp))\lhd b\rhd\perp$

（根据条件法则(10)及不确定性法则(1)和(4)）

$=((P\cup Q)\lhd c\rhd\perp)\lhd b\rhd\perp$

（根据不确定性法则(1)和(2)）

$=(P\cup Q)\lhd c\lhd b\rhd\text{false}\rhd\perp$

（根据条件法则(2)和(6)）。

其中，P和Q代表被|隔开的赋值语句表。因此，P∪Q恰好是这两张表的并。条件◁c◁b▷false▷等价于(c∧b)。由于运算对象是标准形式，这样到处有定义且蕴涵着P∪Q中所有表达式也有定义。

(5) 条件

$(P\lhd c\rhd\perp)\lhd b\rhd(Q\lhd d\rhd\perp)$

$=(P\lhd b\rhd Q)\lhd c\lhd b\rhd d\rhd\perp$ (根据条件法则(12))。

如果$P=\bigcup_{i\le m} x:=E_i$，且$Q=\bigcup_{j\le n} x:=F_j$，

则 $P\lhd b\rhd Q=\bigcup_{i\le m}\bigcup_{j\le n}(x:=E_i\lhd b\rhd x:=F_j)$

(根据条件法则(9))

$=\bigcup_{i\le n}\bigcup_{j\le m} x:=(E_i\lhd b\rhd F_j)$

(根据赋值法则(6))

因为$c\Rightarrow\mathscr{D}Ei$，且$d\Rightarrow\mathscr{D}F_j$，所以，对所有的$i$，$j$，有$c\lhd b\rhd d=\mathscr{D}(Ei\lhd b\rhd F_j)$。

于是，(5)的左边化归到标准形式。

(6) 顺序复合

$((\bigcup_{i\le n} x:=E_i)\lhd b\rhd\perp);\ ((\bigcup_{j\le m} x:=F_j(x))\lhd c(x)\rhd\perp)$

根据对∪的分配，上式化归到

$\bigcup_{i\le n}\bigcup_{j\le m}((x:=E_i\lhd b\rhd\perp);\ (x:=F_j(x)\lhd c(x)\rhd\perp))$

$=\bigcup_{i\le n}\bigcup_{j\le m}(x:=F_j(E_i)\lhd c(E_i)\lhd b\rhd$ false $\rhd\perp)$ (根据赋值法则(9))。

(4) 中所述方法把条件分配到条件，从而得到

$(\bigcup_{i\le n}\bigcup_{j\le m} x:=F_j(E_i))\lhd\bigwedge_{i=1}^{n}(c(E_i))\lhd b\rhd$false$\rhd\perp$，

其中∧归纳定义为

$\bigwedge_{i=1}^{0} c_i=c_0$

$\bigwedge_{i=1}^{n+1} c_i=(\bigwedge_{i=1}^{n} c_i)\lhd c_{n+1}\rhd$false。

这样证明了所有有限程序都是可化归的。

标准形式的重要性在于它们提供了一种完整的检测方法，用以确定两个有限程序文本是否表示相同的程序。首先把两个程序化归到标准形式。若它们的标准形式相等，则两个程序也相等；否则，两个程序不相等。

两个标准形式$(\bigcup_{i\le n} x:=E_i)\lhd b\rhd\perp$和$(\bigcup_{j\le m} x:=F_j)\lhd c\rhd\perp$相等，当且仅当b=c且$\{v\mid \exists\, i\le n\cdot v=E_i\}=\{w\mid \exists\, j\le m\cdot w=F_j\}$，其中，对表达式b、c、$E_i$和$F_j$中所有的变量值，这些方程必须都成立。

这些方程的真假(例如，在整数算术中)可能是不可判定的。这里列出的结果仅建立了相对完备性。

(未完待续)

〔宋国新译自Comm. ACM, vol 30, №8, 1987, 孙永强校〕

(上接28页)

cial Intelligence" Scott E.Fahlman and Geoffrey E. Hinton Carnegie-Mellon University, Computer 1987.1.

2. "The Connection Machine" W.Daniel Hillis 1985,MIT.Press
3. "The Cognitive Architecture Project" Dan Hammerstrom,David Maier,Shreckant Thakkar, Computer Science & Engineering The Oregon Graduate Center,1986.2.7
4. "Connectionism and Cognitive Science" Andy Clark. Cognitive Studies Programe,University of Sussex Brighton. U. K. "Advances in AI"1987.4. Edinburgh.
5. "Applications of the Connection Machine" David I.Waltz,Think Machine Corporation IEEE,1987.1,
6. J.A.Feldman and D.H. Ballard,"Connectionist Models and their Properties", Cognitive Science, Vol.6, no. 3, Ablex, Norwood, NJ, 1982, PP. 205-254.
7. G.E.Hinton, J.L.McClelland. and D. E.Rumelhart,"Distributed Representation."Parallel Distributed Processing: Explorations in the Microstructure of Cognitive, eds.D.E. Rumelhart, J.I. McClelland, and, the PDP Research Group MIT Press, Cambridge, MA, 1986
8. G.E.Hinton, "Learning Distributed Representations of Concepts," Proc. 8th.An.Conf.Cognitive Science,Soc., Lawrence Erlbaum Assoc., Hillsdale, NJ, 1986.