程序设计法则（续）

C.A.R.Hoare等

域的性质

本节使用文献[12]的方法，介绍迭代和递归。

序关系 作为预备知识，我们先考察程序之间序关系$\sqsupseteq$的性质。

定义 $P\sqsupseteq Q \triangleq P \sqcup Q = P$。

这意味着Q是一个比P更确定的程序。Q能做的事情，P也能做，而Q不能做的事情，P也可能不能做。因此，无论从那一点来看，Q是一个比P更能预测、更能控制的程序。在P可靠地用于有用目的的任何情况下，以Q替换P也必然能达到同样的目的。反之却不然。对Q适合的某种目的，由于P有较大的不确定性，它却可能不适合这种目的。于是，$P\sqsupseteq Q$意味着在任何情况下，Q比P更好，或者至少一样好。我们将使用词语"更好"本身去理解"更好或者至少一样好"。

程序之间关系$\sqsupseteq$不是全序的。因为并非对所有的P和Q，命题$P\sqsupseteq Q$为真或$Q\sqsupseteq P$为真。P在某些方面比Q好，而Q在另外的方面比P好。然而，$\sqsupseteq$是偏序的，因为它满足如下法则：

(1) $P\sqsupseteq P$ （自反性）

(2) $P\sqsupseteq Q \wedge Q\sqsupseteq P \Rightarrow P=Q$ （反对称性）

(3) $P\sqsupseteq Q \wedge Q\sqsupseteq R \Rightarrow P\sqsupseteq R$ （传递性）

从$\sqsupseteq$的定义和$\sqcup$的法则，可直接证明上述法则。

证明

(1) $P\sqcup P=P$ （$\sqcup$的幂等性）

(2) $(P\sqcup Q=P) \wedge (Q\sqcup P=Q)$

$\Rightarrow R=P\sqcup Q=Q\sqcup P=Q$ （$\sqcup$的对称性）

(3) $(P\sqcup Q=P) \wedge (Q\sqcup R=Q)$ （前件）

$\Rightarrow P\sqcup R=(P\sqcup Q)\sqcup R$ （第一个前件）

$=P\sqcup(Q\sqcup R)$ （$\sqcup$的结合性）

$=P\sqcup Q$ （第二个前件）

$=P$ （第一个前件）。□

ABORT语句$\perp$是所有程序中最不确定的程序。它是最不能预测、最不能控制的程序。因此，它是最坏的程序。

(4) $\perp \sqsupseteq P$

证明 $\perp \sqcup P=\perp$。 □

一般来说，其行为或者象P或者象Q的机制，比P和Q都坏：

(5) $(P\sqcup Q)\sqsupseteq P \wedge (P\sqcup Q)\sqsupseteq Q$。

证明 $(P\sqcup Q)\sqcup P=P\sqcup(Q\sqcup P)$ （结合性）

$=P\sqcup(P\sqcup Q)$ （对称性）

$=(P\sqcup P)\sqcup Q$ （结合性）

$=P\sqcup Q$ （幂等性）

□

事实上，$P\sqcup Q$是具有性质(5)的最好程序。任何一个比P和Q都坏的程序R也坏于$P\sqcup Q$，反之亦然：

(6) $R\sqsupseteq(P\sqcup Q) \equiv (R\sqsupseteq P \wedge R\sqsupseteq Q)$。

证明 左边$\Rightarrow R\sqsupseteq P$ （根据(5)的传递性）

左边$\Rightarrow R\sqsupseteq Q$ （同上）

右边$\Rightarrow (R\sqcup P=R) \wedge (R\sqcup Q=R)$

（根据$\sqsupseteq$的定义）

$\Rightarrow (R\sqcup P)\sqcup(R\sqcup Q)=R\sqcup R$

（根据方程相加）

$\Rightarrow P\sqcup(P\sqcup Q)=R$

（根据$\sqcup$的性质）

$\Rightarrow$左边 （根据$\sqsupseteq$的定义）。□

如果$P\geqslant Q$，则意味着在各种情况下，Q都比P好。由此可见，每当P出现于较大的程序中，如果用Q替换P，则其唯一结果是改进了原来的较大程序（至少使原来的程序没有改变）。例如，

(7) 如果$P\sqsupseteq Q$，则

$P\sqcup R\sqsupseteq Q\sqcup R$

$\wedge (P;R)\sqsupseteq(Q;R)$

$\wedge (R;P)\sqsupseteq(R;Q)$

$\wedge (P \lhd b \rhd R)\sqsupseteq(Q \lhd b \rhd R)$

$\wedge$ $(R \lhd b \rhd P) \supseteq (R \lhd b \rhd Q)$

$\wedge$ $(b*P) \supseteq (b*Q)$。

总之，上述法则表示，本文语言中所有操作，在保持其运算对象$\supseteq$的意义上是单调的。事实上，对$\cup$分配的每一个操作也是单调的。

定理 如果F是从程序到程序的任一函数，且对所有的程序P和Q，有 $F(P \cup Q)=F(P) \cup F(Q)$，则F是单调的。

证明 $P \supseteq Q \Rightarrow P \cup Q=P$ （根据$\supseteq$的定义）

$\Rightarrow F(P) \cup F(Q)=F(P \cup Q)$

（根据F的分配性）

$=F(P)$ （根据$=$的性质）

$\Rightarrow F(P) \supseteq F(Q)$ （根据$\supseteq$的定义）。

□

单调函数复合定义的任一函数也是单调的。由于本文中程序语言的所有操作都是单调的，所以，借助于这些操作组成的任一程序，对它的每一构成成分也是单调的。于是，如果任一构成成分被较好的构成成分替换了，则从整体上说，其唯一结果是改进了原来的程序。如果新的程序也比老的程序更有效，则得益更多。

上确界 我们已经看到，$P \cup Q$是坏于P和Q的最好程序。假设需要一个好于P和Q的程序。一般来说，这样的程序不存在。考虑两个赋值语句$x:=1$和$x:=2$。这二个程序是不相容的，且不存在各种情况下都比这二个程序好的程序。如果x的最后值是1，则第一个程序满足，但第二个程序不满足。另一方面，如果x的最后值是2，则第一个程序不满足。

考虑如下不确定性程序：

$P=(x:=1 \cup x:=2 \cup x:=3)$，

$Q=(x:=2 \cup x:=3 \cup x:=4)$。

在这种情况下，存在一个比它们都好的程序，即$x:=2$。事实上，存在着好于P和Q的最坏程序。我们以$P \cap Q$表示，

$P \cap Q=(x:=2 \cup x:=3)$。

如果两个程序P和Q有相同的改进部分，则称它们是相容的；于是，$P \cap Q$表示P和Q的最坏的相同改进部分。这种事实总结于下列法则中。

(1) $(P \supseteq R) \wedge (Q \supseteq R) \equiv (P \cap Q) \supseteq R$。

推论 $P \supseteq (P \cap Q) \wedge Q \supseteq (P \cap Q)$

（根据序关系法则（1））。

每当操作$\cap$有定义时，它是幂等的，对称的以及结合的，而且有同一律$\perp$。

(2) $P \cap P=P$

(3) $P \cap Q=Q \cap P$

(4) $P \cap (Q \cap R)=(P \cap Q) \cap R$

(5) $\perp \cap P=P$

操作$\cap$对于任一有限的相容程序集$S=\{P, Q, \cdots, T\}$也适用。只要存在一个好于所有这些相容程序的程序，我们用$\cap S$表示这样一个最小的程序：

$\cap S=P \cap Q \cap \cdots \cap T$，只要$\exists R.P \supseteq R \wedge Q \supseteq R \wedge \cdots \wedge T \supseteq R$。

从而得到

(6) $(\forall P \in S.P \supseteq R) \equiv \cap S \supseteq R$，只要$\cap S$有定义。

重要的是要认识到，$\cap$不是本文程序语言的组合算子，即使P和Q是相容的程序，但严格来说$P \cap Q$不是一个程序。例如，设程序P把任意递升的数列赋给一数组，程序Q对该数组进行任意的排列。这样，程序$P \cap Q$同时满足这两个规格说明，因而把该数组排成递升次序。不巧，程序设计并不如此容易。正如和其它的工程分支一样，一般不可能进行许多不同的设计，其中每个设计都满足一个规格说明需求，然后，把它们合并成满足所有需求的单个产品。相反，工程师必须在单个设计中满足所有的需求。这就是为什么设计和编程变得复杂的原因。

这些问题不单纯由规格说明引起。规格说明由合取and随意连接。可以把$P \cap Q$作为抽象程序，或者作为这样一个程序的规格说明，每当P和Q完成时，该程序也完成，同时，仅当P和Q都不完成时，该程序也不完成。$P \cap Q$规定了一个总好于P和Q的程序（如果该程序存在）。

事实上，合取对于构造大型的规格说明是最有用的操作。为此而设计的任一语言中

应包括合取。使用合取得到的清晰的规格说明比起使用能实现的规格说明语言得到的更重要得多。不巧，合取也同样难于实现。因此，适当的规格说明的形式化和满足这种规格说明的程序设计（例如，软件工程）总是严峻的智力挑战。

极限 设S为非空(可能是无限的)集合，对其中任意的元素序偶，S恰好包含了比这两者都好的元素。称这样的集合是有向的。

定义 S是有向的，意指

$(S\neq\{\})\wedge\forall P,Q\in S.\exists R\in S.P\sqsupseteq R\wedge Q\sqsupseteq R$。

下面是有向集合的例子：

{P}　　只有一个元素的集合，

{P，P∪Q}　因为P∪Q⊒P且P⊒P，

{P，Q，R}　其中P⊒R∧Q⊒R。

若S是有向且有限的，则很显然，它包含一个比其他所有元素都要好的元素，将其定义为∩S且∩S∈S。

若S是有向且无限的，则它未必包含最好的元素。然而，该集合有一个极限∩S。正如我们注意到的，∩S是好于S的所有元素中最坏的程序。集合S象收敛的数列，该数列有一个不在其上的极限。通过挑选该集合的元素，有可能使之紧密逼近其极限。由于没有介绍"紧密"的度量标准，必须间接地对之解释，即，对坏于∩S的任一有限程序P，S中存在着好于P的一个元素。

程序有向集极限的一个有趣性质，是本文语言中所有的操作都保持这个极限。因此，称这样的操作为连续。

(1) $(\cap S)\cup Q=\cap\{P\cup Q\mid P\in S\}$。

(2) $(\cap S)\lhd b\rhd Q=\cap\{P\lhd b\rhd Q\mid P\in S\}$。

(3) $(\cap S);Q=\cap\{P;Q\mid P\in S\}$。

(4) $Q;(\cap S)=\cap\{Q;P\mid P\in S\}$。

(5) $b*(\cap S)=\cap\{b*P\mid P\in S\}$。

连续有这样的性质，即连续函数的任何复合仍是连续函数。设x表示程序，F(x)仅通过连续操作构造起来的可能包含x出现的一个程序。如果S是有向的，则得到

(6) $F(\cap S)=\cap\{F(x)\mid x\in S\}$。

迭代与递归 给定程序P和布尔表达式b，我们归纳定义一个无限集合$\{Q_n\mid n\geqslant 0\}$，其中

$Q_0=\perp$，

$Q_{n+1}=(P;Q_n)\lhd b\rhd\mathrm{I}$。

对所有的$n\geqslant 0$。

很清楚，从定义我们知道，Q_n是对循环体P一直迭代到n次，其行为象$(b*P)$的一个程序，而当第n次迭代时，不再继续执行P，即什么事情也不做($\perp$)。于是，对所有的n显然有$Q_n\sqsupseteq Q_{n+1}$（可用归纳法形式地证明）。因此，集合$\{Q_n\mid n\geqslant 0\}$是有向的，当取n足够大时，我们能紧密逼近循环$(b*P)$的行为。把循环本身定义为所有近似值的极限：

(1) $b*P=\cap\{Q_n\mid n\geqslant 0\}$。

使用同样的技术，定义更一般形式的递归。设x代表欲构造的递归程序名，F(x)定义该程序期望的行为。在F(x)中，x的每次出现代表对整个递归程序的又一次调用。如前所述，我们构造递归程序行为的一组近似值。

$F^0(Q)=Q$，

$F^{n+1}(Q)=F(F^n(Q))$，

对所有的$n\geqslant 0$。

$F^n(\perp)$行为正确；只要递归深度$<n$。因为F是单调的，且$F^0(\perp)\sqsupseteq F^1(\perp)$，从而得到

$F^n(\perp)\sqsupseteq F^{n+1}(\perp)$，

对所有的n。

因此，$\{F^n(\perp)\mid n\geqslant 0\}$是有向集合，我们把递归程序($\mu x.F(x)$)表示定义为它的极限：

(2) $\mu x.F(x)=\cap\{F^n(\perp)\mid n\geqslant 0\}$。

按照上述解释，迭代是递归的特殊情况。

(3) $b*P=\mu x.(P;x)\lhd b\rhd\mathrm{I}$。

递归定义程序的最重要性质是每次递归调用又等于整个程序。或更形式地说，$\mu x.F(x)$ 是方程 $x=F(x)$ 的解。我们以下列法则表示：

(4) $\mu x.F(x) = F(\mu x.F(x))$。

证明　右边 $= F(\sqcap\{F^n(\perp) \mid n\geq 0\})$

（根据μ的定义）

$= \sqcap\{F(F^n(\perp)) \mid n\geq 0\}$

（根据F的连续性）

$= \sqcap(\{F^{n+1}(\perp) \mid n\geq 0\}\cup\{\perp\})$

（根据F^{n+1}的定义以及"最小上界"的法则（5））

$= \sqcap\{F^n(\perp) \mid n\geq 0\}$

（因为$F^0(\perp)=\perp$）

$=$左边　（根据定义）。□

推论　$b*P = (P;\ (b*P))\lhd b\rhd \mathbb{I}$。

一般来说，方程 $x=F(x)$ 不止有一个解。的确，对方程 $x=x$ 来说，任一程序绝对地是它的一个解。然而，所有的解中，$\mu x.F(x)$ 最坏：

(5) $Y=F(Y)\Rightarrow \mu x.F(x)\sqsupseteq Y$。

证明　$(Y=F(Y))$

$\Rightarrow (\perp\sqsupseteq Y)\wedge(Y=F(Y))$

$\Rightarrow (F(\perp)\sqsupseteq F(Y))\wedge(Y=F(Y))$　（根据F的单调性）

$\Rightarrow (F(\perp)\sqsupseteq Y)$。

根据归纳法，对所有的 $n\geq 0$，得到

$Y=F(Y)$

$\Rightarrow F^n(\perp)\sqsupseteq F^n(Y)\wedge Y=F^n(Y)$

$\Rightarrow F^n(\perp)\sqsupseteq Y$

$\Rightarrow \sqcap\{F^n(\perp) \mid n\geq 0\}\sqsupseteq Y$

（根据"最小上界"的法则（6））

即为所求。

规格说明

迄今已介绍了两个重要的概念。第一个概念是规格说明描述了程序期望的行为，它无需指明该程序如何被执行。第二个概念是具体程序P可能好于规格说明S；因此，每当需要行为象S那样的程序时，具体程序P可用于此目的。这种情况下，我们说P满足规格说明S，或记作 $S\sqsupseteq P$。当规格说明S已知时，找出满足S的程序P是程序员的责任。程序设计法则的实际用途会有助于这样的目的。

现在介绍规格说明的演算以帮助程序开发。规格说明不必被机器执行。因此，不必要把我们自己局限于特定的程序语言的记号。同样地，不必要把我们自己局限于能满足的规格说明。作为特殊的例子，我们引进不能被任何程序满足的规格说明T。对所有的规格说明定义一个操作 $\sqcap$：每当R和S不一致时，$(R\sqcap S)$ 的结果为T。进一步，若S是规格说明的任意集合，则

$\sqcap S$表示要求S中所有R满足的规格说明；

$\sqcup S$表示要求S中某些R满足的规格说明。

这些规格说明是S的极限。

(1) $Q\sqsupseteq\sqcup S\equiv \forall R\in S.Q\sqsupseteq R$。

(2) $\sqcap S\sqsupseteq Q\equiv \forall R\in S.R\sqsupseteq Q$。

序$\sqsupseteq$应用于规格说明，正如它应用于程序一样，不过，以新的含义解释序$\sqsupseteq$。如果 $S\sqsupseteq R$，则意味着S是较弱的规格说明，它比R更易满足。满足R的任一程序也一定满足S，但有可能存在更多的满足S的程序。于是，$\perp$是被任何一个程序都能满足的最容易的规格说明，而T是不可能被程序满足的规格说明。

我们不再为规格说明引进专门的记号或语言，在程序执行的前后，允许描述变量值之间关系的任何语言。这样，我们可能使用程序语言记号，甚至用不可计算的操作扩充这些程序语言记号，或者我们把谓词用于谓词程序设计[6]，或者把谓词序偶用于VDM[9]。我们假设规格说明语言至少包括程序设计语言的所有记号，因此，程序本身是最强的规格说明。

于是，所有的程序都是规格说明，但反过来并非如此。因此，存在一些对程序来说为真，但对规格说明来说不为真的法则。特别地，对规格说明来说，顺序复合法则（3）和极限法则（4）不成立。然而，坚信规格说明服从关系演算[13]的所有法则。

最弱前置规格说明　我们以适合于具体程序的所有操作去构造规格说明。例如，如果R和S是规格说明，则$(R;S)$是程序$(P;Q)$满足的规格说明，其中P满足R，Q满足

S（也能被其他程序满足）。这在自顶向下（也称逐步求精）的程序开发中是极其有用的。例如，假设原来的任务是构造满足规格说明W的程序。或许，我们设想把任务W分解为R和S表示的两个较简单的子任务。通过证明W⊒R；S来证明这种分解的正确性。应该在着手设计子任务R和S之前，就完成这种证明。于是，类似的方法用于找出解决这些子任务的程序P和Q，使得R⊒P且S⊒Q。从顺序复合的单调性直接推出，P；Q是解决原来任务W的程序，即，W⊒（P；Q）。

在解决任务W中，假定记得两个子任务中第二个任务S适当的规格说明，但是，我们不知道第一个子任务确切的规格说明。从S和W中计算R是有用的。因此，我们把最弱前置规格说明S＼W定义为这样一个最弱规格说明，第一个子程序R必须满足S＼W，以便复合结构（R；S）用于完成原来的任务W。我们以下述法则表示上述事实。

(1)W⊒(S＼W)；S。

(S＼W)是S左除W；除数S可用后乘抵消，其结果和W相同或更好。

下面是最弱前置规格说明的几个例子，其中x是整型变量：

(x:=3×x)＼(x:=6×y)=(x:=2×y)，因为(x:=2×y；x:=3×x)=(x:=6×y)。

(x:=2×x)＼(x:=3)=T，因为3是奇数，它不可能2倍于某个整数。

(x:=2×x)＼(x:=3∪x:=4)=(x:=2)，因为(x:=3∪x:=4)⊒x:=4=(x:=2；x:=2×x)。

上述法则不是唯一定义的S＼W。但是，在不等式W⊒(x；S)所有的解x中，S＼W最容易得到。于是，为了找出这样一个解，充要条件是它要满足S＼W：

(2)W⊒(x；S)≡(S＼W)⊒x。

于是，在开发满足W的顺序程序中，一般来说，如果已知S是顺序复合右操作对象的规格说明，则把S＼W作为顺序复合左操作对象的规格说明。那就是为什么称之为最弱前置规格说明的原因。为了方便，本文的余下部分中，省略最弱这个词。

前置规格说明P＼R（其中P是程序）所起的作用十分类似于Dijkstra的最弱前置条件。前置规格说明满足类似于最弱前置条件的几个条件。在下述三个法则中，P必然是一个程序。

(3)为完成一个不可能的任务，即使求助于P，也是无济于事的：

P＼T=T。

(4)为借助于P完成两个任务，必须编写同时完成这两个任务的程序。

$P\backslash(R_1 \cap R_2)=(P\backslash R_1)\cap(P\backslash R_2)$。

这个分配律可扩充到任意集合的极限：

$P\backslash(\cap S)=\cap(P\backslash R \mid R\in S)$。

(5)最后，考虑一组规格说明$S=\{R_i \mid i\geqslant 0\}$，使得$R_{i+1}\supseteq R_i$，则

$P\backslash(\cup S)=\cup\{P\backslash R_i \mid i\geqslant 0\}$。

下述法则十分类似于对应的最弱前置条件的法则：

(6)程序Ⅱ什么也没有变。设想在Ⅱ之后做到的任何事情必须在Ⅱ之前就做到：

Ⅱ＼R=R。

(7)为借助于P∪Q得到R，必须用其中的任意一个得到它：

(P∪Q)＼R=(P＼R)∩(Q＼R)。

(8)为借助于(P；Q)得到R，必须借助于P得到(Q＼R)：

(P；Q)＼R=P＼(Q＼R)。

(9)条件的对应法则需要对规格说明的一个新的操作：

(P◁b▷Q)＼R=(P＼R)◁b▷(Q＼R)，其中S◁b▷T规定如下的程序：如果b在执行之后为真，则它对应于规格说明S的行为。如果b在执行之后为假，则它对应于规格说明T的行为。即使P和Q是程序，P◁b▷Q也不是一个程序；事实上，P◁b▷Q甚至是不可实现的。例如，考虑下例：

$x:=false \lhd x \rhd x:=true$。

广义逆 操作$\backslash$有一个对偶/。(R/S)是程序x的最弱规格说明，使得$R \supseteq (S; x)$。其性质十分类似于$\backslash$的性质：例如：

(1) $R \supseteq S; (R/S)$,

(2) $R \supseteq (S; x) \equiv (R/S) \supseteq x$,

(3) $T/P = T$, 若P是一个程序,

(4) $(R_1 \cap R_2)/P = (R_1/P) \cap (R_2/P)$,

(5) $R/\mathbb{I} = R$,

(6) $R/(P \cup Q) = (R/P) \cap (R/Q)$,

(7) $R/(P; Q) = (R/P)/Q$。

在某种意义上，前置规格说明和后置规格说明是顺序复合的右逆和左逆。对任意并集分配的任何操作F，都存在这类逆。逆定义如下：

(8) $F^{-1}(R) = \cup\{P \mid R \supseteq F(P)\}$。

这不是F精确的逆，但它满足如下法则：

(9) $R \supseteq F(F^{-1}(R))$。

证明 右边$= F(\cup\{P \mid R \supseteq F(P)\})$

（根据F^{-1}的定义）

$= \cup\{F(P) \mid R \supseteq F(P)\}$

（根据F的分配性）

$\supseteq R$

（根据集合论）。□

$F^{-1}(R)$是不等式方程$R \supseteq F(x)$中所有解x的并集，所以它必然是最弱的通解：

(10) $R \supseteq F(x) \equiv F^{-1}(R) \supseteq x$。

F必须对$\cup$分配的条件是逆F^{-1}存在的基础。为此，考虑如下计数器的例子，

$F(x) = x; x$

$P = x:=x$

$Q = x:=-x$。

F是一个对其操作数需要执行多次的函数。当把F应用于两个程序P或Q的不确定选择机制时，每次执行可能产生不同的选择。因而，下述例子表明F不是分配的。

$F(P \cup Q) = (P \cup Q); (P \cup Q)$

（根据F的定义）

$= (P; P) \cup (P; Q)$

$\cup (Q; P) \cup (Q; Q)$

（根据析取；）

$= (x:=x; x:=x) \cup (x:=x; x:=-x) \cup$

$\cup (x:=-x; x:=x) \cup (x:=-x; x:=-x)$

$= x:=x \cup x:=-x$

但$F(P) \cup F(Q) = (x:=x; x:=x) \cup (x:=-x; x:=-x)$

$= x:=x$

因为$P \supseteq F(P)$且$P \supseteq F(Q)$，从而得到

$\cup\{x \mid P \supseteq F(x)\} \supseteq P \cup Q$

（根据集合论）。

根据法则(10)以及$F^{-1}(P)$的定义，我们得$P \supseteq F(P \cup Q)$为假。该矛盾表明，即使在法则(10)较弱的意义上，F也没有逆。

逆$F^{-1}(R)$（当它存在时）有助于满足规格说明R的程序的自顶向下开发。假设决定用F来定义顶层的程序结构，则必需计算$F^{-1}(R)$，且把它作为程序成分x的规格说明。确保获得最终程序F(x)满足原来规格说明R，即$R \supseteq F(x)$。

不巧，这种方法不能产生具有两个或两个以上成分的结构F。因此，在计算逆之前，除了一个程序成分之外，必需固定其他所有的成分。

结 论

本文提出的法则有助于程序员开发满足规格说明的程序。运用代数变换也有助于优化。我们的基本观点是把程序本身及其规格说明看成数学表达式。因此，就像表达式表示诸如数、集合、函数、群、范畴等类似的数学概念一样，程序以及规格说明可直接用于数学论证中。在同类框架中，对程序和规格说明一并处理也是很方便的；程序和规格说明之间的主要差别在于程序是规格说明的子类，我们以诸如通用数字计算机能输入、变换、执行的严格限制的记号来表示程序。

（下转10页）

计。例如，人们常常谈论软件重用，仿佛重用已经书写好的代码大概是一种不可思议的方法。实际上，如果不精心设计要重用的软件，那么就没有办法重用软件。重用软件得到的报酬是很大的，但它要求"在前面"付出大量的费用。

最后，正如在全自动化中那样，自动程序设计的真正希望是不仅使现在做什么实现自动化，而且还要完全变革做事的方式。例如，在办公室自动化情况中，自动化就是重新设计办公室中的整个信息流，而不是把同样的旧表格放入电子介质中。对于程序设计，这意味着重新考察软件生存期的传统模型，随着原型方法日益得到接受，这种重新考虑正在发生；这还意味着语言、环境和接口间的传统差别正在消除，正以图形接口和面向对象的程序设计形式出现。

以高级语言的编译程序形式出现的自动程序设计在五十年代末期成为可用的。到六十年代末期，显然认为下一步应是移向甚高级语言。然而，这一步被证明比期望的困难得多，并且自动程序设计的自底向上方法在七十年代基本上停止了发展。

七十年代对自动程序设计的研究工作重点转移到了狭窄领域方法。然后，各种系统（如数据库查询语言）已经构成，在小的领域出现了面向端点用户、全自动的程序设计。

在八十年代，研究兴趣又回到了自底向上方法。这导致了甚高级原型设计语言的出现。此外，我们还看到了第四代语言和程序生成器问世，它们的应用重点更狭窄，也更有效。八十年代，还看到了对所谓的高级设计辅助工具和其它高级程序设计工具这样的自动程序设计的助手方法的研究兴趣日益增长。

进一步观察未来，还没有迹象表明会出现动人的自动程序设计型式。然而，将有一些重大的演变式进展。凑巧，我们将对1998年的自动程序设计得说一句1958年说过的老话：自动程序设计已极大地提高了程序员生产率并且进一步缩小了程序员与用户之差别。

参考文献（略）

〔示　元译自《CGMPUTER》Vol.21 No.8 1988, p40—51　王　心校〕

（上接69页）

然而，我们承认本文的阐述确有不足之处，理论上的一个弱点是本文介绍的这些法则都是不言而喻的公理或假设，本文试图从人们对程序已经理解其性质的基础上来获取这些法则。

对顺序程序及其规格说明来说，有关的数学定义已在关系的经典理论[7]中形式化了。使用这样的定义来证明本文列举的法则，得到了有价值的结果，即这些法则是一致的。进而，这些定义还到划程序设计的数学，以及如何使之应用于实践。特别地，他们还建议了另外有用的法则，并且在下述意义下证明给定的一组法则是完备的，即，部分正确的程序可以从这些法则中直接推导出来，而不必求助于更复杂的定义。这对这样的实际程序员是极其有用的，他们不必知道比科学家必须知道的Dedekind分割实数更多的学科基础。

虽然本文给出将近100条法则，但是如何把他们直接用于设计正确而有效的有一定规模的程序，人们依然要做很多的工作。把这些数学法则应用于程序设计，以取得实践经验是必经之路。必须继续研究更深入且更专门的定理，这些定理能简单地用于较狭窄但不是太狭窄的领域。应用数学家以及纯数学家二千多年来用这种研究方法已经取得了巨大的进步。如果我们遵照这种研究方法，或许我们既在理论研究上又在实际应用上会取得更快的进步。

参考文献（略）

〔宋国新译自Comm.ACM,vol.30 №8, 1987, 孙永强校〕