第十一章 参考

11.0释疑

本节解释了本书在选择和提供材料方面所作的一些取舍, 只关心学习这些材料的学生可能对此不感兴趣, 但教师和研究者也许会感兴趣。

11.0.0记号

在标准的记号和新的完美的记号之间，我选择了标准的记号。例如, 在表示两个数*x*和*y*的最大值时, 使用了函数*max*: *max x y*。 因为最大值是对称和结合的, 所以可以引入一个更好的象↑的对称符号作为中缀操作符：*x*↑ *y*。我个人总是这样做, 但在本书中我所选择的符号尽量保持数量少和符合传统。大多数人在看到*max x y*时不需要预先作任何解释就会明白它的含义, 但对*x* ↑ *y*就不是这样。

在选择操作符的优先次序时遵循两个准则：括号的使用量最少和容易记忆。后者可通过沿袭传统、将相关符号放在一起和使用尽可能少的优先级别来实现。这两个准则有时是相矛盾的, 传统有时也是相矛盾的, 并且以上帮助记忆的三个建议有时也是相矛盾的, 最后 我们必须作一个选择并一直使用它。额外的括号总是可以使用的, 特别是在优先级结构不清晰时更应当使用。为了结构清晰，给 ∧ 和 ∨ 相同的优先级应该更好，但我还是保持了传统。本书采用的优先级比我预想的要多。在第一稿当中, 我用单目–代替¬、×代替∧、二元+代替∨、=和≠代替⇒和⇐, 虽然节约了四个符号, 但是违背了数学传统, 并且多用了许多括号。使用具有较低优先级的大型符号 $=\ \Leftarrow\ \Rightarrow$ 是一个新发明, 我希望它既容易理解也容易书写, 不过, 请用过一阵后再对此进行判断, 它毕竟节约了许多括号。这种用法可以推广到所有符号和各种大小(依次增加)。

----------记号结束

11.0.1布尔理论

布尔理论有时也用其它名称：布尔代数、命题演算、判断逻辑, 它的表达式有时称为“命题”或者“判断”。有时“项”和“命题”又有所区别, “项”表示值, 而“命题”则不表示值, 只表示为真或为假。在“函数”和“谓词”之间也有类似区别, “函数”对参数求值, 而“谓词”则实例化为真或假。但是慢慢地，逻辑的主题从它过去的混淆的哲学中显现出来。我认为命题就是布尔表达式，并把它们等同于数表达式和其它类型的表达式, 而谓词就是布尔函数, 我使用与数、字符、集合、函数一样的符号来表示布尔数。也许将来我们不觉得有必要去想象表达式所表示的抽象对象；我们将通过实际应用来解释它们。我们将通过它们的使用规则而不是其哲学含义来说明我们的形式体系。

为何要引入“反公理”和“反定理”？它们非传统(实际上是我自己创造了这些词)。如同第一章所述，否定操作符和一致性规则的引入使得我们无须这两个新词，我们可以用¬*expression*是定理来代替说明*expression*是反定理。另外，也不必引入⊥，可以用¬T来代替它。引入“反定理”的理由之一是它比说“定理的否定”来得简单，理由之二是它可以帮助我们弄清“证明为假”和“不可证明”这两者之间的重要差别。理由之三是有些逻辑不使用否定操作符和一致性规则。本书中的逻辑是“经典逻辑”；“构造逻辑”省略了完备性规则；“求值逻辑”

省略了一致性规则和完备性规则。

有些书利用一种形式记号提供证明规则(和公理)。本书中没有形式化的元语言，元语言就是自然语言。形式化的元语言可帮助我们提出理论并与其它竞争的形式体系作比较，同时对证明形式体系的定理来说是必须的。但在本书中，仅提出了一种形式体系，如果为了提出这种形式体系而先去学习另一种，显然有点不必要。一种表示置换的形式化元记号可使我们将函数应用规则写成：

$(\lambda v \cdot b)\ a = b[\ a/v\ ]$

但接着就必须说明 $b[\ a/v\ ]$表示“将b中的v置换成a ”，因此不如直接说：

$(\lambda v \cdot b)\ a =$ (将b中的v置换成a)

如果要使用自动证明器就需要一个证明语法(形式化“提示”), 但在本书中没有必要，我也就没有引入。

有些作者可能会区别“公理”和“公理系统”，后者包含了可以实例化生成公理的变量。我所使用的“公理”同时包含了这两层含义。另外，我把“定律”作为“定理”的同义语(我当然很希望减少我的词汇，但这二者都很常用)，而在其它书中可能会通过是否存在变量来区别它们，或者他们可能用“定律”来表示“我希望它为定理，但目前尚未设计一个合适的理论”。

在选择某些公理和定律的名称时我有点随便，我所说的“透明性”常常被称为“置换等价式为等价式”，后者说起来较长且含义也不明确。我的每一个移动定律在历史上都为两个定律，一个方向的蕴含为一个定律，另一个方向的蕴含为另一个定律，其中之一称为“输入”，另一个称为“输出”，但我总是记不住谁是谁。

----------布尔理论结束

11.0.2束论

为什么要引入束? 集合不也一样吗? 束不就是使用了一些特殊记号和术语的集合吗? 其实不然, 详见以下分解。假如只引入集合, 想要能写出{1, 3, 7}及类似的表达式, 我们可能用如下的小语法来描述这些集合表达式:

set = “{” *contents* “}”
contents = *number,*
| *set,*
| *contents* “,” *contents*

我们希望说明集合中元素的顺序无关紧要, 即{1, 2} = {2, 1}；最好是用形式化描述：$A,B = B,A$ (逗号是对称或可交换的)。接着, 我们想说明集合中元素的重复也是无关紧要的, 即{3, 3}={3}；最佳描述是$A,A=A$ (逗号是幂等的)。这里所做的一切正是在刻画束, 只是称它们为集合的“内容”。请注意, 上述语法恰恰等同于束：连接(用并置表示)分布作用于它们操作数的所有元素, 而或者(竖直短线)就是束的并。

当一个孩子初次学习集合时, 常常有一个初始障碍：包含一个元素的集合和该元素是不同的。将集合比作打包就容易理解多了：装有一个苹果的包显然和该苹果不同, 正如{2}和2不同, {2,7}和 2,7 不同一样。束论告诉我们聚合, 而集合论告诉我们打包, 两者是相互独立的。

我们可以在不依赖于束的情况下定义集合(多少年来一直如此), 也可以在我用到束的任何地方使用集合, 那就是说, 束是不必要的。同样, 我们可以在不依赖于集合的情况下定义表(正如我在本书中所做的), 也总是可以用表替换集合, 这就意谓着, 集合也是不必要的。但是, 集合是一个优美的数据结构, 引入了一个概念(包装), 而且我愿意保留它们。同样, 束也是一个优美的数据结构, 也引入了一个概念(聚合), 而且我也愿意保留它。我总是倾向于使用够用的最简单的结构。

函数程序设计一节由于不能方便地表达非确定性而未能尽述。要描述一个值的某些性质但又不完全约束它, 可以用可能值集合来表示, 但不幸的是, 集合不能恰当地确定化；在这种情况下, 包含一个元素的集合不等于元素本身又成了一个难题。所需要的正是束, 一个束总是可以被看作是一个“非确定值”。

束在本书中也被用作“类型论”。不用问, 其他人肯定和我一样也不希望看到类型论重复它值空间中的所有运算符：对作用于值上的每个运算, 在类型空间上都有相应的运算。通过使用束, 这样的重复就被消除了。

很多数学家认为波形括号和逗号只是语法符号, 而语法符号尽管必要, 却是恼人和不重要的。我把它们当作具有代数性质的运算符(在2.1集合论一节中, 我们看到波形括号有逆运算), 这是一个由来已久的历史趋势, 例如, = 最初也是一个语法符号, 表示两件事物(在某些方面)是相同的, 但现在已成为一个具有代数性质的运算符。

----------束论结束

11.0.3串论

在许多文章中, 作者会对其有时把表连接记号错用作表和项的连接而感到抱歉, 或者也许有三个连接记号：一个是连接两个表, 一个是在表头加一项, 还有一个是在表尾加一项。为了弄清楚，可怜的作者不得不与表所提供的包装符号作斗争。我向这些作者提供字符串：不需要包装。(当然, 它们可以在需要时包装到表中, 我并不是在摈弃表)。

----------字符串理论结束

11.0.4函数理论

我使用了单词“局部的”和“非局部的”，而其它人可能使用的是单词“约束的”和“自由的”，或“局部的”和“全局的”，或“隐藏的”和“可见的”，或“私有的”和“公共的”。逻辑传统是从已“存在”的所有可能变量(无穷多)开始, 我并未遵守。函数记号(λ)称为 对变量进行“约束”, 而任何未被约束的变量保持“自由”, 例如,

$$(\lambda x{:}\ int\cdot\ x+3)\ y$$

包含约束变量*x*, 自由变量*y*和无穷多的其它自由变量。本书中, 变量并不会自动“存在”；它们或者通过使用函数记号形式化地引入(不是约束), 或者通过自然语言说明非形式化地引入。

即使其结果可能不是它所应用的函数的任何结果，从 max 形成的量词仍然是*MAX*。名称“最小上界”是传统的。类似地，对于*MIN*，传统地称为“最大下界”。

我忽略了极限“存在”的传统问题；在传统的极限不“存在”的情况下，极限公理不能告诉我们极限是什么，但它仍能告诉我们一些有用的东西。

----------函数理论结束

11.0.5程序理论

赋值语句可能定义为

$$x := e \;=\; defined"e"\wedge e : T \Rightarrow x' = e \wedge y' = y \wedge \ldots$$

其中 *defined* 排除了象1/0这样的表达式，而 T 是 x 的类型。我将 *defined* 留出来是因为对它的完全的定义是不可能的，一个合理的完全定义的复杂程度已经相当于整个程序理论了，而且没有必要。前件 $e : T$ 是有用的，使得 n 为自然数时 $n := n - 1$ 是可实现的。但它的好处不比带来的麻烦多，因为在每个相关组合中都要进行同样的检测。

自从Algol-60设计出来, 顺序执行常常用分号表示, 但分号对我来说已不可用, 因为我已经用它来表示字符串连接。相关组合是一种乘积, 所以我希望句号会是一个可接受的符号, 我考虑交换这两个符号, 用分号表示相关组合, 而用句号表示字符串连接, 但是不可行。

在自然语言中, “前置条件”指的是“事先必要的准备”, 而在很多程序设计书籍中, “前置条件”用来表示“事先充分的准备”。在那些书中, “最弱前置条件”指的是“必要且充分”, 即我所称的“精确前置条件”。

在最早的仍然广为人知的程序设计理论中，我们将变量 x 的增加表示为

$$\{x = X\}S\{x > X\}$$

我们假设在这个规范中，知道 x 是一个状态变量，X 是一个局部变量，其目的是将 x 的初始值和终结值关联起来，而 S 对规范而言也是局部的，是为程序确定位置的。在精化该规范的程序中， X 和 S 都不会出现。形式化地，可以用量词表示它们如下：

$$§\,S\cdot \forall X\cdot \{x=X\}\ S\ \{x > X\}$$

在最弱前置条件理论中, 等价的规范看上去是类似的：

$$§\,S\cdot \forall X\cdot x=X \Rightarrow wp\ S\ (x > X)$$

这些记号有两个问题, 一是它们不提供同时引用前置状态和后置状态两者的任何方法, 因此导致*X*的引入。这个问题在维也纳开发模式中得到解决, 同样的定义为

$$§\,S\cdot \{\mathsf{T}\}\ S\ \{x' > x\}$$

另一个问题是程序设计语言和规范语言分离, 因此导致*S*的引入。在我的理论中, 程序设计语言是规范语言的子语言, 变量*x* 增加的定义是

$$x' > x$$

Z中使用了同样的单式双态规范, 但是精化相当复杂, 在Z中, *P*被*S*精化当且仅当

$$\forall\sigma\cdot (\exists\sigma'\cdot P) \Rightarrow (\exists\sigma'\cdot S) \wedge (\forall\sigma'\cdot P \Leftarrow S)$$

在早期理论中, $§\,S\cdot \{P\}\ S\ \{Q\}$被 $§\,S\cdot \{R\}\ S\ \{U\}$精化当且仅当

$$\forall\sigma\cdot P \Rightarrow R \wedge (Q \Leftarrow U)$$

在我的理论中, *P*被*S*精化当且仅当

$$\forall\sigma, \sigma'\cdot P \Leftarrow S$$

既然精化是我们在程序设计时必须证明的, 最好是使它尽可能简单。

有人也许会推测任何类型的数学表达式都可以用作规范：无论什么都行。某事物的规范,

不管是汽车还是计算, 都要能区分满足规范的事物和不满足的事物。对某事物的观察提供了特定变量的值, 基于这些值, 我们必须能够确定该事物是否满足这条规范。所以, 我们有一条规范, 一些变量的值, 以及两个可能的结果, 这正好是布尔表达式的工作：一条规范(关于任何事物)实际上就是一个布尔表达式。如果我们转而使用一对谓词, 或是一个从谓词到谓词的函数, 或者任何其它别的什么, 我们就得用间接方式书写规范, 并且使确定满足与否的任务更加复杂。

有人也许会想, 任何布尔表达式都可用来刻画任意计算机行为：无论怎么对应都行。在Z中, 表达式T用来刻画(描述)终止计算, 而⊥用来刻画(描述)非终止计算, 理由是：⊥是没有可满足的终结状态的规范；无穷计算是无终结状态的行为；因此 ⊥ 表示无穷计算。虽然我们不能观察无穷计算的终结状态, 但我们可以简单地通过等待10个时间步, 观察到它满足$t' > t+10$, 而不满足 $t' \le t+10$。所以它应当满足由$t' > t+10$所蕴含的任何规范, 包括T, 而不应当满足由$t' \le t+10$所蕴含的任何规范, 包括 ⊥。因为⊥对任何事物都为假, 所以它(实际上)不描述任何事物。一条规范是一个描述, 而⊥是不可满足的, 即使是非终止计算也不例外。因为T对任何事物都为真, 所以它(实际上)描述了一切事物, 即使是非终止计算也不例外。称*P*精化*Q*也就是指所有满足*P*的情形都满足*Q*, 这就是蕴含。规范和计算机行为之间的对应是不可以随心所欲的。

正如第四章中所指出的, 诸如$x' = 2 \wedge t' = \infty$这样的规范有些古怪, 因为它们谈论了无穷时刻变量的“终结”值。我可以修改理论以防止提及无穷时刻的结果, 但我没有这样做, 有两个理由：那会使得理论更加复杂, 并且当我引入交互(第九章)时需要区别无限循环。

----------程序理论结束

11.0.6程序设计语言

第五章中给出的变量说明形式对新的局部变量赋了一个属于其类型的任意值。例如，如果*y*和*z*为整数变量，于是有：

$$\textbf{var}\ x{:}\ nat \cdot y{:=}x = y' : nat \wedge z' = z$$

从实现简单和执行快速角度来看，这种方法比实例化成一个“未定义的值”要好得多。从错误检测的角度来看, 假设我们已经证明了所有的精化，那么这种方法也不坏。进一步地，有时初始化成一个任意值正是我们所希望的(参见练习270 (多数表决))。然而，如果我们不能证明所有的精化，那么初始化成一个未定义的值提供了一种保护手段。如果我们允许将一般操作符(=, ≠, **if then else**)应用于未定义的值，那么就可以证明类似*undefined=undefined*的平凡等式。如果不允许，就无法证明关于未定义值的任何等式。有些程序设计语言为了消除由于使用未实例化变量而引起的错误，将每一个变量初始化成其类型上的一个标准值。这种语言真是糟透了：既不如初始化成任意值有效，又只是消除了错误检测而不是错误本身。

在**while**循环中, 最广为人知和广为使用的规则是不变式和变式方法。令*I*为一个前置条件(称为“不变式”), 令I' 为相应的后置条件，令*v*为一个整数表达式(称为“变式”或“约束函数”), 令v' 为相应的表达式，其中所有变量都带有撇号。于是，不变式和变式规则为：

$$I \Rightarrow I' \wedge \neg b' \Leftarrow \textbf{while}\ b\ \textbf{do}\ \ I \wedge b \Rightarrow I' \wedge 0 \le v' < v\ P$$

粗略地说，这条规则表示：如果循环体保持不变式并减少变式但不小于0, 那么循环能够保持不变式并使得循环条件为假。例如，为了证明：

$$s' = s + \sum L\ [n\ ;..\#L] \Leftarrow \textbf{while}\ n \ne \#L\ \textbf{do}\ (s{:=}s + Ln.\ n{:=}\ n+1)$$

我们必须创造一个不变式

$$s + \sum L\,[n\,;..\#L] = \sum L$$

和一个变式

$$\#L - n$$

并且同时证明

$$s' = \quad s + \sum L\,[n;..\#L]$$

$$\Leftarrow \qquad s + \sum L\,[n;..\#L] = \sum L \;\Rightarrow\; s' + \sum L\,[n';..\#L] = \sum L \wedge n' = \#L$$

和

$$s + \sum L\,[n;..\#L] = \sum L \wedge n \neq \#L \Rightarrow \quad s' + \sum L\,[n';..\#L] = \sum L \wedge 0 \leq \#L - n' < v$$

$$\Leftarrow \; s := s + Ln.\; n := n + 1\;)$$

第五章中给出的证明方法更为简单，并能获得更多的信息(时间)。

如果概率为广义实数，而不是0到1闭区间上的实数，那么概率理论会更简单，在这种情况下我将增加公理$\mathsf{T} = \infty$ □ $\perp = -\infty$ □但发明一个更好的概率理论并非我在本书中的目的。对于概率程序设计，我的第一个方法是将变量的类型重新解释为以函数形式描述的概率分布。如果x是类型T的变量，它变成一个类型$T \to prob$的变量使得$\sum x = \sum x' = 1$。然后所有的运算也必须扩展到函数形式表示的分布上。尽管这种方法是有效的，但它太底层了。以函数形式描述的分布通过它们在参数表中的位置，而不是通过名字来告诉我们其变量的概率。

程序设计的主题常常被误认为是学习大量的程序设计语言“特征”, 这种错误在命令式和函数式程序设计语言中都有犯过。当然，一种程序设计语言所提供的每一个好的操作符都会使某些问题的解决变得简单。在函数式程序设计中，常常提出一种称作“折叠”或“归约”的操作符，它是对某些量词的一个有用的推广。它的符号可能为 /, 其左操作数为一个双目操作符，其右操作数为一张表。表求和问题可通过+/ L解决，而查找问题可类似地通过使用一个合适的查找操作符解决，设计和实现这样一个操作符是最为有用的练习。对一个仅具备找一个已实现的操作符然后应用它的程序设计能力的人来说，这个练习是无法完成的。本书的目的就是教授必要的程序设计技巧。

正如我们的练习所阐明的那样，函数式程序设计和命令式程序设计本质上是一样的：同一个问题在这两种方式下需要同样的解决步骤。所不同的有以下几点：命令式程序员坚持使用烦琐的循环记号，使得证明复杂化；而函数式程序员坚持使用等式，而不是精化，这使得非确定性问题的解决更为困难。

----------程序设计语言结束

11.0.7递归定义

构造和归纳的结合是如此地漂亮和有用，以至于它有一个名字(产生式)和一个记号(::=)。为了保持所用的术语和记号尽可能地少, 我们没有使用它们。

递归结构总是可以通过取一个近似序列的极限完成。我的创新是用序列的索引替代∞，这比寻找极限要容易得多。替换∞并不总能保证产生一个想要的不动点，但寻找极限也是一样。替换∞在除了设法求极限的例子中效果还是不错的。

----------递归定义结束

11.0.8理论设计与实现

我使用名词“数据转换”代替其他人所用的名词“数据精化”, 我看不出有任何理由可以认为其中之一更为“抽象”而另一个更为“具体”。我所谓的“数据转换式”有时也称为“抽象关系”，“连接不变式”，“粘合关系”，“恢复函数”，或“数据不变式”。

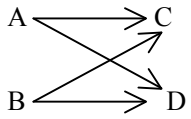
我们使用了一个小心构造的例子，而不是一个在实际中会发生的例子，说明了数据转换的不完备性。我更倾向于采用简单的规则，这些规则对任何会真正发生的问题（而不仅是为说明理论上的不完备性的问题）的转换而言是充分的，而不倾向于一个具有完备性的更复杂的规则或规则的组合。为了重新获得完备性，我们做需要的只是引入局部变量的正常数学实践。这种目的的变量被不同的作者称为“边界变量”，“逻辑常量”，“规范变量”，“灵魂变量”，“抽象变量”和“预言变量”。

----------理论设计和实现结束

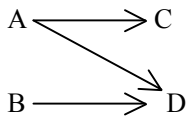
11.0.9并发

在FORTRAN语言中(1977年以前), 我们允许顺序组合包含**if–**语句，但不允许在**if–**语句中包含顺序组合。但在ALGOL 语言中，其语法完全递归，顺序和条件组合可以相互嵌套, 一个在另一个之中。我们吸取了其中的教训了吗？显然还没有学到一个通用的方法：我们现在似乎很高兴可以在并行组合中嵌套顺序组合，但是，如果要在顺序组合中嵌套并行组合就比较勉强。因此在当前流行的语言中，并行组合只能出现在结构的最外层。

正如我们在第8章看到的，执行模式



可以被表示为 $((A\|B).(C\|D))$，而不需任何同步原语。但模式



不能只使用并行和顺序组合表示。该模式在缓冲程序中出现。

在本书的第一版本，并行组合是为具有相同状态空间的进程定义的（半-相关组合）。那个定义比现在的定义（练习378）要复杂得多，但理论上，它免除了对变量的划分。然而在实际中，变量总是要划分的，因此在当前版本中我们使用了一个更简单的定义（独立组合）。

----------并发结束

11.0.10交互

在实现式公式中，不存在合取式$r' \le w'$ 以保证读游标不会向前超过写游标，而在9.1.8死锁这一小节中，我们看到这种情况确实会发生, 当然它要花费无穷时间。在死锁例子中，我们可以证明时间为无穷。但是该理论有一个瑕疵, 考虑以下例子：

$$\begin{aligned} & \textbf{chan } c \cdot t := max\ t\ (\forall r{+}1).\ c? \\ = \quad & \exists \mathcal{M}, \forall, r, r', w, w' \cdot t' = max\ t\ (\forall r + 1) \wedge r' = 1 \wedge w' = 0 \\ = \quad & t' \ge t \end{aligned}$$

我们希望证明$t' = \infty$。为得到这个答案，必须强化对局部信道的定义，即加入合取式$\forall w' \ge t'$。但我倾向于使用比较简单和弱化的理论。

----------交互结束

我们可以谈论信道结构和索引进程, 我们可以谈论一个并行的**for–**循环。总是有更多的东西可以谈论，但我们必须在某处停下来。

----------释疑结束

11.1来源

思想决不是空穴来风，它是人们所受教育、文化熏陶以及与熟人交流的结果。我感谢所有那些给我影响并使我能够完成本书的人, 我可能没有提及那些间接影响我的人，尽管他们的影响可能很大; 我可能没有感谢那些在失意的日子里给我出谋划策的人，因为那时我无心理会; 我可能没有称赞那些独立工作的人，他们的观点也许和我所看到和听到的一样或更好。对所有这样的人，我表示抱歉。我不相信有人会为一个观点来邀功, 理想地说，我们的研究是为了造福于所有的人，也许还有兴致所至，但决不是为了个人荣耀。当然被忽视也是令人失望的。以下就是我所能提供的最完整的来源清单。

本课题的早期工作来自Alan Turing (1949), Peter Naur (1966), Robert Floyd (1967), Tony Hoare (1969) , Rod Burstall (1969), 以及Dana Scott 和Christopher Strachey (1970)。(参见随后的文献目录)。把我自己引入该领域的是Edsger Dijkstra (1976) 的一本书; 读完以后，我在形式化精化方面迈出了第一步(1976)。Ralph Back在该方向进行了进一步的工作(1978), 尽管我直到1984年才知道。本课题的第一批教科书开始出现, 其中也有我的一本(1984)。这些工作都是基于Dijkstra的最弱前置条件谓词转换器, 同一基础上的工作至今仍在继续。我强力推荐Ralph Back和Joachim von Wright所著的“精化演算”一书(1998)。

同一时刻，Tony Hoare正在开发通信顺序进程(1978,1981)。1981年我在牛津呆了一个学期，其间我意识到这些通信顺序进程可以用谓词来描述，因而发表了一个谓词模型(1981,1983)。很快就又发现这种描述方式，即一个单一布尔表达式, 显然可以用于任何一种计算，而且实际上可以用于描述任何其他事物; 回想过去，这一点在一开始就很显然。这些结果发表在一系列文章中(1984, 1986, 1988, 1989, 1990,1994,1998,1999,2004), 最终导致本书的出现。

Netty van Gasteren使我明白了表达式格式和证明格式的重要性(1990)。Chris Lengauer建议我用¢和$分别表示束和集合的基数。 “conflation”一词的使用由Doug McIlroy提议。索引值从0开始这一点赐教于Edsger Dijkstra。 Joe Morris和Alex Bunkenburg找到并修改了束论中的一个问题。“apposition”一词及其用法得自于Lambert Meertens(1986)。Alan Rosenthal建议

我停止担心极限的 “存在”, 只要用公理描述它们即可。我希望这从数学中删除了柏拉图学说的最后痕迹，尽管在英语中还存在一些。Theo Norvell使得我的部分精化定律更为普遍化。我从Chris Lengauer处学会使用计时变量(1981)，他将此归功于Mary Shaw; 我们那时正在使用最弱前置条件，所以我们的时间变量只能下降不能上升。递归时间度量从Paul Caspi、Nicolas Halbwachs、Daniel Pilaud、和John Plaice的工作中得到激发(1987); 在他们的语言LUSTRE中，循环的每一次执行占用一个时间单位，所有其他的执行不需要时间。我从与Andrew Malton的讨论以及Hendrik Boom的一个例子中，学会看轻终止本身，不带有时间界限(1982)。Wlad Turski告诉我斐波那契数问题的对数解, 他在访问Guelph大学时学会这个解。我的关于局部变量说明的不正确的版本得到Andrew Malton的纠正。局部变量的悬挂从Carroll Morgan处改编过来(1990)。**For**循环规则是受了Victor Kwan和Emil Sekerinski的影响。不可实现的规范的回溯实现来自于Greg Nelson(1989)。Carroll Morgan和Annabelle McIver（1996）建议将概率看作可观察的量词，练习284（豆先生的袜子）就源自他们。在函数式程序设计语言的不确定性中和函数精化中使用束这一工作是与Theo Norvell合作完成(1992)。Theo还将时间加入**while**循环的递归定义中。数据类型理论(数据-栈, 数据-队, 数据-树)的风格来自于John Guttag和Jim Horning(1978)。数据-树的实现受到Tony Hoare 的影响(1975)。有些程序-树理论的具体细节归功于Theo Norvell，Yannis Kassios和Peter Kanareitsev。我从He Jifeng和Carroll Morgan处学到数据转换，它基于Tony Hoare的早期工作(1972); 这里出现的公式是我自己的，但我检查过它们与Wei Chen和Jan Tijmen Udding(1989)的公式的等价性。Theo Norvell提供了数据转换式的准则。第二个数据转换例子（取一个数）是从Carroll Morgan(1990)的资源分配例子改编过来的。最后的表示不完备性的数据转换例子是Paul Gardiner和Carroll Morgan发明的(1993)。关于数据转换的百科可参见Willem-Paul deRoever和Kai Engelhardt的书(1998)。我发表了关于独立(并行)组合(1981,1984,1990,1994)的各种各样的公式，第一版的应归功于Theo Norvell，在本版本中作为练习378（半-相关组合）出现，并在Hoare和He的最近工作(1998)中使用到。在本版本中Leslie Lamport说服我回到早先的(1990)版本：简单合取。8.1顺序到并行转换一节是与Chris Lengauer(1981)的合作工作; 自此以后，他在从一般顺序、命令式程序中自动开发高度并行、脉动的计算这一领域取得很大的进展。燃气点燃装置例子是Anders Ravn、Erling Sorensen、和Hans Rischel(1990)的一个类似例子的简化和改编。通信的形式受到Gilles Kahn(1974)的影响。时间脚本由Theo Norvell提议。试验值(probe)是Alain Martin(1985)的一个发明。监控器(Monitors)由Per Brinch Hansen(1973)和Tony Hoare (1974)创造。幂级数相乘是从Doug McIlroy(1990)而来，而它归功于Gilles Kahn。许多练习来自于我早期的一本书(1984), 它们由Wim Feijen给出，并由Edsger Dijkstra、Wim Feijen、Netty van Gasteren、和Martin Rem进一步发展作为Eindhoven技术大学的考试题; 它们自此以后就出现在Edsger Dijkstra和Wim Feijen(1988)合写的一本书中。有些练习来自Martin Rem(1983,..1991)所写的一系列杂志文章。其他的一些练习来源十分广泛，这里无法一一提及。

----------来源结束

11.2 参考文献

R.-J.R.Back: “on the Correctness of Refinement Steps in Program Development”, University of Helsinki, Department of Computer Science, Report A-1978-4, 1978

R.-J.R.Back: “a Calculus of Refinement for Program Derivations”, Acta *Informatica*, volume 25, pages 593,..625, 1988

R.-J.R.Back, J.von Wright: *Refinement Calculus*: a Systematic Introduction, Springer, 1998

H.J.Boom: "a Weaker Precondition for Loops", ACM *Transactions on Programming Languages and Systems*, volume 4, number 4, pages 668,..678, 1982

R.Burstall: "Proving Properties of Programs by Structural Induction", University of Edinburgh, Report 17 DMIP, 1968; also *Computer Journal*, volume 12, number 1, pages 41,..49, 1969

P.Caspi, N.Halbwachs, D.Pilaud, J.A.Plaice: "LUSTRE: a Declarative Language for Programming Synchronous Systems", *fourteenth annual ACM Symposium on Principles of Programming Languages*, pages 178,..189, Munich, 1987

K.M.Chandy, J.Misra: *Parallel Program Design: a Foundation*, Addison-Wesley, 1988

W.Chen, J.T.Udding: "Toward a Calculus of Data Refinement", J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science, volume 375, pages 197,..219, 1989

E.W.Dijkstra: "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", *Communications ACM*, volume 18, number 8, pages 453,..458, 1975 August

E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, 1976

E.W.Dijkstra, W.H.J.Feijen: *a Method of Programming*, Addison-Wesley, 1988

R.W.Floyd: "Assigning Meanings to Programs", *Proceedings of the American Society, Symposium on Applied Mathematics*, volume 19, pages 19,..32, 1967

P.H.B.Gardiner,C.C.Morgan: "a Single Complete Rule for Data Refinement", *Formal Aspects of computing*, volumn 5, number 4, pages 367,..383, 1993

A.J.M.vanGasteren: "on the Shape of Mathematical Arguments", Springer-Verlag Lecture Notes in Computer Science, 1990

J.V.Guttag, J.J.Horning: "the Algebraic Specification of Abstract Data Types", *Acta Informatica*, volume 10, pages 27,..53, 1978

E.C.R.Hehner: "**do** considered **od**: a Contribution to the Programming Calculus", University of Toronto, Technical Report CSRG-75, 1976 November; also *Acta*

Informatica, volume 11, pages 287,..305, 1979

E.C.R.Hehner: "Bunch Theory: a Simple Set Theory for Computer Science", University of Toronto, Technical Report CSRG-102, 1979 July; also *Information Processing Letters*, volume 12, number 1, pages 26,..31, 1981 February

E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", University of Toronto, Technical Report CSRG-134, 1981 September; also *Theoretical Computer Science*, volume 26, numbers 1 and 2, pages 105,..121, 1983 September

E.C.R.Hehner: "Predicative Programming", *Communications ACM*, volume 27, number 2, pages 134,..152, 1984 February

E.C.R.Hehner: *the Logic of Programming*, Prentice-Hall International, 1984

E.C.R.Hehner, L.E.Gupta, A.J.Malton: "Predicative Methodology", *Acta Informatica*, volume 23, number 5, pages 487,..506, 1986

E.C.R.Hehner, A.J.Malton: "Termination Conventions and Comparative Semantics", *Acta Informatica*, volume 25, number 1, pages 1,..15, 1988 January

E.C.R.Hehner: "Termination is Timing", International Conference on Mathematics of Program Construction, The Netherlands, Enschede, 1989 June; also J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science volume 375, pages 36,..48, 1989

E.C.R.Hehner: "a Practical Theory of Programming", *Science of Computer Programming*, volume 14, numbers 2 and 3, pages 133,..159, 1990

E.C.R.Hehner: "Abstraction of Time", *a Classical Mind*, chapter 12, Prentice-Hall, 1994

E.C.R.Hehner: "Formalization of Time and Space", *Formal Aspects of Computing*, volumn 10, pages 290,..307, 1998

E.C.R.Hehner, A.M.Gravell: "Refinement Semantics and Loop Rules", FM'99 World Congress on Formal Methods, pages 20,..25, Toulouse France, 1999 September

E.C.R.Hehner: "Specifications, Programs, and Total Correctness", *Science of Computer Programming* volume 34, pages 191,..206, 1999

E.C.R.Hehner: "Probabilistic Predicative Programming", Conference on Mathematics og Program Construction, Scotland, Stirling, 2004 July 12,..15, and Springer Lecture

Notes in Computer Science, D.M.Kozen(editor), volume 3125, pages 169,..186, 2004

C.A.R.Hoare: “an Axiomatic Basis for Computer Programming”, *Communications ACM*, volume 12, number 10, pages 576,..581, 583, 1969 October

C.A.R.Hoare: “Proof of Correctness of Data Representations”, *Acta Informatica*, volume 1, number 4, pages 271,..282, 1972

C.A.R.Hoare: “Monitors: an Operating System Structuring Concept”, *Communications ACM*, volume 17, number 10, pages 549,..558, 1974 October

C.A.R.Hoare: “Recursive Data Structures”, *International Journal of Computer and Information Sciences*, volume 4, number 2, pages 105,..133, 1975 June

C.A.R.Hoare: “Communicating Sequential Processes”, *Communications ACM*, volume 21, number 8, pages 666,..678, 1978 August

C.A.R.Hoare: “a Calculus of Total Correctness for Communicating Processes”, *Science of Computer Programming*, volume 1, numbers 1 and 2, pages 49,..73, 1981 October

C.A.R.Hoare: “Programs are Predicates”, in C.A.R.Hoare, J.C.Shepherdson (editors): *Mathematical Logic and Programming Languages*, Prentice-Hall Intenational, pages 141,..155, 1985

C.A.R.Hoare, I.J.Hayes, J.He, C.C.Morgan, A.W.Roscoe, J.W.Sanders, I.H.Sϕrensen, J.M.Spivey, B.A.Sufrin: “the Laws of Programming”, *Communications ACM*, volume 30, number 8, pages 672,..688, 1987 August

C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall, 1998

C.B.Jones: *Software Development: a Rigorous Approach*, Prentice-Hall International, 1980

C.B.Jones: *Systematic Software Development using VDM*, Prentice-Hall International, 1986 and 1990

G.Kahn: “the Semantics of a Simple Language for Parallel Programming”, *Information Processing 74*, North-Holland, Proceeding of IFIP Congress, 1974

C.Lengauer, E.C.R.Hehner: “a Methodology for Programming with Concurrency”, CONPAR 81, Nurnberg, 1981 June 10,..13; also Springer-Verlag, Lecture Notes in Computer Science volume 111, 1981 June, pages 259,..271; also *Science of Computer Programming*, volume 2, 1982, pages 1,..53

A.J.Martin: “the Probe: an Addition to Communication Primitives”, *Information Processing Letters*, volume 20, number 3, pages 125,..131, 1985

J.McCarthy: “a Basis for a Mathematical Theory of Computation”, *Proceedings of the Western Joint Computer Conference*, pages 225,..239, Los Angeles, 1961 May; also Computer Programming and Formal Systems, North-Holland, pages 33,..71, 1963

M.D.McIlroy: “Squinting at Power Series”, *Software Practice and Experience*, volume 20, number 7, pages 661,..684, 1990 July

L.G.L.T.Meertens: “Algorithmics - towards Programming as a Mathematical Activity”, Proceedings of CWI Symposium on Mathematics and Computer Science, North-Holland, *CWI Monographs*, volume 1, pages 289,..335, 1986

C.C.Morgan: “the Specification Statement”, *ACM Transactions on Programming Languages and Systems*, volume 10, number 3, pages 403,..420, 1988 July

C.C.Morgan: *Programming from Specifications*, Prentice-Hall International, 1990

C.C.Morgan, A.K.McIver, K.Seidel, J.W.Sanders: “Probabilistic Predicate Transformers”, *ACM Transactions on Programming languages and Systems*, volume 18, number 3, pages 325,..354, 1996 May

J.M.Morris: “a Theoretical Basis for Stepwise Refinement and the Programming Calculus”, *Science of Computer Programming*, volume 9, pages 287,..307, 1987

P.Naur: “Proof of Algorithms by General Snapshots”, *BIT*, volume 6, number 4, pages 310,..317, 1966

G.Nelson: “a Generalization of Dijkstra's Calculus”, *ACM Transactions on Programming Languages and Systems*, volume 11, number 4, pages 517,..562, 1989 October

T.S.Norvell: “Predicative Semantics of Loops”, *Algorithmic Languages and Calculi*, ChapmanHall, 1997

T.S.Norvell, E.C.R.Hehner: “Logical Specifications for Functional Programs”, International Conference on Mathematics of Program Construction, Oxford, 1992 June

A.P.Ravn, E.V.Sϕrensen, H.Rischel: “Control Program for a Gas Burner”, Technical University of Denmark, Department of Computer Science, 1990 March

M.Rem: “Small Programming Exercises”, articles in *Science of Computer Programming*, 1983,..1991

W.-P.deRoever, K.Engelhardt: Data Refinement: *Model-Oriented Proof Methods and their Comparisons*, tracts in Theoretical Computer Science volume 47, Cambrige University Pres, 1998

D.S.Scott, C.Strachey: “Outline of a Mathematical Theory of Computation”, Oxford University Report PRG-2, 1970; also *Proceedings of the fourth annual Princeton Conference on Information Sciences and Systems*, pages 169,..177, 1970

J.M.Spivey: *the Z Notation - a Reference Manual*, Prentice-Hall International, 1989

A.M.Turing: “Checking a Large Routine”, Cambridge University, Report on a Conference on High Speed Automatic Calculating Machines, pages 67,..70, 1949

-参考文献结束

11.3词语对照与索引
（按词语第一字的笔画顺序）

（一画）

一元	unary
一致	consensus
一致的	consistent
一致性规则	consistency rule
一般递归	general recursion

（二画）

几乎有序段	almost sorted segment
二元	binary
二叉决策树	binary decision diagram
二的指数运算	binary exponentiation
二分查找	binary search
二叉树	binary tree
十进制小数	decimal-point numbers

（三画）

下标	subscript
大小	size
广播	broadcast
小器件	widget
三分查找	ternary search
广义整数	extended integers
广义自然数	extended naturals
广义有理数	extended rationals

广义实数 extended reals
上下文 context
卫式命令 guarded command
已排序二维计数 two-dimensional sorted count
已实现的规范 implemented specification
女佣和男佣 maid and butler
（四画）
支点 pivot
尺度 scale
中断 break
元素 element
矛盾 contradition
反公理 antiaxiom
反单调 antimonotonic
反定理 antitheorem
反序计数 inversion count
不等式 unequation
不变式 invariant
不动点 fixed-point
不动点构造 fixed-point construction
不动点归纳 fixed-point induction
不动点定理 fixed-point theorem
不完备的 incomplete
不完备性 incompleteness
不一致的 inconsistent
不可满足的 unsatisfiable
分离定律 detachment
区间合并 interval union
元 arity
元语言 metalanguage
互斥 mutual exclusion
无穷大 infinity
中缀 infix
长度 length
双调表 bitonic list
公理 axiom
分割 partitions
分布 distribution
分配 distribute
分配性 distribution
长正文 long text
计算常量 computing constant
计算变量 computing variable
公共变量 public variable

公理系统 axiom schema
无界的界 unbounded bound
反应控制器 reaction controller
巴科斯诺范式 Backus-Naur Form
无Z的子正文 z-free subtext
以自然数二为底的对数 natural binary logarithm
（五画）
正文 text
术语 term
包装 package
记号 notation
记录 record
对偶 dual
立方 cube
立方测试 cube test
电话 telephone
布尔 Boolean
布尔的布尔 Boole’s Booleans
引用参数 reference parameter
字位和 bit sum
字典序 lexicographic order
写游标 write cursor
头和尾 head and tail
归纳 induction
平均 average
平均空间 average space
存在 existence
存在量 existential quantification
可满足的 satisfiable
可实现的 implementable
可达性 reachability
可靠性 soundness
可重置变量 resettable variable
用户变量 user’s variable
半相关组合 semi-dependent composition
由…导出 follows from
汉诺塔 Towers of Hanoi
未定义的值 undefined value
必要后置条件 necessary postcondition
必要前置条件 necessary precondition
冯诺依曼数 von Neumann numbers
（六画）
网球 tennis
芝诺 Zeno

后件 consequent
后继 successor
后置状态 poststate
同步 synchronous
同步通信 synchronous communication
字符 character
关系 relation
色子 dice
因子 factor
因式计数 factor count
凸等对 convex equal pair
有界队列 limited queue
有界堆栈 limited stack
有序对查找 ordered pair search
自描述 self-describing
自复制 self-reproducing
自动调温器 thermostat
合并 merge
合并 union
合取因子 conjunct
合取式 conjunction
产生式 generation
求总和 running total
求值逻辑 evaluation logic
求值规则 evaluation rule
约束函数 bound function
约束变量 bound variable
全称量 universal quantification
全部出现 all present
机器除法 machine division
机器乘法 machine multiplication
机器平方 machine squaring
多束 multibunch
多维的 multidimensional
多项式 polynomial
多数投票 majority vote
回溯 backtracking
交替和 alternating sum
并置 apposition
并发 concurrency
并行 parallelism
死锁 deadlock
划分 partition
自由的 free

过山车 roller coaster
执行时间 execution time
自然数除法 natural division
自然平方根 natural square root
充分前置条件 sufficient precondition
充分后置条件 sufficient postcondition
交 intersection
交互变量 interactive variable
交互式计算 interactive computing
交互数据转换 interactive data transformation
交换伙伴 swapping partners
传输时间 transit time
传递闭包 transitive closure
共享变量 shared variable
忙式等待循环 busy-wait loop
安全开关 security switch

（七画）

束 bunch
纸牌 blackjack
体 body
别名 alias
时钟 clock
阶乘 factorial
应用 application
极限 limit
余数 remainder
判断 sentence
局部的 local
否定式 negation
尾递归 tail recursion
作用域 scope
近似查找 approximate search
初始化 initializing
初试条件 initial condition
初始状态 initial state
形式化的 formal
完全的 total
批处理 batch processing
私有变量 private variable
证明格式 proof format
时间界 time bound
时间合并 time merge
时间脚本 time script
时间变量 time variable

时间耗尽 timeout
连续的 continuing
灵魂变量 ghost variable
完备的 complete
完备性 completeness
完备性规则 completion rule
直方图的最大方阵列 greatest square under a histogram
状态空间 state space
状态常量 state constant
条件组合 conditional composition
边界变量 boundary variable
传递闭包 transitive closure
条件组合 conditional composition
完美交替 perfect shuffle
快速指数运算 fast exponentiation
克努斯，莫里斯，普拉特 Knuth, Morris, Pratt
麦卡锡的91问题 McCarthy’s 91 problem
豆先生的袜子 Mr.Bean’s socks
（八画）
表 list
表比较 list comparison
表合成 list composition
表并发 list concurrency
表索引 list index
表达式 expression
连接不变式 linking invariant
取一个数 take a number
定理 theorem
定义域 domain
构造 construction
构造逻辑 constructive logic
构造式 constructor
实现者变量 implementer’s variable
单点 one-point
单调的 monotonic
空 nullary
空束 empty bunch
空集 empty set
空串 empty string
空间变量 memory variables
规则 rule
组合 combination
声明 declaration
定律 law

卷起 roll up
抽象空间 abstract space
抽象关系 abstract relation
经典逻辑 classical logic
奇偶校验 parity check
终止 termination
终结条件 final condition
终结状态 final state
变量 variable
变量声明 variable declaration
抽象关系 abstract relation
线性代数 linear algebra
线性查找 linear search
受控循环 controlled iteration
实参 argument
参数 parammeter
连接 catenation
析取因子 disjunct
析取式 disjunction
非确定性 nondeterministic
非局部的 nonlocal
变式 variant
变参调用 call-by-value-result
实例化 instantiation
实例化规则 instance rule
变量悬挂 variable suspension
运算对象 operand
运算符 operator
函数组合 function composition
函数包含 function inclusion
函数精化 function refinement
函数式程序设计 functional programming
命题 proposition
命令式程序设计 imperative programming
侦探小说 whodunit
帕斯卡三角形 Pascal’s triangle
终极周期序列 ultimately periodic sequence
罗素的理发师 Russell’s barber
罗素的悖论 Russell’s paradox

（九画）

段 segment
值域 range
测试 testing
指针 pointer

标尺 rulers
标记 sentinel
转向 go to
信号 signal
信道 channel
信息 information
信息脚本 message script
括号 brackets
括号代数 bracket algebra
选择合并 selected union
独特项 unique items
恢复函数 retrieve function
重排 reformat
重复 repetition
重复计数 duplicate count
前件 antecedent
前缀 prefix
前趋 predecessor
前置状态 prestate
复合数 composite number
查找 search
语法 grammar
语法分析 parsing
首饰盒 caskets
独角兽 unicorn
项计算 item count
真值表 truth table
真实时间 real time
顺序执行 sequential execution
顺序文件更新 file update
段和计数 segment sum count
相关组合 dependent composition
相交组合 disjoint composition
独立组合 independent composition
哥德尔/图灵非完备性 Godel/Turing incompletion

（十画）

弱于 weaker
通用 generic
退出 exit
展开 flatten
索引 index
框架 frame
换名 renaming
倒序 reverse

监视器 monitor
插入排序 insertion sort
读游标 read cursor
预言变量 prophesy variable
家族理论 family theory
缺少的数 missing number
高阶函数 higher-order function
矩阵相乘 matrix multiplication
部分精化法 refinement by parts
递归时间 recursive time
递归程序构造 recursive program construction
递归数据构造 recursive data construction
哲学家就餐 dining philosophers
费马最后程序 Fermat’s last program
（十一画）
堆 heap
堆栈 stack
通信 communication
断言 assertion
谓词 predicate
基本束 elementary bunch
基数 cardinality
副作用 side-effect
编译器 compiler
编辑距离 edit distance
隐藏变量 hidden variable
假言推理 modus ponens
控制进程 control process
旋转测试 rotation test
部分的 partial
情况精化法 refinement by cases
逐步精化 stepwise refinement
逐步精化法 refinement by steps
粘合关系 gluing relation
鸽子洞 pigeon-hole
康托的天国 Cantor’s heaven
康托的对角线 Cantor’s diagonal
骑士和恶棍 Knights and knaves
排序对查找 ordered pair search
（十二画）
等式 equation
等待 wait
量词 quantifier
强于 stronger

超束 hyperbunch
筛法 sieve
滑动 slip
确定的 deterministic
逻辑常量 logical constant
置换定律 substitution law
幂集 powerset
幂级数 power series
幂等排列 idempotent permutation
缓慢增长 grow slow
最短路径 shortest path
最小旋转 smallest rotation
最大真方阵 largest true square
最大下界 greatest lower bound
最小上界 least upper bound
最小差值 minimum difference
最小不动点 least fixed-point
最大子序列 greatest subsequence
最大公约数 greatest common divisor
最小公倍数 smallest common multiple
最小公共项 smallest common item
最大项 maximum item
最大积段 maximum product segment
最小和段 minimum sum segment
最大空间 maximum space
最长公共前缀 longest common prefix
最长平衡段 longest balanced segment
最长回文 longest palindrome
最长平稳段 longest plateau
最长平滑段 longest smooth segment
最长有序子表 longest sorted sublist
最早开会时间 earliest meeting time
最早放弃者 earliest quitter
最弱前置条件 weakest precondition
最弱后置条件 weakest postcondition
最弱前置规范 weakest prespecification
稀疏数组 sparse array
赋值 assignment
缓冲区 buffer
硬币 coin
循环表 circular list
循环数 circular number
递归数据构造 recursive data construction
递归程序构造 recursive program construction

斐波卢契数 Fibolucci
斐波那契数 Fibonacci
随机数产生器 random number generator
（十三画）
解 solution
算术 arithmetic
输入 input
输出 output
数组 array
数字和 digit sum
数字转换器 digitizer
数学常量 mathematical constant
数学变量 mathematical variable
数据不变式 data invariant
数据精化 data refinement
数据结构 data structure
数据转换 data transformation
数据转换式 data transformer
意外的鸡蛋 unexpected egg
（十四画及以上）
熵 entropy
整束 wholebunch
概率 probability
概率分布 probability distribution
蕴含 implication
模糊束 fuzzybunch
模型检测 model-checking
模式查找 pattern search
缩减J表 diminished J-list
霍夫曼编码 Huffman code
精确前置条件 exact precondition
精确后置条件 exact postcondition